Teoría de Lenguajes
Teoría de la Programación

Clase 6: Estado explícito

Estado

¿Qué es el estado?

- Valores que contienen resultados intermedios

Puede ser:

- Implícito
- Explícito

Estado implícito (declarativo)

Estado compuesto por:

- Argumentos
- Variables libres

En el modelo declarativo el estado la clausura de los procedimientos. Esta dado por el entorno.

```
fun {SumList Xs S}
   case Xs of H|T then {SumList T H+S}
   else S end
end
```

Estado explícito

El estado excede al llamado del procedimiento

Celdas de memoria!

Estado explícito - abstracción de datos

Posibilidad de encapsular estados en tipos de datos:

- Interfaz mas sencilla
- Efectos secundarios

Estado explícito - abstracción de datos (ej)

```
local C in
   C = {NewCell 0}
   fun {SumList Xs S}
 C:=@C+1
 case Xs of H|T then {SumList T H+S}
 else S end
   end
   fun {SumCount} @C end
end
```

Sintaxis

Tenemos dos nuevos statements válidos

```
{NewCell <x> <c>}
```

Cell creation

```
{Exchange <c> <x> <m>}
```

Cell exchange

Sintaxis - Cell creation

{NewCell X C}

Crea una nueva celda C con contenido inicial X

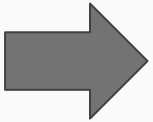
```
local C C2 C3in
   C = {NewCell 0}
   {NewCell 10 C2}
   C3 = {NewCell nil}
end
```

Sintaxis - Cell exchange

{Exchange C X Y}

Bind X con el valor actual de C y set de Y a nuevo valor de C

```
{Exchange C X Y}
```



```
X = @C
C:=Y
```

```
X=C:=Y
```

Sintaxis - Cell exchange

```
local C1 X Y Z in
   C1 = {NewCell 0}
   {Browse @C1}
   {Exchange C1 X 4}
   {Browse X#@C1}
   {Exchange C1 _ 8}
   {Exchange C1 Y @C1}
   {Browse Y#@C1}
   Z=C1:=10
   {Browse Z#@C1}
end
```

Semántica - Cell creation

En el tope tenemos el siguiente semantic statement
({NewCell <x> <y>}, E)

1. Se crea una celda con nombre n
2. Bind E(<y>) y n en el store
3. Se agrega el par (E(<y>) : E(<x>)) al store μ

Si llegan a existir problemas con el bind del paso 2 se hace un raise con error

Semántica - Cell exchange

En el tope tenemos el siguiente semantic statement
({Exchange <x> <y> <z>}, E)

1. Se activa si E(<x>) está determinado
   a. Si E(<x>) no está ligada al nombre de una celda, se levanta un error.
   b. Si existe el par (E(<x>) : w) en el store μ
      i. Se actualiza en el store μ el par con (E(<x>) : E(<z>))
      ii. Se hace un bind de E(<y>) a w en el store σ
2. Si no está determinado se suspende

Sharing, aliasing & equality

Sharing

Dos identificadores se refieren a la misma celda. A veces conocido como aliasing.

Se cambia el valor de una celda, cambian ambos

Equality

Dos valores tienen la misma estructura y valores en todas sus partes.

Dos celdas son iguales no por su valor, sino porque son la misma celda

Modelo declarativo con estado

Es posible escribir un componente con estado, que desde afuera se comporte en forma declarativa.

Se puede utilizar también para **memoization**

Abstracción de datos

Abstracción de datos

Principios

- Encapsulamiento
- Composición
- Instanciación / Invocación

Tipos abstractos de datos (ADTs / TDAs)

ADTs

Categorización

- Abierto / Cerrado
- Con estado / Sin estado
- Empaquetado / No empaquetado

Abierto vs Cerrado

Hace referencia a si la representación interna del TDA es visible o no al resto del programa.

No tiene que ver con si puedo o no ver el código, sino que tiene que ver con el uso

Con estado vs sin estado

Con estado (Stateful): El TDA tiene un estado interno que va variando con el uso

Sin estado (Stateless - Declarative): El TDA no se modifica, cada llamado devuelve siempre una nueva instancia del tipo de dato

Empaquetado vs No empaquetado

Un TDA está compuesto por 2 partes

- Datos
- Operaciones (ej: funciones que aplican a esos datos)

Un TDA empaquetado o no hace referencia a si los datos están junto con las operaciones

En un TDA empaquetado al llamar una función, sabe sobre qué datos aplicarse (bundled)

Implementación de Stack (python vs C)

```c
#define STACK_MAX 100
struct Stack {
    int     data[STACK_MAX];
    int     size;
};
typedef struct Stack Stack;
void Stack_Init(Stack *S){
    S->size = 0;
}
int Stack_Push(Stack *S, int d){
    if (S->size < STACK_MAX)
        S->data[S->size++] = d;
    else
        return -1;
}
int Stack_Pop(Stack *S, int* r){
    if (S->size == 0)
        return -1;
    else
        *r = S->data[S->size-1];
        return S->size--;
}
```

```python
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]
```

```c
Stack s;
int a;
Stack_Init(&s);
Stack_Push(&s,4);
Stack_Push(&s,10);
Stack_Pop(&s,&a);
printf("%d\n",a);
```

```python
s=Stack()
s.push(4)
s.push('dog')
print(s.peek())
s.push(True)
s.push(8.4)
print(s.pop())
print(s.pop())
```

Analicemos el siguiente caso - Stack en Oz

```
fun {NewStack} nil end
fun {Push S E} E|S end
fun {Pop S E}
   case S of X|S1 then E=X S1 end
end
fun {IsEmpty S} S==nil end
```

Le agregamos estado (unbundled - open - stateful

```
   local
      fun {NewStack} {NewCell nil} end
      proc {Push C E} C:= E|@C  end
      proc {Pop C ?E}
            case @C of X|S1 then
                E=X C:=S1
               end
         end
      fun {IsEmpty C} @C==nil end
   in
     Stack=stack(new:NewStack push:Push pop:Pop
isEmpty:IsEmpty)
   end
```

¿Cómo hacemos empaquetado?

- Usamos el scope
- Clausura de las funciones sobre datos predefinidos
- El new en vez de devolver datos, devuelve operaciones sobre estos datos

Lo hacemos empaquetado (bundled - stateful - secure)

```
local NewStack in
   fun {NewStack}
      C = {NewCell nil}
      proc {Push E} C:=E|@C end
      fun {Pop}
         case @C of X|S1 then
               C:=S1 X
            end
      end
      fun {IsEmpty} @C==nil end
   in
      stack(push:Push pop:Pop isEmpty:IsEmpty)
   end
```

Lo hacemos empaquetado sin estado (bundled - declarative - secure)

```
local NewStack in
   local
      fun {StackOps S}
            fun {Push E} {StackOps E|S} end
            fun {Pop ?E}
              case S of X|S1 then
                 E=X
                 {StackOps S1}
              end end
              fun {IsEmpty} S==nil end
         in stack(push:Push pop:Pop isEmpty:IsEmpty) end
   in
      fun {NewStack} {StackOps nil} end
   end
```

¿Cómo hacemos algo seguro / abierto?

Los bundled que vimos son seguros. Para abrirlos:

- Exponer el estado
- Exponer el creador de ops

Los unbundled eran abiertos. Para cerrarlos:

- Envolver la data en algo inmodificable => Wrappers!

Bundled - Stateful - Open

```
fun {NewStack}
   C = {NewCell nil}
   proc {Push E} C:=E|@C end
   fun {Pop}
      case @C of X|S1 then
            C:=S1
            X
      end
   end
   fun {IsEmpty} @C==nil end
in
   stack(push:Push pop:Pop isEmpty:IsEmpty data:C)
end
```

Bundled - Declarative - Open

```
   local NewStack StackObject in
      fun {StackObject S}
            local
                fun {Push E} {StackObject E|S} end
                fun {Pop ?E}
                   case S of X|S1 then E=X {StackObject
         S1} end end
                fun {IsEmpty} S==nil end
            in stack(push:Push pop:Pop isEmpty:IsEmpty
data:S)
         end
      end
      fun {NewStack} {StackObject nil} end
   end
```

Wrapper

```
proc {NewWrapper Wrap Unwrap}
   local Key = {NewName} in
      fun {Wrap X}
         fun {$ K}
            if (K==Key) then X end
         end
      end
      fun {Unwrap W}
         {W Key}
      end
   end
end
```

El Unwrap solo desenvuelve lo que el Wrap de la misma key envolvió

Unbundled - Declarative - Secure

```
 local
    {NewWrapper Wrap Unwrap}
    fun {NewStack} {Wrap nil} end
    fun {Push S E} {Wrap E|{Unwrap S}} end
    fun {Pop S ?E}
      case {Unwrap S} of X|S1 then
            E=X {Wrap S1}
            end end
    fun {IsEmpty S} {Unwrap S}==nil end
  in
    Stack=stack(new:NewStack push:Push pop:Pop
isEmpty:IsEmpty)
  end
```

Unbundled - Stateful- Secure

```
local
         {NewWrapper Wrap Unwrap}
         fun {NewStack} {NewCell {Wrap nil}} end
         proc {Push C E} C:= {Wrap E|{Unwrap @C}}  end
         proc {Pop C ?E}
            case {Unwrap @C} of X|S1 then
                E=X C:={Wrap S1}
            end
         end
         fun {IsEmpty C} {Unwrap @C}==nil end
in
    Stack=stack(new:NewStack push:Push pop:Pop
isEmpty:IsEmpty)
end
```

Capabilities

Un cómputo es seguro si está claramente definido independientemente de la existencia de otros.

Una capability es un token asociado a un objeto que da autoridad para usarlo.

Son parte esencial en “lenguajes seguros”

Wrap/Unwrap son usados como tal

```
NO ES CMMI
```

Pasaje de parámetros

Call by reference / Call by variable

Call by reference

- Es lo que se hizo siempre con Oz
- Como parámetro pasa el identificador de una variable
- La variable en la función se referencia a la misma que se pasó como parámetro

Call by variable: Caso especial de referencia cuando el identificador es una celda

Call by value

Dentro del proc se copia el valor a una nueva variable

Las modificaciones que se hacen no son visibles a quien llama al procedimiento / función

Call by value - result

Variante del call by variable.

El contenido es puesto en una nueva celda y cuando finaliza, recien ahí, se pone en la variable que llegó por parámetro

Hace invisible hacia afuera estados intermedios

Call by name

Crea un procedure value por cada parámetro.

Cada vez que se necesita el parámetro se ejecuta el procedimiento

Call by need

Variante del call by name pero que solo evalúa el parámetro la primera vez que lo necesita

Bibliografía

- **Concepts, Techniques, and Models of Computer Programming - Capítulo 6**, Peter Van Roy and Seif Haridi
- **Extras:**
  - Capabilities:
    - http://srl.cs.jhu.edu/pubs/SRL2003-02.pdf
    - https://en.wikipedia.org/wiki/Capability-based_security