Probabilistic, Modular and Scalable Inference of Typestate Specifications

Nels E. Beckman, CMU/Google Pittsburgh
Aditya V. Nori, Microsoft Research India

Summary

- **Anek** infers specifications needed to check typestate with
- **Plural**: sound static typestate checker,
  - requires pre- & post-conditions
  - Specifications time-consuming

- Anek:
  - Assigns and solves *probabilistic* constraints over program
    - Approximate, but works well in practice
    - Probabilities act as refinable summaries on methods
    - Scales, because it can be applied iteratively
    - Ease of design, heuristics

Plural*: An Overview

- A **sound** static typestate checker for Java programs

```
class File {
  // Creates an
  // open file
  File();

  // File must be open
  void read();

  void close();
}
```

**Bierhoff, Aldrich, OOPSLA '07*

Plural*: An Overview

- A **sound** static typestate checker for Java programs
- Works like a type system
  - Assigns types to variables that vary from line to line
  - Requires pre- and post- types for method parameters
  - Type includes object state & aliasing

```
class File {
  // Creates an
  // open file
  File();

  // File must be open
  void read();

  void close();
}
```

**Bierhoff, Aldrich, OOPSLA '07*

Plural: An Illustration

```
// pre -  f : Open,   unique
// post - f : Closed, unique
String readAndClose(File f)
{
  // f : Open, unique
  Log.log("Reading");
  // f : Open, unique
  String s = f.read();
  // f : Open, unique
  f.close();
  // f : Closed, unique
  return s;
}
```

```
class File {

  // pre  - this : Open, unique
  // post - this : Open, unique
  void read();


  // pre  - this : Open,   unique
  // post - this : Closed, unique
  void close();



}
```

Plural: Other Details

- Other permissions:
  - Unique
  - Immutable
  - Shared
  - 1 Writer, Many Readers
- Fields can have permissions & state
- Concurrency!
- All sound!

The Challenge

- Adding Plural specifications is labor intensive
  - E.g., days for an API in case studies
- Requires tool & methodology expertise
- Requires detailed program expertise (i.e., aliasing)

Anek (अनेक): Specification Inference

- Input: Program with annotated API
- Output: Program with all annotations needed to check use of API

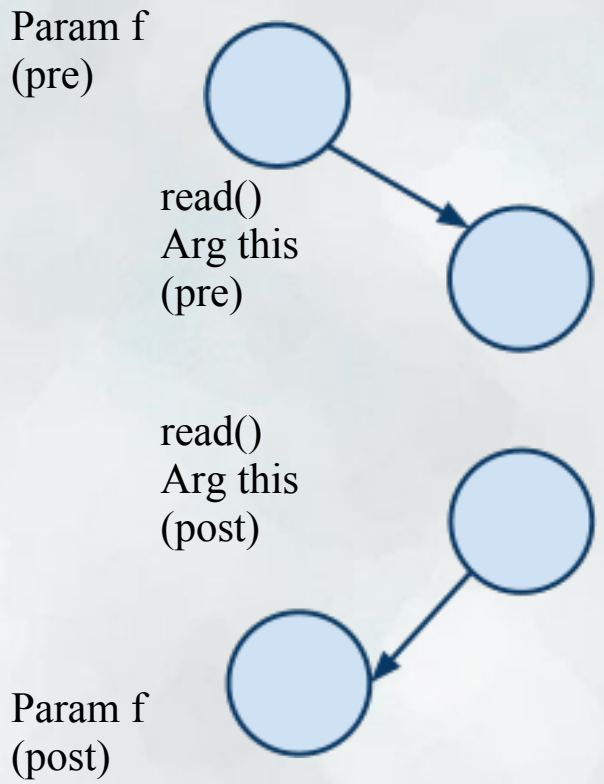
- Process:
  - Turn program into permission-flow graph
  - Solve probabilistic constraints
  - Inferred specification is the 'likeliest' one
- Benefits:
  - Robust to bugs
  - Easy to summarize methods, modularity
  - heuristics

Permission Graph

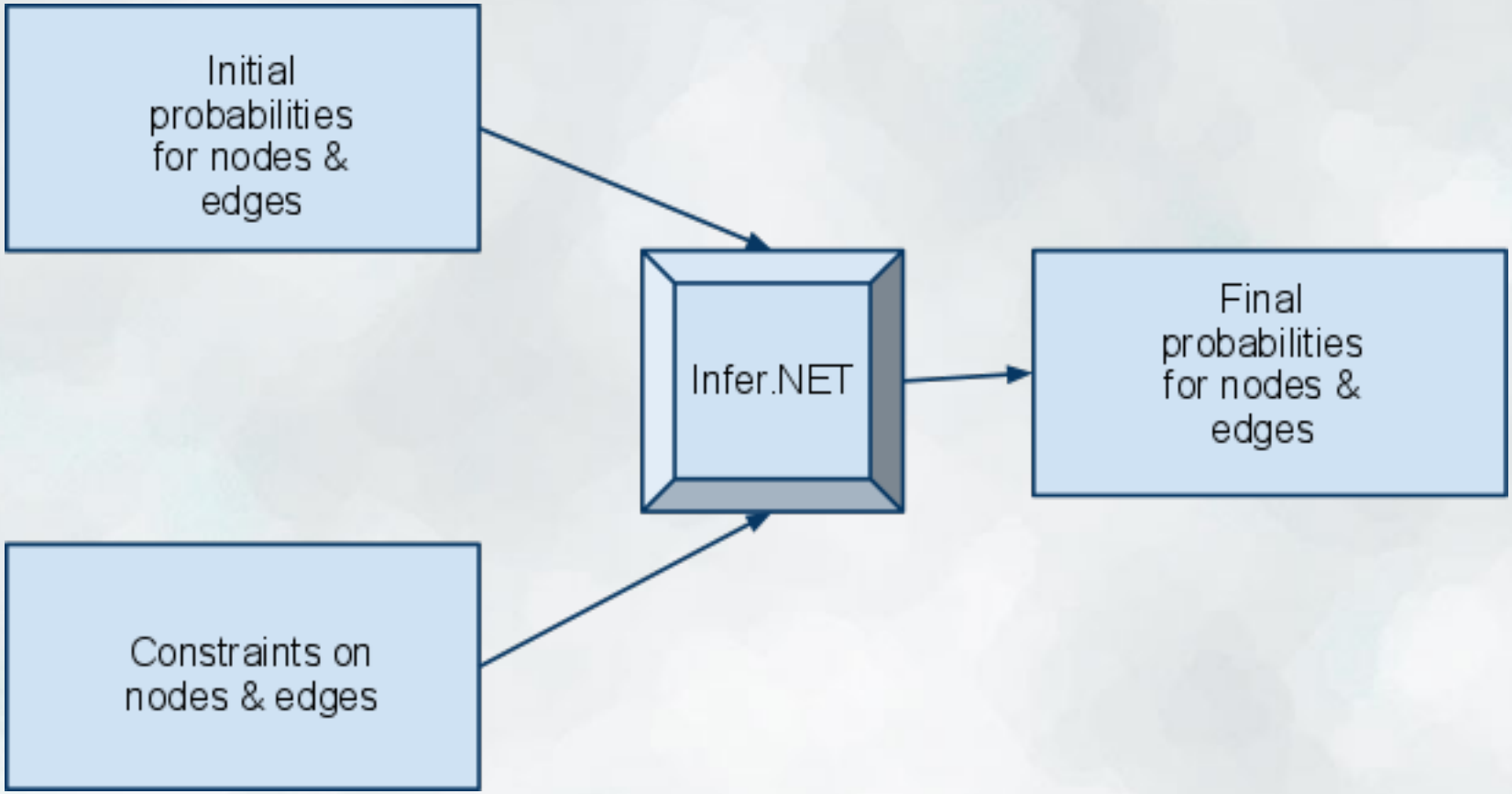
- For all program references, create permission flow graph
- Graph is almost identical to data-flow graph.
  - (Parameter permission implicitly returned)

Permission Graph

```
// pre  - NOT GIVEN
// post - NOT GIVEN
String readAndClose(File f) {
  …
  String s = f.read();
  …
}
```



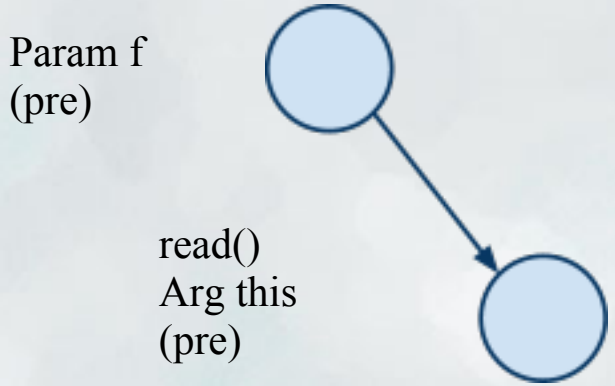
Probabilistic Constraints



Initial Probabilities

```
// pre  - NOT GIVEN
// post - NOT GIVEN
String readAndClose(File f) {
  …
  String s = f.read();
  ...
}

class File {
  ...
  // pre - this: OPEN,
  //       unique
  void read();
}
```



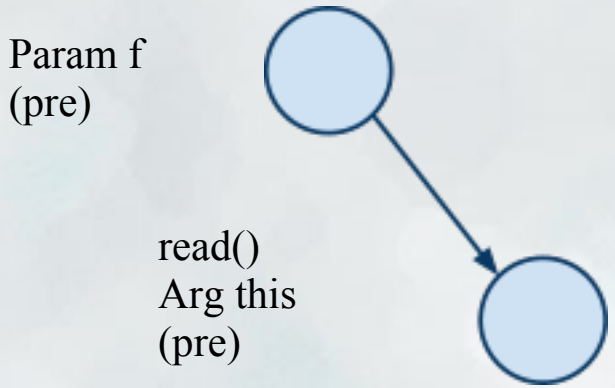
	Initial Value param f pre	Initial Value read() arg pre
OPEN	**?**	**.9999**
CLOSED	?	.0001
unique	**?**	**.9999**
immutable	?	.0001
share	?	.0001
full	?	.0001
pure	?	.0001

Probabilistic Constraints

- Given known specification, how to solve for unknown?
  1. Add constraints to graph
  2. Off-the-shelf solver

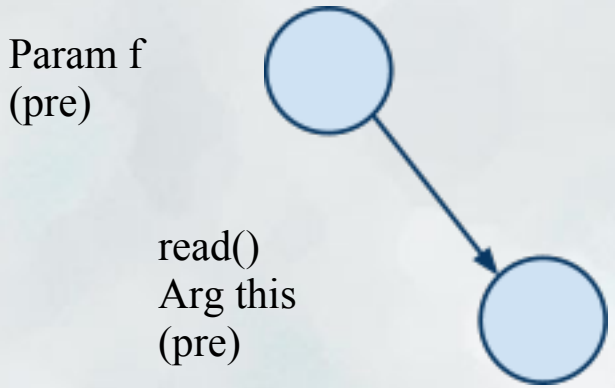
Probabilistic Constraints

- Given known specification, how to solve for unknown?
  1. Add constraints to graph
  2. Off-the-shelf solver



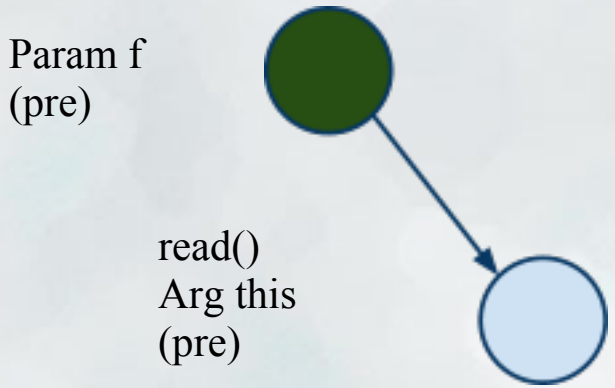
Probabilistic Constraints

- Given known specification, how to solve for unknown?
  1. Add constraints to graph
  2. Off-the-shelf solver
- Constraints:
  - Permission rules
  - Heuristics



Probabilistic Constraints

- Given known specification, how to solve for unknown?
  1. Add constraints to graph
  2. Off-the-shelf solver
- Constraints:
  - Permission rules
  - Heuristics



	Final Value
OPEN	**.997**
CLOSED	.01
unique	**.997**
immutable	.01
share	.01
full	.02
pure	.03

Modularity

- Solver is iterative
  - Probabilities become more accurate over time

Modularity

- Solver is iterative
  - Probabilities become more accurate over time

Param f (pre)

	Intermediate Value
OPEN	**.558**
CLOSED	.456
unique	**.677**
immutable	.33
share	.25
full	.499
pure	.4

Modularity

- Solver is iterative
  - Probabilities become more accurate over time
- Use them as method summaries!
  1. Solve for a method,
  2. Store just values of signature,
  3. Solve other methods,
  4. Iterate
- Avoid storing entire graph
- Sliding scale between precision and time

Param f (pre)

	Intermediate Value
OPEN	**.558**
CLOSED	.456
unique	**.677**
immutable	.33
share	.25
full	.499
pure	.4

Evaluation: Training

- Evaluation Procedure:
  - "Train" algorithm on small benchmarks
  - Evaluate results on large case study
  - Compare time & precision to manual

- Training
  1. Create a number of small benchmarks
  2. Run Anek
  3. Adjust probabilities
  4. Iterate until results are as expected

Evaluation: Iterator API

- Given annotated Iterator API, infer specifications in PMD
  - 40kloc
  - 170 calls to Iterator.next()
  - Initially 45 warnings
  - Case study in Bierhoff thesis

- Results
  - Bierhoff: **26 annotations** in **75 min**, resulted in **3 warnings**
  - Anek: **31 annotations** in **3 min 47 sec,** resulted in **4 warnings**

Evaluation: Comparisons

- In paper we attempt to compare with non-probabilistic algorithm
- Anek is faster because of approximative algorithms

Summary

- Plural: sound static typestate checker,
  - requires pre- & post-conditions containing aliasing permissions
  - Specifications time-consuming
- Anek infers these specifications
  - Assigns and solves *probabilistic* constraints over nodes
    - Works well in practice
    - Probabilities act as refinable summaries on methods
    - Scales, because it can be applied iteratively
    - Ease of design, heuristics
  - We believe in probabilistic inference!
  - Source available soon!