Evaluation of Temporal Queries
with Applications in Program Debugging

Presenter: Demian Lessa
Evaluation of Temporal Queries
with Applications in Program Debugging

1. Debugging with Temporal Queries

2. Query Evaluation: Compilation Approach

3. Query Evaluation: Recursive Temporal Queries
public class BST {

    private final int value;
    private BST left = null;
    private BST right = null;

    public BST(final int n) {
        value = n;
    }

    public void insert(final int n) {
        if (value == n) {
            return;
        }
        if (value < n) {
            if (right == null) {
                right = new BST(n);
            } else {
                right.insert(n);
            }
        } else if (left == null) {
            left = new BST(n);
        } else {
            left.insert(n);
        }
    }
}

- Sample debug questions
  - was there ever a path between...?
  - when was there a path between...?
  - was there ever an invariant violation?
Debugging with Temporal Queries

Example: Binary Search Tree

- JIVE's Object Diagram
  - run-time view of object heap
  - one diagram for each time in execution
  - supports visual debugging
  - supports back-in-time stepping
  - not scalable for large execution traces
  - solution: query-based debugging!
Query-based debugging
- schema: BST(id, key, lid, rid)
- consider the given program state
- recursive Datalog queries (non-temporal)

Q1: Is there a path 100 → K, K < 60?

Q1() :- Path(A,D),
       BST(A,100,_,_),
       BST(D,K,_,_), K < 60

% base cases: direct edges
Path(A,D) :- BST(_,_,D,_)  
Path(A,D) :- BST(_,_,_,D)

% recursive cases
Path(A,D) :- Path(A,N), BST(N,_,D,_)  
Path(A,D) :- Path(A,N), BST(N,_,_,D)
Debugging with Temporal Queries

Example: Binary Search Tree

- **BST Invariant:** given a node N
  - left subtree keys are *smaller* than key(N)
  - right subtree keys are *larger* than key(N)
  - left and right subtrees are BSTs

Q2: Is the BST invariant violated?

Q2() :- Left(A,D), BST(A,KA,_,_), BST(D,KD,_,_), KA < KD
Q2() :- Right(A,D), BST(A,KA,_,_), BST(D,KD,_,_), KA > KD

% D is a node in A's left subtree
Left(A,D) :- BST(A,_,D,_)
Left(A,D) :- Left(A,N), Path(N,D)

% D is a node in A's right subtree
Right(A,D) :- BST(A,_,_,D)
Right(A,D) :- Right(A,N), Path(N,D)
Debugging with Temporal Queries

Example: Binary Search Tree

- Query-based debugging
  - Q1 and Q2 only work for a fixed state
  - solution-- temporal approach

- Challenges
  - how do we incorporate time?
  - data model-- points, intervals, or...?
  - representation of temporal data
  - query language syntax and semantics?
  - query language expressiveness?
    - set and bag operations?
    - grouping and aggregation?
    - recursion?
Debugging with Temporal Queries
Example: Binary Search Tree

- Point-based temporal model
  - BST(id, key, lid, rid, time)
  - time is discrete and linearly ordered
  - conceptually simple
  - query formulation intuitive
  - materializing BST is impractical

Q1: When was there a path 100 → K, K < 60?

Q1(T) :- Path(A,D,T),
          BST(A,100,_,_,T),
          BST(D,K,_,_,T), K < 60

% base cases: direct edges
Path(A,D,T) :- BST(A,_,D,_,T)
Path(A,D,T) :- BST(A,_,_,D,T)

% recursive cases-- temporal equijoins!
Path(A,D,T) :- Path(A,N,T), BST(N,_,D,_,T)
Path(A,D,T) :- Path(A,N,T), BST(N,_,_,D,T)
Debugging with Temporal Queries

Example: Binary Search Tree

**Q1:** When was there a path 100 → K, K < 60?

% does not preserve set semantics! why?

Q1(TS,TE) :- Path(A,D,TSP,TEP),
BST(A,100,_,_,TSA,TEA),
BST(D,K,_,_,TSD,TED), K < 60,
% do intervals overlap? (not transitive!)
TSP < TEA, TSA < TEP,
TSP < TED, TSD < TEP,
TSA < TED, TSD < TEA,
% interval construction
TS = MAX(TSP,TSA,TSD),
TE = MIN(TEP,TEA,TED)

- Interval-based temporal model
  - BST(id, value, lid, rid, time_s, time_e)
  - time is discrete and linearly ordered
  - time_s < time_e
  - space-efficient representation
  - *query formulation much harder*

### BST

<table>
<thead>
<tr>
<th>Object</th>
<th>BST</th>
</tr>
</thead>
<tbody>
<tr>
<td>main:1</td>
<td>args: String[] java.lang.String[0] (id=21)</td>
</tr>
<tr>
<td></td>
<td>return point SYSTEM</td>
</tr>
</tbody>
</table>

```
BST(id, value, lid, rid, time_s, time_e)
```

- time is discrete and linearly ordered
- time_s < time_e
- space-efficient representation
- *query formulation much harder*
Debugging with Temporal Queries
Example: Binary Search Tree

- Interval-based temporal model
  - BST(id, value, lid, rid, time_s, time_e)
  - time is discrete and linearly ordered
  - time_s < time_e
  - space-efficient representation
  - query formulation much harder

Q1: When was there a path 100 → K, K < 60?

% base cases: direct edges
Path(A,D,TS,TE) :- BST(A,_,D,_,TS,TE)
Path(A,D,TS,TE) :- BST(A,_,_,D,TS,TE)

% recursive cases-- temporal equijoins!
Path(A,D,TS,TE) :- Path(A,N,TSP,TEP),
  BST(N,_,_,D,_,TSN,TEN),
  TSP < TEN, TSN < TEP,
  TS = MAX(TSP, TSN), TE = MIN(TEP, TEN)
Path(A,D,TS,TE) :- Path(A,N,TSP,TEP),
  BST(N,_,_,D,_,TSN,TEN),
  TSP < TEN, TSN < TEP,
  TS = MAX(TSP, TSN), TE = MIN(TEP, TEN)
Evaluation of Temporal Queries with Applications in Program Debugging

1. Debugging with Temporal Queries

2. Query Evaluation: Compilation Approach

3. Query Evaluation: Recursive Temporal Queries
Query Evaluation
Compilation Approach

- Temporal Model
  - Abstract Temporal Database (ATDB)
  - Concrete Temporal Database (CTDB)

- ATDB
  - point-based, representation independent
  - possibly infinite, but finitely representable
  - only part of the temporal database exposed to users

- CTDB
  - interval-based encoding of the ATDB
  - used internally, transparent to users
  - an actual SQL/99 RDBMS, so we can leverage existing technology

- Challenge
  - how do we execute point-based queries against the CTDB?
  - through a query compilation technique
Query Evaluation
Compilation Approach

- Semantic mapping, \| | |
  - maps CTDB elements to respective ATDB elements

- Compilation procedure
  - input: point-based temporal query Q
  - output: interval-based temporal query \textit{compile}(Q)

- Query evaluation
  - \textit{compile}(Q) is evaluated against the CTDB
  - \textit{concrete tuples} are returned to user
Query Evaluation

Compilation Approach

```
All ATDBs

Finitely Representable ATDBs

\{ \langle a,1 \rangle, \langle a,2 \rangle, \langle a,3 \rangle, \ldots \} \quad \varphi = R(x,t) \land t < 10 \quad \{ \langle a,1 \rangle, \langle a,2 \rangle, \langle a,3 \rangle, \ldots, \langle a,9 \rangle \} \\

\| E_2 \| \quad \| E_1 \| \quad \| \cdot \| \quad \| \cdot \|

\{ \langle a,1,6 \rangle, \langle a,6,\infty \rangle \} \quad \text{compile}(\varphi)(E_1) \quad \{ \langle a,1,6 \rangle, \langle a,6,10 \rangle \} \\
\{ \langle a,1,\infty \rangle \} \quad \text{compile}(\varphi)(E_2) \quad \{ \langle a,1,10 \rangle \} 

Interval Encoded CTDBs
```
Query Evaluation
Compilation Approach

● Guarantee: compilation preserves semantics w.r.t. ATDBs
  • for every CTDB D, \(|| \text{compile}(Q)(D) || = Q(|| D ||)\)
  • non-trivial!

● Challenges
  • mapping points to intervals: quantifier elimination, well studied
  • however, not sufficient to guarantee preservation of semantics w.r.t. ATDBs

● What is the problem?
  • under set semantics, concrete queries must return disjoint intervals
  • otherwise, we will observe several undesirable consequences...
  • set/bag operations:
    – e.g., expected empty set but tuples are returned
  • duplicate elimination:
    – e.g., [5, 10) is a duplicate if [1, 100) is in the result!
  • aggregation:
    – e.g., inconsistent sums/counts
  • recursion:
    – e.g., blow-up in space/time complexity of the bottom-up evaluation
Time compatibility using the N operator: set difference example
Query Evaluation
Compilation Approach

- Time compatibility using the N operator: set union example
Query Evaluation
Compilation Approach

• Use of N in the compilation of non-recursive queries
  • set/bag operations, grouping, aggregation, duplicate elimination, joins
  • → SQL/TP

• However, recursive queries are not supported...

• N Operator (intuition)
  • collects left (L) and right (R) interval endpoints of input relations
  • splits output relation intervals according to minimal intervals obtained from L and R
  • → must reference each input relation at least once to build L and R
  • → must introduce negation to guarantee minimality

• What is the problem?
  • syntactically, a recursive query is formulated as a union
  • compiling a recursive query introduces the N operator
  • the compiled recursive query is non-linear and has non-stratified negation!
  • SQL/99 and later engines cannot evaluate such queries
  • a more general solution to the bottom-up evaluation is required
Evaluation of Temporal Queries with Applications in Program Debugging

1. Debugging with Temporal Queries

2. Query Evaluation: Compilation Approach

3. Query Evaluation: Recursive Temporal Queries
Query Evaluation
Recursive Temporal Queries

- **Dilemma**
  - using N: semantics w.r.t. ATDBs is preserved but cannot evaluate queries
  - not using N: can evaluate queries but semantics w.r.t. ATDBs is lost
  - our approach:
    - modified compilation to drop the use of N for recursive predicates
    - modified bottom-up evaluation to guarantee preservation of semantics w.r.t. ATDBs

- **Modified Compilation**
  - do not use N for recursive predicates
  - modified bottom-up evaluation code is incorporated in the compiled query
  - cannot be done in plain SQL-- produce a (database) function instead
  - introduce optimizations, e.g., magic sets, index creation, etc

- **Modified Bottom-Up Evaluation**
  - semantics w.r.t. ATDBs is preserved at every stage
  - evaluation terminates in finitely many steps
  - no redundant computation, i.e., new temporal facts are generated at every stage
Query Evaluation
Recursive Temporal Queries

- Traditional Bottom-Up Evaluation
  - $I, J = \emptyset$
  - repeat
    - $J = I$
    - $I = T_p(I)$
  - until $I = J$
  - return $I$;

- Details
  - fixpoint computation
  - based on the immediate consequence operator, $T_p$
  - $T_p$ derives new ground facts from existing ground facts
  - termination: DB has finitely many symbols, no new symbols are introduced
Query Evaluation
Recursive Temporal Queries

- Normalizing Bottom-Up Evaluation
  - $I, J = \emptyset$
  - **repeat**
  - $J = I$
  - $I = T_{NP}(I)$
  - **until** $|I| = |J|$
  - **return** $I$;

- Details
  - fixpoint computation
  - based on the normalizing immediate consequence operator, $T_{NP}$
  - $T_{NP}$ derives new temporal ground facts from existing temporal ground facts
    - guarantees that set unions preserve semantics w.r.t. ATDBs
  - $T_{NP}$ is not sufficient to guarantee termination or non-redundant computation
    - in general, $J \not\subseteq I$ due to representation differences at consecutive stages
    - however, $|J| \subseteq |I|$ holds at every stage, based on the correctness of $T_{NP}$
  - termination: based on the termination of constraint Datalog programs
Abstract relation \( \text{Refs}(O, R, T) \) keeps track of the instants in which object \( O \) references object \( R \) at run-time. \( \text{Refs}(O, R, T_s, T_e) \) is its concrete counterpart.

Now, consider the following \textit{temporal transitive closure} query:

\[
\text{TTC}(X,Y,T) :- \text{Refs}(X,Y,T) \]
\[
\text{TTC}(X,Z,T1), \text{Refs}(Z,Y,T), T > T1 
\]

The point-to-interval translation performed by the compiler yields:

\[
\text{TTC}(X,Y,Ts,Te) :- \text{Refs}(X,Y,Ts,Te) \]
\[
\text{TTC}(X,Z,T1s,T1e), \text{Refs}(Z,Y,T2s,Te), \text{Ts} = \text{MAX}(T2s,T1s+1) 
\]
\[
\text{Te} > T1s+1 \text{ implies that there exists } T \in [T2s,Te) \text{ s.t. } T > T1s 
\]
\[
\text{Ts} = \text{MAX}(T2s,T1s+1) \text{ is the smallest left endpoint contained in } [T2s,Te) 
\]
Now assume that \( \text{Refs} \) contains a single tuple, \(<1, 1, 1, 2^{k+1}>\), for \( k > 0 \).

Our modified evaluation of the concrete query produces:

\[
\begin{align*}
T_{\text{NP}} \uparrow 1 &= \{<1,1,1,2^{k+1}>\} \\
T_{\text{NP}} \uparrow 2 &= \{<1,1,1,2>, <1,1,2,2^{k+1}>\}
\end{align*}
\]

\( || T_{\text{NP}} \uparrow 1 || = || T_{\text{NP}} \uparrow 2 || \), i.e., both represent the same ATDB
Using $T_{NP}$ alone...

- does not guarantee termination for infinite ATDBs, e.g., $<1,1,1,+\infty>$
- causes blowup, e.g., $||J|| = ||I||$ may hold early on even though $J \neq I$
- re-evaluating the previous example...

\[
T_{NP} \uparrow 1 = \{<1,1,1,2^{k+1}>\} \quad \text{% stage #1, base case}
\]
\[
T_{NP} \uparrow 2 = \{<1,1,1,2>,<1,1,2,2^{k+1}>\} \quad \text{% stage #2}
\]

\ldots

\[
T_{NP} \uparrow 2^k = \{<1,1,1,2>,<1,1,2,3>,\ldots,<1<1,1,2^k,2^k+1>\} \quad \text{% stage #2^k}
\]
\[
T_{NP} \uparrow 2^{k+1} = \{<1,1,1,2>,<1,1,2,3>,\ldots,<1<1,1,2^k,2^k+1>\} \quad \text{% stage #2^{k+1}}
\]
Using the modified termination condition alone...
- causes space blowup, e.g., at every stage, I and J may contain temporal duplicates
- re-evaluating the previous example...

\[
\begin{align*}
T_{NP} \uparrow 1 &= \{<1,1,1,2^{k+1}>\} & \text{% stage #1, base case} \\
T_{NP} \uparrow 2 &= \{<1,1,1,2^{k+1}>,<1,1,2,2^{k+1}>\} & \text{% stage #2} \\
\ldots & \\
T_{NP} \uparrow 2^k &= \{<1,1,1,2^{k+1}>,<1,1,2,2^{k+1}>,\ldots,<1<1,1,2^k,2^{k+1}>\} & \text{% stage #2^k} \\
T_{NP} \uparrow 2^{k+1} &= \{<1,1,1,2^{k+1}>,<1,1,2,2^{k+1}>,\ldots,<1<1,1,2^k,2^{k+1}>\} & \text{% stage #2^{k+1}}
\end{align*}
\]
Query Evaluation
Recursive Temporal Queries

- Applications in debugging and program comprehension.
  - questions about temporal state of recursive data structures (Q1, Q2)
  - general questions about object relationships (TTC)
  - our main focus: query-based dynamic analysis!

- Dynamic Analysis
  - analysis of the properties of running programs
  - characteristics: precision, input dependence
  - e.g., dynamic slicing
    - given a variable V and program location L
    - determine the program statements that affected the value of V at L
  - can be implemented as a temporal recursive query
  - further applications
    - scaling our tool's visualizations by removing regions unrelated to the slice(s)
    - enhancing our tool's visual debugging capabilities
Thank You!