Regression Verification: Proving Partial Equivalence

Talk by Dennis Felsing

Seminar within the

Projektgruppe Formale Methoden der Softwareentwicklung

WS 2012/2013

Karlsruhe Institute of Technology

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

Formally prove there are no new bugs

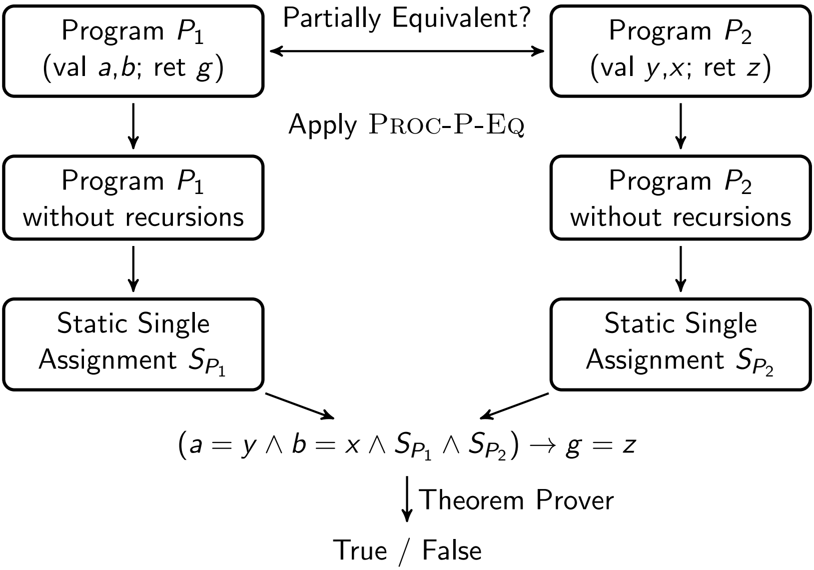
- Goal: Proving the equivalence of two closely related programs
- No formal specification or test cases required
- Instead use old program version
- Make use of similarity between programs

Overview

1. Theoretical Framework
2. Practical Framework
3. Limitations

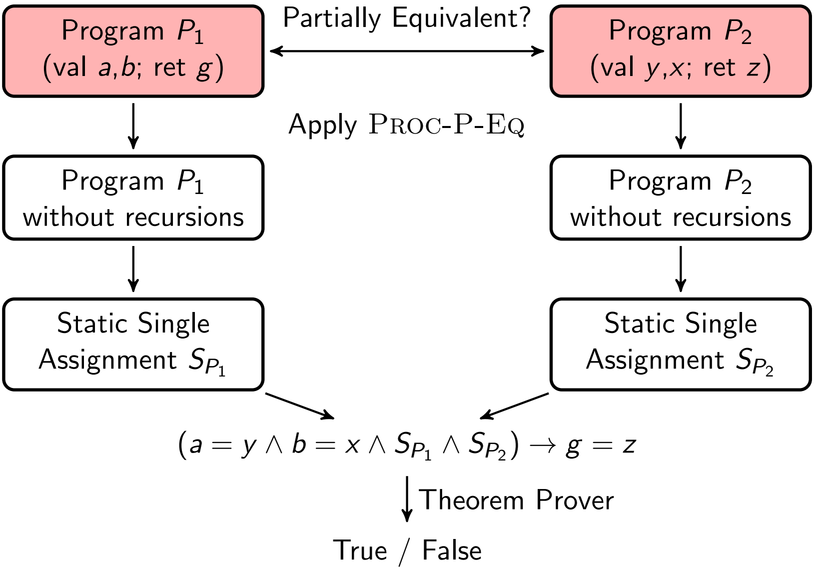
Theoretical Framework

Overview



Linear Procedure Language

Overview



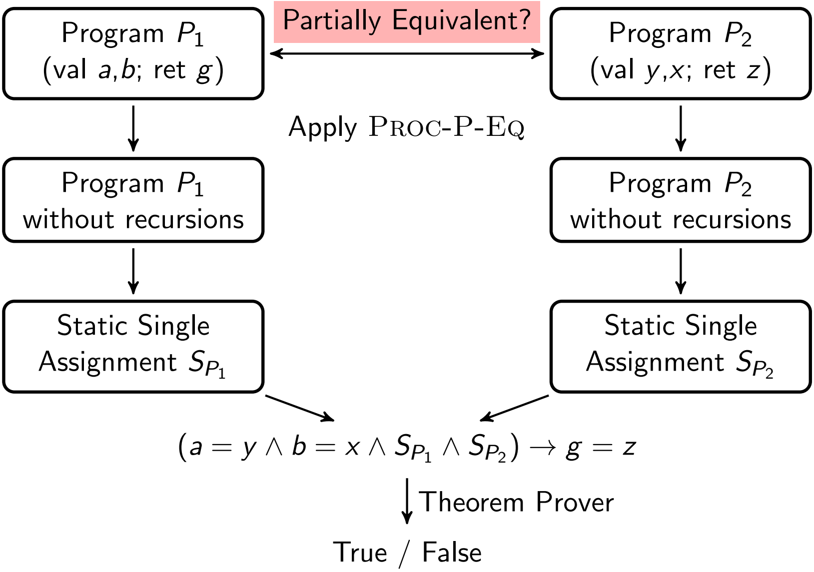
```
procedure gcd3(val x,y,z; ret w):
  call gcd(x,y; a);
  call gcd(a,z; w);
  return

procedure gcd(val a,b; ret g):
  if b = 0 then
    g := a
  else
    a := a%b;
    call gcd(b,a; g)
  fi;
  return
```

$$
\begin{array}{rcl}
\mathit{Program} & :: & \langle \textbf{procedure } \mathrm{p}(\textbf{val } \overline{arg - r_p};\ \textbf{ret } \overline{arg - w_p}) : S_p \rangle_{p \in Proc} \\
S & :: & x := \mathrm{e} \\
 & | & \mathrm{S}\ ;\ \mathrm{S} \\
 & | & \textbf{if } \mathrm{B} \textbf{ then } \mathrm{S} \textbf{ else } \mathrm{S} \textbf{ fi} \\
 & | & \textbf{if } \mathrm{B} \textbf{ then } \mathrm{S} \textbf{ fi} \\
 & | & \textbf{call } p(\overline{e}; \overline{x}) \\
 & | & \textbf{return}
\end{array}
$$

$\Rightarrow$ No loops

Partial Equivalence

Overview



Partial Equivalence

Partial Equivalence: Given the same inputs, any two terminating executions of programs P_1 and P_2 return the same value.

$\Rightarrow$ Partial Equivalence is undecidable

In LPL:

$$\text{part-equiv}(P_1, P_2) = \mathit{in}[P_1] = \mathit{in}[P_2] \rightarrow \mathit{out}[P_1] = \mathit{out}[P_2]$$

Uninterpreted Procedures

Given the same inputs an Uninterpreted Procedure always produces the same outputs.

In LPL:

```
procedure U(val r1, r2, ...; ret w1, w2, ...):
  return
```

Mappings

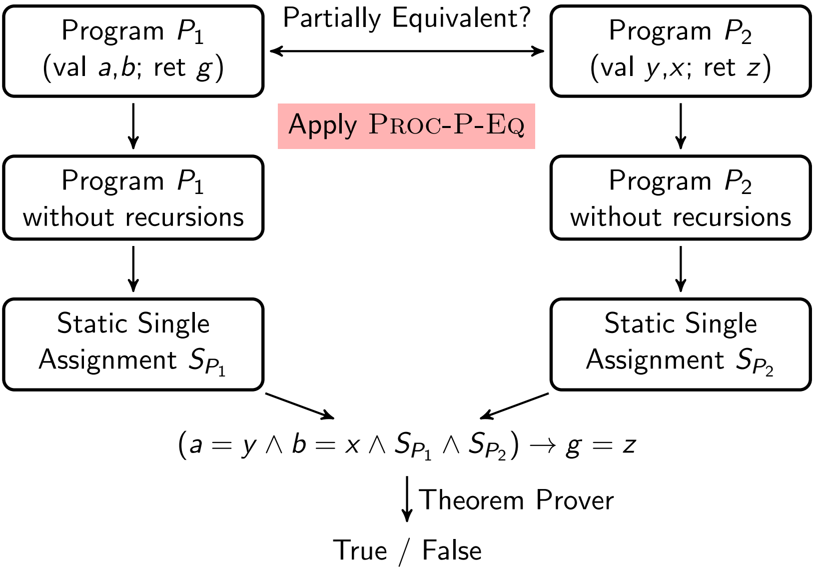
Programs P_1 and P_2 consist of procedures
Map equivalent procedures onto each other

In LPL:
$map : Proc[P_1] \mapsto Proc[P_2]$

UP maps procedures to their respective uninterpreted procedures:

$$\langle F, G\rangle \in map \Longleftrightarrow UP(F) = UP(G)$$

Rule for Proving Partial Equivalence



Example

$$\frac{\text{part-equiv(gcd1, gcd2)} \vdash \text{part-equiv}(\mathit{gcd}1\ \mathbf{body}, \mathit{gcd}2\ \mathbf{body})}{\text{part-equiv}(\mathit{gcd}1, \mathit{gcd}2)}$$

```
procedure gcd1
(val a,b; ret g):
  if b = 0 then
    g := a
  else
    a := a%b;
    call gcd1 (b, a; g)
  fi;
  return
```

```
procedure gcd2
(val x,y; ret z):

  z := x;

  if y > 0 then
    call gcd2 (y, z%y; z)
  fi;
  return
```

Example

$$\frac{\text{part-equiv(gcd1, gcd2)} \vdash \text{part-equiv}(\mathit{gcd1}\ \textbf{body}, \mathit{gcd2}\ \textbf{body})}{\text{part-equiv}(\mathit{gcd1}, \mathit{gcd2})}$$

```
procedure gcd1
(val a,b; ret g):
  if b = 0 then
    g := a
  else
    a := a%b;
    call gcd1(b,a; g)
  fi;
  return
```

```
procedure gcd2
(val x,y; ret z):

  z := x;

  if y > 0 then
    call gcd2(y,z%y; z)
  fi;
  return
```

Example

$$\frac{\vdash_{\mathbb{L}_{\mathbb{UP}}} \text{part-equiv}(gcd1\ \boxed{[gcd1 \leftarrow UP(gcd1)]}\ , gcd2\ \boxed{[gcd2 \leftarrow UP(gcd2)]}\ )}{\text{part-equiv}(gcd1, gcd2)}$$

```
procedure gcd1
(val a,b; ret g):
  if b = 0 then
    g := a
  else
    a := a%b;
    call gcd1(b,a; g)
  fi;
  return
```

```
procedure gcd2
(val x,y; ret z):

  z := x;

  if y > 0 then
    call gcd2(y,z%y; z)
  fi;
  return
```

Example

$$\frac{\vdash_{\mathbb{L}_{\mathbb{UP}}} \text{part-equiv}(gcd1\ \boxed{[gcd1 \leftarrow UP(gcd1)]}\ , gcd2\ \boxed{[gcd2 \leftarrow UP(gcd2)]}\ )}{\text{part-equiv}(gcd1, gcd2)}$$

```
procedure gcd1
(val a,b; ret g):
  if b = 0 then
    g := a
  else
    a := a%b;
    call U (b,a; g)
  fi;
  return
```

```
procedure gcd2
(val x,y; ret z):

  z := x;

  if y > 0 then
    call U (y,z%y; z)
  fi;
  return
```

Rule Proc-P-Eq

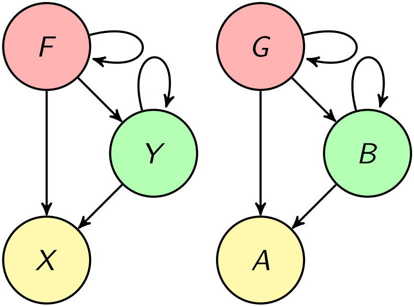
$$\frac{\forall \langle F, G \rangle \in map.\ \{\vdash_{\mathbb{L}_{\mathbb{UP}}} \text{part-equiv}(F^{UP}, G^{UP})\}}{\forall \langle F, G \rangle \in map.\ \text{part-equiv}(F, G)}$$

- $\mathbb{L}_{\mathbb{UP}}$ is a sound proof system for a non-recursive LPL
- $F^{UP} = F[f \leftarrow UP(f) \mid f \in Proc[P]]$ is an isolated procedure

Rule PROC-P-EQ

$$\frac{\forall \langle F, G \rangle \in \mathit{map}.\ \{\vdash_{\mathbb{L}_{\mathbb{UP}}} \text{part-equiv}(F^{UP}, G^{UP})\}}{\forall \langle F, G \rangle \in \mathit{map}.\ \text{part-equiv}(F, G)}$$

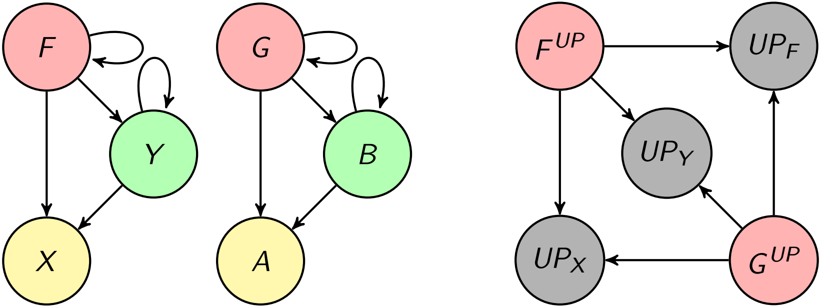
- $\mathbb{L}_{\mathbb{UP}}$ is a sound proof system for a non-recursive LPL
- $F^{UP} = F[f \leftarrow UP(f) \mid f \in Proc[P]]$ is an isolated procedure



Rule Proc-P-Eq

$$\frac{\forall \langle F, G\rangle \in map.\ \{\vdash_{\mathbb{L}_{\mathbb{UP}}} \text{part-equiv}(F^{UP}, G^{UP})\}}{\forall \langle F, G\rangle \in map.\ \text{part-equiv}(F, G)}$$

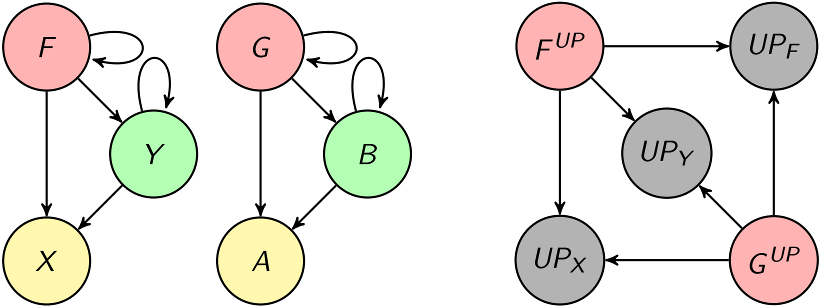
- $\mathbb{L}_{\mathbb{UP}}$ is a sound proof system for a non-recursive LPL
- $F^{UP} = F[f \leftarrow UP(f) \mid f \in Proc[P]]$ is an isolated procedure



Rule PROC-P-EQ

$$\frac{\forall \langle F, G\rangle \in map.\ \{\vdash_{\mathbb{L}_{\mathbb{UP}}} \text{part-equiv}(F^{UP}, G^{UP})\}}{\forall \langle F, G\rangle \in map.\ \text{part-equiv}(F, G)}$$

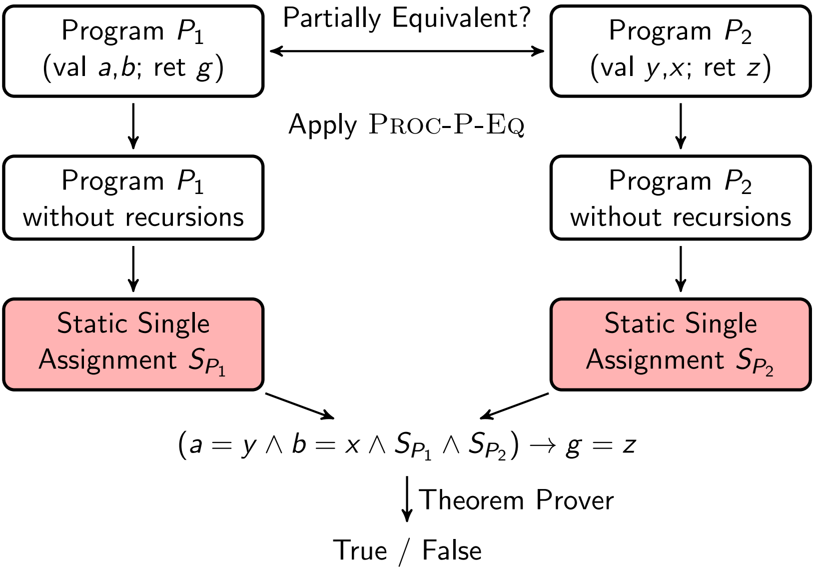
- $\mathbb{L}_{\mathbb{UP}}$ is a sound proof system for a non-recursive LPL
- $F^{UP} = F[f \leftarrow UP(f) \mid f \in Proc[P]]$ is an isolated procedure



$\Rightarrow$ PROC-P-EQ is sound, not complete

Static Single Assignment

Overview



Static Single Assignment

- Translate procedures to formulas
- No loops or recursions
- In assignments $x := exp$ replace x with a new variable x_1
- Represents the states of the program

Static Single Assignment

- Translate procedures to formulas
- No loops or recursions
- In assignments $x := exp$ replace x with a new variable x_1
- Represents the states of the program

Example

```
procedure gcd2
(val x,y; ret z):
  z := x;
  if y > 0 then
    call U(y,z%y; z)
  fi;
  return
```

$$S_{gcd_2} = \left(\begin{array}{ll} x_0 = x & \wedge \\ y_0 = y & \\ & \\ & \\ & \\ & \end{array} \right.$$

Static Single Assignment

- Translate procedures to formulas
- No loops or recursions
- In assignments $x := exp$ replace x with a new variable x_1
- Represents the states of the program

Example

```
procedure gcd2
(val x,y; ret z):
  z := x;
  if y > 0 then
    call U(y,z%y; z)
  fi;
  return
```

$$S_{gcd_2} = \begin{cases} x_0 = x & \wedge \\ y_0 = y & \wedge \\ z_0 = x_0 & \\ & \\ & \end{cases}$$

Static Single Assignment

- Translate procedures to formulas
- No loops or recursions
- In assignments $x := exp$ replace x with a new variable x_1
- Represents the states of the program

Example

```
procedure gcd2
(val x,y; ret z):
  z := x;
  if y > 0 then
    call U(y, z%y; z)
  fi;
  return
```

$$S_{gcd_2} = \begin{cases} x_0 = x & \wedge \\ y_0 = y & \wedge \\ z_0 = x_0 & \wedge \\ y_0 > 0 \rightarrow z_1 = U(y_0, (z_0 \% y_0)) & \\ & \\ & \end{cases}$$

Static Single Assignment

- Translate procedures to formulas
- No loops or recursions
- In assignments $x := exp$ replace x with a new variable x_1
- Represents the states of the program

Example

```
procedure gcd2
(val x,y; ret z):
  z := x;
  if y > 0 then
     call U(y,z%y; z)
  fi;
  return
```

$$S_{gcd_2} = \begin{cases} x_0 = x & \wedge \\ y_0 = y & \wedge \\ z_0 = x_0 & \wedge \\ y_0 > 0 \rightarrow z_1 = U(y_0, (z_0 \% y_0)) & \wedge \\ y_0 \leq 0 \rightarrow z_1 = z_0 & \end{cases}$$

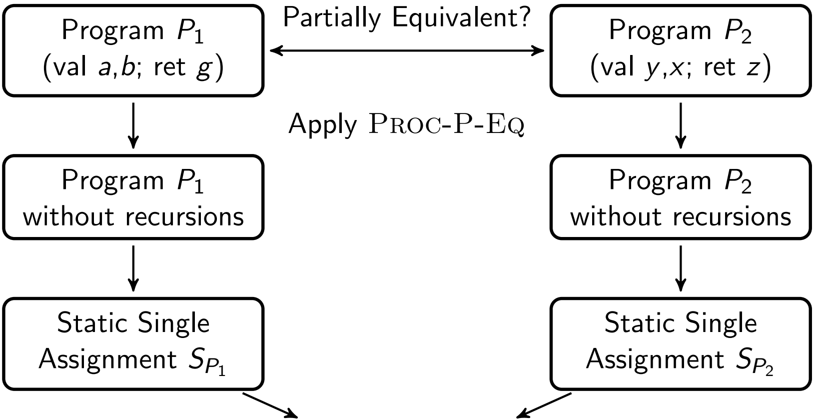
Static Single Assignment

- Translate procedures to formulas
- No loops or recursions
- In assignments $x := exp$ replace x with a new variable x_1
- Represents the states of the program

Example

```
procedure gcd2
(val x,y; ret z):
  z := x;
  if y > 0 then
    call U(y,z%y; z)
  fi;
  return
```

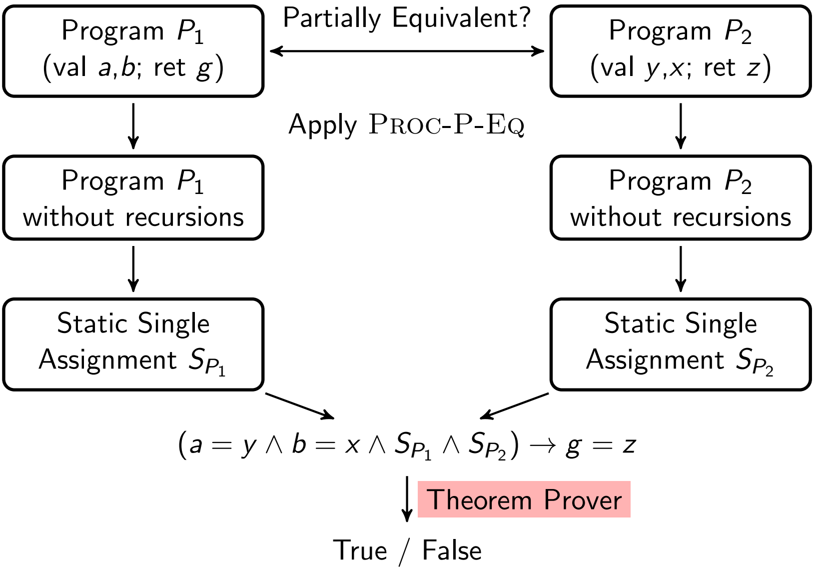
$$S_{gcd_2} = \begin{pmatrix} x_0 = x & \wedge \\ y_0 = y & \wedge \\ z_0 = x_0 & \wedge \\ y_0 > 0 \to z_1 = U(y_0, (z_0 \% y_0)) & \wedge \\ y_0 \le 0 \to z_1 = z_0 & \wedge \\ z = z_1 & \end{pmatrix}$$



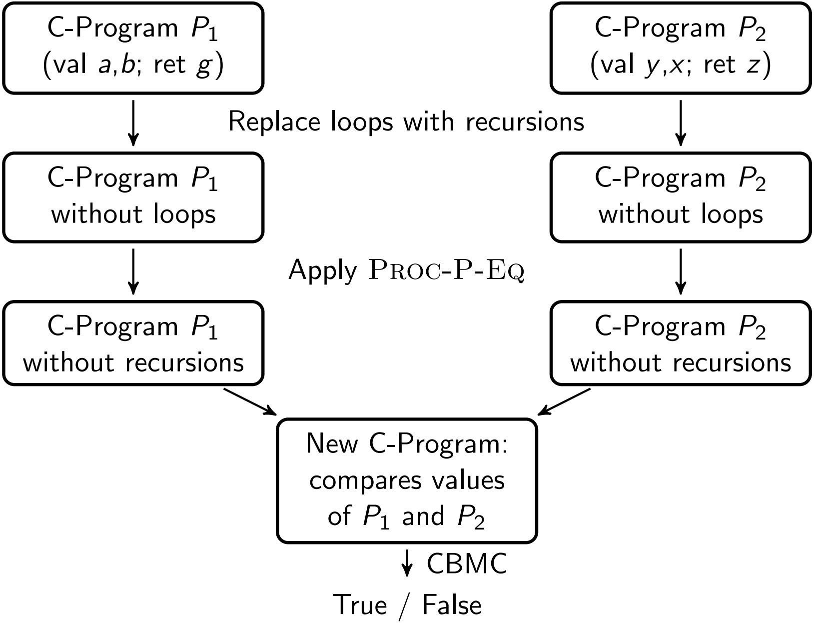
$$\big(\underbrace{a = y \land b = x}_{\text{Equal inputs}} \land S_{P_1} \land S_{P_2}\big) \rightarrow \underbrace{g = z}_{\text{Equal outputs}}$$

Static Single Assignment

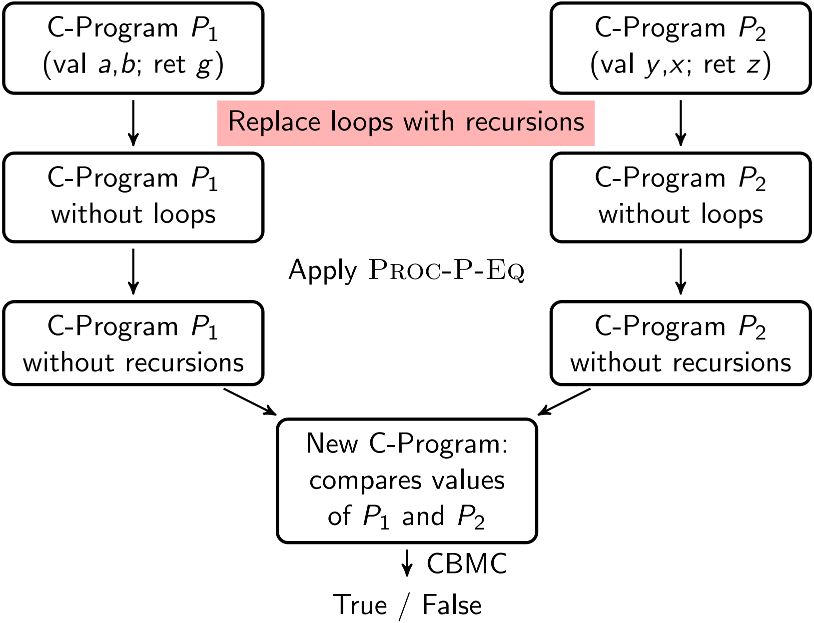
Overview



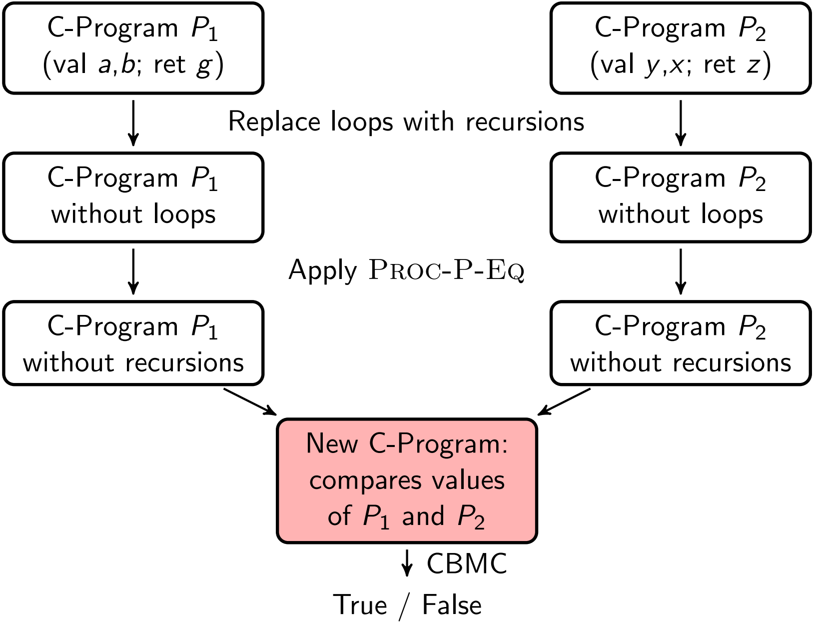
Practice


C-Program P_1 (val a,b; ret g)
C-Program P_2 (val y,x; ret z)
Replace loops with recursions
C-Program P_1 without loops
C-Program P_2 without loops
Apply Proc-P-Eq
C-Program P_1 without recursions
C-Program P_2 without recursions
New C-Program: compares values of P_1 and P_2
CBMC
True / False

Practice


C-Program P_1 (val a,b; ret g)
C-Program P_2 (val y,x; ret z)
Replace loops with recursions
C-Program P_1 without loops
C-Program P_2 without loops
Apply Proc-P-Eq
C-Program P_1 without recursions
C-Program P_2 without recursions
New C-Program: compares values of P_1 and P_2
CBMC
True / False

Practice



C-Program P_1 (val a,b; ret g)
C-Program P_2 (val y,x; ret z)
Replace loops with recursions
C-Program P_1 without loops
C-Program P_2 without loops
Apply Proc-P-Eq
C-Program P_1 without recursions
C-Program P_2 without recursions
New C-Program: compares values of P_1 and P_2
CBMC
True / False

Practice

C-Program P_1 (val a,b; ret g)

C-Program P_2 (val y,x; ret z)

Replace loops with recursions

C-Program P_1 without loops

C-Program P_2 without loops

Apply Proc-P-Eq

C-Program P_1 without recursions

C-Program P_2 without recursions

New C-Program: compares values of P_1 and P_2

CBMC

True / False

Demo

PROC-P-EQ cannot prove recursions where

- procedures are called with different arguments:

```
procedure F
(val n; ret r):
  if n ≤ 1 then
    r := n
  else
    call F(n-1; r);
    r := n + r
  fi
  return
```

```
procedure G
(val n; ret r):
  if n ≤ 1 then
    r := n
  else
    call G(n-2; r);
    r := n+(n−1)+r
  fi
  return
```

- the procedure body is not equivalent
- the Uninterpreted Procedure is too weak

PROC-P-EQ cannot prove recursions where

- procedures are called with different arguments
- the procedure body is not equivalent:

```
procedure F
(val n; ret r):
  if n ≤ 0 then
    r := n
  else
    call F(n−1; r);
    r := n + r
  fi
  return
```

```
procedure G
(val n; ret r):
  if n ≤ 1 then
    r := n
  else
    call G(n−1; r);
    r := n + r
  fi
  return
```

- the Uninterpreted Procedure is too weak

PROC-P-EQ cannot prove recursions where

- procedures are called with different arguments
- the procedure body is not equivalent
- the Uninterpreted Procedure is too weak:

```
procedure F
(val n; ret r):
  if n ≤ 0 then
    r := 0
  else
    call F(n−1; r);
    r := n + r
  fi
  return
```

```
procedure G
(val n; ret r):
  if n ≤ 0 then
    r := 0
  else
    call G(n−1; r);
    if r ≥ 0 then r := n+r
    fi
  fi
  return
```

Limitations

Regression Verification Tool

- Condition of equality cannot be specified
- Counterexample not quickly found because of function inlining
- Mapping only by function names and locations

Conclusion

Regression Verification

- Better chance of being adopted than Functional Verification
- More powerful than Regression Testing
- Simple rule Proc-P-Eq for many cases, but not all
- Regression Verification has recently been extended to multi-threaded programs