Under consideration for publication in Formal Aspects of Computing

Assumption Propagation through Annotated Programs

Dipak L. Chaudhari and Om Damani

Indian Institute of Technology, Bombay, India

Abstract.

In the correct-by-construction programming methodology, programs are incrementally derived from their formal specifications, by repeatedly applying transformations to partially derived programs. At an intermediate stage in a derivation, users may have to make certain assumptions to proceed further. To ensure that the assumptions hold true at that point in the program, certain other assumptions may need to be introduced upstream as loop invariants or preconditions. Typically these other assumptions are made in an ad hoc fashion and may result in unnecessary rework, or worse, complete exclusion of some of the alternative solutions. In this work, we present rules for propagating assumptions through annotated programs. We show how these rules can be integrated in a top-down derivation methodology to provide a systematic approach for propagating the assumptions, materializing them with executable statements at a place different from the place of introduction, and strengthening of loop invariants with minimal additional proof efforts.

Keywords: Assumption Propagation, Annotated Programs, Program Derivation, Correct-by-construction

1. Introduction

In the correct-by-construction style of programming [Dij76, Kal90, Gri87], programs are systematically derived from their formal specifications in a top-down manner. At each step, a derivation rule is applied to a partially derived program at hand, finally resulting in the fully derived program. The refinement calculus [Mor90, BvW98] further formalizes this top-down derivation approach.

Such refinement-based program derivation systems provide a set of formally verified transformation rules. At an intermediate stage in a top down derivation, users may have to make certain assumptions to proceed further. To ensure that the assumptions hold true at that point in the program, certain other assumptions may need to be introduced upstream as loop invariants or preconditions. Typically these other assumptions are made in an ad hoc fashion. It is not always possible to come up with the right predicates on the first

Correspondence and offprint requests to: Dipak L. Chaudhari, Kresit, Indian Institute of Technology, Bombay, Mumbai, 400076, India, e-mail: dipakc@cse.iitb.ac.in

This paper is an extended version of: Dipak L. Chaudhari and Om P. Damani. Combining top-down and bottom-up techniques in program derivation. In Logic-Based Program Synthesis and Transformation - 25th International Symposium, LOPSTR 2015, Lecture Notes in Computer Science, vol 9527, pp 244-258.[CD15].

attempt. Users often need to backtrack and try out different possibilities. The failed attempts, however, often provide added insight which help, to some extent, in deciding the future course of action. In the words of Morgan [Mor90]: *"excursions like the above ... are not fruitless...we have discovered that we need the extra conjunct in the precondition, and so we simply place it in the invariant and try again."* Although the failed attempts are *not fruitless*, and provide some insight, the learnings from these attempts may not be directly applicable; some guesswork is still needed to determine the location for the required modifications and the exact modifications to be made. For example, as we will see in the next section, simply strengthening the loop invariant with the predicate required for the derivation of the loop body might not always work. Moreover, the *trying again* results in rework. The derived program fragments (and the discharged proof obligations) need to be recalculated (redischarged) during the next attempt. The failed attempts also break the flow of the derivations and make them difficult to organize.

Tools supporting the refinement-based formal program derivation (Cocktail [Fra99], Refine [OXC04], Refinement Calculator [BL96] and PRT [CHN+96]) mostly follow the top-down methodology. Not much emphasis has been given on avoiding the unnecessary backtrackings. The refinement strategies cataloged by these tools help to some extent in avoiding the common pitfalls. However, a general framework for allowing users to assume predicates and propagating them to appropriate locations is missing.

In this context, we make the following contributions in this paper:

1. We discuss the problems resulting from ad hoc reasoning involved in propagation of assumptions made during a top-down derivation of programs. To address these problems, we present correctness preserving rules for propagating assumptions through annotated programs.
2. We show how these rules can be integrated in a top-down derivation methodology to provide a systematic approach for propagating the assumptions, materializing them with executable statements at a place different from the place of introduction, and strengthening of loop invariants/preconditions with minimal additional proof efforts.
3. We have implemented these rules in the CAPS[1] system[CD14]. With the help of examples, we demonstrate how these rules help users in avoiding unnecessary rework and also help them explore alternative solutions.

2. Preliminaries

In this section, we present some of the basic definitions and general notations used in the paper.

2.1. Hoare Triple, Weakest Precondition, and Strongest Postcondition

For program S and predicates P and Q, a *Hoare triple* [Hoa69], denoted as $\{P\}\, S\, \{Q\}$, is a boolean that has the value *true* if and only if every terminating execution of program S starting in a state satisfying predicate P terminates in a final state satisfying predicate Q. This notion of correctness is called *partial correctness* since termination is not guaranteed.

The *weakest precondition* of S with respect to Q, denoted as $wp(S, Q)$, is the weakest predicate P for which $\{P\}\, S\, \{Q\}$ holds[Dij76]. More formally $\{P\}\, S\, \{Q\} \equiv [P \Rightarrow wp(S, Q)]$ where the square brackets denote universal quantification over the points in state space[DS90]. The notation $[R]$ is an abbreviation for $(\forall x_1 \ldots \forall x_n\ R)$ where $x_1, \ldots, x_n$ are the program variables in predicate R. The *weakest precondition* for the multiple assignment $x, y := E, F$ is defined as:

$$wp((x, y := E, F), Q) \equiv Q(x, y := E, F)$$

Here, the expression $Q(x, y := E, F)$ denotes a predicate obtained by substituting E and F for the free occurrences of variables x and y in the predicate Q.

Dual to the notion of weakest precondition is the notion of *strongest postcondition* [Gri87, DS90]. The strongest postcondition of program S with respect to predicate P, denoted as $sp(S, P)$, is the strongest predicate Q for which $\{P\}\, S\, \{Q\}$ holds.

[1] The CAPS system is available at `http://www.cse.iitb.ac.in/~damani/CAPS`

2.2. Eindhoven Notation

For representing quantified expressions, we use the *Eindhoven notation* [Dij75, BM06] $(OP\ i:\ R:\ T)$ where OP is the quantifier version of a symmetric and associative binary operator op, i is a list of quantified variables, R is the *Range* - a boolean expression involving the quantified variables, and T is the *Term* - an expression. For example, the expression $\sum_{i=0}^{10} i^2$ in the conventional notation is expressed as $\left(\sum\ i:\ 0 \leq i \leq 10:\ i^2\right)$ in the Eindhoven notation. We also use the Eindhoven notation for the logical quantifiers ($\forall$ and $\exists$). For example, the expressions $\forall i\ R(i) \Rightarrow T(i)$ and $\exists i\ R(i) \wedge T(i)$ in the conventional notation are expressed as $(\forall i : R(i) : T(i))$ and $(\exists i : R(i) : T(i))$ respectively in the Eindhoven notation.

3. Motivating Example

To illustrate top-down derivations using annotated programs, and some of the ad hoc decision making involved in these derivations, we present a sketch of the derivation for the *maximum segment sum* program. The derivation sketch presented in this section is based on the derivations given in [Kal90] and [Coh90]. For readers unfamiliar with the area of calculational program derivation, we have presented a simpler derivation in Appendix A. For a thorough introduction to the area, we refer readers to the excellent textbooks [Kal90] and [Coh90].

3.1. *Maximum Segment Sum* Derivation

In the *Maximum Segment Sum* problem, we are required to compute the maximum of all the *segment sums* of a given integer array. A *segment sum* of an array segment is the sum of all the elements of the segment.

Fig. 1 depicts the derivation process for this program. We start the derivation by providing the formal specification (node A) of the program. In the postcondition of the program, the symbol Max denotes the quantifier version of the binary infix max operator in the Eindhoven notation. After inspecting the postcondition, we apply the *Replace Constant by a Variable*[Kal90] heuristic to replace the constant N with a fresh variable n as shown in node B. We follow the general guideline of adding bounds on the introduced variable n by adding a conjunct $P_1 : 0 \leq n \leq N$ to the postcondition. Although this conjunct looks redundant due to the existence of the stronger predicate $n = N$, it is used later and becomes part of the loop invariant. We then apply the *Take Conjuncts as Invariants*[Kal90] heuristic to select conjuncts P_0 and P_1 as invariants and the negation of the remaining conjunct $n = N$ as a guard of the *while* loop. We choose to traverse the array from left to right and envision an assignment $r, n := r', n+1$, where r' is a metavariable – a placeholder for an unknown program expression (a quantifier free program term). The partially derived program at this stage is shown in node C. To calculate the metavariable r', we now step into the proof obligation for the invariance of P_0 and try to manipulate the formula with the aim of finding a program expression for r'. After several formula transformations, we arrive at a formula $r' = r\ max\ Q(n+1)$ shown in node G where $Q(n)$ is defined as $(Max\ p : 0 \leq p \leq n : Sum.p.n)$, where $Sum.p.n \triangleq (\sum\ i : p \leq i < n : A[i])$.

At this point, we realize that we can not represent r' in terms of the existing program variables as the expression $Q(n+1)$ involves quantifiers. After analyzing the derivation, we speculate that if we introduce a fresh variable (say s) and maintain $s = Q(n)$ as an additional loop invariant then we might be able to express r' in terms of the program variables.

We backtrack to the program shown in node C, introduce a fresh variable s, and envision a *while* program with the strengthened invariant. This time, we are able to calculate r' as $r\ max\ s$ with the help of the newly introduced invariant $s = Q(n)$. After the calculation of r', we proceed further with the derivation of s' and arrive at the formula $s' = (s + A[n])\ max\ 0$ (node L to node M). To make this formula valid, we instantiate the metavariable s' with the expression $(s + A[n])\ max\ 0$. After substituting s' with the expression $(s + A[n])\ max\ 0$ in the program shown in node K, we arrive at the final program shown in node N.

3.2. Ad Hoc Decision Making

The above derivation involves two ad hoc decisions. First, at the time of introducing variable n, we also introduced the upper and lower bounds for n. While the upper bound $n \leq N$ is necessary to ensure that the

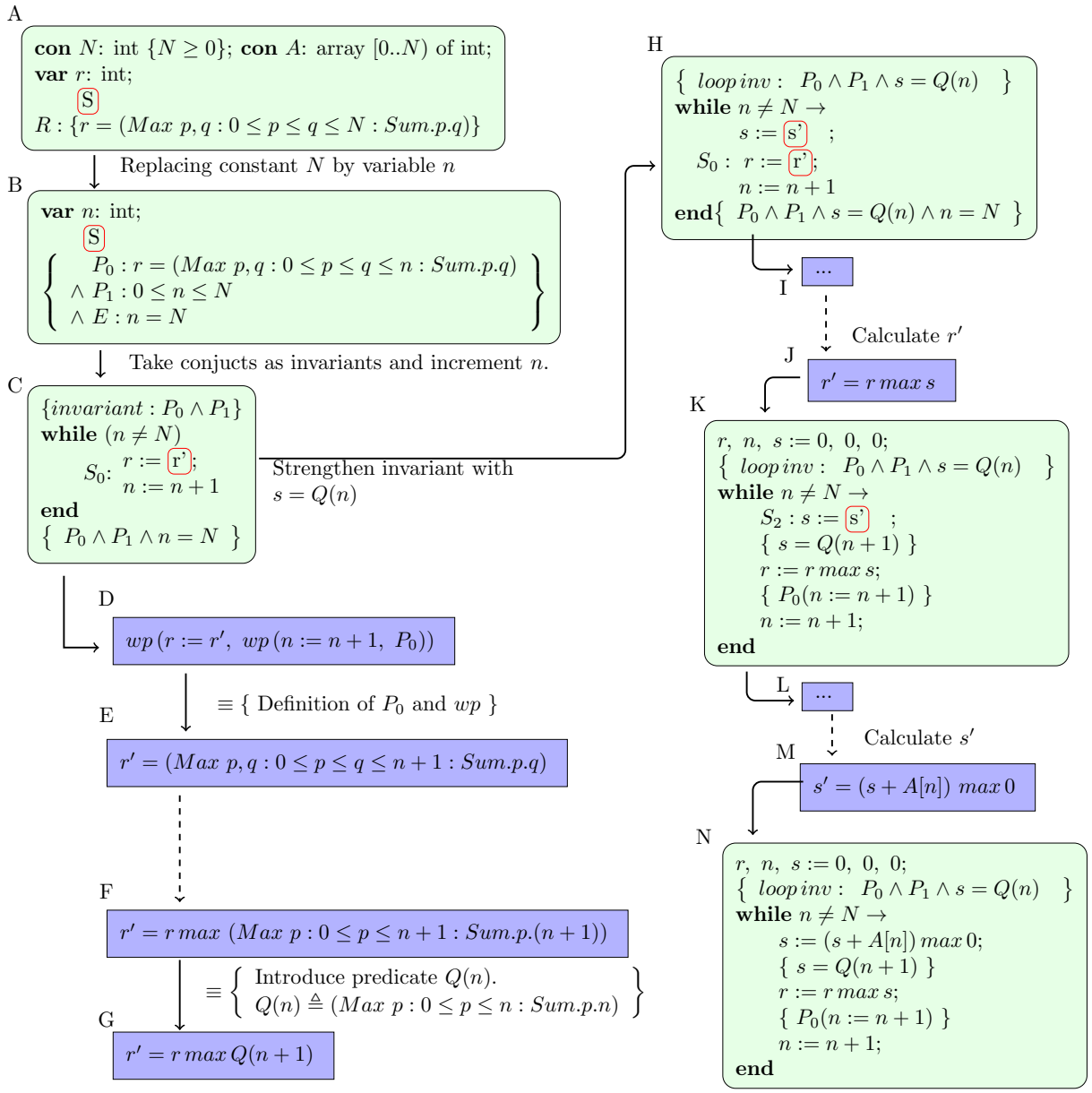


Fig. 1. Sketch of the top-down derivation of the *Maximum Segment Sum* problem.
$Sum.p.q \triangleq (\Sigma i : p \le i < q : A[i])$; $Q(n) \triangleq (Max\ p : 0 \le p \le n : Sum.p.n)$
$P_0 \triangleq (r = (Max\ p, q : 0 \le p \le q \le n : Sum.p.q))$; $P_1 \triangleq 0 \le n \le N$

expression P_0 is well-defined, at that point in derivation, there is no need to introduce the lower bound. The expression remains well-defined even for negative values of n.

The second ad hoc decision was that, we did not select $s = Q(n+1)$ as an invariant even though that is the formula which is required at node F. Instead we selected $s = Q(n)$ as an additional invariant. Selection of this formula needs a foresight that the occurrences of n are textually substituted by $n+1$ during the derivation (step D-E), so we will get the desired formula at node G, if we strengthen the invariant with $s = Q(n)$.

These ad hoc decisions result in the problem of rework and premature reduction of solution space.

$$
\begin{aligned}
\mathit{annGCL} ::= {} & \{\mathit{assertion}\}\ \mathit{program}\ \{\mathit{assertion}\} \\
\mathit{program} ::= {} & \mathbf{skip} \\
& \mid \mathbf{assume}(\mathit{assertion}) \\
& \mid \mathbf{unkprog} \\
& \mid \mathit{var}_1, \ldots, \mathit{var}_n := \mathit{expn}_1, \ldots, \mathit{expn}_n \\
& \mid \mathbf{if}\quad \mathit{bexpn}_1 \rightarrow \mathit{annGCL}_1 \quad [] \ldots [] \ \mathit{bexpn}_n \rightarrow \mathit{annGCL}_n \quad \mathbf{end} \\
& \mid \mathbf{while}\quad \{\text{inv:}\mathit{assertion}\} \quad \mathit{bexpn} \rightarrow \mathit{annGCL} \quad \mathbf{end} \\
& \mid \mathit{annGCL}_1;\ \ldots;\ \mathit{annGCL}_n
\end{aligned}
$$

Fig. 2. *annGCL* grammar

Rework. After backtracking to program C and strengthening invariant, we try to calculate r'. The steps from node I to node J correspond to the calculation of r'. These steps are similar to the calculation of r' in the failed attempt (node E to node F). We need to carry out these steps again to ensure that the newly added invariant does not violate the correctness of the existing program fragments.

Premature reduction of solution space. As shown later in Section 6.2, the above two decisions prematurely reduced the solution space preventing us from arriving at an alternative solution. The alternative solution (Fig. 22, node W) derived using the assumption propagation rules initializes n with -1 and uses an invariant involving the term $s = Q(n+1)$.

3.3. Motivation for Assumption Propagation

As discussed above, having made an arbitrary choice of introducing the invariant $0 \leq n$, when later faced with the problem of materializing the expression $s = Q(n+1)$, a loop invariant $s = Q(n)$ is introduced in an ad hoc fashion. The textbook by Cohen argues that *"The question might arise as to why the following was not chosen instead: $s = Q(n+1)$. The reason is that this invariant cannot be established initially...$A[0]$ is undefined when $N = 0$"* [2][Coh90]. Similarly, the textbook by Kaldewaij does not consider strengthening the invariant with $s = Q(n+1)$ on the ground that *"... for $n = N$ (which is not excluded by P_1) this predicate is not defined. Replacing all occurrences of n by $n-1$ yields an expression that is defined for all $0 \leq n \leq N$."*[Kal90]

We find two justifications for the exclusion of an invariant involving the term $s = Q(n+1)$; one on the ground of an initialization error while the other on the ground of a termination related error. Whereas the real problem lies in the fact that one is trying to make a guess without calculating the logically required expressions. The assumption propagation technique proposed in Section 5 enables the users to make assumptions in order to proceed and later propagate these assumptions to appropriate places where they can be materialized by introducing executable program constructs.

4. Program Derivation by Annotated Program Transformations

The program derivation methodology that we adopt is similar in spirit to the one followed in the motivating example. We start with the specification and incrementally transform it into a fully derived correct program. In this section, we introduce the concepts and notations relevant to the assumption propagation technique.

4.1. Annotated Programs

For representing a program fragment and its specification, we use an extension of the Guarded Command Language (GCL) [Dij75] called *annGCL*. It is obtained by augmenting each program construct in the GCL with its precondition and postcondition. The grammar for the *annGCL* language is given in Fig. 2.

[2] The notations in this quote have be adapted to match our notation.

Table 1. Partial correctness proof obligations for *annGCL* constructs

Annotated program (*annGCL*) $\mathcal{A}$	Correctness proof obligation of $\mathcal{A}$ $po\,(\mathcal{A})$
$\{\alpha\}$ **skip** $\{\beta\}$	$\alpha \Rightarrow \beta$
$\{\alpha\}$ **assume**(θ) $\{\beta\}$	$\alpha \wedge \theta \Rightarrow \beta$
$\{\alpha\}$ **unkprog** $\{\beta\}$	$true$
$\{\alpha\}$ $x_1, \ldots, x_n := E_1, \ldots, E_n$ $\{\beta\}$	$\alpha \Rightarrow \beta(x_1, \ldots, x_n := E_1, \ldots, E_n)$
$\{\alpha\}$ **if** $\mid\ G_1 \rightarrow \{\varphi_1\}\, S_1\, \{\psi_1\}$ $\mid\ \ldots$ $\mid\ G_n \rightarrow \{\varphi_n\}\, S_n\, \{\psi_n\}$ **end** $\{\beta\}$	$po_{coverage} \wedge po_{entry} \wedge po_{body} \wedge po_{exit}$ where, $po_{coverage}$: $\alpha \Rightarrow \bigvee_{i\in[1,n]} G_i$; po_{entry} : $\bigwedge_{i\in[1,n]} (\alpha \wedge G_i \Rightarrow \varphi_i)$; po_{body} : $\bigwedge_{i\in[1,n]} (po\,(\{\varphi_i\}\, S_i\, \{\psi_i\}))$; po_{exit} : $\bigwedge_{i\in[1,n]} (\psi_i \Rightarrow \beta)$
$\{\alpha\}$ **while** $\{\text{Inv: } \omega\}$ $G \rightarrow \{\varphi\}\ S\ \{\psi\}$ **end** $\{\beta\}$	$po_{init} \wedge po_{entry} \wedge po_{body} \wedge po_{inv} \wedge po_{exit}$ where, po_{init} : $\alpha \Rightarrow \omega$; po_{entry} : $\omega \wedge G \Rightarrow \varphi$; po_{body} : $po\,(\{\varphi\}\, S\, \{\psi\})$; po_{inv} : $\psi \Rightarrow \omega$; po_{exit} : $\omega \wedge \neg G \Rightarrow \beta$
$\{\alpha\}$ $\{\varphi_1\}\, S_1\, \{\psi_1\}$ $\ldots$ $\{\varphi_n\}\, S_n\, \{\psi_n\}$ $\{\beta\}$	$po_{entry} \wedge po_{body} \wedge po_{joins} \wedge po_{exit}$ where, po_{entry} : $\alpha \Rightarrow \varphi_1$; po_{body} : $\bigwedge_{i\in[1,n]} (po\,(\{\varphi_i\}\, S_i\, \{\psi_i\}))$; po_{joins} : $\bigwedge_{i\in[1,n-1]} (\psi_i \Rightarrow \varphi_{i+1})$; po_{exit} : $\psi_n \Rightarrow \beta$

For the sake of simplicity, we exclude variable declarations from the grammar. Also, the grammars for variables (*var*), expressions (*expn*), boolean expressions (*bexpn*), and assertions (*assertion*) are not described here. We use the formulas in sorted first-order predicate logic for expressing the assertions. We adopt the Eindhoven notation [BM06] for representing the quantified formulas. We have introduced program constructs *unkprog* and *assume* to represent unimplemented program fragments.

Note that in an *annGCL* program, all its subprograms (and not just the outermost program) are annotated with the pre- and postconditions.

Correctness of Annotated Programs

Definition 1 (Correctness of an annotated program). An *annGCL* program $\mathcal{A}$ is *correct* iff the proof obligation of $\mathcal{A}$ (denoted by $po(\mathcal{A})$ in Table 1) is *valid*.

The proof obligations for the newly introduced program constructs *unkprog* and *assume* deserve some explanation. The proof obligation for the *annGCL* program $\{\alpha\}\, unkprog\, \{\beta\}$ is $true$. In other words, *unkprog* is correct by definition and hence can represent any arbitrary unsynthesized program. The proof obligation for $\{\alpha\}\, assume(\theta)\, \{\beta\}$ is $\alpha \wedge \theta \Rightarrow \beta$. From this it follows that the program $\{\alpha\}\, assume(\theta)\, \{\alpha \wedge \theta\}$ is always correct. The *assume* program is used to represent an unsynthesized program fragment that preserves the precondition α while establishing θ.

The proof obligations of the composite constructs are defined inductively. The po_{body} proof obligation for the *if*, *while*, and *composition* constructs asserts the correctness of corresponding subprograms. We do not

use the Hoare triple notation for specifying correctness of programs since our notation for *annGCL* programs $\{\varphi\}\, S\, \{\psi\}$ conflicts with that of a Hoare triple. Instead, to express that an *annGCL* program $\mathcal{A}$ is correct, we explicitly state that "$po(\mathcal{A})$ is *valid*".

4.2. Transformation Rules

Definition 2 (Annotated program transformation rule). An annotated program transformation rule ($\mathcal{R}$) is a partial function from *annGCL* into itself which transforms a source *annGCL* program $\{\alpha\}\, S\, \{\beta\}$ to a target *annGCL* program $\{\alpha\}\, T\, \{\beta\}$ with the same precondition and postcondition.

Some of the transformation rules have associated *applicability conditions* (also called *proviso*). A rule can be applied only when the associated applicability condition is satisfied.

Definition 3 (Correctness preserving transformation rule). An annotated program transformation rule $\mathcal{R}$ is *correctness preserving* if for all the *annGCL* programs S for which the rule is applicable, if S is correct then $\mathcal{R}(S)$ is also correct.

Nature of the transformation rules. In the stepwise refinement-based approaches [Mor90, BvW98], a formal specification is incrementally transformed into a concrete program. A specification (pre- and post-conditions) is treated as an abstract program (called a specification statement). At any intermediate stage during the derivation, a program might contain specification statements as well as executable constructs. The traditional refinement rules are transformations that convert a specification statement into another program which may in turn contain specifications statements and the concrete constructs. In the conventional approach, once a specification statement is transformed into a concrete construct, its pre- and postconditions are not carried forward.

In contrast to the conventional approach, we maintain the specifications of all the subprogram (concrete as well as unsynthesized). This allows us to provide rules which transform any correct program (not just a specification statement) into another correct program with minimal proof effort. These rules reuse the already derived program fragments and the already discharged proof obligations to ensure correctness.

5. Assumption Propagation

5.1. Assumption Propagation for Bottom up Derivation

Assumption propagation can be seen as a bottom-up derivation approach since we delay certain decisions by making assumptions and then propagate the information upstream. In order to incorporate the bottom-up approach in a primarily top-down methodology, we need a way to accumulate assumptions made during the derivation and then to propagate these assumptions upstream. After propagating the assumptions to appropriate locations in the derived program, user can introduce appropriate program constructs to establish the assumptions.

This bottom-up phase has three main steps.

- **Assume:** To derive a program fragment with precondition α and postcondition β, we start with the *annGCL* program $\{\alpha\}\, unkprog_1\, \{\beta\}$. Now suppose that, in order to proceed further, we decide to assume θ.
  For example, we can envision a program construct in which the unknown program expressions are represented by metavariables. We then focus our attention on the correctness proof obligation of the envisioned program and try to guess suitable expressions for the metavariables with the objective of discharging the proof obligation. While doing so, we might need to assume θ.
  Instead of backtracking and figuring out the modifications to be done to the rest of the program to make θ hold at the point of assumption, we just accumulate the assumptions and proceed further with the derivation to arrive at program S. In the resulting *annGCL* program (Fig. 3), $assume(\theta)$ establishes the assumed predicate θ while preserving α. For brevity, we abbreviate the statement $assume(\theta)$ as $A(\theta)$.
- **Propagate:** We may not want to materialize the program to establish θ at the current program location. We can propagate the $assume(\theta)$ statement upstream to an appropriate program location. The assumed

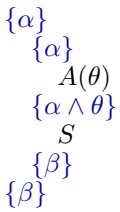


Fig. 3. Result of assuming precondition θ in the derivation of $\{\alpha\}\, unkprog_1\, \{\beta\}$

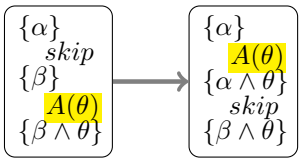


Fig. 4. *SkipUp* rule

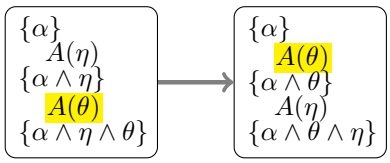


Fig. 5. *AssumeUp* rule

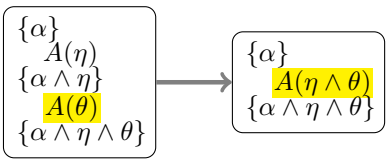


Fig. 6. *AssumeMerge* rule

predicate θ is modified appropriately depending on the program constructs through which it is propagated. The pre- and postconditions of the intermediate program constructs are also updated to preserve correctness.

- **Realize:** Once the *assume* statement is at a desired location, we can materialize it by deriving corresponding executable program constructs that establish the assumption. Note that this might not be a single step process. We might replace the *assume* statement with another partially derived program which might in turn have other *assume/unkprog* statements in addition to some executable constructs.

  We repeat the process till we eliminate all the *assume* and *unkprog* statements.

5.2. Precondition Exploration

We can propagate the assumptions made during the derivation all the way to the top. Let us say, we arrive at a program shown in Fig. 3. If the overall precondition of the program α implies the assumption θ then we can get rid of the *assume* statement and arrive at a program $\{\alpha\}\, S\, \{\beta\}$. If this is not the case, we can either go about materializing the assumption or accept the assumption θ as an additional precondition. So we now have an *annGCL* program $\{\alpha \wedge \theta\}\, S\, \{\beta\}$ which is a solution for a different specification whose precondition is stronger than the original precondition.

This could be called *precondition exploration*; where for the given precondition α and postcondition β, we would like to derive a program S and assumption θ such that the *annGCL* program $\{\alpha \wedge \theta\}\, S\, \{\beta\}$ is correct.

5.3. Rules for Propagating and Establishing Assumptions

The propagation step described in Section 5.1 is an important step in the bottom up phase. We have developed correctness preserving transformation rules for propagating the assumptions upstream through the *annGCL* program constructs. In the coming sections, we present transformation rules for assumption propagation.

5.3.1. Atomic Constructs

Atomic constructs are the program constructs that do not have subprograms. For every atomic construct, there is a rule for up-propagating an assumption through the construct. For atomic constructs that represent unsynthesized programs (*assume* and *unkprog*), there are additional rules for merging statements or establishing the assumptions.

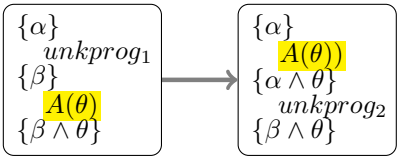


Fig. 7. *UnkProgUp* rule

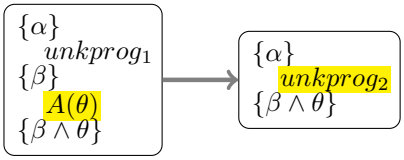


Fig. 8. *UnkProgEst* rule

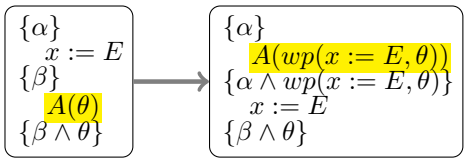


Fig. 9. *AssignmentUp* rule.

skip. The *SkipUp* rule (Fig. 4) propagates an assumption θ through a *skip* statement. No change is required in the assumed predicate as it is propagated through the *skip* statement.

assume. The *AssumeUp* rule (Fig. 5) propagates an assumption θ through an *assume* program $A(\eta)$. This transformation just changes the order of the *assume* statements. Instead of propagating the assumption θ, we can choose to merge it into the statement $A(\eta)$ by applying the *AssumeMerge* rule (Fig. 6). Applying this rule results in a single *assume* statement $A(\eta \wedge \theta)$.

unkprog. Fig. 7 shows the *UnkProgUp* rule which propagates an assumption upward through an unknown program fragment ($unkprog_1$). Note that pre- and post-conditions of $unkprog_2$ are strengthened with θ. Here, we are demanding that $unkprog_2$ should preserve θ. We may prefer to establish θ instead of propagating. The *UnkProgEst* rule (Fig. 8) can be used for this purpose.

assignment. Fig. 9 shows the *AssignmentUp* rule for propagating an assumption upwards through an assignment. The assumed predicate θ gets modified to $wp(x := E, \theta)$ as it is propagated through the assignment $x := E$.

5.3.2. Composition

We need to consider the following two cases for propagating assumptions through a *composition* program.

Case 1: The *assume* statement is immediately after the *composition* program.
Fig. 10 shows a *composition* program which is composed of another *composition* and an *assume(θ)* statement. The *CompositionIn* rule can be used to propagate the assumption θ inside the *composition* construct. The assumption can then be propagated upwards through the subprograms of the composition (S_n to S_1) using appropriate rules.

Case 2: The *assume* statement is the first statement in the *composition* program.
The *CompositionOut* rule (Fig. 11) propagates the *assume* statement at a location before the *composition* statement. The target program does not contain the predicate φ since, from the correctness of the source program, φ is implied by the precondition α.
We also provide the *CompoToIf* rule (Fig. 12) which establishes the assumption θ by introducing an *if* program in which the assumed predicate θ appears as the guard of the program. Another guarded

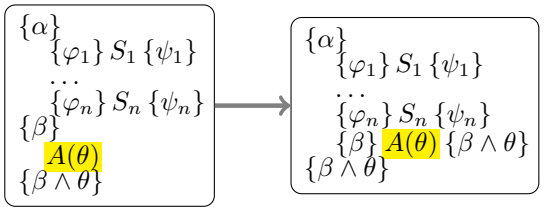


Fig. 10. *CompositionIn* rule

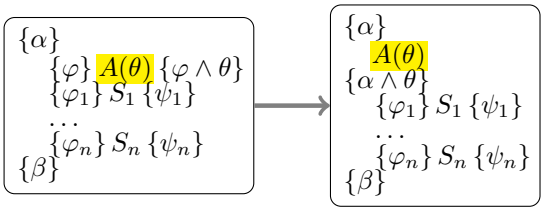


Fig. 11. *CompositionOut* rule

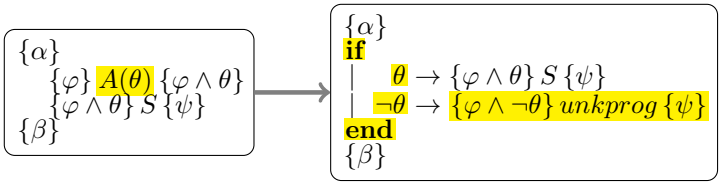


Fig. 12. *CompoToIf* rule: Transforms a *composition* to an *if* program.

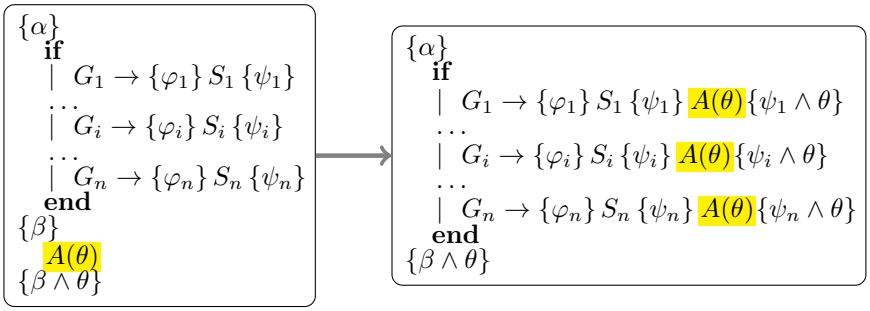


Fig. 13. *IfIn* rule.

command is added to handle the complementary case. This rule has a proviso that θ is a valid program expression. This rule allows users to delay the decision about the type of the program constructs. For example, users may envision an assignment, which can be turned later into an *if* program if required.

If the *assume* statement is at a location inside the composition program which does not fall under these two cases, then appropriate rule should be selected based on the type of the program immediately preceding the *assume* statement. Nested composition construct can be collapsed to form a single composition. However, this construct is useful when we want to apply a rule to a subcomposition.

5.3.3. If

We need to consider the following two cases for propagating assumptions through an *if* program.

Case 1: The *assume* statement is immediately after the *if* program.
In order to make predicate θ available after the *if* program, θ must hold after the execution of every guarded command. The *IfIn* rule (Fig. 13) propagates an *assume* statement that appears immediately after an *if* construct in the source program inside the *if* construct in the target program. In the target program, θ is assumed at the end of every guarded command.

Case 2: The *assume* statement is the first statement in the body of one of the guarded commands.
To make θ available as a precondition of the body of one of the guarded commands, θ must hold as a precondition of the *if* program. (In fact, we can assume a weaker predicate as discussed below.)

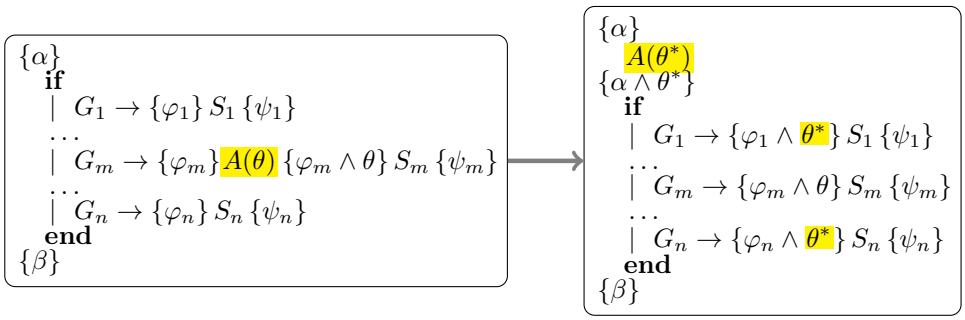


Fig. 14. *IfOut* rule. $\theta^* \triangleq G_m \Rightarrow \theta$

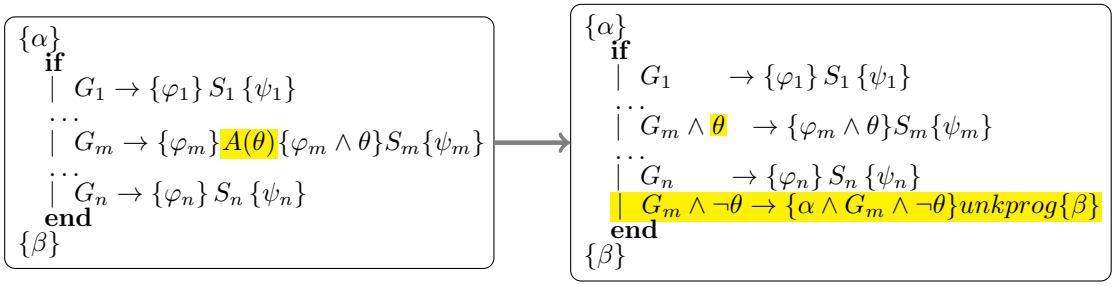


Fig. 15. *IfGrd* rule

Fig. 14 shows the *IfOut* rule corresponding to this case. In the source program, θ is assumed before the subprogram S_m, whereas in the target program, θ^* is assumed before the *if* program. Note that θ^* (which is defined as $(G_m \Rightarrow \theta)$) is weaker than θ. This weakening is possible since, by the semantics of the *if* construct, the guard predicate G_m is already available as a precondition to the program S_m.

As a result of assuming θ^* before the *if* construct, we also strengthen the precondition of the other guarded commands. Note that program fragments S_1 to S_n may contain yet to be synthesized *unkprog* fragments. This strengthening of the preconditions by θ^* might be helpful in the task of deriving these unknown program fragments.

In this case, instead of propagating assumption θ, we can make it available as a precondition to the body of the guarded command (S_m) by simply strengthening the corresponding guard with θ. The *IfGrd* rule (Fig. 15) can be applied for this purpose. Since we are strengthening the guard of one of the guarded commands (G_m) with θ, an additional guarded command (with a guard $G_m \wedge \neg\theta$) needs to be added to ensure that all the cases are handled.

5.3.4. While

The assumption propagation rules involving the *while* construct are more complex than those for the other constructs since strengthening an invariant strengthens the precondition as well as the postcondition of the loop body. Depending on the location of the *assume* statement with respect the *while* construct, we have the following two cases.

Case 1: The *assume* statement is immediately after the *while* program.
Fig. 16 shows the *WhileIn* rule applicable to this case. The source program has an assumption after the while loop. In order to propagate the assumption θ upward, we strengthen the invariant of the while loop with $\neg G \Rightarrow \theta$. This is the weakest formula that will assert θ after the while loop. We add an *assume* statement after the loop body to maintain the invariant and another *assume* statement before the loop to establish the invariant at the entry of the loop.

Case 2: The *assume* statement is the first statement in the body of *while* program.
There are two options available to the user in this case depending on whether the user chooses to strengthen the postcondition of the loop body by propagating the assumed predicate forward through the loop body. The rules corresponding to these two alternatives are given below.

***WhileStrInv* rule.** In the source program (Fig. 17), the predicate θ is assumed at the start of the loop body. To make θ valid at the start of the loop body S, we strengthen the invariant with $(G \Rightarrow \theta)$. An *assume* statement $A(G \Rightarrow \theta)$ is added after the loop body to ensure that invariant is preserved. Another *assume* statement is added before the while loop to establish the invariant at the entry of the loop.

***WhilePostStrInv* rule.** There are two steps in this rule (Fig. 18). In the first step, the postcondition of the program S is strengthened with θ^* which is the strongest postcondition of θ with respect to S. In the second step, the invariant of the while loop is strengthened with θ^*. An unknown program fragment is added before S to establish θ. An *assume* statement is added before the while program to establish θ^* at the entry of the loop.
Strongest postconditions involve existential quantifiers. To simplify the formulas, these quantifiers should

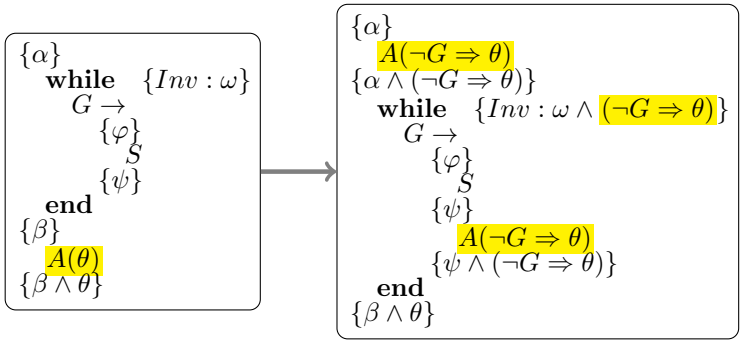


Fig. 16. *WhileIn* rule: Strengthens the invariant with $\neg G \Rightarrow \theta$

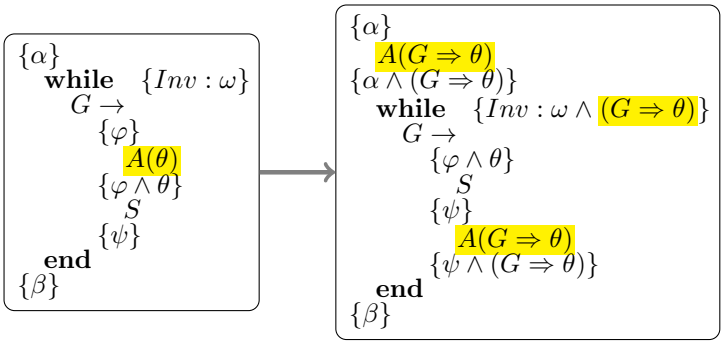


Fig. 17. *WhileStrInv* rule: Strengthens the invariant with $G \Rightarrow \theta$

be eliminated whenever possible. In this rule, we have defined θ^* to be the $sp(S, \theta)$. However, any formula θ^w weaker than the strongest postcondition will also work provided the program $\{\varphi \wedge \theta^w\}\ unkprog\{\varphi \wedge \theta\}$ can be derived.

5.4. Correctness of the Transformation Rules

To prove that a transformation rule $\mathcal{R} : src \mapsto target$ is *correctness preserving*, we need to prove $[proviso(src)] \Rightarrow [po(src)] \Rightarrow [po(target)]$ where the square brackets denote universal quantification over the points in state space as explained in Section 2.1. The *proviso* is optional and is assumed to be *true* if not mentioned. In this section, we give the proof of correctness preservation of the *WhilePostStrInv* rule. The proofs of correctness preservation for the other rules are given in Appendix B.

Theorem 5.1. *WhilePostStrInv* rule is correctness preserving.

Proof. There is no proviso for this rule. To prove that the rule is correctness preserving, we need to prove $[po(src)] \Rightarrow [po(target)]$. In this proof, we will prove $[po(src) \Rightarrow po(target)]$ which implies $[po(src)] \Rightarrow [po(target)]$.

Proof obligations of the src program and the $target$ program for the *WhilePostStrInv* are given in Fig 19.

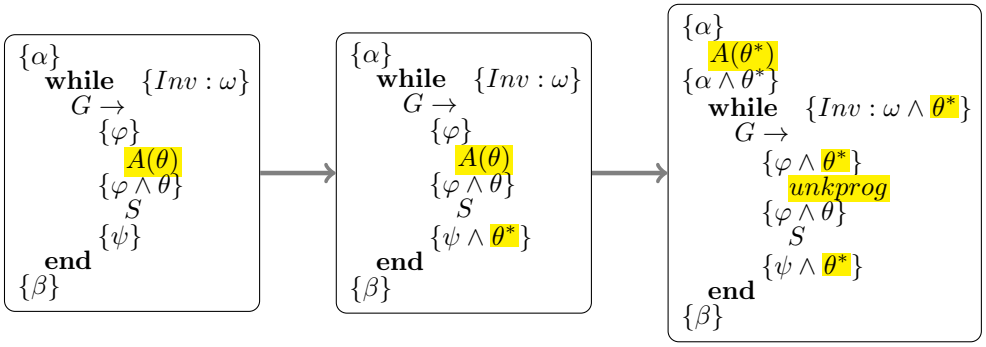


Fig. 18. *WhilePostStrInv* rule: Strengthens the loop invariant with θ^* where $\theta^* \triangleq sp(S, \theta)$

$po(src) = \bigwedge_i \Gamma_i$

$\Gamma_1 \quad : \alpha \Rightarrow \omega$
$\Gamma_2 \quad : \omega \wedge G \Rightarrow \varphi$
$\Gamma_3 \quad : po\,(\{\varphi\}\, A(\theta)\, \{\varphi \wedge \theta\})$
$\Gamma_4 \quad : po\,(\{\varphi \wedge \theta\}\, S\, \{\psi\})$
$\Gamma_5 \quad : \psi \Rightarrow \omega$
$\Gamma_6 \quad : \omega \wedge \neg G \Rightarrow \beta$

$po(target) = \bigwedge_i \Delta_i$

$\Delta_1 \quad : po\,(\{\alpha\}\, A(\theta^*)\, \{\alpha \wedge \theta^*\})$
$\Delta_2 \quad : \alpha \wedge \theta^* \Rightarrow \omega \wedge \theta^*$
$\Delta_3 \quad : \omega \wedge \theta^* \wedge G \Rightarrow \varphi \wedge \theta^*$
$\Delta_4 \quad : po\,(\{\varphi \wedge \theta^*\}\, unkprog\, \{\varphi \wedge \theta\})$
$\Delta_5 \quad : po\,(\{\varphi \wedge \theta\}\, S\, \{\psi \wedge \theta^*\})$
$\Delta_6 \quad : \psi \wedge \theta^* \Rightarrow \omega \wedge \theta^*$
$\Delta_7 \quad : \omega \wedge \theta^* \wedge \neg G \Rightarrow \beta$

Fig. 19. Proof obligations of the *src* and the *target* programs for the *WhilePostStrInv* Rule.

We assume the correctness of the *src* program and prove the proof obligations of the target program, i.e. $\bigwedge_{i\in[1,6]} \Gamma_i \Rightarrow \bigwedge_{i\in[1,7]} \Delta_i$.

Δ_1 :
$po\,(\{\alpha\}\, A(\theta^*)\, \{\alpha \wedge \theta^*\})$
$\equiv$ { definition of *assume* }
$\alpha \wedge \theta^* \Rightarrow \alpha \wedge \theta^*$
$\equiv$ { predicate calculus }
true

Δ_2 :
$\alpha \wedge \theta^* \Rightarrow \omega \wedge \theta^*$
$\Leftarrow$ { predicate calculus }
$\alpha \Rightarrow \omega$
$\equiv$ { Γ_1 }
true

Δ_3 :
$\omega \wedge \theta^* \wedge G \Rightarrow \varphi \wedge \theta^*$
$\Leftarrow$ { predicate calculus }
$\omega \wedge G \Rightarrow \varphi$
$\equiv$ { Γ_2 }
true

Δ_4 :
$po\,(\{\varphi \wedge \theta^*\}\, unkprog\, \{\varphi \wedge \theta\})$
$\equiv$ { definition of *unkProg* }
true

Δ_5 :
$po\,(\{\varphi \wedge \theta\}\, S\, \{\psi \wedge \theta^*\})$
$\equiv$ { *po* is conjunctive in postcondition }
$po\,(\{\varphi \wedge \theta\}\, S\, \{\psi\}) \wedge po\,(\{\varphi \wedge \theta\}\, S\, \{\theta^*\})$
$\equiv$ { Γ_4 }
$po\,(\{\varphi \wedge \theta\}\, S\, \{\theta^*\})$
$\equiv$ { definition of θ^* }
true

Δ_6 :
$\psi \wedge \theta^* \Rightarrow \omega \wedge \theta^*$
$\Leftarrow$ { predicate calculus }
$\psi \Rightarrow \omega$
$\equiv$ { Γ_5 }
true

Δ_7 :
$\omega \wedge \theta^* \wedge \neg G \Rightarrow \beta$
$\Leftarrow$ { predicate calculus }
$\omega \wedge \neg G \Rightarrow \beta$
$\equiv$ { Γ_6 }
true

□

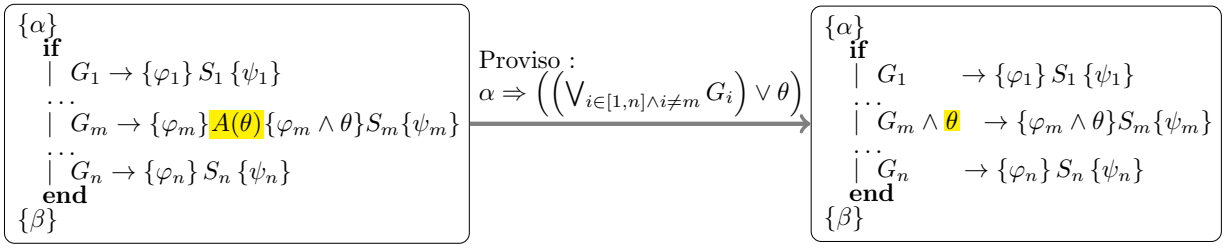


Fig. 20. *IfGrd2* rule: A variation of the *IfGrd* rule(Fig. 15).

5.5. Adding New Transformation Rules

New rules can be introduced as long as they are *correctness preserving*. For example, we can come up a *IfGrd2*(Fig. 20) rule which is a variation of the *IfGrd* rule (Fig. 15) where the rule is similar to the *IfGrd* except for the following differences.

1. This rule has a proviso $\alpha \Rightarrow \Big(\Big(\bigvee_{i\in[1,n]\wedge i\neq m} G_i\Big) \vee \theta\Big)$.
2. The target program in this case does not contain the guarded command with the guard $(G_m \wedge \neg\theta)$.

In the next section we present some guidelines for selecting appropriate assumption propagation rules that are likely to result in concrete programs.

5.6. Selecting Appropriate Rules

When encountered with an *assume* statement, we need to decide whether to establish (materialize) the assumption at its current location or to propagate it upstream. In many cases, this decision depends on the location of the *assume* statement. For example, if the *assume* statement is inside the body of a while loop, and materializing it will result in an inner loop, we may prefer to propagate it upstream and strengthen the invariant in order to derive an efficient program.

For propagating assumptions, our choices are limited by the location of the *assume* statement and the preceding program construct. For example, consider a scenario where an *assume* statement comes immediately after an *if* program. Although there are four rules for the *if* construct, only the *IfIn* rule is applicable in this case. In cases where multiple rules are applicable, select a rule that results in a simpler program. For example, if the *assume* statement is the first statement in the body of an *if* construct, there are two choices namely the *IfGrd* rule and the *IfGrd2* rule. In this case, it is desirable to apply the *IfGrd2* rule (provided the corresponding proviso is valid) since it results in a simpler program.

Another choice that one has to make quite often is between the *WhileStrInv* rule and the *WhilePostStrInv* rule. We select a rule that results in invariants that are easier to establish at the entry of the loop. In some cases (as discussed later in Section 6.2), both the paths may lead to concrete programs. One might have to proceed one or two steps and decide if a particular line of derivation is worth trying. This is much simpler than the ad hoc trial and error discussed in Section 3.2 where we had to guess the right predicates.

5.7. Down-propagating the Assertions

Note that dual to act of propagating assumptions upstream is the act of propagating assertions downstream by computing the strongest postconditions. A typical derivation involves interleaved instances of up-propagation of the *assume* statements and down-propagation of the assertions. We present one such example in Section 6.1.

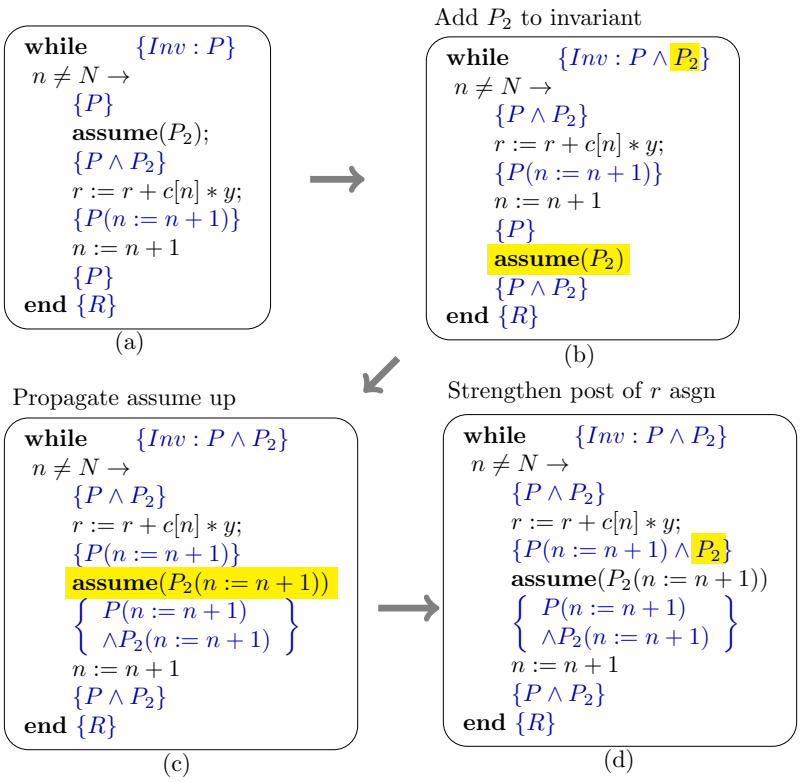


Fig. 21. Some steps in the derivation of a program for the *Horner's rule*. Invariant initializations at the entry of the loop are not shown.

6. Derivation Examples

6.1. Evaluating Polynomials

To demonstrate the interleaving of up-propagation of the assumptions and down-propagation of the assertions, we present some of the steps from the derivation of a program for evaluating a polynomial whose coefficients are stored in an array (also called *Horner's rule*). The program is specified as follows.

con $A[0..N)$ array of int $\{N \geq 0\}$;
con x : int; **var** r : int;
S
$\{R : r = (\sum i : 0 \leq i < N : c[i] * x^i)\}$

We skip the initial rule applications and directly jump to the program shown in Fig. 21(a). The user has already assumed predicate $P_2 : y = x^n$ during the calculation of r' (not shown). We next apply the *WhileStrInv* rule to strengthen the invariant with P_2 to arrive at program shown in the figure (b). We then propagate the *assume* statement upwards through $n := n+1$ to arrive at the program shown in figure (c). We would like to synthesize the assumption here but the precondition is not sufficient. Next, we strengthen the postcondition of the assignment statement for r to arrive at program shown in the figure (d). The assumption $P_2(n := n + 1)$ can now be easily established as $y := y * x$. Note that alternative solutions are also possible.

With the combinations of steps involving up-propagation of the *assume* statements and down-propagation

of the assertions, we can propagate the missing fragments to an appropriate location and then synthesize them.

6.2. Back to the Motivating Example

Next, we derive the motivating example from Section 3 using the assumption propagation approach. The initial derivation up to node G in Fig. 1 will remain the same except for the fact that we do not add $0 \leq n$ as an invariant initially. For the purpose of this example, P_1 is just $n \leq N$.

At node G, we are not able to express the formula $Q(n+1)$ as a program expression. Instead of speculating about what we should add at an upstream location so that we get $Q(n+1)$ at the current node, node G, we just assume the predicate that is needed at the current location. Instead of backtracking, we introduce a fresh variable s and assume the formula $s \equiv Q(n+1)$ and proceed further with the calculation.

$$
\begin{array}{ll}
 & \ldots \\
 & r' = r \ max \ Q(n+1) \\
\equiv & \left\{ \begin{array}{l} \text{introduce variable } s \text{ and} \\ \text{assume } s \equiv Q(n+1) \end{array} \right\} \\
 & r' = r \ max \ s \\
\equiv & \{ \ldots \}
\end{array}
$$

After instantiating r' to $r \ max \ s$, we arrive at a *while* program (node O in Fig. 22) where the body of the loop contains the statement $assume(s \equiv Q(n+1))$ as the first statement. We can establish the assumption at the current location, however that would be expensive since we would need to traverse the array inside the loop body. We therefore decide to propagate the assumption upwards out of the loop body. We now have two choices; we can apply the *WhilePostStrInv* rule or the *WhileStrInv* rule. We first show application of the *WhilePostStrInv* rule.

Application of the *WhilePostStrInv* rule strengthens the invariant by $s \equiv Q(n)$ and yields the program shown in node P in Fig. 22. We can now proceed further with the derivation of the *unkprog* fragment and the initialization *assume* statement as usual. The additional invariant $0 \leq n$ is added later when another *assume* statement (in node R) is propagated upwards. The final solution is shown in node S. This solution is derived in a linear fashion without any backtracking, thus avoiding the unnecessary rework.

Alternative solution

We now apply the *WhileStrInv* rule at node O in Fig. 22. Application of this rule adds $n \neq N \Rightarrow s = Q(n+1)$ as an invariant and results in the program shown in node T. We can now materialize the assume statements by deriving the corresponding program. The final solution is shown in node W. (For brevity, we have not shown the guarded command if the body of the guarded command contains only a *skip* statement.)
Remark. In section 3.2, we did not select $s = Q(n+1)$ as an invariant since our informal analysis warned us of an access to an undefined array element. As a result of this analysis, we discarded a possible program derivation path. However, if we apply the *WhileStrInv* rule, the array initialization problem does not occur as the term $s = Q(n+1)$ is suitably modified before adding it to the invariant. The rules are driven by logical necessity; program constructs are added only when they are logically necessary to preserve correctness. In this case, the appropriate guards are automatically added to safeguard us from accessing an undefined array element.

7. Implementing Assumption Propagation

The assumption propagation technique as described here can be adapted for use in any correct by construction program development environment. We have implemented it in the CAPS system [CD14]. CAPS is a tactic based interactive program derivation system. In the CAPS system, users incrementally transform a formal specification into a fully synthesized *annGCL* program by repeatedly applying predefined transformation rules called *Derivation Tactics*. The complete derivation history is recorded in the form of a *Derivation Tree*. The system provides various features like stepping into subcomponents, backtracking, branching. The system automates most of the mundane tasks and employs the automated theorem provers Alt-Ergo [CC], CVC3

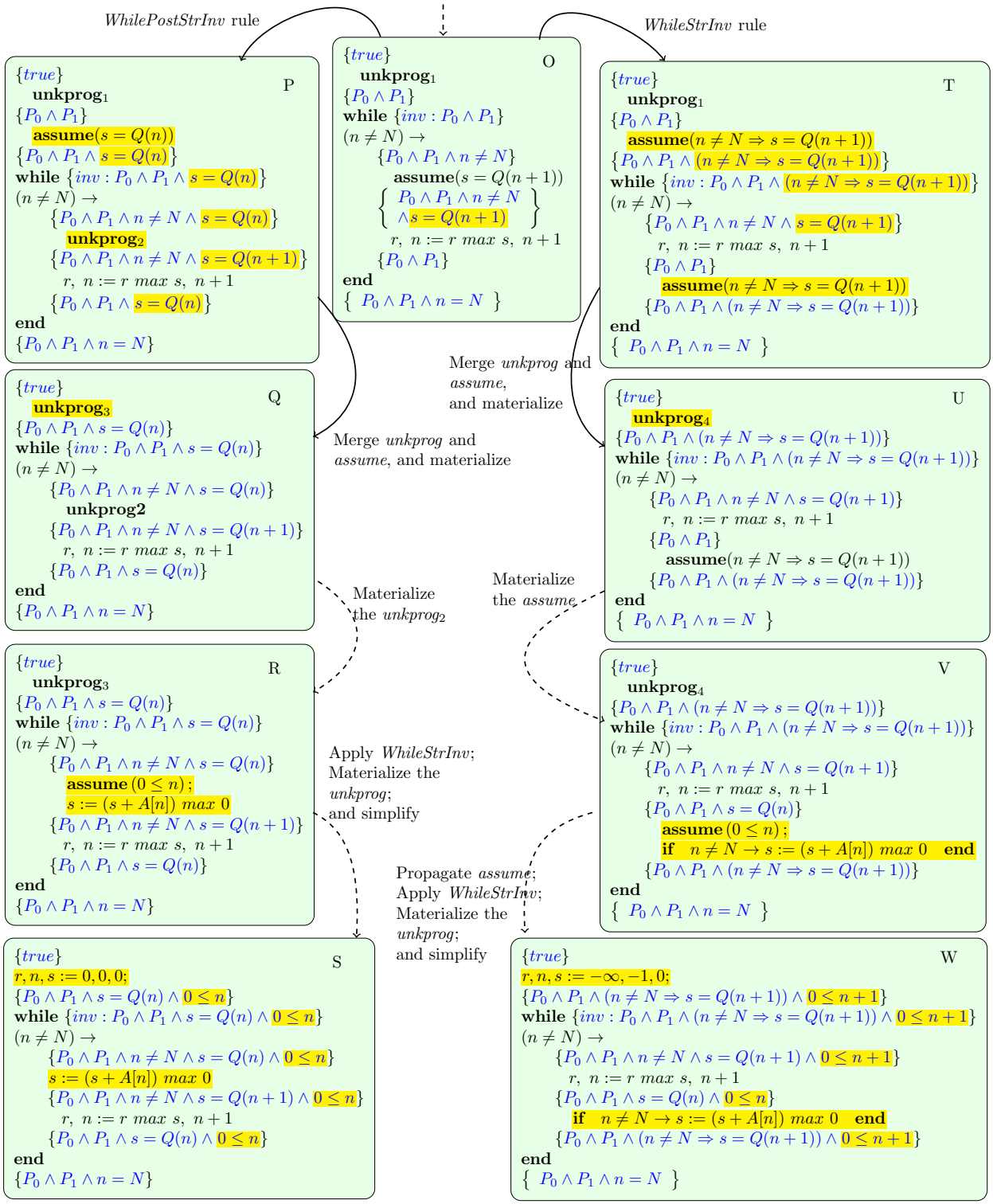


Fig. 22. Derivations using the assumption propagation rules *WhilePostStrInv* and *WhileStrInv*. The following predicate definitions are same as those in Fig. 1 except for P_1.

$P_0 \triangleq r = (Max\ p, q : 0 \le p \le q \le n : Sum.p.q)\,; \quad P_1 \triangleq n \le N$

$Q(n) \triangleq (Max\ p :\ 0 \le p \le n :\ Sum.p.n)$

[BT07], SPASS [WBH+02] and Z3 [DMB08] for discharging proof obligations. The Why3 tool [FP13] is used to interface with these theorem provers.

Program and Formula Modes

The CAPS system provides tactics for transforming partially derived programs as well as the proof obligation formulas. These two modes are referred as the *Program Mode* and the *Formula Mode* respectively. Users can envision missing program fragments in terms of metavariables which are then derived by manipulating the proof obligation formulas. The *StepIntoPO* (Step Into Proof Obligation) tactic is used to transition from programs to corresponding proof obligation formulas. On applying the tactic to an *annGCL* program containing metavariables, a new formula node representing the proof obligations (verification conditions) is created in the derivation tree. This formula is then incrementally transformed to a form, from which it is easier to instantiate the metavariables. After successfully discharging the proof obligation and instantiating all the metavariables, a tactic called *StepOut* is applied to get an *annGCL* program with all the metavariables replaced by the corresponding instantiations.

Assumptions are typically made in the formula mode after stepping into the proof obligation. After stepping out of the formula mode, they are added as *assume* statements to the resulting *annGCL* programs. The assumption propagation tactics available in the program mode can then be used to move these *assume* statements.

For the ease of derivation, the CAPS system provides a metatactic called *PropagateAssumption* which can propagate a selected assumption from its current location to any desired upstream location, provided there are no intermediate loops in the path. When multiple rules are applicable during this propagation, the metatactic chooses certain predefined default rules. For propagating assumptions through loops, *WhileIn*, *WhileStrInv*, and *WhilePostStrInv* rules are implemented. Assertions can also be propagated downwards when needed. We have implemented heuristics for simplifying the formulas by eliminating the existential quantifiers in the strongest postconditions.

8. Related Work

The work most closely related to our *assumption propagation* is that of [LvW97] and [BvW98] on *context assumptions*. However, their main purpose in propagating assumptions is to move them to another place in the program where the existing annotations would imply the assumptions being made. In contrast, our focus is to propagate assumptions to a suitable place where they can be materialized by introducing concrete program fragments. The rule set given in [LvW97] and [BvW98] is weaker in that they do not have assumption propagation rules related to loops. As Back et. al. say, *"...there is no rule for loops, we assume that whenever a loop is introduced, sufficient information about the loop is added as an assertion after the loop"*. As shown in the examples in Section 6, for the purpose of our work, the assumption propagation rules related to loops are often the most important ones in practice.

In the context of data refinement, Morgan [Mor90] introduces the concept of *coercions* for *making* a formula true at a given point in a program. However, the focus of their work is on refining abstract variables with concrete variables and has rules for adding variables (*augment coercion*) and removing variables (*diminish coercion*). Groves uses in [Gro98] the concept of *coercions* in the context of specification modifications. His purpose is to modify a given implementation when the postcondition of the program is strengthened.

Various tools exist for refinement-based formal program derivation. Refinement Calculator [BL96] provides a general mechanism for refinement-based transformational reasoning on top of HOL. The PRT tool [CHN+96] extends the Ergo theorem prover and supports refinement-based program development with a close integration of refinements and proof support in a single tool. The Cocktail [Fra99] tool supports the derivation programs from specifications using the Hoare/Dijkstra method with support for interactive proof construction as well as automatic theorem proving.

9. Conclusion

We have discussed the problems associated with the ad hoc reasoning involved during the top-down derivation of programs. To address these problems, we have proposed an assumption propagation technique wherein users can make assumptions whenever needed and then propagate them to an appropriate location before finally materializing them. We have presented transformation rules for propagating assumptions through annotated programs while preserving correctness with minimal additional proof efforts. We have implemented these rules in the CAPS system and described how these rules can be systematically integrated in a top-down derivation methodology. With the help of examples, we have shown how the assumption propagation rules reduce the need for ad hoc reasoning. These rules help exploration of alternative solutions and also help avoid unnecessary branching and rework during program derivations.

Acknowledgements.

The work of the first author was supported by the Tata Consultancy Services (TCS) Research Fellowship and an assistantship from the Ministry of Human Resource Development(MHRD), Government of India.

References

[BL96] Michael Butler and Thomas Långbacka. Program derivation using the refinement calculator. In *Theorem Proving in Higher Order Logics: 9th International Conference, volume 1125 of LNCS*, pages 93–108. Springer Verlag, 1996.

[BM06] Roland Backhouse and Diethard Michaelis. Exercises in quantifier manipulation. In *Mathematics of program construction*, pages 69–81. Springer, 2006.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, Berlin, 1998.

[CC] Sylvain Conchon and Evelyne Contejean. The alt-ergo automatic theorem prover, 2008.

[CD14] Dipak L. Chaudhari and Om Damani. Automated theorem prover assisted program calculations. In Elvira Albert and Emil Sekerinski, editors, *Integrated Formal Methods*, Lecture Notes in Computer Science, pages 205–220. Springer International Publishing, 2014.

[CD15] Dipak L. Chaudhari and Om P. Damani. Combining top-down and bottom-up techniques in program derivation. In *Logic-Based Program Synthesis and Transformation - 25th International Symposium, LOPSTR 2015, Siena, Italy, July 13-15, 2015. Revised Selected Papers*, pages 244–258, 2015.

[CHN+96] David Carrington, Ian Hayes, Ray Nickson, G. N. Watson, and Jim Welsh. A tool for developing correct programs by refinement. Technical report, 1996.

[Coh90] Edward Cohen. *Programming in the 1990s - An Introduction to the Calculation of Programs*. Texts and Monographs in Computer Science. Springer, 1990.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[Dij76] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, Berlin, 1990.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – Where Programs Meet Provers. In *ESOP'13 22nd European Symposium on Programming*, volume 7792 of *LNCS*, Rome, Italie, 2013. Springer.

[Fra99] Michael Franssen. Cocktail: A tool for deriving correct programs. In *Workshop on Automated Reasoning*, 1999.

[Gri87] David Gries. *The Science of Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1987.

[Gro98] Lindsay Groves. Adapting program derivations using program conjunction. In *International Refinement Workshop and Formal Methods Pacific*, volume 98, pages 145–164. Citeseer, 1998.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM: Communications of the ACM*, 12, 1969.

[Kal90] Anne Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice-Hall, Inc., NJ, USA, 1990.

[LvW97] Linas Laibinis and Joakim von Wright. Context handling in the refinement calculus framework. Technical Report TUCS-TR-118, Turku Centre for Computer Science, Finland, August 21, 1997.

[Mor90] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, Inc., 1990.

[OXC04] Marcel Oliveira, Manuela Xavier, and Ana Cavalcanti. Refine and gabriel: support for refinement and tactics. In *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, pages 310–319. IEEE, 2004.

[WBH+02] Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobalt, and Dalibor Topic. SPASS version 2.0. In Andrei Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 275–279. Springer-Verlag, 2002.

A. An Introductory Program Derivation Example

To introduce beginners to the calculational style of program derivation, we present a simple but interesting derivation of a program for constructing a table of cubes using only additive operations. This derivation is based on the derivation in [Coh90].

The problem can be specified as follows.

$$
\begin{array}{l}
\textbf{con}\ N : \text{int}\,\{N \geq 0\}\,; \\
\textbf{var}\ c : \text{array}\ [0..N)\ \text{of int}; \\
\quad S \\
\{R : (\forall i : 0 \leq i < N : c[i] = i^3)\}
\end{array}
$$

Our task is to derive program S to establish the postcondition R. We are not allowed to use exponentiation or multiplication in the solution.

Step 1

We start by applying the heuristic of replacing the constant N by a fresh variable n and rewrite the postcondition as $P_0 \wedge (n = N)$ where P_0 is defined as follows.

$$P_0 : \ (\forall i : 0 \leq i < n : c[i] = i^3)$$

After rewriting the postcondition, we arrive at the following program.

$$
\begin{array}{l}
\textbf{con}\ N : \text{int}\,\{N \geq 0\}\,; \\
\textbf{var}\ c : \text{array}\ [0..N)\ \text{of int}; \\
\textbf{var}\ n : \text{int}; \\
\quad S \\
\{P_0 \wedge (n = N)\}
\end{array}
$$

Note that the new postcondition $P_0 \wedge (n = N)$ implies the original postcondition R.

Step 2

We now apply the well-known heuristic of *taking a conjunct as an invariant*. We introduce a while loop and select P_0 as an invariant and the negation of the remaining conjunct as the guard of the while loop. We follow the general guideline of adding bounds on the introduced variables by adding $P_1 : 0 \leq n \leq N$ as an additional invariant. We also initialize n with 0 to establish the invariants at the start of the loop and increment n inside the loop body.

$$
\begin{array}{l}
\textbf{var}\ n : \text{int}; \\
n := 0; \\
\textbf{while} \quad \{Inv : P_0 \wedge P_1\} \\
\quad n \neq N \rightarrow \\
\quad\quad S_1; \\
\quad\quad n := n + 1 \\
\textbf{end} \\
\{P_0 \wedge (n = N)\}
\end{array}
$$

Step 3

We can select S_1 to be $c[n] := n^3$ as it preserves the invariants. However, we are not allowed to use exponentiation. To eliminate exponentiation, we introduce fresh variable x, and rewrite our program as:

var n : int;
$n := 0$;
while $\{Inv : P_0 \wedge P_1\}$
  $n \neq N \rightarrow$
    //establish $P_2 : x = n^3$
    $c[n] := x$;
    $n := n + 1$;
end
$\{P_0 \wedge (n = N)\}$

This program is correct provided $P_2 : x = n^3$ is a precondition of the statement $c[n] := x$. In order to establish P_2 as a precondition to $c[n] := x$, we maintain P_2 as a loop invariant. After adding P_2 as an invariant we arrive at the program

var $n,\ x$: int;
$n,\ x := 0,\ 0$;
while $\{Inv : P_0 \wedge P_1 \wedge P_2\}$
  $n \neq N \rightarrow$
    $c[n] := x$;
    $n,\ x := n + 1,\ x'$;
end

where x' must be chosen to maintain P_2.

Step 4

The invariant $P2$ is available before the assignment $n, x := n+1, x'$ since the preceding statement ($c[n] := x$) does not change its validity. We now calculate x' such that P_2 in preserved by the loop body:

$wp((n,\ x := n + 1,\ x'), P_2)$
$\equiv$ { Definition of P_2 and assignment }
$x' = (n + 1)^3$
$\equiv$ { Arithmetic }
$x' = n^3 + 3 * n^2 + 3 * n + 1$
$\equiv$ { P_2, to eliminate an exponentiation }
$x' = x + 3 * n^2 + 3 * n + 1$
$\equiv$ { Assume $P_3 : y = 3 * n^2 + 3 * n + 1$ to eliminate the exponentiation and the multiplications }
$x' = x + y$

During the calculation of x', to eliminate exponentiation and multiplications, we assumed that P_3 holds. This can be ensured by maintaining P_3 as a loop invariant. The following program is correct provided y' is chosen to maintain the invariance of P_3.

var $n,\ x,\ y$: int;
$n,\ x,\ y := 0,\ 0,\ 1$;
while $\{Inv : P_0 \wedge P_1 \wedge P_2 \wedge P_3\}$
  $n \neq N \rightarrow$
    $c[n] := x$;
    $n,\ x,\ y := n + 1,\ x + y,\ y'$;
end

Step 5

We now calculate x' such that P_3 in preserved by the loop body:

$$
\begin{array}{ll}
& wp((n,\ x,\ y := n+1,\ x+y,\ y'),\ P_3) \\
\equiv & \quad \{ \text{ Definitions of } P_3 \text{ and assignment } \} \\
& y' = 3*(n+1)^2 + 3*(n+1) + 1 \\
\equiv & \quad \{ \text{ Arithmetic} \} \\
& y' = 3*n^2 + 9*n + 7 \\
\equiv & \quad \{ P_3 \} \\
& y' = y + 6*n + 6
\end{array}
$$

After substituting $y + 6*n + 6$ for y in the program from the previous step, we arrive at the following program.

con N : int $\{N \geq 0\}$;
var c : array $[0..N)$ of int;
var $n,\ x,\ y$: int;
$n, x, y := 0, 0, 1$;
while $\{Inv : P_0 \wedge P_1 \wedge P_2 \wedge P_3\}$
  $n \neq N \rightarrow$
    $c[n] := x$;
    $n,\ x,\ y := n+1,\ x+y,\ y + 6*n + 6$
end

The term $6*n$ can be easily rewritten using only the addition operation. We now have a linear-time solution for the problem.

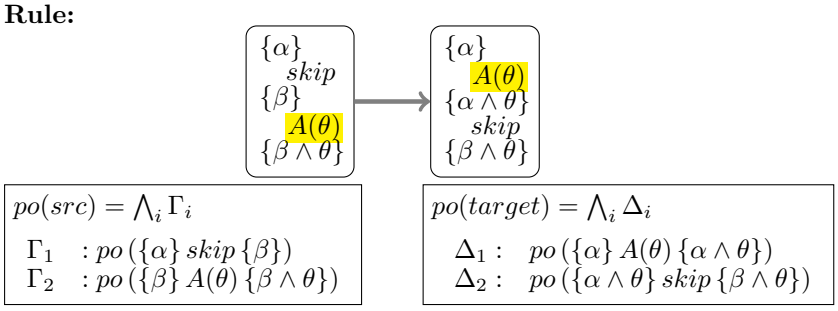
B. Correctness Proofs

To prove that a rule $\mathcal{R} : src \mapsto target$ is correctness preserving, we need to prove $[proviso(src)] \Rightarrow [po(src)] \Rightarrow [po(target)]$. In the following proofs, we will prove $[proviso(src) \Rightarrow po(src) \Rightarrow po(target)]$ which implies $[proviso(src)] \Rightarrow po(src)] \Rightarrow [po(target)]$.

Let Γ_i be the proof obligation of the *src* program and Δ_i be the proof obligations of the target program. To prove $[\bigwedge_i \Gamma_i \Rightarrow \bigwedge_i \Delta_i]$, we will assume the antecedents and prove the proof obligations of the target programs(Δ_i) separately using the calculational style.

B.1. SkipUp Rule

Rule:



Theorem B.1. *SkipUp* rule is correctness preserving.

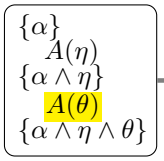
Proof. We assume the correctness of the *src* program and prove the proof obligations of the target program(Δ_i) separately.

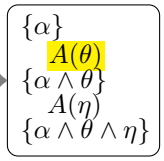
Δ_1 :
$po\,(\{\alpha\}\,A(\theta)\,\{\alpha \wedge \theta\})$
$\equiv$ { definition of *assume* }
$\alpha \wedge \theta \Rightarrow \alpha \wedge \theta$
$\equiv$ { predicate calculus }
true

Δ_2 :
$po\,(\{\alpha \wedge \theta\}\,skip\,\{\beta \wedge \theta\})$
$\equiv$ { definition of *skip* }
$\alpha \wedge \theta \Rightarrow \beta \wedge \theta$
$\Leftarrow$ { predicate calculus }
$\alpha \Rightarrow \beta$
$\equiv$ { Γ_1 }
true

B.2. AssumeUp Rule

Rule:





$po(src) = \bigwedge_i \Gamma_i$
Γ_1 : $po\,(\{\alpha\}\,A(\eta)\,\{\alpha \wedge \eta\})$
Γ_2 : $po\,(\{\alpha \wedge \eta\}\,A(\theta)\,\{\alpha \wedge \eta \wedge \theta\})$

$po(target) = \bigwedge_i \Delta_i$
Δ_1 : $po\,(\{\alpha\}\,A(\theta)\,\{\alpha \wedge \theta\})$
Δ_2 : $po\,(\{\alpha \wedge \theta\}\,A(\eta)\,\{\alpha \wedge \theta \wedge \eta\})$

Theorem B.2. *AssumeUp* rule is correctness preserving.

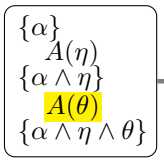
Proof. We assume the correctness of the *src* program and prove the proof obligations of the target program(Δ_i) separately.

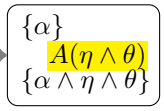
Δ_1 :
$po\,(\{\alpha\}\,A(\theta)\,\{\alpha \wedge \theta\})$
$\equiv$ { definition of *assume* }
$\alpha \wedge \theta \Rightarrow \alpha \wedge \theta$
$\equiv$ { predicate calculus }
true

Δ_2 :
$po\,(\{\alpha \wedge \theta\}\,A(\eta)\,\{\alpha \wedge \theta \wedge \eta\})$
$\equiv$ { definition of *assume* }
$\alpha \wedge \theta \wedge \eta \Rightarrow \alpha \wedge \theta \wedge \eta$
$\equiv$ { predicate calculus }
true

B.3. AssumeMerge Rule

Rule:





$po(src) = \bigwedge_i \Gamma_i$
Γ_1 : $po\,(\{\alpha\}\,A(\eta)\,\{\alpha \wedge \eta\})$
Γ_2 : $po\,(\{\alpha \wedge \eta\}\,A(\theta)\,\{\alpha \wedge \eta \wedge \theta\})$

$po(target) = \Delta_1$
Δ_1 : $po\,(\{\alpha\}\,A(\eta \wedge \theta)\,\{\alpha \wedge \eta \wedge \theta\})$

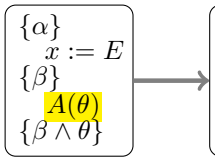
Theorem B.3. *AssumeMerge* rule is correctness preserving.

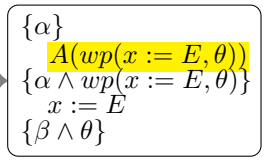
Proof. We assume the correctness of the *src* program and prove the proof obligation of the target program(Δ_1).

Δ_1 :
$po\,(\{\alpha\}\, A(\eta \wedge \theta)\, \{\alpha \wedge \eta \wedge \theta\})$
$\equiv$ { definition of *assume* }
$\alpha \wedge \eta \wedge \theta \Rightarrow \alpha \wedge \eta \wedge \theta$
$\equiv$ { predicate calculus }
$true$

B.4. AssignmentUp Rule

Rule:





$po(src) = \bigwedge_i \Gamma_i$
Γ_1 : $\alpha \Rightarrow \beta(x := E)$
Γ_2 : $po\,(\{\beta\}\, A(\theta)\, \{\beta \wedge \theta\})$

$po(target) = \bigwedge_i \Delta_i$
Δ_1 : $po\,(\{\alpha\}\, A(wp(x := E, \theta))\, \{\alpha \wedge wp(x := E, \theta)\})$
Δ_2 : $po\,(\{\alpha \wedge wp(x := E, \theta)\}\, x := E\, \{\beta \wedge \theta\})$

Theorem B.4. *AssignmentUp* rule is correctness preserving.

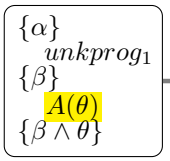
Proof. We assume the correctness of the *src* program and prove the proof obligations of the target program(Δ_i) separately.

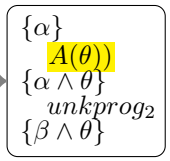
Δ_1 :
$po\,(\{\alpha\}\, A(wp(x := E, \theta))\, \{\alpha \wedge wp(x := E, \theta)\})$
$\equiv$ { definition of assume }
$\alpha \wedge wp(x := E, \theta) \Rightarrow \alpha \wedge wp(x := E, \theta)$
$\equiv$ { predicate calculus }
$true$

Δ_2 :
$po\,(\{\alpha \wedge wp(x := E, \theta)\}\, x := E\, \{\beta \wedge \theta\})$
$\equiv$ { definition of assignment }
$\alpha \wedge wp(x := E, \theta) \Rightarrow (\beta \wedge \theta)(x := E)$
$\equiv$ { definition of weakest precondition; substitution distributes over conjunction }
$\alpha \wedge \theta(x := E) \Rightarrow \beta(x := E) \wedge \theta(x := E)$
$\Leftarrow$ { predicate calculus }
$\alpha \Rightarrow \beta(x := E)$
$\equiv$ { Γ_1 }
$true$

B.5. UnkProgUp Rule

Rule:





$po(src) = \bigwedge_i \Gamma_i$
Γ_1 : $po\,(\{\alpha\}\, unkprog_1\, \{\beta\})$
Γ_2 : $po\,(\{\beta\}\, A(\theta)\, \{\beta \wedge \theta\})$

$po(target) = \bigwedge_i \Delta_i$
Δ_1 : $po\,(\{\alpha\}\, A(\theta)\, \{\alpha \wedge \theta\})$
Δ_2 : $po\,(\{\alpha \wedge \theta\}\, unkprog_2\, \{\beta \wedge \theta\})$

Theorem B.5. *UnkProgUp* rule is correctness preserving.

Proof. We assume the correctness of the *src* program and prove the proof obligations of the target program(Δ_i) separately.

Δ_1 :

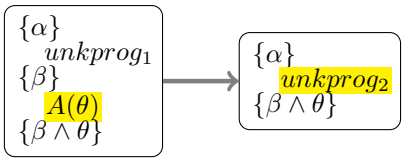
$po\,(\{\alpha\}\,A(\theta)\,\{\alpha \wedge \theta\})$
$\equiv$ { definition of assumption }
$\alpha \wedge \theta \Rightarrow \alpha \wedge \theta$
$\equiv$ { predicate calculus }
$true$

Δ_2

$po\,(\{\alpha \wedge \theta\}\,unkprog_2\,\{\beta \wedge \theta\})$
$\equiv$ { definition of *unkprog* }
$true$

B.6. UnkProgEst Rule

Rule:



$po(src) = \bigwedge_i \Gamma_i$

Γ_1 : $po\,(\{\alpha\}\,unkprog_1\,\{\beta\})$
Γ_2 : $po\,(\{\beta\}\,A(\theta)\,\{\beta \wedge \theta\})$

$po(target) = \bigwedge_i \Delta_i$

Δ_1 : $po\,(\{\alpha\}\,unkprog_2\,\{\beta \wedge \theta\})$

Theorem B.6. *UnkProgEst* rule is correctness preserving.

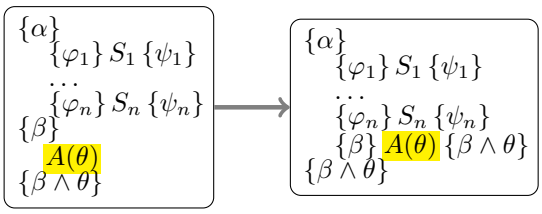
Proof. We assume the correctness of the *src* program and prove the proof obligations of the target program(Δ_i) separately.

Δ_1 :

$po\,(\{\alpha\}\,unkprog_2\,\{\beta \wedge \theta\})$
$\equiv$ { definition of *unkprog*}
$true$

B.7. CompositionIn Rule

Rule:



$po(src) = \bigwedge_i \Gamma_i$

Γ_1 : $\alpha \Rightarrow \varphi_1$
Γ_2 : $\left(\bigwedge_{i\in[1,n]} po\,(\{\varphi_i\}\,S_i\,\{\psi_i\})\right)$
Γ_3 : $\left(\bigwedge_{i\in[1,n-1]} \psi_i \Rightarrow \varphi_{i+1}\right)$
Γ_4 : $\psi_n \Rightarrow \beta$
Γ_5 : $po\,(\{\beta\}\,A(\theta)\,\{\beta \wedge \theta\})$

$po(target) = \bigwedge_i \Delta_i$

Δ_1 : $\alpha \Rightarrow \varphi_1$
Δ_2 : $\left(\bigwedge_{i\in[1,n]} po\,(\{\varphi_i\}\,S_i\,\{\psi_i\})\right)$
Δ_3 : $\left(\bigwedge_{i\in[1,n-1]} \psi_i \Rightarrow \varphi_{i+1}\right)$
Δ_4 : $\psi_n \Rightarrow \beta$
Δ_5 : $po\,(\{\beta\}\,A(\theta)\,\{\beta \wedge \theta\})$
Δ_6 : $\beta \wedge \theta \Rightarrow \beta \wedge \theta$

Theorem B.7. *CompositionIn* rule is correctness preserving.

Proof. We assume the correctness of the *src* program and prove the proof obligations of the target program(Δ_i) separately.

Δ_1 :
$\alpha \Rightarrow \varphi_1$
$\equiv$ { Γ_1 }
true

Δ_2 :
$\left(\bigwedge_{i\in[1,n]} po\left(\{\varphi_i\}\, S_i\, \{\psi_i\}\right)\right)$
$\equiv$ { Γ_2 }
true

Δ_3 :
$\left(\bigwedge_{i\in[1,n-1]} \psi_i \Rightarrow \varphi_{i+1}\right)$
$\equiv$ { Γ_3 }
true

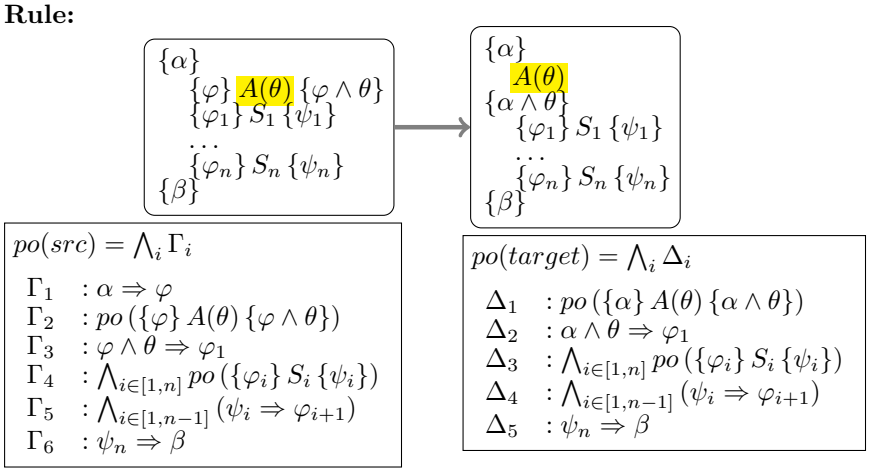
Δ_4 :
$\psi_n \Rightarrow \beta$
$\equiv$ { Γ_4 }
true

Δ_5 :
$po\left(\{\beta\}\, A(\theta)\, \{\beta \wedge \theta\}\right)$
$\equiv$ { Γ_5 }
true

Δ_6 :
$\beta \wedge \theta \Rightarrow \beta \wedge \theta$
$\equiv$ { predicate calculus }
true

B.8. CompositionOut Rule

Rule:



Theorem B.8. *UnkProgUp* rule is correctness preserving.

Proof. We assume the correctness of the *src* program and prove the proof obligations of the target programs(Δ_i) separately.

Δ_1 :
$po\left(\{\alpha\}\, A(\theta)\, \{\alpha \wedge \theta\}\right)$
$\equiv$ { definition of assume }
$\alpha \wedge \theta \Rightarrow \alpha \wedge \theta$
$\equiv$ { predicate calculus }
true

Δ_2 :
$\alpha \wedge \theta \Rightarrow \varphi_1$
$\Leftarrow$ { Γ_1; Γ_3 }
true

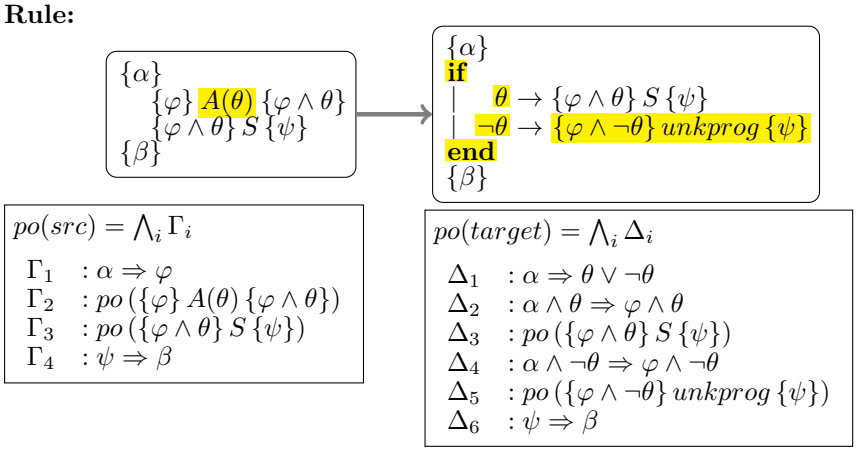
Δ_3 :
$\bigwedge_{i\in[1,n]} po\,(\{\varphi_i\}\, S_i\, \{\psi_i\})$
$\equiv$ { Γ_4 }
$true$

Δ_4 :
$\bigwedge_{i\in[1,n-1]} (\psi_i \Rightarrow \varphi_{i+1})$
$\equiv$ { Γ_5 }
$true$

Δ_5 :
$\psi_n \Rightarrow \beta$
$\equiv$ { Γ_6 }
$true$

B.9. CompoToIf Rule

Rule:



Theorem B.9. *CompoToIf* rule is correctness preserving.

Proof. We assume the correctness of the *src* program and prove the proof obligations of the target program(Δ_i) separately.

Δ_1 :
$\alpha \Rightarrow \theta \vee \neg\theta$
$\equiv$ { predicate calculus }
$\alpha \Rightarrow true$
$\equiv$ { predicate calculus }
$true$

Δ_2 :
$\alpha \wedge \theta \Rightarrow \varphi \wedge \theta$
$\Leftarrow$ { predicate calculus }
$\alpha \Rightarrow \varphi$
$\equiv$ { Γ_1 }
$true$

Δ_3 :
$po\,(\{\varphi \wedge \theta\}\, S\, \{\psi\})$
$\equiv$ { Γ_3 }
$true$

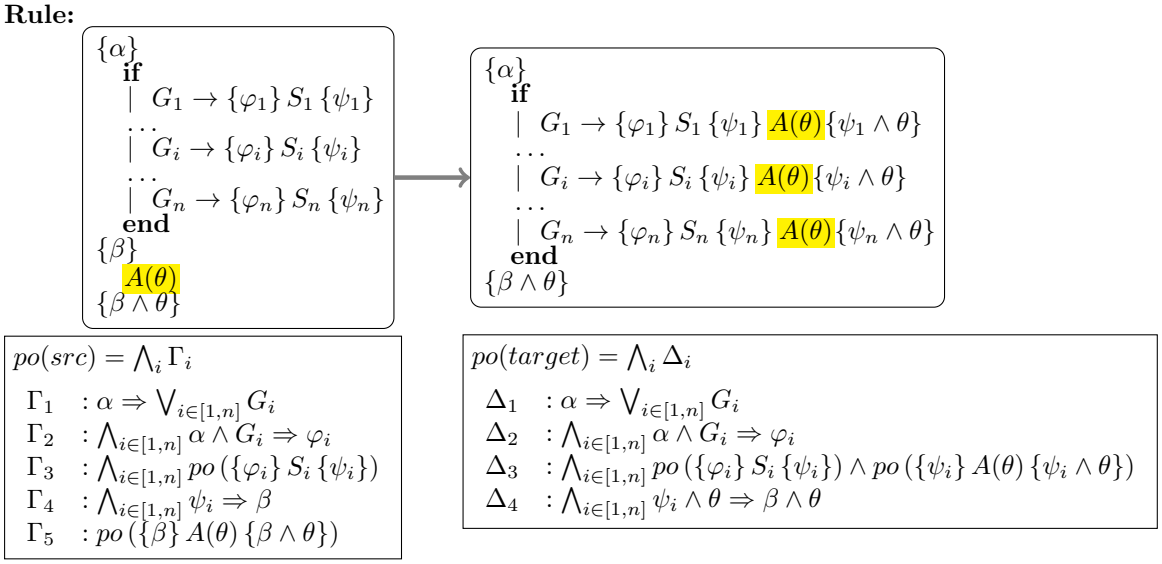
Δ_4 :
$\alpha \wedge \neg\theta \Rightarrow \varphi \wedge \neg\theta$
$\Leftarrow$ { predicate calculus }
$\alpha \Rightarrow \varphi$
$\equiv$ { Γ_1 }
$true$

Δ_5 :
$po\,(\{\varphi \wedge \neg\theta\}\, unkprog\, \{\psi\})$
$\equiv$ { definition of *unkprog* }
$true$

Δ_6 :
$\psi \Rightarrow \beta$
$\equiv$ { Γ_4 }
$true$

B.10. IfIn Rule

Rule:



Theorem B.10. *IfIn* rule is correctness preserving.

Proof. We assume the correctness of the *src* program and prove the proof obligations of the target program(Δ_i) separately.

Δ_1 :

$\alpha \Rightarrow \bigvee_{i\in[1,n]} G_i$

$\equiv$ { Γ_1 }

true

Δ_2 :

$\bigwedge_{i\in[1,n]} \alpha \wedge G_i \Rightarrow \varphi_i$

$\equiv$ { Γ_2 }

true

Δ_3 :

$\bigwedge_{i\in[1,n]} po(\{\varphi_i\}\ S_i\ \{\psi_i\}) \wedge po(\{\psi_i\}\ A(\theta)\ \{\psi_i \wedge \theta\})$

$\equiv$ { definition of *assume* }

$\bigwedge_{i\in[1,n]} po(\{\varphi_i\}\ S_i\ \{\psi_i\}) \wedge (\psi_i \wedge \theta \Rightarrow \psi_i \wedge \theta)$

$\equiv$ { predicate calculus }

$\bigwedge_{i\in[1,n]} po(\{\varphi_i\}\ S_i\ \{\psi_i\})$

$\equiv$ { Γ_3 }

true

Δ_4 :

$\bigwedge_{i\in[1,n]} \psi_i \wedge \theta \Rightarrow \beta \wedge \theta$

$\Leftarrow$ { predicate calculus }

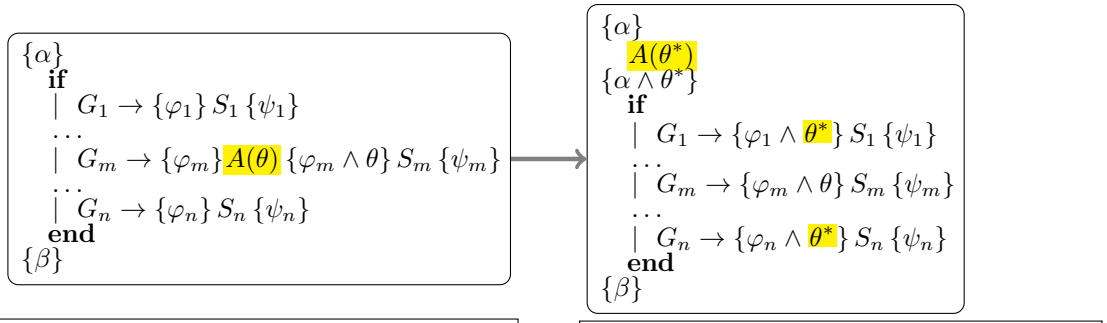
$\bigwedge_{i\in[1,n]} \psi_i \Rightarrow \beta$

$\equiv$ { Γ_4 }

true

B.11. IfOut Rule

Rule:



$po(src) = \bigwedge_i \Gamma_i$

$\Gamma_1 \quad : \alpha \Rightarrow \bigvee_{i\in[1,n]} G_i$
$\Gamma_2 \quad : \bigwedge_{i\in[1,n]} \alpha \wedge G_i \Rightarrow \varphi_i$
$\Gamma_3 \quad : \bigwedge_{i\in[1,n]\wedge i\neq m} po\,(\{\varphi_i\}\, S_i\, \{\psi_i\})$
$\Gamma_4 \quad : po\,(\{\varphi_m\}\, A(\theta)\, \{\varphi_m \wedge \theta\}) \wedge po\,(\{\varphi_m \wedge \theta\}\, S_m\, \{\psi_m\})$
$\Gamma_5 \quad : \bigwedge_{i\in[1,n]} \psi_i \Rightarrow \beta$

$po(target) = \bigwedge_i \Delta_i$

$\Delta_1 \quad : po\,(\{\alpha\}\, A(\theta^*)\, \{\alpha \wedge \theta^*\})$
$\Delta_2 \quad : \alpha \wedge \theta^* \Rightarrow \bigvee_{i\in[1,n]} G_i$
$\Delta_3 \quad : \bigwedge_{i\in[1,n]\wedge i\neq m} \alpha \wedge \theta^* \wedge G_i \Rightarrow \varphi_i \wedge \theta^*$
$\Delta_4 \quad : \alpha \wedge \theta^* \wedge G_m \Rightarrow \varphi_m \wedge \theta$
$\Delta_5 \quad : \bigwedge_{i\in[1,n]\wedge i\neq m} po\,(\{\varphi_i \wedge \theta^*\}\, S_i\, \{\psi_i\})$
$\Delta_6 \quad : po\,(\{\varphi_m \wedge \theta\}\, S_m\, \{\psi_m\})$
$\Delta_7 \quad : \bigwedge_{i\in[1,n]} \psi_i \Rightarrow \beta$

IfOut rule. $\theta^* \triangleq G_m \Rightarrow \theta$

Theorem B.11. *IfOut* rule is correctness preserving.

Proof. We assume the correctness of the *src* program and prove the proof obligations of the target program(Δ_i) separately.

Δ_1 :
$po\,(\{\alpha\}\, A(\theta^*)\, \{\alpha \wedge \theta^*\})$
$\equiv$ { definition of *assume* }
$\alpha \wedge \theta^* \Rightarrow \alpha \wedge \theta^*$
$\equiv$ { predicate calculus }
true

Δ_2 :
$\alpha \wedge \theta^* \Rightarrow \bigvee_{i\in[1,n]} G_i$
$\Leftarrow$ { predicate calculus }
$\alpha \Rightarrow \bigvee_{i\in[1,n]} G_i$
$\equiv$ { Γ_1 }
true

Δ_3 :
$\bigwedge_{i\in[1,n]\wedge i\neq m} \alpha \wedge \theta^* \wedge G_i \Rightarrow \varphi_i \wedge \theta^*$
$\Leftarrow$ { predicate calculus }
$\bigwedge_{i\in[1,n]\wedge i\neq m} \alpha \wedge G_i \Rightarrow \varphi_i$
$\Leftarrow$ { Γ_2 }
true

Δ_4 :
$\alpha \wedge \theta^* \wedge G_m \Rightarrow \varphi_m \wedge \theta$
$\Leftarrow$ { definition of θ^* }
$\alpha \wedge (G_m \Rightarrow \theta) \wedge G_m \Rightarrow \varphi_m \wedge \theta$
$\equiv$ { predicate calculus }
$\alpha \wedge G_m \wedge \theta \Rightarrow \varphi_m \wedge \theta$
$\Leftarrow$ { predicate calculus }
$\alpha \wedge G_m \Rightarrow \varphi_m$
$\Leftarrow$ { Γ_2 }
true

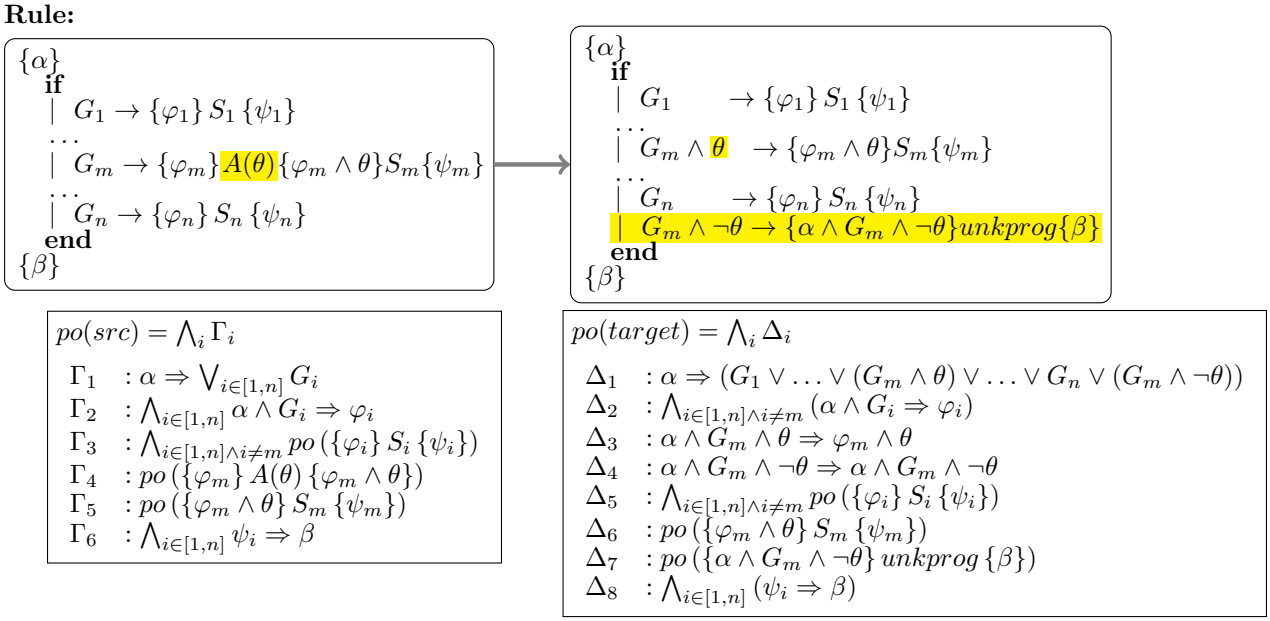
Δ_5 :
$\bigwedge_{i\in[1,n]\wedge i\neq m} po\,(\{\varphi_i \wedge \theta^*\}\, S_i\, \{\psi_i\})$
$\Leftarrow$ { $\varphi_i \wedge \theta^* \Rightarrow \varphi_i$ }
$\bigwedge_{i\in[1,n]\wedge i\neq m} po\,(\{\varphi_i\}\, S_i\, \{\psi_i\})$
$\equiv$ { Γ_3 }
true

Δ_6 :
$po\,(\{\varphi_m \wedge \theta\}\, S_m\, \{\psi_m\})$
$\Leftarrow$ { Γ_4 }
true

Δ_7 :
$\bigwedge_{i\in[1,n]} \psi_i \Rightarrow \beta$
$\equiv$ { Γ_5 }
$true$

B.12. IfGrd Rule

Rule:



$po(src) = \bigwedge_i \Gamma_i$

Γ_1 : $\alpha \Rightarrow \bigvee_{i\in[1,n]} G_i$
Γ_2 : $\bigwedge_{i\in[1,n]} \alpha \wedge G_i \Rightarrow \varphi_i$
Γ_3 : $\bigwedge_{i\in[1,n]\wedge i\neq m} po\,(\{\varphi_i\}\, S_i\, \{\psi_i\})$
Γ_4 : $po\,(\{\varphi_m\}\, A(\theta)\, \{\varphi_m \wedge \theta\})$
Γ_5 : $po\,(\{\varphi_m \wedge \theta\}\, S_m\, \{\psi_m\})$
Γ_6 : $\bigwedge_{i\in[1,n]} \psi_i \Rightarrow \beta$

$po(target) = \bigwedge_i \Delta_i$

Δ_1 : $\alpha \Rightarrow (G_1 \vee \ldots \vee (G_m \wedge \theta) \vee \ldots \vee G_n \vee (G_m \wedge \neg\theta))$
Δ_2 : $\bigwedge_{i\in[1,n]\wedge i\neq m} (\alpha \wedge G_i \Rightarrow \varphi_i)$
Δ_3 : $\alpha \wedge G_m \wedge \theta \Rightarrow \varphi_m \wedge \theta$
Δ_4 : $\alpha \wedge G_m \wedge \neg\theta \Rightarrow \alpha \wedge G_m \wedge \neg\theta$
Δ_5 : $\bigwedge_{i\in[1,n]\wedge i\neq m} po\,(\{\varphi_i\}\, S_i\, \{\psi_i\})$
Δ_6 : $po\,(\{\varphi_m \wedge \theta\}\, S_m\, \{\psi_m\})$
Δ_7 : $po\,(\{\alpha \wedge G_m \wedge \neg\theta\}\, unkprog\, \{\beta\})$
Δ_8 : $\bigwedge_{i\in[1,n]} (\psi_i \Rightarrow \beta)$

Theorem B.12. *IfGrd* rule is correctness preserving.

Proof. We assume the correctness of the *src* program and prove the proof obligations of the target program(Δ_i) separately.

Δ_1 :
$\alpha \Rightarrow (G_1 \vee \ldots \vee (G_m \wedge \theta) \vee \ldots \vee G_n \vee (G_m \wedge \neg\theta))$
$\equiv$ { $(G_m \wedge \theta) \vee (G_m \wedge \neg\theta) \equiv G_m$ }
$\alpha \Rightarrow (G_1 \vee \ldots \vee G_m \vee \ldots \vee G_n)$
$\equiv$ { Γ_1 }
$true$

Δ_2 :
$\bigwedge_{i\in[1,n]\wedge i\neq m} (\alpha \wedge G_i \Rightarrow \varphi_i)$
$\equiv$ { Γ_2 }
$true$

Δ_3 :
$\alpha \wedge G_m \wedge \theta \Rightarrow \varphi_m \wedge \theta$
$\Leftarrow$ { predicate calculus }
$\alpha \wedge G_m \Rightarrow \varphi_m$
$\equiv$ { Γ_2 }
$true$

Δ_4 :
$\alpha \wedge G_m \wedge \neg\theta \Rightarrow \alpha \wedge G_m \wedge \neg\theta$
$\equiv$ { predicate calculus }
$true$

Δ_5 :
$\bigwedge_{i\in[1,n]\wedge i\neq m} po\,(\{\varphi_i\}\, S_i\, \{\psi_i\})$
$\equiv$ { Γ_3 }
$true$

Δ_6 :
$po\,(\{\varphi_m \wedge \theta\}\, S_m\, \{\psi_m\})$
$\equiv$ { Γ_5 }
$true$

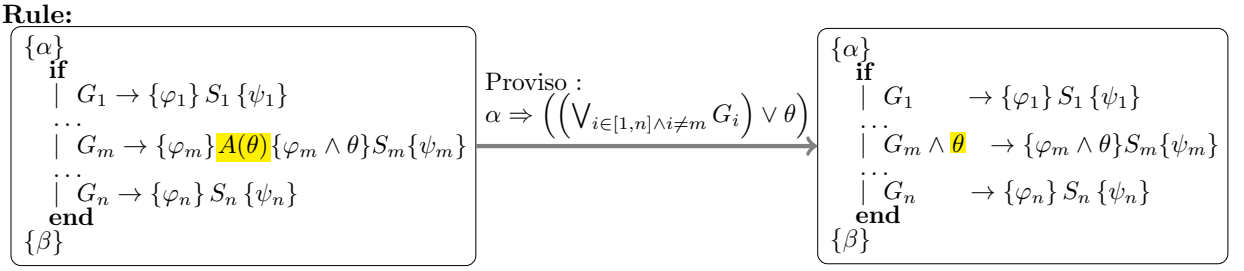
Δ_7 :

$$
\begin{array}{ll}
& po\,(\{\alpha \wedge G_m \wedge \neg\theta\}\, unkprog\, \{\beta\}) \\
\equiv & \quad \{ \text{ definition of } unkProg \ \} \\
& true
\end{array}
$$

$$
\begin{array}{ll}
& \bigwedge_{i\in[1,n]} (\psi_i \Rightarrow \beta) \\
\equiv & \quad \{\ \Gamma_6\ \} \\
& true
\end{array}
$$

B.13. IfGrd2 Rule

Rule:



$po(src) = \bigwedge_i \Gamma_i$

$$
\begin{array}{ll}
\Gamma_1 & : \alpha \Rightarrow \bigvee_{i\in[1,n]} G_i \\
\Gamma_2 & : \bigwedge_{i\in[1,n]} \alpha \wedge G_i \Rightarrow \varphi_i \\
\Gamma_3 & : \bigwedge_{i\in[1,n]\wedge i\neq m} po\,(\{\varphi_i\}\, S_i\, \{\psi_i\}) \\
\Gamma_4 & : po\,(\{\varphi_m\}\, A(\theta)\, \{\varphi_m \wedge \theta\}) \\
\Gamma_5 & : po\,(\{\varphi_m \wedge \theta\}\, S_m\, \{\psi_m\}) \\
\Gamma_6 & : \bigwedge_{i\in[1,n]} \psi_i \Rightarrow \beta
\end{array}
$$

$po(target) = \bigwedge_i \Delta_i$

$$
\begin{array}{ll}
\Delta_1 & : \alpha \Rightarrow (G_1 \vee \ldots \vee (G_m \wedge \theta) \vee \ldots \vee G_n) \\
\Delta_2 & : \bigwedge_{i\in[1,n]\wedge i\neq m} (\alpha \wedge G_i \Rightarrow \varphi_i) \\
\Delta_3 & : \alpha \wedge G_m \wedge \theta \Rightarrow \varphi_m \wedge \theta \\
\Delta_4 & : \bigwedge_{i\in[1,n]\wedge i\neq m} po\,(\{\varphi_i\}\, S_i\, \{\psi_i\}) \\
\Delta_5 & : po\,(\{\varphi_m \wedge \theta\}\, S_m\, \{\psi_m\}) \\
\Delta_6 & : \bigwedge_{i\in[1,n]} (\psi_i \Rightarrow \beta)
\end{array}
$$

IfGrd2 rule: A variation of the *IfGrd* rule(Fig. 15).

Theorem B.13. *IfGrd2* rule is correctness preserving.

Proof. We assume the correctness of the *src* program and the proviso, and prove the proof obligations of the target program(Δ_i) separately.

Δ_1 :

$$
\begin{array}{ll}
& \alpha \Rightarrow (G_1 \vee \ldots \vee (G_m \wedge \theta) \vee \ldots \vee G_n) \\
\equiv & \quad \{ \text{ distributivity } \} \\
& \alpha \Rightarrow \left(\bigvee_{i\in[1,n]} G_i\right) \wedge \left(\left(\bigvee_{i\in[1,n]\wedge i\neq m} G_i\right) \vee \theta\right) \\
\equiv & \quad \{\ \Gamma_1\ \} \\
& \alpha \Rightarrow \left(\bigvee_{i\in[1,n]\wedge i\neq m} G_i\right) \vee \theta \\
\equiv & \quad \{ \text{ proviso } \} \\
& true
\end{array}
$$

Δ_2 :

$$
\begin{array}{ll}
& \bigwedge_{i\in[1,n]\wedge i\neq m} (\alpha \wedge G_i \Rightarrow \varphi_i) \\
\equiv & \quad \{\ \Gamma_2\ \} \\
& true
\end{array}
$$

Δ_3 :

$$
\begin{array}{ll}
& \alpha \wedge G_m \wedge \theta \Rightarrow \varphi_m \wedge \theta \\
\Leftarrow & \quad \{ \text{ predicate calculus } \} \\
& \alpha \wedge G_m \Rightarrow \varphi_m \\
\equiv & \quad \{\ \Gamma_2\ \} \\
& true
\end{array}
$$

Δ_4 :

$$
\begin{array}{ll}
& \bigwedge_{i\in[1,n]\wedge i\neq m} po\,(\{\varphi_i\}\, S_i\, \{\psi_i\}) \\
\equiv & \quad \{\ \Gamma_3\ \} \\
& true
\end{array}
$$

Δ_5 :

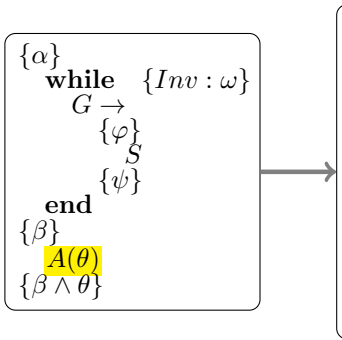
$$
\begin{array}{ll}
& po\,(\{\varphi_m \wedge \theta\}\, S_m\, \{\psi_m\}) \\
\equiv & \quad \{\ \Gamma_5\ \} \\
& true
\end{array}
$$

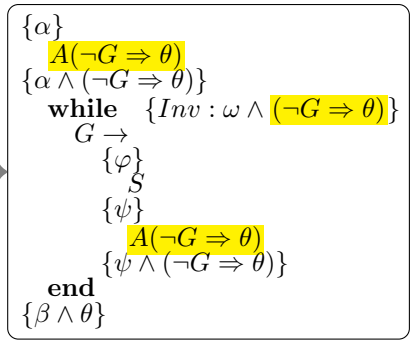
Δ_6 :

$$
\begin{array}{ll}
& \bigwedge_{i\in[1,n]} (\psi_i \Rightarrow \beta) \\
\equiv & \quad \{\ \Gamma_6\ \} \\
& true
\end{array}
$$

B.14. WhileIn Rule

Rule:





$po(src) = \bigwedge_i \Gamma_i$

$\Gamma_1 \quad : \alpha \Rightarrow \omega$
$\Gamma_2 \quad : \omega \wedge G \Rightarrow \varphi$
$\Gamma_3 \quad : po\,(\{\varphi\}\, S\, \{\psi\})$
$\Gamma_4 \quad : \psi \Rightarrow \omega$
$\Gamma_5 \quad : \omega \wedge \neg G \Rightarrow \beta$
$\Gamma_6 \quad : po\,(\{\beta\}\, A(\theta)\, \{\beta \wedge \theta\})$

$po(target) = \bigwedge_i \Delta_i$

$\Delta_1 \quad : po\,(\{\alpha\}\, A(\neg G \Rightarrow \theta)\, \{\alpha \wedge (\neg G \Rightarrow \theta)\})$
$\Delta_2 \quad : \alpha \wedge (\neg G \Rightarrow \theta) \Rightarrow \omega \wedge (\neg G \Rightarrow \theta)$
$\Delta_3 \quad : \omega \wedge (\neg G \Rightarrow \theta) \wedge G \Rightarrow \varphi$
$\Delta_4 \quad : po\,(\{\varphi\}\, S\, \{\psi\})$
$\Delta_5 \quad : po\,(\{\psi\}\, A(\neg G \Rightarrow \theta)\, \{\psi \wedge (\neg G \Rightarrow \theta)\})$
$\Delta_6 \quad : \psi \wedge (\neg G \Rightarrow \theta) \Rightarrow \omega \wedge (\neg G \Rightarrow \theta)$
$\Delta_7 \quad : \omega \wedge (\neg G \Rightarrow \theta) \wedge \neg G \Rightarrow \beta \wedge \theta$

Theorem B.14. *WhileIn* rule is correctness preserving.

Proof. We assume the correctness of the *src* program and prove the proof obligations of the target program(Δ_i) separately.

Δ_1 :
$po\,(\{\alpha\}\, A(\neg G \Rightarrow \theta)\, \{\alpha \wedge (\neg G \Rightarrow \theta)\})$
$\equiv$ { definition of *assume*}
$\alpha \wedge (\neg G \Rightarrow \theta) \Rightarrow \alpha \wedge (\neg G \Rightarrow \theta)$
$\equiv$ { predicate calculus}
true

Δ_2 :
$\alpha \wedge (\neg G \Rightarrow \theta) \Rightarrow \omega \wedge (\neg G \Rightarrow \theta)$
$\equiv$ { predicate calculus }
$\alpha \wedge (\neg G \Rightarrow \theta) \Rightarrow \omega$
$\Leftarrow$ { predicate calculus }
$\alpha \Rightarrow \omega$
$\equiv$ { Γ_1 }
true

Δ_3
$\omega \wedge (\neg G \Rightarrow \theta) \wedge G \Rightarrow \varphi$
$\Leftarrow$ { predicate calculus }
$\omega \wedge G \Rightarrow \varphi$
$\equiv$ { Γ_2 }
true

Δ_4 :
$po\,(\{\varphi\}\, S\, \{\psi\})$
$\equiv$ { Γ_3 }
true

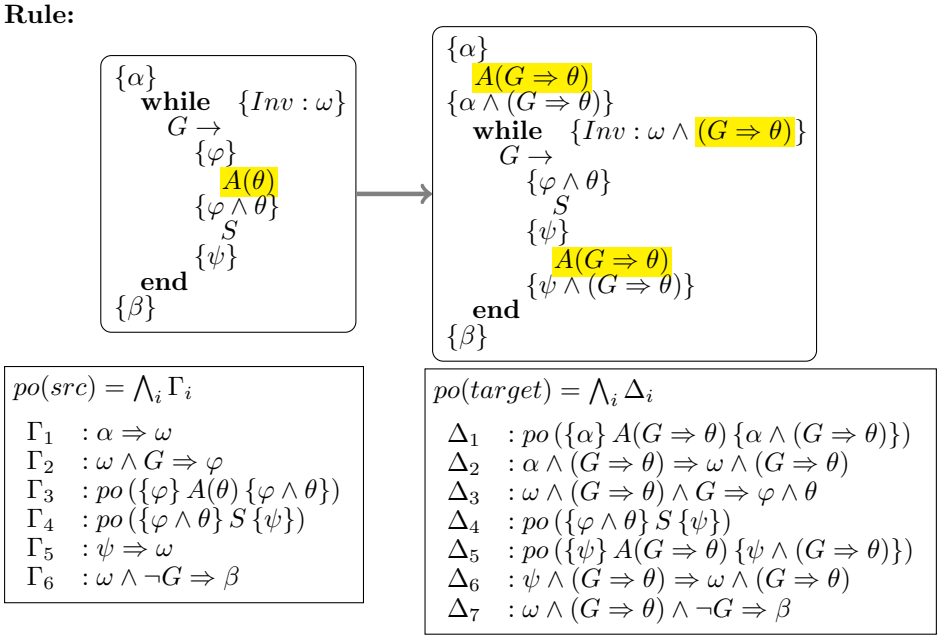
Δ_5 :
$po\,(\{\psi\}\, A(\neg G \Rightarrow \theta)\, \{\psi \wedge (\neg G \Rightarrow \theta)\})$
$\equiv$ { definition of *assume* }
$\psi \wedge (\neg G \Rightarrow \theta) \Rightarrow \psi \wedge (\neg G \Rightarrow \theta)$
$\equiv$ { predicate calculus }
true

Δ_6 :
$\psi \wedge (\neg G \Rightarrow \theta) \Rightarrow \omega \wedge (\neg G \Rightarrow \theta)$
$\equiv$ { predicate calculus }
$\psi \wedge (\neg G \Rightarrow \theta) \Rightarrow \omega$
$\Leftarrow$ { Γ_4 }
true

Δ_7 :
$\omega \wedge (\neg G \Rightarrow \theta) \wedge \neg G \Rightarrow \beta \wedge \theta$
$\equiv$ { predicate calculus }
$\omega \wedge \neg G \wedge \theta \Rightarrow \beta \wedge \theta$
$\Leftarrow$ { predicate calculus }
$\omega \wedge \neg G \Rightarrow \beta$
$\equiv$ { Γ_5 }
true

B.15. WhileStrInv Rule

Rule:



Theorem B.15. *WhileStrInv* rule is correctness preserving.

Proof. We assume the correctness of the *src* program and prove the proof obligations of the target program(Δ_i) separately.

Δ_1 :
$po\,(\{\alpha\}\, A(G \Rightarrow \theta)\, \{\alpha \wedge (G \Rightarrow \theta)\})$
$\equiv$ { definition of *assume* }
$\alpha \wedge (G \Rightarrow \theta) \Rightarrow \alpha \wedge (G \Rightarrow \theta)$
$\equiv$ { predicate calculus }
true

Δ_2 :
$\alpha \wedge (G \Rightarrow \theta) \Rightarrow \omega \wedge (G \Rightarrow \theta)$
$\equiv$ { predicate calculus }
$\alpha \wedge (G \Rightarrow \theta) \Rightarrow \omega$
$\equiv$ { Γ_1 }
true

Δ_3 :
$\omega \wedge (G \Rightarrow \theta) \wedge G \Rightarrow \varphi \wedge \theta$
$\equiv$ { predicate calculus }
$\omega \wedge G \wedge \theta \Rightarrow \varphi \wedge \theta$
$\Leftarrow$ { preciate calculus }
$\omega \wedge \theta \Rightarrow \varphi$
$\equiv$ { Γ_2 }
true

Δ_4 :
$po\,(\{\varphi \wedge \theta\}\, S\, \{\psi\})$
$\equiv$ { Γ_4 }
true

Δ_5 :

$po\,(\{\psi\}\,A(G \Rightarrow \theta)\,\{\psi \wedge (G \Rightarrow \theta)\})$
$\equiv$ { definition of *assume* }
$\psi \wedge (G \Rightarrow \theta) \Rightarrow \psi \wedge (G \Rightarrow \theta)$
$\equiv$ { predicate calculus }
$true$

Δ_6 :

$\psi \wedge (G \Rightarrow \theta) \Rightarrow \omega \wedge (G \Rightarrow \theta)$
$\Leftarrow$ { predicate calculus }
$\psi \Rightarrow \omega$
$\equiv$ { Γ_5 }
$true$

Δ_7 :

$\omega \wedge (G \Rightarrow \theta) \wedge \neg G \Rightarrow \beta$
$\equiv$ { predicate calculus }
$\omega \wedge \neg G \Rightarrow \beta$
$\equiv$ { Γ_6 }
$true$