Regression Verification: Project Proposal

Presentation by Dennis Felsing
within the
Projektgruppe Formale Methoden der Softwareentwicklung

SS 2013

How to prevent regressions in software development?

Introduction

Formal Verification

Formally prove correctness of software
$\Rightarrow$ Requires formal specification

Regression Testing

Discover new bugs by testing for them
$\Rightarrow$ Requires test cases

Introduction

Formal Verification

Formally prove correctness of software
$\Rightarrow$ Requires formal specification

Regression Testing

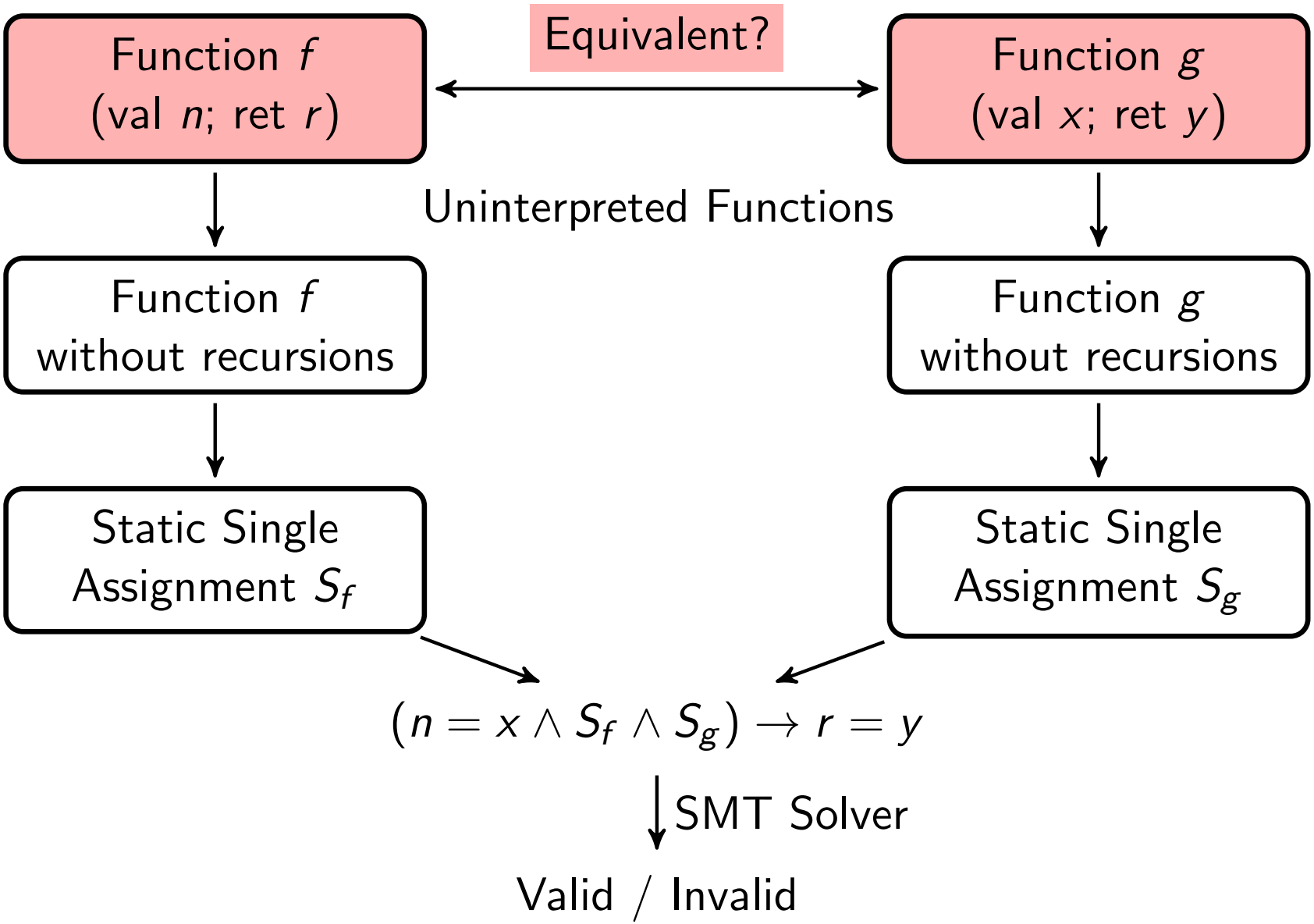
Discover new bugs by testing for them
$\Rightarrow$ Requires test cases

Regression Verification

Formally prove there are no new bugs

Regression Verification

Overview



Regression Verification

Formally prove there are no new bugs

- Goal: Proving the equivalence of two closely related programs
- No formal specification or test cases required
- Instead use old program version
- Make use of similarity between programs

Regression Verification

Formally prove there are no new bugs

- Goal: Proving the equivalence of two closely related programs
- No formal specification or test cases required
- Instead use old program version
- Make use of similarity between programs

```
int f(int n) {
  int r = 0;
  if (n ≤ 0) {
    r = n;
  } else {
    r = n + f(n - 1);
  }
  return r;
}
```

```
int g(int x) {
  int y = 0;
  if (x ≤ 1) {
    y = x;
  } else {
    y = x + g(x - 1);
  }
  return y;
}
```

Regression Verification

Formally prove there are no new bugs

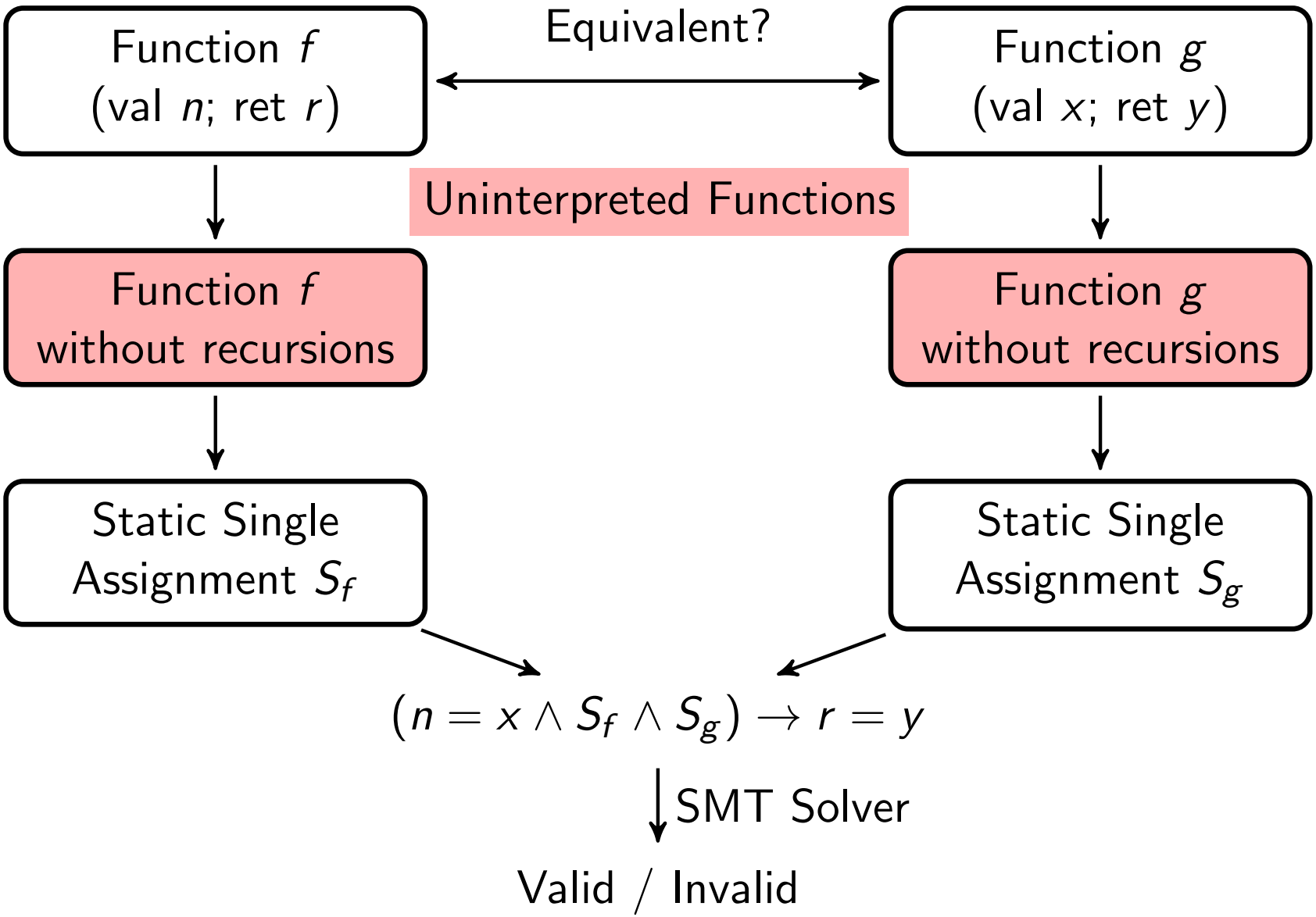
- Goal: Proving the equivalence of two closely related programs
- No formal specification or test cases required
- Instead use old program version
- Make use of similarity between programs

```
int f(int n) {
  int r = 0;
  if (n ≤ 0) {
    r = n;
  } else {
    r = n + f(n - 1);
  }
  return r;
}
```

```
int g(int x) {
  int y = 0;
  if (x ≤ 1) {
    y = x;
  } else {
    y = x + g(x - 1);
  }
  return y;
}
```

Uninterpreted Functions

Overview



Uninterpreted Functions

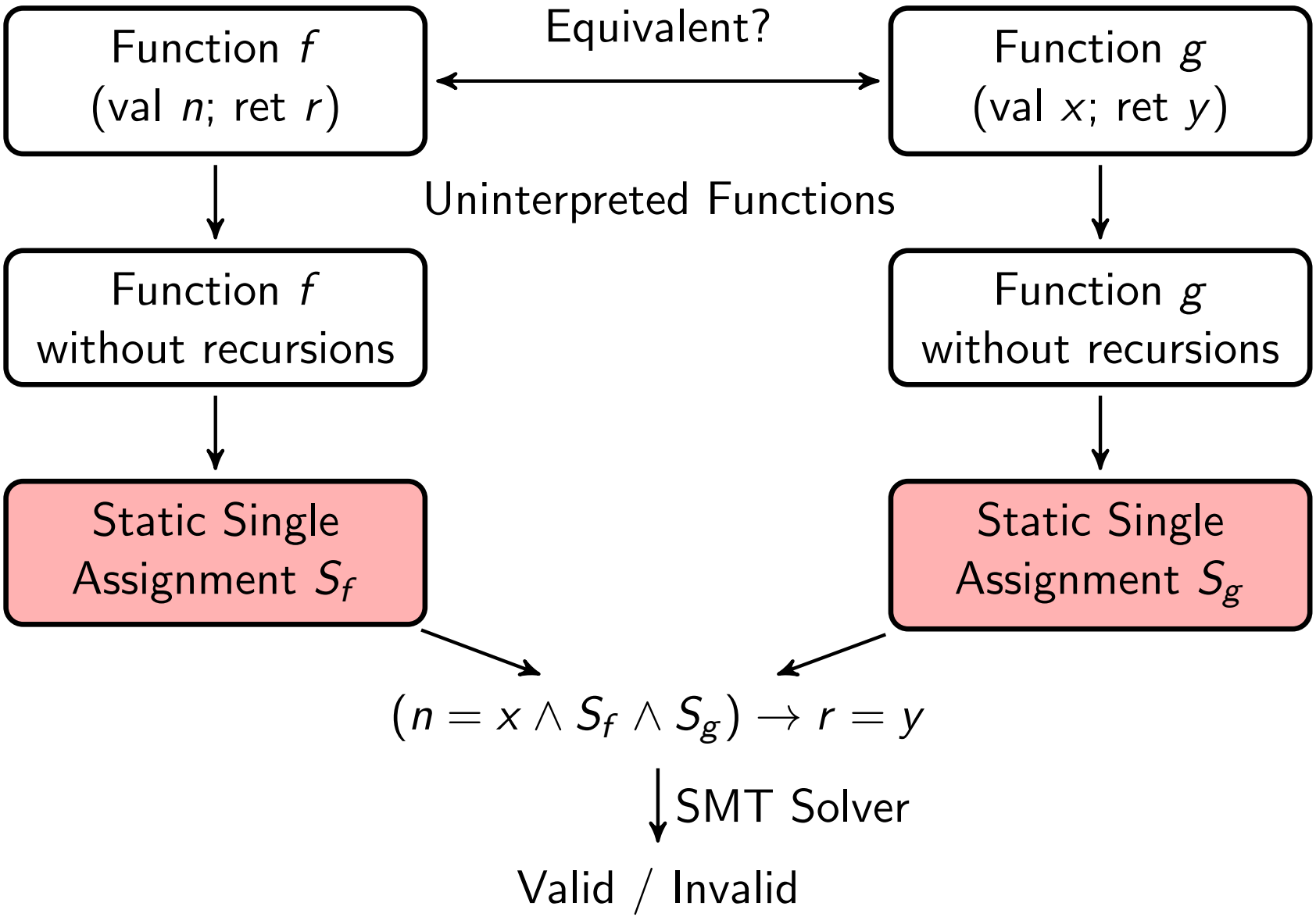
- Given the same inputs an Uninterpreted Function always returns the same outputs.
- Motivation: Proof by Induction, to prove $f(n) = g(n)$ assume $f(n-1) = g(n-1)$

```
int f(int n) {
  int r = 0;
  if (n ≤ 0) {
    r = n;
  } else {
    r = n + U(n - 1);
  }
  return r;
}
```

```
int g(int x) {
  int y = 0;
  if (x ≤ 1) {
    y = x;
  } else {
    y = x + U(x - 1);
  }
  return y;
}
```

Static Single Assignment

Overview



Static Single Assignment

- Translate program functions to formulas
- Recursions: Abstraction by Uninterpreted Function
- In assignments $x = exp$ replace x with a new variable x_1
- Represents the states of the program

Static Single Assignment

- Translate program functions to formulas
- Recursions: Abstraction by Uninterpreted Function
- In assignments $x = exp$ replace x with a new variable x_1
- Represents the states of the program

```
int f(int n) {
  int r = 0;
  if (n ≤ 0) {
    r = n;
  } else {
    r = n + U(n − 1);
  }
  return r;
}
```

$$S_f = \begin{pmatrix} r_0 = 0 \end{pmatrix}$$

Static Single Assignment

- Translate program functions to formulas
- Recursions: Abstraction by Uninterpreted Function
- In assignments $x = exp$ replace x with a new variable x_1
- Represents the states of the program

```
int f(int n) {
  int r = 0;
  if (n ≤ 0) {
    r = n;
  } else {
    r = n + U(n - 1);
  }
  return r;
}
```

$$S_f = \begin{pmatrix} r_0 = 0 \\ n \leq 0 \rightarrow r_1 = n \\ \\ \end{pmatrix} \wedge$$

Static Single Assignment

- Translate program functions to formulas
- Recursions: Abstraction by Uninterpreted Function
- In assignments $x = exp$ replace x with a new variable x_1
- Represents the states of the program

```
int f(int n) {
  int r = 0;
  if (n ≤ 0) {
    r = n;
  } else {
    r = n + U(n - 1);
  }
  return r;
}
```

$$S_f = \begin{pmatrix} r_0 = 0 & \wedge \\ n \leq 0 \rightarrow r_1 = n & \wedge \\ n > 0 \rightarrow r_1 = n + U(n-1) & \end{pmatrix}$$

Static Single Assignment

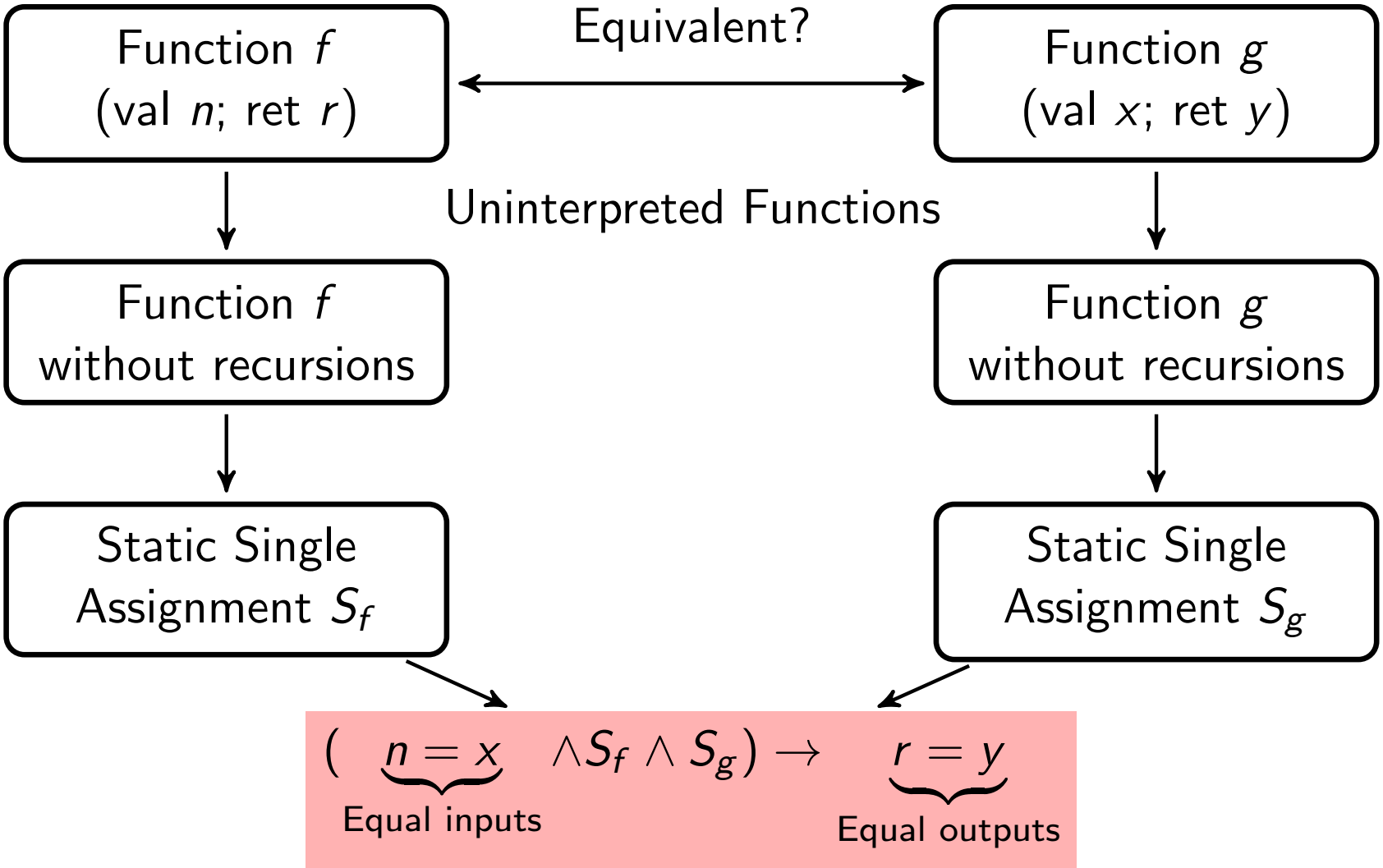
- Translate program functions to formulas
- Recursions: Abstraction by Uninterpreted Function
- In assignments $x = exp$ replace x with a new variable x_1
- Represents the states of the program

```
int f(int n) {
  int r = 0;
  if (n ≤ 0) {
    r = n;
  } else {
    r = n + U(n − 1);
  }
  return r;
}
```

$$S_f = \begin{pmatrix} r_0 = 0 & \wedge \\ n \leq 0 \rightarrow r_1 = n & \wedge \\ n > 0 \rightarrow r_1 = n + U(n-1) & \wedge \\ r = r_1 & \end{pmatrix}$$

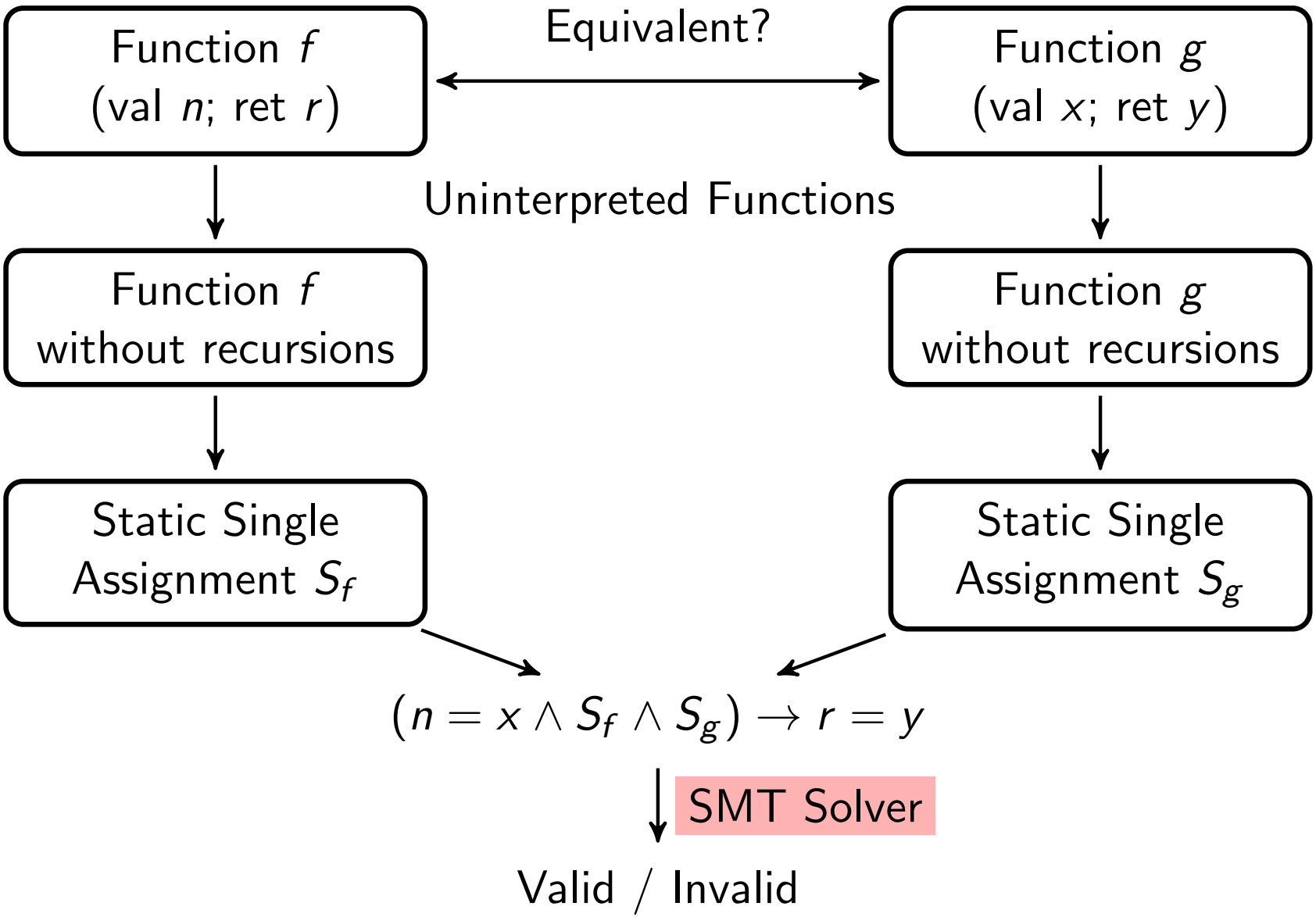
Formula

Overview



SMT Solver

Overview



Extensions

- SMT solver still complains:

$$f(n) = \begin{cases} -1 & \text{if } n = 0 \\ g(n) & \text{otherwise} \end{cases}$$

Extensions

- SMT solver still complains:

$$f(n) = \begin{cases} -1 & \text{if } n = 0 \\ g(n) & \text{otherwise} \end{cases}$$

```
int f(int n) {
  int r = 0;
  if (n ≤ 0) {
    r = n;
  } else {
    r = n + f(n − 1);
  }
  return r;
}
```

```
int g(int x) {
  int y = 0;
  if (x ≤ 1) {
    y = x;
  } else {
    y = x + g(x − 1);
  }
  return y;
}
```

Extensions

- SMT solver still complains:

$$f(n) = \begin{cases} -1 & \text{if } n = 0 \\ g(n) & \text{otherwise} \end{cases}$$

```
int f(int n) {
  int r = 0;
  if (n ≤ 0) {
    r = n;
  } else {
    r = n + f(n − 1);
  }
  return r;
}
```

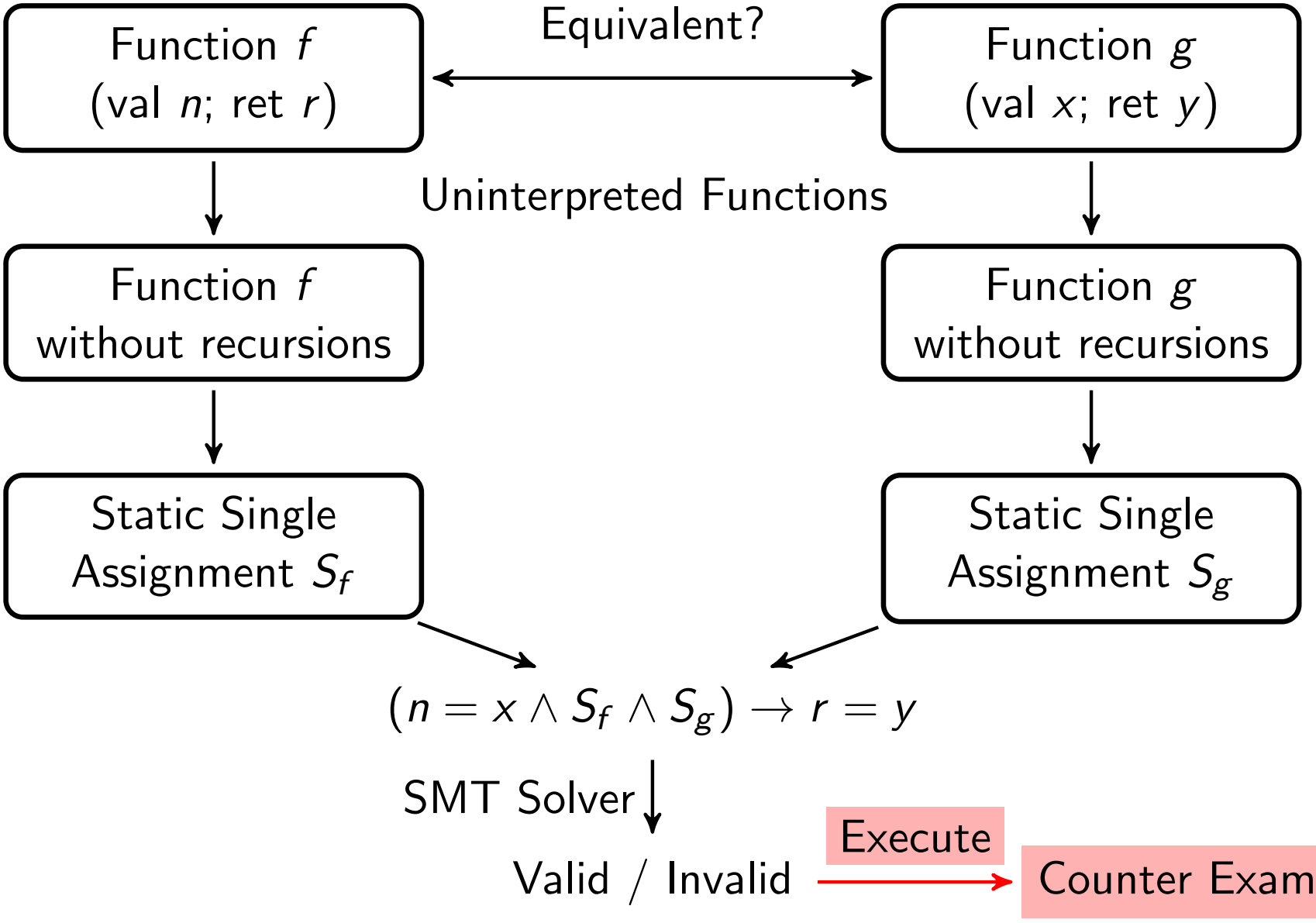
```
int g(int x) {
  int y = 0;
  if (x ≤ 1) {
    y = x;
  } else {
    y = x + g(x − 1);
  }
  return y;
}
```

- But we can fix it:

$$f(0) = 0$$

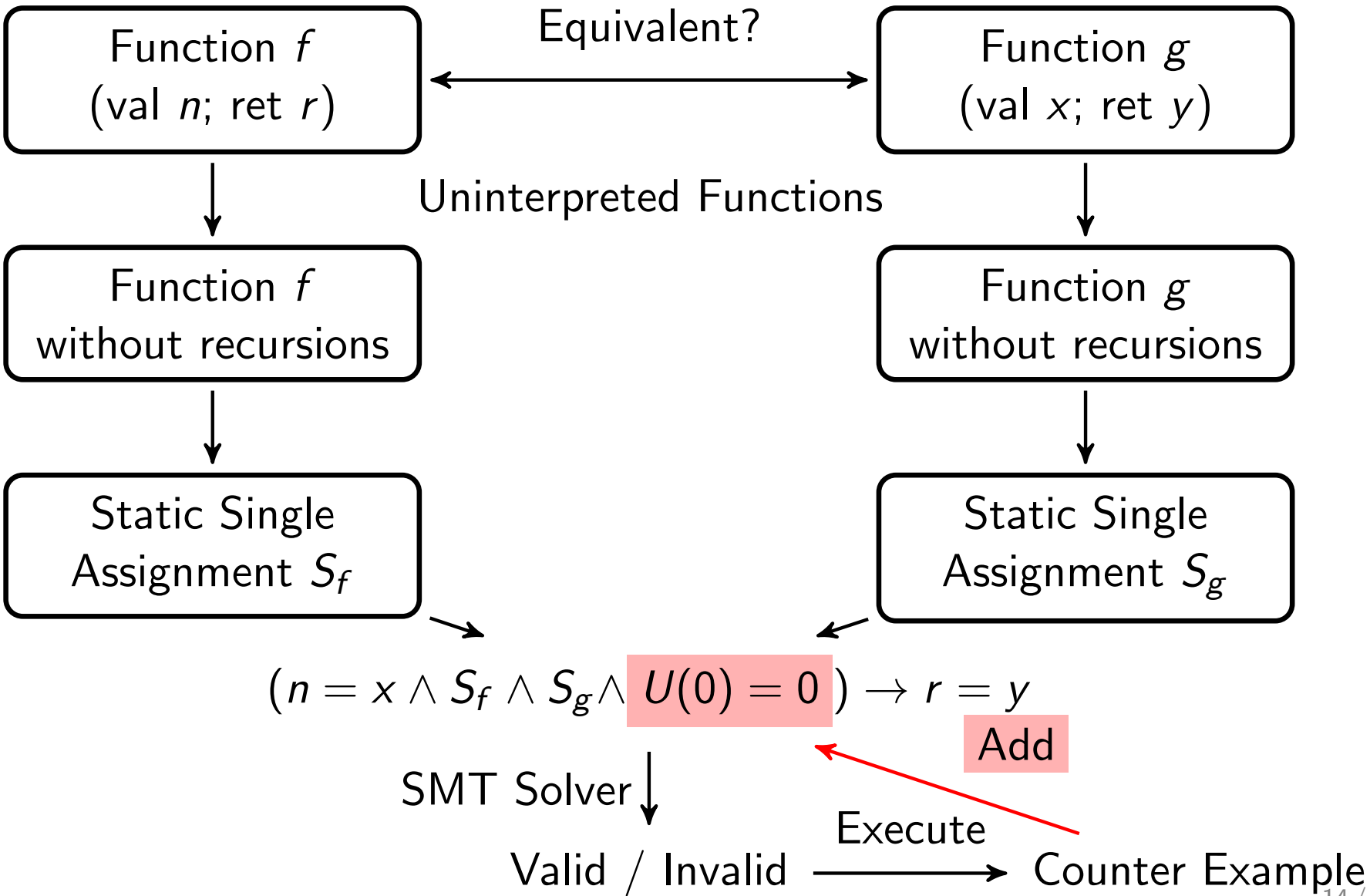
Extensions

Finding Counter Examples



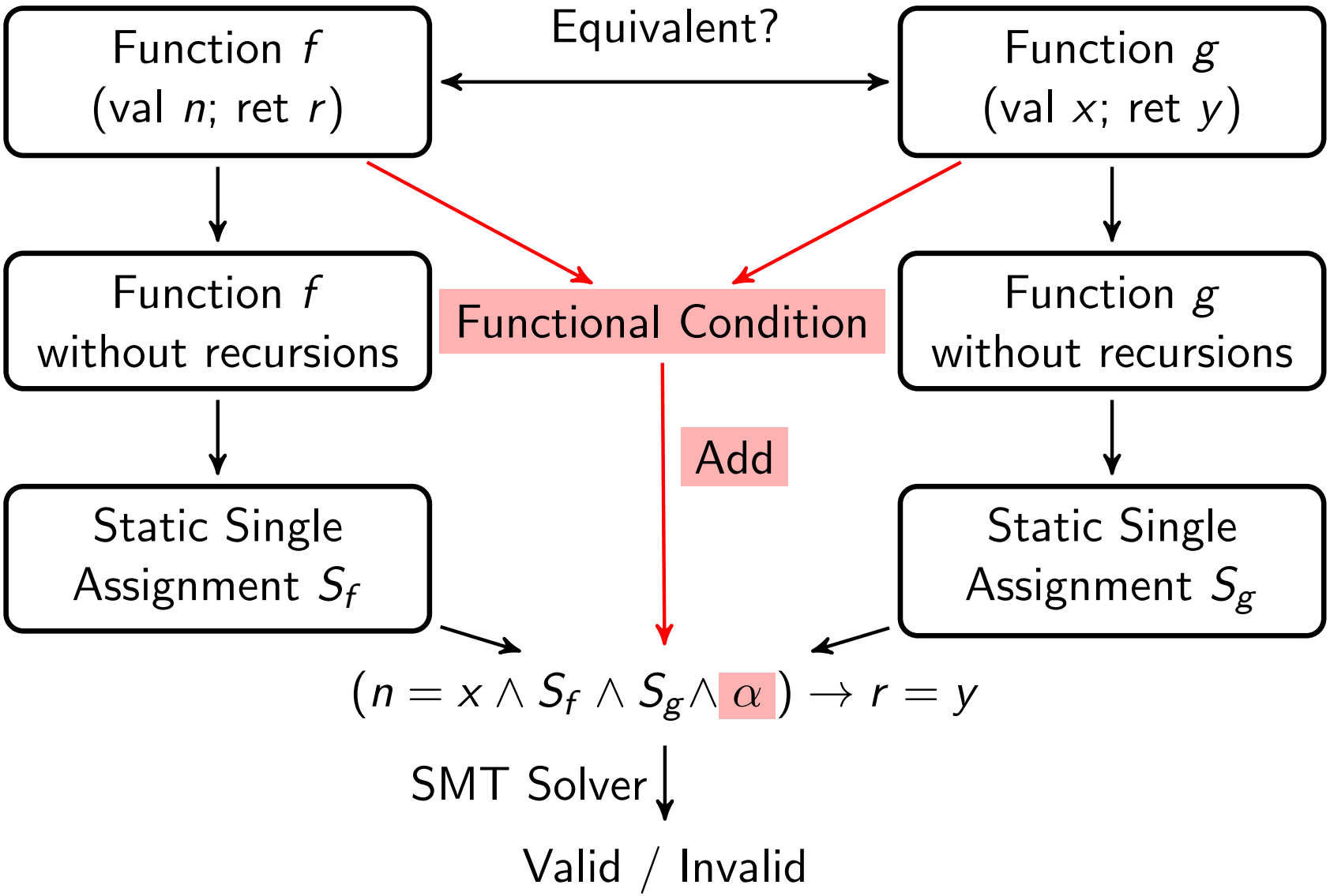
Extensions

Determining Corner Cases



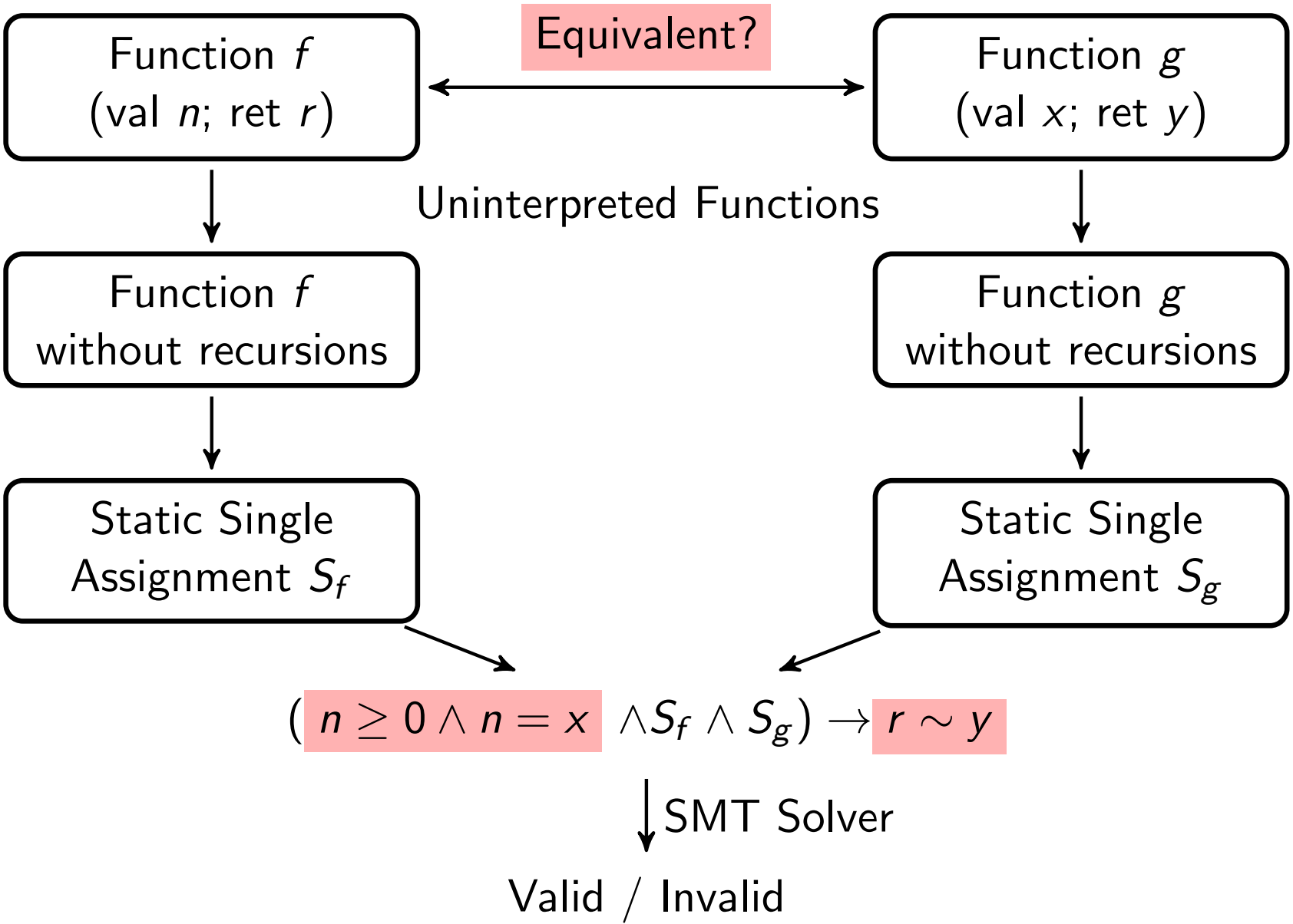
Extensions

Functional Condition Extraction



Extensions

Relational Equivalence



Example Catalog

- Collect examples: Papers, Refactoring Rules, ...
- 51 program pairs so far
- Test how well approach and extensions work

Conclusion

Regression Verification

- Better chance of being adopted than Formal Verification
- More powerful than Regression Testing
- Extensions to cover more cases
- Example Catalog for evaluation