II. Paradigmas de programación - 1C2024

Todo el contenido en este archivo es una recopilación de las teóricas/prácticas del campus de la materia. Puede contener errores.

Resumen para el segundo parcial
- LPO, Resolución, Prolog, Smalltalk

Lógica de primer orden (LPO)

Sintaxis de LPO

Sea P un símbolo de predicado de aridad n y X una variable ligada por los cuantificadores ∀ y ∃, la gramática de la lógica de primer orden es la siguiente:

```
-- Términos
σ ::= P(t1, ..., tn)  -- Formula atómica
        | ⊥               -- Contradicción
        | σ -> σ
        | σ ∧ σ
        | σ ∨ σ
        | ¬σ
        | ∀X. σ           -- Cuantificación universal
        | ∃X. σ           -- Cuantificación existencial


-- Ejemplo: (= es un simbolo de predicado)
σ := succ(X) = Y -> ∃Z. X + Z = Y
```

Deducción natural

Se extiende la deducción natural de la lógica proposicional a primer orden:

Reglas para el cuantificador universal:

$$\frac{\Gamma \vdash \forall X.\sigma}{\Gamma \vdash \sigma\{X := \mathrm{t}\}}\forall_E$$

$$\frac{\Gamma \vdash \sigma \quad X \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash \forall X.\sigma}\forall_I$$

Reglas para el cuantificador existencial:

$$\frac{\Gamma \vdash \exists X.\sigma \quad \Gamma, \sigma \vdash \tau \quad X \notin \mathrm{fv}(\Gamma, \tau)}{\Gamma \vdash \tau} \exists_E$$

$$\frac{\Gamma \vdash \sigma\{\mathrm{X} := \mathrm{t}\}}{\Gamma \vdash \exists X.\sigma} \exists_I$$

Algoritmo de unificación:

Es el mismo algoritmo de unificación (Martelli-Montanari) para cálculo lambda pero adaptado a términos de primer orden.

1. **Delete**

   $$\{X \stackrel{?}{=} X\} \cup E \quad \to \quad E$$

2. **Decompose**

   $$\{\, \mathrm{f}\,(t_1, ..., t_n) \stackrel{?}{=} \mathrm{f}\,(s_1, ..., s_n)\,\} \cup E \quad \to \quad \{t_1 \stackrel{?}{=} s_1, ..., t_n \stackrel{?}{=} s_n\} \cup E$$

3. **Swap**

   $$\{\mathrm{t} \stackrel{?}{=} X\} \cup E \quad \to \quad \{X \stackrel{?}{=} \mathrm{t}\} \cup E$$

   (Si t no es una variable)

4. **Elim**

   $$\{X \stackrel{?}{=} \mathrm{t}\} \cup E \quad \to_{\{X := \mathrm{t}\}} \quad E' = E\{X := \mathrm{t}\}$$

   (Si $X \notin \mathrm{fv}(\mathrm{t})$)

5. **Clash**

   $$\{\mathrm{f}(\tau_1, ..., \tau_n) \stackrel{?}{=} \mathrm{g}(\sigma_1, ..., \sigma_n)\} \cup E \quad \to \quad \mathrm{falla} \quad (\text{Si } \mathrm{f} \neq \mathrm{g})$$

6. **Occurs-Check**

   $$\{X \stackrel{?}{=} \mathrm{t}\} \cup E \quad \to \quad \mathrm{falla} \quad (\text{Si } X \neq \mathrm{t} \;\; \mathrm{y} \;\; X \in \mathrm{fv}(\mathrm{t}))$$

Resolución general para lógica de primer orden

Entrada: fórmula σ de LPO
Salida: booleano que indica si
σ es válida

Procedimiento:

1. Escribir $\neg\sigma$ como conjunto C de **claúsulas**.
2. Buscar una **refutación** de C:
   - Si existe, el método encuentra **alguna.**

- Si no, el método puede colgarse.

1. Pasaje a forma clausal

1. Reescribir $\rightarrow$ usando $\boxed{a \rightarrow b = \neg a \vee b}$
2. Pasar a forma normal **negada**, empujando $\neg$ "hacia adentro".
   -
   Forma normal negada: una fórmula está en **FNN** si todos sus $\neg$ afectan solo a fórmulas atómicas.

   ```
   -- Reglas para pasar a FNP
   ¬(σ ∧ τ)  ->  ¬σ ∨ ¬τ
   ¬(σ ∨ τ)  ->  ¬σ ∧ ¬τ
   ¬¬σ       ->  σ
   ¬∀X.σ     ->  ∃X.¬σ
   ¬∃X.σ     ->  ∀X.¬σ
   ```

3. Pasar a forma normal **prenexa**, extrayendo $\forall$, $\exists$ "hacia afuera"
   -
   Forma normal prenexa: una fórmula está en **FNP** si está escrita con únicamente cuantificadores ($\exists$ y/o $\forall$) seguido de una fórmula sin cuantificadores.

   ```
   -- Reglas para pasar a FNP
   (∀X.σ) ∧ τ  -> ∀X.(σ ∧ τ)
   (∀X.σ) ∨ τ  -> ∀X.(σ ∨ τ)
   (∃X.σ) ∧ τ  -> ∃X.(σ ∧ τ)
   (∃X.σ) ∨ τ  -> ∃X.(σ ∨ τ)
   ```

4. Pasar a forma normal de **Skolem** (eliminar los $\exists$)
   -
   Forma normal de Skolem: Una fórmula está skolemizada si está en FNP y solo contiene $\forall$.

   ```
   -- Regla para pasar a forma normal de skolem
   ∀X1 ... ∀Xn.∃Y.σ   ->   ∀X1 ... ∀Xn.σ{Y := f(X1,...,Xn)}
   con f un nuevo simbolo de funcion de aridad n>=0
   ```

5. Pasar a forma normal **conjuntiva**, distribuyendo $\vee$ sobre $\wedge$.
   -
   Forma normal conjuntiva: Una fórmula está en **FNC** cuando está escrita enteramente como una conjunción de **cláusulas** (disyunción de literales).

   ```
   -- Reglas para pasar a forma normal conjuntiva
   σ ∨ (τ ∧ ρ)   ->   (σ ∨ τ) ∧ (σ ∨ ρ)
   (τ ∧ ρ) ∨ σ   ->   (σ ∨ τ) ∧ (σ ∨ ρ)
   ```

6. "Empujar" los cuantificadores universales hacia adentro de las conjunciones.
   De esta manera obtenemos un conjunto

$C = \{k_1, ..., k_n\}$ que representa una conjunción de cláusulas.

2. Refutación

Regla de resolución para lógica de primer orden:

$$\frac{\{\sigma_1, ..., \sigma_p, \ell_1, ..., \ell_n\} \quad \{\neg\tau_1, ..., \neg\tau_q, ..., \ell_1^{'}, ..., \ell_m^{'}\} \quad S = \text{mgu}(\sigma_1 \stackrel{?}{=} ... \stackrel{?}{=} \sigma_p \stackrel{?}{=} \neg\tau_1 \stackrel{?}{=} ... \stackrel{?}{=} \neg\tau_q)}{S(\{\ell_1, ..., \ell_n, \ell_1^{'}, ..., \ell_m^{'}\})}$$

- (Con $p > 0$ y $q > 0$)

Resolución SLD

El método de resolución SLD (Selective Linear Definite) es un método de resolución **lineal** eficiente que solo puede aplicarse a un conjunto de cláusulas dadas:

Cláusulas de Horn

Son cláusulas de la forma $\forall X_1 \ ... \ \forall X_m . \ C$ tal que la disyunción de literales C solamente tiene uno o ningún **literal positivo**.

Ejemplo :

$\{P(\text{x}), P(\text{y}), \neg Q(\text{y,z})\}$ no es una cláusula de horn

$\{P(\text{x}), \neg P(\text{y}), \neg Q(\text{y,z})\}$ es una cláusula de horn

Cláusulas de definición:

Son cláusulas de la forma $\forall X_1 \ ... \ \forall X_m . \ C$ tal que la disyunción de literales C tiene **exactamente un** literal positivo.

Cláusulas objetivo:

Son cláusulas de la forma $\forall X_1 \ ... \ \forall X_m . \ C$ tal que la disyunción de literales C **no tiene ningún** literal positivo.

Cláusulas de entrada:

Es un conjunto de cláusulas de Horn de la forma $\boxed{H = P \cup \{G\}}$

P: conjunto de cláusulas de definición. (se lo llama **programa** o **base de conocimiento**)

C: cláusula objetivo. (se lo llama **goal** o **meta**)

Secuencia de pasos para resolución SLD

Una secuencia de pasos de resolución SLD para un conjunto de cláusulas de Horn H (cláusulas de entrada) es una secuencia de **cláusulas objetivo** de la forma: $< N_0, N_1, ..., N_p >$ tal que satisfacen las siguientes condiciones:

1. $N_0 \in H$ (N_0 es la única cláusula objetivo de H).

2. Para todo $0 < \mathrm{i} < \mathrm{p}$, si N_i es:

$$\{\neg A_1, ..., \neg A_{k-1}, \neg A_k, \neg A_{k+1}, ..., \neg A_n\} \text{o}$$

Entonces hay
alguna cláusula de definición C_i de la forma $\{A, \neg B_1, ..., \neg B_m\}$ en H, tal que A_k y A son unificables con MGU S y $N_{\mathrm{i}+1}$ es:

$$S\{(\neg A_1, ..., \neg B_1, ..., \neg B_m, \neg A_{k+1}, ..., \neg A_n)\}$$

En resumen:

1. Resolvemos la cláusula objetivo N_i con las cláusulas de definición en H.
2. El resultado es otra cláusula objetivo $N_{\mathrm{i}+1}$.
3. Repetimos el proceso hasta llegar a $N_\mathrm{p} = \varnothing$.
4. El resultado es la sustitución compuesta de todas las sustituciones que se fueron realizando

La regla es la siguiente:

$$\frac{\overbrace{\{A, \neg B_1, ..., \neg B_n\}}^{\text{definición}} \quad \overbrace{\{\neg A_1, ..., \neg A_{k-1}, \neg A_k, \neg A_{k+1}, ..., \neg A_m\}}^{\text{objetivo}}}{\underbrace{S(\{\neg A_1, ..., \neg A_{k-1}, B_1, ..., \neg B_n, \neg A_{k+1}, ..., \neg A_m\})}_{\text{nuevo objetivo}}}$$

Con S = $\mathrm{mgu}\{A \stackrel{?}{=} A_k\}$

La resolución es SLD cuando:

1. La estrategia es lineal (utilizando en cada paso el resolvente obtenido en el paso anterior)
2. Se usan solamente cláusulas de Horn.
3. Se empieza por una cláusula objetivo.
4. La selección es binaria (un literal de cada cláusula)
5. La resolvente (nuevo objetivo) es una nueva cláusula objetivo

Prolog

Semántica de Prolog

Un programa en Prolog es un conjunto de **cláusulas de definición**.
Una consulta en Prolog es una
cláusula objetivo.

```
% Ejemplo:
% Para las siguientes clausulas:

% {a(0,X,X)}
% {a(s(X),Y,s(Z)) , ¬a(X,Y,Z)}

% {¬a(s(0),X,s(s(0)))}      %% goal
```

```
% el programa en prolog es el siguiente:
a(0,X,X).
a(s(X),Y,s(Z))) :- a(X,Y,Z).

?- a(s(0),X,s(s(0)))        %% consulta
```

La ejecución de un programa se basa en la **resolución SLD.**
Criterio de búsqueda:
busca las cláusulas objetivo por orden de aparición (desde la primera hasta la última) haciendo DFS.
Criterio de selección:
elegir siempre el primer literal de la cláusula objetivo elegida.

Predicados

sort, msort, length, nth1, nth0, member, append, last, between, is_list, list_to_set, is_set, union, intersection, subset, subtract, select, delete, reverse, atom, number, numlist, sum_list, flatten.

```
sort(+List, -Sorted)
% ordena List de forma creciente, remueve duplicados
msort(+List, -Sorted)
% igual a sort pero no elimina repetidos
length(?List, ?Length)
% true si Length representa la cantidad de elementos en List
nth1(?Index, ?List, ?Elem)
% true si Elem es el n(Index)-esimo elemento de List
nth0(?Index, ?List, ?Elem)
% igual que nth1 pero indexa desde cero
member(?Elem, ?List)
% true si Elem pertenece a List
append(+ListOfLists, ?List)
%concatena en List todas la listas que pertenecen a ListOfLists
append(?List1, ?List2, ?List1AndList2)
% List1AndList2 es la concatenacion de List1 y List2
between(+Low, +High, ?Value)
% Value se unifica con todos los valores entre Low y High
subset(+SubSet, +Set)
% true si SubSet es un subconjunto de Set (todo elemento en SubSet pertenece
subtract(+Set, +Delete, -Result)
% elimina todos los elementos de Delete que se encuentren en Set (todas las a
select(?Elem, ?List1, ?List2)
% true si List2 resulta en eliminar una aparicion de Elem en List1
atom(@Term)
% true si Term esta ligado a un átomo
numlist(+Low, +High, -List)
% List es una lista creciente [Low, Low+1, ..., High]
sum_list(+List, -Sum)
% Sum es la suma de todos los elementos de List
```

```
flatten(+NestedList, -FlatList)
% FlatList es el resultado de aplanar la lista de listas NestedList
```

Motor aritmético de Prolog

Predicados:

```
between(+Low, +High, ?Value)
% Value se unifica con todos los valores entre Low y High
succ(?Int1, ?Int2)
% true si Int2 = Int1 +1 , alguno de los dos arg. debe estar instanciado
plus(?Int1, ?Int2, ?Int3)
% true si Int3 = Int1 + Int2, al menos 2 de los 3 arg. debe estar instanciado
divmod(+Dividend, +Divisor, -Quotient, -Remainder)
% computa el cociente y resto de Dividend/Divisor
```

Operadores

```
% Evaluan de ambos lados las expresiones
+Expr1 > +Expr2
+Expr1 < +Expr2
+Expr1 =< +Expr2
+Expr1 >= +Expr2
+Expr1 =\= +Expr2
+Expr1 =:= +Expr2

-Number is +Expr
% true si Number es el valor el cual Expr evalúa
% solo se evalúa Expr

+IntExpr1 mod +IntExpr2
% Res = IntExpr1 modulo IntExpr2
+IntExpr1 rem +IntExpr2
% Res = resto de la IntExpr1 // IntExpr2
+IntExpr1 // +IntExpr2
% division entera
+IntExpr1 gcd +IntExpr2
% maximo comun divisor
+IntExpr1 lcm +IntExpr2
% minimo comun multiplo
abs(+Expr)
% valor absoluto
max(+Expr1, +Expr2)
min(+Expr1, +Expr2)
```

Meta-predicados

```
not(:Goal)
% true si Goal no puede ser satisfacido
```

Smalltalk

Objetos

Un objeto es una entidad con:

- **Estado**
- **Comportamiento**
- **Identidad**

Ejemplo:

Un objeto 'archivo' es una entidad que consta de:

- Estado: dirección, contenido
- Comportamiento: mostrar contenido, cambiar su contenido, renombrar
- Identidad: cada archivo es distinto uno del otro

Mensajes

Los objetos se comunican a través de mensajes, cada mensaje invoca un método.

Distintos objetos pueden "**entender**" el mismo mensaje, pero pueden tener distinta respuesta a determinado mensaje.

```
aWindow open
aFile open
aZip open
```

Un mensaje tiene **receptores** y **colaboradores.**

```
objeto mensaje: colaborador

!Ejemplos:
aCar goTo: aLocation
aWindow open
aFile open
```

Métodos

Un método es una secuencia de instrucciones a ser ejecutadas por el receptor de un mensaje. Operan con los datos encapsulados en los objetos y pueden modificar el estado de un objeto o realizar otras operaciones.

```
Clase >> nombre_método
   ...
```

```
    ... (def del metodo)
```

En resumen:

Mensaje: Indican a un objeto **QUE** hacer.

Método: Indican a un objeto **COMO** implementar determinado mensaje.

Clases

Una clase es una **plantilla** o **molde** a partir de la cual se crean instancias de un objeto.
Definen las
variables y el **comportamiento** de todas sus instancias.
Una clase puede
heredar características de otras clases.
Las clases también son objetos, por lo que pueden recibir mensajes y tener su propio comportamiento.

```
-- La respuesta de la clase Car al mensaje new es una nueva instancia de Car
Car new
>>> aCar
```

Pseudo-variables

Son variables que vienen predefinidas en Smalltalk y no pueden ser redefinidas.

self: se refiere siempre al objeto del método en donde se define, se pueden enviar mensajes a self y puede ser enviado como colaborador de otros mensajes
Ejemplo:

```
self computeTaxes

phoneBook at:self put: i
```

super: se refiere al mismo objeto que self (self == super). La diferencia está en que cuando super recibe un mensaje, la búsqueda del método empieza en la superclase de la clase que contiene el método que se está definiendo.
Ejemplo:

Cuando se envíe el mensaje show a una instancia de BorderedPanedWindow, va a buscar el método en su propia clase, como no lo encuentra, sigue buscando en su superclase (PanedWindow), donde efectivamente lo encuentra.
Evalúa
super show, buscando el método show primero en la superclase de PanedWindow (el objeto receptor sigue siendo aBorderedPanedWindow).
Encuentra show en Window y evalúa self drawBorder (Ahora la búsqueda empieza en la clase del objeto al que se refiere self, o sea, BorderedPanedWindow).

```
-- Jerarquía:
-- Window --> PanedWindow --> BorderedPanedWindow
```

```
Window
    show
        self drawBorder
        self showTitle

PanedWindow
    show
        super show
        self showPanes

BorderedPanedWindow
    drawBorder
```

nil: es la única instancia de la clase UndefinedObject. Todas las variables de instancias, clases, y variables locales son inicializadas como **nil.**

Block closures

Un bloque es una **función anónima.** Es evaluada al enviarle el mensaje **#value,** #**value**:, **#value:value:, #value:value:value, #value:value:value:value** (solo hasta cuatro parámetros) o **#valueWithArguments:** (cualquier número de parámetros, el colaborador es un array de parametros). También pueden ser pasados como colaboradores de otros mensajes (como cualquier objeto)

Ejemplos:

```
[:x | 1 + x] value: 2
-> 3

|z|
[:x :y | z:= x + y. z+1] value:3 value:5
-> 9

| n m |
n := 60
m := 45
(1 to n) select: [:d | n//d == 0].
-> #(1 2 3 4 5 6 10 12 15 20 30 60)
```

Condicionales y estructuras de control

- **If**
  Hay tres mensajes para el condicional if:
  #ifTrue: - #ifFalse: - #ifTrue:ifFalse:

  Los colaboradores de los tres mensajes deben ser block closures.
  Ejemplo:

```
-- Codigo en java/c++
if(x>y){
    max = x;
    i = j
}else{
    max = y;
    i = k
}

-- Codigo en smalltalk
(x>y) ifTrue: [max := x. i := k] ifFalse : [max := y. i := k]
```

- **While**

  Dos mensajes: **#whileTrue:, #whileFalse:**

  La expresión condicional tiene que estar en un bloque debido a que en cada iteración vamos a tener que reevaluarla.

```
-- Codigo en c++/java
while(i<100){
    sum = sum + 1;
    i = i + 1;
}

-- Codigo en smalltalk
[i < 100] whileTrue: [sum := sum + i. i := i+1]
```

- For
  Dos tipos de mensajes (aunque hay más mensajes que implementan estructuras similares):

  #to:do: - #timesRepeat:

```
-- Codigo en c++/java
for(i = 0; i<n; i++){
    sum = sum + i
}

-- Codigo en smalltalk
|sum n|
sum := 0
n := 5
1 to: n do: [:i | sum := sum + i]

sum := 0
```

```
n:= 1
5 timesRepeat: [sum := sum+n. n := n + 1]
```

Colecciones

```
Collection
    |__ Bag                     -- coleccion no-ordenada - permite repetidos
    |__ Set                     -- coleccion no-ordenada - no hay repetidos
    |   |__ Dictionary
    |
    |__ SequenceableCollection  -- collecciones ordenadas
            |__ SortedCollection    -- internamente ordenadas
            |__ OrderedCollection   -- ordenadas por aparicion
            |__ Interval
            |__ ArrayedCollection   -- ordenadas por indice
                    |__ Array
                    |__ String
```

Principales métodos en la clase Collection:

```
add: newObject
    "añade newObject al receptor del msj"

addAll : aCollection
    "añade todos los objetos en aCollection al receptor del msj"

remove: oldObj
    "elimina oldObj del receptor del msj"

remove: oldObj ifAbsent: aBlock
    "elimina oldObj, evalua aBlock si no encuentra oldObj en el receptor del

removeAll: aCollection
    "elimina todos los objetos de aCollection en el receptor del msj"

isEmpty
    "true si la colleccion está vacía"

size
    "devuelve la cantidad de objetos en la colección"


do: aBlock
    "aplica aBlock para cada elemento"

collect: aBlock
    "devuelve una nueva coleccion aplicando aBlock a cada elemento"
```

```
select: aBlock
    "devuelve una nueva colleccion con los elementos que devuelven true segun

otros: reject, inject, detect
```

Ejemplos:

```
|col arr sum|
col := OrderedCollection new.

col size.
    -> 0

col add: 4; add: 5.
col
    -> #(4 5)

sum := 0.
#(1 3 4) do: [:e | sum := sum + e].
sum.
    -> 8

#(1 2 3) collect: [:e | e + 10].
    -> #(11 12 13)

#(47 29 12 1 89 37845 12 94 58) select: [:e | e \\ 2 == 0].
    -> #(12 12 94 58)

arr := Array new: 5
arr at: 1 put: 23
arr
    -> #(23 nil nil nil nil)
```