Modelos y Paradigmas de Programación clase 1

Camilo Rueda

Propósito

La programación como disciplina de ingeniería

- Existe una teoría
- Los que practican la disciplina usan la teoría para predecir el resultado de sus acciones
- La teoría usa conceptos cercanos a la práctica de la profesión
- Ejemplo: Ingeniería civil
  - Teoría: estática, dinámica, resistencia de materiales
  - Permiten chequear el diseño de un puente y optimizarlo
  - Permiten entender el diseño de un puente hecho por otra persona

Teorías en computación

Las informática empezó en las matemáticas. Varias teorías entonces, derivadas de la noción de *computabilidad*:

- Máquinas de Turing
- Cálculo lambda
- π-Cálculo

Muy utilizadas entre la comunidad de investigadores en Ciencias de la computación.

- Ventajas: Rigor semántico, generalidad
- Desventajas: Alejadas de la *práctica*

Teorías de programación “prácticas”

Existe alguna teoría de la programación para el programador?

Teorías de programación “prácticas”

Existe alguna teoría de la programación para el programador?
Al menos dos:

- Basada en la noción de transformadores de predicados (Dijkstra)
- Basada en la noción de lenguaje de núcleo (“kernel”)

Principios:

- Dijkstra:
  - Sintaxis: Un programa es una secuencia de instrucciones
    $< st >::=< st >; < st > \mid < var >:=< exp > \mid \mathbf{if}\ ..\mathbf{fi} \mid \mathbf{while}...\mathbf{od}$
  - Semántica: Precondición, poscondición, invariante
    $wp([x := x + y],\ x < y + 1) = (x < 1)$

Lenguaje de núcleo

Un lenguaje de núcleo (KL) debe ser:

- Turing completo
- Implementable e implementado
- Simple
- Modular (o “incrementable”)
- Realísticamente cercano a lenguajes existentes.
- Semántica precisa.

Nombre de moda: “framework”

Por qué este y no aquel?

- Por ser dispositivo de cómputo: refuerza conceptos teóricos viendo su realización práctica
- Puede cubrir varios paradigmas de programación
- Puede construírse incrementable para cubrir distintos *modelos de computación*

Argumento de última moda: La arquitectura .NET debe permitir construír aplicaciones con múltiples lenguajes. El kernel Oz:

- Captura la esencia de los principales paradigmas
- Y de la computación multihilo
- Luego:
  Ofrecería la teoría ideal para programación en .NET

El KL Oz (para recordar)

$< st >::=$

$< st_1 >< st_2 >$

local $< var >$ **in** $< st >$ **end**

if $< var >$ **then** $< st_1 >$ **else** $< st_2 >$ **end**

case $< var >$ **of** $< patron >$ **then** $< st_1 >$ **else** $< st_2 >$ **end**

$\{< var >< arg_1 > \ldots < arg_n >\}$

Por qué no hay declaración de procedimientos?

El KL Oz (para recordar)

$< st >::=$

$< st_1 >< st_2 >$

local $< var >$ **in** $< st >$ **end**

if $< var >$ **then** $< st_1 >$ **else** $< st_2 >$ **end**

case $< var >$ **of** $< patron >$ **then** $< st_1 >$ **else** $< st_2 >$ **end**

$\{< var >< arg_1 > \ldots < arg_n >\}$

Por qué no hay declaración de procedimientos?

Porque son “de primera clase” (valores):

$< valor >::=< numero > \mid < registro > \mid < proced >$

$< proced >::=$ **proc** $\{\$\ arg_1 > \ldots < arg_n >\}$ $< st >$ **end**

Extensiones

El anterior es el modelo de computación *declarativo* (secuencial). Pero hay más modelos, que requieren nuevas instrucciones del KL:

- Concurrente (declarativo)
  $< st >::= \ldots \mathbf{thread} < st > \mathbf{end}$
- Manejo de errores
  $< st >::= \ldots \mathbf{try} < st_1 > \mathbf{catch} < var > \mathbf{then} < st_2 > \mathbf{end}$
  $\mathbf{raise} < var > \mathbf{end}$
- Estado explícito
  $< st >::= \ldots \{NewCell\, C\, X\} \mid \{Access\, C\, X\} \mid \{Assign\, C\, X\}$
- Programación orientada-Objetos: Agregar *clase* y *objeto*

Modelos de computación (versión PVR&SH

- Declarativo
  - Abstracción procedimental y recursión
  - Evaluación perezosa
  - Programación de alto orden
  - Lenguajes: ML (caML)
- Estado explícito (imperativo)
  - Funciones con memoria
  - Tipos abstractos de datos
  - Lenguajes: C
- Concurrente declarativo o dirigido por datos
  - Programación multi-hilos
  - Flujos (streams)
  - Sincronización de procesos
  - Lenguajes: Haskell

Modelos (2)

- Concurrencia por paso de mensaje
  - Asincronía
  - Puertos
  - Agentes
  - Lenguajes: Earlang
- Orientado-objetos
  - Herencia
  - Clase, objeto
  - Lenguajes: Java, Eiffel, C++, et al.
- Concurrencia de estado compartido
  - Candados
  - Monitores
  - Transacciones
  - Lenguajes: Java concurrente

Modelos (3)

- Relacional
  - Variables entrada/salida (dependiente del llamado)
  - Escogencia (“choice”)
  - Programación en lógica
  - Bases de datos y parsing
  - Lenguajes: Prolog

Metodología

Conocer cómo razonar formalmente sobre un programa concurrente

y

Saber cómo implementar un programa concurrente usando el lenguaje de núcleo

Programa concurrente vs secuencial

- Secuencial:
  - Observables: Parejas de estados iniciales y finales i.e. Valores de las variables de entrada/salida
  - El cómo se llega al estado final no importa
  - Son unidades atómicas
- Concurrente:
  - Estados intermedios son tan importantes como los finales (sincronización)
  - Observables:
    - Conjunto de variables que ocurren en un componente
    - Conjunto de variables con las que se comunica

Semántica de la concurrencia

traza: una secuencia particular de transiciones atómicas de estado

$$\sigma_0 \rightarrow^{\alpha_1} \sigma_1 \rightarrow^{\alpha_2} \ldots \rightarrow^{\alpha_i} \sigma_i \rightarrow^{\alpha_{i+1}} \sigma_{i+1}$$

- σ_i : Estados
- α_i : acciones atómicas
- $\alpha_1\alpha_2\ldots$ es un intercalamiento (*interleaving*). Corresponde a una historia del proceso

Semántica de un programa:
conjunto de todas sus historias posibles

Cómo razonar sobre prog. concurrentes?

- Inspeccionar cada historia?

Cómo razonar sobre prog. concurrentes?

- Inspeccionar cada historia? son demasiadas!!

Cómo razonar sobre prog. concurrentes?

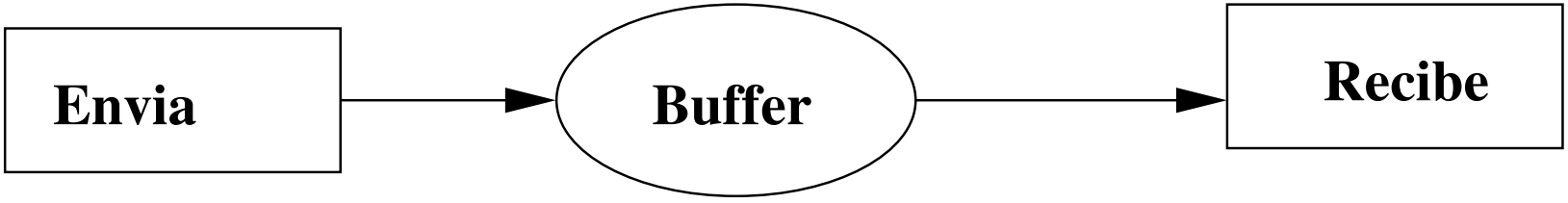
- Inspeccionar cada historia? son demasiadas!!
- Mejor:
  Representar conjuntos enteros de trazas por sus propiedades
  un predicado representa un conjunto de estados

  Un programa es una relación entre predicados

Cómo involucrar sincronización?

- La comunicación permite a un proceso influir sobre otro
- Posibilidades:
  - Variables compartidas
  - Paso de mensajes (canales compartidos)
- Esquemas de sincronización:
  - Exclusión mutua
  - Sincronización condicional
  - Comunicación síncrona

Ejemplo



- Exclusión mutua entre envío y recepción
- Sincronización condicional: Evitar que el mismo mensaje se envíe dos veces.
- Protocolo de comunicación: sincronía.

Dificultades: Exclusión mutua

$P_1 \equiv$	$P_2 \equiv$
l_0 : **loop forever do**	m_0 : **loop forever do**
l_1 : sección no crítica	m_1 : sección no crítica
l_2 : $y_1, s := 1, 1;$	m_2 : $y_1, s := 1, 2;$
l_3 : **wait** $(y_2 = 0) \vee (s \neq 1);$	m_3 : **wait** $(y_1 = 0) \vee (s \neq 2);$
l_4 : sección crítica	m_4 : sección crítica
l_5 : $y_1 := 0$	m_5 : $y_2 := 0$
od	**od**

$$Ej1 \equiv s := 1; y_1, y_2 := 0, 0; [P_1 \,||\, P_2]$$

Funciona? Cómo estar seguro? Por qué dos variables?

Ejemplo: Exclusión mutua(2)

$P_1 \equiv$	$P_2 \equiv$
l_0 : **loop forever do**	m_0 : **loop forever do**
l_1 : sección no crítica	m_1 : sección no crítica
l_2 : $s := 1;$	m_2 : $s := 2;$
l_3 : $y_1 := 1;$	m_3 : $y_2 := 1;$
l_4 : **wait** $(y_2 = 0) \vee (s \neq 1);$	m_4 : **wait** $(y_1 = 0) \vee (s \neq 2);$
l_5 : sección crítica	m_5 : sección crítica
l_6 : $y_1 := 0$	m_6 : $y_2 := 0$
od	**od**

$$Ej1 \equiv s := 1; y_1, y_2 := 0, 0; [P_1 \,||\, P_2]$$

No funciona? Exactamente por qué?

Ejemplo: Exclusión mutua(2)

$P_1 \equiv$	$P_2 \equiv$
l_0 : **loop forever do**	m_0 : **loop forever do**
l_1 : sección no crítica	m_1 : sección no crítica
l_2 : $y_1 := 1;$	m_2 : $y_2 := 1;$
l_3 : $s := 1;$	m_3 : $s := 2;$
l_4 : **wait** $(y_2 = 0) \vee (s \neq 1);$	m_4 : **wait** $(y_1 = 0) \vee (s \neq 2);$
l_5 : sección crítica	m_5 : sección crítica
l_6 : $y_1 := 0$	m_6 : $y_2 := 0$
od	**od**

$$Ej1 \equiv s := 1; y_1, y_2 := 0, 0; [P_1 \,||\, P_2]$$

funciona? Exactamente por qué?

Ejemplo: Exclusión mutua(3)

$P_i \equiv$

l_0 : **loop forever do**

l_1 : sección no crítica

l_2 : $flag[i] := 1;$

l_3 : **wait** $\forall j : 0 \leq j < n \wedge j \neq i. (flag[j] = 0);$

l_4 : sección crítica

l_5 : $flag[i] := 0$

od

mutex1 $\equiv flag := 0; [P_1 \,||\, P_2 \,||\, ... || \, P_n]$

Funciona? por qué?

Ejemplo: Exclusión mutua(4)

$P_i \equiv$

l_0 : **loop forever do**

l_1 : sección no crítica

l_2 : $flag[i] := 1;$

l_3 : **wait** $\forall j : 0 \leq j < n \wedge j \neq i. (flag[j] = 0 \vee flag[j] = 1);$

l_4 : $flag[i] := 4;$

l_5 : **wait** $\forall j : 0 \leq j < i \wedge j \neq i. (flag[j] = 0 \vee flag[j] = 1);$

l_4 : sección crítica

l_5 : $flag[i] := 0$

od

mutex2 $\equiv flag := 0; [P_1 \,||\, P_2 \,||\, ... || \, P_n]$

Funciona? por qué?

Ejemplo: Exclusión mutua(4)

$P_i \equiv l_0 :$ **loop forever do**

$l_1 :$ sección no crítica

$l_2 : flag[i] := 1;$

$l_3 :$ **wait** $\forall j : 0 \leq j < n \wedge j \neq i. (flag[j] = 0 \vee flag[j] = 1 \vee flag[j] = 2);$

$l_4 :$ **if** $\exists j : 0 \leq j < n. (flag[j] = 1)$ **then**

$l_5 : flag[i] := 2; l_{5'} :$ **wait** $\exists j : 0 \leq j < n. (flag[j] = 4)$

$l_7 : flag[i] := 4;$

$l_8 :$ **wait** $\forall j : 0 \leq j < i \wedge j \neq i. (flag[j] = 0 \vee flag[j] = 1);$

$l_9 :$ sección crítica

$l_10 : flag[i] := 0$

od

tarea

Hacer las mínimas modificaciones de manera que el algoritmo anterior funcione. Razone cuidadosamente sobre la corrección del algoritmo.
Su algoritmo debe ser "justo".
Impleméntelo en su lenguaje de programación concurrente favorito. Córralo para 10 procesos.

Cómo pensar un Sistema concurrente?

Sistema: Algo pasa coordinadamente en el tiempo.
Cómo se da cuenta del paso del tiempo?

El mundo es:

Cómo pensar un Sistema concurrente?

Sistema: Algo pasa coordinadamente en el tiempo.
Cómo se da cuenta del paso del tiempo?

- Observa cambios.
- Qué cambia?

El mundo es:

Cómo pensar un Sistema concurrente?

Sistema: Algo pasa coordinadamente en el tiempo.
Cómo se da cuenta del paso del tiempo?

- Observa cambios.
- Qué cambia? Mi información sobre el sistema
- Qué causa el cambio?

El mundo es:

Cómo pensar un Sistema concurrente?

Sistema: Algo pasa coordinadamente en el tiempo.
Cómo se da cuenta del paso del tiempo?

- Observa cambios.
- Qué cambia? Mi información sobre el sistema
- Qué causa el cambio?
  Un evento

El mundo es:

Cómo pensar un Sistema concurrente?

Sistema: Algo pasa coordinadamente en el tiempo.
Cómo se da cuenta del paso del tiempo?

- Observa cambios.
- Qué cambia? Mi información sobre el sistema
- Qué causa el cambio?
  Un evento

El mundo es:

- Objetos
- Eventos
- Estado:
  mi información sobre acumulación de eventos

Diseñar un sistema concurrente (o no)

Diferenciar entre

- Modelar:
  Construir un sistema con el objeto de probar sus propiedades
- Implementar:
  Construir un sistema para simular u observar comportamiento

  La clave:
  Cómo establecer una relación precisa entre ambos

Metodología de diseño

- Construír uno (o varios) modelos
- Usar lenguaje matemático para razonar sobre propiedades
- Construir refinamientos del modelo
- Establecer criterios para probar coherencia entre ambos

Cómo diseñar?

Paradigma del paracaidas:

- Primero ubicarse mentalmente muy por encima del sistema y observar:
  - Objetos (personas, buses, canales, mensajes)
  - Eventos: Movimientos de personas, buses, etc
- Representar los anteriores en un formalismo (modelar)
- Inferir una propiedad que determina la coherencia de lo que se observa:
  Un invariante del sistema
- Asegurarse de que la evolución del sistema mantiene el invariante
- Comprobar que los eventos no se bloquean (deadlock
- Comprobar justicia (fairness) en ejecución de eventos

Guía

- Empezar con un modelo simple (estados+eventos)
- Volverlo más preciso: Refinar
- Introducir control y comunicación
- Repetir 2,3
- Descomponer
- Repetir 2,3,4,5
- Probar formalmente las propiedades

El formalismo: Eventos

xxx=

ANY $x, y, z, ...$ **WHERE**

$P(x, y, ..., v, w, ...)$

THEN

$S(x, y, ..., v, w, ...)$

END

Para cualquier $(x, y, ...)$ tales que $P(x, y, ..., v, w, ...)$ entonces

Nuevo estado es $S(x, y, ..., v, w, ...)$

Eventos (2)

```
XXX=
        SELECT
            P(v, w, ...)
        THEN
        S(v, w, ...)
        END
```

Cuando $P(v, w, ...)$ es cierto, el nuevo estado es $S(v, w, ...)$

S es de la forma $a, b, ... := E, F, ...$

Prueba de consistencia

XXX=

ANY x **WHERE**

$P(x, v)$

THEN

$v := E(x, v)$

END

$$\exists x. \, (\, I(v) \wedge P(x, v) \Rightarrow I(E(x, v)) \,)$$

Prueba de consistencia (2)

```
XXX=
        SELECT
          P(v, w)
        THEN
        v := E(v, w)
        END
```

$$I(v,w) \wedge P(v,w) \Rightarrow I(E(v,w))$$

Ausencia de bloqueo

XXX=	XXX=
SELECT	**SELECT**
$P_1(v, w)$	$P_2(v, w)$
THEN	**THEN**
$v := E_1(v, w)$	$v := E_2(v, w)$
END	**END**

$$I(v, w) \Rightarrow P_1(v, w) \vee P_2(v, w)$$

Justicia

Ausencia de ciclos infinitos:

- $I(x) \Rightarrow V(x) \geq 0$
- $I(x) \wedge G(x) \Rightarrow V(E(x)) < V(x)$

Es decir,

Ningún evento acapara siempre el control

Ejemplo: exclusión mutua

El problema (especificación):

- Los procesos pertenecen a un conjunto P
- cada proceso $x \in P$ hace un ciclo infinito en las fases:
  - x está en una zona no crítica
  - x está en una zona pre-crítica
    i.e. compite con otros procesos para entrar en zona crítica
  - x está en la zona crítica

Los objetos son, en este caso, procesos

Transiciones entre fases

Los eventos ocasionan transiciones. Tipos de eventos:

- ask: transición no-crítica $\longrightarrow$ pre-crítica
- entrar: transición pre-crítica $\longrightarrow$ crítica
- salir: transición crítica $\longrightarrow$ no-crítica

Restricciones:

- Un proceso en la zona pre-crítica no puede bloquear procesos en zona no-crítica
- Ningún proceso se queda esperando por siempre en zona pre-crítica.
- la zona crítica contiene a lo sumo un proceso

Especificación: invariante

- p es el conjunto de procesos en zona pre-crítica o crítica.
- c es el conjunto de procesos en zona crítica.

$$
\begin{array}{|l|}
\hline
p \subseteq P \\
c \subseteq p \\
\forall (x, y).\, (x \in c \wedge y \in c \Rightarrow x = y) \\
\hline
\end{array}
$$

Invariante(2)

- Una relación r de precedencia entre procesos en P y procesos en p
- Si (x, y) está en r, entonces el proceso x no puede entrar a zona crítica antes que y.
- Lo anterior se cumple si la última vez que x visitó la zona crítica, y estaba ya en la precrítica.

$$\boxed{\begin{array}{l} r \in P \leftrightarrow p \\ r \circ r \subseteq r \\ r \cap r^{-1} = \emptyset \end{array}}$$

Eventos

ask $=$

ANY x **WHERE**

$x \in P - p$

THEN

$p := p \cup \{x\}$

END

entrar $=$

ANY x **WHERE**

$x \in p \wedge c = \emptyset \wedge x \notin dom(r)$

THEN

$c := \{x\}$

END

salir $=$

ANY x **WHERE** $x \in c$ **THEN**

$c := \emptyset \parallel p := p - \{x\}$

$\parallel r := (r - (P \times \{x\})) \cup (\{x\} \times (p - \{x\})$

END

Del modelo a la implementación

Se trata de refinar la especificación:

- Reemplazar construcciones abstractas por otras "programables"
- Concretar escogencias abstractas

Por ejemplo:
Cómo hacer el test $c = \emptyset$?

Una opción: reemplazar la relación de orden parcial r por un orden total. El mínimo en este orden es el proceso en la zona crítica.

Un refinamiento

- Los procesos se comunican con una única agencia central
- La agencia escoge quién entra a zona crítica
- Hay un canal "injusto" que comunica procesos con la agencia
- La agencia tiene una puerta y una bodega ("justa") de procesos
- La bodega contiene procesos en espera para entrar en zona crítica
- Cuando la puerta está abierta, la agencia lee el canal y transfiere un proceso a la bodega
- Cuando está cerrada, la agencia escoge de la bodega un proceso para zona crítica.

Funcionamiento

El conjunto p particionado en dos: cn, el canal y bd, la bodega (que incluye a c).

- Un proceso entra a zona pre-crítica mediante el evento *ask* (como antes). Se agrega a cn.
- La agencia lee el canal mediante el evento *pedir* (cuando la puerta está abierta).
  Se transfiera a alguien de cn a bd.
- A veces, cuando bd no está vacío y c está vacío, la agencia cierra la puerta, mediante el evento *loop1*.
- Antes de cerrar la puerta se asegura de que cn esté vacío.

Funcionamiento (2)

- Mientras la puerta esté cerrada:
  - La agencia realiza su escogencia.
  - Los procesos continuan enviando pedidos al canal, pero la agencia no ve esto.
- Una vez hecha la escogencia, la agencia abre la puerta, mediante el evento *entrar*
- La agencia usa la misma relación r. La agencia debe entonces:
  - Cuando x sale de zona crítica, nuevas relaciones deben establecer con procesos en bd (fácil)
  - Nuevas relaciones deben establecerse con procesos en cn (cómo hacer esto???)

Especificación

$$
\begin{array}{|l|}
\hline
cn \subseteq P \\
bd \subseteq P \\
pr \in \{abierta, cerrada\} \\
p = cn \cup bd \\
cn \cap bd = \emptyset \\
\hline
\end{array}
\qquad
\begin{array}{|l|}
\hline
pr = cerrada \Rightarrow c = \emptyset \\
pr = cerrada \Rightarrow bd \neq \emptyset \\
pr = cerrada \Rightarrow cn \cap ran(r) = \emptyset \\
\hline
\end{array}
$$

Observe la fuerte restricción sobre precedencia de procesos en cn cuando la puerta está cerrada.

Eventos

ask $=$

ANY x **WHERE**

$x \in P - p$

THEN

$p := p \cup \{x\}$

$cn := cn \cup \{x\}$

END

entrar $=$

ANY y **WHERE**

$y \in bd \wedge pr = cerrada$

$\wedge y \notin dom(r)$

THEN

$c := \{y\} \,||\, pr := abierta$

END

salir $=$

ANY x **WHERE** $x \in c$ **THEN**

$c := \emptyset \;\|\; p := p - \{x\} \,||\, bd := bd - \{x\}$

$||\, r := (r - (P \times \{x\})) \cup (\{x\} \times (p - \{x\})$

END

Eventos(2)

pedir $=$

ANY x **WHERE**

$x \in cn \wedge pr = abierta$

THEN

$bd := bd \cup \{x\} \,||$

$cn := cn - \{x\}$

END

loop1 $=$

SELECT

$c = \emptyset \wedge cn = \emptyset$

$bd \neq \emptyset \wedge pr = abierta$

THEN

$pr := cerrada$

END

Segundo refinamiento

Implementar la relación r mediante otra más débil

- Usamos números $1..n$ para identificar los procesos
- Una relación h implementa r usando precedencias sobre esos números dispuestos en un anillo

$$
\begin{array}{l}
h \in P \leftrightarrow P \\
\forall (x,y).\ \left(\begin{array}{l}
(x,y) \in h \\
\Leftrightarrow \\
(y = w \Rightarrow x \neq w) \wedge \\
(y > w \Rightarrow y < x \vee x \leq w) \wedge \\
(y < w \Rightarrow y < x \wedge x \leq w)
\end{array} \right) \\
r \subseteq h
\end{array}
$$

Eventos

entrar $=$

ANY y **WHERE**

$y \in bd \wedge pr = cerrada \wedge$

$w < y \Rightarrow (w + 1..y - 1) \cap bd = \emptyset \wedge$

$y \leq w \Rightarrow bd \subseteq y..w$

THEN

$c := \{y\} \,||\, pr := abierta \,||\, w := y$

END

salir $=$

ANY x **WHERE** $x \in c$ **THEN**

$c := \emptyset \;||\; p := p - \{x\} \,||\, bd := bd - \{x\}$

END