Conceptos transversales de los paradigmas de programación

- Si hay efecto de lado => **NO** hay transparencia referencial.
- Si hay transparencia referencial => **NO** hay efecto de lado.
- Si **NO** hay efecto de lado, no puedo asegurar que haya trasparencia referencial. Por ej. el mensaje “date today” de Smalltalk.
- Una solución es más **expresiva** que otra si resuelve un problema en forma más clara, menos compleja y usa nombres de variables bien claros.
- Una solución es más **declarativa** que otra si tiene menor grado de detalle sobre los algoritmos de cómo se resuelve el problema. Se delega el mecanismo de resolución a un motor.

Una Operación para que tenga transp. Refencial debe ser:

- Independiente (no depender del estado de nada externo)
- Sin estado (no tiene un estado que se mantenga de llamada en llamada)
- Determinística (devuelve el mismo valor dado los mismo argumentos)

Transparencia referencial (Haskell y Prolog): ya que no existe el concepto de estado ni de asignación destructiva. En Smalltalk al haber características de paradigmas imperativos como asignación destructiva y efecto de lado, no hay transparencia referencial.

Los tipos siempre existen. Si el compilador del lenguaje los chequea, es fuertemente tipado. De lo contrario, débilmente tipado.

Paradigma Funcional

Conceptos

Alto Nivel, gran claridad. Declarativo.

Abstracción, el programador se concentra en lo esencial del problema y se desentiende de los detalles de cómo se termina resolviendo. Una abstracción para representar a una entidad (una persona por ej.) puede ser una TUPLA.

Se especifica el QUE de la cuestión y no el COMO (de eso se ocupa el motor).

El ENFASIS está puesto en la acción o verbo (que acción es hecha).

Menos posibilidades de control sobre la maquina por parte del programador.

Transparencia Referencial: Cada función devuelve un valor, y este concepto hace referencia a que es posible reemplazar esa función por su resultado (el valor que devuelve) ya que este no depende del contexto de ejecución ni del orden de evaluación de la expresión. Es decir, el valor que devuelve una función está UNICAMENTE determinado por el VALOR de sus ARGUMENTOS.

Efecto de lado (o efecto colateral): Es una modificación que sobrevive (o sobrepasa) al bloque de código que la genera. NO es lo mismo que asignación destructiva, que es la simple modificación del valor de una variable. **NO existe efecto de lado en el paradigma funcional** ya que no permite asignar en más de un contexto el valor de una variable. Ej. de efecto de lado: cuando se asigna una variable de índole GLOBAL dentro de una función en un lenguaje imperativo como C. Se modifica su valor para el uso dentro de la función, pero como es GLOBAL está afectando a otras partes del código que la utilicen, por eso esa acción tiene efecto colateral.

Concepto de variable: NO son posiciones de memoria, son INCOGNITAS TEMPORALES que se resuelven cuando se quiere conocer el valor de la función para determinados valores del dominio (representado por los argumentos de la función).

Las listas por comprensión dotan al programa de mayor EXPRESIVIDAD. Esta es una medida para saber qué tan fácil el código comunica las intenciones e ideas del programador.

Una TUPLA es un tipo de dato compuesto que acepta elementos de distintos tipos. NO es un tipo de dato recursivo, mientras que una LISTA SÍ lo es.

Una forma de almacenar estados intermedios es pasando esos valores como argumentos a una función RECURSIVA y relacionándolos por medio del PATTERN MATCHING.

Recursividad: una función que se llama a sí misma.

Es un paradigma ATEMPORAL, ya que en la mayoría de los casos no interesa la secuencia.

Funciones de ORDEN SUPERIOR son las que reciben o devuelven FUNCIONES. La ventaja de usarlas es que permite separar responsabilidades. Las funciones son expresiones y son valores de primer orden. Aumentan declaratividad y expresividad.

Funciones CONSTANTES, no tiene argumentos. Sirve para darle un nombre representativo a un juego de datos en particular.

Las expresiones LAMBDA son funciones anónimas, que no tienen nombre y se usan por ej. con funciones de orden superior actuando como argumentos (definiciones locales). Me ahorra definir una función auxiliar aparte (para usarla una sola vez) pero este recurso dota al programa de menor expresividad. Para foldl por ej.

APLICACIÓN PARCIAL: dependiendo de la cantidad de parámetros aplicados a una función, obtengo una función o un valor resultante de la expresión calculada. Cuando la función no recibe la cantidad de parámetros que necesita, queda EVALUADA PARCIALMENTE y representa una función. Esta técnica permite GENERAR nueva funciones. Interviene el concepto de evaluación diferida.

La aplicación parcial es posible gracias a la CURRIFICACION: Considera a una función como una secuencia de funciones que reciben un único argumento. Una forma de evitar la aplicación parcial es definir un tupla como parámetro de la función, en lugar de los n parámetros que reciba.

CURRIFICAR UNA FUNCION: tener n argumentos en lugar de un argumento solo (lógicamente, una tupla de n elementos). Esto permite aplicarla parcialmente.

POLIMORFISMO PARAMETRICO: Cuando una función recibe parámetros de distinto tipo. Acepta un tipo genérico. Por ej. las funciones lenght, fst, snd, head y tail.

El objetivo de una función representa su responsabilidad.

POLIMORFISMO AD-HOC: la función en cuestión recibe como parámetro un conjunto de tipos en particular, es decir tiene **restricciones de tipo**.

PATTERN MATCHING: tratamos de unificar el argumento con los distintos patrones que tengamos. Una vez que matchea con una función, devuelve la expresión correspondiente. Lo importante es destacar que matchea SOLO una vez, justamente por el concepto matemático de función (unicidad, para un determinado valor del dominio hay una ÚNICA imagen).

Tipos de datos compuestos: * **Listas** -> Tipo de dato recursivo

* **Tuplas** -> Es una abstracción que permite representar entidades.

Haskell – Lenguaje del Paradigma funcional

Compilador: WinHugs

Características

Prelude.hs contiene definiciones de funciones estándares.

Lenguaje fuertemente tipado. El compilador hace chequeo de tipos.

Normalmente no se declaran de qué tipo son las cosas, el compilador infiere los tipos a través de los operadores o funciones que se aplican sobre los argumentos. El compilador chequea que los tipos del dominio y la imagen de la funciones sean válidos, acorde a su inferencia o declaración hecha en el caso que el programador haya hecho la declaración de tipos de su función.

Haskell, como muchos otros lenguajes del paradigma funcional, trabajan con el concepto de EVALUACION DIFERIDA (lazy evaluation), con lo que se van evaluando los argumentos recién a medida que los voy necesitando. Solo se evalua aquello que realmente se necesita. Gracias a esta característica de Haskell podemos tener **listas potencialmente infinitas**. En los lenguajes con EVALUACION ANSIOSA (imperativos) se reduce la expresión que se tiene como parámetro y después se invoca a la función. No hay lazy evaluation en lenguajes como C por el tema del efecto colateral, ya que si no se evalúa todo, luego un argumento malo (de un tipo distinto al esperado, por ej.) puede causar un error en el programa (que se rompa…).

Paradigma Lógico

Conceptos

Declarativo, no hay algoritmo. Sólo definiciones y un motor que resuelve el problema. Me concentro en lo esencial del problema. Abstracción.

En mi base de conocimiento tengo HECHOS (son axiomas, son ciertos porque existen, de lo contrario se presumen falsos) y REGLAS (que contienen un consecuente y uno o varios antecedentes).

UNIFICAR: igualar dos términos (variables).

LIGAR: darle un valor (constante) a una variable.

A diferencia de funcional, el motor de lógico buscará probar la verdad de la consulta usando todas las cláusulas que haya de un predicado.

El concepto de PATTERN MATCHING asociado con functores puede lograr polimorfismo definiendo varias cláusulas de un mismo predicado, una por cada tipo de functor que quiera tratar polimórficamente.

Declaro CONOCIMIENTO a través de predicados (afirmaciones respecto a algo). Tanto los hechos como las reglas son predicados. Un predicado de un solo argumento representa una propiedad mientras que uno de mayor aridad representa una relación.

Una VARIABLE es una incógnita, algo que está sin resolver.

No existe la ASIGNACION de variables ya que no son posiciones de memoria. No existe por lo tanto el EFECTO COLATERAL.

A través del mecanismo BACKTRACKING es como el motor encuentra todas las unificaciones posibles para una variable.

UNIVERSO CERRADO o MUNDO ACOTADO: todo lo que no está en mi base de conocimiento, no se puede inferir su veracidad, por lo tanto se presume FALSO.

Tenemos consultas EXISTENCIALES (algún argumento sin ligar) y consultas INDIVIDUALES (se refieren a individuos, son argumentos ligados o instanciados). Las consultas existenciales por lo general devuelven varios resultados, esto quiere decir que el motor encuentra más de una manera de resolver/unificar las incógnitas.

PATTERN MATCHING: Se produce en todos los argumentos donde pueda unificarse valor o variable contra lo que yo mandé (mi consulta). Suele haber muchos matcheos posibles, lo que se traduce en MUCHAS respuestas.

VENTAJA lógico sobre funcional: Una relación es más abarcativa que una función, me permite ver más cosas. No tengo que pensar de antemano que consultas satisfacer, el motor se encarga de responder varias preguntas a la vez.

INVERSIBILIDAD: Es la propiedad de hacer consultas sin importar si los argumentos están instanciados o no (si le paso una constante o una incógnita). Las restricciones para que los predicados no sean inversibles son las comparaciones, las operaciones aritméticas, el operador IS y el operador NOT. Si un predicado es inversible aumenta la DECLARATIVIDAD de la solución.

Para una correcta inversibilidad, en ciertas reglas se usan predicados GENERADORES (le aplican un dominio al argumento) que lo que hacen es ligar una incógnita con algún valor de nuestra base de conocimiento. Por ejemplo antes de un forall o findall.

FUNCTOR: No es un predicado, aunque la notación se le parezca. Denota un individuo compuesto, no una regla o un hecho. Tiene una cantidad fija de elementos que pueden ser de distinto tipo. Admiten ser tratados POLIMORFICAMENTE y además se puede usar PATTERN MATCHING dentro de ellos.

PREDICADO DE ORDEN SUPERIOR: trabajan sobre predicados. Nos dan un grado mayor de abstracción.

RECURSIVIDAD: predicados que se llaman a sí mismos. Esta técnica reemplaza a las estructuras de iteración de los paradigmas imperativos.

Prolog – Lenguaje del Paradigma Lógico

Compilador: SWI-Prolog

Características

Todas las variables del lado derecho del IS deben estar unificadas, ya que el motor evalúa esta expresión. El = unifica términos directamente.

El predicado FINDALL dota de mayor DECLARATIVIDAD a la solución.

Es un lenguaje débilmente tipado, no hay chequeo de tipos. En Prolog puedo tener una lista con elementos de distinto tipo, cosa que en Haskell NO se puede.

EVALUACION ANSIOSA.

forall(A,B) es igual a hacer not((A,not(B))), es lo mismo decir:

No hay UNO que cumpla A y no cumpla B, que decir todos los que cumple A también cumplen B.

Unificacion y Pattern Matching en Logico y Funcional

El concepto de pattern matching existe en ambos paradigmas. En lógico existe la posibilidad de dejar variables sin unificar en una consulta (sin darle un valor) para que el motor encuentre todas las soluciones posibles. En funcional esto no es posible. La inversibilidad es un concepto del paradigma LOGICO.

En lógico, el motor busca todas las posibilidades para cada patrón mientras que en Funcional se encaja siempre el primer patrón encontrado.

Paradigma orientado a Objetos

Conceptos

Nació con una idea procedimental, ya que tengo una secuencia de pasos y el orden es importante, como también tengo mayor control sobre el algoritmo. Así y todo, depende del programador, pero puede usarse más DECLARATIVAMENTE ya que nos abstraemos de cómo se realizan algunas cosas.

Un objeto es algo que puedo representar a través de una idea, un concepto. Tiene entidad.

Un sistema es un conjunto de objetos que se envían mensajes para alcanzar un determinado objetivo.

AMBIENTE: lugar donde “viven” los objetos.

VARIABLE: es un puntero a un objeto.

MENSAJE: es lo que el objeto EMISOR le envía como orden al RECEPTOR. El emisor no se entera de cómo se resuelve el mensaje, solo lo pide. -> ABSTRACCION.

METODO: porción de código que se ejecuta cuando un objeto receptor recibe un MENSAJE con el mismo selector. La búsqueda del método se llama “METHOD LOOKUP”. Si el método no se encuentra en la clase del objeto al cual se mandó el mensaje, se buscará en las superclases de esa clase. Si tampoco se encuentra, se devuelve un error.

ABSTRACCION: nos concentramos SOLO en lo que queremos resolver y dejamos los detalles que NO son esenciales de lado. En un ejemplo práctico, cuando le mando un mensaje a un objeto, no me interesa que haga el objeto internamente sino que me devuelva lo que espero. Esta abstracción favorece a que yo sepa menos de los objetos que manejo.

ENCAPSULAMIENTO (concepto de caja negra): Agrupación de funcionalidades que son propias de un objeto en particular. Separa las interfaces de las implementaciones de la funcionalidad del sistema (métodos) y oculta la información (variables). Un objeto no conoce el funcionamiento interno de los demás objetos y TAMPOCO lo necesita para interactuar con ellos, le es suficiente con conocer su interfaz (los mensajes que acpetan).

COLABORACION: los objetos colaboran para la realización de una acción.

EFECTO COLATERAL: Existe en este paradigma. Cuando le mando un mensaje binario a un objeto, este puede que cambie alguna de sus atributos, por lo tanto tenemos efecto de lado. Tiene ASIGNACION DESTRUCTIVA.

INTERFAZ: es lo que un objeto publica para que otro objeto lo use.

IMPLEMENTACION: es lo que un objeto ENCAPSULA para definir cómo se resuelve un mensaje.

POLIMORFISMO: a TODOS los objetos les hablo igual, solo que cada uno responde a su manera. El polimorfismo es para el OBSERVADOR que usa esos objetos. Estos objetos en si no tienen idea que son polimórficos, sólo responden a una serie de cosas que forman parte de sus responsabilidades.

Un objeto CLONADO (no tiene métodos) usa los métodos del objeto original del cual fue clonado. Evalúa el método en su propio contexto, por lo cual modifica SUS atributos y no los del objeto original.

CLASE: es un TEMPLATE para la generación de objetos. Cada objeto es una instancia de una y sola una clase. Describe la estructura de información y el comportamiento que tendrá todo objeto de esa clase.

HERENCIA: una clase hereda de otra, comportamiento y atributos. Además la nueva clase puede agregar más atributos, comportamientos o redefinir comportamientos heredados pero adaptándolo a la necesidad de la nueva clase.

CLASE ABSTRACTA: Es una clase que no tengo intención de crearle instancias, porque no tiene sentido, quizá su comportamiento y sus atributos sean muy generales para el sistema que quiero modelar.

Clase no es igual a TIPO. Un tipo es un conjunto de mensajes que entiende un objeto.

METODOS GETTERS: Devuelven los valores de un atributo en particular. Llevan el mismo nombre que el atributo.

METODOS SETTERS: asigna a un atributo un valor que se le pasa como parámetro. Sirve para inicializar por lo general.

COLECCIONES: conjunto de elemento de distinto tipo, sin orden alguno. Lo copado es que tenga dentro de una colección elementos que los pueda tratar POLIMORFICAMENTE,

aunque sean de distintas clases. Las colecciones permiten modelas RELACIONES del tipo 1 a N.

Relación de ASOCIACION entre dos CLASES: indica que una de ellas tiene como atributo a la otra. Este puede ser también una colección, depende del tipo de asociación, 1 a 1, 1 a N, etc.

OBJETOS BLOQUE: Es un objeto que representa una porción de código. Por ejemplo, un criterio de selección con select:. Les puedo mandar el mensaje value para que se ejecuten.

El ENFASIS está puesto en el sustantivo (quien hace las cosas).

VARIABLES (o atributos) DE CLASE: son globales y comunes para todas las instancias de esa clase. En Smalltalk comienzan con mayúscula.

METODOS DE CLASE: son los métodos en los que el objeto receptor es la clase (new por ej.). En Smalltalk lo llamo "self class nombreMetodo".

DELEGACION: Un objeto delega a otro una responsabilidad al enviarle un mensaje solicitándole un servicio. Es decir, el pedido de alguna tarea en particular puede solicitarse a un objeto en particular y este a su vez delegar responsabilidades a otros.

Smalltak – Lenguaje del Paradigma orientado a Objetos

Características

TODO es un OBJETO. La CLASE es un objeto.

SINTAXIS: objeto mensaje parámetro/s

El AMBIENTE es un archivo de imagen (*.img).

SELF: apunta al propio objeto. Para invocar un método propio.

SUPER: Apunta también al objeto receptor, pero la búsqueda del método comienza a partir de la superclase del objeto receptor.

Si bien HAY tipos, no se hace ningún chequeo en tiempo de compilación, por lo tanto Smalltalk es DEBILMENTE TIPADO.

EVALUACION ANSIOSA.

El select: no tiene efecto colateral ya que filtra una colección dada, y los elementos que cumplen el criterio de filtro se agregan a la colección resultante (permaneciendo también en la original). Lo mismo con collect:, genera una nueva colección.

BINDING DINAMICO (Enlace Dinámico o en Run Time): Cuando compilo NO SE exactamente que código (método) se va a ejecutar ante un mensaje que es entendido por varios objetos (polimorfismo), ya que no sé qué objeto voy a llamar en particular. Esto me permite tener polimorfismo ya que difiero hasta último momento posible el enlace entre operador (mensaje) y operando (objeto).

HERENCIA SIMPLE: cada clase puede heredar de una sola.

El **self** dentro de un método SIEMPRE va a referenciar a la instancia que está ejecutando ese método. Por más que self sea el parámetro de un mensaje que le enviamos a otro objeto para delegar comportamiento.