Automated Theorem Prover Assisted Program Calculations

Dipak L. Chaudhari and Om Damani

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

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 provers 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. We have implemented this approach in a Calculational Assistant for Programming from Specifications (CAPS). With the help of simple examples, we show how the calculational assistant helps in taking the drudgery out of the derivation process while ensuring correctness.

Keywords: Calculational Style, Program Derivation, Correct by Construction, Program Correctness

1 Introduction

Calculational Style of Programming [11], [18] is a programming methodology wherein programs are systematically derived from their formal specifications. At every step in the derivation process, a partially derived program/formula is transformed into another form, by following certain heuristic guidelines. The derived programs are correct-by-construction since correctness is implicit in the derivation. The calculational style is known for its readability and rigour. 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 [18]. Although very appealing, there are several practical difficulties in effectively adopting this methodology. For many programming problems, the derivations are long and difficult to organize. As a result, the derivations, if done manually, are error-prone and cumbersome. To address these issues, the present work takes inspiration from various approaches from the fields of program verification, automated theorem proving, and interactive theorem proving to design and build a Calculational Assistant for Programming from Specifications (CAPS)[1]. Our aim has been to address the difficulties as-

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

sociated with the pen-and-paper calculational style while retaining the positive aspects.

Various tools exists to assist programmers in verifying the correctness of programs during the implementation phase itself. Tools like Dafny [19], Why3 [13], VCC [7] and VeriFast [17] generate the verification conditions and try to automatically prove these verification conditions. Dafny even has an extension called *poC* (program-oriented calculations) [20] which provides support for automatic verification of calculational proofs. However, the primary focus of such tools being the verification of programs, these tools provide limited guidance to the programmer in the actual task of deriving the programs.

There is a recent trend in program synthesis in which the programmer provides a syntactic template for the desired program in addition to the correctness specification [16], [23], [24]. These are automatic approaches which require a syntactic template to be provided by the user. Our focus, on the other hand, is on calculational derivation in an interactive setting.

Tools like Refinement Calculator [5] and PRT [6] provide tool support for the refinement based formal program derivation. Refinement Calculator uses HOL as an underlying proof engine. The PRT tool has similar goals; it extends the Ergo theorem prover and provides an Emacs based user interface. With these tools, the program constructs need to be encoded in the language of the underlying theorem prover. We chose not to tightly couple the system with any particular theorem prover. CAPS has built-in refinement rules and the system generates the required correctness proof obligations. We have tried to keep the notation and style of the derivation as close as possible to the pen-and-paper calculational style which is known for its readability. Our main emphasis has been on developing theorem prover assisted tactics to reduce the length of the derivations.

The main contributions of this work are as follows. (a) We have designed and implemented a calculational assistant for derivation of imperative programs. The tool provides a tactic based framework for carrying out program as well as formula transformations in a coherent way. (b) We have extended the *Structured Calculational Proof* format [2] by making the transformation relation explicit and by adding metavariable support. 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. The automated theorem prover (ATP) assisted tactics also help in carrying out derivations that are not amenable to the calculational style. (c) By providing a unified framework for carrying out formula as well as program transformations, we have kept the derivation style in CAPS close to the calculational style.

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 while ensuring correctness.

2 Motivating Example

In this section, we derive – without using any tool support – a simple program by following the calculational style of program derivation. This exercise highlights the complex user interactions usually involved in a typical program derivation session. We use this example in later sections to motivate and illustrate the main features of CAPS.

Consider the following programming problem specified in a natural language (adapted from exercise 4.3.4 in [18]).

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

One of the several ways to formally specify this problem is shown in Fig. 1(a) where S denotes the program to be derived. For representing quantified expressions, we use the *Eindhoven* notation $(OP\, i : R : T)$ [18] 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 typically involving the quantified variables, and T is the *Term* - an expression. This notation is usually used for arithmetic quantified terms ($\sum$, $\prod$). By using the same notation for all the quantified terms – including the logical quantified terms ($\forall$, $\exists$) – we can have generalized calculational rules.

We start by analysing the shape of the postcondition R and apply the *Replacing Constants by Variables* heuristics [18]. In particular, we introduce a fresh variable n, add bounds on n, and rewrite postcondition R as

$$\begin{array}{l}\left(r \equiv \left(\exists p : 0 \le p \le n : \left(\begin{array}{c}(\forall i : 0 \le i < p : f[i]) \\ \wedge\, (\forall i : p \le i < n : \neg f[i])\end{array}\right)\right)\right) \\ \wedge\, 0 \le n \le N \;\wedge\; n = N\end{array}$$

We then apply the *Taking Conjuncts as Invariant* heuristics [18] to arrive at loop invariant $P_0 \wedge P_1$ and guard $\neg\,(n = N)$ where P_0 and P_1 are as follows.

$$\begin{array}{l}P_0 : \left(r \equiv \left(\exists p : 0 \le p \le n : \left(\begin{array}{c}(\forall i : 0 \le i < p : f[i]) \\ \wedge\, (\forall i : p \le i < n : \neg f[i])\end{array}\right)\right)\right) \\ P_1 : 0 \le n \le N\end{array}$$

We observe that P_0 and P_1 can be established initially by $r, n := true, 0$. At this stage, we arrive at the program shown in Fig. 1(b) as the solution for S. We investigate an increase of n by 1 and envision the multiple assignment $r, n := r', n + 1$ for S_0 where r' is a placeholder for the unknown expression.

From the *Invariance Theorem* [18], the proof obligation for invariance of P_0 is $P_0 \wedge P_1 \wedge n \neq N \Rightarrow wp\,(r, n := r', n + 1;, P_0)$ where wp is the *weakest precondition predicate transformer* [12]. To calculate r', we assume P_0, P_1, and $n \neq N$ and simplify the consequent of this formula as shown in Fig. 1(c). Every step in the calculation is associated with the relation to be maintained ($\equiv$ in this case) and a hint justifying the step. For brevity, we skip the proof of preservation of P_1.

con N: int $\{N \geq 0\}$; **var** f: array $[0..N)$ of bool;
var r: bool;
S
$R : \left\{ r \equiv \left(\exists p : 0 \leq p \leq N : \begin{pmatrix} (\forall i : 0 \leq i < p : f[i]) \\ \wedge (\forall i : p \leq i < N : \neg f[i]) \end{pmatrix} \right) \right\}$

(a)

$r, n := true, 0;$
$\left\{ \begin{array}{l} inv : P_0 \wedge P_1 \\ bound : N - n \end{array} \right\}$
do $n \neq N \rightarrow$
$\quad S_0$
od

(b)

0	$wp\,(S_0, P_0)$
1	$\equiv$ { envision $r, n := r', n+1$ for S_0}
2	$wp\,(r, n := r', n+1;, P_0)$
4	$\equiv$ { definition of P_0 and assignment }
5	$r' \equiv (\exists p : 0 \leq p \leq n+1 : (\forall i : 0 \leq i < p : f[i]) \wedge (\forall i : p \leq i < n+1 : \neg f[i]))$
6	$\equiv$ { split off $p = n+1$; $0 \leq n+1$}
7	$r' \equiv \left(\begin{array}{l} (\exists p : 0 \leq p \leq n : (\forall i : 0 \leq i < p : f[i]) \wedge (\forall i : p \leq i < n+1 : \neg f[i])) \\ \vee (\forall i : 0 \leq i < n+1 : f[i]) \wedge (\forall i : n+1 \leq i < n+1 : \neg f[i]) \end{array} \right)$
8	$\equiv$ { empty range rule }
9	$r' \equiv \left(\begin{array}{l} (\exists p : 0 \leq p \leq n : (\forall i : 0 \leq i < p : f[i]) \wedge (\forall i : p \leq i < n+1 : \neg f[i])) \\ \vee (\forall i : 0 \leq i < n+1 : f[i]) \end{array} \right)$
10	$\equiv$ { split off $i = n$}
11	$r' \equiv \left(\begin{array}{l} (\exists p : 0 \leq p \leq n : (\forall i : 0 \leq i < p : f[i]) \wedge (\forall i : p \leq i < n : \neg f[i]) \wedge \neg f[n]) \\ \vee (\forall i : 0 \leq i < n+1 : f[i]) \end{array} \right)$
12	$\equiv$ { distribute $\wedge$ over $\exists$ since $\neg f[n]$ does not have free occurrence of p}
13	$r' \equiv \left(\begin{array}{l} ((\exists p : 0 \leq p \leq n\colon (\forall i : 0 \leq i < p\colon f[i]) \wedge (\forall i : p \leq i < n\colon \neg f[i])) \wedge \neg f[n]) \\ \vee (\forall i : 0 \leq i < n+1 : f[i]) \end{array} \right)$
14	$\equiv$ { invariant P_0}
15	$r' \equiv (r \wedge \neg f[n]) \vee (\forall i : 0 \leq i < n+1 : f[i])$
16	$\equiv$ { add invariant $P_2 : s \equiv (\forall i : 0 \leq i < n : f[i])$ and assume $P_2(n := n+1)$.}
17	$r' \equiv (r \wedge \neg f[n]) \vee s$
18	$\equiv$ { instantiating r' to $(r \wedge \neg f[n]) \vee s$}
19	$true$

(c)

$r, n, s := true, 0, true;$
$\left\{ \begin{array}{l} inv : P_0 \wedge P_1 \wedge P_2 \\ bound : N - n \end{array} \right\}$
do $n \neq N \rightarrow$
$\quad \{P_2\}\, S_1\, \{P_2(n := n+1)\};$
$\quad r, n := (r \wedge \neg f[n]) \vee s, n+1;$
od

(d)

$r, n, s := true, 0, true;$
$\left\{ \begin{array}{l} inv : P_0 \wedge P_1 \wedge P_2 \\ bound : N - n \end{array} \right\}$
do $n \neq N \rightarrow$
$\quad s := s \wedge f[n];$
$\quad \{P_2(n := n+1)\}$
$\quad r, n := (r \wedge \neg f[n]) \vee s, n+1$
od

(e)

Fig. 1. Calculational derivation of the motivating example.

In step 15 in Fig. 1(c), the expression under consideration is neither easily computable nor easily expressible in terms of the program variables. We, therefore, introduce a variable s and add an invariant $P_2 : s \equiv (\forall i : 0 \leq i < n : f[i])$. We now observe that P_2 can be established initially by $s := true$ since the uni-

versal quantification over an empty range equals *true*. With this we arrive at the program shown in Fig. 1(d).

Program S_1 has been added to ensure that $P_2(n := n + 1)$ is a precondition of the assignment to r. For S_1, we envision the assignment $s := s'$. By following the same procedure as before, we can calculate the value of s' to be $s \wedge f[n]$. The final derived program is presented in Fig. 1(e).

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.

3 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, 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 [2], 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 [16], [23], [24] 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 the user more control to explore alternative solutions since all the design decisions are manifest in the derivation.

Using the Why3 tool [13] as an interface, we have integrated automated theorem provers Alt-Ergo [8], CVC3 [4], SPASS [25] and Z3 [9] with CAPS. The Why3 platform provides two languages: a logic language (Why) and a ML-like programming language (WhyML). In CAPS, we use only the logic language since we are using Why3 only for the purpose of interacting with various theorem provers. The proof obligation formulas are transformed to Why3's logic language and Why3 is invoked in the background to prove these proof obligations with the help of various theorem provers. Using Why3 as an interface saves us from dealing with the different logical languages and predefined theories of various theorem provers.

We next describe the CAPS system and its specific features which play vital role in designing the theorem prover assisted tactics.

4 Calculational Assistant

CAPS is a calculational assistant for programming from specifications (precondition and postcondition) specified in first-order logic. The core component of CAPS is implemented in the Scala programming language. An interactive user interface is provided in the form of a web application. The web application uses AJAX to provide a responsive user interface.

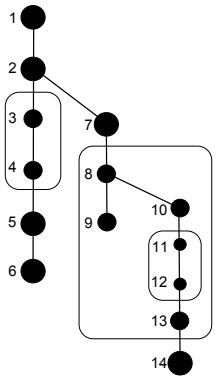


Fig. 2. A schematic representation of a Synthesis Tree

The derivation style in CAPS is very similar to the calculational style explained in Section 2. Users start the derivation by providing the formal specification of the program and then incrementally transform it into a fully derived annotated program by applying predefined transformation rules called *Synthesis Tactics*. The complete derivation history is recorded in the form of a tree called *Synthesis Tree*. Fig. 2 shows a schematic representation of the *Synthesis Tree*. The portions of the tree enclosed by rectangles correspond to the transformations performed on the subprograms. This functionality is explained in Section 5.1. Users have a facility to backtrack to any node in the Synthesis Tree and branch out to explore different derivation possibilities. The final output of the derivation is a fully synthesized *AnnotatedProgram* (explained in Section 4.1) along with the complete derivation history.

The GUI of the tool is shown in Fig. 3. The tactics panel shows the list of applied tactics (a path in the synthesis tree), the content panel shows the program/formula corresponding to the selected node, and the input panel is shows a input form used for a applying tactic. CAPS has in-built tactics for transforming partially derived annotated programs and proof obligations formulas.

While reasoning about a program fragment, it is natural to treat them as a formula (a Hoare triple) whose details needs to be worked out to make the formula valid. In CAPS, however, we treat programs and formulas differently as

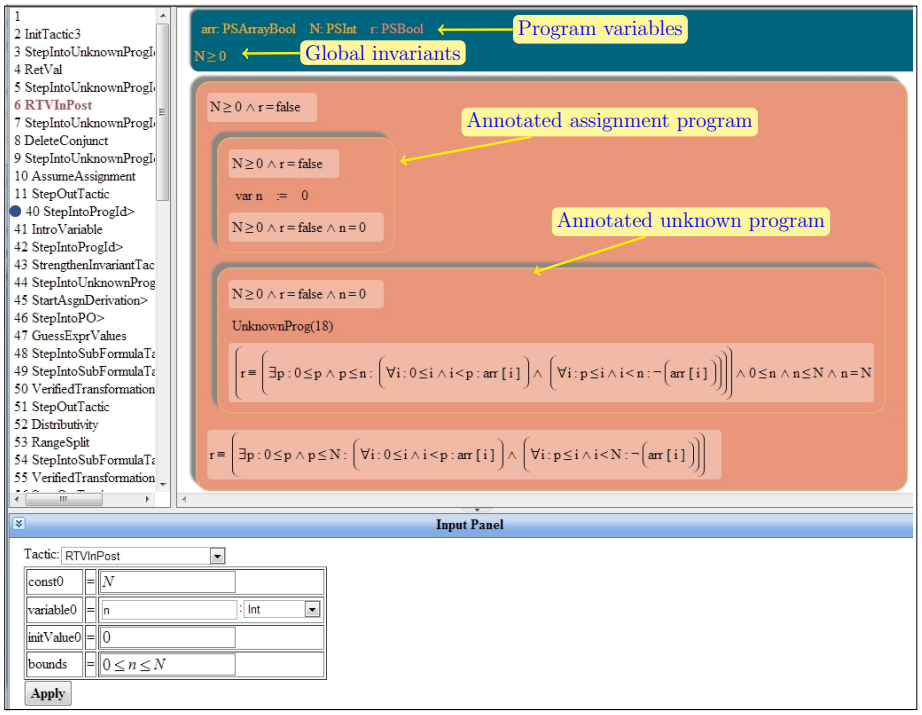


Fig. 3. CAPS GUI. There are three panels: *Tactics Panel* (left), *Content Panel*(center) and *Input Panel* (bottom). The input panel shows the input form for the *RTVInPost* (Replace term by variable) tactic.

they differ in many aspects: (a) Transformation rules for programs and formulas are quite different. (b) Program context consists of program variables whereas formula context consists of set of assumptions. (c) Visual representations of programs and formulas are quite different. By treating programs and formulas differently and keeping separate context management mechanisms for them, we are able to reason directly at the program level. This also helps in displaying programs and formulas in their natural form in the graphical user interface.

4.1 Program Transformations

For representing a program fragment and its specification, we introduce a data structure called *AnnotatedProgram*. It is obtained by augmenting each program construct in the Guarded Command Language (GCL) [10] with its precondition and postcondition. We also introduce a new program construct *UnknownProg* to represent an unsynthesized program fragment.

The program transformation tactics are based on the refinement rules from the refinement calculus [3], [21] and the high level program derivation heuristics

from the literature on calculational program derivation [12], [18]. For example, consider the program transformation tactics shown in Fig. 4. The *Weaken the Precondition* tactic captures the 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 [18]. The main difference between the rules in the refinement calculus and the transformation tactics in CAPS is that the refinement rules gradually transform the specification to a program (without annotations) whereas our program transformation tactics transform a partially derived annotated program to a fully derived annotated program.

Tactic: Weaken the Precondition. Input: R Applicability condition: $P \Rightarrow R$	$\{P\}$ $UnknownProg\,(1)$ $\{Q\}$ → $\{P\}$ $\{P\}\ SkipProg\ \{R\}$; $\{R\}\ UnknownProg(2)\ \{Q\}$ $\{Q\}$

Tactic:
Take Conjuncts as Invariants.
Inputs:
Invariant conjuncts: R_1
Variant: t
Applicability condition:
$P \Rightarrow R_1$
	$\{P\}$
	$UnknownProg\,(1)$
	$\{R_1 \wedge R_2\}$ → $\{P\}$
	$\{inv : R_1\}\{variant : t\}$
	While($\neg R_2$) {
	$\{R_1 \wedge \neg R_2\}$
	$UnknownProg\,(2)$
	$\{R_1\}$
	}
	$\{R_1 \wedge R_2\}$

Fig. 4. Program transformation tactics.

4.2 Formula Transformations

As discussed in Section 2, 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. This functionality is implemented in CAPS by using metavariables to represent the placeholders.

Some steps in the derivations involve transformation of annotated 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 annotated program containing metavariables, a new formula node representing the proof obligations (verification conditions) is created in the synthesis 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 annotated program with all the metavariables replaced by the corresponding instantiations.

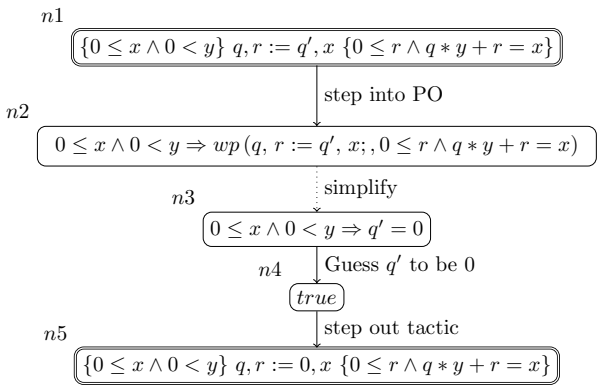


Fig. 5. A path in the synthesis tree for the Integer Division program. The *StepIntoPO* tactic is used to create a formula node corresponding to the proof obligation of the program node.

Example. Fig. 5 shows a path in the synthesis 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 synthesis 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 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. 6.

	F_0
R	{ hint justifying F_0 R F_1 }
	F_1
R	{ hint justifying F_1 R F_2 }
$\cdots$	$\cdots$
R	{ hint justifying F_{n-1} R F_n }
	F_n

Fig. 6. Calculation representation

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, $\ldots$ 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 $n-1$.

5 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.

5.1 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. 1(b), user has focused on S_0 and derived it separately as shown in Fig. 1(c).

The activity of focusing on a subproblem is error-prone if carried out without tool support. In Fig. 1(d), subprogram S_1 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 provided S_1 does not modify variables r and n. User has to keep this fact in mind while deriving S_1 separately. Due care must be taken during manual derivation to ensure that after any modification, the earlier assumptions still continue to hold. In CAPS, every program fragment is associated with its full specification (precondition, postcondition) and the context, and the corresponding proof obligations are automatically generated.

Since the precondition and postcondition of each program construct are made explicit, user can focus on synthesizing a subprogram in isolation. Focusing on a subprogram is achieved by applying the *StepInTactic* which displays the subprogram under consideration along with the context and hides the rest of the program. User can then transform this subprogram to a desired form and apply the *StepOutTactic* when done. In Fig. 2, the portions of the tree enclosed by rectangles correspond to the transformations performed on the subprograms.

5.2 Extracting Context of Subformulas

CAPS also provides a functionality to focus on a subformula of the formula under consideration. 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 [14], [15], [22]. Our implementation differs from the stack based implementation in [15] since we maintain the history of all the transformations.

Table 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
	$\Rightarrow$	$\Leftarrow$	
	$\Leftarrow$	$\Rightarrow$	
$\boxed{A} \implies B$	$\equiv$	$\equiv$	$\Gamma \cup \{\neg B\}$
	$\Rightarrow$	$\Leftarrow$	
	$\Leftarrow$	$\Rightarrow$	
$B \implies \boxed{A}$	$\equiv$	$\equiv$	$\Gamma \cup \{B\}$
	$\Rightarrow$	$\Rightarrow$	
	$\Leftarrow$	$\Leftarrow$	

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

Extracting the Context. 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 inference pattern [26].

$$\frac{\Gamma' \vdash f\ r\ f'}{\Gamma \vdash F[f]\, R\, F[f']} \tag{1}$$

Table 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 [2]. 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. 7 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 1.

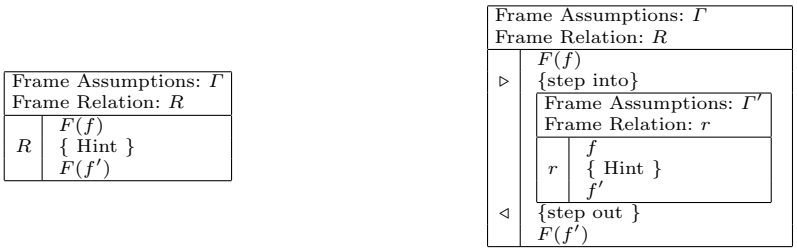


Fig. 7. Focusing on subformula.

Fig. 8 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.

5.3 Automation at Tactic Level

We now describe the various functions of CAPS that are automated with the help of ATPs and the scenarios in which these automations are helpful.

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. For example, the *Split Range Tactic* and the *Empty Range Tactic* for the universal quantifier are shown below.

Split Range Tactic	**Empty Range Tactic**
$(\forall\, i : P.i \vee Q.i : T.i)$	$(\forall\, i : R.i : T.i)$
$\equiv$ { Split Range }	$\equiv$ { Empty Range; $R.i \equiv false$}
$(\forall\, i : P.i : T.i) \wedge (\forall\, i : Q.i : T.i)$	$true$

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.). These conditions are automatically verified in CAPS using ATPs. Note that in the absence of this integration, the 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 that do not involve any metavariable are good candidates for full automation. In Section 2, we skipped the proof for preservation of the loop invariant $P_1 : 0 \leq n \leq N$. This invariant proof obligation

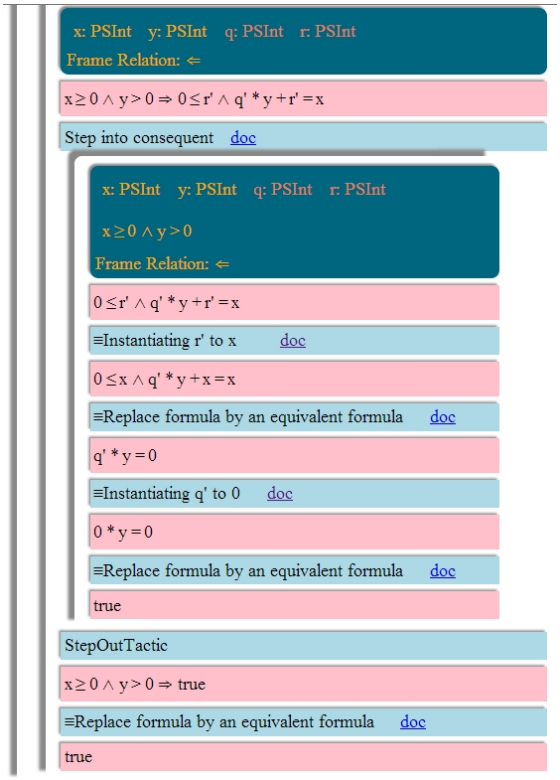
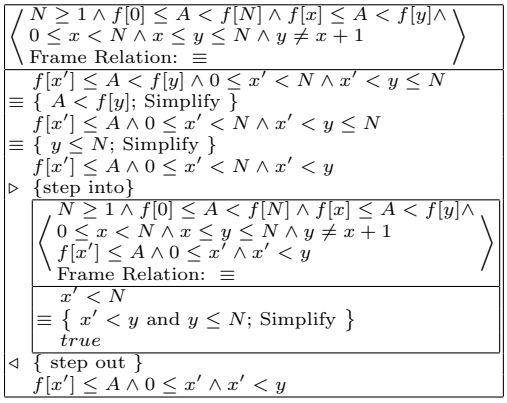


Fig. 8. Calculation of initialization assignment $(q, r := 0, x)$ to establish invariant $0 \leq r \wedge q * y + r = x$ in the derivation of *Integer Division* program

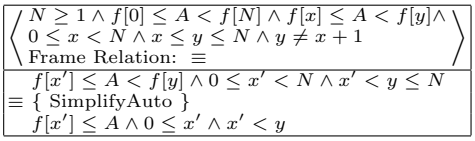
does not involve any metavariable, and hence is not of interest from the synthesis point of view. We automatically prove such proof obligations with the help of ATPs. In case the automated provers fail to discharge the proof obligation or prove it invalid, we have to revert back to the step-by-step way 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 *Verified-Transformation* 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 transformation functionality offered by the *poC* (program-oriented calculations) [21] extension of Dafny. The derivation in Fig. 8 has three instances of application of this tactic (labeled by a hint "Replace formula by an equivalent formula"). This tactic greatly reduces 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



(a)



(b)

Fig. 9. (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.

proof by contradiction, case analysis, induction, etc.) are not easily expressed in a purely calculational style [2]. Although, with some effort, these proofs can be discharged by using the functionality for focusing on subcomponents (which is based on the structured calculational approach in [2]), 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, the user directly specifies an arbitrary formula as the transformed form of a given formula and the tactic application just verifies the correctness of the transformation.

Simplification. The *Simplify* tactic simplifies the current formula by eliminating the $true/false$ subformulas. For example, it transforms the formula $\varphi \wedge true$ to φ. The *SimplifyAuto* tactic takes this idea further by recursively focusing on the subformulas in bottom-up fashion and verifying – 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. 9(a) shows an excerpt from the derivation of the binary search program whereas Fig. 9(b) shows how the same outcome can be accomplished in a single step using the *SimplifyAuto* tactic.

6 Conclusions and Future Work

To address the problem of lengthy and tedious calculational program derivations, we have proposed an approach to integrate automated theorem provers at a tactic level and implemented it in a calculational assistant (CAPS) which we have built to assist users in deriving imperative programs from formal specifications. We have adapted various techniques from the fields of program verification and theorem proving for providing features like ability to step into proof obligations, metavariable support, and ability to extract context of a subformula, which help in realizing the tactic level automation. The introduced tactics 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.

This tool will be used in the future offerings of the "Program Derivation" (CS420) class at IIT Bombay. To improve the usability, we plan to develop heuristics to rank the tactics in a given context so that at every stage in the derivation, users can be presented with a list of tactics sorted by descending likelihood of application. We also plan to develop high level program derivation tactics where the low level synthesis tasks (like synthesizing loop-free programs) are taken care of by the syntax-guided synthesis solvers [1].

Acknowledgements. The authors would like to thank the anonymous referees for their helpful comments. The work of the first author was supported by the Tata Consultancy Services (TCS) Research Fellowship.

References

1. Alur, R., Bodik, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD) (2013)
2. Back, R., Grundy, J., Von Wright, J.: Structured calculational proof. Formal Aspects of Computing 9(5-6), 469–483 (1997)
3. Back, R.J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Graduate Texts in Computer Science, Springer-Verlag, Berlin (1998)
4. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV. LNCS, vol. 4590, pp. 298–302. Springer (2007)
5. Butler, M., Långbacka, T.: Program derivation using the refinement calculator. In: Theorem Proving in Higher Order Logics: 9th International Conference, volume 1125 of LNCS. pp. 93–108. Springer Verlag (1996)

6. Carrington, D., Hayes, I., Nickson, R., Watson, G.N., Welsh, J.: A tool for developing correct programs by refinement. Tech. rep. (1996), http://espace.library.uq.edu.au/view/UQ:10768
7. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: Vcc: A practical system for verifying concurrent c. In: Theorem Proving in Higher Order Logics. Springer (2009)
8. Conchon, S., Contejean, E.: The alt-ergo automatic theorem prover, 2008 http://alt-ergo.lri.fr
9. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer (2008)
10. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM 18(8), 453–457 (1975)
11. Dijkstra, E.W., Feijen, W.H.: A Method of Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1988)
12. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, NJ, USA (1997)
13. Filliâtre, J.C., Paskevich, A.: Why3 – Where Programs Meet Provers. In: ESOP'13 22nd European Symposium on Programming. LNCS, vol. 7792. Springer, Rome, Italie (2013)
14. Grundy, J.: A window inference tool for refinement. In: 5th Refinement Workshop. pp. 230–254. Springer (1992)
15. Grundy, J.: A Method of Program Refinement. Ph.D. thesis, University of Cambridge Computer Laboratory, Cambridge, England (1993)
16. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: Hall, M.W., Padua, D.A. (eds.) Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011. pp. 62–73. ACM (2011)
17. Jacobs, B., Piessens, F.: The VeriFast program verifier. Technical Report CW-520, Dept. of Computer Science, Katholieke Universiteit Leuven (2008), http://www.cs.kuleuven.be/~bartj/verifast/verifast.pdf
18. Kaldewaij, A.: Programming: The Derivation of Algorithms. Prentice-Hall, Inc., NJ, USA (1990)
19. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Logic for Programming, Artificial Intelligence, and Reasoning. Springer (2010)
20. Leino, K.R.M., Polikarpova, N.: Verified calculations. In: Verified Software: Theories, Tools, Experiments, pp. 170–190. Springer (2014)
21. Morgan, C.: Programming from Specifications. Prentice-Hall, Inc. (1990)
22. Robinson, P.J., Staples, J.: Formalizing a hierarchical structure of practical mathematical reasoning. Journal of Logic and Computation 3(1), 47–61 (1993)
23. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. ACM SIGARCH Computer Architecture News 34(5), 404–415 (2006)
24. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: POPL 2010. pp. 313–326. New York, NY, USA (2010)
25. Weidenbach, C., Brahm, U., Hillenbrand, T., Keen, E., Theobalt, C., Topic, D.: SPASS version 2.0. In: Voronkov, A. (ed.) Automated Deduction – CADE-18. Lecture Notes in Computer Science, vol. 2392, pp. 275–279. Springer-Verlag (2002)
26. von Wright, J.: Extending window inference. In: Theorem Proving in Higher Order Logics, pp. 17–32. Springer (1998)