Separate compilation and analysis in go and gopls

Alan Donovan
November 15, 2022
Go tools team

We've been talking a lot about "export data" recently, and a number of people asked me what that means.
So in this talk I hope to explain it.

This topic lies at the boundary of compilers and build systems
It's an unsexy part of the programming language field,
much neglected by existing literature,
yet important in practice.

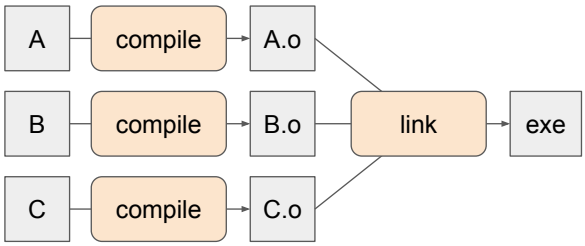
I hope to give you a little bit of historical background,
some technical insight into how separate compilation works (while keeping it accessible for non-compiler folks)
and to show you how we are using this technique to make gopls more efficient.

Separate compilation

Separate compilation is the compilation of a program in several parts, the results of which are then linked together to produce an executable.

Goals:

1. Save space
2. Save time



What do we mean by separate compilation?

[Quote definition]

The figure shows a program made of three parts,
all compiled separately into .o files (o for object code),
and then linked together to produce an executable.

Why is this important?

There are two main reasons: space, time

Separate compilation saves space

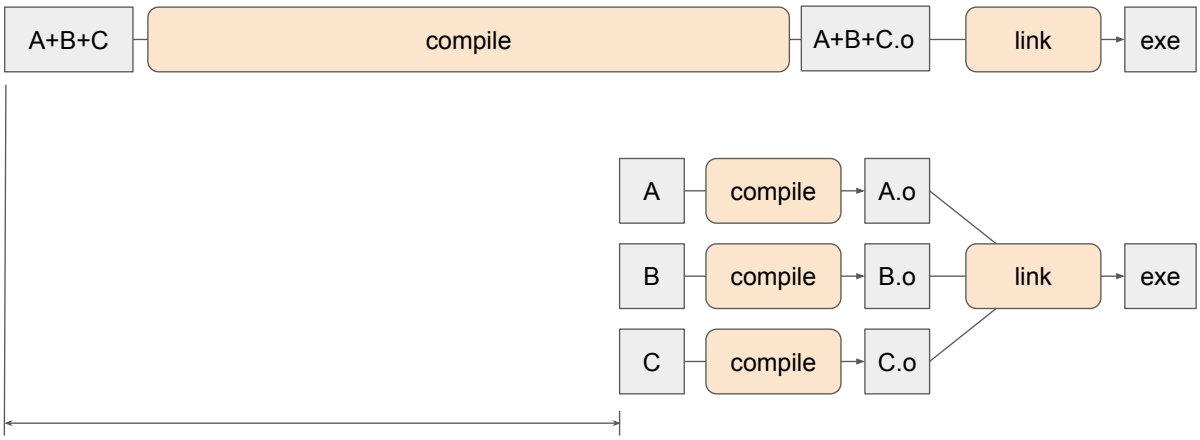
By splitting compilation into parts, the whole program doesn't need to be loaded into RAM at once.

These days the source code of even the largest programs easily fits in RAM,
but in the early days of computing RAM was extremely limited.

The picture shows an IBM 704, the first mainframe, which had 18KB of magnetic core memory.
It could run FORTRAN II, arguably the first separate compiler.

And even if you can hold the source text in RAM,
the process of compilation may require hundreds of times more than that for its working memory.

Separate compilation saves time



Also, by splitting up compilation, we save time, for two reasons.

The first is that we can avoid recompiling everything after a minor change.
Instead, we recompile only what has changed.
Incremental rebuild systems have been popular ever since Make in the mid 1970s.
(The author of Make, Stu Feldman, was later the site lead of Google NYC.)

The second (and more modern) reason is that splitting up compilation creates opportunities for parallelism.
My laptop has 16 threads; Forge has over a million. The summary of a large Forge build will sometimes report that it spent a CPU-day in a few minutes.

Ancient history of separate compilation with types

Fortran II (1958) was the first toolchain with separate compilation
…but you couldn't import type information.
NU (Norway) ALGOL (1965) had dynamic checks for type consistency.
A PASCAL for OS/360 (1978) had link-time checking for consistency.

Not until the 1970s did type consistency begin to be checked at compile time:
SIMULA 67 (1971)
Cambridge ALGOL 68C (1975)
UW (Madison) PASCAL (1977)
Xerox MESA (1978)

Modula-2 (1978) was the first language spec to address this requirement.

Ancient history of separate type checking

In a sense: separate compilation is as old as computing:
the original compilers and assemblers were humans (mostly women), using pen and paper,
and once a routine was compiled to binary it was reused in many other programs.
Recompiling was extremely expensive!

(John D. Levine's Linkers and Loaders book claims to have a source for Mauchly talking about separate compilation on the ENIAC in 1947. I'm skeptical and have sent him a query. He said perhaps this was anticipating the design of the UNIVAC, and that he would get back to me on return from a conference. Update: he couldn't find it and agreed it seemed unlikely.)

But what about automated compilation?
FORTRAN (1954) was the first real compiler. It didn't support separate compilation, but FORTRAN II in 1958 did.
So clearly it was the second most important feature. :)
But it was up to the programmer to ensure that the parts of the program were consistent in the types ascribed to common variables.
(The legacy of this approach survives in C/C++ to this day: it's still possible to get it wrong.)

What about type consistency?

- NU (norway) Algol 1950 for Univac had dynamic checking.
  At run-time the program would check that type descriptors from each part of the program agreed.
  More of a self-consistency assertion than a type error, I suspect.
- a SUNY Stony Brook PASCAL for OS/360 1978 had link-time checking, again I suspect in a similar vein.
  https://dl.acm.org/doi/pdf/10.5555/800099.803186

What about true compile-time type checking? Apparently not until the 1970s.

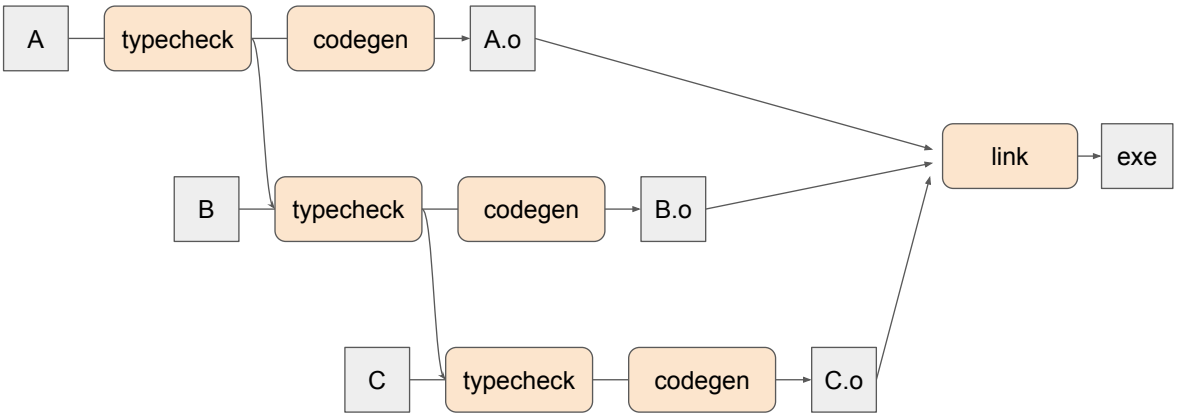
- Sweden SIMULA 67 - ref 1971 (“Part Compilation in High Level Languages”)
- UW-PASCAL has a static system - ref 1977
- ALGOL 68C at Cambridge - Steven Bourne ref 1975
- Xerox MESA - ref 1978
- Modula-2 - 1983 dissertation https://www.astrobe.com/Modula2/ETH7286.pdf

Ref: On implementing separate compilation in block-structured languages.
https://dl.acm.org/doi/pdf/10.1145/800229.806963

M2 (a Pascal descendent) seems to be the first language spec (not implementation) to take this requirement seriously.

Le Blanc + Fischer UW-Madison 1984 TR
https://www.cs.rochester.edu/u/scott/papers/1984_UW-TR541.pdf

Separate compilation with types



The ancient approaches to type checking took a number of shortcuts for types.

This diagram shows the idealized structure of the problem.

Consider a program of three parts ABC in which C imports B imports A.

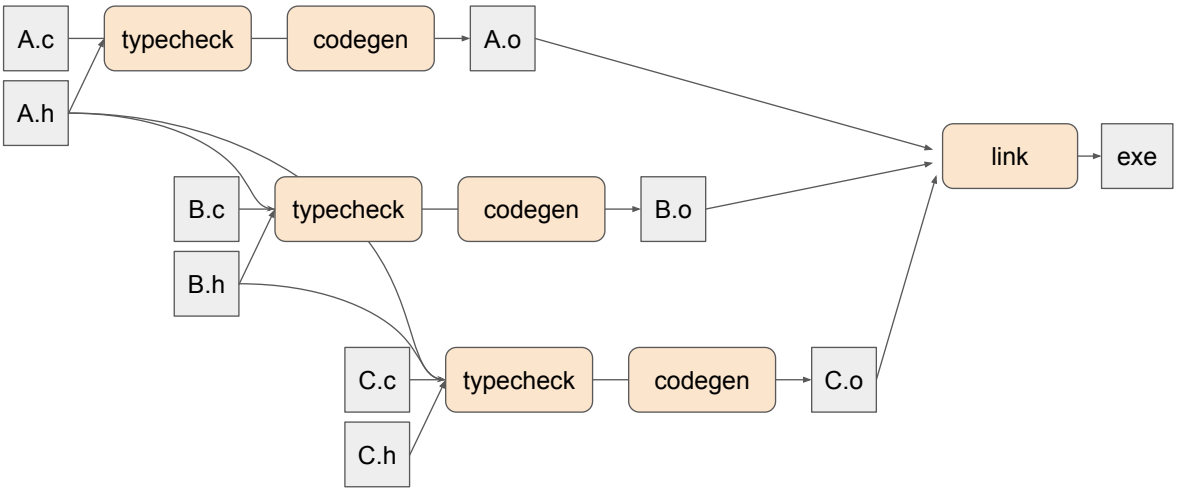
In this diagram we've split compilation into two parts.
The first, typecheck, figures out the types in the compilation unit.
The second, codegen, turns the functions into object code.

Observe that information must flow from the output of A's typechecking step to the input of B's typechecking step.
Similarly for B to C.

The challenge is: in what form should this type information flow?

Before Go, there were two main models which we might call "C" and "Java".
(To be clear, neither of these languages invented the respective approach.)

The C approach



This slide shows the C approach (which is also used by C++)

The source files are of two types:
.h header files define the types
.c files define the implementations (function bodies)

The compilation of B consumes both B files, but also A.h.
The compilation of C consumes both C files, but also B.h and A.h.
And so on.

This approach has a number of pros and cons.
The pros:

- All the header files all exist at once right from the start.
- This means all the compilations can proceed at once
- The critical path is only two steps: compile everything, link everything.
- This is great for build parallelism.

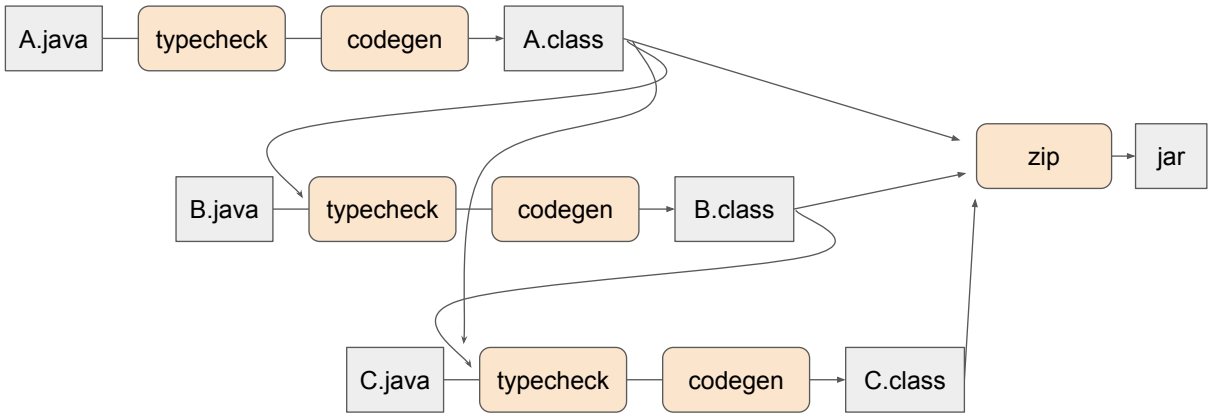
However, the cons:

- There are a lot of edges in the graph (O(n^2) for depth n).
- That means there are a lot of file-reads during a build…
- …and the same files are read, parsed, and typechecked several times. (3x for A.h here.)
- C compilers process a lot of text!
- They need very highly optimized scanners and parsers to chew through those headers

- and very efficient internal representations of the syntax tree.

(BTW C’s textual #include mechanism is rightly criticized for the ease with which it can be {mis,ab}used but that’s not the point of this slide.)

The Java approach



This slide shows what might be called the Java approach.
(Java didn't originate this idea; it dates from the 1970s and was used by some Pascal implementations, and the Modula language family.)

In this approach, the compiler turns Java source into JVM class files, which serve a dual purpose:
They contain the executable code used by the interpreter, and
they contain type information.
So, the compiler both produces class files, and consumes the ones produced by the previous step.

(In reality many .java files produce many .class files in each step, which are then bundled together, but that's a detail.)
(Also notice that in Java there's no linker. The class files are interpreted, so the linker is just zip.)

The pro of this approach is that the B compiler doesn't need to work very hard to get the type information for A because it is already validated and in an easy-to-read binary form.
This significantly reduces the amount of downstream computation.
But it has several downsides:

- As before, there are still a quadratic number of import edges.
  The A.class file may need to be parsed by B and by C.
- Each downstream compilation may need access to a large number of inputs:

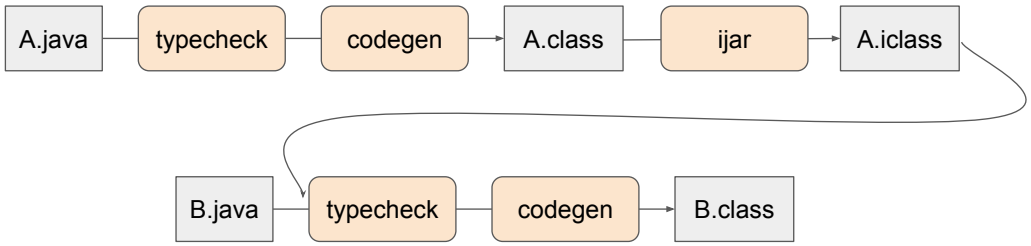
- all the dependencies. C needs files from B and A.
- The critical path is much longer. Notice that compilation of B cannot start until A is finished. Builds become highly sequential.
- Also, compiler inputs are large, because they contain Java source, plus .class files containing not just types but also (unnneed) function bodies too.

Before joining the Go team (both times!) I worked on Blaze, the build system used in google3.
Many of these system design issues around separate compilation were particularly acute there, for three reasons:

1. the vast scale of software in that repository: apps may have 10 million LoC and 10,000 build steps;
2. all of the compilation steps are executed remotely in a Borg service called Forge, so inputs and outputs must traverse the network; and
3. Forge makes cache hits only if the inputs are unchanged, so spurious changes are costly.

For these reasons, we deployed a number of tricks to improve performance: [NEXT SLIDE]

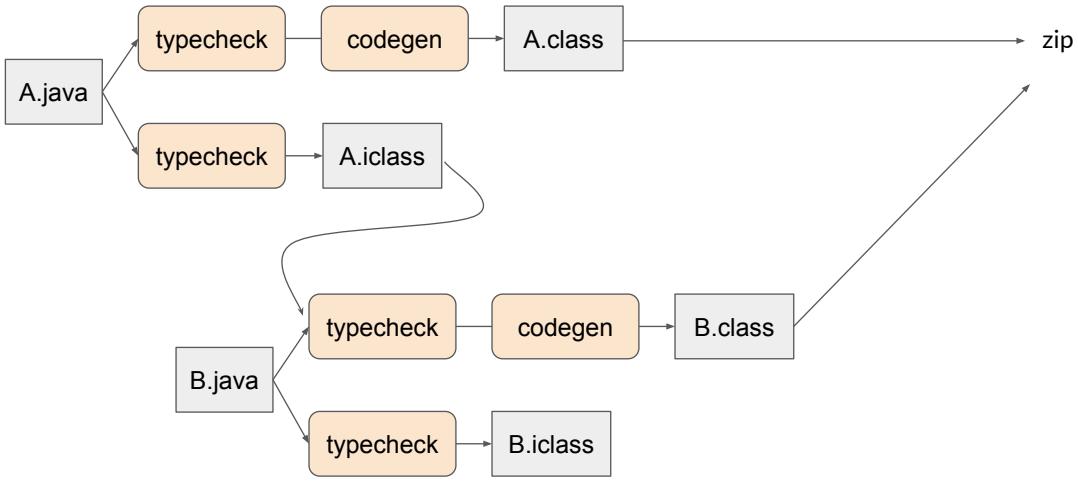
The Java approach (+ijar)



For these reasons, we deployed a number of tricks to improve performance:

For example, we inserted a filter (“ijar”) after the compiler.
It would make a copy of the .class files in which the function bodies were removed, and provide this copy to the compilation of B.
This reduced the amount of data feeding into B, and improved the cache hit rate, but as you can see it made the critical path longer.

The Java approach (+propeller)



A later approach (“propeller”) ran two tasks in parallel:

- the compiler (above) to produce the .class file to be executed, and
- the typechecker (below) to produce the .iclass type information for downstream compilation.

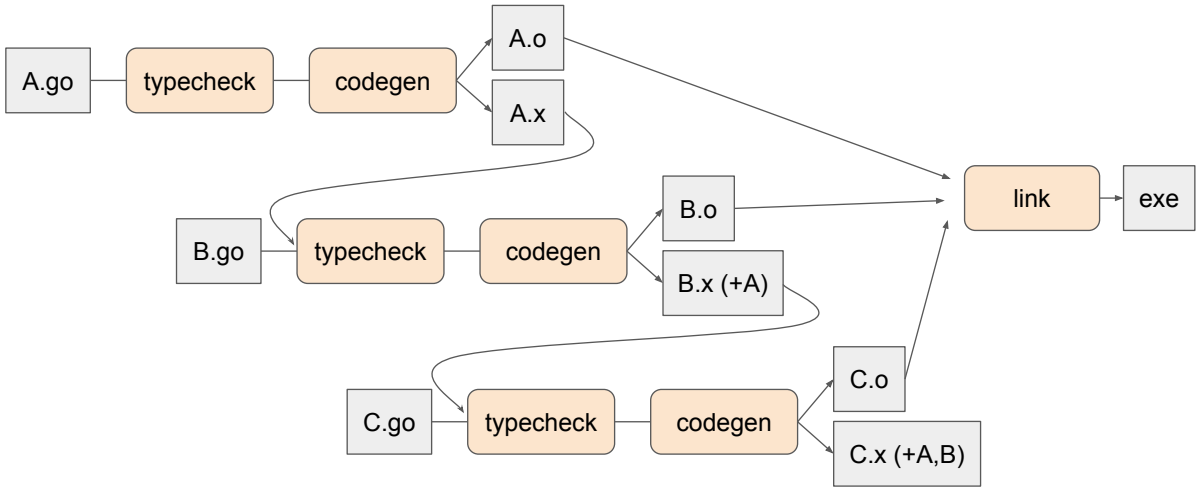
Because the typechecker is cheaper than the compiler, and more likely to make cache hits,
the critical path was much reduced, along with the “blast radius” of a small change to the source.
But it needs to do typechecking twice.

My point here is that there’s nothing really intrinsic to the language about which of those approaches you take.
Indeed, C++20’s new modules feature is causing that language to adopt many of the same ideas as used in Java.

(One thought: has anyone explored a compiler that writes the type information early, in parallel with code generation, and then notifies the build system so that it can get downstream units started even before waiting for the compiler to exit. This is close to the “idealized” scheme I started with. The benefit is that it would avoid the need to run the type checker twice.)

Enough about C and Java. What about Go?

The Go approach



To a first glance, Go takes approximately the same approach as Java (and most modern languages):
the types output from one compilation step are fed as input to another.

But the output of the compiler consists of two parts:

- the .o file containing object code, for the linker; and
- the .x file containing serialized type information, for the downstream compilation. We call these “export data” files.

Depending on the build system these parts may be:
- two sections of a single .a archive file (as in the Go command), or
- two separate files (as in Blaze).

The first case is analogous to vanilla Java; the second to Java+ijar.
The reason for the difference is that the ‘go’ command is all local whereas blaze execution is remote, so it makes sense to split the files.

But there’s an important difference:
The output of the B compiler includes not just the type information for B, but also for A (or at least, the parts of A that B mentions.)
Consequently, the compilation for C doesn’t need to include A.x, because B.x tells it all it needs to know.

This has two important consequences.
The good news: the set of edges is linear. If C imports only B, it needs to loads only B.x. That’s all.

Historically this has been one of the reasons that Go compilation is so fast relative to C++ and Java.

But the bad news: the size of the .x file grows quadratically:
a small high-level package may include a lot of type information about its indirect dependencies.

There’s really no getting away from the quadratic term.

We’ll come back to this in a moment.

Evolution of Go's export data

2008: "text": pseudo-Go type/func/var/const declarations.

2017: "binary": a compact binary encoding of an object graph; incl. function bodies.

2018: "indexed": a variant that allows lazily decoding parts as they are needed.

2021: "unified": a new representation of function bodies for generics and inlining.

So what's actually in an export data file?

The answer has changed over time.
Unlike Java, were the .class file is a fixed standard, we have a lot of latitude to change things,
so long as the compiler and the x/tools repo agree.

At the dawn of time (2008), ken coded the original text format,
which looked a bit like ordinary Go declarations: var/func/const/type.
It was parsed using a simplified fork of the Go parser that generated types.Type data instead of syntax trees.

In 2017, the compiler started supporting cross-package inlining, meaning that not only types
but the bodies of functions needed to be passed from one compilation unit to another.
gri implemented a binary form that compactly encoded general object graphs,
including not just function bodies but all the data structures produced during
type checking, thereby avoiding the need to repeat that step downstream.
(when inlining is disabled) the files were 29% smaller. Also faster to read and write.
See gri's talk from https://go.dev/talks/2017/exporting-go.pdf
CL https://go-review.googlesource.com/13937

In 2018, mdempsky added an indexed form that allowed the compiler to read the whole file
but to selectively decode parts of it as the need arises. Given that in many cases most of the file is unneeded, this was a big saving: builds became 20% faster.
CL https://go-review.googlesource.com/106796

Then last year (2021), mdempsky unified the representation of function bodies for inlining and generics.
https://go-review.googlesource.com/324573

Example using golang.org/x/tools/go/packages

```
package main

import ("fmt"; "log"; "golang.org/x/tools/go/packages")

func main() {
     cfg := &packages.Config{Mode: packages.NeedTypes}
     pkgs, err := packages.Load(cfg, "net/http") // 90ms
     if err != nil { log.Fatal(err) }
     fmt.Println(pkgs[0].Types.Scope().Lookup("Request"))
}
```

```
type net/http.Request struct{Method string; URL *net/url.URL; Proto string; ProtoMajor int; ProtoMinor
int; Header net/http.Header; Body io.ReadCloser; GetBody func() (io.ReadCloser, error); ContentLength
int64; TransferEncoding []string; Close bool; Host string; Form net/url.Values; PostForm net/url.Values;
MultipartForm *mime/multipart.Form; Trailer net/http.Header; RemoteAddr string; RequestURI string; TLS
*crypto/tls.ConnectionState; Cancel <-chan struct{}; Response *net/http.Response; ctx context.Context}
```

The golang.org/x/tools repo has a copy of the encoding and decoding routines for export data.
This way the type checker package (go/types) can read and produce files in the the same format as the compiler.
(but only the type information: function bodies for inlining+generics are in a form that is private to the Go compiler.)

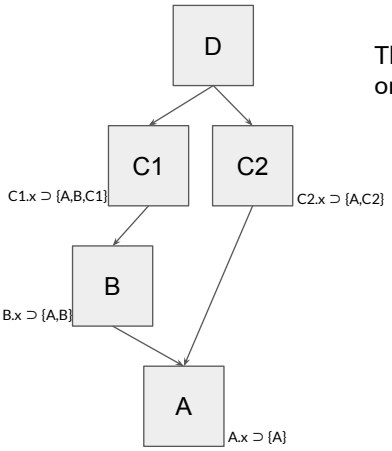
The go/packages package takes advantage of the compiler's export data, making it very efficient.

The example program prints the type of http.Request, from the net/http package, which is substantial.
It runs ‘go list -export’ to compile the export data for net/http.
If it’s already built, the whole operation takes less than 100ms.

It loads only a single .x file for net/http, but notice that it knows about types such as net/url.URL, io.ReadCloser, mime/multipart.Form, and crypto/tls.ConnectionState.

(It would take 4x as long to do the compute the types from source even for this low-level package.)

“Deep” export data



The type checker for D needs to load only C1.x and C2.x, not A.x or B.x.

`import` graph

Export data is "deep"

We mentioned a few slides back that Go export data is usually "deep".
Consider this package import graph.
If the export data for C1 mentions a declaration in B,
and that declaration in B mentions a type defined in A, then
the export data for C1 includes the A type.
The type graph is transitively closed.

The advantage of this approach is that when compiling D, which depends on C1,
we can load the export data for the direct imports C1 and C2
without needing A and B. C1 and C2 together tell us all we need to know about A and B.
This means a build system need only provide one "summary" per direct import.

The export data files C1.x and C2.x may contain overlapping information about indirect dependencies in A.
It is the build tool's job to ensure that they are consistent.

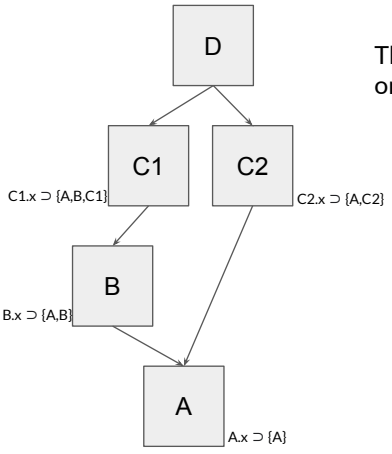
This is especially important in Blaze, because it must send all the .x
files over the network to the forge cluster, and we don't want that
set to grow quadratically.

HOWEVER

The transitively closed information grows quickly.

NEXT SLIDE

“Deep” export data



import graph

The type checker for D needs to load
only C1.x and C2.x, not A.x or B.x.

```
type PodsByCreationTime []*v1.Pod

func (s PodsByCreationTime) Len() int           { … }
func (s PodsByCreationTime) Swap(i, j int)      { … }
func (s PodsByCreationTime) Less(i, j int) bool { … }
```

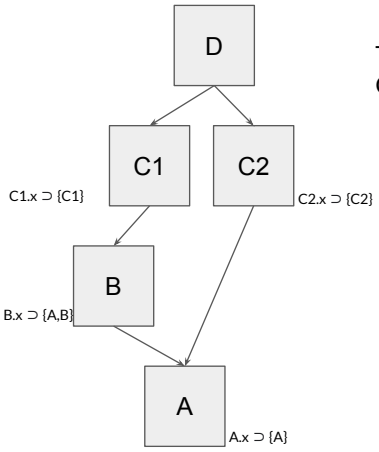
The transitively closed information grows quickly.

In k8s, our canonical big Go project, we found a package that defines
only a sort ordering for a particular type -- a handful of lines of code --
but that type was the main type in k8s and it depends on information from hundreds of packages.
A few lines of source text result in more than 1.1MB of export data.

Now consider that every package that mentions this type (C1, C2, D, etc) all have to contain a copy of that information.

The total amount of export data for this build is about 550MB,
even though a single complete copy of each package would be only 42MB.

“Shallow” export data



The type checker for D needs to load
C1.x and C2.x, and perhaps A.x and B.x too.

The compiler team (esp. mdempsky) is currently evaluating the approach of using “shallow” export data.
(This is effectively the same as the the Java+ijar approach mentioned earlier.)

The object graph encoder would stop each time it encounters a name from another package.
The decoder must read many files --- possibly the entire transitive closure.
But in practice only a small subset.

Usually it reads the direct deps C1, C2, etc in parallel ahead of time, then the indirect deps (A, B) on demand.
There is a risk that this becomes very sequential if indirect deps are numerous.
(There are tricks we could use to improve that, such as remembering and prefetching the set that was actually needed “last time”.)

Export data and gopls

TODO: stunning visual aid goes here

What this means for gopls

So far I've been talking about compilers and build systems.
I've been talking about the compiler a lot.
What does all this have to do with gopls?

gopls is not so different from a compiler.
Though it doesn't need object code, it does want type errors, type information,
and static analysis information for all the packages in the workspace.
(e.g. hover, cross-refs, analysis).

Until now, we've been loading everything into gopls memory---just like FORTRAN 1.
This is not scaling well, the k8s team tells us.
They find gopls slow, and hungry for memory and CPU.

So, we're trying to make gopls more like an incremental build system (go or blaze).
And that means saving the results of computation on each package in the file system
and discarding the data from memory.
Naturally that means using export data for type information.

Rob described this approach back in the late Spring.
The first CL of this new approach is out this week:
https://go-review.git.corp.google.com/c/tools/+/443099

Our first draft used deep export data, but spent a lot of time encoding and decoding the bulky files.
(Without the expensive object code generation step of a compiler, this became the dominant cost.)
So we switched to shallow export data.

Pruning works.

It takes about 8s to analyze all of k8s from cold. (A little slower than the current 5s)
But then all of that state is saved in the file system, not in memory.
The next request for analysis takes only 250ms---even after killing the process.

Summary of the benefits:

like separate compilation:
- space efficiency
- CPU efficiency
- incremental builds become the norm

But this is the beginning of a process. Analysis is relatively separable from the rest of gopls.
gopls is monolithic in assumptions about package/object identity.
We need to get away from that, and towards a more build-system like model.

[linker analogy: symbol index?]
need to handle errors in types
optimize for latency vs throughput - completion ??

“Eager” vs. “lazy” import

TODO: brilliant animation of type-checker at work

Another trick that we could use is “lazy” importing.

Normally, importing is “eager”.
That is, when the type checker imports an export data file, it decodes the whole thing and creates data structures for it.

However, often you only need a single symbol from a large package,
so it would be cheaper to avoid decoding symbols until their are actually needed:
lazily.

The go compiler already does this. TODO: quote the benefit

We did an experiment to use lazy importing in gopls.
But we rejected it on grounds of complexity:

- with a lazy approach you never know whan you are “done”
  Unlike a compiler, which generates assembly and exits, gopls’ analysis can make arbitrary requests at some later time.
- Potential risks wrt concurrency.
- Hard to integrate into dependency analysis.
- Also requires unsafe hacks to get at experimental features of the type checker.

Separate analysis

TODO

go vet -- tool for static anlaysis
unitchecker -- core of incremental build system approach to modular analysis
facts -- serializable data associated with declarations. An extension to the type system.
objectpath -- serializable stable names for all objects in the export data (incl fields, methods, etc)

Analysis facts are like export data
though we use "deep facts" because quantities of data are so small
that the shallow/deep arguments don't really apply.

Questions?