Bazel

{fast, correct} – choose two

Klaus Aehlig

February 4–5, 2017

Bazel

What is Bazel?

Bazel

What is Bazel?

- Bazel is a build tool

Bazel

What is Bazel?

- Bazel is a build tool
  like `make`, *etc, it organises compiling/creating artifacts (libraries, executables, . . .) from sources*

Bazel

What is Bazel?

- Bazel is a build tool

Bazel

What is Bazel?

- Bazel is a build tool
- core part of a tool
  used internally at Google since over a decade

Bazel

What is Bazel?

- Bazel is a build tool
- core part of a tool
  used internally at Google since over a decade

⇝ optimized for Google's internal use case

Bazel

What is Bazel?

- Bazel is a build tool
- core part of a tool
  used internally at Google since over a decade

⇝ optimized for Google's internal use case

  - large code base in a single source tree ($\approx 10^7$ files)

Bazel

What is Bazel?

- Bazel is a build tool
- core part of a tool
  used internally at Google since over a decade

⇝ optimized for Google's internal use case

  - large code base in a single source tree ($\approx 10^7$ files)
  - majority of engineers ($\approx 10^{4.5}$) actively working on that single code base.

Bazel

What is Bazel?

- Bazel is a build tool
- core part of a tool
  used internally at Google since over a decade

⇝ optimized for Google's internal use case (large mono-repo)

Bazel

What is Bazel?

- Bazel is a build tool
- core part of a tool
  used internally at Google since over a decade

⇝ optimized for Google's internal use case (large mono-repo)

- open-sourced only in 2015
  (in fact, still going on)

What is Bazel Good for?

What is Bazel? And what is special about it?

What is Bazel Good for?

What is Bazel? And what is special about it?

- optimized for large mono-repos, therefore...

What is Bazel Good for?

What is Bazel? And what is special about it?

- optimized for large mono-repos, therefore...
- aggressive caching

What is Bazel Good for?

What is Bazel? And what is special about it?

- optimized for large mono-repos, therefore...
- aggressive caching *without* losing correctness
  (i.e., all artifacts as if freshly built from source)

What is Bazel Good for?

What is Bazel? And what is special about it?

- optimized for large mono-repos, therefore...
- aggressive caching *without* losing correctness

What is Bazel Good for?

What is Bazel? And what is special about it?

- optimized for large mono-repos, therefore...
- aggressive caching *without* losing correctness
- declarative style of `BUILD` files

What is Bazel Good for?

What is Bazel? And what is special about it?

- optimized for large mono-repos, therefore...
- aggressive caching *without* losing correctness
- declarative style of `BUILD` files
  - separation of concerns
    writing code *vs* choosing correct (cross) compiling strategy

What is Bazel Good for?

What is Bazel? And what is special about it?

- optimized for large mono-repos, therefore...
- aggressive caching *without* losing correctness
- declarative style of `BUILD` files
  - separation of concerns
    writing code *vs* choosing correct (cross) compiling strategy
  - central maintenance point for build rules

What is Bazel Good for?

What is Bazel? And what is special about it?

- optimized for large mono-repos, therefore...
- aggressive caching *without* losing correctness
- declarative style of `BUILD` files

Overview of a `bazel build`

What is Bazel? And how does it build?

Overview of a `bazel build`

What is Bazel? And how does it build?

- load the `BUILD` files *(all that are needed)*

Overview of a `bazel build`

What is Bazel? And how does it build?

- load the `BUILD` files *(all that are needed)*
- analyze dependencies between targets

Overview of a `bazel build`

What is Bazel? And how does it build?

- load the `BUILD` files *(all that are needed)*
- analyze dependencies between targets
- from rules generate action graph

Overview of a `bazel build`

What is Bazel? And how does it build?

- load the `BUILD` files *(all that are needed)*
- analyze dependencies between targets
- from rules generate action graph
- execute actions *(unless already cached)*

Overview of a `bazel build`

What is Bazel? And how does it build?

- load the `BUILD` files *(all that are needed)*
- analyze dependencies between targets
- from rules generate action graph
- execute actions *(unless already cached)*

on subsequent builds, update the graphs
(client-server architecture to keep graph in memory)

An Example

Let's look at a `helloworld` example.

An Example

- main program `helloworld.c`

An Example

- main program `helloworld.c`

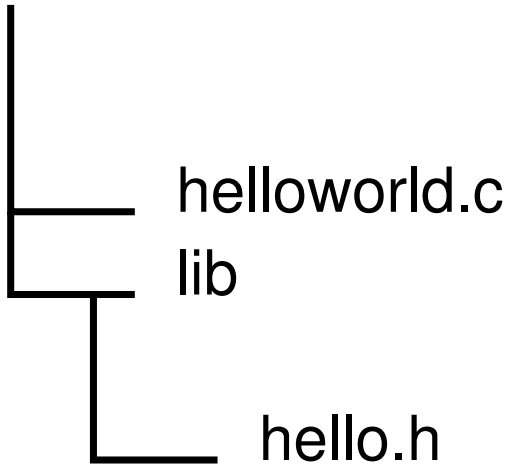

```
#include "lib/hello.h"

int main(int argc, char **argv) {
  greet("world");
  return 0;
}
```

An Example

- main program `helloworld.c`, depending on a library

An Example

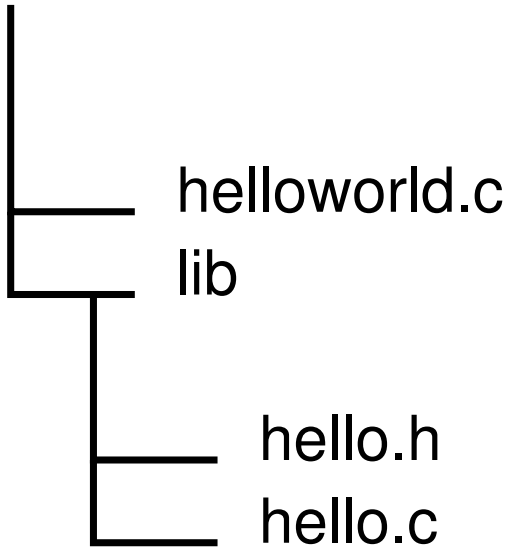
- main program `helloworld.c`, depending on a library
- a library with headers (`lib/hello.h`)




```
#ifndef HELLO_H
#define HELLO_H

void greet(char *);

#endif
```

An Example

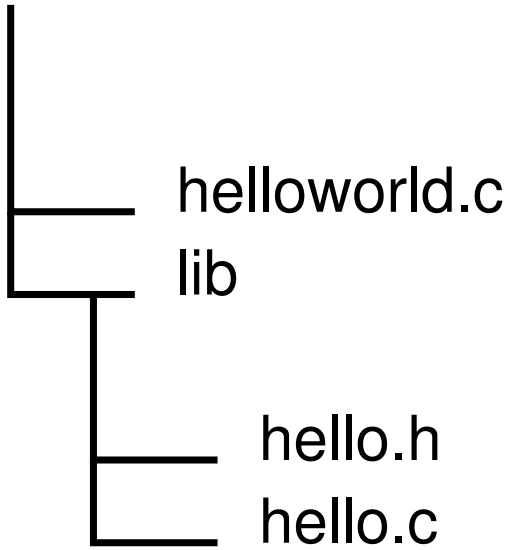
- main program `helloworld.c`, depending on a library
- a library with headers (`lib/hello.h`)
  ... and implementation (`lib/hello.c`)




```
#include "hello.h"
#include <stdio.h>

void greet(char *it) {
  printf("Hello %s!", it);
}
```

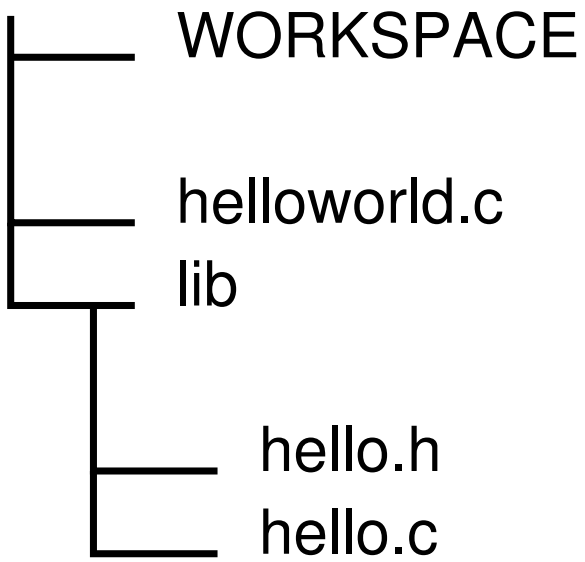
An Example

- main program `helloworld.c`, depending on a library
- a library with headers (`lib/hello.h`)
  ... and implementation (`lib/hello.c`)

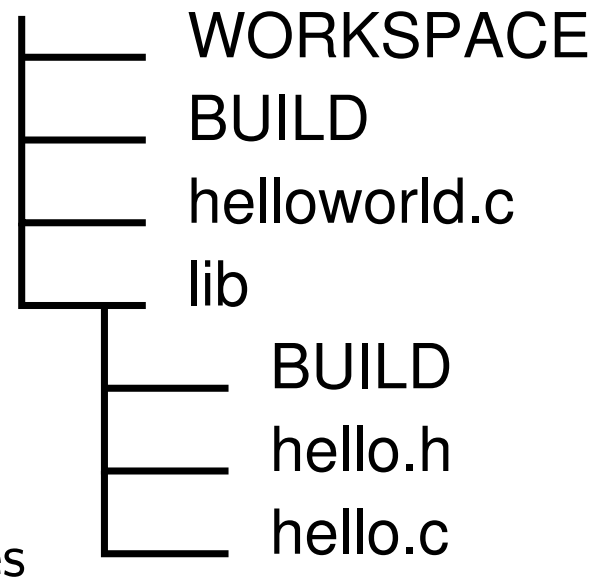


An Example

- main program `helloworld.c`, depending on a library
- a library with headers (`lib/hello.h`) ... and implementation (`lib/hello.c`)
- then we can have an empty `WORKSPACE` file



An Example

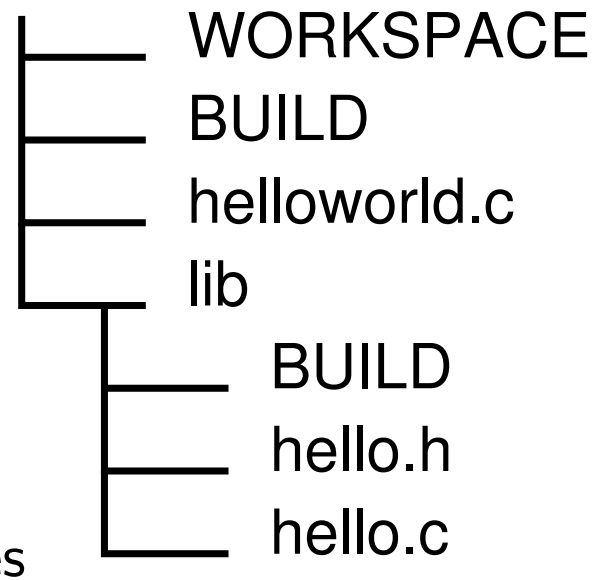


- main program `helloworld.c`, depending on a library
- a library with headers (`lib/hello.h`) ... and implementation (`lib/hello.c`)
- then we can have an empty `WORKSPACE` file ... and the following declarative `BUILD` files

```
cc_binary(
  name="helloworld",
  srcs=["helloworld.c"],
  deps=["//lib:hello"],
)
```

```
cc_library(
  name="hello",
  srcs=glob(["*.c"]),
  hdrs=glob(["*.h"]),
)
```

An Example



- main program `helloworld.c`, depending on a library
- a library with headers (`lib/hello.h`) ... and implementation (`lib/hello.c`)
- then we can have an empty `WORKSPACE` file ... and the following declarative `BUILD` files

```
cc_binary(
  name="helloworld",
  srcs=["helloworld.c"],
  deps=["//lib:hello"],
)
```

```
cc_library(
  name="hello",
  srcs=glob(["*.c"]),
  hdrs=glob(["*.h"]),
)
```

Note: `CC`, link options, host/target architecture, etc, taken care of elsewhere.

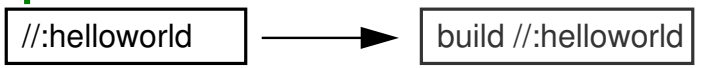
Example cont'd: Dependencies

build //:helloworld

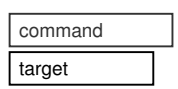
Now let's see what happens if we want to build `:helloworld`...

command

Example cont'd: Dependencies

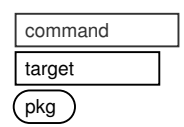


We look at the target `:helloworld`

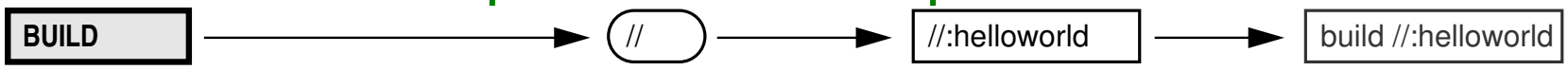


Example cont'd: Dependencies

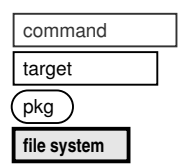
We look at the target `:helloworld`, in package `//`



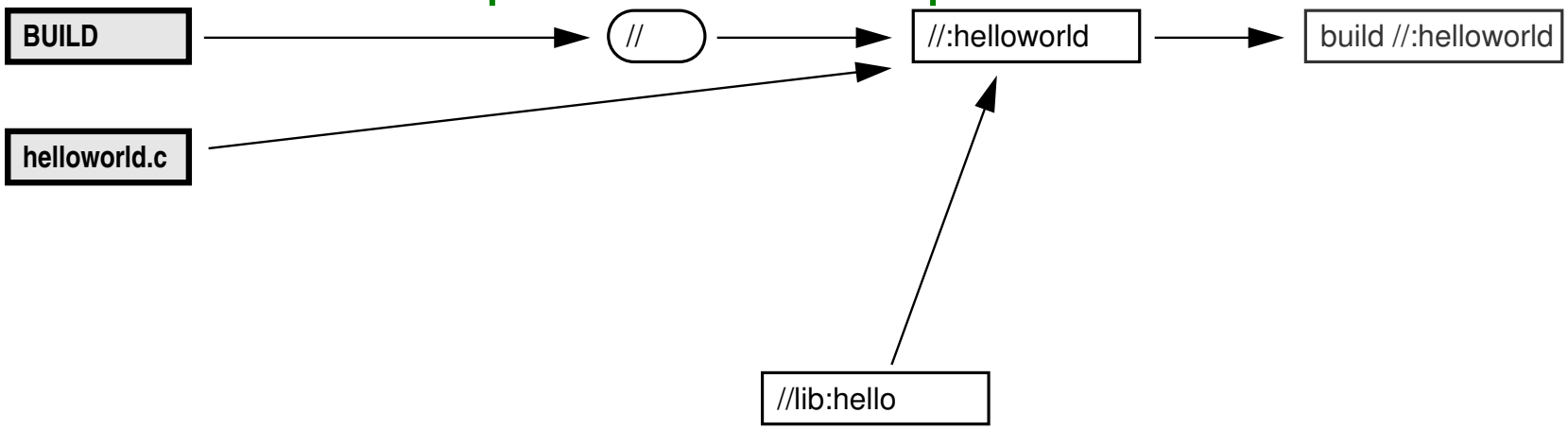
Example cont'd: Dependencies



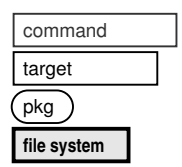
We look at the target `:helloworld`, in package `//`, in file `BUILD`



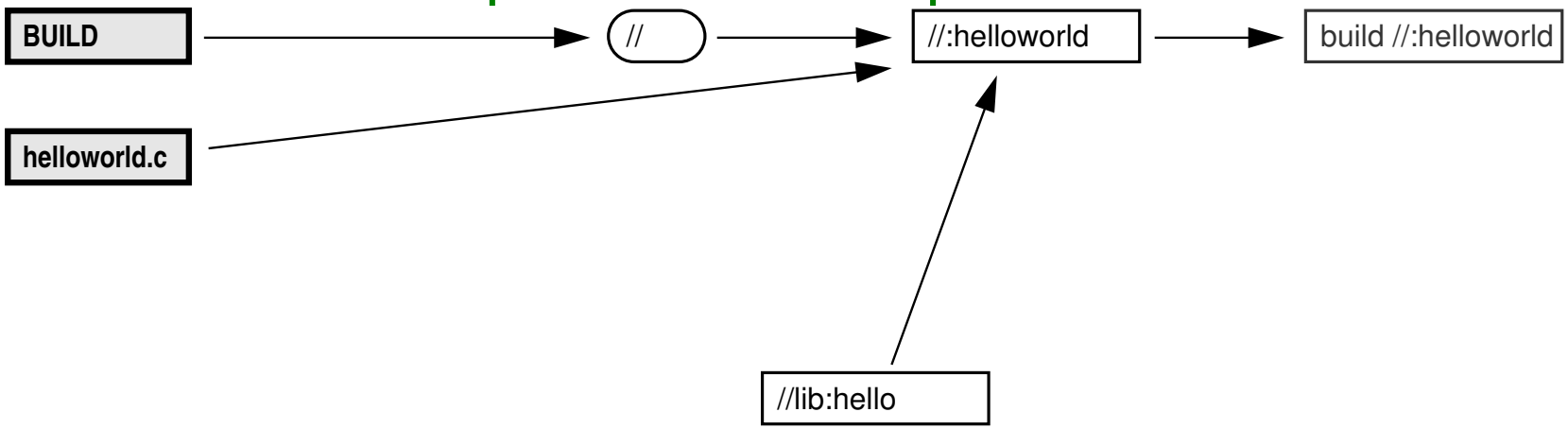
Example cont'd: Dependencies



Two declared dependencies

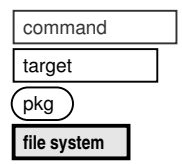


Example cont'd: Dependencies

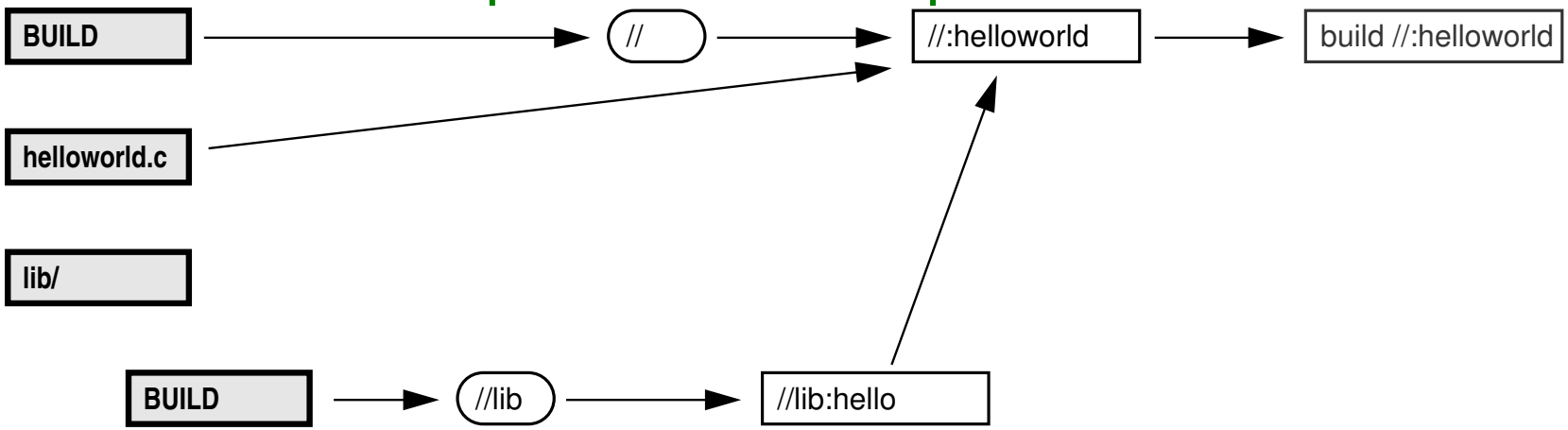


Two declared dependencies

. . . and implicit dependency on the `C` *tool chain (not drawn in this diagram)*



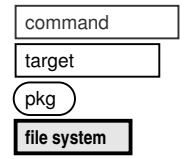
Example cont'd: Dependencies



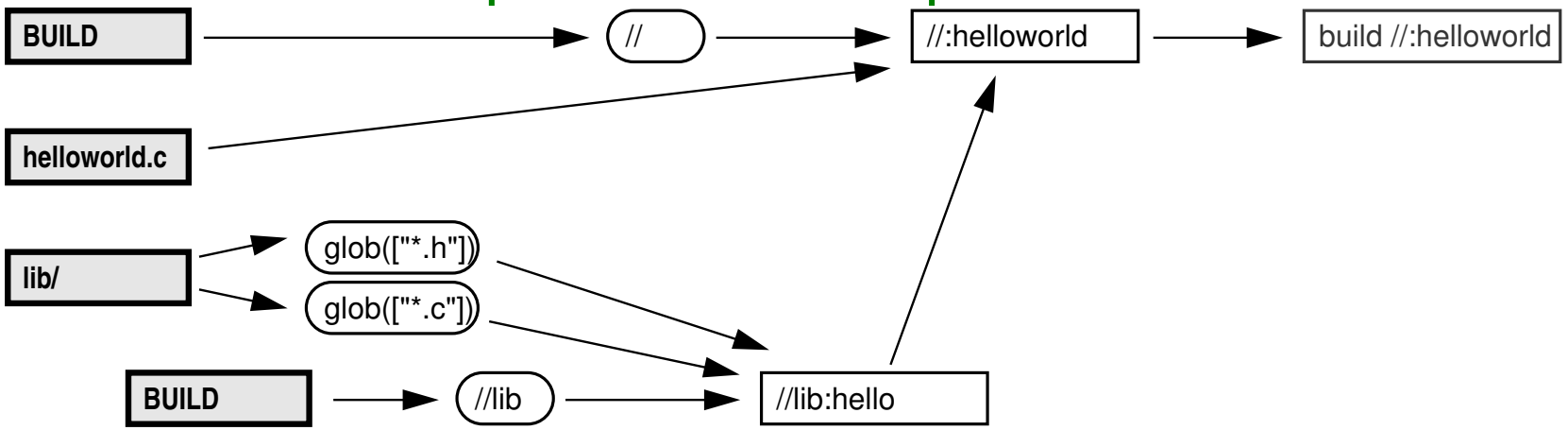
Two declared dependencies, one in a different package

Note: We construct dependency graph over package boundaries!

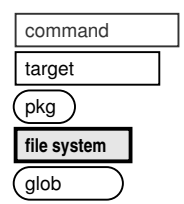
(no recursive calling)



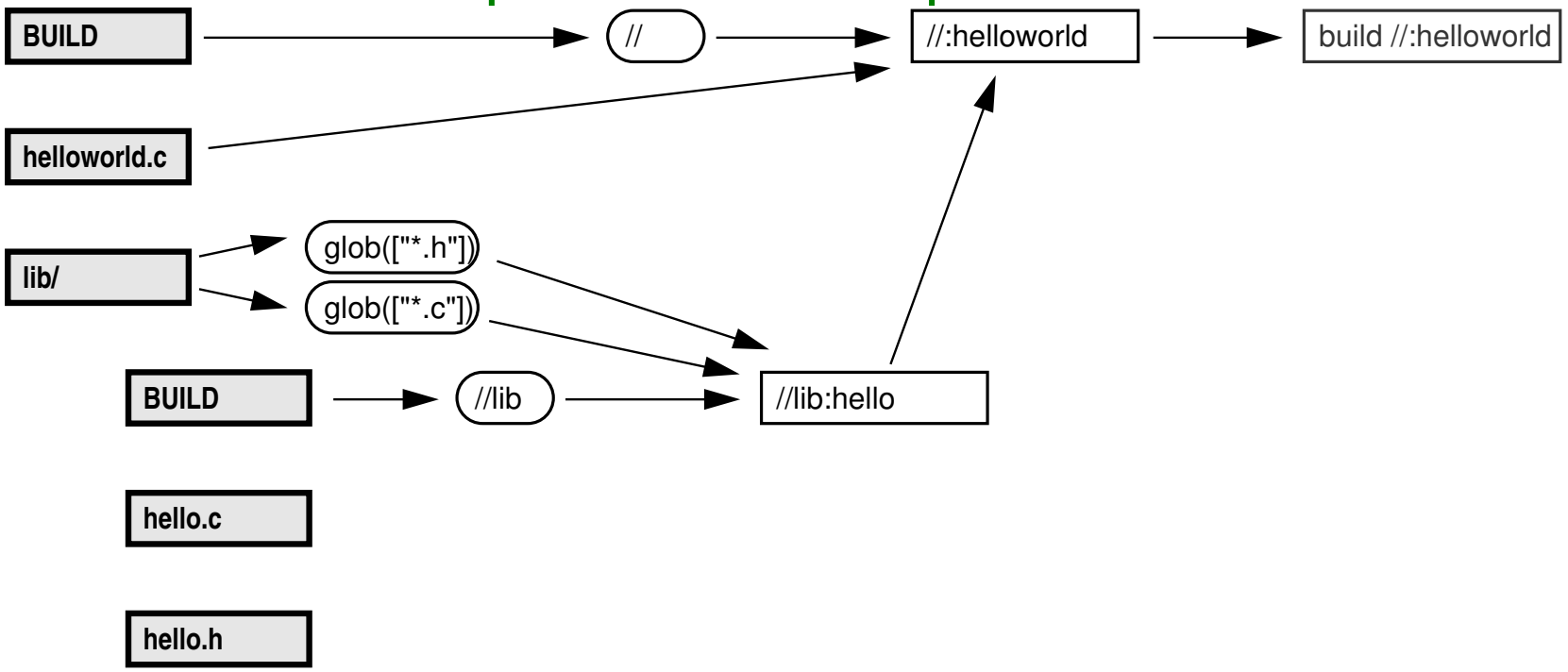
Example cont'd: Dependencies



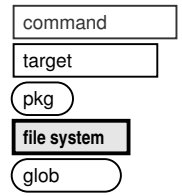
We discover `glob` expressions



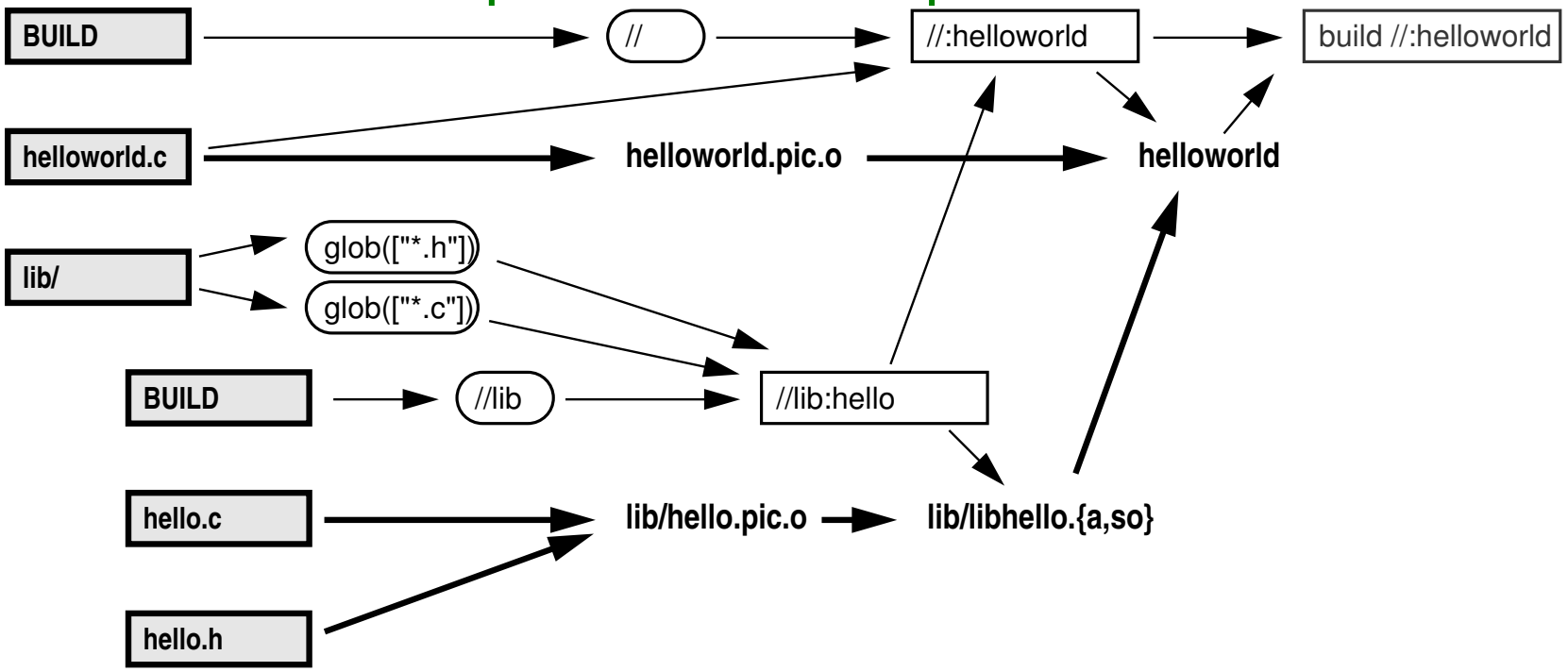
Example cont'd: Dependencies



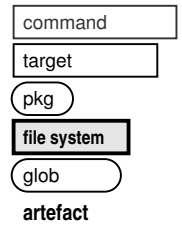
We discover `glob` expressions, and read the directory.



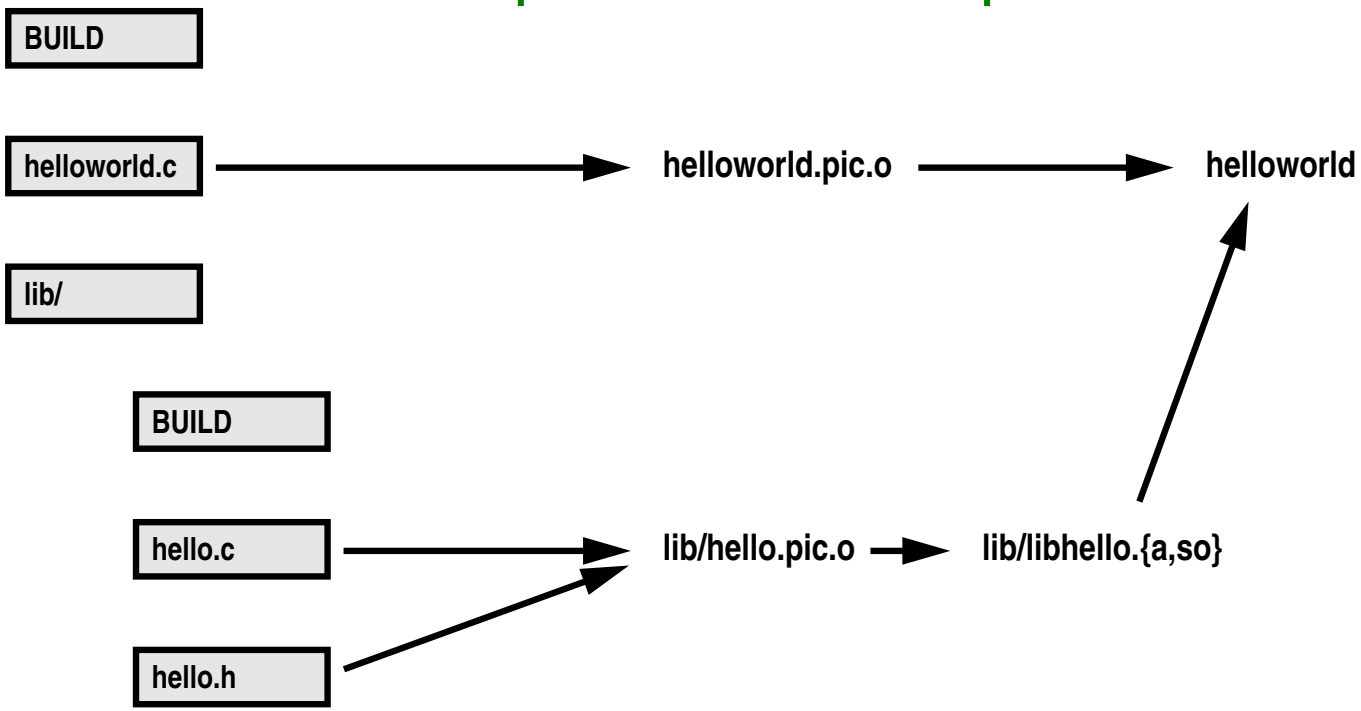
Example cont'd: Dependencies



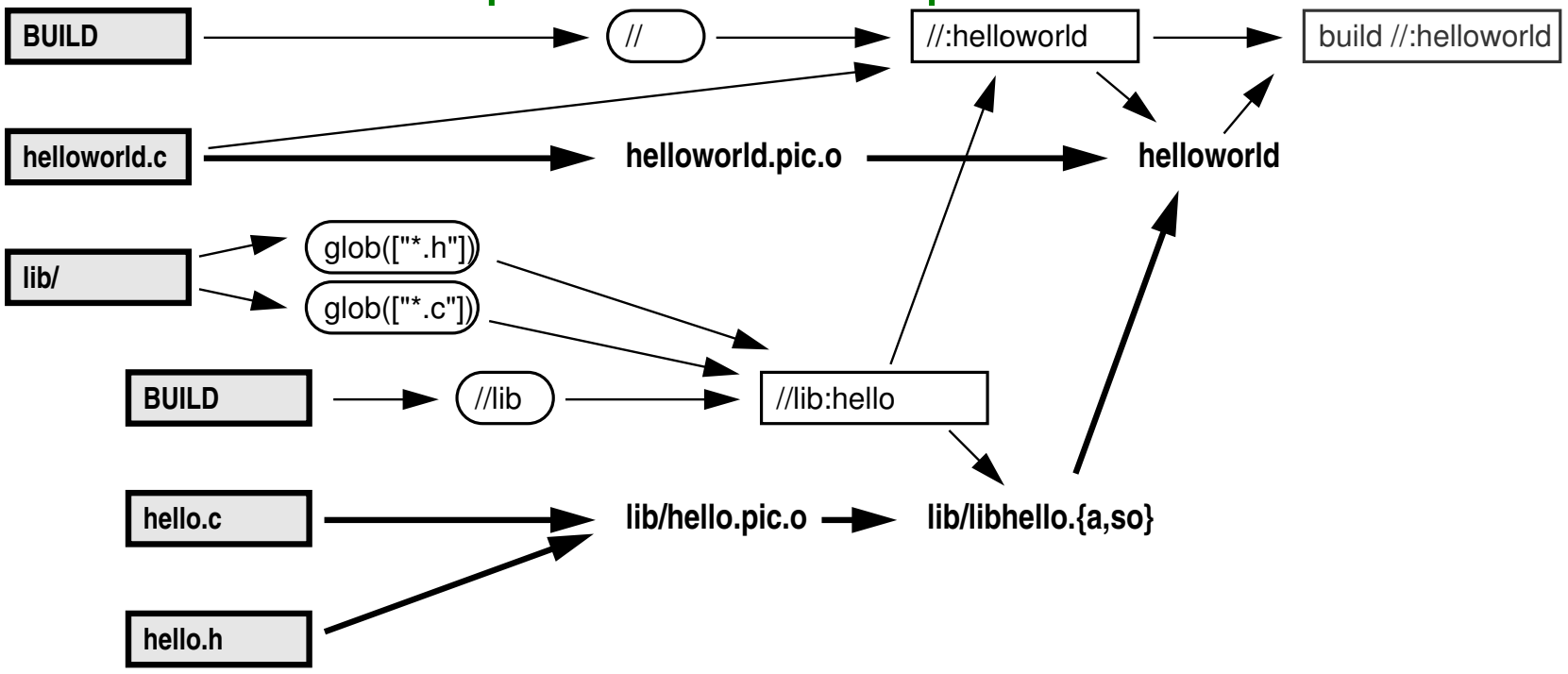
The rules tell us, which artifacts to build.

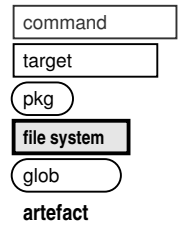


Example cont'd: Dependencies

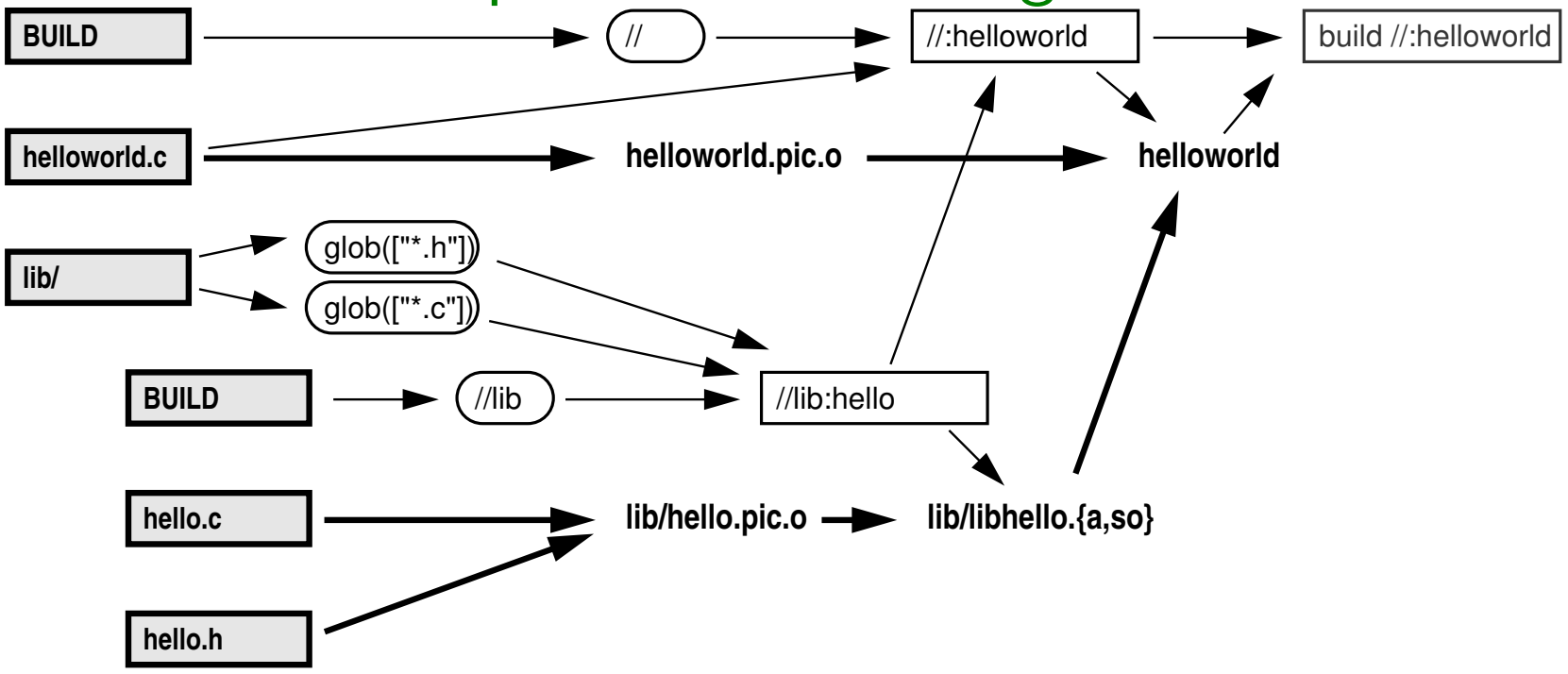


Example cont'd: Dependencies

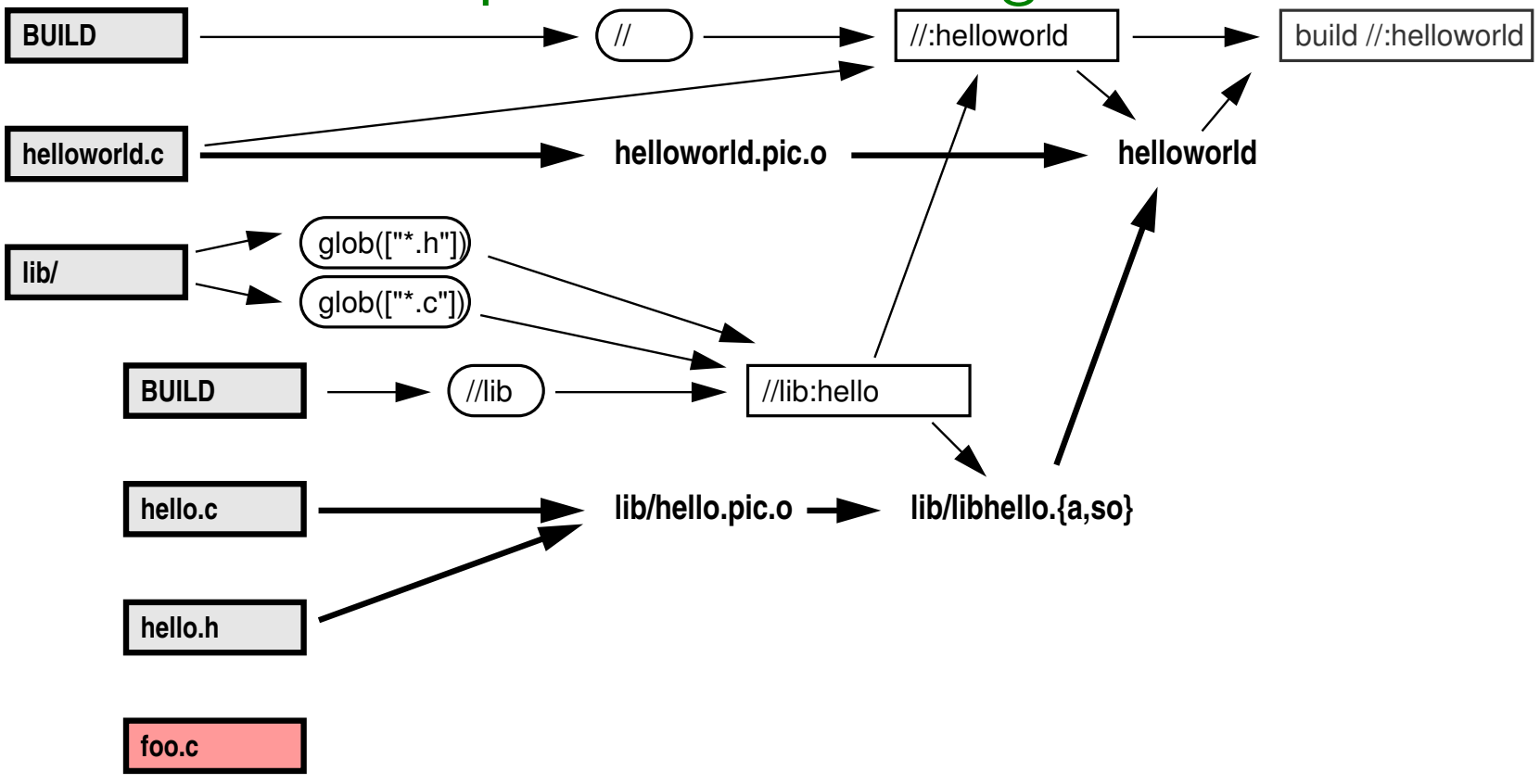




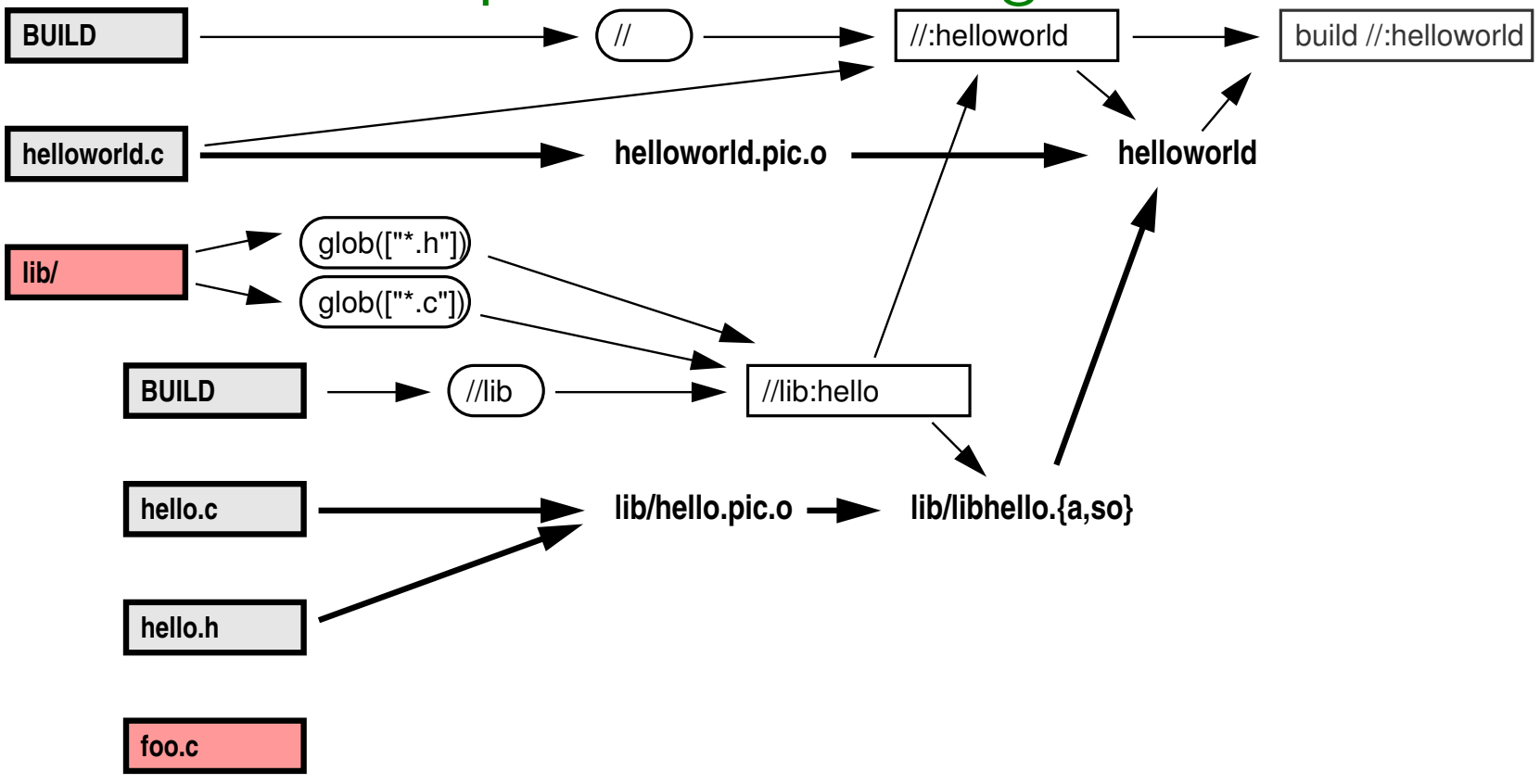
Example cont'd: Adding a File



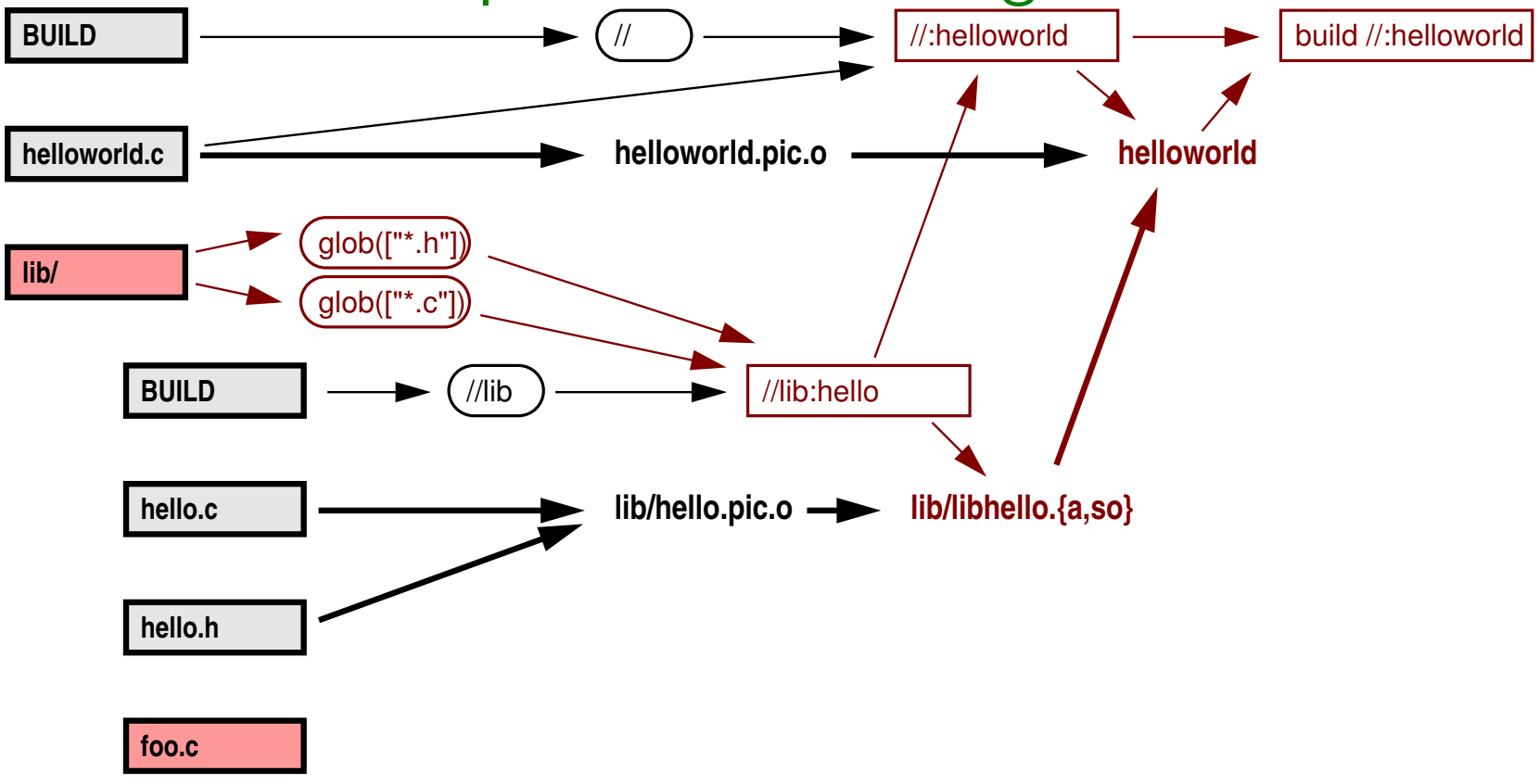
Example cont'd: Adding a File

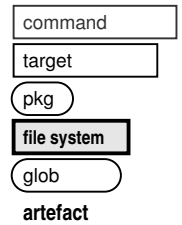


Example cont'd: Adding a File

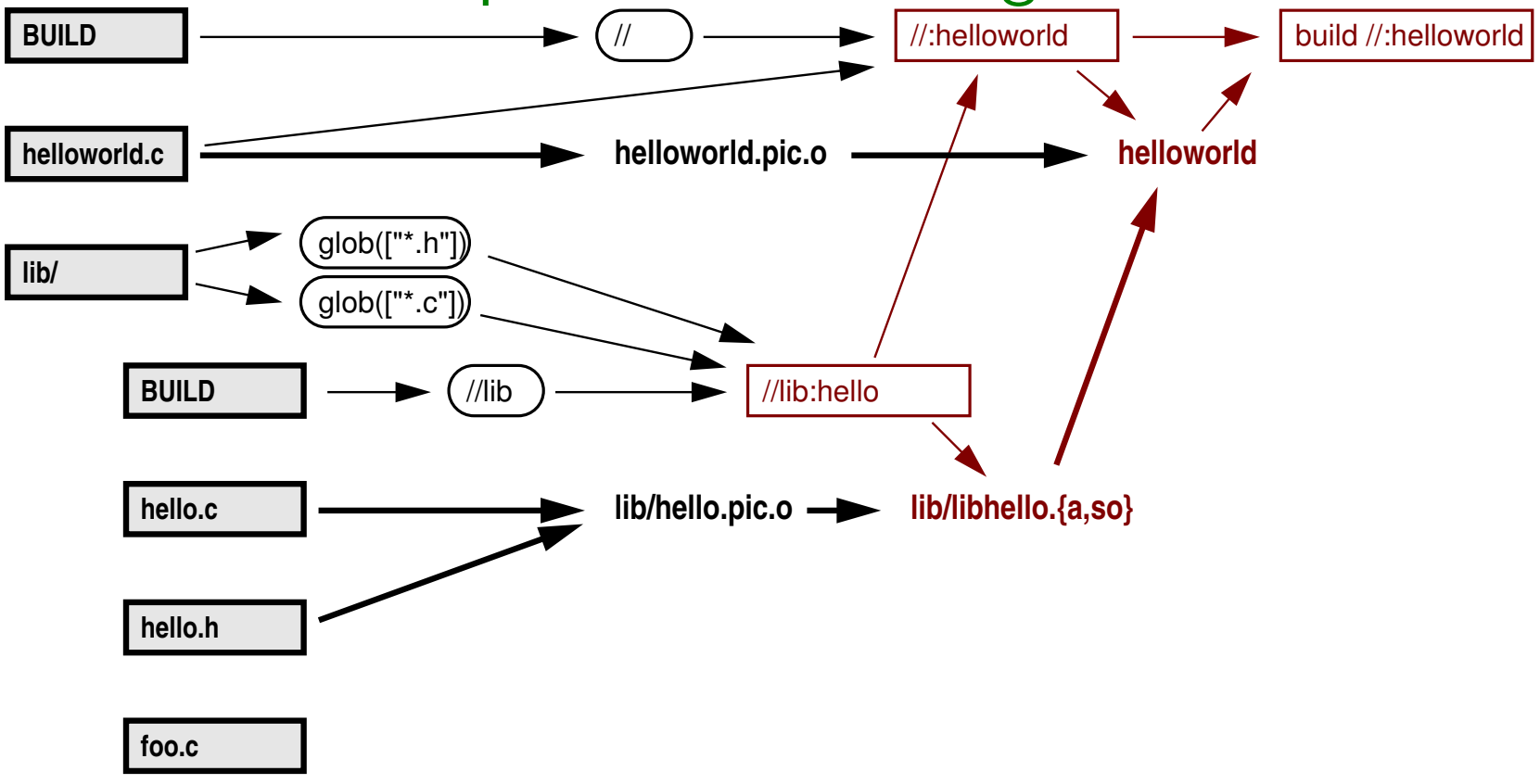


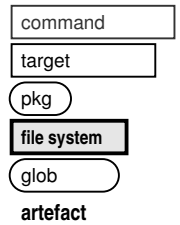
Example cont'd: Adding a File



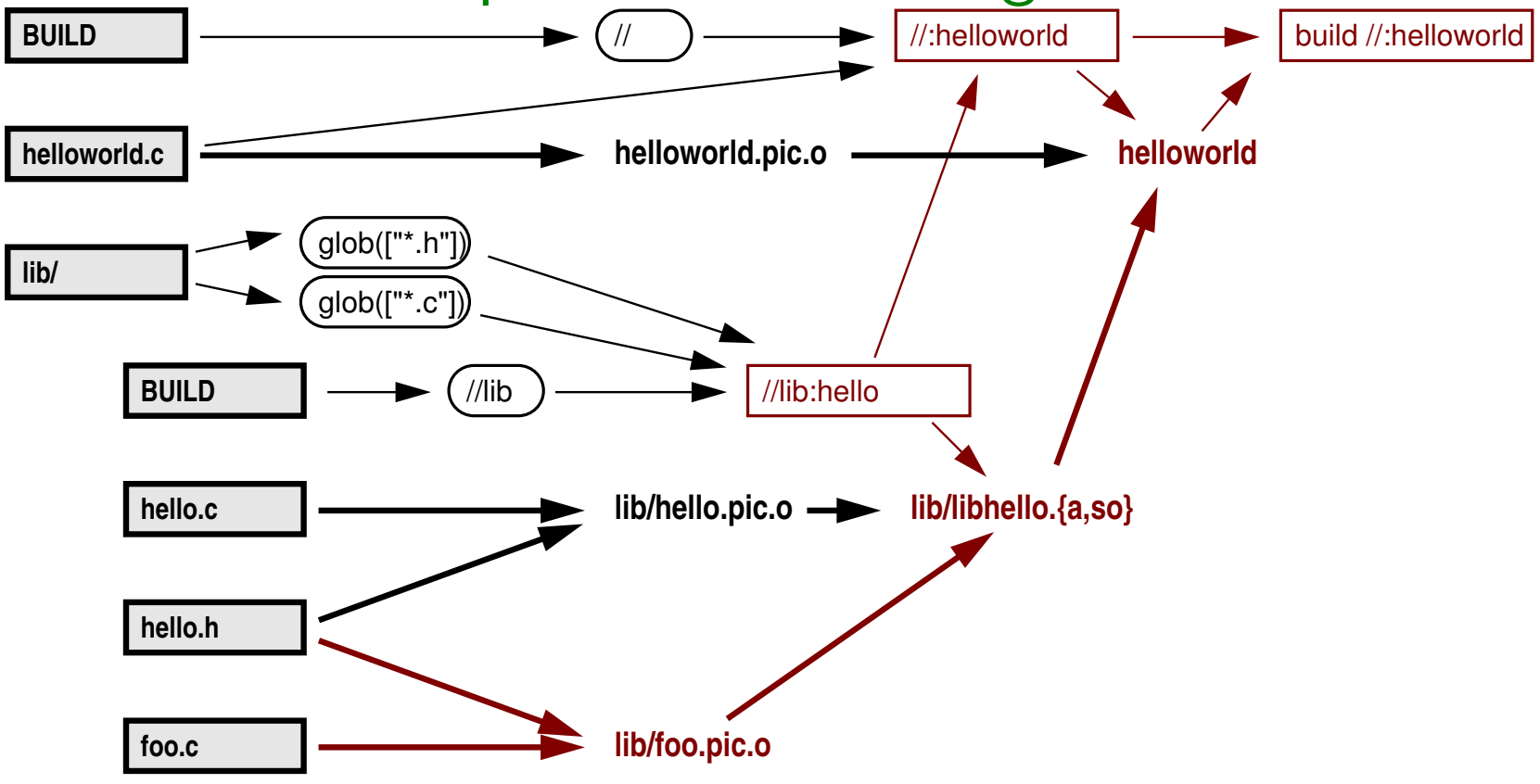


Example cont'd: Adding a File

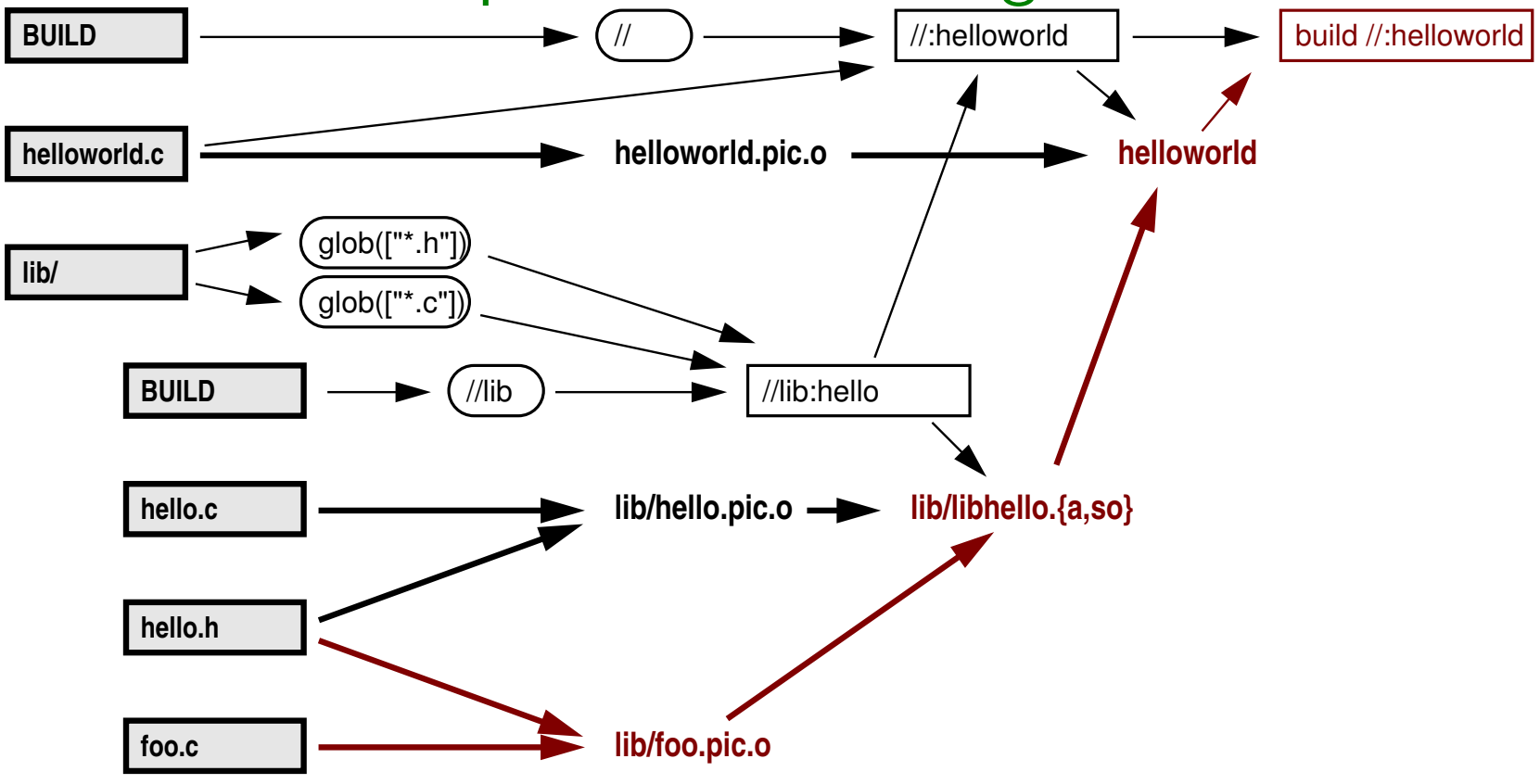




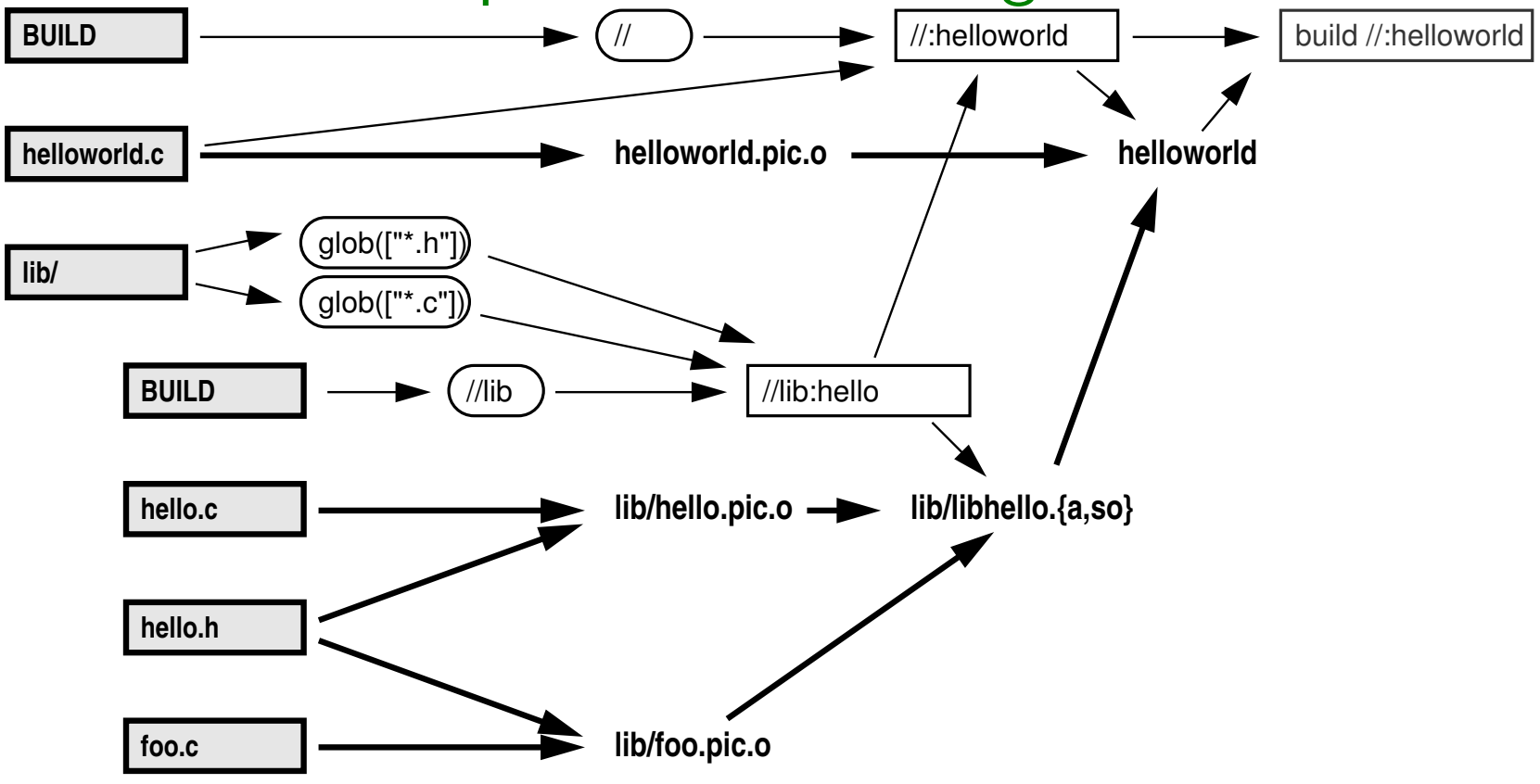
Example cont'd: Adding a File



Example cont'd: Adding a File



Example cont'd: Adding a File



Actions

Actions

- action do the actual work of building

Actions

- action do the actual work of building
  . . . and hence take the most time

Actions

- action do the actual work of building

  . . . and hence take the most time

⇝ particularly interesting to avoid unnecessary actions

Actions

- action do the actual work of building

  . . . and hence take the most time

⇝ particularly interesting to avoid unnecessary actions

  - dependency graph shows if prerequisites changed

Actions

- action do the actual work of building

  . . . and hence take the most time

⇝ particularly interesting to avoid unnecessary actions

  - dependency graph shows if prerequisites changed
  - caching of input/output-relation itself

Actions

- action do the actual work of building

  . . . and hence take the most time

⇝ particularly interesting to avoid unnecessary actions
  - dependency graph shows if prerequisites changed
  - caching of input/output-relation itself

! requires all inputs/outputs to be known to `bazel`

Actions

- action do the actual work of building

  . . . and hence take the most time

⇝ particularly interesting to avoid unnecessary actions
  - dependency graph shows if prerequisites changed
  - caching of input/output-relation itself

! requires all inputs/outputs to be known to `bazel`
  - so, no `.done_foo` targets,
  - and only reading *declared inputs*

Actions

- action do the actual work of building

  . . . and hence take the most time

⇝ particularly interesting to avoid unnecessary actions
  - dependency graph shows if prerequisites changed
  - caching of input/output-relation itself

! requires all inputs/outputs to be known to `bazel`

Actions

- action do the actual work of building

  . . . and hence take the most time

⇝ particularly interesting to avoid unnecessary actions

  - dependency graph shows if prerequisites changed
  - caching of input/output-relation itself

! requires all inputs/outputs to be known to `bazel`

⇝ facilitate correct I/O by running actions in “sandboxes”

Actions

- action do the actual work of building

  . . . and hence take the most time

⇝ particularly interesting to avoid unnecessary actions
  - dependency graph shows if prerequisites changed
  - caching of input/output-relation itself

! requires all inputs/outputs to be known to `bazel`

⇝ facilitate correct I/O by running actions in "sandboxes"
  - isolated environment
    - only declared inputs/tools present
    - only declared outputs copied out

Actions

- action do the actual work of building

  . . . and hence take the most time

⇝ particularly interesting to avoid unnecessary actions
  - dependency graph shows if prerequisites changed
  - caching of input/output-relation itself

! requires all inputs/outputs to be known to `bazel`

⇝ facilitate correct I/O by running actions in "sandboxes"
  - isolated environment
    - only declared inputs/tools present
    - only declared outputs copied out
  - depending on OS, different approaches (`chroot`, temp dir, . . .)

Actions

- action do the actual work of building

  . . . and hence take the most time

⇝ particularly interesting to avoid unnecessary actions

  - dependency graph shows if prerequisites changed
  - caching of input/output-relation itself

! requires all inputs/outputs to be known to `bazel`

⇝ facilitate correct I/O by running actions in “sandboxes”

Actions

- action do the actual work of building

  . . . and hence take the most time

⇝ particularly interesting to avoid unnecessary actions

  - dependency graph shows if prerequisites changed
  - caching of input/output-relation itself

! requires all inputs/outputs to be known to `bazel`

⇝ facilitate correct I/O by running actions in "sandboxes"

- bonus: remote execution

Actions

- action do the actual work of building

  . . . and hence take the most time

⇝ particularly interesting to avoid unnecessary actions
  - dependency graph shows if prerequisites changed
  - caching of input/output-relation itself

! requires all inputs/outputs to be known to `bazel`

⇝ facilitate correct I/O by running actions in "sandboxes"

- bonus: remote execution

  ⇒ enables shared caches.

  (All engineers working on the same code base!)

Extending Bazel

Extending Bazel

- Bazel has built-in rules

Extending Bazel

- Bazel has built-in rules
  - specialized rules with knowledge about certain languages

    `cc_library`, `cc_binary`, `java_library`, `java_binary`, ...

Extending Bazel

- Bazel has built-in rules
  - specialized rules with knowledge about certain languages
    `cc_library`, `cc_binary`, `java_library`, `java_binary`, ...
  - generic ones, in particular `genrule`

Extending Bazel

- Bazel has built-in rules
  - specialized rules with knowledge about certain languages
    `cc_library`, `cc_binary`, `java_library`, `java_binary`, ...
  - generic ones, in particular `genrule`
    → just specify a shell command (with `$@`, `$<`, ...)

Extending Bazel

- Bazel has built-in rules
  - specialized rules with knowledge about certain languages
    `cc_library`, `cc_binary`, `java_library`, `java_binary`, ...
  - generic ones, in particular `genrule`
    → just specify a shell command (with `$@`, `$<`, ...)
    (basically the only rule available in a `Makefile`*)*

Extending Bazel

- Bazel has built-in rules

Extending Bazel

- Bazel has built-in rules
- but adding specialized rules for every language doesn't scale

Extending Bazel

- Bazel has built-in rules
- but adding specialized rules for every language doesn't scale

⇝ need ways to extend `BUILD` language

Extending Bazel

- Bazel has built-in rules
- but adding specialized rules for every language doesn't scale

⇝ need ways to extend `BUILD` language: Skylark

Extending Bazel

- Bazel has built-in rules
- but adding specialized rules for every language doesn't scale

⇝ need ways to extend `BUILD` language: Skylark

  - Python-like language *(familiar syntax)*

Extending Bazel

- Bazel has built-in rules
- but adding specialized rules for every language doesn't scale

⇝ need ways to extend `BUILD` language: Skylark

- Python-like language *(familiar syntax)*
- but restricted to a simple core
  without global state, complicated features, . . .

Extending Bazel

- Bazel has built-in rules
- but adding specialized rules for every language doesn't scale

⇝ need ways to extend `BUILD` language: Skylark

- Python-like language *(familiar syntax)*
- but restricted to a simple core
  without global state, complicated features, . . .

  ⇝ deterministic, hermetic evaluation

Extending Bazel

- Bazel has built-in rules
- but adding specialized rules for every language doesn't scale

⇝ need ways to extend `BUILD` language: Skylark

Extending Bazel

- Bazel has built-in rules
- but adding specialized rules for every language doesn't scale

⇝ need ways to extend `BUILD` language: Skylark

- simple case: can compose it from existing rules

Extending Bazel

- Bazel has built-in rules
- but adding specialized rules for every language doesn't scale

⇝ need ways to extend `BUILD` language: Skylark

- simple case: can compose it from existing rules

  "that `sh`*-script with these params; always create 5 targets . . ."*

Extending Bazel

- Bazel has built-in rules
- but adding specialized rules for every language doesn't scale

⇝ need ways to extend `BUILD` language: Skylark

- simple case: can compose it from existing rules ⇝ macros

Extending Bazel

- Bazel has built-in rules
- but adding specialized rules for every language doesn't scale

⇝ need ways to extend `BUILD` language: Skylark

- simple case: can compose it from existing rules ⇝ macros

```
def mylang(name="", param="default", srcs=[]):
  script = str(Label("//rules/mylang:bld.sh"))
  native.genrule(
    name = name + "_out",
    tools = [script],
    cmd = "env ...  $(location " + script + ")"
          + "...  $@ $(SRCS)",
    ...)
  native....
```

Extending Bazel

- Bazel has built-in rules
- but adding specialized rules for every language doesn't scale

⇝ need ways to extend `BUILD` language: Skylark

- simple case: can compose it from existing rules ⇝ macros

Extending Bazel

- Bazel has built-in rules
- but adding specialized rules for every language doesn't scale

⇝ need ways to extend `BUILD` language: Skylark

- simple case: can compose it from existing rules ⇝ macros
- all extensions are loaded in `BUILD` files

```
load("//....bzl", "mylang")
```

Extending Bazel

- Bazel has built-in rules
- but adding specialized rules for every language doesn't scale

⇝ need ways to extend `BUILD` language: Skylark

- simple case: can compose it from existing rules ⇝ macros
- all extensions are loaded in `BUILD` files
  `load("//....bzl", "mylang")`
- not so simple case: rules
  freely specify actions, argument declaration, ...

The Task of Open-Sourcing Bazel

The Task of Open-Sourcing Bazel

Bazel became open-source only after years of internal use

The Task of Open-Sourcing Bazel

Bazel became open-source only after years of internal use

. . . on a single repository. *(large, but just one)*

The Task of Open-Sourcing Bazel

Bazel became open-source only after years of internal use . . . on a single repository. *(large, but just one)*

- lot of dependencies, including Google-specific ones

The Task of Open-Sourcing Bazel

Bazel became open-source only after years of internal use . . . on a single repository. *(large, but just one)*

- lot of dependencies, including Google-specific ones
  "We have those libs anyway, so let's just use them."

The Task of Open-Sourcing Bazel

Bazel became open-source only after years of internal use . . . on a single repository. *(large, but just one)*

- lot of dependencies, including Google-specific ones

The Task of Open-Sourcing Bazel

Bazel became open-source only after years of internal use . . . on a single repository. *(large, but just one)*

- lot of dependencies, including Google-specific ones
- focus on the "Google languages" (and that built in)

The Task of Open-Sourcing Bazel

Bazel became open-source only after years of internal use . . . on a single repository. *(large, but just one)*

- lot of dependencies, including Google-specific ones
- focus on the "Google languages" (and that built in)
- no stable interfaces

The Task of Open-Sourcing Bazel

Bazel became open-source only after years of internal use
. . . on a single repository. *(large, but just one)*

- lot of dependencies, including Google-specific ones
- focus on the "Google languages" (and that built in)
- no stable interfaces
  "I know all the uses of my interface, so I can easily change it."

The Task of Open-Sourcing Bazel

Bazel became open-source only after years of internal use . . . on a single repository. *(large, but just one)*

- lot of dependencies, including Google-specific ones
- focus on the "Google languages" (and that built in)
- no stable interfaces

The Task of Open-Sourcing Bazel

Bazel became open-source only after years of internal use . . . on a single repository. *(large, but just one)*

- lot of dependencies, including Google-specific ones
- focus on the "Google languages" (and that built in)
- no stable interfaces
- hard-coded paths everywhere

The Task of Open-Sourcing Bazel

Bazel became open-source only after years of internal use
. . . on a single repository. *(large, but just one)*

- lot of dependencies, including Google-specific ones
- focus on the "Google languages" (and that built in)
- no stable interfaces
- hard-coded paths everywhere
  "I know how my environment and how my compiler is called."

The Task of Open-Sourcing Bazel

Bazel became open-source only after years of internal use . . . on a single repository. *(large, but just one)*

- lot of dependencies, including Google-specific ones
- focus on the "Google languages" (and that built in)
- no stable interfaces
- hard-coded paths everywhere

The Task of Open-Sourcing Bazel

Bazel became open-source only after years of internal use . . . on a single repository. *(large, but just one)*

- lot of dependencies, including Google-specific ones
- focus on the "Google languages" (and that built in)
- no stable interfaces
- hard-coded paths everywhere
- . . .

Bazel Roadmap

Bazel Roadmap

- Big goal “1.0”. Properly open-source (expected 2018).

Bazel Roadmap

- Big goal “1.0”. Properly open-source (expected 2018).
  - public primary repository

Bazel Roadmap

- Big goal “1.0”. Properly open-source (expected 2018).
  - public primary repository
    $\rightsquigarrow$ clear interfaces between bazel, and, e.g., Google’s use

Bazel Roadmap

- Big goal “1.0”. Properly open-source (expected 2018).
  - public primary repository

Bazel Roadmap

- Big goal "1.0". Properly open-source (expected 2018).
  - public primary repository
  - all design reviews public

Bazel Roadmap

- Big goal “1.0”. Properly open-source (expected 2018).
  - public primary repository
  - all design reviews public
  - core team consisting not only of Google employees

Bazel Roadmap

- Big goal "1.0". Properly open-source (expected 2018).
  - public primary repository
  - all design reviews public
  - core team consisting not only of Google employees
  - stable build language and APIs

Bazel Roadmap

- Big goal "1.0". Properly open-source (expected 2018).
  - public primary repository
  - all design reviews public
  - core team consisting not only of Google employees
  - stable build language and APIs
- On the way there, technical improvements

Bazel Roadmap

- Big goal "1.0". Properly open-source (expected 2018).
  - public primary repository
  - all design reviews public
  - core team consisting not only of Google employees
  - stable build language and APIs
- On the way there, technical improvements
  - remote execution API

Bazel Roadmap

- Big goal "1.0". Properly open-source (expected 2018).
  - public primary repository
  - all design reviews public
  - core team consisting not only of Google employees
  - stable build language and APIs
- On the way there, technical improvements
  - remote execution API
  - community repositories of Skylark rules

Bazel Roadmap

- Big goal "1.0". Properly open-source (expected 2018).
  - public primary repository
  - all design reviews public
  - core team consisting not only of Google employees
  - stable build language and APIs
- On the way there, technical improvements
  - remote execution API
  - community repositories of Skylark rules

  Bazel more language agnostic tool

Bazel Roadmap

- Big goal "1.0". Properly open-source (expected 2018).
  - public primary repository
  - all design reviews public
  - core team consisting not only of Google employees
  - stable build language and APIs
- On the way there, technical improvements
  - remote execution API
  - community repositories of Skylark rules

Bazel Roadmap

- Big goal "1.0". Properly open-source (expected 2018).
  - public primary repository
  - all design reviews public
  - core team consisting not only of Google employees
  - stable build language and APIs
- On the way there, technical improvements
  - remote execution API
  - community repositories of Skylark rules
  - good story for remote repositories (including proper caching)

Bazel Roadmap

- Big goal "1.0". Properly open-source (expected 2018).
  - public primary repository
  - all design reviews public
  - core team consisting not only of Google employees
  - stable build language and APIs
- On the way there, technical improvements
  - remote execution API
  - community repositories of Skylark rules
  - good story for remote repositories (including proper caching)
  - ...

Summary

- declarative `BUILD` files
  . . . also supporting your own extensions
- *all* dependencies tracked $\rightsquigarrow$ correctness
  (sandboxes to ensure all I/O is known)
- full knowledge enables fast builds
  (caching of actions, remote execution, parallelism, . . .)
- open-source

Try Bazel

Try Bazel yourself.

- Homepage `https://bazel.build/`
- Mailing lists
  - `bazel-discuss@googlegroups.com`
  - `bazel-dev@googlegroups.com`
- Repository and issue tracker
  `https://github.com/bazelbuild/bazel`
- IRC `#bazel` on `irc.freenode.net`
- Release key fingerprint
  `71A1 D0EF CFEB 6281 FD04 37C9 3D59 19B4 4845 7EE0`

Thanks for your attention. Questions?