PREDICATE-TRANSFORMER–BASED VERIFICATION (LN CHAPTER 2)

Wishnu Prasetya
wishnu@cs.uu.nl
www.cs.uu.nl/docs/vakken/pv

PLAN

- Hoare logic
- Verification using predicate transformers
- Verification with “Guarded Command Language” (GCL)
- Objects and exceptions
- Should at end give an answer to “how to mechanically verify a real program”.

SPECIFYING PROGRAMS

- We can use “Hoare triples” :

$\{ \text{true} \}$ plusOne(x) *body* $\{ \text{return} = x+1 \}$

$\{ \#S>0 \}$ max(S) *body* $\{ \text{return} \in S \;\wedge\; (\forall x: x \in S : \text{return} \geq x) \}$

- *total* and *partial correctness* interpretation.

HOARE LOGIC

- Provides a set of “inference rules” to prove the validity of Hoare triples. Example:

$$\frac{O \Rightarrow P \;,\quad \{P\}\;\; \mathrm{S} \;\; \{Q\}}{\{O\}\;\; \mathrm{S} \;\; \{Q\}}$$

- **Note** : LN uses the notation $\vdash\ O \Rightarrow P$

THE LOGIC'S GENERAL IDEA: BREAK IT DOWN!

$$\frac{\{P\}\ S_1\ \{Q\}\quad,\quad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1 ; S_2\ \{R\}}$$

$$\frac{\{P \wedge g\}\ S_1\ \{Q\}\quad,\quad \{P \wedge \neg g\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ g\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}}$$

ASSIGNMENT

- Eventually, everything boils down to "assignments"
- An assignment is correct if .....

$$\frac{P \Rightarrow Q[e/x]}{\{\,P\,\}\ x{:=}e\ \{\,Q\,\}}$$

- **Note**: this assumes that the assignment only changes the value of x (it does not silently affect other variables)

LOOP

- A loop is correct if you can find an “invariant” :

$$\frac{\begin{array}{c} P \Rightarrow I \\ \{\, g \wedge I \,\} \quad S \quad \{\, I \,\} \\ I \wedge \neg g \Rightarrow Q \end{array}}{\{\, P \,\} \ \ \underline{\textbf{while}} \ \ g \ \ \underline{\textbf{do}} \ \ S \quad \{\, Q \,\}}$$

- E.g. a trivial loop:

{ k=1000 } **while** k>0 **do** k := k-1 { k=0 }

FEW MORE EXAMPLES

{ y=0 /\ k=0 } **while** k<10 **do** { y := y+2 ; k++ } { y = 20 }

{ y=0 /\ k=10 } **while** k ≠ 0 **do** { y := y+2 ; k-- } { y = 20 }

FEW MORE EXAMPLES

Home work:

$\{ s=false \ \wedge \ k=0 \}$

while k<#b **do** {
    s = s $\vee$ b[k] ;
    k = k + 1
}

$\{ \ s \ = \ (\exists i: 0 \leq i < \#b : b[i]) \ \}$

PROVING TERMINATION (OF LOOP)

- Extend the previous rule to:

$P \Rightarrow I$
$\{ g \wedge I \} \quad S \quad \{ I \}$
$I \wedge \neg g \Rightarrow Q$

$\{ I \wedge g \} \quad C{:=}m \,;\, S \quad \{ m{<}C \}$ // m decreasing
$I \wedge g \Rightarrow m > 0$ // m bounded below

$\{ P \}$ **<u>while</u>** g **<u>do</u>** S $\{ Q \}$

EXAMPLE

```
{  x≥0  /\  y≥0 }

  while x+y>0  do  {
     if  x>0  then  { x--   ;   y := y+100 }
     else  y--
}

 { x=0 /\  y=0 }
```

HOME WORK

```
{  x>1 }


  while x>0  do  {
     if  x>1  then  x := x - 2
     else  x :=  x + 1
}


{ true }
```

HOARE LOGIC CANNOT BE DIRECTLY AUTOMATED

- Problem:

$$\frac{\{P\}\ S_1\ \{Q\}\quad,\quad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1 ; S_2\ \{R\}}$$

- Let's now look at "predicate transformer-based verification". A *predicate transformer* is a function of type :

  Statement $\rightarrow$ Predicate $\rightarrow$ Predicate

FORWARD AND BACKWARD TRANSFORMER

- **cp**$\triangleright$ S *P* (forward) : transform a given pre-condition *P* to a post-condition.
- **cp**$\triangleleft$ S *Q* (backward) : transform a given post-condition *Q* to a pre-condition.

- Do they produce valid (sound) pre/post conditions? Yes if :

  { *P* } S { **cp**$\triangleright$ S *P* }

  { **cp**$\triangleleft$ S *Q* } S { *Q* }

FORWARD AND BACKWARD TRANSFORMER

- Sound and complete if :

$$\{\,P\,\}\ \ \mathrm{S}\ \ \{\,Q\,\} \qquad \equiv \qquad \mathbf{cp}\triangleright \mathrm{S}\ \ P \Rightarrow Q$$

$$\{\,P\,\}\ \ \mathrm{S}\ \ \{\,Q\,\} \qquad \equiv \qquad P \Rightarrow \mathbf{cp}\triangleleft \mathrm{S}\ \ Q$$

WP AND WLP TRANSFORMER

- **wp** *S* Q : the weakest pre-condtion so that S terminates in *Q*.
- **wlp** S *Q* : the weakest pre-condition so that S, if it terminates, will terminate in *Q*.

  wlp = weakest liberal pre-conidtion.

- Though, we will see later, that we may have to drop the completeness property of **wp**/**wlp**, but we will still call them **wp**/**wlp**.

GUARDED COMMAND LANGUAGE (GCL)

- Simple language to start
- Expressive enough to encode larger languages
- So that you can keep your logic-core simple
- Constructs
  - assignment, seq, **while, if-then-else**
  - **var** x **in** S , uninitialized local-var.
  - **assert** *e* **,** **assume** *e*
  - **try-catch**
  - program decl, program call
  - primitive types, arrays

WLP

- **wlp** **skip** Q = Q
- **wlp** (x:=e) Q = Q[e/x]
- **wlp** (S_1 ; S_2) Q = **wlp** S_1 (**wlp** S_2 Q)

We don't need to propose our own intermediate predicate!

- Example, prove:

$$\boxed{\{\, x \neq y \,\} \quad tmp := x \ ; x := y \ ; y := tmp \quad \{\, x \neq y \,\}}$$

WLP

- **wlp** (**assert** e) Q = $e \wedge Q$
- **wlp** (**assume** e) Q = $e \Rightarrow Q$
- **wlp** (S [] T) Q = (**wlp** S Q) $\wedge$ (**wlp** T Q)

- With respect to Hoare triples these two are equivalent (any Hoare triple satisfied by one is satisfied by the other) :

if g **then** S_1 **else** S_2

$\equiv$

(**assume** g ; S_1) [] (**assume** $\neg g$ **;** S_2 **)**

WLP

- So, it follows:

  $\textbf{wlp}\ \ (\textbf{if}\ \ g\ \ \textbf{then}\ \mathrm{S}_1\ \textbf{else}\ \mathrm{S}_2)\ \ \ Q$

  $=$

  $(g\ \Rightarrow\ \textbf{wlp}\ \mathrm{S}_1\ Q)\ \ \bigwedge\ \ (\neg g\ \Rightarrow\ \textbf{wlp}\ \mathrm{S}_2\ \ Q)$

- **Note** : it is equivalent to $(g \bigwedge \textbf{wlp}\ \mathrm{S}_1\ Q)\ \ \bigvee\ \ (\neg g\ \bigwedge\ \textbf{wlp}\ \mathrm{S}_2\ \ Q)$

FORMULA GROWTH

- Note that wlp of if-then-else "duplicates" Q (2x).
- So a series like:

wlp (**if** *g* **then** S1 **else** S2 ;
if *h* **then** S3 **else** S4 ;
if *l* **then** S5 **else** S5) *Q*

will duplicate Q 8x. (exponential growth in the size of the resulting wlp)

PATH-BASED VERIFICATION

- Any program S that does not contain a loop or recursion can be equivalently decomposed into linear "program paths".
- Example :

$$\textbf{if } g \textbf{ then } x:=e_1 \textbf{ else } x := e_2 \text{ ;}$$
$$\textbf{if } h \textbf{ then } y:=e_3 \textbf{ else } y := e_4$$

Can be decomposed to:

1. **assume** g ; $x:=e_1$; **assume** h ; $y := e_3$
2. **assume** g ; $x:=e_1$; **assume** $\neg h$; $y := e_4$
3. **assume** $\neg g$; $x:=e_2$; **assume** h ; $y := e_3$
4. **assume** $\neg g$; $x:=e_2$; **assume** $\neg h$; $y := e_4$

PATH-BASED VERIFICATION

- {P} S {Q} is valid ≡ forall program path σ of S: {P} σ {Q} is valid.
- E.g. to verify :

$$\{\ P\ \}$$
$$\textbf{if } g \textbf{ then } x{:=}e_1 \textbf{ else } x := e_2\ ;$$
$$\textbf{if } h \textbf{ then } y{:=}e_3 \textbf{ else } y := e_4$$
$$\{\ Q\ \}$$

1. $\{ P \}$ **assume** g ; $x{:=}e_1$; **assume** h ; $y := e_3$ $\{ Q \}$
2. $\{ P \}$ **assume** g ; $x{:=}e_1$; **assume** $\neg h$; $y := e_4$ $\{ Q \}$
3. $\{ P \}$ **assume** $\neg g$; $x{:=}e_2$; **assume** h ; $y := e_3$ $\{ Q \}$
4. $\{ P \}$ **assume** $\neg g$; $x{:=}e_2$; **assume** $\neg h$; $y := e_4$ $\{ Q \}$

Each is reducible to pred. logic formula. We can automate this!

- This approach of verification is also called "**symbolic execution**", because it as if we symbolically execute each control path in the target program.
- The number of paths can still be a lot, but we can verify them incrementally, and even choose which ones to verify.

UNFEASIBLE PATH

- Consider as an example of program path (recall that the "assumes" came originally from branch-guards) :

$$\{ P \} \ \textbf{assume}\ g\ ;\ x{:=}x{+}1\ ;\ \textbf{assume}\ h\ ;\ y := x\ \ \{ y{>}z \}$$

$$C_1 = P \wedge g \qquad C_2 = P \wedge g\ \wedge h[x{+}1/x]$$

- A program path can turn out to be **unfeasible** if no actual execution can trigger it. This happens if one of its branch-guard is unfeasible towards the path pre-condition (P above) :
  - Condition g above is unfeasible if C_1 is *unsatisfiable*
  - Condition h is unsatisfiable if C_2 in *unsatisfiable*
- Verifying an unfeasible path is waste of effort, but checking if a path in unfeasible also takes effort (above, you need to check C_1 and C_2).

FORWARD SYMBOLIC EXECUTION

- **Forward** symbolic execution: executing a program path, using "variables" to symbolically represent all possible inputs.

```
{ x=y }
x:=x+2 ;
z:= x-1
{ x>y }
```

We maintain a "symbolic" state. Initially

$x \longmapsto x_0$

$y \longmapsto y_0$

$z \longmapsto z_0$

Constraint: $x_0 = y_0$

- "$x \longmapsto x_0$" means the variable x has the value x_0.
- x0, y0, z0 : fresh variables representing initial values of x,y,z.
- "Constraint" is a condition that the symbolic values must satisfy, e.g. because it is imposed by the program's pre-condition.

FORWARD SYMBOLIC EXECUTION

- **Forward** symbolic execution: we execute a program, taking a formula as its input :

0	$\{ x=y \}$	(0) $x \longmapsto x_0$, $y \longmapsto y_0$, $z \longmapsto z_0 \wedge x_0=y_0$
1	$x:=x+2$;	(1) $x \longmapsto x_0+2$, $y \longmapsto y_0$, $z \longmapsto z_0 \wedge x_0=y_0$
2	$z:= x-1$	(2) $x \longmapsto x_0+2$, $y \longmapsto y_0$, $z \longmapsto x_0+2-1 \wedge x_0=y_0$
3	$\{ x>y \}$	

- The Hoare triple on the left is valid if and only if the final symbolic state implies the post-condition. That is, if this is valid:

$$x=x_0+2 \ \wedge \ y=y_0 \wedge z=x_0+2-1 \wedge \ x_0=y_0 \quad \Longrightarrow \quad x=y$$

FORWARD SYMBOLIC EXECUTION

- More generally a symbolic state consist of a pair (C,s) where C is a constrain and s is a mapping **$x \longmapsto e$** that maps every program variable to its symbolic value (which can be an expression e.g. α+1). To keep it simple, we won't keep track of a stack (needed if you have recursive calls).
- Initially the symbolic state is ($true$, **[$x_1 \longmapsto \alpha_1$, $x_2 \longmapsto \alpha_2$, ...])** where α_k is a fresh variable representing an arbitrary initial value of x_k .
- Given a program path ρ (sequence of instructions), and an initial symbolic state (C,s), to symbolically execute ρ we simply execute the instructions in ρ in the order as they appear, passing on the new symbolic state after an instruction ρ_k to the execution of the instruction ρ_{k+1}.
- executing **assume** P on (C,s) gives (P ∧ C, s) as the new symbolic state.
- executing **assert** Q on (C,s) gives the same state (C, s) as the new symbolic state, if C ∧ s ⟹ Q holds. Else the execution is aborted, reporting an error (the asserted Q is violated).
- executing **y:=e** on (C,s) gives (C, **update** y e' s) as the new symbolic state, where e' is the symbolic value of the expression e on the state s, which can be obtained by replacing every program variable z that occurs in e with s(z) (the symbolic value of z in the state s).

BACKWARD VS FORWARD SYMBOLIC EXECUTION

- Consider again the verification of:

$$\{\ x=y\ \}$$
$$x:=x+2\ ;\ z:=x-1$$
$$\{\ x>y\ \}$$

wlp-based verification

$$x=y\ \Rightarrow\ x+2>y$$

- Wlp-based. The program is valid iff $x=y\ \Rightarrow\ x+2>y$

- Forward symbolic execution: the program is valid iff

$$x=x_0+2\ \land\ y=y_0\ \land\ z=x_0+2-1\ \land\ x_0=y_0\ \Longrightarrow\ x=y$$

BACKWARD VS FORWARD TRANSFORMATION

- Backward execution yields: $x{=}y \Rightarrow$ **$x{+}2{>}y$**.
- Forward execution yields:

$$\boxed{y=y_0 \wedge x_0=y_0 \wedge x = x_0+2 \wedge z=x_0+2-1} \Rightarrow x>y$$

- Backward transformation: yields cleaner formulas, containing only conditions relevant towards the post-cond.
- Forward transformation
  - The direction is more intuitive
  - The intermediate formulas also produce conditions that correspond to the feasibility of each branch-guard (1x, the blue box above), so you can immediately check the guard feasibility. Note we can also check this via wlp; it is just that we need more staging in the corresponding symbolic execution.

CAN WE PROVE THE SOUNDNESS AND COMPLETENESS OF THE WLP TRANSFORMER ?

- Yes, e.g. wrt a denotational semantic. I will just give a sketch of how to do this.

- Consider the following semantical domains:
  - **State :** the space of all possible program states.
  - **val** : the space of all possible values of program variables

- Note: Chapter 1 and 2 propose two different representations of states. They are both usable, as they both support state query and update.

THE SEMANTIC OF EXPRESSIONS AND PREDICATES

- $\mathcal{E}$: expr $\longrightarrow$ (**State** $\longrightarrow$ **val**)
- $\mathcal{E}$: **Pred** $\longrightarrow$ (**State** $\longrightarrow$ **bool**)   // overloading $\mathcal{E}$
- e.g.

  $$\mathcal{E}[\![x>y]\!] \;=\; (\lambda s.\;\; s\,x \;>\, s\;y)$$

- Note that P:**State** $\rightarrow$ **bool** can equivalently be seen as a set of all the states on which P is true:

  “P as a set” $=$ $\{\; s \;\mid\; s \in$ **State** $\bigwedge\; P\,s\,\}$

- Predicate operators translate to set operators, e.g. $\bigwedge$, $\bigvee$ to $\cap\ \cup$, negation to complement wrt **State**.
- This ordering: “$P \Rightarrow Q$ is valid” translates to $P \subseteq Q$

THE SEMANTIC OF STATEMENTS

- $\mathcal{S}$: stmt $\longrightarrow$ (**State** $\longrightarrow$ **State**)

- Alternatively: $\mathcal{S}$: stmt $\longrightarrow$ (**State** $\longrightarrow$ **Pow**(**State**)) to allow non-determinism.

  where **Pow**(**State**) is the domain of all subsets of **State**. On this domain, we have: $\cap$ $\cup$ $\subseteq$ $\supseteq$

THE SEMANTIC OF STATEMENTS (DETERMINISTIC)

- $\mathcal{S}\,[\![\textbf{skip}]\!] \quad = (\lambda s.\ s)$
- $\mathcal{S}\,[\![\ \mathrm{x} := \mathrm{e}\ ]\!] = (\lambda \mathrm{s}.\ \textbf{update}\ \mathrm{s}\ \mathrm{x}\ \ (\mathcal{E}\,[\![\ \mathrm{e}\ ]\!]\ \mathrm{s})\ )$
- $\mathcal{S}\,[\![\ \mathrm{S}_1\ ;\ \mathrm{S}_2\ ]\!] = (\lambda \mathrm{s}.\ \ \mathcal{S}\,[\![\ \mathrm{S}_2\ ]\!]\ \ (\mathcal{S}\,[\![\ \mathrm{S}_1\ ]\!]\ \mathrm{s})\ )$

SEMANTIC OF HOARE TRIPLE (DETERMINISTIC)

- $\mathcal{H}$: Hoare-triple $\longrightarrow$ bool

  $\mathcal{H}$({P} S {Q}) = $\forall$s: $\mathcal{E}[\![P]\!]$ s : $\mathcal{E}[\![Q]\!]$ ($\mathcal{S}[\![S]\!]$ s)

- wlp is sound and complete if:

  {P} S {Q} if and only if P $\Rightarrow$ wlp S Q is valid

- In comes down to proving that: for all state s, and post-cond Q: $\mathcal{E}[\![Q]\!]$ ($\mathcal{S}[\![S]\!]$s) if and only if $\mathcal{E}[\![$ wlp S Q $]\!]$ s
- Can be proven inductively over the stucture of S.

DEALING WITH LOOP

- Unfortunately, no general way to calculate **wlp** of loops.
- For annotated loop, let's "define" :

wlp (**inv** I **while** g **do** S) Q = I , provided

- $I \wedge \neg g \Rightarrow Q$
- $I \wedge g \Rightarrow$ **wlp** S I

WHAT IF THERE IS NO I ANNOTATED ?

- Heuristics to construct invariants e.g. based on the form of the post-condition → not in scope.
- Dynamically infer invariants → not in scope.
- Non-heuristic approaches, simple; can be used as starting points.
  - Fix point based
  - Unfolding

WLP AS FIX POINT

- Note this first:

  loop

  while *g* **do** S

  ≡

  if g **then** { S ; **while** *g* **do** S } **else** skip

- let *W* = **wlp** *loop* *Q*.

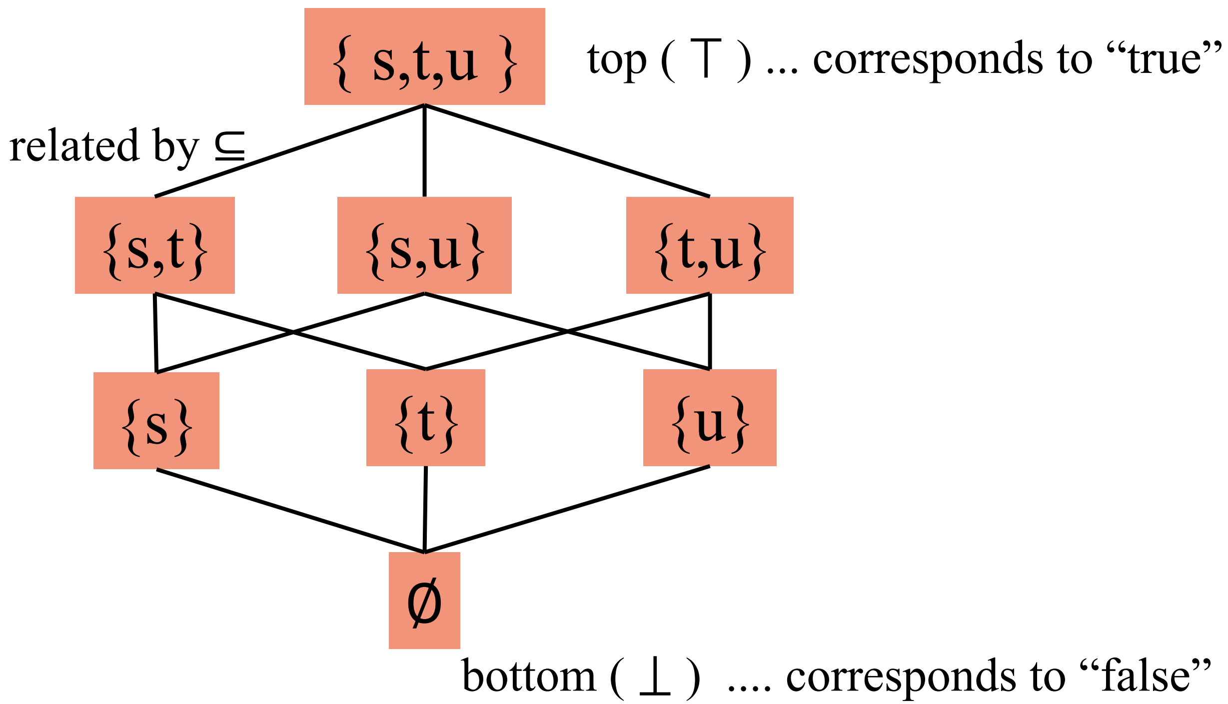
$$= (g \Rightarrow \textbf{wlp}\ \mathrm{S}\ (\textbf{wlp}\ loop\ \mathrm{Q})) \land (\neg g \Rightarrow Q)$$

$$= \underbrace{(g \land \textbf{wlp}\ \mathrm{S}\ \underbrace{(\textbf{wlp}\ loop\ \mathrm{Q})}_{W}) \lor (\neg g \land Q)}_{\mathbf{F}(W)}$$

- We are looking for the "weakest" solution of *W* = **F**(*W*)

THE DOMAIN OF STATE PREDICATES, POW(∑)

- Suppose ∑ (set of all states) = { s,t,u }. The domain Pow(∑), ordered by ⊆ :



$X \cup Y$ (union) acts as the least upper bound of X and Y

$X \cap Y$ (intersection) acts as the greatest lower bound of X and Y

SOME BIT OF FIX POINT THEORY

- $(A,\leq)$ where $\leq$ is a p.o., is a *complete lattice*, if every subset $X \subseteq A$ has a supremum and infimum.
  - Supremum = least upper bound = join $\bigvee X$ $\bigsqcup X$
  - Infimum = greatest lower bound = meet $\bigwedge X$ $\bigsqcap X$
  - So, we will also have the supremum and infimum of the whole A, often called top $\top$ and bottom $\bot$.
- Let $f : A \rightarrow A$. An x such that x = f(x) is a *fix point* of f.
- **Knaster-Tarski**. Let $(A,\leq)$ be a complete lattice. If $f : A \rightarrow A$ is monotonic over $\leq$, and P is the set of all its fix points, then P is non-empty, and $(P, \leq)$ is a complete lattice.
- (thus, both $\bigsqcup P$ and $\bigsqcap P$ are also in P).

SOME BIT OF FIX POINT THEORY

- The domain (**pow**($\sum$), $\subseteq$) is a complete lattice.
  - $\sum$ = top
  - $\emptyset$ = bottom
  - A$\cup$B : least upper bound ($\bigvee$)
  - A $\cap$ B : greatest lower bound ($\bigwedge$)
- So, if **F** defined before is monotonic within this domain, then it has a greatest fix-point.
- Is **F** monotonic ?
- Is **wlp** monotonic ?

SOME BIT OF FIX POINT THEORY

- Consider now f : **pow**(∑) $\rightarrow$ **pow**(∑). It is $\cap$-continuous if for all decreasing chain $X_0 \supseteq X_1 \supseteq X_2$ ... :

$$f\,(X_0 \cap X_1 \cap \dots) \;= f(X_0) \cap f(X_1) \cap \dots$$

- If f is $\cap$-continuous, it is also monotonic.
- Is **wlp** $\cap$-continuous ?

FP ITERATION

- *Define:*
  - $f^0(X) = X$,
  - $f^{k+1}(X) = f(f^k(X))$
- Suppose f is $\cap$-continuous. Consider the series
  $f^0(\Sigma), f^1(\Sigma), f^2(\Sigma) \ldots$
  Corollaries:
  - f is also monotonic.
  - The series is a decreasing chain.
  - $\alpha = f^0(\Sigma) \cap f^1(\Sigma) \cap f^2(\Sigma) \cap \ldots$ is a fix point of f.
  - α is the greatest fix point of f.

FP ITERATION

- How to compute $\cap\{ f^{k}(\Sigma) \mid k \geq 0 \}$?
  - compute $f^{0}, f^{1}, f^{2}, \ldots$ but notice you only need to keep track of the last.
  - $X := \Sigma$;
    while $X \neq f(X)$ **do** $X := f(X)$

- Will give the greatest FP, if it terminates.
- For **wlp** :
  - W := **true** ;
    while $W \neq \mathbf{F}(W)$ **do** $W := \mathbf{F}(W)$

    where $\mathbf{F}(W) = (g \wedge \mathbf{wlp}\ \mathrm{S}\ W) \vee (\neg g \wedge Q)$

EXAMPLE 1

while y>0 **do** { y := y−1 } **{ y=0 }**

- Q is y=0
- W_0 = true
- W_1 = (y>0 $\wedge$ **wlp** S W_0) $\vee$ ($\cancel{\neg(y>0)}$ $\wedge$ y=0)
  = (y>0 $\wedge$ true) $\vee$ (y=0)

  = y≥0
- W_2 = (y>0 $\wedge$ y-1≥0) $\vee$ ($\cancel{\neg(y>0)}$ $\wedge$ y=0)
  = y ≥1 $\vee$ y=0
  = y≥0
- $W_2 = W_1$

EXAMPLE 2

while y>0 **do** { y := y−1 } { *d* }

- Q is d -- c,d are bool vars
- W_0 = true
- W_1 = (y>0 $\bigwedge$ **wlp** S W_0) $\bigvee$ ($\neg$(y>0) $\bigwedge$ d)
  = (y>0 $\bigwedge$ true) $\bigvee$ (y$\leq$0 $\bigwedge$ d)
- W_2 = (y>0 $\bigwedge$ **wlp** S W_1) $\bigvee$ (y$\leq$0 $\bigwedge$ d)
  = (y>0 $\bigwedge$ y>1) $\bigvee$ (y=1 $\bigwedge$ d) $\bigvee$ (y$\leq$0 $\bigwedge$ d)
- W_3 = (y>2) $\bigvee$ (y=2 $\bigwedge$ d) $\bigvee$ (y=1 $\bigwedge$ d) $\bigvee$ (y$\leq$0 $\bigwedge$ d)
- does not terminate ☹

EXAMPLE 3

$\textbf{while}\ \ y>0\ \textbf{do}\ \{\ \ \textbf{assert}\ d\ ;\ y := y-1\ \ \}\ \ \{\ d\ \}$

- Q is d -- c,d are bool vars
- W_0 = true
- W_1 = $(y>0 \wedge \textbf{wlp}\ S\ W_0) \vee (\neg(y>0) \wedge d)$
  = $(y>0 \wedge d) \vee (y\leq 0 \wedge d)$
  = d
- W_2 = $(y>0 \wedge \textbf{wlp}\ S\ W_1) \vee (y\leq 0 \wedge d)$
  = $(y>0 \wedge d) \vee (y\leq 0 \wedge d)$

  = d
- $W_2 = W_1$

FINITE UNFOLDING

- Define

  $[\textbf{while}]^0 \; (g,\mathrm{S}) \;=\; \textbf{assert} \neg g$
  $[\textbf{while}]^{k+1} (g,\mathrm{S}) \;=\; \textbf{if } g \textbf{ then } \{ \mathrm{S} \,;\, [\textbf{while}]^k (g,\mathrm{S}) \} \;\textbf{else}\; \mathrm{skip}$

  $\langle\textbf{while}\rangle^0 \; (g,\mathrm{S}) \;=\; \textbf{assume} \neg g$
  $\langle\textbf{while}\rangle^{k+1} (g,\mathrm{S}) \;=\; \textbf{if } g \textbf{ then } \{ \mathrm{S} \,;\, \langle\textbf{while}\rangle^k (g,\mathrm{S}) \} \;\textbf{else}\; \mathrm{skip}$

- Iterate at most *k*-times.
- Iterate at most *k* times, then miracle.

REPLACING WHILE WITH $[\text{WHILE}]^K$

$\{\,P\,\}$ **while** $y>0$ **do** $\{\, y := y-1 \,\}$ $\{\, y=0 \,\}$

- **wlp** ([**while**]^2 $(y>0 ,\, y := y-1)$) $(y=0)$

  $= (\, y=2 \;\vee\; y=1 \;\vee\; y=0 \,)$

- Works if *P* says y is exactly that (0,1,2).
- Does not work if *P* is e.g. y=3 or y≥0
- Such unfolding produces *sound* wlp, but *incomplete*.

REPLACING WHILE WITH $\langle$WHILE$\rangle^K$

- **wlp** $\boxed{(\langle \textbf{while} \rangle^2\ (y>0\ ,\ y := y-1))}$ $(y=0 \wedge b)$

  $= \quad y>2 \vee (y=2 \wedge b) \vee (y=1 \wedge b) \vee (y=0 \wedge b)$

- P : $y=0 \vee y=1$ ... works
- P : $y \geq 0$ ... "works" as well
- This unfolding yields wlp which is complete but unsound. So, if $P \Rightarrow W$, W is the above wlp, is valid, we don't know if the original specification is also valid. However, if $P \Rightarrow W$ is invalid, then so is the orig. spec.

VERIFICATION OF OO PROGRAMS

- GCL is not OO, but we can encode OO constructs in it. We'll need few additional ingredients :
  - local variables
  - simultant assignment
  - program call
  - array
  - object
  - method

LOCAL VARIABLE

- **wlp** (**var** x **in** x:=1 ; y := y+x **end**) (x=y)

  Rename loc-vars to fresh-vars, to avoid captures:

  wlp (**var** x' **in** x':=1 ; y := y+x' **end**) (x=y)

- Let's try another example :

  wlp (**var** x' **in** **assume** x'>0 ; y := y+x' **end**) (x=y)

- Note that if x' is fresh we can alternatively treat this as:

  assume x'>0 ; y := y+x'

SIMULTANT ASSIGNMENT

- For example: $x,y := y , x+y$ **{ x=y }**

  wlp $(x,y := e_1,e_2)\ Q\ =\ Q[e_1,e_2 / x,y]$

- Compare it with : $(x=y)\ [y/x]\ [x+y/y]$

- But e.g. this is **not allowed**: $x,x := e_1, e_2$

PROGRAM AND PROGRAM CALL

- Syntax: *Pr*(*x*, *y* | *r*) *body*
- *x*,*y* are input parameters → passed by value
- *r* is an output parameter.
- Syntax of program call: *a* := *Pr*(e_1,e_2)

  Example :
  - *inc*(*x* | *r*) { r := x+1 }
  - x := inc(x)
  - y := inc(x)

ENCODING PROGRAM CALL

- Consider : *Pr*(*x*, *y* | *r*) *body*
- We treat program call as syntactic sugar :

  $a := Pr(e_1,e_2)$

  $\equiv$

  var x,y, *r* **in** x,y := e1,e2 ; *body* ; *a* := *r* **end**

  Rename to make the loc-vars fresh.
- This allows you to using existing rules to calculate the wlp of calls (except when we have recursion, see next slide).
- example:

  wlp (*x* := *inc*(*x*)) (x>1) $\rightarrow$ x+1 > 1

BLACK-BOX CALL

- What if we don't know the body? Assuming you still have the specification, e.g. :

  $\{\, x{>}0 \,\}$ $\;$ $drop(x \mid r)$ $\;$ $\{\, r < x \,\}$

  We treat this as having this body :

  $drop(x \mid r)$ $\{$ **assert** $x{>}0$; **assume** $r < x$ $\}$

- **Note**: reference to input params (e.g. "x") in a method post-cond is intended to refer to their initial value.
- example: **wlp** $(x := drop(x))$ $(x{<}7)$
  $\rightarrow$ $x{>}0$ $\wedge$ $(r < x \Rightarrow r < 7)$
- **(assuming *r* is fresh!)**
- **Note:** this allows us to handle call to recursive programs, if they provide specifications (but proving the correctness of a recursive program is still another problem to solve)

ARRAYS

- Arrays in this GCL are infinite (but not in your project).
- a := b is assumed to clone b into a.
- Introduce the notation a(i **repby** e) to denote a clone of the array a, but differs at i-th element, which now has the value e. See LN, Rule 2.6.12.
- Treat a[i] := e as a := a(i **repby** e)
- Example :
  wlp (a[i] := 0) (a[i] = a[3])
- Encode a[i] := 0 in an RL language e.g. as :

  assert $0 \leq i < \#a$; a[i] := 0

ARRAY ASSIGNMENTS TRIGGER CASES

- Consider wlp calculation of:

$$(k+1=1 \rightarrow 2 \mid (k+1=k \rightarrow 1 \mid a[k+1])) = 2$$

$$(k+1=1 \rightarrow 2 \mid a[k+1]) = 2$$

```
0 {a[k+1]=2}
1   a[k] := 1 ;
2   a[1] := 2
3 { a[k+1] = 2 }
```

- For each array expression in the post-condition, each array assignment adds a nested conditional expression. Above this leads to 3 cases that the back-end prover must consider. If the post-cond has one more array expression, it will create its own 3 cases. In combination with the first, now there are 3x3 cases to consider!

ARRAY ASSIGNMENTS TRIGGER CASES

- Consider again this wlp calculation of:

$(k+1=1 \rightarrow 2 \mid (k+1=k \rightarrow 1 \mid a[k+1])) = 2$

$(k+1=1 \rightarrow 2 \mid a[k+1]) = 2$

```
0 { a[k+1]=2 }
1    a[k] := 1 ;
2    a[1] := 2
3 {  a[k+1] = 2 }
```

- We can reduce/simplify **this**
- Simplifying cost effort.

OBJECTS

- Introduce 'heap' H : [int][fieldname] $\rightarrow$ value
- We may want to introduce one heap for each Class; but let's ignore this here.
- N, representing the number of existing objects.
- Objects are stored in H[0] ... H[N-1]
- Two types of values : primitive values, or reference to another object.
- Encoding :
  - x := o , unchanged, but note o is thus an int
  - o.fn := 3 is encoded by H[o][fn] := 3
  - Suppose C has a single int-field named a

    x = **new** C() is encoded by { H[N][a] := 0 ; x:=N ; N++ }

METHODS

- Let m(x|r) be a method of class C. It implicitly has the instance-object and the whole heap as input and output parameter.
- Example: **class** C { a:int
  inc(d) { **this**.a := **this**.a+d ; } }

  The method is translated to:

  inc(this,H,d | H') { H[this][a] := H[this][a] + d ;
  H' := H }

- call o.inc(3) is translated to ... ?

EXCEPTION

- Exception introduces non-standard flow of execution, which are often error prone.
- The problem is, exception can be thrown from many points in the program, basically exploding the goals to solve for verification.
- But first ... how to deal with exception in Hoare logic / **wlp** ?
  - Approach-1 : introduce a variable *exc* to represent that the program has entered an exceptional state, and explicitly encode the flow in the **wlp**.
  - Approach-2: extend post-condition to be a pair (Q_n,Q_e) representing the desired situation when a program terminates normally, and when it terminates exceptionally.

APPROACH 1

- Introduce a global variable *exc :* bool, initially false
- **raise** sets this variable to true
- entering a handler resets it to false
- A state where *exc* is true, is an ‘exceptional’ state, else it is a normal state.
- Multiple exception types can be encoded, but let’s ignore this here...
- The obvious:

  wlp raise Q = Q[true/*exc*]

APPROACH 1

- S_1 ; S_2 is treated as: S_1 ; **if** exc **then** skip **else** S_2

Downside: this blows up the resulting wlp since each ";" will add one conditional clause.

Can we avoid that?

  - We can optimize this by statically checking if exc would be true after S_1. If so, there is no need to expand S_2.
  - Caution: what should we do with "**if** g **then** raise **else** skip ; S" ?

- **while** also need to be transformed due to the implicit sequencing between iterations

ASSIGNMENTS AND GUARDS

- In GCL evaluating an expression does not crash; so you either have to **insist** that they don’t. E.g. :

  x := a[k]/y

  is transformed to:

  assert 0≤k<#a ;
  assert y≠0 ;
  x := a[k]/y ;

- Similar situation with guards in ite and loop.

ASSIGNMENTS AND GUARDS

- Or we model exception throwing executions. E.g. :

  x := a[k]/y

  is then transformed to :

  if $k<0 \lor k \geq \#a$ **then raise** ;
  if $y=0$ **then raise** ;
  x := a[k]/y ;

- Similar situation with guards in ite and loop.

EXCEPTION HANDLER

- S ! H (**try** S **catch** H)

  is treated as:

  S ; **if** exc **then** { exc:=false ; H } **else** skip

  (";" should not be expanded here)

APPROACH 2

- Extend the post-condition from a single predicate to a pair of predicate:

  {P} S {Q,R} where Q describes the expectation when S terminates normally, and R the expectation when S terminates by exception.

- **wlp** $(x := e)\ (Q,R) = Q[e/x]$ -- assuming e does not throw an exception.
- **wlp raise** $(Q,R) = R$
- **wlp** $(S_1\ ;\ S_2)\ (Q,R) =$ **wlp** S_1 (**wlp** S_2 $(Q,R), R)$
- **wlp** $(S\ !\ H)\ (Q,R) =$ **wlp** S $(Q,$ **wlp** $H\ (Q,R))$
- This avoids blow up that we saw in Approach-1, but on the other hand wlp becomes more complicated.

EXCEPTION

- Approach 1:
  - wlp-rules are still the same
  - But it leads to blow up, unless we do the optimization.
- Approach 2:
  - does not blow up, but we need to keep track of two post-conditions.
- **If** g **then** raise **else** skip ; S → needs some pre-processing to avoid blow up.
- To consider: using a control flow graph.

SUMMARY

- You have now the full **wlp**-based logic to verify GCL programs.
- The calculation of **wlp** is syntax driven.
- If loops are annotated, the calculation of **wlp** is fully automatic, else you may have to do some trade off.
- RL languages can be translated to GCL; we have shown how some core OO constructs can be translated.
- **wlp**-based logic does *not* deal with concurrency.