**GHC**

**GRUPO 20**

1. **¿Por qué fue creado y cuales son sus objetivos de diseño? ¿Cuán grande es el equipo de desarrollo desde entonces?**

GHC fue creado por estas tres principales razones:

* Para hacer disponible libremente un compilador robusto y portable para Haskell que genera código de alto rendimiento;
* Proporcionar una base modular que otros investigadores puedan extender y desarrollar;
* Para aprender cómo se comportan los programas reales, para que podamos diseñar y construir mejores compiladores.

A lo largo de su vida, GHC generalmente tuvo y tiene alrededor de dos o tres desarrolladores activos, aunque la cantidad de personas que han contribuido con algún código para GHC es de cientos.

1. **¿Cuales son sus tres principales componentes?**

En el nivel más alto, GHC se puede dividir en tres componentes distintos:

* El compilador en sí. Este es esencialmente un programa Haskell cuyo trabajo es convertir el código fuente Haskell en código máquina ejecutable.
* Las bibliotecas de arranque. GHC viene con un conjunto de bibliotecas que llamamos bibliotecas de inicio, porque constituyen las bibliotecas de las que depende el propio compilador. Tener estas bibliotecas en el árbol fuente significa que GHC puede iniciarse él mismo. Algunas de estas bibliotecas están estrechamente acopladas con GHC, porque implementan funcionalidades de bajo nivel como el tipo Int en términos de primitivas definidas por el compilador y el sistema de tiempo de ejecución. Otras bibliotecas son de más alto nivel e independientes del compilador, como la biblioteca Data.Map.
* El sistema de tiempo de ejecución (RTS). Esta es una gran biblioteca de código C que maneja todas las tareas asociadas con la *ejecución* del código Haskell compilado, incluida la recolección de basura, programación de hilos, creación de perfiles, manejo de excepciones, etc. El RTS está vinculado a cada programa compilado de Haskell. El RTS representa una parte importante del esfuerzo de desarrollo puesto en GHC, y las decisiones de diseño tomadas allí son responsables de algunas de las fortalezas clave de Haskell, como su soporte eficiente para concurrencia y paralelismo.

1. **¿Cuánto creció el código a lo largo de 20 años? ¿Por qué?**

GHC tiene ahora más de 20 años y ha estado en continuo desarrollo activo desde sus inicios. Hoy, los lanzamientos de GHC son descargados por cientos de miles de personas, el repositorio en línea de las bibliotecas de Haskell tiene más de 3.000 paquetes, GHC se usa para enseñar Haskell en muchos cursos de enseñanza, y hay un número creciente de casos de dependencia de Haskell comercialmente.

1. **¿Cómo logra el compilador desacoplarse de los cambios (frecuentes) al lenguaje?**

Una estructura típica para un compilador de un lenguaje de tipo estático es la siguiente: el programa se comprueba por tipo y se transforma en un lenguaje intermedio sin *tipo* , antes de ser optimizado. GHC es diferente: tiene un *lenguaje intermedio de tipo estático* . Como resultado, esta elección de diseño ha tenido un efecto generalizado en el diseño y desarrollo de GHC.

El lenguaje intermedio de GHC se llama Core. Los siguientes puntos son las principales ventajas de este:

* Haskell es un idioma fuente muy grande. El tipo de datos que representa su árbol de sintaxis tiene literalmente cientos de constructores.

Por el contrario, Corees un cálculo lambda minúsculo y de principios. Tiene muy pocas formas sintácticas, sin embargo podemos traducir todo Haskell a Core.

* Haskell es un *lenguaje* fuente *tipeado implícitamente* . Un programa puede tener pocas o ninguna anotación de tipo; en cambio, depende del algoritmo de inferencia de tipo averiguar el tipo de cada enlace y subexpresiones. Este tipo de algoritmo de inferencia es complejo, y ocasionalmente algo ad hoc (específico), lo que refleja los compromisos de diseño que todo lenguaje de programación real incorpora.

Por el contrario, Core es un lenguaje *explícitamente tipado* . Cada carpeta tiene un tipo explícito, y los términos incluyen abstracciones y aplicaciones de tipo explícito. Core disfruta de un algoritmo de verificación de tipo muy simple y rápido. El algoritmo es completamente directo; no hay compromisos ad hoc.

Todos los pases de análisis y optimización de GHC funcionan Core. Esto es genial: debido a que Core es un lenguaje tan pequeño, una optimización tiene solo unos pocos casos para tratar. Aunque Core es pequeño, es extremadamente expresivo: después de todo, el Sistema F se desarrolló originalmente como un cálculo fundamental para la computación mecanografiada. Cuando se agregan nuevas características de idioma al idioma de origen (y eso ocurre todo el tiempo), los cambios generalmente se restringen a la interfaz; Core se mantiene sin cambios, y por lo tanto también lo hace la mayoría del compilador.

1. **¿De qué forma fomenta el compilador la modularización del código Haskell? ¿Qué ventajas ofrece escribir funciones pequeñas y qué desventajas acarrea esta decisión?**

Mediante Cross-module inlining.

GHC debe poder hacer cross-module y, de hecho, cross-package, inline. La idea es:

* Al compilar un módulo Haskell Lib.hs, GHC produce un código de objeto Lib.o y un "archivo de interfaz" en Lib.hi. Este archivo de interfaz contiene información sobre todas las funciones que Lib exporta, incluidos sus tipos y, para funciones suficientemente pequeñas, sus definiciones.
* Al compilar un módulo Client.hs que importa Lib, GHC lee la interfaz Lib.hi. Entonces, si el Client llama a una función Lib.f definida en Lib, GHC puede usar la información en Lib.hi línea Lib.f.

La incorporación de cross-modules es absolutamente esencial para definir bibliotecas súper eficientes, pero tiene un costo. Si el autor actualiza su biblioteca, no es suficiente volver a vincularla Client.o con la nueva Lib.o, porque Client.o contiene fragmentos en línea de la anterior Lib.hs y es posible que no sean compatibles con la nueva. Otra forma de decir esto es que la ABI (Application Binary Interface) Lib.o ha cambiado de una manera que requiere la recompilación de sus clientes.

1. **¿Por qué ha sido fácil refactorizar el código de GHC a lo largo de los años?**

Se debe a tres motivos:

* Una buena ingeniería de software, basada en Modularidad, que siempre vale la pena. Haciendo a las APIs entre componentes bien pequeños, mantienen la flexibilidad en los componentes individuales haciéndolos tener pocas dependencias.
* Está desarrollado en un lenguaje fuertemente tipado, lo que hace que la refactorización sea muy sencilla. Siempre que se necesita modificar el tipo de dato, o cambiar la cantidad de argumentos o el tipo de una función, el compilador nos indica inmediatamente que otras partes del código necesitan ser retocadas.
* Cuando programas en un lenguaje puramente funcional, es difícil introducir dependencias accidentalmente. Que sea puramente funcional te fuerza a hacer las dependencias explícitas, lo que te quita presión en agregar nuevas, y pocas dependencias se traducen en una mejor modularidad.