Declarative Debugging

PROMETIDOS SUMMER SCHOOL
Rafael Caballero Roldán
Universidad Complutense de Madrid
Summary

- Introduction
- Computation Trees
- Functional-Logic paradigm
- Relational Databases: SQL views
- The strange case of Datalog and Declarative Debugging
Introduction
Buggy Systems

SOFTWARE SYSTEM

Intended Semantics
Buggy Systems

SOFTWARE SYSTEM
Buggy Systems

Initial Symptom
Buggy Systems

Where is the bug?
Declarative Debugging

- Also known as **algorithmic debugging**

**Characteristics:**
- A general, paradigm independent, framework
- Detect erroneous components in software systems
- Black Box approach: the components code is not important, only their **intended semantics**
Declarative Debugging

The oracle
Declarative Debugging

The oracle

Concrete Computation
Declarative Debugging

The oracle

Trusted specification

Concrete Computation
Declarative Debugging

The oracle

Concrete Computation

The Poor User
Detecting a component $C$ producing a non-valid result

- Implies that there is a bug
- But...the bug is not necessarily in $C \rightarrow$ components are interconnected!

http://www.imdb.com/title/tt0298879/
Idea 1: Detect a component C such that
- C contains a non-valid result, and such that
- All the components connected to C contain valid results

Idea 2: Employ some data structure (usually a tree) in order to represent the conexions between components in an erroneous computation
Declarative Debugging

- Declarative Debugging has been applied to:
  - Declarative Languages
    - Logic Programming (Shapiro, 1984)
    - Functional Programming (Nilsson, Sparoud, Naish)
    - Rewriting Systems (Maude -> Martí, Riesco, Verdejo)
    - Functional-Logic programming
    - Deductive Databases
  - Object Oriented Programming
  - Relational Databases (SQL views)
Computation Trees
**Computation Trees**

- Represent the computation that produced the unexpected result
- Each node contains:
  - An intermediate result \( R \) (the initial symptom in the case of the root)
  - A reference to the component (fragment of code) that produced \( R \)
- Children of a node \( N \):
  - Contain the results needed to obtain the result at \( N \)
Computation Trees

Example:

\[
\begin{align*}
good(tweety) \\
good(silvester) \\
loyal(tweety) \\
trustable(X) & :\ - \ good(X) \\
trustable(X) & :\ - \ loyal(X) \\
likes(tweety,silvester) \\
likes(silvester,tweety) \\
goodFriend(X,Y) & :\ - \ likes(X,Y), \ trustable(X)
\end{align*}
\]
Goal: \textit{goodFriend}(X,Y)

First answer \{ X \rightarrow \text{tweety}, \quad Y \rightarrow \text{silvester} \}

Second answer \{ X \rightarrow \text{tweety}, \quad Y \rightarrow \text{silvester} \}

Third answer \{ X \rightarrow \text{silvester}, \quad Y \rightarrow \text{tweety} \}
Computation Trees

goodFriend(silvester,tweety)

likes(silvester,tweety)  trustable(silvester)

  good(silvester)
Computation Trees

\( \text{goodFriend(silvester,tweety)} \)

\( \text{likes(silvester,tweety)} \) \quad \text{trustable(silvester)} \quad \text{good(silvester)} \)
goodFriend(silvester,tweety)

likes(silvester,tweety)?

trustable(silvester)

good(silvester)
Computation Trees

goodFriend(silvester,tweety)

likes(silvester,tweety)  trustable(silvester)?

good(silvester)
Computation Trees

\[
\text{goodFriend(silvester,tweety)}
\]

\[
\text{likes(silvester,tweety)} \quad \text{trustable(silvester)}
\]

\[
\text{good(silvester)?}
\]
The goal of declarative debugging is to find a **buggy node**: non-valid node with only valid children.

- A non-valid leaf is always buggy.
- Assuming an non-valid root there is at least a buggy node connected by the root by a path of non-valid nodes.
Computation Trees

- Completeness property of declarative debugging

Every computation tree with non-valid root (initial symptom) contains at least one buggy node

- Easy to prove by induction on the size of the tree
Computation Trees

How to obtain computation trees?

1. Define a program transformation
   - The components return their own computation trees
   - Pros: Convenient for proving correctness
   - Cons: Slow

2. Modify the runtime procedure of the system
   - The system produces the computation tree when executing the goal in debug mode
Program transformation for the example?
good(tweety)
good(silvester)
loyal(tweety)
trustable(X) :- good(X)
trustable(X) :- loyal(X)
likes(tweety,silvester)
likes(silvester,tweety)
goodFriend(X,Y) :- likes(X,Y), trustable(X)
Declarative Debugging Schema

Declarative debugging consists of the following phases:

1. An initial symptom is detected (by the user)
2. A computation tree for this computation is built (by the debugger)
3. The tree is navigated (interaction user/oracle and debugger)
4. A buggy node is detected, and its associated component pointed out as erroneous (by the debugger)
Declarative Debugging Schema

- From the theoretical point of view:
  - Completeness is ensured
  - Soundness must be established
    - Prove that a buggy node implies an incorrect component
  - If possible, it is interesting to prove that the debugger generates suitable computation trees
Declarative Debugging Schema

- Soundness. **Idea:** Use some semantic proof calculus, such that
  - **Parent node** → conclusion of an inference
  - **Children** → premises of the same inference

$$P_1 \ldots P_n \quad \rightarrow \quad C$$

$$\begin{array}{c}
\text{P}_1 \ldots \text{P}_n \\
\text{C}
\end{array} \rightarrow \quad C$$
Declarative Debugging Schema

Logic Programming

\[
\frac{P_1 \Theta \ldots P_n \Theta}{C \Theta} \quad (\text{MP}) \quad P ::= P_1 \ldots P_n \quad \text{a program clause}
\]
Declarative Debugging Schema

- Logic Programming

\[ P_1 \Theta \quad \ldots \quad P_n \Theta \quad \text{(MP)} \quad P : = P_1 \ldots P_n \text{ a program clause} \]

- Buggy Node: the instance \( P_1 \Theta \ldots P_n \Theta \rightarrow C \Theta \) is not valid in the intended model \( \rightarrow \) the clause is not valid in the intended model
Different Types of Errors

- The example above presents a **wrong answer**
- The cause was an **incorrect clause** (Lloyd)
- But bugs can also generate **missing answers**
- In this case we have an **uncovered atom** and an **incomplete predicate**
- Missing answers are more **difficult to debug**
Example:

\[\text{good(tweety)}\]
\[\text{good(grandma)}\]
\[\text{trustable(X) :- good(X)}\]
\[\text{likes(tweety,grandma)}\]
\[\text{likes(tweety,silvester)}\]
\[\% \text{ forgotten} \]
\[\% \text{likes(grandma,tweety)} \]
\[\text{goodFriend(X,Y) :- likes(X,Y), trustable(X)}\]
Goal: GoodFriend(X,Y)

Answer 1: { X -> tweety, Y -> grandma }
Answer 2: { X -> tweety, Y -> silvester }

Missing answer!

Answer 3: { X -> grandma, Y -> tweety}
Missing Answers

- The same computation tree?
- Program transformation?
- The same inference rule?
Limitations

- Non-terminating (buggy) computations
  - no computation tree
  - declarative debugging not applicable
Limitations

- Concurrent programming
  - Many different trees that interact
Drawbacks

- Sometimes it is difficult to determine the validity of intermediate results → complex questions

http://currentconfig.com/2005/02/22/essential-life-lesson-1-over-is-right-under-is-wrong/
Drawbacks

- The system can ask too many questions to the user
Drawbacks

- Reducing the number of questions → navigation strategies
- Idea: choose carefully the node that is selected at each step
- See works by Silva & Insa
Divide & Query Strategy

Valid

Non valid
Drawbacks

- Reducing the number of questions
- Replace the user as oracle when possible:
  - Using an old version of the program that worked properly → trusted specifications
  - Using assertions
  - Introduce auxiliary nodes that combine the results of several nodes
Functional-Logic Paradigm
Functional-Logic Paradigm

- Combines features of Functional and Logic Programming:
  - Logic Variables
  - Non-determinism
  - Higher-order values
  - Lazyness
Program Example

data nat = z | suc nat

take :: nat → [A] → [A]
take z Xs = []
take (suc N) [] = []
take (suc N) (X:Xs) = X : take N Xs

from :: nat → [nat]
from X = X : from X ← debería ser X : from (suc X)

Goal: take (suc (suc z)) (from X) == R

Answer: R → X:X:[ ] Wrong Answer
Semantic Calculus

\[
\begin{align*}
\text{BT} & \quad e \rightarrow \bot \\
\text{RR} & \quad X \rightarrow X \text{ con } X \in V ar \\
\text{DC} & \quad \frac{e_1 \rightarrow t_1 \ldots e_m \rightarrow t_m}{h \overline{t}_m \in \text{Pat}_\bot} \quad h \overline{e}_m \rightarrow h \overline{t}_m \\
\text{JN} & \quad \frac{e \rightarrow t \quad e' \rightarrow t}{t \in \text{Pat} \text{ (patrón total)}} \quad e \equiv e' \\
\text{AR+FA} & \quad \frac{e_1 \rightarrow t_1 \ldots e_n \rightarrow t_n}{f \overline{t}_n \rightarrow s \quad s \overline{a}_k \rightarrow t} \quad f \overline{e}_n \overline{a}_k \rightarrow t \\
& \quad t \neq \bot \quad (f \overline{t}_n \rightarrow r \leftarrow C') \in [P]_\bot
\end{align*}
\]
Abbreviated Proof Tree

\[
\text{take } (\text{suc } (\text{suc } z)) \text{ (from } X) \to X:X:[]
\]

\[
\text{from } X \to X:X: \perp
\]

\[
\text{take } (\text{suc } (\text{suc } z)) \text{ (X:X:} \perp\text{) } \to X:X:[]
\]

\[
\text{from } X \to X: \perp
\]

\[
\text{take } (\text{suc } z) \text{ (X:} \perp\text{) } \to X:[]
\]

\[
\text{take } z \perp \to []
\]
Abbreviated Proof Trees

- Computation trees contain unnecessary nodes.

- The abbreviated proof trees:
  - Only includes those nodes that can be potentially buggy.
  - Independent of the execution order.
  - Include the arguments in their most evaluated forms (simplifies questions).
Implementation

- Based on Program Transformation
- Structure of the debugger
Relational Databases: SQL Views
SQL Views

- SQL views defined in terms of tables and other views
  - Set of correlated SQL views
  - Debugging in this context becomes is a difficult task
  - Computation Tree ~ Syntactic Dependency Tree

```
create view V(...) as
select ...
from V1, V2, ....
where ....;

create view V2(...) as
select ...
from V3, ....
where ....;
```
The trees are usually small (less than 100 views)

Both missing and wrong are covered → all the possible errors (queries are always terminating)

Problem: answers can contain thousands of rows

Test-Case generator included in the implementation available in DES

Trusted specifications
We improve $S$ because...

It is not efficient or it must be extended with new requirements

- $S$ substitutes the user as oracle when possible
- Reduces the number of questions asked to the user

Set SQL view definitions works correctly

$S'$ = new version of $S$

Does not work

$V_1$
$V_2'$
...
$V_n$

$V_1$
$V_2$
...
$V_n$
Implementations

- SQL Test Generator + Declarative Debugger included in the Datalog System DES
- Prototype in Java
The Dog and Cat Club annual dinner is going to take place in a few weeks, and the organizing committee is preparing the dinner guest list. Each year they browse the database of the All Pets Club looking for people that own at least one cat and one dog. This year two additional constraints have been introduced:

- People owning more than 5 animals are not allowed (the dinner would become too noisy).
- No animals sharing the same name are allowed at the party. This means that if two people have a cat or dog sharing the same name neither of them will be invited.
SQL Example: Tables

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mark Costas</td>
</tr>
<tr>
<td>2</td>
<td>Helen Kaye</td>
</tr>
<tr>
<td>3</td>
<td>Robin Scott</td>
</tr>
<tr>
<td>4</td>
<td>Tom Cohen</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>code</th>
<th>name</th>
<th>species</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Wilma</td>
<td>dog</td>
</tr>
<tr>
<td>101</td>
<td>Kitty</td>
<td>cat</td>
</tr>
<tr>
<td>102</td>
<td>Wilma</td>
<td>cat</td>
</tr>
<tr>
<td>103</td>
<td>Lucky</td>
<td>dog</td>
</tr>
<tr>
<td>104</td>
<td>Rocky</td>
<td>dog</td>
</tr>
<tr>
<td>105</td>
<td>Oreo</td>
<td>cat</td>
</tr>
<tr>
<td>106</td>
<td>Cecile</td>
<td>turtle</td>
</tr>
<tr>
<td>107</td>
<td>Chelsea</td>
<td>dog</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>102</td>
</tr>
<tr>
<td>2</td>
<td>103</td>
</tr>
<tr>
<td>3</td>
<td>104</td>
</tr>
<tr>
<td>3</td>
<td>105</td>
</tr>
<tr>
<td>4</td>
<td>106</td>
</tr>
<tr>
<td>4</td>
<td>107</td>
</tr>
</tbody>
</table>
SQL Example: Intended Interpretation

<table>
<thead>
<tr>
<th>AnimalOwner</th>
<th>CatsAndDogsOwner</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>aname</td>
</tr>
<tr>
<td>1</td>
<td>Wilma</td>
</tr>
<tr>
<td>1</td>
<td>Kitty</td>
</tr>
<tr>
<td>2</td>
<td>Wilma</td>
</tr>
<tr>
<td>2</td>
<td>Lucky</td>
</tr>
<tr>
<td>3</td>
<td>Rocky</td>
</tr>
<tr>
<td>3</td>
<td>Oreo</td>
</tr>
<tr>
<td>4</td>
<td>Cecile</td>
</tr>
</tbody>
</table>

NoCommonName

<table>
<thead>
<tr>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
create or replace view animalOwner(id,aname,specie) as select O.id, P.name, P.specie from owner O, pet P, petOwner PO where O.id = PO.id and P.code = PO.code;

create or replace view LessThan6(id) as select id from animalOwner where specie='cat' or specie='dog' group by id having count(*)<6;

create or replace view CatsAndDogsOwner(id,aname) as select AO1.id,AO1.aname from animalOwner AO1, animalOwner AO2 where AO1.id = AO2.id and AO1.specie='dog' and AO2.specie='cat';
create view commonName(ID) as
  select distinct CDO1.ID
  from CatsAndDogsOwner CDO1, CatsAndDogsOwner CDO2
  where CDO1.aname = CDO2.aname and CDO1.ID <> CDO2.ID;

create view NoCommonName(ID) as
  select ID
  from CatsAndDogsOwner C
  where not exists (select ID from commonName N where N.id = C.id);

create or replace view guest(id,name) as
  select distinct O.id, O.name
  from owner O, NoCommonName N, LessThan6 L
  where O.id = N.id and N.id = L.id;
Select * from guest;
(1, Mark Costas)
(2, Helen Kaye)
(3, Robin Scott)
SQL Example

- Guest
  - Owner
  - NoCommonName
    - CatsAndDogsOwner
    - AnimalOwner
      - Owner
      - Pet
      - PetOwner
  - LessThan6
    - AnimalOwner
      - Owner
      - Pet
      - PetOwner
The strange case of Datalog and Declarative Debugging
Datalog

- **Datalog** is a deductive database (see next talk)
- Syntax: restriction of Prolog
  - No functions allowed
  - Only stratified negation
- Bottom-up semantics: fix points are actually computed answers
Consider the following example:

- Program: \{p(X) :- q(X), q(X) :- p(X)\}
- Goal: p(X)
- Computed Answer: {} (Prolog\(\rightarrow\)non-terminating)
- Intended Interpretation \{p(a), q(a)\}
- p(a) is a missing answer
- But p(a) is not uncovered! (\(\sigma = \{X\rightarrow a\}\))
Datalog

- We cannot blame either p or q alone
- We say that \{p,q\} form an incomplete set of relations
- What about computation trees?
  - Is better to consider computation graphs
  - Apart from buggy nodes we have **buggy circuits**!
Example:

\[
\begin{align*}
&\text{star(sun).} \\
&\text{orbits(earth, sun).} \\
&\text{orbits(moon, earth).} \\
&\text{orbits}(X,Y) :\neg\text{orbits}(X,Z), \text{orbits}(Z,Y). \\
&\text{planet}(X) \ :\neg\text{orbits}(X,Y), \text{star}(Y), \text{not(intermediate}(X,Y)) \\
&\text{intermediate}(X,Y) :\neg\text{orbits}(X,Y), \text{orbits}(Z,Y).
\end{align*}
\]
Datalog

- Goal: \texttt{planet(X)}
- Answer: \{\}
- Expected answer: \{\texttt{planet(earth)}\}
- Missing answer

We use the debugger included in the datalog system DES
DES> /debug planet(X) p

Is orbits(sun,sun) = {} valid? v
Is orbits(earth,Y) = {orbits(earth,sun)} valid? v
Is intermediate(earth,sun) = {intermediate(earth,sun)}
  valid? n
Is orbits(sun,Y) = {} valid? v
Is orbits(X,sun) = {orbits(earth,sun),orbits(moon,sun)}
  valid? v
The debugger indicates that \texttt{intermediate} is wrong:

Error in relation: intermediate/2"

Witness query:

\texttt{intermediate(earth,sun) = intermediate(earth,sun)}}

The witness query can be useful for finding the error in the predicate
Conclusion

- Declarative Debugging: A general framework for debugging
- Based on the semantics of the components
- Applicable to many different paradigms
- Difficulty: complexity of the questions
- The features included in the implementations (trusted specifications, undo, maybe answers) are very important to obtain tools applicable in practice
Thank you!