1. Programs as State Relations

- Specification by State Predicates
  - Hoare calculus and predicate transformers use state predicates.
  - Formulas that talk about a single (pre/post-)state.
  - In such a formula, a reference $x$ means “the value of program variable $x$ in the given state”.
- Relationship between pre/post-state is not directly expressible.
  - Requires uninterpreted mathematical constants.
    - $\{x = a\} x := x + 1 \{x = a + 1\}$
- Unchanged variables have to be explicitly specified.
  - $\{x = a \wedge y = b\} x := x + 1 \{x = a + 1 \wedge y = b\}$
- The semantics of a command $c$ is only implicitly specified.
  - Specifications depend on auxiliary state conditions $P, Q$.
    - $\{P\} c(Q)$
    - $\wp(c, Q) = P$

Let us turn our focus from individual states to pairs of states.

2. The RISC ProgramExplorer

We introduce formulas that denote state relations.
- Talk about a pair of states (the pre-state and the post-state).
  - $\text{old } x$: “the value of program variable $x$ in the pre-state”.
  - $\text{var } x$: “the value of program variable $x$ in the post-state”.
- We introduce the logical judgment $c : [F]^x$:
  - If the execution of $c$ terminates normally, the resulting post-state is related to the pre-state as described by $F$.
  - Every variable $y$ not listed in the set of variables $x, \ldots$ has the same value in the pre-state and in the post-state.
    - $c : F \wedge \text{var } y = \text{old } y \wedge \ldots$
    - $x := x + 1 : [\text{var } x = \text{old } x + 1]^x$
    - $x := x + 1 : \text{var } x = \text{old } x + 1 \wedge \text{var } y = \text{old } y \wedge \text{var } z = \text{old } z \wedge \ldots$

We will discuss the termination of commands later.
Let us calculate this "semantic essence" of the program.

### Example

\[ c_1 = y := y + 1; \]
\[ c_2 = x := x + y \]
\[ c_1 : [\text{var } y = \text{old } y + 1] x \]
\[ c_2 : [\text{var } x = \text{old } x + \text{old } y] x \]
\[ c_1 : [\text{var } y = \text{old } y + 1 \land \text{var } x = \text{old } x] x y \]
\[ c_2 : [\text{var } x = \text{old } x + \text{old } y \land \text{var } y = \text{old } y] x y \]
\[ c_1 ; c_2 : [\exists x_0, y_0 : y_0 = \text{old } y + 1 \land x_0 = \text{old } x \land \text{var } x = x_0 \land y_0 \land \text{var } y = y_0] x y \]
\[ c_1 ; c_2 : [\text{var } x = \text{old } x + \text{old } y + 1 \land \text{var } y = \text{old } y + 1] x y \]

Mechanical translation and logical simplification.

---

\[ c = \]
\[ \text{if } n < 0 \]
\[ s := -1 \]
\[ \text{else} \]
\[ s := 0 \]
\[ i := 0 \]
\[ \text{while } i < n \text{ do } \{ l, t \} \]
\[ s := s + i \]
\[ i := i + 1 \]

\[ c : [\text{var } i = \text{old } i + 1] s, i \]
\[ s, i : [\text{var } n + \text{var } s = \sum_{j=0}^{\text{var } i - 1} j] \]

The loop relation is derived from the invariant (not the loop body); we have to prove the preservation of the loop invariant.
Example

\[ c = \text{if } n < 0 \text{ then } s := -1 \text{ else } b \]
\[ b = (s := 0; i := 0; w) \]
\[ w = \text{while } i < n \text{ do } \{ l, t \} \text{ (} s := s + i; i = i + 1 \) \]
\[ s := 0; \]
\[ i := 0; \]
\[ s := 0; \]
\[ w : [-\{ \text{var } i < \text{var } n \} \land (0 \leq \text{old } i \leq \text{old } n \land \text{var } s = \sum_{j=0}^{\text{old } n-1} j \Rightarrow l)]^{s,i} \]
\[ w : [\text{var } i \geq \text{old } n \land (0 \leq \text{old } i \leq \text{old } n \land \text{var } s = \sum_{j=0}^{\text{old } n-1} j \Rightarrow l)]^{s,i} \]

Partial Correctness

- Specification \((xs, P, Q)\)
  - Set of program variables \(xs\) (which may be modified).
  - Precondition \(P\) (a formula with "old \(xs\" but no "var \(xs\".
  - Postcondition \(Q\) (a formula with both "old \(xs\" and "var \(xs\")
- Partial correctness of implementation \(c\)
  1. Derive \(c : [F]^{xs}\).
  2. Prove \(F \Rightarrow (P \Rightarrow Q)\)
     - Or: \(P \Rightarrow (F \Rightarrow Q)\)
     - Or: \((P \land F) \Rightarrow Q\)

Verification of partial correctness leads to the proof of an implication.
Relationship to Other Calculi

Let all state conditions refer via "old xs" to program variables xs.

- Hoare Calculus
  - For proving \( \{ P \} c \{ Q \} \), it suffices to derive \( c : [F]a \) and prove \( P \land F \Rightarrow Q[\text{var} \; xs/old \; xs] \).
- Predicate Transformers
  - Assume we can derive \( c : [F]a \).
  - If \( c \) does not contain loops, then
    \[
    \text{wp}(c, Q) = \forall xs : F[xs/\text{var} \; xs] \Rightarrow Q[\text{x}s/old \; xs] \\
    \text{sp}(c, P) = \exists xs : P[\text{x}s/old \; xs] \land F[xs/old \; xs, old \; xs/\text{var} \; xs]
    \]
  - If \( c \) contains loops, the result is still a valid pre/post-condition but not necessarily the weakest/strongest one.

A generalization of the previously presented calculi.

Termination Condition Rules

- \( \text{skip} \downarrow \text{true} \)
- \( \text{abort} \downarrow \text{true} \)
- \( \text{x} := e \downarrow \text{true} \)
- \( \frac{c_1 \downarrow T_1 \quad c_2 \downarrow T_2}{c_1 ; c_2 \downarrow T_1 \land \text{wp}(c_1, T_2)} \)
- \( \frac{\text{if } e \; \text{then } c \downarrow T}{\text{if } e \; \text{then } c_1 \downarrow T_1 \quad \text{else } \frac{c_2 \downarrow T_2}{\text{if } e \; \text{then } c_1 \text{ else } c_2 \downarrow T}} \)
- \( c : [F]a \quad c \downarrow T \quad \forall xs, ys, zs : I[ys/old \; xs, zs/\text{var} \; xs] \land e[ys/xs] \land F[ys/old \; xs, zs/\text{var} \; xs] \Rightarrow T[ys/old \; xs] \land 0 \leq t[zs/old \; xs] < t[ys/old \; xs] \quad \text{while } e \text{ do } \{l, t\} \quad c \downarrow t \geq 0 \)

In every iteration of a loop, the loop body must terminate and the termination term must decrease (but not become negative).

Example

\[
\text{c} = \begin{cases} 
\text{if } n < 0 
\quad s := -1 \\
\text{else} 
\quad s := 0 \\
\quad i := 0 \\
\quad \text{while } i < n \text{ do } \{l, t\} 
\quad s := s + i \\
\quad i := i + 1 
\end{cases}
\]

We still have to prove the constraint on the loop iteration.
Example

\[ s := s + i; \quad i := i + 1 \downarrow \text{true} \]

\[ \forall s_x, s_y, s_z, i_x, i_y, i_z : \]
\[ (0 \leq i_y \leq \text{old } n \land s_y = \sum_{j=0}^{i_y-1} j) \land \]
\[ i_y < \text{old } n \land \]
\[ (s_z = s_y + i_y \land i_z = i_y + 1) \Rightarrow \]
\[ \text{true} \land \]
\[ 0 \leq \text{old } n - i_z < \text{old } n - i_y \]

Also this constraint is simple to prove.

Abortion

Also abortion can be ruled out by proving side conditions in the usual way.


See the report for the full calculus.

The RISC ProgramExplorer

- An integrated environment for program reasoning.
  - Research Institute for Symbolic Computation (RISC), 2008–.
  - http://www.risc.jku.at/research/formal/software/ProgramExplorer
  - Integrates the RISC ProofNavigator for computer-assisted proving.
  - Written in Java, runs under Linux (only), freely available (GPL).
- Programs written in "MiniJava":
  - Subset of Java with full support of control flow interruptions.
  - Value (not pointer) semantics for arrays and objects.
- Theories and specifications written in a formula language.
  - Derived from the language of the RISC ProofNavigator.
- Semantic analysis and verification.
  - Program methods are translated into their "semantic essence".
  - Open for human inspection.
  - From the semantics, the verification tasks are generated.
  - Solved by automatic decision procedure or interactive proof.

Tight integration of executable programs, declarative specifications, mathematical semantics, and verification tasks.
Using the Software

See “The RISC ProgramExplorer: Tutorial and Manual”.

- Develop a theory.
  - File “Theory.theory” with a theory Theory of mathematical types, constants, functions, predicates, axioms, and theorems.
  - Can be also added to a program file.
- Develop a program.
  - File “Class.java” with a class Class that contains class (static) and object (non-static) variables, methods and constructors.
  - Class may be annotated by a theory (and an object invariant).
  - Methods may be annotated by method specifications.
  - Loops may be annotated by invariants and termination terms.
- Analyze method semantics.
  - Transition relations, termination conditions, ... of the method body and its individual commands.
- Perform verification tasks.
  - Frame, postcondition, termination, preconditions, loop-related tasks, type-checking conditions.

Starting the Software

Starting the software:
module load ProgramExplorer (only at RISC)
ProgramExplorer &

Command line options:
Usage: ProgramExplorer [OPTION]...
OPTION: one of the following options:
-h, --help: print this message.
-cp, --classpath [PATH]: directories representing top package.

Environment Variables:
PE_CLASSPATH: the directories (separated by ":") representing the top package (default the current working directory) ...

- Task repository created/read in current working directory:
  Subdirectory .PETASKS.timestamp (ProgramExplorer tasks)
  Subdirectory .ProofNavigator (ProofNavigator legacy)

The Graphical User Interface

A Program

/*@..
class Sum
{
static int sum(int n) /*@..
{
  int s;
  if (n < 0)
    s = -1;
  else
  {
    s = 0;
    int i = 1;
    while (i <= n) /*@..
    {
      s = s+i;
      i = i+1;
    }
  }
  return s;
}
@*/

Markers /*@.. indicate hidden mathematical annotations.
A Theory

/*@
theory {
  sum: (INT, INT) -> INT;
  sumaxiom: AXIOM
  FORALL(m: INT, n: INT):
    IF n<m THEN
      sum(m, n) = 0
    ELSE
      sum(m, n) = n+sum(m, n-1)
    ENDIF;
}
@*/

class Sum
...

The introduction of a function \( \text{sum}(m, n) = \sum_{j=m}^{n} j \).
The Semantics View

Select method symbol "sum" and menu entry "Show Semantics".

The Method Body

Body Knowledge
[[Show Original Formulas]]

Pre-State Knowledge
old x < Base.MAXixa

Effects
executes: true, continues: false, breaks: false, returns: true

Transition Relation
if old x < 0 then
  returns@next ∧ value@next = -1
else
  returns@next ∧
  (BaseBase.int ln old x + 1 ∧ ln ∧ value@next = sum(1, ln-1)) ^
  old x < Base.MAXixa
endif

Select method symbol "sum" and menu entry “Show Semantics”.

A Body Command

Statement Knowledge
[[Show Original Formulas]]

Pre-State Knowledge
old x < Base.MAXixa ∧ old n ≥ 0 ∧ old r = 0 ∧ old i = I

Precondition
old x < Base.MAXixa ∧ 1 ≤ old i ∧ old n ≤ 1 ∧ old r = sum(1, old l-1)

Effects
executes: true, continues: false, breaks: false, returns: false

variables: x, i, exceptions:

Transition Relation
var i = old i ∧ old x < Base.MAXixa ∧ 1 ≤ var i = sum(1, var i-1)

Termination Condition
executes@now ⇒ old x = old l ≥ -1

The Semantics Elements

- Pre-State Knowledge
  What is known about the pre-state of the command.

- Precondition
  What has to be true for the pre-state of the command such that the command may be executed.

- Effects
  Which kind of effects may the command have.
  - variables: which variables may be changed.
  - exceptions: which exceptions may be thrown.
  - executes, continues, breaks, returns: may the execution terminate normally, may it be terminated by a continue, break, return.

- Transition Relation
  The prestate/poststate relationship of the command.

- Termination
  What has to be true for the pre-state of the command such that the command terminates.

Formulas are shown after simplification (see “Show Original Formulas”).
Constraining a State

Select the loop body, enter in the box the condition \( \text{VAR } s=2 \text{ AND VAR } i=1 \), press “Submit”, and move the mouse to \( i=i+1 \).

The Verification Tasks

- **Effects**: does the method only change those global variables indicated in the method’s assignable clause?
- **Postcondition**: do the method’s precondition and the body’s state relation imply the method’s postcondition?
- **Termination**: does the method’s precondition imply the body’s termination condition?
- **Precondition**: does a statement’s prestate knowledge imply the statement’s precondition?
- **Loops**: is the loop invariant preserved, the measure well-formed (does not become negative) and decreased?
- **Type checking conditions**: are all formulas well-typed?
- **Specification validation**: does for every input that satisfies a precondition exist a result that does (not) satisfy the postcondition?

Partially solved by automatic decision procedure, partially by an interactive computer-supported proof.

The Task States

The task status is indicated by color (icon).

- **Blue (sun)**: the task was solved in the current execution of the RISC ProgramExplorer (automatically or by an interactive proof).
- **Violet (partially clouded)**: the task was solved in a previous execution by an interactive proof.
  - Nothing has changed, so we need not perform the proof again.
  - However, we may replay the proof to investigate it.
- **Red (partially clouded)**: there exists a proof but it is either not complete or cannot be trusted any more (something has changed).
- **Red (fully clouded)**: there does not yet exist a proof.

Select “Execute Task” to start/replay a proof, “Show Proof” to display a proof, “Reset Task” to delete a proof.
A Postcondition Proof

Linear Search

/*@..*/
public class Searching
{
public static int search(int[] a, int x) /*@..
{
    int n = a.length;
    int r = -1;
    int i = 0;
    while (i < n && r == -1) /*@..
    {
        if (a[i] == x)
            r = i;
        else
            i = i+1;
    }
    return r;
}
}

The Representation of Arrays

The program type int[] is mapped to the mathematical type Base.IntArray.
theory Base
{
    ...
    IntArray: TYPE =
        [#value: ARRAY int OF int, length: nat, null: BOOLEAN#];
    ...
}

- (VAR a).length: the number of elements in array a.
- (VAR a).value[i]: the element with index i in array a.
- (VAR a).null: a is the null pointer.

Program type Class is mapped to mathematical type Class.Class;
Class[] is mapped to Class.Array.

Theory

/*@*/
theory uses Base {
    int: TYPE = Base.int;
    intArray: TYPE = Base.IntArray;
    smallestPosition: FORMULA
        FORALL(a: IntArray, n: NAT, x: int):
            (EXISTS(i:int): 0 <= i AND i < n AND a.value[i] = x)
            =>
            (EXISTS(i:int): 0 <= i AND i < n AND a.value[i] = x AND
                (FORALL(j:int): 0 <= j AND j < n AND a.value[j] = x
                    =>
                        j >= i));
}
@*/
public class Searching
...
Method Specification

```java
public static int search(int[] a, int x) /*@ requires NOT (VAR a).null; ensures LET result = VALUE@NEXT, n = (VAR a).length IN IF result = -1 THEN FORALL(i: INT): 0 <= i AND i < n => (VAR a).value[i] /= VAR x ELSE 0 <= result AND result < n AND (FORALL(i: INT): 0 <= i AND i < result => (VAR a).value[i] /= VAR x) AND (VAR a).value[result] = VAR x ENSURE */ ...
```

Loop Annotation

```java
while (i < n && r == -1) /*@
invariant NOT (VAR a).null AND VAR n = (VAR a).length AND 0 <= VAR i AND VAR i <= VAR n AND (FORALL(i: INT): 0 <= i AND i < VAR i => (VAR a).value[i] /= VAR x) AND (VAR r = -1 OR (VAR r = VAR i AND VAR i < VAR n AND (VAR a).value[VAR r] = VAR x)); decreases IF VAR r = -1 THEN VAR n - VAR i ELSE 0 ENDIF;@g*/
{
if (a[i] == x)
r = i;
else
i = i+1;
}
```

Method Semantics

Transition Relation

\[(\exists inBase : int, n \in Base : int : n = old a.length \land (in \geq n \land value@next \neq -1) \land 0 \leq in \land in \leq \sigma) \land \forall i \in \mathbb{Z} : 0 \leq i \land i < in \Rightarrow old a.value[i] \neq old x) \land \forall i \land value@next = -1 \land value@next \leq in \land in \leq n \land old a.value[value@next] \neq old x) \land \sim old a.null \land returns@next\]

Termination Condition

executes@now \Rightarrow old a.length \geq 0

Verification Tasks

- [Searching.search] effects
- [Searching.search] postcondition
- [Searching.search] termination
- preconditions
  - [Searching.search-0] declaration precondition
  - [Searching.search-1] declaration precondition
  - [Searching.search-2] while loop precondition
  - [Searching.search-3] conditional precondition
  - [Searching.search-4] assignment precondition
- loops
  - [Searching.search-5] invariant is preserved
- type checking conditions
  - [Searching.local:pxx] value is in interval
  - [Searching.local:smu] value is in interval
  - [Searching.local:uns] value is in interval
- specification validation (optional)
Working Strategy

- Develop theory.
  - Introduce interesting theorems that may be used in verifications.
- Develop specifications.
  - Validate specifications, e.g. by showing satisfiability and non-triviality.
- Develop program with annotations.
  - Validate programs/annotations by investigating program semantics.
- Prove postcondition and termination.
  - Partial and total correctness.
  - By proofs necessity of additional theorems may be detected.
- Prove precondition tasks and loop tasks.
  - By proofs necessity of additional theorems may be detected.
- Prove mathematical theorems.
  - Validation of auxiliary knowledge used in verifications.

The integrated development of theories, specifications, programs, annotations is crucial for the design of provably correct programs.