8. Program Verification

8.4 Program verification with Isabelle/HOL

Extension: IMP plus arrays

Case study:

To illustrate a more realistic example, we

- extend IMP by arrays
- extend the Hoare logic appropriately
- verify the splitting step according to the program and specification
  given in the introduction

See » HoareIMParray.thy

8.5 Software verification tools

Considered techniques and tools:

- Embedding programming into HO interactive theorem proving (e.g., theory Simpl.thy written in Isabelle/HOL)
- Tools for interactive software verification (e.g., KeY, Why)
- Extended static checking (e.g., Spec#, Chalice, VeriFast)
- Specification and refinement (e.g., Event B, KIV)
- Automated theorem proving, in particular:
  - Superposition provers (e.g., SPASS, E)
  - SMT solvers and model checkers (e.g., Z3, SPIN)
8. Program Verification 8.5 Software verification tools

General approach

- Specify programming language syntax and semantics in an interactive theorem prover for HOL
- Use HOL as specification language for program properties
- Use the HOL proof system for verification

Example: Verifying Simpl programs

Simpl (by N. Schirmer [Archives of formal proofs]):
A sequential imperative programming language:
- mutually recursive procedures
- abrupt termination and exceptions
- runtime faults
- local and global variables
- pointers and heaps
- expressions with side-effects
- pointers to procedures
- partial application and closures
- dynamic method invocation
- unbounded nondeterminism

Example program: Quicksort on heap lists

Illustrating:
- Handling of programming language aspects:
  - recursive procedures
  - local and global variables
  - pointers and heaps
- Handling of specification aspects:
  - heap-manipulation using abstraction
  - frame properties
- Problems of embedding

Modeling of a program specific state

Heap for singly-linked lists (sll-heaps):

record globals_heap =
  next_' :: "ref ⇒ ref"
  cont_' :: "ref ⇒ nat"

A predicate to abstract sll-heaps:

primrec List :: ref ⇒ (ref ⇒ ref) ⇒ ref list ⇒ bool
  where
  List x h [] = (x = Null) |
  List x h (p#ps) = (x = p ∧ x ≠ Null ∧ List (h x) h ps)
Modeling of a program specific state (2)

The variables for procedures `append` and `quickSort`

```plaintext
record 'g vars = "'g state" +
  p_ :: "ref"
  q_ :: "ref"
  le_ :: "ref"
  gt_ :: "ref"
  hd_ :: "ref"
  tl_ :: "ref"
```

Implementation and specification of procedure `quickSort`

```plaintext
procedures quickSort(p|p) =
  "IF p=Null THEN SKIP
  ELSE tl := p→next;; le := Null;; gt := Null;;
      WHILE tl≠Null DO
        hd := tl;; tl := tl→next;;
        IF hd→cont ≤ p→cont
           THEN hd→next := le;; le := hd
           ELSE hd→next := gt;; gt := hd
        FI
      OD;;
    le := CALL quickSort(le);;
    gt := CALL quickSort(gt);;
    p→next := gt;;
    le := CALL append(le,p);
    p := le
  FI"
```

Specification of procedure `quickSort`

```plaintext
quickSort_spec:
  "∀ σ Ps. Γ ⊢
    { σ. List p→next Ps ∧ List q→next Qs ∧
      set Ps ∩ set Qs = {} }
    p := PROC append(p,q)
    { ∃ sortedPs. List p→next (Ps@Qs) ∧
      sorted (op ≤) (map "cont sortedPs") ∧
      Ps <~~> sortedPs } ∧
    (∀ x. x∉set Ps → next x = "next x )}"
```

Implementation and specification of procedure `append`

```plaintext
procedures append(p,q|p) =
  "IF p=Null THEN p :== q
  ELSE p→next := CALL append(p→next,q) FI"
```

```plaintext
append_spec:
  "∀ σ Ps Qs. Γ ⊢
    { σ. List p→next Ps ∧ List q→next Qs ∧
      set Ps ∩ set Qs = {} }
    p := PROC append(p,q)
    { List p→next (Ps@Qs) ∧
      (∀ x. x∉set Ps → next x = "next x )}"
```

```plaintext
append_modifies:
  "∀ σ. Γ ⊢
    { t. t may_only_modify_globals σ in [next] }"
```

Implementation of procedure `quickSort`

```plaintext
procedures quickSort(p|p) =
  "IF p=Null THEN SKIP
  ELSE tl := p→next;;
      hd := tl;; tl := tl→next;;
      WHILE tl≠Null DO
        hd := tl;; tl := tl→next;;
        IF hd→cont ≤ p→cont
           THEN hd→next := le;; le := hd
           ELSE hd→next := gt;; gt := hd
        FI
      OD;;
    le := CALL quickSort(le);;
    gt := CALL quickSort(gt);;
    p→next := gt;;
    le := CALL append(le,p);
    p := le
  FI"
```
Discussion of embedding approach

• Advantages:
  ▶ powerful specification language and reasoning support
  ▶ flexible for experimenting with languages
  ▶ meta-logical aspects can be handled

• Disadvantages:
  ▶ handling (realistic) programs can be a bit cumbersome
  ▶ realizing convenient IDEs expensive

Tools for interactive software verification

Example systems

• KeY: http://www.key-project.org/
  ▶ programming language: JavaCard; specifications in JML
  ▶ based on an interactive prover for dynamic logic
  ▶ makes extensive use of symbolic evaluation

• Why, Why3: http://why.lri.fr/
  ▶ programming languages: Java subset, C subset
  ▶ specific specification languages
  ▶ general-purpose verification condition generator
  ▶ uses many interactive and automated provers