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.
  \[
  \{x = a \land y = b\} x := x + 1 \{x = a + 1 \land y = b\}
  \]
- The semantics of a command \( c \) is only implicitly specified.
  - Specifications depend on auxiliary state conditions \( P, Q \).
    \[
    \{P\} c \{Q\}
    \]

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

2. The RISC ProgramExplorer

Specification by State Relations

- 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 \( c : [F]^{\ast} \cdot \cdot \cdot \)
  - 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.
    \[
    c : F \land \text{var } y = \text{old } y \land \ldots
    \]
    \[
    x := x + 1 : [\text{var } x = \text{old } x + 1]^{y}
    \]
    \[
    x := x + 1 : \text{var } x = \text{old } x + 1 \land \text{var } y = \text{old } y \land \text{var } z = \text{old } z \land \ldots
    \]

We will discuss the termination of commands later.
have to prove the preservation of the loop invariant.

Let us calculate this "semantic essence" of the program.

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

Example

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

\[ I \quad \text{s.t.} \quad 0 \leq i \leq n \quad \text{and} \quad s = \sum_{j=0}^{i-1} j \]
\[ (s := s + i; i := i + 1) \quad \text{var } s = \text{old } s + \text{old } i \quad \text{and} \quad i = \text{old } i + 1 \]
\[ \text{while } i < n \text{ do } \{ I, t \} \]
\[ s := s + i \]
\[ i := i + 1 \]

\[ I \quad \text{s.t.} \quad 0 \leq i \leq n \quad \text{and} \quad s = \sum_{j=0}^{i-1} j \]
\[ t = \text{old } n - \text{old } i \]

Let us calculate this "semantic essence" of the program.
Example

c = if n < 0 then s := −1 else b
b = (s := 0; i := 0; w)
w = while i < n do {I, t} (s := s + i; i = i + 1)

s := 0; i := 0: [var s = 0 ∧ var i = 0]s,i

Partial Correctness

Verification of partial correctness leads to the proof of an implication.

Example

c = if n < 0 then s := −1 else b
b = (s := 0; i := 0; w)
w = while i < n do {I, t} (s := s + i; i = i + 1)

s := 0; i := 0: [var s = 0 ∧ var i = 0]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]c.
  2. Prove F ⇒ (P ⇒ Q)
     - Or: P ⇒ (F ⇒ Q)
     - Or: (P ∧ F) ⇒ 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 \(c : [F]^{xs}\)
  - and prove \(P \land F \Rightarrow Q[\text{var } xs/old \ xs]\).

- **Predicate Transformers**
  - Assume we can derive \(c : [F]^{xs}\).
  - If \(c\) does not contain loops, then
    \[
    \begin{align*}
    \text{wp}(c, Q) &= \forall \text{xs} : F[\text{xs}/\text{var } \text{xs}] \Rightarrow Q[\text{xs}/\text{old } \text{xs}] \\
    \text{sp}(c, P) &= \exists \text{xs} : P[\text{xs}/\text{old } \text{xs}] \land F[\text{xs}/\text{old } \text{xs}, \text{old } \text{xs}/\text{var } \text{xs}]
    \end{align*}
    \]
  - 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

- We introduce a judgment \(c \downarrow T\).
  - State condition \(T\) (a formula with "old xs" but no "var xs").
  - Starting with a pre-state that satisfies condition \(T\) the execution of
    command \(c\) terminates.

- **Total correctness** of implementation \(c\).
  - Specification \((xs, P, Q)\).
  - Derive \(c \downarrow T\).
  - Prove \(P \Rightarrow T\).

Also verification of termination leads to the proof of an implication.

Example

\[
\begin{align*}
\text{c} &= \text{if } n < 0 \\
&\quad \text{then } s := -1 \\
&\quad \text{else } \\
&\quad \text{if } i = 0 \\
&\quad \quad \text{then } s := s + i \\
&\quad \quad \text{else } i := i + 1 \\
\text{i} &\Rightarrow 0 \leq \text{var } i \leq \text{old } n \land \text{var } s = \sum_{j=0}^{i-1} j \\
\text{t} &\Rightarrow \text{old } n - \text{old } i \\
\text{c} &\downarrow \text{if old } n < 0 \text{ then true else ...}
\end{align*}
\]

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

\[
\begin{align*}
  s &:= s + i; \\
i &:= i + 1 \quad \downarrow \text{true}
\end{align*}
\]

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

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

1. Programs as State Relations

2. 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 (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.
The introduction of a function \( \text{sum}(m, n) = \sum_{j=m}^{n} j \).
The Semantics View

A Body Command

Statement Knowledge

Pre-State Knowledge

Pre-condition

Effects

Transition Relation

Termination Condition

The Method Body

Body Knowledge

Pre-State Knowledge

Effects

Transition Relation

Termination

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

Select the loop body, enter in the box the condition VAR s=2 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 (VAR a).null = FALSE; 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 ENDIF; @*/ ...
```

Loop Annotation

```java
while (i < n && r == -1) /*@ invariant (VAR a).null = FALSE 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; @*/
{
if (a[i] == x)
r = i;
else
i = i+1;
}
```

Method Semantics

Verification Tasks
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.