Persistent Functional Languages: towards Functional Relational Databases

Lesley Wevers
Supervised by Marieke Huisman & Maurice van Keulen

Formal Methods and Tools & Databases
University of Twente

SIGMOD PhD Symposium 2014, Snowbird, Utah

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.

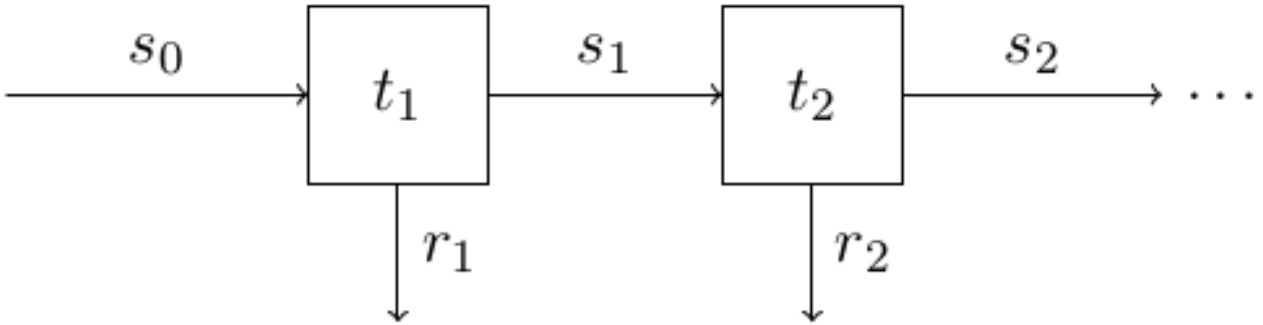
Using functional languages for the *updating* of databases has not really been explored.

Functional Transactions

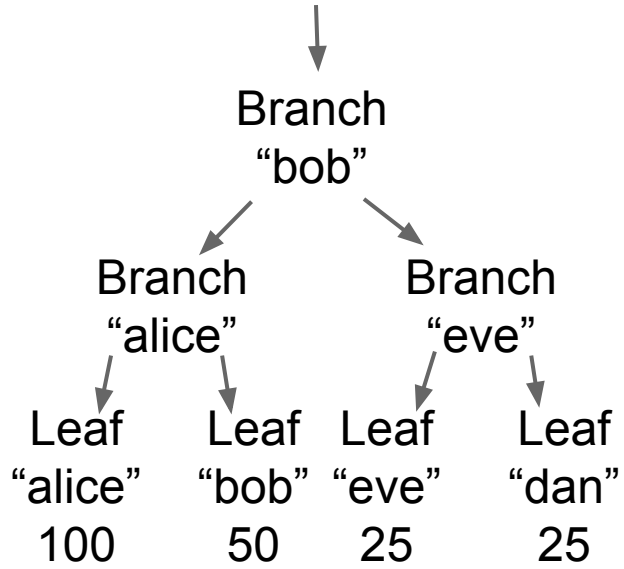
```
transaction : State -> State x Result
```

Functional Transaction Processing

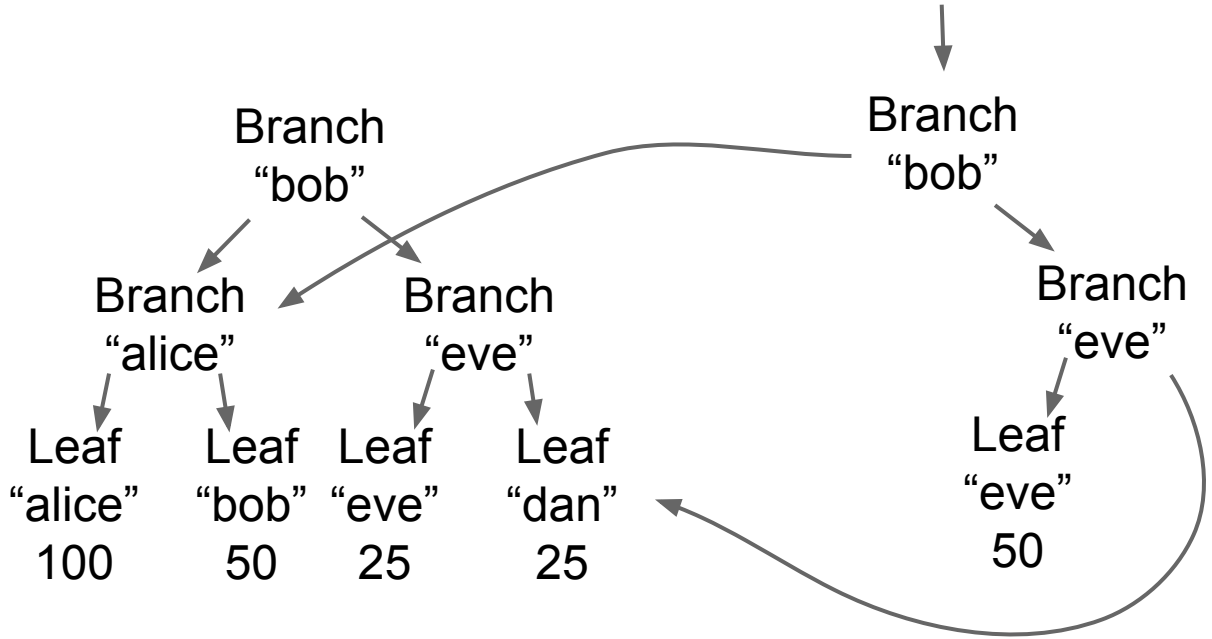
```
tm : State x [Transaction] -> [Result]
```



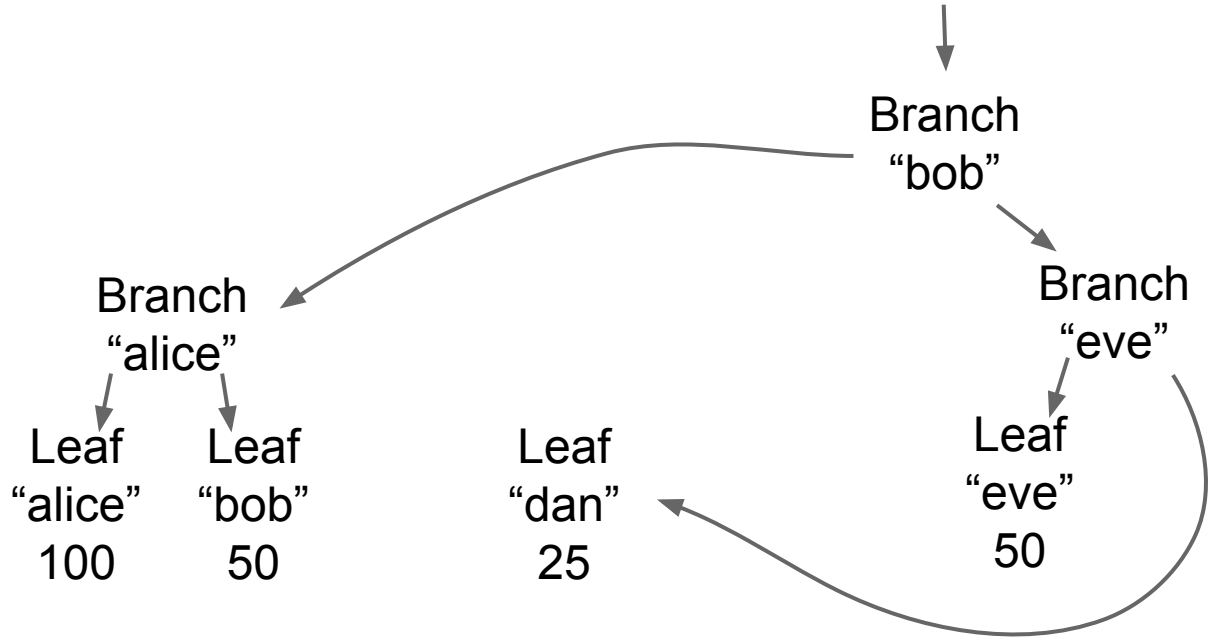
Functional States



Functional Updates



Functional Updates



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:

```
if check(next_state)
  then (next_state, result)
  else (prev_state, Error)
```

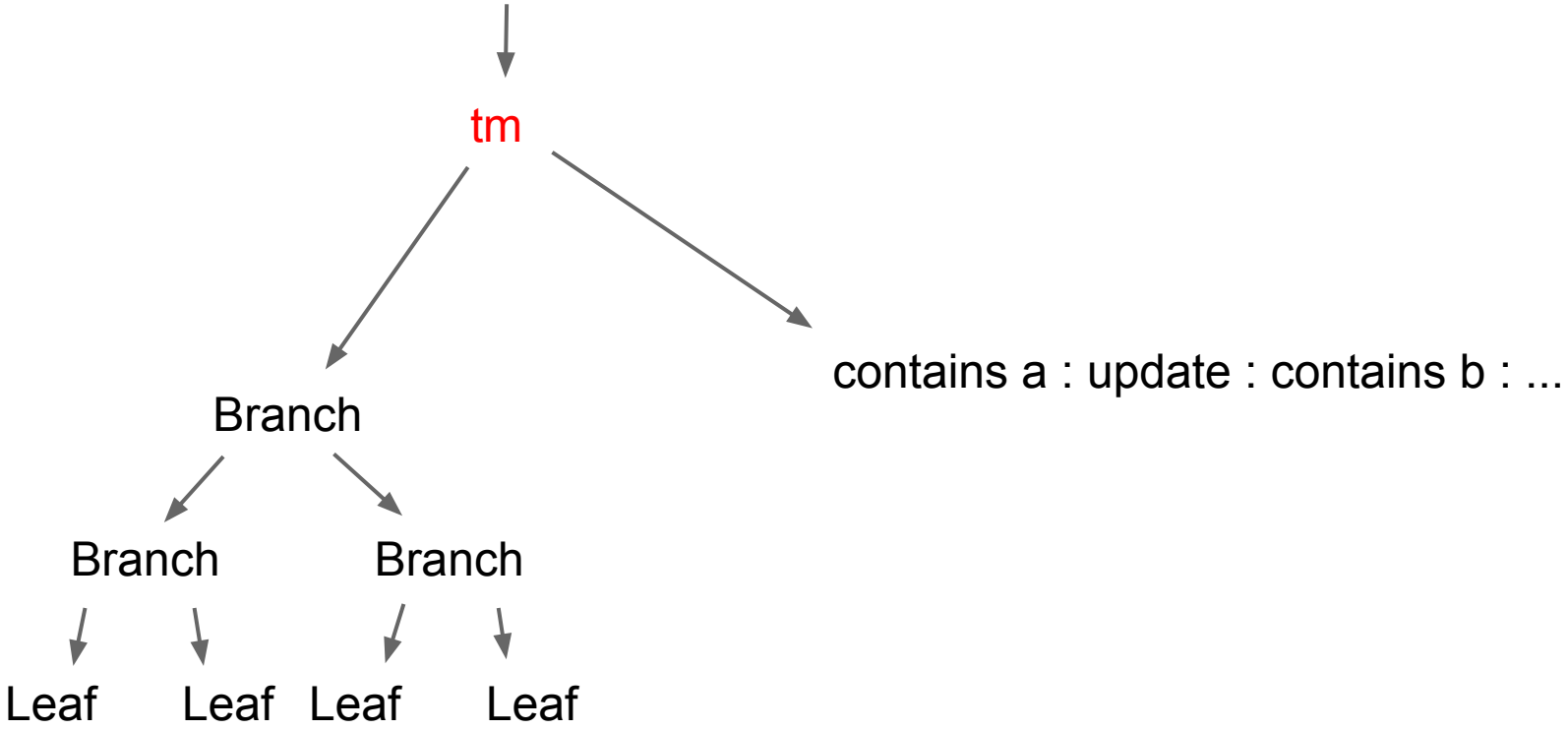
Transactions

This model satisfies the ACID properties:

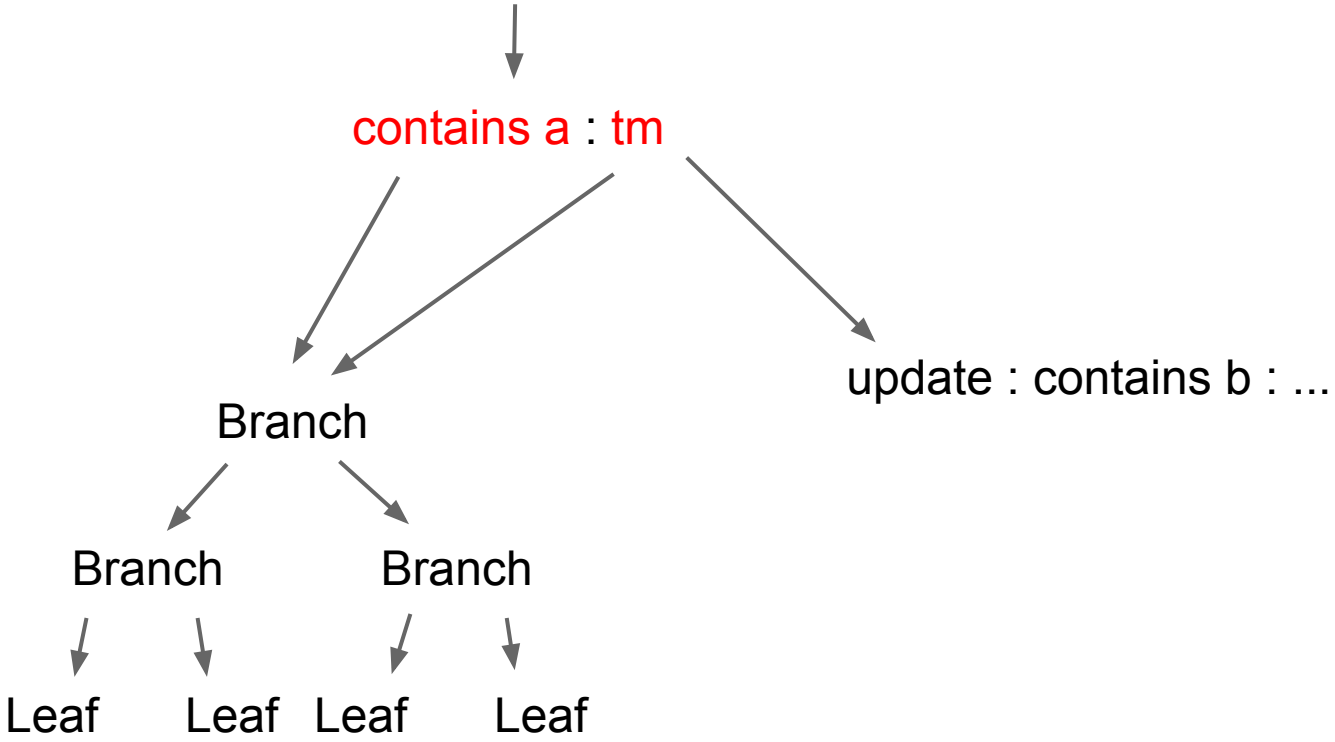
- Atomic
- Consistent
- Isolated
- Durable

But how do we execute transactions in parallel?

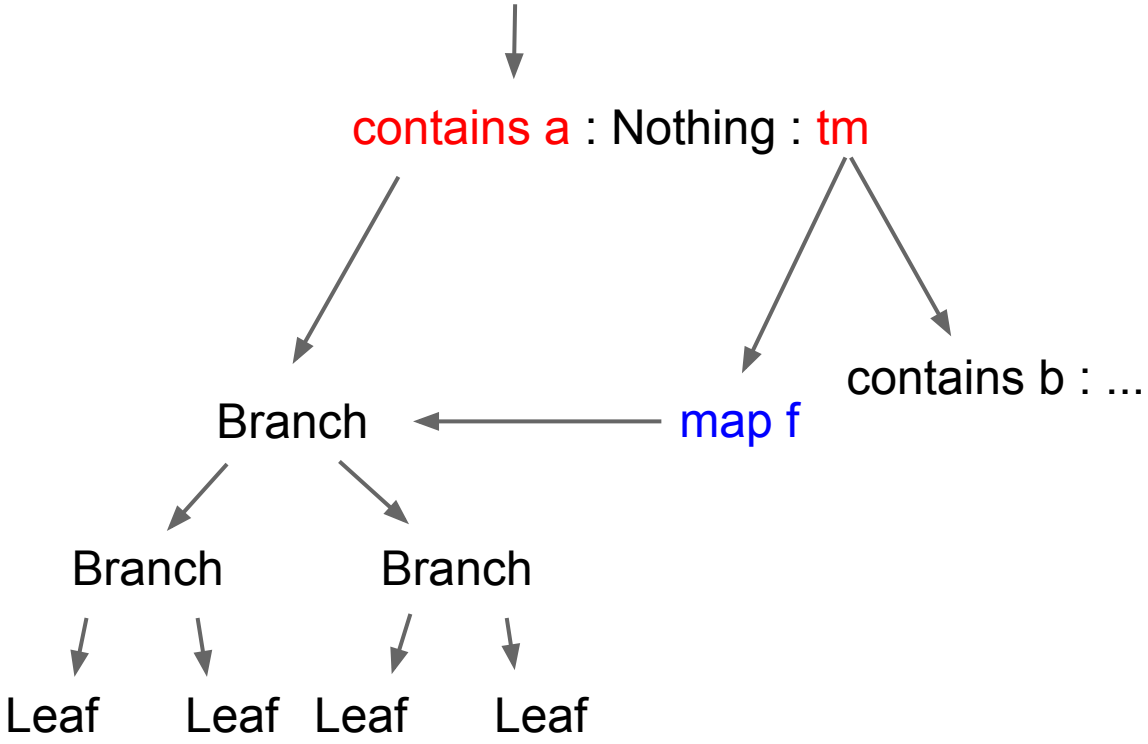
Concurrent Transaction Processing



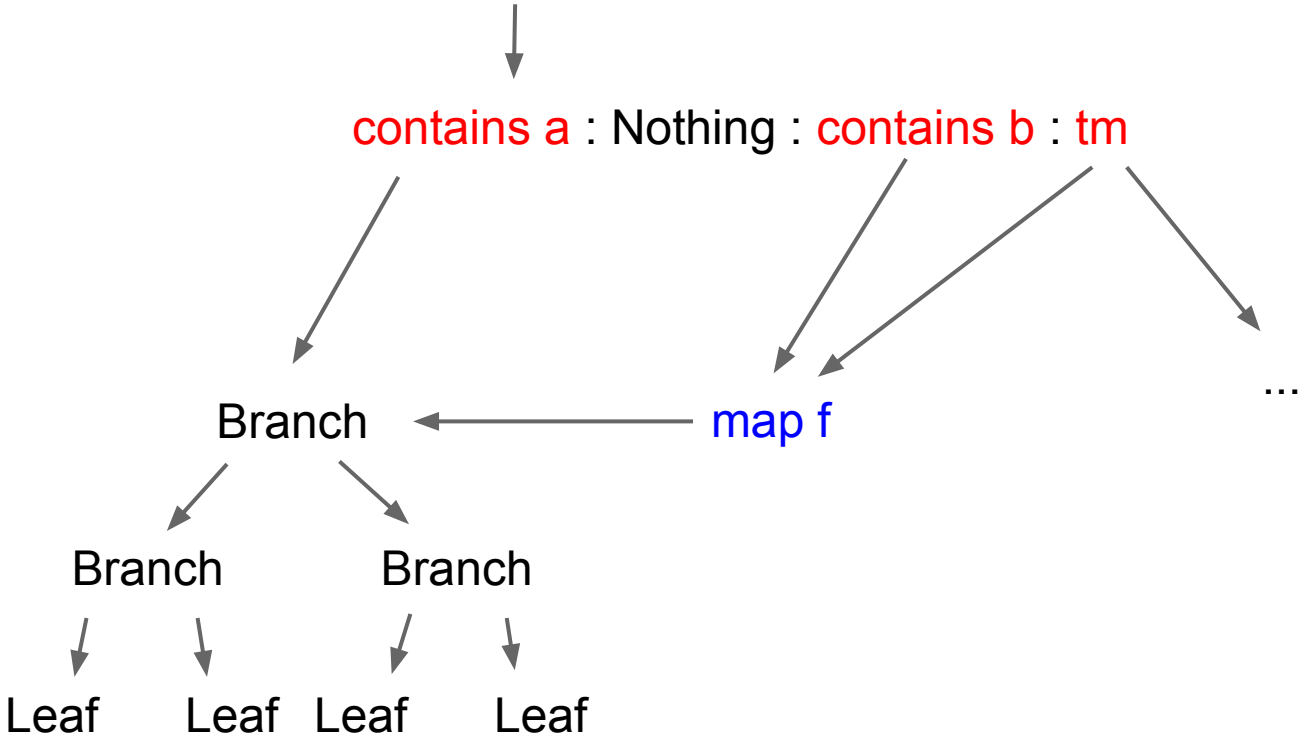
Concurrent Transaction Processing



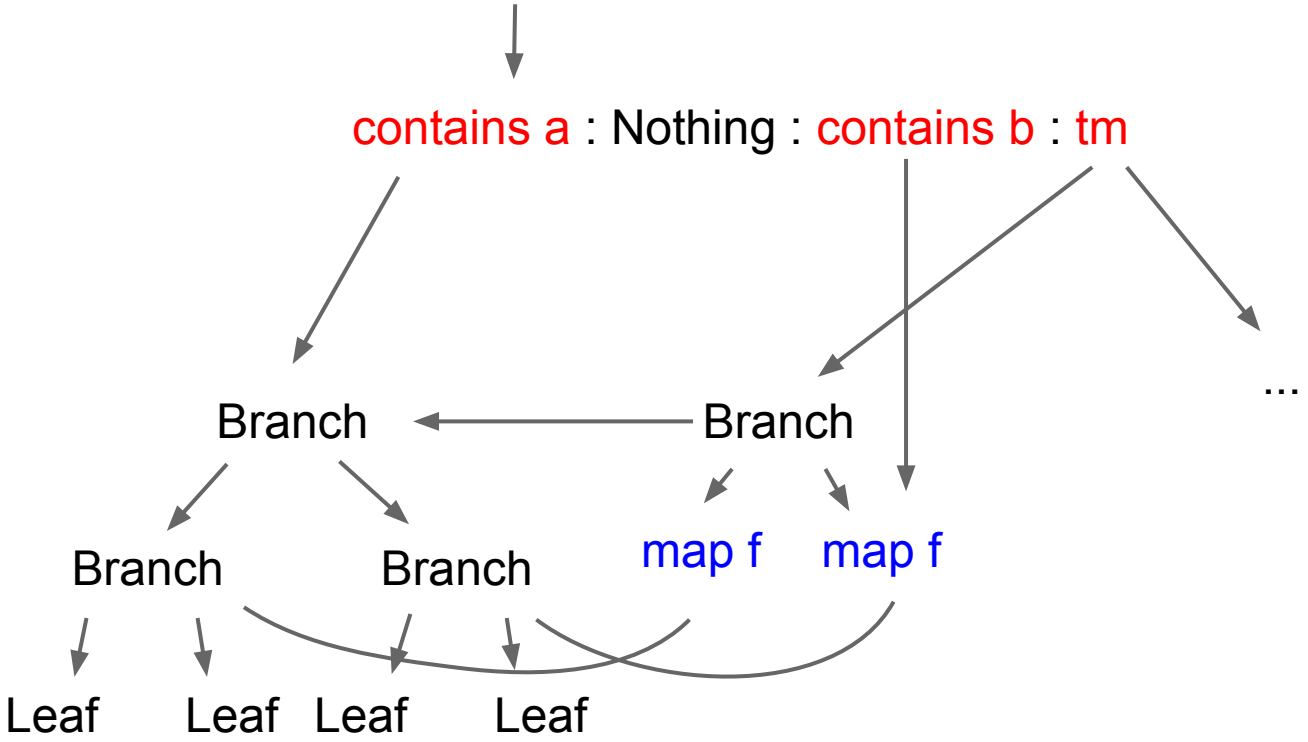
Concurrent Transaction Processing



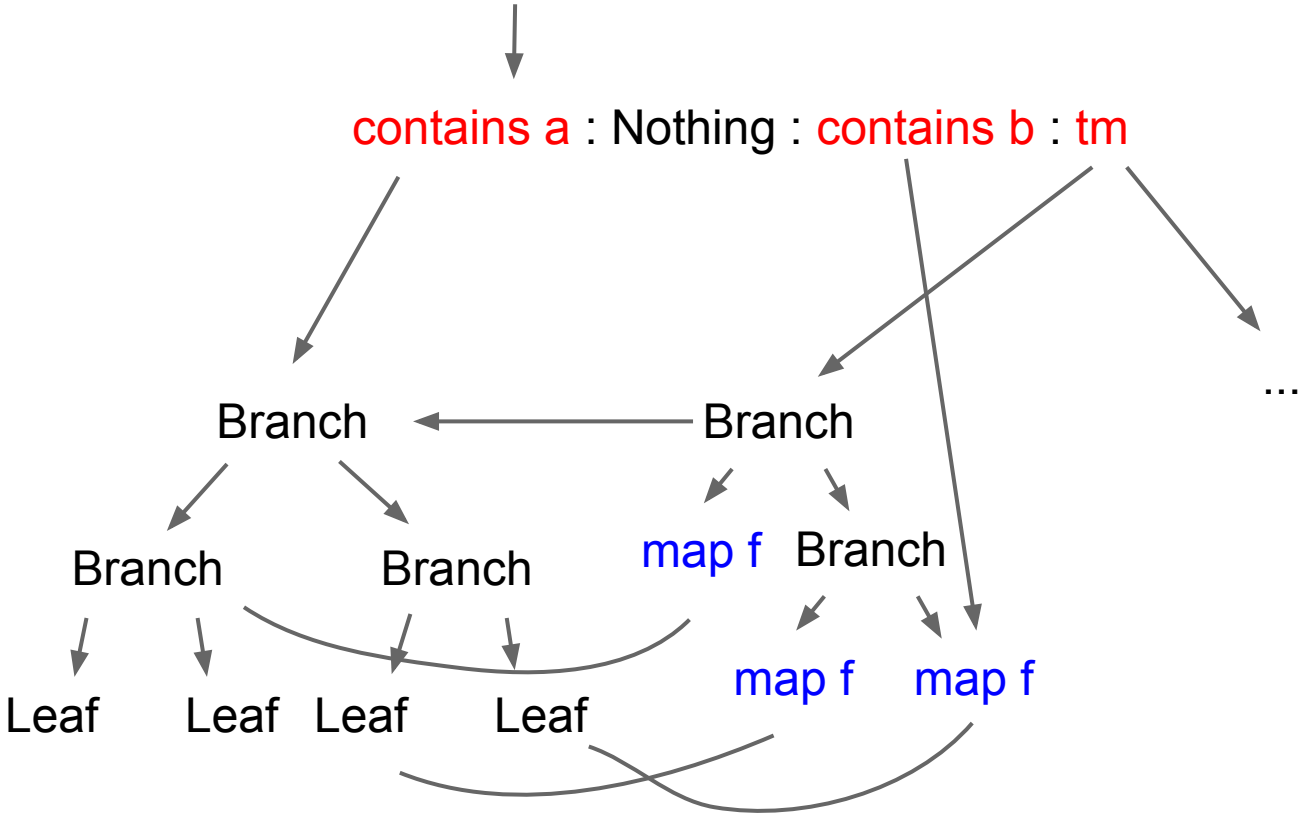
Concurrent Transaction Processing



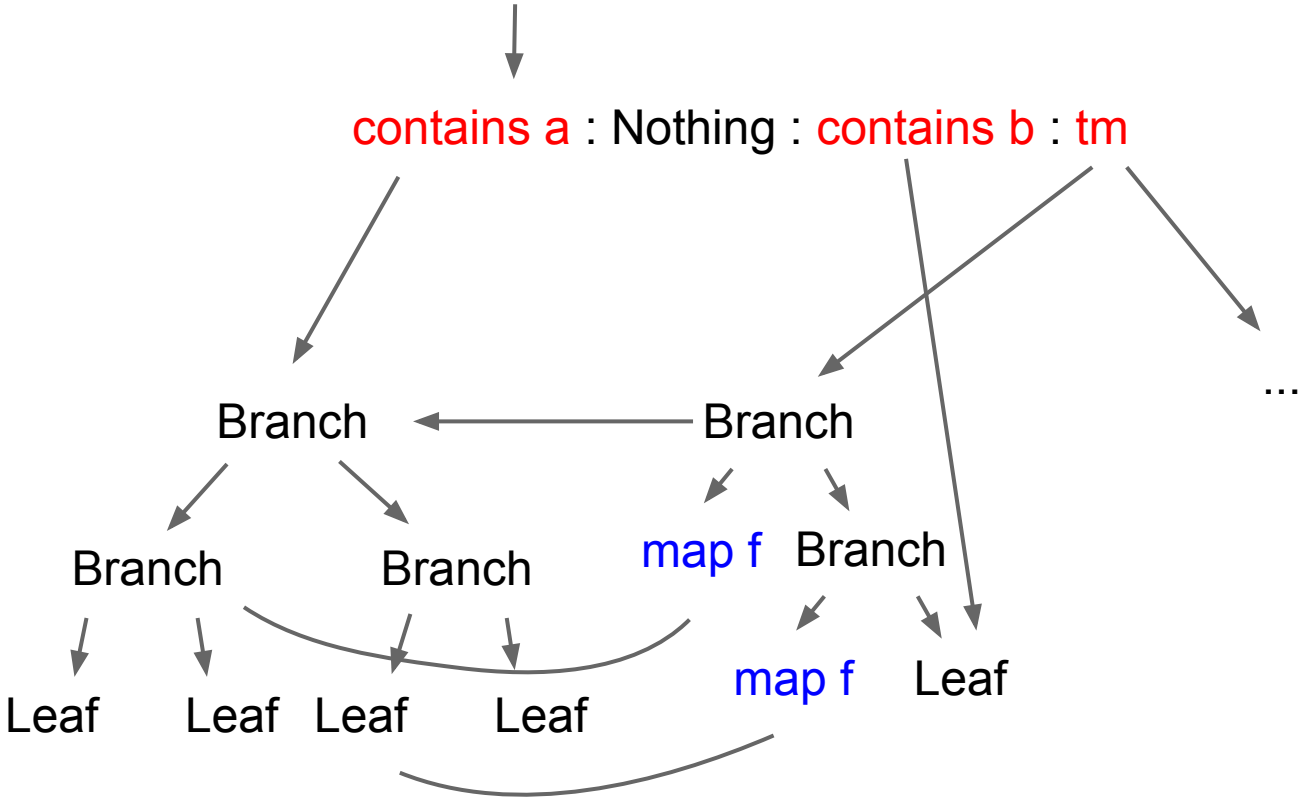
Concurrent Transaction Processing



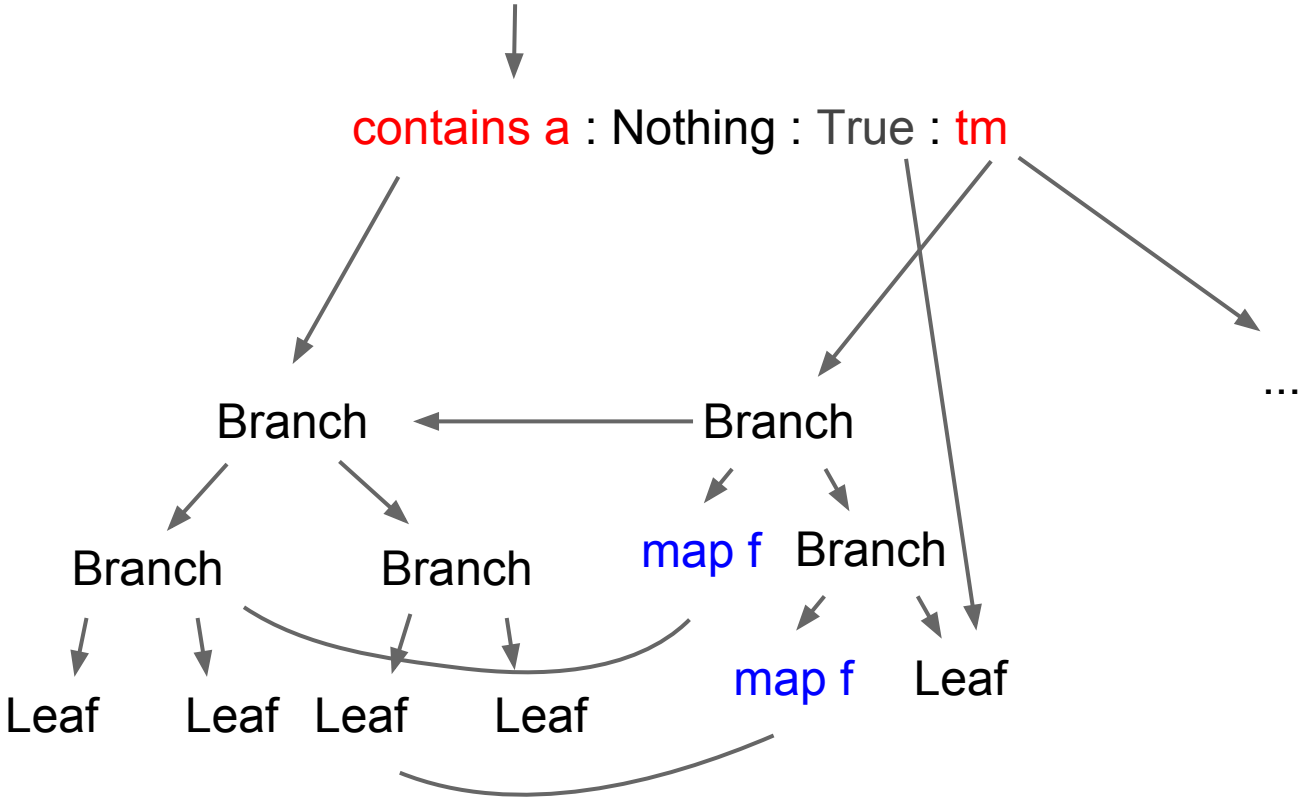
Concurrent Transaction Processing



Concurrent Transaction Processing



Concurrent Transaction Processing



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

{ names = [“alice”] }

names’ = “bob” : names

result = names

[“alice”]

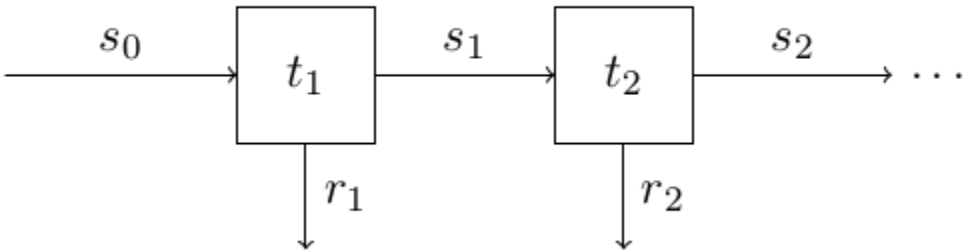
{ names = [“bob”, “alice”] }

Persistent Functional Language

```
{ names = [“bob”, “alice”] }
    length’ list = list match
        [] ->  0
        (x:xs) -> 1 + length’ xs
{ names = [“bob”, “alice”], length = λ list → ... }
    result = length names
    2
{ names = [“bob”, “alice”], length = λ list → ... }
```

Functional Transaction Processing

```
tm : DB x [Transaction] -> [Result]
tm s (tx:txs) =
   let (ns, r) = tx(s) in
      r : (tm ns 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

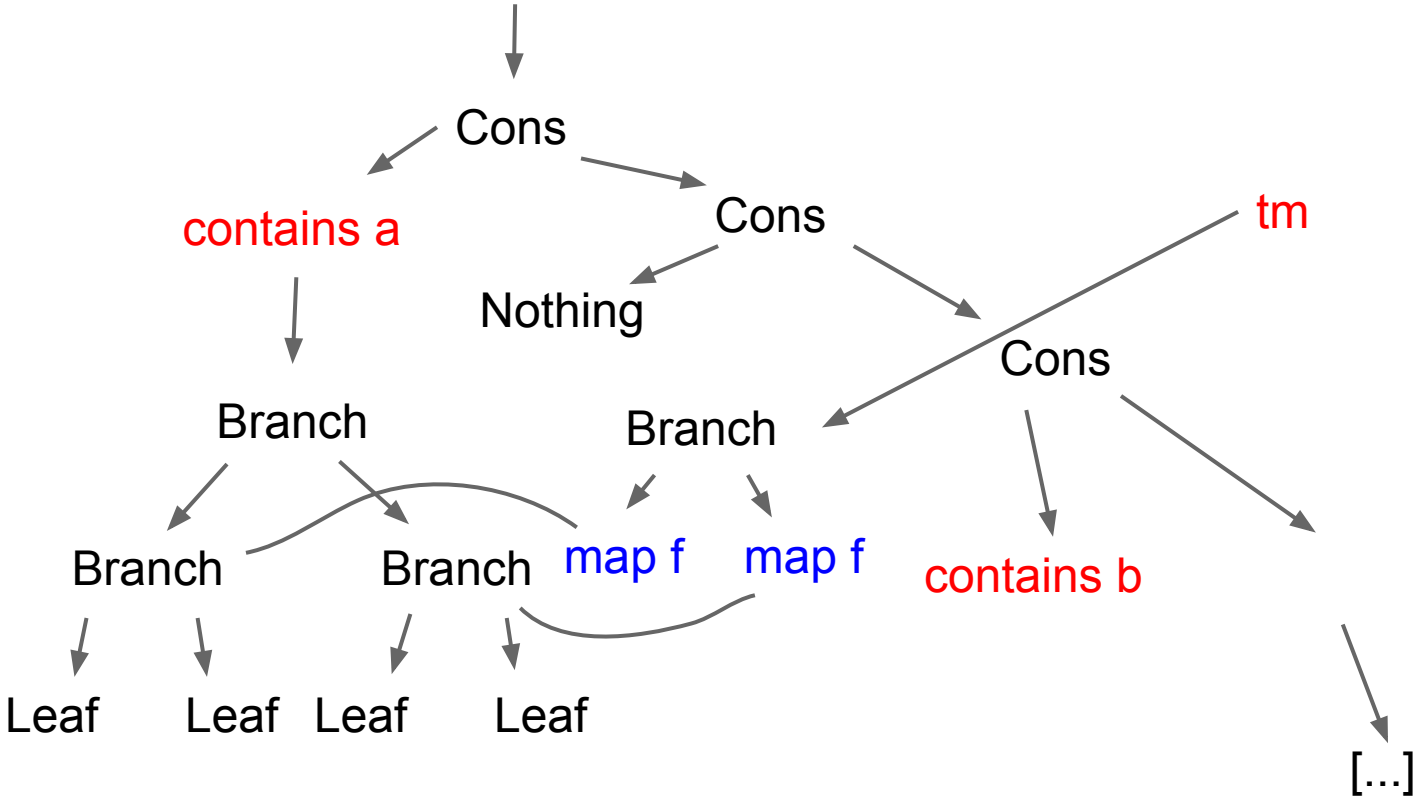
tm (Branch ...) [contains a, update, contains b, ...]

Future work: Memoization

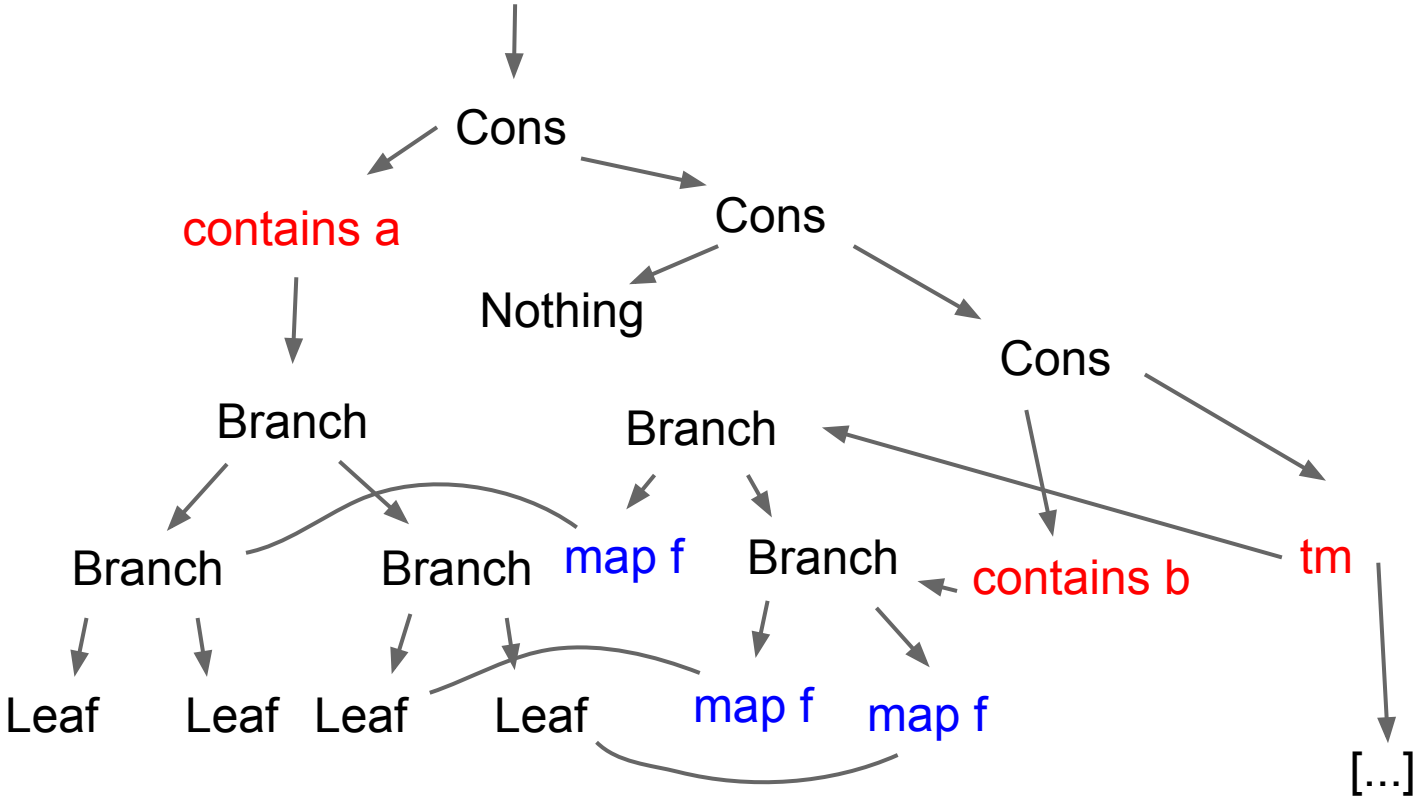
Remember results of function applications:

- Optimistic execution & retrying transactions
- Aggregate functions:
  - sum, min, max, …
  - constraint checks
- Materialized views

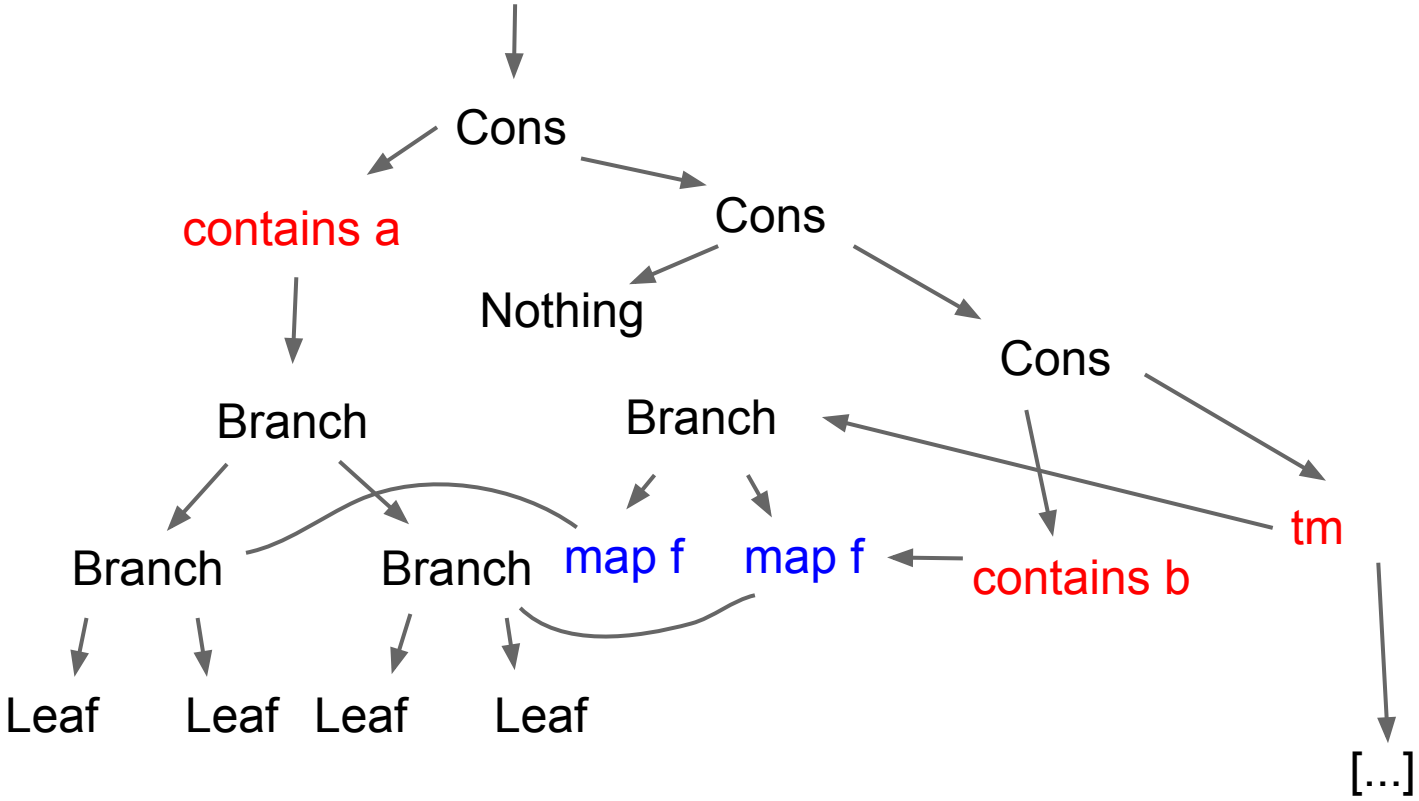
Concurrent Transaction Processing



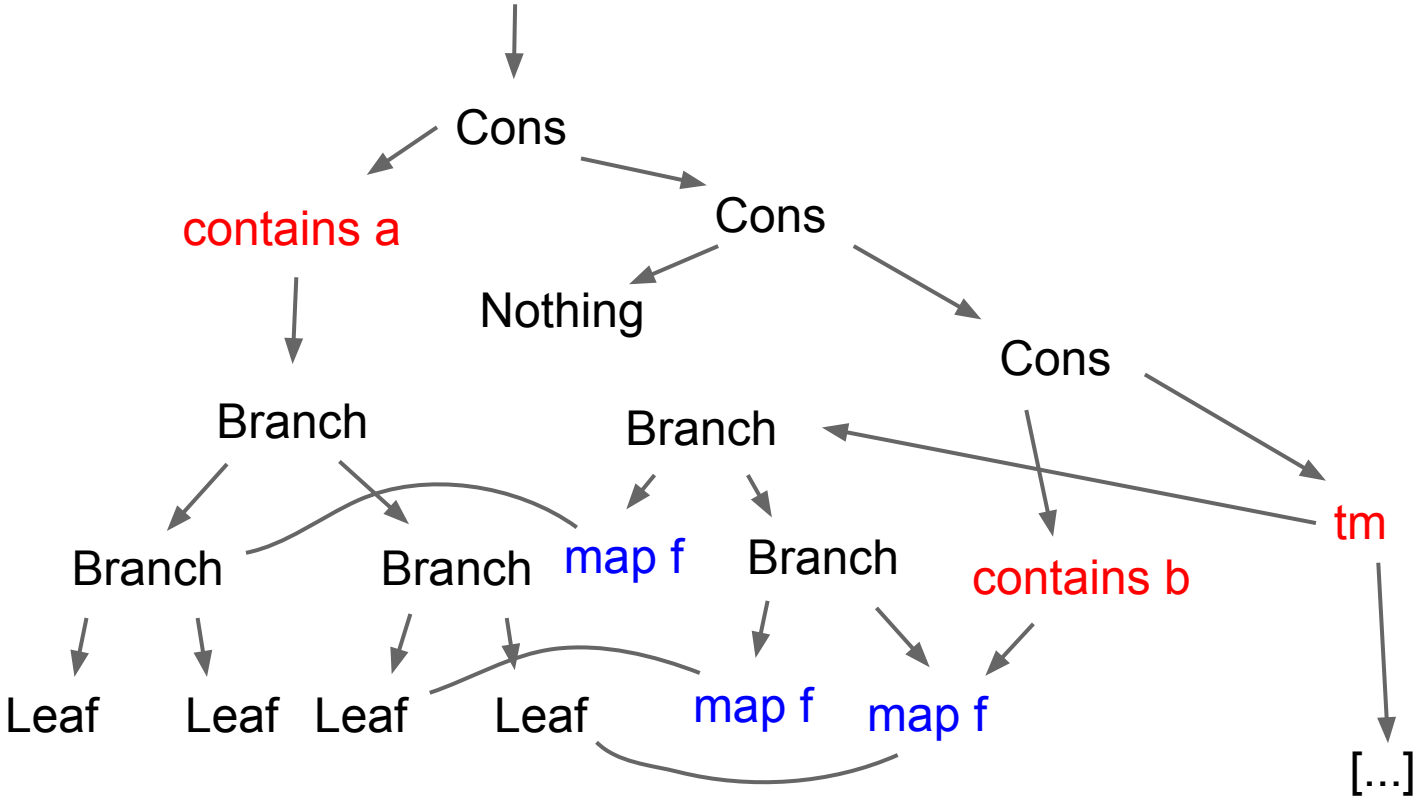
Concurrent Transaction Processing



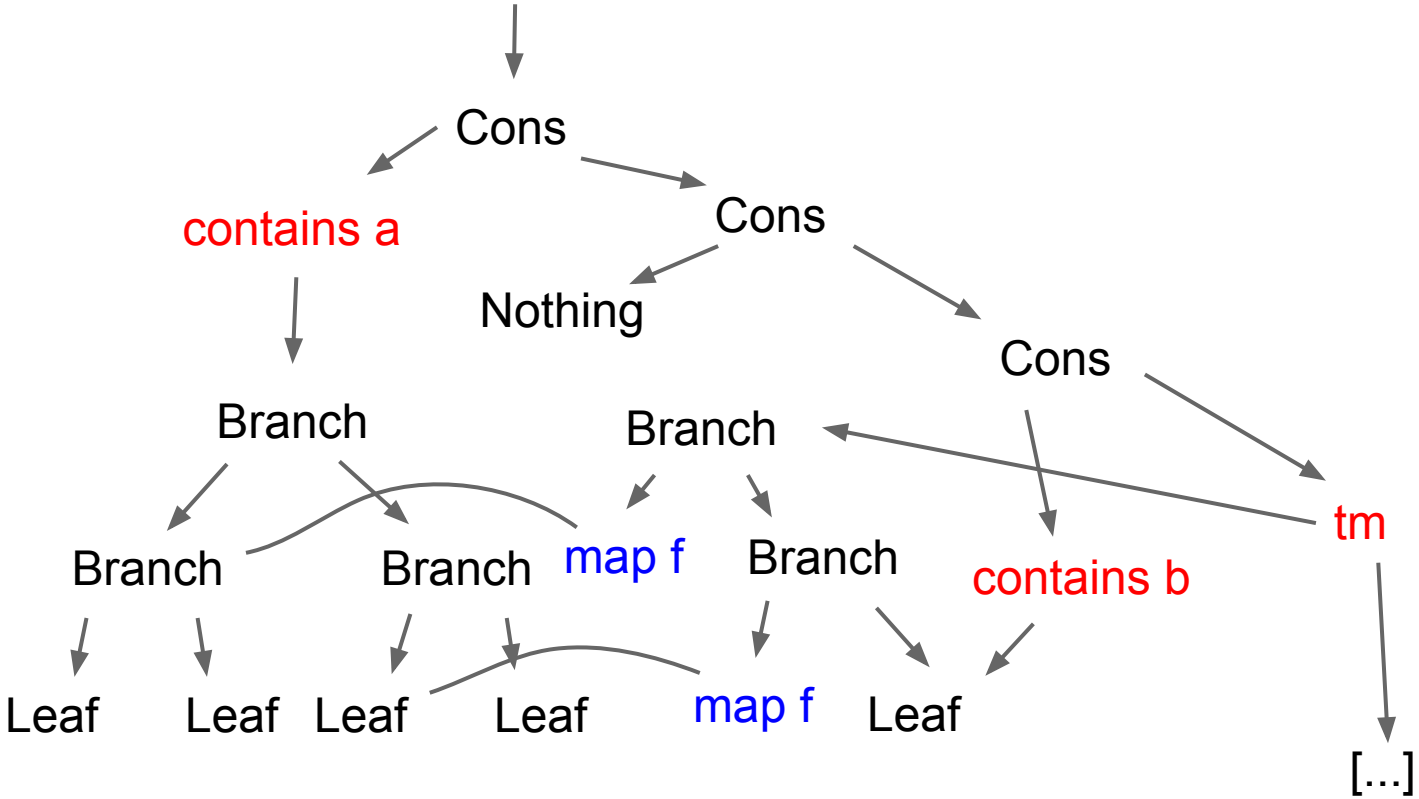
Concurrent Transaction Processing



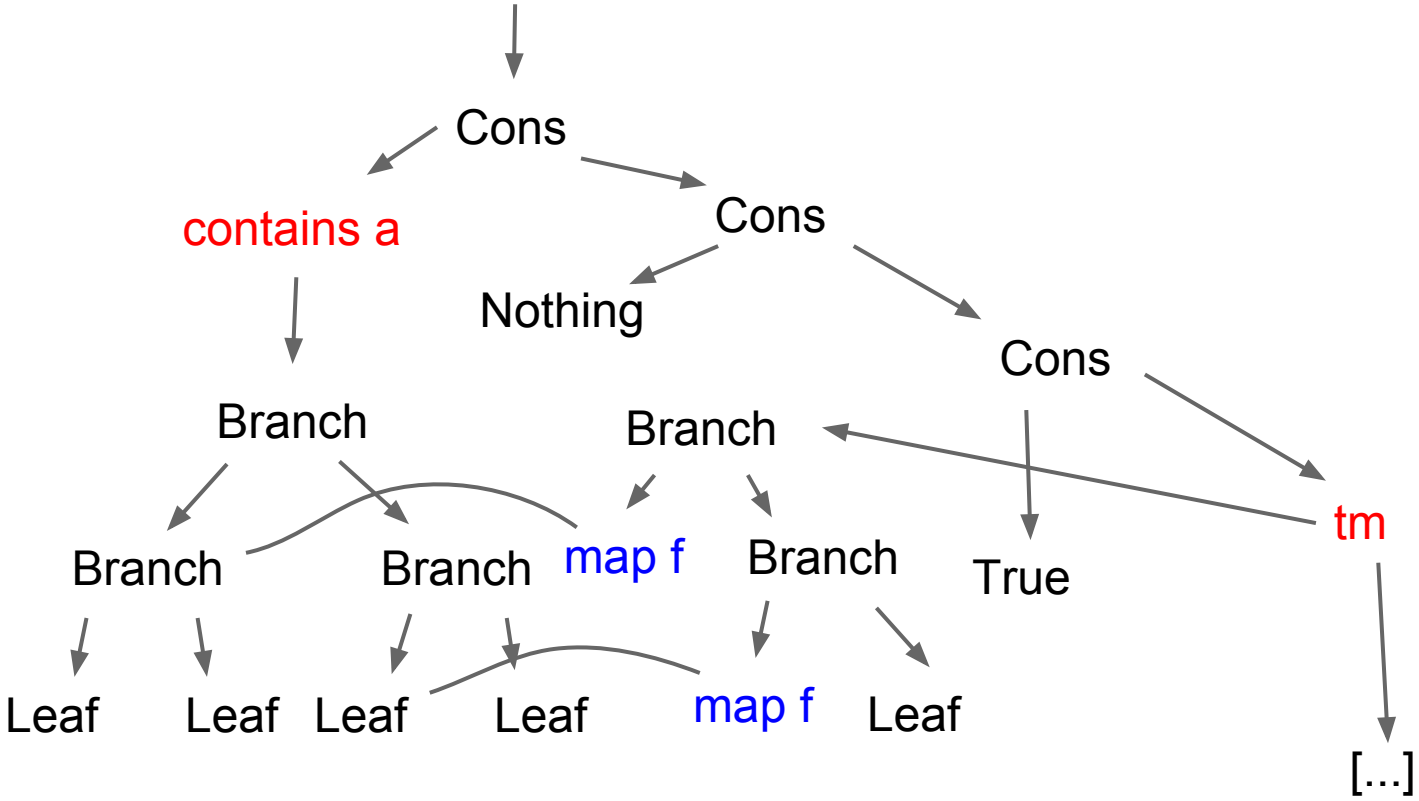
Concurrent Transaction Processing



Concurrent Transaction Processing



Concurrent Transaction Processing



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.