Tesina

Iñaki Garay

Septiembre 2020

Contents

Introducción / Esquema General

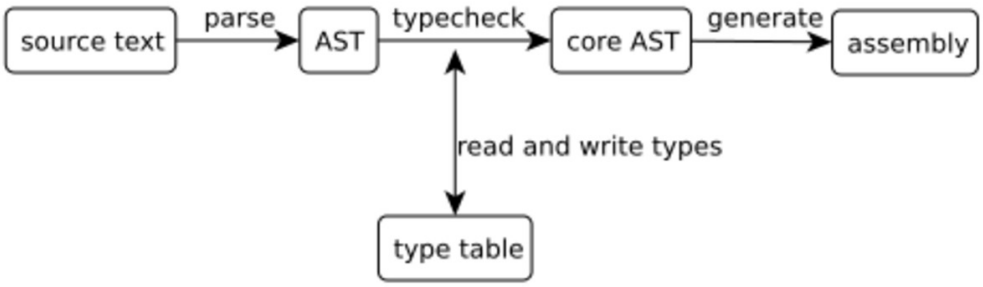
- Los compiladores tradicionales tienen una arquitectura de pipeline.
- Dos requerimientos nuevos paralelos: minimizar tiempo de compilacion y proveer mas informacion de manera mas interactiva a las herramientas de edicion.
- Los editores y entornos de desarrollo modernos usan LSP (Language Server Protocol). 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

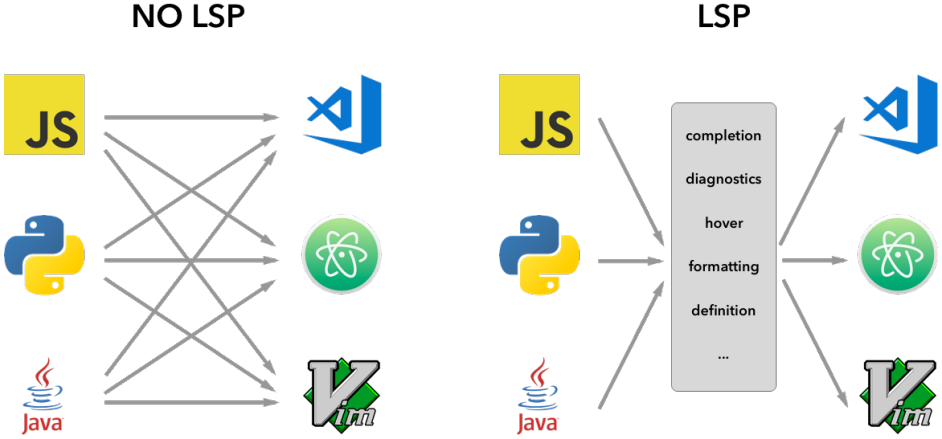
El Language Server Protocol (LSP) es una protocolo abierto basado en JSON-RPC para el uso entre editores de codigo fuente o entornos de desarrollo integrados (IDE) y servidores que proveen funcionalidades especificas a lenguajes de programacion. El objetivo del protocolo es permitir que se implemente 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. Estas

```
- However, when control flow reaches a point where it is
about to compute some non-trivial intermediate result, it
will try to load that result from the incremental
compilation cache on disk instead.

- If there is a valid entry in the cache, the compiler can
just skip computing that particular piece of data. Let's
take a look at a (simplified) overview of the different
compilation phases and the intermediate results they
produce:

- First the compiler will parse the source code into an
abstract syntax tree (AST). The AST then goes through the
analysis phase which produces type information and the MIR
for each function. After that, if analysis did not find any
```

errors, the codegen phase will transform the MIR version of the program into its machine code version, producing one object file per source-level module. In the last step all the object files get linked together into the final output binary which may be a library or an executable.

- So, this seems pretty simple so far: Instead of computing something a second time, just load the value from the cache. Things get tricky though when we need to find out if it's actually valid to use a value from the cache or if we have to re-compute it because of some changed input.

[Woe16]

Seguimiento de Dependencias [Olle20]

- Rock, Shake, Salsa

- This functionality is provided by Rock, a library that packages up some functionality for creating query-based compilers.

- Rock is an experimental library heavily inspired by Shake and the Build systems à la carte paper.

- It essentially implements a build system framework, like
- make.

- Build systems have a lot in common with modern compilers since we want them to be incremental, i.e. to take advantage of previous build results when building anew with few changes.

- But there's also a difference: Most build systems don't care about the types of their queries since they work at the level of files and file systems.

- Build systems à la carte is closer to what we want.

- There the user writes a bunch of computations, tasks,

choosing a suitable type for keys and a type for values.

- The tasks are formulated assuming they're run in an environment where there is a function fetch of type Key -> Task Value, where Task is a type for describing build system rules, that can be used to fetch the value of a dependency with a specific key.

- In our above example, the key type might look like this:

- The build system has control over what code runs when we do a fetch, so by varying that it can do fine-grained dependency tracking, memoisation, and incremental updates.

- Build systems à la carte is also about exploring what kind of build systems we get when we vary what Task is allowed to do, e.g. if it's a Monad or Applicative.

- In Rock, we're not exploring that, so our Task is a thin layer on top of IO.

- A problem that pops up now, however, is that there's no satisfactory type for Value.

- We want fetch (ParsedModuleKey "Data.List") to return a ParsedModule, while fetch (TypeKey "Data.List.map") should return something of type Type.

[arc20]

Indexed queries [Olle20]

- Rock allows us to index the key type by the return type of the query. The Key type in our running example becomes the following GADT:

```
data Key a where
  ParsedModuleKey :: ModuleName -> Key ParsedModule
  ResolvedModuleKey :: ModuleName -> Key ResolvedModule
  TypeKey :: QualifiedName -> Key Type
```

- The fetch function gets the type forall a. Key a -> Task a, so we get a ParsedModule when we run fetch (ParsedModuleKey "Data.List"), like we wanted, because the return type depends on the key we use.

- Now that we know what fetch should look like, it's also worth revealing what the Task type looks like in Rock, more concretely.

- As mentioned, it's a thin layer around IO, providing a way to fetch keys (like Key above):

- The rules of our compiler, i.e. its "Makefile", then becomes the following function, reusing the functions from above:

```
rules :: Key a -> Task a
rules key = case key of
  ParsedModuleKey moduleName ->
    fetchParsedModule moduleName
  ResolvedModuleKey moduleName ->
    fetchResolvedModule moduleName
  TypeKey qualifiedName ->
    fetchType qualifiedName
```

[arc20]

Caching [Olle20]

- The most basic way to run a Task in Rock is to directly call the rules function when a Task fetches a key.

- This results in an inefficient build system that recomputes every query from scratch.

- But the Rock library lets us layer more functionality onto our rules function, and one thing that we can add is memoisation.

- If we do that Rock caches the result of each fetched key by storing the key-value pairs of already performed fetches in a dependent hashmap.

- This way, we perform each query at most once during a single run of the compiler.

[arc20]

Verifying dependencies and reusing state [Olle20]

- Another kind of functionality that can be layered onto the rules function is incremental updates. When it's used, Rock keeps track of what dependencies a task used when it was executed (much like Shake) in a table, i.e. what keys it fetched and what the values were.

- Using this information it's able to determine when it's safe to reuse the cache from a previous run of the compiler even though there might be changes in other parts of the dependency graph.

- This fine-grained dependency tracking also allows reusing the cache when a dependency of a task changes in a way that has no effect.

- For example, whitespace changes might trigger a re-parse, but since the AST is the same, the cache can be reused in queries that depend on the parse result.

[arc20]

Reverse dependency tracking [Olle20]

- Verifying dependencies can be too slow for real-time tooling like language servers, because large parts of the dependency graph have to be traversed just to check that most of it is unchanged even for tiny changes.

- For example, if we make changes to a source file with many

large imports, we need to walk the dependency trees of all of the imports just to update the editor state for that single file.

- This is because dependency verification by itself needs to go all the way to the root queries for all the dependencies of a given query, which can often be a large proportion of the whole dependency tree.

- To fix this, Rock can also be made to track reverse dependencies between queries.

- When e.g. a language server detects that a single file has changed, the reverse dependency tree is used to invalidate the cache just for the queries that depend on that file by walking the reverse dependencies starting from the changed file.

- Since the imported modules don't depend on that file, they don't need to be re-checked, resulting in much snappier tooling!

[arc20]

Caso de Estudio: Rustc

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

Rustc Dependency graphs [Woe16]

```
- There is a formal method that can be used to model a
computation's intermediate results and their individual
"up-to-dateness" in a straightforward way: dependency
graphs.

- It looks like this: Each input and each intermediate
result is represented as a node in a directed graph. The
edges in the graph, on the other hand, represent which
intermediate result or input can have an influence on some
other intermediate result.

- Note, by the way, that the above graph is a tree just
because the computation it models has the form of a tree. In
general, dependency graphs are directed acyclic graphs

- What makes this data structure really useful is that we
can ask it questions of the form "if X has changed, is Y
still up-to-date?". We just take the node representing Y and
collect all the inputs Y depends on by transitively
following all edges originating from Y. If any of those
inputs has changed, the value we have cached for Y cannot be
relied on anymore.
```

[Woe16]

Dependency Tracking in the Compiler [Woe16]

- When compiling in incremental mode, we always build the dependency graph of the produced data: every time, some piece of data is written (like an object file), we record which other pieces of data we are accessing while doing so.

- The emphasis is on recording here. At any point in time the compiler keeps track of which piece of data it is currently working on (it does so in the background in thread-local memory).

- This is the currently active node of the dependency graph. Conversely, the data that needs to be read to compute the value of the active node is also tracked: it usually already resides in some kind container (e.g. a hash table) that requires invoking a lookup method to access a specific entry.

- We make good use of this fact by making these lookup methods transparently create edges in the dependency graph: whenever an entry is accessed, we know that it is being read and we know what it is being read for (the currently active node).

- This gives us both ends of the dependency edge and we can simply add it to the graph. At the end of the compilation sessions we have all our data nicely linked up, mostly automatically.

- This dependency graph is then stored in the incremental compilation cache directory along with the cache entries it describes.

- At the beginning of a subsequent compilation session, we detect which inputs (=AST nodes) have changed by comparing them to the previous version. Given the graph and the set of changed inputs, we can easily find all cache entries that are not up-to-date anymore and just remove them from the cache.

- Anything that has survived this cache validation phase can safely be re-used during the current compilation session.

- There are a few benefits to the automated dependency tracking approach we are employing. Since it is built into the compiler's internal APIs, it will stay up-to-date with changes to the compiler, and it is hard to accidentally forget about. And if one still forgets using it correctly (e.g. by not declaring the correct active node in some place) then the result is an overly conservative, but still "correct" dependency graph: It will negatively impact the re-use ratio but it will not lead to incorrectly re-using some outdated piece of data.

- Another aspect is that the system does not try to predict or compute what the dependency graph is going to look like, it just keeps track. A large part of our (yet to be written) regression tests, on the other hand, will give a description of what the dependency graph for a given program ought to look like. This makes sure that the actual graph and the reference graph are arrived at via different methods, reducing the risk that both the compiler and the test case agree on the same, yet wrong, value.

[Woe16]

- Let's take a look at some of the implications of what we've learned so far:
  - The dependency graph reflects the actual dependencies between parts of the source code and parts of the output binary.
  - If there is some input node that is reachable from many intermediate results, e.g. a central data type that is used in almost every function, then changing the definition of that data type will mean that everything has to be compiled from scratch, there's no way around it.
- In other words, the effectiveness of incremental compilation is very sensitive to the structure of the program being compiled and the change being made. Changing a single character in the source code might very

```
well invalidate the whole incremental compilation cache.
Usually though, this kind of change is a rare case and
most of the time only a small portion of the program has
to be recompiled.
```

[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 la construccion 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]

2016 LLVM Developers' Meeting: D. Dunbar "A New Architecture for Building Software"

[Dun16]

```
## Overview

compile times impact developers
clang was designed from the beginngin to be a very fast c/c++
that was one of the motivations

the strategy was
- to have a tuned lex and parse implementation
- focused heavily on having a low overhead -O0 path, no unnecesary
  optimizations on -O0, and do things such as fast instruction
  selection, etc
- redesigned pch (pre compiled headers), try to pull the minimal
  amount of data from headers
- integrated the assembler into the compiler, to avoid the time of
  emitting assembly code and loading it back into the assembler
```

this proved succesful, clang was 3x faster than gcc
but over time this lead has diminished
in part because gcc itself got faster but mainly because

performance regresses over time because features are added, and
tuning can break

improving compile time, options:
- distributed compilation
- improved caching, ideally distributed and shared
- do less work

there are things that in the case of clang could be done today,
such as reuse the frontend for compiling several files sharing the
same compile flags, allowing the frontend to better chache
e.g. file stat calls and other metadata, or build pch for hotly
edited files.

while this works, the problem is that there is no control in the
compiler over how it is invoked, as this depends on the build system
one proposal is to build a compiler-service, but this presents other
problems

How we build software today

for most languages, traditional unix/build system model
- compiler runs as a separate process
- primitive mechanisms for communicating dependencies between build
  systems and the compiler
  - the build system tells us about our inputs andoutpus through
    command line arguemtns
  - mechanism by which the compiler can write out additional
    dependencies the build system can ingest, we dont even have a
    real file format we use for that, we just write out a makefile
    format and force the build system to ingest it
  - fixed input/output, we only have a limited set of places where
    were supposed read and write data from, if the compiler wanted
    to cache data on the side, we dont even have an agreed upon
    location to do that

so from a certain perspective, this is basically an api contract,
and its an api that hasnt changed in decades

we last major advancement was this argument that lets the compiler send dependencies back to the build system

how software could be built

what would happen if we were willing to break this api?

things that could be done
- ad hoc lookup tables:
  if you think about the lifetime of the compiler throughout the build, there are many places where the compiler ends up computing some amount of information in every build that could be accelerated through a lookup table
  and if we had a way to keep that lookup table around, that would make the build faster
  but today that would require us to serialize it out to disk and read it back in, and that would mitigate some of the speedup, so we dont do that
- early exit via output signatures:
  we could implement in the compiler the ability to take a signature of the llvm ir after it comes out of codegen, and if that code signature is the same as the last time that file was compiled, we can skip the backend, we know we are going to get the same object file out
  this is a big win if you change a comment in a file that causes all the project to rebuild, but none of your object files are going to change.
- dealing with redundant template instantiaions
  we spend a fair amount of time throughout the build doing a lot of work for the same template instantiation
  creating them, typechecking them, generating the code for them, and then finally passing them all off to the linker, for it to get rid of most of them

for all these things whats need is the ability to evolve the api between the build systema and the compiler

what about a module cache?

it caches something throughout the whole build.

yes:
- automatically builds modules when needed

- shares results across the build
- no build system changes needed

nonexample because it involves significant implementation complexity
- has posix file lockingg in order to manage coordination
- has its own cache consistency management scheme, few debugging tools, it has to watch and understand when it needs to rebuild modules
- has to have its own eviction mechanism (automatic pruning, tuning parameters)
- opaque to the build system scheduler, if one starts a bunch of build jobs and they all want the same model, from the build systems perspective, it going to be blocked, waiting for all of those, but it actually could be doing more work on the core that it has, it just doesnt know that.

ideal model for building software

- what we really want is some kind of flexible api between the c ompiler and the build system

goals:
- it should be really easy to share redundant work, if we see a place in the compiler where we realize its redundant throughout the lifetime of the build, we should be able to implement that quickly
- optimize the compiler for the entire build: most users, other than compiler engineers, dont care about objetct files, what they care about is getting their final product out
  thats the situation we should optimize the compiler at, being fast for doing the entire build
- conversely, we should be able to optimize the build system via a rich compiler api
  if theres a place where the compiler realizes it could do some amount of work in parallel, it would be relly nice if we could talk back to the build system and communicate with it so that the job can be scheduled effectively
- consistent incremental builds and debuggable architecture: one of the benefitrs of the curent strategy is that we try very hard on the compiler to make this strong guarantee that the same input will produce the exact same object file output, that 's a very good basis for having reliable reproducible builds.

- need ability yo integrate the build system and the compiler

```
requires:
- library based compiler: a compiler that has been architected to be
  able to be used that way
- extensible build system, a build system that has been designed with
  the idea in mind that some of the tools that it interacts with
  should be able to plug in more deeply than just as an extensible
  process.
  when you call make or ninja, they expect to call you via subprocess
  and theres a few ways you can interact with them but not a lot
- compiler plugin for that

## introducing llbuild

goals:
- ignore build description / input language
  not make a new syntax as a replacement for cmake
  most build systems, somewhere inside they have a little engine
  that is capable of evaluating a dependency graph
  that engine is usually fairly simple
- with llbuild focus on building a powerful engine
  - support work being discovered on the fly
    very few build can support the situation where your executing
    some task and during the execution of that task you realize you
    need to do more work
  - scale to millions of tasks
    because the goal was to take our existing build and partition
    them into much smaller pieces to be able to get better
    incremental beehaviour it really needed to be able to scale
  - support sophisticated scheduling
    during a build of llvm we have a lot of cpu boudn stuff running
    the compipler, and we have a lot of io bound stuff, like runner
    the linker , which sucks in all the object files
  - powerful debugging tools
- support a pluggable task api

### llbuild architecture

- flexible underlying core engine
  the llvm ir equivalent, a common denominator between the high level
  things, build systems, and the lower level optimization
  - library for persistent, incremental computation
  - heavily inspired by a haskell build system called shake
```

```
  - low-level
    - inputs and outputs are byte strings
      there are no files, just strings of bytes, and the expectation
      that as a client, you'll encode data in those if necessary
    - functions are abstract
    - use c++ api between tasks
  - on top of this core: hiher-level build systems are built on the
    core there's a ninja implementation on top of this, a package
    manager

### llbuild engine

what would it mean to factor out this kind of more minimal engine
what it actually looks like

- minimal, functional model
  four basic concepts
  - keys: unambiguos name for some comutation you want to perform
  - value: the result of a computation
  - rule: how to produce a value for a key
  - task: a runing instance of a rule
    - a task can request other input keys as part of its work

the core engine can be used directly for general purpose
computation

recursive functions foprma a natural graph each result
depends on the recursive inputs

### example: ackerman

to build ackerman we encode the ackerman invocation we want
to make as a key, we encode the integer result as a value,
then we take those keys and we map them to tasks using our
rules, and then the tasks themselves implement the ackerman
function

we have simple wrapper struct that wraps the integer pair
that the arguments,

it has a couple functions to convert it to and from a
serialized representation the value is the same thing
```

where it gets more interesting with rules

the way llbuild works is you give it a delegate,

it can handle this idea of work being generated on the fly, and the reason it can do it is you jhust give it a delegate, and you give it a function that provides it rules when it wants them

so unlike a ninja file where all of the rules are present in advance, from the engines perspective it just has a function to request rules as they come in

so in this case our function just decodes that key and will create a task if its requested

it has some other stuff like it can report to you if it found a cycle

so tasks are where the real work happens

the way that the tasks work is that the engine will call back into your tasks as interesting stuff happens happens

when a task is started, youre notified

when the engine has computed a result you asked for, then its given to you

when all of the inputs that your task has asked for have been computed, then it shluld complete

a new architecture

requires:
- library based compiler
- extensible build system
- compiler plugin
  the las tthing we need to try to gete performance
  imporments is this

straw man proposal:

```
## What's the status of this 2 years later?

Funny you should ask! llbuild is alive and strong, and now
powers the default build system used in Xcode 10. Getting
the "new architecture" proposed here off the ground has
continued to be difficult, mostly due to the challenge of
refactoring the existing compilers, and that the payoffs
only start to get realized once a large amount of work has
been done. It is hard to unwind 40 years of CS-precedent for
how compilers are built overnight, but I am still optimistic
about the potential here!

... and probably worth adding: since this was published
llbuild's own Ninja implementation usually outperforms the
actual Ninja on most real world situations we've seen (such
as building LLVM/Clang on a variety of different hardware).
```

Responsive compilers - Nicholas Matsakis - PLISS 2019

[Mat19b]

```
# pipelines and passes

when I started writing compilers, we used the dragon book I
learned this classic structure of how to write a compiler
which was all about passes

that is precisely how rustc used to look

the reason that there has been a change is that the way you
interact with comilers has changed

the way the rust compiler was written and the way compilers
usually work is this batch compilation model

where you run it, process the whole source and you produce
an output and maybe get an error out of it
```

these days people are working with ides and you want a different way to interact with the source when your in this model

you want to take messed up inputs, make sense of them, you want to be able to do compiletions and jump to definitions in an interactive way

my goal today:
what if the dragon book were written today

i think there are more questions marks than answer, we dont have all the things written down to make the book

but I have a lot of experiences of what we tried and the challenves

the first thing to learn about this environment today is that there has been a big shift the last couple of years in how ides are written is that microsoft introduced vs code, which is an amazing editor, but among the many amazing things it introduced is the LSP, which is an intermediate protocol for interfacing between the language that's being compiled and the editor that's interacting, wo that neither have to be tied to each other

it used to be that when you wrote an eclipse plugin for your language it just worked in exclsipse, and then if you wanted to 4extend for netbeans, and emacs, and vim, etc

but LSP lets you sidestep that

so for example in the rust compiler we have a language service, we actually ahve to and they work for emacws, vi, whatever

The "responsive" compiler

- compiler as an actor
- editor sends diffs and requests completions, diagnostics
- compiler responds

what you wind up in this model is that instead of having the

compiler be something that you run in the command line it's more of an actor, the editor is sending you diffs of the files youre compoiling, and you are responding to them and sending back diagnostics,

and it might say, ok, we want to know what are the completions at this point, and your compiler responds with a vector of responses

key point: need to be able to respond as quickly as possible

so you wind up with a pretty different structure

andd you ahve to think about what is the minimum information you need to answer that requests, and can you process just that so you can get responses as fast as possible

so instead of a type checker that walks all the sources and does all the things for all the functions, ok i just need to type check this statement, how much context do i need to do that

and i go back out to the function, i have to go back out to other functions sometimes, but not all of them, find their signatures, etc

demand driven

so what weve been trying to do is move to a more demand driven architecture

- start from goal
- figure out what is needed for that

you have a given goal you need to do, and that goal is implemented by some function that uses other functions, and you go backwards, but you try to keep this set to a minimum

at the end of the day you still have thesed traditional compiler passes, logical, but you might not be executing them completely and you might now tbe executing them in order

```
# why should you care about IDEs?

there are some things ive noticed in trying to make this
transformation that surprised me

and there some reasons to try to do an ide friendly
approach, even if its not a full ide, from the beginning

- you have to write code in this language your making
- it really informs your language design,
  you becoime much more aware of what depoendencies you need
  to figure out bits of information and that might lead you
  to make or not make certain desicions
- stric phase separation is impossible anyway

# dependencies matter

rust allows arbitrryu nesting of delcarations inside
fuinctions an example of something we would not have done
had we implemented ides earlier on

rust has always allowed you to nest things rather
arbitrarily for example a function with a struct inside of
it

thats kinda handy, sometimes you want some local data that's
not needed outside the function

fn foo() {
  // Equivalent to a struct declared at the root of the
  // file, but only visible inside this function.
  struct Bar{}
  let x = Bar{};
}

you can also put methods on the struct

fn foo() {
  struct Bar{}
  impl Bar {
    pub fn method() { ... }
  }
  let x = Bar{};
```

```
}
```

a side effect of this is that auto-completion requires looking inside a lof of function bodies

what tha means is that I could have a struct visible from outside the function, and put methods on it inside the function, and I can call those methods from outside the function, because the methods are dispatched based on the type, and we attached the method to type Bar, and if I have an instance of Bar from outsidfe the function I can call and what that means is that I'm doing completion on a value of type Bar, I really need to parse the inside of my function bodies to figure out if there is an impl that muight be releant, or else I wont get those methods

```
struct Bar {}
fn some_method() {
  let bar = Bar::new();
  bar. // <-- what methods should we offer as
       // auto-completion here?
}
```

strict phase separation is impossible anyway

the other reason that led us in this direction, is that in most compiler that I've worked on , if you have a strict phase spearation in which you fully resolve all the symbols, then type check all the bodies, and then... it e3nds up kind of constraining, and you often need to process your source in a difficult order

in many languages its a constraint you dont want

Rust, for example, lets you do this:

```
const LEN: u8 = 1 + 1 + 1;
const DATA: [u8; LEN] = [1, 1, 1]:
```

what this means now is that there is an interdependency, in order to know the full type of DATA, I have to evaluate the constant LEN, and in order to evaluate the constant LEN, I have to type check its body and execute in some way,

interpret it, symbolically execute it, to figure out what it's value is, and that means I can't fully type check all the constants in one order, without considering the dependencies between them before I can even figure out the type of data.

What we used to do is some horrible hack.

We essentially had two implementations of some parts of the compiler, because we needed to have some subset of the typechecker and evaluator that was good enough to evaluate things like LEN and that could execute at any part on demand, and then we had the real code that did the full check that came after.

It was a horrible pain.

And now in this more demand based system this isnt a problem because we can go and execute LEN on its own.

What if we were to do this?

```
const LEN: u8 = DATA[0] + DATA[1] + DATA[2];
const DATA: [u8; LEN] = [1,1,1];
```

Usually when you're doing this sort of thing you end up needed to detect cycles and this kind of falls out out of the framework we're working on basically.

Other examples of phase separation:
- inferred types across function boundaries in e.g ML
- in rust, theres a bunch of things in the logic language: specialization, which requires solving some traits
- java and its lazy class file loading, how many different things the dot operator does in java, you will find that there is a bunch of lazyness, the set of classes a given classs can touch is determined as you walk the file you compile
- how racket deals with phase separation and scheme macros

in a lot of languages you find you want to evaluate some subset of the source and type check and be able to woirk with thme without necesarily processin ght whoel thing

Not a solved problem

what have we been actually doing to solve this

- hand coded
- salsa
- more formal technique
  - attribute grammars
  - datalog or structured queries

rustc takes an approach based on a framework called salsa
it enables you to still write your compiler in a general
purpose programming language what feels familiar

there are two different alternative to this,

one of them, the hand coded version, ive seen more in
practice in other places, is not having a framework but
thinking very carefully about things, doing the same things
but open coded, doing things by hand figuring out if im
going to type check this, i need to figure out these
dependencies, and making it work

that is of course very practical, it just can have bugs,
incremental inconsistencies, etc if you have forgotten about
a dependency between things

another way is more formal, higher level expression

then you have to make sure that everything you do fits in to
this framework or extend the framework

so salsa is kind of a middle ground

Salsa
- high level idea:
  - inputs
  - derived queries

- most closely related work:
  - adapton
  - glimmer from the ember web framework

- build systems a la carte

the high level idea of this framework is that you separate out the inputs to your compilation and then a bunch of derived stuff.

the derived queries are basically pure functions that get to demand other results/queries, that they needs, some of which may be inputs and when we use one of these functions we track what bits of data did it use and ultimately which inputs did it use, and then whene theres a cahnge to one of these inputs we can propagate the change and try to avoid re executing some of these things.

there are a lot of systems in this space

the three that I know of are these

two of these are academic, adapton approach by mathew hammer and BSALC a paper by simon peyton jones that built a very flexible system in haskell that has a similar basis

teh difference would be that ours is in between BSALC allows you to customize a lot of different thing and tweak many things, adapton is also a little more flexible but also more complicated

an interesting approach is glimmer engine in the ember web framework, which does incremental updates. i havent looked deeply at what react or elm does but i imagine they are kinda related

turns out this is a problem that applies to many areas, not just compilation

Salsa core idea

```
let mut db = Database::new();
loop {
  db.set_input_1( ... );
  db.set_input_2( ... );

  db.derived_value_1();
```

```
  db.derived_value_2();
}
```

when you are writing a program in salsa it kinda looks like this and then you essentialy have a loop where you set some inputs, and that's like when you get a diff from the editor, and you compute some derived values

and the idea is that these things are memoized, so whenever you ask for a derived value it will always be up to date, for whatever the input changes you have made

Entity component System

- entity: unit of entity
- component: data about an entity

```
fn move_left(entity: Entity, amount: usize) {
  let pos = DB.position(entity);
  pos.x -= amount;
  DB.set_position(entity, pos);
}
```

when you really try to write a whole compiler withj this kind of model one of the things that arises is this relationship to ECS, this is kind of a right turn but I want to explain this as a sort of background as a way to think about how this feels in practice

an ECS is something that arises from game proigramming and is an alternative to OOP

where you separate out data and the identity

so in an OOP you have a class and you make an instance of it and all of iots dasta and its operation are defined at that moment when you created it

whereas in an ECS you create a new entitty aND it has nothing associated with thi but identity, and then you can have separately data that you attache to it

in games this is useful because of the very dynamic nature
of data in a game.

it allows you to be unstructured and in a compilar that part
is not as importnat but its pretty useful

Entities in a compiler

- often called symbols
- things like
  - input files
  - struct declarations
  - fields
  - function declarattions
  - parameters or local variables
- something "addressable" by other parts of the system

so what you wind iup with is a system where you entities
correspond to things that get declared in the program
language and you layer on different bits of data about this
symbols so you might have a type, etc

and ther reason you layer this on is this is what allows us
to be so demand driven

we can ask about the type of a symbol and get that without
getting all the other bits of data that might eventually
come to be associated with it

Components in a compiler

things like
- the type of an entity
- the signature of a function

there are a couple different things that are like
components, type is one, signature is another, sometimes
there are more like unit results, or lists of errors, the
result of applying some analysis that can reject

```
# Salsa queries

Q(K0 .. Kn) -> V
- Q is the query name (like AST)
- the K0 .. Kn are the query keys
  - atomic values of any type
- The V is the value associated with the query

the basis structure of this salsa  systems is that you have
queries, which have a name
its not quite an ECS in that we dont have a formal concept
of entities instead what we have is queries which are kind
of like components
a query name might be like the type or the signature
and then we have a set of keys that go into the query
and often there is only one

# Example queries
# Query group
# Input queries
# Derived Queries
# How salsa works
# Recomputation (simplified)
# But suppose input change is not important
# Recomputation (less simplified)
# Order matters
# Minimizing redundant checks
# Garbage collection
# General idea
# Layering
# Represent layers with maps
# Rust compiler of yore
# Trees are your friends
# Interning
# Tree-based entities also give context
# Signature
# Tightening queries with projection
# The "outer spine"
# Error handling
# Recovery from day one
# Example: error type
```

```
# Diminishing returns
# Handling cycles
# Example: inlining (take 1)
# Example: inlining (take 2)
# Non-deterministic results
# One better approach
# Other cases involving cycles
# Tracking location information ("spans")
# What to put in your AST
# Incremental re-parsing
# Example: swift red and black trees
# Alternative: "zooming out" or "zooming in"
# Threading
# Salsa's threading model
# Cancellation
# Conclusion
```

How Salsa Works (2019.01)

[Mat19a]

Salsa In More Depth (2019.01)

[Mat19c]

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]
- 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
  - ☐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: Responsive compilers - Nicholas Matsakis - PLISS 2019*. June 2019. URL: https://www.youtube.com/watch?v=N6b44kMS6OM.

[Mat19c] 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.