Regression Verification: Proving Partial Equivalence

Dennis Felsing

Seminar within the "Projektgruppe Formale Methoden der Softwareentwicklung"
Application-oriented Formal Verification
Institute for Theoretical Informatics
Am Fasanengarten 5, 76131 Karlsruhe, Germany
dennis.felsing@student.kit.edu

Abstract. Regression Verification aims at proving the equivalence of two closely related versions of a program, as they often exist in the life cycle of programs. We present a proof rule developed by Benny Godlin and Ofer Strichman for proving the partial equivalence of such program versions. We also illustrate the theoretical and practical frameworks surrounding this proof rule and discuss their limitations.

1 Introduction

In *Formal Verification* the correctness of some software is formally proven. For this a formal specification of the correct behaviour of the program has to be given, which poses a difficulty in practice.

Regression Testing on the other hand is the act of testing new versions of some software to discover new bugs. This requires writing extensive test cases, which is laborious and still does not guarantee that all bugs will be found.

The approach of *Regression Verification* is to formally prove that no bugs have been introduced into a new version of some software. This means it is situated between Formal Verification and Regression Testing. Based on two closely related programs, usually an older and a newer version of a program in development, their equivalence shall be proven. Instead of comparing the two programs to a common formal specification Regression Verification tries to make use of their similarity. Therefore neither a formal specification nor test cases are required. Basically the old program version specifies the correct behaviour. This means that the "correctness" that Regression Verification can prove is of a weaker form, as it can only show that a program is as correct as an older version of it.

The work presented in this paper has been developed by Benny Godlin and Ofer Strichman.[1][2]

Figure 1 visualizes the theoretical framework of Regression Verification, which will be detailed in this paper. The question is whether two closely related programs P_1 and P_2 are partially equivalent. The necessary terms for this will be discussed in Section 2.

To prove partial equivalence the programs are transformed using the proof rule Proc-P-Eq to a non-recursive form, which will be the topic of Section 3.

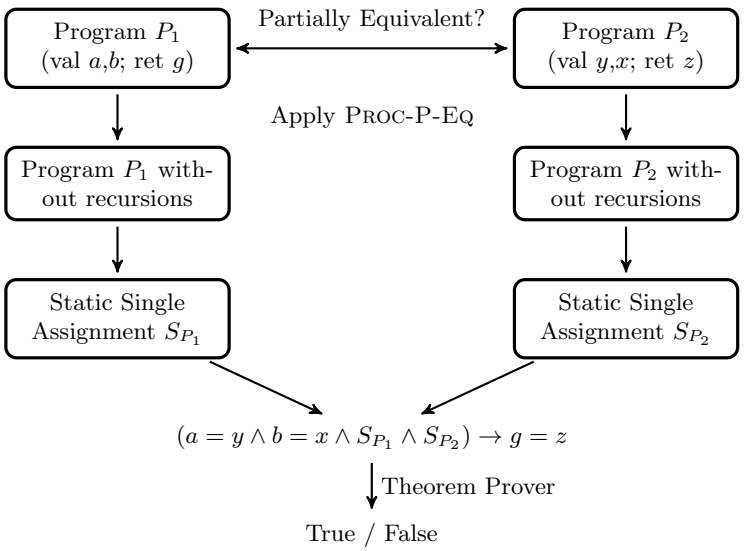


Fig. 1: Outline of the theoretical framework of Regression Verification

Finally in Section 4 we look at how these non-recursive programs can then be transformed into a logical formula, which is passed to a theorem prover to determine whether the programs are equivalent. In Section 5 the practical aspect of Regression Verification is illuminated by discussing the Regression Verification Tool (RVT). Finally in Section 6 the limitations of proof rule Proc-P-Eq and RVT are considered.

2 Preliminaries

We will use a simple programming language for all examples in this paper, the *Linear Procedure Language* (LPL). It is also for this language that the central proof rule Proc-P-Eq has been proven correct.

Definition 1. *The following grammer defines the* Linear Procedure Language*:*

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

where $p \in Proc$ is a procedure, e is an expression, and B is a predicate. The notations ***val*** $\overline{arg - r_p}$ *and* ***ret*** $\overline{arg - w_p}$ *respectively indicate a sequence of input and output arguments.*

Note that the Linear Procedure Language does not support loops. This constraint is merely syntactic and does not restrict the expressiveness of the language, as loops can be reformulated as recursions.

Figure 2 contains an example program in the Linear Procedure Language, which consists of two procedures. Procedure *gcd3* calculates the greatest common divisor by calling *gcd* twice. Procedure *gcd* implements a variant of Euclid's algorithm for calculating the greatest common divisor of two numbers.

```
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
```

Fig. 2: An example program in the Linear Procedure Language

The notion of equivalence we consider in this paper is *partial equivalence*, which means that we do not consider the case of non-terminating programs:

Definition 2. *Two programs P_1 and P_2 are called* partially equivalent *iff given the same inputs, any two terminating executions of P_1 and P_2 return the same values:*

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

Definition 3. *An* uninterpreted procedure *is a procedure of which it is only known that it returns the same outputs when called with the same inputs. It is denoted in the Linear Procedure Language as follows:*

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

We will use uninterpreted procedures in the proof rule Proc-P-Eq to reduce recursive procedures to non-recursive ones. Which procedures of program P_1 map to which in P_2 is determined by *map*:

Definition 4. *map* : $Proc[P_1] \mapsto Proc[P_2]$ *is a relation which maps partially equivalent procedures of Program* P_1 *to those of* P_2*.*

Uninterpreted procedures and procedure mappings can now be used together in the definition of UP, which will be used in Proc-P-Eq:

Definition 5. UP *maps procedures to their respective uninterpreted procedures, so that:*

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

3 Proof Rule Proc-P-Eq

We will use the programs P_1 and P_2 in Figure 3 to motivate the final form of proof rule Proc-P-Eq in three steps.

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

(a) Program P_1

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

  z := x;

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

(b) Program P_2

Fig. 3: Two programs calculating the greatest common divisor of two positive integers

The procedures `gcd1` and `gcd2` are partially equivalent. In order to prove this, we can assume that the recursive calls to `gcd1` and `gcd2` respectively have the same effect on the output variable depending on the input variables. Assuming the partial equivalence of the recursive calls we now have to prove that the bodies of `gcd1` and `gcd2` are partially equivalent. This is the essence of the first form of rule Proc-P-Eq:

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

This rule can be reformulated. We assumed the equivalence of the recursive calls. The uninterpreted procedures $UP(gcd1)$ and $UP(gcd2)$ are equal by the definition of UP. Therefore we can replace the recursive calls in the procedure bodies by calls to the resulting uninterpreted procedure. At this point the procedures have been modified to the ones in Figure 4.

This is the essence of the second form of rule Proc-P-Eq:

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

(a) Program P_1 without recursions

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

  z := x;

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

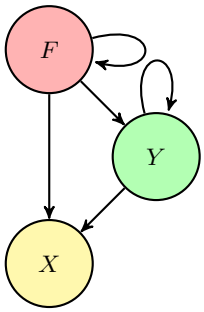
(b) Program P_2 without recursions

Fig. 4: The programs from Figure 3 with uninterpreted procedures replacing the recursive calls

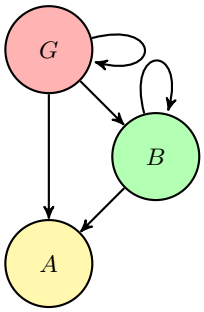
$$\frac{\vdash_{\mathbb{L}_{\mathbb{UP}}} \text{part-equiv}(gcd1[gcd1 \leftarrow UP(gcd1)], gcd2[gcd2 \leftarrow UP(gcd2)])}{\text{part-equiv}(gcd1, gcd2)}$$

We are now using the proof system $\mathbb{L}_{\mathbb{UP}}$, which is a sound proof system for a non-recursive LPL.

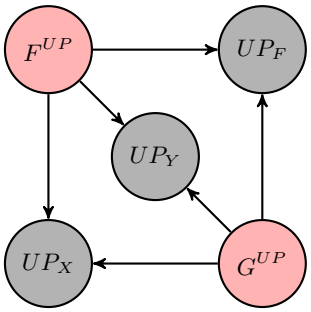
In a more general case a procedure may not only call itself but also other procedures, which may themselves be recursive or form a more complex kind of circle. An example for this is visible in the call graphs in Figure 5.



(a) Call graph of some program Q_1



(b) Call graph of some program Q_2



(c) Call graphs after applying Proc-P-Eq

Fig. 5: Transformation of program call graphs by applying Proc-P-Eq

These procedure calls also have to be replaced by uninterpreted procedures, which leaves us with the following final proof rule Proc-P-Eq:

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

$F^{UP} = F[f \leftarrow UP(f) \mid f \in Proc[P]]$ is an *isolated procedure*, which means that all procedure calls from the procedure are replaced by their corresponding uninterpreted procedure.

4 Using Proc-P-Eq

After applying Proc-P-Eq to all procedures in a program they are transformed into procedures which contain no calls to actual procedures, only uninterpreted ones. These procedures can be transformed into formulas in *Static Single Assignment* form. This means that in all assignments of the form $x := exp$ we replace x with a new variable x_1. Every line of the Static Single Assignment represents a state of the corresponding procedure.

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

  z := x;

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

(a) Procedure *gcd2* after applying Proc-P-Eq

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

(b) Static Single Assignment of *gcd2*

Fig. 6: Example of transforming a procedure into Static Single Assignment

The Static Single Assignments can be integrated into a formula, as is seen in the example formula in Figure 7. At this point the formula can be passed to a theorem prover, like Z3, to prove its validity or find a counter example.

$$(\underbrace{a = y \wedge b = x}_{\text{Equal inputs}} \wedge S_{gcd_1} \wedge S_{gcd_2}) \rightarrow \underbrace{g = z}_{\text{Equal outputs}}$$

Fig. 7: Final formula for deciding partial equivalence of gcd_1 and gcd_2

5 Regression Verification Tool

The approach Strichman and Godley chose in practice differs in some parts from the one described in their papers. They have developed a Regression Verification

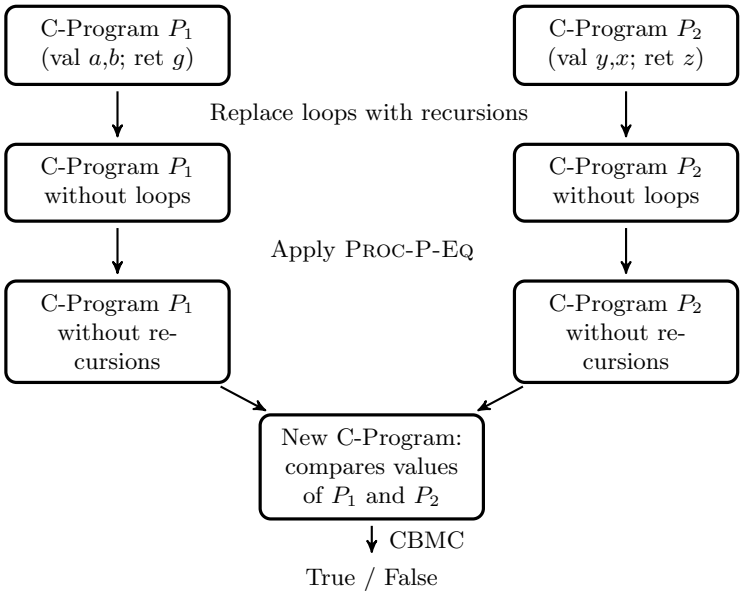


Fig. 8: Outline of the practical framework implemented in the Regression Verification Tool

Tool[1], which can be used to prove the partial equivalence of two programs written in C.

As the proof rule Proc-P-Eq works on recursions initially all loops are replaced by recursions. Afterwards a mapping of all functions in the programs is determined by mapping functions with the same name and functions which appear at the same syntactic position in an otherwise unchanged function onto each other.

Now Proc-P-Eq is applied to the functions in the call graph bottom up. If partial equivalence cannot be shown for a function, it is inlined into all calls to it, and Proc-P-Eq is applied to the resulting functions.

6 Limitations

While the equivalence of many procedures can be shown using Proc-P-Eq there are some cases which do not work. The following procedures calculating $r = \Sigma_{i=1}^{n} i$ illustrate the limitations.

In the example in Figure 9 the procedures F and G call themselves recursively with different arguments $n-1$ and $n-2$. This represents a difference in the procedure bodies. Therefore the premise of rule Proc-P-Eq does not hold and

[1] `https://code.google.com/p/rvt/`

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

Fig. 9: Examples

it can not be applied. Partial Equivalence of procedures F and G can not be proven even though they are equivalent.

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

Fig. 10: Examples

In a similar way the procedures in Figure 10 can not be shown to be equivalent, because their procedure bodies differ.

In Figure 11 partial equivalence can not be shown because of the Uninterpreted Procedure. By replacing the recursive calls to F and G by an Uninterpreted Procedure U all semantic information about them is lost, except that they return the same values for the same inputs. For this reason the proof rule Proc-P-Eq can not recognize that the check for $r \geq 0$ after calling G will always return $True$.

The Regression Verification Tool has some further limitations, which do not result from using Proc-P-Eq:

The condition of equality can not be specified for a program or function. This means that there is no way to tell that fixes for bugs, which were introduced in the program, are not bugs instead.

When two programs are partially equivalent, RVT usually proves this equivalence in a short time. But when the programs are not partially equivalent RVT tries to inline functions in all possible ways to still prove the equivalence, which

```
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
```

Fig. 11: Examples

leads to a long running time. For the user of RVT this means that no counterexample for the two programs being partially equivalent is provided.

The mapping of functions to be proven equivalent works in RVT by comparing the names of procedures. If the names are equal, two procedures are mapped. Otherwise the syntactic appearence of function calls in another function determines the mapping of those functions called. This works well for many small changes, but may occasionally need user interaction to guide the mapping process, which is not provided by RVT.

7 Conclusion

Because Regression Verification does not require a formal specification it has a higher chance than Formal Verification of being adopted in actual software projects. Regression Verification is also more powerful than Regression Testing, as it covers all possible cases.

The proof rule Proc-P-Eq is very simple and yet covers a lot of refactorings. The simple examples which fail to be proven equivalent by Proc-P-Eq show that there is still room for improvement. Whether another simple proof rule could be used to replace Proc-P-Eq or whether a group of rules would be necessary to cover these examples remains to be examined.

The Regression Verification Tool provides a working system, which, although not being complete, demonstrates how Regression Verification can be used in practice for a real programming language.

Regression Verification is still an active topic of research, as can for example be seen by the recent paper by Ofer Strichman et al. which adds support for multi-threaded programs to the Regression Verification framework.[3]

References

1. B. Godlin and O. Strichman, "Regression verification," in *DAC*, 2009, pp. 466–471.
2. ——, "Inference rules for proving the equivalence of recursive procedures," *Acta Inf.*, vol. 45, no. 6, pp. 403–439, Jul. 2008.
3. S. Chaki, A. Gurfinkel, and O. Strichman, "Regression verification for multi-threaded programs," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, V. Kuncak and A. Rybalchenko, Eds. Springer Berlin Heidelberg, 2012, vol. 7148, pp. 119–135.