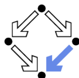
SEMANTIC TECHNOLOGIES FOR CS EDUCATION

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Wolfgang.Schreiner@risc.jku.at
http://www.risc.jku.at

JKU
JOHANNES KEPLER
UNIVERSITY LINZ



Formal Methods Education

Formal specification and verification of programs/algorithms.

- Specification S, program/algorithm P^A with annotations A.
  - S: Pre- and post-conditions.
  - A: Loop invariants, termination measures.
- Derive set of verification conditions $VC = vc(P^A, S)$.
  - Verification condition generator vc.
- Proof of VC often fails because it is actually invalid.
  - Error in program.
  - Specification does not match program.
  - Loop invariant too strong or too weak.

Most time spent in trying to prove something which is not true.

1. The RISC ProgramExplorer

2. The RISC Algorithm Language (RISCAL)

3. Conclusions

Program Verification (Classical)

$$
P^A \quad + \quad S \quad \begin{array}{c} \nearrow \\ \longrightarrow \\ \searrow \end{array} \quad \begin{array}{c} VC_1 \\ \vdots \\ VC_i \\ \vdots \\ VC_n \end{array}
$$

- Annotated Program P^A
- Specification S
- Verification Conditions VC_i

Only when proving VC_i, we learn whether P, A, S "match".

Soundness of Verification

- Program P ... command
  - Program semantics $[\![P]\!] \subseteq State \times State$
- Specification S ... formula
  - Specification semantics $[\![S]\!] \subseteq State \times State$
- Verification Condition VC_i ... formula
  - Condition semantics $[\![VC_i]\!] \in \{\mathsf{true}, \mathsf{false}\}$

$$[\![VC_i]\!] \Rightarrow \forall s, s' \in State : [\![P]\!](s, s') \Rightarrow [\![S]\!](s, s')$$

If the verification conditions are valid, the state transitions performed by the program are allowed by the specification.

Program Reasoning (Alternative)

$$
\begin{array}{ccccccc}
 & & \circlearrowright & & & & \\
P^A & \longrightarrow & F & + & S & \longrightarrow & F \Rightarrow S \\
 & \searrow & & & & & \\
 & & TC_1 & & & & \\
 & & \dots & & & & \\
 & & TC_n & & & &
\end{array}
$$

- Annotated program P^A
- Transition formula F
- Specification S
- Verification Condition $F \Rightarrow S$
- Translation Conditions $TC_1, \ldots, TC_n$

F is the "semantic essence" of P^A open for investigation.

Soundness of Translation

- Program P ... command
  - Program semantics $[\![P]\!] \subseteq \mathit{State} \times \mathit{State}$
- Formula F ... formula
  - Formula semantics $[\![F]\!] \subseteq \mathit{State} \times \mathit{State}$
- Translation Conditions TC_i ... formula
  - Condition semantics $[\![TC_i]\!] \in \{\mathsf{true}, \mathsf{false}\}$

$$[\![TC_i]\!] \Rightarrow \forall s, s' \in \mathit{State} : [\![P]\!](s, s') \Rightarrow [\![F]\!](s, s')$$

If the translation conditions are valid, the state transitions performed by the program are captured by the formula.

Specification by State Relations

- We introduce formulas that denote state relations.
  - Talk about a pair of states (the pre-state and the post-state).
  - old x: "the value of program variable x in the pre-state".
  - var x: "the value of program variable x in the post-state".
- We introduce the logical judgment $c : [f]^{xs}_{g,h}$
  - 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 xs has the same value in the pre-state and in the post-state.

$$c : f \wedge \text{var } y = \text{old } y \wedge \dots$$

  - VCs g (state relation) and h (state condition).

$x := x + 1 : [\text{var } x = \text{old } x + 1]^x$

$x := x+1 : \text{var } x = \text{old } x+1 \wedge \text{var } y = \text{old } y \wedge \text{var } z = \text{old } z \wedge \dots$

State Relation Rules

$$\frac{c : [f]^{xs}_{g,h} \qquad x \notin xs}{c : [f \wedge \mathsf{var}\ x = \mathsf{old}\ x]^{xs\ \cup\ \{x\}}_{g,h}}$$

$$\frac{e \simeq_h t}{x = e : [\mathsf{var}\ x = t]^{\{x\}}_{\mathsf{true},h}}$$

$$\frac{c : [f]^{xs}_{g,h}}{\{\mathtt{var}\ x\mathtt{;}\ c\} : [\exists x_0, x_1 : f[x_0/\mathsf{old}\ x, x_1/\mathsf{var}\ x]]^{xs \setminus x}_{g, \forall x : h[x/\mathsf{old}\ x]}}$$

$$\frac{c_1 : [f_1]^{xs}_{g_1,h_1} \qquad c_2 : [f_2]^{xs}_{g_2,h_2} \qquad \mathsf{PRE}(c_1, h_2) = h_3}{\{c_1\,\mathtt{;}\,c_2\} : [\exists ys : f_1[ys/\mathsf{var}\ xs] \wedge f_2[ys/\mathsf{old}\ xs]]^{xs}_{g_1 \wedge g_2, h_1 \wedge h_3}}$$

$$\frac{e \simeq_h f_e \qquad c_1 : [f_1]^{xs}_{g_1,h_1} \qquad c_2 : [f_2]^{xs}_{g_2,h_2}}{\mathtt{if}\ (e)\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 : [\mathsf{if}\ f_e\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2]^{xs}_{g_1 \wedge g_2, h \wedge \mathsf{if}\ f_e\ \mathsf{then}\ h_1\ \mathsf{else}\ h_2}}$$

$$\frac{\begin{array}{c} e \simeq_h f_e \qquad c : [f_c]^{xs}_{g_c,h_c} \\ g \equiv \forall xs, ys, zs : f[xs/\mathsf{old}\ xs, ys/\mathsf{var}\ xs] \wedge f_e[ys/\mathsf{old}\ xs] \wedge f_c[ys/\mathsf{old}\ xs, zs/\mathsf{var}\ xs] \Rightarrow \\ h[ys/\mathsf{old}\ xs] \wedge f[xs/\mathsf{old}\ xs, zs/\mathsf{var}\ xs] \end{array}}{\mathtt{while}\ (e)^{f,t}\ c : [f \wedge \neg f_e[\mathsf{var}\ xs/\mathsf{old}\ xs]]^{xs}_{f_c \wedge g, h \wedge f[\mathsf{old}\ xs/\mathsf{var}\ xs]}}$$

Example

```
if (n < 0)
  s = -1;                F1
else {
  var i;
   s = 0;                F2 } Fs
   i = 1;                F3 }
   while (i <= n) {
      s = s+i;           F4 } Fb
      i = i+1;           F5 }
   }^{F,T}
}
```

$F_2, F_3 \to F_s$; $F_4, F_5 \to F_b$; $F_b \to F_w$; $F_s, F_w \to F_v$; $F_v \to F_e$; $F_1, F_e \to F_c$

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

$T :=$ var n - var i + 1

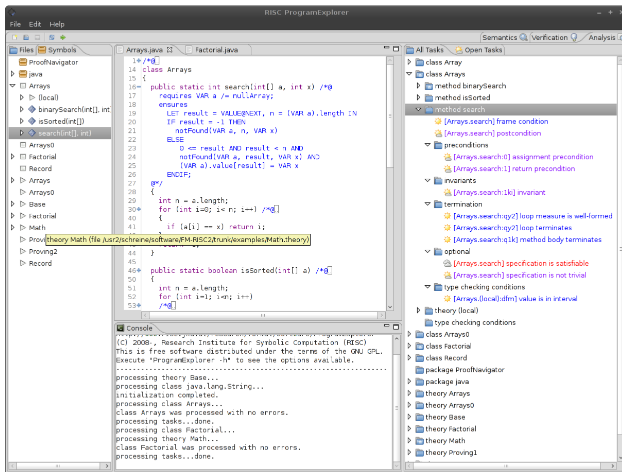
$$F_c \Leftrightarrow [\text{if old } n < 0 \text{ then var } s = \text{-1 else var } s = \sum_{j=1}^{\text{old } n} j]^{\{s\}}$$

The program computes for non-negative n the sum from 1 to n.

The RISC ProgramExplorer

- An integrated program reasoning environment.
  - Programming language MiniJava.
  - Theory/specification language in the style of PVS/CVC.
  - Semi-automatic proving assistant RISC ProofNavigator.
- Semantics view.
  - Semantics of a method body.
  - Pre/post-condition reasoning.
- Analyze view (verification tasks).
  - Type checking conditions.
  - Statement preconditions.
  - Loop invariants.
  - Method frame preservation.
  - Method termination.
  - Method postcondition.
- Verify view.
  - Proof construction and management.

The RISC ProgramExplorer



http://www.risc.jku.at/research/formal/software/ProgramExplorer

1. The RISC ProgramExplorer

2. The RISC Algorithm Language (RISCAL)

3. Conclusions

Verifying via Checking Finite Instances

A step-wise approach to verification.

- Algorithm/program $P^A[n]$ and specification $S[n]$.
  - Parameter $n \in \mathbb{N}$ bounds size of variable domain $D[n]$.
    - May have different bounds for different domains.
  - Value of parameter is arbitrarily large (not fixed in program).
- Program operates over a finite domain.
  - Can be executed for all inputs of the domain.
- Specification and annotations are decidable.
  - By evaluating their semantics over the domain.

Structure of program/specification can be used for the validation of correctness before its actual verification.

Verifying via Checking Finite Instances

Verify some finite instance $P^A[c]$ and $S[c]$.

- Testing
  - Run $P[c]$ with some input $i \in D[c]$ and watch output.
  - Validate informal correctness of program for some inputs.
- Runtime assertion checking
  - Additionally evaluate $A[c]$ and $S[c]$ and report violations.
  - Validate formal correctness of program for some inputs.
- Model checking
  - Runtime assertion checking for every input $i \in D[c]$.
  - Validate formal correctness for all inputs in $D[c]$.
    - May detect that $A[c]$ respectively $S[c]$ is too strong.
- Generate verification condition $VC[c] = vc(P^A[c], S[c])$.
  - Decidable by evaluation or SMT solver.
    - May detect that $A[c]$ is too weak.

Verify correctness of some $P^A[c]$ with respect to $S[c]$.

Verifying via Checking Finite Instances

Verify every instance $P^A[n]$ and $S[n]$.

- Prove $\forall n \in \mathbb{N}.\ VC[n]$
  - Computer-assisted reasoning again.
  - But now, by previous validation, high chance of being valid.

Verify correctness of $P^A[n]$ w.r.t. $S[n]$ for arbitrary $n \in \mathbf{N}$.

The RISC Algorithm Language (RISCAL)

- Formal theory and algorithm specification language.
  - Static type system with parameterized domains $T[n]$.
  - Functions (implicit, explicit, recursive).
  - Predicates (explicit, recursive).
  - Theorems (predicates claimed to be always true).
  - Procedures (functions defined by commands).
  - Pre-/post-conditions, loop invariants, termination measures.
- Non-deterministic semantics.
  - Implicit function definitions and non-deterministic choices.
- Formulas currently decided by evaluation.
  - (Parallel) model checker to validate theories and algorithms.
- Future work on generation of VCs and their verification.
  - Decidable by evaluation and by SMT solvers.
- Ultimately extension to proof-based verification.

Implementation based on semantics of programs and formulas.

Example

```
val n: ℕ;
type Literal = ℤ[-n,n];
...
type Formula = Set[Clause];
...
theorem notValid(f:Formula) ... ⇔ valid(f) ⇔ ¬satisfiable(not(f));
pred satisfiable(f:Formula) ⇔ ∃v:Valuation. valuation(v) ∧ satisfies(v,f);
fun substitute(f:Formula,l:Literal):Formula = {c\{-l} | c∈f with ¬(l∈c)};
...
multiple pred DPLL(f:Formula)
  requires formula(f); ensures result ⇔ satisfiable(f);
  decreases |literals(f)|;
⇔
  if f = ∅[Clause] then
    ⊤
  else if ∅[Literal] ∈ f then
    ⊥
  else
    choose l∈literals(f) in
    DPLL(substitute(f,l)) ∨ DPLL(substitute(f,-l));
```

Semantics-based Implementation

$$ComSem := Single + Multiple$$

$$Single := Command \rightarrow (Context \rightarrow Context)$$

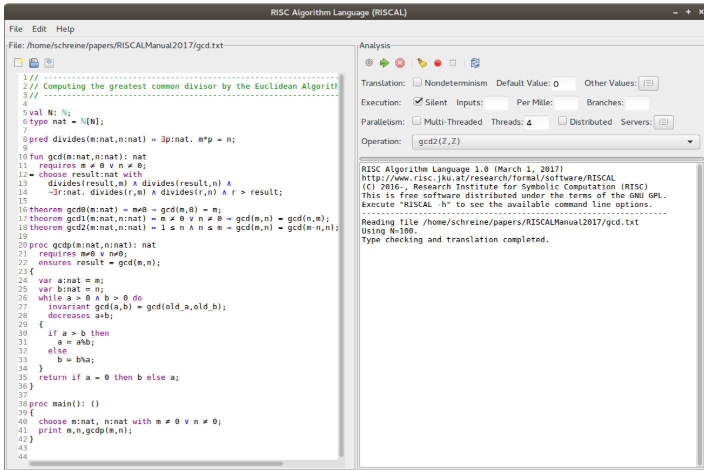
$$Multiple := Command \rightarrow (Context \rightarrow Seq(Context))$$

$$Seq(T) := Unit \rightarrow (Null + Next(T, Seq(T)))$$

```
public static interface ComSem  {
  public interface Single extends ComSem, Function<Context,Context> { }
  public interface Multiple extends ComSem, Function<Context,Seq<Context>> { }
}
public interface Seq<T> extends Supplier<Seq.Next<T>> {
  // public Seq.Next<T> get();
  public final static class Next<T> {
    public final T head; public final Seq<T> tail;
    ...
  }
}
```

Non-deterministic semantics based on lazy sequences.

The RISCAL Software



http://www.risc.jku.at/research/formal/software/RISCAL

1. The RISC ProgramExplorer

2. The RISC Algorithm Language (RISCAL)

3. Conclusions

Conclusions

- RISC ProgramExplorer in use since 2011.
  - Regular course “Formal Methods in Computer Science” for master students of computer science and mathematics.
  - Students master specifications and verifications of moderate to medium difficulty.
  - Main challenge is to get the invariants right.
  - Some profit from formal semantics view, but also quite a few seem to consider this as a superfluous “extra”.
- RISC Algorithm Language released in 2017.
  - Soon to be used in the course mentioned above.
  - Sample contents currently being developed by students (discrete mathematics, computer algebra, number theory).
  - Very positive feedback due to “full automation”.
  - Further development in the frame of LOGTECHEDU.
    - LOGTECHEDU: Logic Technology for Computer Science Education, LIT seed project, 2017–2019.
    - LIT: Linz Institute of Technology of the JKU.