Algoritmos y Estructuras de Datos II

Tomás Agustín Hernández

1. Especificación

Consideraciones importantes / Reminders

- Utilizar operadores luego: Si estoy en LPO (Lógica de Primer Orden) utilizar los operadores luego si vemos que hay una posible indefinición como una división, o ingresar a una lista a un índice. Recordar que el para todo y un existe, aunque esté acotado por un rango, los cuantificadores predican IGUAL para todos los valores. Entonces, aunque diga que x es positivo, también probará dividir inclusive por 0 y estallará.
- Recordar las condiciones bidireccionales
  - Si por algún motivo tengo que armar una "lista", como, por ejemplo, los divisores de un número x tengo que indicar que, si el número divide a x, entonces ese número está en res, pero además todos los valores que están en res DIVIDEN a x. Es una condición bidireccional.
  - Otro ejemplo puede ser que tenga que considerar el máximo de una lista, si todos los valores y que están en la lista son menores que res entonces significa que res también pertenece a esa lista original.
- Recordar el significado de los cuantificadores con dos variables al mismo tiempo: En la lógica se ejecutan todos de uno a la vez. Es decir, si tengo que poner un para todo adentro de un para todo entonces hago un para todo solo con dos variables y listo.
- Recordar que cuando en un procedimiento llamo a un predicado y ese predicado devuelve algo de un para todo, existe (básicamente un valor de verdad) tengo que castear ese valor en el procedimiento porque son dos mundos distintos. Ej: asegura: res = True $\iff$ predicado
- Los predicados y funciones auxiliares no describen problemas. Son herramientas sintácticas para descomponer predicados.
  - Los procedimientos pueden llamar a funciones auxiliares o predicados. Un procedimiento no puede llamar a otro procedimiento.
  - Los predicados pueden llamar a predicados o auxiliares.
  - Las auxiliares solo pueden llamar auxiliares.
- No usamos nunca == en especificación, usamos siempre = y estamos comparando, no asignando.
- No existe el guardar o asignar en el mundo de la lógica. No puedo guardar en una lista en un índice específico porque si un valor. Para esto solemos usar que x valor pertenecerá a esta lista, por ejemplo.
- Si tengo un algoritmo que cumple una funcionalidad específica con un require más débil, puedo poner el require más restrictivo y va a funcionar igual pero NO al revés.

Fórmulas compuestas

Decimos que una fórmula es compuesta a una fórmula que tiene más de una operación y esa operación necesita realizarse antes de conocer su valor.

- $(p \land q) \lor m$
- $((p \land q) \lor m) \implies n$

Fórmula atómica

Decimos que una fórmula es atómica si se puede inferir su valor con una, o ninguna operación. Es irreducible.

- p
- $p \land q$

Fórmulas bien definidas

Decimos que una fórmula está bien definida cuando el orden que hay que hacer las operaciones es clara. Es decir, cuando cada operación toma dos variables proposicionales, y al realizar la operación termina siendo una fórmula atómica.

- $p \land q \lor r$ está mal formada. No se especifica si primero se realiza el $\land$ o el $\lor$.
- $(p \land q) \lor r$ está bien formada.
- $p \land q \land r \land m$ está bien formada porque son todas conjunciones.
- $p \lor q \lor r \lor m$ está bien formada porque son todas disyunciones.

Cuantificadores

- Para todo: $\forall$
  - *Garantiza la* conjunción : $p(1) \land p(2) \land p(3) \cdots \land p(m)$. Todos los casos deben ser true para que el cuantificador sea true.
  - Se acompaña por un $\longrightarrow$ a la hora de predicar sobre los elementos.
  - $(\forall i : \mathbb{Z})(0 \leq i < |s| \longrightarrow_L s[i] \ mod \ 2 = 0)$. Todos los elementos de la lista son divisibles por 2.
  - Estructura: $\forall$ + rango + $\longrightarrow_L$
- Existe: $\exists$
  - *Garantiza la* disyunción : $p(1) \lor p(2) \lor p(3) \cdots \lor p(m)$. Con un caso true el cuantificador es true.
  - Se acompaña por un $\land$ a la hora de predicar sobre los elementos.
  - $(\exists i : \mathbb{Z})(0 \leq i < |s| \land_L s[i] \geq 0)$. Existe algún elemento en la lista que es mayor o igual a 0.
  - $\exists$ + rango + $\land_L$

Equivalencias entre fórmulas

Decimos que dos fórmulas son equivalentes $\iff$ los valores de la tabla de verdad al aplicar la operación arroja el mismo resultado.

Valuaciones

Las valuaciones surgen en base a la tabla de verdad. Las valuaciones serian darle valor a las variables proposicionales y ver el resultado de la operación. Solo hacen referencias a fórmulas atómicas.

Tautologias, contradicciones y contingencias

- Una fórmula es tautología $\iff$ el resultado de la operación en cada fila arroja siempre V.
- Una fórmula es contradicción $\iff$ el resultado de la operación en cada fila arroja siempre F.
- Una fórmula es contradicción $\iff$ el resultado de la operación en cada fila arroja siempre V y F.

Relaciones de fuerza entre fórmulas

Decimos que una fórmula es más fuerte que la otra $\iff$ una fórmula es más restrictiva que la otra, o está incluida en la otra.
En el mundo de la lógica, decimos que A es más fuerte que B $\iff A \implies B$

- Si $(A \implies B)$ y $(B \implies A)$ son tautologías, entonces A y B son equivalentes.
- Si $(A \implies B)$ es tautología y $(B \not\Longrightarrow A)$ no es tautología, entonces decimos que A es más fuerte que B.
- Si $(A \not\Longrightarrow B)$ y $(B \not\Longrightarrow A)$ son contigencias, entonces no existe relación de fuerza entre A y B.

Algunos ejemplos:

- $|s| = 0 \implies |s| \geq 0$. En este caso vemos que $|s| = 0$ es más fuerte que $|s| \geq 0$ pues $|s| = 0$ está incluido en $|s| \geq 0$. Por lo tanto, $A \implies B$
- $|s| = 0 \implies |s| \geq 3$. En este caso vemos que $|s| = 0$ no es más fuerte que $|s| \geq 3$ pues $|s| = 0$ no está incluido en $|s| \geq 3$. Por lo tanto, $A \not\Longrightarrow B$
- $2 \leq i < |s| \implies 1 \leq i < |s|$. En este caso A $\implies$ B, pues i = 2 está incluido en el rango de B. Por lo tanto, $A \implies B$
- $0 \leq i < |s| \implies 1 \leq i < |s|$. En este caso A $\not\Longrightarrow$ B, pues el 0 de A no es parte de B. Por lo tanto, $A \not\Longrightarrow B$

Tipos de parámetros en especificacion

- in: Solo nos interesa el valor de entrada de una variable. No la vamos a modificar. Ya están inicializados
- out: Donde se retornará el resultado. No nos importa el valor inicial ni tampoco determina nada en nuestra función.
- inout: Necesitamos el valor original aunque lo terminamos modificando y devolviendo.

Lógica trivaluada

También llamada lógica secuencial porque se procesa de izquierda a derecha; Nos introduce los conceptos de $\wedge_L$ $\vee_L$ $\longrightarrow_L$ y el valor de indefinido $\perp$.

Se termina de evaluar una expresión cuando se puede deducir el valor de verdad.

Considere $x = \text{true} \wedge y = \perp \wedge z = \text{false}$

- $x \vee_L y$: Como el $\vee_L$ necesita uno solo para ser verdadero, entonces como x ya es true entonces toda la fórmula es verdadera.
- $x \wedge_L y$: Como el $\wedge_L$ necesita que ambas variables sean verdaderas, evalúa indefinido y el programa estalla.
- $\neg x \longrightarrow_L y$: Como el $\longrightarrow_L$ solo es falso si el antecedente es true y el consecuente false, como en este caso el antecedente ya es falso, toda la implicación es verdadera.
- $(x \wedge z) \wedge_L y$: Como el $\wedge_L$ necesita que ambas fórmulas sean true, en este caso, como $(x \wedge z)$ es falso, entonces ya toda la fórmula es falsa. Nótese que el $\wedge$ de la condición interna no contiene el luego porque jamás se indefinirá.
- $(\forall i : \mathbb{Z})(0 \leq i < |s| \longrightarrow_L s[i] \geq 0)$ Nótese que aquí usamos un $\longrightarrow_L$ porque podría ser que la lista esté indefinida o no exista el valor en s[i]

Predicados

- Viven en el mundo de la lógica.
- Nos sirven para poder modularizar nuestras especificaciones.
- Solamente devuelven valores de verdad True y False y es necesario castearlos en caso de querer devolver bool como tipo de dato.
- Los predicados pueden llamar a otros predicados o funciones auxiliares.
- Pueden utilizar cuantificadores.
- No tienen requiere ni asegura.
- No admite parámetros in, out, inout.

Ejemplo cuando tenemos que transformar el valor de verdad a tipo de dato:

```
pred divisiblePorDos (n: ℤ) {
    n mod 2 = 0
}
proc esMultiploDeDos (in n: ℤ) : Bool
      requiere {true}
      asegura {res = true ⟺ divisiblePorDos(n)}
```

Ejemplo usando un predicado sin necesidad de transformar el valor de verdad a tipo de dato:

```
   pred todosSonPares (l: seq⟨ℤ⟩) {
    (∀i : ℤ) (0 ≤ i < |l| ⟶_L l[i]mod2 = 0)
}
proc todosPares (in l: seq⟨ℤ⟩) : Bool
      requiere {todosSonPares(l)}
```

Funciones Auxiliares

- Son reemplazos sintácticos.
- Nos ayudan a modularizar las especificaciones.
- No pueden ser recursivas.
- Solo hacen cuentas.
- No pueden utilizar cuantificadores.
- Pueden llamar a predicados.
- Devuelven un tipo de dato.
- No tienen requiere ni asegura.
- No admite parámetros in, out, inout.

`aux sumar` (n: $\mathbb{Z}$, m: $\mathbb{Z}$) : $\mathbb{Z}$ $= n + m$;
`aux sumarTodos` (s: $seq\langle\mathbb{Z}\rangle$) : $\mathbb{Z}$ $= \sum_{i=0}^{|s|-1} s[i]$;

Aridad

Decimos que una función es de aridad n cuando la función recibe n cantidad de parámetros.

Variables Ligadas y Libres

Las variables son ligadas $\iff$ están dentro de un cuantificador mientras que son libres cuando no lo están.

- $(\forall i : \mathbb{Z})(0 \leq i < |s| \longrightarrow_L n \geq s[i])$ i es una variable ligada mientras que n y s son variables libres.
- $(\exists j : \mathbb{Z})(0 \leq j < |s| \wedge_L n \geq s[i])$ j es una variable ligada mientras que n y s son variables libres.
- $(\forall i : \mathbb{Z})(0 \leq i < |s| \longrightarrow_L n \geq s[i]) \wedge P(i)$ Ojo acá. i es una variable ligada, pero la i que está fuera del cuantificador $P(i)$ no está ligada. Esta última debería ser renombrada para no tener problemas y confusiones.

Cuando tenemos variables ligadas **no** podemos hacer nada sobre ellas, entre esas cosas, no podemos reemplazarlas porque no dependen de nosotros sino de los cuantificadores.

Cuantificadores anidados

Anidamos cuantificadores cuando el rango de las variables es exactamente el mismo.

- $(\forall i, j : \mathbb{Z})(0 \leq i, j < |s| \longrightarrow_L n \geq s[i][j]) \equiv (\forall i : \mathbb{Z})((0 \leq i < |s| \longrightarrow_L (\forall j : \mathbb{Z})(0 \leq j < |s| \longrightarrow_L n \geq s[i][j])))$

Estado

Llamamos estado a los valores de las variables en un punto de ejecución específico. El estado de un programa es importante porque muta al asignar valores a las variables. Cuando necesitamos hablar del estado de una variable en un instante específico, hablamos de **metavariables**

Metavariables

Llamamos metavariable a una variable en un instante dado. Es útil cuando tenemos que predicar como cambio el valor de una variable con respecto al inicial.
Cuando tenemos que utilizar metavariables, sea S una variable cualquiera podemos referirnos al instante de tiempo de S como S_t donde t indica el momento.

Notación $S = S_0$

`proc multiplicarPorDosAImpares` (inout l: $seq\langle\mathbb{Z}\rangle$)
`requiere` $\{l = l_0\}$
`asegura` $\{|l| = |l_0|\}$
`asegura` $\{(\forall i : \mathbb{Z})(0 \leq i < |s| \longrightarrow_L if(s_0[i] \ mod \ 2 \neq 0) \ then \ (s[i] = s_0[i] * 2) \ else \ (s[i] = s_0[i]) \ fi)\}$

Nota: Cuando utilizamos metavariables tenemos que indicar que al modificar algo directamente, si no modificamos todo el

conjunto de valores tenemos que indicar que los demás permanecen inalterados. En este caso, como estamos editando los valores, no tendría sentido que la lista salga con mayor longitud, es por eso que garantizamos que no cambia.

Otra manera de resolver el ejemplo anterior es utilizando old(s)

`proc multiplicarPorDosAImpares` (inout l: $seq\langle\mathbb{Z}\rangle$)
`asegura` $\{|l| = |old(l)|\}$
`asegura` $\{(\forall i : \mathbb{Z})(0 \leq i < |s| \longrightarrow_L if(old(s)[i] \ mod \ 2 \neq 0) \ then \ (s[i] = old(s)[i] * 2) \ else \ (s[i] = old(s)[i]) \ fi)\}$

Correctitud de un Programa

Decimos que un programa S es correcto respecto a una especificación si se cumple la precondición P, el programa termina su ejecución y se cumple la postcondición Q.

Tripla de Hoare

Notación para indicar que S es correcto respecto a la especificación (P, Q)

$$\{P\} \ S \ \{Q\}$$

SmallLang

Es un lenguaje que nos permitirá poder validar la correctitud de un programa. Solo tiene dos operaciones:

- x := E ≡ asignación
- skip ≡ no hace nada

Nota: E es una expresión cualquiera. Un valor, una función, cualquier cosa.

Estructuras de Control en SmallLang

- Secuencia de pasos: S1; S2 es un programa $\iff$ S1 y S2 son dos programas.
- Condicionales: if B then S1 else S2 endif es un programa $\iff$ B es una condición lógica (guarda) y S1 y S2 son programas.
- Ciclo: while B do S endwhile es un programa $\iff$ B es una condición lógica y S un programa.

Validez de una tripla de Hoare

$\{x \geq 4\} \ x := x + 1 \ \{x \geq 7\}$ Donde,

- P = $\{x \geq 4\}$
- S = $x := x + 1$
- Q = $\{x \geq 7\}$

¿Vale que $\{P\} \ S \ \{Q\}$? Solo vale $\iff$ $x \geq 6$ por lo tanto, como la precondición P falla en los casos de x = 4, x = 5 podemos decir que la tripla de Hoare no es válida.

Esto que acabamos de hacer se llama demostrar la correctitud de un programa, y acabamos de demostrar que la precondición P para el programa S es demasiado débil pues no nos garantiza que llegaremos a Q cumpliendo P.

Existe una manera formal que nos permite conocer la precondición más débil de un algoritmo.

Predicado def(E)

Dada una expresión E, llamamos def(E) a las condiciones para que E esté definida. Todas las constantes están definidas, por lo tanto def(x) ≡ True. La idea es ir separando en términos e ir colocando las definiciones necesarias para esa operación específica.

- $def(x+1) \equiv def(x) \wedge def(1) \equiv True \wedge True \equiv True$
- $def(x/y) \equiv def(x) \wedge (def(y) \wedge y \neq 0) \equiv True \wedge (True \wedge y > 0) \equiv y \neq 0$
- $def(\sqrt{x}) \equiv (def(x) \wedge x \geq 0)$
- $def(a[i]+3) \equiv (def(a) \wedge def(i)) \wedge_L 0 \leq i < |a| \wedge def(3) \equiv (True \wedge True) \wedge_L 0 \leq i < |a| \wedge True \equiv 0 \leq i < |a|$

Predicado Q_E^x

Cuando hablamos de este predicado hablamos de reemplazar las ocurrencias de x por E en el programa. Solo se reemplazan las ocurrencias libres, no las ligadas.

Axiomas

- Axioma 1: $wp(x := E, Q) \equiv def(E) \wedge_L Q_E^x$
- Axioma 2: $wp(skip, Q) \equiv Q$
- Axioma 3: $wp(S1; S2, Q) \equiv wp(S_1, wp(S_2, Q))$
- Axioma 4: $wp(S, Q) \equiv def(B) \wedge_L ((B \wedge wp(S_1, Q)) \vee (\neg B \wedge wp(S_2, Q)))$

Axioma 1 con secuencias

El axioma 1 nos sirve para asignar una expresión a una variable; Sin embargo, si tenemos que guardar algo en una secuencia debemos utilizar el setAt.

- $wp(b[i] := E, Q)$
  $\equiv def(setAt(b, i, E)) \wedge_L Q^{b[i]}_{setAt(b,i,E)}$
  $\equiv (def(b) \wedge def(i) \wedge def(E)) \wedge_L 0 \leq i < |b| \wedge_L Q^{b[i]}_{setAt(b,i,E)}$
  $\equiv 0 \leq i < |b| \wedge_L Q^{b[i]}_{setAt(b,i,E)}$
  $\equiv setAt(b, i, E)[j] = \{E \ si \ i = j, b[j] \ si \ i \neq j\}$

Algunos ejemplos:
$wp(s[i] := s[i-1], Q)$
$wp(setAt(s, i, s[i-1]), Q) \equiv$
$def(setAt(s, i, s[i-1])) \equiv (def(s) \wedge def(i)) \wedge_L 0 \leq i < |s| \wedge_L def(s[i-1]) \equiv$
$0 \leq i < |s| \wedge_L (def(s) \wedge def(i)) \wedge_L 0 \leq i - 1 < |s| \equiv 0 \leq i < |s| \wedge_L 1 \leq i < |s| + 1 \equiv 1 \leq i < |s|$

TODO: Luego mostrar un ejercicio y aclarar que por cada condición se separan n cuantificadores.

Precondición más débil (Weakest Precondition)

Es la precondición más débil que se necesita para poder ejecutar un algoritmo y satisfacer la postcondición Q.

Notación: $wp(S, Q)$ donde S es el programa y Q la postcondición.

Teorema: Una tripla de Hoare $\{P\}\ S\ \{Q\}$ es válida $\iff P \longrightarrow_L wp(S, Q)$.

Sea el siguiente enunciado, calcule la precondición más debil.

- P = $\{x \geq 4\}$
- S = $x := x + 1$
- Q = $\{x \geq 5\}$

$P \longrightarrow_L wp(S, Q) \equiv wp(x := x + 1, x \geq 5) \equiv def(x+1) \wedge_L Q^x_{x+1} \equiv def(x) \wedge_L def(1) \wedge_L x + 1 \geq 5$
$\equiv True \wedge_L True \wedge_L x \geq 4 \equiv x \geq 4$

Luego, $\{x \geq 4\} \longrightarrow_L \{x \geq 4\}$ es true
Por lo tanto, $wp(x := x + 1, x \geq 5) \equiv \{x \geq 4\}$
Finalmente, probamos que para poder satisfacer Q la precondición más debil que cumple P es cuaqluier $x \geq 4$.

Nota importante: Muchas veces puede ser que se nos solicite indicar que *wp* es incorrecta. Cuando se dice esto, significa que son precondiciones válidas pero hay algunas que no son la más débil.

Precondición más débil en ciclos

Consideremos el siguiente ejemplo $\{???\}S\{x = 0\}$ donde S es el siguiente programa

```
1  while(x>0) do
2      x := x–1
3  endwhile
```

Recordemos que esto es una tripla de Hoare, pero no podemos utilizar wp(S, Q) porque el programa es en base a ciclos. La precondición P más débil acá sería que x≥0 porque para cumplir la postcondición Q me da igual si entra al ciclo o no. Eso tiene que quedar siempre claro, cuando estamos hablando en ciclos, si no entra al ciclo y satisface igual ya está.

Predicado H_k

Definimos el predicado H_k para poder controlar la cantidad de iteraciones que hace un ciclo y poder calcular la precondición más débil.

- $H_0(Q) \equiv def(B) \land \neg B \land Q$
- $H_{k+1}(Q) \equiv def(B) \land B \land wp(S, H_k(Q))\ para\ k \geq 0$

Axiomas

Definimos el Axioma 5 para poder verificar la correctitud de ciclos que hacen una cantidad de iteraciones fijas como

$$wp(while\ B\ do\ S\ endwhile, Q) \equiv (\exists_{i \geq 0})(H_i(Q))$$

¿Cual es el problema del Axioma 5 y el Predicado H_k? El problema es que ambos nos sirven para poder validar la precondición de un ciclo que sabemos que finaliza luego de n iteraciones.

Predicado I - Invariante

Definimos el Invariante de un ciclo como un predicado que:

- Demuestra que el ciclo realmente funciona y nos ayuda a demostrar la correctitud parcial "si termina el ciclo... entonces se cumple Q"
- Vale antes de entrar al ciclo y luego de salir del ciclo.
- En cada iteración, el invariante debe volver a ser válido. En el medio de la iteración puede que deje de valer.
- Un buen invariante incluye el rango de la(s) variable(s) de control del ciclo.
- Un buen invariante tiene alguna afirmación sobre el acumulador del ciclo.

Teorema del Invariante

Para poder probar que un ciclo es correcto, usamos el teorema del invariante.

- $P \equiv$ Las condiciones del requiere
- $P_c \equiv$ Las precondiciones que necesita el ciclo para poder ingresar, muchas veces son las líneas que están por encima del while y el requiere (¡no todas las del requiere!)
- $I \equiv$ Las condiciones que suceden antes y después de entrar al ciclo
- $B \equiv$ La guarda del ciclo
- $Q_c \equiv$ La postcondición del ciclo

Luego,

- $\{P\}\ S\ \{P_c\}$
- $P_c \implies I$
- $\{I \land B\}\ S\ \{I\}$
- $\{I \land \neg B\} \implies Q_c$

Recordatorio: Si aparece la S es que tenemos que calcular la wp porque es parte de la Tripla de Hoare.
Recordatorio importante: Para todo lo que es wp usamos SmallLang, es decir, cuando tenemos que validar la tripla de Hoare. En todos los demás casos usamos lógica.
Ej: No sería válido tener un if en un invariante.

Teorema de Terminación de Ciclo

Utilizamos el teorema de terminación de ciclo para garantizar que cumpliendo la precondición de un ciclo, el programa siempre termina. Para poder probar esto, necesito una función variante f_v.
Llamamos función variante f_v a una función que es siempre estrictamente decreciente y representa una cantidad que se va reduciendo a lo largo de las iteraciones. La función f_v debe garantizar

- $\{I \land B \land v_0 = f_v\}\ S\ \{f_v < v_0\}$
- $\{I \land f_v \leq 0 \implies \neg B\}$

Reglas generales para validar correctitud de ciclo y terminación

- Cuando tenemos que iterar sobre listas, tendremos un índice i que irá hasta incluida la longitud (pues nos sirve) para negar la guarda, mientras que el j será para usar dentro del ciclo. $(0 \leq i \leq |s| \land_L (0 \leq j < i \longrightarrow_L res = \sum_{j=0}^{i} s[j]))$
- Si tengo algo en I deberá estar en P_c y/o Q_c. Esto es porque cuando tengamos que hacer las implicancias, deberíamos comparar de ambos lados y si el invariante tiene algo que los demás no, es siempre falso.
- Si tengo algo en P_c o Q_c no necesariamente tiene que estar en I
- En Q_c se acostumbra a colocar que $i = |s|$ porque nos ayuda a demostrar la terminación del ciclo.
- En la función variante: i va negada si i crece en el ciclo; i va positivo si i decrece en el ciclo.
- En Q_c rara vez tenemos que hablar de rangos de variables.

TADs (Tipos Abstractos de Datos)

Es un tipo de datos porque define un conjunto de valores y las operaciones que se pueden realizar sobre ellos. Es abstracto ya que para utilizarlos no necesitamos saber como está implementado. Describe el qué y no el cómo. Son una forma de modularizar a nivel de los datos.

Importante: En todo enunciado ambiguo, queda en nosotros interpretarlo y eliminar esas ambiguedades decidiendo como lo vamos a considerar.

Ej: En varios ejercicios te piden que un punto deba cambiarse el centro y hay dos maneras de plantearlo

- Agarrar el centro que vienen por parámetro y sumar coordenada a coordenada con respecto a las de la instancia de mi TAD.
- Agarrar el centro que viene por parámetro y considerarlo como el centro final en la instancia del TAD.

Importante: En un TAD podemos usar todos los mismos tipos de datos de especificación y sus funciones correspondientes.

Observadores

Los observadores son una especie de atributo en un objeto en POO. Nos permiten saber qué valores tienen y en qué momento.

- Sirven para describir el estado de una instancia de un TAD y nos permiten consultar su estado virtual.
- Tenemos que poder observar todas las características que nos interesan de las instancias.
- En un instante de tiempo, el estado de una instancia del TAD estará dado por el estado de todos sus observadores (como un debugger).
- Nos permiten distinguir si dos instancias son distintas.
- Todas las operaciones tienen que poder ser descriptas a partir de los observadores.

Ejemplo de un TAD con observadores de tipo dato:

```
1  TAD lista {
2      obs elems: conj<T>
3      obs cantElems: int
4  }
```

Los observadores también pueden ser funciones auxiliares.

- Son las auxiliares de nuestro lenguaje de especificación
- No pueden tener efectos colaterales ni modificar los parámetros
- Pueden usar tipos de nuestro lenguaje de especificación

Ejemplo de un TAD con observadores de tipo dato:

```
1  TAD lista {
2      obs elems: conj<T>
3      obs estaElem(e: T): bool
4  }
```

Igualdad Observacional

Decimos que dos TADs son iguales $\iff$ todos sus observadores son iguales.
Muchas veces no nos basta con la igualdad por defecto con los observadores, y tenemos que declarar nuestra propia igualdad observacional.

Operaciones de un TAD

Las operaciones de un TAD deben estar especificadas y nos indican qué se puede hacer con una instancia. Antes y una vez aplicada una operación tenemos que hablar del estado en el que quedó el TAD con respecto a sus observadores.

Implementación de un TAD

Módulos

Los módulos son la implementación de un TAD. Los módulos implementan el TAD.

Variables de Estado

Lo que en los TADs eran observadores, acá son variables de estado. Serán manipuladas por las operaciones mediante el código de los algoritmos. Tipos válidos para variables de estado:

- int, real, bool, char, string
- tupla<T1, T2, T3>, struct<campo1: val1, campo2: val2>
- Array<T> (arrays de tamaño fijo)
- No es posible usar tipos de especificación como conj<T> o seq<T>

El módulo puede tener variables de estado que no hagan referencia a ningún observador de un TAD, por ejemplo: guardar un máximo en un módulo aunque el TAD no lo pida. A veces estas cosas se hacen solo por temas de a la hora de implementarlo en un lenguaje de programación.

Invariante de Representación

- Define una restricción sobre el conjunto de valores que pueden tomar las variables de estado para que se considere una instancia válida.
- Es equivalente al invariante en ciclos, pero acá aplicado a TAD's
- Este es inicializado en el constructor una vez realizada la instancia
- Vale siempre antes de llamar a los métodos y luego de cada método. Siempre y cuando tenemos un parámetro inout, tenemos que ver que los cambios que hagamos hagan seguir valiendo a la invariante de representación.
- Todos los métodos pueden asumir que el invariante de representación siempre vale al ser llamados.
- ¡Prestar atencion a las verificaciones bidireccionales! En el sentido de que si tengo $dict < Alarma, Conj < Sensores \gg$ y $dict < Sensor, Conj < Alarmas \gg$ si en $dict < Alarma, ... >$ tengo varios sensores, esos sensores estan en Sensor y ademas, si busco en las alarmas de cada sensor deberia estar la $dict < Alarma, ... >$

Función de Abstracción

Definimos función de abstracción (α) como la representación física de lo que tiene el código, es decir, cuando acá hablamos de instancia.elems decimos que el código tiene un campo llamado elems. En TAD cuando nosotros hablabamos de un observador podía existir o no en la implementación como atributo pero seguir cumpliendo la especificación.

- Es un predicado
- Toma como parámetro una instancia del módulo y una instancia del TAD.
- Hacemos referencia a observadores y a otros predicados, no procs
- Todos los observadores del TAD deben tener un vinculo con una variable de estado pero no necesariamente al revés
- Para poder escribir, usamos lenguaje de especificación como lo conocemos. Podemos usar por ejemplo —a.data— o a.data.length da igual, pero hay que ser consistentes
- Rara vez se colocan rangos acá, casi nunca, excepto cuando hay alguna variable de estado y observadora que hablan de capacidades.

Es importante considerar que todo lo que está en el invariante de representación ya está implícito en la función de abstracción. Ej: Tenemos un módulo que tiene ventas, mayorProductoVentas, y mayorPrecio pero el TAD solo tiene ventas. En este caso, en el invariante de representación hablamos de mayorProductoVentas y mayorPrecio haciendo las relaciones con ventas (en temas de key del producto) pero en el predicado de astracción NO hablás de mayorProductoVentas y mayorPrecio primero porque no son observadores del TAD pero tampoco hay que garantizar nada porque ya está implicado por el invariante de representación

Operaciones de un Módulo

Como un módulo implementa un TAD, todas sus operaciones deben estar implementados en código. Para escribir las implementaciones de los procs usamos una especie de SmallLang.

- Declaración de variables:
  - var x: int
  - var c: ConjuntoArr¡int¿
- Asignación: x := valor
- Condicional: if condicion then codigo else codigo endif
- Ciclo: while condicion do codigo endwhile
- Llamar a un proc:
  - c.vacio()
  - b : = c.vacio();

Memoria dinámica

Los tipos complejos son usados siempre por referencia. El valor indefinido es identificado por la palabra null. Acceder a algo null explota.
Las variables de tipos complejos deben ser inicializadas mediante el operador new. Tipos nativos:

```
1  var a: Array<int>
2  var b: int
3  if a == null then
4  ...codigo
5  endif
6  a := new Array<int>(10)
7  b := a[0]
```

Tipos de otros módulos:

```
1  var a: ConjArr<int>
2  a := new ConjArr(10) //longitud 10
```

Pasaje por referencia:

```
var a: ConjArr<int>
var b: ConjArr<int>
a := new ConjArr(10); //longitud 10
b := a
```

Nota: El pasaje por referencia, a veces se le conoce comúnmente como aliasing y para evitarlo, hay que crear una nueva instancia con los valores de la otra instancia en vez de asignarla directamente.

Contrato entre métodos de un módulo

Los métodos de un Módulo tienen un contrato entre sí. Supongamos que necesitamos que la complejidad de búsqueda de un método de mi módulo sea o(log n) sabiendo que mi lista está ordenada. Aquí, el método busqueda podrá usar la búsqueda binaria para poder hacer el proceso lo más rápido posible, pero ¿qué sucede si al agregar un elemento en el método agregar() la lista de salida que se modifica en el módulo no está ordenada?
El contrato de búsqueda no se cumpliría porque la búsqueda binaria no funcionaría porque necesita que la lista esté ordenada.

En este caso, que la lista esté ordenada debería ser una condición del invariante de representación para evitar estos problemas.

Este módulo es bueno para la búsqueda de un elemento pero el agregar un elemento es mucho más costoso por el tema de reordenar la lista nuevamente.

Mirada al futuro: Los métodos deben cumplir una complejidad algorítimica dada.

Estructuras de Datos

Colección

Representa un grupo de objetos. Provee de una arquitectura para su almacenamiento y manipulación.

- Secuencia
- Conjunto
- Multiconjunto
- Diccionario

¿Por qué una tupla no es parte del grupo? Porque la tupla es fija, una vez definida e inicializada su longitud no cambia.

Tipos paramétricos

Son variables de tipo, es decir, variables con tipo genérico. La manera más común de indicar un genérico son T, K o V. Hay que tener cuidado con las operaciones que se realizan con las variables genéricas porque algunos operadores nos restringuen su uso a ciertos tipos.
Ej: No puedo utilizar X - Y si X, Y son genéricos y nos envían X, Y como char.

Iteradores

Nos permiten recorrer colecciones de una manera abstracta sin saber su estructura.
Un buen iterador se distingue de otro iterador por la velocidad de iterar la estructura.
Operaciones con iteradores:

- ¿Estoy sobre un elemento?
- Obtener el elemento actual
- Avanzar al siguiente elemento
- Retroceder al elemento anterior (sii es bidireccional)

Los Iteradores son una clase privada que van dentro de la clase que queremos que se instancie y se pueda recorrer.
Es privada porque no queremos que sea instanciada más que por la instancia de la clase que queremos recorrer.
Los Iteradores no van hasta n, van hasta n+1.


```java
public class Vector<T> implements List<T>{
    private T[] elementos;
    private int size;

    private class Iterador implements Iterator<T>{
        int indice;

        Iterador(){
            indice = 0;
        }
        public boolean hasNext(){
            return indice != size;
        }
        public T next(){
            int i = indice;
            indice = indice + 1;
            return elementos[i];
        }
    }

    public Iterator<T> iterator(){
        return new Iterador();
    }
}

Iterator it = vector.iterator();
while(it.hasNext()){
    System.out.println(it.next());
}
```

Singly Linked Lists - Listas simplemente enlazadas

Es una estructura que sirve para representar una secuencia de elementos donde cada elemento es un Nodo que tiene un valor y una referencia al siguiente.
Nota: Si el Nodo actual es el último de la Singly Linked List lo notamos porque el ultimo.siguiente = null. El Nodo debe ser responsable de guardar al siguiente porque caso contrario, no existirá otra referencia al siguiente Nodo. Ventajas de las Singly Linked Lists:

- Mas fino uso de la memoria.
- Insertamos fácilmente al principio y al final.
- El costo de insertar al final es o(n) pues debemos recorrer todos los nodos para insertar uno nuevo. Es por eso que es común guardar el último nodo almacenado en la clase.
- Eficiente para reacomodar elementos.
- Es malo en rendimiento o(n) si necesito buscar un elemento en específico. Es decir, perdemos el acceso aleatorio a los elementos.

Double Linked Lists - Listas doblemente enlazadas

Exactamente igual que la Singly Linked Lists pero acá los Nodos tienen almacenado: Referencia al nodo previo, valor y referencia al nodo siguiente. Nota: Si el Nodo actual es el último de la Double Linked List lo notamos porque el ultimo.siguiente = null.

Listas vs Linked Lists

Ambas son bastante rápidas a la hora de iterar sobre ellas, sin embargo:

- Las listas nos permiten acceder rápidamente a un elemento mientras que las Linked Lists no.
- Las Linked Lists nos permiten agregar elementos rápidamente al inicio (O(1)) y no necesitamos reacomodar nada más que decir que el anterior primero ahora es el segundo mientras que en la lista tenemos que reacomodar todos los elementos como antes pero colocar el nuevo al principio (O(n))