The RISC ProgramExplorer: Reasoning about Programs as State Relations (Extended Abstract)

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.jku.at

1 Introduction

We report on the formal framework underlying the *RISC ProgramExplorer* [12], a program reasoning environment which is under development at the Research Institute for Symbolic Computation (RISC) and which integrates the previously developed *RISC ProofNavigator* [14] as an interactive proving assistant. The current release of the software is a first demonstrator that incorporates the overall technological framework (including an elaborated graphical user interface) and language processors (for a simple subset of Java as a programming language and a formal specification language). Work is going on to provide this skeleton with the envisioned program reasoning capabilities. The goal of this presentation is to outline the formal basis underlying these capabilities and to explain the rationale for its particular design.

In the "Hoare Calculus" [5], a program command `x=x*x` is specified by a triple $\{x = a\}$`x=x*x`$\{x = a^2\}$ with two logical formulas $x = a$ and $x = a^2$ evaluated on the pre-state and on the post-state of the command. Here a "fresh" logical constant a is "pulled out of the hat" to let the post-condition refer to the value of the program variable `x` in the pre-state. In "Dynamic Logic" [3], the same command is specified by the formula $\forall a : \mathtt{x} = a \Rightarrow [\mathtt{x=x*x}]\mathtt{x} = a^2$ where a becomes a bound logical variable and the two formulas $\mathtt{x} = a$ and $\mathtt{x} = a^2$ separated by the modality $[\mathtt{x=x*x}]$ have to be interpreted over two different states, the pre-state of `x=x*x` and its post-state. Both formalisms suffer from the fact that they squeeze the formulation of a binary state *relation* (the relation between the pre- and the post-state of a command) into two unary state *conditions* and thus need logical constants/variables to "glue" them together. In both formalisms a layer of reasoning with unfamiliar rules (on Hoare triples respectively formulas with modalities) is required before ultimately the well-known layer of classical predicate logic is reached.

We have chosen another possibility described in [13], the *relational* view of program commands. This allows us to write a specification of form $\mathtt{x=x*x} : \mathtt{x}' = \mathtt{x}^2$ which associates to a program command `x=x*x` a single formula $\mathtt{x}' = \mathtt{x}^2$ in classical logic (the actual syntax described later is slightly different). This formula expresses a state relation with the unprimed variable `x` denoting the value of the corresponding program variable in the pre-state and the primed version $\mathtt{x}'$ its value in the post-state. A judgment $C : S$ can now be read as "program command C implements the specification S"; this also gives immediately rise to a notion of *refinement*, since $C : S'$ and $S' \Rightarrow S$ implies $C : S$. For verification, this calculus can be exploited in the following way: given a command C specified by state relation S, we first derive a judgment $C : S'$, i.e. we *translate* the command C into a logical formula S', and then prove $S' \Rightarrow S$ using the rules of classical logic.

However, to translate a program command to a state relation alone is not sufficient, because it only describes which state transitions are allowed but does not require that any state transition actually takes place. Thus for example the judgment `while(true){}` $: S$ is trivially satisfied for any specification S, since the non-terminating loop implements the empty state relation. Our solution to this problem is straight-forward: we complement the specification of a command by a judgment $C \downarrow T$ where T is a logical formula that describes a state condition, namely a condition for the pre-state of C under which the command must terminate and produce a post-state. Here the refinement works in the other direction: $C \downarrow T'$ and $T \Rightarrow T'$ implies $C \downarrow T$. Both kinds of judgments $C : S$ and $C \downarrow T$ allow to specify a program command's safety (partial correctness) and liveness (total correctness).

Actually, a specification `x=x*x` : $\mathtt{x}' = \mathtt{x}^2$ is too weak since it does not state that every variable `y` different from `x` has the same value in the pre- and in the post-state (assuming that program variables are not aliased); also it does not allow to specify the effect of commands like `continue`, `break`, `return` E or `throw` E which change the state of a program, not by the modification of a user-visible variable, but by altering the flow of control. Our calculus is therefore based on a more general notion of state which also includes the current *execution mode* (executing normally, continuing a loop iteration, breaking from a loop, returning from a method, throwing an exception). Furthermore, it considers the *frame* of a command execution, i.e. the set of variables that may be modified. Judgments are thus actually of the form $C : [S]^V_m$ where V is the set of variables that may be modified by C and m constrains the execution mode of the post-state of C. We can thus write a specification `x=x*x` : $[\mathtt{x}' = \mathtt{x}^2]^{\{x\}}_{m_e}$ where m_e encodes the mode "normally executing"; another specification is `return x` : $[\text{value}(\text{next}) = x]^{\{\}}_{m_r}$ where 'next' is a logical constant representing the post-state and 'value' is a logical function that extracts the return value from the state.

The derivation of judgments is compositional such that e.g. the judgment `x=x+1;x=x*x` : $[S]^{\{x\}}_{m_e}$ with $S \equiv \exists x_0 : x_0 = x + 1 \wedge x' = x_0^2$ can be derived from the individual judgments for `x=x+1` and `x=x*x`. Further on the formula S can be logically simplified to $x' = (x + 1)^2$ which concisely describes the *semantic essence* of command `x=x+1;x=x*x`. A core goal of the RISC ProgramExplorer is to make this computation of a program's semantic essence transparent to the programmer as a means to better understand the meaning of a program: program code is translated to a logical formula in a format that makes the relationship to the code explicit; after suitable simplification, the formula displays the code's semantic essence. Thus e.g. the RISC ProgramExplorer will be able to illustrate the relationship between the states at two control points in a declarative way.

Still there is a strong relation to the classical calculus, e.g. from a pre/post-condition pair $x = a$ and $x = a^2$, a state relation $\forall a : x = a \Rightarrow x' = a^2$ can be derived (which can then be further simplified to $x' = x^2$); a state relation $x' = x^2$ for command C can be immediately translated to a weakest precondition $\text{wp}(C,Q) \equiv \forall b : b = x^2 \Rightarrow Q[b/x]$ (which can be simplified to $Q[x^2/x]$) or to a strongest postcondition $\text{sp}(C,P) \equiv \exists a : P[a/x] \wedge x = a^2$. The RISC ProgramExplorer will provide these translations and make use of them, e.g. to describe the information known about the state at a particular control point or to derive the condition required to reach a subsequent control point.

In this extended abstract, we present only a simplified version of the calculus that does not include the execution modes sketched above and does correspondingly not handle statements that interrupt the control flow; also we do not discuss the treatment of global program variables, program methods, and recursion. See [13] for a treatment of these aspects. The organization of the presentation is as follows: in Section 2 we discuss the core calculus for deriving judgments $C : [F]^V$; in Section 3 we introduce auxiliary judgments for deriving pre- and post-conditions; in Section 4 we describe the judgment $C \downarrow T$. Section 5 discusses related work and Section 6 concludes.

2 The Core Calculus

Let *State* be the set of program states (to be defined below) and let $\mathit{StateRelation} := \mathbb{P}(\mathit{State} \times \mathit{State})$ be the set of binary relations on states. We interpret a state relation $R \in \mathit{StateRelation}$ as a set of possible state transitions: a pair of states $\langle s, s' \rangle \in R$ (i.e. two states s, s' such that the relation $R(s,s')$ holds) is considered as a transition from pre-state s to post-state s' that is possible according to R.

We are now going to introduce a domain of (essentially classical) logical formulas such that every formula F is translated to a state relation $[\![F]\!] \in \mathit{StateRelation}$, i.e. as the set of transitions that are allowed by F. The formula domain is equipped with a (essentially classical) logical calculus such that, if the formula $F_1 \Rightarrow F_2$ can be proved, then $[\![F_1]\!] \subseteq [\![F_2]\!]$ holds, i.e. F_1 only allows those transitions that are also allowed by F_2. Likewise, we are going to introduce a domain of program commands and translate every command C to a state relation $[\![C]\!] \in \mathit{StateRelation}$, i.e. as the set of transitions that can be performed by the command (the relation $[\![C]\!]$ need not be a function, since C may be non-deterministic, or at least non-deterministically specified). The core of the calculus is a judgment $C : F$ that translates every command C to some formula F such that the soundness condition $[\![C]\!] \subseteq [\![F]\!]$ holds, i.e. every transition that can be performed by C is also allowed by F.

The domain of commands is defined by the grammar

$$C := I\texttt{=}E \mid \texttt{\{var}\ I\texttt{;}C\texttt{\}} \mid C_1\texttt{;}C_2 \mid \texttt{if (}E\texttt{)}\ C_1\ \texttt{else}\ C_2 \mid \texttt{while (}E\texttt{)}\ C^{F,T}$$

where E is an unspecified domain of program expressions denoting values. The programming language thus supports variable assignments, local variable declarations, command sequences, conditional commands, and loops (annotated with invariance formulas and termination terms). An invariance formula F describes not a condition on a single state but a relation between two states, the pre-state of the loop and the state before/after every loop iteration. On the other hand, the value of the termination term T is determined by a single state; this value must become smaller by every iteration according according to some well-founded ordering. The phrases F and T do not influence the execution of the program but aid reasoning about the execution.

For the semantic interpretation of this language, let $\mathit{State} := \mathit{Identifier} \to \mathit{Value}$ be the set of program states where $\mathit{Identifier} = \{I_1, \ldots, I_n\}$ is a finite set of identifiers and Value is a set of values; every state thus is just a mapping of program variables to values. For every command C, we are now going to define the semantics $[\![C]\!]$ by writing $[\![C]\!](s,s') :\Leftrightarrow \ldots$ to denote $[\![C]\!] := \{\langle s,s'\rangle : \ldots\}$.

$$\begin{array}{l}
[\![I\texttt{=}E]\!](s,s') :\Leftrightarrow \\
\quad \forall J \in \mathit{Identifier} : \text{ IF } J = I \text{ THEN } s'(I) = [\![E]\!](s) \text{ ELSE } s'(I) = s(I) \\
[\![\texttt{\{var}\ I\texttt{;}C\texttt{\}}]\!](s,s') :\Leftrightarrow \\
\quad \exists t,t' \in \mathit{State} : [\![C]\!](t,t') \wedge \forall J \in \mathit{Variable} : J \neq I \Rightarrow s(J) = t(J) \wedge t'(J) = s'(J) \\
[\![C_1\texttt{;}C_2]\!](s,s') :\Leftrightarrow \\
\quad \exists t \in \mathit{State} : [\![C_1]\!](s,t) \wedge [\![C_2]\!](t,s') \\
[\![\texttt{if (}E\texttt{)}\ C_1\ \texttt{else}\ C_2]\!](s,s') :\Leftrightarrow \\
\quad \text{IF } [\![E]\!](s) = \text{TRUE THEN } [\![C_1]\!](s,s') \text{ ELSE } [\![C_2]\!](s,s') \\
[\![\texttt{while (}E\texttt{)}\ C^{F,T}]\!](s,s') :\Leftrightarrow \\
\quad \exists n \in \mathbb{N}, t \in \mathit{State}^* : \\
\qquad t(0) = s \wedge t(n) = s' \wedge [\![E]\!](s') = \text{FALSE} \wedge \\
\qquad \forall i \in \mathbb{N} : i < n-1 \Rightarrow [\![E]\!](s) = \text{TRUE} \ \wedge\ [\![C]\!](t(i),t(i+1))
\end{array}$$

Most cases are self-evident; the semantics of a while loop is specified by the sequence t of states arising from the iteration of the loop. With the help of some auxiliary functions, the last three cases can be also written more concisely. Let $R_1 \circ R_2 := \{\langle s,s'\rangle : \exists t \in \mathit{State} : \langle s,t\rangle \in R_1 \wedge \langle t,s'\rangle \in R_2\}$ denote the composition of state relations R_1 and R_2. Let $R^* := \bigcup_{i \in \mathbb{N}} R^i$ with $R^0 := \{\langle s,s\rangle : s \in \mathit{State}\}$ and $R^{i+1} := R^i \cup R^i \circ R$ denote the reflexive transitive closure of state relation R. Let $S \rhd R := \{\langle s,s'\rangle \in R : s \in S\}$ denote the restriction of the domain of state relation R to state set (condition) S and let $R \lhd S := \{\langle s,s'\rangle \in R : s' \in R\}$ denote the restriction of the range of R to S. Let $f^{\mathbb{T}} := \{s \in \mathrm{dom}(f) : [\![f]\!](s) = \text{TRUE}\}$ denote that subset of the domain of function f for which the value is TRUE, and let $f^{\mathbb{F}} := \mathrm{dom}(f) \backslash f^{\mathbb{T}}$ denote the complement of that set. Then we have

$$\begin{array}{l}
[\![C_1\,;C_2]\!] = [\![C_1]\!] \circ [\![C_2]\!] \\
[\![\texttt{if (}E\texttt{)}\ C_1\ \texttt{else}\ C_2]\!] = ([\![E]\!]^{\mathbb{T}} \rhd [\![C_1]\!]) \cup ([\![E]\!]^{\mathbb{F}} \rhd [\![C_2]\!]) \\
[\![\texttt{while (}E\texttt{)}\ C^{F,T}]\!] = ([\![E]\!]^{\mathbb{T}} \rhd [\![C]\!])^* \lhd [\![E]\!]^{\mathbb{F}}
\end{array}$$

While this "point-free" algebraic definition style has some appeal, the verbose original one makes the logic underlying the command semantics more explicit. In particular, for the more general notion of states described in [13] which supports commands that interrupt the control-flow, the underlying logic is considerably more complicated such that an algebraic definition would not really give more insight than the explicit logical characterization.

Next, we define the domain of formulas by the grammar

$$\begin{array}{l}
F := p_n(T_1,\ldots,T_n) \mid \,!F \mid F_1 \text{ and } F_2 \mid F_1 \text{ or } F_2 \mid F_1 \Rightarrow F_2 \mid \ldots \mid \text{forall } I\text{: } F \mid \text{exists } I\text{: } F \\
T := I \mid \text{var } I \mid \text{old } I \mid f_n(T_1,\ldots,T_n)
\end{array}$$

where p_n stands for a family of n-ary predicate constants and f_n for a family of n-ary function constants. Let $\mathit{Environment} := \mathit{Identifier} \to \mathit{Value}$ be the set of formula environments that map logical variables to values. Then the formula semantics is defined as $[\![F]\!](s,s') :\Leftrightarrow \forall e \in \mathit{Environment} : [\![F]\!]^e(s,s')$ where $[\![F]\!]^e(s,s')$ is essentially the classical interpretation of formula F in environment e and $[\![T]\!]^e(s,s')$ is essentially the classical interpretation of term T in environment e. The only places where s and s' matter are in the rules

$$\begin{array}{l}
[\![\text{var } I]\!]^e(s,s') := s'(I) \\
[\![\text{old } I]\!]^e(s,s') := s(I)
\end{array}$$

i.e. var I denotes the value of the program-variable I in the post-state and old I denotes its value in the pre-state (we use this syntax rather than the syntax I' and I indicated in the introduction in order to differentiate between a reference var I respectively old I to a program variable I in a state and a reference I to a logical variable I in the environment). Furthermore, we define the the evaluation of a term $[\![T]\!]^e(s) := [\![T]\!]^e(s,s)$ respectively a formula $[\![F]\!](s) := [\![F]\!](s,s)$ on a single state (such that var I and old I have the same value) which will become handy later. Given a formula F and a set of program variables $\mathit{Is} \subseteq \mathit{Identifier}$ we also define a syntactic abbreviation

$$[F]^{\mathit{Is}} \equiv F \text{ and var } I_1 = \text{old } I_1 \text{ and } \ldots \text{ and var } I_n = \text{old } I_n$$

where $\{I_1,\ldots,I_n\} = \mathit{Identifier} \backslash \mathit{Is}$. In a formula $[F]^{\mathit{Is}}$, the set I_s defines the "frame" of formula F, i.e. the set of variables that may possibly be changed by a transition and whose values must therefore be described by F; all other variables remain unchanged.

The rules for deriving judgments of the form $C : [F]^{\mathit{Is}}$ are given in Figure 1.

- Rule *weaken* allows to weaken the formula derived from a judgment by logical implication. This rule makes use of a judgment of form which $\models_{\mathit{Is}} F$ which derives the classical validity of formula F where all occurrences of terms var I and old I are considered as (different) logical variables but an axiom var I = old I is added for every variable $I \notin \mathit{Is}$.

- Rule *frame* describes how to extend the frame of a rule by a new variable which has the same value in the pre-state as in the post-state.

- Rule *assign* defines the value of variable I in the post-state by a term T derived from the expression E by a judgment $E \simeq T$ which is sound if and only if $\forall s \in \mathit{State} : e \in \mathit{Environmente} : [\![E]\!](s) = [\![T]\!]^e(s)$, i.e. T is interpreted over a single state and has the same value as E. To simplify further substitutions, we assume that all references to program variables in T are of the form old I (not of the form var I).

$$(\mathit{weaken})\ \dfrac{\begin{array}{l} C : [F_1]^{I_s} \\ \models_{I_s} F_1 \Rightarrow F_2 \end{array}}{C : [F_2]^{I_s}}$$

$$(\mathit{frame})\ \dfrac{C : [F]^{I_s}}{C : [\ F\ \textsf{and var}\ I\ \textsf{=}\ \textsf{old}\ I\ ]^{I_s \cup \{I\}}}$$

$$(\mathit{assign})\ \dfrac{E \simeq T}{I\texttt{=}E : [\ \textsf{var}\ I\ \textsf{=}\ T\ ]^{I}}$$

$$(\mathit{var})\ \dfrac{\begin{array}{l} C : [F]^{Is} \\ I_o, I_v \text{ do not occur as terms in } F \end{array}}{\texttt{\{var}\ I\texttt{;}C\texttt{\}} : [\ \textsf{exists}\ I_o, I_v\textsf{:}\ F[I_o/\textsf{old}\ I, I_v/\textsf{var}\ I]\ ]^{Is\setminus\{I\}}}$$

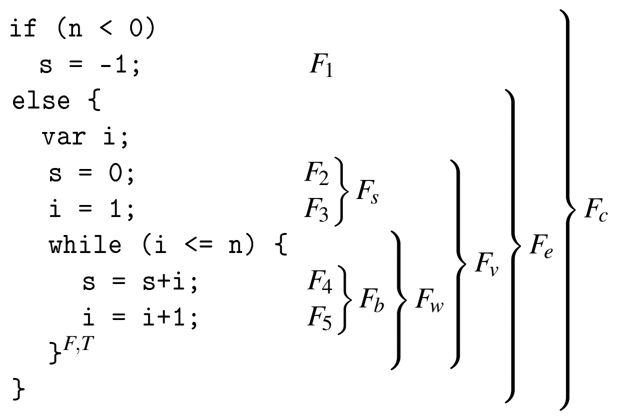
$$(\mathit{seq})\ \dfrac{\begin{array}{l} C_1 : [F_1]^{\{I_1,\dots,I_n\}} \\ C_2 : [F_2]^{\{I_1,\dots,I_n\}} \\ J_1,\dots,J_n \text{ do not occur as terms in } F \end{array}}{\begin{array}{r} C_1\,\texttt{;}\,C_2 : [\ \textsf{exists}\ J_1,\dots,J_n : F_1[J_1/\textsf{var}\ I_1,\dots,J_n/\textsf{var}\ I_n] \land \\ F_2[J_1/\textsf{old}\ I_1,\dots,J_n/\textsf{old}\ I_n]\ ]^{\{I_1,\dots,I_n\}} \end{array}}$$

$$(\mathit{if})\ \dfrac{\begin{array}{l} E \simeq F \\ C_1 : [F_1]^{Is} \\ C_2 : [F_2]^{Is} \end{array}}{\texttt{if (}E\texttt{)}\ C_1\ \texttt{else}\ C_2 : [\ \textsf{if}\ F\ \textsf{then}\ F_1\ \textsf{else}\ F_2\ ]^{Is}}$$

$$(\mathit{while})\ \dfrac{\begin{array}{l} E \simeq F_E \\ C : [F_C]^{I_1,\dots,I_n} \\ \mathit{Invariant}(F, F_E, F_C)^{\{I_1,\dots,I_n\}} \\ J_1,\dots,J_n \text{ do not occur as terms in } F, F_E, F_C \end{array}}{\begin{array}{l} \texttt{while (}E\texttt{)}\ C^{F,T} : \\ \quad [\ \textsf{!}F_E[\textsf{var}\ I_1/\textsf{old}\ I_1,\dots,\textsf{var}\ I_n/\textsf{old}\ I_n]\ \textsf{and} \\ \qquad (F[\textsf{old}\ I_1/\textsf{var}\ I_1,\dots,\textsf{old}\ I_n/\textsf{var}\ I_n]\ \textsf{=>}\ F)\ ]^{I_1,\dots,I_n} \end{array}}$$

$$\begin{array}{l} \mathit{Invariant}(F, F_E, F_C)^{\{I_1,\dots,I_n\}} \equiv \\ \quad \models_{\{I_1,\dots,I_n\}} \\ \qquad \textsf{forall}\ J_1,\dots,J_n\textsf{:} \\ \qquad\quad (F[J_1/\textsf{var}\ I_1,\dots,J_n/\textsf{var}\ I_n]\ \textsf{and} \\ \qquad\quad F_E[J_1/\textsf{old}\ I_1,\dots,J_n/\textsf{old}\ I_n]\ \textsf{and}\ F_C[J_1/\textsf{old}\ I_1,\dots,J_n/\textsf{old}\ I_n])\ \textsf{=>}\ F \end{array}$$

Figure 1: Judgment $C : F$



```
if (n < 0)
  s = -1;
else {
  var i;
  s = 0;
  i = 1;
  while (i <= n) {
    s = s+i;
    i = i+1;
  }^{F,T}
}
```

$F :\Leftrightarrow$ 1 <= var i <= var n+1 and var s = $\sum_{j=1}^{\text{var } i\text{-}1} j$
$T :=$ var n - var i + 1

Figure 2: A Sample Program

- Rule *var* constructs from the formula F describing the behavior of the body of a block C the behavior of the block by existentially quantifying the value of the variable in the pre-state and in the post-state (the phrase $P[V/U]$ is identical to P except that V replaces every occurrence of U).

- Rule *seq* constructs the formula for a command sequence from the formulas of the individual commands with common frame by existentially quantifying the values of the variables in the intermediate state of the sequence (with the help of rule *frame* formulas can be always be extended to a common frame).

- Rule *if* constructs the formula for a conditional statement from the formulas of the individual statements with a common frame. The judgment $E \simeq F$ is sound if and only if $\forall s \in State : [\![E]\!](s) =$ TRUE $\Leftrightarrow [\![F]\!](s)$, i.e. F is interpreted over a single state only. To simplify further substitutions, we assume that all references to program variables in F are of form old I (not of form var I).

- Rule *while* constructs the formula for a loop from the formulas F_E and F_C derived from the loop condition E and body C, respectively, and from a given loop invariant F. The proof obligation *Invariant*$(\ldots)$ states that F indeed remains invariant the execution of the loop body. The derived state relation formula expresses the fact that F_E does not hold in the post-state s' of the loop and that the invariant F holds on the post-state of the loop, provided that it also holds on the pre-state s (i.e. $[\![F]\!](s,s')$ holds, provided that $[\![F]\!](s,s)$ holds).

The rules satisfy the following theorem.

Theorem 1 (Soundness of Core Judgment)**.** *For every command C and formula F, if a judgment C* : *F can be derived, then we have*

$$\forall s,s' \in State : [\![C]\!](s,s') \Rightarrow [\![F]\!](s,s')$$

In the following, we illustrate the derivation of a judgment $C : F_C$ for the program fragment C given in Figure 2 which sums the values from 1 to n. The core of the program is a while loop annotated by an invariant F and a termination term T. For the moment, only F is used, which constrains the state/before after every loop iteration (by references to program variables of the form var I); in general it may also refer to the pre-state of the loop (by references of the form old I). We assume that $Value = \mathbb{Z}$, i.e. all variables denote integer numbers.

First we derive the formulas for the assignment statements:

$$\begin{aligned}
F_1 &\Leftrightarrow [\mathsf{var}\ s\ \mathsf{=}\ \mathsf{-1}]^{\{s\}}\\
F_2 &\Leftrightarrow [\mathsf{var}\ s\ \mathsf{=}\ \mathsf{0}]^{\{s\}}\\
F_3 &\Leftrightarrow [\mathsf{var}\ i\ \mathsf{=}\ \mathsf{1}]^{\{i\}}\\
F_4 &\Leftrightarrow [\mathsf{var}\ s\ \mathsf{=}\ \mathsf{old}\ s + \mathsf{old}\ i]^{\{s\}}\\
F_5 &\Leftrightarrow [\mathsf{var}\ i\ \mathsf{=}\ \mathsf{old}\ i + \mathsf{1}]^{\{i\}}
\end{aligned}$$

Next we extend F_2 and F_3 to a common frame and derive a judgment F_s for the sequence of the two assignment statements before the loop which is simplified to a logically equivalent formula:

$$\begin{aligned}
F_2 &\Leftrightarrow [\mathsf{var}\ s\ \mathsf{=}\ \mathsf{0}\ \mathsf{and}\ \mathsf{var}\ i\ \mathsf{=}\ \mathsf{old}\ i]^{\{s,i\}}\\
F_3 &\Leftrightarrow [\mathsf{var}\ i\ \mathsf{=}\ \mathsf{1}\ \mathsf{and}\ \mathsf{var}\ s\ \mathsf{=}\ \mathsf{old}\ s]^{\{s,i\}}\\
F_s &\Leftrightarrow [\mathsf{exists}\ v_s, v_i\colon v_s\ \mathsf{=}\ \mathsf{0}\ \mathsf{and}\ v_i\ \mathsf{=}\ \mathsf{old}\ i\ \mathsf{and}\ \mathsf{var}\ i\ \mathsf{=}\ \mathsf{1}\ \mathsf{and}\ \mathsf{var}\ s\ \mathsf{=}\ v_s]^{\{s,i\}}\\
&\Leftrightarrow [\mathsf{var}\ i\ \mathsf{=}\ \mathsf{1}\ \mathsf{and}\ \mathsf{var}\ s\ \mathsf{=}\ \mathsf{0}]^{\{s,i\}}
\end{aligned}$$

We do the same to derive a formula F_b for the sequence of assignment statements that represent the body of the loop.

$$\begin{aligned}
F_4 &\Leftrightarrow [\mathsf{var}\ s\ \mathsf{=}\ \mathsf{old}\ s + \mathsf{old}\ i\ \mathsf{and}\ \mathsf{var}\ i\ \mathsf{=}\ \mathsf{old}\ i]^{\{s,i\}}\\
F_5 &\Leftrightarrow [\mathsf{var}\ i\ \mathsf{=}\ \mathsf{old}\ i + \mathsf{1}\ \mathsf{and}\ \mathsf{var}\ s\ \mathsf{=}\ \mathsf{old}\ s]^{\{s,i\}}\\
F_b &\Leftrightarrow [\mathsf{exists}\ v_s, v_i\colon v_s\ \mathsf{=}\ \mathsf{old}\ s + \mathsf{old}\ i\ \mathsf{and}\ v_i\ \mathsf{=}\ \mathsf{old}\ i\ \mathsf{and}\ \mathsf{var}\ i\ \mathsf{=}\ v_i + \mathsf{1}\ \mathsf{and}\ \mathsf{var}\ s\ \mathsf{=}\ v_s]^{\{s,i\}}\\
&\Leftrightarrow [\mathsf{var}\ i\ \mathsf{=}\ \mathsf{old}\ i\ + \mathsf{1}\ \mathsf{and}\ \mathsf{var}\ s\ \mathsf{=}\ \mathsf{old}\ s + \mathsf{old}\ i]^{\{s,i\}}
\end{aligned}$$

From the loop invariant F, we construct the formula F_w for the while loop and simplify it logically:

$$\begin{aligned}
F_w \Leftrightarrow\ & [\mathsf{!}(\mathsf{var}\ i\ \mathsf{<=}\ \mathsf{var}\ n)\ \mathsf{and}\\
&\quad ((\mathsf{1}\ \mathsf{<=}\ \mathsf{old}\ i\ \mathsf{<=}\ \mathsf{old}\ n\mathsf{+}1\ \mathsf{and}\ \mathsf{old}\ s\ \mathsf{=}\ \textstyle\sum_{j=1}^{\mathsf{old}\ i\mathsf{-1}} j)\ \mathsf{=>}\\
&\qquad (\mathsf{1}\ \mathsf{<=}\ \mathsf{var}\ i\ \mathsf{<=}\ \mathsf{var}\ n\mathsf{+}1\ \mathsf{and}\ \mathsf{var}\ s\ \mathsf{=}\ \textstyle\sum_{j=1}^{\mathsf{var}\ i\mathsf{-1}} j))]^{\{s,i\}}\\
\Leftrightarrow\ & [(\mathsf{old}\ n\ \mathsf{<}\ \mathsf{var}\ i)\ \mathsf{and}\\
&\quad ((\mathsf{1}\ \mathsf{<=}\ \mathsf{old}\ i\ \mathsf{<=}\ \mathsf{old}\ n\mathsf{+}1\ \mathsf{and}\ \mathsf{old}\ s\ \mathsf{=}\ \textstyle\sum_{j=1}^{\mathsf{old}\ i\mathsf{-1}} j)\ \mathsf{=>}\\
&\qquad (\mathsf{1}\ \mathsf{<=}\ \mathsf{var}\ i\ \mathsf{<=}\ \mathsf{old}\ n\mathsf{+}1\ \mathsf{and}\ \mathsf{var}\ s\ \mathsf{=}\ \textstyle\sum_{j=1}^{\mathsf{var}\ i\mathsf{-1}} j))]^{\{s,i\}}\\
\Leftrightarrow\ & [(\mathsf{old}\ n\ \mathsf{<}\ \mathsf{var}\ i)\ \mathsf{and}\\
&\quad (\mathsf{1}\ \mathsf{<=}\ \mathsf{old}\ i\ \mathsf{<=}\ \mathsf{old}\ n\mathsf{+}1\ \mathsf{and}\ \mathsf{old}\ s\ \mathsf{=}\ \textstyle\sum_{j=1}^{\mathsf{old}\ i\mathsf{-1}} j)\ \mathsf{=>}\\
&\qquad (\mathsf{var}\ i\ \mathsf{=}\ \mathsf{old}\ n\mathsf{+}1\ \mathsf{and}\ \mathsf{var}\ s\ \mathsf{=}\ \textstyle\sum_{j=1}^{\mathsf{old}\ n} j)]^{\{s,i\}}
\end{aligned}$$

To prove the invariance of F, we generate the following proof obligation

$$\begin{aligned}
&\models_{\{s,i\}}\\
&\quad \mathsf{forall}\ v_s, v_i\colon\\
&\qquad \mathsf{1}\ \mathsf{<=}\ v_i\ \mathsf{<=}\ \mathsf{var}\ n\mathsf{+}1\ \mathsf{and}\ v_s\ \mathsf{=}\ \textstyle\sum_{j=1}^{v_i\mathsf{-1}} j\ \mathsf{and}\\
&\qquad v_i\ \mathsf{<=}\ \mathsf{var}\ n\ \mathsf{and}\ \mathsf{var}\ i\ \mathsf{=}\ v_i\ + \mathsf{1}\ \mathsf{and}\ \mathsf{var}\ s\ \mathsf{=}\ v_s + v_i\ \mathsf{=>}\\
&\qquad\quad \mathsf{1}\ \mathsf{<=}\ \mathsf{var}\ i\ \mathsf{<=}\ \mathsf{var}\ n\mathsf{+}1\ \mathsf{and}\ \mathsf{var}\ s\ \mathsf{=}\ \textstyle\sum_{j=1}^{\mathsf{var}\ i\mathsf{-1}} j
\end{aligned}$$

whose correctness can be easily established.

The formula F_w derived above contains contains an implication which expresses the truth of the invariant only under a condition on the range of the program variable i and the value of s in the pre-state of the loop. By combining this condition with the formula F_s that establishes the pre-state of the loop, the precondition is logically simplified to the condition that program variable n is not negative:

$$\begin{aligned} F_v \;\Leftrightarrow\;& [\text{exists } v_s, v_i : v_s = 0 \text{ and } v_i = 1 \text{ and} \\ &\quad \text{old } n < \text{var } i \text{ and} \\ &\quad (1 <= v_i <= \text{old } n{+}1 \text{ and } v_s = \textstyle\sum_{j=1}^{v_i\text{-}1} j => \\ &\qquad \text{var } i = \text{old } n{+}1 \text{ and var } s = \textstyle\sum_{j=1}^{\text{old } n} j)]^{\{s,i\}} \\ \Leftrightarrow\;& [\text{old } n < \text{var } i \text{ and} \\ &\quad (0 <= \text{old } n => \text{var } i = \text{old } n{+}1 \text{ and var } s = \textstyle\sum_{j=1}^{\text{old } n} j)]^{\{s,i\}} \end{aligned}$$

The program variable i is local to the else-branch of the conditional statement: its effect is thus captured by existential quantification; by logical simplification, the variable is removed from the formula F_e for this branch:

$$\begin{aligned} F_e \;\Leftrightarrow\;& [\text{exists } v_i : \text{old } n < v_i \text{ and } (0 <= \text{old } n => v_i = \text{old } n{+}1 \text{ and var } s = \textstyle\sum_{j=1}^{\text{old } n} j)]^{\{s\}} \\ \Leftrightarrow\;& [0 <= \text{old } n => \text{var } s = \textstyle\sum_{j=1}^{\text{old } n} j]^{\{s\}} \end{aligned}$$

Finally, we combine the effect of both branches of the conditional statement and derive the formula for the whole command:

$$\begin{aligned} F_c \;\Leftrightarrow\;& [\text{if old } n < 0 \text{ then var } s = \text{-1 else } 0 <= \text{old } n => \text{var } s = \textstyle\sum_{j=1}^{\text{old } n} j]^{\{s\}} \\ \Leftrightarrow\;& [\text{if old } n < 0 \text{ then var } s = \text{-1 else var } s = \textstyle\sum_{j=1}^{\text{old } n} j]^{\{s\}} \end{aligned}$$

We note that F_c describes the "semantic essence" of the program fragment: if the program variable n is initially negative, the program variable s has finally the value -1, and else the sum of all values from 1 to n. The calculation of this semantic essence is the core of the calculus: it proceeds "inside-out" from atomic commands to composed commands and may apply logic to simplify formulas; its soundness depends on the generated proof obligation that the formula annotating the loop is indeed invariant invariant with respect to the execution of the loop body.

3 Pre- and Post-Conditions

From the judgment of the previous section, the auxiliary judgments presented in Figure 3 can be derived:

- $\text{PRE}(C,Q) = P$: Given a state formula Q and a command C, this judgment determines a state formula P, such that if P holds in the state in which C is executed, Q holds afterward. P is thus a pre-condition of C with respect to post-condition Q.
- $\text{POST}(C,P) = Q$: Given a state formula P and a command C, this judgment determines a state formula Q, such that if P holds in the state in which C is executed, Q holds afterwords. Q is thus a post-condition of C with respect to pre-condition P.

Formulas P and Q represent state conditions, i.e. are evaluated over single states; to simplify further substitutions, we assume that they refer to program variables in the form old I (and not var I); the generated pre-/post-conditions preserve this property.

In general, the judgments do not always determine the weakest pre- respectively strongest post-condition, because the reasoning about loops is based on externally provided invariants, which are not necessarily the strongest possible ones. Still they preserve the following constraint.

Theorem 2 (Soundness of Pre/Post-Conditions)**.** *For all commands C and state formulas P,Q, if the judgment* $\text{PRE}(C,Q) = P$ *or the judgment* $\text{PRE}(C,P) = Q$ *can be derived, we have:*

$$\forall s,s' \in State : [\![P]\!](s) \wedge [\![C]\!](s,s') \Rightarrow [\![Q]\!](s')$$

$$(pre)\ \frac{\begin{array}{l} C : [F]^{\{I_1,\ldots,I_n\}} \\ \mathsf{var}\ \text{\textvisiblespace does not occur in } Q \\ J_1,\ldots,J_n \text{ do not occur as terms in } F,Q \end{array}}{\begin{array}{l} \textsc{pre}(C,Q) = \\ \quad \mathsf{forall}\ J_1,\ldots,J_n\text{: } F[J_1/\mathsf{var}\ I_1,\ldots,J_n/\mathsf{var}\ I_n] \texttt{ => } \\ \qquad Q[J_1/\mathsf{old}\ I_1,\ldots,J_n/\mathsf{old}\ I_n] \end{array}}$$

$$(pre)\ \frac{\begin{array}{l} C : [F]^{\{I_1,\ldots,I_n\}} \\ \mathsf{var}\ \text{\textvisiblespace does not occur in } P \\ J_1,\ldots,J_n \text{ do not occur as terms in } F,P \end{array}}{\begin{array}{l} \textsc{post}(C,P) = \\ \quad \mathsf{exists}\ J_1,\ldots,J_n\text{: } P[J_1/\mathsf{old}\ I_1,\ldots,J_n/\mathsf{old}\ I_n]\ \mathsf{and} \\ \qquad F[J_1/\mathsf{old}\ I_1,\ldots,J_n/\mathsf{old}\ I_n,\mathsf{old}\ I_1/\mathsf{var}\ I_1,\ldots,\mathsf{old}\ I_n/\mathsf{var}\ I_n] \end{array}}$$

Figure 3: Judgments $\textsc{pre}(C,Q) = P$ and $\textsc{post}(C,P) = Q$

As an example, for command $C =$ `x=x+1` with state relation $[\mathsf{var}\ x \texttt{ = } \mathsf{old}\ x \texttt{ + } 1]^{\{x\}}$, we get

$$\begin{array}{lcl} \textsc{pre}(C,Q) & = & \mathsf{forall}\ v_x\text{: } v_x \texttt{ = } \mathsf{old}\ x \texttt{ + } 1 \texttt{ =>} Q[v_x/\mathsf{old}\ x] \\ & = & Q[\mathsf{old}\ x \texttt{ + } 1/\mathsf{old}\ x] \\ \\ \textsc{post}(C,P) & = & \mathsf{exists}\ v_x\text{: } P[v_x/\mathsf{old}\ x]\ \mathsf{and}\ \mathsf{old}\ x \texttt{ = } v_x \texttt{ + } 1 \end{array}$$

The computation of pre- and post-conditions has various applications; one of them is in the termination calculus presented in the following section.

4 Termination

The judgments presented up to now are only concerned with the partial correctness of a command: they constrain which transitions may take place but do not demand that any transition actually must take place, i.e. a command may also block respectively not terminate. To support also total correctness arguments, we are now going to capture the termination of a command C by an interpretation $\langle\!\langle C \rangle\!\rangle \in \mathit{StateCondition} := \mathbb{P}(\mathit{State})$ such that $\langle\!\langle C \rangle\!\rangle(s)$ (i.e. $s \in \langle\!\langle C \rangle\!\rangle$) holds if the execution of C in state s terminates. For our programming language, the definition of $\langle\!\langle C \rangle\!\rangle$ is as follows:

$$\begin{array}{l}
\langle\!\langle I\texttt{=}E \rangle\!\rangle(s) :\Leftrightarrow \textsc{true} \\
\langle\!\langle \texttt{\{var } I\texttt{;}C\texttt{\}} \rangle\!\rangle(s) :\Leftrightarrow \forall s' \in \mathit{State} : (\forall J \in \mathit{Identifier} : J \neq I \Rightarrow s(J) = s'(J)) \Rightarrow \langle\!\langle C \rangle\!\rangle(s') \\
\langle\!\langle C_1\texttt{;}C_2 \rangle\!\rangle(s) :\Leftrightarrow \langle\!\langle C_1 \rangle\!\rangle(s) \wedge \forall s' \in \mathit{State} : [\![C]\!](s,s') \Rightarrow \langle\!\langle C_2 \rangle\!\rangle(s) \\
\langle\!\langle \texttt{if (}E\texttt{) } C_1 \texttt{ else } C_2 \rangle\!\rangle(s) :\Leftrightarrow \\
\quad \textsc{if } [\![E]\!](s) = \textsc{true then } \langle\!\langle C_1 \rangle\!\rangle(s) \textsc{ else } \langle\!\langle C_2 \rangle\!\rangle(s) \\
\langle\!\langle \texttt{while (}E\texttt{) } C^{F,T} \rangle\!\rangle(s) :\Leftrightarrow \\
\quad (\forall n \in \mathbb{N}, t \in \mathit{State}^* : \\
\qquad t(0) = s \wedge (\forall i \in \mathbb{N} : i < n-1 \Rightarrow [\![E]\!](s) = \textsc{true} \wedge [\![C]\!](t(i),t(i+1))) \Rightarrow \langle\!\langle C \rangle\!\rangle(t(n))) \wedge \\
\quad (\neg \exists t \in \mathit{State}^\omega : \\
\qquad t(0) = s \wedge \forall i \in \mathbb{N} : [\![E]\!](s) = \textsc{true} \wedge [\![C]\!](t(i),t(i+1)))
\end{array}$$

Most cases are self-evident; the termination semantics for while loops demands that the execution of each loop iteration must terminate and that there are not infinitely many such iterations. The relationship between both semantic interpretations of commands is determined by the following constraint.

$$(\mathit{strengthenT})\ \frac{\begin{array}{l} C \downarrow F_1 \\ \models_{\emptyset} F_2 \Rightarrow F_1 \end{array}}{C \downarrow F_2}$$

$$(\mathit{assignT})\ \ I\text{=}E \downarrow \textsf{true}$$

$$(\mathit{varT})\ \frac{\begin{array}{l} C \downarrow F \\ J \text{ does not occur as a term in } F \end{array}}{\texttt{\{var } I\texttt{;} C\texttt{\}} \downarrow \textsf{forall } J\textsf{: } F[J/\textsf{old } I]}$$

$$(\mathit{seqT})\ \frac{\begin{array}{l} C_1 \downarrow F_1 \\ C_2 \downarrow F_2 \\ \text{PRE}(C_1, F_2) = F_3 \end{array}}{C_1\,;C_2 \downarrow F_1 \textsf{ and } F_3}$$

$$(\mathit{ifT})\ \frac{\begin{array}{l} E \simeq F \\ C_1 \downarrow F_1 \\ C_2 \downarrow F_2 \end{array}}{\texttt{if (}E\texttt{) } C_1 \texttt{ else } C_2 \downarrow \textsf{if } F \textsf{ then } F_1 \textsf{ else } F_2}$$

$$(\mathit{whileT})\ \frac{\begin{array}{l} E \simeq F_E \\ C : [F_C]^{I_1,\ldots,I_n} \\ \mathit{Invariant}(F, F_E, F_C)^{\{I_1,\ldots,I_n\}} \\ C \downarrow F_T \\ J_1,\ldots,J_n \text{ do not occur as terms in } F, F_E, F_C, F_T, T \\ \models_{\{I_1,\ldots,I_n\}} \\ \quad \textsf{forall } J_1,\ldots,J_n\textsf{:} \\ \qquad F[J_1/\textsf{old } I_1,\ldots,J_n/\textsf{old } I_n, \textsf{old } I_1/\textsf{var } I_1,\ldots,\textsf{old } I_n/\textsf{var } I_n] \textsf{ and} \\ \qquad F_E \textsf{ and } F_C \textsf{ =>} \\ \qquad\quad F_T \textsf{ and } T \textsf{ < } T[\textsf{var } I_1/\textsf{old } I_1,\ldots,\textsf{var } I_n/\textsf{old\_}I_n] \\ \textsf{<} \text{ represents a well-founded ordering} \end{array}}{\texttt{while (}E\texttt{) } C^{F,T} \downarrow F[\textsf{old } I_1/\textsf{var } I_1,\ldots,\textsf{old } I_n/\textsf{var } I_n]}$$

Figure 4: Judgment $C \downarrow F$

Theorem 3 (Termination Semantics). *For every command C, we have*

$$\forall s \in \mathit{State} : \langle\!\langle C \rangle\!\rangle(s) \Rightarrow \exists s' \in \mathit{State} : [\![C]\!](s, s')$$

i.e. the state condition $\langle\!\langle C \rangle\!\rangle$ represents a (possibly proper) subset of the domain of the state relation $[\![C]\!]$.

In the following, we derive judgments of the form $C \downarrow F$ where F is represents a state condition (i.e. F is a formula evaluated over a single state) such that $[\![F]\!]$ describes those pre-states for which the execution of command C must terminate; F is thus called a termination condition of C. Figure 4 gives the rules for deriving termination conditions. Rule *strengthenT* shows that a termination condition can be strengthened by logical implication. Most other rules are pretty self-evident, with the exception of (*whileT*): this rule claims the termination of a loop in a state in which the loop invariant F holds provided that it can be proved that F is indeed an invariant and that in every iteration the execution of the loop body terminates and the termination term decreases according to some well-founded ordering <. For this proof, the pre-state of the loop iteration may be assumed to satisfy the invariant and the loop condition.

The termination calculus meets the following soundness constraint.

Theorem 4 (Soundness of Termination Judgment)**.** *For all commands C and state formulas F, if the judgment $C \downarrow F$ can be derived, we have*

$$\forall s \in State : [\![F]\!](s) \Rightarrow \langle\!\langle C \rangle\!\rangle(s)$$

As an example, for the sample program C given in Figure 2, we can derive $C \downarrow$ true where, in addition to the obligation of proving the correctness of the loop invariant, the following proof obligation is generated (after some logical simplification):

$\models_{s,i}$
 forall v_s, v_i :
  1 <= old i <= old n+1 and old s = $\sum_{j=1}^{\text{old } i\text{-}1} j$ and
  old i <= old n and var s = old s + old i and var i = old i + 1 =>
   old n - old i + 1 <= var n - var i + 1

From the last two lines and the fact that in frame $\{s,i\}$ we have axiom var n = old n, this condition is clearly valid.

5 Related Work

The idea of "programs as state relations" is not new; it has been variously advocated, e.g. in Lamport's "Temporal Logic of Actions" [7], Boute's "Calculational Semantics" [2], Hehner's "Practical Theory of Programming" [4], Hoare and Jifeng's "Unifying Theory of Programming" [6], Morgan's calculus for "Programming from Specifications" [9], and the "Refinement Calculus" of Back and von Wright [1] respectively Morris [10]. Still most program reasoning texts and tools are due to historical reasons dominated by the pre/post-condition view; this has been criticized in [11].

As a concrete example, the recent calculus of Mili et al [8] describes a loop by a reflexive and transitive state relation that is determined by the loop invariant; if the relation (that characterizes the relationship between two states that are separated by an arbitrary number of loop iterations) is also symmetric, (a lower bound approximation of) the state function implemented by the loop can be automatically derived. This relational framework differs from our in various respects:

- The interpretation of a relation $R \subseteq State \times State$ as a program specification is that, for every pre-state $s \in \text{dom}(R)$, a program described by R *must* terminate with some post-state s' such that $\langle s, s' \rangle \in R$. This gives rise to the notion that a relation R_1 refines a relation R_2 if and only if $(R_1 \circ S) \cap (R_2 \circ S) \cap (R_1 \cup R_2) = R_2$ where $S := State \times State$ is the universal relation.

  In our framework, a program is characterized by a pair of a state relation $R \subseteq State \times State$ and a termination condition $T \subseteq State$ such that $T \subseteq \text{dom}(R)$; here R describes only the space of possible transitions while T describes those pre-states for which a transition must take place. A specification $\langle R_1, T_1 \rangle = \langle [\![F_1]\!], [\![G_1]\!] \rangle$ refines a specification $\langle R_2, T_2 \rangle = \langle [\![F_1]\!], [\![G_1]\!] \rangle$ if and only if $R_1 \subseteq R_2$ and $T_2 \subseteq T1$, i.e. if $F_1 \Rightarrow F_2$ and $G_2 \Rightarrow G_1$ are valid formulas. Our framework thus provides a simple logical characterization of the semantics of a program and the notions of program specification and specification refinement.

- An invariant F is required to be a reflexive and transitive state relation, i.e. $[\![F]\!](s,s)$ must hold for every state s independently of the program context into which the loop is embedded, i.e., whether s is a possible pre-state of the loop or not.

  In our framework, F is need not be reflexive. The relation $[\![F]\!](s,s)$ must only hold for every state s that is a possible pre-state for the context in which the loop is embedded (the relation derived from the loop has a corresponding pre-state condition as the antecedent of an implication that has the invariant as its consequent). The proof of invariance also does not establish general transitivity of the relation but only that the invariance (with respect to the pre-state of the loop) is preserved by every iteration. While it would be also possible to strengthen the proof obligation for loops to check the reflexivity and transitivity of F, our framework thus allows to describe the invariance of a loop with respect to the actual context in which it is executed.

- Our framework uses a more general notion of state than described in this extended abstract. A state not only contains the values of program variables but also execution modes that arise from executing statements that interrupt the control-flow (loop continuations and breaks, function returns, exception throws). These extensions give a rise to an essentially more complex execution behavior, in particular with respect to loops, that can be nevertheless characterized in a relational framework and handled by the judgments introduced in the previous sections. The framework also considers global program variables, modular reasoning about methods specified by contracts, and (direct and indirect) recursive method invocations. Details can be found in [13].

Similar differences hold for most of the calculi cited above.

6 Conclusions

The "state relation" view by itself does not give rise to bigger automation, in particular the derivation of a loop's state relation still depends on an externally provided invariant. Nevertheless, we believe that switching from state conditions to state relations substantially alters one's mind set and is useful beyond the mere purpose of verification: with appropriate tool support (automatic derivation and simplification of formulas characterizing commands and program fragments) it is ultimately able to give humans *insight* into the programs they write.

References

[1] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer, New York, 1998.

[2] Raymond T. Boute. Calculational Semantics: Deriving Programming Theories from Equations by Functional Predicate Calculus. *ACM Transactions on Programming Languages and Systems*, 28(4):747–793, July 2006.

[3] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic.* MIT Press, Cambridge, MA, USA, 2000.

[4] Eric C.R. Hehner. *A Practical Theory of Programming.* Springer, New York, 2006. `http://www.cs.utoronto.ca/~hehner/aPToP`.

[5] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[6] C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice Hall, London, UK, 1998. `http://www.unifyingtheories.org`.

[7] Leslie Lamport. *Specifying Systems; The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002. `http://research.microsoft.com/users/lamport/tla/book.html`.

[8] Ali Mili, Shir Aharon, Chaitanya Nadkarni, Lamia Labed Jilani, Asma Louhichi, and Olfa Mraihi. Reflexive transitive invariant relations: A basis for computing loop functions. *Journal of Symbolic Computation*, in press:DOI: 10.1016/j.jsc.2008.11.007, 2010.

[9] Carroll Morgan. *Programming from Specifications*. Prentice Hall, London, UK, 2nd edition, 1998. `http://web2.comlab.ox.ac.uk/oucl/publications/books/PfS`.

[10] J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[11] David Lorge Parnas. Really Rethinking 'Formal Methods'. *Computer*, 43(1):28–34, 2010.

[12] The RISC ProgramExplorer, 2010. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, `http://www.risc.jku.at/research/formal/software/ProgramExplorer`.

[13] Wolfgang Schreiner. A Program Calculus. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, September 2008. `http://www.risc.jku.at/people/schreine/papers/ProgramCalculus2008.pdf`.

[14] Wolfgang Schreiner. The RISC ProofNavigator: A Proving Assistant for Program Verification in the Classroom. *Formal Aspects of Computing*, 21(3):277–291, 2009.