# Program Repair Guided by Datalog-defined Static Analysis

Liu Yu
liuyu@comp.nus.edu.sg
National University of Singapore

Sergey Mechtaev
s.mechtaev@ucl.ac.uk
University College London

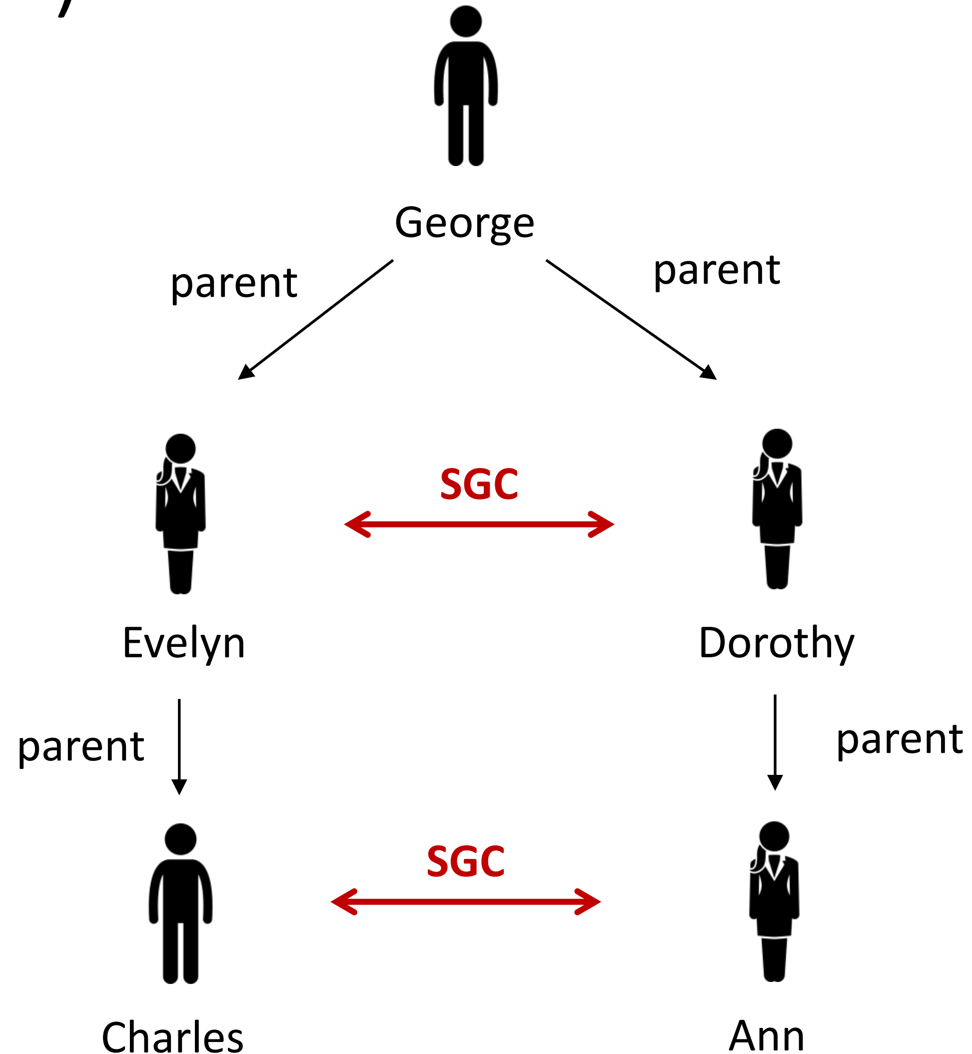
Pavle Subotic
psubotic@fantom.foundation
Fantom Foundation

Abhik Roychoudhury
abhik@comp.nus.edu.sg
National University of Singapore

# Static vs Test-Based Repair

- Test-Driven Repair
  + General-purpose, addresses wide class of bugs
  - Test-overfitting
  - No explanations
  - Inefficient (expensive test execution)
- Static Analysis Guided Repair
  + Property-driven (provides guarantees)
  + Efficient (does not require program execution)
  + Provides explanations
  - Limited class of addressed bugs, tied to specific analysers

# Introduction To Datalog: Same Generation Cousins (SGC)



# Example of Rules and Facts

Rule 1: $sgc(X, X) \leftarrow person(X)$

Rule 2: $sgc(X, Y) \leftarrow parent(X1, X),$
$parent(Y1, Y),$
$sgc(X1, Y1).$

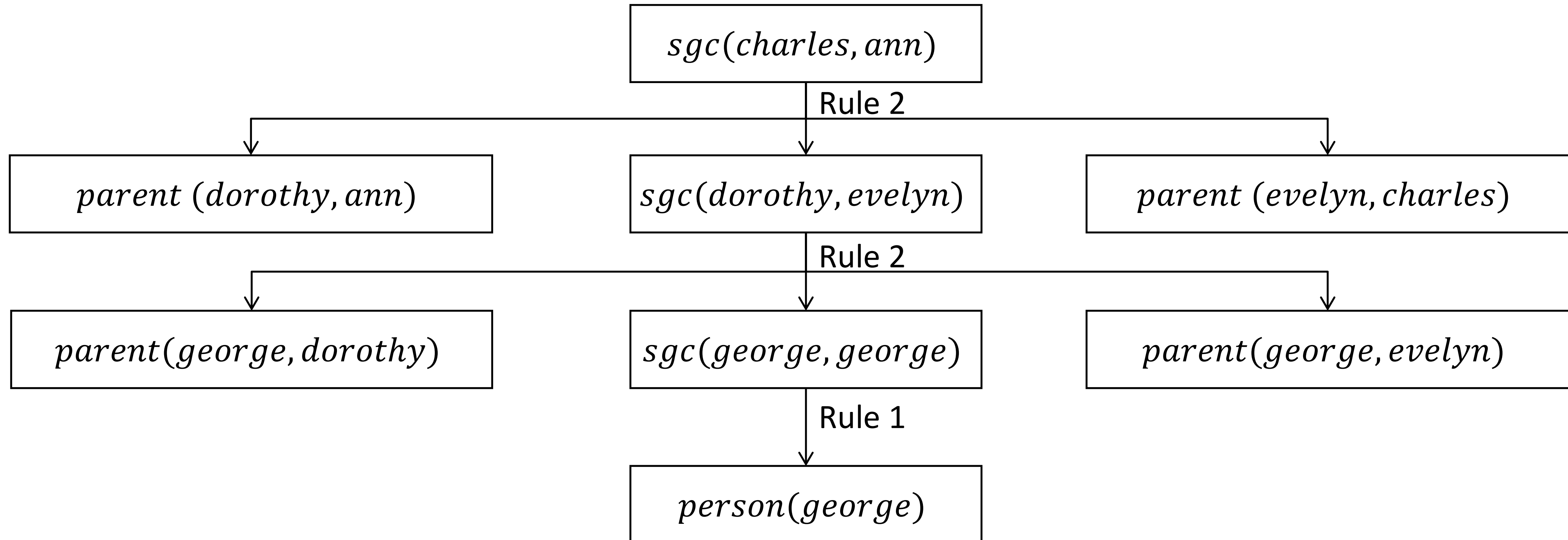
Facts:

$person(ann)$
$person(bertrand)$
$person(charles)$
$person(dorothy)$
$person(evelyn)$
$person(fred)$
$person(george)$
$person(hilary)$

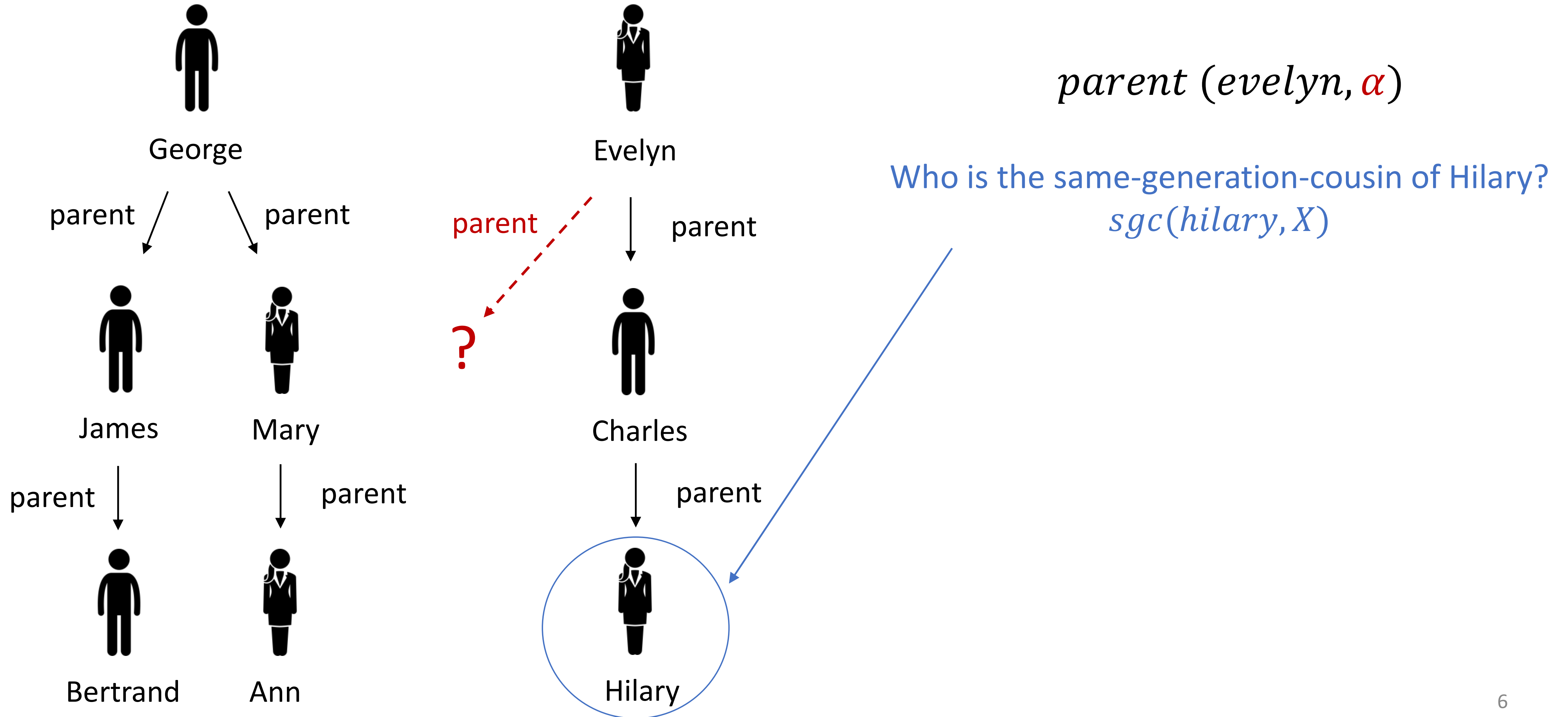
$parent(george, dorothy)$
$parent(george, evelyn)$
$parent(dorothy, bertrand)$
$parent(dorothy, ann)$
$parent(hilary, ann)$
$parent\ (evelyn, charles)$

# Proof Tree

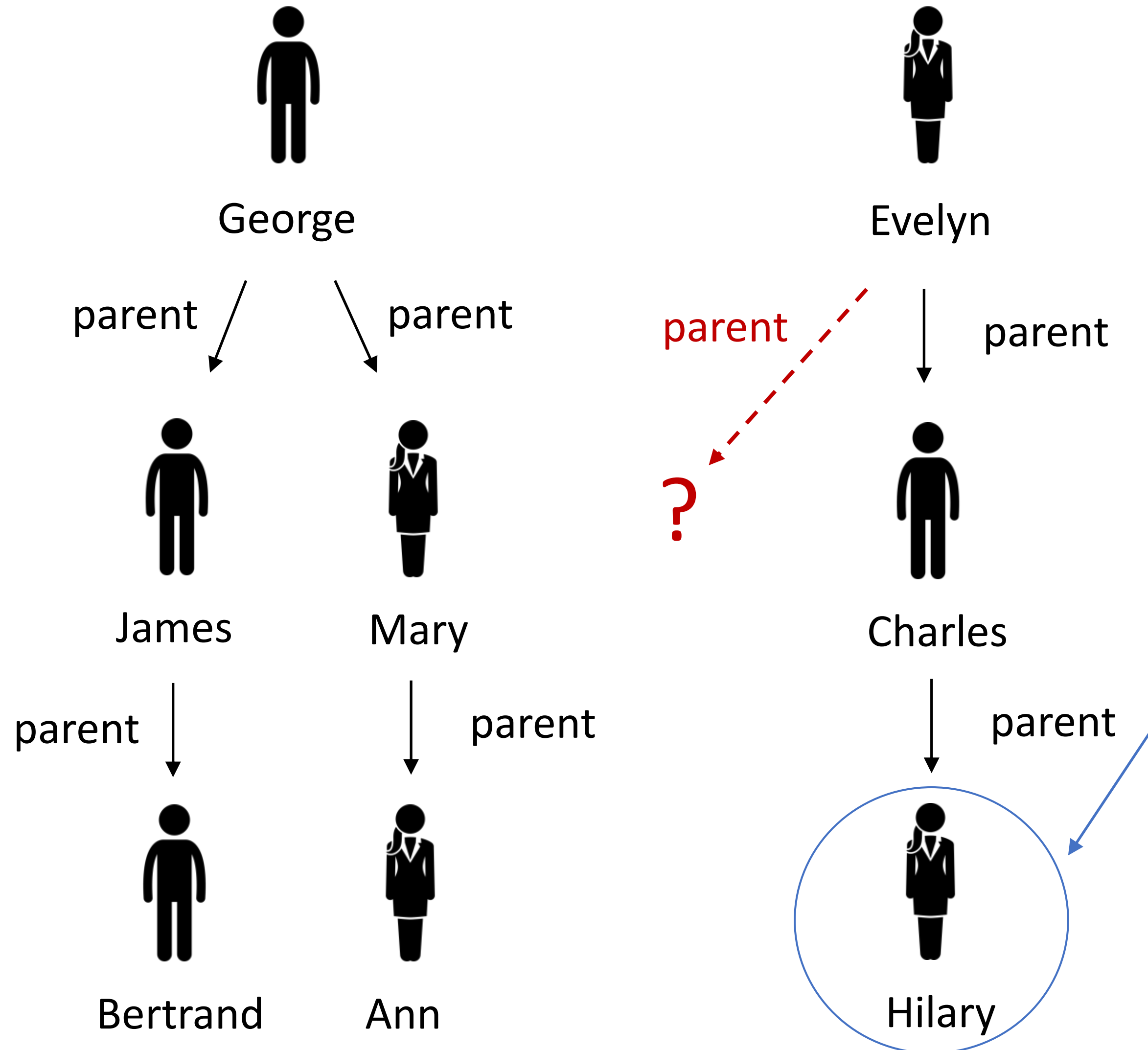
A **proof tree** for $sgc(charles, ann)$ visualises $\vdash$ relationship:



# Symbolic Execution of Datalog (SEDL)



# Symbolic Execution of Datalog (SEDL)




$parent\ (evelyn, \alpha)$

Who is the same-generation-cousin of Hilary?
$sgc(hilary, X)$

$\alpha = mary \quad \rightarrow \quad sgc(hilary, ann)$

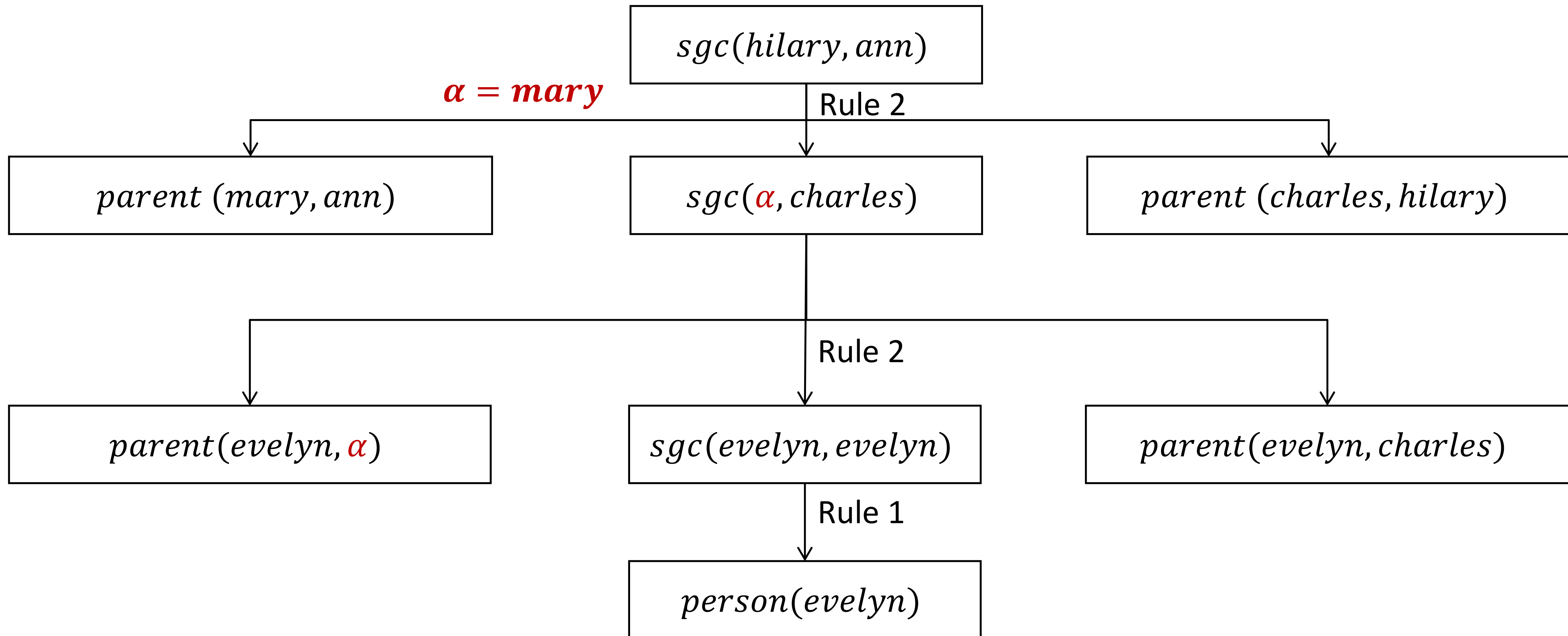
$\alpha = james \quad \rightarrow \quad sgc(hilary, bertrand)$

$\alpha = george \quad \rightarrow \quad sgc(hilary, james)$
$sgc(hilary, mary)$

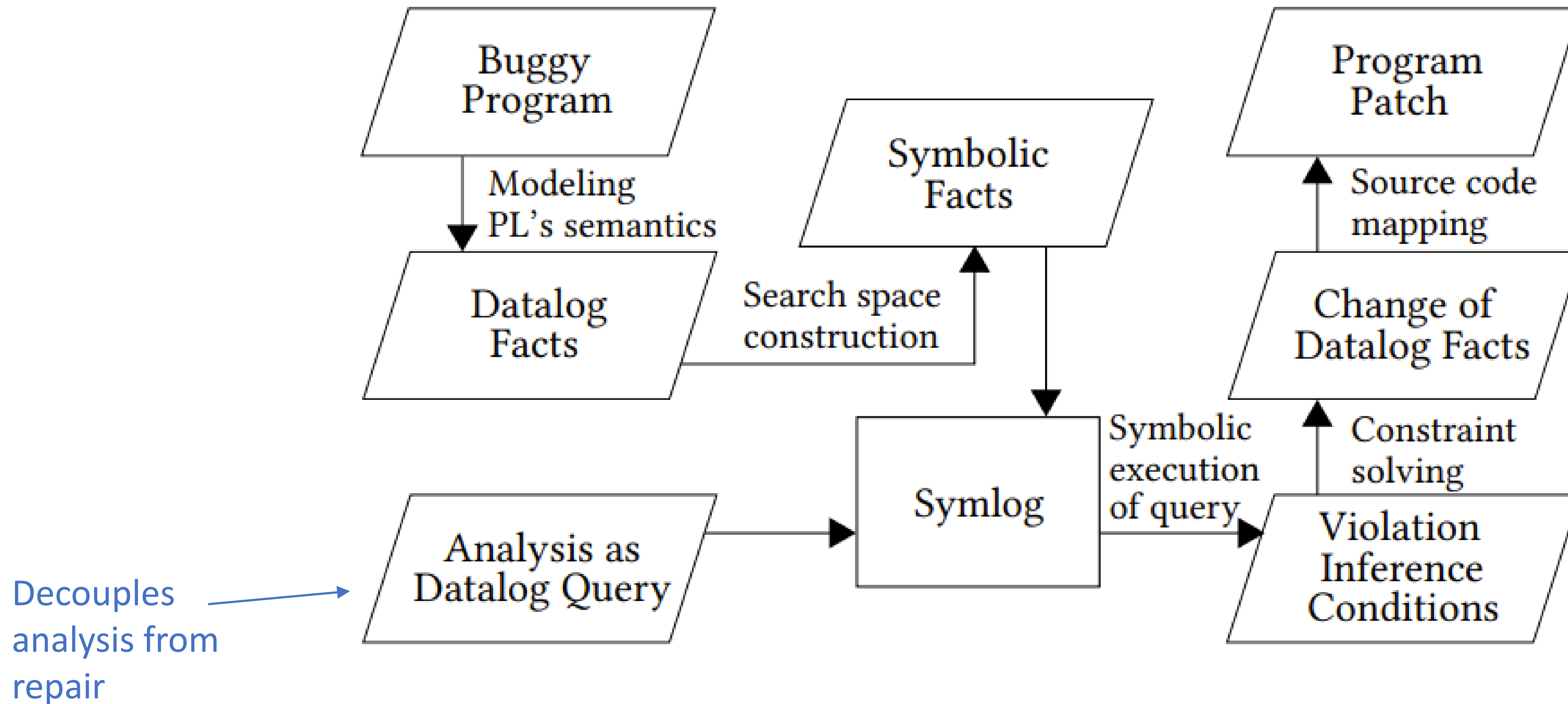
Inference condition

symbolic output

# Symbolic Proof Tree



# General-Purpose Static Analysis Guided Program Repair



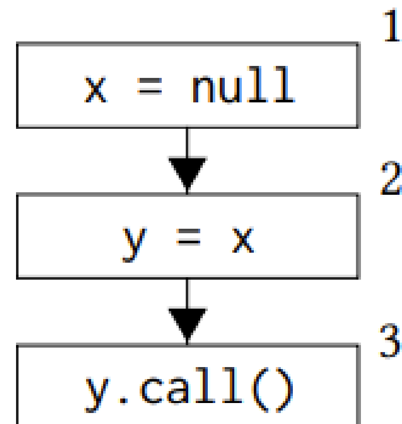

Decouples analysis from repair

# Datalog Representation of Programs

Program:

```
x = null
y = x
y.call()
```

Control Flow Graph:

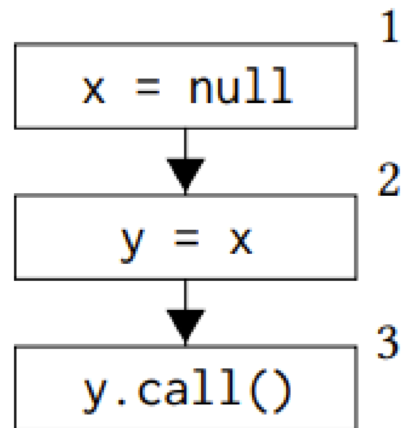



Datalog Representation:

```
flow(1, 2).
flow(2, 3).
assign_null("x", 1).
assign("y", "x", 2).
call("y", 3).
```

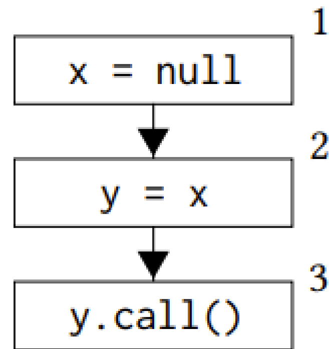
# NPE Analysis as Datalog Query

Control Flow Graph:




```
npe(V, L) :- call(V, L),
             null(V, L),
             ! guard(V, L).
null(V, L) :- flow(L1, L),
              assign_null(V, L1).
null(V, L) :- flow(L1, L),
              null(V, L1),
              ! assign(V, _, L1),
              ! assign_obj(V, L1).
null(V, L) :- flow(L1, L),
              assign(V, V1, L1),
              null(V1, L1).
```

# Proof Tree Capturing an NPE




```
flow(1, 2).
flow(2, 3).
assign_null("x", 1).
assign("y", "x", 2).
call("y", 3).
```

$$\dfrac{\dfrac{\dfrac{\texttt{flow(1,2)} \quad \texttt{assign_null("x",1)}}{\texttt{null("x",2)}} \quad \texttt{flow(2,3)} \quad \texttt{assign("y","x",2)}}{\texttt{null("y",3)}} \quad \texttt{call("y",3)} \quad \texttt{!guard("y",3)}}{\texttt{npe("y",3)}}$$

# Repairing NPE with SEDL

Concrete EDB (representation of the program):

```
flow(1, 2).
flow(2, 3).
assign_null("x", 1).
assign("y", "x", 2).
call("y", 3).
```

Symbolic EDB (representation of a set of modifications):

$\xi_1$ `flow(1, 2).`
$\xi_2$ `flow(2, 3).`
$\xi_3$ flow($\alpha_1$, $\alpha_2$).
$\xi_4$ flow($\alpha_2$, $\alpha_3$).
`assign_null("x", 1).`
`assign("y", "x", 2).`
`call("y", 3).`
$\xi_5$ guard($\alpha_4$, 3).
$\xi_6$ assign_obj($\alpha_5$, $\alpha_6$).

# A Symbolic Proof Tree of NPE

**Element of the search space (insert a statement between lines 1 and 2):**

```
x = null
y = new obj()
y = x
y.call()
```

**NPE inference condition:**

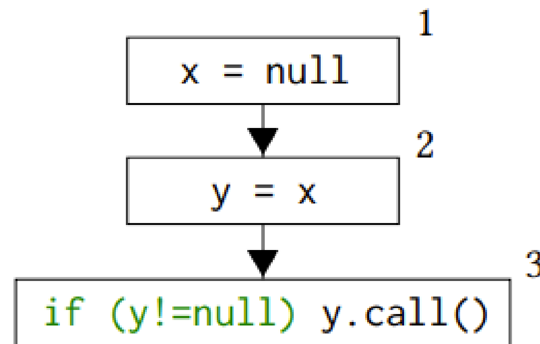
$$(\xi_2 \land \xi_3 \land \alpha_1 = 1 \land \xi_4 \land \alpha_3 = 2$$
$$\land \neg(\xi_6 \land \alpha_5 = \text{“}x\text{”} \land \alpha_6 = \alpha_2) \land \neg(\xi_5 \land \alpha_4 = \text{“}y\text{”}))$$

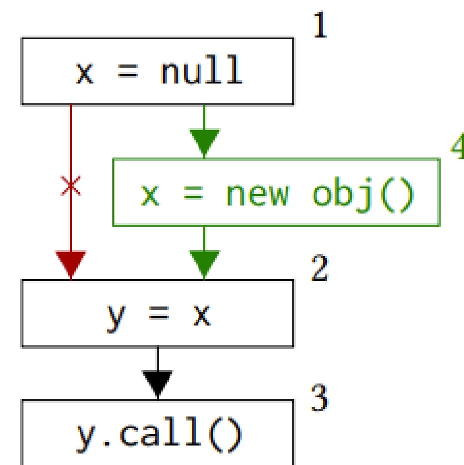
$$\cfrac{\cfrac{\xi_3\,\texttt{flow}(\alpha_1,\alpha_2) \quad \texttt{assign_null("x",1)}}{\texttt{null("x",}\alpha_2\texttt{)}}\;\xi_1 \land \alpha_1 = 1 \quad \xi_4\,\texttt{flow}(\alpha_2,\alpha_3) \quad \texttt{!assign("x",_,}\alpha_2\texttt{)} \quad \xi_6\,\texttt{assign_obj}(\alpha_5,\alpha_6)}{\texttt{null("x",}\alpha_2\texttt{)}}\;\xi_4 \land \alpha_3 = 3 \land \neg(\xi_6 \land \alpha_5 = \text{“}x\text{”} \land \alpha_6 = \alpha_2)$$

$$\cfrac{\cfrac{\texttt{null("x",}\alpha_2\texttt{)} \quad \xi_2\,\texttt{flow(2, 3)} \quad \texttt{assign("y","x",2)}}{\texttt{null("y",3)}}\;\xi_2 \quad \texttt{call("y",3)} \quad \xi_5\,\texttt{guard}(\alpha_4\texttt{, 3)}}{\texttt{npe("y",3)}}\;\neg(\xi_5 \land \alpha_4 = \text{“}y\text{”})$$

# Repair Synthesis

Solve **repair condition** $\neg\phi \wedge \psi$, where $\phi$ is the inference condition of the bug, and $\psi$ are structural constraints.




```
+ guard("x", 3).
```

```
- flow(1, 2).
+ flow(1, 4).
+ flow(4, 2).
+ assign_obj("x", 4).
```

# Experiments on Three Classes of Bugs

## 10 Java NPEs (correct + plausible incorrect):

| NPEX | AlphaRepair | InCoder | Symlog nonopt | Symlog | Nonopt time | Opt time | Nonopt memory | Opt memory |
|---|---|---|---|---|---|---|---|---|
| 7+1 | 2+2 | 0+0 | **6+1** | **8+2** | >3m 50s | 3m 50s | >9.7Gb | 6.1Gb |

## 11 preprocessing leakage bugs (correct + plausible incorrect):

| AlphaRepair | InCoder | Symlog nonopt | Symlog | Nonopt time | Opt time | Nonopt memory | Opt memory |
|---|---|---|---|---|---|---|---|
| 0+0 | 0+0 | **0+0** | **6+4** | - | 24m 7s | all OOM | 200Mb |

## 63 smart contracts bugs (correct + plausible incorrect):

| Elysium | Symlog | Opt time |
|---|---|---|
| 15+0 | **62+1** | <4m |

# Example Repair (Preprocessing Leakage)

```
from sklearn.preprocessing import MinMaxScaler

dataset = load_data()
scaler = MinMaxScaler(feature_range=(0, 1))
- scale_data=scaler.fit_transform(dataset)

- train_data, test_data = split_data(scale_data)
+ train_data, test_data = split_data(dataset)

x_train, y_train = split_train_data(train_data)
x_test, y_test = split_test_data(test_data)
model = LSTM_model()

+ x_train_new = scaler.fit_transform(x_train)
+ x_test_new = scaler.transform(x_test)

- model.fit(x_train, y_train)
+ model.fit(x_train_new, y_train)
- predictions = model.predict(x_test)
+ predictions = model.predict(x_test_new)
```

# Conclusion

- **Symbolic execution of Datalog**, a new reasoning approach that computes dependency between the input and the output of a Datalog query.
- Its application to **static analysis guided program repair**
- Paper:

  **Program Repair Guided by Datalog-Defined Static Analysis**
  Liu Yu, Sergey Mechtaev*, Pavle Subotic, Abhik Roychoudhury
  FSE 2023
- Tool Symlog:

  github.com/symlog/symlog