Procedimientos y Funciones

Algoritmos y Estructuras de Datos

DC-UBA

2024

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 `PROC Sumatoria (in hasta:ℕ):ℕ`

```
VAR s:ℕ;
VAR 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)
- 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
VAR s:ℕ;
VAR 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
```
$\{z = \sum_{k=m}^{n} k\}$

Razonamiento con Proc: Inlining

```
{true}
VAR s:ℕ;
VAR 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

Podemos aprovechar que sabemos que contamos con una implementación probada de Sumatoria? O sea dada la especificación:

```
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. Dado que sabemos que:

$\{\texttt{True}\}\ Cuerpo_{Sumatoria}\ \{\texttt{result} = \sum_{k=1}^{hasta} \texttt{k}\}$

Ejemplo:

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

Cuál es la Wp(`x := Call Sumatoria (n)`, Q)?

Wp (`x:=Call P(E)`, Q)

Dado un procedimiento (o función) P tal que sabemos que $\{\texttt{Pre}\}C_P\{\texttt{Pos}\}$ (donde C_P es el cuerpo de P).

- Vamos a razonamiento de forma modular
  - Vamos a aprovechar que ya probamos que $\{\texttt{Pre}\}\ C_P\ \{\texttt{Pos}\}$ cada vez que aprece invocación a P.
  - Como podemos definir Wp (`x:=Call P(E)`, Q).

Wp (`x := Call P(E)`, Q)

Asumamos que:

`PROC P (in pf: T) :` $\mathtt{T}_r$

  `Requiere` {`PRE`}

  `Asegura` {`POS`}

- `pf` no cambia (es `in`)
- el resultado va a parar antes del retorno a la variable distinguida `result`
- `Pre` predica sobre `pf` y `Post` sobre `pf` y `result`

Definimos:

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

Ejemplo

```
{true}

x:= Sumatoria(n);

y:= Sumatoria(m-1);

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

Ejemplo

{true}

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

Ejemplo

$\{\texttt{true}\}$

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

Ejemplo

$\{$`true`$\}$

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

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

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

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

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

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

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

Ejemplo

$\{$`true`$\} \not\Rightarrow$

$\{(\sum_{k=1}^{n}$ `k`$)$ `-` $(\sum_{k=1}^{m-1}$ `k`$)$ `=` $\sum_{k=m}^{n}$ `k`$\} \equiv \{$`n` $\geq$ `m`$\}$

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

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

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

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

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

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

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 cuerpo 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
- Razonamiento modular es fundamental para poder analizar programas que usan librerias (que implementan por ej estructuras de datos)
- Lo que viene: Tipo Abstractos de Datos