Java Composition with Labels and AI Planning

Johan Nyström-Persson (johan@nii.ac.jp)

Dept. of Computer Science
University of Tokyo / NII (Honiden laboratory), Japan

Shinichi Honiden (honiden@nii.ac.jp)
Dept. of Computer Science, University of Tokyo / NII, Japan

Motivation

- **Software construction from libraries and components is now the dominant paradigm**
- Java is an excellent language for component-based software development
- **However, component integration and re-integration remains difficult**
  - Integrating components in the first place
  - Re-integrating after evolution

Difficulty of integration

- Dependencies between classes from different components in Java are encoded as **method calls** or as **field reads/writes**
- **In order to use an API, we need knowledge that is not formally specified**
    - The meaning and usage of each argument of a method, of return values etc.
    - Temporal constraints (correct order of method invocations etc.)
- These constraints evolve over time, causing breaking changes; a well known problem[1,2]

1. Shaw, “Procedure Calls are the Assembly Language of Software Interconnections.” Proc Workshop Studies of Software Design, 1993
2. Kell, “The Mythical Matched Modules.” OOPSLA, 2009

A typical syntactic breaking change

- The time and date API changed substantially between Java 1.4 and 1.5 (see below)
- **The *capabilities* of the time and date component have not been reduced, but the *structure of the interface* has changed.**
- Idea: we should depend on and provide *capabilities*, not interface details

Java 1.4

```
Date now = new Date();
int hour = now.getHour();
```

Java 1.5

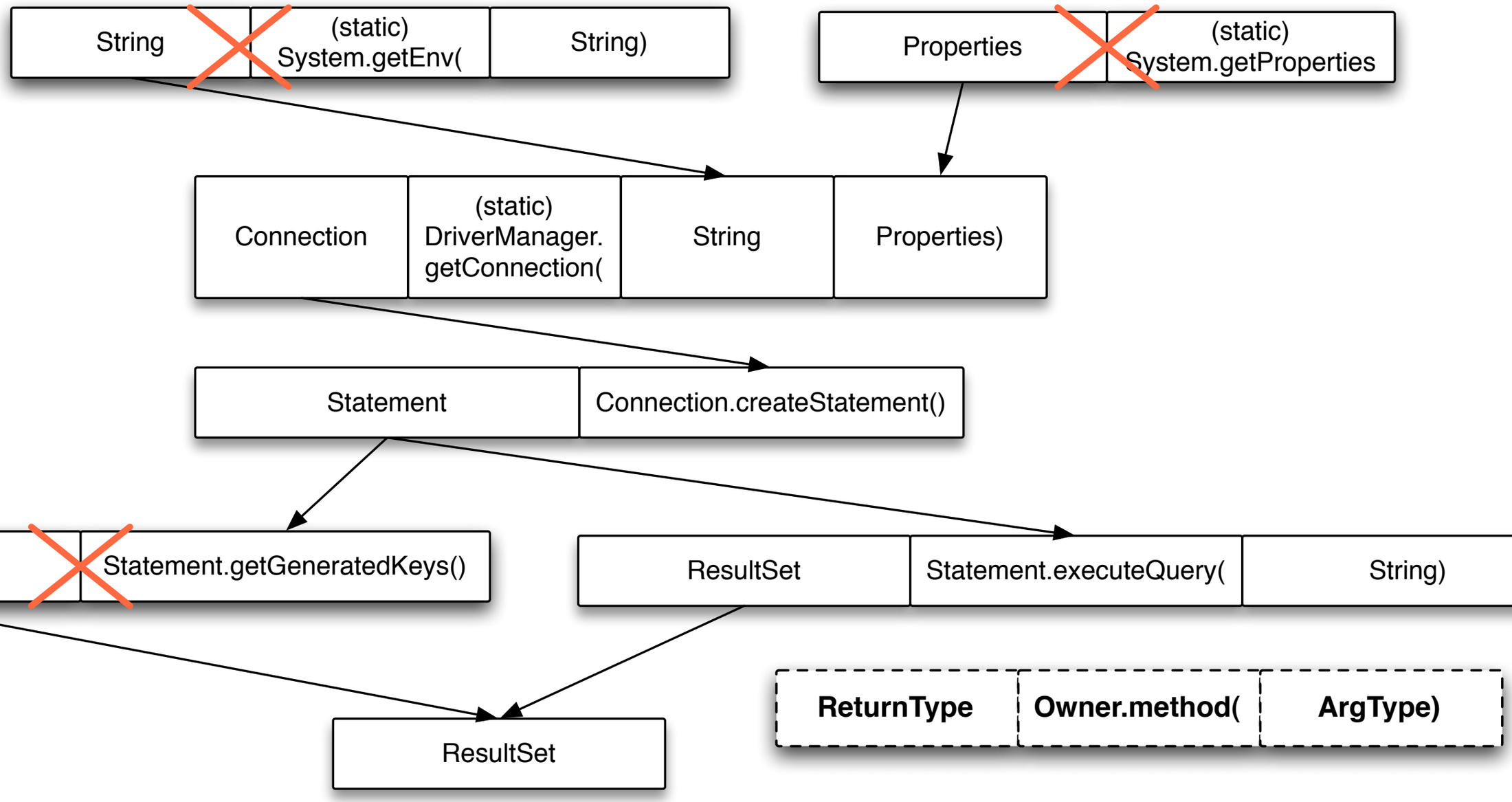
```
Calendar now = Calendar.getCalendar();
int hour = now.get(Calendar.HOUR_OF_DAY);
```

Producing values: JDBC example

```
void m(Properties p) { /* Inputs: p, url, query, column */
    String url = "jdbc:mysql://localhost:3306/...";
    String query = "select * from data where item.value > 5;";
    int column = 1;

    Connection c = DriverManager.getConnection(url, p);
    Statement s = c.createStatement();
    ResultSet rs = s.executeQuery(query);
    while (rs.next()) {
        int target = rs.getInt(column); //Target value
        //...
    }
}
```

- **Goal: produce a certain kind of value (the integer field from the database) given certain inputs**
- This code depends on method names, relations between methods, type signatures etc.
- **Idea: try to find the necessary code using a search algorithm**

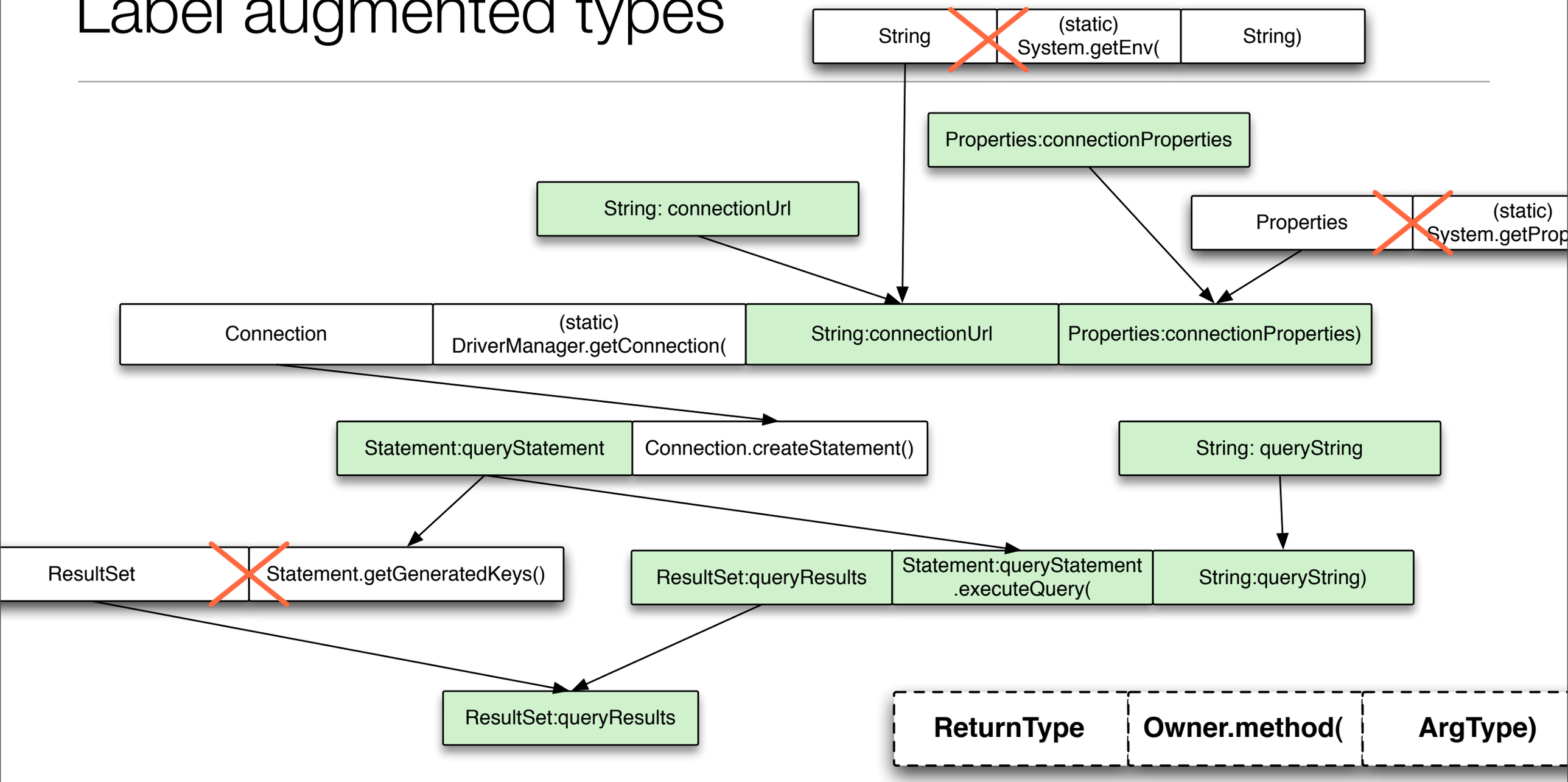
Type based search, leads to errors



- Much of the code fragment can be constructed by a **reverse type based search**
- In case of very general types (String, int...) many possibilities are incorrect

Inspired by: Mandelin, Xu, Kimelman and Bodik. “Jungloid Mining: Helping to Navigate the API Jungle.” PLDI 2005

Label augmented types



- Can rule out incorrect choices by specifying variables with labels
- Label based selection has been attempted for ML and Lambda calculus[1], not Java

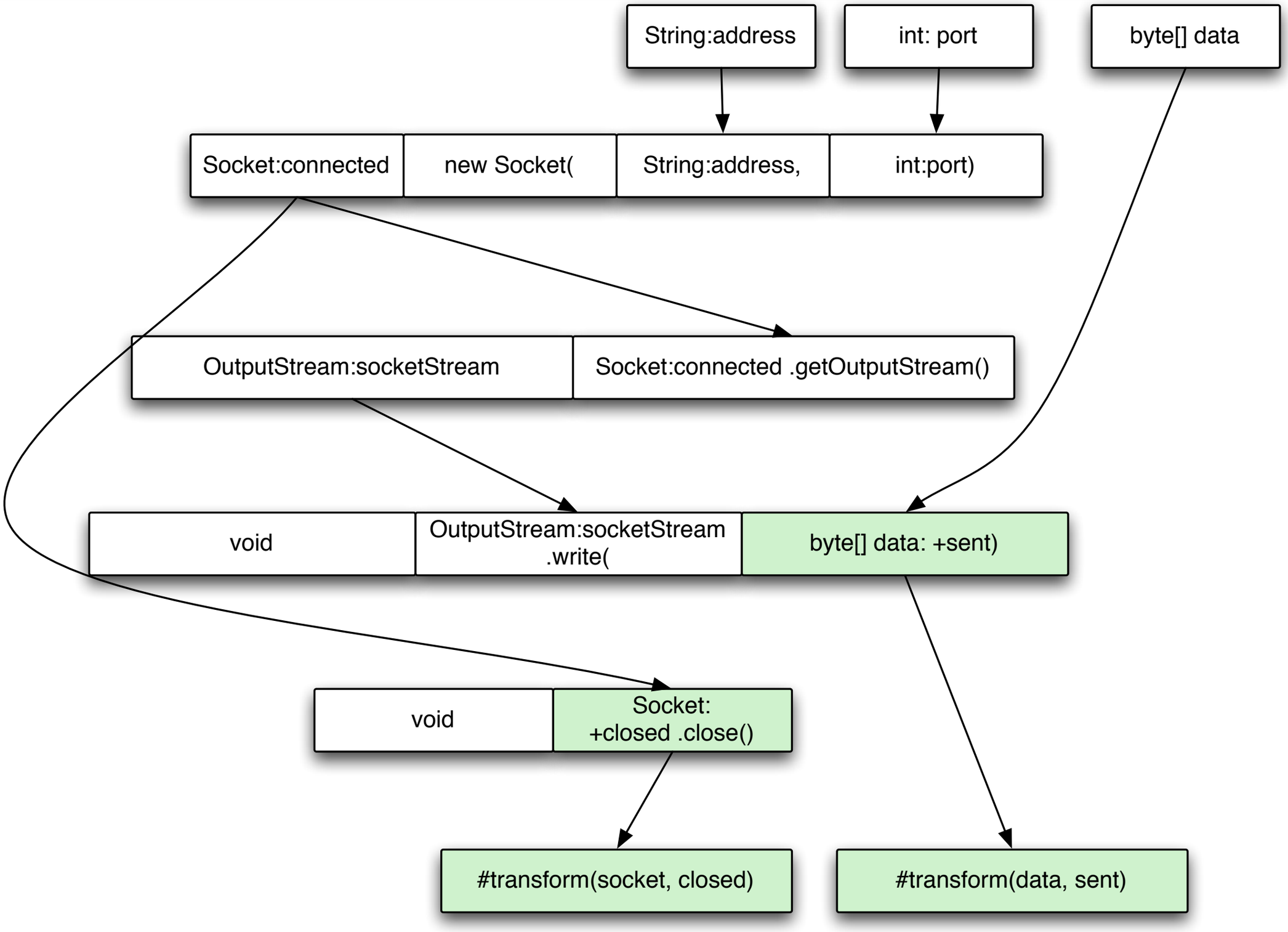
1. Garrigue and Furuse. “A Label-Selective Lambda Calculus with Optional Arguments and its Compilation Method.” 1995

Transforming values: Socket example

```
void m(byte[] data) {
    Socket s = new Socket("localhost", 3000);
    OutputStream o = s.getOutputStream();
    o.write(data);
    s.close();
}
```

- The purpose of this code is to send the data, i.e. **to cause a side effect**
- Can we achieve this through search?
- Idea: **describe label changes in method signatures**

Requesting transformations (additional labels)



Two kinds of capabilities

- Given some existing variables, with known types and labels:
  - **Produce** capability
    - Produce a new variable of a given type, with given labels
  - **Transform** capability
    - Transform an existing variable to give it additional labels

AI planning at the level of variables

- Searching for code to integrate components resembles an *AI planning problem*
    - Identifying a sequence of actions that may interfere with each other
- **Approach: describe variables in interfaces in such a way that we can apply planning algorithms to generate integration code**
    - Generate **safe and correct** code that can be mixed with handwritten code
    - Describe and request **useful**, composable API usage patterns

Core features of Poplar

- A Java extension that compiles to pure Java code (no runtime support)
- Associate a dynamic *set of labels* with each variable
    - Specify how methods depend on and modify labels
- A planning algorithm searches for *solutions* (code) by using the labels. Solutions satisfy *queries*.
    - **A query and a solution form an *integration link*.**

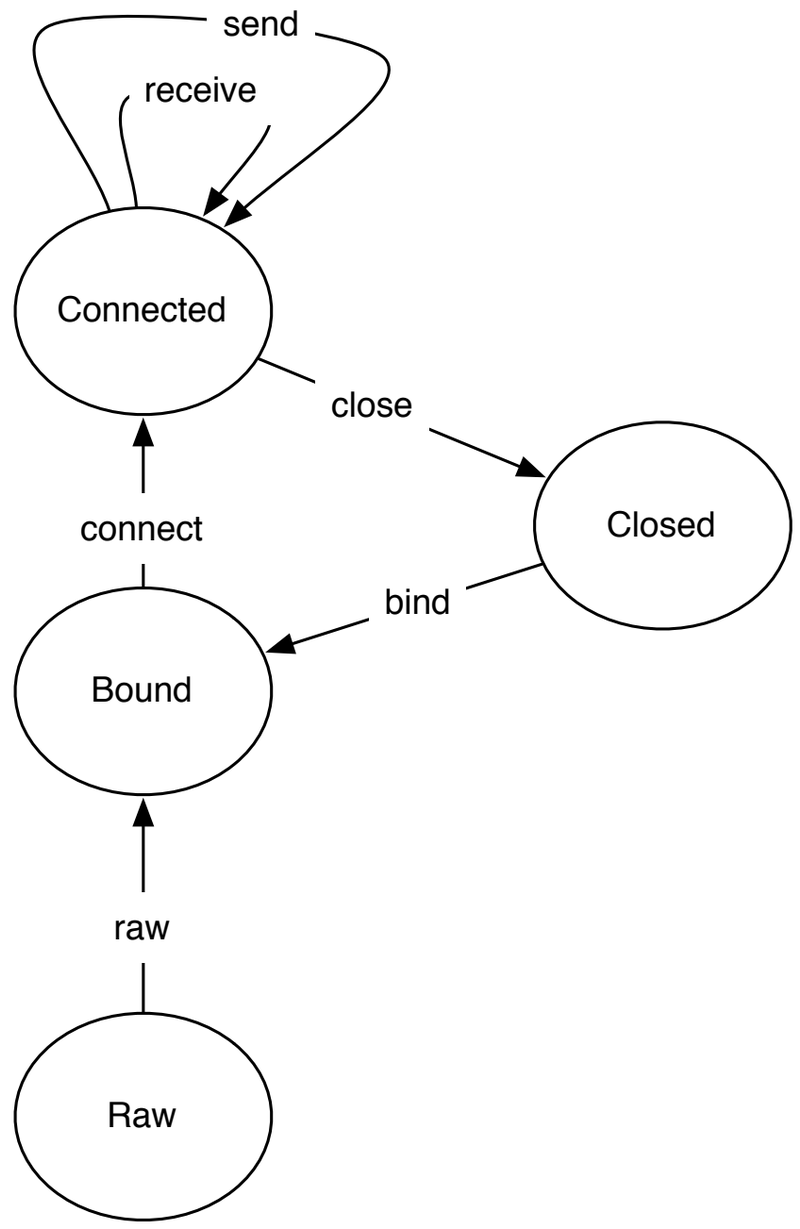
Example: a Poplar declaration

```
class Connector {
    tags(Address) connectionAddress;

    resource connection {
        properties @connected, @configured;

        void connect(Address a)
            this: +@connected, @configured;
            a: connectionAddress. {
                //...
            }

        void disconnect() {
            //...
        }
    }
}
```

- *Properties* (name begins with @) are labels that can be erased
- A property represents a configuration of the object's internal state
- @x: *invariant*
- +@x: *postcondition*

Related: Typestate Checking

- Typestate[1,2,3 etc.] typically describes objects as state machines in order to check for invalid API usage
  - For example, Sockets can be in a “connected” state or a “closed” state
  - Methods may express pre- and postconditions on object state



1. Yellin and Strom. “Typestate: A New Programming Language Concept for Software Reliability.” IEEE Trans Softw Eng. 1986.

2. Deline and Fähndrich. “Typestates for Objects.” ECOOP, 2004.

3. Bierhoff and Aldrich. “Modular Typestate Checking of Aliased Objects.” OOPSLA, 2007

Poplar compared to typestate

- We must describe not just *legal* API sequences but also *useful* API sequences.

- A Poplar state is a set of labels, so clients can depend on any subset of a state.

  - This is a source of slack for future evolution!

- We allow any class to provide labels for any other classes

  - Typestates are only provided by a class for itself

Tracing properties

```
class Connector {
    tags(Address) connectionAddress;

    resource connection {
        properties @connected, @configured;

        void connect(Address a)
            this: +@connected, @configured;
            a: connectionAddress. {
                //...
            }

        void disconnect() {
            //...
        }
    }
```

- Properties are erased through *resource mutations*, an idea adapted from the Boyland-Greenhouse system
- **Mutating a resource erases all of its properties**, unless we specify otherwise
- In addition to generating code, we can verify handwritten code w.r.t. a *mutation summary* such as mutates c.connection

```
class Client {
    Address a(connectionAddress);
    void m(Connector c) mutates
    c.connection: c: -@configured. {
                c.connect(a);
                c.disconnect();
    }
}
```

Labels of c after execution

@configured
@connected, @configured
(none)

Related: Boyland-Greenhouse Effect System[1]

- Every object has a hierarchy of *abstract regions*, which are groups of fields (polymorphic)

- B-G system tracks reads and writes of fields; the purpose is to know whether two statements may interfere with each other (boolean)

- Our system tracks **creation and destruction of labels**. If there is interference, **we can know which labels may be destroyed as a result.**

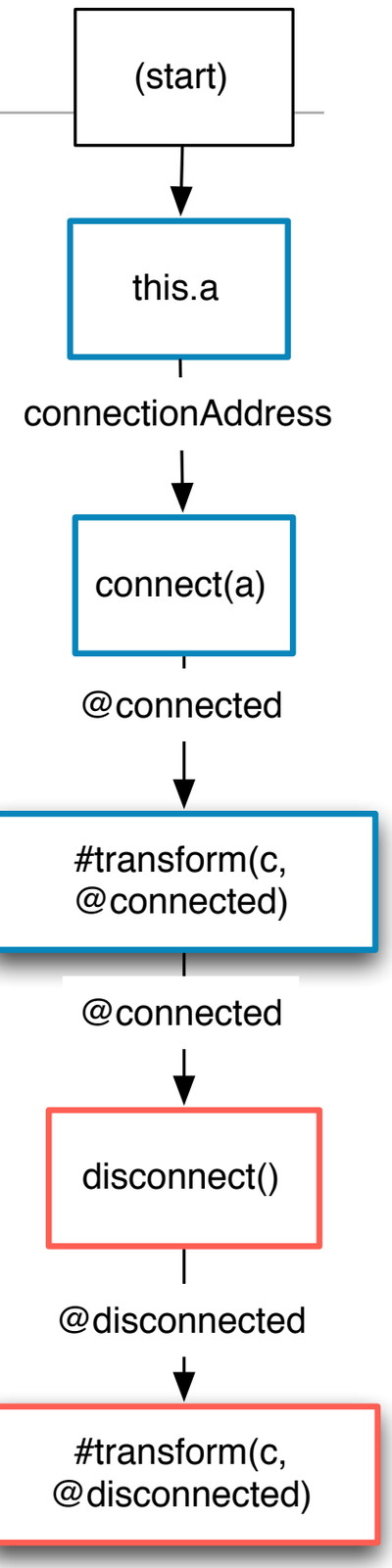
1. Boyland and Greenhouse. “An Object-Oriented Effects System.” ECOOP, 1999

Queries and solutions

```
class Connector {
    tags(Address) connectionAddress;

    resource connection {
        properties @connected, @disconnected;

        void connect(Address a)
            this: +@connected;
            a: connectionAddress. {
                //...
            }

        void disconnect()
            this: -@connected, +@disconnected. {
                //...
        }
```

- When we solve a query, we generate a set of *goal conditions* from the query and *assumptions* from the context

- In general, we construct a partially ordered sequence of actions using a planning algorithm




```
class Client {                                          Generated code
    Address a(connectionAddress);
    void m(Connector c) mutates c.connection:
        c: {
            #transform(c, @connected);                  c.connect(a);
            #transform(c, @disconnected);               c.disconnect();
    }
}
```

Evolving the Connector

Old

```
class Connector {
    tags(Address) connectionAddress;

    resource connection {
        properties @connected, @disconnected;

        void connect(Address a)
            this: +@connected;
            a: connectionAddress. {
                //...
            }

        void disconnect()
            this: -@connected, +@disconnected. {
                //...
    }
```

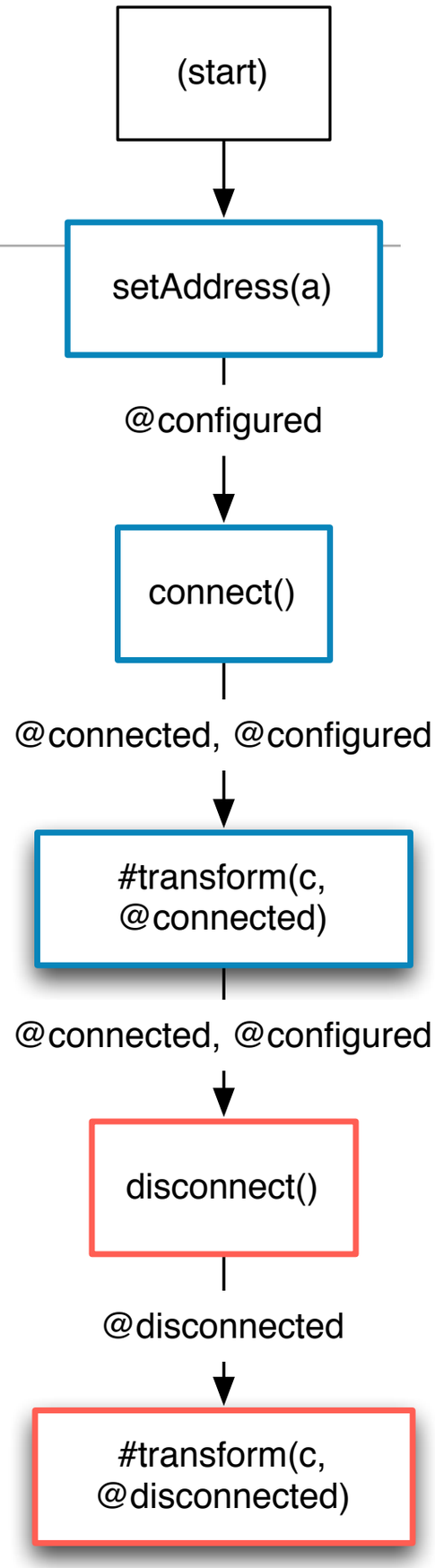
New

```
class Connector {
    tags(Address) connectionAddress;

    resource connection {
        properties @connected, @disconnected,
        @configured;

        void setAddress(Address a)
            a: connectionAddress;
             this: +@configured. { // ... }

        void connect()
            this: +@connected, @configured. {
                //...
            }

        void disconnect()
            this: -@connected, +@disconnected. {
                //...
            }
    }
}
```

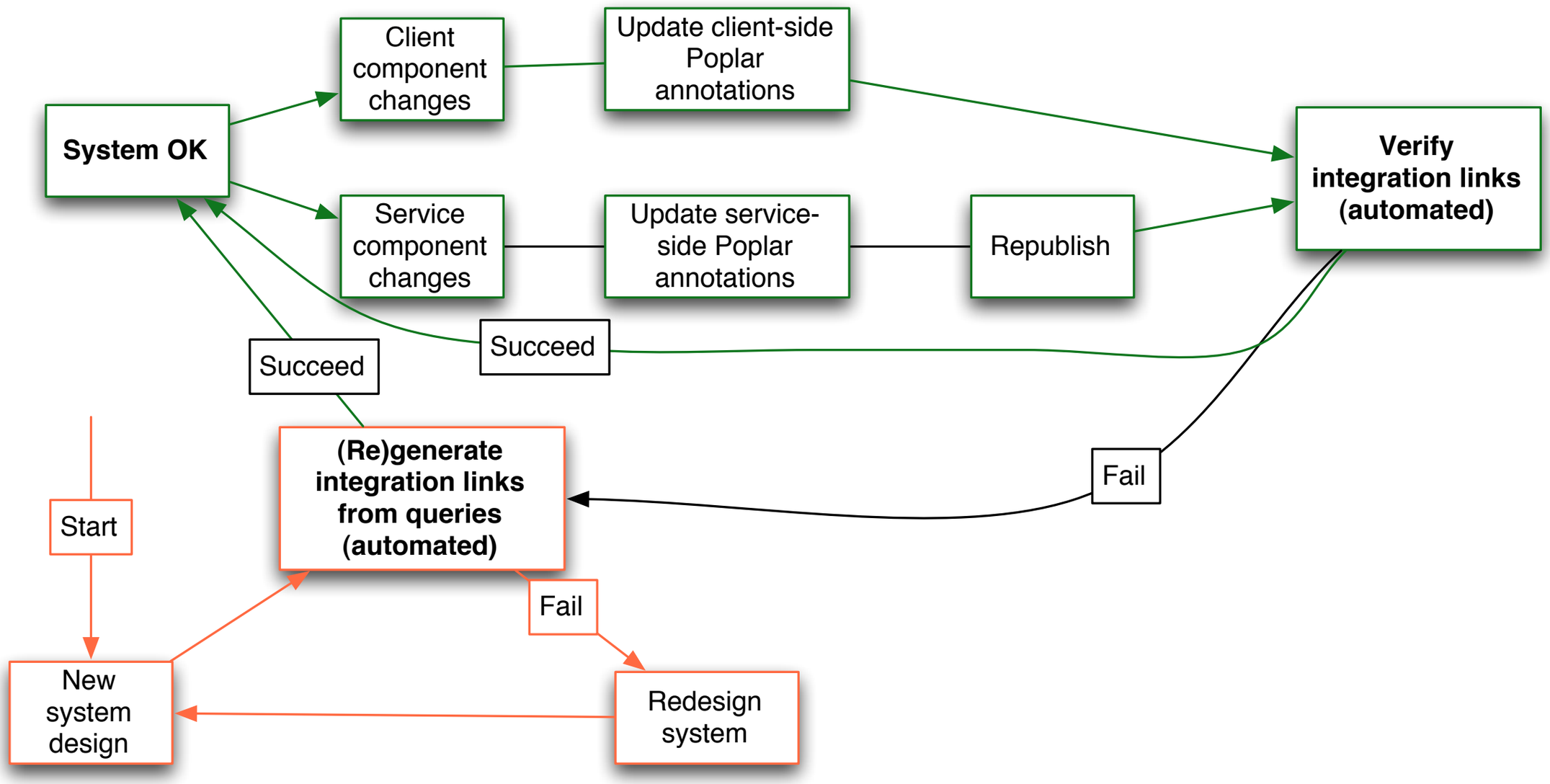
Integrating with the new Connector

```
class Connector {
    tags(Address) connectionAddress;

    resource connection {
        properties @connected, @disconnected, @configured;

        void setAddress(Address a)
            a: connectionAddress;
             this: +@configured. { // ... }

        void connect()
            this: +@connected, @configured. { //... }

        void disconnect()
            this: -@connected, +@disconnected. { //... }
    }
}
```

```
class Client {
    Address a(connectionAddress);                    Generated code
    void m(Connector c) mutates c.connection:
        c: {
            #transform(c, @connected);               c.setAddress(a);
                                                     c.connect();
            #transform(c, @disconnected);            c.disconnect();
    }
}
```



Intended workflow



Limitations of our design

- We don’t encode control flow information (branches, loops and so on), so this must be handled externally
- No support for exceptions, concurrency, generics
- It is essential to preserve the meaning of *each individual label* across program upgrades

Jardine, a Poplar implementation

- The first prototype release will be available for download at http://www.poplar-lang.org very soon

- Based on David Pearce's JKit[1] Java compiler; written in Java and Scala

- Checks Poplar constraints and solves queries, compiles Poplar source files to Java classes

- Co-implemented with Alexandre Pichot, UPMC, France

1. Pearce, David. "JPure: A Modular Purity System for Java.", CC 2011

Summary of results

- The language design appears to be useful for describing real APIs (Swing, JDBC)
  - Modular compilation, polymorphism
  - A strategy for handling aliasing
  - Built-in slack allows for future evolution
- A formalisation based on Middleweight Java[1]

1. Nyström-Persson, Pichot and Honiden. *Evolvable Java Composition with Stateful Labels and Effects.* ESOP 2012, under review.

Conclusion

- We propose a novel approach to Java component specification and integration, based on AI planning
  - We combine ideas from typestate checking and from effect systems
  - Verification of code, generation of integration links and verification of integration links are all possible
- **Initial integration becomes easy**
- **Staying integrated when components evolve becomes easy**

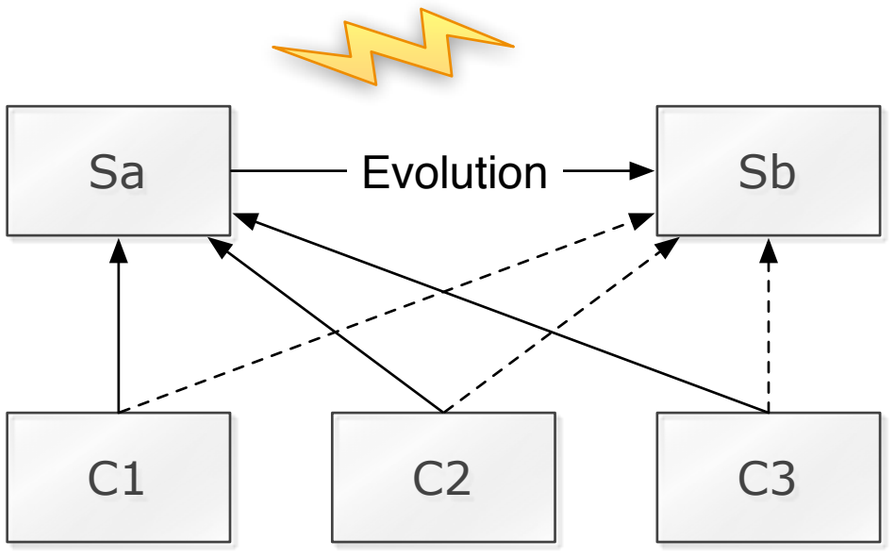
Future work

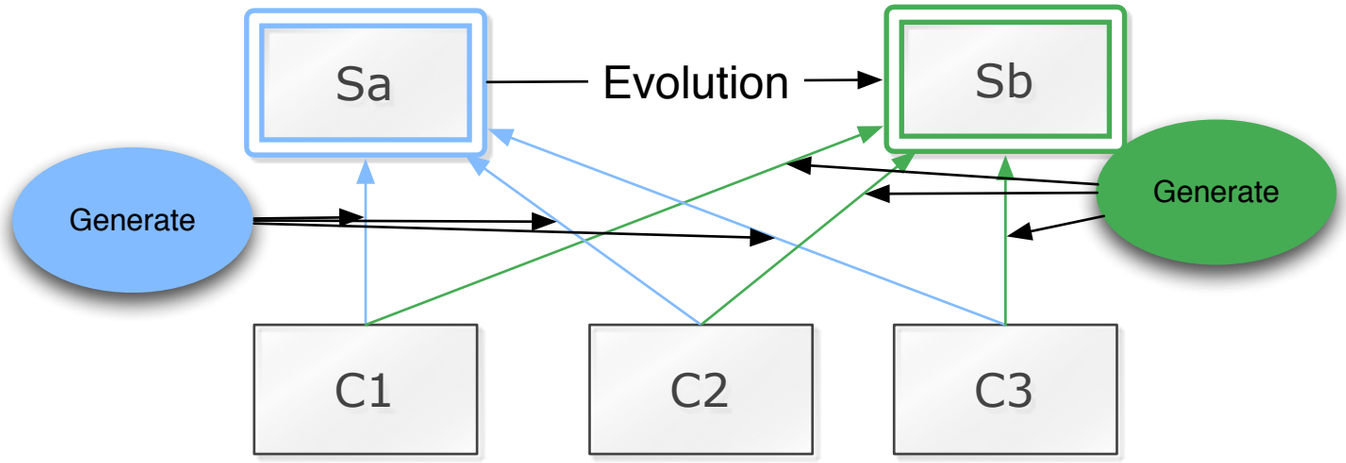
- Case studies on software evolution in practice
- Further Jardine improvements
    - Minor enhancements and bugfixes (short term)
    - Verification of integration links (long term)
- More comprehensive formalisation
    - Type safety proof
- Full “Poplarisation” of selected Java libraries

The presentation ends here. Extra slides after this point.

Generating and evolving integration links

Integration link = query + solution





Related: Middleweight Java (MJ)

- Minimal calculus for an imperative fragment of Java (with side effects) by Parkinson, Bierman, Pitts[1]

- The MJ authors prove the safety of a simplified form of the Boyland-Greenhouse effect system

- **We use MJ as a basis for formalisation of Poplar.**

- FJ is a more well-known core calculus for Java, but unsuitable, since it has no side effects (in particular, no assignment)[2]

1. Bierman, Parkinson and Pitts. “MJ: An Imperative Core Calculus for Java and Java with Effects.” 2003
2. Igarashi, Pierce and Wadler. “Featherweight Java: A Minimal Core Calculus for Java and GJ.” TOPLAS, 2001

Other related work

- Label-selective lambda calculus[1]
    - As far as we know, Poplar is the first application of label based argument selection to Java
- Fully automated adaptation of ML components[2]
- Ownership systems[3]
- Jungloid mining[4]

1. Garrigue and Furuse. “A Label-Selective Lambda Calculus with Optional Arguments and its Compilation Method.” 1995
2. Haack, Howard, Stoughton and Wells. “Fully Automated Adaptation of Software Components Based on Semantic Specifications”, AMAST 2002
3. Clarke, Potter and Noble. “Simple Ownership Types for Object Containment.” ECOOP 2001
4. Mandelin, Xu, Kimelman and Bodik. “Jungloid Mining: Helping to Navigate the API Jungle.” PLDI 2005

Conflict of evolution and integration

- Software developers often want to refactor their code, i.e. make systematic structural changes

- It has been found[1] that a majority of breaking changes (changes that cause bugs or compilation errors) in large software systems stem from refactoring

- Bloch[2] and Fowler[3] recommend that published interfaces should be changed minimally and that they should be as small as possible

- **The greater the number of components that must remain integrated, the harder it is to evolve the program**

1. Dig and Johnson. “The Role of Refactorings in API Evolution.” ICSM 2005
2. Bloch. “Effective Java.” 2001
3. Fowler et al. “Refactoring.” 1999

Additional features of Poplar

- Composite properties: alternative names for conjunctions of labels
- External resources/properties: labels of one object can be represented in another object

Handling aliasing with uniqueness kinds

- **Uniqueness kinds** classify each variable according to a) uniqueness or non-uniqueness, and b) preservation of uniqueness
  - **Unique**: definitely unique, preserves uniqueness
  - **Maintain**: possibly unique, preserves uniqueness
  - **Normal**: not unique, does not preserve uniqueness
  - **Fresh**: new and unique, can be converted to any other uniqueness kind

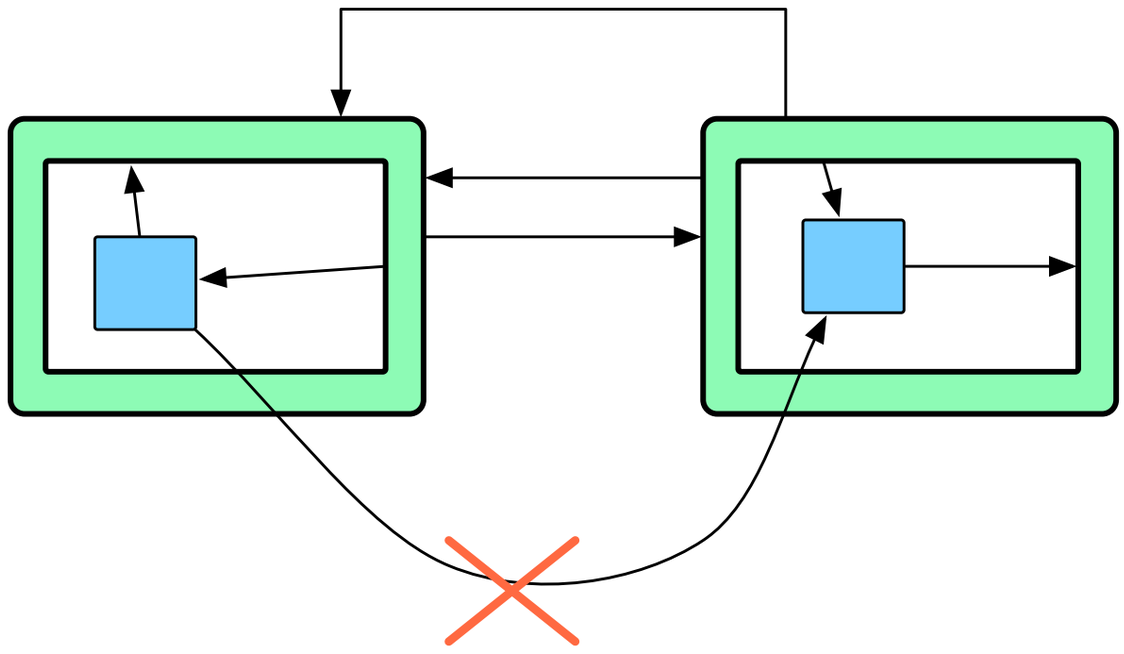
```
class Foo {

    static Foo getFoo() result: normal. { ... }

    Foo() result: fresh. { ... }

    void baz() mutates this.baz;

    void m1() mutates any(Foo).baz: {
        Foo f = getFoo(); //f is aliased
        f.baz();
    }

    void m2 {
        Foo f(unique) = new Foo(); //f is now
        unique
        f.baz;
    }
```

Managed/plain boundary

- We divide each component into *plain code* and *managed code*

- Constraints such as **unique** need only be true in managed code, and Poplar will only use managed code for integrations (logical uniqueness rather than true uniqueness)

- We make this practical by constraining the flow of data between components to be in managed code

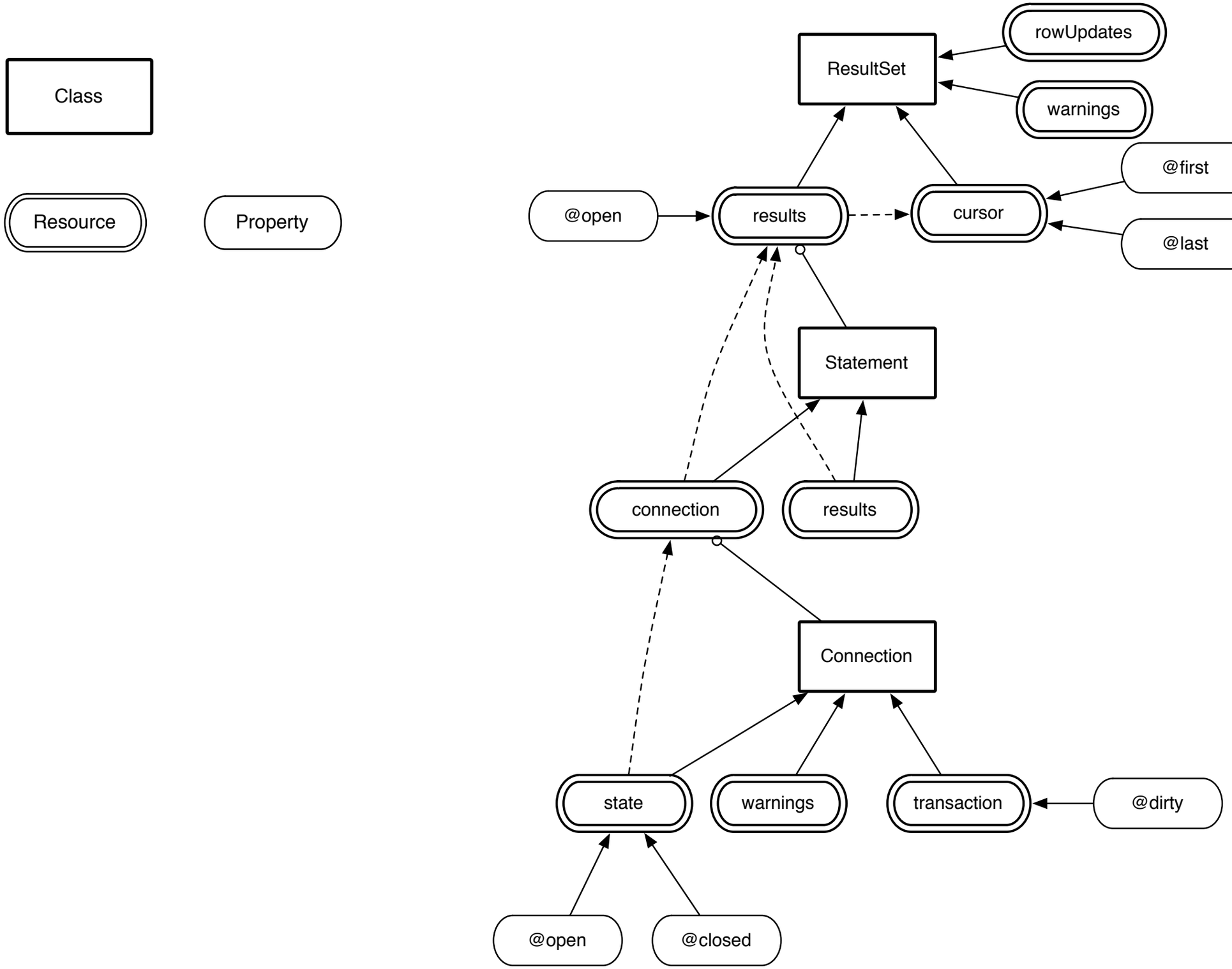


Example: registry with managed/plain code

- Managed code is simply code that has associated Poplar signatures
- The result of Item() is not truly unique, but it is unique within the managed code - “logically unique”
- We may view the managed code as an *integration interface*
- The extra aliases that are created in the plain code must not be exposed in managed code

```
class Item {
    static List<Item> allItems;

    //...
    tags usefulItem;
    tags(int) numItems;

    Item() result: +usefulItem, unique. {
        allItems.add(this);
    }

    resource state {
        properties @used;
        void use()
            this: ++@used. {
        }
    }

    int countItems() result: +numItems. {
        return allItems.size();
    }

}

class User {
    void useItem(Item i) mutates i.state:
        i: unique, +@used. {
            i.use();
        }
    }


    //if “i” were not unique, the above would be:
    //mutates any(Item).state
}
```

Resource structure (partial) for JDBC



Example code

- Tags are like properties, but cannot be erased

```
class Statement {
   tags(ResultSet) statementResult;
   tags(String) sqlQuery;

   resource(ResultSet) results {
        properties @open;
        links results;
        links ext(Connection).connection;
   }

   resource results {
        ResultSet executeQuery(String query)
           query: sqlQuery, result: statementResult,
+@open. { }
   }
}
```

```
class ResultSet {
   tags(int) resultData, columnIndex;
   tags(String) resultData;

   tags updatable;

   resource warnings {
        void clearWarnings() {}
   }

   resource rowUpdates {
        properties @dirty;
        void cancelRowUpdates() this:updatable. {}
        void updateBoolean(int idx, boolean x) this: updatable, +@dirty. {}
        void updateInt(int idx, int x) this: updatable, +@dirty. {}
        // etc...

        void updateRow() {}
   }

   resource cursor {
        properties @first, @last;

        boolean relative(int rows) this: @open. {}
        boolean absolute(int pos) this: @open. {}
        boolean next() this: @open. {}
        boolean previous() this: @open. {}
        void first() this: @open, +@first. {}
        void last() this: @open, +@last. {}
   }

   int getInt(int index)
        this: @open, index: columnIndex, result: resultData. {}

   String getString(int index)
        this: @open, index: columnIndex, result: resultData. {}
}
```

Partial Order Planning algorithm (POP)

1. Initialise plan to have two pseudo-actions, **start** and **finish**
2. If there are no open preconditions, stop
3. Select an open precondition
4. Create each possible successor with new and existing actions
5. Resolve conflicts in each successor by strengthening ordering
6. Go to step 2