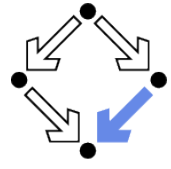
IMPLEMENTING LOGIC BY SEMANTICS

The RISCAL Approach to Automating Program Reasoning over Finite Domains

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)



Formal Methods in Computer Science

- Specification:
  - Describe precisely a computational problem to be solved.
    - Precondition: what can be assumed about the inputs.
    - Postcondition: what has to be established for the outputs.
  - Formal specification: conditions are precisely formulated in the language of formal (first order predicate) logic.
- Verification:
  - Show that a program correctly implements the specification.
    - For all inputs that satisfy the precondition, the program must produce outputs that satisfy the postcondition.
  - Formal verification: given a formal program semantics, correctness can be established with mathematical rigor.

Formal logic is the fundament of precise program reasoning.

1. **Formal Specification**

2. **Proof-Based Verification**

3. **Semantics-Based Checking**

4. **Conclusions**

A Program Specification

The specification of a “conditional swap” problem.

- Input: an integer array a of length N and indices i, j in a:

```
requires 0 ≤ i ∧ i < N ∧ 0 ≤ j ∧ j < N;
```

- Output: an array $result$ that is identical to a except that the elements at i and j are in ascending order:

```
ensures a[i] ≤ a[j] ⇒ result[i] = a[i] ∧ result[j] = a[j];
ensures a[i] > a[j] ⇒ result[i] = a[j] ∧ result[j] = a[i];
ensures ∀k:index with 0 ≤ k ∧ k < N.
          k ≠ i ∧ k ≠ j ⇒ result[k] = a[k];
```

Conditional Swap

An implementation of the “conditional swap” problem.

```
val N:ℕ; val M:ℕ;
type index = ℤ[-N,N]; type elem = ℤ[-M,M]; type array = Array[N, elem];

proc cswap(a:array, i:index, j:index): array
  requires 0 ≤ i ∧ i < N ∧ 0 ≤ j ∧ j < N;
  ensures a[i] ≤ a[j] ⇒ result[i] = a[i] ∧ result[j] = a[j];
  ensures a[i] > a[j] ⇒ result[i] = a[j] ∧ result[j] = a[i];
  ensures ∀k:index with 0 ≤ k ∧ k < N. k ≠ i ∧ k ≠ j ⇒ result[k] = a[k];
{
  var b:array = a;
  if b[i] > b[j] then
  {
    var x:elem := b[i];
    b[i] := b[j];
    b[j] := x;
  }
  return b;
}
```

Formal Verification

How to rigorously demonstrate the correctness of the program with respect to its specification?

- Generate verification conditions.
  - Logic formulas whose validity implies the correctness.
  - Can be automatically generated by a formal calculus.
  - Requires the annotation of the program with sufficiently strong extra-information (loop invariants/termination terms).
- Prove the conditions.
  - Possibly performed/supported by an automated reasoner/interactive proof assistant.
  - First order logic is not decidable, thus fully automatic proofs can generally not be expected.
  - In general, (also) human effort is required.

The traditional (and only fully general) approach.

1. Formal Specification

2. Proof-Based Verification

3. Semantics-Based Checking

4. Conclusions

Verification Condition Generation

E.g., Dijkstra's weakest precondition calculus.

- Verification condition: $pre \Rightarrow wp(prog, post)$

$$
\begin{aligned}
wp(x := e, post) &:= post[e/x] \\
wp(\texttt{skip}, post) &:= post \\
wp(c_1; c_2, post) &:= wp(c_1, wp(c_2, post)) \\
wp(\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2, post) &:= (b \Rightarrow wp(c_1, post)) \\
&\quad \wedge (\neg b \Rightarrow wp(c_2, post)) \\
&\ldots
\end{aligned}
$$

Fully automatic; without loops, no extra information is required.

Verification of the Program

```
var b:array = a;
if b[i] > b[j] then
{
  var x:elem := b[i];
  b[i] := b[j];
  b[j] := x;
}
return b;
```

$$wp(cswap, post) :=$$

$$(b[i] \not> b[j] \Rightarrow post[b/result][a/b]) \wedge$$

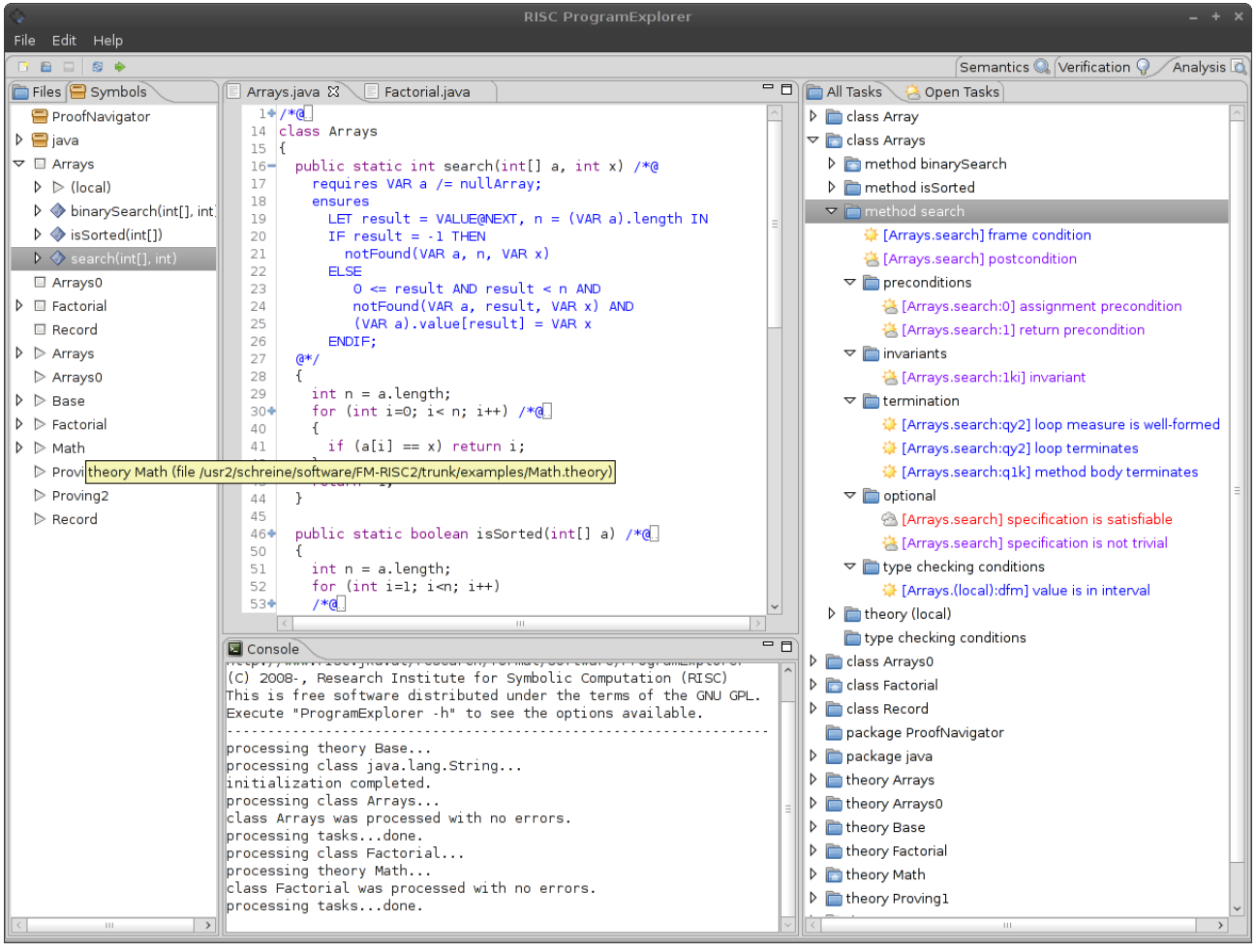
$$(b[i] > b[j] \Rightarrow post[b/result][b[j \mapsto x]/b][b[i \mapsto b[j]]/b][b[i]/x][a/b])$$

We have to prove $pre \Rightarrow wp(cswap, post)$;
for this we use some automation support.

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



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

BubbleSort

```
pred sorted(a:array, i:index)
  requires 0 ≤ i ∧ i < N;
⇔ ∀k:index. i ≤ k ∧ k < N-1 ⇒ a[k] ≤ a[k+1];

proc bubbleSort(a:array): array
 ensures sorted(result,0);
{
  var b:array = a;
  for var i:index := 0; i < N-1; i := i+1 do
    for var j:index := 0; j < N-i-1; j := j+1 do
      b := cswap(b,j,j+1);
  return b;
}
```

How to verify the correctness of this program?

Verification Condition Generation for Loop

- Weakest precondition of a loop annotated with an invariant:

$$wp(\texttt{while}\ b\ \texttt{do}^{inv(x,x')}\ c^x, post) := inv(x,x) \\ \wedge\ (\forall x'.inv(x,x') \Rightarrow post[x'/x])$$

  - c^x: a command that only changes variable x.
  - $inv(x,x')$: a formula that relates a variable's prestate value x to its poststate value x'.
  - Also have to prove that the loop body maintains the invariant:

$$inv(x,x') \wedge b[x'/x] \Rightarrow wp(c^x, inv(x,x'))$$

Only partial correctness: for termination, also a "termination measure" is required.

BubbleSort

```
pred sorted(a:array, i:index) requires 0 ≤ i ∧ i < N;
⇔ ∀k:index. i ≤ k ∧ k < N-1 ⇒ a[k] ≤ a[k+1];
pred lesseq(a:array, i:index) requires 0 ≤ i ∧ i < N;
⇔ ∀k:index. 0 ≤ k ∧ k < i ⇒ a[k] ≤ a[i];

proc bubbleSort(a:array): array
 ensures sorted(result,0);
{
  var b:array = a;
  for var i:index := 0; i < N-1; i := i+1 do
    invariant 0 ≤ i ∧ (N > 0 ⇒ i < N);
    invariant sorted(b,N-1-i) ∧ (i > 0 ⇒ lesseq(b,N-i));
  {
    for var j:index := 0; j < N-i-1; j := j+1 do
      invariant 0 ≤ i ∧ (N > 0 ⇒ i < N) ∧ 0 ≤ j ∧ j < N-i;
      invariant sorted(b,N-1-i) ∧ (i > 0 ⇒ lesseq(b,N-i));
      invariant lesseq(b,j);
      b := cswap(b,j,j+1);
  }
  return b;
}
```

Verification of Loop-Based Programs

- Many potential sources of errors.
  - Errors in the program.
  - Errors in the specification.
  - Errors in the loop invariants.
  - Failure to find the adequate proof strategy.
- If a proof fails, it is hard to determine the reason.
  - Most time in proof-based verification is wasted by attempting to prove invalid verification conditions!

It would be good to have an easier way to find errors.

1. Formal Specification

2. Proof-Based Verification

3. Semantics-Based Checking

4. Conclusions

Formal Verification

Is there an alternative to proof-based verification?

- Model Checking
  - Check whether program runs satisfy the specification.
  - Runtime assertion checking: user selects certain runs.
  - Model checking: automatic consideration of all possible runs.
- Problem: only complete under restrictive assumptions.
  - Decidable conditions to be checked.
  - Original model checking: state space is finite (domains of all program variables are finite).
  - Abstraction-based model checking: program can be checked in a finite abstraction of the state space.
  - Bounded model checking: program runs with a bounded number of loop iterations.

Usually applied only to automatic detection of runtime errors.

The RISC Algorithm Language (RISCAL)

- Formal theory and algorithm specification language.
  - Static type system with parameterized types $T[n]$.
  - 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 in formulas and programs.
- Semantics-based implementation of programs/formulas.
  - All phrases are translated to their denotational semantics.
  - Model checker executes semantics for all possible inputs.
  - Parallel implementation allows to check large state spaces.

A semantics-based approach to checking and verification.

Denotational Semantics of Programs

$[\![.]\!] : \mathit{Command} \times \mathit{State} \to \mathit{State}$

$$
\begin{aligned}
[\![x := e]\!](s) &:= s[x \mapsto [\![e]\!](s)] \\
[\![\texttt{skip}]\!](s) &:= s \\
[\![c_1 ; c_2]\!](s) &:= [\![c_2]\!]([\![c_1]\!](s)) \\
[\![\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2]\!](s) &:= \text{if } [\![b]\!](s) = \text{true then} [\![c_1]\!](s) \text{ else } [\![c_2]\!](s) \\
[\![\texttt{while } b \texttt{ do } c]\!](s) &:= w(b, c, s) \text{ where} \\
w(b, c, s) &:= \text{if } [\![b]\!](s) = \text{false} \\
&\quad \text{then } s \\
&\quad \text{else } w(b, c, [\![c]\!](s))
\end{aligned}
$$

Executable by a direct implementation.

Denotational Semantics of Formulas

$$[\![.]\!] : \mathit{Formula} \times \mathit{State} \to \{\text{true}, \text{false}\}$$

$$[\![p(t_1, \ldots, t_n)]\!](s) := [\![p]\!]([\![t_1]\!](s), \ldots, [\![t_n]\!](s))$$

$$[\![\neg F]\!](s) := \begin{cases} \text{true} & \text{if } [\![F]\!](s) = \text{false} \\ \text{false} & \text{else} \end{cases}$$

$$[\![F_1 \wedge F_2]\!](s) := \begin{cases} \text{true} & \text{if } [\![F_1]\!](s) = [\![F_2]\!](s) = \text{true} \\ \text{false} & \text{else} \end{cases}$$

$$\ldots$$

$$[\![\forall x : T.\ F]\!](s) := \begin{cases} \text{true} & \text{if } \forall a \in [\![T]\!].\ [\![F]\!](s[x \mapsto a]) = \text{true} \\ \text{false} & \text{else} \end{cases}$$

$$\ldots$$

Executable by a direct implementation (provided that the semantics $[\![T]\!]$ of every type T is finite).

Relationship Semantics to Logic

The weakest precondition calculus is sound with respect to the denotational semantics of programs and formulas:

$$(\forall s \in \mathit{State}.\ [\![\, \mathit{pre} \Rightarrow \mathit{wp}(c, \mathit{post}) \,]\!](s) = \mathrm{true}) \Leftrightarrow$$
$$(\forall s \in \mathit{State}.\ [\![\, \mathit{pre} \,]\!](s) \Rightarrow [\![\, \mathit{post} \,]\!]([\![\, c \,]\!](s)))$$

- If the derived verification condition is valid, then every execution of the program that starts in a state that satisfies the precondition yields a result state that satisfies the postcondition . . .
- . . . and vice versa.

Proof-based verification and semantics-based checking yield the same result.

Semantics-based Implementation

$$ComSem := Single + Multiple$$

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

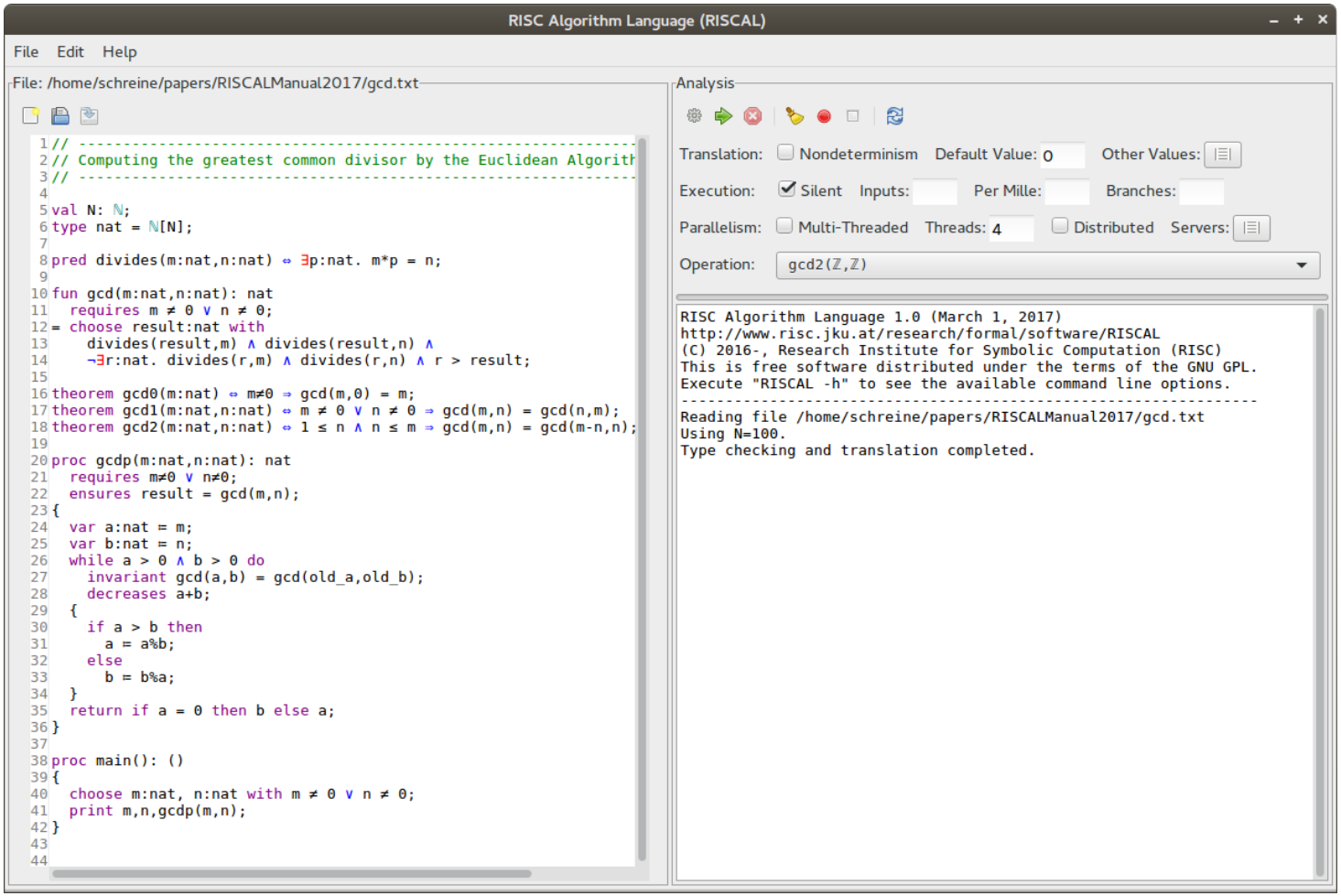
$$Multiple := Command \to (Context \to Seq(Context))$$

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

```
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

Verifying via Checking Finite Instances

A step-wise approach to verification.

- Program $P^A[n]$ and specification $S[n]$.
  - Parameter $n \in \mathbb{N}$ bounds size of every variable type $T[n]$.
    - May have different bounds for different types.
  - 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.

Verification 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.
    - 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 $n \in \mathbf{N}$.

1. Formal Specification

2. Proof-Based Verification

3. Semantics-Based Checking

4. Conclusions

Conclusions

- RISC Algorithm Language released in 2017.
  - Since then used in formal method courses
  - Contents developed by students (computer science, discrete mathematics, number theory, computer algebra, ...).
  - Very positive initial feedback due to “full automation”.
  - Visualization component recently added.
  - Current work on generation of verification conditions.
  - Further development within LOGTECHEDU and SEMTECH.
    - LOGTECHEDU: Logic Technology for Computer Science Education, LIT seed project, 2017–2019.
    - SEMTECH: Austrian OEAD WTZ and the Slovak SRDA project SK 14/2018, 2018–2019.

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