Derivation of Imperative Sequential Programs from Formal Specifications

thesis

Submitted in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

submitted by

Dipak Liladhar Chaudhari
(Roll No. 08305901)

Under the supervision of

Prof. Om Damani

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
August, 2016

To my father

Late Shri. Liladhar Atmaram Chaudhari

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Dipak Liladhar Chaudhari

Roll No: 08305901

Abstract

Calculational Style of Programming, while very appealing, has several practical difficulties when done manually. Due to the large number of proofs involved, the derivations can be cumbersome and error-prone. To address these issues, we have developed automated theorem prove assisted program and formula transformation rules, which when coupled with the ability to extract context of a subformula, help in shortening and simplifying the derivations.

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 upstream assumptions are made in an ad hoc fashion and may result in unnecessary rework, or worse, complete exclusion of some of the alternative solutions. We present correctness-preserving 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.

We have developed a program derivation system called CAPS for the interactive, calculational derivation of imperative programs. We have implemented the automated theorem prover assisted rules and the assumption propagation rules in the CAPS system. We show how the CAPS system helps in taking the drudgery out of the derivation process while ensuring correctness.

Keywords: Program Derivation; Correct-by-construction Programming; Assumption Propagation; Annotated Programs; Program Transformations

Contents

Abstract ii

Contents iii

List of Tables vii

List of Figures viii

Nomenclature x

1 Introduction 1

1.1 Objectives 2

1.2 Contributions 3

1.3 Organization of the Thesis 4

2 Calculational Style of Programming 6

2.1 Calculational Style 6

2.2 Preliminaries 7

2.2.1 Hoare Triple, Weakest Precondition, and Strongest Postcondition . 7

2.2.2 Eindhoven Notation 8

2.3 Motivating Example 8

2.4 Teaching Calculational Style of Programming 11

3 Derivation Methodology 14

3.1 Derivation Process 14

3.2 Annotated Programs 15

3.3 Annotated Program Transformation Rules 17

3.4 Formula Transformations 19

4 Theorem Prover Assisted Program Derivation 21

4.1 Harnessing the Automated Theorem Provers 22

4.2 Theorem Prover Assisted Tactics 23

4.2.1 Extracting Context of Subprograms 23

4.2.2 Extracting Context of Subformulas 24

4.2.3 Automation at Tactic Level 26

4.3 Why3 Encoding 29

4.3.1 Encoding Scheme 29

4.3.2 Encoding Function ($\mathcal{E}$) 30

4.3.3 Create a Why3 theory 32

4.3.4 Example 32

4.4 Related Work 34

5 Assumption Propagation 37

5.1 Introduction 37

5.2 *Maximum Segment Sum* Revisited 38

5.2.1 *Maximum Segment Sum* Derivation 38

5.2.2 Ad Hoc Decision Making 40

5.2.3 Motivation for Assumption Propagation 41

5.3 Assumption Propagation 42

5.3.1 Assumption Propagation for Bottom up Derivation 42

5.3.2 Precondition Exploration 43

5.3.3 Rules for Propagating and Establishing Assumptions 43

5.3.4 Adding New Transformation Rules 51

5.3.5 Selecting Appropriate Rules 51

5.3.6 Down-propagating the Assertions 52

5.4 Derivation Examples 52

5.4.1 Evaluating Polynomials 52

5.4.2 Back to the Motivating Example 53

5.5 Related Work 57

6 Correctness of Assumption Propagation Rules **58**

6.1 SkipUp Rule . . . 59
6.2 AssumeUp Rule . . . 60
6.3 AssumeMerge Rule . . . 60
6.4 AssignmentUp Rule . . . 61
6.5 UnkProgUp Rule . . . 62
6.6 UnkProgEst Rule . . . 63
6.7 CompositionIn Rule . . . 63
6.8 CompositionOut Rule . . . 65
6.9 CompoToIf Rule . . . 66
6.10 IfIn Rule . . . 67
6.11 IfOut Rule . . . 68
6.12 IfGrd Rule . . . 70
6.13 IfGrd2 Rule . . . 71
6.14 WhileIn Rule . . . 73
6.15 WhileStrInv Rule . . . 75
6.16 WhilePostStrInv Rule . . . 77

7 CAPS **79**

7.1 Introduction . . . 79
7.2 Graphical User Interface . . . 79
7.3 Textual vs Structured Representation . . . 80
7.4 Focusing on Subcomponents . . . 82
7.5 Selective Display of Information . . . 83
7.6 Maintaining Derivation History . . . 85
7.7 Implementing Assumption Propagation . . . 86
7.8 System Architecture . . . 87
7.9 Using the CAPS System . . . 88
7.9.1 Evaluation . . . 90
7.10 Related Work . . . 91

8 Conclusion and Future Work **92**

Appendix A BigMax Theory in Why3 **95**

Bibilography **97**

List of Publications **103**

Acknowledgments **104**

List of Tables

3.1 Partial correctness proof obligations for *annGCL* constructs 16

4.1 Contextual assumptions 24

List of Figures

2.1 Selected stages in the derivation of the *Maximum Segment Sum* problem. . 9

3.1 Schematic Derivation Tree. 15
3.2 *annGCL* grammar 17
3.3 Program transformation tactics. 18
3.4 A path in the synthesis tree for the Integer Division program 20
3.5 Calculation representation 20

4.1 Focusing on a subformula. 25
4.2 Calculation of initialization assignment in *Integer Division* program 26
4.3 An example of application of the SimplifyAuto tactic 28
4.4 Example of Why3 encoding 35

5.1 Sketch of the derivation of the *Maximum Segment Sum* problem. 39
5.2 Result of assuming precondition θ in the derivation of $\{\alpha\}\, unkprog_1\, \{\beta\}$. . 43
5.3 *SkipUp* rule 44
5.4 *AssumeUp* rule 44
5.5 *AssumeMerge* rule 44
5.6 *UnkProgUp* rule 44
5.7 *UnkProgEst* rule 44
5.8 *AssignmentUp* rule 45
5.9 *CompositionIn* rule 45
5.10 *CompositionOut* rule 45
5.11 *CompoToIf* rule 45
5.12 *IfIn* rule. 47
5.13 *IfOut* rule 47

5.14 *IfGrd* rule 47
5.15 *WhileIn* rule 50
5.16 *WhileStrInv* rule 50
5.17 *WhilePostStrInv* rule 50
5.18 *IfGrd2* rule 51
5.19 Some steps in the derivation of a program for the *Horner's rule*. 54
5.20 Maximum Segment Sum derivations using the assumption propagation rules. 56

7.1 CAPS GUI 80
7.2 Structured representation of a formula 80
7.3 InputPanel 81
7.4 Formula transformations from the derivation of the Binary Search program. 83
7.5 Annotation Modes 84
7.6 Navigating the derivation tree 86
7.7 CAPS Architecture 87

Nomenclature

ATP	Automated Theorem Prover
$A(\theta)$	assume(θ)
$wp(S, R)$	Weakest precondition of program S with respect to postcondition R
$sp(S, P)$	Strongest postcondition of program S with respect to precondition P
$Q(x := E)$	A predicate obtained by substituting expressions E for the free occurrences of variable x in the predicate Q
$po(\mathcal{A})$	Proof Obligation of the annotated program $\mathcal{A}$
$\mathcal{E}$	Encoding function to encode formulas to *Why3* declarations

Chapter 1

Introduction

Importance of software correctness can not be overemphasized in today's world wherein our lives are becoming increasingly dependent on the reliable functioning of computer systems. Software testing is the most predominant approach used in the industry to evaluate correctness. Although testing helps in finding software defects, it does not guarantee absence of them. As the state space of the systems is usually very large, it is infeasible to test the functionality for all the possible inputs.

In contrast to testing, formal verification can prove correctness of systems with respect to their formal specifications. Traditional formal verification work flow typically involves an implementation phase followed by a separate formal verification phase. In such *implement-and-verify* approaches, formal specification does not play an active role in the implementation phase. In reality, systems are formally specified often after the implementation phase and just before the verification stage. Verification efforts involve reconstruction of the correctness arguments which might have been considered, albeit informally, during the implementation phase. For example, for sequential programs, automatically inferring loop invariants is a hard problem. It can be argued however that, for the designer of program, it would be easier to suggest the right invariants during the implementation itself. Moverover, the verification methodologies do not help the programmers during the implementation phase.

Program verification systems like Why3 [FP13], Dafny [Lei10], VCC [CDH$^+$09] and VeriFast [JP08] are trying to bring the verification phase closer to the implementation phase. Programmers can annotate programs with the invariants and the verification systems can run the verification engine immediately after compilation. Although the failed proof

obligations provide some hint, there is no structured help available to the programmer in the actual task of implementing the programs. Programmers often rely on ad hoc use cases and informal reasoning to guess the program constructs.

In contrast to this, in the correct-by-construction style of programming, programs and the correctness proofs are developed hand in hand. The calculational style of programming, which is the subject of this thesis, is one such correct-by-construction methodology wherein proofs are carried out in a particular style, called calculational style, which allows proofs to be presented at right level of granularity. Formal specification plays a central role in this method as programs are incrementally derived by manipulating the formal specification.

1.1 Objectives

Calculational style of programming, while very appealing, has several practical difficulties when done manually. Due to the large number of proofs involved, the derivations can be cumbersome and error-prone. Derivations are often non-linear and involve intricate user interactions making them difficult to manage. To address these issues, we set out to develop a program derivation methodology based on the calculational style of programming with an overall objective of taking the drudgery out of the program derivation process. Our aim has been to automate the mundane tasks without sacrificing the readability that is so characteristic of the calculational style, and to provide support for capturing the creative aspects of the derivation process in the form of program/formula transformation rules.

One gaping hole in the existing literature of program derivation has been the lack of a mechanism for propagating assumptions. 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 upstream assumptions are made in an ad hoc fashion and may result in unnecessary rework, or worse, complete exclusion of some of the alternative solutions. We set out to fill this gap by developing an approach for propagating assumptions through annotated programs while ensuring correctness. Our aim has been to eliminate the unnecessary branchings and associated rework due to the trial and errors involved in establishing the predicates at downstream locations.

Scope of the work. In this thesis, we restrict our attention to the derivation of sequential imperative programs. Large systems are composed of several small modules which in turn are composed of many small programs. We focus on programs at the *programming-in-the-small* level with the basic tenet being that, in David Gries's words, *"ability to develop small programs is a necessary condition for developing large ones — although it may not sufficient"*[Gri87]. We hope that, as argued in [Sol86], the insights gained in research into *programming-in-the-small can to lead to effective, productive research into programming-in-the-large.*

1.2 Contributions

The main contributions of this work are as follows.

Methodological Contributions

Program derivation methodology. We have developed a tactic based interactive program derivation methodology based on the calculational style of program derivation. By providing a unified framework for carrying out program as well as formula transformations, we have kept the derivation style and notation close to the pen-and-paper style of derivation which is known for its rigor and readability.

Theorem prover assisted tactics. We have automated the mundane formula manipulation tasks and exploited the power of automated theorem proving to design powerful transformation rules (tactics) which help in shortening and simplifying the derivations without sacrificing the correctness. We have extended the Structured Calculational Proof format [BGVW97] by making the transformation relation explicit and by adding metavariable support. We have developed tactics for automatically simplifying proof obligation formulas with the help of external theorem provers.

Theoretical Contributions

To address the problem of ad hoc reasoning involved in propagation of assumptions made during a top-down derivation of programs, we have developed correctness preserving 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/preconditions with minimal additional proof efforts. With the help of examples, we demonstrate how these rules help users in avoiding unnecessary rework and also help them explore alternative solutions.

Systems-level Contributions

We have designed and implemented a system, called CAPS, for the calculational derivation of imperative programs. We have implemented the program derivation tactics and the assumption propagation tactics in the CAPS system. In building CAPS, our main emphasis has been on the usability; in particular on being able to model and replay the complex interactions and iterations that usually occur during the manual program derivation.

1.3 Organization of the Thesis

The organization of the thesis is as follows.

Chapter 2: Calculational Style of Programming. In this chapter, we introduce the reader to the calculational style of programming with the help of an example. We discuss the virtues of the methodology and the problems in the adoption of the methodology. We motivate the need for a tool-supported methodology to address the discussed problems.

Chapter 3: Derivation Methodology. We propose an interactive program derivation methodology wherein programs are derived by incrementally applying correctness preserving annotated program transformation rules. We give a high level description of the proposed methodology and introduce notations and definitions which are used in the remainder of the thesis.

Chapter 4. Theorem Prover Assisted Program Derivation. In this chapter, we present the theorem prover assisted tactics which automate various program derivation tasks by employing external theorem provers. We discuss various techniques employed to

make the tactic level theorem prover integration possible. We demonstrate how the theorem prover assisted tactics help in shortening and simplifying the derivations.

Chapter 5: Assumption Propagation. We discuss the problem of ad hoc reasoning in top-down program derivations and its repercussions. To address this problem, we introduce the concept of assumption propagation rules. We present correctness preserving assumption propagation rules for various program constructs. With the help of simple examples, we explain how these rules reduce the ad hoc reasoning and the associated rework during the derivations.

Chapter 6: CAPS. In this chapter, we discuss the architecture of the CAPS system. We discuss various features of the CAPS system and explain how these features help in addressing the common problems. We also share our experience of using the CAPS system in the classroom setting.

Chapter 7: Conclusion and Future Work. We conclude by summarizing the work and outlining a few future research directions.

Appendix A. In this appendix, we prove that the assumption propagation rules presented in Chapter 5 preserve the correctness of annotated programs.

Chapter 2

Calculational Style of Programming

The purpose of this chapter is to introduce the calculational style of programming and to motivate the work presented in this thesis. We start by giving a brief historical account of the development of the calculational style of programming. With the help of an example derivation, we discuss the appeal of the methodology and various problems in its adoption. We discuss our experience in teaching the methodology to sophomore students and present common difficulties faced by the students.

2.1 Calculational Style

During the 1970s, Dijkstra, Feijen, Gries, and others [Dij76, Gri87, DF88] developed an approach to programming wherein the *program and its correctness arguments are designed hand in hand*. With this approach, programming becomes more of a discipline, in a sense that instead of relying solely on intuition and trial-and-error, programs are derived from their formal specifications by applying well-established principles and heuristics. The derived programs are correct-by-construction since correctness is implicit in the derivation. This method was then refined by using a proof format in which a *hint* justifying the transformation is written on a dedicated line between the formulas connected by it. This proof style came to be known as the *calculational* style. The program derivation textbooks [Kal90] and [Coh90] extensively use the calculational style while manipulating correctness proof obligation formulas of partially derived programs. Back and von Wright [BGVW97] refined the calculational proof format further by providing support for hierarchical decomposition of larger proofs into smaller ones.

The calculational program derivation as presented in [Dij76, Gri87, DF88], although rigorous, were not mechanically verifiable. In support of the case for mechanized checking of proofs, Manolios and Moore [MM01] pointed out to some errors in the calculational proofs in Dijkstra's work. The refinement calculus [Mor90, BvW98] formalizes program derivation in the form of a top-down methodology. However, as we will see in the next section, program derivations often do not proceed in strictly top-down manner.

The calculational style is known for its readability and rigor. The calculational derivation helps in understanding the rationale behind the introduction of the program constructs and associated invariants thereby providing more opportunities to explore alternative solutions. This method often results in simple and elegant programs since program constructs are introduced only when logical manipulations show them to be necessary for discharging the correctness proof obligations.

2.2 Preliminaries

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

2.2.1 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.

2.2.2 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 $(\sum\ i:\ 0 \leq i \leq 10:\ i^2)$ 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.

2.3 Motivating Example

In this section, we present a calculational derivation of the well-known *Maximum Segment Sum* problem. This derivation highlights the complex user interactions usually involved in a typical program derivation session. The derivation is based on the derivations given in [Kal90] and [Coh90].

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.

Main steps in the derivation are given in Fig. 2.1. We start the derivation by providing the formal specification of the program as shown in subfigure (b). The postcondition R of the program is as follows:

$$R : r = (Max\ p, q:\ 0 \leq p \leq q \leq N:\ Sum(p, q))$$

$Sum(p,q) : (\Sigma\ i : p \le i < q : A[i])$
$R : r = (Max\ p,q : 0 \le p \le q \le N : Sum(p,q))$
$P_0 : r = (Max\ p,q : 0 \le p \le q \le n : Sum(p,q))$
$P_1 : 0 \le n \le N$
$Q(n) : (Max\ p : 0 \le p \le n : Sum(p,n))$

(a) Term and Predicate definitions

con N : $\{N \ge 0\}$;
con A : array$[0..N)$ of int;
var r : int;
  S
$\{R\}$

(b) Specification

$r, n := 0, 0;$
$\{loopinv : P_0 \wedge P_1\}$
while $(n \neq N)$
  $r := r';$
  $\{P_0\,(n := n+1)\};$
  $n := n+1$
end

(c) After introduction of the loop

1		$wp(r := r', wp(n := n+1, P_0))$
2	$\equiv$	{ definition of P_0 }
3		$wp(r := r', wp(n := n+1, r = (Max\ p,q : 0 \le p \le q \le n : Sum(p,q))))$
4	$\equiv$	{ definition of wp}
5		$wp(r := r', (Max\ p,q : 0 \le p \le q \le n+1 : Sum(p,q)))$
6	$\equiv$	{ definition of wp}
7		$r' = (Max\ p,q : 0 \le p \le q \le n+1 : Sum(p,q))$
8	$\equiv$	$\{q \le n+1 \equiv (q \le n \wedge q = n+1)\}$
9		$r' = (Max\ p,q : (0 \le p \le q \le n) \vee (0 \le p \le q = n+1) : Sum(p,q))$
10	$\equiv$	{ range split }
11		$r' = \left(\begin{array}{l}(Max\ p,q : 0 \le p \le q \le n : Sum(p,q)) \\ max\,(Max\ p,q : 0 \le p \le q = n+1 : Sum(p,q))\end{array}\right)$
12	$\equiv$	{ definition of P_0 }
13		$r' = r\ max\,(Max\ p,q : 0 \le p \le q = n+1 : Sum(p,q))$
14	$\equiv$	{ $q = n+1$ }
15		$r' = r\ max\,(Max\ p,q : 0 \le p \le n+1 : Sum(p,n+1))$
16	$\equiv$	{ define $Q(n)$ as $(Max\ p : 0 \le p \le n : Sum(p,n))$ }
17		$r' = r\ max\ Q(n+1)$
18	$\equiv$	{ introduce fresh variable s; assume $s = Q(n+1)$ }
19		$r' = r\ max\ s$

(d) Calculation of r'

$r, n, s := 0, 0, 0;$
$\{loopinv : P_0 \wedge P_1 \wedge (s = Q(n))\}$
while $(n \neq N)$
  $s := s';$
  $\{s = Q(n+1)\}$
  $r := r\ max\ s;$
  $\{P_0\,(n := n+1)\};$
  $n := n+1;$
end

(e) After introducing statement for updating r

$r, n, s := 0, 0, 0;$
$\{loopinv : P_0 \wedge P_1 \wedge (s = Q(n))\}$
while $(n \neq N)$
  $s := (s + A[n])\ \ max\ 0;$
  $\{s = Q(n+1)\}$
  $r := r\ max\ s;$
  $\{P_0\,(n := n+1)\}$
  $n := n+1$
end

(f) Final derived program

Figure 2.1. Selected stages in the derivation of the *Maximum Segment Sum* problem.

where the symbol Max denotes the quantifier version of the binary infix max operator in the Eindhoven notation and $Sum(p, q)$ is defined as given in subfigure (a).

We introduce a fresh variable n and rewrite the postcondition R as

$$P_0 \wedge (n = N) \wedge P_1$$

where P_0 and P_1 are defined as follows.

$$P_0 : r = (MAX\ p, q :\ 0 \leq p \leq q \leq n :\ Sum(p, q))$$
$$P_1 : 0 \leq n \leq N$$

The predicate P_0 is obtained by replacing constant N with a fresh variable n. We follow the general guideline of adding bounds on the introduced variable n by adding a conjunct P_1 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. Note that the new postcondition $P_0 \wedge (n = N) \wedge P_2$ implies the original postcondition R. This commonly used heuristic is called the *Replace Constant by a Variable*[Kal90]. This heuristic is applied to bring the postcondition in the form of a conjunction.

We then apply the *Take Conjuncts as Invariants*[Kal90] heuristic to select conjuncts P_0 and P_1 as invariants and 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 a program $r := r'; n := n + 1$, 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 subfigure (c). The proof obligation for the invariance of P_0 is :

$$P_0 \wedge P_1 \wedge n \neq N \Rightarrow wp(r := r', wp(n := n + 1), P_0)$$

We now assume P_0, P_1 and $n \neq N$ and manipulate the consequent of the formula with the aim of finding a program expression for the metavariable r'. From the definition of P_0 and the definition of weakest precondition for the assignment construct, we get the following formula (step 7 in subfigure (c)).

$$r' = (Max\ p, q : 0 \leq p \leq q \leq n + 1 : Sum(p, q))$$

To separate out the $q = n + 1$ case, we split the range of the formula by applying the *range split* rule and arrive at the following formula (step 11 in subfigure (c)).

$$r' = \left(\begin{array}{l} (Max\ p, q :\ 0 \leq p \leq q \leq n :\ Sum(p, q)) \\ max\,(Max\ p, q :\ 0 \leq p \leq q = n + 1 :\ Sum(p, q)) \end{array} \right)$$

From the definition of P_0, the first operand on the right hand side is equal to r. On replacing it with r, we arrive at the following formula (step 13 in subfigure (c)).

$$r' = r \ max \ (Max \ p, q : \ 0 \leq p \leq q = n + 1 : \ Sum(p, q))$$

We now apply the *one point* rule and substitute $n + 1$ for q to arrive at the formula (step 15 in subfigure (c)) :

$$r' = r \ max \ \ (Max \ p : \ 0 \leq p \leq n + 1 : \ Sum(p, n + 1))$$

At this point, we realize that we can not represent r' in terms of the existing program variables as the expression on the right hand side involves quantifiers. We define $Q(n)$ as $(Max \ p : \ 0 \leq p \leq n : \ Sum(p, n))$ so that we can write the formula in step 15 as follows.

$$r' = r \ max \ Q(n + 1)$$

After analyzing the derivation, we realize that if we introduce a fresh variable (say s) and maintain $s = Q(n)$ as an additional loop invariant then we can express r' in terms of the program variables. After this manipulation, we can now instantiate r' as $r \ max \ s$.

We now arrive at a program shown in subfigure (e). We have introduced an unknown program S_1 to establish the newly added invariant $s = Q(n)$. For the calculation of s', we follow similar process and arrive at the final program shown in subfigure (f).

As this example shows, the final derived program, even when annotated with the invariants, may not be sufficient to provide the reader with the rationale behind the introduction of the program constructs and the invariants; whole derivation history is required. The calculational derivation involves program transformations as well as formula transformations.

2.4 Teaching Calculational Style of Programming

In the traditional implement-and-verify methodology, there is very little help available to the user in the actual task of programming. In contrast, in the *Calculational Style of Programming* programmers see the program transformation strategy that led to the introduction of a particular programming construct, and hence they understand why the construct was introduced at a particular point in a program. These positive aspects make

the calculational style an attractive option for introducing students to the concepts of formal methods.

In its final report [ADG+01], the ITiCSE 2000 Working Group on Formal Methods Education aspired *to see the concepts of formal methods integrated seamlessly into the computing curriculum.* Fifteen years later that aspiration still remains an aspiration. In our opinion, the major reason for this is the fact that the points of integration identified in the report, in Appendices C and E, come much later in the curriculum. By that time, the students are already used to the informal ways of developing programs and software and the old habits die hard. Ideally formal methods should be introduced as early as possible, particularly when students are just learning how to design programs [Cow10].

Existing attempts in this direction focus on employing formal verification for teaching program correctness [SW14, CGG12, DLC06, Lau04]. The Implement-and-Verify program development methodology involves an implementation phase followed by a separate verification phase. Although the failed proof obligations provide some hint, there is no structured help available to the students in the actual task of implementing the programs. Students often rely on ad hoc use cases and informal reasoning to guess the program constructs.

I have been a teaching assistant of the Program Derivation elective course to sophomores at IIT Bombay. The students' interest in the methodology is reflected in the course feedback where we received 87% score in the last offering of the course. Following two comments exemplify the students' excitement: "*A quite different approach to programming, very innovating and interesting too. Some really great insights.*" and "*We learned many good things. I never thought that program could be derived. The experience was enriching.*" Despite the mostly positive feedback, we also realized that students were facing a number of difficulties in manually (without using any tool support) deriving the programs:

Common Difficulties:

(CD0) Difficulty in understanding formal logic: Used to informal reasoning, students make several mistakes in understanding and applying inference rules.

(CD1) Not checking transformation applicability conditions: Many of the program transformation rules have prerequisites that need to be checked. For example, the $+$ operator distributes over quantified Max only if the range is non-empty. Students often forget to check such conditions.

(CD2) Long derivations: Compared to the guess and test approach, the calculational derivations are longer even for simple programs. Students get restless if the derivation runs too long, leading to more errors.

(CD3) Mistakes made during guessing: Manual derivations often involve small jumps where the unknown program expressions are simple enough to be guessed easily. Students often inadvertently take big steps during guessing, resulting in incorrect program expressions. For example, for program S_1 in subfigure 2.1(f), many students make a jump and guess the value of s' to be $s + A[n]$.

(CD4) Forgetting to add bounds to the introduced variables: It is a general guideline to add bounds for a newly introduced variable, such as the bounds for n in the maximum segment sum problem. Students often forget to add such bounds, and later in the derivation, when the bound constraints are needed, they have to backtrack and take the corrective actions.

(CD5) Forgetting to prove proof obligations: With their focus on unraveling the unknown program fragments, students often forget to prove some of the proof obligations.

(CD6) Problem with organizing derivations: The derivation process is not always linear; it involves multiple iterations involving failed derivation attempts. Students often fail organize the derivation in cases where they need to go back and make some corrective changes. Unorganized derivations often lead to some missing proofs of correctness.

To address the problems discussed above, we set out to develop a tool-supported methodology for calculational derivation of programs. In the next chapter, we discuss the proposed program derivation methodology which tries to retain the positive aspects of the calculational method while trying to address the problems described in this chapter.

Chapter 3

Derivation Methodology

In this chapter, we present a high-level description the methodology. We have implemented this methodology in the CAPS system. Design and implementation details are discussed later in chapter 7.

3.1 Derivation Process

The program derivation methodology that we propose is similar in spirit to the one we followed in the motivating example. We start with the formal specification of the program and incrementally transform it into a fully derived correct program. The process, however, is not linear and users often need to backtrack and try out different options. Fig. 3.1 shows a schematic diagram of a derivation tree. Node 1 is the starting node representing the specification and node 12 represents the final derived program. The intermediate nodes can represent programs or formulas. (The way to transition between these to modes is explained in Section 3.4.) Node 6 and node 9 are the nodes where the user faces some difficulties with the derivation and decides not to carry out the derivation further and prefers to backtrack and branch out. The paths in the derivation tree enclosed in rectangles correspond to transformations on subcomponents of the program/formula nodes. This functionality is discussed later in Section 4.2.

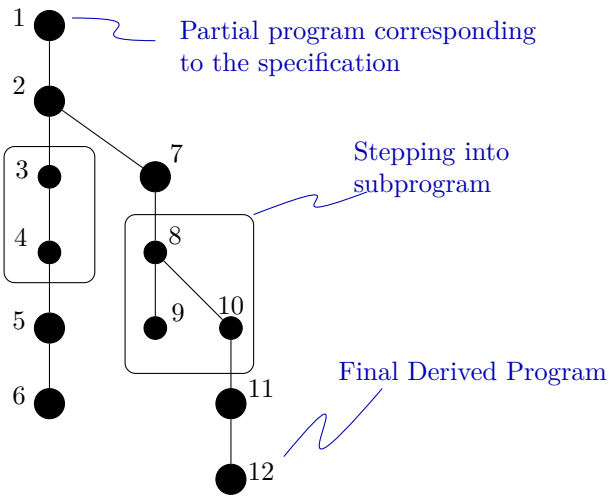


Figure 3.1. Schematic Derivation Tree.

3.2 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. 3.2.

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

Correctness of Annotated Programs

Definition 3.1 (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 3.1) 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* $\{\beta\}$

Table 3.1. 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** $\mid \quad G_1 \to \{\varphi_1\}\, S_1\, \{\psi_1\}$ $\ldots \qquad\qquad \ldots$ $\mid \quad G_n \to \{\varphi_n\}\, S_n\, \{\psi_n\}$ **end** $\{\beta\}$	$po_{coverage} \wedge po_{entry} \wedge po_{body} \wedge po_{exit}$ where, $po_{coverage}: \quad \alpha \Rightarrow \bigvee_{i\in[1,n]} G_i$ $po_{entry}: \quad \bigwedge_{i\in[1,n]} (\alpha \wedge G_i \Rightarrow \varphi_i)$ $po_{body}: \quad \bigwedge_{i\in[1,n]} (po\,(\{\varphi_i\}\, S_i\, \{\psi_i\}))$ $po_{exit}: \quad \bigwedge_{i\in[1,n]} (\psi_i \Rightarrow \beta)$
$\{\alpha\}$ **while** $\{\text{Inv: } \omega\}$ $G \to \{\varphi\}$ S $\{\psi\}$ **end** $\{\beta\}$	$po_{init} \wedge po_{entry} \wedge po_{body} \wedge po_{inv} \wedge po_{exit}$ where, $po_{init}: \quad \alpha \Rightarrow \omega$ $po_{entry}: \quad \omega \wedge G \Rightarrow \varphi$ $po_{body}: \quad po\,(\{\varphi\}\, S\, \{\psi\})$ $po_{inv}: \quad \psi \Rightarrow \omega$ $po_{exit}: \quad \omega \wedge \neg G \Rightarrow \beta$
$\{\alpha\}$ $\{\varphi_1\}\, S_1\, \{\psi_1\}$ $\ldots$ $\{\varphi_n\}\, S_n\, \{\psi_n\}$ $\{\beta\}$	$po_{entry} \wedge po_{body} \wedge po_{joins} \wedge po_{exit}$ where, $po_{entry}: \quad \alpha \Rightarrow \varphi_1$ $po_{body}: \quad \bigwedge_{i\in[1,n]} (po\,(\{\varphi_i\}\, S_i\, \{\psi_i\}))$ $po_{joins}: \quad \bigwedge_{i\in[1,n-1]} (\psi_i \Rightarrow \varphi_{i+1})$ $po_{exit}: \quad \psi_n \Rightarrow \beta$

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

Figure 3.2. *annGCL* grammar

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*".

3.3 Annotated Program Transformation Rules

Definition 3.2 (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

as *proviso*). A rule can be applied only when the associated applicability condition is satisfied.

Definition 3.3 (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.

Tactic: Weaken the Precondition. Input: R Applicability condition: $P \Rightarrow R$	$\{P\}$ **unkprog**$_1$ $\{Q\}$ ➜ $\{P\}$ $\{P\}$ **skip** $\{R\}$; $\{R\}$ **unkprog**$_2$ $\{Q\}$ $\{Q\}$
Tactic: Take Conjuncts as Invariants. Inputs: Invariant conjuncts: R_1 Applicability condition: $P \Rightarrow R_1$	$\{P\}$ **unkprog**$_1$ $\{R_1 \wedge R_2\}$ ➜ $\{P\}$ **while** $\{inv : R_1\}$ $\neg R_2 \rightarrow$ $\{R_1 \wedge \neg R_2\}$ **unkprog**$_2$ $\{R_1\}$ **end** $\{R_1 \wedge R_2\}$

Figure 3.3. Program transformation tactics.

The program transformation tactics are based on the refinement rules from the refinement calculus [BvW98, Mor90] and the high level program derivation heuristics from the literature on calculational program derivation [Kal90, Coh90]. For example, consider the program transformation tactics shown in Fig. 3.3. The *Weaken the Precondition* tactic captures the Hoare triple rule "$\{R\}\, S\, \{Q\}$ and $(P \Rightarrow R)$ implies $\{P\}\, S\, \{Q\}$" whereas the *Take Conjuncts as Invariants* tactic captures the program derivation heuristics with the same name in [Kal90].

3.4 Formula Transformations

As we saw in Section 2.3, program derivation often involves guessing the unknown program fragments in terms of placeholders and then deriving program expressions for the placeholders in order to discharge the correctness proof obligations. We use metavariables to represent the placeholders.

Program and Formula modes. Some steps in the derivations involve transformation of *annGCL* programs whereas others involve transformation of proof obligation formulas. We call these two modes of the derivation as *program mode* and *formula mode* respectively. In order to emulate this functionality in a tactic based framework, we devised a tactic called *StepIntoPO*. On applying this 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.

Example. Fig. 3.4 shows a path in the derivation tree corresponding to the derivation of the Integer Division program (compute the quotient (q) and the remainder (r) of the integer division of x by y where $x \geq 0$ and $y > 0$). Node *n1* in the derivation tree represents an assignment program which contains a metavariable q'. In order to discharge the corresponding proof obligation, the user applies a *StepIntoPO* tactic resulting in a formula node *n2*. The task for the user now is to derive an expression for q' that will make the formula valid. On further transformations, the user arrives at node *n3* from which

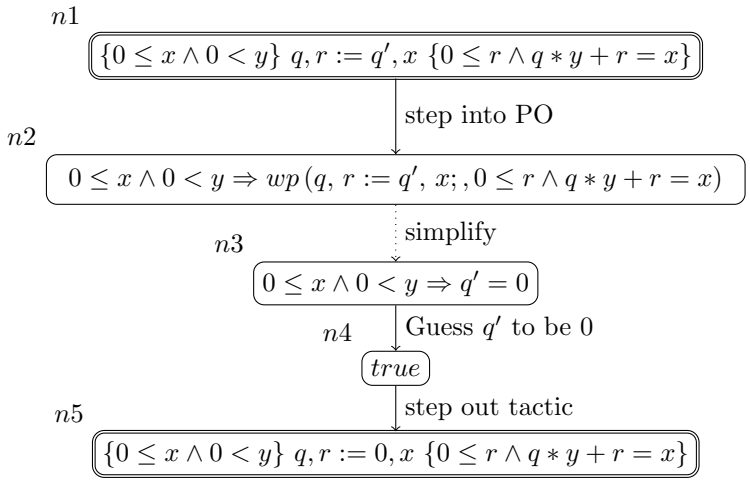


Figure 3.4. A path in the derivation tree for the Integer Division program. The *StepIntoPO* tactic is used to create a formula node corresponding to the proof obligation of the program node.

	F_0
R	{ hint justifying F_0 R F_1 }
	F_1
R	{ hint justifying F_1 R F_2 }
...	...
R	{ hint justifying F_{n-1} R F_n }
	F_n

Figure 3.5. Calculation representation

it is easier to instantiate q' as "0". Finally, the application of *StepOut* tactic results in a program node *n5* where the metavariable q' is replaced with the instantiated expression "0".

Formula Transformations. We adopt a transformational style of inference wherein a formula F_0 is transformed step by step while preserving a reflexive and transitive relation R. Because of the transitivity of R, the sequence of transformations F_0 R F_1 R, ... R F_n implies that F_0 R F_n holds. This derivation is represented in the calculational notation as shown in Fig. 3.5.

Note that the relation maintained at an individual step can be stronger than the overall relation as the sequence of transformations F_0 R_0 F_1 R_1, ... R_{n-1} F_n implies F_0 R F_n, provided relation R_i is at least as strong as the relation R for all i from 0 to $n-1$.

Chapter 4

Theorem Prover Assisted Program Derivation

Having explained our general program derivation methodology in the previous chapter, we now turn our focus towards automating various program derivation tasks by employing external theorem provers. With the help of simple examples, we show how the theorem prover assisted tactics help in shortening and simplifying the derivations thereby taking the drudgery out of the derivation process.

As we saw in Chapter 2, calculational proofs can be cumbersome and error-prone. Manolios and Moore have even found errors in some of the calculational proofs in Dijkstra's work. In support of the case for mechanized checking of calculational proofs, they appeal:

> "The calculational proof community has spent decades exploring the basic issues, refining proof methods, settling on format, and so on. Our advice to the community then is simple: implement a program to check your proofs." [MM01]

Although the appeal is in the context of calculational proofs, the correctness concerns discussed there are applicable to calculational derivations as well. We have addressed these concerns by designing theorem prover assisted program derivation tactics. Proofs in program derivations have an added complexity due to the presence of unsynthesized terms (metavariables) in the formulas. Additionally, the task in discharging proofs is to prove the proof obligation formulas *valid* whereas the task in program derivation is to find the unknown terms (program fragments) to make the formulas *valid*. We have extended the *Structured Calculational Proof* format [BGVW97] by making the transformation relation

explicit and by adding metavariable support.

4.1 Harnessing the Automated Theorem Provers

Readability of the calculational style comes from its ability to express all the important steps in the derivation, and at the same time being able to hide the secondary steps. By secondary steps, we mean the steps that are of secondary importance in deciding the direction of the derivation. For example, the steps involved in the proof for the justifications of the transformations do not change the course of the derivation. These justifications, when obvious, are often stated as hints without explicitly proving them. However, when the justifications are not obvious, it might take several steps to prove them. During the pen-and-paper calculational derivations, the transformation steps are kept small enough to be verifiable manually. Doing low level reasoning involving simple propositional reasoning, arithmetic reasoning, or equality reasoning (replacing equals by equals), can get very long and tedious if done in a completely formal way. Moreover, the lengthy derivations involving the secondary steps often hamper the readability. In such cases, there is a temptation to take long jumps while doing such derivations manually (without a tool support) resulting in correctness errors. With the help of automated theorem provers(ATPs), however, we can afford to take long jumps in the derivation without sacrificing the correctness.

Many common proof paradigms like proof by contradiction, case analysis, induction, etc., are not easily expressed in a purely calculational style. Although, with some effort, these proofs can be handled by the structured calculational approach [BGVW97], employing automated theorem provers greatly simplifies the proof process. We use ATP assisted tactics to automate transformation steps that may not always be amenable to the calculational style.

The template based program synthesis approaches [GJTV11], [SLTB$^+$06], [SGF10] take the specification and the syntactic template of the program as an input and automatically generate the whole program. In contrast, we are interested not just in the final program but also in the complete derivation as it helps in understanding the rationale behind the introduction of the program constructs and the associated invariants. Therefore, we employ the automated theorem provers at a much lower level in an interactive setting. This choice gives users more control to explore alternative solutions since all the design

decisions are manifest in the derivation in the form of tactic applications.

To carry out various proof tasks, we have integrated automated theorem provers (Alt-Ergo [CC], CVC3 [BT07], SPASS [WBH+02] and Z3 [DMB08]) with the CAPS system (refer details about the integration in Section 7.8).

4.2 Theorem Prover Assisted Tactics

In order to integrate ATPs at local level, we first need to extract the context of the subprogram/subformula under consideration. The extracted context can then be used as assumptions while discharging the corresponding proof obligations.

4.2.1 Extracting Context of Subprograms

A partially derived program at some intermediate stage in the program derivation may contain multiple unsynthesized subprograms. Users may want to focus their attention on the derivation of one of these unknown subprograms. The derivation of a subprogram is, for the most part, independent of the rest of the program. Hence it is desirable to provide a mechanism wherein all the contextual information required for the derivation of a subprogram is extracted and presented to users so that they can carry out the derivation independently of the rest of the program. For example, in Fig. 2.1, the user has focused on the fragment $r := r'$ in subfigure (c) and calculated r' separately as shown in subfigure (d).

The activity of focusing on a subprogram is error-prone if carried out without any tool support. In Fig. 2.1(e), subprogram $s := s'$ is added to establish $P_2(n := n + 1)$. We do not recalculate r' since the assumptions during the derivation of r' (invariant P_0 and P_1) still continue to hold since variables r and n are not modified by assignment to s. Users have to keep this fact in mind while calculating s' separately. Due care must be taken during manual derivation to ensure that after any modification, the earlier assumptions still continue to hold. With every program fragment, we associate its full specification (precondition, postcondition) and the context. Since the precondition and postcondition of each program construct are made explicit, users can focus on transforming a subprogram in isolation.

Table 4.1. Contextual assumptions: The R-preserving transformation from $F[f]$ to $F[f']$ under the assumptions Γ can be achieved by r-preserving transformation from f to f' under the assumptions Γ'. (It is assumed that Γ does not contain a formula with i as a free variable. This is ensured during the derivation by appropriately renaming the bound variables.)

F[f]	**R**	**r**	Γ'
$\boxed{A} \wedge B$	$\equiv$	$\equiv$	$\Gamma \cup \{B\}$
	$\Rightarrow$	$\Rightarrow$	
	$\Leftarrow$	$\Leftarrow$	
$\boxed{A} \vee B$	$\equiv$	$\equiv$	$\Gamma \cup \{\neg B\}$
	$\Rightarrow$	$\Rightarrow$	
	$\Leftarrow$	$\Leftarrow$	
$\neg \boxed{A}$	$\equiv$	$\equiv$	Γ
	$\Rightarrow$	$\Leftarrow$	
	$\Leftarrow$	$\Rightarrow$	
$\boxed{A} \Rightarrow B$	$\equiv$	$\equiv$	$\Gamma \cup \{\neg B\}$
	$\Rightarrow$	$\Leftarrow$	
	$\Leftarrow$	$\Rightarrow$	
$B \Rightarrow \boxed{A}$	$\equiv$	$\equiv$	$\Gamma \cup \{B\}$
	$\Rightarrow$	$\Rightarrow$	
	$\Leftarrow$	$\Leftarrow$	

F[f]	**R**	**r**	Γ'
$\boxed{A} \equiv B$	$\equiv$	$\equiv$	Γ
	$\Rightarrow$	$\equiv$	
	$\Leftarrow$	$\equiv$	
$\left(\forall i : \boxed{R.i} : T.i\right)$	$\equiv$	$\equiv$	$\Gamma \cup \{\neg T.i\}$
	$\Rightarrow$	$\Leftarrow$	
	$\Leftarrow$	$\Rightarrow$	
$\left(\exists i : \boxed{R.i} : T.i\right)$	$\equiv$	$\equiv$	$\Gamma \cup \{T.i\}$
	$\Rightarrow$	$\Rightarrow$	
	$\Leftarrow$	$\Leftarrow$	
$\left(\forall i : R.i : \boxed{T.i}\right)$	$\equiv$	$\equiv$	$\Gamma \cup \{R.i\}$
	$\Rightarrow$	$\Rightarrow$	
	$\Leftarrow$	$\Leftarrow$	
$\left(\exists i : R.i : \boxed{T.i}\right)$	$\equiv$	$\equiv$	$\Gamma \cup \{R.i\}$
	$\Rightarrow$	$\Rightarrow$	
	$\Leftarrow$	$\Leftarrow$	

4.2.2 Extracting Context of Subformulas

As the proof obligation formulas contain metavariables, it is important to provide a functionality for focusing on subformulas, so that the focused subformulas can be simplified with the help of theorem provers. Besides the obvious advantage of restricting attention to the subformula, this functionality also makes the additional contextual information available to the user which can be used for manipulating the subformula.

We adopt a style of reasoning similar to the window inference proof paradigm [Gru92], [Gru93], [RS93]. Our implementation differs from the stack based implementation in [Gru93] since we maintain the history of all the transformations.

Let $F[f]$ be a formula with an identified subformula f and Γ be the set of current assumptions. Now, we want to transform the subformula f to f' (keeping the rest of the formula unchanged) such that $F[f]\, R\, F[f']$ holds where R is a reflexive and transitive relation to be preserved. The relationship to be preserved (r) and the contextual assumptions that can used (Γ') during the transformation of f to f' are governed by the following

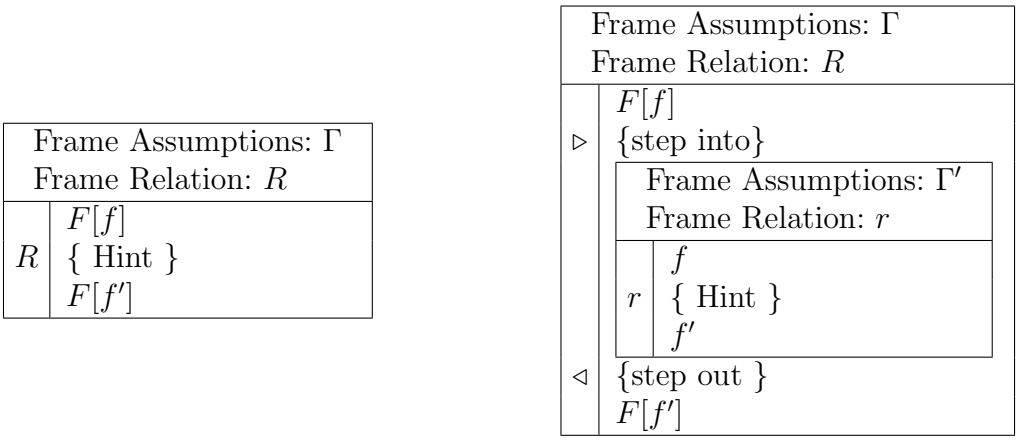


Figure 4.1. Focusing on a subformula.

inference pattern [vW98].

$$\frac{\Gamma' \vdash f\ r\ f'}{\Gamma \vdash F[f]\ R\ F[f']} \tag{4.1}$$

Table 4.1 lists the assumptions Γ' and the relation r for a few combinations of $F[f]$ and R. The *StepInTactic* applications can be chained together. For example, if we want to transform $A \wedge B \Rightarrow C$ while preserving implication ($\Rightarrow$) relation, we may focus on the subformula A and preserve reverse implication ($\Leftarrow$) assuming $\neg C$ and B.

Our representation is an extension of the *Structured Calculational Proof* format [BGVW97]. In our representation, the transformations on the subformulas are indented and contextual information is stored in the top row of the indented derivation. Each indented derivation is called a *frame*. Besides the assumptions, a frame also stores the relation to be maintained by the transformations in the frame. Tactic applications ensure that the actual relation maintained is at least as strong as the frame relation. Fig. 4.1 shows two calculational derivations. In the first derivation, formula $F[f]$ is transformed into $F[f']$ by preserving relation R. The same outcome is achieved in the second derivation by focusing on the subformula f and transforming it to f' under the assumptions Γ' while preserving r provided $F[\_]$, Γ, R, Γ', and r are in accordance with Equation 4.1.

Fig. 4.2 shows application of this tactic in CAPS. The user focuses on a subformula and manipulates it further while preserving the equivalence ($\equiv$) relation (which is stronger than the frame relation $\Leftarrow$). The assumptions extracted from the context can be used during the transformation of the subformula.

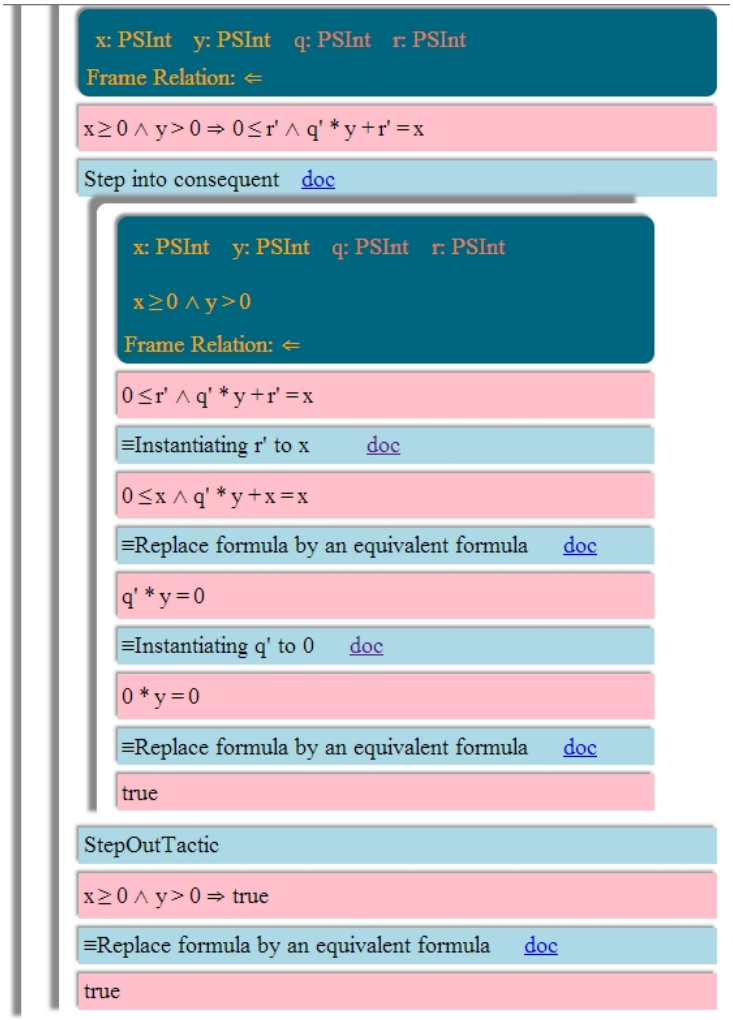


Figure 4.2. Calculation of initialization assignment $(q, r := 0, x)$ to establish invariant $0 \leq r \wedge q * y + r = x$ in the derivation of *Integer Division* program

4.2.3 Automation at Tactic Level

We now describe various program derivation tasks which are automated with the help of ATPs

Tactic Applicability Conditions. Some of the tactics are purely syntactic manipulations and are correct by construction whereas others have applicability conditions which need to be verified. The *Split Range Tactic* and the *Empty Range Tactic* for the universal quantifier are shown below.

Split Range Tactic

$$
\begin{array}{ll}
 & (\forall\ i:\ P.i \vee Q.i:\ T.i) \\
\equiv & \{\ \text{Split Range}\ \} \\
 & (\forall\ i:\ P.i:\ T.i) \wedge (\forall\ i:\ Q.i:\ T.i)
\end{array}
$$

Empty Range Tactic

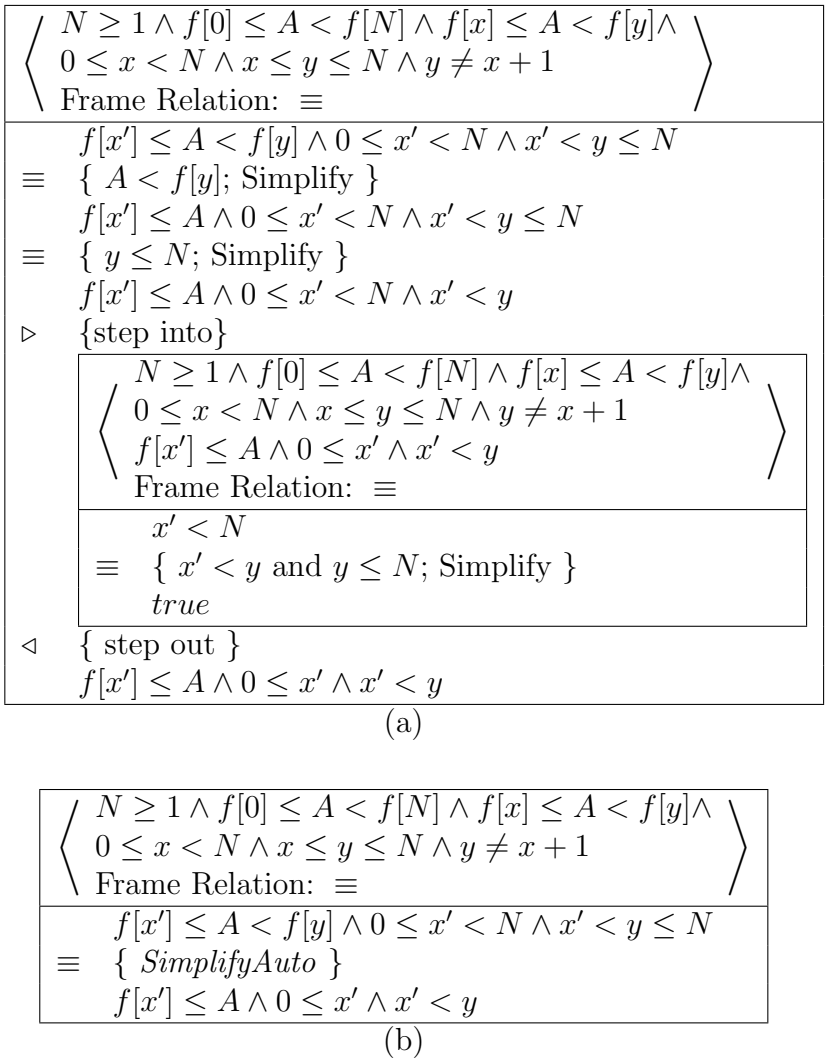
$$
\begin{array}{ll}
 & (\forall\ i:\ R.i:\ T.i) \\
\equiv & \{\ \text{Empty Range; } R.i \equiv false\} \\
 & true
\end{array}
$$

The *Split Range Tactic* does not have any applicability condition whereas the *Empty Range Tactic* has an additional applicability condition ($\forall i :: R.i \equiv false$) (i.e. $R.i$ is unsatisfiable). Such conditions are automatically verified in the CAPS system using ATPs. Note that in the absence of this integration, one way to accomplish this transformation – at the risk of making the derivation lengthy – is to focus onto $R.i$ and transform it to $false$ and then step out and transform the whole formula to $true$.

Proofs involving no metavariables. Proofs of formulas free of metavariables are good candidates for full automation. In Section 2.3, we skipped the proof for preservation of the loop invariant $P_1 : 0 \leq n \leq N$. This invariance proof obligation does not involve any metavariable, and hence is not of interest from the synthesis point of view. We automatically prove such proof obligations with the help of ATPs. In case the automated provers fail to discharge the proof obligation or prove it invalid, we have to revert back to the step-by-step calculational way of proving.

Verifying the transformations. During the calculational derivations, it is sometimes easier to directly specify the desired formula and verify it to be correct instead of deriving the formula in a purely interactive way. We have a *VerifiedTransformation* tactic that serves this purpose. This tactic takes the formula corresponding to the next step and the relation to be maintained as an input and verifies if the relation holds. This functionality is similar in spirit to the verified calculations functionality [LP14] in Dafny. The derivation in Fig. 4.2 has three instances of application of this tactic (labeled by a hint "Replace formula by an equivalent formula"). This tactic helps in reducing the length of the derivations.

The *VerifiedTransformation* tactic is also helpful in discharging proofs which are not amenable to the calculational style. Many common proof paradigms (like proof by contradiction, case analysis, induction, etc.) are difficult to express in a purely calculational style [BGVW97]. Although, with some effort, these proofs can be discharged by using the functionality for focusing on subcomponents (which is based on the structured calculational



(a)

(b)

Figure 4.3. (a) Excerpt from the derivation of the binary search program using multiple applications of the *Simplify* tactic, (b) The same derivation performed using the *SimplifyAuto* tactic.

approach in [BGVW97]), employing the automated theorem provers greatly simplifies the derivation. Note that this tactic is different from the earlier tactics; in all the other tactics a formula is transformed in a specific way and only the applicability condition is proved automatically, whereas in this tactic, users directly specify an arbitrary formula as the transformed form of a given formula and the tactic application just verifies the correctness of the transformation.

Automatic Simplification of Formulas. The *SimplifyAuto* tactic recursively focuses on subformulas in bottom-up fashion and verifies – with the help of ATPs – if the subformulas are valid/invalid. The same effect can be achieved by interactively focusing on each subformula, proving/disproving the subformula under the modified assumptions, and

then simplifying the formula. The *SimplifyAuto* tactic automates this process resulting in simpler derivations in many cases. Fig. 4.3(a) shows an excerpt from the derivation of the binary search program whereas Fig. 4.3(b) shows how the same outcome can be accomplished in a single step using the *SimplifyAuto* tactic.

4.3 Why3 Encoding

Why3 [FP13] is a framework for deductive program verification. The base logic of *Why3* is an extension of first-order logic with polymorphism, algebraic data types, and inductive predicates. The framework provides two languages: a logic language (Why) and a ML-like programming language (WhyML). We use only the logic language since we are using the framework only for the purpose of interfacing with the theorem provers. We assume basic familiarity with the logic language (refer the *Why3* reference manual [Why] for details). We use the common interface provided by the *Why3* framework to communicate with various automated theorem provers (Alt-Ergo, CVC3, SPASS, and Z3). Using *Why3* as an interface saves us from dealing with the different logical languages and predefined theories of various theorem provers.

We encode every proof obligation formula as a *Why3* theory and invoke the *Why3* framework in the background to prove the validity. A *Why3* theory is a list of declarations which introduce new functions and predicates, state axioms and goals. We have encoded the arithmetic quantifiers ($Max, Min, Sum,$ and $Prod$) as *Why3* theories. The *Why3* theory for the Max quantifier is given in Appendix A.

In this section, we discuss the encoding of expressions involving the arithmetic quantifiers. Encodings of quantifier-free expressions and expressions involving boolean quantifiers ($\forall, \exists$) is straightforward and not discussed here.

4.3.1 Encoding Scheme

Consider a quantified expression $(OP\ i :\ R :\ T)$ where OP is the quantifier version of a symmetric and associative binary operator `op: int -> int -> int`, i is a list of quantified variables, R is the *Range* - a boolean expression involving the quantified variables, and T is the *Term* - an integer expression.

The range R and the term T of the quantified expression are encoded as functions

with the following types.

```
constant r: int -> bool
constant t: int -> int
```

The quantifier OP is encoded as a function with the following declaration.

```
constant bigOp: (int -> bool) -> (int -> int) -> int
```

And finally, the quantified expression $(OP\ i :\ R :\ T)$ is encoded as a function application as shown below.

```
bigOp r t
```

The functions `r` and `t` are then defined inductively by encoding the expressions R and T respectively and `bigOp` is axiomatized by a set of axioms. The final encoding thus consists of two parts: the *Why3* expression and the declarations for the introduced functions.

4.3.2 Encoding Function ($\mathcal{E}$)

We define an encoding function $\mathcal{E} : (f, bvs) \mapsto (expr, decls)$ which transforms a tuple of a first order expression (f) and a list of context bound variables (bvs) into a tuple of corresponding expression $(expr)$ and declarations $(decls)$ in the *Why3* language. The context bound variables argument (bvs) comes into play for nested quantified terms. It represents a list of accumulated bound variables from the context of a subexpression. For example, in the quantified expression $(Max\ p : 0 \leq p \leq n : (Max\ q : 0 \leq p \leq q \leq n : (\sum i : p \leq i < q : arr[i])))$, for the outermost Max term there are no context bound variable, for the inner Max term, p is the only context bound variable, and for the $\sum$ term p and q are the context bound variables.

We now present the steps for encoding a quantified expression $(OP\ i :\ R :\ T)$ where OP with a single context bound variable bv. To simplify the presentation, we restrict to a single bound variable.

Encode the *range* and the *term*.

We first encode the range R and the term T. When encoding the range and the term, we need to add the bound variable i to the context bound variables.

$$\text{Let } (expr_r, decl_r) = \mathcal{E}(R, [bv, i]) \text{ and}$$
$$(expr_t, decl_t) = \mathcal{E}(T, [bv, i]).$$

After encoding the R and T expressions, we declare the functions `r` and `t` as follows.

```
<decl_r>
<decl_t>
constant r: int -> int -> bool = (\bv: int, i: int. <expr_r>)
constant t: int -> int -> int = (\bv: int, i: int. <expr_t>)
```

Note that since the context bound variable can not declared globally, we add an extra parameter of type `int` for both the functions `r` and `t`.

Encode the quantified expression.

We add the following statement to import the declaration and axioms for the `bigOp` function.

```
use import bigOp.BigOp as BigOp
```

We encode the quantified operation as the following function application.

```
bigOp (r bv) (t bv)
```

The terms `(r bv)` and `(t bv)` are partially applied functions of type "`int -> bool`" and "`int -> int`" respectively.

Combine the declarations and expressions

We combine the declaration from the previous steps to arrive at the following expression and declarations.

Why3 Declarations:

```
use import bigOp.BigOp as BigOp
```
$decl_r$

$decl_t$

```
constant r: int -> int -> bool = (\bv: int, i: int. exprr)
constant t: int -> int -> int = (\bv: int, i: int. exprt)
```

Why3 Expression:

```
bigOp (r bv) (t bv)
```

4.3.3 Create a Why3 theory

To create a *Why3* theory for a proof obligation formula f, we apply an encoding function $\mathcal{E}$ to the formula f and an empty list of context bound variables.

Let $(expr_f, decl_f) = \mathcal{E}(f, [\,])$ and $decl_{global}$ be the declarations of free variables in f. The resulting *Why3* theory is given below.

```
theory Test
    use import bool.Bool
    use import int.Int
    use import array.Array
    <declglobal>
    <declr>
    <declt>
    <declf>
    goal G: <exprf>
end
```

4.3.4 Example

In this section we give example of encoding of the formula $(n = 0) \Rightarrow (E = 0)$ where expression E is defined as follows.

$$E \triangleq (Max\ p : (\exists q : 0 \leq p \leq q \leq n) : \\ (Max\ q : 0 \leq p \leq q \leq n : \\ (\sum i : p \leq i < q : arr[i])))$$

We first encode E as shown below.

Step 1:

The result of $\mathcal{E}(E, [\,])$ is given below.

Why3 declarations:

```
constant r: int -> bool
   = (\p: int. (\exists q: int. 0 <= p <= q <= n))
constant t: int -> int
   = (\p: int. E((Max q : 0 ≤ p ≤ q ≤ n : (∑ i : p ≤ i < q : arr[i]))))
```

Why3 expression:

```
bigMax r t
```

Step 2:

We expand the body of constant t as follows.

$$\mathcal{E}((Max\ q : 0 \leq p \leq q \leq n : \left(\sum i : p \leq i < q : arr[i]\right)), [p]) = (decl_2, expr_2)$$

where $decl_2$ and $expr_2$ are defined as given below.

$decl_2$:

```
constant r1: int -> int -> bool
   = (\p: int, \q: int. 0 <= p <= q <= n)
constant t1: int -> int -> bool
   = (\p: int, \q: int. E((∑ i : p ≤ i < q : arr[i])))
```

$expr_2$:

```
bigMax (r1 p) (t1 p)
```

Step 3:

We expand the body of constant $t1$ as follows.

$$\mathcal{E}(\Big(\sum i : p \leq i < q : arr[i]\Big)), [p, q]) = (decl_3, expr_3)$$

where $decl_3$ and $expr_3$ are defined as given below.

$decl_3$:

```
constant r2: int -> int -> int -> bool
   = (\p: int, \q: int, \i: int. p <= i < q)
constant t2: int -> int -> int -> bool
   = (\p: int, \q: int, \i: int. arr[i]))
```

$expr_3$:

```
bigSum (r2 p q )(t2 p q)
```

Step 4:

Finally, we import the basic theories, declare the free variables, collect the declarations and expressions from the previous steps, and build a theory. Final theory for the formula $(n = 0) \Rightarrow (E = 0)$ is presented in Fig. 4.4. The expression corresponding to the proof obligation formula becomes the *goal* statement in the theory. The theory can then be sent to the Why3 tool to check validity of the proof obligation formula.

Our approach of integrating the automated theorem provers at a tactic level is effective in automating the mundane tasks at the same time allowing users to decide the granularity of the derivation. The tactics introduced in this chapter help in shortening the derivations and also in carrying out derivations that are not amenable to the calculational style. We have managed to keep the derivation style close to the pen-and-paper calculational style thereby retaining the benefits of readability and rigour.

4.4 Related Work

The work that comes closest to our work in this chapter is the *verified calculations* [LP14] functionality in Dafny. However, the primary focus of Dafny — and many other similar

```
theory Test
    use import bool.Bool
    use import int.Int
    use import array.Array

    use import bigPlus.BigPlus as BigPlus
    use import bigMax.BigMax as BigMax

    constant n: int
    constant arr : array int

    constant r2 : int  -> int  -> int  -> bool
        = (\p: int, q: int, i: int. p <= i < q)

    constant t2 : int  -> int  -> int  -> int
        = (\p: int, q: int, i: int. arr[i])

    constant r1 : int  -> int  -> bool
        = (\p: int, q: int. 0 <= p <= q <= n)

    constant t1 : int  -> int  -> int
        = (\p: int, q: int. bigPlus (r2 p q) (t2 p q))

    constant r : int  -> bool
        = (\p: int. (exists q: int.
                          0 <= p <= q <= n))
    constant t : int  -> int
        = (\p: int. bigMax (r1 p) (t1 p))

    goal G:
        n = 0 ->
            (bigMax r  t) = 0
end
```

Figure 4.4. Why3 encoding of the formula $(n = 0) \Rightarrow (Max\ p : (\exists q : 0 \leq p \leq q \leq n) : (Max\ q : 0 \leq p \leq q \leq n : (\sum i : p \leq i < q : arr[i]))) = 0$

tools like Why3 [FP13], VCC [CDH+09] and VeriFast [JP08] — is on the verification of already implemented programs. The template based program synthesis approaches [GJTV11], [SLTB+06], [SGF10] are automatic in nature and require a syntactic template of the solution to be provided by the user. Our focus, however, is on the calculational derivation in an interactive setting.

Chapter 5

Assumption Propagation

5.1 Introduction

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 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:

> "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." [Mor90]

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 making them difficult

to organize.

Tools supporting the refinement based formal program derivation (Cocktail [Fra99], Refine [OXC04], Refinement Calculator [BL96a] and PRT [CHN+96a]) mostly follow the top-down methodology. Not much emphasis has been given on avoiding the guesswork and 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 an appropriate location is missing.

In this chapter, 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. 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. With the help of examples, we demonstrate how these rules help users in avoiding unnecessary rework and also help them explore alternative solutions.

5.2 *Maximum Segment Sum* Revisited

In Section 2.3, we discussed derivation of the *Maximum Segment Sum* problem performed without using any tool support. In this section, we revisit the problem, and present derivation sketch using our methodology.

5.2.1 *Maximum Segment Sum* Derivation

Fig. 5.1 depicts the derivation process for this program. As earlier, we start the derivation by providing the formal specification (node A) of the program. For brevity, we show only selected annotations. After applying a transformation rule (tactic) to a node, another node is created as a child node in the tree. We can transition between program and formula nodes using the functionality discussed in Section 3.4. Till node G, the derivation is similar to the one discussed in Section 2.3.

At node G, we realize that we can not represent r' in terms of the existing program variables since the expression $Q(n+1)$ involves quantifiers. After analyzing the derivation,

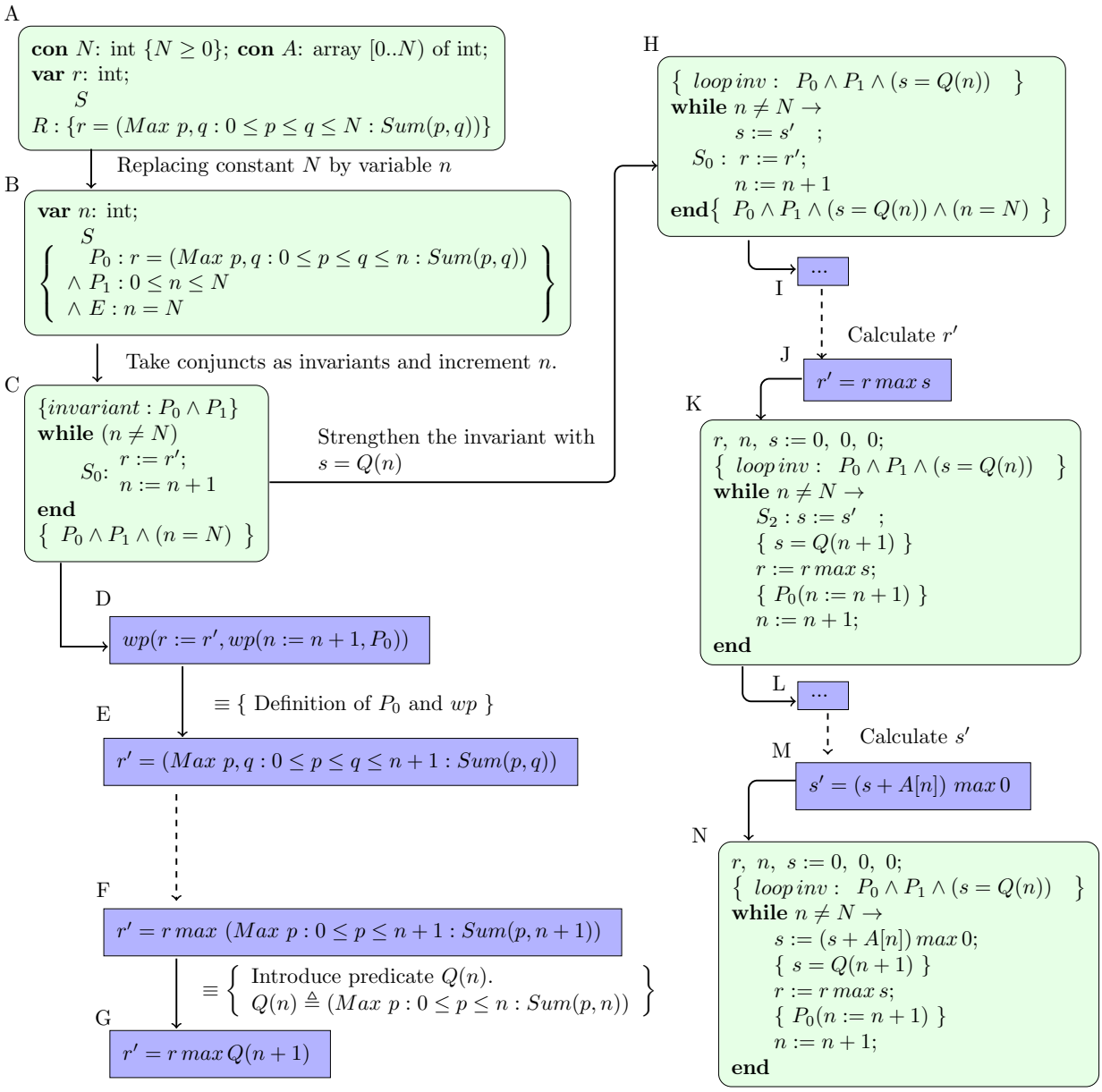


Figure 5.1. Sketch of the top-down derivation of the *Maximum Segment Sum* problem.

$Sum(p, q) \triangleq (\Sigma i : p \leq i < q : A[i])$; $Q(n) \triangleq (Max\ p : 0 \leq p \leq n : Sum(p, n))$

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

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. In the informal derivation in Section 2.3, we assumed $s = Q(n+1)$ and proceeded further. However, there were some back-of-the-envelope arguments involved. We need to be sure that adding $s = Q(n)$ as an invariant makes the predicate $s = Q(n+1)$ at the desired location. In general, when some changes are made upstream (like strengthening invariants), calculations from that point onward must be thoroughly checked for correctness.

Therefore, we backtrack to the program shown in node C, introduce a fresh variable s, and envision a *while* program with the strengthened invariant. 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.

The methodology solves the problem of correctness, but the problem of ad hoc reasoning remains.

5.2.2 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 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.

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 5.4.2, the above two decisions prematurely reduced the solution space preventing us from arriving at an alternative solution. The alternative solution (Fig. 5.20, node W) derived using the assumption propagation rules initializes n with -1 and uses an invariant involving the term $s = Q(n + 1)$.

5.2.3 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:

> "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 $N = 0$" [1][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 5.3 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.

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

5.3 Assumption Propagation

5.3.1 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 location 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. 5.2), $assume(\theta)$ 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 can propagate the $assume(\theta)$ statement upstream to an appropriate program location. The assumed 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.

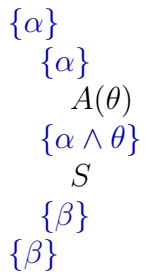


Figure 5.2. Result of assuming precondition θ in the derivation of $\{\alpha\}\, unkprog_1\, \{\beta\}$.

- **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.

5.3.2 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. 5.2. 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 the 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.

5.3.3 Rules for Propagating and Establishing Assumptions

The propagation step described in Section 5.3.1 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.

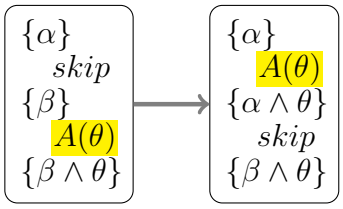


Figure 5.3. *SkipUp* rule

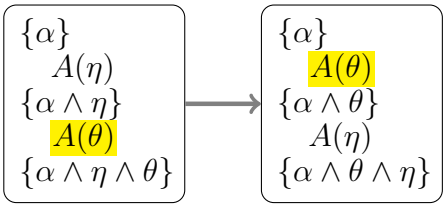


Figure 5.4. *AssumeUp* rule

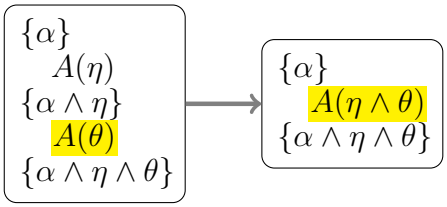


Figure 5.5. *AssumeMerge* rule

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.

skip. The *SkipUp* rule (Fig. 5.3) 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. 5.4) 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. 5.5). Applying this rule results in a single *assume* statement $A(\eta \wedge \theta)$.

unkprog. Fig. 5.6 shows the *UnkProgUp* rule which propagates an assumption up-

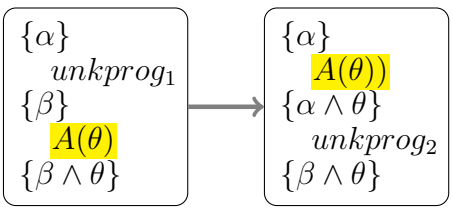


Figure 5.6. *UnkProgUp* rule

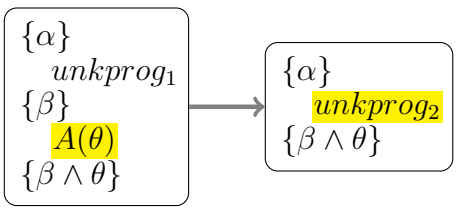


Figure 5.7. *UnkProgEst* rule

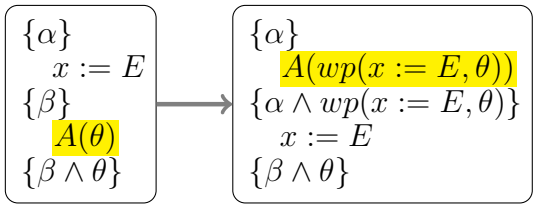


Figure 5.8. *AssignmentUp* rule

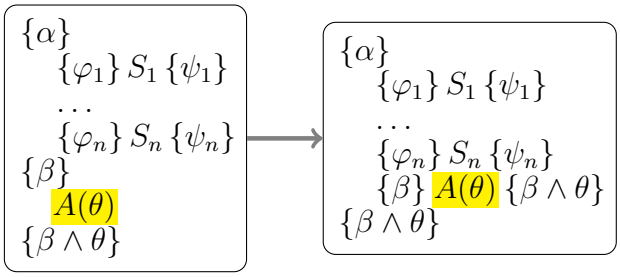


Figure 5.9. *CompositionIn* rule

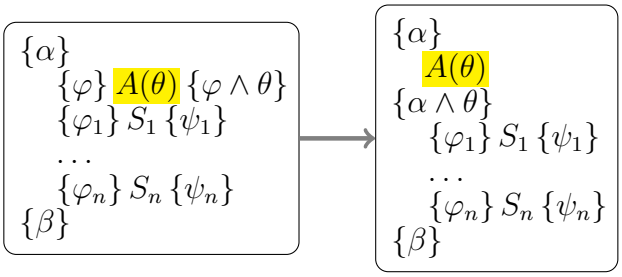


Figure 5.10. *CompositionOut* rule

ward 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. 5.7) can be used for this purpose.

assignment. Fig. 5.8 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 $x := E$.

Composition

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

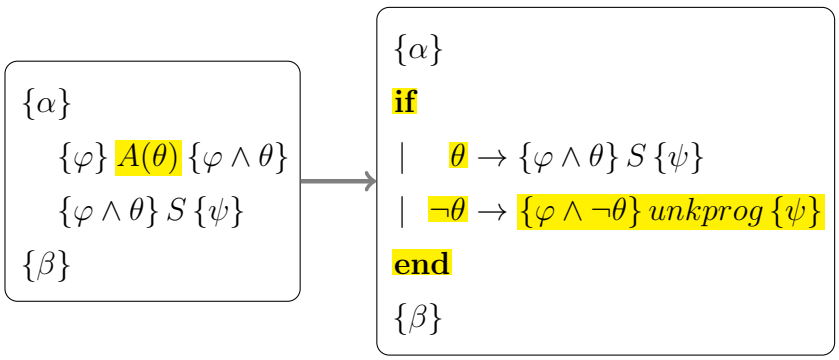


Figure 5.11. *CompoToIf* rule: Transforms a *composition* to an *if* program.

Case 1: The *assume* statement is immediately after the *composition* program.

Fig. 5.9 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. 5.10) 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. 5.11) 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 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.

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. 5.12) propagates an *assume*

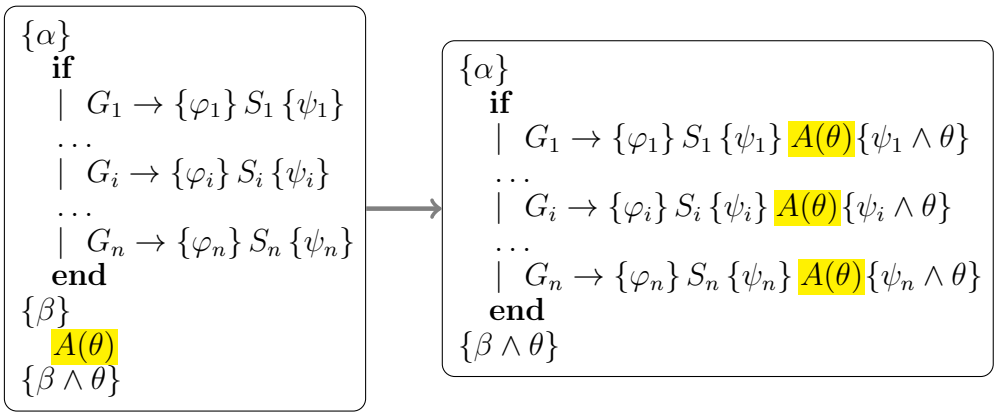


Figure 5.12. *IfIn* rule.

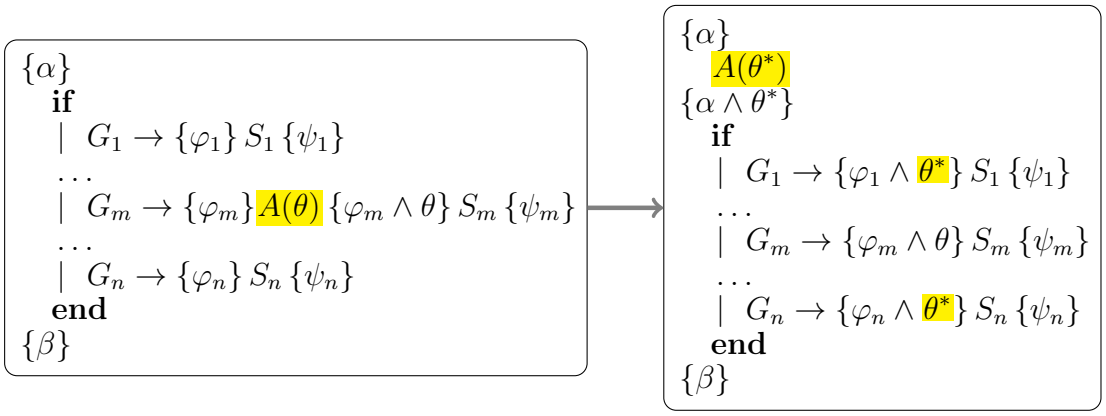


Figure 5.13. *IfOut* rule. $\theta^* \triangleq G_m \Rightarrow \theta$

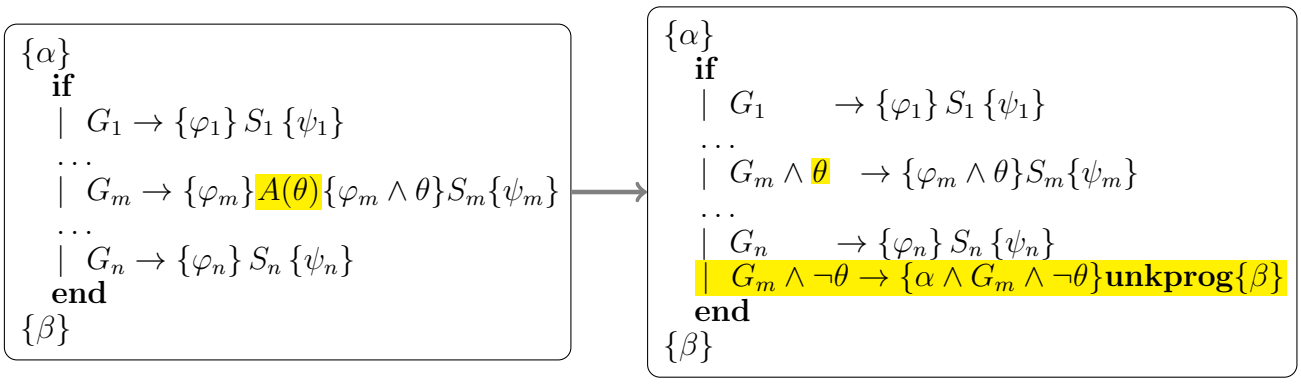


Figure 5.14. *IfGrd* rule

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. 5.13 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. 5.14) 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.

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. 5.15 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. 5.16), 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. 5.17). 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 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.

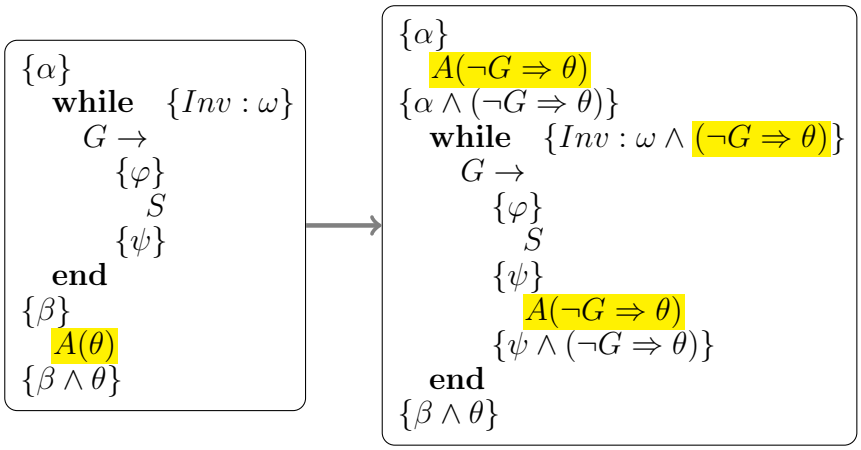


Figure 5.15. *WhileIn* rule: Strengthens the invariant with $\neg G \Rightarrow \theta$

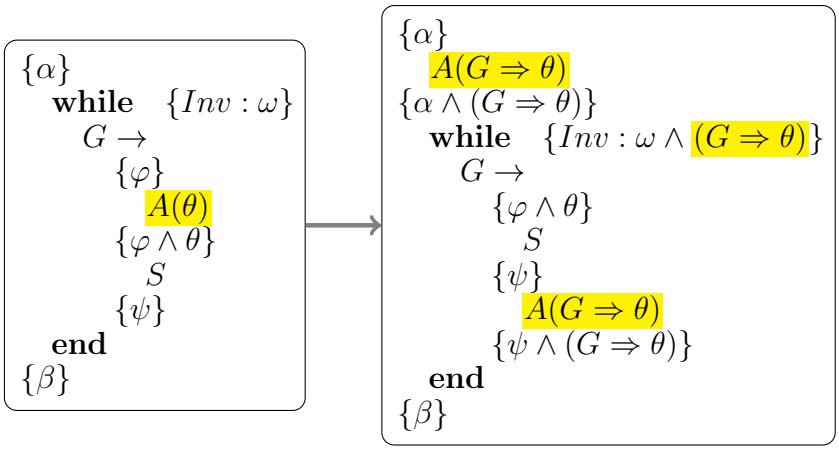


Figure 5.16. *WhileStrInv* rule: Strengthens the invariant with $G \Rightarrow \theta$

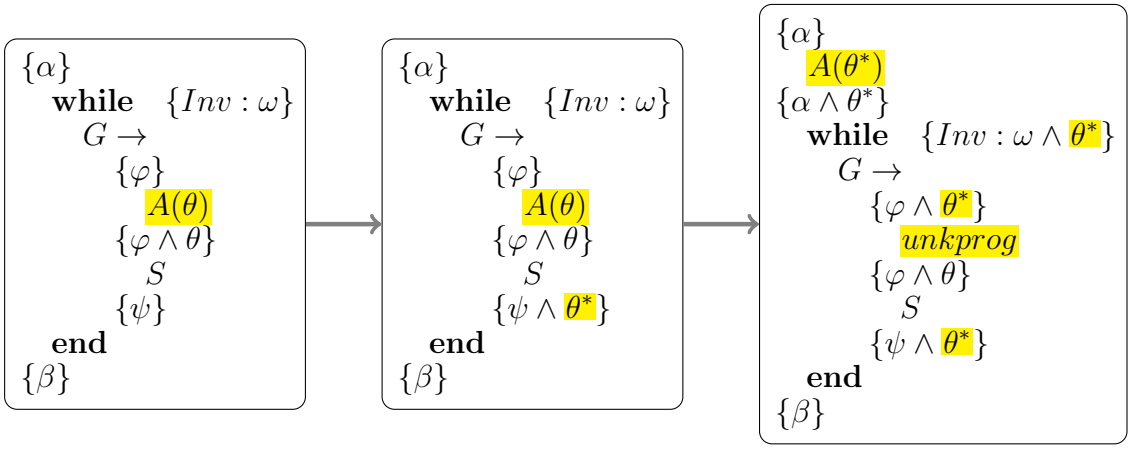
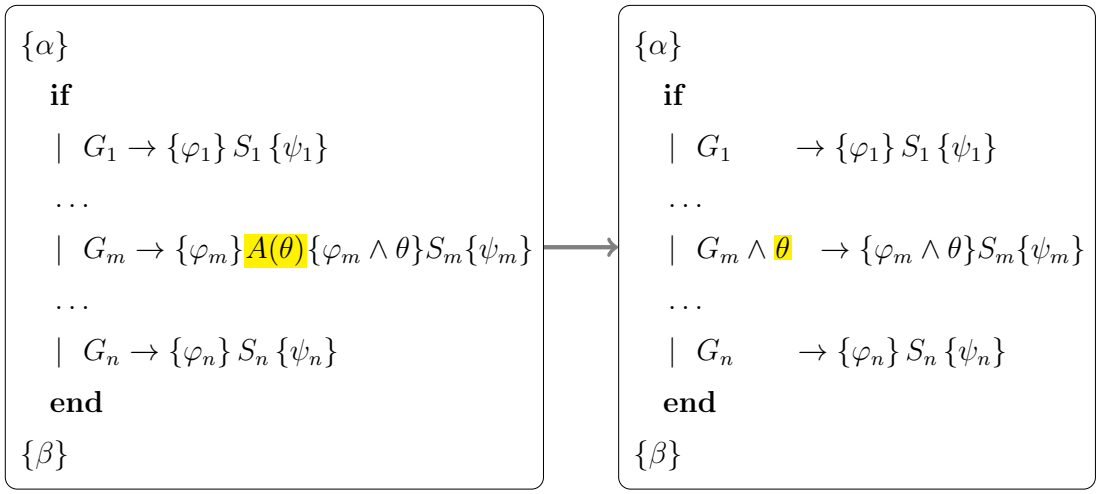


Figure 5.17. *WhilePostStrInv* rule: Strengthens the loop invariant with θ^* where $\theta^* \triangleq sp(S, \theta)$



Proviso : $\alpha \Rightarrow \left(\left(\bigvee_{i\in[1,n]\wedge i\neq m} G_i\right) \vee \theta\right)$

Figure 5.18. *IfGrd2* rule: A variation of the *IfGrd* rule(Fig. 5.14).

5.3.4 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. 5.18) rule which is a variation of the *IfGrd* rule (Fig. 5.14) where the rule is similar to the *IfGrd* except for the following differences.

1. This rule has a proviso $\alpha \Rightarrow \left(\left(\bigvee_{i\in[1,n]\wedge i\neq m} G_i\right) \vee \theta\right)$.
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.

5.3.5 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 5.4.2), 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 5.2.2 where we had to guess the right predicates.

5.3.6 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 5.4.1.

5.4 Derivation Examples

5.4.1 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. 5.19(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.

5.4.2 Back to the Motivating Example

Next, we derive the *Maximum Segment Sum* program using the assumption propagation approach. The initial derivation up to node G in Fig. 5.1 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

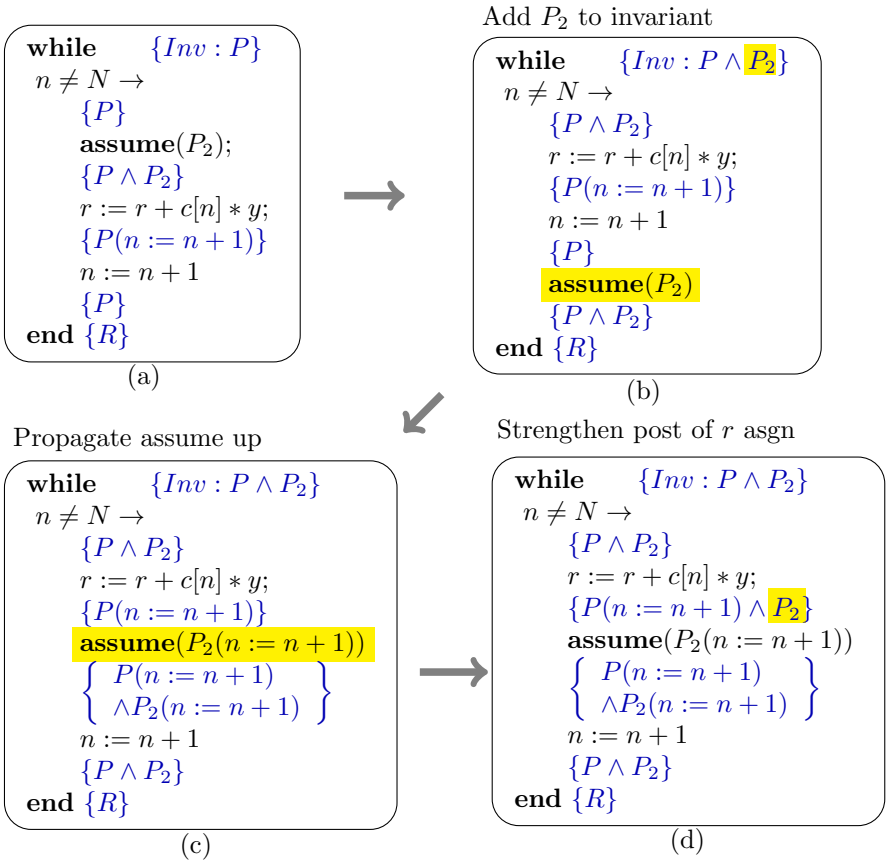


Figure 5.19. Some steps in the derivation of a program for the *Horner's rule*. Invariant initializations at the entry of the loop are not shown.

formula $s \equiv Q(n+1)$ and proceed further with the calculation.

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

After instantiating r' to $r \ max \ s$, we arrive at a *while* program (node O in Fig. 5.20) 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 $s = Q(n)$ and yields the program shown in node P in Fig. 5.20. 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. 5.20. 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 as 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 5.2.2, 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

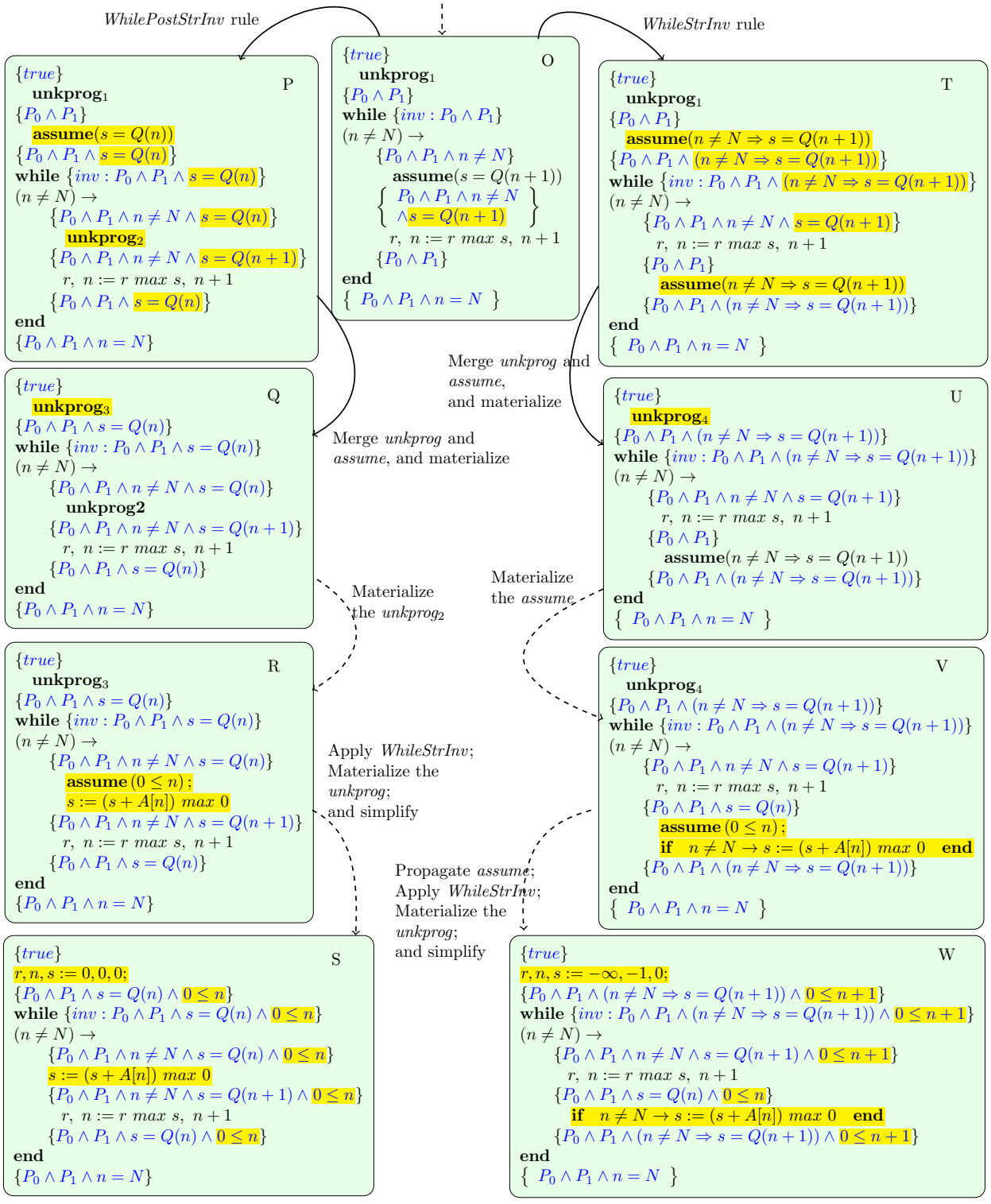


Figure 5.20. Maximum Segment Sum derivations using the *WhilePostStrInv* and *WhileStrInv* assumption propagation rules.

The following predicate definitions are same as those in Fig. 5.1 except for P_1.

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

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

this case, the appropriate guards are automatically added to safeguard us from accessing an undefined array element.

We have implemented the assumption propagation technique in the CAPS system (refer Section 7.7 for the implementation details). The approach described here is general enough to be adaptable for use in other formal program development environments supporting invariant annotations.

5.5 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 5.4, 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 [BL96a] provides a general mechanism for refinement based transformational reasoning on top of HOL. The PRT tool [CHN+96a] 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.

Chapter 6

Correctness of Assumption Propagation Rules

To prove that a rule $\mathcal{R} : src \mapsto target$ is correctness preserving, we need to prove that the following formula $\mathcal{A}$ is valid.

$$\mathcal{A} : [proviso(src)] \Rightarrow [po(src)] \Rightarrow [po(target)]$$

The square brackets denote universal quantification over the points in state space as explained in Section 2.2.1.

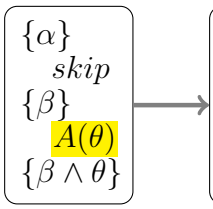
In the following proofs, instead of proving validity of $\mathcal{A}$, we will prove the validity of the following formula $\mathcal{B}$ which is stronger than $\mathcal{A}$.

$$\mathcal{B} : [proviso(src) \Rightarrow po(src) \Rightarrow po(target)]$$

The *proviso* is optional and is assumed to be *true* if not mentioned. 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.

6.1 SkipUp Rule

Rule:



$po(src) = \bigwedge_i \Gamma_i$

$\Gamma_1 \quad : po\,(\{\alpha\}\ skip\ \{\beta\})$
$\Gamma_2 \quad : po\,(\{\beta\}\ A(\theta)\ \{\beta \wedge \theta\})$

$po(target) = \bigwedge_i \Delta_i$

$\Delta_1 : \quad po\,(\{\alpha\}\ A(\theta)\ \{\alpha \wedge \theta\})$
$\Delta_2 : \quad po\,(\{\alpha \wedge \theta\}\ skip\ \{\beta \wedge \theta\})$

Theorem 6.1. *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 }
$true$

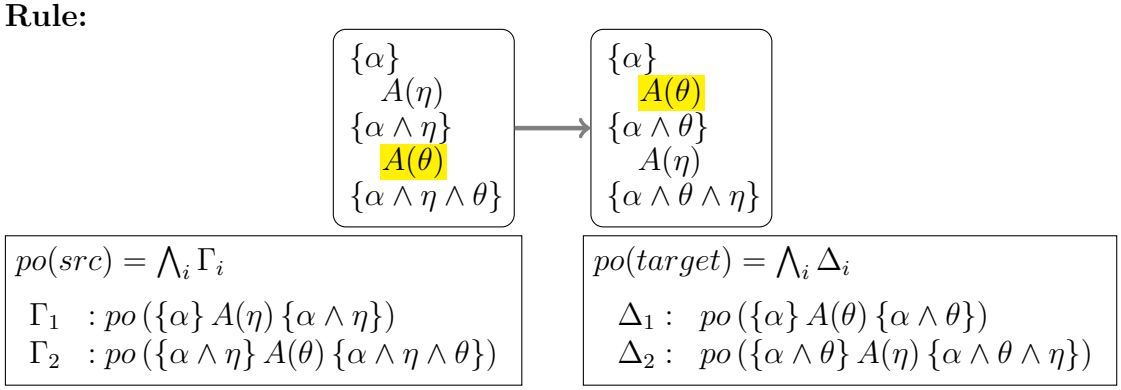
Δ_2 :

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

□

6.2 AssumeUp Rule

Rule:



Theorem 6.2. *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* }

$\alpha \wedge \theta \wedge \eta \Rightarrow \alpha \wedge \theta \wedge \eta$

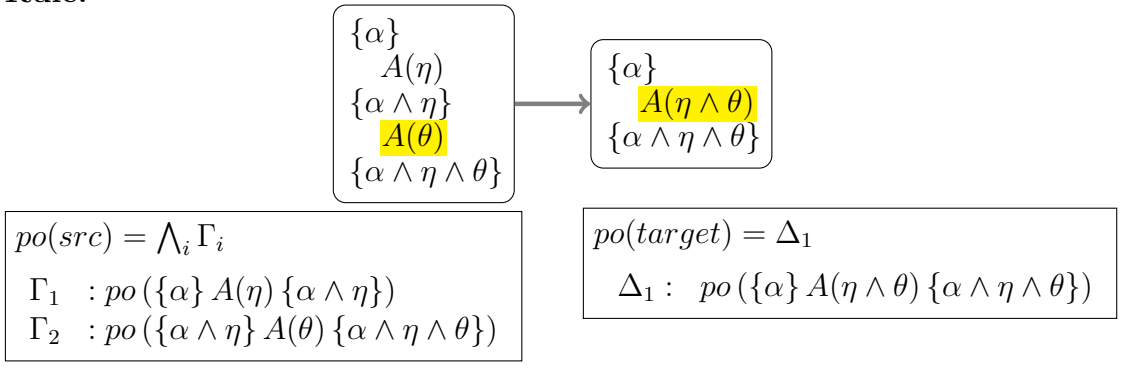
$\equiv$ { predicate calculus }

$true$

□

6.3 AssumeMerge Rule

Rule:



Theorem 6.3. *AssumeMerge* rule is correctness preserving.

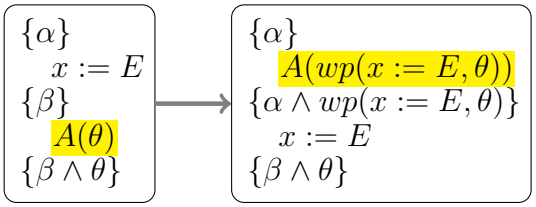
Proof. We assume the correctness of the *src* program and prove the proof obligation of the target program(Δ_1).

$$
\begin{aligned}
&\Delta_1 : \\
&\quad po\,(\{\alpha\}\ A(\eta \wedge \theta)\ \{\alpha \wedge \eta \wedge \theta\}) \\
\equiv &\quad \{\ \text{definition of } \textit{assume}\ \} \\
&\quad \alpha \wedge \eta \wedge \theta \Rightarrow \alpha \wedge \eta \wedge \theta \\
\equiv &\quad \{\ \text{predicate calculus}\ \} \\
&\quad \textit{true}
\end{aligned}
$$

□

6.4 AssignmentUp Rule

Rule:



$po(src) = \bigwedge_i \Gamma_i$

$\Gamma_1 : \quad \alpha \Rightarrow \beta(x := E)$

$\Gamma_2 : \quad po\,(\{\beta\}\ A(\theta)\ \{\beta \wedge \theta\})$

$po(target) = \bigwedge_i \Delta_i$

$\Delta_1 : \quad po\,(\{\alpha\}\ A(wp(x := E, \theta))\ \{\alpha \wedge wp(x := E, \theta)\})$

$\Delta_2 : \quad po\,(\{\alpha \wedge wp(x := E, \theta)\}\ x := E\ \{\beta \wedge \theta\})$

Theorem 6.4. *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 }

$\alpha \Rightarrow \beta(x := E)$

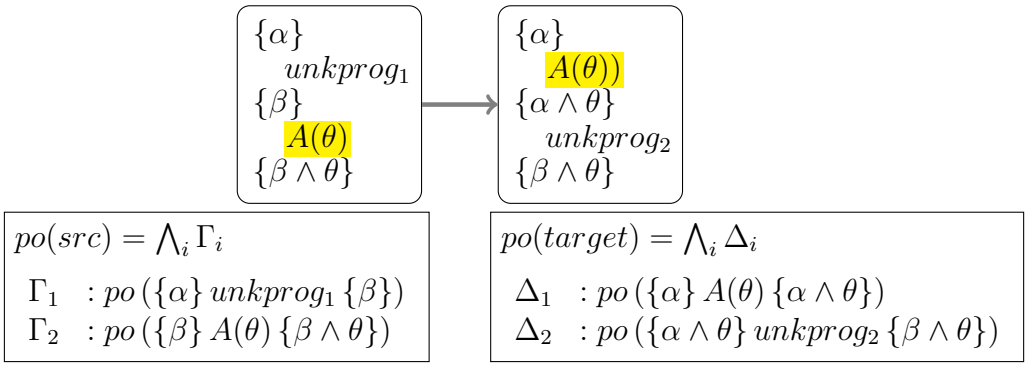
$\equiv$ { Γ_1 }

$true$

□

6.5 UnkProgUp Rule

Rule:



Theorem 6.5. *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

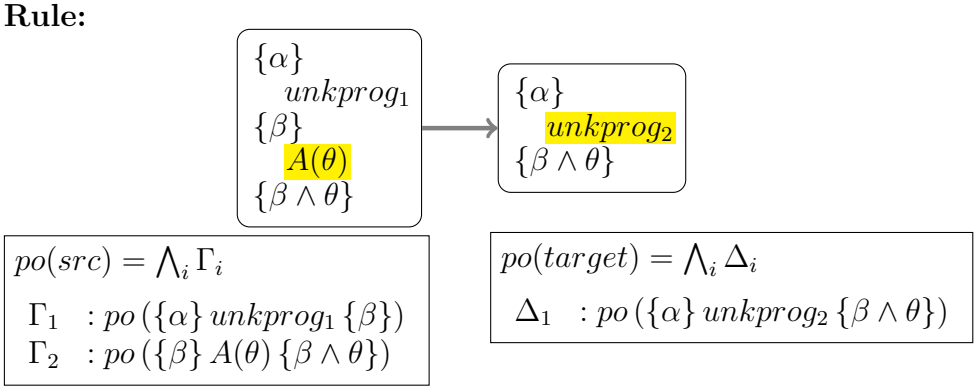
$po\,(\{\alpha \wedge \theta\}\,unkprog_2\,\{\beta \wedge \theta\})$

$\equiv$ { definition of *unkprog* }

$true$

6.6 UnkProgEst Rule

Rule:



Theorem 6.6. *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 :

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

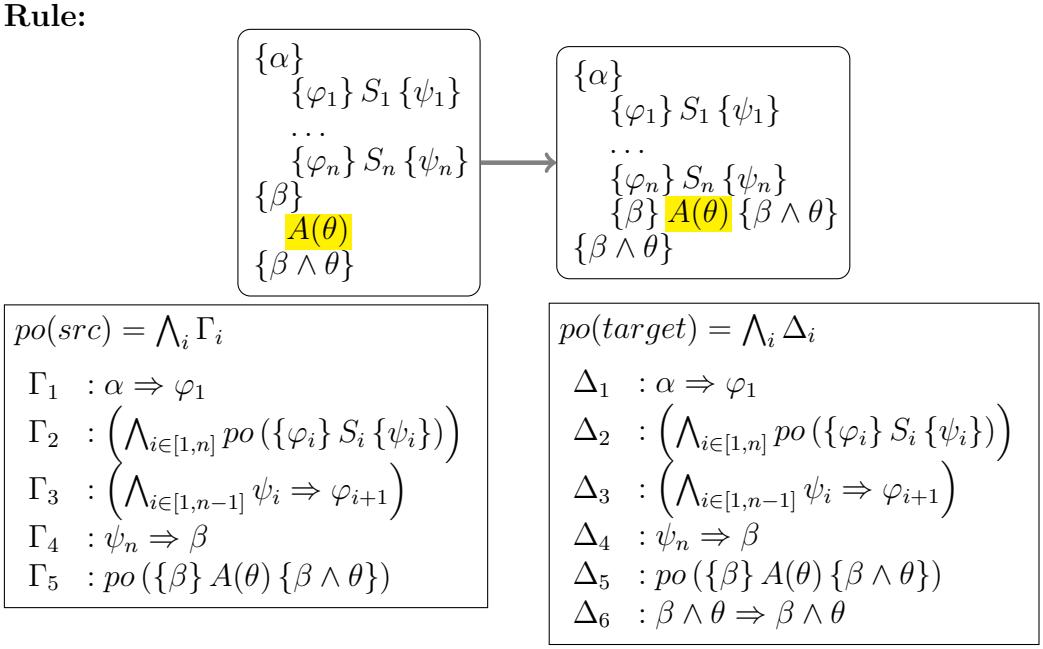
$\equiv$ { definition of *unkprog* }

$true$

□

6.7 CompositionIn Rule

Rule:



Theorem 6.7. *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 :

$\beta \wedge \theta \Rightarrow \beta \wedge \theta$

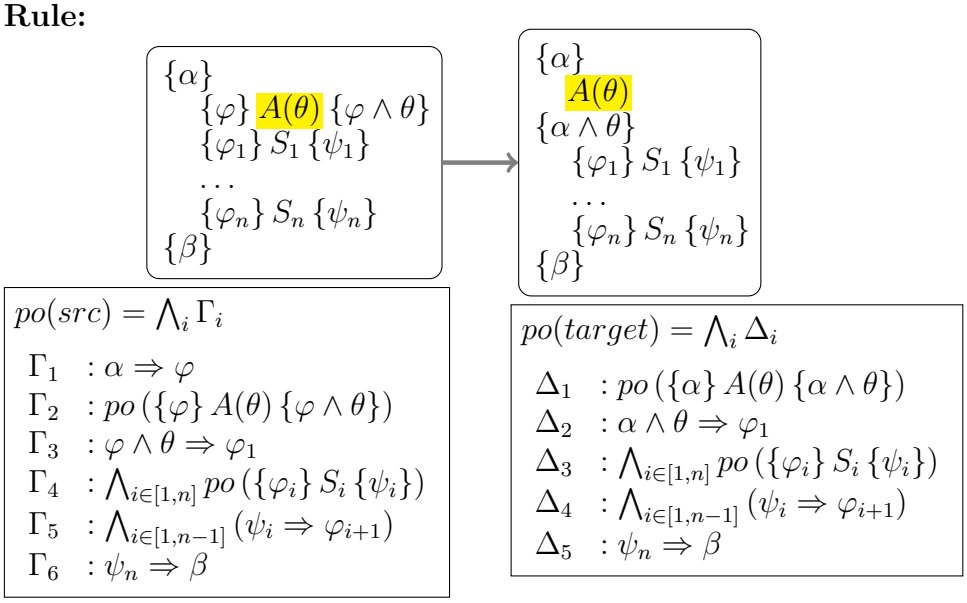
$\equiv$ { predicate calculus }

$true$

□

6.8 CompositionOut Rule

Rule:



Theorem 6.8. *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 :

$po\,(\{\alpha\}\,A(\theta)\,\{\alpha \wedge \theta\})$

$\equiv$ { definition of assume }

$\alpha \wedge \theta \Rightarrow \alpha \wedge \theta$

$\equiv$ { predicate calculus }

$true$

Δ_2 :

$\alpha \wedge \theta \Rightarrow \varphi_1$

$\Leftarrow$ { $\Gamma_1;\ \Gamma_3$ }

$true$

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 :

$\psi_n \Rightarrow \beta$

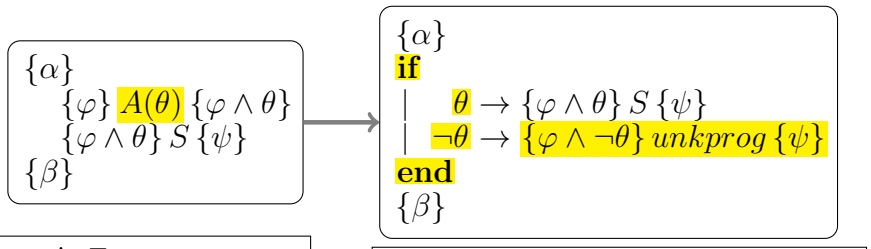
$\equiv$ { Γ_6 }

$true$

□

6.9 CompoToIf Rule

Rule:



$po(src) = \bigwedge_i \Gamma_i$

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

$po(target) = \bigwedge_i \Delta_i$

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

Theorem 6.9. *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 :

$\psi \Rightarrow \beta$

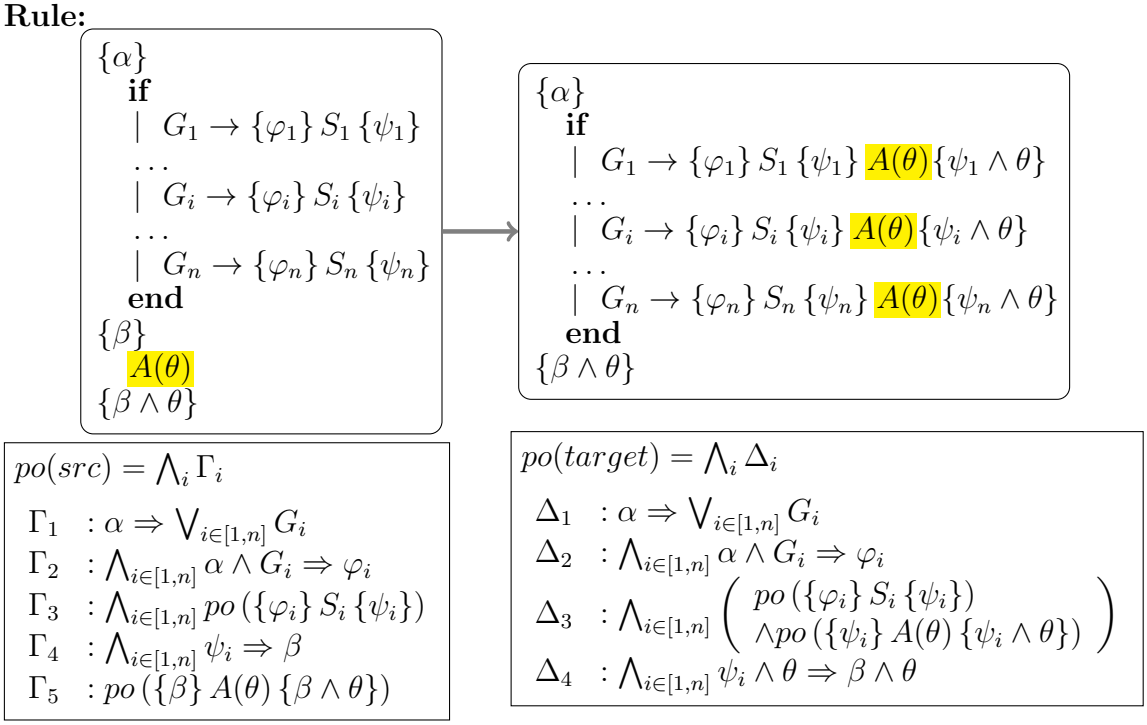
$\equiv$ { Γ_4 }

$true$

□

6.10 IfIn Rule

Rule:



Theorem 6.10. *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 :

$$\bigwedge_{i\in[1,n]} \left(\begin{array}{l} po\,(\{\varphi_i\}\ S_i\ \{\psi_i\}) \\ \wedge po\,(\{\psi_i\}\ A(\theta)\ \{\psi_i \wedge \theta\}) \end{array} \right)$$

$\equiv$ { definition of *assume* }

$$\bigwedge_{i\in[1,n]} \left(\begin{array}{l} po\,(\{\varphi_i\}\ S_i\ \{\psi_i\}) \\ \wedge\,(\psi_i \wedge \theta \Rightarrow \psi_i \wedge \theta) \end{array} \right)$$

$\equiv$ { 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 }

$true$

□

6.11 IfOut Rule

Rule:

$$\{\alpha\}\ \textbf{if}\ |\ G_1 \to \{\varphi_1\}\ S_1\ \{\psi_1\}\ \ldots\ |\ G_m \to \{\varphi_m\}A(\theta)\ \{\varphi_m \wedge \theta\}\ S_m\ \{\psi_m\}\ \ldots\ |\ G_n \to \{\varphi_n\}\ S_n\ \{\psi_n\}\ \textbf{end}\ \{\beta\}$$

$$\longrightarrow$$

$$\{\alpha\}\ A(\theta^*)\ \{\alpha \wedge \theta^*\}\ \textbf{if}\ |\ G_1 \to \{\varphi_1 \wedge \theta^*\}\ S_1\ \{\psi_1\}\ \ldots\ |\ G_m \to \{\varphi_m \wedge \theta\}\ S_m\ \{\psi_m\}\ \ldots\ |\ G_n \to \{\varphi_n \wedge \theta^*\}\ S_n\ \{\psi_n\}\ \textbf{end}\ \{\beta\}$$

$po(src) = \bigwedge_i \Gamma_i$

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

$po(target) = \bigwedge_i \Delta_i$

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

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

Theorem 6.11. *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 :

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

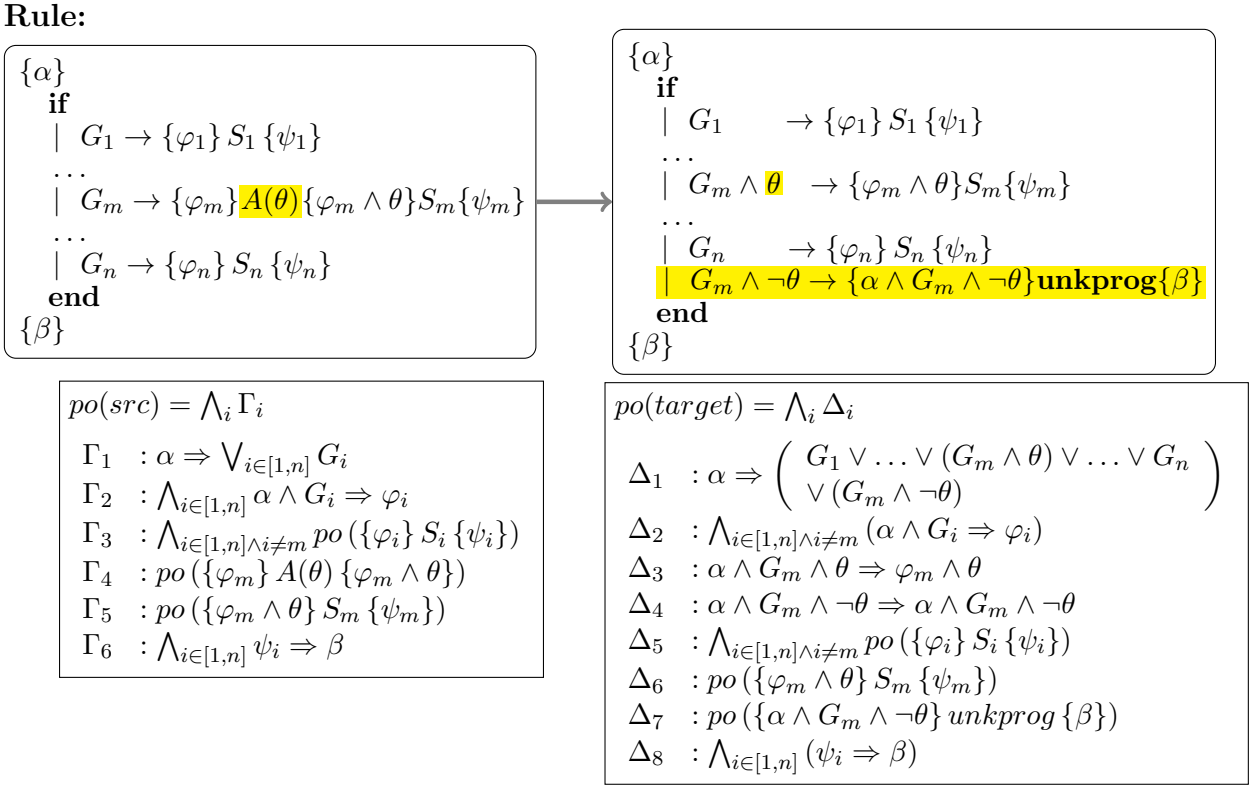
$\equiv$ { Γ_5 }

$true$

□

6.12 IfGrd Rule

Rule:



Theorem 6.12. *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 :

$$\alpha \Rightarrow \left(\begin{array}{l} G_1 \vee \ldots \vee (G_m \wedge \theta) \vee \ldots \\ \vee G_n \vee (G_m \wedge \neg\theta) \end{array} \right)$$

$\equiv$ { $(G_m \wedge \theta) \vee (G_m \wedge \neg\theta) \equiv G_m$ }

$$\alpha \Rightarrow (G_1 \vee \ldots \vee G_m \vee \ldots \vee G_n)$$

$\equiv$ { 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_7 :

$po\,(\{\alpha \wedge G_m \wedge \neg\theta\}\ unkprog\ \{\beta\})$

$\equiv$ { definition of $unkProg$ }

$true$

Δ_6 :

$po\,(\{\varphi_m \wedge \theta\}\ S_m\ \{\psi_m\})$

$\equiv$ { Γ_5 }

$true$

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

$\equiv$ { Γ_6 }

$true$

□

6.13 IfGrd2 Rule

Rule:

$\{\alpha\}$
if
| $G_1 \rightarrow \{\varphi_1\}\ S_1\ \{\psi_1\}$
...
| $G_m \rightarrow \{\varphi_m\} A(\theta) \{\varphi_m \wedge \theta\} S_m \{\psi_m\}$
...
| $G_n \rightarrow \{\varphi_n\}\ S_n\ \{\psi_n\}$
end
$\{\beta\}$

$\rightarrow$

$\{\alpha\}$
if
| $G_1 \rightarrow \{\varphi_1\}\ S_1\ \{\psi_1\}$
...
| $G_m \wedge \theta \rightarrow \{\varphi_m \wedge \theta\} S_m \{\psi_m\}$
...
| $G_n \rightarrow \{\varphi_n\}\ S_n\ \{\psi_n\}$
end
$\{\beta\}$

$po(src) = \bigwedge_i \Gamma_i$

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

Proviso : $\alpha \Rightarrow \left(\left(\bigvee_{i\in[1,n]\wedge i\neq m} G_i\right) \vee \theta\right)$

IfGrd2 rule: A variation of the *IfGrd* rule(Fig. 5.14).

Theorem 6.13. *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 :

$\alpha \Rightarrow (G_1 \vee \ldots \vee (G_m \wedge \theta) \vee \ldots \vee G_n)$

$\equiv$ { distributivity }

$\alpha \Rightarrow \left(\bigvee_{i\in[1,n]} G_i\right) \wedge \left(\left(\bigvee_{i\in[1,n]\wedge i\neq m} G_i\right) \vee \theta\right)$

$\equiv$ { Γ_1 }

$\alpha \Rightarrow \left(\bigvee_{i\in[1,n]\wedge i\neq m} G_i\right) \vee \theta$

$\equiv$ { proviso }

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 :

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

$\equiv$ { Γ_3 }

true

Δ_5 :

$po\,(\{\varphi_m \wedge \theta\}\, S_m\, \{\psi_m\})$

$\equiv$ { Γ_5 }

true

Δ_6 :

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

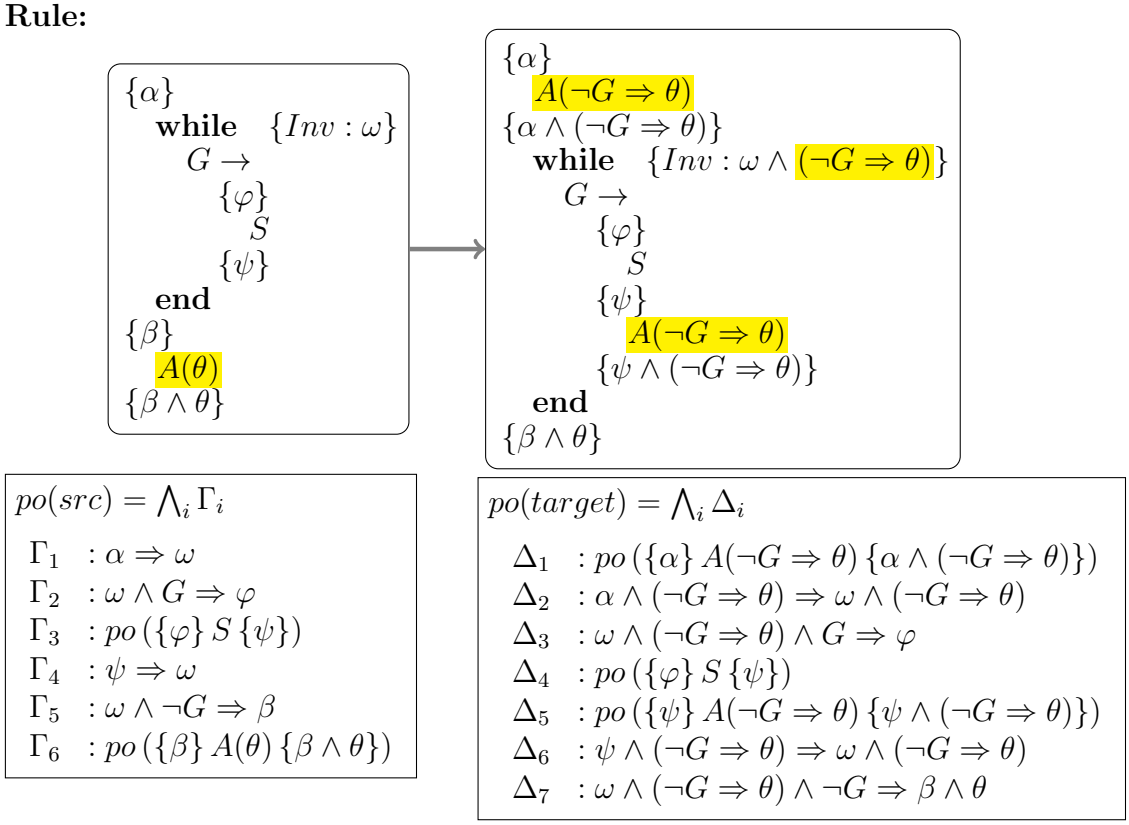
$\equiv$ { Γ_6 }

true

□

6.14 WhileIn Rule

Rule:



Theorem 6.14. *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 }

$\omega \wedge \neg G \Rightarrow \beta$

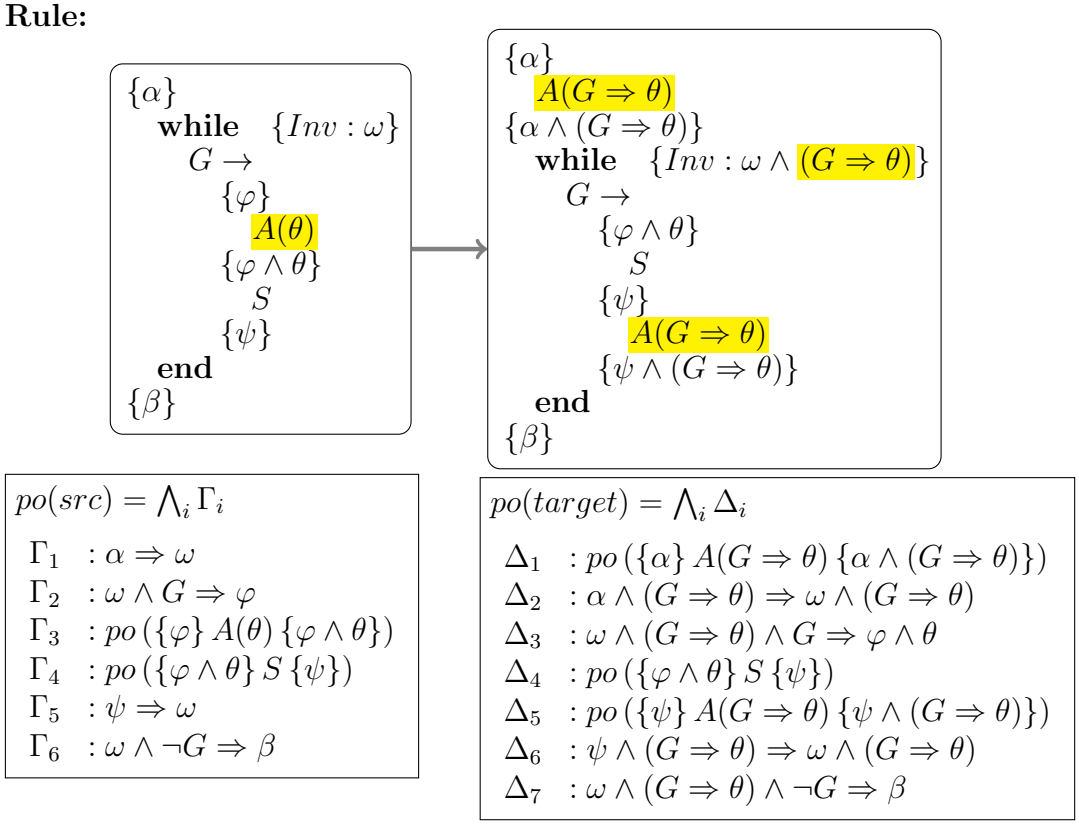
$\equiv$ { Γ_5 }

$true$

□

6.15 WhileStrInv Rule

Rule:



Theorem 6.15. *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 }

$\omega \wedge \neg G \Rightarrow \beta$

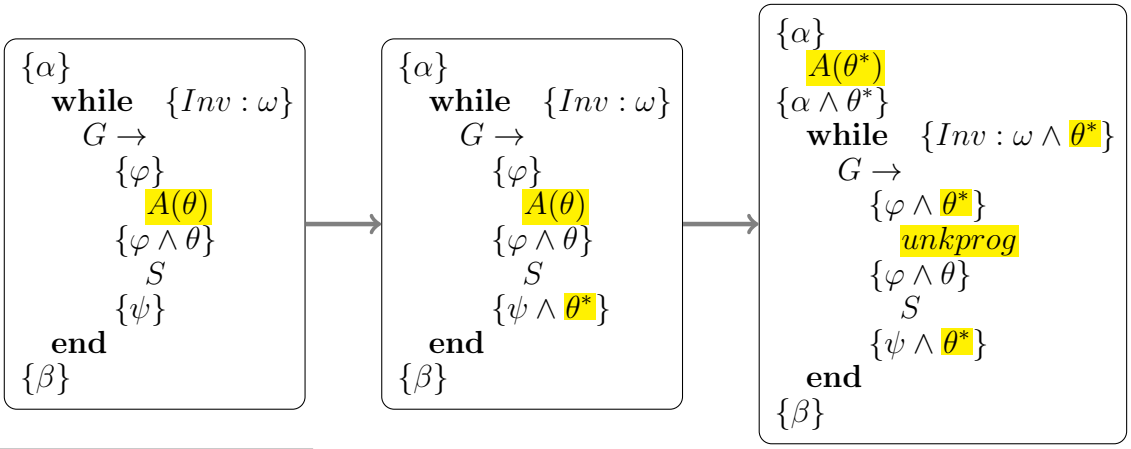
$\equiv$ { Γ_6 }

true

□

6.16 WhilePostStrInv Rule

Rule:



$po(src) = \bigwedge_i \Gamma_i$

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

$po(target) = \bigwedge_i \Delta_i$

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

Theorem 6.16. *WhilePostStrInv* 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 :

$\alpha \wedge \theta^* \Rightarrow \omega \wedge \theta^*$
$\Leftarrow$ { 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 }

$true$

□

Chapter 7

CAPS

7.1 Introduction

We have implemented our methodology in the CAPS system [1]. In building CAPS, our aim has been to build an easy to use IDE that automates the mundane tasks while retaining the calculational flavor of the derivations. In this chapter, we discuss main features of the system, the GUI design, the overall architecture of the CAPS system, and the various design trade-offs involved. We discuss how various features of the system address the issues identified in Section 2.4.

7.2 Graphical User Interface

The Graphical User Interface of the CAPS system has three main panels (Fig. 7.1). The bottom panel, called the input panel, is used for selecting a tactic (from a list of available tactics) to be applied next and for providing the corresponding input parameters. There is a special tactic called *InitTactic* which is used for starting a new derivation by providing a specification. The left panel, also called the tactics panel, shows the list of the tactics applied so far. It corresponds to a path the derivation tree. Users can navigate back to an earlier point in the derivation by clicking on the corresponding node in the left panel. The central panel is called the contents panel. It shows a partially derived program (or a formula) at the current stage of the derivation.

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

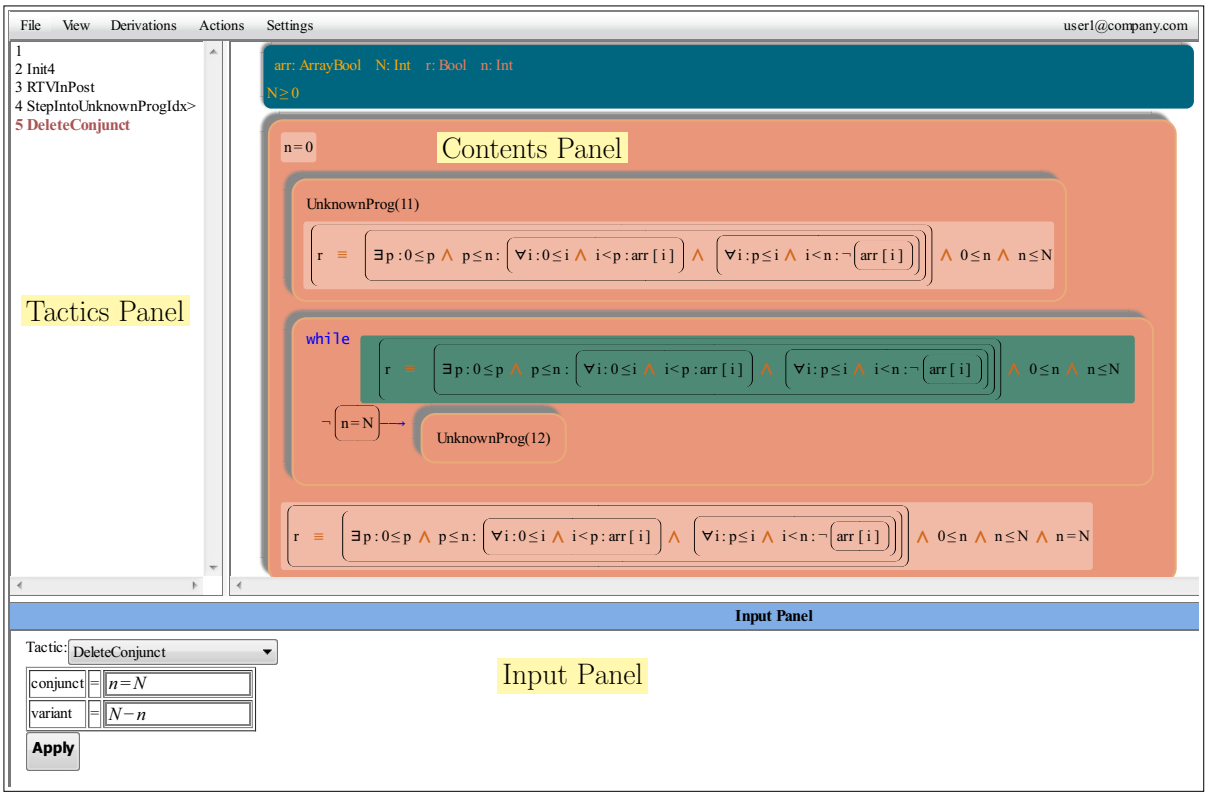


Figure 7.1. CAPS GUI

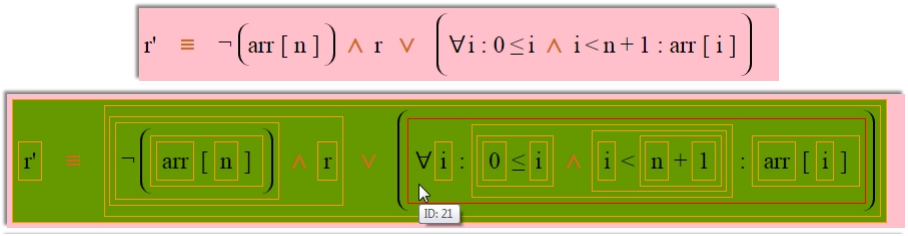


Figure 7.2. Structured representation of a formula in *normal mode* and *selection mode*. Users can select a subformula by simply clicking on it.

7.3 Textual vs Structured Representation

One important decision in developing an IDE is the choice between a textual representation and a structural one. While the tools like Dafny [Lei10] and Why3 [FP13] use textual representations, the structural representation is more suitable for a tactic based framework like CAPS. An *annGCL* program in CAPS has a hierarchical structure consisting of nested programs and formulas. By *structured* representation, we mean that such hierarchical elements are identifiable in the GUI. As discussed later, this allows the user to select and focus on a subprogram or a subformula. Note that doing the same in a text based representation will require extra processing [BKSS97].

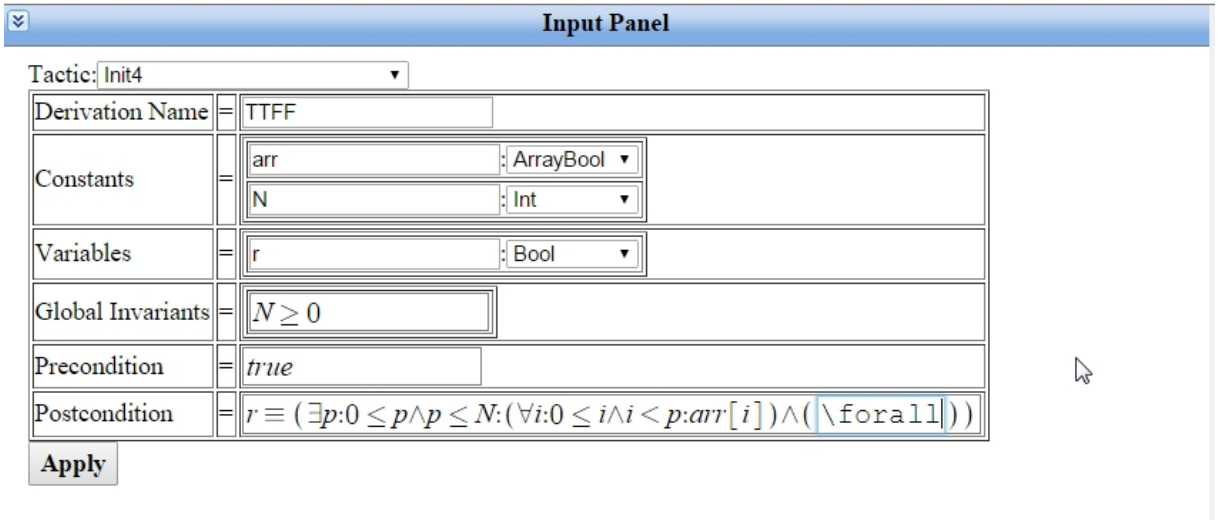


Figure 7.3. Input Panel: On selection of a tactic to be applied, the corresponding input form is dynamically generated.

Direct editing of the *annGCL* program may destroy the structure and is disallowed in CAPS; the only way to generate/modify a program is through a tactic application. This discipline allows us to capture all the design decisions taken during the derivation. However, to allow some informality, we do have tactics to directly guess a program fragment (or the next formula). In such cases, the role of a tactic application is just to ensure - with the help of theorem provers - that the transformation is correct, and that the structure is maintained.

The contents panel in Fig. 7.1 shows the structured representation of an *annGCL* program. Fig. 7.2 shows the structured representation of a formula in the normal and the selection mode. The binary logical operators are shown using the infix notation. Only necessary parentheses are displayed assuming the usual precedence. We put more space around the lower precedence operators (like $\equiv$) to improve readability.

For inputting the tactic parameters, we prefer a dynamically generated GUI instead of a static textual input form. On selecting a tactic to be applied next, the corresponding input form is dynamically generated. Users need not remember the input parameters required for the tactic. Fig. 7.3 shows the tactic input panel for the *Init4* tactic which is used for specifying the program. Since CAPS is a web-based application, the hypertext-based display enables providing a help menu for input parameters in a user-friendly way.

For entering formulas, however, we prefer textual input. The formulas are entered in the LaTeX format. The formula input box is responsive; as soon as a LaTeX expression is

typed, it converts the expression into the corresponding symbol immediately. We use the MathQuill [Mat] library for this purpose.

7.4 Focusing on Subcomponents

During the program derivation process, an annotated program is nothing but a partially derived program containing multiple unsynthesized subprograms. The derivation of these unsynthesized subprograms is, for the most part, independent of the rest of the program. Hence the CAPS system provides a facility to extract all the contextual information required for the derivation of a subprogram so that users can focus their attention on the derivation of one of these unknown subprograms. A subprogram can be selected by simply clicking on it. On selecting a subprogram, only the extracted context of the subprogram, and its precondition and postcondition are shown whereas the rest of the program is hidden.

Similar to the subprogram extraction, users can choose to restrict attention to a subformula of the formula under consideration. On focusing on a subformula, the system extracts and presents the contextual information necessary for manipulating the subformula. Our subformula representation is an extension of the *Structured Calculational Proof* format [BGVW97].

The derivation is displayed in a nested fashion. Fig. 7.4 shows a snapshot of the formula transformations from the derivation of the binary search program. Whenever a user focuses on a subformula, an inner frame is created inside the outer frame. The assumptions available in each frame are displayed on the top of the frame. In the figure, as the user focuses on the consequent of the implication, the antecedent is added to the assumptions. On successful derivation of all the metavariables, users can step out from the formula mode. The system then creates a program where the metavariables are replaced with the corresponding derived expressions.

Unlike the hierarchical program structure, the hierarchical formula structure is not explicitly shown in the GUI. This is done to reduce the clutter as the hierarchical formula structure can get very large. It is only displayed when we intend to select a subformula. This user interaction mode, called a selection mode, is used to select subformulas to be focused on. Fig. 7.2 shows a formula in the normal mode and in the selection mode.

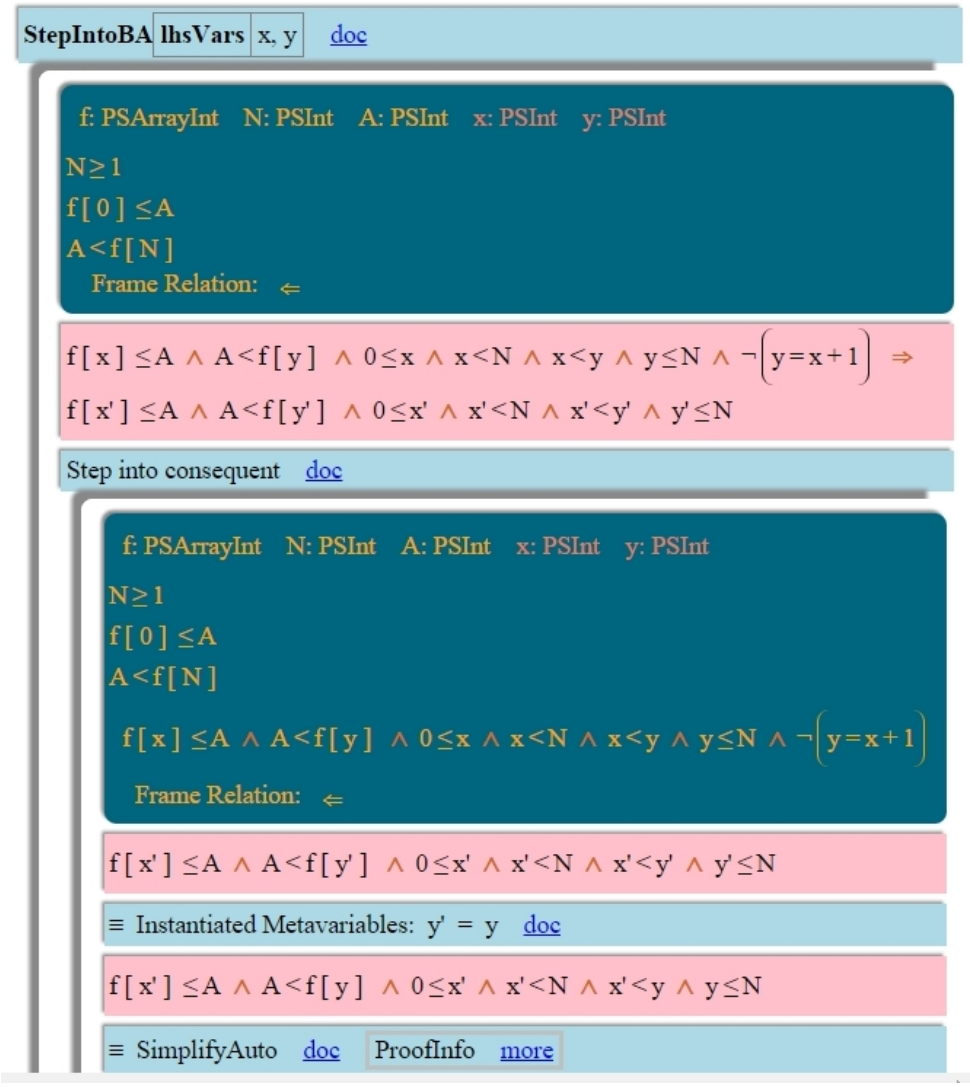
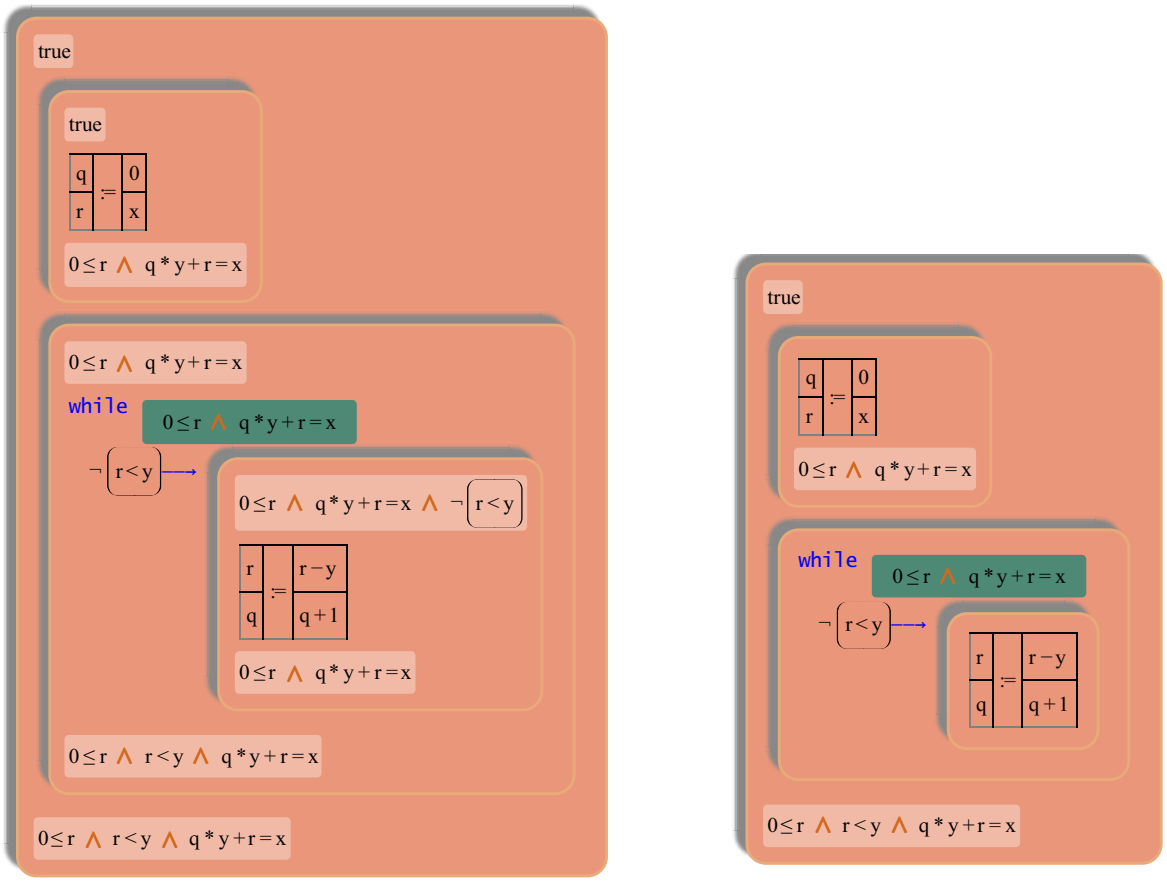


Figure 7.4. Formula transformations from the derivation of the Binary Search program.

7.5 Selective Display of Information

In the *annGCL* representation, all the subprograms are annotated with the respective precondition and postcondition. Although this creates a nice hierarchical structure, it results in a cluttered display which places higher cognitive demand on the attention and mental resources of the users. An effective way to keep the cognitive load low, is to hide the irrelevant information in a given context. For example, the annotations that can be inferred easily from the other annotations should be hidden to reduce the clutter. CAPS provides a *Minimal Annotations* mode which displays only the following annotations.

- Precondition and postcondition of the outermost program
- Loop invariants



(a) (b)

Figure 7.5. Final *annGCL* for the Integer Division problem: a) Full annotations mode, b) Minimal annotation mode.

- The intermediate-assertion of *Composition* constructs

All other annotations can be inferred from these annotations without performing a textual substitution required for computing the weakest precondition with respect to an assignment statement. Fig. 7.5 shows the Integer Division program with full annotations and with minimal annotations. All the hidden annotations can be easily inferred from the displayed annotations. The minimal annotations reduce the clutter to a great extent.

In addition to the annotations, there are lots of other details that can be hidden. For example, the discharge status of various proof obligations for the *SimplifyAuto* tactic can run into several pages, and is hidden by default (The *ProofInfo* link in the Fig. 7.4). The annotated programs can also be collapsed by double clicking on them.

7.6 Maintaining Derivation History

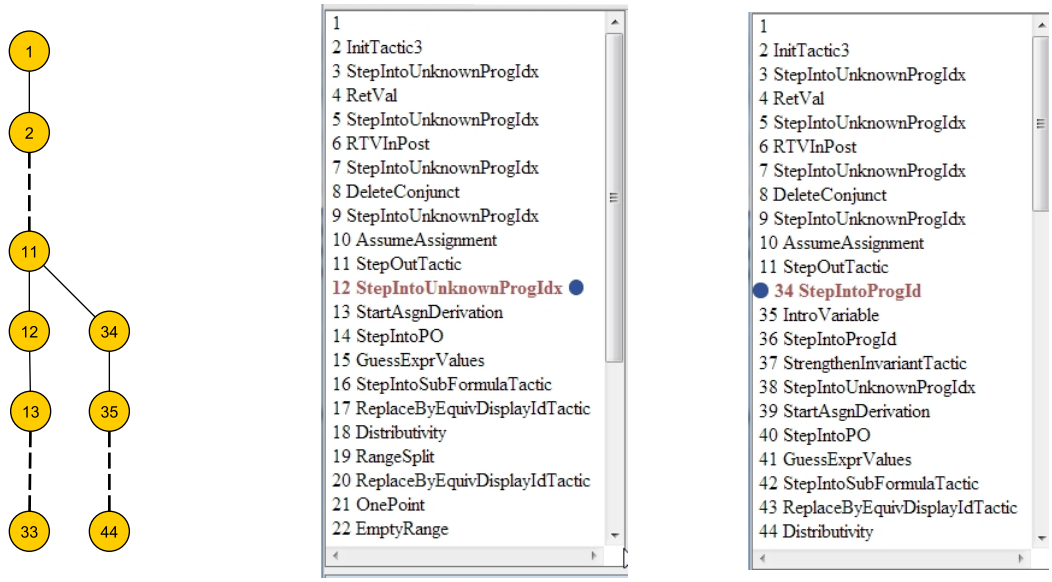
Loop invariants and other assertions help in understanding and verifying a program. However, they provide little clue about how the program designer might have discovered them. For example, at node G in the derivation in Fig. 5.1, we are unable to express the expression under consideration in terms of the program variables. This guides us to introduce a fresh variable s and strengthen the invariant with $s = Q(n)$. This crucial information is missing from the final annotated program. It is therefore desirable to preserve the complete derivation history to fully understand the derivation of the program. CAPS maintains the derivation history in the form a derivation tree. Maintaining history also facilitates backtracking and branching if users want to try out alternative derivation strategies.

Backtracking and Branching.

In CAPS, we do not allow programmers to directly edit the program; users have to backtrack and branch to try out different derivation strategies. This restriction ensures that the derivation tree contains all the information necessary to reconstruct the program from scratch. All the design decisions are manifest in the derivation tree which helps in understanding the rationale behind the introduction of various program constructs and invariants. Using the branching functionality, users can also explore multiple solutions for the given programming task.

Navigating the Derivation tree

The conventional tree interface is not suitable for displaying the derivation tree. At any point during a derivation, we are mostly interested in the active branch of the derivation tree. This active branch is shown in the left panel in the GUI. To make it easy to navigate to other branches, we show the sibling nodes of every node in the active branch. Users can navigate across the branches by clicking the sibling markers as shown in Fig. 7.6. If there are multiple branches under the selected sibling node, then the rightmost branch is selected.



(a) (b) (c)

Figure 7.6. Navigating the derivation tree: Fig. (a) shows schematic diagram of a derivation tree. Fig. (b) shows the path in the derivation tree containing the currently selected node (node 12). A marker (a filled circle) to the right of node 12 indicates the presence of a right-sibling node (node 34) in the derivation tree. Users can click on this sibling marker to switch to the branch containing *node 34*. The resulting path is shown in Fig. (c).

7.7 Implementing Assumption Propagation

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 *Propagate-Assumption* 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.

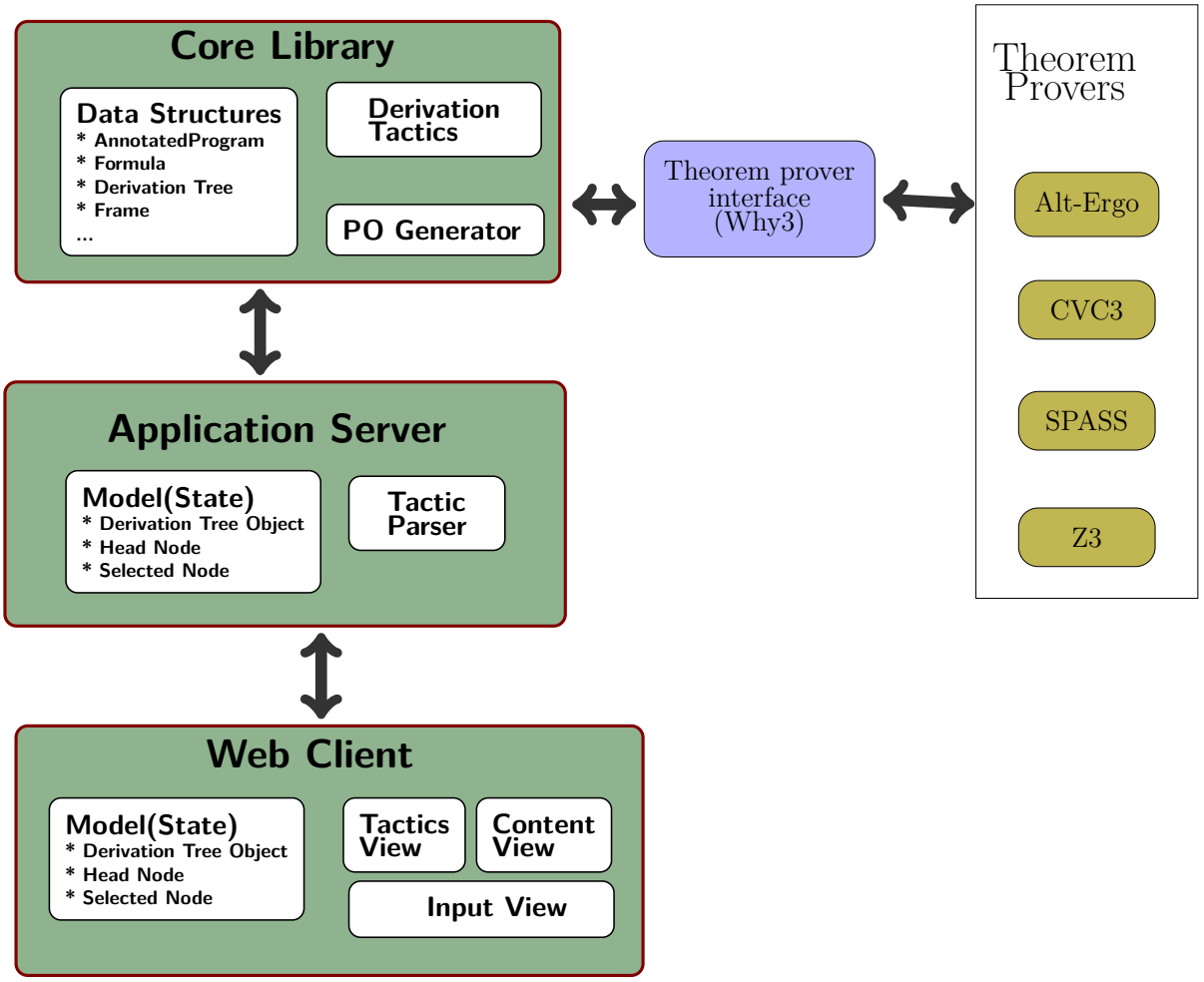


Figure 7.7. CAPS Architecture

7.8 System Architecture

The CAPS system is designed as a web application so that it can be deployed easily. The system is divided into following 3 main components (Fig. 7.7).

Core Library. The core library implements the main tactic application functionality and the underlying data structures (*annGCL*, *Formula*, *DerivationTree*, *DerivationTactic*, *Frame*, etc.). It also contains a repository of the program and the formula manipulation tactics. The library is integrated with various automated theorem provers (Alt-Ergo, CVC3, SPASS, and Z3) via the common interface provided by the *Why3* framework [FP13]. The library is implemented in *Scala* and uses the *Kiama* library [Slo11] for rewriting.

Application Server. The server stores the current state of the derivation. The appli-

cation also implements a tactic parser which parses the tactic request. The server component is implemented using the *Scala play* web framework [Pla].

Web Client. The *CAPS* application is implemented as a single-page web application based on the *Backbone.js* framework [Bac]. The client also maintains a state of the derivation in order to reduce server trips for navigational purpose to increase responsiveness of the application. The in-browser functionality is implemented in the *Typescript* language [Typ] (which complies to Javascript). The GUI module has different views to display the current state of the derivation.

7.9 Using the CAPS System

In this section, we discuss the use of the CAPS system in the classroom setting. We introduced the CAPS system in class during the last offering of the course. In Section 2.4, based on our experience during several offerings of the *Program Derivation* course, we have presented common difficulties faced by students in deriving programs without using any tool support. We discuss how various features of the system helped in addressing these difficulties.

Tactic Based Methodology. Students incrementally transform a formal specification into a fully derived program by applying *Derivation Tactics*. For example, two of the tactics that we employed in derivation in Figure 2.1 are *Replace constant by a variable*, and *Range Split*. To apply a tactic, one needs to select a tactic from a list and provide the required input parameters, and the tool automatically performs the corresponding formula manipulations. By forcing students to enter the required parameters, errors such as *CD4* are prevented. The tool ensures correctness after application of every tactic.

Derivation History and Backtracking. As the CAPS tool maintains the entire derivation history in the form a derivation tree, users can branch off from any point in the derivation to explore different derivation strategies. This helps take care of the errors resulting from *CD4* and *CD6*.

Focusing on Subcomponents. The functionality of stepping into subcomponents allows users to focus on the subprograms or subformulas of interest. With this functionality,

users need not drag along the context - which remains unchanged - while manipulating a subformula or subprogram thereby simplifying the derivations (*CD2*). This helps take care of the errors resulting from *CD6* as the derivation steps for manipulating the subcomponent are nested and can be collapsed while browsing the proofs.

Automating Formula Transformations. In the manual calculations, all the steps are kept small enough to be manually verified by the user. This is the main reason why the program derivations are long even for simple problems, and formal methods are hated by several students. With a tool support, however, we can afford to take large steps, as long as the readability is maintained. In general, small steps are good for readability. However, there are situations where certain calculation is not important from the derivation point of view. We would like to automate such calculations. We employ a backend theorem prover to perform required proofs. This makes the program calculations flexible and reduces the derivation length. This helps with the observed errors *CD2* and *CD5*.

Calculations not involving any metavariables should be automated to the extent possible. For example, in Fig. 2.1, we skipped the proof of preservation of the invariant $P_1 : 0 \leq n \leq N$. As no metavariable is involved this proof obligation, it is uninteresting from the derivation viewpoint. Students resent doing such proofs. We, however, still need to discharge them to ensure correctness. The proof obligation for P_1 can be directly transformed to *true* by applying the *VerifiedTransformation* tactic which uses backend theorem provers. The introduction of this tactic takes care of the errors *CD1* and *CD3*.

In case of failure of external theorem provers in automatically discharging a proof obligation, we have to carry out the detailed interactive step-by-step proof. A failed calculational proof often provides clues about how to proceed further with the derivation. Automated formula transformations take care of the most of the common logic related errors (*CD0*).

Assumption Propagation. The functionality of assumption propagation reduces the ad hoc reasoning and the resulting trial and error by allowing users to propagate assumptions to appropriate places. As seen in Section 5.4.2, this simplifies the derivations by avoiding unnecessary backtrackings. This feature helps in reducing the errors *CD2* and *CD6*.

7.9.1 Evaluation

The CAPS system received very enthusiastic response from the students. We did an anonymous survey to get specific feedback about the system. There were a total of fourteen responses. Ten students felt that the use of the tool increased their confidence in the correctness of the derived program, while three did not feel so, and one student was unsure. Same pattern was observed for the question whether the tool simplifies or complicates the task of the derivation. To the question of how would they like to derive the programs in future, five said using the tool alone, six said that they would like to use the tool along with paper and pencil, and three students commented that they would not use the tool. Eight out of the fourteen students also felt that the tool should have been introduced right from the beginning of the semester, while three suggested introduction around the middle of the semester, and three students felt that the tool should not be introduced at all but they did not write any comments. Due to the anonymity of the survey, we are unable to determine why three students did not like the tool at all.

Overall, we are quite happy with the use of tool in the course. The biggest advantage was that the students could not submit incorrect derivation. They could only submit either correct or partially correct answers; since programs were correct-by-construction at all stages (although they may have been incomplete). We could look at the derivation history of partial submissions and identify the problems because of which they were stuck at a particular point. Students were happy about the fact that they knew that their solution was correct before making the submission. Note that this adds a completely new dimension to the concept of automatic grading of assignments [ER09].

One unexpected downside of the introduction of the tool was the increase in ad-hocism in some of the derivations. In the class, we teach various derivation heuristics and the associated proof obligations, and students are supposed to follow them in the manual derivation. However, with the tool trying to automatically discharge proof obligations, some students make wild guesses about the required program constructs, resulting in very inelegant programs. For example, rather than deriving the value of s' in Figure 2.1(f), many students introduce several *if* statements enumerating different cases involving positive and negative values of s and $A[n]$. In comparison, *max* operator in our derivation can be implemented using a single *if*. Note that these programs were inelegant compared to what is possible with the derivation methodology, and not compared to what is achieved

in the standard guess and test methodology. Essentially, these students use the tool as a program verification system and not as a program derivation system.

7.10 Related Work

All the publicly available IDEs lack in one respect or another with respect to the features important for our purpose (for example, structured calculations, integration with multiple theorem provers, backtracking and branching, assumption propagation, etc.). Tools like Why3 [FP13], Dafny [Lei10], VCC [CDH+09] and VeriFast [JP08] generate the proof obligations and try to automatically discharge these proof obligations. Although the failed proof obligations provide some hint, there is no structured help available to the users in the actual task of implementing the programs. Users often rely on ad hoc use cases and informal reasoning to guess the program constructs.

Systems like Cocktail [Fra99], Refine [OXC04], Refinement Calculator [BL96b] and PRT [CHN+96b] provide tool support for the refinement based formal program derivation. Cocktail offers a proof-editor for first-order logic which is partially automated by a tableau based theorem prover. However, the proof style is different from the calculational style. Refine has a plug-in called Gabriel which allows users to create tactics using a tactic language called *ArcAngel*. Refine and Gabriel are not integrated with theorem provers and do not support discharging of proof obligations. In case of Refinement Calculator and PRT, the program constructs need to be encoded in the language of the underlying theorem prover. In CAPS, our goal has been to be theorem-prover agnostic, so that we can exploit the advances made in different theorem provers.

The KIDS [Smi90] and the Specware [SJ95] systems provide operations for the transformational development of programs and have been very successful in synthesizing efficient scheduling algorithms. However, these systems are targeted towards expert users. Jape [BS97] is a proof calculator for interactive and step-by-step construction of proofs in natural-deduction style. Although Jape supports Hoare logic, it is mainly intended for proof construction whereas CAPS is focused on program derivation and has many tactics specific to program calculations.

Chapter 8

Conclusion and Future Work

In this thesis, we set out to address the problem of cumbersome and error-prone derivations in the context of calculational program derivation. We also aimed at developing a theory for propagating assumptions in order to reduce the ad hoc reasoning and associated rework. Towards these goals, we have developed a tool and a methodology for calculational derivation of imperative programs. We have developed automated theorem prover assisted tactics, which when coupled with the functionality of extracting context of subcomponents, significantly reduce human efforts and simplify the derivations. We have filled a gap in the program derivation literature by developing correctness preserving assumption propagation rules which help in reducing guesswork and associated rework. We have designed and implemented a program derivation system called CAPS and demonstrated the utility of the proposed techniques in the system. The system was used in the Program Derivation class at IIT Bombay, and the overall feedback was encouraging. The system avoided, reduced, or otherwise managed most of the common errors we had identified in the previous offerings of the course.

Although the contributions of the thesis were discussed in the context of the calculational style and the CAPS system, they are applicable to broader settings. For example, the automatic simplification of formulas can be useful in any interactive system that deals with large formulas. The assumption propagation rules we proposed can be adapted for use in other formal program construction systems. We give one application in the next section where these rules can be used for formal refactoring of programs. Apart from the theoretical contributions, other practical details like the interface for carrying out structured proofs, tactic mechanism for transitioning between program and formula modes,

and the integration with theorem provers can be useful to other researchers interested in building similar systems. We have open-sourced the system for others to use and improve upon.

Future Work

Expanding the Scope of the Methodology. In the current work, the target programming language we have chosen is very simple. To expand the scope of the methodology, we can extend the methodology to handle richer programming language constructs like function calls, recursion, algebraic data types and polymorphic types. This work will involve development of transformation rules for these constructs and encoding the advanced data types in the external theorem provers.

Integrating Synthesis Solvers. Automated theorem provers help in automatically discharging the applicability conditions. The theorem provers however are of not much help during application of rules involving instantiation of metavariables. The syntax guided synthesis solvers (*SyGuS* solvers) [ABJ+13] can be used to find appropriate expressions for the metavariables. This will enable us to develop powerful meta-tactics to handle quantifiers in the program calculations. We can also employ a *SyGuS* solver to synthesize loop-free subprograms during an interactive derivation. The focus of *SyGuS* is on theories T for which satisfiability modulo T decision procedures are available. In CAPS, we do not want to restrict the specification language to certain theories. We can, however, employ a *SyGuS* solver during the derivation when the specification of the subprogram under consideration is in a theory for which a *SyGuS* solver is available. This has the potential of significantly simplifying the program calculations.

Formal Code Refactoring. Code refactoring involves restructuring existing computer code without changing its external behavior. It typically involves incrementally applying small transformations to the code. Before refactoring, one must have a comprehensive test suite so that after the refactoring, it can be verified that the refactored code is behaving as expected. However, there is always a chance of introducing errors as the test suite does not capture the complete functional behavior. In this thesis, we focused on annotated program transformation rules for synthesizing programs and propagating assumptions. Similar rules

can be developed for transforming a concrete annotated program to another functionally equivalent annotated program. Such rules can be used for refactoring the code while ensuring correctness. This functionality would be helpful in development environments that support invariant annotations.

Calculational Methodology for the Design of Reactive Systems. In [CD11], we have proposed a methodology for generating a hierarchical representation of the system for visualizing Event-B [Abr10] models. We use Hierarchical Abstract State Transition Machine (HASTM) representation, which uses the concepts of hierarchical states and guarded transitions similar to those in statechart diagrams. This representation makes the pre- and postconditions of the transitions explicit. Each transition is labeled with a guard and an action. The HASTM representation can be used to devise a calculational methodology for the design of discrete transition systems. Given a partial description of a transition, the missing constituents can be calculated systematically. For example, users can guess the precondition, the postcondition, and the action for a transition, and then calculate the guard of a transition by adopting the calculational methodology. This functionality would be beneficial in reducing the ad hoc steps in the system construction.

Appendix A

BigMax Theory in Why3

```
module CapsUtil

    use import int.Int

    use HighOrd as HO

    predicate non_empty_range(r: int -> bool) = (exists i: int. r i)

    predicate empty_range(r: int -> bool) = not(non_empty_range r)

    predicate one_point(r: int -> bool)(x: int) =
        r x /\ (forall i: int. r i  -> i = x)

    predicate is_union_of(u p q: int -> bool) =
        forall x: int. u x <-> p x \/ q x

    predicate is_union_of_v (ru r: int -> bool)(v: int) =
        forall x: int. ru x <-> (r x \/ x = v)

    predicate disjoint(p q: int -> bool) =
        forall x: int.  not(p x /\ q x)

    predicate are_same(t1 t2: int -> int) =
        forall x: int. t1 x = t2 x

end
```

```
module BigMax

    use import int.Int

    use HighOrd as HO

    use import capsUtil.CapsUtil as CapsUtil

    constant bigMax: (int -> bool) -> (int -> int) -> int

    function max(x y: int): int = if (x > y) then x else y

    axiom onePointMax:
        forall r: int -> bool, t: int -> int, x: int.
            one_point r x ->
                bigMax r t = t x

    axiom splitRangeMax:
        forall r r1 r2: int -> bool, t t1 t2: int -> int.
            non_empty_range r -> non_empty_range r1 -> non_empty_range r2 ->
            is_union_of r r1 r2 ->
            are_same t t1 -> are_same t t2 ->
                bigMax r t = max (bigMax r1 t1) (bigMax r2 t2)

    axiom splitRangeRightMax:
        forall r r1: int -> bool, t t1 t2: int -> int, v: int.
            non_empty_range r -> non_empty_range r1 ->
            is_union_of_v r r1 v ->
            are_same t t1 -> are_same t t2 ->
                bigMax r t = max (bigMax r1 t1) (t2 v)

end
```

Bibliography

[ABJ+13] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2013.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering.* Cambridge University Press, 2010.

[ADG+01] Vicki L. Almstrum, C. Neville Dean, Don Goelman, Thomas B. Hilburn, and Jan Smith. Support for teaching formal methods. *SIGCSE Bull.*, 33(2):71–88, June 2001.

[Bac] Backbonejs javascript library. http://backbonejs.org/.

[BGVW97] Ralph Back, Jim Grundy, and Joakim Von Wright. Structured calculational proof. *Formal Aspects of Computing*, 9(5-6):469–483, 1997.

[BKSS97] Yves Bertot, Thomas Kleymann-Schreiber, and Dilip Sequeira. Implementing proof by pointing without a structure editor. Technical Report ECS-LFCS-97-368, University of Edinburgh, 1997.

[BL96a] 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.

[BL96b] Michael Butler and Thomas Långbacka. Program derivation using the refinement calculator. In *Theorem Proving in Higher Order Logics: 9th Inter-*

national 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.

[BS97] Richard Bornat and Bernard Sufrin. Jape: A calculator for animating proof-on-paper. In *Automated DeductionCADE-14*, pages 412–415. Springer, 1997.

[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.

[CD11] Dipak L Chaudhari and Om P Damani. Generating hierarchical state based representation from event-b models. *Proceedings of the B 2011 Workshop, Electronic Notes in Theoretical Computer Science*, 280:35–46, 2011.

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *Theorem Proving in Higher Order Logics*. Springer, 2009.

[CGG12] Guido Caso, Diego Garbervetsky, and Daniel Gorín. Integrated program verification tools in education. *Software: Practice and Experience*, 2012.

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

[CHN+96b] 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.

[Cow10] A. J. Cowling. Stages in teaching formal methods. In *23rd IEEE Conference on Software Engineering Education and Training*, 2010.

[DF88] Edsger W. Dijkstra and W. H. Feijen. *A Method of Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[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.

[DLC06] Isabelle Dony and Baudouin Le Charlier. A tool for helping teach a programming method. In *Proc. of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE*, 2006.

[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.

[ER09] J. English and T. Rosenthal. Evaluating students' programs using automated assessment - a case study. In *Proc. of the Conference on Integrating Technology into Computer Science Education, ITiCSE*, 2009.

[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.

[GJTV11] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In Mary W. Hall and David A. Padua,

editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 62–73. ACM, 2011.

[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.

[Gru92] Jim Grundy. A window inference tool for refinement. In *5th Refinement Workshop*, pages 230–254. Springer, 1992.

[Gru93] Jim Grundy. *A Method of Program Refinement*. PhD thesis, University of Cambridge Computer Laboratory, Cambridge, England, 1993.

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

[JP08] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Dept. of Computer Science, Katholieke Universiteit Leuven, 2008.

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

[Lau04] Kung-Kiu Lau. A beginner's course on reasoning about imperative programs. In C.Neville Dean and RaymondT. Boute, editors, *Teaching Formal Methods*, volume 3294 of *LNCS*. Springer Berlin Heidelberg, 2004.

[Lei10] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, 2010.

[LP14] K Rustan M Leino and Nadia Polikarpova. Verified calculations. In *Verified Software: Theories, Tools, Experiments*, pages 170–190. Springer, 2014.

[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.

[Mat] Mathquill: a javascript library for latex, http://mathquill.com/.

[MM01] Panagiotis Manolios and J. Strother Moore. On the desirability of mechanizing calculational proofs. *Inf. Process. Lett.*, 77(2-4):173–179, February 2001.

[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.

[Pla] Play: A web framework for java and scala. http://www.playframework.com/.

[RS93] Peter J Robinson and John Staples. Formalizing a hierarchical structure of practical mathematical reasoning. *Journal of Logic and Computation*, 3(1):47–61, 1993.

[SGF10] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *POPL 2010*, pages 313–326, New York, NY, USA, 2010.

[SJ95] Yellamraju V. Srinivas and Richard Jüllig. *Specware: Formal support for composing software*, pages 399–422. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[Slo11] Anthony M. Sloane. Lightweight language processing in kiama. In JooM. Fernandes, Ralf Lmmel, Joost Visser, and Joo Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 408–425. Springer Berlin Heidelberg, 2011.

[SLTB+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *ACM SIGARCH Computer Architecture News*, 34(5):404–415, 2006.

[Smi90] Douglas R. Smith. Kids: A semiautomatic program development system. *IEEE transactions on software engineering*, 16(9):1024–1043, 1990.

[Sol86] Elliot Soloway. What to d0 next: Meeting the challenge of programming-in-the-large. In *Empirical Studies of Programmers: First Workshop*, volume 1, page 263. Intellect Books, 1986.

[SW14] M. Sitaraman and B.W. Weide. Special session: "hands-on" tutorial: Teaching software correctness with resolve. In *SIGCSE 2014 - Proc. of the 45th ACM Technical Symposium on Computer Science Education*, 2014.

[Typ] Typescript: a language for application-scale javascript development, http://www.typescriptlang.org/.

[vW98] Joakim von Wright. Extending window inference. In *Theorem Proving in Higher Order Logics*, pages 17–32. Springer, 1998.

[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.

[Why] The why3 platform reference manual. http://why3.lri.fr/doc-0.85/.

List of Publications

Journals

- Chaudhari, D. L. and Damani, O. Assumption Propagation through Annotated Programs. *Formal Aspects of Computing (Accepted with minor revisions).*

Conferences

- Chaudhari, D. L. and Damani, O. Combining Top-down and Bottom-up Techniques in Program Derivation. *25th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2015.*

- Chaudhari, D. L. and Damani, O. Introducing Formal Methods via Program Derivation. *20th Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE 2015.*

- Chaudhari, D. L. and Damani, O. Automated Theorem Prover Assisted Program Calculations. *The 11th International Conference on Integrated Formal Methods, iFM 2014.*

Workshops

- Chaudhari, D. L. and Damani, O. Building an IDE for the Calculational Derivation of Imperative Programs. *Workshop on Formal-IDE, 2015.*

- Chaudhari, D. L. and Damani, O. Generating Hierarchical State Based Representation From Event-B Models. *B Workshop, 2011.*

Acknowledgments

I would like to thank my advisor, Prof. Om Damani, for his guidance, support and encouragement throughout my years of study. He has been an excellent advisor, mentor, and friend. He has always been available and willing to discuss various issues be they technical, administrative, or personal. He believed in my abilities and gave me the freedom to pursue the topic of my interest. The five offerings of his *Program Derivation* course, for which I was a teaching assistant, have been instrumental in building the foundations and in shaping my research interests and thesis goals.

I am thankful to my research progress committee members Prof. Supratik Chakraborty, Prof. Abhiram Ranade, and Prof. Amitabha Sanyal who have all been very supportive, encouraging, and challenging. With their expertise in diverse fields such as formal methods, algorithms, and functional programming, they brought diverse perspectives which enriched my educational and research experience.

I am grateful to Prof. Claude Marché for hosting me at Toccata research group, INRIA Saclay. He along with Jean-Christophe Filliâtre, Andrei Paskevich helped me in integrating the *Why3* framework and in coming up with the theories for the arithmetic quantifiers.

I thank Prof. Ralph-Johan Back for hosting me at the Åbo Akademy, Finland and giving my valuable feedback on my work. I thank Dr. Johannes Eriksson for explaining the workings of the SOCOS environment and Dr. Viorel Preoteasa for the insightful discussions on theorem prover integration.

I thank Prof. Dominique Méry for giving valuable feedback at the FIDE workshop. I thank Prof. Reiner Hähnle, Dr. Richard Bubel, and Dr. Martin Hentschel for helping me with the Key Verification framework during my stay at TU Darmstadt, Germany. Special thanks to Dr. Stijn de Gouw for explaining the verification of Tim Sort algorithm in the Key framework.

Many thanks to my friends and colleagues for their support and encouragement. Special thanks to Nikhil Hooda, Pratik Jawanpuria, Anindya Sen, Naveen Nair, Mitesh Khapra, Devendra Bhave, Abhisekh Sankaran, Hrishikesh Karmarkar, Ramesh Gopalakrishnan, Vaibhao Tatte, Dhaval Bonde, Dhritiman Das, Aditya Joshi, Balamurali A R. I always turned to them for advice on various issues. I would also like to thank my friend Avinash Mane who is my go-to man for practical advice in times of uncertainty.

Further thanks go to the students of the *Program Derivation* class who participated in the surveys and gave constructive feedback on the program derivation system. I would also like to thank the office staff of the Computer Science Department at IIT Bombay for their assistance. In particular, I would like to thank Vijay Ambre for prompt handling of all the requests.

I would like to thank Mrs. Seema Periwal for all the help and making my family feel at home during our stay in the campus.

I thank the Ministry of Human Resource Development (MHRD), Government of India for the PhD assistantship, the Tata Consultancy Services (TCS) for supporting the work through a research fellowship, IIT Bombay and Google India for the conference travel grants.

Finally, I can never thank enough my family: my mother Mrs. Usha Chaudhari and my sister Mrs. Bimba Chaudhari for their unconditional love and support; my wife Deepa for single handedly managing the household in addition to a full time job; my son Pinak, who is now five, for being quite understanding for his age. Without your support I would never have been able to achieve my goals. Thank you!

Date: _____ Dipak Liladhar Chaudhari