Procedimientos y Funciones

Cátedra AED

DC-UBA

2 cuat, 2023

Procedimientos y funciones: por qué?

- Reuso de código

Procedimientos y funciones: por qué?

- Reuso de código
- Razonamiento más compacto y efectivo

Procedimientos y funciones: por qué?

- Reuso de código
- Razonamiento más compacto y efectivo
- Evolución (correcta) de código

Ejemplo Proc y Uso

Notar que el lenguaje SmallLang no tenía ni definiciones ni invocaciones a procedimientos. Agregamos la definición de procedmientos (ya vista de alguna manera) y la invocación `x := Call P (E)` al lenguaje. Lo mantenemos simple para ilustrar el concepto

Ejemplo Proc y Uso

Notar que el lenguaje SmallLang no tenía ni definiciones ni invocaciones a procedimientos. Agregamos la definición de procedmientos (ya vista de alguna manera) y la invocación `x := Call P (E)` al lenguaje. Lo mantenemos simple para ilustrar el concepto

```
PROC Sumatoria (in hasta:ℕ):ℕ
AUX s:ℕ;
AUX i:ℕ;
s:=0;
i:=1;
While i ≤ hasta
 s:=s+i;
 i:=i+1
EndWhile;
Result:=s;
Return
```

Ejemplo de (Re)Uso de Proc

```
x:= Sumatoria(n);
y:= Sumatoria(m-1);
z:= x - y
```

Procedimientos y funciones: por qué?

- Reuso de código: Ok, es más o menos obvio (abstracción procedimental)

Procedimientos y funciones: por qué?

- Reuso de código: Ok, es más o menos obvio (abstracción procedimental)
- Razonamiento más compacto/abstracto: Usar la abstracción procedimental para no pensar en cómo hace lo que hace. ¿O sea?...

Ejemplo Proc y Uso con Contratos

```
PROC Sumatoria (in hasta:ℕ):ℕ
AUX s:ℕ;
AUX i:ℕ;
s:=0;
i:=1;
While i ≤ hasta
 s:=s+i;
 i:=i+1
EndWhile;
Result:=s;
Return
```

```
{true}
x:= Sumatoria(n);
y:= Sumatoria(m-1);
z:= x - y
```

$\{z = \Sigma_{k=m}^{n} k\}$

Razonamiento con Proc: Inlining

```
{true}
AUX s:ℕ;
AUX i:ℕ;
s:=0;
i:=1;
While i ≤ n
 s:=s+i;
 i:=i+1
EndWhile;
x:=s;
s:=0;
i:=1;
While i ≤ m-1
 s:=s+i;
 i:=i+1
EndWhile;
y:=s;
z:= x - y
```

$\{z = \sum_{k=m}^{n} k\ \}$

Razonamiento modular basado en procedimientos

Inlining no es problemático. PERO qué pasa si sabemos que es cierta tupla de Hoare: $\{\texttt{Pre}\}\ Cuerpo_P\ \{\texttt{Pos}\}$ (por ejemplo porque lo dice el **`requiere`** y el **`asegura`** y lo hemos probado). Ejemplo:

Razonamiento modular basado en procedimientos

Inlining no es problemático. PERO qué pasa si sabemos que es cierta tupla de Hoare: $\{\texttt{Pre}\}\ Cuerpo_P\ \{\texttt{Pos}\}$ (por ejemplo porque lo dice el `requiere` y el `asegura` y lo hemos probado). Ejemplo:

PROC Sumatoria (in hasta: $\mathbb{N}$):$\mathbb{N}$

Requiere {TRUE}

Asegura {result = $\sum_{k=1}^{hasta}$ k }

Razonamiento modular basado en procedimientos

Inlining no es problemático. PERO qué pasa si sabemos que es cierta tupla de Hoare: {`Pre`} $Cuerpo_P$ {`Pos`} (por ejemplo porque lo dice el **`requiere`** y el **`asegura`** y lo hemos probado). Ejemplo:

```
PROC Sumatoria (in hasta:  ℕ):ℕ
Requiere {TRUE}
Asegura {result = Σ_{k=1}^{hasta} k }
```

Queremos usar esa información para probar el código que invoca al procedimiento. Ejemplo:

```
{true}
x:= Sumatoria(n);
y:= Sumatoria(m-1);
z:= x - y
{z = Σ_{k=m}^{n} k }
```

Razonamiento modular basado en procedimientos

Inlining no es problemático. PERO qué pasa si sabemos que es cierta tupla de Hoare: {`Pre`} $Cuerpo_P$ {`Pos`} (por ejemplo porque lo dice el **`requiere`** y el **`asegura`** y lo hemos probado). Ejemplo:

```
PROC Sumatoria (in hasta: ℕ):ℕ
Requiere {TRUE}
```
Asegura {result = $\sum_{k=1}^{hasta}$ k }

Queremos usar esa información para probar el código que invoca al procedimiento. Ejemplo:

```
{true}
x:= Sumatoria(n);
y:= Sumatoria(m-1);
z:= x - y
```
{z = $\sum_{k=m}^{n}$ k }

Surge la pregunta: Sabiendo esto. Cuál es la Wp(`x := Call Proc (E)`, Q)?

Wp (`x:=Call P(E)`, Q) sabiendo {`Pre`}C_P{`Pos`}

- Qué quiero lograr?: Razonamiento Modular! O sea:

Wp (`x:=Call P(E)`, Q) sabiendo $\{\texttt{Pre}\}C_P\{\texttt{Pos}\}$

- Qué quiero lograr?: Razonamiento Modular! O sea:
  - Reusar de alguna manera lo que sé del procedimiento y no reproducir los pasos de la prueba $\{\texttt{Pre}\}$ $Cuerpo_P$ $\{\texttt{Pos}\}$ cada vez que me encuentro con una invocación del procedimiento

Wp (`x:=Call P(E)`, Q) sabiendo $\{\texttt{Pre}\}C_P\{\texttt{Pos}\}$

- Qué quiero lograr?: Razonamiento Modular! O sea:
  - Reusar de alguna manera lo que sé del procedimiento y no reproducir los pasos de la prueba $\{\texttt{Pre}\}$ $Cuerpo_P$ $\{\texttt{Pos}\}$ cada vez que me encuentro con una invocación del procedimiento
  - Veamos en concreto esto del razonamiento modular con Wp

Wp (`x := Call P(E)`, Q) sabiendo {`Pre`}C_P{`Pos`}

Asumamos que

- `P` tiene un parámetro formal `pf` que es `in`

Wp (`x := Call P(E)`, Q) sabiendo {`Pre`}C_P{`Pos`}

Asumamos que

- `P` tiene un parámetro formal `pf` que es `in`
- el resultado va a parar antes del retorno a la variable distinguida `result`

Wp (`x := Call P(E)`, Q) sabiendo $\{\texttt{Pre}\}C_P\{\texttt{Pos}\}$

Asumamos que

- `P` tiene un parámetro formal `pf` que es `in`
- el resultado va a parar antes del retorno a la variable distinguida `result`
- `Pre` predica sobre `pf`

Wp (`x := Call P(E)`, Q) sabiendo {`Pre`}C_P{`Pos`}

Asumamos que

- `P` tiene un parámetro formal `pf` que es `in`
- el resultado va a parar antes del retorno a la variable distinguida `result`
- `Pre` predica sobre `pf`
- `Post` sobre `pf` y `result` (i.e, `Pre(pf)` y `Post(pf,result)`)
- Asumamos que además probamos que $\texttt{pf} = \texttt{pf}_0$ en el retorno (Wp ó analizando en el código) ya que lo pide el hecho de ser un parámetro `in`

Entonces:

Wp (x := Call P(E), Q) sabiendo {Pre}C_P{Pos}

Asumamos que

- P tiene un parámetro formal pf que es in
- el resultado va a parar antes del retorno a la variable distinguida result
- Pre predica sobre pf
- Post sobre pf y result (i.e, Pre(pf) y Post(pf,result))
- Asumamos que además probamos que pf = $\mathtt{pf}_0$ en el retorno (Wp ó analizando en el código) ya que lo pide el hecho de ser un parámetro in

Entonces:

Wp (x := Call P(E), Q) $=_{def}$ Def(E) $\wedge_l$ Pre[pf/E] $\wedge_l$
$\forall$ r :: (Post[pf/E,result/r] $\Rightarrow$ Q[x/r])

Wp (`x := Call P(E)`, Q) sabiendo {`Pre`}C_P{`Pos`}

Asumamos que

- `P` tiene un parámetro formal `pf` que es `in`
- el resultado va a parar antes del retorno a la variable distinguida `result`
- `Pre` predica sobre `pf`
- `Post` sobre `pf` y `result` (i.e, `Pre(pf)` y `Post(pf,result)`)
- Asumamos que además probamos que `pf = pf`$_0$ en el retorno (Wp ó analizando en el código) ya que lo pide el hecho de ser un parámetro `in`

Entonces:

`Wp (x := Call P(E), Q)` $=_{def}$ `Def(E)` $\wedge_l$ `Pre[pf/E]` $\wedge_l$
$\forall$ `r ::  (Post[pf/E,result/r]` $\Rightarrow$ `Q[x/r] )`

Dónde / es sustitución de variable libre a la izquierda por expresión a la derecha. Nota, `Q[x/r]` es lo mismo que $\mathtt{Q}^x_r$

Ejemplo

```
{true}

x:= Sumatoria(n);

y:= Sumatoria(m-1);

z:= x - y
```
$\{z = \sum_{k=m}^{n} k\}$

Ejemplo

$\{\texttt{true}\}$

```
x:= Sumatoria(n);
```
$\{\forall \texttt{r}.(\texttt{r} = \sum_{k=1}^{m-1} \texttt{k}) \Rightarrow \texttt{x-r} = \sum_{k=m}^{n} \texttt{k}\} \equiv \{\texttt{x-}(\sum_{k=1}^{m-1} \texttt{k}) = \sum_{k=m}^{n} \texttt{k}\}$
```
y:= Sumatoria(m-1);
```
$\{\texttt{x-y} = \sum_{k=m}^{n} \texttt{k}\}$
```
z:= x - y
```
$\{\texttt{z} = \sum_{k=m}^{n} \texttt{k}\}$

Ejemplo

$\{\texttt{true}\}$

```
x:= Sumatoria(n);
```
$\{\texttt{x-(}\sum_{k=1}^{m-1}\texttt{ k) = }\sum_{k=m}^{n}\texttt{ k}\}$
```
y:= Sumatoria(m-1);
```
$\{\texttt{x-y = }\sum_{k=m}^{n}\texttt{ k}\}$
```
z:= x - y
```
$\{\texttt{z = }\sum_{k=m}^{n}\texttt{ k}\}$

Ejemplo

$\{\texttt{true}\}$

$\{\forall \texttt{r}.(\texttt{r = } \sum_{k=1}^{n} \texttt{ k }) \Rightarrow \{\texttt{r-(}\sum_{k=1}^{m-1} \texttt{ k) = } \sum_{k=m}^{n} \texttt{ k}\}$

```
x:= Sumatoria(n);
```

$\{\texttt{x-(}\sum_{k=1}^{m-1} \texttt{ k) = } \sum_{k=m}^{n} \texttt{ k}\}$

```
y:= Sumatoria(m-1);
```

$\{\texttt{x-y = } \sum_{k=m}^{n} \texttt{ k}\}$

```
z:= x - y
```

$\{\texttt{z = } \sum_{k=m}^{n} \texttt{ k}\}$

Ejemplo

$\{\texttt{true}\} \not\Rightarrow$
$\{(\sum_{k=1}^{n} \texttt{k})\ \texttt{-}\ (\sum_{k=1}^{m-1} \texttt{k})\ \texttt{=}\ \sum_{k=m}^{n} \texttt{k}\} \equiv \{\texttt{n} \geq \texttt{m}\}$

```
x:= Sumatoria(n);
```

$\{\texttt{x-}(\sum_{k=1}^{m-1} \texttt{k})\ \texttt{=}\ \sum_{k=m}^{n} \texttt{k}\}$

```
y:= Sumatoria(m-1);
```

$\{\texttt{x-y}\ \texttt{=}\ \sum_{k=m}^{n} \texttt{k}\}$

```
z:= x - y
```

$\{\texttt{z}\ \texttt{=}\ \sum_{k=m}^{n} \texttt{k}\}$

Algunas Conclusiones

Algunas Conclusiones

- Usamos el **qué** del procedimiento para probar el **cómo** del código que lo usa (código "cliente"). **Abstracción procedimental acompañada de razonamiento modular!**

Algunas Conclusiones

- Usamos el **qué** del procedimiento para probar el **cómo** del código que lo usa (código "cliente"). **Abstracción procedimental acompañada de razonamiento modular!**
- Cualquier cambio del procedimiento que deje igual o debilite su precondición y deje igual o fortalezca la postcondición NO impacta en la corrección del código "cliente" (Design by Contracts (Meyer)/ **Principio de Sustitución** de Liskov). Evolución disciplinada del software

Algunas Conclusiones

- Usamos el **qué** del procedimiento para probar el **cómo** del código que lo usa (código "cliente"). **Abstracción procedimental acompañada de razonamiento modular!**
- Cualquier cambio del procedimiento que deje igual o debilite su precondición y deje igual o fortalezca la postcondición NO impacta en la corrección del código "cliente" (Design by Contracts (Meyer)/ **Principio de Sustitución** de Liskov). Evolución disciplinada del software
- Lo que viemos es una pieza central en el camino hacia mecanismos que ponen -de manera abstracta- a disposición procedimientos (y estructuras de datos) que el código cliente puede invocar (ej. liberías) o ser invocado (ej. framework)
- Lo que viene: Tipo Abstractos de Datos