Tesina

Iñaki Garay

Septiembre 2020

Contents

Introducción / Esquema General

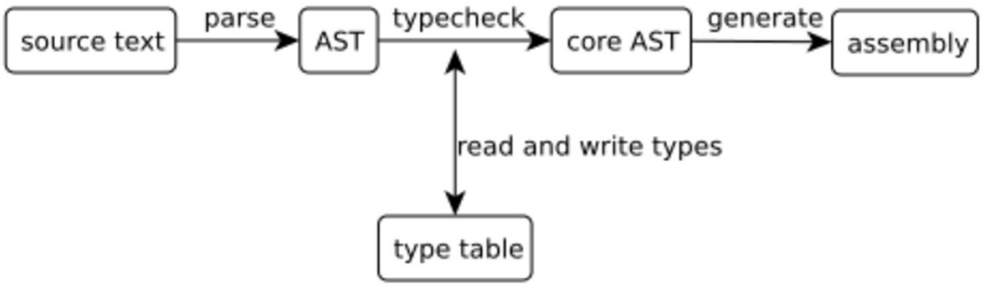
- Los compiladores tradicionales tienen una arquitectura de pipeline.
- Dos requerimientos nuevos paralelos: minimizar tiempo de compilacion y proveer mas informacion de manera mas interactiva a las herramientas de edicion.
- Los editores y entornos de desarrollo modernos usan LSP (Language Server Protocol). Porque?
- Una implementacion de LSP require de componentes de compiladores (especialmente analisis).
- La arquitectura de pipeline no se adapta bien a estos requerimientos modernos de reducir tiempos de compilacion y de proveer informacion online durante edicion.
- Estos objetivos puede ser logrados mediante compilacion incremental.
- La compilacion incremental puede ser lograda mediante una arquitectura basada en queries.
- La arquitectura basada en queries es implementada tomando inspiracion de build systems.
- Como funciona el sistema de queries de rustc?
- Rust-analyzer es la segunda implementacion de LSP para rust.
- Porque no funciono la primera iteracion?
- Rust-Analyzer usa una libreria, Salsa, para cachear las queries parciales.
- Como funciona Salsa?

Motivaciones

Los compiladores ya no son cajas negras que ingestan un conjunto de archivos fuente y producen codigo ensamblador. De compiladores modernos se espera que:

- Sean incrementales, es decir, si se recompila el proyecto despues de producir modificaciones en el código fuente, solo se recompila lo que fue afectado por esas modificaciones.
- Provean funcionalidad para editores, e.g. saltar a definición, encontrar el tipo de una expresión en una ubicación dada, y mostrar errores al editar.

Arquitecturas Tradicionales de Compiladores [Olle20]



[arc20]

Hay muchas variaciones, y frecuentemente mas pasos y representaciones intermedias que las ilustradas, pero la idea esencial es la misma: se empuja codigo fuente por un pipeline y corremos una sequencia fija de transformaciones hasta que finalmente emitimos codigo ensamblador o algun otro lenguaje. En el camino frecuentemente se necesita leer y actualizar estado interno. Por ejemplo, se puede actualizar la tabla de tipos durante la fase de verificacion de tipado, para que mas adelante se pueda verificar el tipo de las entidades a las cuales el codigo se refiere. [arc20]

Language Server Protocol

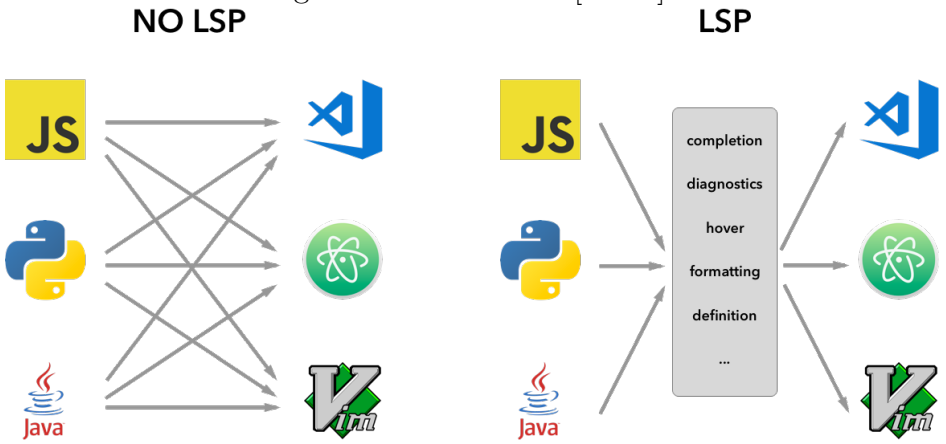
El Language Server Protocol (LSP) es una protocolo abierto basado en JSON-RPC para el uso entre editores de codigo fuente o entornos de desarrollo integrados (IDE) y servidores que proveen funcionalidades especificas a lenguajes de programacion. El objetivo del protocolo es permitir que se implemente y distribuya el soporte para un lenguaje de programacion independientemente de un editor o IDE determinado. Implementar funcionalidad tal como autocompletado, ir a definicion, o mostrar documentacion relacionada a una entidad para un lenguaje de programacion determinado, requieren de esfuerzos significantes. Tradicionalmente este trabajo se debia repetir para cada herramienta de desarrollo, dado que cada herramienta proveia un API diferente al impl Un Servidor de Lenguaje provee inteligencia especifica a un lenguaje y se comunica con herramientas de desarrollo mediante un protocolo que permite comunicacion entre procesos. La idea detras de LSP es estandarizar el protocolo con el cual se comunican los las herramientas de desarrollo y los servidores. De esta manera, un unico servidor de lenguaje puede ser reutilizado por multiples herramientas de desarrollo, las cuales a su vez pueden soportar multiples lenguajes, con esfuerzo minimo. [Mic]

Los entornos de desarrollo integrados (IDEs) modernos proveen a desarrolladores funcionalidad sofisticada tale como completado de codigo, refactoreo, navegacion a definicino de un simbolo, resaltamiento de sintaxis, y marcadores de errores y avisos. Por ejemplo, en un lenguaje basado en texto, un programador podria querer renombra un metodo. El programador podria o bien manualmente editar los archivos de codigo fuente respectivos y cambiar las ocurrencias apropiadas del nombre viejo del metodo al nuevo, o usar las capacidades para refactorear del IDE para hacer los cambios automaticamente. Para poder soportar este tipo de refactoreo, un IDE necesita una sofisticado comprehension del lenguaje de programacion en que esta escrito el codigo. Una herramienta de desarrollo sin ese entendimiento, por ejemplo una que hace una busqueda y reemplazo simple, podria introducir

errores. Por ejemplo, al renombrar un un metodo "read", la herramienta no deberia reemplazar identificadores como "readyState" que contienen el identificador a renombrar, ni tampoco reemplazar porciones de comentarios. Tampoco deberia suceder que renombrar una variable local afecte a variables con nombres similares en otros alcances. [Wikb]

Los compiladores e interpretees convencionales son usualmente incapaces de proveer estos servicios de lenguaje, dado que estan implementados con el objetivo de o bien transformar el codigo fuente en codigo objeto o ejecutar inmediatamente el codigo. Adicionalmente, los servicios de lenguaje deben poder manejar codigo que no esta bien formado, e.g. cuando el programador esta en el medio de editar y no ha terminado de escribir una expresion, procedimiento, u otra construccion del lenguaje. Mas aun, pequeños cambios que ocurren durante la escritura normalmente cambian la semantica del programa. Para poder proveer feedback instantaneo al usuario, la herramienta de edicion debe poder evaluar muy rapidamente las consequencias sintacticas y semanticas de una modificacion especifica. Los compiladores e interpretes, por lo tanto, son pobres candidatos para la produccion de la informacion necesaria para la consumicion por una herramienta de edicion. [Wikb]

Previamente al diseño e implementacion de LSP para el desarrollo de Visual Studio Code, la mayoria de los servicios de lenguaje estaban atados a un IDE o editor especifico. En la ausencia del LSP, los servicios son tipicamente implementados utilizando un API de extension especifica a una herramienta. Proveer el mismo servicio a otra herramienta de edicion requiere de un esfuerzo para adaptar el codigo existente para que el servicio pueda soportar las interfaces de la segunda herramienta. [Wikb]



LSP permite desacoplar los servicios de lenguaje del editor de tal manera que los servicios se pueden contener en un servidor de lenguaje de proposito general. Cual editor puede acceder a soporte sofisticado para muchos lenguajes diferentes al hacer uso de los servidores de lenguaje existentes. Similar-

mente, un programador involucrado en el desarrollo de un lenguaje nuevo puede crear servicios para ese lenguaje y hacerlo inmediatamente disponible para editores existentes. Hacer uso de servidores de lenguaje a traves del LSP por lo tanto tambien reduce la carga sobre los desarrolladores de herramientas de edicion, dado que no necesitan desarrollar sus propios servicios de lenguaje para cada lenguaje que quieren soportar. El LSP tambien habilita la distribucion y desarrollo de servidores contribuidos por terceros, tales como usuarios finales, sin participacion ni por parte de los desarrolladores del compilador del lenguaje o por los desarrolladores de la herramienta de edicion para la cual se esta agregando soporte de lenguaje. [Wikb]

LSP no se limita a lenguajes de programacion. Puede ser utilizacion para cualquier lenguaje basado en texto, tales como lenguajes de especificacion o especificos a dominions (DSL). [Wikb]

Velocidad de Compilación

Mejorar los tiempos de compilacion ha sido un foco principal despues de que Rust llego a la version 1.0. Sin embargo, gran parte del trabajo en pos de este objetivo ha sido sentar las bases arquitectonicas dentro del compilador.

Uno de los proyectos que esta construyendo sobre esta base, y que deberia mejorar los tiempos de compilacion para flujos de trabajo tipicos es la compilacion incremental. La compilacion incremental evita repetir trabajo cuando se compila un paquete, lo cual llevara en ultima instancia a un ciclo de edicion-compilacion-debug mas rapido. [Woe16]

Mecanismos

Compilación Incremental

La compilación incremental es una forma de computación incremental aplicada a la compilación. En contraste con compiladores comúnes que realizan "builds limpios" y ante un cambio en el código fuente recompilan todas las unidades de compilación, un compilador incremental solo recompila las unidades modificadas. Al construir sobre el trabajo hecho previamente, el compilador incremental evita la ineficiencia de repetir trabajo ya realizado. Se puede decir que un compilador incremental reduce la granularidad de las unidades de compilación tradicionales a la vez que mantiene la semántica del lenguaje. [Wika]

Muchas herramientas de desarrollo aprovechan compiladores incrementales para proveer a sus usuarios un entorno mucho mas interactivo. No es inusual que un compilador incremental sea invocado por cada cambio en un archivo fuente, de tal manera que el usuario es informado inmediatamente de cualquier error de compilación causado por sus modificaciones. Este esquema, en contraste con el modelo de compilación tradicional, acorta el ciclo de desarrollo considerablemente. [Wika]

Una desventaja de este esquema es que el compilador no puede optimizar fácilmente el código que compila, dada la localidad y el alcance reducido de los cambios. Normalmente esto no es un problema, dado que la optimización del código generado se aplica solamente al producir un *release build*, instancia en la cual se puede usar el compilador tradicional. [Wika]

Arquitectura Basada en Queries [Olle20]

Pasar de pipelines a queries [Olle20]

Que se necesita para obtener el tipo de un identificador calificado? En una arquitectura basada en pipelines, se buscaria el tipo en la tabla de simbolos. Con queries, hay que pensarlo de manera distinta. En lugar de depen-

der de haber actualizado un fragmento de estado, se computa de cero. En una primera iteracion, se hace siempre completamente de cero. Primero se averigua de cual archivo viene el nombre, y luego se leen el contenido del archivo, se parsea, posiblemente se realize algo de resolucion de nombres para saber a que nombres se refiere el codigo dado lo que se importa, y por ultimo se busca la definicion cuyo nombre ha sido resuelto y se verifica su tipo, finalmente retornandolo.

We first find out what file the name comes from, then read the contents of the file, parse it, perhaps we do name resolution to find out what the names in the code refer to given what is imported, and last we look up the name-resolved definition and type check it, returning its type.

```
fetchType :: QualifiedName -> IO Type
fetchType (QualifiedName moduleName name) = do
    fileName <- moduleFileName moduleName
    sourceCode <- readFile fileName
    parsedModule <- parseModule sourceCode
    resolvedModule <- resolveNames parsedModule
    let definition = lookup name resolvedModule
    inferDefinitionType definition
```

Refactoreando este esquema en funcionas mas chicas.

```
fetchParsedModule :: ModuleName -> IO ParsedModule
fetchParsedModule moduleName = do
  fileName <- moduleFileName moduleName
  sourceCode <- readFile fileName
  parseModule moduleName

fetchResolvedModule :: ModuleName -> IO ResolvedModule
fetchResolvedModule moduleName = do
  parsedModule <- fetchParsedModule moduleName
  resolveNames parsedModule

fetchType :: QualifiedName -> IO Type
fetchType (QualifiedName moduleName name) = do
  resolvedModule <- fetchResolvedModule moduleName
  let definition = lookup name resolvedModule
  inferDefinitionType definition
```

Notemos que cada una de estas funciones hace todo de cero, i.e. cada una realiza un prefijo cada vez mas largo del trabajo total que se haria en un

pipeline. Esto ha resultado ser un patron comun en compiladores basados en queries. Una forma de mejorar la eficiencia de este esquema es agregar una capa de memoizacion alrededor de cada funcion. De esta manera, se ejecuta trabajo copmutacionalmente demandante la primera vez que se invoca una funcion con un argumento dado, pero las llamadas subsiguientes son mas baratas porque pueden devolver el resultado cacheado. Esto es la esencia la arquitectura basadas en queries, pero en lugar de usar un cache por funcion, se utiliza un cache central, indexado por query. [arc20]

Why Incremental Compilation in the First Place? [Woe16]

Gran parte del tiempo de un programador se pasa en el ciclo de trabajo editar-compilar-debuguear:

- se realiza un pequeño cambio (frecuentemente en un unico modulo o funcion),
- se corre el compilador para convertir el codigo en un objeto ejecutable,
- se ejecuta el programa resultante o un conjunto de pruebas unitarias para ver el resultado del cambio.

Despues de eso, se vuelve al primer paso, realizar otro pequeño cambio informado por el conocimiento adquirido en la iteracion previa. Este bucle de alimentacion esencial es el nucleo de la actividad diaria de un programador. Se busca que el tiempo que se pasa detenido mientras se espera que el compilador produzca el compilador ejecutable sea lo mas breve posible.

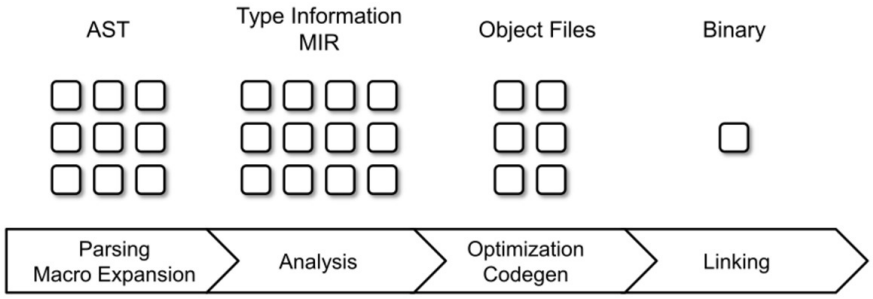
La compilacion incremental es una forma de aprovechar el hecho que poco cambia entre compilaciones durante el flujo de trabajo normal. Muchos, si no la mayoria, de los cambios entre sesiones de compilacion solo tienen un impacto local en el codigo maquina del binario producido, mientras que la mayor parte del programa, al igual que a nivel codigo, termina igual, bit a bit. La compilacion incremental apunta a retener la mayor parte posible de estas partes sin cambios a la vez que se rehace solo la cantidad de trabajo que debe hacerse. [Woe16]

How Do You Make Something "Incremental"? [Woe16

Ya se detallo que computar algo incrementalmente significa actualizar solo aquellas partes de la salida de la computacion que necesita ser adaptada en respuesta a los cambios dados en las entradas de la computacion. Una estrategia basica que podemos emplear para lograr esto es ver una computacion grande (tal como compilar un programa completo) como una composicion de muchas computaciones pequeñas interrelacionadas que construyen una sobre otra. Cada una de estas computaciones mas pequeñas producira un resultado intermedio que puede ser cacheado y reutilizado en una iteracion subsiguiente, evitando la necesidad de re-computar ese resultado intermedio en particular. [Woe16]

An Incremental Compiler [Woe16]

La forma en que se eligio implementar la incrementalidad en el compilador de Rust es directa: una sesion de compilacion incremental sigue exactamente los mismos pasos que una sesion de compilacion batch. Sin embargo, cuando el flujo de control llegue a un punto en el cual esta a punto de computar un resulto intermedio no trivial, intentara cargar ese resultado del cache de compilacion incremental en disco en su lugar. Si existe una entrada valida en el cache, el copmilador puede saltear la computacion de ese dato en particular. Este es un esquema (simplificado) de de las diferentes fases de compilacion y los resultados intermedios que producen: [Woe16]



Primero, el compilador parse el codigo fuente en un arbol de sintaxis abstracto (AST). El AST pasa luego a la fase de analisis que produce informacion de tipos y el MIR para cada funcion. Luego de eso, si el analisis no encontro ningun error, la fase de generacion de codigo transformara la version MIR del programa en su version de codigo maquina, emitiendo un archivo objeto por cada modulo de codigo. En la ultima fase todos los archivo

objeto son linkeados juntos en el binario final, que puede ser una libreria o un ejecutable. Hasta ahora las cosas parecen bastante simples: en lugar de computar algo por segunda vez, sencillamente cargar el valor desde el cache. Las cosas se complican cuando es necesario saber si es efectivamente valido usar un valor del cache o si hay que recomputarlo porque alguna entrada ha cambiado. [Woe16]

Seguimiento de Dependencias [Olle20]

El seguimiento, o *tracking*, de dependencias, es provista por librerias tales como llbuild, Shake, Rock, o Salsa. Estas librerias proveen parte de la funcionalidad necesaria para crear compiladores basados en queries.

Rock es una libreria experimental fuertemente inspirada por Shake y el paper *Build systems à la carte* [MMP18]. Esencialmente implementa un framework de sistema de build *build system framework*, como `make`. Los sistemas de build tienen mucho en comun con compiladores modernos dado que tambien queremos que sean incrementales, i.e. que aprovechen los resultados de builds anteriores al construir uno nuevo con pocos cambios. Sin embargo, tambien tienen una diferencia: a la mayoria de los sistemas de build no les importa el tipo de sus queries dado que trabajan sobre archivos y sistema de archivos. El esquema detallado en *Build systems à la carte* se aproxima mejor a lo necesario en el caso de un compilador. En esta publicacion detallan un sistema en el cual el usuario escribe un conjunto de computaciones, tareas, que toman una clave y retornan un valor, y elige un tipo adecuado para las claves y otro para los valores. Las tareas se formulan asumiendo que van a ser ejecutadas en un entorno en el cual existe una funcion `fetch` de tipo `Key -> Task[Value]`, donde `Task` es un tipo que describe las reglas del sistema de build, la cuale puede ser usada para obtener los valores de una dependencia con una clave dada. El sistema de build tiene control sobre que codigo corre al momento de ejecutar `fetch`, de tal manera que se puede variar la granularidad del seguimiento de dependencias, memoisacion, y actualizaciones incrementales. *Build systems à la carte* tambien explora que tipo de sistemas de build obtenemos cuando variamos lo que puede realizar una tarea *Task*, e.g. si es una monada o un aplicativo. Un problema que surge inmediatamente es que no hay ningun tipo satisfactorio para `Value`. E.g. puede haber una query para obtener el modulo donde esta definido un tipo, y otra para obtener la representacion del tipo dado el nombre calificado de un tipo. [arc20]

Verifying dependencies and reusing state [Olle20]

Una funcionalidad que se puede incorporar a las funciones regla son las actualizaciones incrementales. Cuando se utiliza, el sistema de build mantiene un registro en una tabla de cuales dependencias uso una tarea al ejecutarse, i.e. cuales claves busco y cuales fueron sus valores asociados. Usando esta informacion es posible determinar cuando es seguro reutilizar el cache de una corrida previa del compilador aunque haya habido cambios en otras partes del grafo de dependencias. Este seguimiento de grano fino de las dependencias tambien permite reutilizar el cache cuando la dependencia de una tarea cambia de tal manera que el cambio no tiene efecto. E.g. cambios en el espaciado pueden disparar un reparseo, pero dado que el AST permanece igual, los valores del AST cacheados pueden ser reutilizados en queries que dependan de ese resultado de parseo. [arc20]

Reverse dependency tracking [Olle20]

Verificar las dependencias puede ser demasiado lento para herramientas que deben proveer informacion en tiempo real como servidores de lenguaje, dado que puede ser necesario recorrer partes grandes del grafo de dependencias solo para verificar que no cambio nada aun para pequeños cambios. E.g. si se realizan cambios en un archivo fuente con una gran cantidad de imports, se debe caminar los grafos de dependencias de todos los imports para actualizar el estado del editor para ese unico archivo. Para evitar esto, se puede registrar las dependencias inversas entre queries. E.g. cuando un servidor de lenguaje detecta que un unico archivo ha cambiado, el grafo de dependencia inverso se usa para invalidar el cache unicamente para las queries que dependen de ese archivo, recorriendo las dependencias inversas comenzando desde el archivo modificado. Dado que los modulos importados no dependen de ese archivo, no deben ser verificados nuevamente, resultando en una respuesta mucho mas pronta del servidor de lenguaje. [arc20]

Caso de Estudio: Rustc

Rustc, el compilador de rust, tiene su propia implementacion de queries.

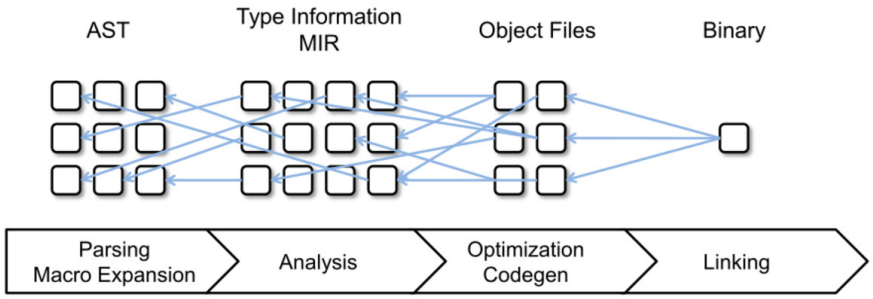
Rustc Dependency graphs [Woe16]

Existe un modelo formal que puede usarse para modelar los resultados intermedios de una computacion y su propiedad de estar actualizad de una manera directa: los grafos de dependencias. Cada entrada y cada resultado intermedio son representados como un nodo en un grafo dirigido. Los arcos en el grafo representan cual resultado intermedio o entrada puede tener influencia en otro resultado intermedio. En general no se puede asumir que los grafos de dependencias son arboles, sino grafos dirigidos aciclicos. Lo que convierte a esta estructura de datos en realmente util es que permite realizar consultas tales como "si X ha cambiado, entonces Y aun esta actualizado?". Para resolver esta consulta se examina un nodo Y y se colectan las entradas de las cuales Y depende transitivamente, siguiendo los arcos salientes de Y. Si alguna de esas entradas cambio, el valor cacheado para Y esta desactualizado y debe ser recomputado. [Woe16]

Dependency Tracking in the Compiler [Woe16]

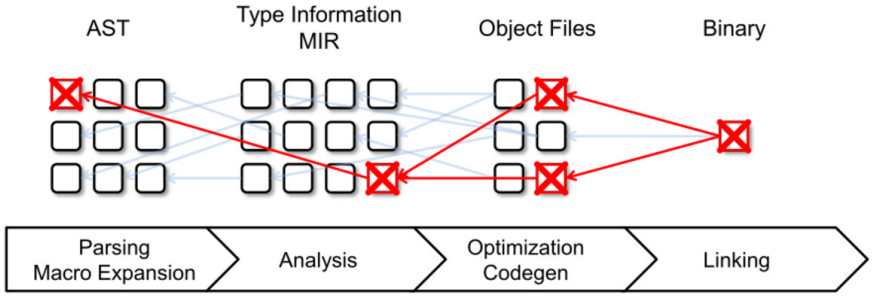
Al compilar en modo incremental, siempre se construye el grafo de dependencia de los datos producidos: cada vez que se escribe un dato (como un archivo objeto), se registra cuales otros datoa se acceden en el proceso. Se hace enfasis en el registro. En todo momento el compilador mantiene el registro de sobre cual dato esta trabajando (esto lo hace en background en memoria local al thread). Este es el nodo actualmente activo del grafo de dependencias. Por otro lado, el dato que necesita ser leido para computar el valor del nodo actual tambien se registra: usualmente ya reside en algun tipo de contenedor (e.g. una tabla hash) que requiere invocar un metodo de lookup para acceder a una entrada especifica. Se hace uso de este hecho,

haciendo que los metodos de lookup crean los arcos del grafo de dependencia, y sencillamente se agregan al grafo. Al final de las sesiones de compilacion se tienen todos los nodos enlazados, automaticamente. [Woe16]



Este grafo de dependencia se almacena en un directorio de cache de compilacion incremental, junto con las entradas que el cache describe. Al comienzo de una sesion de compilacion siguiente, se detecta cuales entradas (i.e. nodos

del AST) han cambiado, comparandolas con sus versiones previas. Dado el grafo y el conjunto de entradas que cambiaron, se puede facilmente encontrar todas las entradas en el cache que no estan actualizadas y eliminarlas del cache. Cualquier elemento que sobrevivio esta fase de validacion de cache puede ser re-utilizado durante la sesion de compilacion actual. [Woe16]



Existen algunos beneficios al esquema de seguimiento de dependencias

automatico que se esta empleando. Dado que esta incorporado a las interfaces internas del compilador, es dificil olvidarse accidentalmente de el. Si aun asi un desarrollador se olvida (e.g. no declarando el nodo activo) entonces el resultado es demasiado conservador, pero el grafo de dependencia aun sera correcto, impactara negativamente en la cantidad de reutilizacion pero no llevara a usar incorrectamente un dato desactualizado. [Woe16]

Tambien es de notar que el sistema no intenta predecir o computar como sera el grafo de dependencia. Una gran parte de las pruebas de regresion, sin embargo, si tendran una descripcion de como deberia ser el grafo de dependencia para un programa dado. Esto asegura que el grafo efectivo y el grafo de referencia se construyen por distintos metodos, reduciendo el riesgo de que tanto el compilador y el test esten de acuerdo en un valor incorrecto. [Woe16]

Algunas implicaciones: El grafo de dependencia refleja las dependencias efectivas entre partes del codigo fuente y partes del binario emitido. Si existe un nodo de entrada que es alcanzable de muchos nodos intermedios, e.g. un tipo de dato central que es utilizado en casi toda funcion, entonces cambiar la definicion de ese tipo de dato causara que casi todo debe ser compilado de cero.

En otras palabras, la efectividad de la compilacion incremental es muy sensible a la estructura del programa siendo recompilado y al cambio realizado. Cambiar un unico caracter del codigo fuente podria invalidar completamente el cache de compilacion incremental. Sin embargo, este tipo de cambio es un caso raro y la mayor parte del tiempo solo una pequeña porcion del programa debe ser recompilado. [Woe16]

The Current Status of the Implementation [Woe16]

El estado actual de rustc a fines de 2019. Para la primera implementacion de compilacion incremental, implementada a principios de 2019, el equipo de rustc de focalizo en cachear archivos objeto. Consequentemente, si esta fase se puede saltear aunque sea para parte de un codigo, se puede lograr el mayor impacto en tiempos de compilacion. Con esto en mente, tambien se puede estimar la cota superior de cuanto tiempo se puede ahorrar: si el compilador pasa N segundos optimizando cuando compila un crate, entonces la compilacion incremental puede reducir los tiempos de compilado en a lo sumo esos N segundos. Otra area que tiene una gran influencia en la efectividad de la primera implementacion de es la granularidad del seguimiento de dependencias. Depende de la implementacion cuan fina es la granularidad de los grafos de dependencias, y la implementacion actual es media gruesa. Por

ejemplo, el grafo de dependencias solo tiene un unico nodo para todos los metodos en un `impl` (bloque de implementacion de un trait). En consecuencia, el compilador considerara que cambiaron todos los metodos de ese `impl` aunque solo haya cambiado uno solo. Esto por supuesto significa que mas codigo sera recompilado de lo que seria estrictamente necesario. [Woe16]

Caso de Estudio: Rust-analyzer y Salsa

Rust-Analyzer utiliza una libreria llamada salsa.

Como funciona salsa?

La idea central de salsa es definir el programa como un conjunto de *queries*. Cada query se usa como una función $K \to V$ que mapea de una clave de tipo K a un valor de tipo V.

Las queries en salsa son de dos variedades basicas:

- **Entradas:** definen los inputs basicos al sistema, los cuales pueden cambiar en cualquier momento.
- **Funciones:** funciones puras (sin efectos secundarios) que transforman las entradas en otros valores. Los resultados de estas queries se memoizan para evitar recomputarlas. Cuando se modifican las entradas, salsa determina cuales valores memoizados pueden ser reutilizados y cuales deben ser recomputados.

El esquema general de utilizacion de salsa consiste en tres pasos:

1. Definir uno o mas grupos de queries que contendran las entradas y las queries requeridas. Se puede definir mas de un grupo para separar las queries en componentes.
2. Definir las queries.
3. Definir la base de datos, la cual contendra el almacenamiento para las entradas y queries utilizadas.

Conclusiones

La mayoria de los lenguajes modernos necesitan tener una estrategia en cuanto a la provision de herramientas de desarrollo, y el build de compiladores en base a sistemas de queries parace ser un acercamiento con mucha promesa. Con queries el desarrollador del compilador no necesita manejar explicitamente la actualizacion e invalidacion de un conjunto de caches ad hoc, lo cual puede ser el resultado cuando se agregan actualizaciones incrementales a una arquitectura de compilador tradicional en pipeline. En un sistema basado en queries se maneja el estado incremental de manera centralizada, reduciendo la posibilidad de errores. Las queries son excelentes para las herramientas porque permiten pedir por el valor de cualquier query en cualquier momento sin tener que preocuparse sobre el orden o efectos temporales, al igual que con un Makefile bien escrito. El sistema copmutara o recuperar valores cacheads por la query y sus dependencias automaticamente de una manera incremental. Los compialdores basados en queries son ademas sorprendentemente facil de paralelizar. Dado que se puede ejecutar cualquier query en cualquier momento, y se memoizan la primera vez que corren, se pueden disparar varias queries en paralelo sin preocuparse demasiado. [arc20]

Planes futuros para rustc (09/2019)

Los dos ejes principales a lo largo de los cuales se buscara mejorar la eficiencia de rustc son:

- Cachear mas resultados intermediosm como MIR e informacion de tipo, permitiendo que el compilador evite repetir mas y mas pasos.
- Mejorar la precision del seguimiento de dependencias, para que el compilador encuentre menos falsos positivos durante la invalidacion del cache.

[Woe16]

Source Notes

Anders Hejlsberg on Modern Compiler Construction

[Hej]

Build Systems a la Carte

[MMP18]

How Salsa Works (2019.01)

[Mat19a]

Salsa In More Depth (2019.01)

[Mat19b]

Fuentes

- General
  - ☑Youtube: Anders Hejlsberg on Modern Compiler Construction [Hej]
  - ☑Wikipedia: Incremental Compiler [Wika]
  - ☑Olle Fredriksson: Query-based compiler architectures [arc20]
  - ☑Rust Blog: Incremental Compilation [Woe16]
  - ☐Build Systems A La Carte [MMP18]
  - ☑Youtube: 2016 LLVM Developers' Meeting: D. Dunbar "A New Architecture for Building Software" [Dun16]

  - ☐Codebase as Database: Turning the IDE Inside Out with Datalog
  - ☐Three Architectures for a Responsive IDE
- Rustc Dev Guide
  - ☐Overview of the Compiler
  - ☐High-level overview of the compiler source
  - ☐Queries: demand-driven compilation
    - ☐The Query Evaluation Model in Detail
    - ☐Incremental compilation
    - ☐Incremental Compilation In Detail
    - ☐Debugging and Testing Dependencies
    - ☐Profiling Queries
    - ☐How Salsa works
- Rust Analyzer
  - ☐Rust Analyzer
  - ☐Manual
  - ☐Blog
  - ☐rust-analyzer/tree/master/docs/dev
  - ☐Rust Analyzer in 2018 and 2019
  - ☐Status of rust-analyzer
  - ☐2020 Intro to Rust Analyzer
  - ☐2020 What I learned contributing to Rust-Analyzer
  - ☐Youtube: Are we *actually* IDE yet? A look on the Rust IDE Story - Igor Matuszewski
  - ☐Youtube: Rust analyzer guide
  - ☐Youtube: rust analyzer syntax trees
  - ☐Youtube: rust-analyzer type-checker overview by flodiebold
  - ☐Youtube: Rust Analyzer Q&A
- Salsa
  - Github: salsa
  - ☐The Salsa Book
  - ☐Youtube: Incremental Compilation Working Group
  - ☐Youtube: Responsive compilers - Nicholas Matsakis - PLISS 2019
  - ☑Youtube: Things I Learned (TIL) - Nicholas Matsakis - PLISS 2019
  - ☐Youtube: How Salsa Works (2019.01)
  - ☐Youtube: Salsa In More Depth (2019.01)
  - ☐Youtube: RLS 2.0, Salsa, and Name Resolution

- Rust Compilation Speed
  - □How to alleviate the pain of Rust compile times
  - □Nethercote: How to speed up the Rust compiler
  - □Nethercote: How to speed up the Rust compiler some more
  - □Nethercote: How to speed up the Rust compiler in 2018
  - □Nethercote: How to speed up the Rust compiler some more in 2018
  - □Nethercote: How to speed up the Rust compiler in 2018: NLL edition
  - □Nethercote: The Rust compiler is getting faster
  - □Nethercote: The Rust compiler is still getting faster
  - □Nethercote: How to speed up the Rust compiler in 2019
  - □Nethercote: How to speed up the Rust compiler some more in 2019
  - □Nethercote: How to speed up the Rust compiler one last time in 2019
  - □Nethercote: How to speed up the Rust compiler in 2020
  - □Nethercote: How to speed up the Rust compiler some more in 2020
  - □Nethercote: How to speed up the Rust compiler one last time
  - □PingCAP Blog: The Rust Compilation Model Calamity
  - □PingCAP Blog: Generics and Compile-Time in Rust
  - □PingCAP Blog: Rust's Huge Compilation Units
  - □PingCAP Blog: A Few More Reasons Rust Compiles Slowly
- Miscellaneous
  - □Youtube: Making Fast Incremental Compiler for Huge Codebase - Michał Bartkowiak - code::dive 2019
  - □Youtube: Starting with Semantics - Sylvan Clebsch - PLISS 2019
  - □Youtube: Polyhedral Compilation as a Design Pattern for Compilers (1/2) - Albert Cohen - PLISS 2019
  - □Youtube: Polyhedral Compilation as a Design Pattern for Compilers (2/2) - Albert Cohen - PLISS 2019
  - □Youtube: First-Class Continuations: What and Why - Arjun Guha
  - □Youtube: Implementing First-Class Continuations by Source to Source Translation - Arjun Guha - PLISS 2019
  - □Youtube: Static Program Analysis (part 1/2) - Anders Møller - PLISS 2019
  - □Youtube: Static Program Analysis (part 2/2) - Anders Møller -

PLISS 2019

Bibliografia

[Dun16] Daniel Dunbar. *2016 LLVM Developers' Meeting: A New Architecture for Building Software*. Dec. 2016. URL: https://www.youtube.com/watch?v=b_T-eCToX1I.

[Woe16] Michael Woerister. *Rust Blog: Incremental Compilation*. Sept. 2016. URL: https://blog.rust-lang.org/2016/09/08/incremental.html.

[MMP18] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. "Build systems a la carte". In: *Proc International Conference on Functional Programming (ICFP'18)*. ACM, Sept. 2018. URL: https://www.microsoft.com/en-us/research/publication/build-systems-la-carte/.

[Mat19a] Nicholas Matsakis. *Youtube: How Salsa Works (2019.01)*. Jan. 2019. URL: https://www.youtube.com/watch?v=_muY4HjSqVw.

[Mat19b] Nicholas Matsakis. *Youtube: Salsa In More Depth (2019.01)*. Jan. 2019. URL: https://www.youtube.com/watch?v=i_IhACacPRY.

[arc20] Query-based compiler architectures. *Olle Fredriksson*. June 2020. URL: https://ollef.github.io/blog/posts/query-based-compilers.html.

[Hej] Anders Hejlsberg. *Anders Hejlsberg on Modern Compiler Construction*. URL: https://www.youtube.com/watch?v=wSdV1M7n4gQ.

[Mic] Microsoft. *Language Server Protocol*. URL: https://microsoft.github.io/language-server-protocol/.

[Wika] Wikipedia. *Incremental Compiler*. URL: https://en.wikipedia.org/wiki/Incremental_compiler.

[Wikb] Wikipedia. *Language Server Protocol*. URL: https://en.wikipedia.org/wiki/Language_Server_Protocol.