Software Engineering using Formal Methods

Reasoning about Programs with Loops and Method Calls

Wolfgang Ahrendt

15 October 2015

Program Logic Calculus – Repetition

Calculus realises symbolic interpreter:

- works on first active statement
- decomposition of complex statements into simpler ones
- simple assignments to updates
- accumulated update captures changed program state (abbr. w. $\mathcal{U}$)
- control flow branching induces proof splitting
- application of update computes weakest precondition of $\mathcal{U}'$ wrt. ϕ

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma' \Longrightarrow \{\mathcal{U}'\}\phi \qquad \ldots}{\ldots \qquad \ldots}}{\begin{array}{ll}\textit{`branch1'} & \Gamma, \{\mathcal{U}\}(\mathtt{isValid} = \mathtt{TRUE}) \Longrightarrow \{\mathcal{U}\}\langle\{\mathtt{ok{=}true;}\}\ldots\rangle\phi \\ \textit{`branch2'} & \Gamma, \{\mathcal{U}\}(\mathtt{isValid} = \mathtt{FALSE}) \Longrightarrow \{\mathcal{U}\}\langle\ldots\rangle\phi\end{array}}}{\Gamma \Longrightarrow \{\mathtt{t} := \mathtt{j} \| \mathtt{j} := \mathtt{j} + 1 \| \mathtt{i} := \mathtt{j}\}\{\mathcal{U}\}\{\mathcal{U}\}\langle\mathtt{if(isValid)\{ok{=}true;\}}\ldots\rangle\phi}}{\ldots}}{\Gamma \Longrightarrow \{\mathtt{t} := \mathtt{j}\}\langle\mathtt{j{=}j{+}1;i{=}t;if(isValid)\{ok{=}true;\}}\ldots\rangle\phi}}{\dfrac{\Gamma \Longrightarrow \langle\mathtt{t{=}j;j{=}j{+}1;i{=}t;if(isValid)\{ok{=}true;\}}\ldots\rangle\phi}{\Gamma \Longrightarrow \langle\mathtt{i{=}j{+}{+};if(isValid)\{ok{=}true;\}}\ldots\rangle\phi}}$$

An Example

```
\javaSource "src/";

\programVariables{
 Person p;
 int j;
}

\problem {
  (\forall int i;
    (!p=null ->
      ({j := i}\<{p.setAge(j);}\>(p.age = i))))
}
```

Method Calls

Method Call with actual parameters $arg_0, \ldots, arg_n$

$$\langle \pi \ \texttt{o.m}(arg_0, \ldots, arg_n); \ \ \omega \rangle \phi$$

where m declared as **void** $\texttt{m}(\tau_0\ \texttt{p}_0, \ldots, \tau_\texttt{n}\ \texttt{p}_\texttt{n})$

Actions of rule methodCall

1. For each formal parameter $\texttt{p}_\texttt{i}$ of m:
   declare and initialize new local variable τ_i p#i $=arg_i$;
2. Look up implementation class C of m and split proof
   if implementation cannot be uniquely determined
   (necessitated by dynamic dispatch in general)
3. Create statically resolved method invocation o.m(p#0, ..., p#n)@C

Method Calls Cont'd

Method Body Expand

1. Execute code that binds actual to formal parameters $\tau_{\mathtt{i}}$ `p#i` $=arg_i$;
2. Call rule methodBodyExpand

$$\frac{\Gamma \Longrightarrow \langle \pi \ \mathtt{method\text{-}frame(source=C,\ this=o)\{}\ \mathtt{body}\ \mathtt{\}}\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \pi \ \mathtt{o.m(p\#0,\ldots,p\#n)@C;}\ \omega\rangle\phi, \Delta}$$

   2.1 Rename p_i in body to `p#i`
   2.2 Replace method invocation by method frame and method body

Method frames:
Required in canculus to mirror call stack

Demo

```
methods/instanceMethodInlineSimple.key
methods/inlineDynamicDispatch.key
```

Localisation of Fields and Method Implementations

JAVA has complex rules for localisation of fields and method implementations

- Polymorphism
- Late binding (dynamic dispatch)
- Scoping (class vs. instance)
- Visibility (private, protected, public)

Proof split into cases when implementation not statically determined

Object initialization

JAVA has complex rules for object initialization

- Chain of constructor calls until Object
- Implicit calls to super()
- Visibility issues
- Initialization sequence

Coding of initialization rules in methods `<createObject>()`, `<init>()`,...
which are then symbolically executed

Limitations of Method Inlining: methodBodyExpand

- Source code might be unavailable
  - source code often unavailable for commercial APIs, even for some JAVA API methods (& implementation vendor-specific)
  - method implementation deployment-specific
- Method is invoked multiple times in a program
  - avoid multiple symbolic execution of identical code
- Cannot handle unbounded recursion
- Not modular: changes to called methods require re-verification of caller even

Use method contract instead of method implementation

1. Show that requires clause is satisfied
2. Continue after method call
   - 'Ignoring' ealier values of modifiable locations
   - assuming ensures clause

Method Contract Rule: Normal Behavior Case

Warning: Simplified version

```
/*@ public normal_behavior
  @ requires preNormal;
  @ ensures postNormal;
  @ assignable mod;
  @*/ // implementation contract of m()
```

$$\frac{\Gamma \Longrightarrow \mathcal{U}\mathcal{F}(\texttt{preNormal}), \Delta \quad \text{(precondition)} \qquad \Gamma \Longrightarrow \mathcal{U}\mathcal{V}_{\texttt{mod}}(\mathcal{F}(\texttt{postNormal}) \rightarrow \langle\pi\ \omega\rangle\phi), \Delta \quad \text{(normal)}}{\Gamma \Longrightarrow \mathcal{U}\langle\pi\,\texttt{result} = \texttt{m}(\texttt{a}_1, \ldots, \texttt{a}_\texttt{n});\ \omega\rangle\phi, \Delta}$$

- $\mathcal{F}(\cdot)$: translation from JML to Java DL
- $\mathcal{V}_{\texttt{mod}}$: anonymising update,
  forgetting pre-values of modifiable locations

JML Method Contracts Revisited

```
/*@ public normal_behavior
  @ requires preNormal;
  @ ensures postNormal;
  @ assignable mod;
  @*/
T m(T1 a1, ..., Tn an) { ... }
```

Implicit Preconditions and Postconditions

- The object referenced by **this** is not **null**: **this**!=**null** (precondition only; **this** cannot be changed by method)
- The heap is wellformed: `wellFormed(heap)` (precondition only)
- Invariant for 'this': `\invariant_for(this)`

Method Contract Rule: Normal Behavior Case

Warning: Simplified version

```
/*@ public normal_behavior
  @ requires preNormal;
  @ ensures postNormal;
  @ assignable mod;
  @*/ // implementation contract of m()
```

$$\frac{\Gamma \Longrightarrow \mathcal{U}\mathcal{F}(\mathtt{preNormal}), \Delta \quad \text{(precondition)} \qquad \Gamma \Longrightarrow \mathcal{U}\mathcal{V}_{\mathtt{mod}}(\mathcal{F}(\mathtt{postNormal}) \rightarrow \langle \pi\ \omega \rangle \phi), \Delta \quad \text{(normal)}}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\, \mathtt{result} = \mathtt{m}(\mathtt{a}_1, \ldots, \mathtt{a}_\mathtt{n}); \omega \rangle \phi, \Delta}$$

- $\mathcal{F}(\cdot)$: translation from JML to Java DL
- $\mathcal{V}_{\mathtt{mod}}$: anonymising update,
  forgetting pre-values of modifiable locations

Keeping the Context

- Want to keep part of prestate $\mathcal{U}$ that is unmodified by called method
- **assignable** clause of contract tells what can possibly be modified

```
@ assignable mod;
```

- How to erase all values of **assignable** locations in state $\mathcal{U}$?
- Anonymising updates $\mathcal{V}$ erase information about modified locations

Anonymising Heap Locations

Define anonymising function $\texttt{anon}$: Heap × LocSet × Heap → Heap

The resulting heap $\texttt{anon}(...)$ coincides with the first heap on all locations except for those specified in the location set. Those locations attain the value specified by the second heap.

Definition:

$$\texttt{select}(\texttt{anon}(h1, locs, h2), o, f) = \begin{cases} \texttt{select}(h2, o, f) & \text{if } (o, f) \in locs \\ \texttt{select}(h1, o, f) & \text{otherwise} \end{cases}$$

Usage:

$$\mathcal{V}_{mod} = \{\texttt{heap} := \texttt{anon}(\texttt{heap}, locs_{mod}, \texttt{h}_a)\}$$

where $\texttt{h}_a$ a new (not yet used) constant of type Heap

Effect: After $\mathcal{V}_{mod}$, modfied locations have unknown values

Anonymising Heap Locations: Example

```
@ assignable o.a, this.*;
```

To erase all knowledge about the values of the locations of the assignable expression:

- anonymise the current heap on the designated locations:

$$\mathtt{anon}(\mathtt{heap}, \{(\mathtt{o}, \mathtt{a})\} \cup \mathtt{allFields}(\mathtt{this}), \mathtt{h}_a)$$

- assign the current heap the new value

$$\mathcal{V}_{mod} = \{\mathtt{heap} := \mathtt{anon}(\mathtt{heap}, \{(o, a)\} \cup \mathtt{allFields}(\mathtt{this}), \mathtt{h}_a)\}$$

Method Contract Rule: Exceptional Behavior Case

Warning: Simplified version

```
/*@ public exceptional_behavior
  @ requires preExc;
  @ signals (Exception exc) postExc;
  @ assignable mod;
  @*/
```

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}\mathcal{F}(\mathtt{preExc}), \Delta \quad \text{(precondition)} \\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}_{mod}((\mathtt{exc} \neq \mathbf{null} \wedge \mathcal{F}(\mathtt{postExc})) \\ \qquad\qquad\qquad\qquad \rightarrow \langle \pi\, \mathbf{throw}\ \mathtt{exc};\ \omega\rangle\phi), \Delta \quad \text{(exceptional)}\end{array}}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\, \mathtt{result} = \mathtt{m}(\mathtt{a}_1, \ldots, \mathtt{a}_\mathtt{n});\ \omega\rangle\phi, \Delta}$$

- $\mathcal{F}(\cdot)$: translation from JML to Java DL
- $\mathcal{V}_{mod}$: anonymising update

Method Contract Rule – Combined

Warning: Simplified version

KeY uses actually only one rule for both kinds of cases.

Therefore translation of postcondition ϕ_{post} as follows (simplified):

$$\begin{aligned}
\phi_{post_n} &\equiv \mathcal{F}(\texttt{\textbackslash old}(\texttt{normalPre})) \land \mathcal{F}(\texttt{normalPost}) \\
\phi_{post_e} &\equiv \mathcal{F}(\texttt{\textbackslash old}(\texttt{excPre})) \land \mathcal{F}(\texttt{excPost})
\end{aligned}$$

$$\frac{\begin{array}{l}
\Gamma \Longrightarrow \mathcal{U}(\mathcal{F}(\texttt{normalPre}) \lor \mathcal{F}(\texttt{excPre})), \Delta \quad \text{(precondition)} \\
\Gamma \Longrightarrow \mathcal{U}\mathcal{V}_{mod_{normal}}(\phi_{post_n} \rightarrow \langle \pi\ \omega \rangle \phi), \Delta \quad \text{(normal)} \\
\Gamma \Longrightarrow \mathcal{U}\mathcal{V}_{mod_{exc}}((\texttt{exc} \neq \texttt{null} \land \phi_{post_e}) \\
\qquad\qquad \rightarrow \langle \pi\ \texttt{throw exc};\ \omega \rangle \phi), \Delta \quad \text{(exceptional)}
\end{array}}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\ \texttt{result} = \texttt{m}(\texttt{a}_1, \ldots, \texttt{a}_\texttt{n});\ \omega \rangle \phi, \Delta}$$

- $\mathcal{F}(\cdot)$: translation to Java DL
- $\mathcal{V}_{mod}$: anonymising update (similar to loops)

Method Contract Rule: Example

```
class Person {
 private /*@ spec_public @*/ int age;
 /*@ public normal_behavior
   @ requires age < 29;
   @ ensures age == \old(age) + 1;
   @ assignable age;
   @ also
   @ public exceptional_behavior
   @ requires age >= 29;
   @ signals_only ForeverYoungException;
   @ assignable \nothing;
   @//allows object creation (else use \strictly_nothing)
   @*/
 public void birthday() {
   if (age >= 29) throw new ForeverYoungException();
   age++;
 } }
```

Method Contract Rule: Example Cont'd

Demo

`methods/useContractForBirthday.key`

- Proof without contracts (all except object creation)
  - Method treatment: Expand
- Proof with contracts (until method contract application)
  - Method treatment: Contract
- Proof contracts used
  - Method treatment: Expand
  - Select contracts for `birthday()`in `src/Person.java`
  - Prove both specification cases

Verification of Loops

Symbolic execution of loops: unwind

$$\text{unwindLoop}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\pi\, \texttt{if (b)\{p; while (b) p\}}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\pi\, \texttt{while (b) p}\ \omega]\phi, \Delta}$$

How to handle a loop with...

- 0 iterations? Unwind $1\times$
- 10 iterations? Unwind $11\times$
- 10000 iterations? Unwind $10001\times$
- an unknown number of iterations?

We need an invariant rule (or some form of induction)

Loop Invariants

Idea behind loop invariants

- A formula *Inv* whose validity is preserved by loop guard and body
- Consequence: if *Inv* was valid at start of the loop, then it still holds after arbitrarily many loop iterations
- If the loop terminates, then *Inv* holds afterwards
- Construct *Inv* such that, together with loop exit condition, it implies postcondition of loop

Basic Invariant Rule

$$\text{loopInvariant}\quad \frac{\begin{array}{ll} \Gamma \Longrightarrow \mathcal{U}\, Inv, \Delta & \text{(valid when entering loop)} \\ Inv,\ b = \texttt{TRUE} \Longrightarrow [\texttt{p}]Inv & \text{(preserved by p)} \\ Inv,\ b = \texttt{FALSE} \Longrightarrow [\pi\ \omega]\phi & \text{(assumed after exit)} \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\, \texttt{while(b) p}\ \omega]\phi, \Delta}$$

How to Derive Loop Invariants Systematically?

Example (First active statement of symbolic execution is loop)

```
n >= 0  & wellFormed(heap) ->
{i := 0} \[{
 while (i < n) {
      i = i + 1;
    }
 }\](i = n)
```

Look at desired postcondition (`i = n`)

What, in addition to negated guard (`i >= n`), is needed? (`i <= n`)

Is (`i <= n`) established at beginning and preserved?

Yes! We have found a suitable loop invariant!

Demo `loops/simple.key` (auto after inv)

Obtaining Invariants by Strengthening

Example (Slightly changed loop)

```
n >= 0 & n = m & wellFormed(heap) ==>
{i := 0}\[{
  while (i < n) {
    i = i + 1;
  }
}\] (i = m)
```

Look at desired postcondition `(i = m)`

What, in addition to negated guard `(i >= n)`, is needed? `(i = m)`

Is `(i = m)` established at beginning and preserved? Neither!

Can we use something from the precondition or the update?

- If we know that `(n = m)` then `(i <= n)` suffices
- Strengthen the invariant candidate to: `(i <= n & n = m)`

Generalization

Example (Addition: `x,y` program variables, `x0,y0` rigid constants)

```
x = x0 & y = y0 & y0 >= 0  & wellFormed(heap) ==>
\[{
 while (y > 0) {
  x = x + 1;
  y = y - 1;
} }\] (x = x0 + y0)
```

Finding the invariant

First attempt: use postcondition `x = x0 + y0`

- Not true at start whenever `y0 > 0`
- Not preserved by loop, because `x` is increased

Generalization

Example (Addition: x,y program variables, x0,y0 rigid constants)

```
x = x0 & y = y0 & y0 >= 0  & wellFormed(heap) ==>
\[{
 while (y > 0) {
  x = x + 1;
  y = y - 1;
} }\] (x = x0 + y0)
```

Finding the invariant

What stays invariant?

- The sum of `x` and `y`:   `x + y = x0 + y0`   "Generalization"
- Can help to think of "δ" between `x` and `x0 + y0`

Generalization

Example (Addition: `x,y` program variables, `x0,y0` rigid constants)

```
x = x0 & y = y0 & y0 >= 0  & wellFormed(heap) ==>
\[{
 while (y > 0) {
  x = x + 1;
  y = y - 1;
} }\] (x = x0 + y0)
```

Checking the invariant

Is `x + y = x0 + y0` a good invariant?

- Holds in the beginning and is preserved by loop
- But postcondition not achieved by `x + y = x0 + y0 & y <= 0`

Generalization

Example (Addition: `x,y` program variables, `x0,y0` rigid constants)

```
x = x0 & y = y0 & y0 >= 0  & wellFormed(heap) ==>
\[{
 while (y > 0) {
  x = x + 1;
  y = y - 1;
} }\] (x = x0 + y0)
```

Strenghtening the invariant

Postcondition holds if `y = 0`

- Sufficient to add `y >= 0` to `x + y = x0 + y0`

Demo `loops/simple3.key`

Basic Loop Invariant: Context Loss

Basic Invariant Rule: a Problem

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U}\, Inv, \Delta & \text{(initially valid)}\\ Inv,\ b = \mathtt{TRUE} \Longrightarrow [\mathtt{p}]Inv & \text{(preserved)}\\ Inv,\ b = \mathtt{FALSE} \Longrightarrow [\pi\ \omega]\phi & \text{(use case)}\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \mathbf{while}(\mathtt{b})\ \mathtt{p}\ \omega]\phi, \Delta}$$

- Context Γ, Δ, $\mathcal{U}$ must be omitted in 2nd and 3rd premise:

  Γ, Δ in general don't hold in state reached by $\mathcal{U}$

  2nd premise *Inv* must be invariant for any state, not only $\mathcal{U}$

  3rd premise We don't know the state after the loop exits
- But: context contains (part of) precondition and class invariants
- Required context information must be added to loop invariant *Inv*

Example

Precondition: $\texttt{a} \neq \texttt{null}$ & *ClassInv*

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Postcondition: $\forall\, \texttt{int}\ x; (0 \leq x \;\&\; x < \texttt{a.length} \rightarrow \texttt{a}[x] = 1)$

Loop invariant: $0 \leq \texttt{i} \;\&\; \texttt{i} \leq \texttt{a.length}$
$\&\; \forall\, \texttt{int}\ x; (0 \leq x \;\&\; x < \texttt{i} \rightarrow \texttt{a}[x] = 1)$
$\&\; \texttt{a} \neq \texttt{null}$
& *ClassInv*

Keeping the Context (As In Method Contract Rule)

- Want to keep part of the context that is unmodified by loop
- `assignable` clauses for loops tell what can possibly be modified

```
@ assignable i, a[*];
```

- How to erase all values of `assignable` locations?
- Anonymising updates $\mathcal{V}$ erase information about modified locations

Anonymising JAVA Locations

```
@ assignable i, a[*];
```

To erase all knowledge about the values of the locations of the assignable expression:

- introduce a new (not yet used) constant of type `int`, e.g., c
- introduce a new (not yet used) constant of type Heap, e.g., $\mathtt{h}_a$
  - anonymise the current heap: $\mathtt{anon}(\mathtt{heap}, \mathtt{allFields}(\mathtt{this.a}), \mathtt{h}_a)$
- compute anonymizing update for assignable locations

$$\mathcal{V} = \{\mathtt{i} := \mathtt{c} \mid\mid \mathtt{heap} := \mathtt{anon}(\mathtt{heap}, \mathtt{allFields}(\mathtt{this.a}), \mathtt{h}_a)\}$$

For local program variables (e.g., `i`) KeY computes assignable clause automatically

Loop Invariants Cont'd

Improved Invariant Rule

$$\frac{\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U} Inv, \Delta & \text{(initially valid)}\\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}(Inv \;\&\; b = \mathtt{TRUE} \rightarrow [\mathtt{p}]Inv), \Delta & \text{(preserved)}\\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}(Inv \;\&\; b = \mathtt{FALSE} \rightarrow [\pi\ \omega]\phi), \Delta & \text{(use case)}\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \mathbf{while}(\mathtt{b})\ \mathtt{p}\ \omega]\phi, \Delta}$$

- Context is kept as far as possible:
  $\mathcal{V}$ wipes out only information in locations assignable in loop
- Invariant *Inv* does not need to include unmodified locations
- For **`assignable \everything`** (the default):
  - $\mathtt{heap} := \mathtt{anon}(\mathtt{heap}, \mathtt{allLocs}, \mathtt{h}_a)$ wipes out **all** heap information
  - Equivalent to basic invariant rule
  - Avoid this! Always give a specific **`assignable`** clause

Example with Improved Invariant Rule

Precondition: $\mathtt{a} \neq \mathbf{null}$ & *ClassInv*

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Postcondition: $\forall\, \mathbf{int}\ x;\ (0 \leq x\ \&\ x < \mathtt{a.length} \rightarrow \mathtt{a}[x] = 1)$

Loop invariant: $0 \leq \mathtt{i}\ \&\ \mathtt{i} \leq \mathtt{a.length}$
$\&\ \forall\, \mathbf{int}\ x;\ (0 \leq x\ \&\ x < \mathtt{i} \rightarrow \mathtt{a}[x] = 1)$

Example in JML/Java – Loop.java Demo

```
public int[] a;
/*@ public normal_behavior
  @  ensures (\forall int x; 0<=x && x<a.length; a[x]==1);
  @  diverges true;
  @*/
public void m() {
  int i = 0;
  /*@ loop_invariant
    @  0 <= i && i <= a.length &&
    @  (\forall int x; 0<=x && x<i; a[x]==1);
    @ assignable a[*];
    @*/
  while(i < a.length) {
    a[i] = 1;
    i++;
  }
}
```

Example from a Previous Lecture

$\forall$ **int** x;
$(x = \mathtt{n} \land x >= 0 \rightarrow$
$[$
```
i = 0; r = 0;
while (i<n) { i = i + 1; r = r + i;}
r=r+r-n;
```
$]\,(\mathtt{r} = x * x)$

How can we prove that the above formula is valid
(i.e., satisfied in all states)?

Needed Invariant:

```
@ loop_invariant
@     i>=0  && i <= n && 2*r == i*(i + 1);
@ assignable \nothing; // no heap locations changed
```

Demo `Loop2.java`

Hints

Proving `assignable`

- Invariant rule above assumes that **`assignable`** is correct
  E.g., possible to prove nonsense with incorrect
  `assignable \nothing;`
- Invariant rule of KeY generates proof obligation that ensures correctness of **`assignable`**
  This proof obligation is part of (Body preserves invariant) branch

Setting in the KeY Prover when proving loops

- Loop treatment: Invariant
- Quantifier treatment: No Splits with Progs
- If program contains `*`, `/`: Arithmetic treatment: DefOps
- Is search limit high enough (time out, rule apps.)?
- When proving partial correctness, add **`diverges true;`**

What is still missing?

Is the sequent

$$\Longrightarrow [\texttt{i = -1; while (true)\{\}}]\texttt{i} = 4711$$

provable?

Yes, e.g.,

```
@ loop_invariant true;
@ assignable \nothing;
```

Possible to prove correctness of non-terminating loop

- Invariant trivially initially valid and preserved ⇒
  Initial Case and Preserved Case immediately closable
- Loop condition never false: Use case immediately closable

But need a method to prove termination of loops

Mapping Loop Execution to Well-Founded Order

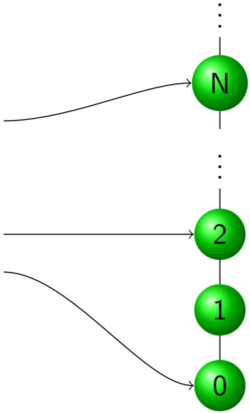
```
while (b) {
  body
}
```

if (b) { body }$_1$

⋮

if (b) { body }$_{17}$

if (b) { body }$_{18}$



Need to find expression getting smaller wrt $\mathbb{N}$ in each iteration

Such an expression is called a decreasing term or variant

Total Correctness: Decreasing Term (Variant)

Find a decreasing integer term v (called variant)

Add the following premisses to the invariant rule:

- $v \geq 0$ is initially valid
- $v \geq 0$ is preserved by the loop body
- v is strictly decreased by the loop body

Proving termination in JML/Java

- Remove directive `diverges true;` from contract
- Add directive `decreasing v;` to loop invariant
- KeY creates suitable invariant rule and PO (with $\langle \ldots \rangle \phi$)

Example (The `array` loop)

```
@ decreasing  a.length - i;
```

Files:

- `LoopT.java`
- `Loop2T.java`

Final Example: Computing the GCD

```
public class Gcd {
 /*@ public normal_behavior
   @ requires _small>=0 && _big>=_small;
   @ ensures _big!=0 ==>
   @  (_big % \result == 0 && _small % \result == 0 &&
   @    (\forall int x; x>0 && _big % x == 0
   @       && _small % x == 0; \result % x == 0));
   @ assignable \nothing;
   @*/
 private static int gcdHelp(int _big, int _small) {
   int big = _big; int small = _small;
   while (small != 0) {
     final int t = big % small;
     big = small;
     small = t;
   }
   return big;
 }
}
```

Computing the GCD: Method Specification

```
public class Gcd {
 /*@ public normal_behavior
   @ requires _small>=0 && _big>=_small;
   @ ensures _big!=0 ==>
   @ (_big % \result == 0 && _small % \result == 0 &&
   @   (\forall int x; x>0 && _big % x == 0
   @      && _small % x == 0; \result % x == 0));
   @ assignable \nothing;
   @*/
 private static int gcdHelp(int _big, int _small) {...}
```

requires normalization assumptions on method parameters (both non-negative and `_big` $\geq$ `_small`)

ensures if `_big` positive, then

- the return value `\result` is a divider of both arguments
- all other dividers `x` of the arguments are also dividers of `\result` and thus smaller or equal to `\result`

Computing the GCD: Specify the Loop Body

```
int big = _big; int small = _small;
while (small != 0) {
  final int t = big % small;
  big = small;
  small = t;
}
return big;
```

Which locations are changed (at most)?

```
@ assignable \nothing; // no heap locations changed
```

What is the variant?

```
@ decreases small;
```

Computing the GCD: Specify the Loop Body Cont'd

```
int big = _big; int small = _small;
while (small != 0) {
  final int t = big % small;
  big = small;
  small = t;
}
return big;
```

Loop Invariant

- Order between `small` and `big` preserved by loop: `big>=small`
- Possible for `big` to become 0 in a loop iteration? No.
- Adding `big>0` to loop invariant? No. Not initially valid.
- Weaker condition necessary: `big==0 ==> _big==0`

Computing the GCD: Specify the Loop Body Cont'd

```
int big = _big; int small = _small;
while (small != 0) {
  final int t = big % small;
  big = small;
  small = t;
}
return big;
```

Loop Invariant

- Order between `small` and `big` preserved by loop: `big>=small`
- Weaker condition necessary: `big==0 ==> _big==0`
- What does the loop preserve? The set of dividers!
  All common dividers of `_big, _small` are also dividers of `big, small`

```
(\forall int x; x > 0;
  (_big%x == 0 && _small%x == 0) <==>
            (big%x == 0 && small%x == 0));
```

Computing the GCD: Final Specification

```
int big = _big; int small = _small;
/*@ loop_invariant small >= 0 && big >= small &&
  @   (big == 0 ==> _big == 0) &&
  @   (\forall int x; x > 0; (_big % x == 0 && _small % x == 0)
  @            <==>
  @   (big % x == 0 && small % x == 0));
  @ decreases small;
  @ assignable \nothing;
  @*/
  while (small != 0) {
    final int t = big % small;
    big = small;
    small = t;
  }
  return big; // assigned to \result
```

Why does `big` divides `_small` and `_big` follow from the loop invariant?

If `big` is positive, one can instantiate `x` with it, and use `small == 0`

Computing the GCD: Demo

Demo `loops/Gcd.java`

1. Show `Gcd.java` and `gcd(a,b)`
2. Ensure that "DefOps" and "Contracts" is selected, $\geq$ 10,000 steps
3. Proof contract of `gcd()`, using contract of `gcdHelp()`
4. Note KeY check sign in parentheses:
   - 4.1 Click "Proof Management"
   - 4.2 Choose tab "By Proof"
   - 4.3 Select proof of `gcd()`
   - 4.4 Select used method contract of `gcdHelp()`
   - 4.5 Click "Start Proof"
5. After finishing proof obligations of `gcdHelp()` parentheses are gone

Some Hints On Finding Invariants

General Advice

- Invariants must be developed, they don't come out of thin air!
- Be as systematic in deriving invariants as when debugging a program
- Don't forget: the program or contract (more likely) can be buggy
  - In this case, you won't find an invariant!

Some Hints On Finding Invariants, Cont'd

Technical Hints

- The desired postcondition is a good starting point
  - What, in addition to negated loop guard, is needed for it to hold?
- If the invariant candidate is not preserved by the loop body:
  - Can you add stuff from the precondition?
  - Does it need strengthening?
  - Try to express the relation between partial and final result
- Simulate a few loop body executions to discover invariant patterns
- If the invariant is not initially valid:
  - Can it be weakened such that the postcondition still follows?
  - Did you forget an assumption in the requires clause?
- Several "rounds" of weakening/strengthening might be required
- Use the KeY tool for each premiss of invariant rule
  - After each change of the invariant make sure all cases are ok
  - Interactive dialogue: previous invariants available in "Alt" tabs

Understanding Unclosed Proofs

Reasons why a proof may not close

- Buggy or incomplete specification
- Bug in program
- Maximal number of steps reached: restart or increase # of steps
- Automatic proof search fails: manual rule applications necessary

Understanding open proof goals

- Follow the control flow from the proof root to the open goal
- Branch labels give useful hints
- Identify unprovable part of post condition or invariant
- Sequent remains always in "pre-state"
  Constraints on program variables refer to value at start of program
  (exception: formula is behind update or modality)
- NB: $\Gamma \Longrightarrow o = \mathtt{null}, \Delta$ is equivalent to $\Gamma, o \neq \mathtt{null} \Longrightarrow \Delta$

Literature for this Lecture

Essential

KeY Book Verification of Object-Oriented Software (see course web page), Chapter 10: Using KeY

KeY Book Verification of Object-Oriented Software (see course web page), Chapter 3: Dynamic Logic, Sections 3.1, 3.2, 3.4, 3.5, 3.6.1, 3.6.2, 3.6.3, 3.6.4, 3.6.5, 3.6.7, 3.7