Customised Induction Rules for Proving Correctness of Imperative Programs

Angela Wallenburg

angelaw@cs.chalmers.se

CHALMERS | GÖTEBORG UNIVERSITY

4th International KeY Symposium

June 9, 2005, Lökeberg

Outline

1. Problem: Induction and Loops
2. First approach: Use idea from software testing to create induction rules
3. Next approach: Use KeY to customise the rules instead and tie up loose ends
4. Ongoing work: Rippling – can it be used for the remaining challenges?

Problems in Semi-Interactive Theorem Proving

1. Level of automation (a lot of user-interaction)
2. User-interaction complicated

Loops present the real challenge.

- Induction used to prove loops in KeY
- Induction hypothesis, required by the user
- Can be rather complicated, everything at once
- Recursion, similar problems

This holds for KeY!

Motivating Example

Proof obligation: $\forall i \in \mathbb{N} \cdot \varphi(i)$,

where $\varphi(i)$:

$$\forall c \in \mathbb{N} \cdot i \geq 0 \wedge c \geq 1 \rightarrow \langle \textbf{while } (\mathsf{i} > 0) \{ \textbf{if } (\mathsf{i} >= \mathsf{c}) \{ \mathsf{i} = \mathsf{i} - \mathsf{c}; \} \textbf{ else } \{ \mathsf{i}{-}{-}; \} \} \rangle\ i = 0$$

Motivating Example

Standard induction step: $\forall n \in \mathbb{N} \cdot \varphi(n) \rightarrow \varphi(n+1)$

- Symbolic execution
- Unwind loop
- Two branches:

$$\begin{array}{cc} (1) & (2) \\ i := i - c; & i - -; \\ \forall n \in \mathbb{N} \cdot \varphi(n) \wedge n \geq c \rightarrow \varphi(n+1-c) & \forall n \in \mathbb{N} \cdot \varphi(n) \wedge n < c \rightarrow \varphi(n) \end{array}$$

Problem!

Goal

- Derive induction rule
- Automatically
- Program-specific induction rule
- Minimise *user-interaction*, not necessarily interested in proof-strength

First Approach - Partition Testing as an Inspiration

- Using technique from software testing: *partitioning*
- Divide and Conquer!
- Partition analysis can be performed *automatically*
- White-box partition analysis using branch predicates
- Partition the proof!

Example

```
int russianMultiplication(int a,int b) {
   int z = 0;
   while (a != 0) {
      if (a mod 2 != 0) {
           z = z + b;
      }
      a = a/2;
      b = b*2;
   }
   return z;
}
```

Example Partition

Partition of domain of a ($\mathbb{N}$), based on the branch predicates:

$$
\begin{aligned}
D_1 &= \{x \in \mathbb{N} \,|\, x = 0\} = \{0\} \\
D_2 &= \{x \in \mathbb{N} \,|\, x \neq 0 \wedge x \bmod 2 \neq 0\} \\
D_3 &= \{x \in \mathbb{N} \,|\, x \neq 0 \wedge x \bmod 2 = 0\}
\end{aligned}
$$

Overview of the method

1. **Construct partition of induction variable's domain**
   - using branch predicates
   - automatically
2. **Refine the partition**
   - using implicit case distinctions of operators
   - desired format
3. **Create new induction rule**
   - based on refined partition
   - k base cases, matching finite subdomains
   - l step cases, matching infinite subdomains
4. Hopefully less user-interaction required

Method by Example

The partitioned induction rule

$$\varphi(0) \tag{1}$$
$$\forall n \in \mathbb{N}_1 \cdot \varphi(n) \rightarrow \varphi(2 * n) \tag{2}$$
$$\forall n \in \mathbb{N} \cdot \varphi(n) \rightarrow \varphi(2 * n + 1) \tag{3}$$

to prove $\forall n \in \mathbb{N} \cdot \varphi(n)$

Resulting User Interaction

User interaction required with **partitioned induction rule**:

- Instantiation
- Induction rule application
- Unwinding of the loop
- Decision procedure
- Arithmetic

Next Approach – Generate Partitions with KeY

Problems with the approach described so far:

- Branch predicates might not be related to the update of the induction variable – resulting induction rule provides no simplification!
- Relies on quite sophisticated refinement of the partitions.

Rather we would like to:

- Let the side effects on the induction variable performed inside loop decide the induction steps.
- Use *failed proof attempts* and *updates*!

Generate Partitions Using a Theorem Prover

The productive use of failure:

- perform an attempt at proving the loop
- get stuck
- figure out why
- use this when starting over

Use the machinery of semi-automatic theorem prover KeY, in particular the *updates*, to do this.

Example of a Failed Proof Attempt with Update

$$
\begin{array}{l}
\vdash \forall il \in \mathbb{Z} \cdot il \geq 0 \rightarrow \\
\quad \{\mathsf{i} := il\} \\
\quad \langle \texttt{while (i > 0) \{} \\
\qquad \texttt{i = i - 2;} \\
\quad \texttt{\}} \rangle\, i = 0 \lor i = -1
\end{array}
$$

Stuck after unwinding of the loop:

$$
\begin{array}{l}
il_c > 0 \\
\vdash \\
\{\mathsf{i} := il_c - 2\} \\
\langle \texttt{while (i > 0) \{} \\
\qquad \texttt{i = i - 2;} \\
\quad \texttt{\}} \rangle\, i = 0 \lor i = -1
\end{array}
$$

Destructor Style Induction

- Avoid inverting functions during creation of induction step
- Use "predecessor functions", starting "one step earlier"
- Process of proving still the same: unwind right-hand side to attain syntactic equivalence
- Computations only performed in the forwards direction

$$\frac{\Gamma \vdash \forall i \in \mathbb{D}_b \cdot \varphi(i) \quad \Gamma \vdash \forall i \in \mathbb{D}_s \cdot \varphi(p(i)) \rightarrow \varphi(i)}{\Gamma \vdash \forall i \in \mathbb{N} \cdot \varphi(i)}$$

Example Constructor versus Destructor Style Induction

Induction rule for previous example, in *constructor* style:

$$\frac{\Gamma \ \vdash \ \forall i \in \mathbb{D}_b \cdot \varphi(i) \quad \Gamma \ \vdash \ \forall i \in \mathbb{D}_s \cdot \varphi(i) \rightarrow \varphi(i+2)}{\Gamma \ \vdash \ \forall i \in \mathbb{Z} \cdot \varphi(i)}$$

and in *destructor* style:

$$\frac{\Gamma \ \vdash \ \forall i \in \mathbb{D}_b \cdot \varphi(i) \quad \Gamma \ \vdash \ \forall i \in \mathbb{D}_s \cdot \varphi(i-2) \rightarrow \varphi(i)}{\Gamma \ \vdash \ \forall i \in \mathbb{Z} \cdot \varphi(i)}$$

Soundness

Customised induction rule so far:

$$\frac{\begin{array}{c}\Gamma \ \vdash \ \forall i \cdot BC(i) \rightarrow \varphi(i) \\ \Gamma \ \vdash \ \forall i \cdot BP_1(i) \wedge \varphi(p_1(i)) \rightarrow \varphi(i) \quad \ldots \quad \Gamma \ \vdash \ \forall i \cdot BP_n(i) \wedge \varphi(p_n(i)) \rightarrow \varphi(i)\end{array}}{\Gamma \ \vdash \ \forall i \cdot \varphi(i)} \tag{4}$$

where $BC(i) \leftrightarrow \neg BP_1(i) \wedge \ldots \wedge \neg BP_n(i)$.

Noetherian induction: proving

$$\forall m \in M \cdot (\forall k \in M \cdot k \prec_M m \rightarrow \varphi(k)) \rightarrow \varphi(m) \tag{5}$$

and that $(M, \prec_M)$ is a well-founded set, together with the well-founded induction principle means that we have verified $\forall m \in M \cdot \varphi(m)$.

Soundness (ii)

To ensure well-foundedness of the induction set we need some extra proof obligations:

- Allow only predecessor functions that decrease the argument:

$$(\forall i \cdot BP_1(i) \rightarrow p_1(i) < i) \wedge \ldots \wedge (\forall i \cdot BP_n(i) \rightarrow p_n(i) < i) \wedge \\ \forall i, j \cdot BC(i) \wedge \neg BC(j) \rightarrow i < j \tag{6}$$

- Make sure there exists some element in the domain of the base case:

$$\exists i \cdot BC(i) \tag{7}$$

The Customised Induction Rule

Now this rule is sound (proof in thesis):

$$\frac{\begin{array}{c} \Gamma \vdash \forall i \cdot BC(i) \rightarrow \varphi(i) \\ \Gamma \vdash \forall i \cdot BP_1(i) \wedge \varphi(p_1(i)) \rightarrow \varphi(i) \quad \ldots \quad \Gamma \vdash \forall i \cdot BP_n(i) \wedge \varphi(p_n(i)) \rightarrow \varphi(i) \\ \Gamma \vdash \begin{array}{l} (\forall i \cdot \bigwedge_{k=1\ldots n} BP_k(i) \rightarrow p_k(i) < i) \wedge \forall i, j \cdot BC(i) \wedge \neg BC(j) \rightarrow i < j \vee \\ (\forall i \cdot \bigwedge_{k=1\ldots n} BP_k(i) \rightarrow p_k(i) > i) \wedge \forall i, j \cdot BC(i) \wedge \neg BC(j) \rightarrow i > j \end{array} \\ \Gamma \vdash \exists i \cdot BC(i) \end{array}}{\Gamma \vdash \forall i \cdot \varphi(i)} \tag{8}$$

Russian Multiplication Example Revisited

$$\frac{\begin{array}{c}\Gamma \vdash \forall i \cdot i \leq 0 \rightarrow \varphi(i) \\ \Gamma \vdash \forall i \cdot i > 0 \wedge i \bmod 2 \neq 0 \wedge \varphi(i/2) \rightarrow \varphi(i) \\ \Gamma \vdash \forall i \cdot i > 0 \wedge i \bmod 2 = 0 \wedge \varphi(i/2) \rightarrow \varphi(i) \\ \Gamma \vdash \begin{array}{l}((\forall i \cdot (i > 0 \wedge i \bmod 2 \neq 0 \rightarrow i/2 < i) \wedge \\ (i > 0 \wedge i \bmod 2 = 0 \rightarrow i/2 < i)) \wedge \forall i, j \cdot i \leq 0 \wedge \neg j \leq 0 \rightarrow i < j) \vee \\ ((\forall i \cdot (i > 0 \wedge i \bmod 2 \neq 0 \rightarrow i/2 > i) \wedge \\ (i > 0 \wedge i \bmod 2 = 0 \rightarrow i/2 > i)) \wedge \forall i, j \cdot i \leq 0 \wedge \neg j \leq 0 \rightarrow i > j)\end{array} \\ \Gamma \vdash \exists i \cdot i \leq 0\end{array}}{\Gamma \vdash \forall i \cdot \varphi(i)}$$

Comparison to Noetherian Induction

Differences mainly in usability and interaction requirements, not proof-strength

- WFI introduces only one new proof branch – at least four for PI
- a failed proof attempt in PI is *much* easier to debug
  - PI separates the different concerns of the proof
  - PI "knows" more about the problem, presents the branches "up-front"
- base case is separated in PI, implicit in WFI.
- WFI beyond PI in application domain
- additional well-foundedness-proof-obligations in PI

Customised Induction Rules – Summary

- automatic creation of customised induction rules for proving the total correctness of loops
- the resulting rules are
  - tailor-made for the respective loops to be verified
  - sound
- in comparison to Peano induction or Noetherian induction, the customised induction rules significantly simplify the user interaction required
- using a customised induction rule, the resulting proof becomes more modularised
- a shift of focus for the user interacting with the prover

Customised Induction Rules – Summary

- limitations and future work
  - Other data structures: so far only integers, extend with lists, trees
  - Nested loops and multiple induction variables
  - Expression simplification
  - Partial correctness/box modality
  - Separate termination analysis
  - Hybrid with Noetherian induction
  - Generalisation of post-conditions
  - Towards full automation...

Induction Proving Process

1. Apply strategy (without unwinding of loops).

2. Decide induction variable. Look at the termination condition.

3. Decide which kind of induction rule to use. Look at the update to the induction variable inside the loop.

4. Induction hypothesis. Start with the proof obligation.

5. Apply the induction rule. Apply strategy and Simplify.
   - Use case: a lot of instantiations should do the trick.
   - Base/Step cases: Unwind loop in subsequent. Instantiations, arithmetic.

6. Generalise the induction hypothesis, if needed. It is the updates and the postconditions that have to be changed, the program will stay the same.

Cubic Sum Example

```
i=0;
r=0;
while (i < n) {
   i++;
   r = r + (i*i*i);
}
```

Precondition $n \geq 0$

Postcondition $4 * r = nl^2 * (nl + 1)^2$

Cubic Sum Example

Induction variable: new variable kl $(nl - il)$. Generalised induction hypothesis:

```
all nl:int.(all rl:int.(
      (geq(nl, 0) & geq(rl, 0) & geq(kl, 0) & geq(nl, kl)) ->
           {i:=+(nl,~m(kl)),
            n:=nl,
            r:=rl}
            <{
               while ( i<n ) {
                 i++;
                 r=r+(i*i*i);
               }
             }>   mul(4, +(r,~m(rl)))
                = +((mul(mul(mul(nl,nl),+(nl,1)),+(nl,1))),
                   ~m(mul(mul(mul(+(nl,~m(kl)),+(nl,~m(kl))),
                     +(+(nl,~m(kl)),1)),+(+(nl,~m(kl)),1)))) ))
```

Cubic Sum Example

Original postcondition:

$4 * r = nl^2 * (nl + 1)^2$

Generalised postcondition:

$4 * (r - rl) = nl^2 * (nl + 1)^2 - (nl - kl)^2 * (nl - kl + 1)^2$

Generalisation can be non-trivial!

Rippling

In general:

- technique to annotate formulas, colouring
- restrict rewriting rules, wave rules
- allow only rewrites that make the conjecture similar to lemma or hypothesis
- originates from Bundy, Ireland etc, functional programming

Rippling

In particular:

- useful for proving the induction step
- can be used together with "productive use of failure" approach
- creating induction rules
- generalising induction formula
- currently investigating
- translate concept of rippling to KeY

The End

Thanks.

Related Work

Combinig testing and proving

- FATES
- Z, B, VDM, ASML, Haskell...
- Partition testing, Howden 76 etc.
- Using testing to aid in proving Geller 78
- Dynamic analysis, generating invariants, Nimmer et al.
- Avoiding failed proof attempts, Qiao

Related Work

Mechanizing induction proving

- Explicit induction
- Implicit induction
- Walther, Bundy, Boyer and Moore, see refs thesis
- Generating induction schemas, Slind
- Cyclic reasoning, Sprenger and Dam

Related Work

Simplifying user interaction

- Automtic generation of loop invariants, Kapur et el.
- Rippling, Bundy
- B method
- ACL2, industrial strength theorem prover