PROGRAM INVERSION: MORE THAN FUN!

Wei CHEN

Department of Computer Science, Washington University, St. Louis, MO 63130, USA

Jan Tijmen UDDING

Department of Mathematics and Computing Science, Groningen University, Groningen, Netherlands

Communicated by M. Rem
Received August 1989
Revised April 1990

Abstract. We introduce proof rules for inverting a program. We derive an algorithm to compute the preorder and inorder traversals of a binary tree. Subsequently, we invert this algorithm to arrive at an algorithm to construct a tree from its preorder and inorder traversals. We prove this program correct using the proof rules for inversion rather than directly. Since a proper formulation of a provable invariant of this program appears to be quite awkward, this example reinforces the view that program inversion is a useful technique and more than fun.

Introduction

The concept of program inversion is due to Edsger W. Dijkstra and W.H.J. Feijen [2]. Subsequently, it was explored by David Gries in [3]. Since then, the concept has popped up every so often, e.g. in [4, 5]. However, to our knowledge no formal definition of program inversion has ever been given and neither have proof rules been formulated for the individual program constructs.

In this paper we provide a formal basis for program inversion. It allows us to invert a program and to establish the correctness of the result without a direct proof for it. One might think that the invariant and variant function of the program to be inverted can be used to prove the correctness of the inverted program. It turns out that this is not always the case. The example of constructing a binary tree from its preorder and inorder traversals demonstrates this exquisitely.

The paper is organized as follows. In the next section we introduce the proof rules for program inversion. Subsequently, we derive an algorithm to compute the preorder and inorder traversals of a binary tree. In order to aid the inversion process we examine this solution closely and strengthen the invariant slightly, requiring the tree to be uniquely labeled. In Section 3 we do the actual inversion. We conclude with a few remarks.

We assume the reader to be familiar with a Hoare triple and its definition in terms of the weakest precondition wp [1]. For predicates P and Q, and for program S we
mean by \( \{ P \} S \{ Q \} \) that \( P \Rightarrow wp(S, Q) \). We use Dijkstra’s guarded command language [1]. We typically use \( B_i \) and \( C \), for guards, \( S \), and \( T \), for commands, and \( E_0 \) and \( E_1 \) for expressions. With if \( B_i \rightarrow S_i \), \( \text{if} \) we denote the alternative command consisting of the guarded commands \( B_i \rightarrow S_i \). for \( i \) ranging over some fixed domain. A similar convention applies to the repetitive construct. The predicate that is true everywhere is denoted by \textit{true}. We allow unbounded nondeterminism.

1. Proof rules for program inversion

We introduce the proof rules for program inversion; their proofs are collected in Appendix A. Informally, we say that program \( T \) is an inverse of program \( S \) if \( T \) exactly retraces the steps of \( S \) and ends up in the state from which \( S \) started. One can also think of inversion more liberally and allow the inverse \( T \) to end up in a state differing from the one that \( S \) started from. Given a precondition of the program \( S \) one would only require the inverse \( T \) to end in a state satisfying that precondition. We aim at an inversion process that yields totally correct programs by construction. By the more liberal view of inversion we would not be able to guarantee termination of an inverted repetition if we were to allow the inverse to reach new states. Therefore, we take the more strict view of inversion in which the inverted program exactly retraces the steps of the program to be inverted, which is more formally stated in the next definition.

\textbf{Definition 1.1.} Program \( T \) is said to be an inverse of program \( S \) under precondition \( P \) exactly when \( \{ P \land Q \} S ; T \{ Q \} \) for all predicates \( Q \).

Program inversion can be an alternative method for finding a program \( T \) satisfying \( \{ P \} T \{ Q \} \). Rather than solving this problem directly using standard techniques, it might be easier to find a program \( S \) satisfying \( \{ Q \} S \{ P \} \) and then to invert it. Notice that the definition of an inverse program does not state properties of the inverse in isolation. It expresses that execution of program \( S \) starting in a certain initial state followed by execution of its inverse \( T \) amounts to skip. Hence, only if we start execution of \( T \) in a state that can be viewed as the result of a computation of \( S \) can we draw any conclusion about the final state of \( T \) in isolation.

We state for all four constructs in the language sufficient conditions to invert them. For sequential composition, for the alternative command, and for the repetition, these conditions are expressed in terms of the statements from which these commands are constructed. This allows a stepwise approach to program inversion, as demonstrated in the next sections. In this way we end up with the obligation to invert primitive assignment statements.

Not all assignment statements can be inverted. A necessary condition to invert the assignment \( x := E \) under precondition \( P \) is that for any value \( A \) the equation \( x : P \land (A = E) \) has at most one solution. Even if an assignment statement enjoys
this property the inverse assignment need not exist. This depends upon the permissible expressions in the language. For example, $2^x$ may be an expression in the language, whereas its inverse $\log x$ may not be. This does not mean that such an assignment cannot be inverted. It just means that a simple assignment will not do the job. The following proof rule states under which conditions an assignment statement can be inverted by another assignment statement.

**Proof Rule for the Assignment Statement**

$$
P \Rightarrow \text{def}(E_0) \land (\text{def}(E_1)) \land \left(x = (E_1) \right)
\{P \land Q\} x := E_0 ; x := E_1 \{Q\} \quad \text{for all } Q
$$

Here $\text{def}(E)$ means that expression $E$ is well defined.

The proof rule for sequential composition basically states that we can invert the sequential composition of two statements if we can invert each of them individually under the appropriate preconditions.

**Proof Rule for Sequential Composition**

$$
\{P\} S_0 \{R\}
\{P \land Q\} S_0 ; T_0 \{Q\} \quad \text{for all } Q
\{R \land Q\} S_1 ; T_1 \{Q\} \quad \text{for all } Q
\{P \land Q\} S_0 ; S_1 ; T_1 ; T_0 \{Q\} \quad \text{for all } Q
$$

The first two conditions of the proof rule for the alternative statement state that mutually exclusive postconditions exist for the guarded commands of the alternative construct to be inverted. The third condition guarantees its termination (nonterminating constructs cannot be inverted). The last condition states that each guarded command has an inverse under the appropriate preconditions.

**Proof Rule for the Alternative Statement**

$$
\{P \land B_i\} S_i \{C_i\} \quad \text{for all } i
\{P \land B_i\} S_i \{\neg C_i\} \quad \text{for all } i \text{ and } j \text{ with } i \neq j
P \Rightarrow (\exists i :: B_i)
\{P \land B_i \land Q\} S_i ; T_i \{Q\} \quad \text{for all } i \text{ and } Q
\{P \land Q\} \text{ if } B_i \rightarrow S_i ; \text{ if } C_i \rightarrow T_i \{Q\} \quad \text{for all } Q
$$

The first condition of the proof rule for the repetitive construct states that the repetition to be inverted terminates, provided that initially $P$ and none of the $C_i$ hold. The next two conditions require $P$ to be an invariant of the repetition and the $C_i$ to be mutually exclusive postconditions of the guarded commands of the repetition. The last condition states that the body of the repetition can be inverted under the appropriate preconditions.
Proof Rule for the Repetitive Construct

\[ \{ P \land (\forall i : (\neg C_i)) \} \rightarrow B \rightarrow S, \od \{ \text{true} \} \]
\[ \{ P \land B \}, S, \{ P \land C \} \quad \text{for all } i \]
\[ \{ P \land B \}, S, \{ \neg C \} \quad \text{for all } i \text{ and } j \text{ with } i \neq j \]
\[ \{ P \land B, \land Q \} \rightarrow S, T, \{ Q \} \quad \text{for all } i \text{ and } Q \]
\[ \{ P \land (\forall i : (\neg C_i)) \land Q \} \rightarrow B, \rightarrow S, \od ; \od C, \rightarrow T, \od \{ Q \} \quad \text{for all } Q \]

2. Constructing traversals from a tree

In order to demonstrate the usefulness of the proof rules stated in the previous section we set out to solve the problem of constructing a binary tree from its preorder and inorder traversals. In this section we derive a program that constructs the preorder and inorder traversals of a binary tree. In the next section we invert that program so as to solve the given problem. It turns out to be much easier to derive a program to construct traversals from a tree rather than the other way around. This again shows the usefulness of program inversion.

We use the definition of a binary tree and of its traversals from [5].

Definition 2.1 (Sequences). Catenation of sequences is denoted by juxtaposition. The empty sequence is denoted by \( \varepsilon \). The first element of a non-empty sequence \( s \) is denoted by \( \text{hd} \cdot s \) and its last element by \( \text{last} \cdot s \). The sequence constructed by deleting the first element of a non-empty sequence \( s \) is denoted by \( \text{tl} \cdot s \) and the one constructed by deleting its last element by \( \text{fr} \cdot s \).

Definition 2.2 (Binary tree). A finite (labeled) binary tree is empty, denoted by \( \emptyset \), or is a triple \((l, d, r)\) constructed from finite binary trees \( l \) and \( r \) and label \( d \). We denote the left subtree of non-empty tree \( u \) by \( u.l \), its label by \( u.d \), and its right subtree by \( u.r \).

Definition 2.3 (Preorder and inorder traversal). The preorder traversal, \( \text{pre} \cdot t \), of a finite binary tree \( t \) is \( \varepsilon \) if \( t = \emptyset \) and \( d \cdot \text{pre} \cdot l \cdot \text{pre} \cdot r \) if \( t = (l, d, r) \). The inorder traversal, \( \text{in} \cdot t \), is \( \varepsilon \) if \( t = \emptyset \) and \( \text{in} \cdot l \cdot \text{in} \cdot r \) if \( t = (l, d, r) \).

We derive an algorithm for inorder and preorder traversals, given a binary tree, in the usual way by starting with a postcondition \( R \) and stipulating an invariant \( P \). We follow the derivation of [5]. The reader who is already familiar with this algorithm can skip to page 6, where we strengthen the invariant slightly so as to allow the program to be inverted, or to the end of this section, where the final and fully annotated program is given.

For \( t \) a binary tree, the postcondition of a program to compute \( \text{pre} \cdot t \) and \( \text{in} \cdot t \) from \( t \) is

\[ R: \quad \text{pre} \cdot t = p \land \text{in} \cdot t = q. \]
A first attempt to obtain an invariant would be to replace $p$ with $p\text{pre}.t$ and $q$ with $q\text{in}.t$, so that $p$ and $q$ can be initialized to $\varepsilon$, and then replace $t$ in the right-hand sides of the equalities with program variable $u$. This leads to an invariant of the form

$$p\text{pre}.t = p\text{pre}.u \land q\text{in}.u = q\text{in}.u,$$

which is easily established by $p,q,u := \varepsilon,\varepsilon,t$. When we expand $u$ in the invariant, assuming $u = (l,d,r)$ for some trees $l$ and $r$ and label $d$, we find

$$p\text{pre}.t = p d\text{pre}.l \text{pre}.r \land q\text{in}.l d\text{in}.r,$$

which is of another form than that of the invariant. Tree $l$ can be associated with $u$ in the invariant, but $\text{pre}.r$ and $d\text{in}.r$ are new terms. Expanding $l$ once more, assuming $l \neq \emptyset$, reveals that the new terms are in fact catenations of preorder and inorder traversals of certain trees. Therefore, we propose the following invariant $P$:

$$P: \quad p\text{pre}.t = p\text{pre}.u x.S \land q\text{in}.u y.S,$$

where $x$ and $y$ are defined on sequences of pairs. Each pair consists of a label and a finite binary tree. For such a pair $s$, we denote its label by $s.d$ and its tree by $s.r$.

$$x.\varepsilon = y.\varepsilon = \varepsilon,$$

and for $d$ a label and $r$ a finite binary tree

$$x.((d,r) S) = \text{pre}.r x.S,$$
$$y.((d,r) S) = d\text{in}.r y.S.$$

This invariant is established by $p,q,u,S := \varepsilon,\varepsilon,t,\varepsilon$. We now develop the body of the repetition by manipulating $P$ so as to find an assignment statement that gets us closer to the postcondition. Under the condition that $u = (l,d,r)$ we derive

$$P$$

$$= \{\text{definition of } P, \text{ using } u = (l,d,r)\}$$

$$= \text{pre}.t = p\text{pre}.(l,d,r) x.S \land q\text{in}.(l,d,r) y.S$$

$$= \{\text{definitions of } \text{pre} \text{ and } \text{in}\}$$

$$= \text{pre}.t = p d\text{pre}.l \text{pre}.r x.S \land q\text{in}.l d\text{in}.r y.S$$

$$= \{\text{definitions of } x \text{ and } y\}$$

$$= \text{pre}.t = p d\text{pre}.l x.((d,r) S) \land q\text{in}.l y.((d,r) S)$$

$$= \{\text{definition of substitution}\}$$

$$p, u, S := p u.d, u.l.((u.d, u.r) S)$$

Hence, the assignment $p, u, S := p u.d, u.l.((u.d, u.r) S)$ maintains $P$, provided that $u \neq \emptyset$. 
Moreover, the length of $p$ increases by this statement. Under the condition $u = \emptyset \land S = (d, r) \cup$ we derive

$$P$$

$$= \{ \text{definition of } P, \text{ using that } u = \emptyset \}$$

$$\text{pre}_t = p \times S \land \text{in}_t = q \times S$$

$$= \{ \text{definitions of } x \text{ and } y, \text{ using that } S = (d, r) \cup \}$$

$$\text{pre}_t = p \\text{pre}_r \times U \land \text{in}_t = q \times d \times r \times U$$

$$= \{ \text{definition of substitution} \}$$

$$P_{q,u,S}$$

Hence, the assignment $q, u, S := q \times d, t, S$ maintains $P$, provided that $u = \emptyset \land S \neq \varepsilon$. Moreover, the length of $q$ increases by this statement. Therefore, the following program has been proved to compute the preorder and inorder traversals of a finite binary tree.

$$p, q, u, S := \varepsilon, \varepsilon, t, \varepsilon,$$

$$\text{do } u \neq \emptyset \lor S \neq \varepsilon \rightarrow$$

$$\text{if } u \neq \emptyset \rightarrow p, u, S := p \times u, u, l, (u, d, u, r) \times S$$

$$\square u = \emptyset \rightarrow q, u, S := q \times d, t, S \times S$$

$$\text{od } \{ p = \text{pre}_r \land q = \text{in}_t \land S = \varepsilon \land u = \emptyset \}$$

We intend to invert this repetition. From the proof rule for the inversion of a repetition we see that we have to find a predicate $C$ that holds after each iteration, but that does not hold initially. Moreover, we have to invert the body. We start with the inversion of the body, in this case an alternative statement. In order to invert an alternative statement we need to find mutually exclusive postconditions of its guarded commands. Examination of the above program reveals that $\text{last}_p = \text{hd}_S \times d$ after the first guarded command. It would be nice if we could conclude that this does not hold after the second guarded command. In order to be able to do so we strengthen the invariant slightly.

From the first guarded command we infer that the labels of the elements of $S$ are appended to $p$ when they are prepended to $S$. Therefore, the labels of the elements of $S$ occur in $p$ in reverse order. More formally, we define the function $z$ on sequences of pairs as the sequence of labels of those pairs in reverse order

$$z.\varepsilon = \varepsilon,$$

and for label $d$, tree $r$, and sequence $S$

$$z.((d, r) \times S) = z.\times d.$$

Now we add to our invariant $P$ the predicate

$$z.\times S \subseteq p,$$

where $a \subseteq b$ means that $a$ is a subsequence of $b$. It is obvious that this predicate holds initially and that removing an element of $S$ or adding an element to $p$ maintains it. Prefixing $S$ with a pair maintains this also if at the same time $p$ is extended (to
the right) with that pair's label. This is exactly what happens in the two guarded commands. Hence, $P$ is still an invariant.

Next we investigate the postcondition of the second guarded command. We assume for the time being that $S$ contains at least two elements. In the second guarded command the first pair of $S$ is removed and $p$ does not change. We want to conclude as a postcondition of this statement that $\text{last.p} \neq \text{hd.S.d}$. (Notice that $\text{last.p}$ is well defined due to $z.S \subseteq p$ and $S \neq \varepsilon$.) Hence, as a precondition we need that the last element of $p$ and the label of the second element of $S$ are distinct. Given that $z.S \subseteq p$, this is implied if the last element of $p$ does not occur elsewhere in $p$, which is implied if we require the labels to be unique. From now on we assume the labels of the tree to be unique.

The second guard of the alternative statement guarantees $S$ to contain at least one element rather than two. The proper postcondition is therefore

$$S = \varepsilon \text{ cor } \text{hd.S.d} \neq \text{last.p}.$$  

In order to get rid of this cor we change the program a little bit, although this is not strictly necessary for the inversion. We replace $S$ with $T$ and we add to the invariant $T = S(\bot, \emptyset)$, where $\bot$ is a label not occurring in the tree. Then, the operations $\text{hd}$ and $\text{tl}$ on $S$ are the same operations on $T$ and the predicate $S \neq \varepsilon$ becomes $T \neq (\bot, \emptyset)$. Since $\bot$ is a fresh label, we also have that the postcondition

$$S = \varepsilon \text{ cor } \text{hd.S.d} \neq \text{last.p}$$

is equivalent to $\text{hd.T.d} \neq \text{last.p}$. Variable $S$ has now become auxiliary and does no longer occur in the program, which operates on $T$ instead. The invariant, however, is still expressed using $S$.

As said earlier, in order to invert the repetition we do not only have to invert the body of the repetition, but we also have to find a predicate $C$ that does not hold initially and holds after each iteration. Obviously, we can take $p \neq \varepsilon \lor q \neq \varepsilon$ for $C$, since the length of $p$ or the length of $q$ increases in each iteration. This is equivalent to $p \neq \varepsilon$ on account of the invariant.

The final and fully annotated program that constructs the preorder and inorder traversals of a uniquely labeled binary tree is

$$p, q, u, T := \varepsilon, \varepsilon, t, (\bot, \emptyset) \{ P \land \neg(p \neq \varepsilon) \land u = t \}$$

$$\text{do } u \neq \emptyset \lor T \neq (\bot, \emptyset) \Rightarrow \{ P \land (u \neq \emptyset \lor T \neq (\bot, \emptyset)) \}$$

$$\text{if } u \neq \emptyset \Rightarrow p, u, T := p \cdot u \cdot d, u \cdot l, (u \cdot d, u \cdot r) T$$

$$\{ P \land \text{hd.T.d} = \text{last.p} \}$$

$$\Box u = \emptyset \Rightarrow q, u, T := q \cdot \text{hd.T.d}, \text{hd.T.r}, \text{tl.T}$$

$$\{ P \land \text{hd.T.d} \neq \text{last.p} \}$$

$$\text{fi } \{ P \land p \neq \varepsilon \}$$

$$\text{od } \{ p = \text{pre.t} \land q = \text{in.t} \land T = (\bot, \emptyset) \land u = \emptyset \}$$

with invariant $P$:

$$\text{pre.t} = p \cdot \text{pre.u} \cdot x \cdot S \land \text{in.t} = q \cdot \text{in.u} \cdot y \cdot S \land$$

$$z.S \subseteq p \land T = S(\bot, \emptyset).$$
3. The program inverted

We now derive a program that, given the preorder and inorder traversals of a uniquely labeled binary tree, constructs that tree. We do so by program inversion, using the proof rules of Section 1.

Before actually inverting the repetition, we argue that the inverted program solves the problem of constructing a binary tree from its preorder and inorder traversals. Let \( t \) be that binary tree. For \( Q \) in the conclusion of the proof rule for a repetition we take \( u = t \), which is a precondition of the repetition above and exactly the condition we want to end up with. From the definition of inversion we cannot immediately conclude anything for the inverse in isolation. A closer look at the above repetition, however, reveals that any state satisfying the postcondition can be viewed as the result of an execution of the repetition. The program terminates and the postcondition is a one-point predicate. (In other words, the postcondition is the strongest postcondition.) Therefore, the inverse of the above repetition, provided it exists, ends in a state \( u = t \) when it starts in a state satisfying

\[
p = \text{pre}.t \land q = \text{in}.t \land T = (\perp, \emptyset) \land u = \emptyset.
\]

Hence, the inverse solves the problem and its initialization is

\[
p, q, u, T := \text{pre}.t, \text{in}.t, 0, (\perp, \emptyset).
\]

We now invert the repetition of the previous section. On account of the proof rule for inverting a repetition we should choose as guard of the inverted repetition a predicate that holds after each iteration and that does not hold initially. From the annotation we infer that \( p \neq \epsilon \) satisfies that condition. The only remaining obligation is to find the body of the inverted repetition, i.e. a statement \( E \) such that

\[
\{ P \land (u \neq \emptyset \lor T \neq (\perp, \emptyset)) \land Q \} \text{D;} E \{ Q \}
\]

for all predicates \( Q \), where \( D \) is the if-statement of the above repetition.

Looking at the proof rule for inversion of an alternative command, we choose for \( E \) an alternative statement with guards \( \text{hd}.T.d = \text{last}.p \) and its negation, which are mutually exclusive postconditions of the above if-statement. From the same proof rule and some predicate calculus we infer that the remaining obligation is to find statements \( T_0 \) and \( T_1 \) such that

\[
\{ P \land u \neq \emptyset \land Q \} S_0; T_0 \{ Q \}
\]

and

\[
\{ P \land T \neq (\perp, \emptyset) \land u = \emptyset \land Q \} S_1; T_1 \{ Q \}
\]

for all predicates \( Q \), where \( S_0 \) and \( S_1 \) are the two guarded commands of the if-statement.
Statements $S_0$ and $S_1$ are simple assignment statements, which can easily be inverted. Statement $T_0$ becomes

$$p, u, T := \text{fr}.p, (u, \text{last}.p, \text{hd}.T.r), \text{tl}.T$$

and statement $T_1$ becomes

$$q, u, T := \text{fr}.q, 0, (\text{last}.q, u) T.$$ 

We apply the proof rule only for the latter of these two. Left to prove is

$$(\text{def}(\text{fr}.q, 0, (\text{last}.q, u) T))^{q,u,T}_{q(\text{hd}.T.d), \text{hd}.T.r, \text{tl}.T}$$

and

$$q, u, T = (\text{fr}.q, 0, (\text{last}.q, u) T)^{q,u,T}_{q(\text{hd}.T.d), \text{hd}.T.r, \text{tl}.T}$$

given that

$$P \land T \neq (\bot, 0) \land u = 0.$$ 

Straightforward substitution yields the result desired.

This concludes the derivation of the inverse program. The final program is

$$p, q, u, T := \text{pre}.t, \text{in}.t, 0, (\bot, 0);$$
$$\text{do } p \neq \varepsilon \rightarrow$$
$$\quad \text{if } \text{hd}.T.d = \text{last}.p \rightarrow p, u, T := \text{fr}.p, (u, \text{last}.p, \text{hd}.T.r), \text{tl}.T$$
$$\quad \square \text{hd}.T.d \neq \text{last}.p \rightarrow q, u, T := \text{fr}.q, 0, (\text{last}.q, u) T$$
$$\quad \text{fi}$$
$$\text{od } \{u = t\}$$

4. Concluding remarks

We have given proof rules for program inversion. Subsequently, we have used the rules to derive an algorithm for the construction of a uniquely labeled binary tree from its preorder and inorder traversals. This program can, of course, also be proved by the more common proof technique of an invariant and a variant function. Predicate $P$, the invariant of the first program is, of course, also maintained by the inverted program, which exactly retraces the steps of the first one. Hence, one might ask why to use the proof rules for program inversion, other than for the fun of it, instead of proving $P$ to be an invariant of the program. We invite the reader to prove that $P$ is invariant of the inverted program in the usual way to discover the answer. It is not clear at all why $z.S \subseteq p$ is invariant under the second of the two guarded commands. One way out is to strengthen this part of the invariant so as to express more precisely how $z.S$ is embedded in $p$. This is done in [7], where two more program variables are used to formulate the stronger invariant. In [6], where the inverted program is very similar to ours, a function is defined that basically builds a stack in order to show that the inverse exactly retraces the steps of the first
program. That proof is quite awkward and has nothing to do with the problem of tree traversal. The correctness of the inverted program follows from a programming principle rather than from properties of the problem at hand. Therefore, we believe that program inversion, with its proof rules, provides a supplementary and useful technique for program derivation.

Appendix A. Proofs of the proof rules

In this appendix we prove the four proof rules of Section 1.

Proof Rule for the Assignment Statement

\[ P \Rightarrow \text{def}(E_0) \land \text{def}(E_1)) \Rightarrow \text{def}(x = (E_1)_{E_0}) \]
\[
\{ P \land Q \} x := E_0 ; x := E_1 \{ Q \} \quad \text{for all } Q
\]

Proof.

\[ wp("x := E_0 ; x := E_1", Q) \]
\[ = \quad \{ \text{rule for sequential composition} \} \]
\[ wp("x := E_0", wp("x := E_1", Q)) \]
\[ = \quad \{ \text{rule for the assignment statement} \} \]
\[ wp("x := E_0", \text{def}(E_1) \land Q_{E_1}) \]
\[ = \quad \{ \text{rule for the assignment statement} \} \]
\[ \text{def}(E_0) \land (\text{def}(E_1) \land Q_{E_1})_{E_0} \]
\[ = \quad \{ \text{calculus} \} \]
\[ \text{def}(E_0) \land (\text{def}(E_1))_{E_0} \land Q_{E_1} \]
\[ \Leftarrow \quad \{ \text{premise} \} \]
\[ P \land Q \quad \square \]

Proof Rule for Sequential Composition

\[ \{ P \} S_0 \{ R \} \]
\[ \{ P \land Q \} S_0 ; S_i ; T_0 \{ Q \} \quad \text{for all } Q \]
\[ \{ R \land Q \} S_i ; T_1 \{ Q \} \quad \text{for all } Q \]
\[ \{ P \land Q \} S_0 ; S_i ; T_1 ; T_0 \{ Q \} \quad \text{for all } Q \]

Proof. Let Q be a predicate. We may assume R_0 to be such that

\[ \{ P \land Q \} S_0 \{ R_0 \} \quad \text{and} \quad \{ R_0 \} T_0 \{ Q \} \quad (0) \]
on account of the second premise. Moreover, we assume $R_1$ to be such that

\[ \{ R \land R_0 \} S_i \{ R_i \} \quad \text{and} \quad \{ R_1 \} T_i \{ R_0 \} \]  

(1)

on account of the third premise. Now we derive

\[ wp(\text{"}S_0; S_i; T_i; T_0\text{"}, Q) \]

\[ = \{ \text{rule for sequential composition} \}
\]

\[ wp(S_0, wp(S_i, wp(T_i, wp(T_0, Q)))) \]

\[ \Leftarrow \{ wp \text{ is monotonic and } \{ R_0 \} T_0 \{ Q \} \text{ on account of (0)} \}
\]

\[ wp(S_0, wp(S_i, wp(T_i, R_0))) \]

\[ \Leftarrow \{ wp \text{ is monotonic and } \{ R_1 \} T_i \{ R_0 \} \text{ on account of (1)} \}
\]

\[ wp(S_0, wp(S_i, R_1)) \]

\[ \Leftarrow \{ wp \text{ is monotonic and } \{ R \land R_0 \} S_i \{ R_i \} \text{ on account of (1)} \}
\]

\[ wp(S_0, R \land R_0) \]

\[ = \{ wp \text{ is conjunctive} \}
\]

\[ wp(S_0, R) \land wp(S_0, R_0) \]

\[ \Leftarrow \{ (0) \text{ and the first premise} \}
\]

\[ P \land Q \quad \square \]

**Proof Rule for the Alternative Statement**

\[
\begin{align*}
\{ P \land B_i \} & S_i \{ C_i \} \quad \text{for all } i \\
\{ P \land B_i \} & S_i \{ \neg C_j \} \quad \text{for all } i \text{ and } j \text{ with } i \neq j \\
P & \Rightarrow (\exists i :: B_i) \\
\{ P \land B_i \land Q \} & S_i ; T_i \{ Q \} \quad \text{for all } i \text{ and } Q \\
\{ P \land Q \} & \text{if } B_i \Rightarrow S_i ; \text{if } C_i \Rightarrow T_i ; \text{if } \{ Q \} \quad \text{for all } Q
\end{align*}
\]

**Proof.** We abbreviate if $B_i \Rightarrow S_i$ by $if_b$ and if $C_i \Rightarrow T_i$ by $if_c$. Let $Q$ be a predicate. We assume $P \land Q$ and prove $wp(\text{"}if_b \text{;} if_c\text{"}, Q)$. From the first and third premise and the assumption $P$ we infer

\[ wp(if_b, (\exists i :: C_i)) \]

(0)

and from assumption $P$ and the second premise

\[ B_i \Rightarrow wp(S_i, \neg C_j) \quad \text{for all } i \neq j. \]

(1)
We derive

\[ wp(\text{"if } b \text{ : if } c \text{"}, Q) \]
\[ = \{ \text{rule for sequential composition} \} \]
\[ wp(\text{if } b, wp(\text{if } c, Q)) \]
\[ = \{ \text{rule for the alternative statement} \} \]
\[ wp(\text{if } b, (\exists j :: C_j) \land (\forall j :: C_j \Rightarrow wp(T_j, Q))) \]
\[ = \{ wp \text{ is conjunctive} \} \]
\[ wp(\text{if } b, (\exists j :: C_j) \land (\forall j :: wp(\text{if } b, C_j \Rightarrow wp(T_j, Q)))) \]
\[ = \{ (0) \text{ and the rule for the alternative statement} \} \]
\[ (\forall j :: (\exists i :: B_i) \land (\forall i :: B_i \Rightarrow wp(S_i, C_i \Rightarrow wp(T_i, Q)))) \]
\[ = \{ \text{third premise, using that } P \text{ holds by assumption} \} \]
\[ (\forall i,j :: B_i \Rightarrow wp(S_i, C_j \Rightarrow wp(T_i, Q))) \]
\[ \Rightarrow \{ \text{monotonicity of } wp \text{ and predicate calculus} \} \]
\[ (\forall i,j :: B_i \Rightarrow wp(S_i, \neg C_j) \lor wp(S_i, wp(T_i, Q))) \]
\[ \Rightarrow \{ (1) \} \]
\[ (\forall i :: B_i \Rightarrow wp(S_i, wp(T_i, Q))) \]
\[ = \{ \text{rule for sequential composition} \} \]
\[ (\forall i :: B_i \Rightarrow wp(S_i, T_i, Q)) \]
\[ = \{ \text{last premise, using assumption } P \land Q \} \]
\[ true \]

Proof Rule for the Repetitive Construct

\[ \{ P \land (\forall i :: \neg C_i) \} \text{do } B_i \rightarrow S_i \text{ od } \{ true \} \]
\[ \{ P \land B_i \} S_i \{ P \land C_i \} \text{ for all } i \]
\[ \{ P \land B_i \} S_i \{ \neg C_i \} \text{ for all } i \text{ and } j \text{ with } i \neq j \]
\[ \{ P \land B_i \land Q \} S_i ; T_i \{ Q \} \text{ for all } i \text{ and } Q \]
\[ \{ P \land (\forall i :: \neg C_i) \land Q \} \text{do } B_i \rightarrow S_i \text{ od ; do } C_i \rightarrow T_i \text{ od } \{ Q \} \text{ for all } Q \]

Proof. Let \( Q \) be a predicate. We abbreviate \( wp(\text{"do } C_i \rightarrow T_i \text{ od"}, Q) \) by \( WP \). We show that \( WP \) holds after termination of the first repetition. We do so by showing that it holds initially and that it is kept invariant. This is sufficient since the first premise and the precondition \( \{ P \land (\forall i :: \neg C_i) \land Q \} \) guarantee the first repetition to terminate. We have the following relation for \( WP \), obtained by unfolding.

\[ WP = ((\exists i :: C_i) \lor Q) \land (\forall i :: \neg C_i \lor wp(T_i, WP)) \tag{0} \]
From (0) we immediately infer that \( P \land (\forall i :: \neg C_i) \land Q \) implies \( WP \), so that \( WP \) holds initially. Furthermore, we derive for any \( i \)

\[
wp(S_i, WP) \\
\iff \{ (0), \text{using the monotonicity of } wp \} \\
wp(S_i, (\exists j :: C_j) \land (\forall j :: \neg C_j \lor wp(T_j, WP))) \\
= \{ wp \text{ is conjunctive} \} \\
wp(S_i, (\exists j :: C_j)) \land (\forall j :: wp(S_i, \neg C_j \lor wp(T_j, WP))) \\
\iff \{ \text{second premise, monotonicity of } wp \text{ and predicate calculus} \} \\
P \land B_i \land (\forall j :: wp(S_i, \neg C_j) \lor wp(S_i, wp(T_j, WP))) \\
\iff \{ \text{third premise, rule for sequential composition} \} \\
P \land B_i \land wp("S_i; T_i", WP) \\
\iff \{ \text{last premise, taking } WP \text{ for } Q \} \\
P \land B_i \land WP \quad \square

Acknowledgement

We are grateful to Jan L.A. van de Snepscheut for redrawing our attention to the problem of constructing a tree from its preorder and inorder traversals. Moreover, he suggested that one could try to solve this problem by bridling the nondeterminism in the program to construct a tree from its inorder traversal, which David Gries and he derived by a process of inversion [5]. We owe the proof of the proof rule for inversion of a repetition to Wim H. Hesselink. It reduces the length of our original proof dramatically. Thanks are also due to Netty van Gasteren, Wim H. Hesselink, Anne Kaldewaij, and David Gries for comments upon earlier versions of this paper.

References