Formal Development and Verification of Approximation Algorithms Using Auxiliary Variables

Rudolf Berghammer[1] and Markus Müller-Olm[2]

[1] Institut für Informatik und Praktische Mathematik
Universität Kiel
Olshausenstraße 40, D-24098 Kiel, Germany
`rub@informatik.uni-kiel.de`

[2] Fachbereich 4, Lehrstuhl V
Universität Dortmund
Baroper Straße 301, D-44221 Dortmund, Germany
`mmo@ls5.cs.uni-dortmund.de`

Abstract. For many intractable optimization problems efficient approximation algorithms have been developed that return near-optimal solutions. We show how such algorithms and worst-case bounds for the quality of their results can be developed and verified as structured programs. The proposed method has two key steps. First, auxiliary variables are introduced that allow a formal analysis of the worst-case behavior. In a second step these variables are eliminated from the program and existential quantifiers are introduced in assertions. We show that the elimination procedure preserves validity of proofs and illustrate the approach by two examples.

1 Introduction

Algorithm design and formal program development and verification are two well established domains in computer science and applied mathematics. Formerly they mostly have been coexisting. But in recent years many computer scientists noticed that the design and verification of efficient algorithms which are not only correct "in principle" but in all details can benefit from techniques of formal program development and verification. Therefore, there is an increasing cooperation between the two fields.

In this paper techniques of formal program development and verification are applied to *approximation algorithms*. Such algorithms (see [11] for an overview) have been developed because a great variety of important optimization problems cannot be solved efficiently. Approximation algorithms are usually very fast but return only near-optimal solutions (unless $P = NP$). Hence, besides feasibility of their results, estimates for the closeness to the optimal solutions are of interest. We show how approximation algorithms and, in particular, their worst-case bounds can formally be derived and verified as structured programs using the well-known assertion method pioneered by Floyd and Hoare [7, 10, 5, 1].

The proposed method for proving worst-case bounds has two key steps. In the first step, auxiliary variables are added to the program. They are used to collect information that is referred to in the informal proofs but is not present in the algorithm itself. The worst-case behaviour can then be analysed formally by strengthening the assertions used in the feasibility proof. In the second step, the auxiliary variables are removed from the program and existential quantifiers are introduced in assertions. This avoids the inefficient calculation of auxiliary variables at run-time. It also allows to use non-constructive or expensive operations in assignments to auxiliary variables. We show that this elimination procedure preserves validity of partial correctness proofs.

The paper is organized as follows: Section 2 illustrates that auxiliary variables are useful for a formal verification of worst-case behaviour. For this purpose, we recall a well-known approximation algorithm for the minimum vertex cover problem and its proof in the usual semi-formal mathematical way. Afterwards, we show how this proof can be formalized. Proof outlines, which are decisive for our reasoning, are considered in Section 3 and three simple conditions for their validity are presented. In Section 4 we show how auxiliary variables can be eliminated from proof outlines and prove that the resulting proof outline shows the same partial correctness property as the original proof outline. Section 5 applies our method to a second example, an approximation algorithm for the problem of computing an independent set of maximum size. We also discuss some variants of this algorithm. We conclude in Section 6 with some further applications and ideas for future research.

2 An Illustrative Example

Let $g = (V, E)$ be an undirected and loop-free graph with finite (and non-empty) set V of vertices and set E of edges, where each edge is a set $\{x, y\}$ with $x, y \in V$ and $x \neq y$. A *vertex cover* of g is a subset C of V such that every edge in g is incident to some vertex in C, i.e., $e \cap C \neq \emptyset$ for all $e \in E$. To compute a vertex cover of minimum size is an *NP*-hard problem; see [3]. There is a simple greedy approximation algorithm for that problem attributed to Gavril and Yannakakis in [3]. Expressed as a while-programs it reads as given below, where we assume that the non-deterministic assignment $e :\in F$ assigns an arbitrary element of $F \neq \emptyset$ to e and the call $inc(e)$ yields the set $\{f \in E \mid e \cap f \neq \emptyset\}$ of all edges incident to edge e:

```
$C := \emptyset; F := E;$
**while** $F \neq \emptyset$ **do**
  $e :\in F;$
  $C := C \cup e; F := F \setminus inc(e)$ **od**.          (VC$_1$)
```

In [3] it is also shown that this program always returns a vertex cover C of g whose size $|C|$ is guaranteed to be no greater than twice the minimum size c_{opt} of a vertex cover of g. The idea underlying this proof is to consider the set M

of all edges that were picked by the statement $e :\in F$ and to show that M is a matching of g with $|C| \leq 2 * |M|$. (A set of edges is a matching if no two different edges are incident.) The estimation $|C| \leq 2 * c_{\text{opt}}$ then follows from the fact that a vertex cover C^* of g of minimum size must include at least one vertex of any edge of M, i.e., $|M| \leq |C^*| = c_{\text{opt}}$.

As prevalent in algorithmics, the correctness proof of Gavril and Yannakakis' algorithm in [3] is done in a "free-style" mathematical way without formal problem specification and program verification. But it can also be formalized in the assertion approach if the program (VC_1) is refined as follows:

```
{ $true$ }
$C := \emptyset; F := E; M := \emptyset;$
{ $inv(C, F, M)$ }
while $F \neq \emptyset$ do                                   ($\text{VC}_2$)
  $e :\in F;$
  $C := C \cup e; F := F \setminus inc(e); M := M \oplus e$ od
{ $post(C)$ } .
```

In this annotated program a call of the operation $\oplus$ is assumed to insert an element into a set, i.e., $M \oplus e$ yields $M \cup \{e\}$; the loop invariant $inv(C, F, M)$ is defined as the conjunction of

$$C \text{ vertex cover of } g_F = (V, E \setminus F)\,, \tag{1}$$

$$M \text{ matching of } g = (V, E)\,, \tag{2}$$

$$|C| \leq 2 * |M|\,, \tag{3}$$

$$\forall\, e \in M, f \in F : e \cap f = \emptyset\,, \tag{4}$$

and the post-condition $post(C)$ as the conjunction of

$$C \text{ vertex cover of } g = (V, E)\,, \tag{5}$$

$$|C| \leq 2 * c_{\text{opt}}\,. \tag{6}$$

Termination of the annotated program (VC_2) is obvious since $e :\in F$ is defined because of the condition of the while-loop and, due to $e \in inc(e)$, the value of F is strictly decreased in every run throuh the while-loop. To show partial correctness of (VC_2) wrt. the pre-condition $true$ and the post-condition $post(C)$, three proof obligations have to be discharged (see Section 3 or [6, 9] for details). First of all, the initialization must establish the loop invariant if the pre-condition holds, i.e.,

$$true \implies inv(\emptyset, E, \emptyset)\,.$$

Secondly, each execution of the loop's body must maintain the loop invariant, i.e., the implication

$$F \neq \emptyset \wedge inv(C, F, M) \implies inv(C \cup e, F \setminus inc(e), M \oplus e)$$

must be shown for all C, F, and M and for any e in F. The proofs of both implications are easy exercises and, therefore, left out. Note, however, that in the second case assertion (4) is necessary to obtain the matching property of $M \oplus e$ from $e \in F$ and the matching property of M. The third and final task is to show that upon termination of the loop the post-condition follows from the loop's exit condition and the loop invariant, which leads to

$$F = \emptyset \land \mathit{inv}(C, F, M) \implies \mathit{post}(C)$$

for any F as last proof obligation. Here (5) follows from (1) and $F = \emptyset$ and (6) is a consequence of (2) and (3), as already shown.

3 Proof Outlines

Proof methods for correctness of programs have been the topic of intense research for more than three decades (see [5] for an overview). Floyd's method [7] of inductive assertions and Hoare logic [10] are particularly well known. In this paper we use proof outlines for proof presentation which combine the strength of both methods. They allow a proof presentation on the level of structured programs but lead to a more compact proof representation as full proof trees. A *proof outline* is a program annotated with assertions as the example (VC_2) in Section 2.

An *assertion* is a predicate on the values of the variables used in a program. In practice, assertions are given by predicate-logic formulas. Suppose we have two assertions *pre* and *post*. Then, program π is called *partially correct* with respect to pre-condition *pre* and post-condition *post* if any terminating execution of π from an initial state that satisfies *pre* ends in a state that satisfies *post*; if, in addition π terminates from any state in *pre*, it is called *totally correct* with respect to *pre* and *post*. While we are ultimately interested in total correctness, it is in most cases easier in formal program development and verification to first concentrate on partial correctness and prove termination separately afterwards. Because there is nothing special about termination proofs of approximation algorithms, we focus on partial correctness reasoning in this paper.

A *program element* is a Boolean expression (condition) or an atomic statement. We will discuss the following types of atomic statements in this paper: the "do-nothing" statement **skip**, (deterministic) assignments $x := t$, and non-deterministic assignments $x :\in S$. For simplicity, we assume that t and S are total expressions. The exposition can straightforwardly be extended to other kinds of atomic statements.

Now, let p and q be two assertions in a proof outline. Then a *segment* from p to q is a sequence of program elements that may be traversed successively in an execution of the underlying program on the way from p to q; a segment is not allowed to extend over an assertion. We refrain from a more formal definition but provide an illustrative example. Consider the following simple generic proof outline, where S_1, S_2, S_3, and S_4 are atomic statements, b is a condition, and *pre*, *inv*, and *post* are assertions:

$$\begin{array}{l} \{\ \mathit{pre}\ \} \\ S_1; S_2; \\ \{\ \mathit{inv}\ \} \\ \textbf{while}\ b\ \textbf{do} \\ \quad S_3; S_4\ \textbf{od} \\ \{\ \mathit{post}\ \}\,. \end{array}$$

In it, we have the three segments $\langle S_1, S_2\rangle$ from *pre* to *inv*, $\langle b, S_3, S_4\rangle$ from *inv* to *inv*, and $\langle \neg b\rangle$ from *inv* to *post*.

As demonstrated in Section 2, each segment in a proof outline gives rise to a proof obligation: it must be partially correct with respect to the surrounding assertions. This proof obligations is best captured in terms of the weakest liberal pre-condition of the segment, which is inductively defined by

$$\begin{array}{rcl} \mathsf{wlp}(\varepsilon, q) & :\Longleftrightarrow & q\,, \\ \mathsf{wlp}(e\cdot s, q) & :\Longleftrightarrow & \mathsf{wlp}(e, \mathsf{wlp}(s, q))\,. \end{array}$$

Here ε is the empty sequence (*empty segment*) and $e\cdot s$ is the concatenation of the program element e and the segment s. This definition refers to the weakest liberal pre-condition of program elements, which is given by

$$\begin{array}{rcl} \mathsf{wlp}(b, q) & :\Longleftrightarrow & b \rightarrow q\,, \\ \mathsf{wlp}(\textbf{skip}, q) & :\Longleftrightarrow & q\,, \\ \mathsf{wlp}(x := e, q) & :\Longleftrightarrow & q[e/x]\,, \\ \mathsf{wlp}(x :\in S, q) & :\Longleftrightarrow & S \neq \emptyset \wedge (\forall\, s \in S : q[s/x])\,. \end{array}$$

It is understood that s is a fresh variable in the clause for $x :\in S$, i.e. a variable q is independent of, and $q[s/x]$ is obtained by substituting s for x in q.

Note that the conjunct $S \neq \emptyset$ in the clause for $x :\in S$ puts the obligation on the algorithm designer to prove that S is non-empty. Omitting this conjunct results in an alternative definition in which $\mathsf{wlp}(x :\in S, q)$ holds trivially for all states, in which S evaluates to the empty set. One could argue that this alternative definition would be in accordance with the philosophy of partial correctness: to choose a value from an empty set should result in a run-time error and, like divergent paths, execution paths leading to run-time errors could be ignored in partial correctness. However, without the conjunct $S \neq \emptyset$ we cannot allow non-deterministic assignments to auxiliary variables as otherwise the elimination procedure described in Section 4 would become unsound. Here is a minimal example for the problem: Without the conjunct $S \neq \emptyset$, the proof outline $\{\mathit{true}\}\ a :\in \emptyset\ \{\mathit{false}\}$ is valid in the sense defined below. Step (i) of the elimination procedure of Section 4, however, results in $\{\mathit{true}\}$ **skip** $\{\mathit{false}\}$ which is certainly wrong.

Now, a proof outline is *valid* for assertions *pre* and *post* if it satisfies the following three conditions:

(a) Assertion *pre* is placed at the beginning and assertion *post* at the end of the proof outline.
(b) Any loop in the underlying program is broken by an assertion; typically this is achieved by placing a loop invariant right in front of every loop.
(c) For every segment s from an assertion p to another assertion q in the proof outline, p implies $\mathsf{wlp}(s, q)$.

Intuitively, the second condition guarantees that a proof outline induces only a finite number of segments and the last condition says that all segments are partially correct wrt. the surrounding assertions. As every execution of the program is composed of executions of segments, a valid proof outline proves partial correctness of the underlying program π wrt. *pre* and *post*.

4 Auxiliary Variables and Their Elimination

The example in Section 2 illustrates that the enrichment of programs and proof outlines by auxiliary variables often allows a clearer statement of the underlying argument in a formal verification. However if the auxiliary variables are left in the executed version of the program they may lead to inefficiencies caused by additional computations, which in certain cases even forbids the use of the modified programs in practice.

In order to overcome this disadvantage, we show in this section that auxiliary variables can always be eliminated from proof outlines without affecting their validity. From a theoretical point of view, this result proves that auxiliary variables are unnecessary in order to perform a correctness proof. Nevertheless, we recommend their practical use because of the above reason. Note that our result even allows the use of pre-algorithmic constructs (like set comprehension or quantification) in assignments to auxiliary variables. This often simplifies formal reasoning considerably.

In view of our applications, a finite set of variables A is called a *set of auxiliary variables* in a program π if variables $a \in A$ are used only in assignments of the form $x := t$ or $x :\in S$ where $x \in A$. That is: auxiliary variables must not be used in assignments to non-auxiliary variables and in guards of loops or conditionals.[1] Therefore, they can neither influence the control flow nor the values held by non-auxiliary variables Auxiliary variables may be used freely in the assertions of a proof outline. In order to ensure that the specification proved by a proof outline is independent of auxiliary variables, we require, however, that auxiliary variables do not appear freely in the pre- and the post-condition. In the proof outline (VC_2) in Section 2, for instance, M is an auxiliary variable.

In order to eliminate the auxiliary variables $a \in A$ from π, we perform the following simple step; the resulting program is called $\tilde{\pi}$ in the following.

(i) Remove any assignment of the form $a := t$ or $a :\in S$ with $a \in A$ from π.

[1] Some authors, e.g., [12], use the notion "auxiliary variables" for variables whose appearance is restricted to assertions and whose values may not be changed by programs. Usually, in the literature such variables are called *logical variables*.

We can think of this step as a replacement of all these assignments in π by the neutral statement **skip**. Of course, a valid proof outline for π will in general no longer be valid if π is replaced by the modified program $\tilde{\pi}$, as the assertions may use auxiliary variables in an essential way. As we will prove in a moment, however, we can regain a valid proof outline for $\tilde{\pi}$ by the following second step:

(ii) Replace in addition any assertion p in the old proof outline in which auxiliary variables occur freely by the assertion $(\exists\, a_1, \ldots, a_k : p)$, where $a_1, \ldots, a_k \in A$ are the auxiliary variables occurring free in p.

Note that this transformation of the proof outline leaves both pre- and post-condition unchanged, as they do not contain free occurrences of auxiliary variables. Hence, the modified proof outline proves, if indeed valid, the same partial correctness property as the original one, but for the modified program $\tilde{\pi}$.

If we apply (i) and (ii) to the valid proof outline (VC$_2$) of Section 2 we get the following valid proof outline (VC$_3$) showing the partial correctness of the original program (VC$_1$) wrt. the pre-condition *true* and the post-condition *post*(C):

```
{ $true$ }
$C := \emptyset; F := E;$
{ $\exists\, M : inv(C, F, M)$ }
while $F \neq \emptyset$ do                                  (VC$_3$)
  $e :\in F;$
  $C := C \cup e; F := F \setminus inc(e)$ od
{ $post(C)$ } .
```

Let $\bar{a}$ be a list of variables that contains each variable of A exactly once. We now show that the application of steps (i) and (ii) to a valid proof outline leads always again to a valid proof outline. As the first two conditions (a) and (b) of Section 3 for validity of proof outlines are not affected by the transformation via (i) and (ii), we only have to show that the third condition (c) remains true. Without loss of generality, we can assume that step (ii) replaces *all* assertions p in the proof outline by $(\exists\, \bar{a} : p)$. This assumption which smoothes the proof is justified by the fact that existential quantification over variables a predicate q is independent of results in an equivalent predicate.

As a preparation for the correctness proof, we show an interesting relationship between a program element e in the original program and the program element $\tilde{e}$ which replaces it in the modified program when step (i) is applied:

Lemma 4.1. *Let q be an assertion. Then*

$$(\exists\, \bar{a} : \mathsf{wlp}(e, q)) \quad \Longrightarrow \quad \mathsf{wlp}(\tilde{e}, (\exists\, \bar{a} : q)) \,.$$

Proof. The assertions $(\exists\, \bar{a} : \mathsf{wlp}(e, q))$ and $\mathsf{wlp}(\tilde{e}, (\exists\, \bar{a} : q))$ are even equivalent if the program element e is **skip**, a deterministic assignment $x := t$ to a non-auxiliary variable x, or a guard b. For e being **skip** this is obvious: both assertions reduce to $(\exists\,\bar a : q)$. For deterministic assignments equivalence is proved by

$$\begin{array}{lll} & (\exists\,\bar a : \mathsf{wlp}(x{:=}t, q)) & \\ \Longleftrightarrow & (\exists\,\bar a : q[t/x]) & \text{definition } \mathsf{wlp} \\ \Longleftrightarrow & (\exists\,\bar a : q)[t/x] & \text{predicate logic} \\ \Longleftrightarrow & \mathsf{wlp}(x{:=}t, (\exists\,\bar a : q)) & \text{definition } \mathsf{wlp}. \end{array}$$

The second equivalence exploits that x is a non-auxiliary variable and that t does not depend on auxiliary variables.

The case of guards is shown by

$$\begin{array}{lll} & (\exists\,\bar a : \mathsf{wlp}(b, q)) & \\ \Longleftrightarrow & (\exists\,\bar a : b \rightarrow q) & \text{definition } \mathsf{wlp} \\ \Longleftrightarrow & b \rightarrow (\exists\,\bar a : q) & \text{predicate logic} \\ \Longleftrightarrow & \mathsf{wlp}(b, (\exists\,\bar a : q)) & \text{definition } \mathsf{wlp}. \end{array}$$

Note that the second equivalence depends on b being independent of auxiliary variables and the fact that the variables of $\bar a$ have a non-empty type, a standard assumption for program variables.

In the case of a non-deterministic assignment $x{:\in}S$ to a non-auxiliary variable x we calculate as follows, where the second step exploits that S is independent of all the variables in $\bar a$ because $\bar a$ consists of auxiliary variables and the third step that for the fresh variable s the property $s \notin A$ can be assumed:

$$\begin{array}{lll} & (\exists\,\bar a : \mathsf{wlp}(x{:\in}S, q)) & \\ \Longleftrightarrow & (\exists\,\bar a : S \neq \emptyset \wedge (\forall\, s \in S : q[s/x])) & \text{definition } \mathsf{wlp} \\ \Longrightarrow & S \neq \emptyset \wedge (\forall\, s \in S : (\exists\,\bar a : q[s/x])) & \text{predicate logic} \\ \Longleftrightarrow & S \neq \emptyset \wedge (\forall\, s \in S : (\exists\,\bar a : q)[s/x]) & \text{since } x, s \notin A \\ \Longleftrightarrow & \mathsf{wlp}(x{:\in}S, (\exists\,\bar a : q))\,. & \text{definition } \mathsf{wlp}. \end{array}$$

For deterministic assignments to an auxiliary variable $a \in A$ the proof looks as follows, where in the fourth step of the derivation s is replaced by a and the two quantifiers are combined.

$$\begin{array}{lll} & (\exists\,\bar a : \mathsf{wlp}(a := t, q)) & \\ \Longleftrightarrow & (\exists\,\bar a : q[t/a]) & \text{definition } \mathsf{wlp} \\ \Longleftrightarrow & (\exists\,\bar a : (\exists s : s = t \wedge q[s/a])) & \text{one-point rule} \\ \Longrightarrow & (\exists\,\bar a : (\exists s : q[s/a])) & \text{weakening} \\ \Longleftrightarrow & (\exists\,\bar a : q) & \text{see above} \\ \Longleftrightarrow & \mathsf{wlp}(\mathbf{skip}, (\exists\,\bar a : q)) & \text{definition } \mathsf{wlp}. \end{array}$$

Finally, for a non-deterministic assignment $a{:\in}S$ to an auxiliary variable we calculate as follows, where in the second step the conjunct $S \neq \emptyset$ ensures that the range of the universal quantification is non-empty such that it can be strengthened to an existential quantification:

$$\begin{array}{lll} & (\exists\,\bar a : \mathsf{wlp}(a{:\in}S, q)) & \\ \Longleftrightarrow & (\exists\,\bar a : S \neq \emptyset \;\wedge\; (\forall\, s \in S : q[s/a])) & \text{definition } \mathsf{wlp} \\ \Longrightarrow & (\exists\,\bar a : S \neq \emptyset \;\wedge\; (\exists\, s \in S : q[s/a])) & \text{see above} \\ \Longrightarrow & (\exists\,\bar a : (\exists\, s : q[s/a])) & \text{weakening.} \end{array}$$

The last formula is equivalent to $\mathsf{wlp}(\mathbf{skip}, (\exists \bar{a} : q))$ as we have seen in the proof for deterministic assignments to auxiliary variables. □

A simple inductive argument shows that the implication in Lemma 4.1 transfers from program elements to segments:

Lemma 4.2. *Suppose s is a segment in the original proof outline, $\tilde{s}$ is the corresponding segment in the modified proof outline, and q is an assertion. Then*

$$(\exists\, \bar{a} : \mathsf{wlp}(s, q)) \quad\Longrightarrow\quad \mathsf{wlp}(\tilde{s}, (\exists\, \bar{a} : q))\,.$$

Proof. The induction base of s being empty is trivial: since $\tilde{s}$ is empty, too, both sides of the implication reduce to $(\exists\, \bar{a} : q)$.

For the induction step of s being of the form $e\cdot r$ we get the derivation

$$\begin{array}{lll} & (\exists\, \bar{a} : \mathsf{wlp}(e\cdot r, q)) & \\ \Longleftrightarrow & (\exists\, \bar{a} : \mathsf{wlp}(e, \mathsf{wlp}(r, q))) & \text{definition } \mathsf{wlp} \text{ for segments} \\ \Longrightarrow & \mathsf{wlp}(\tilde{e}, (\exists\, \bar{a} : \mathsf{wlp}(r, q))) & \text{Lemma 4.1} \\ \Longrightarrow & \mathsf{wlp}(\tilde{e}, \mathsf{wlp}(\tilde{r}, (\exists\, \bar{a} : q))) & \text{induction hypothesis, monotonicity} \\ \Longleftrightarrow & \mathsf{wlp}(\tilde{e}\cdot\tilde{r}, (\exists\, \bar{a} : q)) & \text{definition } \mathsf{wlp} \text{ for segments} \end{array}$$

which concludes the proof since the concatenation $e\cdot r$ is modified to $\tilde{e}\cdot\tilde{r}$. □

After these preparations, it is now easy to show correctness of the elimination procedure.

Theorem 4.1. *The application of steps (i) and (ii) to a valid proof outline leads again to a valid proof outline.*

Proof. As conditions (a) and (b) of Section 3 for validity of proof outlines are not affected by the transformation via (i) and (ii), we only have to show that condition (c) remains true.

For the purpose of proving (c), assume that the original proof outline satisfies condition (c) and suppose we are given a segment t from some assertion $(\exists\, \bar{a} : p)$ to some assertion $(\exists\, \bar{a} : q)$ in the transformed proof outline. As the transformation affects only the form of single statements and assertions but not the global structure of the proof outline, there is a corresponding segment s from p to q in the original proof outline such that the modification $\tilde{s}$ of s equals t. By assumption, the original proof outline satisfies condition (c); thus, p implies $\mathsf{wlp}(s, q)$. We can now show the desired implication by

$$\begin{array}{lll} & (\exists\, \bar{a} : p) & \\ \Longrightarrow & (\exists\, \bar{a} : \mathsf{wlp}(s, q)) & p \Rightarrow \mathsf{wlp}(s, q), \text{ monotonicity} \\ \Longrightarrow & \mathsf{wlp}(\tilde{s}, (\exists\, \bar{a} : q)) & \text{Lemma 4.2} \\ \Longleftrightarrow & \mathsf{wlp}(t, (\exists\, \bar{a} : q)) & t = \tilde{s}. \end{array}$$

□

5 A Further Example

Let $g = (V, E)$ be an undirected and loop-free graph and assume that the call $ngb(x)$ computes the set $\{y \in V \mid \{x, y\} \in E\}$ of all neighbour vertices of vertex x. A set of vertices of g is called *independent* if no two vertices in it are connected via an edge from E. To derive a program for computing such a set S, we start with the assertion

$$S \text{ independent set of } g = (V, E) \tag{7}$$

as post-condition $post(S)$. Using a new variable X, then we generalize (7) to

$$X \subseteq V \text{ and } S \text{ independent set of } g_X = (X, E_X)\,, \tag{8}$$

where $E_X = \{e \in E \mid e \subseteq X\}$, i.e., g_X is the subgraph of g generated by X. Choosing the conjunction of (8) and the assertion

$$\forall\, x \in V \setminus X, y \in S : \{x, y\} \notin E \tag{9}$$

as loop invariant $inv(S, X)$, it is straightforward to derive the program for computing S shown in the following valid proof outline:

```
{ $true$ }
$S := \emptyset; X := \emptyset;$
{ $inv(S, X)$ }
while $X \neq V$ do                                        (IS$_1$)
  $x :\in V \setminus X;$
  $S := S \oplus x; X := X \cup (ngb(x) \oplus x)$ od
{ $post(S)$ } .
```

The definedness of the non-deterministic assignment $x :\in V \setminus X$ in this proof outline follows from the assumption $V \neq \emptyset$, the part $X \subseteq V$ of the loop invariant, and the condition of the while-loop. Termination of the while-loop is obvious.

Like the minimum vertex cover problem of Section 2, the problem of computing an independent set of maximum size is *NP*-hard and the program of the proof outline (IS$_1$) implements a well-known approximation algorithm proposed and studied by Wei [16].

Now, we formally analyze the worst-case behaviour of Wei's algorithm using our method. To this end, we enrich the program of (IS$_1$) by an auxiliary variable U, which consists of all the sets $ngb(x) \oplus x$ which are added to X on all iterations of the while loop. This step leads to a first strengthening of the original loop invariant $inv(S, X)$: we add the conjunct

$$X = \textstyle\bigcup_{u \in U} u\,. \tag{10}$$

Let $\Delta(g) = \max_{x \in V} |ngb(x)|$ be the so-called *degree* of the graph g. It is obvious that each set in U has at most $\Delta(g) + 1$ elements and we add also the corresponding assertion

$$\forall\, u \in U : |u| \leq \Delta(g) + 1 \tag{11}$$

to the original loop invariant. Finally, we notice that whenever some set $ngb(x) \oplus x$ is added to U, the vertex x is added to S. Due to its choice via the assignment $x :\in V \setminus X$ and assertion (8), x is not yet a member of S. Thus, S cannot be smaller than U. The corresponding estimation

$$|U| \leq |S| \tag{12}$$

is the third addition to the original loop invariant. Altogether, we obtain the following proof outline with auxiliary variable U and a loop invariant $inv(S, X, U)$ given as conjunction of the assertions (8) to (12); its validity proof is rather simple since it contains all necessary information:

```
{ $true$ }
$S := \emptyset; X := \emptyset; U := \emptyset;$
{ $inv(S, X, U)$ }
while $X \neq V$ do                                                    (IS$_2$)
  $x :\in V \setminus X;$
  $S := S \oplus x; X := X \cup (ngb(x) \oplus x); U := U \oplus (ngb(x) \oplus x)$ od
{ $post(S)$ } .
```

Now, we are able to estimate the worst-case behaviour of Wei's algorithm formally. Assume $inv(S, X, U)$ and $X = V$ and let s_{opt} be the size of a maximum independent set of g. Then, we obtain

$$s_{\text{opt}} \leq |V| = |\bigcup_{u \in U} u| \leq \sum_{u \in U} |u| \leq |U| * (\Delta(g) + 1) \leq |S| * (\Delta(g) + 1) .$$

Here the second step follows from assertion (10) and the loop's exit condition $X = V$ and the fourth and fifth step use assertions (11) and (12), respectively. Hence, the proof outline (IS$_2$) remains valid if its post-condition $post(S)$ is strengthened to $post'(S)$ by adding the conjunct

$$s_{\text{opt}} \leq |S| * (\Delta(g) + 1) . \tag{13}$$

Now the auxiliary variable U can be eliminated using steps (i) and (ii) of Section 4 and this, finally, yields the following valid proof outline for Wei's algorithm which includes a worst-case estimation in its post-condition as desired:

```
{ $true$ }
$S := \emptyset; X := \emptyset;$
{ $\exists U : inv(S, X, U)$ }
while $X \neq V$ do                                 (IS$_3$)
  $x :\in V \setminus X;$
  $S := S \oplus x; X := X \cup (ngb(x) \oplus x)$ od
{ $post'(S)$ } .
```

We sketch some variants of Wei's algorithm in the remainder of this section. They are based on a slight modification of the above estimation, viz.

$$s_{\text{opt}} \leq |V| = |\bigcup_{u \in U} u| \leq \sum_{u \in U} |u| \leq |U| * \max_{u \in U} |u| \leq |S| * \max_{u \in U} |u| \,.$$

From this property, we get $\max_{u \in U} |u|$ as a worst-case bound of Wei's algorithm which is potentially better than the previous bound $\Delta(g) + 1$. This maximum is not known in advance, but it can easily be computed as a further result. A corresponding modification of the proof outline (IS_2) followed by the elimination of the auxiliary variable U leads to

```
$\{ \; \mathit{true} \; \}$
$S := \emptyset; X := \emptyset; s := 0;$
$\{ \; \exists\, U : \mathit{inv}(S, X, U, s) \; \}$
**while** $X \neq V$ **do**                                                    $(\text{IS}_4)$
  $x :\in V \setminus X;$
  $S := S \oplus x; X := X \cup (\mathit{ngb}(x) \oplus x); s := \mathit{max}(s, |\mathit{ngb}(x) \oplus x|)$ **od**
$\{ \; \mathit{post}(S, s) \; \}\,.$
```

Here the maximum $\max_{u \in U} |u|$ is stored in the variable s and the postcondition $\mathit{post}(S, s)$ consists of the conjunction of the assertion (7) and the estimation

$$s_{\text{opt}} \leq |S| * s \,. \tag{14}$$

Furthermore, the assertion $\mathit{inv}(S, X, U, s)$ occurring in the loop invariant is obtained from the assertion $\mathit{inv}(S, X, U)$ in (IS_3) by replacing (11) with

$$s = \max_{u \in U} |u| \,. \tag{15}$$

With this information, it is rather simple to show that (IS_4) is indeed a valid proof outline. Of course, the program can slightly be improved by the use of a further variable which avoids the two-fold evaluation of the expression $\mathit{ngb}(x) \oplus x$.

By a little modification of the hitherto derivation, the computed worst-case bound can be improved as follows: Instead of collecting all sets $\mathit{ngb}(x) \oplus x$ in U, we only collect their "non-visited parts". I.e., we replace the assignment $U := U \oplus (\mathit{ngb}(x) \oplus x)$ by $U := U \oplus ((\mathit{ngb}(x) \setminus X) \oplus x)$ in the proof outline (IS_2). To minimize the value of $\max_{u \in U} |u|$ further, we refine the non-deterministic assignment $x :\in V \setminus X$ of (IS_2) and pick now (via a simple loop) the vertex x in such a way from $V \setminus X$ that the size of the set $\mathit{ngb}(x) \setminus X$ is minimal. Obviously, both steps do not affect the invariance property of $\mathit{inv}(S, X, U)$. Hence, the modified proof outline is also valid. Starting with this proof outline, we obtain the desired algorithm following the above derivation of (IS_4) from (IS_2).

To choose in each iteration a node of minimal degree in the non-visited part of the graph is a common heuristic in connection with Wei's algorithm. The idea, however, to enrich the algorithm by a variable s that computes the resulting worst-case bound and the formal correctness proof seems not to appear in the previous literature.

6 Concluding Remarks

When verifying approximation algorithms, we are particularly interested in estimates for the maximal deviation of their results from optimal solutions of the given problem. While the informal feasibility proofs found in the literature on algorithmics can usually be reformulated as formal assertional proofs straightforwardly, the informal proofs of worst-case bounds often refer to entities that are not present in the program and are thus difficult to formalize. As a solution, we proposed in this paper to collect additional information in auxiliary variables inserted into the program which can be referred to in assertions. These auxiliary variables are introduced solely for the purpose of the formal assertional proof and are not meant to stay in the executed version of the program. Therefore, we showed how they can be eliminated from proof outlines without affecting validity. Their elimination also allows references to non-constructive or expensive operations in assignments to auxiliary variables.

Our approach can be seen as a particular application of the well-known "invent and verify" technique for program development combined with transformational programming. In doing so, sometimes the specific form of the assignments to auxiliary variables formally can be calculated using, e.g., the postcondition and/or the invariant(s). But the usual way to introduce auxiliary variables is by an "invent" step. In the latter case only the "verify" step is a formal one. The elimination of auxiliary variables corresponds to the application of a schematic transformation rule.

The approach has been applied to many other, more complex approximation algorithms besides the two examples considered in this paper, in particular, to all approximation algorithms discussed in [14]. This includes e.g., the formal development of a program implementing Chvátal's well-known approximation algorithm for set covering. (A description of this algorithm in the common informal style can be found in [3].)

When investigating the formal development and verification of approximation algorithms we have also discovered a new result viz. an approximation algorithm for the bin packing problem which possesses $\frac{3}{2}$ as absolute worst-case approximation bound and runs in linear time[2]. It follows the Best Fit idea. In contrast with the original approach, however, it works with two partial solutions P_1 and P_2 instead of one and two auxiliary bins B_1 and B_2 – one for each partial solution. Roughly speaking, it proceeds as follows: First, the objects are packed one by one into B_1 until its capacity would be exceeded by the insertion of some object u. In this situation the contents of B_1 is inserted into P_1, the bin B_1 is cleared, u is packed into B_2, and the process starts again with the remaining objects. If, however, the insertion of u would lead to an overfilling of B_1 as well as B_2,

[2] As shown in [15], there is no approximation algorithm for the bin packing problem with an absolute approximation factor smaller than $\frac{3}{2}$, unless $P = NP$. But all approximation algorithms with this – in all probability – optimal absolute approximation factor which can be found in the literature are non-linear; the best running time is $\mathcal{O}(n * \log n)$ with n as number of objects. See the overview paper [4].

then additionally the contents of B_2 is inserted into P_2 and u is packed into the cleared B_2. This "book-keeping" in combination with a suitable selection of the next object (based on a partition of the objects into small and large ones at the beginning of the algorithm) allows one to avoid the costly search of a bin the next object will fit in optimally. The decisive idea for proving the bound $\frac{3}{2}$ is to use a function f which yields for a bin B of P_1 the unique object whose insertion into B would lead to an overfilling.

Details of the algorithm can be found in [2], where its correctness (i.e., the feasibility of the result and the absolute approximation factor $\frac{3}{2}$) formally is verified using the assertion technique. In doing so, f is not used as an auxiliary variable but as a logical variable in assertions. It is, however, not hard to develop the algorithm in the style proposed in the current paper using an auxiliary variable that holds the function f. We believe that this development is more clear and easier to follow than the original one.

Besides their use in approximation algorithms we have investigated also some other applications of auxiliary variables, for instance, their utility for formal termination proofs and for program specialization. We illustrate the latter application by a somewhat academic example. Consider the following proof outline for a program calculating quotient and remainder of two positive integers x and y by iterated subtraction:

```
$\{ x > 0 \wedge y > 0 \}$
$q := 0; r := x;$
$\{ 0 \leq r \wedge x = q \cdot y + r \}$
**while** $r \geq y$ **do**
   $q := q + 1; r := r - y$ **od**
$\{ x = q \cdot y + r \wedge 0 \leq r < y \}$
```

If we weaken the post-condition to $r = x \bmod y$, then q becomes an auxiliary variable. Subsequently applying elimination leads to the following proof outline:

```
$\{ x > 0 \wedge y > 0 \}$
$r := x;$
$\{ (\exists q : 0 \leq r \wedge x = q \cdot y + r) \}$
**while** $r \geq y$ **do**
   $r := r - y$ **od**
$\{ r = x \bmod y \}$
```

The underlying program computes just the remainder of integer division, i.e., is a specialization of the program of the first proof outline.

Interestingly, auxiliary variables are of crucial importance for the approach of Owicki aud Gries to verification of concurrent programs too: they are necessary for achieving completeness [8]. Therefore, proof rules for elimination of auxiliary variables are found in connection with that method. Rule 17 in [1, p. 193], for instance, reads as follows, where S_0 is the program obtained from program S by deleting all assignments to auxiliary variables.

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ S_0\ \{q\}}\ .$$

This rule is usually proved to be correct by referring to an underlying operational semantics; see, e.g., the proof of Lemma 5.17 in [1, p. 196]. We believe that our proof of the correctness of elimination on the level of predicate transformers is much more elegant. Moreover, our result is stronger. We not only show that partial correctness is preserved but validity of whole proof outlines, if auxiliary variables are removed from the programs and existentially quantified in the assertions.

Auxiliary variables can also be used for performing data refinements. In doing so, firstly, concrete variables intended to replace abstract variables are added to the program as auxiliary variables and updated in parallel with the abstract variables such that a coupling invariant is preserved. In a second step then the program is algorithmically refined in such a way that the abstract variables become auxiliary variables. Finally, the abstract variables are eliminated. Morgan [13] investigates this method in the context of predicate transformer semantics. But he does not describe a systematic transformation for eliminating auxiliary variables from proof outlines as we do here.

To sum up: Due to the experiences gained so far, we believe that auxiliary variables in combination with proof outlines and our elimination procedure are a valuable means for formal program development, verification and proof documentation.

Acknowledgments: We gratefully acknowledge discussions with F. Reuter and thank the referees for their valuable comments.

References

1. Apt K.R., Olderog E.-R.: Verification of sequential and concurrent programs. Springer (1991)
2. Berghammer R., Reuter F.: A linear approximation algorithm for bin packing with absolute approximation factor $\frac{3}{2}$. Science of Computer Programming 48, 67-80 (2003)
3. Cormen T.H., Leiserson C.E., Rivest R.L.: Introduction to algorithms. The MIT Press (1990)
4. Coffmann Jr. E.G., Garay M.R., Johnson D.S.: Approximation algorithms for bin packing: A survery. In: Hochbaum D.S. (ed.): Approximation algorithms for *NP*-hard problems. PWS Publishers, 46-93 (1996)
5. Cousot P.: Methods and logics for proving programs. In: van Leeuwen J. (ed.), Handbook of Theoretical Computer Science, Vol. B, pp. 841-993, Elsevier (1990)
6. Dijkstra E.W.: A discipline of programming. Prentice-Hall (1976)
7. Floyd R.W.: Assigning meanings to programs. In: Schwartz J.T. (ed.), Proc. Symp. on Applied Mathematics 19, American Mathematical Society, pp. 19-32 (1967)
8. Francez N.: Program Verification. Addison-Wesley (1992)

9. Gries D.: The science of computer programming. Springer (1981)
10. Hoare C.A.R.: An axiomatic basis of computer programming. Comm. ACM 12, pp. 576-583 (1969)
11. Hochbaum D.S. (ed.): Approximation algorithms for *NP*-hard problems. PWS Publishers (1996)
12. Kleymann T.: Hoare logic and auxiliary variables. Formal Aspects of Computing 11, 541-566 (1999)
13. Morgan C.: Auxiliary variables in data refinement. Information Processing Letters 29, pp. 293-296 (1988)
14. Reuter F.: On the formal specification and derivation of approximation algorithms using assertions (in German). Diploma thesis, Inst. für Informatik und Praktische Mathematik, Universität Kiel (2000)
15. Simchi-Levi D.: New worst-case results for the bin packing problem. Naval Research Logistics 41, 479-485 (1994)
16. Wei V.K.: A lower bound for the stability number of a simple graph. Bell Lab. Tech. Memor. 81-11217-9 (1981)