

# Using Induction in Interactive Verification

**Angela Wallenburg**

`angelaw@cs.chalmers.se`

University of Koblenz-Landau

ESF Workshop

Nijmegen, October 17 2006

# Problems in Semi-Interactive Theorem Proving

1. Level of automation
2. User-interaction complicated
3. Difficult to debug a failed proof attempt

This holds for the theorem prover in  – a system with the outspoken aim to be software engineer-friendly!

# Problems in Semi-Interactive Theorem Proving

**Loops** present the real challenge.

- **Induction** is used to prove loops in KeY
- **Induction variable** and **formula**, supplied by the user
- Can be very complicated
- **Poor modularisation** of proofs (and failed proof attempts)
- Recursion, similar problems

# Simple (!?) Induction

(first order) Peano induction rule:

$$\frac{\vdash \varphi(0) \quad \vdash \forall n \in \mathbb{N} \cdot \varphi(n) \rightarrow \varphi(n+1)}{\vdash \forall n \in \mathbb{N} \cdot \varphi(n)}$$

- Induction variable is  $n$ .
- Induction formula is  $\varphi(n)$ .
  - Induction hypothesis.
  - Induction conclusion.

# Example

Russian multiplication in JAVA CARD DL:

$$\varphi(a_0) \leftrightarrow$$

$$\forall b_0 \in \mathbb{Z} \cdot a_0 \geq 0 \rightarrow$$

```
{a := a_0}{b := b_0}{z := 0}⟨ while (a != 0){  
    if (a % 2 != 0){  
        z = z + b;  
    }  
    a = a / 2;  
    b = b * 2;  
}⟩ z = a_0 * b_0
```

Prove  $\forall a \in \mathbb{Z} \cdot \varphi(a)$ .

# Naive Proof Attempt

$\forall bl \in int.$

```
{a := alc, b := bl, z := 0}
⟨ while (a != 0) {
    if (a % 2 != 0) z = z + b;
    a = a / 2;
    b = b * 2;
} ⟩ z = alc * bl
```

⊢

```
{a := alc + 1, b := blc, z := 0}
⟨ while (a != 0) {
    if (a % 2 != 0) z = z + b;
    a = a / 2;
    b = b * 2;
} ⟩ z = (alc + 1) * blc
```

Step case:

- $\forall a \in \mathbb{Z} \cdot \varphi(a) \rightarrow \varphi(a + 1)$  has become  $\varphi(al_c) \rightarrow \varphi(al_c + 1)$
- $al_c$  is a skolem constant
- $bl_c$  is a skolem constant (succedent only).

# Naive Proof Attempt

$\forall bl \in int.$

```
{a := alc, b := bl, z := 0}
⟨ while (a != 0) {
    if (a % 2 != 0) z = z + b;
    a = a / 2;
    b = b * 2;
} ⟩ z = alc * bl
```

⊢

```
{a := alc + 1, b := blc, z := 0}
⟨ while (a != 0) {
    if (a % 2 != 0) z = z + b;
    a = a / 2;
    b = b * 2;
} ⟩ z = (alc + 1) * blc
```

Step case:

- $\forall a \in \mathbb{Z} \cdot \varphi(a) \rightarrow \varphi(a + 1)$  has become  $\varphi(al_c) \rightarrow \varphi(al_c + 1)$
- $al_c$  is a skolem constant
- $bl_c$  is a skolem constant (succedent only).

Proceed with proof attempt:

- Unwind loop in succedent (induction conclusion).
- Symbolic execution of loop body.

After symbolic execution of the loop body in the induction conclusion ( $al_c$  even):

$\forall bl \in int.$

```
{a := alc, b := bl, z := 0}
⟨ while (a != 0) {
    if (a % 2 != 0) z = z + b;
    a = a / 2;
    b = b * 2;
} ⟩ z = alc * bl
```

⊢

```
{a := alc + 1, b := blc, z := 0}
⟨ while (a != 0) {
    if (a % 2 != 0) z = z + b;
    a = a / 2;
    b = b * 2;
} ⟩ z = (alc + 1) * blc
```

$\forall bl \in int.$

```
{a := alc, b := bl, z := 0}
⟨ while (a != 0) {
    if (a % 2 != 0) z = z + b;
    a = a / 2;
    b = b * 2;
} ⟩ z = alc * bl
```

⊢

```
{a := (alc + 1)/2, b := 2 * blc, z := 0}
⟨ while (a != 0) {
    if (a % 2 != 0) z = z + b;
    a = a / 2;
    b = b * 2;
} ⟩ z = (alc + 1) * blc
```

## Naive Proof Attempt, continued

$\forall \mathbf{bl} \in \mathit{int}.$

$\{a := al_c, b := \mathbf{bl}, z := 0\}$

```
 $\langle$  while (a  $\neq$  0) {  
    if (a % 2  $\neq$  0) z = z + b;  
    a = a / 2;  
    b = b * 2;  
}  $\rangle$  z =  $al_c * \mathbf{bl}$ 
```

$\vdash$

$\{a := (al_c + 1)/2, b := \mathbf{2} * \mathbf{bl}_c, z := 0\}$

```
 $\langle$  while (a  $\neq$  0) {  
    if (a % 2  $\neq$  0) z = z + b;  
    a = a / 2;  
    b = b * 2;  
}  $\rangle$  z =  $(al_c + 1) * \mathbf{bl}_c$ 
```

Step case:

- Proceed with **instantiation** of  $\forall \mathbf{bl}$ .
- Supply  $2 * \mathbf{bl}_c$  for  $\mathbf{bl}$ .

After instantiation of  $bl$ :

$\forall \mathbf{bl} \in \mathit{int}$ .

$\{a := al_c, b := \mathbf{bl}, z := 0\}$

$\langle$  while (a  $\neq$  0) {  
     if (a % 2  $\neq$  0) z = z + b;  
     a = a / 2;  
     b = b \* 2;  
 }  $\rangle$  z =  $al_c * \mathbf{bl}$

$\vdash$

$\{a := (al_c + 1)/2, b := \mathbf{2} * \mathbf{bl}_c, z := 0\}$

$\langle$  while (a  $\neq$  0) {  
     if (a % 2  $\neq$  0) z = z + b;  
     a = a / 2;  
     b = b \* 2;  
 }  $\rangle$  z =  $(al_c + 1) * bl_c$

$\{a := al_c, b := \mathbf{2} * \mathbf{bl}_c, z := 0\}$

$\langle$  while (a  $\neq$  0) {  
     if (a % 2  $\neq$  0) z = z + b;  
     a = a / 2;  
     b = b \* 2;  
 }  $\rangle$  z =  $al_c * \mathbf{2} * \mathbf{bl}_c$

$\vdash$

$\{a := (al_c + 1)/2, b := \mathbf{2} * \mathbf{bl}_c, z := 0\}$

$\langle$  while (a  $\neq$  0) {  
     if (a % 2  $\neq$  0) z = z + b;  
     a = a / 2;  
     b = b \* 2;  
 }  $\rangle$  z =  $(al_c + 1) * bl_c$

## Naive Proof Attempt — Failed

$$\{a := \mathbf{al}_c, b := 2 * bl_c, z := 0\}$$
$$\langle \text{while } (a \neq 0) \{$$
$$\quad \text{if } (a \% 2 \neq 0) \ z = z + b;$$
$$\quad a = a / 2;$$
$$\quad b = b * 2;$$
$$\} \rangle z = \mathbf{al}_c * \mathbf{2} * \mathbf{bl}_c$$

⊢

$$\{a := (\mathbf{al}_c + 1)/\mathbf{2}, b := 2 * bl_c, z := 0\}$$
$$\langle \text{while } (a \neq 0) \{$$
$$\quad \text{if } (a \% 2 \neq 0) \ z = z + b;$$
$$\quad a = a / 2;$$
$$\quad b = b * 2;$$
$$\} \rangle z = (\mathbf{al}_c + 1) * \mathbf{bl}_c$$

# Observations

```
{a := alc, b := 2 * blc, z := 0}
⟨ while (a != 0) {
  if (a % 2 != 0) z = z + b;
  a = a / 2;
  b = b * 2;
} ⟩ z = alc * 2 * blc
```

⊢

```
{a := (alc + 1)/2, b := 2 * blc, z := 0}
⟨ while (a != 0) {
  if (a % 2 != 0) z = z + b;
  a = a / 2;
  b = b * 2;
} ⟩ z = (alc + 1) * blc
```

Why we are stuck.

- Mismatch in updates.
  - Update to induction variable by loop body is different from increment by the step case in induction rule.
- Mismatch in postcondition.
  - Instantiation of variables must achieve syntactic equivalence for the updates.

# Induction Proving Process

1. **Apply strategy** (without unwinding of loops).
2. Choose **induction variable**. See the termination condition.
3. Choose **induction rule**. See the update to the induction variable inside the loop.
4. Choose **induction formula**. Start with the proof obligation.
5. **Apply the induction rule**. Apply strategy and Simplify.
  - Use case: a lot of instantiations.
  - Base/Step cases: Unwind loop in subsequent. Instantiations, arithmetic.
6. **Generalise** the induction hypothesis, if needed. Updates and postconditions change, the program will stay the same.

# Goals

- Simplify user-interaction
- Minimise user-interaction
- **Simplify debugging of failed proof attempts**
  - proof attempts of valid formulae often fails
  - modularise proofs
- **Generalise induction formulae**
  - avoid generalisation if possible
  - assist when necessary
- Easy on the user!

# Using a Theorem Prover to Learn from Failed Attempts

*“The productive use of failure”*

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

Use the machinery of semi-automatic theorem prover [KEY](#), in particular the *updates*, to do this.

# Customised Induction Rules [Hähnle, Olsson and me]

Program:

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

Partition of induction variable:

$$D_1 = \{0\}$$

$$D_2 = \{1, 3, 5, \dots\}$$

$$D_2 = \{2, 4, 6, \dots\}$$

Customised induction rule:

$$\frac{\vdash \varphi(0) \quad \vdash \forall n \in \mathbb{N} \cdot \varphi(n) \rightarrow \varphi(2 * n) \quad \vdash \forall n \in \mathbb{N} \cdot \varphi(n) \rightarrow \varphi(2 * n + 1)}{\vdash \forall n \in \mathbb{N} \cdot \varphi(n)}$$

## Another Proof Attempt

$\forall bl \in int.$

```
{a := ac, b := bl, z := 0}
⟨ while (a != 0) {
    if (a % 2 != 0) z = z + b;
    a = a / 2;
    b = b * 2;
} ⟩ z = ac * bl
```

⊢

```
{a := 2 * ac + 1, b := blc, z := 0}
⟨ while (a != 0) {
    if (a % 2 != 0) z = z + b;
    a = a / 2;
    b = b * 2;
} ⟩ z = (2 * ac + 1) * blc
```

Step case, odd:

- $\forall a \in \mathbb{N} \cdot \varphi(a) \rightarrow \varphi(2 * a + 1)$
- Unwind loop in induction conclusion.
- Symbolic execution of loop body.

After symbolic execution of the loop body in the induction conclusion:

$\forall bl \in int.$

```
{a := alc, b := bl, z := 0}
⟨ while (a != 0) {
    if (a % 2 != 0) z = z + b;
    a = a / 2;
    b = b * 2;
} ⟩ z = alc * bl
```

⊢

```
{a := 2 * alc + 1, b := blc, z := 0}
⟨ while (a != 0) {
    if (a % 2 != 0) z = z + b;
    a = a / 2;
    b = b * 2;
} ⟩ z = (2 * alc + 1) * blc
```

$\forall bl \in int.$

```
{a := alc, b := bl, z := 0}
⟨ while (a != 0) {
    if (a % 2 != 0) z = z + b;
    a = a / 2;
    b = b * 2;
} ⟩ z = alc * bl
```

⊢

```
{a := (2 * alc + 1) / 2, b := 2 * blc, z := blc}
⟨ while (a != 0) {
    if (a % 2 != 0) z = z + b;
    a = a / 2;
    b = b * 2;
} ⟩ z = (2 * alc + 1) * blc
```

After instantiation of  $bl$ :

|  |   |
|--|---|
| $\forall \mathbf{bl} \in \mathit{int}.$ $\{a := al_c, b := \mathbf{bl}, z := 0\}$ $\langle \text{while } (a \neq 0) \{$ $\quad \text{if } (a \% 2 \neq 0) \ z = z + b;$ $\quad a = a / 2;$ $\quad b = b * 2;$ $\} \rangle z = al_c * \mathbf{bl}$ $\vdash$ $\{a := (2 * al_c + 1)/2, b := \mathbf{2} * bl_c,$ $\quad z := bl_c\}$ $\langle \text{while } (a \neq 0) \{$ $\quad \text{if } (a \% 2 \neq 0) \ z = z + b;$ $\quad a = a / 2;$ $\quad b = b * 2;$ $\} \rangle z = (2 * al_c + 1) * bl_c$ | $\{a := al_c, b := \mathbf{2} * bl_c, z := 0\}$ $\langle \text{while } (a \neq 0) \{$ $\quad \text{if } (a \% 2 \neq 0) \ z = z + b;$ $\quad a = a / 2;$ $\quad b = b * 2;$ $\} \rangle z = al_c * \mathbf{2} * bl_c$ $\vdash$ $\{a := al_c, b := 2 * bl_c, z := bl_c\}$ $\langle \text{while } (a \neq 0) \{$ $\quad \text{if } (a \% 2 \neq 0) \ z = z + b;$ $\quad a = a / 2;$ $\quad b = b * 2;$ $\} \rangle z = (2 * al_c + 1) * bl_c$ |
| <p>Failed!</p>   |   |

## More Observations

```
{a := alc, b := 2 * blc, z := 0}
⟨ while (a != 0) {
    if (a % 2 != 0) z = z + b;
    a = a / 2;
    b = b * 2;
} ⟩ z = alc * 2 * blc
```

⊢

```
{a := alc, b := 2 * blc, z := blc}
⟨ while (a != 0) {
    if (a % 2 != 0) z = z + b;
    a = a / 2;
    b = b * 2;
} ⟩ z = (2 * alc + 1) * blc
```

- Customised induction rules help, but only for the induction variable.
- Still problem with  $\{z := 0\}$  and  $\{z := bl_c\}$
- Solution: Generalise  $\varphi(al)$  over  $z$ !

# Induction Formulae Generalisation using

**Idea:** Mechanical approach, let the theorem prover work to help itself.

- Make use of failed proof attempt to **learn where to generalise**.
- **Generalise mechanically.**
  - introduce a new universally quantified variable to the syntactically mismatched update
  - introduce a higher order meta variable  $F$  that depends on the new variable and the post-condition (term).
- **Start over and learn** from a second failed proof attempt.
  - **generate constraints** from open proof goals.
  - **solve constraints** (using higher order unification).
- If you succeed, you have found a proper generalisation that you can prove!

# Generalisation Example

```
{a := alc, b := 2 * blc, z := 0}
⟨ while (a != 0) {
  if (a % 2 != 0) z = z + b;
  a = a / 2;
  b = b * 2;
} ⟩ z = alc * 2 * blc
```

⊢

```
{a := alc, b := 2 * blc, z := blc}
⟨ while (a != 0) {
  if (a % 2 != 0) z = z + b;
  a = a / 2;
  b = b * 2;
} ⟩ z = (2 * alc + 1) * blc
```

After first failed proof attempt:

- Updates not syntactically equivalent?
- If not induction variable — create naive generalisation.
  - introduce universally quantified variable  $z_l$ .
  - introduce higher order variable  $F$ .
- Construct a new induction formula:

# Generalisation Example

Original:

$$\begin{aligned} \varphi(al) &\leftrightarrow \\ \forall bl \in int. & \\ \{a := al, b := bl, z := 0\} & \\ \langle \text{while } (a \neq 0) \{ & \\ \quad \text{if } (a \% 2 \neq 0) \ z = z + b; & \\ \quad a = a / 2; & \\ \quad b = b * 2; & \\ \} \rangle z = al * bl & \end{aligned}$$

Generalisation:

$$\begin{aligned} \varphi(al) &\leftrightarrow \\ \forall bl, \mathbf{z1} \in int. & \\ \{a := al, b := bl, z := \mathbf{z1}\} & \\ \langle \text{while } (a \neq 0) \{ & \\ \quad \text{if } (a \% 2 \neq 0) \ z = z + b; & \\ \quad a = a / 2; & \\ \quad b = b * 2; & \\ \} \rangle z = \mathbf{F}(al * bl, \mathbf{z1}) & \end{aligned}$$

2nd failure, step case:

$$\{a := al_c, b := 2 * bl_c, z := zl_c + bl_c\}$$
$$\langle \text{while } (a \neq 0) \{$$
$$\quad \text{if } (a \% 2 \neq 0) \ z = z + b;$$
$$\quad a = a / 2;$$
$$\quad b = b * 2;$$
$$\} \rangle \ z = F(al_c * 2 * bl_c, zl_c + bl_c)$$

⊢

$$\{a := al_c, b := 2 * bl_c, z := zl_c + bl_c\}$$
$$\langle \text{while } (a \neq 0) \{$$
$$\quad \text{if } (a \% 2 \neq 0) \ z = z + b;$$
$$\quad a = a / 2;$$
$$\quad b = b * 2;$$
$$\} \rangle \ z = F((2 * al_c + 1) * bl_c, zl_c)$$

2nd failure, base case:

⊢

$$\{a := 0, b := bl_c, z := zl_c\}$$
$$\langle \text{while } (a \neq 0) \{$$
$$\quad \text{if } (a \% 2 \neq 0) \ z = z + b;$$
$$\quad a = a / 2;$$
$$\quad b = b * 2;$$
$$\} \rangle \ z = F(0 * bl_c, zl_c)$$

Constraints:

$$zl_c = F(0, zl_c),$$
$$F(al_c * 2 * bl_c, zl_c + bl_c) =$$
$$F((2 * al_c + 1) * bl_c, zl_c)$$

Solution:  $F$  is +.

# Resulting User Interaction

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

# Problems and Limitations

Oups.

- We have introduced higher concepts.
- KeY is not higher order.
- Higher order unification is undecidable.

and

- Still rather informal experiments by hand.
- Not clear how to introduce the higher order meta variables in the general case.

# Ways to Go

Some alternatives:

1. Introduce higher order variables and higher order unification into KeY.
2. Use external higher order unification algorithm, and just “learn” the generalisation if possible.
3. Use existing meta variables to be syntactic placeholders.

# Higher order pattern unification

- Higher order pattern is a term where the free (higher order) variables are only applied to  $\eta$ -equivalent bound variables.
- Decidable algorithm [Miller, Nipkow] finds the most general unifier.

Made experiments with this. Pros and cons:

- Fragment of the problem.
- Fragment of what we need.

## Challenge: Cubic Sum Example

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

Precondition  $n \geq 0$

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

# Cubic Sum Example

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

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!

KeY -- Prover

File View Proof Options Tools Help

Simple JavaCardDL without unwinding loops Autoresume Run Simplify Goal Back

Tasks

Env. with no model #1  
cubicSum.key

Proof Search Strategy Rules

Proof Goals User Constraint

Proof

```

308:int [r] =r;
309:int [r];
310:[r]=r;
311:Update Simplification
312:int [r] =(i*i);
313:int [r];
314:[r]=(i*i);
315:[r]=i*i;
316:int [i]=i*i;
317:int [i];
318:[i]=i*i;
319:Update Simplification
320:[r]=(i*i)*i;
321:Update Simplification
322:mul_assoc
323:r=[r]_1+[r];
324:Update Simplification
325:{}
326:inst_all
327:hide_left
328:imp_left
  Case 1
    329:not_right
    412:while (i<n) { i++; r=r
      413:OPEN GOAL
  Case 2
    330:minus_distribute
    331:hide_right
    410:hide_right
      411:OPEN GOAL

```

Current Goal


```

{i:=n1_0 + -k1_0,
 n:=n1_0,
 r:= r1_1
 + (1 + (-1 + (n1_0 + -k1_0)))
 * ((1 + (-1 + (n1_0 + -k1_0))) * (1 + (-1 + (n1_0 + -k1_0))))}
\<{
 while ( i<n ) {
 i++;
 r=r+(i*i*i);
 }
}\>
 * ( r
 + (-r1_1
 + -(1 + (-1 + (n1_0 + -k1_0)))
 * ((1 + (-1 + (n1_0 + -k1_0))) * (1 + (-1 + (n1_0 + -k1_0))))))
 = n1_0 * (n1_0 * ((1 + n1_0) * (1 + n1_0)))
 + -( (n1_0 + -k1_0)
 * ((n1_0 + -k1_0) * ((1 + (n1_0 + -k1_0)) * (1 + (n1_0 + -k1_0))))),
-1 + (n1_0 + -k1_0) < n1_0
=>
{i:=1 + (-1 + (n1_0 + -k1_0)),
 n:=n1_0,
 r:= r1_1
 + (1 + (-1 + (n1_0 + -k1_0)))
 * ((1 + (-1 + (n1_0 + -k1_0))) * (1 + (-1 + (n1_0 + -k1_0))))}
\<{
 while ( i<n ) {
 i++;
 r=r+(i*i*i);
 }
}\>
 n1_0 * (n1_0 * ((1 + n1_0) * (1 + n1_0)))
 + -( (-1 + (n1_0 + -k1_0))
 * ( (-1 + (n1_0 + -k1_0))
 * ((1 + (-1 + (n1_0 + -k1_0))) * (1 + (-1 + (n1_0 + -k1_0))))))
 = 4 * (r + -r1_1)

```

Integrated Deductive Software Design: Ready

# Related Work – Rippling

- In general
  - technique to annotate formulae, colouring
  - restrict rewriting rules, wave rules
  - allow only rewrites that make the conjecture similar to lemma or hypothesis
  - originates from Bundy, Basin, Ireland, Stark, Hutter
  - functional programming
- In particular
  - useful for proving the induction step
  - can be used together with “productive use of failure” approach
  - creating induction rules, Gow
  - generalising induction formula
  - investigated translating concept of rippling to 

**The End**

Thanks.