Ciclos repetitivos – invocación de métodos

Ma. Laura Cobo

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Argentina

Cálculo para programas

El cálculo realiza la interpretación simbólica del programa

El proceso sigue los siguientes lineamientos:

- Trabaja sobre la primer sentencia activa
- Descompone sentencias complejas en sentencias más simples
- Transforma asignaciones simples en updates
- La acumulación de updates captura los cambios en los estados del programa.
- Las ramificaciones en el flujo de control inducen particiones en la prueba.
- Los updates computan la precondición más débil de U con respecto a la fórmula ϕ, $\Gamma \Rightarrow \{U\}\phi$

Departamento de Ciencias e Ingeniería de la Computación – Universidad Nacional del Sur, Argentina

Updates cuantificados

Los updates secuenciales y paralelos cubren una buena parte de lo que se requiere para evaluar una fórmula en un estado pero …. no todo

Algunas situaciones comunes son:

- Una variable de programa es seteada a una constante:

  `{i:= 5} ϕ`

- Una variable de programa es incrementada en uno:

  `{i:= i+1} ϕ`

- Dos variables "swapean" valores

  `{i:= j || j:=i} ϕ`

- Todas las componentes de un arreglo de una longitud constante, por ejemplo tamaño 2, tienen un valor

  `{arr[0]:=0 || arr[1]:=0} ϕ`

Updates cuantificados

Los updates secuenciales y paralelos cubren una buena parte de lo que se requiere para evaluar una fórmula en un estado pero …. no todo

¿Qué sucede si la situación se plantea para un arreglo de longitud **`n`**?

- Es decir se manipula una expresión como:

```
< int[] a = new int[n]; >
∀ int x; (0<= x < a.length → a[x]≐0)
```

Se requiere de un update cuantificado

Updates cuantificados

La idea del update cuantificado es realizar updates paralelos para todos los objetos de un determinado tipo que estén en el dominio

$$\{\backslash\mathtt{for}\ T\ x; \backslash\mathtt{if}\ \phi(x); l(x) := r(x)\}$$

Es importante tener en cuenta que:

- La expresión condicional es opcional
- Generalmente la variable x se menciona en ϕ, l y r, aunque no es necesario.
- Hay una forma normal que permite la computación eficiente de los updates.

El tipo debe ser bien-odenado. No tiene cadenas descendientes infinitas
El tipo int es bien-ordenado en KeY

Updates cuantificados: ejemplos

La idea del update cuantificado es realizar updates paralelos para todos los objetos de un determinado tipo que estén en el dominio

$$\{\backslash\mathtt{for}\ T\ x; \backslash\mathtt{if}\ \phi(x); l(x) := r(x)\}$$

- Inicialización del campo `a` para objetos de clase `C`:

  `{\for C obj; obj.a:= 9}`

- Inicialización de las componentes de un arreglo `arreglo`:

  `{\for int i; arreglo[i]:= 1}`

Invariantes de ciclo

La regla de aplicación del ciclo en general se aplica un número desconocido de veces

Este hecho hace que se requiera una regla invariante o alguna otra forma de inducción

La idea de los invariantes de ciclo:

- Una fórmula `Invariante` cuya validez sea preservada por el ciclo repetitivo, tanto por la condición como por el cuerpo.
- De esta manera `Invariante` fue válida al comenzar el ciclo, y aún lo es luego de una cantidad arbitraria de iteraciones.
- Claramente si el ciclo termina, entonces `Invariante` se mantiene.
- La construcción de `Invariante` debe ser tal que implique la postcondición del ciclo

Invariantes de ciclo

Symbolic execution of loops: unwind

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

La regla de aplicación del ciclo en general se aplica un número desconocido de veces

Basic Invariant Rule

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

Departamento de Ciencias e Ingeniería de la Computación – Universidad Nacional del Sur, Argentina

Invariantes de ciclo

Basic Invariant Rule

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

El contexto Γ, Δ, U debe omitirse en la segunda y tercer premisa.

- U representa el estado de comienzo del ciclo, no el estado alcanzado luego de algunas iteraciones
- El mantener Γ, Δ sin U significaría ejecutar $\mathtt{P}$ en el pre-estado del programa.
- Pero … El contexto es importante para las precondiciones e invariantes. La única solución es agregar lo que se necesite del contexto en el invariante.

Ejemplo

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

Invariante de clase: `arreglo` $\not\stackrel{\circ}{=}$ `null`

Post-condición:

$\forall$ `int x;(0<=x<arreglo.longitud -> arreglo[x]` $\stackrel{\circ}{=}$ `1)`

Invariante

`0 <= i & i <arreglo.longitud)`

`& arreglo` $\not\stackrel{\circ}{=}$ `null`

`&` $\forall$ `int x;(0<= x< i -> arreglo[x]` $\stackrel{\circ}{=}$ `1)`

El invariante hace referencia a las condiciones previas y establecidas por el ciclo, junto con el invariante de clase y la post-condición.

Manteniendo el contexto

- Se espera poder mantener parte del contexto que se mantiene inmodificable en el ciclo.
- Las cláusulas `assignable` para ciclos indican qué puede modificarse

```
@ assignable i, arreglo[*]
```

- Reemplazar x por constantes frescas en la regla de cuantificación derecha $(\Rightarrow \forall x; \phi)$ o en el antecedente (Γ). Para cambiar el valor de una locación del programa se utiliza el update, no la sustitución.
- Anonimización de updates, $\mathcal{V}$ borra información sobre las locaciones modificadas

$$\mathcal{V} = \{i{:=}c \;||\; \backslash\mathbf{for}\ x,\ arreglo[x]{:=}f_a(x)\}$$

c, f_a son contantes frescas

Departamento de Ciencias e Ingeniería de la Computación – Universidad Nacional del Sur, Argentina

Manteniendo el contexto

Improved Invariant Rule

$$\frac{\begin{array}{ll} \Gamma \Longrightarrow \mathcal{U} Inv, \Delta & \text{(valid when entering loop)} \\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}(Inv \;\&\; b \doteq \mathtt{TRUE} \rightarrow [\mathrm{p}]Inv), \Delta & \text{(preserved by p)} \\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}(Inv \;\&\; b \doteq \mathtt{FALSE} \rightarrow [\pi\ \omega]\phi), \Delta & \text{(assumed after exit)} \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \mathtt{while(b)\ p}\ \omega]\phi, \Delta}$$

- El contexto se mantiene tanto como es posible
- El invariante no necesita incluir las locaciones inmodificables
- Para la instrucción por defecto, `assignable \everything`
  - $\mathcal{V} = \{* := *\}$ Limpia toda la información
  - Equivale a la regla de invariante clásica
  - **IMPORTANTE**: tratar de evitar esta situación indicando siempre una clausula de asignabilidad.

Ejemplo

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

Invariante de clase: `arreglo` $\not\overset{\circ}{=}$ **`null`**

Post-condición:

$\forall$ **`int`** `x;(0<=x<arreglo.longitud -> arreglo[x]` $\overset{\circ}{=}$ `1)`

Invariante

`0 <= i` **`&`** `i < arreglo.longitud`

~~**`&`** `arreglo` $\not\overset{\circ}{=}$ **`null`**~~

`&` $\forall$ **`int`** `x;(0<=x< i -> arreglo[x]` $\overset{\circ}{=}$ `1)`

No se necesita para el invariante

Ejemplo

```
 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 i,a[*];
    @*/};
  while(i<a.length) {
       a[i]=1;
       i++
  }
}
```

Ejemplo

```
∀ int x;
  (x≗n ∧ x>= 0 →
    [i = 0; r=0;
    while (i<n) {i=i+1;r=r+i}
    r=r+r-n;]
   r≗x*x
```

Invariante

```
@ loop_invariant
@       i>=0 && 2*r==i*(i+1)&& i <=n;
@ assignable i,r;
```

Departamento de Ciencias e Ingeniería de la Computación – Universidad Nacional del Sur, Argentina

Tips

- La regla de invariante asume que la cláusula `assignable` es correcta. Si `assignable \nothing` cerrar pruebas sin sentido
- La regla del invariante de KeY genera proof-obligations que aseguran la correctitud de lo expresado en la cláusula `assignable`

Al probar ciclos Key debe tener los siguientes seteos establecidos:

- **Loop treatment: invariant**
- **Quantifier treatment: no splits with progs**
- **Si el programa contine *, /: arithmetic treatment: DefOps**
- **Diverges true; si se prueba correctitud parcial**

Correctitud total en ciclos

Hay que encontrar un término entero decreciente v (llamado variante)

Se agregan las siguientes premisas a la regla del invariante:

- ▶ $v >= 0$ es inicialmente válida
- ▶ $v >= 0$ es preservada por el cuerpo del ciclo
- ▶ v es estrictamente decrementado por el cuerpo del ciclo

Para probar terminación en JML/Java

- **Remover la directiva `diverges true;`**
- **Agregar la directiva `decreasing v;` al invariante de ciclo**
- **KeY crea una regla de invariante apropiada y proof obligation con <...>ϕ**

Ejemplo

```
 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 i,a[*];
    @ decreasing a.length - i
    @*/};
  while(i<a.length) {
      a[i]=1;
      i++
  }
}
```

Problemas con las invocaciones a métodos

La ejecución simbólica de métodos JAVA API puede resolverse de dos maneras:

- ▶ Los métodos tienen una implementación de referencia en Java. El cuerpo del método inline puede ejecutarse en forma simbólica. Problemas:
    1. La implementación de referencia no siempre está disponible.
    2. Demasiado costoso
    3. Imposible lidiar con la recursión
- ▶ Utilizar el **contrato** del método en lugar de su implementación

Comprensión de las pruebas …

Razones por las cuales una prueba puede no **cerrar**:

- Especificación incompleta o con bugs
- Bugs en el programa
- Numero máximo de pasos alcanzado: recomenzar o incrementar la cantidad de pasos.
- La búsqueda de una prueba automática falla y la aplicación manual de reglas se vuelve necesaria.

Comprensión de las pruebas ...

Comprensión de la situación de metas **abiertas**:

- Seguir el flujo de control tomado desde la raíz hacia la meta abierta.
- Las etiquetas de ramificación pueden dar pistas útiles.
- Identificar (parte de) la post-condición o invariante que no puede ser probado
- El secuente se mantiene siempre en el “pre-estado”. Es decir, una restricción como `indice>0` hace referencia siempre al valor de `indice` anterior a la ejecución del programa (recodar como excepción la situación en la cual la fórmula es posterior a un update o modalidad)