
Reasoning About Class Behavior Using the Bisimulation Technique

No.63 NII Shonan Meeting

Vasileios Koutavas

supported, in part, by Science
Foundation Ireland grant 13/RC/2094

Trinity College Dublin

joint work with **Mitchell Wand**

Northeastern University

Roadmap

- A Motivating Example
- Contextual Equivalence
- Technology for Proving Equivalences
- Deriving Bisimulations for Equivalence
- An Actual Proof of Equivalence
- Language Extensions

A Cell Class

```
class Cell extends Object {  
    private Object g;  
    Cell() { g = null; }  
    public void set(Object b) {  
        g = b;  
    }  
    public Object get() {  
        return g;  
    }  
}
```

A Cell Class

```
class Cell extends Object {
    private Object g;
    Cell() { g = null; }
    public void set(Object b) {
        g = b;
    }
    public Object get() {
        return g;
    }
}
```

Some Program

```
// Project: ACellTestDemo
// File: ACellTestDemo, ACellTestDemo
// Author: 2008
// Description:
// This program demonstrates the use of the Cell class.
// It shows how to create a Cell object and use its set and get methods.
// The program is divided into two parts: a main method and a test method.
// The main method creates a Cell object and sets its value to "A".
// The test method creates a Cell object and sets its value to "B".
// The program is compiled and run using the following commands:
// javac ACellTestDemo.java
// java ACellTestDemo

public class ACellTestDemo {
    private ACellTestDemo acd = null;
    private String testString = "A"; // See ACellTestDemo.java

    public ACellTestDemo(String testString) {
        this.testString = testString;
    }

    // Create a Cell object and set its value to "A".
    public void main() {
        Cell c = new Cell();
        c.set("A");
        System.out.println("Cell value: " + c.get());
    }

    // Create a Cell object and set its value to "B".
    public void test() {
        Cell c = new Cell();
        c.set("B");
        System.out.println("Cell value: " + c.get());
    }

    // Create a Cell object and set its value to "A".
    public void test2() {
        Cell c = new Cell();
        c.set("A");
        System.out.println("Cell value: " + c.get());
    }
}
```


A Cell Class

```
class Cell extends Object {
  private Object g1, g2;
  private int cnt;
  Cell() {
    g1 = null; g2 = null; cnt = 0;
  }
  public void set(Object b) {
    cnt = cnt + 1;
    if ((cnt % 2) == 0)
      g1 = b;
    else
      g2 = b;
  }
  public Object get() {
    if ((cnt % 2) == 0)
      return g1;
    else
      return g2;
  }
}
```

A Cell Class

```
class Cell extends Object {
  private Object g1, g2;
  private int cnt;
  Cell() {
    g1 = null; g2 = null; cnt = 0;
  }
  public void set(Object b) {
    cnt = cnt + 1;
    if ((cnt % 2) == 0)
      g1 = b;
    else
      g2 = b;
  }
  public Object get() {
    if ((cnt % 2) == 0)
      return g1;
    else
      return g2;
  }
}
```

A Cell Class

```
class Cell extends Object {
    private Object g1, g2;
    private int cnt;
    Cell() {
        g1 = null; g2 = null; cnt = 0;
    }
    public void set(Object b) {
        cnt = cnt + 1;
        if ((cnt % 2) == 0)
            g1 = b;
        else
            g2 = b;
    }
    public Object get() {
        if ((cnt % 2) == 0)
            return g1;
        else
            return g2;
    }
}
```


A Cell Class

```
class Cell extends Object {
    private Object g1, g2;
    private int cnt;
    Cell() {
        g1 = null; g2 = null; cnt = 0;
    }
    public void set(Object b) {
        cnt = cnt + 1;
        if ((cnt % 2) == 0)
            g1 = b;
        else
            g2 = b;
    }
    public Object get() {
        if ((cnt % 2) == 0)
            return g1;
        else
            return g2;
    }
}
```

A Cell Class

```
class Cell extends Object {
  private Object g1, g2;
  private int cnt;
  Cell() {
    g1 = null; g2 = null; cnt = 0;
  }
  public void set(Object b) {
    cnt = cnt + 1;
    if ((cnt % 2) == 0)
      g1 = b;
    else
      g2 = b;
  }
  public Object get() {
    if ((cnt % 2) == 0)
      return g1;
    else
      return g2;
  }
}
```

The Same Program

```
// File: ACell.java
// Author: A. K. Kulkarni, A. Kulkarni
// Date: 2001
// Description: A Cell class that implements the Cell interface.
// It has two private fields g1 and g2, and a private field cnt.
// The class has a constructor, a set method, and a get method.
// The set method increments cnt and sets g1 or g2 based on cnt.
// The get method returns g1 or g2 based on cnt.

public class ACell implements Cell {
  private Object g1 = null;
  private Object g2 = null;
  private int cnt = 0;

  public ACell() {
    g1 = null;
    g2 = null;
    cnt = 0;
  }

  public void set(Object b) {
    cnt = cnt + 1;
    if ((cnt % 2) == 0)
      g1 = b;
    else
      g2 = b;
  }

  public Object get() {
    if ((cnt % 2) == 0)
      return g1;
    else
      return g2;
  }
}
```

A Cell Class

The Same Program

```
// File: ACellClass.java
// Author: A. K. ...
// Date: 2014
// ...
public class MMRepresentation
{
    private boolean has = null;
    private String[][] graph = null; // see String[][] graph.java
    public MMRepresentation(String headerLine, String[][] graph)
    {
        this.has = has;
        this.graph = graph;
        ...
    }
    ...
}

// ...

public String getHeaderLine() // Returns the header line as a String
{
    return headerLine;
}

// ...

public String getGraph() // Returns the graph as a String
{
    ...
}

// ...

public String getHeaderLineAndGraph() // Returns the header line and graph as a String
{
    ...
}

// ...
```



A Cell Class

```
// Project: AtomsCellClass
// Author(s): A.McCarthy, A.Kochergov
// 2016-2017
//
// Description: This class implements the Cell class, which is used to represent a cell in a grid.
// It contains methods for getting and setting the cell's state, and for checking if the cell is
// adjacent to another cell.
//
// Usage:
//   Cell cell;
//   cell.setState(1);
//   int state = cell.getState();
//   bool adjacent = cell.isAdjacent(otherCell);
```

```
public class AtomsCellClass
{
private:
    int m_state; // state of the cell
    int m_x; // x coordinate of the cell
    int m_y; // y coordinate of the cell

public:
    AtomsCellClass(int state, int x, int y) : m_state(state), m_x(x), m_y(y) {}

    // Getters
    int getState() const { return m_state; }
    int getX() const { return m_x; }
    int getY() const { return m_y; }

    // Setters
    void setState(int state) { m_state = state; }
    void setX(int x) { m_x = x; }
    void setY(int y) { m_y = y; }

    // Adjacency
    bool isAdjacent(const AtomsCellClass& other) const
    {
        return (abs(m_x - other.getX()) == 1 || abs(m_y - other.getY()) == 1);
    }

    // String representation
    string toString() const
    {
        return to_string(m_state) + " " + to_string(m_x) + " " + to_string(m_y);
    }
};
```



```
// Project: AtomsCellClass
// Author(s): A.McCarthy, A.Kochergov
// 2016-2017
//
// Description: This class implements the Cell class, which is used to represent a cell in a grid.
// It contains methods for getting and setting the cell's state, and for checking if the cell is
// adjacent to another cell.
//
// Usage:
//   Cell cell;
//   cell.setState(1);
//   int state = cell.getState();
//   bool adjacent = cell.isAdjacent(otherCell);
```

```
public class AtomsCellClass
{
private:
    int m_state; // state of the cell
    int m_x; // x coordinate of the cell
    int m_y; // y coordinate of the cell

public:
    AtomsCellClass(int state, int x, int y) : m_state(state), m_x(x), m_y(y) {}

    // Getters
    int getState() const { return m_state; }
    int getX() const { return m_x; }
    int getY() const { return m_y; }

    // Setters
    void setState(int state) { m_state = state; }
    void setX(int x) { m_x = x; }
    void setY(int y) { m_y = y; }

    // Adjacency
    bool isAdjacent(const AtomsCellClass& other) const
    {
        return (abs(m_x - other.getX()) == 1 || abs(m_y - other.getY()) == 1);
    }

    // String representation
    string toString() const
    {
        return to_string(m_state) + " " + to_string(m_x) + " " + to_string(m_y);
    }
};
```



A Cell Class

```
// Project: AtomsOfGenetics
// LeetCode, 461, Hamming Distance, A.K. Rodrigo
// 03/18/2016
//
// Description:
// Given two integers x and y, calculate the Hamming distance between them. The Hamming distance between two integers is the number of positions at which the corresponding bits are different.
// For example, x = 1 (0000 1000) and y = 4 (0000 0100), the Hamming distance between x and y is 2 because they differ at the second and third bits from the right.
//
// Input: Two integers, x and y.
// Output: An integer, the Hamming distance between x and y.
// Example:
// Input: 1, 4
// Output: 2
```

```
public class HammingDistance {
    private HammingDist ham = null;
    private HammingDistance hamming = null; // see HammingDistance.java

    public HammingDistance(HammingDistance ham, HammingDistance hamming) {
        this.ham = ham;
        this.hamming = hamming;
    }

    public HammingDistance() {
        ham = null;
        hamming = null;
    }
}
```

```
public String getHammingDistance() {
    String str = "";
    if (ham == null)
        return str;
    for (int i = 0; i < ham.getLength(); i++)
        str += ham.getBit(i);
    return str;
}
```

```
public String getHammingDistanceFromStrings() {
    String str = "";
    HammingDistance ham = hamming.getHammingDistanceFromStrings();
    for (int i = 0; i < ham.getLength(); i++)
        str += ham.getBit(i);
    return str;
}
```

```
public String getHammingDistanceFromStringsAndStrings() {
    String str = "";
    HammingDistance ham = null;
    HammingDistance hamming = hamming.getHammingDistanceFromStringsAndStrings();
    for (int i = 0; i < ham.getLength(); i++)
        str += ham.getBit(i);
    return str;
}
```



```
public String getHammingDistanceFromStringsAndStringsAndStrings() {
    String str = "";
    HammingDistance ham = null;
    HammingDistance hamming = hamming.getHammingDistanceFromStringsAndStringsAndStrings();
    for (int i = 0; i < ham.getLength(); i++)
        str += ham.getBit(i);
    return str;
}
```

```
// Project: AtomsOfGenetics
// LeetCode, 461, Hamming Distance, A.K. Rodrigo
// 03/18/2016
//
// Description:
// Given two integers x and y, calculate the Hamming distance between them. The Hamming distance between two integers is the number of positions at which the corresponding bits are different.
// For example, x = 1 (0000 1000) and y = 4 (0000 0100), the Hamming distance between x and y is 2 because they differ at the second and third bits from the right.
//
// Input: Two integers, x and y.
// Output: An integer, the Hamming distance between x and y.
// Example:
// Input: 1, 4
// Output: 2
```

```
public class HammingDistance {
    private HammingDist ham = null;
    private HammingDistance hamming = null; // see HammingDistance.java

    public HammingDistance(HammingDistance ham, HammingDistance hamming) {
        this.ham = ham;
        this.hamming = hamming;
    }

    public HammingDistance() {
        ham = null;
        hamming = null;
    }
}
```

```
public String getHammingDistance() {
    String str = "";
    if (ham == null)
        return str;
    for (int i = 0; i < ham.getLength(); i++)
        str += ham.getBit(i);
    return str;
}
```

```
public String getHammingDistanceFromStrings() {
    String str = "";
    HammingDistance ham = hamming.getHammingDistanceFromStrings();
    for (int i = 0; i < ham.getLength(); i++)
        str += ham.getBit(i);
    return str;
}
```

```
public String getHammingDistanceFromStringsAndStrings() {
    String str = "";
    HammingDistance ham = null;
    HammingDistance hamming = hamming.getHammingDistanceFromStringsAndStrings();
    for (int i = 0; i < ham.getLength(); i++)
        str += ham.getBit(i);
    return str;
}
```



```
public String getHammingDistanceFromStringsAndStringsAndStrings() {
    String str = "";
    HammingDistance ham = null;
    HammingDistance hamming = hamming.getHammingDistanceFromStringsAndStringsAndStrings();
    for (int i = 0; i < ham.getLength(); i++)
        str += ham.getBit(i);
    return str;
}
```

A Cell Class

Is the behavior the same for any possible pair of programs?

```
// File: Cell.java
// Author: A. A. Chaitin, A. A. Chaitin
// Date: 2001
// Description: A class representing a cell in a grid.
// The cell has a state and a color.
// The state is an integer from 0 to 9.
// The color is a string from "black" to "white".
// The cell can be created with a state and a color.
// The cell can be updated with a new state and color.
// The cell can be compared with another cell.
// The cell can be converted to a string.
// The cell can be converted from a string.
// The cell can be converted to a double.
// The cell can be converted from a double.
// The cell can be converted to a float.
// The cell can be converted from a float.
// The cell can be converted to a long.
// The cell can be converted from a long.
// The cell can be converted to a short.
// The cell can be converted from a short.
// The cell can be converted to a byte.
// The cell can be converted from a byte.
// The cell can be converted to a char.
// The cell can be converted from a char.
// The cell can be converted to a boolean.
// The cell can be converted from a boolean.
// The cell can be converted to a byte array.
// The cell can be converted from a byte array.
// The cell can be converted to a short array.
// The cell can be converted from a short array.
// The cell can be converted to a long array.
// The cell can be converted from a long array.
// The cell can be converted to a float array.
// The cell can be converted from a float array.
// The cell can be converted to a double array.
// The cell can be converted from a double array.
// The cell can be converted to a string array.
// The cell can be converted from a string array.
// The cell can be converted to a double array.
// The cell can be converted from a double array.
// The cell can be converted to a float array.
// The cell can be converted from a float array.
// The cell can be converted to a long array.
// The cell can be converted from a long array.
// The cell can be converted to a short array.
// The cell can be converted from a short array.
// The cell can be converted to a byte array.
// The cell can be converted from a byte array.
// The cell can be converted to a char array.
// The cell can be converted from a char array.
// The cell can be converted to a boolean array.
// The cell can be converted from a boolean array.
// The cell can be converted to a byte array.
// The cell can be converted from a byte array.
// The cell can be converted to a short array.
// The cell can be converted from a short array.
// The cell can be converted to a long array.
// The cell can be converted from a long array.
// The cell can be converted to a float array.
// The cell can be converted from a float array.
// The cell can be converted to a double array.
// The cell can be converted from a double array.
// The cell can be converted to a string array.
// The cell can be converted from a string array.
```

```
public class CellRepresentation {
    private boolean debug = false;
    private String[] grid;
    private int[][] state;
    private int[][] color;

    public CellRepresentation(String[] grid, int[][] state, int[][] color) {
        this.grid = grid;
        this.state = state;
        this.color = color;
    }

    public String toString() {
        String s = "";
        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[i].length; j++) {
                s += grid[i][j] + " ";
            }
            s += "\n";
        }
        return s;
    }
}
```

```
public String toString() {
    // Method to return the string representation of the grid.
    String s = "";
    for (int i = 0; i < grid.length; i++) {
        for (int j = 0; j < grid[i].length; j++) {
            s += grid[i][j] + " ";
        }
        s += "\n";
    }
    return s;
}
```

```
public String toString() {
    // Method to return the string representation of the grid.
    String s = "";
    for (int i = 0; i < grid.length; i++) {
        for (int j = 0; j < grid[i].length; j++) {
            s += grid[i][j] + " ";
        }
        s += "\n";
    }
    return s;
}
```



```
public String toString() {
    // Method to return the string representation of the grid.
    String s = "";
    for (int i = 0; i < grid.length; i++) {
        for (int j = 0; j < grid[i].length; j++) {
            s += grid[i][j] + " ";
        }
        s += "\n";
    }
    return s;
}
```

```
// File: Cell.java
// Author: A. A. Chaitin, A. A. Chaitin
// Date: 2001
// Description: A class representing a cell in a grid.
// The cell has a state and a color.
// The state is an integer from 0 to 9.
// The color is a string from "black" to "white".
// The cell can be created with a state and a color.
// The cell can be updated with a new state and color.
// The cell can be compared with another cell.
// The cell can be converted to a string.
// The cell can be converted from a string.
// The cell can be converted to a double.
// The cell can be converted from a double.
// The cell can be converted to a float.
// The cell can be converted from a float.
// The cell can be converted to a long.
// The cell can be converted from a long.
// The cell can be converted to a short.
// The cell can be converted from a short.
// The cell can be converted to a byte.
// The cell can be converted from a byte.
// The cell can be converted to a char.
// The cell can be converted from a char.
// The cell can be converted to a boolean.
// The cell can be converted from a boolean.
// The cell can be converted to a byte array.
// The cell can be converted from a byte array.
// The cell can be converted to a short array.
// The cell can be converted from a short array.
// The cell can be converted to a long array.
// The cell can be converted from a long array.
// The cell can be converted to a float array.
// The cell can be converted from a float array.
// The cell can be converted to a double array.
// The cell can be converted from a double array.
// The cell can be converted to a string array.
// The cell can be converted from a string array.
```

```
public class CellRepresentation {
    private boolean debug = false;
    private String[] grid;
    private int[][] state;
    private int[][] color;

    public CellRepresentation(String[] grid, int[][] state, int[][] color) {
        this.grid = grid;
        this.state = state;
        this.color = color;
    }

    public String toString() {
        String s = "";
        for (int i = 0; i < grid.length; i++) {
            for (int j = 0; j < grid[i].length; j++) {
                s += grid[i][j] + " ";
            }
            s += "\n";
        }
        return s;
    }
}
```

```
public String toString() {
    // Method to return the string representation of the grid.
    String s = "";
    for (int i = 0; i < grid.length; i++) {
        for (int j = 0; j < grid[i].length; j++) {
            s += grid[i][j] + " ";
        }
        s += "\n";
    }
    return s;
}
```

```
public String toString() {
    // Method to return the string representation of the grid.
    String s = "";
    for (int i = 0; i < grid.length; i++) {
        for (int j = 0; j < grid[i].length; j++) {
            s += grid[i][j] + " ";
        }
        s += "\n";
    }
    return s;
}
```



```
public String toString() {
    // Method to return the string representation of the grid.
    String s = "";
    for (int i = 0; i < grid.length; i++) {
        for (int j = 0; j < grid[i].length; j++) {
            s += grid[i][j] + " ";
        }
        s += "\n";
    }
    return s;
}
```

A Cell Class

Is the behavior the same for any possible pair of programs?

```
// File: CellRepresentation.cpp
// Author: A. K. ...
// Date: 2014
// Description: ...
```

```
public class CellRepresentation {
public:
    CellRepresentation(int x, int y) : m_x(x), m_y(y) {}
    // ...
};
```

```
public string getAsString() const {
    return m_x + " " + m_y;
}
```

```
public string getAsStringIncludingNeighbors() const {
    // ...
}
```

```
public string getAsStringIncludingNeighborsAndDirections() const {
    // ...
}
```

```
public string getAsStringIncludingNeighborsAndDirectionsAndColors() const {
    // ...
}
```

```
public string getAsStringIncludingNeighborsAndDirectionsAndColorsAndWeights() const {
    // ...
}
```

```
public string getAsStringIncludingNeighborsAndDirectionsAndColorsAndWeightsAndDirections() const {
    // ...
}
```

```
public string getAsStringIncludingNeighborsAndDirectionsAndColorsAndWeightsAndDirectionsAndWeights() const {
    // ...
}
```

```
public string getAsStringIncludingNeighborsAndDirectionsAndColorsAndWeightsAndDirectionsAndWeightsAndDirections() const {
    // ...
}
```

```
// File: CellRepresentation.cpp
// Author: A. K. ...
// Date: 2014
// Description: ...
```

```
public class CellRepresentation {
public:
    CellRepresentation(int x, int y) : m_x(x), m_y(y) {}
    // ...
};
```

```
public string getAsString() const {
    return m_x + " " + m_y;
}
```

```
public string getAsStringIncludingNeighbors() const {
    // ...
}
```

```
public string getAsStringIncludingNeighborsAndDirections() const {
    // ...
}
```

```
public string getAsStringIncludingNeighborsAndDirectionsAndColors() const {
    // ...
}
```

```
public string getAsStringIncludingNeighborsAndDirectionsAndColorsAndWeights() const {
    // ...
}
```

```
public string getAsStringIncludingNeighborsAndDirectionsAndColorsAndWeightsAndDirections() const {
    // ...
}
```

```
public string getAsStringIncludingNeighborsAndDirectionsAndColorsAndWeightsAndDirectionsAndWeights() const {
    // ...
}
```

```
public string getAsStringIncludingNeighborsAndDirectionsAndColorsAndWeightsAndDirectionsAndWeightsAndDirections() const {
    // ...
}
```

Contextual Equivalence

Contextual Equivalence (\equiv)

- **C** and **C'** are **contextually equivalent** iff:

For any class-table context **CT[]**:

CT[C] and **CT[C']** have the same behavior

Contextual Equivalence (\equiv)

- **C** and **C'** are **contextually equivalent** iff:

For any class-table context **CT[]**:

$$\mathbf{CT[C]} \Downarrow \text{ iff } \mathbf{CT[C']} \Downarrow$$

Contextual Equivalence (\equiv)

- **C** and **C'** are **contextually equivalent** iff:

For any class-table context **CT[]**:

$$\mathbf{CT[C]} \Downarrow \text{ iff } \mathbf{CT[C']} \Downarrow$$

- The “golden standard of equivalences”
- Hard to use directly to prove non-trivial equivalences

Technology for Proving Equivalence

- Denotational Semantics
 - Usual Denotational Semantics + Logical Relations [Pitts]
 - Usual Denotational Semantics + Bisimulations [Banerjee&Naumann]
 - Game Semantics + “quotient” [Reynolds, Hyland&Ong]
 - Fully abstract Game Semantics [Abramsky+, Murawski&Tzevelekos,Laird]
- Trace Equivalence [Jeffrey&Rathke,Laird]
- Operational Semantics + Logical Relations [Pitts, Pitts&Stark,Appel&McAllester,Ahmed,Dreyer+]
- Operational Semantics + Bisimulations [Abramsky, Sumii&Pierce, Koutavas&Wand, Sangiorgi+]

Technology for Proving Equivalence

Java

- Denotational Semantics
 - Usual Denotational Semantics + Logical Relations [Pitts]
 - Usual Denotational Semantics + Bisimulations [Banerjee&Naumann]
 - Game Semantics + “quotient” [Reynolds, Hyland&Ong]
 - Fully abstract Game Semantics [Abramsky+, Murawski&Tzevelekos, Laird]
- Trace Equivalence [Jeffrey&Rathke, Laird]
- Operational Semantics + Logical Relations [Pitts, Pitts&Stark, Appel&McAllester, Ahmed, Dreyer+]
- Operational Semantics + Bisimulations [Abramsky, Sumii&Pierce, Koutavas&Wand, Sangiorgi+]

Technology for Proving Equivalence

This talk: Environmental
Bisimulations for Java

- Denotational Semantics
 - Usual Denotational Semantics + Logical Relations [Pitts]
 - Usual Denotational Semantics + Bisimulations [Banerjee&Naumann]
 - Game Semantics + “quotient” [Reynolds, Hyland&Ong]
 - Fully abstract Game Semantics [Abramsky+, Murawski&Tzevelekos,Laird]
- Trace Equivalence [Jeffrey&Rathke,Laird]
- Operational Semantics + Logical Relations [Pitts, Pitts&Stark,Appel&McAllester,Ahmed,Dreyer+]
- Operational Semantics + Bisimulations [Abramsky, Sumii&Pierce, **Koutavas&Wand**, Sangiorgi+]

Technology for Proving Equivalence

- The standard definition of (\equiv) describes the largest set of equivalent classes

$$(\equiv) = \{ (C, C') \mid \dots \}$$

Technology for Proving Equivalence

- The standard definition of (\equiv) describes the largest set of equivalent classes

$$(\equiv) = \{ (C, C') \mid \dots \}$$

- Find a more “convenient” definition for a set R and show:
 - $R \subseteq (\equiv)$ (Soundness of R)
 - $(\equiv) \subseteq R$ (Completeness of R)

Technology for Proving Equivalence

- The standard definition of (\equiv) describes the largest set of equivalent classes

$$(\equiv) = \{ (C, C') \mid \dots \}$$

- Find a more “convenient” definition for a set R and show:
 - $R \subseteq (\equiv)$ (Soundness of R)
 - $(\equiv) \subseteq R$ (Completeness of R)
- Use the definition of R in proofs instead of (\equiv)

Challenges of the Defined Set R

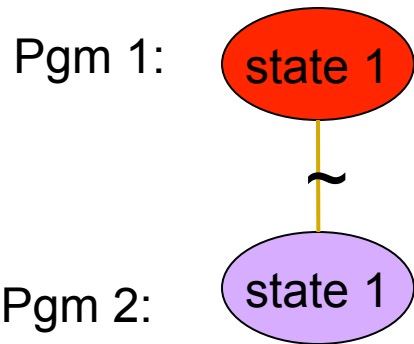
- For a Java-like language R should help reason about:
 - Different **state fields** in related classes
 - **Inheritance** and **(down-)casting** as a way of distinguishing classes by the context
 - Higher-order features—**Callbacks**

Roadmap

- *A Motivating Example*
- *Contextual Equivalence*
- *Technology for Proving Equivalences*
- **Deriving Bisimulations for Equivalence**
- **An Actual Proof of Equivalence**
- **Language Extensions**

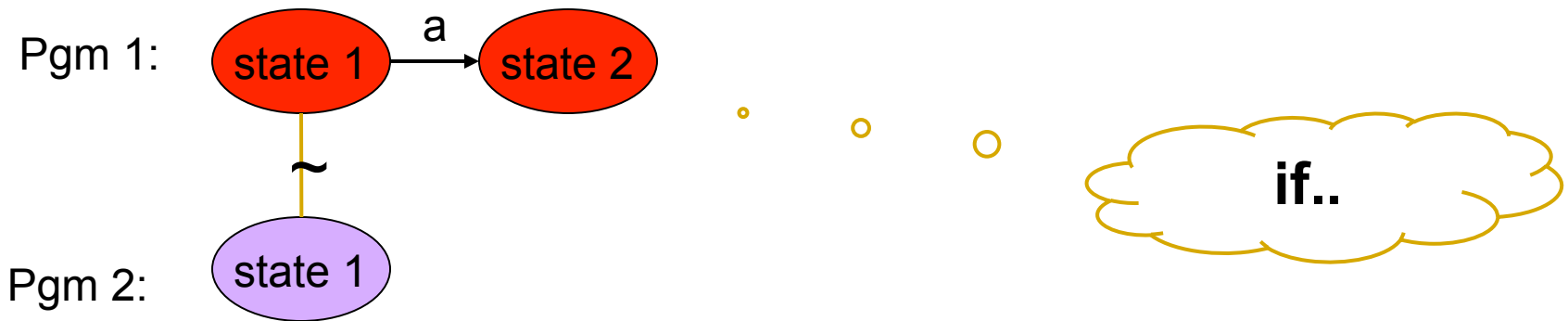
Bisimulations

- Introduced in Concurrent Calculi [Milner], and adapted to functional languages [Abramsky].



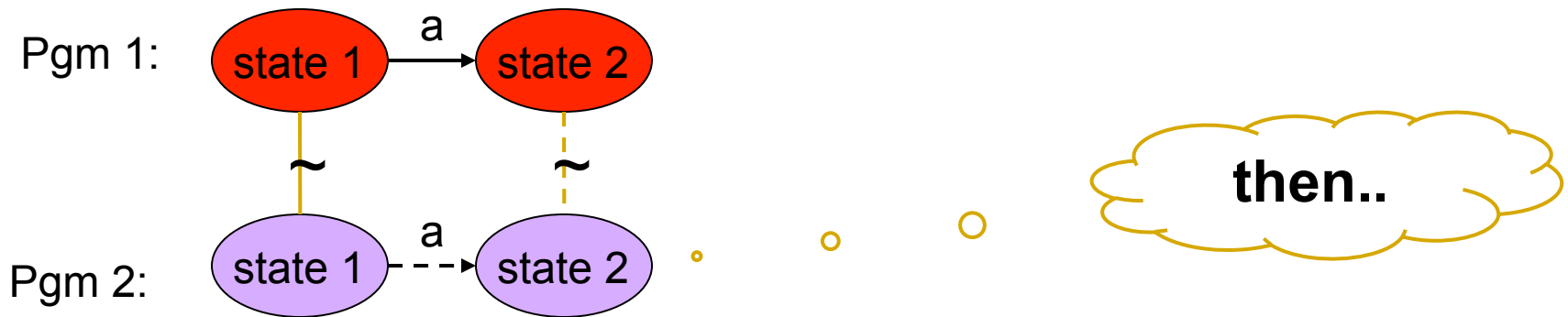
Bisimulations

- Introduced in Concurrent Calculi [Milner], and adapted to functional languages [Abramsky].



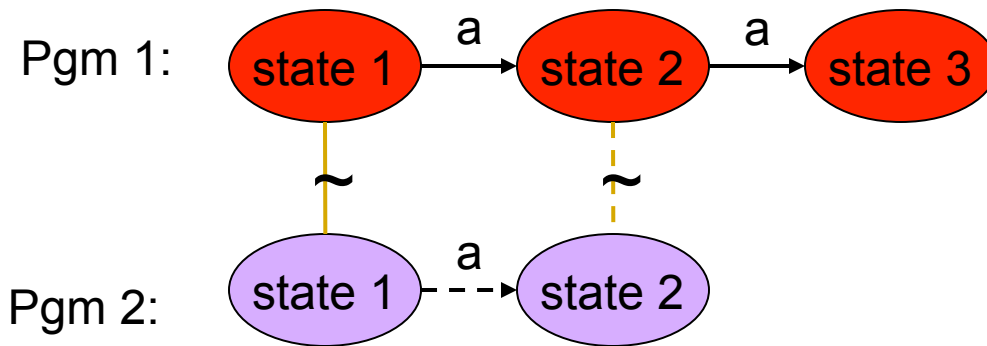
Bisimulations

- Introduced in Concurrent Calculi [Milner], and adapted to functional languages [Abramsky].



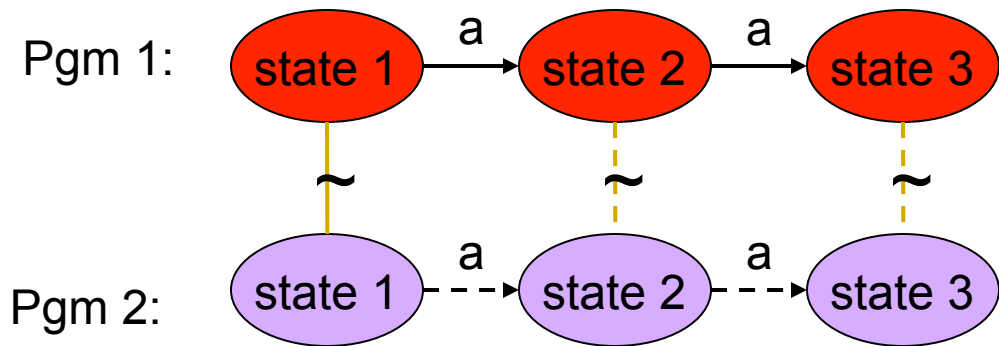
Bisimulations

- Introduced in Concurrent Calculi [Milner], and adapted to functional languages [Abramsky].



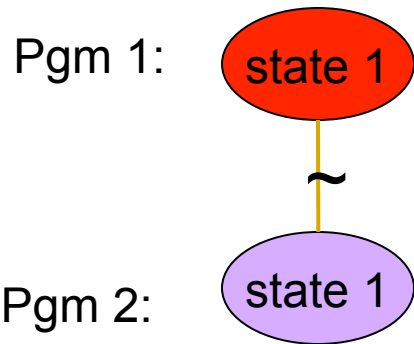
Bisimulations

- Introduced in Concurrent Calculi [Milner], and adapted to functional languages [Abramsky].



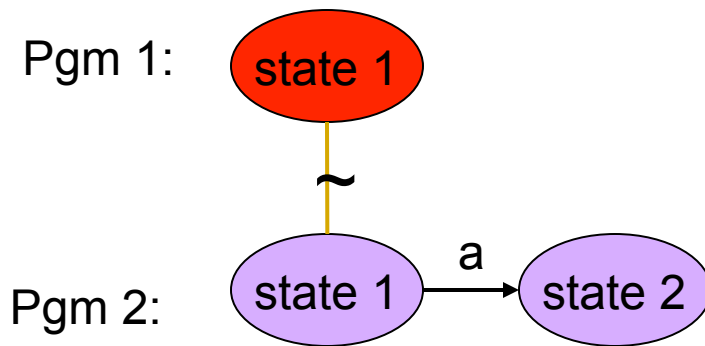
Bisimulations

- Introduced in Concurrent Calculi [Milner], and adapted to functional languages [Abramsky].



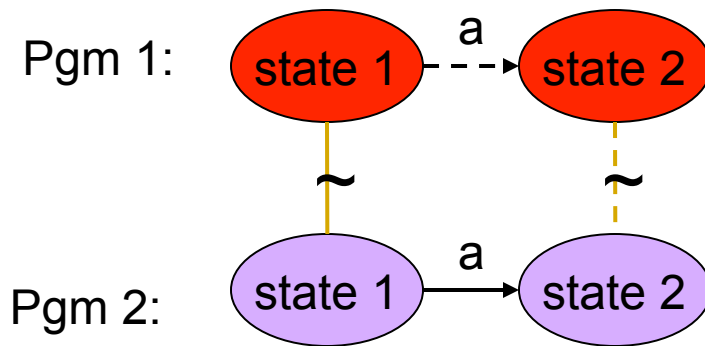
Bisimulations

- Introduced in Concurrent Calculi [Milner], and adapted to functional languages [Abramsky].



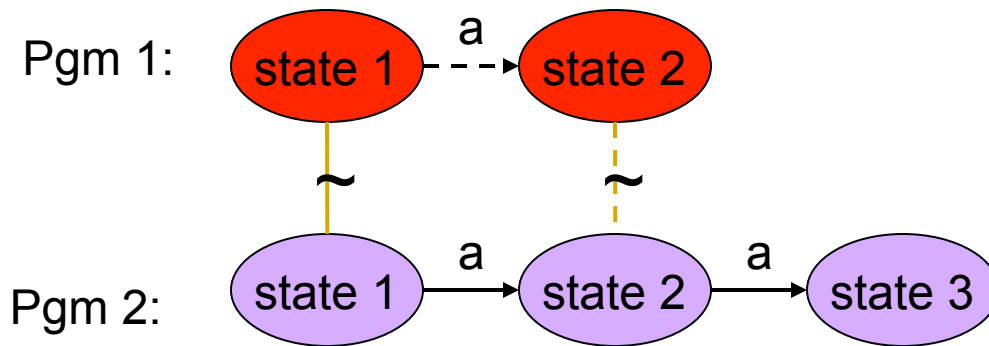
Bisimulations

- Introduced in Concurrent Calculi [Milner], and adapted to functional languages [Abramsky].



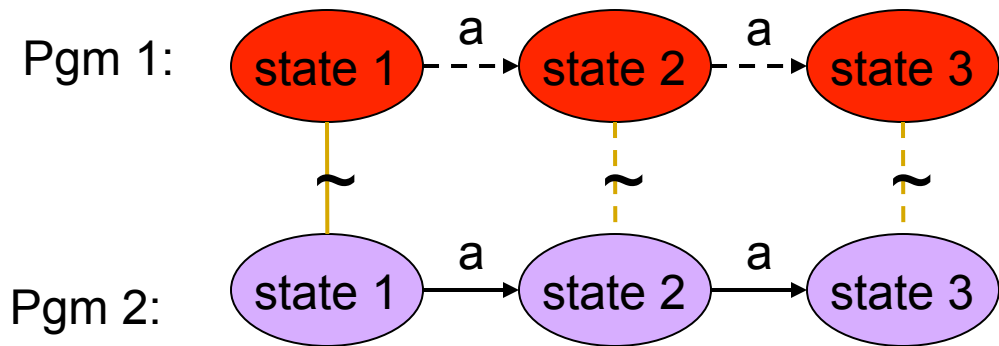
Bisimulations

- Introduced in Concurrent Calculi [Milner], and adapted to functional languages [Abramsky].



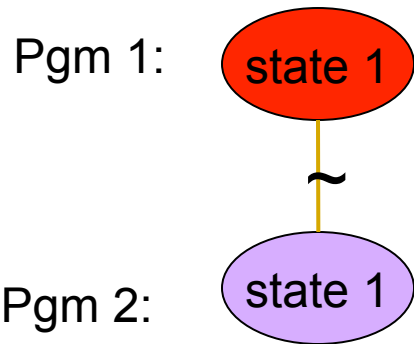
Bisimulations

- Introduced in Concurrent Calculi [Milner], and adapted to functional languages [Abramsky].



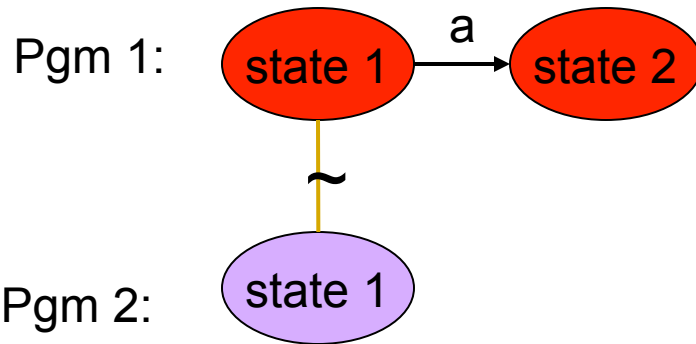
Bisimulations

- Introduced in Concurrent Calculi [Milner], and adapted to functional languages [Abramsky].



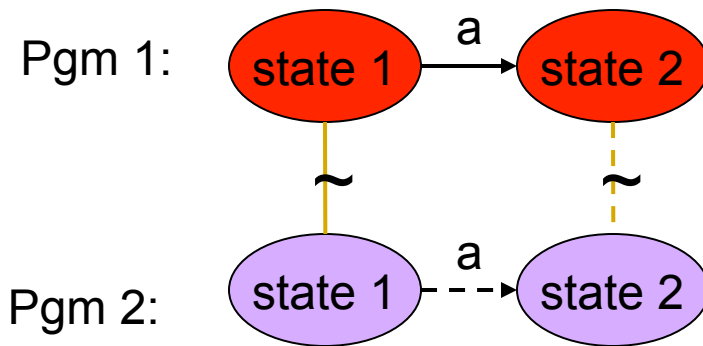
Bisimulations

- Introduced in Concurrent Calculi [Milner], and adapted to functional languages [Abramsky].



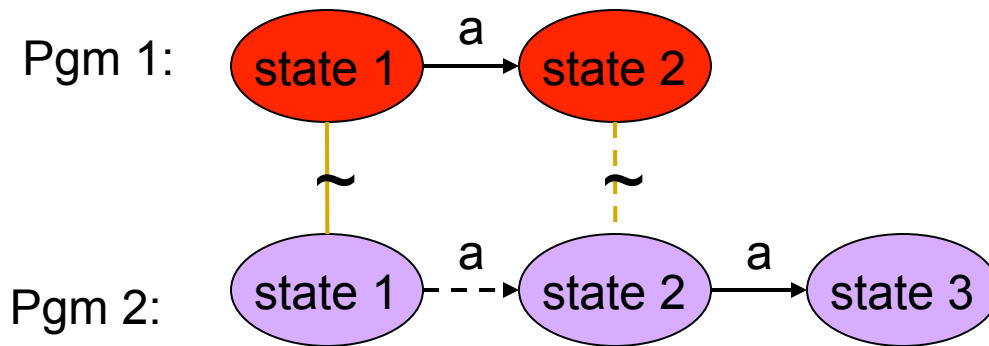
Bisimulations

- Introduced in Concurrent Calculi [Milner], and adapted to functional languages [Abramsky].



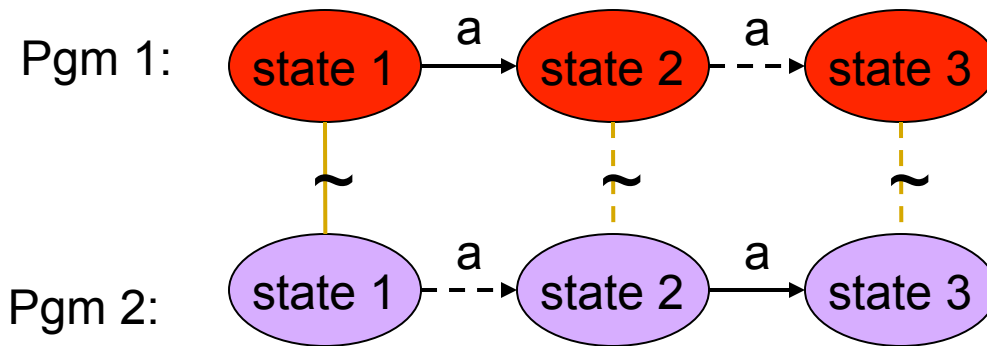
Bisimulations

- Introduced in Concurrent Calculi [Milner], and adapted to functional languages [Abramsky].



Bisimulations

- Introduced in Concurrent Calculi [Milner], and adapted to functional languages [Abramsky].



Deriving Bisimulations for (\equiv)

- $(C, C') \in (\equiv)$ iff:

For any class-table context $CT[]$:

$$(0, CT[C]) \Downarrow \text{ iff } (0, CT[C']) \Downarrow$$

Deriving Bisimulations for (\equiv)

- $(C, C') \in (\equiv)$ iff:

For any class-table context $CT[]$:

$$(0, CT[C]) \Downarrow \text{ iff } (0, CT[C']) \Downarrow$$



Empty
store

Deriving Bisimulations for (\equiv)

- $(\mathbf{C}, \mathbf{C}') \in (\equiv)$ iff:

For any class-table context $\mathbf{CT}[\]$:

$$(\mathbf{0}, \mathbf{CT}[\mathbf{C}]) \Downarrow \text{ iff } (\mathbf{0}, \mathbf{CT}[\mathbf{C}']) \Downarrow$$

Define Adequacy (\approx) as the largest set s.t.:

- $(\mathbf{s}, \mathbf{s}', \mathbf{C}, \mathbf{C}') \in (\approx)$ iff:

For any class-table context $\mathbf{CT}[\]$:

$$\begin{aligned} (\mathbf{s}, \mathbf{CT}[\mathbf{C}]) \Downarrow \mathbf{t}, \mathbf{a} \implies (\mathbf{s}', \mathbf{CT}[\mathbf{C}']) \Downarrow \mathbf{t}', \mathbf{a} \\ \& (\mathbf{t}, \mathbf{t}', \mathbf{C}, \mathbf{C}') \in (\approx) \end{aligned}$$

Deriving Bisimulations for (\equiv)

- $(C, C') \in (\equiv)$ iff:

For any class-table context $CT[]$:

$$(0, CT[C]) \Downarrow \text{ iff } (0, CT[C']) \Downarrow$$

Define Adequacy (\approx) as the large

and the reverse

- $(s, s', C, C') \in (\approx)$ iff:

For any class-table context $CT[]$:

$$(s, CT[C]) \Downarrow t, a \iff (s', CT[C']) \Downarrow t', a$$

$$\& (t, t', C, C') \in (\approx)$$

Deriving Bisimulations for (\equiv)

- $(C, C') \in (\equiv)$ iff:

For any class-table context $CT[]$:

$$(0, CT[C]) \Downarrow \text{ iff } (0, CT[C']) \Downarrow$$

Define Adequacy (\approx) as the largest

- $(s, s', C, C') \in (\approx)$ iff:

For any class-table context $CT[]$:

$$(s, CT[C]) \Downarrow t, a \Rightarrow (s', CT[C']) \Downarrow t', a \\ \& (t, t', C, C') \in (\approx)$$



This is a
big-step
bisimulation

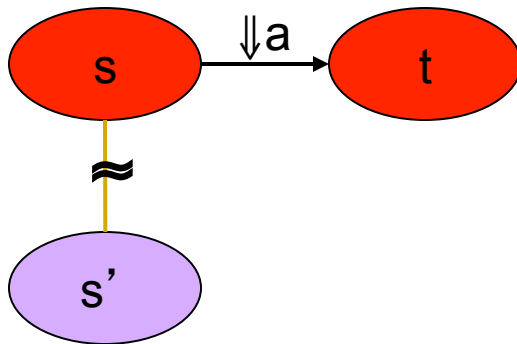
Deriving Bisimulations for (\equiv)

Define Adequacy as the largest set s.t.:

- $(s, s', C, C') \in (\approx)$ iff:

For any class-table context $CT[]$:

$$(s, CT[C]) \Downarrow t, a \Rightarrow (s', CT[C']) \Downarrow t', a$$
$$\& (t, t', C, C') \in (\approx)$$



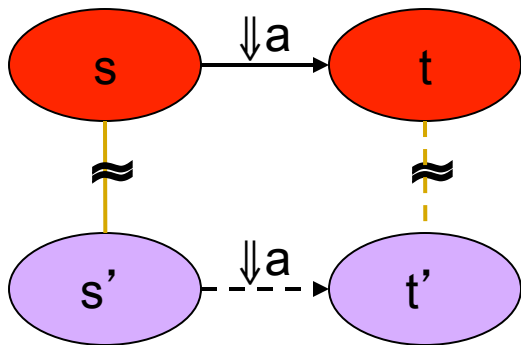
Deriving Bisimulations for (\equiv)

Define Adequacy as the largest set s.t.:

- $(\mathbf{s}, \mathbf{s}', \mathbf{C}, \mathbf{C}') \in (\approx)$ iff:

For any class-table context $\mathbf{CT}[\]$:

$$(\mathbf{s}, \mathbf{CT}[\mathbf{C}]) \Downarrow \mathbf{t}, \mathbf{a} \Rightarrow (\mathbf{s}', \mathbf{CT}[\mathbf{C}']) \Downarrow \mathbf{t}', \mathbf{a} \\ \& (\mathbf{t}, \mathbf{t}', \mathbf{C}, \mathbf{C}') \in (\approx)$$



Deriving Bisimulations for (\equiv)

Define Adequacy as the largest set s.t.:

- $(\mathbf{s}, \mathbf{s}', \mathbf{C}, \mathbf{C}') \in (\approx)$ iff:

For any class-table context $\mathbf{CT}[\]$:

$$(\mathbf{s}, \mathbf{CT}[\mathbf{C}]) \Downarrow \mathbf{t}, \mathbf{a} \Rightarrow (\mathbf{s}', \mathbf{CT}[\mathbf{C}']) \Downarrow \mathbf{t}', \mathbf{a} \\ \& (\mathbf{t}, \mathbf{t}', \mathbf{C}, \mathbf{C}') \in (\approx)$$

- Thm (Soundness & Completeness):

$$(\mathbf{0}, \mathbf{0}, \mathbf{C}, \mathbf{C}') \in (\approx) \text{ iff } (\mathbf{C}, \mathbf{C}') \in (\equiv)$$

Deriving Bisimulations for (\equiv)

To prove C and C' equivalent we must:

- provide a set R
- $(0, 0, C, C') \in R$
- show $R \subseteq (\approx)$

Deriving Bisimulations for (\equiv)

To prove C and C' equivalent we must:

- provide a set R
- $(0, 0, C, C') \in R$
- show $R \subseteq (\approx)$
 - find an **inductive principle**

Examples of Inductive Principles

If $S \subseteq \text{Nat}$, to show **$S = \text{Nat}$** , we must show:

1. $0 \in S$
2. $\forall k. k \in S \Rightarrow k+1 \in S$

Examples of Inductive Principles

If $S \subseteq \text{Nat}$, to show $S = \text{Nat}$, we must show:

1. $0 \in S$
2. $\forall k. k \in S \Rightarrow k+1 \in S$

OR:

1. $0 \in S$
2. $\forall k. (\forall j. j < k \Rightarrow j \in S) \Rightarrow k \in S$

Examples of Inductive Principles

If $S \subseteq \text{Nat}$, to show $S = \text{Nat}$, we

1. $0 \in S$
2. $\forall k. k \in S \Rightarrow k+1 \in S$

The inductive principle is independent of S

OR:

1. $0 \in S$
2. $\forall k. (\forall j. j < k \Rightarrow j \in S) \Rightarrow k \in S$

Examples of Inductive Principles

If $S \subseteq \text{Nat}$, to show $S = \text{Nat}$, we

1. $0 \in S$
2. $\forall k. k \in S \Rightarrow k+1 \in S$



**Proof
Obligations
of S**

OR:

1. $0 \in S$
2. $\forall k. (\forall j. j < k \Rightarrow j \in S) \Rightarrow k \in S$

Deriving Bisimulations for (\equiv)

To prove C and C' equivalent we must:

- provide a set R
- $(0, 0, C, C') \in R$
- show $R \subseteq (\approx)$
 - find an **inductive principle**

Deriving Bisimulations for (\equiv)

To prove C and C' equivalent we must:

- provide a set R
- $(0, 0, C, C') \in R$
- show $R \subseteq (\approx)$
 - find an **inductive principle**
 - depends on the language, operational semantics, and (\approx)
 - it is **independent of the set R**
 - it requires some **proof obligations on R**

Deriving Bisimulations for (\equiv)

To prove C and C' equivalent we must:

- provide a set R
- $(0, 0, C, C') \in R$
- R satisfies the **Proof Obligations**

Inductive Principle from the def. of (\approx)

$R \subseteq (\approx)$ iff

■ $\forall (s, s', C, C') \in \boxed{R} :$

For any class-table context $CT[] :$

$(s, CT[C]) \Downarrow t, a \Rightarrow (s', CT[C']) \Downarrow t', a$

$\& (t, t', C, C') \in \boxed{R}$

Inductive Principle from the def. of (\approx)

$R \subseteq (\approx)$ iff $\forall k.$

■ $\forall (s, s', C, C') \in R.$

For any class-table context $CT[]:$

$$(s, CT[C]) \stackrel{\leftarrow k}{\Downarrow} t, a \Rightarrow (s', CT[C']) \Downarrow t', a$$
$$\& (t, t', C, C') \in R$$

Inductive Principle from the def. of (\approx)

$R \subseteq (\approx)$ iff $\forall k. \text{IH}(k)$

■ $\forall (s, s', C, C') \in R.$

IH(k)

For any class-table context $\text{CT}[\]$:

$$(s, \text{CT}[C]) \Downarrow^k t, a \Rightarrow (s', \text{CT}[C']) \Downarrow t', a \\ \& (t, t', C, C') \in R$$

Inductive Principle from the def. of (\approx)

$R \subseteq (\approx)$ iff $\forall k. IH(k)$
iff

$\forall k. (\forall j. j < k \Rightarrow IH(j)) \Rightarrow IH(k)$

Inductive Principle from the def. of (\approx)

$R \subseteq (\approx)$ iff $\forall k. IH(k)$
iff

$$\forall k. (\forall j. j < k \Rightarrow IH(j)) \Rightarrow IH(k)$$

- Try the proof for **arbitrary R** (proof construction scheme)
 - find **necessary & sufficient proof obligations on R**
 - turn proof obligations into **Bisimulation Conditions**

Inductive Principle from the def. of (\approx)

$R \subseteq (\approx)$ iff $\forall k. IH(k)$
iff

$\forall k. (\forall j. j < k \Rightarrow IH(j)) \Rightarrow IH(k)$

Great to have in
proofs of
equivalence

- Try the proof for **arbitrary R** (proof construction scheme)
 - find **necessary & sufficient proof obligations on R**
 - turn proof obligations into **Bisimulation Conditions**

Application of Our Technique

- We have derived Proof Obligations for
 - An imperative, class-based language without inheritance (J1)
 - 6 bisimulation conditions: typing&closure conditions on R
 - Context can **read/update public fields**
 - Context can **call public methods with arguments from R^***
 - The extension of J1 with **inheritance** (J2)
 - The context can **override methods in subclasses**
 - The context can **access protected fields/methods**
 - The extension of J2 with **downcasting** (J3)
 - No change in proof obligations

Proving the Equivalence of Cells (J1)

```
class Cell {
  private Object g;
  Cell() { g = null; }
  public void set(Object b) {
    g = b;
  }
  public Object get() {
    return g;
  }
}
```

```
class Cell {
  private Object g1, g2;
  private int cnt;
  Cell() {
    g1 = null; g2 = null; cnt = 0;
  }
  public void set(Object b) {
    cnt = cnt + 1;
    if ((cnt % 2) == 0)
      g1 = b;
    else
      g2 = b;
  }
  public Object get() {
    if ((cnt % 2) == 0)
      return g1;
    else
      return g2;
  }
}
```

Proving the Equivalence of Cells (J1)

Let $R = \{ (s, s', \text{Cell}, \text{Cell}') \mid$
 $s = [l = \text{obj Cell}\{g = \mathbf{l_1}\}] \cdot s_0$
 $(s' = [l = \text{obj Cell}\{g_1 = \mathbf{l_1}, g_2 = l_2, \text{cnt} = \mathbf{2n}\}] \cdot s_0')$
OR
 $s' = [l = \text{obj Cell}\{g_1 = l_2, g_2 = \mathbf{l_1}, \text{cnt} = \mathbf{2n+1}\}] \cdot s_0') \}$

Proving the Equivalence of Cells (J1)

$\exists l, l_1, n, \dots$

Let $R = \{ (s, s', \text{Cell}, \text{Cell}') \mid$
 $s = [l = \text{obj Cell}\{g = \mathbf{l_1}\}] \cdot s_0$
 $(s' = [l = \text{obj Cell}\{g_1 = \mathbf{l_1}, g_2 = l_2, \text{cnt} = \mathbf{2n}\}] \cdot s_0'$
OR
 $s' = [l = \text{obj Cell}\{g_1 = l_2, g_2 = \mathbf{l_1}, \text{cnt} = \mathbf{2n+1}\}] \cdot s_0') \}$

Proving the Equivalence of Cells (J1)

Let $R = \{ (s, s', \text{Cell}, \text{Cell}') \mid$
 $s = [l = \text{obj Cell}\{g = \mathbf{l}_1\}] \cdot s_0$
 $(s' = [l = \text{obj Cell}\{g_1 = \mathbf{l}_1, g_2 = l_2, \text{cnt} = \mathbf{2n}\}] \cdot s_0')$
OR
 $s' = [l = \text{obj Cell}\{g_1 = l_2, g_2 = \mathbf{l}_1, \text{cnt} = \mathbf{2n+1}\}] \cdot s_0') \}$

PO 6: Must show that if

$s, \text{CT}[\text{Cell}], l.\text{get}() \Downarrow t, a$

then

$s', \text{CT}[\text{Cell}'], l.\text{get}() \Downarrow t', a$

and $(t, t', \text{Cell}, \text{Cell}') \in R$

Proving the Equivalence of Cells (J1)

Let $R = \{ (s, s', \text{Cell}, \text{Cell}') \mid$
 $s = [l = \text{obj Cell}\{g = \mathbf{l}_1\}] \cdot s_0$
 ($s' = [l = \text{obj Cell}\{g_1 = \mathbf{l}_1, g_2 = l_2, \text{cnt} = \mathbf{2n}\}] \cdot s_0'$
OR
 $s' = [l = \text{obj Cell}\{g_1 = l_2, g_2 = \mathbf{l}_1, \text{cnt} = \mathbf{2n+1}\}] \cdot s_0') \}$

PO 6: Must show that if

$s, \text{CT}[\text{Cell}], l.\text{get}() \Downarrow \mathbf{s}, \mathbf{l}_1$

then

$s', \text{CT}[\text{Cell}'], l.\text{get}() \Downarrow \mathbf{s}', \mathbf{l}_1$

and $(\mathbf{s}, \mathbf{s}', \text{Cell}, \text{Cell}') \in R$

which is **true**.

Proving the Equivalence of Cells (J1)

Let $R = \{ (s, s', \text{Cell}, \text{Cell}') \mid$
 $s = [l = \text{obj Cell}\{g = \mathbf{l}_1\}] \cdot s_0$
 ($s' = [l = \text{obj Cell}\{g_1 = \mathbf{l}_1, g_2 = l_2, \text{cnt} = \mathbf{2n}\}] \cdot s_0'$
OR
 $s' = [l = \text{obj Cell}\{g_1 = l_2, g_2 = \mathbf{l}_1, \text{cnt} = \mathbf{2n+1}\}] \cdot s_0') \}$

PO 6: Must show that **for all l_3**

$s, \text{CT}[\text{Cell}], l.\text{set}(l_3) \Downarrow t, a$

then

$s', \text{CT}[\text{Cell}'], l.\text{set}(l_3) \Downarrow t', a$

and $(t, t', \text{Cell}, \text{Cell}') \in R$

Proving the Equivalence of Cells (J1)

Let $R = \{ (s, s', \text{Cell}, \text{Cell}') \mid$
 $s = [l = \text{obj Cell}\{g = \mathbf{l_1}\}] \cdot s_0$
 $([s' = [l = \text{obj Cell}\{g_1 = \mathbf{l_1}, g_2 = l_2, \text{cnt} = \mathbf{2n}\}] \cdot s_0'$
OR
 $s' = [l = \text{obj Cell}\{g_1 = l_2, g_2 = \mathbf{l_1}, \text{cnt} = \mathbf{2n+1}\}] \cdot s_0') \}$

PO 6: Must show that for all l_3

$s, \text{CT}[\text{Cell}], l.\text{set}(l_3) \Downarrow t, a$

then

$s', \text{CT}[\text{Cell}'], l.\text{set}(l_3) \Downarrow t', a$

and $(t, t', \text{Cell}, \text{Cell}') \in R$

Proving the Equivalence of Cells (J1)

Let $R = \{ (s, s', \text{Cell}, \text{Cell}') \mid$

t $s = [l = \text{obj Cell}\{g = \mathbf{l_1}\}]$

$s' = [l = \text{obj Cell}\{g_1 = \mathbf{l_1}, g_2 = l_2, \text{cnt} = \mathbf{2n}\}]$

OR

t'

$s' = [l = \text{obj Cell}\{g_1 = l_2, g_2 = \mathbf{l_1}, \text{cnt} = \mathbf{2n+1}\}] \cdot s_0'$

PO 6: Must show that for all l_3

$s, \text{CT}[\text{Cell}], l.\text{set}(l_3) \Downarrow t, a$

then

$s', \text{CT}[\text{Cell}'], l.\text{set}(l_3) \Downarrow t', a$

and $(t, t', \text{Cell}, \text{Cell}') \in R$

which is **true**.

Proving the Equivalence of Cells (J1)

Let $R = \{ (s, s', \text{Cell}, \text{Cell}') \mid$
 $s = [l = \text{obj Cell}\{g = \mathbf{l}_1\}] \cdot s_0$
 $(s' = [l = \text{obj Cell}\{g_1 = \mathbf{l}_1, g_2 = l_2, \text{cnt} = \mathbf{2n}\}] \cdot s_0'$
OR
 $s' = [l = \text{obj Cell}\{g_1 = l_2, g_2 = \mathbf{l}_1, \text{cnt} = \mathbf{2n+1}\}] \cdot s_0') \}$

PO 6: Must show that for all l_3

$s, \text{CT}[\text{Cell}], l.\text{set}(l_3) \Downarrow t, a$

then

$s', \text{CT}[\text{Cell}'], l.\text{set}(l_3) \Downarrow t', a$

and $(t, t', \text{Cell}, \text{Cell}') \in R$

Proving the Equivalence of Cells (J1)

Let $R = \{ (s, s', \text{Cell}, \text{Cell}') \mid$

t $s = [l = \text{obj Cell}\{g = \mathbf{l_1}\}] \cdot s_0$
 t' $(s' = [l = \text{obj Cell}\{g_1 = \mathbf{l_1}, g_2 = l_2, \text{cnt} = \mathbf{2n}\}] \cdot s_0'$
 OR $s' = [l = \text{obj Cell}\{g_1 = l_2, g_2 = \mathbf{l_1}, \text{cnt} = \mathbf{2n+1}\}] \cdot s_0') \}$

Note: Handwritten annotations include yellow circles around t and t' , and yellow callouts pointing to $\mathbf{l_3}$ in the original image.

PO 6: Must show that for all l_3

$s, \text{CT}[\text{Cell}], l.\text{set}(l_3) \Downarrow t, a$

then

$s', \text{CT}[\text{Cell}'], l.\text{set}(l_3) \Downarrow t', a$

and $(t, t', \text{Cell}, \text{Cell}') \in R$

which is **true**.

Proving the Equivalence of Cells (J1)

Let $R = \{ (s, s', \text{Cell}, \text{Cell}') \mid$
 $s = [l = \text{obj } \text{Cell}\{g = \mathbf{l}_1\}] \cdot s_0$
 $(s' = [l = \text{obj } \text{Cell}\{g_1 = \mathbf{l}_1, g_2 = l_2, \text{cnt} = \mathbf{2n}\}] \cdot s_0'$
OR
 $s' = [l = \text{obj } \text{Cell}\{g_1 = l_2, g_2 = \mathbf{l}_1, \text{cnt} = \mathbf{2n+1}\}] \cdot s_0') \}$

**The rest of the PO are satisfied
by construction of R**

Proving the Equivalence of Cells (J2, J3)

Let $R = \{ (s, s', \text{Cell}, \text{Cell}') \mid$
 $s = [l = \text{obj } \mathbf{C}\{g = \mathbf{l}_1, \dots\}] \cdot s_0$
 $(s' = [l = \text{obj } \mathbf{C}\{g_1 = \mathbf{l}_1, g_2 = \mathbf{l}_2, \text{cnt} = \mathbf{2n}, \dots\}] \cdot s_0'$
OR
 $s' = [l = \text{obj } \mathbf{C}\{g_1 = \mathbf{l}_2, g_2 = \mathbf{l}_1, \text{cnt} = \mathbf{2n+1}, \dots\}] \cdot s_0') \}$

Proving the Equivalence of Cells (J2, J3)

Let $R = \{ (s, s', \text{Cell}, \text{Cell}') \mid$
 $s = [l = \text{obj } \mathbf{C}\{g = \mathbf{l}_1, \dots\}] \cdot s_0$
 $(s' = [l = \text{obj } \mathbf{C}\{g_1 = \mathbf{l}_1, g_2 = \mathbf{l}_2, \text{cnt} = \mathbf{2n}, \dots\}] \cdot s_0')$
OR
 $s' = [l = \text{obj } \mathbf{C}\{g_1 = \mathbf{l}_2, g_2 = \mathbf{l}_1, \text{cnt} = \mathbf{2n+1}, \dots\}] \cdot s_0') \}$

$\mathbf{C} <: \text{Cell}$

Proving the Equivalence of Cells (J2, J3)

More Fields

Let $R = \{ (s, s', \text{Cell}, \text{Cell}') \mid$
 $s = [l = \text{obj } \mathbf{C}\{g = \mathbf{l}_1, \dots\}]$
 $(s' = [l = \text{obj } \mathbf{C}\{g_1 = \mathbf{l}_1, g_2 = \mathbf{l}_2, \text{cnt} = \mathbf{2n}, \dots\}] \cdot s_0$
OR
 $s' = [l = \text{obj } \mathbf{C}\{g_1 = \mathbf{l}_2, g_2 = \mathbf{l}_1, \text{cnt} = \mathbf{2n+1}, \dots\}] \cdot s_0') \}$

Proving the Equivalence of Cells (J2, J3)

More Fields

Let $R = \{ (s, s', \text{Cell}, \text{Cell}') \mid$
 $s = [l = \text{obj } \mathbf{C}\{g = \mathbf{l}_1, \dots\}]$
 $(s' = [l = \text{obj } \mathbf{C}\{g_1 = \mathbf{l}_1, g_2 = \mathbf{l}_2, \text{cnt} = \mathbf{2n}, \dots\}] \cdot s_0$
OR
 $s' = [l = \text{obj } \mathbf{C}\{g_1 = \mathbf{l}_2, g_2 = \mathbf{l}_1, \text{cnt} = \mathbf{2n+1}, \dots\}] \cdot s_0') \}$

The rest of the proof is the same

Application of Our Technique

- We have derived Proof Obligations for
 - An imperative, class-based language without inheritance (J1)
 - 6 bisimulation conditions: typing&closure conditions on R
 - Context can **read/update public fields**
 - Context can **call public methods with arguments from R***
 - The extension of J1 with **inheritance** (J2)
 - The context can **override methods in subclasses**
 - The context can **access protected fields/methods**
 - The extension of J2 with **downcasting** (J3)
 - No change in proof obligations

Application of Our Technique

- We have derived Proof Obligations for
 - An imperative, class-based language without inheritance (J1)
 - 6 bisimulation conditions: typing&closure
 - Context can **read/update public fields**
 - Context can **call public methods**
 - The extension of J1 with **inheritance**
 - The context can **override methods**
 - The context can **access protected fields**
 - The extension of J2 with **dynamic casting** (J3)
 - No change in proof obligations

Are there no extra interactions b/w term-context?
(conservative extension?)

Downcasting breaks equivalences [A. Kennedy]

```
class RO {
  protected int val;
  public RO(int val) { this.val = val; }
  public int get() { return this.val; }
}

class RW extends RO {
  public RW(int val) { super(val); }
  public void set(int val) { this.val = val; }
}

class Callback { public void act(RO arg) { } }
```

```
class C {
  public int foo(Callback f) {
    RW rw = new RW(5);
    f.act(rw);
    return rw.get();
  }
}
```



```
class C {
  public int foo(Callback f) {
    RW rw = new RW(5);
    f.act(rw);
    return 5;
  }
}
```

Downcasting breaks equivalences [A. Kennedy]

```
class RO {
  protected int val;
  public RO(int val) { this.val = val; }
  public int get() { return this.val; }
}

class RW extends RO {
  public RW(int val) { super(val); }
  public void set(int val) { this.val = val; }
}

class Callback { public void act(RO arg) { }
```

```
class C {
  public int foo(Callback f) {
    RW rw = new RW(5);
    f.act(rw);
    return rw.get();
  }
}
```

≠

```
class C {
  public int foo(Callback f) {
    RW rw = new RW(5);
    f.act(rw);
    return 5;
  }
}
```

Proof technique is correct:
we need to consider all callbacks (including those that downcast);
val can have any value after call to f

Downcasting breaks equivalences [A. Kennedy]

```
class RO {  
    protected int val;  
    public RO(int val) { this.val = val; }  
    public int get() { return this.val; }  
}  
  
class RW extends RO {  
    public RW(int val) { super(val); }  
    public void set(int val) { this.val = val; }  
}  
  
class Callback { public void act(RO arg) { }
```

```
class C {  
    public int foo(Callback f) {  
        RW rw = new RW(5);  
        f.act(rw);  
        return rw.get();  
    }  
}
```

≠

```
class C {  
    public int foo(Callback f) {  
        RW rw = new RW(5);  
        f.act(rw);  
        return 5;  
    }  
}
```

Proof technique is correct:
we need to consider all callbacks (including

We adapted the relations to show intuitive change in POs when adding downcasting.

Conclusions

- Bisimulation proof technique of equivalence
 - derived by a “brute-force” but well-structured proof of equivalence
- It works really well in proofs
 - without a detailed model of interactions
- Can help us understand the semantics of the language
 - by looking at example equivalences
 - not by looking at definitions

Thanks!