Building an IDE for the Calculational Derivation of Imperative Programs

Dipak L. Chaudhari Om Damani
Indian Institute of Technology Bombay, India
dipakc@cse.iitb.ac.in damani@cse.iitb.ac.in

In this paper, we describe an IDE called CAPS (Calculational Assistant for Programming from Specifications) for the interactive, calculational derivation of imperative programs. In building CAPS, our aim has been to make the IDE accessible to non-experts while retaining the overall flavor of the pen-and-paper calculational style. We discuss the overall architecture of the CAPS system, the main features of the IDE, the GUI design, and the trade-offs involved.

1 Introduction

Correct by Construction is a programming methodology, wherein programs are derived from a given formal specification of the problem to be solved, by repeatedly applying transformation rules to partially derived programs. Within this broad framework, Dijkstra and Wim Feijen [15] popularized the Calculational style for deriving sequential programs, where unknown program fragments are *calculated* from their pre- and post- conditions. By *calculation*, we mean that program constructs are introduced only when logical manipulations show them to be sufficient for discharging the correctness proof obligations.

Despite resulting in simple and elegant programs [19], the Calculational Style of Program Derivation did not become popular due to the various practical difficulties that prevented wider adoption of this methodology. Even for small programming problems, the derivations are often long and difficult to organize. As a result, the derivations, if done manually, are error-prone and cumbersome.

To address these issues, we have built an IDE called CAPS (Calculational Assistant for Programming from Specifications)[1]. CAPS has built-in refinement rules and the system generates the required correctness proof obligations. In building CAPS, our aim has been to make the IDE accessible to nonexperts while retaining the overall flavor of the pen-and-paper style derivation.

Towards this goal, in our earlier work, we described the use of theorem prover assisted tactics [11] to automate the mundane tasks during the derivations. In this paper, we discuss the overall architecture of CAPS, the main features of the IDE, the GUI design, and the design trade-offs involved. For the automation to fit into the overall calculational methodology, we have developed several features, like stepping into subcomponents, backtracking, and metavariable support. With the help of small examples, we discuss how these features address various issues with particular emphasis on usability.

Related Work.

The Implement-and-Verify program development methodology involves an implementation phase followed by a verification phase. Tools like Why3 [16], Dafny [20], VCC [13] and VeriFast [18] generate the proof obligations and try to automatically discharge these proof obligations. Although the failed

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

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 [17], Refine [22], Refinement Calculator [9] and PRT [10] 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 and the Specware[24] 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[8] 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.

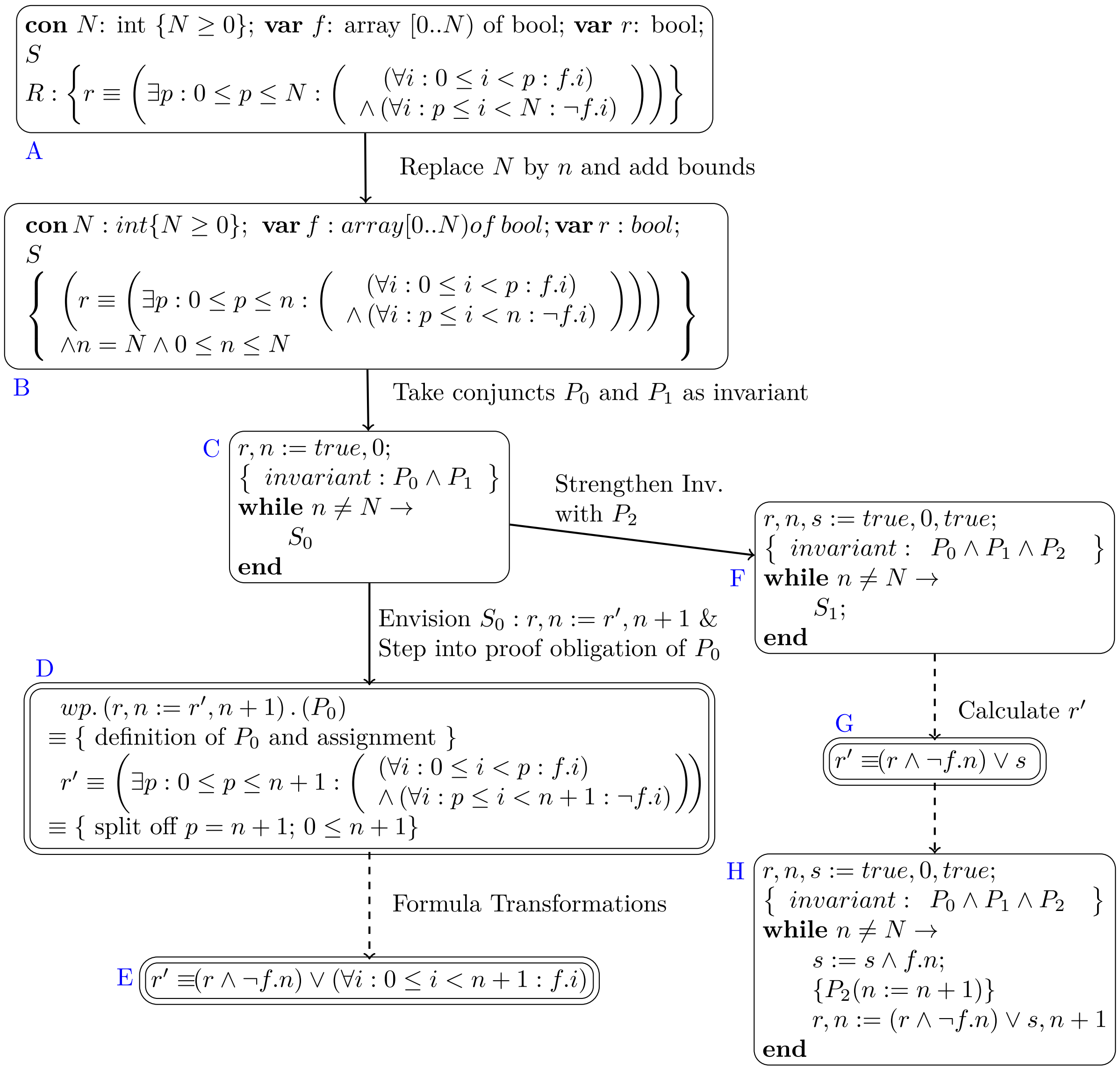
2 An Example of a Calculational Derivation

We now present a sketch of the calculational derivation for a simple program. Consider the following programming task (adapted from exercise 4.3.4 in [19]. The informal derivation of this problem also appears in [11]).

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

Fig. 1 depicts the derivation process for this program. We start the derivation by providing the formal specification (node A) of the unknown program S. We apply the *Replace Constant by a Variable* [19] heuristic. In particular, we replace constant N by a fresh variable n and add bounds on n to arrive at program B. After inspecting the postcondition of program shown in node B, we decide to apply another well known heuristic *Take Conjuncts as Invariants* to arrive at a *While* program (node C) with P_0 and P_1 as loop invariants. Here, S_0 denotes the unknown loop body. (Derivation of the initialization of the variables r and n is skipped.) To ensure loop progress, we envision an assignment $r,n := r', n+1$ for S_0 where r' is placeholder for the unknown expression (also called a metavariable). We then step into the proof obligation for preservation of invariant P_0 and try to manipulate the formula with the aim of finding a program expression for the metavariable r'. After several formula transformations we arrive at a formula E $(r' \equiv (r \wedge \neg f[n]) \vee (\forall i : 0 \leq i < n+1 : f[i]))$. At this point, we realize that we can not represent r' in terms of the program variables unless we introduce a fresh variable to maintain $(\forall i : 0 \leq i < n : f[i])$. We then backtrack to program B, introduce a fresh variable s and strengthen the invariant of the *While* program with P_2. For the derivation of program S_1, we follow the same process as that of S_0 with the strengthened invariant. On this derivation attempt, we are able to calculate r' with the help of the newly added invariant P_2. Finally we derive $s := s \wedge f.n$ to establish $P_2(n := n+1)$.[2] The final derived program in shown in node H. (Note that we can further improve the program by strengthening the guard.)

[2]$P_2(n := n+1)$ represents a formula obtained by textual substitution of the free occurrences of n with $n+1$ in P_2


A
B
C
D
E
F
G
H
Replace N by n and add bounds
Take conjuncts P_0 and P_1 as invariant
Strengthen Inv. with P_2
Envision $S_0 : r, n := r', n+1$ & Step into proof obligation of P_0
Formula Transformations
Calculate r'

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

B:
```
con $N : int\{N \geq 0\}$; var $f : array[0..N) of\ bool$; var $r : bool$;
$S$
$\left\{ \begin{array}{l} \left( r \equiv \left( \exists p : 0 \leq p \leq n : \left( \begin{array}{c} (\forall i : 0 \leq i < p : f.i) \\ \wedge (\forall i : p \leq i < n : \neg f.i) \end{array} \right) \right) \right) \\ \wedge n = N \wedge 0 \leq n \leq N \end{array} \right\}$
```

C:
```
$r, n := true, 0$;
{ $invariant : P_0 \wedge P_1$ }
while $n \neq N \rightarrow$
    $S_0$
end
```

D:
```
  $wp.(r, n := r', n+1).(P_0)$
$\equiv$ { definition of $P_0$ and assignment }
  $r' \equiv \left( \exists p : 0 \leq p \leq n+1 : \left( \begin{array}{c} (\forall i : 0 \leq i < p : f.i) \\ \wedge (\forall i : p \leq i < n+1 : \neg f.i) \end{array} \right) \right)$
$\equiv$ { split off $p = n+1$; $0 \leq n+1$ }
```

E: $r' \equiv (r \wedge \neg f.n) \vee (\forall i : 0 \leq i < n+1 : f.i)$

F:
```
$r, n, s := true, 0, true$;
{ $invariant : \ P_0 \wedge P_1 \wedge P_2$ }
while $n \neq N \rightarrow$
    $S_1$;
end
```

G: $r' \equiv (r \wedge \neg f.n) \vee s$

H:
```
$r, n, s := true, 0, true$;
{ $invariant : \ P_0 \wedge P_1 \wedge P_2$ }
while $n \neq N \rightarrow$
    $s := s \wedge f.n$;
    $\{P_2(n := n+1)\}$
    $r, n := (r \wedge \neg f.n) \vee s, n+1$
end
```

Figure 1. Sketch of the calculational derivation for a simple program. Symbols S, S_0, and S_1 are the placeholders for the unknown program fragments. The single bordered boxes represent program nodes whereas the double bordered boxes represent formula nodes.
$P_0 : (r \equiv (\exists p : 0 \leq p \leq n : ((\forall i : 0 \leq i < p : f[i]) \wedge (\forall i : p \leq i < n : \neg f[i]))))$
$P_1 : 0 \leq n \leq N$; $\quad P_2 : s \equiv (\forall i : 0 \leq i < n : f[i])$

As can be seen in this example, the calculational derivation involves program transformations as well as formula transformations. The derivation process is non-linear involving backtracking and branching.

3 CAPS

In building CAPS, our aim has been to build an easy to use IDE for the calculational derivation of imperative programs. We have tried to automate the mundane tasks while striving to keep the overall approach close to the pen-and-paper calculational style. 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).

3.1 Derivation Methodology

We use a hierarchical representation called *AnnotatedProgram* for representing a program fragment along with its specification (precondition and postcondition). The *AnnotatedProgram* representation can be thought of as an extension of the Guarded Command Language (GCL) [14] where each program construct in the GCL is augmented with its precondition and postcondition. We also introduce a new program construct *UnkProg* to represent an unsynthesized program fragment. Each subprogram in the annotated program representation has its own precondition and postcondition. As we will see in section 5, this hierarchical structure is helpful when the user wants to focus on each subprogram independently.

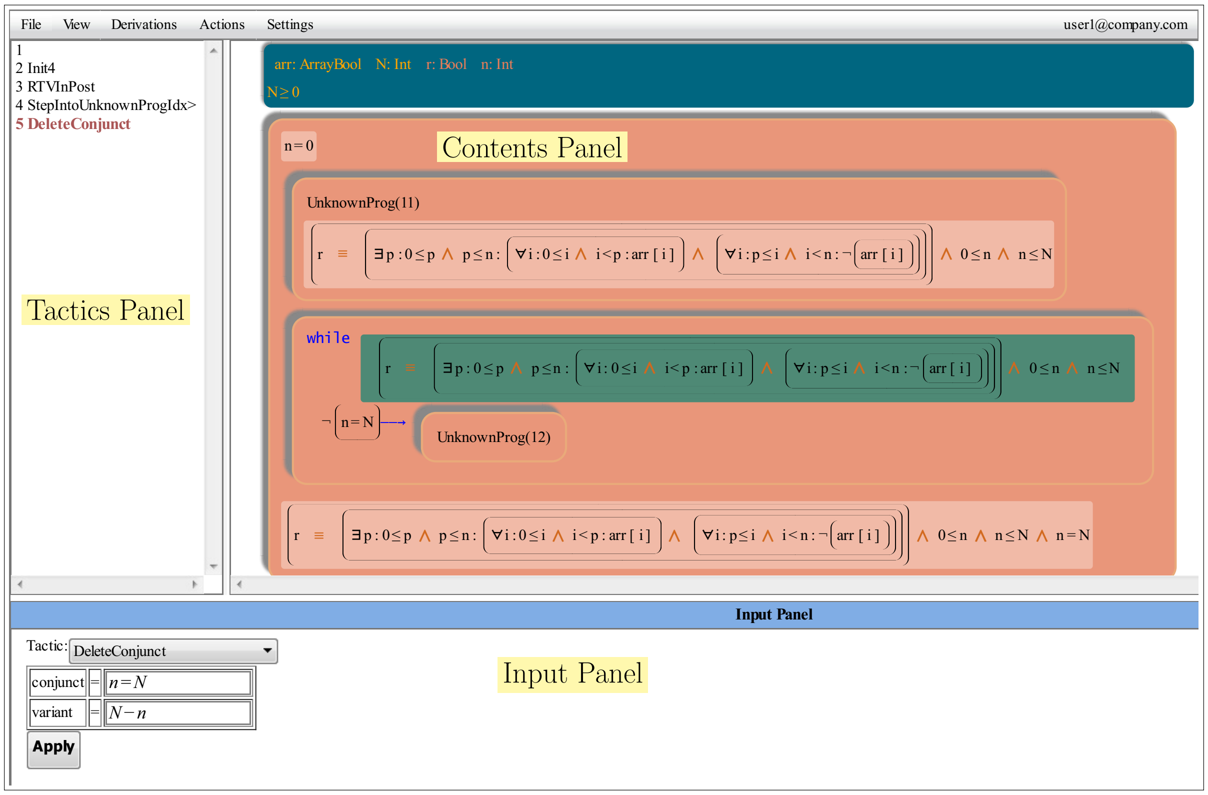
We use the formulas in sorted first-order predicate logic for expressing the precondition and the postcondition of the programs. We use the Eindhoven notation [6] for expressing the quantified formulas. In the quantified formula $(OP\,i : R : T)$, The symbol *OP* is the quantifier version of a symmetric 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.

Users start a derivation by providing the formal specification of a program and then incrementally transform it into a fully derived program by applying predefined transformation rules called *Derivation Tactics*. For example, in Fig. 1, the user starts the derivation by providing the postcondition *R* (node *A*). This program is then transformed incrementally to the final program shown in node *H*. During the derivation, a user might envision a subprogram in terms of the metavariables. The next task for the user is to find a program expression for the metavariable such that the proof obligation is discharged. This requires formula transformations to simplify the proof obligation. The derivation thus consists of the program transformations as well as the formula transformations. These derivation modes are called the program mode and the formula mode respectively. A way of transitioning between these two modes is described in section 5. The derivation process ends when all the unknown programs are derived. The complete derivation history is recorded in the form of the *Derivation Tree*.

The final outcome of the program derivation process is the fully annotated program along with the complete derivation tree. The *AnnotatedProgram* can be easily transformed to a program in a real programming language.

3.2 Graphical User Interface

Fig. 2 shows the Graphical User Interface of the CAPS system. It has three panels. The central panel, also called the contents panel, shows a partially derived program (or a formula) at the current stage of the derivation. For example, the schematic node *C* in Fig. 1 corresponds to the program in the contents panel in Fig. 2. The contents in this panel can be shown at different levels of details, as discussed in section 6. 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. For example, the tactics applied from node *A* to node *C* in Fig. 1 are listed in the tactics panel in Fig. 2. Users can navigate back to an earlier point in the derivation by clicking on the corresponding node in the left panel. The bottom panel is the input panel. This panel is used for selecting a tactic to be applied next and for providing the corresponding tactic parameters.


File
View
Derivations
Actions
Settings
user1@company.com
1
2 Init4
3 RTVInPost
4 StepIntoUnknownProgIdx>
5 DeleteConjunct
Tactics Panel
arr: ArrayBool N: Int r: Bool n: Int
N≥0
n=0
Contents Panel
UnknownProg(11)
r ≡ ∃p:0≤p ∧ p≤n: (∀i:0≤i ∧ i<p:arr[i]) ∧ (∀i:p≤i ∧ i<n:¬(arr[i])) ∧ 0≤n ∧ n≤N
while
r ≡ ∃p:0≤p ∧ p≤n: (∀i:0≤i ∧ i<p:arr[i]) ∧ (∀i:p≤i ∧ i<n:¬(arr[i])) ∧ 0≤n ∧ n≤N
¬(n=N) ⟶
UnknownProg(12)
r ≡ ∃p:0≤p ∧ p≤n: (∀i:0≤i ∧ i<p:arr[i]) ∧ (∀i:p≤i ∧ i<n:¬(arr[i])) ∧ 0≤n ∧ n≤N ∧ n=N
Input Panel
Input Panel
Apply

Tactic: DeleteConjunct

conjunct	=	$n = N$
variant	=	$N - n$

Figure 2. CAPS GUI

3.3 System Architecture

The architecture of the CAPS system is shown in Fig. 3. There are 3 main components of the system:

- Core Library. The Core library contains the data structures for *AnnotatedPrograms*, *Formula*, *DerivationTree*, *DerivationTactic* and *Frame*. It also contains a repository of the program and the formula manipulation tactics. The Core library is integrated with various automated theorem provers (Alt-Ergo, CVC3, SPASS, Z3) via the common interface provided by the *Why3* framework [16]. The Derivation Tree management utilities are also implemented in this library. The library is implemented in *Scala* and uses the *Kiama* library [23] for rewriting.

- Application Server. The server component is implemented using the *Scala play* web framework [2]. The server stores the current state of the derivation. The application also implements a tactic parser which parses the tactic request.

- Web Client. The *CAPS* application is implemented as a single-page web application based on the *Backbone.js* framework [1]. 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 GUI part is implemented in the *Typescript* language [3] (which complies to Javascript). The GUI module has different views to display the current state of the derivation.

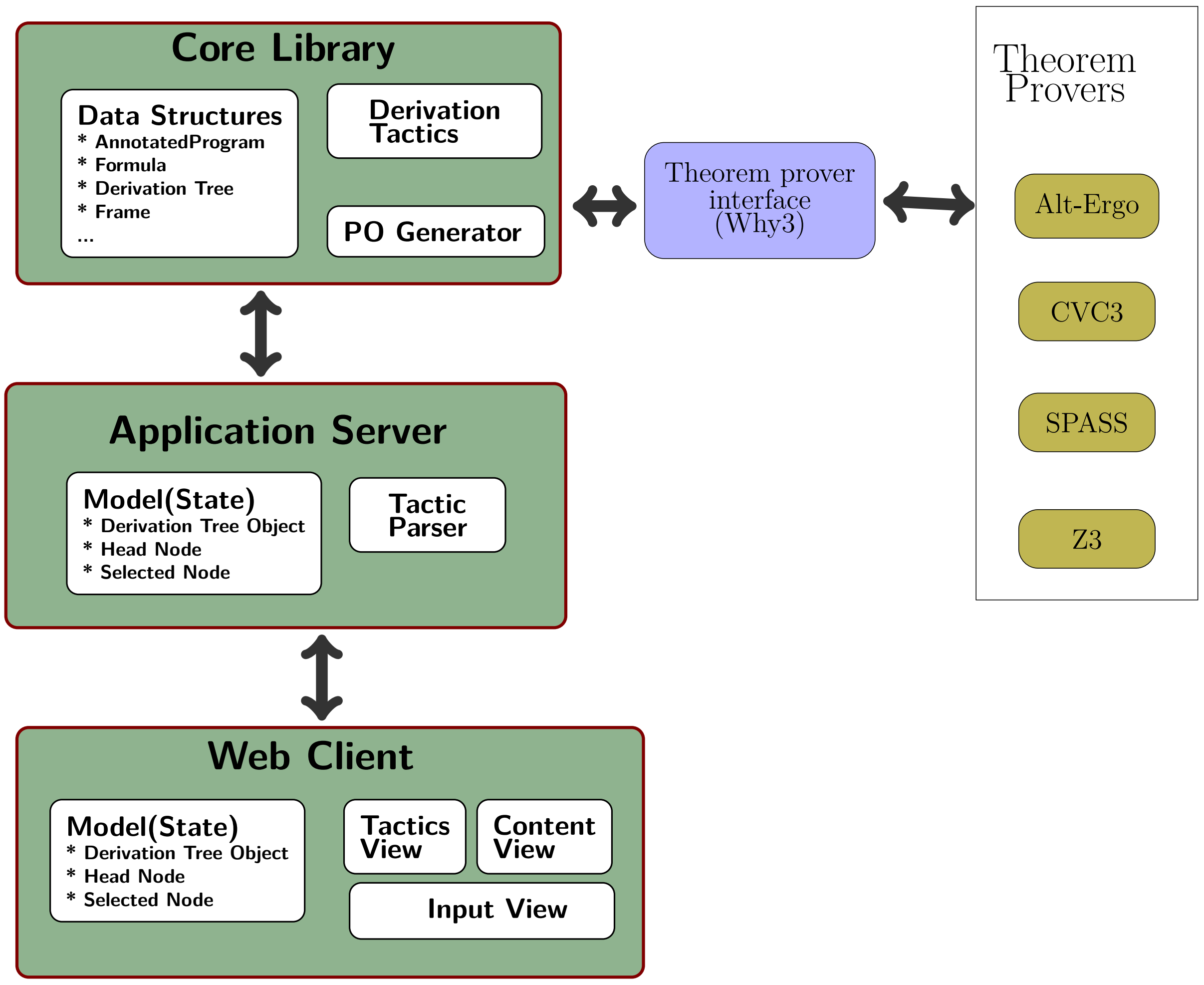
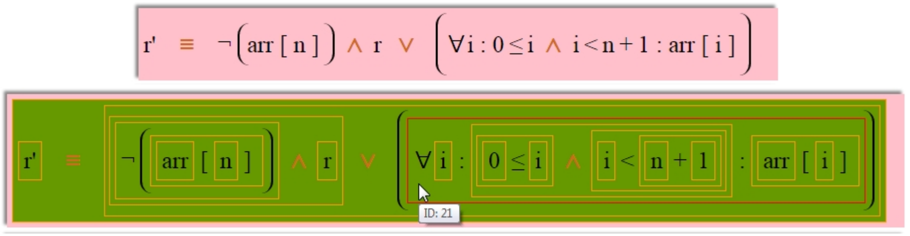


Figure 3. CAPS Architecture

4 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 [20] and Why3 [16] use textual representations, the structural representation is more suitable for a tactic based framework like CAPS. An Annotated 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 [7].

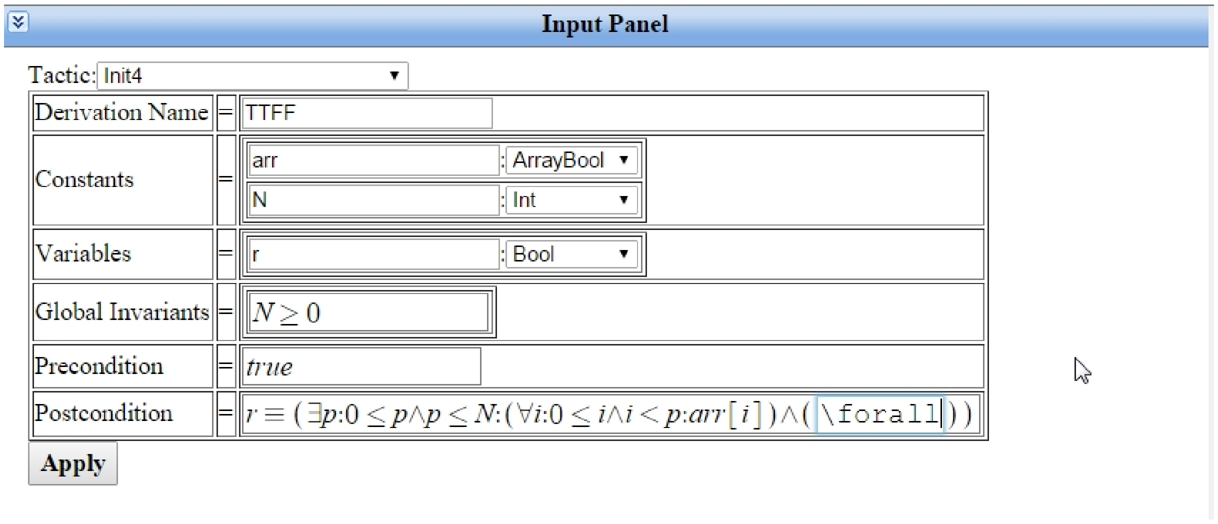
Direct editing of the Annotated Program may destroy the structure and is disallowed in CAPS; the only way to generate a program is through a tactic application. This discipline allows us to capture all the design decisions taken during the derivation. However, to allow some informality, we do have tactics 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.



r' ≡ ¬(arr [n]) ∧ r ∨ (∀i : 0≤i ∧ i<n+1 : arr [i])

r' ≡ ¬(arr [n]) ∧ r ∨ (∀ i : 0 ≤ i ∧ i < n + 1 : arr [i])

Figure 4. Structured representation of a formula in *normal mode* and *selection mode*. Users can select a subformula by simply clicking on it.



Input Panel

Tactic: Init4

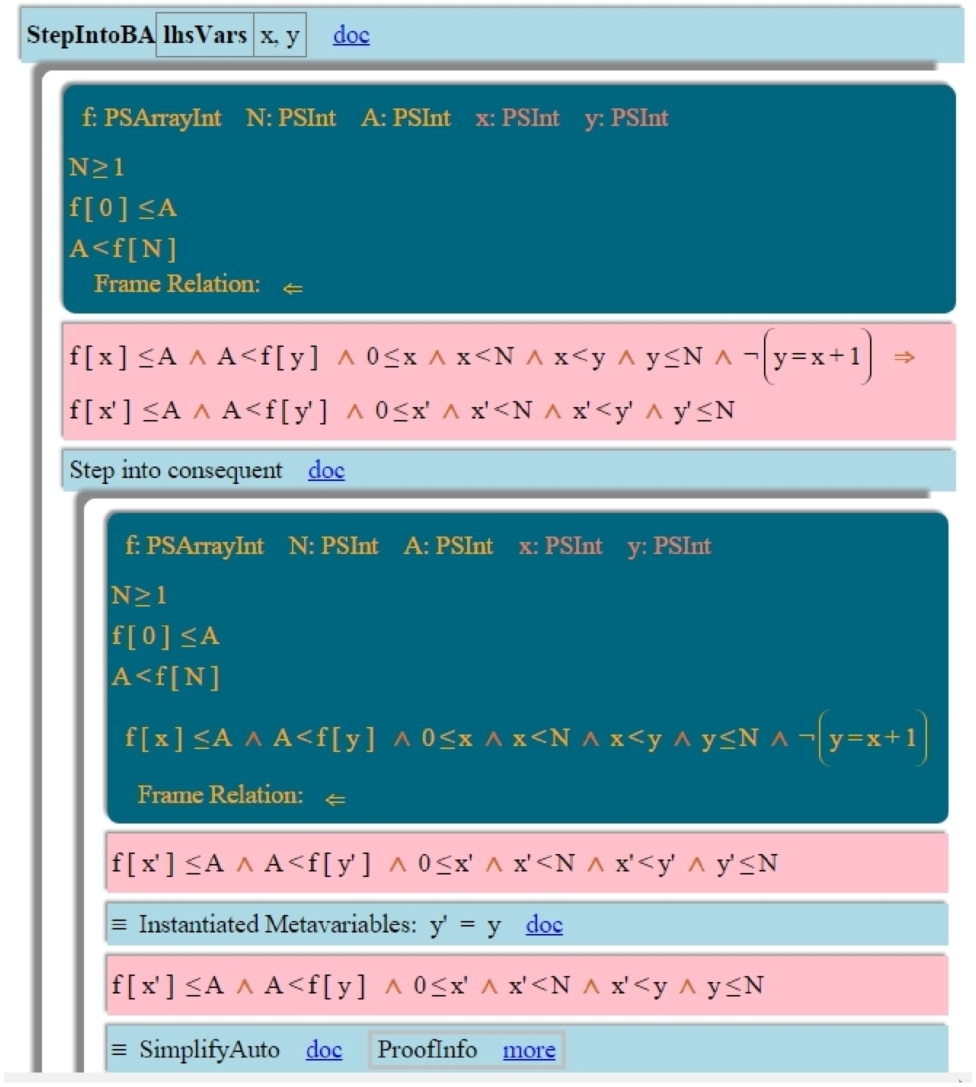
Derivation Name	=	TTFF
Constants	=	arr : ArrayBool; N : Int
Variables	=	r : Bool
Global Invariants	=	$N \geq 0$
Precondition	=	*true*
Postcondition	=	$r \equiv (\exists p{:}0 \leq p \wedge p \leq N{:}(\forall i{:}0 \leq i \wedge i < p{:}arr[i]) \wedge($ \forall$))$

Figure 5. Input Panel: On selection of a tactic to be applied, the corresponding input form is dynamically generated.

The contents panel in Fig. 2 shows the structured representation of an annotated program. Fig. 4 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 ≡) 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. 5 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.



StepIntoBA lhsVars x, y doc

f: PSArrayInt N: PSInt A: PSInt x: PSInt y: PSInt

$N \geq 1$

$f[0] \leq A$

$A < f[N]$

Frame Relation: $\Leftarrow$

$f[x] \leq A \wedge A < f[y] \wedge 0 \leq x \wedge x < N \wedge x < y \wedge y \leq N \wedge \neg(y = x+1) \Rightarrow$

$f[x'] \leq A \wedge A < f[y'] \wedge 0 \leq x' \wedge x' < N \wedge x' < y' \wedge y' \leq N$

Step into consequent doc

f: PSArrayInt N: PSInt A: PSInt x: PSInt y: PSInt

$N \geq 1$

$f[0] \leq A$

$A < f[N]$

$f[x] \leq A \wedge A < f[y] \wedge 0 \leq x \wedge x < N \wedge x < y \wedge y \leq N \wedge \neg(y = x+1)$

Frame Relation: $\Leftarrow$

$f[x'] \leq A \wedge A < f[y'] \wedge 0 \leq x' \wedge x' < N \wedge x' < y' \wedge y' \leq N$

$\equiv$ Instantiated Metavariables: $y' = y$ doc

$f[x'] \leq A \wedge A < f[y] \wedge 0 \leq x' \wedge x' < N \wedge x' < y \wedge y \leq N$

$\equiv$ SimplifyAuto doc ProofInfo more

Figure 6. Formula transformations from the derivation of the Binary Search program.

5 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 the user 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 chose 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 [5]. The implementation details and the theoretical basis of the contextual extraction is given in [11].

Fig. 6 shows a snapshot of the formula transformations involved in the derivation of the binary search program. The derivation is displayed in a nested fashion. Whenever the 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, user can step out from the formula mode to create a program where the metavariables are replaced with the corresponding derived expressions.

Unlike the hierarchical program structure, the hierarchical formula structure is not usually 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. 4 shows a formula in the normal mode and in the selection mode.

6 Selective Display of Information

In the *AnnotatedProgram* 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 information that is not relevant in any given context, such as the annotations that can be easily inferred from the other annotations. CAPS provides a *Minimal Annotations* mode which displays only the following annotations.

- Precondition and postcondition of the outermost program
- Loop invariants
- The intermediate-assertion of the *Composition* construct

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 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. 6). The annotated programs can also be collapsed by double clicking on them.

7 Maintaining Derivation History

Invariant and assertion annotations 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 E in the derivation in Fig. 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 P_2. 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

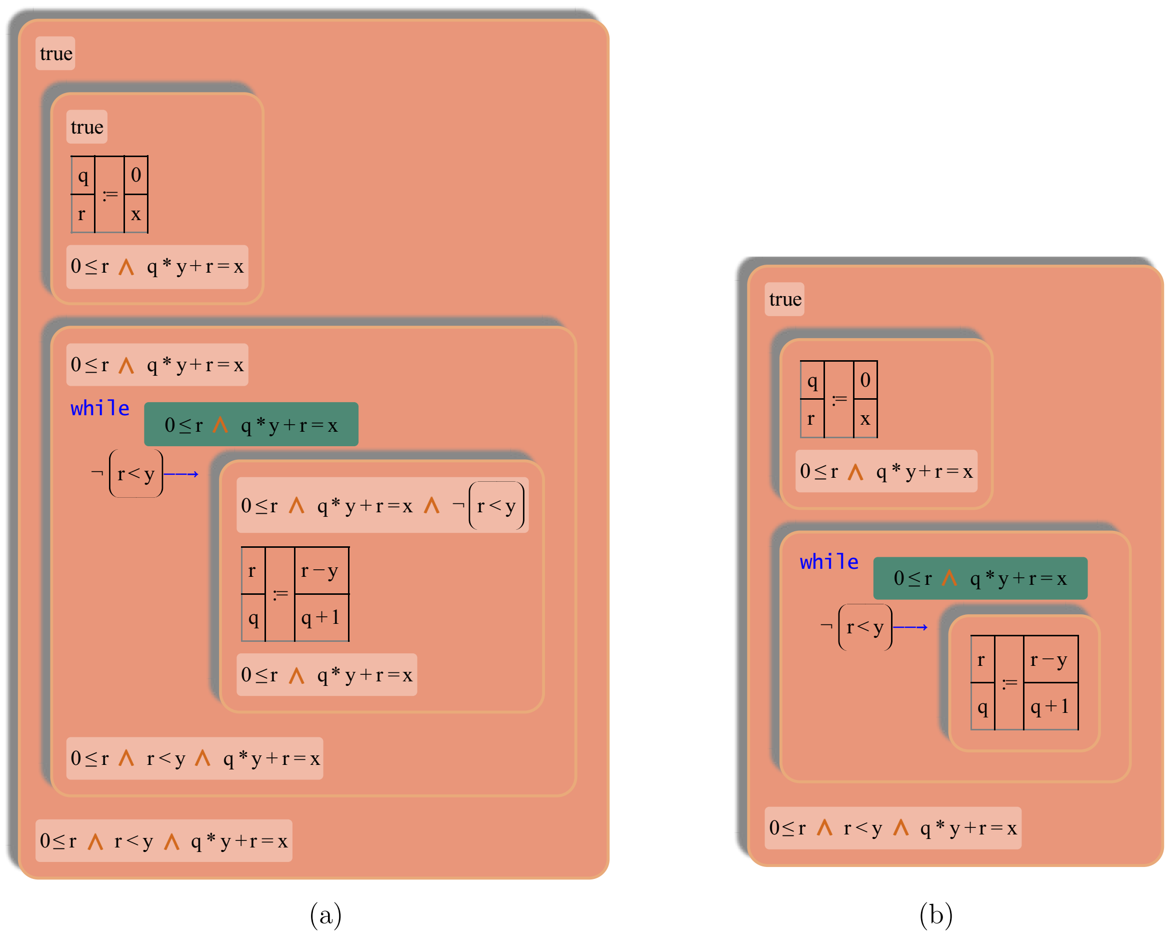


Figure 7. Final *AnnotatedProgram* for the Integer Division problem: a) Full annotations mode, b) Minimal annotation mode.

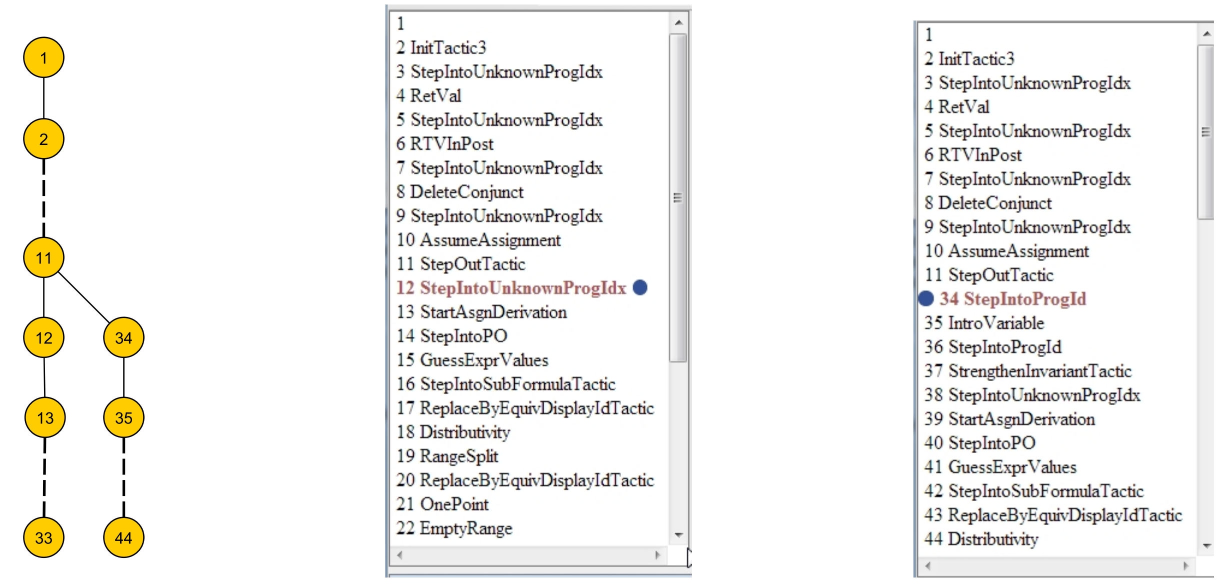
derivation history in the form a derivation tree. Maintaining history also facilitates backtracking and branching if the user wants to try out an alternative derivation strategy.

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 explore multiple solutions for the given programming task.

Navigating the Derivation tree

The conventional tree interface is not suitable to showing the derivation tree. At any point during a derivation, we are interested in only the active path of the derivation tree. This active path is shown in



```
1
2 InitTactic3
3 StepIntoUnknownProgIdx
4 RetVal
5 StepIntoUnknownProgIdx
6 RTVInPost
7 StepIntoUnknownProgIdx
8 DeleteConjunct
9 StepIntoUnknownProgIdx
10 AssumeAssignment
11 StepOutTactic
12 StepIntoUnknownProgIdx ●
13 StartAsgnDerivation
14 StepIntoPO
15 GuessExprValues
16 StepIntoSubFormulaTactic
17 ReplaceByEquivDisplayIdTactic
18 Distributivity
19 RangeSplit
20 ReplaceByEquivDisplayIdTactic
21 OnePoint
22 EmptyRange
```

```
1
2 InitTactic3
3 StepIntoUnknownProgIdx
4 RetVal
5 StepIntoUnknownProgIdx
6 RTVInPost
7 StepIntoUnknownProgIdx
8 DeleteConjunct
9 StepIntoUnknownProgIdx
10 AssumeAssignment
11 StepOutTactic
● 34 StepIntoProgId
35 IntroVariable
36 StepIntoProgId
37 StrengthenInvariantTactic
38 StepIntoUnknownProgIdx
39 StartAsgnDerivation
40 StepIntoPO
41 GuessExprValues
42 StepIntoSubFormulaTactic
43 ReplaceByEquivDisplayIdTactic
44 Distributivity
```

Figure 8. 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 37) in the derivation tree. Users can click on this sibling marker to switch to the branch containing *node 37.*. The resulting path is shown in Fig. (c).

the left panel in the GUI. To make it easy to navigate to other branches, we show siblings of the nodes in the path. Users can navigate across the branches by clicking the sibling markers as shown in Fig. 8. If there are multiple branches under the selected sibling, then the rightmost branch is selected.

8 Conclusions and Future Work

In this work, we have described the design of an IDE for the Calculational Derivation of Imperative Programs. Our design focus has been on making the IDE accessible to nonexperts while retaining the overall flavor of the pen-and-paper style derivation. We have used the CAPS system in an elective course on *Program Derivation* taken by 2nd year students. The preliminary student response to the tool has been very positive[12]. However, a thorough evaluation needs to be done on more challenging problems.

Based on the learnings from the first offering of the tool, we plan to enhance the tool in a number of ways.

Richer Language Constructs.

We plan to target programs with richer constructs involving recursion, algebraic data types, and polymorphic types.

Executing programs.

We currently do not have a functionality to execute the derived programs in CAPS. We plan to explore the possibility of executing not only the final program, but also the intermediate partially derived programs.

Being able to simulate programs at the intermediate stages of behavioral abstraction has already been identified[21] as one of the barriers in the adoption of the stepwise refinement based methods.

Integrating Synthesis Solvers.

We plan employ the synthesis solvers [4] during the interactive derivation when the specification of the subprogram under consideration falls in a theory for which a synthesis solver is available. We will, however, restrict the use of these solvers to the synthesis of loop-free programs.

Acknowledgements.

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

References

[1] *Backbone.js: a JavaScript library with a RESTful JSON interface*. Available at `http://backbonejs.org/`.

[2] *Play: an open source MVC web application framework*. Available at `http://www.playframework.com/`.

[3] *TypeScript: a language for application-scale JavaScript development, http://www.typescriptlang.org/*. Available at `http://www.typescriptlang.org/`.

[4] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak & Abhishek Udupa (2013): *Syntax-Guided Synthesis*. In: *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, doi:10.1109/FMCAD.2013.6679385.

[5] Ralph Back, Jim Grundy & Joakim Von Wright (1997): *Structured Calculational Proof*. *Formal Aspects of Computing* 9(5-6), pp. 469–483, doi:10.1007/BF01211456.

[6] Roland Backhouse & Diethard Michaelis (2006): *Exercises in quantifier manipulation*. In: *Mathematics of program construction*, Springer, pp. 69–81, doi:10.1007/11783596_7.

[7] Yves Bertot, Thomas Kleymann-Schreiber & Dilip Sequeira (1997): *Implementing Proof by Pointing without a Structure Editor*. Technical Report ECS-LFCS-97-368, University of Edinburgh. Available at `http://www.inria.fr/RRRT/RR-3286.html`.

[8] Richard Bornat & Bernard Sufrin (1997): *Jape: A calculator for animating proof-on-paper*. In: *Automated DeductionCADE-14*, Springer, pp. 412–415, doi:10.1007/3-540-63104-6_41.

[9] Michael Butler & Thomas Långbacka (1996): *Program Derivation Using the Refinement Calculator*. In: *Theorem Proving in Higher Order Logics: 9th International Conference, volume 1125 of LNCS*, Springer Verlag, pp. 93–108, doi:10.1007/BFb0105399.

[10] David Carrington, Ian Hayes, Ray Nickson, G. N. Watson & Jim Welsh (1996): *A Tool for Developing Correct Programs by Refinement*. Technical Report. Available at `http://espace.library.uq.edu.au/view/UQ:10768`.

[11] Dipak L. Chaudhari & Om Damani (2014): *Automated Theorem Prover Assisted Program Calculations*. In Elvira Albert & Emil Sekerinski, editors: *Integrated Formal Methods*, Lecture Notes in Computer Science, Springer International Publishing, pp. 205–220, doi:10.1007/978-3-319-10181-1_13.

[12] Dipak L. Chaudhari & Om Damani (2015): *Introducing Formal Methods via Program Derivation*. In: *Innovation and Technology in Computer Science Education, ITiCSE 15*.

[13] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte & Stephan Tobies (2009): *VCC: A Practical System for Verifying Concurrent C*. In: *Theorem Proving in Higher Order Logics*, Springer, doi:10.1007/978-3-642-03359-9_2.

[14] Edsger W. Dijkstra (1975): *Guarded Commands, Nondeterminacy and Formal Derivation of Programs*. *Commun. ACM* 18(8), pp. 453–457, doi:10.1145/360933.360975.

[15] Edsger W. Dijkstra & W. H. Feijen (1988): *A Method of Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[16] Jean-Christophe Filliâtre & Andrei Paskevich (2013): *Why3 – Where Programs Meet Provers*. In: *ESOP'13 22nd European Symposium on Programming*, *LNCS* 7792, Springer, Rome, Italie, doi:10.1007/978-3-642-37036-6_8.

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

[18] Bart Jacobs & Frank Piessens (2008): *The VeriFast Program Verifier*. Technical Report CW-520, Dept. of Computer Science, Katholieke Universiteit Leuven. Available at `http://www.cs.kuleuven.be/~bartj/verifast/verifast.pdf`.

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

[20] K. Rustan M. Leino (2010): *Dafny: An Automatic Program Verifier for Functional Correctness*. In: *Logic for Programming, Artificial Intelligence, and Reasoning*, Springer, doi:10.1007/978-3-642-17511-4_20.

[21] K. Rustan M. Leino (2011): *Tools and Behavioral Abstraction: A Direction for Software Engineering*. In Sebastian Nanz, editor: *The Future of Software Engineering*, Springer Berlin Heidelberg, pp. 115–124, doi:10.1007/978-3-642-15187-3_7.

[22] Marcel Oliveira, Manuela Xavier & Ana Cavalcanti (2004): *Refine and Gabriel: support for refinement and tactics*. In: *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, IEEE, pp. 310–319, doi:10.1109/SEFM.2004.1347535.

[23] Anthony M. Sloane (2011): *Lightweight Language Processing in Kiama*. In JooM. Fernandes, Ralf Lmmel, Joost Visser & Joo Saraiva, editors: *Generative and Transformational Techniques in Software Engineering III*, *Lecture Notes in Computer Science* 6491, Springer Berlin Heidelberg, pp. 408–425, doi:10.1007/978-3-642-18023-1_12.

[24] Douglas R. Smith (2008): *Generating Programs Plus Proofs by Refinement*. In Bertrand Meyer & Jim Woodcock, editors: *Verified Software: Theories, Tools, Experiments*, *Lecture Notes in Computer Science* 4171, Springer Berlin Heidelberg, pp. 182–188, doi:10.1007/978-3-540-69149-5_20.