The RISC ProgramExplorer Second Status Report

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.jku.at

Goals

An integrated program reasoning environment that provides insight into the semantic essence of a program.

- Is based on the concept of programs as state relations.
  - A program implements a relation on states.
  - A specification describes a relation on states.
  - The program relation must imply the specification relation.
- Addresses various semantic questions.
  - Is a specification satisfiable and not trivial?
  - What is the state relation described by a command/method?
  - What state condition is known at a particular program point?
  - Are methods only called in states that satisfy the methods' preconditions?
  - Does the method meet its specification (assuming that loop invariants hold and termination terms are appropriate)?
  - Do the invariants indeed hold?
  - Are the termination terms indeed appropriate?
- Provides a state-of-the-art graphical user interface.
  - Tight links between syntactic source code and semantic essence.
  - Helps to gain insight as much as possible.

Program Calculus

- Hoare Calculus: $\{\mathtt{x} = a\}\mathtt{x=x*x}\{\mathtt{x} = a^2\}$
  - Pair of *state conditions* "glued together" by a logical constant a.
  - Reasoning based on Hoare triples that mix program and logic.
- Dynamic Logic: $\forall a : \mathtt{x} = a \Rightarrow [\mathtt{x=x*x}]\mathtt{x} = a^2$
  - Two *state conditions* separated by a modality $[\mathtt{x=x*x}]$.
  - Reasoning based on modal formulas that mix program and logic.
- Relational Calculus: $\mathtt{x=x*x}$: $\mathtt{x}' = \mathtt{x}^2$
  - Single *state relation* $\mathtt{x}' = \mathtt{x}^2$.
    - Captures the (denotational) semantics of the command.
  - Reasoning based on classical logic.
    - The command is translated into a classical logical formula.
    - All further reasoning about the command is based on the formula.

Our approach is to use the relational calculus to give programmers *insight*.

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   Fw   Fv   Fe   Fc
      i = i+1;            F5 }
   }^{F,T}
}
```

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

$T := \text{var } n \text{ - var } i + 1$

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

Translation into a formula that captures the program's semantic essence.

Abstract Framework

Formal syntax and semantics of various languages.

- An abstract imperative programming language.
  - Commands operating on states.

    `=`, `var`, `if`, `while`, `continue`, `break`, `return`, `throw`, `try`.
  - Methods with results, (direct and indirect) recursion.
- An abstract logic formula language.
  - Predicate logic formulas with functions and predicates on states.
- A program specification language based on the formula language.
  - Assertions, loop invariants, termination terms.
  - Method specifications with preconditions, postconditions, frame conditions, exception conditions, recursion measures.

The formal reasoning calculus was elaborated and its soundness was proved within this framework.

Concrete Programming Language

A subset of Java ("MiniJava") that can be mapped to the abstract programming language in a rather straight-forward way.

- Classes as modules with class variables and class methods.
  - Treatment as global variables and methods of the basic calculus.
- Classes as types with object variables, constructors, object methods.
  - Object functions receive the `this` object as an additional argument and return it as an additional result.
- Value semantics for arrays and objects.

  Type checker prevents aliasing (i.e. that different variables refer to same object) and thus hides difference to reference semantics.
  - Assignment to variable only from a constructor call.
  - Return as function result only from locally owned object.
  - Passing as an argument only from a constructor call or from a local variable that does not appear as another argument.
  - No (directly or indirectly) recursive class references.

Classes as modules and types, no inheritance, no reference semantics.

Example

```
class Record {
  String key; int value;
  Record(String k, int v)  { key = k; value = v; }
  boolean equals(String k) { boolean e = key.equals(k); return e; }

  public static int search(Record[] a, String key) {
    int n = a.length;
    for (int i=0; i<n; i++) {
      Record r = new Record(a[i].key, a[i].value); // copy of a[i]
      boolean e = r.equals(key);                   // a[i].equals(key) illegal
      if (e) return i;
    }
    return -1;
  }

  public static void main() {
    Record[] a = new Record[10];
    for (int i=0; i<10; i++) a[i] = new Record("abc", i);
    int i = search(a, "abc");
    System.out.println(i);
  }
}
```

Concrete Specification Language

- Typed higher-order predicate logic.
  - ProofNavigator syntax (inherited from CVS/PVS).

    ```
    FORALL(i:INT): 0 <= i AND i < n => a0[i].key /= k0
    ```
- Program variables.
  - $x \rightsquigarrow$ `OLD x`, $x' \rightsquigarrow$ `VAR x`.
- State types, constants functions, predicates.
  - `STATE(`*T*`)`, `NOW`, `NEXT`, `EXECUTES@`*s*, `VALUE@`*s*, ...
- Method specifications
  - `requires` ... `assignable` ... `signals` ... `ensures` ... `decreases`
- Code annotations
  - Loops: `invariant` ... `decreases` ...
  - Statements: `assert` ...

Tradition of JML et al, extended by an explicit notion of states.

Example

```
public static int search(Record[] a, String key) /*@
  requires var a /= Record.nullArray;
  ensures
    (LET result=VALUE@NEXT, a0=VAR a, n=Record.length(a0), k0=VAR key IN
      IF result = -1 THEN
        FORALL(i:INT): 0 <= i AND i < n => a0[i].key /= k0
      ELSE
        0 <= result AND result < n AND a0[result].key = k0
      ENDIF);
@*/
{
  int n = a.length;
  for (int i=0; i<n; i++)
  {
    Record r = new Record(a[i].key, a[i].value);
    boolean e = r.equals(key);
    if (e) return i;
  }
}
```

Theories

- Automatically generated theories.
  - `theory Base`
    - MiniJava types and operations.
  - `class` C $\rightsquigarrow$ `theory` C.
    - Classes as records.
- Named theories (user-defined).
  - File *Theory*`.theory`.
    - Abstract datatypes etc.
- Local theories (user-defined).
  - `/*@ theory { ...} @*/ class` C
    - Local definitions inside a class.

Building blocks for specifications.

Example

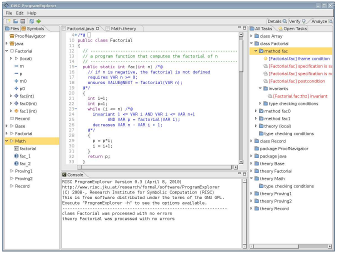
```
theory Record uses java.lang.String, Base { // generated from class Record
  Record: TYPE = [#key: java.lang.String.String, value: Base.int#];
  null: Record; nullArray: ARRAY Base.int OF Record;
  length: (ARRAY Base.int OF Record) -> Base.nat;
}

theory Stack {                                  // file Stack.theory
  Elem: TYPE = INT; Stack: TYPE;
  empty: Stack; cons: (Elem, Stack) -> Stack;
  isempty: PREDICATE(Stack);
  IE: AXIOM FORALL(s: Stack): isempty(s) <=> s=empty;
}

/*@
theory uses Record, java.lang.String {          // file Record.java
  Record: TYPE = Record.Record;
  String: TYPE = java.lang.String.String;
  notFound: PREDICATE(ARRAY INT OF Record, INT, STRING) =
    PRED(a:ARRAY INT OF Record, i:INT, key: String):
      (FORALL(i:INT): 0 <= i AND i < Record.length(a) => a[i].key /= key);
} @*/
class Record {...}
```

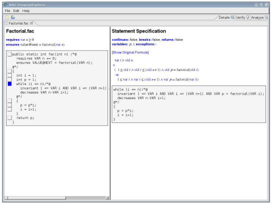
The Software

- Integrated environment built on top of the Eclipse SWT.
  - Provides graphical user interface and editing framework.
- Analyze view.
  - Console.
    - Plain text output.
  - Source code editor.
    - Syntax highlighting, specification text folding, error annotations. active identifiers.
  - Files/Symbols and Tasks/Open tasks.
    - Symbols and tasks linked to source.
- Verify view.
  - Embeds the RISC ProofNavigator.
- Details view.
  - State relations of method bodies.



Internal Operation

Constructs/maintains the internal model of the program/specification.

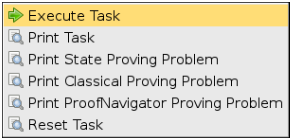


- Annotated abstract syntax trees.
  - Nodes linked to source code positions.
  - Identifiers linked to symbols.
  - Terms linked to types.
  - State relations linked to commands.
- Symbol tables.
  - Collections of symbols introduced in same scope.
  - Symbols linked to abstract syntax tree nodes.
- Verification tasks.
  - Organized in nested folders, linked to abstract syntax tree nodes.
  - Currently: type-checking tasks, specification tasks, frame condition tasks, postcondition tasks, invariant tasks.
  - Missing: precondition tasks, termination-related tasks.

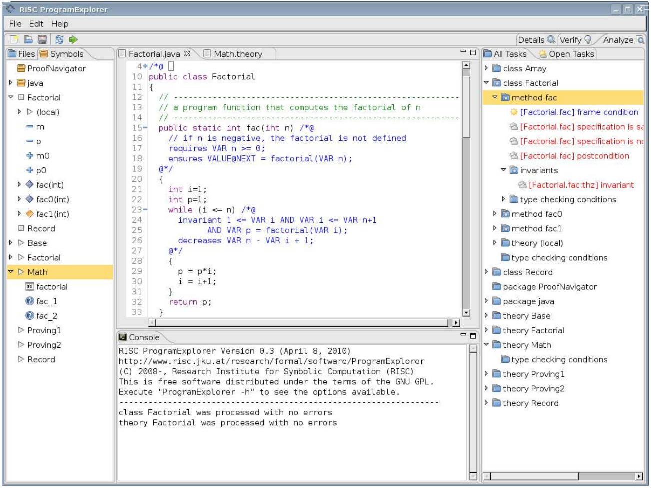
Task Management

Framework for generation and maintenance of tasks.

- Tasks organized in nested folders.
  - Corresponding to source code structure.
  - Linked to source code positions.
- Strategies may be associated to tasks.
  - Currently: automatic decision by CVCL and manual verification.
- Tasks may be translated to proving problems.
  - E.g. postcondition task → state logic problem → classical logic problem → RISC ProofNavigator problem.
    - Translation to logical problem on program variables/states, translation of program variables to mathematical constants considering the problem frame, translation to ProofNavigator format.
- Proofs are persistent.
  - Stored in RISC ProofNavigator format.
  - Reused in new RISC ProgramExplorer invocations.
  - RISC ProofNavigator dependence control maintains trust status.



Demonstration



Current State and Further Work

- Software in alpha2 status.
  - Reasonably stable (tested with toy examples only).
  - Classes: ca. 120 ProgramExplorer, 100 ProofNavigator, 300 syntax.
  - Lines of code: about 130K with comments (perhaps 60-70K without).
- Website and user manual.
  - Still presenting the alpha1 status (April 2010).
- Further work:
  - Precondition/assertion checking.
  - Forward/backward propagation of conditions.
  - Termination calculus.
  - Checking loop termination terms and recursive method measures.

First functionality-complete prototype expected by summer 2011.