

Lenguajes de Programación 2020-1

Facultad de Ciencias UNAM

Tarea Examen Parcial 4

Sandra del Mar Soto Corderi Edgar Quiroz Castañeda

Fecha de entrega: 2 de diciembre de 2019

Esta tarea vale 8 puntos sobre el parcial 4, la calificación se completa con una pregunta presencial el 29 de noviembre de 2019

1. (1pt) Se desea implementar una función `ct` que reciba un árbol heterogéneo de naturales o booleanos y devuelva la conjunción de sus elementos siempre y cuando todos sean booleanos y en otro caso devuelva el valor `n+1`, donde `n` es el primer natural encontrado en el árbol. Para este propósito defina una función `ctaux` y una expresión `e` tal que la función `ct` quede implementada como

```
ct t = handle ctaux t with x => e
```

Bosqueje la evaluación de la expresión

```
Node (iszero 9) (Node False Void Void) (Node 5 Void Void)
```

en la máquina \mathcal{K} .

Puede omitir varios pasos pero no los que involucren el manejo de excepciones.

Sugerencia: Es más fácil si define `ctaux` a partir de una función binaria `vand` que realice la conjunción si sus argumentos son booleanos y en caso contrario lance una excepción adecuada de forma que sea manejada por `e`. Puede suponer que existe una función unaria `isbool` que verifica si su argumento es o no un booleano.

Solución:

Definiendo `vand`.

```
-- Función auxiliar para mandar excepciones
bool_panic p = let x = p in if isbool x then x else raise(x)

-- Realiza la conjunción si sus argumentos son booleanos o lanza una
-- excepción con el primer argumento que no lo sea.
vand p q = and (bool_panic p) (bool_panic q)
```

Ahora, hay que usar esta función para definir `ctaux`.

```
ctaux Void = True
ctaux (Node r i d) = vand r (vand (ctaux i) (ctaux d))
```

Y usando esto, se define `ct` como

```
ct t = handle ctaux t with x => x + 1
```

Se definen algunos sinónimos para hacer más corta la evaluación

```
-- constantes
f = False
t = True
-- arboles
i = (Node False Void Void)
d = (Node 5 Void Void)
```

```

a = (Node (iszero 9) i d)
-- expresiones booleanas simples
p1 = iszero 9
-- pilas
s1 = and(-, e2), handle(-, x.x+1)
s2 = and(f, -), handle(-, x.x+1)
s3 = and(-, e7), let(-, x.e10), s2
s4 = and(f, -), let(-, e10), s3
-- otras expresiones
e3 = (vand (ctaux i) (ctaux d))
e2 = bool-panic e3
e4 = ctaux i
e5 = (bool-panic (ctaux i))
e6 = ctaux d
e7 = (bool-panic (ctaux d))
e11 = ctaux Void
e12 = bool-panic e11
e8 = (vand e11 e11)
e9 = bool-panic e8
e10 = if (isbool x) then x else raise(x)

```

Ahora, vamos a evaluar la expresión:

```

□ ⊢ ct a
→β □ ⊢ handle ctaux a with x => x+1
→κ handle(-, x.x+1) ⊢ ctaux a
→β handle(-, x.x+1) ⊢ and (bool-panic p1) e2
→κ and(-, e2), handle(-, x.x+1) ⊢ bool-panic p1
→β s1 ⊢ let x = p1 in e10
→κ let(-, x.e10), s1 ⊢ iszero 9
→κ* let(-, e10), s1 ⊢ f
→κ s1 ⊢ if isbool x then x else raise(x) [x := f]
→κ* and(-, e2), handle(-, x.x+1) ⊢ f
→κ and(f, -), handle(-, x.x+1) ⊢ bool-panic e3
→β s2 ⊢ let x = e3 in e10
→κ let(-, x.e10), s2 ⊢ e3
→β let(-, x.e10), s2 ⊢ and e5 e7
→κ and(-, e7), let(-, x.e10), s2 ⊢ e5
→β s3 ⊢ let x = e4 in e10
→κ let(-, e10), s3 ⊢ e4
→β let(-, e10), s3 ⊢ and (bool-panic f) e9
→κ and(-, e9), let(-, e10), s3 ⊢ bool-panic f
→β and(-, e9), let(-, e10), s3 ⊢ let x = f in e10
→κ* and(-, e9), let(-, e10), s3 ⊢ f
→κ and(f, -), let(-, e10), s3 ⊢ e9
→β s4 ⊢ let x = e8 in e10
→κ let(-, x.e10), s4 ⊢ e8
→β let(-, x.e10), s4 ⊢ and e12 e12
→κ and(-, e12), let(-, x.e10), s4 ⊢ e12
→β and(-, e12), let(-, x.e10), s4 ⊢ let x = e11 in e10

```

```

→K let(, x.e10), and(-, e12), let(-, x.e10), s4 > ctaux Void
→K let(, x.e10), and(-, e12), let(-, x.e10), s4 < t
→K and(-, e12), let(-, x.e10), s4 > (if isbool x then x else raise(x))[x := true]
→K and(true, -), let(-, x.e10), s4 > e12
→K and(true, -), let(-, x.e10), s4 < t
→K s4 > (if isbool x then x else raise(x))[x := t]
→K and(f, -), let(-, e10), s3 < t
→K s3 > (if isbool x then x else raise(x))[x := f]
→K and(-, e7), let(-, x.e10), s2 < f
→K and(f, -), let(-, x.e10), s2 > e7
→β and(f, -), let(-, x.e10), s2 > let x = e6 in e10
→K let(-, x.e10), and(f, -), let(-, x.e10), s2 > e6
→β let(-, x.e10), and(f, -), let(-, x.e10), s2 > and (bool-panic 5) e8
→K and(-, e8), let(-, x.e10), and(f, -), let(-, x.e10), s2 > bool-panic 5
→β and(-, e8), let(-, x.e10), and(f, -), let(-, x.e10), s2 > let x = 5 in e10
→K and(-, e8), let(-, x.e10), and(f, -), let(-, x.e10), s2 > if isbool 5 then 5 else raise(5)
→K and(-, e8), let(-, x.e10), and(f, -), let(-, x.e10), s2 << raise(5)
→K and(f, -), let(-, x.e10), s2 << raise(5)
→K and(f, -), handle(-, x.x+1) << raise(5)
→K handle(-, x.x+1) << raise(5)
→K □ > (x+1)[x := 5]
→K □ < 6

```

2. (1pt) Considere el siguiente programa, donde suponemos que el lenguaje contiene un operador primitivo `not` para la negación booleana.

```

not letcc k2 in
  iszero(2 + letcc k1 in
    3 + if x = pred 8 then
      4 * pred (continue k1 6)
    else 5 * suc (continue k2 false)
  end
end
end

```

- a) ¿Cuáles son los tipos de `k1` y `k2`?

`k1 : Cont(Integer) k2 : Cont(Bool)`

- b) ¿A qué continuaciones se ligan las variables `k1` y `k2`?

`k1 -->(continue k1 6) →K (continue (not(-), iszero(-), suma(2, -)) 6)`

`k2 -->(continue k2 false) →K (continue (not(-)) false)`

- c) ¿A qué se evalúa el programa para `x = 7` y para `x ≠ 7`?

`e[x := 7] →K True, e[x /= 7] →K True`

3. (1.5pt) Considere la siguiente función `N` que depende de ciertas funciones dadas `f`, `g`, `h`.

`N 0 = 17`

`N 1 = f (1 + 13)`

`N 2 = 22 + (2-3) + 2`

`N 3 = 22 + (f 3) + 37`

`N 4 = g 22 (f 4)`

`N 4 = 22 + (f 4) + 33 + (g y)`

`N x = h (f x) (44 - y) (g y)`

Defina la versión `cps` de `N`, denotada por `cpsN`. Para esto, puede suponer definidas las versiones `cps` de `f`, `g`, `h`, denotadas por `cpsf`, `cpsg`, `cps h`.

Atención: Las operaciones aritméticas deben permanecer sin cambios. Es decir, no se piden las versiones `cps` de `+`, `-`.

Solución

`cpsN 0 k = 17`

`cpsN 1 k = cpsf (1 + 13) (\v -> k v)`

`cpsN 2 k = k (22 + (2-3) + 2)`

`cpsN 3 k = cpsf 3 (\v -> k (22 + v + 37))`

`cpsN 4 k = cpsg 22 (\v1 -> cpsf 4 (\v2 -> k(v1 + v2)))`

`cpsN 5 k = cpsf 4 (\v1 -> cpsg y (\v2 -> k(22 + v1 + 33+ v2)))`

`cpsN x k = cpsf x (\v1 -> cpsg y (\v2 -> cps h v1 (44 -y) v2 (\v3 -> k v3)))`

4. (1.5pt) Considere las siguientes cuatro características para un lenguaje de programación.

- Existe un tipo de cadena `String` y se cumple `Int ≤ String` mediante una conversión implícita que transforma a un entero en una cadena.
Por ejemplo, `123` a `"123"`.
- Existe un tipo cadena `String` que cumple que `String ≤ Int` mediante una conversión implícita que transforma cadenas a enteros ignorando los caracteres que no sean dígitos, excepto por el carácter inicial `-`.
Por ejemplo, `"-0aw23r4"` corresponde a `-234`.
- Existe un operador binario `+` que denota a ambas la suma de enteros y la concatenación de cadenas.
- Existe un operador binario `=` que denota a ambas la igualdad de enteros y de cadenas.

Para cada par de estas características, discuta si se violan o no los principios fundamentales del subtipado. En caso afirmativo escriba un ejemplo de un programa simple que cause un comportamiento ambiguo o contraintuitivo.

- `String < Int` e `Int < String`

Si se tiene algún operador `*` sobrecargado (como en los demás casos), entonces podría existir ambigüedad en la operación.

¿Cuál de los dos valores debería recibir el casting?

En una implementación se podría resolver fijando un elemento como el tipo por omisión para estos casos, pero significaría que la función podría dar diferentes valores en ciertas condiciones dependiendo de las decisiones de diseño del lenguaje, independientemente de la definición de la función.

Por ejemplo, con una expresión `1+"2"`, ¿cuál es el tipo del resultado? Usando únicamente los principios básicos del subtipado, ambos tipos serían correctos, lo que haría que una expresión tenga dos tipos.

Además, el casting hacia entero provoca pérdida de información en los datos.

- `String <Int` y `+` sobrecargado

Considere

```
("1"+"1")+1 = 12
"1"+"(1+1) = 3
```

Es es que `+` ya no sería asociativa, lo cuál sí cumpliría si `String < Int`.

- `String <Int` y `=` sobrecargado

Considere

```
"1q" = 1
1 = "1e"
"1q" /= "1e"
```

Es decir, la igualdad ya no es transitiva, lo cuál es chocante porque ya no sería una relación de equivalencia.

- `Int <String` y `+` sobrecargado

Considere

```
(1+2)+"a" = "3a"
1+(2+"a") = "12a"
```

Es es que `+` ya no sería asociativa, lo cuál sí cumpliría si `Int < String`.

Fuera de este inconveniente, todo respecto a los tipos se mantiene en regla.

- `Int <String` y `=` sobrecargado

El único caso donde se haría coherción sería al tener una cadena y un número. Pero la única cadena a la que el número es igual es a su conversión dígito a dígito.

Por lo que aún cuando se mezclen los argumentos, se mantiene la igualdad.

No hay ningún aspecto inusual en este caso.

- `+` y `=` sobrecargados

Si suponemos que no se cumple que algún tipo es subtipo del otro, entonces toda aplicación de estos operadores siempre resultaría en operaciones entre cosas del mismo tipo.

Por lo que no hay ningún cambio o aspecto contraintuitivo en este caso.

5. (1pt) Usando la definición de números naturales en `Java` `Peso` `Pluma` vista en clase.

- Agregue un método `pot` y `leq` para las operaciones de potencia y el orden \leq .

Solución:

```
// Clase de naturales definida en las notas
class Nat extends Object {
    Object p;

    Nat (Object n) { super(); this.p = n;}

    Nat suc() { return new Nat(this); }

    Nat pred() { return (Nat) this.p; }

    Nat suma(Nat n) { return this.pred().sum(n.suc());}

    Nat multi(Nat n) { return n.suma(this.pred().multi(n));}

    /** método 'pot'
     * @param Nat n exponente al que se va a elevar la potencia.
     * @return this^n
     */
    Nat pot(Nat n) { return n.flipPot(this) }

    /** método auxiliar para calcular 'pot'
     * @param Nat n base del números del que this es exponente.
     * @return n^this
     */
}
```

```

*/
Nat flipPot(Nat n) {return n.multi(this.pred().flipPot(n));}

/** método 'leq'
 *  $this \leq n \iff this \not\leq n \iff n \not\leq this$ 
 * @param Nat n Número a comparar con this.
 * @return this  $\leq n$ 
 */
Boolean leq(Nat n) {return n.lt(this).not();}

/** método auxiliar para calcular 'leq'
 *  $this < n \iff 0 < n - this$ 
 * @param Nat n Número a comparar con this.
 * @return this < n
 */
Boolean lt(Nat n) {return this.pred().lt(n.pred());}

// método auxiliar para calcular 'not' en Boolean
Nat inv() {return new Cero(this);}
}

// Clase del Cero definida en las notas
class Cero extends Nat {
    Cero (Object n) {super(n);}

    Nat pred() {return this;}

    Nat suma(Nat n) {return n;}

    Nat multi(Nat n) {return this;}

    /** calcular el caso base del método auxiliar 'flipPot'
     * @param Nat n número del que this es exponente.
     * @return  $n^0 = 1 = suc(0) = this.suc()$ 
     */
    Nat flipPot(Nat n) {return this.suc();}

    /** calcular el caso base del método auxiliar 'lt'
     *  $0 < n \iff bool(n) = True$ 
     * @param Nat n número a comparar
     * @return this < n
     */
    Boolean lt(Nat n) {return new Boolean(n);}

    // método auxiliar para calcular 'not' en Boolean
    Nat inv() {return this.suc();}
}

```

Donde not está definido como

```

class Boolean extends Object {
    ...
    Boolean not() {return new Boolean(this.m.inv());}
}

```

b) Modele la clase Boolean para el manejo de valores booleanos

- La clase debe extender de Object
- El constructor recibirá un objeto de la clase Nat para definir su valor. 0 representa false y 1 representa true.
- Debe tener métodos true y false que regresen una instancia de Boolean según el caso.

Solución:

```
class Boolean extends Object {
    Nat m;

    Boolean (Nat n) { super(); this.m = n;}

    Boolean true() { return new Boolean (new Cero(this).suc());}

    Boolean false() { return new Boolean (new Cero(this));}

    Boolean not() { return new Boolean(this.m.inv());}
}
```

6. (2pt) Definina el siguiente lenguaje MinEAB

$e ::= n \mid \text{true} \mid \text{false} \mid e + e \mid e < e$

en Java Peso Pluma.

Hay que seguir los siguientes pasos

a) Defina una clase Expr que incluya los siguientes métodos

- `isAtom` que devuelve `true` si la expresión no tiene subexpresiones propias.
- `lsub` que devuelve la subexpresión izquierda de una expresión no atómica.
- `rsub` que devuelve la subexpresión derecha de una expresión no atómica.
- `eval` que devuelve el valor de la expresión.

Esta clase debe ser abstracta en el sentido de que no tiene atributos y por lo tanto sus métodos no tiene cuerpo. Es una interfaz.

Solución:

```
class Expr extends Object {

    Boolean isAtom() {return error;}

    Expr lsub() {return error;}

    Expr rsub() {return error;}

    Expr eval() {return error;}
}
```

b) Defina las siguientes clases que implemente a Expr

- `NumExpr` que implemente los métodos para manejar números.

Solución:

```
class NumExpr extends Expr {
    Nat v;

    NumExpr(Nat w) {super(); this.v = w;}

    Boolean isAtom() {return (new Boolean(v)).true();}

    Expr eval() {return this;}
}
```

- `BoolExpr` que implemente los métodos para manejar booleanos.

Solución:

```
class BoolExpr extends Expr {
    Boolean v;

    BoolExpr(Boolean v) {super(); this.v = v;}
}
```

```

    Boolean isAtom() {return this.v.true();}

    Expr eval() {return this;}
}

```

- SumExpr que implemente los métodos para manejar sumas.

Solución:

```

class SumExpr extends Expr {
    Expr i;
    Expr d;

    SumExpr(Expr e1, Expr e2) {super(); this.i = e1; this.d = e2}

    Boolean isAtom() {return (new Boolean(new Cero(this))).false();}

    Expr eval() {return new NumExpr(this.i.eval().v.suma(this.d.eval().v));}
}

```

- LExpr que implemente los métodos para manejar comparaciones de orden.

Solución:

```

class LExpr extends Expr {
    Expr i;
    Expr d;

    LExpr(Expr e1, Expr e2) {super(); this.i = e1; this.d = e2}

    Boolean isAtom() {return (new Boolean(new Cero(this))).false();}

    Expr eval() {return new BoolExpr(this.i.eval().v.lt(this.d.eval().v));}
}

```

- c) Dé ejemplos de instancias de cada una de las clases anteriores.

```

// Cero
Nat cero = new Cero(this);

// NumExpr
NumExpr ceroE = new NumExpr(cero);

// BoolExpr
BoolExpr falseE = new BoolExpr(new Boolean(cero));

// SumExpr
SumExpr sumCE = new SumExpr(ceroE, ceroE);

// LExpr
LExpr lC = new lteXPR(ceroE, ceroE);

```

- d) Extienda MinEAB con las expresiones `-e`, `iszero e`.

Con esta nueva definición, cree las siguientes clases

- NegExpr

Solución:

Como ahora hay números negativos, hay que definir la clase de enteros.

```

// definición de enteros usando naturales
// parejas de naturales cuya diferencia indica el número a
// representar.
//  $n \in \mathbb{N}$ 
//  $n \mapsto (n, 0)$ 
//  $-n \mapsto (0, n)$ 
class Int extends Object {

```



```

Nat i;
Nat d;

//  $\mathbb{Z} \cong \mathbb{N}^2$ 
Int(Nat i, Nat d) {super(); this.i = i; this.d = d;}

//  $(x, y) +_{\mathbb{Z}} (z, w) = (x +_{\mathbb{N}} z, y +_{\mathbb{N}} w)$ 
Int suma(Int n) {return new Int(this.i.suma(n.i), this.d.suma(n.d));}

//  $(x, y) \leq_{\mathbb{Z}} (z, w) \iff xz \leq_{\mathbb{N}} yz$ 
Boolean lt(Int n) {return this.i.suma(n.d).lt(n.i.suma(this.d));}

//  $-(a, b) = (b, a)$ 
Int neg() {return new Int(this.d, this.i);}
}

```

Hay que modificar la clase de NumExpr para manejar enteros

```

// modificaciones para manejar enteros
class NumExpr extends Expr {
    Int v;

    NumExpr(Int v) {super(); this.v = v;}
    ...
}

```

Y con estas nuevas modificaciones se define NegExpr

```

// manejar expresiones con negativos
class NegExpr extends Expr {
    Expr e;

    NegExpr(Expr e) {this.e = e;}

    Boolean isAtom() {return (new Boolean(new Nat(this))).false();}

    Expr eval() {return this.e.eval().neg();}
}

```

- IsZero

Para evaluar esto, se modifica la clase Nat agregando los siguientes método.

```

// algunas modificaciones para Booleanos
class Boolean extends Object {
    ...
    /**
     *  $p \vee q = \text{False} \iff p = q = \text{False} = \text{Bool}(0) = \text{Bool}(0 + 0)$ 
     */
    Boolean or(Boolean p) {return new Boolean(this.m.suma(p.m));}

    /**
     *  $p \wedge q = \neg(\neg p \vee \neg q)$ 
     */
    Boolean and(Boolean p) {return this.not().or(p.not()).not();}
}

// modificaciones para Nat
class Nat extends Object {
    ...
    /**
     *  $a = b \iff a \leq b \wedge b \leq a$ 
     */
    Nat eq(Nat n) {return this.leq(n).and(n.leq(this));}
    ...
}

```

```

}

// modificaciones para Int
class Int extends Object {
    ...
    Boolean isZero(Int i) {return this.i.eq(this.d);}
    ...
}
Y se usa este método para
class IsZero extends Expr {
    Expr e;

    isZero(Expr e) {super(); this.e = e;}

    Boolean isAtom() {return (new Boolean(new Nat(this))).false();}

    Expr eval() {return new BoolExpr(e.eval().v.isZero());}
}

```

e) Dé ejemplos de instancias de estas dos clases.

```

// cero
Int ceroI = new Int(new Cero(this), new Cero(this));

// NegExpr
NegExpr ceroE = new NegExpr(ceroI);

// IsZero
IsZero isZE = new IsZero(ceroI);

```

f) ¿Cómo se modifican las subclases de `Expr` definidas en puntos anteriores?

Para manejar `NegExpr`, definió la clase `Int` y se reemplazó `Nat` en `NumExpr` por `Int`.

Para definir `IsZero`, sólo se modificaron las clases de `Boolean`, `Nat` y `Int` agregando los métodos necesarios para usar `isZero` en `Int`.

Puede suponer definida la clase `Value` (esencialmente `Nat + Bool`) cuyas instancias sean los valores del lenguaje.

Además de otras clases primitivas con los métodos que requiera.

También se puede usar la constante de error en cualquier método.

7. Extra (hasta 2pt): Privacidad en Java Peso Pluma

Java proporciona mecanismos para controlar el acceso. Un método o atributo de una clase pueden declararse como *público*, *protegido* o *privado*. En Java *Peso Pluma* es posible agregar este mecanismo como sigue

```

C ::= class C extends C {p C f ;K M } // declaración de clases
M ::= p C m (C x ) {return e;} // declaración de métodos
p ::= public | protected | private // modificador de privacidad

```

EL significado intuitivo de los modificadores de privacidad es el siguiente:

- Si un método o atributo de una clase `C` se declara `public`, entonces se permite el acceso desde cualquier lugar.
- Si un método o atributo de una clase `C` se declara `protected`, entonces se permite el acceso únicamente desde métodos de `C` y subclases de `C`.
- Si un método o atributo de una clase `C` se declara `private`, entonces se permite el acceso únicamente desde métodos de `C`.

Extienda la semántica estática para filtrar programas que contengan violaciones de privacidad de acuerdo a las reglas dadas arriba. Especifique claramente cuales reglas de tipado originales se eliminan o se sustituyen por nuevas y cuáles se mantienen.

Sugerencia: Cuando se verifique si un método está bien formado en una clase `C`, se debe verificar si la expresión en el cuerpo del método no se refiere a métodos o atributos en otra clase `D` que sean privados o protegidos si $C \not\leq D$. Podría necesitarse el paso de algo más en el contexto Γ en el juicio de tipado para expresiones.