A Note on the Automated Generation of an Algorithm Verification Method

Bruno Buchberger
Research Institute for Symbolic Computation
Johannes Kepler University, Linz, Austria
bruno.buchberger@jku.at

Acknowledgement: Sponsored by FWF (Österreichischer Fonds zur Förderung der Wissenschaftlichen Forschung; Austrian Science Foundation), project SFB 1302 ("*Theorema*") of the SFB 013 ("Numerical and Symbolic Scientific Computing").

1 Abstract

A natural verification rule for proving that simple recursive programs satisfy in-out problem specifications is automatically generated by the author’s lazy thinking algorithm synthesis method.

Keywords: algorithm schemes, verification of recursive algorithms, lazy thinking, algorithm synthesis, failure analysis, *Theorema*.

2 In−OutProblems

Throughout this paper, K is a theory (or "knowledge base"), i.e. a collection of (predicate logic) formulae. Let P be a binary predicate constant occurring in K and f a unary function constant. Then the formula

(in-out) $\underset{x}{\forall} P[x, f[x]]$

is called an "in−outspecification of f".

(Of course, we could also consider multivariate functions f. Also, we could restrict the inputs to those satisfying a certain input specification. However, we want to keep the presentation as simple as possible and avoid technicalities which do not contribute to the essence of this paper. Also, in this paper, we do not consider the question of termination but only the question of partial correctness. We could add termination considerations but the emphasis of this paper is on the relation between a general algorithm synthesis method and a verification method for recursive programs, which we want to present without being distracted by termination.)

If P and f are "given", i.e. some knowledge on P and f is contained in K, then the task of proving (in−out) is called "the verification of the correctness of f w.r.t. to the problem P".

If P is "given", i.e. some knowledge on P is contained in K, then finding a definition of f (in terms of other functions in the knowledge base) is called the "synthesis of a function (in particular, an algorithm) for the problem P".

Note that many of the (algorithmic) problems are in−outproblems but also note that there are many interesting (algorithmic) problems that are not in−out problems. For example, the problem of sorting is a typical in−outproblem: Prove that

$$\underset{x}{\forall}\,(\text{is-sorted-version}[x,\ f[x]]).$$

In contrast, the problem of "canonical simplification"

$$\underset{x}{\forall}\,(x \sim f[x])$$
$$\underset{x,y}{\forall}\,(x \sim y \Rightarrow (f[x] = f[y])),$$

where '~' is typically an equivalence, is not an in−out problem: i.e. we cannot formulate a predicate P such that the two formulae can be written in the form (in-out).

3 Simple Recursive Programs

The recursive programs which we consider in this paper have the form

$$\text{(rec)}\qquad f[x] = \begin{cases} s[x] & \Leftarrow\ c[x] \\ h[x,\ f[g1[x]],\ f[g2[x]]] & \Leftarrow\ \text{otherwise} \end{cases}$$

where c, s, g1, g2, and h are known auxiliary functions.

(Of course, we could also consider the case where, instead of the two recursive calls f[g1[x]] and f[g2[x]], we have only one recursive call f[g[x]] or we have more than two recursive calls.)

4 A Natural Verification Rule for Explicit Problems and Simple Recursive Programs

[Popov 2004] observes that, if we apply Scott induction (defined in [DeBakker 1969]) for proving that a function f defined by (rec) satisfies (in−out), one obtains the following verification rule:

In order to prove that

$$\underset{x}{\forall} P[x,\ f[x]]$$

it suffices to prove that

$$\underset{x}{\forall} (c[x] \Rightarrow P[x, s[x]])$$

and

$$\underset{x,y1,y2}{\forall} \begin{pmatrix} \neg\, c[x] \\ P[g1[x], y1] \\ P[g2[x], y2] \end{pmatrix} \Rightarrow P[x, h[x, y1, y2]].$$

It is clear that the above verification rule for recursive programs could also be guessed directly (or stipulated axiomatically) from the intuitive understanding of function substitution and the iteration of recursive calls.

5 The Automated Synthesis of the Verification Rule by Lazy Thinking

Now, we want to show that the above verification rule can also be automatically (!) synthesized by applying the "lazy thinking" algorithm synthesis method introduced in [Buchberger 2003], which demonstrates the power of this method.

Roughly, the lazy thinking synthesis method suggests to try a scheme for f, e.g. the recursive scheme

$$f[x] = \begin{cases} s[x] & \Leftarrow\ c[x] \\ h[x, f[g1[x]], f[g2[x]]] & \Leftarrow\ \text{otherwise} \end{cases}$$

and to attempt a proof of

$$\underset{x}{\forall} P[x, f[x]]$$

by any proof method whatsoever.

The proof will most probably fail because nothing is known about the ingredient functions c, s, g1, g2, and h. Then one apply a "requirements generating" algorithm that generates requirements for the ingredient functions which leads to a synthesis problem for the ingredient functions, and so on until we arrive at requirements for auxiliary functions that can be fulfilled by functions already available in the knowledge base.

The current requirements generating algorithm proposed in [Buchberger 2003] consists of the following rule:

Generalization Rule: If the failing proof situation consists of the temporary assumption(s)

$$A$$

and the temporary goal

$$G$$

containing the Skolem constant(s) ("arbitrary but fixed" constants) x0 and, maybe, containing terms that start with the function symbol f, then do the following: Replace in A and G different terms starting with f by different new variables y1, y2, etc. and replace x0 by a new variable x yielding, say, formulae A’ and G’. Then the requirement is

$$\underset{x,y1,y2,\ldots}{\forall} (A' \Rightarrow G').$$

Special Case: If in A and G, there are no terms starting with f then x0 has to be replaced by a new variable x yielding A’ and G’ and the requirement is just.

$$\underset{x}{\forall} (A' \Rightarrow G').$$

Let us now apply this synthesis method to the above recursive scheme. Let us apply Noetherian induction w.r.t. to some Noetherian relation $\succ$ on the given domain, for which

$$\underset{x}{\forall} \left(\neg\, c[x] \Rightarrow \begin{matrix} g1[x] \prec x \\ g2[x] \prec x \end{matrix} \right)$$

holds.

Then we obtain the following proof attempt:

We take x0 arbitrary but fixed and assume as induction hypothesis

$$\underset{x<x0}{\forall} P[x, f[x]].$$

We want to prove

$$P[x0, f[x0]].$$

Now, we have two cases:

Case c[x0]: In this case, by the recursive presentation of f, we have to prove

$$P[x0, s[x0]].$$

Here the proof is stuck. By the requirements generation algorithm, we obtain the requirement

$$\underset{x}{\forall} (c[x] \Rightarrow P[x, s[x]]).$$

*Case ¬c[x0]:*In this case, by the recursive presentation of f, we have to prove

$$P[x0, h[x0, f[g1[x0]], f[g2[x0]]]].$$

Now, by the induction hypothesis,

$$\begin{array}{l} P[g1[x0], f[g1[x0]]], \\ P[g2[x0], f[g2[x0]]]. \end{array}$$

Here the proof is stuck. By the requirements generation algorithm, we obtain the requirement

$$\underset{x,y1,y2}{\forall} \left(\begin{array}{l} \neg\, c[x] \\ P[g1[x], y1] \\ P[g2[x], y2] \end{array} \Rightarrow P[x, h[x, y1, y2]] \right).$$

Hence, we see that we obtain, completely automatically, the following verification rule for recursive algorithms of the form (rec) and in−outproblem specifications, which is exactly the rule in [Popov 2004]:

In order to prove that

$$\underset{x}{\forall} P[x, f[x]]$$

it suffices to prove that

$$\underset{x}{\forall} (c[x] \Rightarrow P[x, s[x]])$$

and

$$\underset{x,y1,y2}{\forall}\left(\begin{array}{l}\neg\, c[x]\\ P[g1[x],\, y1]\ \Rightarrow P[x,\, h[x,\, y1,\, y2]]\\ P[g2[x],\, y2]\end{array}\right).$$

References

[Buchberger 2003]
B. Buchberger. Algorithm Invention and Verification by Lazy Thinking.
In: D. Petcu, V. Negru, D. Zaharie, T. Jebelean (eds), Proceedings of SYNASC 2003 (Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania, October 1-4, 2003), Mirton Publishing, ISBN 973-661-104-3, pp. 2-26.

[Popov 2004]
N. Popov. Verification of Simple Recursive Programs.
Manuscript, RISC (Research Institute for Symbolic Computation), Johannes Kepler University, January 2004.

[DeBakker 1969]
J. W. DeBakker and D. Scott. A Theory of Programs.
IBM Seminar, Vienna, Austria, unpublished notes, 1969.