Lenguaje de programación Haskell

Lenguaje de programación Haskell

Concepto:	Es un lenguaje de programación funcional puro, de propósito general, que incluye muchas de las últimas innovaciones en el desarrollo de los lenguajes de programación funcional.

Lenguaje de programación Haskell. Es un lenguaje de programación puramente funcional, de propósito general. El nombre proviene del matemático y lógico estadounidense Haskell Curry.

Características

Incluye muchas de las últimas innovaciones en el desarrollo de los lenguajes de programación funcional, como son las funciones de orden superior, evaluación perezosa, tipos polimórficos estáticos, tipos definidos por el usuario, encaje por patrones, y definiciones de listas.

Incorpora, además, otras características interesantes como el tratamiento sistemático de la sobrecarga, la facilidad en la definición de tipos abstractos de datos, el sistema de entrada/salida puramente funcional y la posibilidad de utilización de módulos.

Se utiliza como referencia el entorno de programación Hugs y se supone que el lector tiene unos mínimos conocimientos del modelo de programación imperativo o tradicional.

Historia

A principios de la década de los setenta aparecieron los primeros síntomas de lo que se ha denominado crisis del software. Los programadores que se enfrentan a la construcción de grandes sistemas de software observan que sus productos no son fiables. La alta tasa de errores conocidos (bugs) o por conocer pone en peligro la confianza que los usuarios depositan en sus sistemas.

Cuando los programadores quieren corregir los errores detectados se enfrentan a una dura tarea de mantenimiento. Cuando se intenta corregir un error detectado, una pequeña modificación trae consigo una serie de efectos no deseados sobre otras partes del sistema que, en la mayoría de las ocasiones, empeora la situación inicial.

La raíz del problema radica en la dificultad de demostrar que el sistema cumple los requisitos especificados. La verificación formal de programas es una técnica costosa que en raras ocasiones se aplica.

Por otro lado, el incremento en la potencia de procesamiento lleva consigo un incremento en la complejidad del software. Los usuarios exigen cada vez mayores prestaciones cuyo cumplimiento sigue poniendo en evidencia las limitaciones de recursos y la fragilidad del sistema.

Las posibles soluciones podrían englobarse en las siguientes líneas de investigación:

- Ofrecer nuevos desarrollos de la Ingeniería del Software que permitan solucionar el problema del Análisis y Diseño de grandes Proyectos Informáticos. Actualmente las metodologías orientadas a objetos han aportado una evolución importante dentro de este campo.
- Proporcionar sistemas de prueba y verificación de programas cuya utilización no sea costosa.
- Construir técnicas de síntesis de programas que permitan obtener, a partir de unas especificaciones formales, código ejecutable.
- Diseñar nuevas arquitecturas de computadoras, en particular, técnicas de procesamiento en paralelo.
- Proponer un modelo de computación diferente al modelo imperativo tradicional.

Esta última solución se basa en la idea de que los problemas mencionados son inherentes al modelo computacional utilizado y su solución no se encontrará a menos que se utilice un modelo diferente. Entre los modelos propuestos se encuentra la programación funcional o aplicativa, cuyo objetivo es describir los problemas mediante funciones matemáticas puras sin efectos laterales, y la programación lógica o declarativa, que describe los problemas mediante relaciones entre objetos.

Problemas del modelo imperativo

Los lenguajes de programación tradicionales como Pascal, C, C++, ADA, entre otros forman una abstracción de la máquina de Von-Neumann caracterizada por:

- Memoria principal para almacenamiento de datos y código máquina.
- Unidad Central de Proceso (CPU) con una serie de registros de almacenamiento temporal y un conjunto instrucciones de cálculo aritmético, modificación de registros y acceso a la memoria principal.

Los programas imperativos poseen una serie de datos globales y una secuencia de comandos ó código. Estos dos elementos forman una abstracción de los datos y código de la memoria principal. Para hacer efectiva dicha abstracción se compila el código fuente para obtener código máquina.

El modelo imperativo ha tenido un gran éxito debido a su sencillez y proximidad a la arquitectura de los computadores convencionales.

El programador trabaja en un nivel cercano a la máquina que le permite generar programas eficientes. Con esta proximidad aparece, sin embargo, una dependencia entre el algoritmo y la arquitectura que impide, por ejemplo, utilizar algoritmos programados para arquitecturas secuenciales en arquitecturas paralelas.

Los algoritmos en lenguajes imperativos se expresan mediante una secuencia de órdenes que modifican el estado de un programa accediendo a los datos globales de la memoria.

Las instrucciones de acceso a los datos globales destruyen el contenido previo de un dato asignando un nuevo valor. Las asignaciones introducción al lenguaje Haskell producen una serie de efectos laterales que oscurecen la semántica del lenguaje.

Modelo funcional

El modelo funcional, tiene como objetivo la utilización de funciones matemáticas puras sin efectos laterales y, por tanto, sin asignaciones destructivas.

El esquema del modelo funcional es similar al de una calculadora. Se establece una sesión interactiva entre sistema y usuario: el usuario introduce una expresión inicial y el sistema la evalúa mediante un proceso de reducción. En este proceso se utilizan las definiciones de función realizadas por el programador hasta obtener un valor no reducible.

El valor que devuelve una función está únicamente determinado por el valor de sus argumentos consiguiendo que una misma expresión tenga siempre el mismo valor (esta propiedad se conoce como transparencia referencial). Es más sencillo demostrar la corrección de los programas ya que se cumplen propiedades matemáticas tradicionales como la propiedad commutativa, asociativa y otras.

El programador se encarga de definir un conjunto de funciones sin preocuparse de los métodos de evaluación que posteriormente utilice el sistema. Este modelo deja mayor libertad al sistema de evaluación para incorporar pasos intermedios de transformación de código y paralelización ya que las funciones no tienen efectos laterales y no dependen de una arquitectura concreta.

La importancia de la programación funcional no radica únicamente en no utilizar asignaciones destructivas. Por el contrario, este modelo promueve la utilización de una serie de características como las funciones de orden superior, los sistemas de inferencia de tipos, el polimorfismo, la evaluación perezosay otras.

Evolución del modelo funcional

Los orígenes teóricos del modelo funcional se remontan a los años treinta en los cuales Church propuso un nuevo modelo de estudio de la computabilidad mediante el cálculo lambda. Este modelo permitía trabajar con funciones como objetos de primera clase. En esa misma época, Shönfinkel y Curry construían los fundamentos de la lógica combinatoria que tendría gran importancia para la implementación de los lenguajes funcionales.

Hacia 1950, John McCarthy diseñó el lenguaje LISP (List Processing) que utilizaba las listas como tipo básico y admitía funciones de orden superior.

Este lenguaje se ha convertido en uno de los lenguajes más populares en el campo de la Inteligencia Artificial. Sin embargo, para que el lenguaje fuese práctico, fue necesario incluir características propias de los lenguajes imperativos como la asignación destructiva y los efectos laterales que lo alejaron del paradigma funcional.

En 1964, Peter Landin diseñó la máquina abstracta SECD para mecanizar la evaluación de expresiones, definió un subconjunto no trivial de Algol–60 mediante el cálculo lambda e introdujo la familia de lenguajes ISWIM (*If You See What I Mean*) con innovaciones sintácticas (operadores infijos y espaciado) y semánticas importantes.

En 1978 J. Backus (uno de los diseñadores de "*FORTRAN*" y "*ALGOL*") consiguió que la comunidad informática prestara mayor atención a la programación funcional con su artículo *Can Programming be liberated from the Von Neumann style?* en el que criticaba las bases de la programación imperativa tradicional mostrando las ventajas del modelo funcional.

Además Backus diseñó el lenguaje funcional FP (Functional Programming) con la filosofía de definir nuevas funciones combinando otras funciones.

A mediados de los setenta, Gordon trabajaba en un sistema generador de demostraciones denominado LCF que incluía el lenguaje de programación ML (Metalenguaje).

A mediados de los ochenta se realizó un esfuerzo de estandarización que culminó con la definición de SML (Stándar ML). Este lenguaje es fuertemente tipado con resolución estática de tipos, definición de funciones polimórficas y tipos abstractos. Actualmente, los sistemas en SML compiten en eficiencia con los sistemas en otros lenguajes imperativos y han aparecido proyectos como Fox Project 1994 [HL94] que pretenden desarrollar un nuevo lenguaje ML2000 con subtipos y módulos de orden superior.

Al mismo tiempo que se desarrollaban FP y ML, David Turner (primero en la Universidad de St. Andrews y posteriormente en la Universidad de Kent) trabajaba en un nuevo estilo de lenguajes funcionales con evaluación perezosa y definición de funciones mediante encaje de patrones. El desarrollo de los lenguajes SASL (St. Andrews Static Language), KRC (Kent Recursive Calculator) y Miranda5 tenía como objetivo facilitar la tarea del programador incorporando facilidades sintácticas como las guardas, el encaje de patrones, las listas y las secciones.

A comienzos de los ochenta surgieron una gran cantidad de lenguajes funcionales debido a los avances en las técnicas de implementación. Entre éstos, se podrían destacar Hope, LML, Orwell, Erlang, FEL, Alfl, etc.

En septiembre de 1987, se celebró la conferencia FPCA en Portland, Oregon, en la que se discutieron los problemas que creaba esta proliferación. Se decidió formar un comité internacional que diseñase un nuevo lenguaje puramente funcional de propósito general denominado Haskell (Hud92).

Con el lenguaje Haskell se pretendía unificar las características más importantes de los lenguajes funcionales, como las funciones de orden superior, evaluación perezosa, inferencia estática de tipos, tipos de datos definidos por el usuario, encaje de patrones y listas por comprehensión.

Al diseñar el lenguaje se observó que no existía un tratamiento sistemático de la sobrecarga con lo cual se construyó una nueva solución conocida como las clases de tipos.

En mayo de 1996 aparecía la versión 1.3 del lenguaje Haskell [Has95] que incorporaba, entre otras características, mónadas para entrada/salida, registros para nombrar componentes de tipos de datos, clases de constructores de tipos y diversas librerías de propósito general.

Posteriormente, surge la versión 1.4 con ligeras modificaciones. En 1998 se decide proporcionar una versión estable del lenguaje, que se denominó Haskell98 a la vez que se continúa la investigación de nuevas características.

Funciones orden superior

Un lenguaje utiliza funciones de orden superior cuando permite que las funciones sean tratadas como valores de primera clase, permitiéndo que sean almacenadas en estructuras de datos, que sean pasadas como argumentos de funciones y que sean devueltas como resultados.

La utilización de funciones de orden superior proporciona una mayor flexibilidad al programador, siendo una de las características más sorbresalientes de los lenguajes funcionales. De hecho, en algunos ámbitos se considera que la propiedad que distingue un lenguaje funcional de otro es la utilización de funciones de orden superior.

Sistemas de inferencia de tipos y polimorfismo

Muchos lenguajes funcionales han adoptado un sistema de inferencia de tipos que consiste en:

- El programador no está obligado a declarar el tipo de las expresiones
- El compilador contiene un algoritmo que infiere el tipo de las expresiones
- Si el programador declara el tipo de alguna expresión, el sistema chequea que el tipo declarado coincide con el tipo inferido.

Los sistemas de inferencia de tipos permiten una mayor seguridad evitando errores de tipo en tiempo de ejecución y una mayor eficiencia, evitando realizar comprobaciones de tipos en tiempo de ejecución.

Polimorfismo

El polimorfismo permite que el tipo de una función dependa de un parámetro.

En un lenguaje sin polimorfismo sería necesario definir una función para cada tipo de lista que se necesitase. El polimorfismo permite una mayor reutilización de código ya que no es necesario repetir algoritmos para estructuras similares.

Evaluación perezosa

Los lenguajes tradicionales, evalúan todos los argumentos de una función antes de conocer si éstos serán utilizados.

Por otra parte, en ciertos lenguajes funcionales se utiliza evaluación perezosa (*Lazy Evaluation*) que consiste en no evaluar un argumento hasta que no se necesita.

Uno de los beneficios de la evaluación perezosa consiste en la posibilidad de manipular estructuras de datos infinitas.

Evidentemente, no es posible construir o almacenar un objeto infinito en su totalidad. Sin embargo, gracias a la evaluación perezosa se puede construir objetos potencialmente infinitos pieza a pieza según las necesidades de evaluación.

Fuentes

- Sitio Oficial Haskell