Combining Top-down and Bottom-up Techniques in Program Derivation

Dipak L. Chaudhari and Om Damani

Indian Institute of Technology Bombay, India.
{dipakc, damani}@cse.iitb.ac.in

Abstract. The traditional stepwise refinement based program derivation methodologies are primarily top-down. Strictly following the top-down program derivation approach may require backtracking resulting in rework. Moreover, the top down approach does not directly help in suggesting the next course of action in case of a failed derivation attempt. In this work we seamlessly incorporate a bottom up assumption propagation technique into a primarily top down derivation methodology. We present new tactics for back-propagating the assumptions made during the top-down phase. These tactics help in reducing the guesswork during the derivations. We have implemented these tactics in a program derivation system. With the help of simple examples, we show how this approach is useful for avoiding backtracking thereby simplifying the derivations.

Keywords: Calculational Style, Program Derivation

1 Introduction

In the calculational style of programming [10,14,13], 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. Although systematic, this approach can still be considered as informal. The refinement calculus [15,1] formalizes this top-down derivation approach. It provides a set of formally verified refinement rules (transformations).

At an intermediate stage in a top down derivation, users have to select an appropriate refinement rule by analyzing the structure of the specification under consideration. However it is not always possible to come up with the right choice on the first attempt. Users often need to backtrack and try out different rules. 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 [15]: "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 the required insight, the *trying again* results in rework. The derived program

fragments (and the discharged proof obligations) need to be recalculated (re-discharged) during the next attempt. The failed attempts also break the flow of the derivations and make them difficult to organize. Moreover, the learnings from the failed attempt are not directly applicable; some guesswork is needed in deciding the future course of action.

The non-linear and lengthy derivations are the major hindrance in widespread adoption of the calculational derivation methodology. In our earlier work [6], we developed a system called CAPS[1] (Calculational Assistant for Programming from the Specifications). The CAPS system provides, among other features, support for backtracking and branching by maintaining the complete derivation tree. Although these features helps in managing the non-linear derivations (along with the failed attempts), the problem of rework still remains. Users have to repeat most of the steps carried out during the failed attempt with slight modifications. In the manual derivations (e.g. as in [14]), users do not actually backtrack and redo the complete derivations; they just figure out the impact of the modifications and add relevant program fragments to maintain the correctness. However, without proper formalization and tool support, this approach remains error prone.

Tools supporting the refinement based formal program derivation (Cocktail [12], Refine [16], Refinement Calculator [4] and PRT [5]) 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 the users to assume predicates and then propagating these predicates to appropriate location is missing.

In this work, we have seamlessly incorporated the bottom-up techniques into a top-down derivation methodology in order to avoid the unnecessary backtrackings and the associated rework. We present derivation tactics for capturing the assumptions made during the top-down phase and subsequently back-propagating these assumptions to appropriate program locations. We have implemented this approach in the CAPS system. With the help of small examples, we explain how this approach avoids unnecessary backtracking, reduces guesswork, and results in simpler derivations in the CAPS system.

2 Motivating Example

In this section, we present a sketch of the calculational derivation for a simple program performed in a top-down manner. We discuss how the top-down approach is insufficient to capture the natural flow of the derivation and results in additional guesswork and rework. Consider the following programming task (adapted from exercise 4.3.4 in [14]. The informal derivation of this problem also appears in [6]).

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

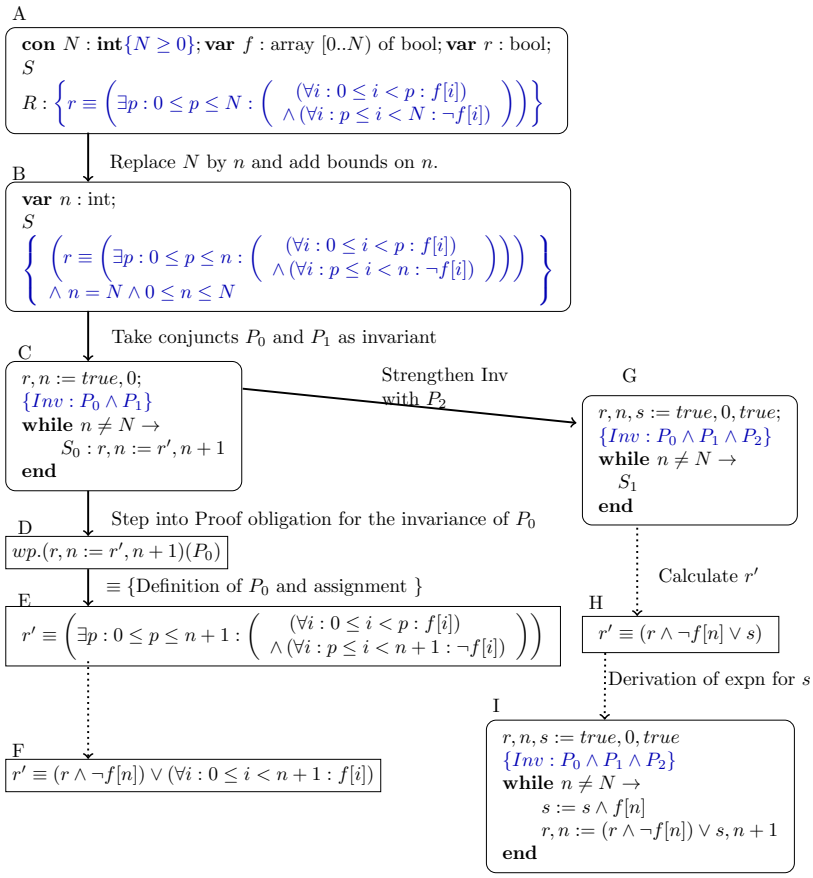


Fig. 1. Sketch of the top-down derivation of the motivating example.
$P_0 : (r \equiv (\exists p : 0 \leq p \leq n : ((\forall i : 0 \leq i < p : f[i]) \wedge (\forall i : p \leq i < n : \neg f[i]))))$
$P_1 : 0 \leq n \leq N$; $\quad P_2 : s \equiv (\forall i : 0 \leq i < n : f[i])$

Let $f[0..N)$ be an array of booleans where N is a natural number. Derive a program for the computation of a boolean variable r such that r is true iff all the true *values in the array come before all the* false *values.*

Fig. 1 depicts the derivation process for this program. We start the derivation by providing the formal specification (node A) of the program. We then apply the *Replace Constant by a Variable* [14] 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 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* heuristics to select conjuncts P_0 and P_1 as invariants and negation of the remaining conjunct $n = N$ as the

guard of the while loop. To ensure termination, we choose to increment variable n by 1 and envision an assignment $r, n := r', n + 1$, where r' is a metavariable. 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 $(r' \equiv (r \wedge \neg f[n]) \vee (\forall i : 0 \leq i < n + 1 : f[i]))$ shown in node F.

At this point, we realize that we can not represent r' in terms of the existing program variables. After analyzing the derivation, we speculate that if we introduce a fresh variable (say s) and maintain $P_2 : s \equiv (\forall i : 0 \leq i < n : f[i])$ as an additional loop invariant then we would be able to express r' in terms of the program variables.

We backtrack to program C, introduce a fresh variable s, and envision a *While* program with the strengthened invariant. For the derivation of the program S_1, we follow the same process as that of S_0. The steps from node G to node H correspond to the calculation of r'. These steps are similar to the calculation of r' in the failed attempt (node E to node F). However, this time, we are able to instantiate r' with the help of the newly introduced invariant P_2. After calculation of r', we proceed further for the derivation of assignment for the variable s. The program can be improved further by strengthening the guard to ensure early termination.

Note that we did not select $s \equiv (\forall i : 0 \leq i < \boldsymbol{n + 1} : f[i])$ as an invariant even though the formula is required at node F. This comes from the observation that it would not be possible to establish the invariant at the start of the loop. Since n is initially 0, assignment $s := f[0]$ would be needed to establish the invariant. However, $f[0]$ is undefined when $N = 0$. Instead we added P_2 as an invariant. Selection of this formula needs foresight that the occurrences of n are textually substituted by $n + 1$ during the derivation (step *D-E*), so we will get the formula we want at node F, if we strengthen the invariant with P_2.

As we saw in this example, some ingenuity is required to figure out the next course of action after a failed derivation attempt. We need to decide the location from where to branch and what new things to try. The backtracking results in rework and breaks the linear flow of the derivation making the derivation complex.

3 Mixing Top-Down and Bottom-Up Approaches

In this section, we first describe the derivation methodology adopted in CAPS and then present our approach for incorporating the bottom-up reasoning in a primarily top-down approach.

3.1 Derivation Methodology

For representing a program fragment and its specification, we use an extension of the Guarded Command Language (GCL) [9] called *AnnotatedProgram*. It is

obtained by augmenting each program construct in the GCL with its precondition and postcondition. It is different from the Hoare triple in a sense that, in addition to the program, every subprogram is also annotated with the pre- and post-conditions. We also introduce a new program construct *UnkProg* to represent an unsynthesized program fragment. Annotated program with a precondition α, a postcondition β, and body S is represented as $\{\alpha\}\, S\, \{\beta\}$. We use the formulas in sorted first-order predicate logic for expressing the precondition and the postcondition of the programs. We adopt the Eindhoven notation [2] for representing the quantified formulas.

Users start a derivation by providing a formal specification of a program and then incrementally transform it into a fully synthesized annotated program by 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 structured derivations, stepping into subcomponents, and backtracking. The system automates most of the mundane tasks and employs the automated theorem provers Alt-Ergo [7], CVC3 [3], SPASS [17] and Z3 [8] for discharging proof obligations. The Why3 tool [11] is used to interface with these theorem provers.

Nature of the transformation rules. In the stepwise refinement based approaches [15,1], 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 post-conditions 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. These rules try to reuse the already derived program fragments and utilize the already discharged proof obligations to ensure correctness.

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 annotated 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 obliga-

tion and instantiating all the metavariables, a tactic called *StepOut* is applied to get an annotated program with all the metavariables replaced by the corresponding instantiations.

3.2 Incorporating the Bottom-up Approach

In order to incorporate the bottom-up approach in the 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 location in the derived program, user can introduce appropriate program constructs to establish the assumptions.

The bottom-up phase has three main steps.

- **Assume:** To derive an annotated program $\{\alpha\}\, UnkProg(1)\, \{\beta\}$, we envision an assignment containing metavariables and step into the proof obligation for the program. We then try to simplify the formula with the objective of guessing the expressions for the metavariables. However, to do so, imagine that we need to assume θ. Instead of backtracking, we just accumulate the assumption and proceed further to derive a program S. In the derived annotated program (Fig. 2), $assume(\theta)$ 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 then propagate the assumption upstream to an appropriate program location. Depending on the program constructs through which the assumption is propagated, the assumed predicate at the new upstream location might be different from the one being propagated.
- **Realize:** Materialize the *assume* statement by converting it to an unknown program fragment which can be derived subsequently from its specification.

$\{\alpha\}$
  $\{\alpha\}$
    $A(\theta)$
  $\{\alpha \wedge \theta\}$
    S
  $\{\beta\}$
$\{\beta\}$

Fig. 2. Result of assuming precondition θ in the derivation of $\{\alpha\}\, UnkProg(1)\, \{\beta\}$

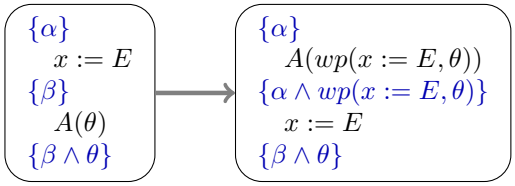
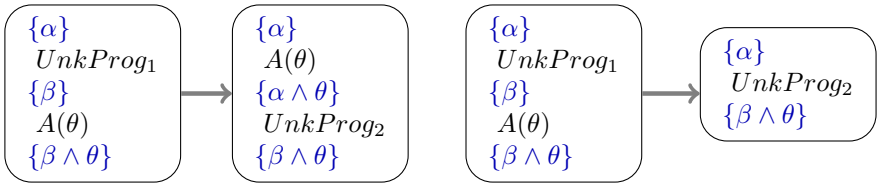


Fig. 3. *AssignmentUp* tactic.



(a) *UnkProgUp* tactic (b) *UnkProgEst* tactic

Fig. 4. *UnkProg* tactics

4 Propagating and Establishing Assumptions

The propagation step mentioned in the previous section is an important step in the bottom up phase. We have developed transformation rules for propagating the assumptions upstream through various program constructs. Some of these rules also establish the assumptions after propagating them. The transformation rules transform an annotated program (source program) into another annotated program (target program) with the same specification (i.e. with the same precondition and postcondition). The transformation rules are verified for correctness: if the source program is correct, then the transformed program is also correct. To prove correctness of a rule, we prove the validity of the formula $PO(S) \Rightarrow PO(T)$ where $PO(S)$ and $PO(T)$ are the proof obligations of the source program S and target program T respectively. The transformation rules are implemented in the CAPS system as tactics. Some of the tactics have associated applicability conditions (also called as *proviso*). A tactic can be applied only when the associated proviso is discharged successfully.

4.1 Atomic Constructs

Atomic constructs are the program constructs that do not have subprograms. In this section, we present some rules for the *Assignment* and *UnkProg* constructs.

Assignment. Fig. 3 shows the *AssignmentUp* tactic for propagating an assumption upwards through an assignment.

UnkProg. Fig. 4(a) shows the *UnkProgUp* tactic 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. User may prefer to establish θ instead of propagating. The *UnkProgEst* tactic (Fig. 4(b)) can be used for this purpose.

We have not presented the rules for the simple constructs like *skip* and *assume*, since the propagation rules for these constructs are simple.

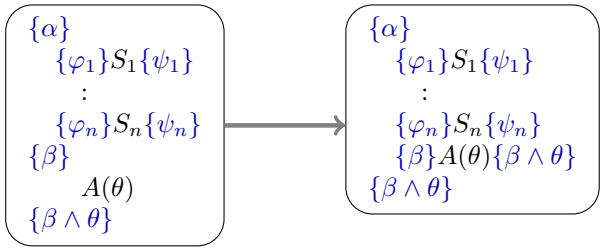


Fig. 5. *CompositionIn* tactic

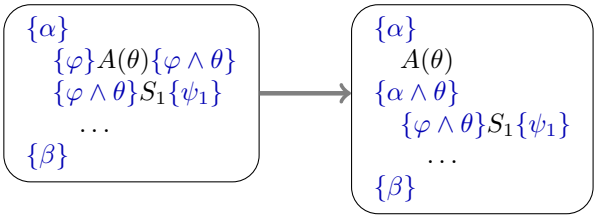


Fig. 6. *CompositionOut* tactic

4.2 Composition

Fig. 5 shows a *Composition* program which is composed of another *Composition* and an *assume(θ)* statement. The *CompositionIn* tactic 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. The *CompositionOut* tactic (Fig. 6) propagates the *assume* statement before the composition statement.

The CAPS system supports nested composition constructs (Composition constructed out of other compositions). Although a nested composition can be collapsed to form a single composition, this construct is useful when we want to apply a tactic to a subcomposition.

Fig. 7 shows the *CompoToIf* tactic 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 command is added to handle the other case. This tactic has a proviso that θ is a valid program expression. This tactic 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.

4.3 If

Fig. 8 shows the *IfIn* tactic. An *assume* statement that appears after the *if* construct in the source program is pushed inside the *if* construct in the tar-

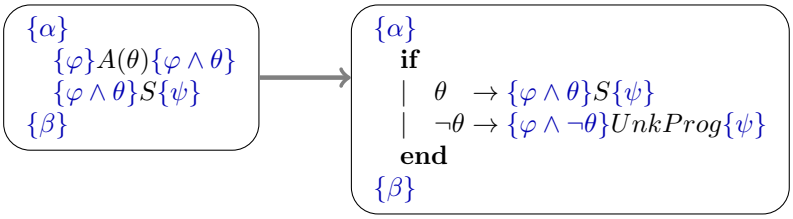


Fig. 7. *CompoToIf* tactic: Transforms a *composition* to an *if* program.

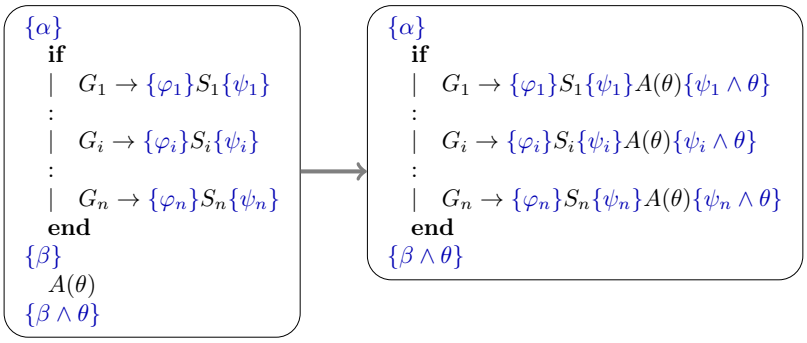


Fig. 8. *IfIn* tactic.

get program. In the target program, θ is assumed at the end of every guarded command.

Fig. 9 shows the *IfOut* tactic. In the source program, θ is assumed before the subprogram S_i, whereas in the target program, θ^* is assumed before the *if* program. Note that θ^* (which is defined as $(G_i \Rightarrow \theta)$) is weaker than θ. As a result of assuming θ^* before the *if* construct, we also strengthen the precondition of the other guarded commands. This strengthening of the precondition is beneficial for the unsynthesized program fragments as it may make the task of derivation simpler.

Instead of propagating the assumption, it can be established by strengthening the guard. This can be achieved by applying the *IfGrd* tactic. An additional guarded command needs to be added to the *if* program to preserve correctness. This tactic does not propagate the assumption; instead it establishes it.

4.4 While

The assumption propagation tactics 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.

***WhileIn* tactic.** Fig. 11 shows the *WhileIn* tactic. The source program has an assumption after the while loop. In order to propagate the assumption θ

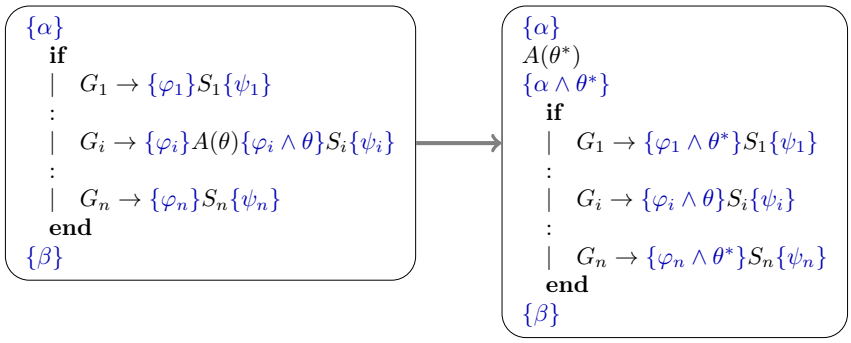


Fig. 9. *IfOut* tactic. ($\theta^\star \triangleq G_i \Rightarrow \theta$)

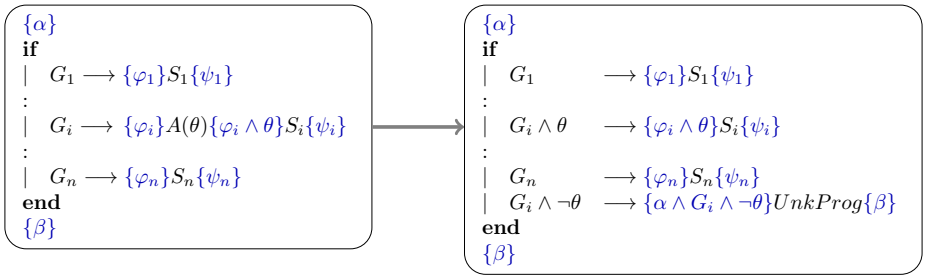


Fig. 10. *IfGrd* tactic

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.

***WhileStrInv* tactic.** Fig. 12 shows the *WhileStrInv* tactic. In the source program, 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* tactic.** Fig. 13 shows the *WhilePostStrInv* tactic. There are two steps in this tactic. In the first step, 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. We have implemented heuristics for eliminating the quantifiers to simplify the formulas. In this tac-

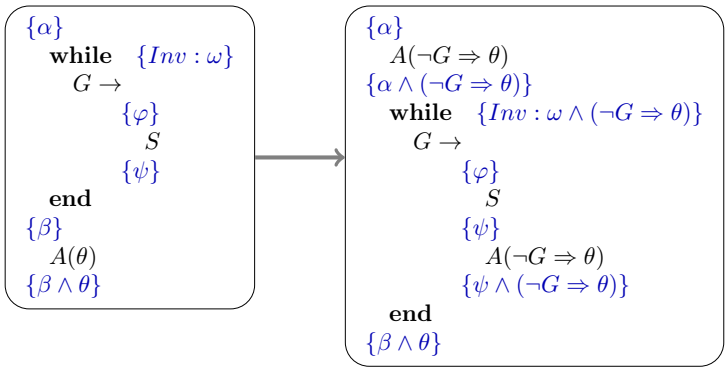


Fig. 11. *WhileIn* tactic: Strengthens the invariant with $\neg G \Rightarrow \theta$

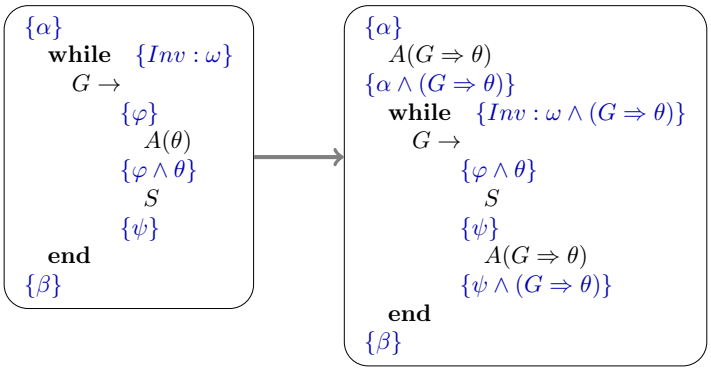


Fig. 12. *WhileStrInv* tactic: Strengthens the invariant with $G \Rightarrow \theta$

tic, we have defined θ^* to be the $sp(S, \theta)$. However, any formula θ^w weaker than the strongest postcondition will also work as long as the program $\{\varphi \wedge \theta^w\}$ $UnkProg\{\varphi \wedge \theta\}$ can be derived.

4.5 Down-propagating the assumed predicates

As we move the assumptions upstream, they become available to various downstream constructs. For example, in the *IfOut* tactic the assumption θ in the i^{th} guarded command is moved upwards before the *if* construct. As a result of this, the propagated assumption θ^* percolates down to the other guarded commands. The predicates can be further propagated downwards using the *StrengthenPost* tactic.

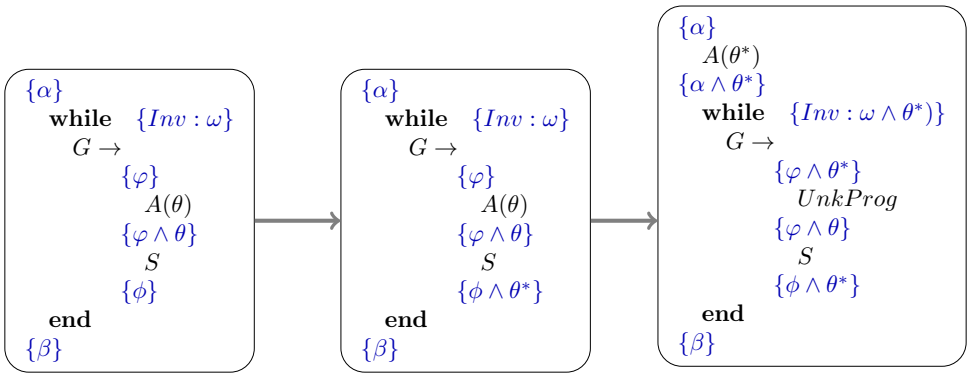


Fig. 13. *WhilePostStrInv* tactic: Strengthens the loop invariant with θ^* where $\theta^* \triangleq sp(S, \theta)$

5 Derivation Examples

5.1 Evaluating Polynomials

A typical derivation involves interleaved instances of up-propagation of the *assume* statements and down-propagation of the assumed predicates. To demonstrate this, 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.

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

We skip the initial tactic applications and directly jump to the program shown in Fig. 14(a). The user has already assumed predicate $P_2 : y = x^n$ during the calculation of r' (not shown). We next apply the *WhileStrInv* tactic to stengthen 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 predicates, we can propagate the missing fragments to an appropriate location and then synthesize them.

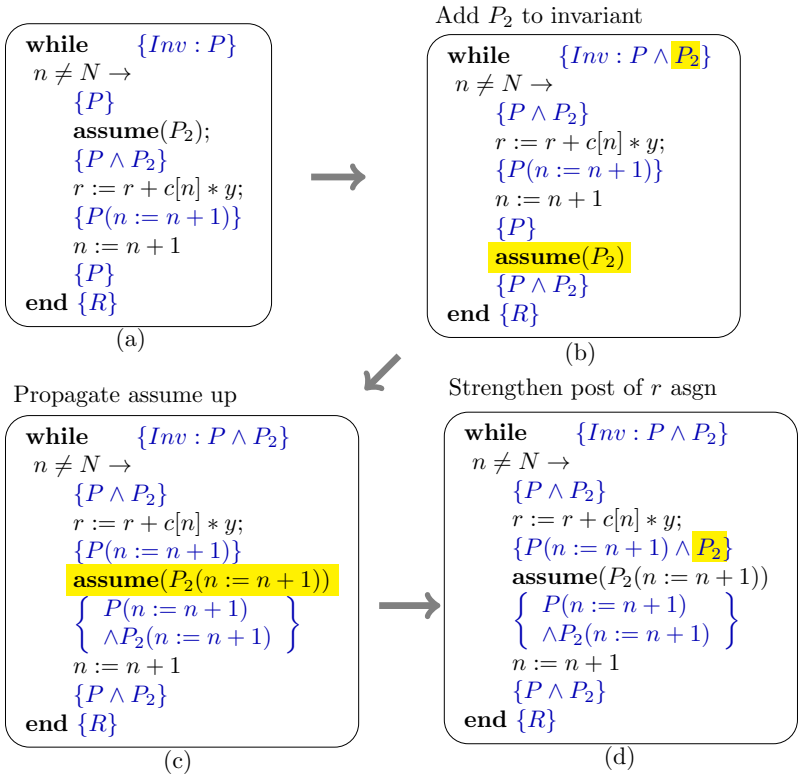


Fig. 14. Some steps in the derivation of a program for the *Horner's rule*. Invariant initializations at the entry of the loop are not shown.

5.2 Back to the Motivating Example

Next, we derive the motivating example from section 2 using our approach. We start from formula F in Fig. 1. At this point, we are not able to express the formula $tt(n+1)$ (where $tt(n) \triangleq (\forall i : 0 \leq i < n : f[i])$) as a program expression. Instead of backtracking, we introduce a fresh variable s and assume the formula $s \equiv tt(n+1)$ and proceed further with the calculation.

$$
\begin{array}{ll}
 & \ldots \\
 & r' \equiv (r \wedge \neg f[n]) \vee (\forall i : 0 \leq i < n+1 : f[i]) \\
\equiv & \{ \text{ Introduce variable } s \text{ and assume } s \equiv (\forall i : \ 0 \leq i < n+1 : \ f[i]) \ \} \\
 & r' \equiv (r \wedge \neg f[n]) \vee s \\
\equiv & \{\text{Step out from formula mode}\}
\end{array}
$$

After stepping out from the formula mode, we arrive at the while loop where the body of the loop contains the *assume* statement.

$\{P_0 \wedge P_1\}$
while $\{Inv : P_0 \wedge P_1\}$
$n \neq N \rightarrow$
$\{P_0 \wedge P_1 \wedge n \neq N\}$
assume$(s \equiv tt(n+1))$
$\{P_0 \wedge P_1 \wedge n \neq N \wedge s \equiv tt(n+1)\}$
$r, n := (r \wedge \neg f[n]) \vee s, n+1$
$\{P_0 \wedge P_1\}$
end
$\{R\}$

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. We can apply the *WhileStrInv* tactic or the *WhilePostStrInv* tactic. Applying the *WhileStrInv* would add $n \neq N \Rightarrow s \equiv tt(n+1)$ as an invariant. With this invariant the initialization problem discussed in section 2 does not arise and this choice results in a different solution. Here, we apply the *WhilePostStrInv* tactic which adds $s \equiv tt(n)$ as an invariant. By applying this tactic, we arrive at the following program.

$\{P_0 \wedge P_1\}$
assume$(s \equiv tt(n))$
$\{P_0 \wedge P_1 \wedge s \equiv tt(n)\}$
while $\{Inv : P_0 \wedge P_1 \wedge s \equiv tt(n)\}$
$n \neq N \rightarrow$
$\{P_0 \wedge P_1 \wedge n \neq N \wedge s \equiv tt(n)\}$
$UnkProg$
$\{P_0 \wedge P_1 \wedge n \neq N \wedge s \equiv tt(n+1)\}$
$r, n := (r \wedge \neg f[n]) \vee s, n+1$
$\{P_0 \wedge P_1 \wedge s \equiv tt(n)\}$
end
$\{R\}$

We can now proceed further with the derivation of the *UnkProg* fragment and the initialization *assume* statement as usual.

Using the bottom-up assumption propagation technique, we could maintain the natural flow of the derivation. This derivation reduces the guesswork and avoids unnecessary branching.

6 Conclusion

We have developed tactics (rules) for up-propagating the information assumed during the top down phase. These tactics have been implemented in the CAPS system. With the help of simple examples we have demonstrated that the seamless integration of the bottom-up and top-down techniques help in reducing the

unnecessary backtrackings and associated rework. The methodology also helps in reducing the guesswork involved in the derivations by allowing the user to delay decisions.

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

References

1. Back, R.J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Graduate Texts in Computer Science, Springer-Verlag, Berlin (1998)
2. Backhouse, R., Michaelis, D.: Exercises in quantifier manipulation. In: Mathematics of program construction. pp. 69–81. Springer (2006)
3. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV. LNCS, vol. 4590, pp. 298–302. Springer (2007)
4. Butler, M., Långbacka, T.: Program derivation using the refinement calculator. In: Theorem Proving in Higher Order Logics: 9th International Conference, volume 1125 of LNCS. pp. 93–108. Springer Verlag (1996)
5. Carrington, D., Hayes, I., Nickson, R., Watson, G.N., Welsh, J.: A tool for developing correct programs by refinement. Tech. rep. (1996), `http://espace.library.uq.edu.au/view/UQ:10768`
6. Chaudhari, D., Damani, O.: Automated theorem prover assisted program calculations. In: Albert, E., Sekerinski, E. (eds.) Integrated Formal Methods, pp. 205–220. Lecture Notes in Computer Science, Springer International Publishing (2014), `http://dx.doi.org/10.1007/978-3-319-10181-1_13`
7. Conchon, S., Contejean, E.: The alt-ergo automatic theorem prover, 2008 `http://alt-ergo.lri.fr`
8. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer (2008)
9. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM 18(8), 453–457 (1975)
10. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, NJ, USA (1997)
11. Filliâtre, J.C., Paskevich, A.: Why3 – Where Programs Meet Provers. In: ESOP'13 22nd European Symposium on Programming. LNCS, vol. 7792. Springer, Rome, Italie (2013)
12. Franssen, M.: Cocktail: A tool for deriving correct programs. In: Workshop on Automated Reasoning (1999)
13. Gries, D.: The Science of Programming. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edn. (1987)
14. Kaldewaij, A.: Programming: The Derivation of Algorithms. Prentice-Hall, Inc., NJ, USA (1990)
15. Morgan, C.: Programming from Specifications. Prentice-Hall, Inc. (1990)
16. Oliveira, M., Xavier, M., Cavalcanti, A.: Refine and gabriel: support for refinement and tactics. In: Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on. pp. 310–319. IEEE (2004)
17. Weidenbach, C., Brahm, U., Hillenbrand, T., Keen, E., Theobalt, C., Topic, D.: SPASS version 2.0. In: Voronkov, A. (ed.) Automated Deduction – CADE-18. Lecture Notes in Computer Science, vol. 2392, pp. 275–279. Springer-Verlag (2002)