Regression Verification: Status Report

Presentation by Dennis Felsing
within the
Projektgruppe Formale Methoden der Softwareentwicklung

2013-12-11

Karlsruhe Institute of Technology

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

Project Objectives

1. Develop a tool for Regression Verification for recursive programs in a simple imperative programming language
2. Case study to evaluate how well our approaches work for different examples in comparison to other systems
3. Extend the tool to work with more programs and to be more general

Preliminary Considerations I

Unbounded Integers vs Bit Vectors

- Unbounded Integers don't overflow
- Bit Vectors can be limited to simplify the problem
- **Solution:** Support both:
  - Proofs are supposed to be over unbounded Integers
  - For comparison Bit Vectors can also be used

Preliminary Considerations II

Division by 0

In Z3, division by zero is allowed, but the result is not specified. Division is not a partial function. Actually, in Z3 all functions are total, although the result may be underspecified in some cases like division by zero.

- **Possible Solutions:**
  - Check that there are no divisions by 0
  - It could be verified that the result is independent of the result of division by 0

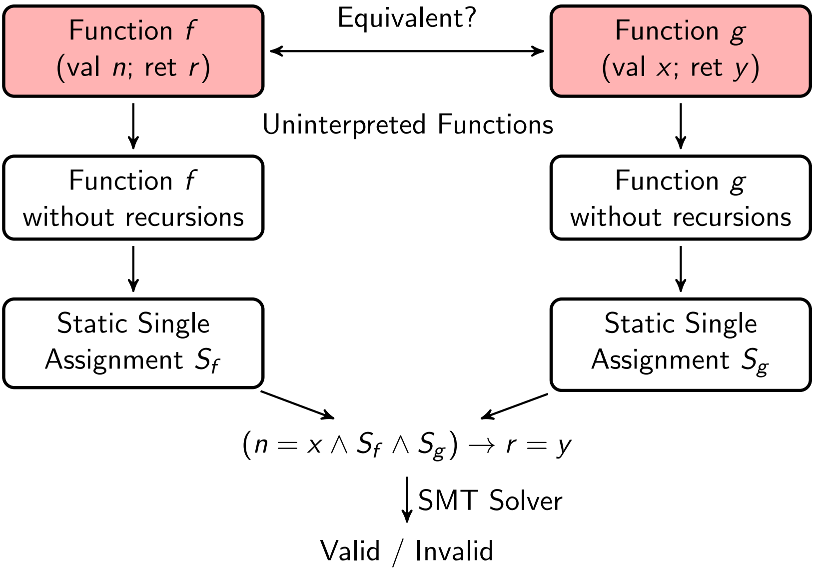
Preliminary Considerations III

Array Access over Boundaries

- Arrays have infinite size in Z3
- Possibility: Check array boundaries on every access
- Programs can be proven to honor array boundaries
- **Solution:** Assume programs have been proven to honor array boundaries

Tool for Regression Verification

Overview



Tool for 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

Tool for Regression Verification

Formally prove there are no new bugs

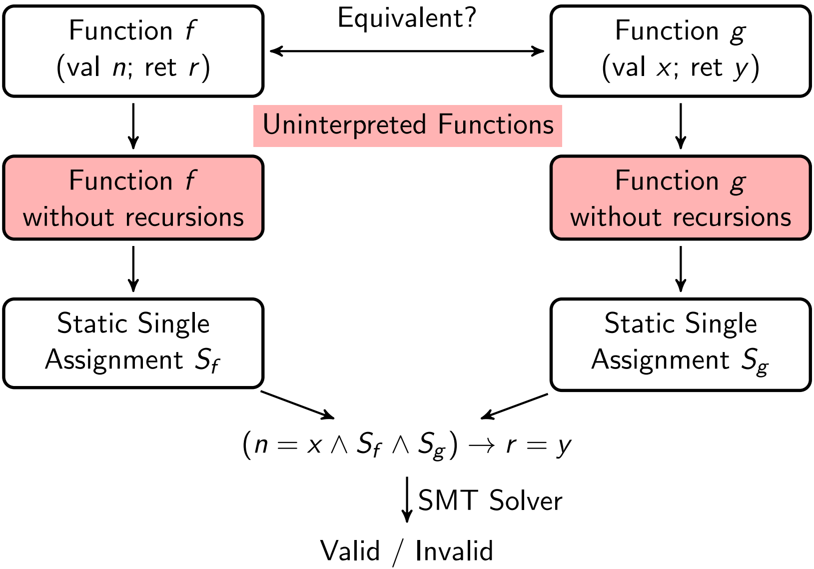
- Goal: Proving the equivalence of two closely related programs
- No formal specification or test cases required
- Instead use old program version
- Make use of similarity between programs

```
int gcd1(int a, int b) {
  int g = 0;
  if (b == 0) {
    g = a;
  } else {
    a = a % b;
    g = gcd1(b, a);
  }
  return g;
}
```

```
int gcd2(int x, int y) {
  int z = x;


  if (y > 0) {

    z = gcd2(y, z % y);
  }
  return z;
}
```


Function f (val n; ret r)
Equivalent?
Function g (val x; ret y)
Uninterpreted Functions
Function f without recursions
Function g without recursions
Static Single Assignment S_f
Static Single Assignment S_g
$(n = x \wedge S_f \wedge S_g) \to r = y$
SMT Solver
Valid / Invalid

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 gcd1(int a, int b) {
  int g = 0;
  if (b == 0) {
    g = a;
  } else {
    a = a % b;
    g = U(b, a);
  }
  return g;
}
```

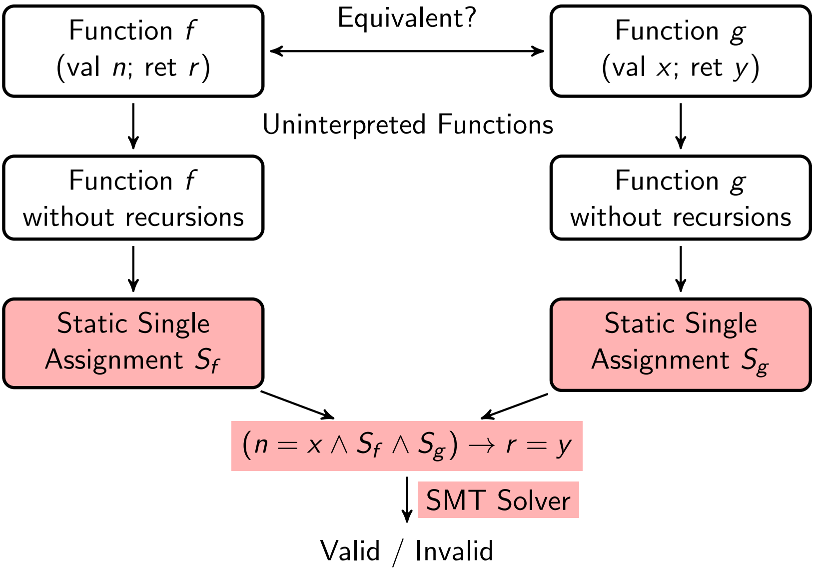
```
int gcd2(int x, int y) {
  int z = x;


  if (y > 0) {

    z = U(y, z % y);
  }
  return z;
}
```

Conversion of Programs to Formulae

Overview



Conversion of Programs to Formulae I

General idea

- Walk Abstract Syntax Tree of both programs
- Convert every SimPL construct to SMT formula:

```
int x = y;        ⇒    declare-fun x_0 () Int
                       assert (x_0 = y_i)

                  ⋮

if (y) {
  x = b;               assert (x_i = b)
} else {          ⇒    assert (x_(i+1) = c)
  x = c;               assert (x_(i+2) = (ite y
}                        x_i x_(i+1))) ; Phi node
```

- Use new variable for every assignment

Conversion of Programs to Formulae II

Regression Verification

- Uninterpreted Functions:

```
assert (forall ((u Int) (v Int)
  ((gcd1 u v) = (gcd2 u v))))
```

- Proof $f = g$:

```
assert (not (gcd1_result = gcd2_result))
check-sat
get-model
exit
```

$\Rightarrow$ **Objective "Regression Verification proofs": Done**

Case Study

Done

- Collect examples: Papers, Refactoring Rules, ...
- 51 program pairs so far

Planned

- Framework for testing them
- Check how well extensions work
- More (interesting) examples

⇒ **Objective "Case Study": Work in Progress**

Convert Loops to Recursions

Idea

- Convert every loop to a new recursive function
- Handling multiple loop variables: Return a tuple

```
while (x < 10) {
  y = y + x;
  x = x - 1;
}
```

```
  (x,y) = loop(x,y);
    ⋮
tuple loop(int x, int y) {
  if (x < 10) {
    y = y + x;
    x = x - 1;
    (x,y) = loop(x,y);
  }
  return (x,y); }
```

Initial work

- Added tuples to SimPL grammar and AST

Function Inlining

Idea

- Specify how often a function call is inlined:

```
y = f(x) inline 3;
```

- Same for loops (converted to functions):

```
while (x < y) inline 5 {
  z;
}
```

- Possibility later: Inlining strategies

Initial work

- Modified grammar to support inlining

Abstraction Refinement I

- Recursive Functions are the main problem
- Two ways of dealing with them:

Most general abstraction

- Classical Regression Verification approach
- Uninterpreted functions
- $\forall x : f(x) = g(x)$
- No further information about the functions

$\Rightarrow$ **Only works when the function bodies are equivalent**

Abstraction Refinement II

No abstraction

- Give recursive definition:

```
forall x. f(n) =
    let r0 = 0
        r1 = n
        r2 = f(n-1)
        r3 = n + r2
        r4 = ite(n <= 1, r1, r3)
    in r4
```

- Experiments for a few simple functions

$\Rightarrow$ **Only works when the function bodies differ for finite number of inputs**

Abstraction Refinement III

Problem: Find an abstraction inbetween

CEGAR Loop

- Counter Example Guided Abstraction Refinement
- Start with simple over-approximation
- Extract patterns from counter examples
- Refine Abstraction
- Repeat if proof still fails

Abstraction Refinement IV

Problem: Find an abstraction inbetween

Horn Clauses

- $(p \wedge q \wedge \cdots \wedge t) \rightarrow u$
- Postcondition PC is true after recursive call
- $r = f(n) \rightarrow PC(n, r)$
- Solver figures out Postcondition on its own (e.g. using CEGAR)

Summary

Regression Verification

- Prove that two similar programs are equivalent
- Better chance of being adopted than Formal Verification
- More powerful than Regression Testing

Project Status

1. Develop Regression Verification tool:
    - Basic tool: **Done**
    - Loops to Recursions: **WIP**
    - Function Inlining: **WIP**
2. Case study to compare approaches: **WIP**
3. Extend tool: **Planning and Experimentation**