Bazel

{fast, correct} – choose two

Klaus Aehlig

August 19–20, 2017

Bazel

What is Bazel?

Bazel

What is Bazel?

- Bazel is a build tool
  I.e., organizes compiling/creating artifacts (libraries, executables, . . .) from sources.

Bazel

What is Bazel?

- Bazel is a build tool
  I.e., organizes compiling/creating artifacts (libraries, executables, . . .) from sources.
- open-source since 2015

Bazel

What is Bazel?

- Bazel is a build tool
  I.e., organizes compiling/creating artifacts (libraries, executables, . . .) from sources.
- open-source since 2015
- . . . but a longer (a decade) history as a Google-internal tool

Bazel

What is Bazel? And why yet another *make?

Bazel

What is Bazel? And why yet another `*make`?

- Scales to large repos with complex dependencies *(e.g.,* $10^{4.5}$ *engineers working on* 10^7 *files)*

Bazel

What is Bazel? And why yet another `*make`?

- Scales to large repos with complex dependencies *(e.g.,* $10^{4.5}$ *engineers working on* 10^7 *files)*
  - aggressive parallelism

Bazel

What is Bazel? And why yet another `*make`?

- Scales to large repos with complex dependencies *(e.g.,* $10^{4.5}$ *engineers working on* 10^7 *files)*
  - aggressive parallelism
  - aggressive caching

Bazel

What is Bazel? And why yet another `*make`?

- Scales to large repos with complex dependencies *(e.g., $10^{4.5}$ engineers working on 10^{7} files)*
  - aggressive parallelism
  - aggressive caching
  - . . . without losing correctness *(i.e., all artifacts as if freshly built from source)*

Bazel

What is Bazel? And why yet another `*make`?

- Scales to large repos with complex dependencies *(e.g.,* $10^{4.5}$ *engineers working on* 10^{7} *files)*
  - aggressive parallelism
  - aggressive caching
  - . . . without losing correctness *(i.e., all artifacts as if freshly built from source)*
- declarative style of `BUILD` files

Bazel

What is Bazel? And why yet another `*make`?

- Scales to large repos with complex dependencies *(e.g.,* $10^{4.5}$ *engineers working on* 10^7 *files)*
  - aggressive parallelism
  - aggressive caching
  - . . . without losing correctness *(i.e., all artifacts as if freshly built from source)*
- declarative style of `BUILD` files
  - separation of concerns writing code *vs* choosing correct (cross) compiling strategy

Bazel

What is Bazel? And why yet another `*make`?

- Scales to large repos with complex dependencies
  (e.g., $10^{4.5}$ engineers working on 10^7 files)
  - aggressive parallelism
  - aggressive caching
  - . . . without losing correctness
    (i.e., all artifacts as if freshly built from source)
- declarative style of `BUILD` files
  - separation of concerns
    writing code *vs* choosing correct (cross) compiling strategy
  - central maintenance point for build rules

Bazel

What is Bazel? And why yet another `*make`?

- Scales to large repos with complex dependencies *(e.g., $10^{4.5}$ engineers working on 10^7 files)*
  - aggressive parallelism
  - aggressive caching
  - . . . without losing correctness *(i.e., all artifacts as if freshly built from source)*
- declarative style of `BUILD` files
  - separation of concerns writing code *vs* choosing correct (cross) compiling strategy
  - central maintenance point for build rules
- generic tool *Can bring your own declarative rules for* `BUILD` *files*

An Example

Let's look at a `helloworld` example.

An Example

- main program `helloworld.c`

helloworld.c

An Example

helloworld.c

- main program `helloworld.c`

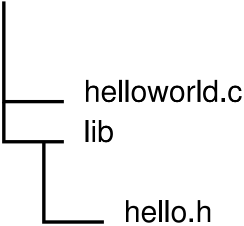
```
#include "lib/hello.h"

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`)



helloworld.c

lib

hello.h

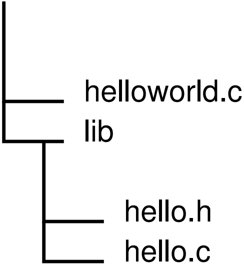
```
#ifndef HELLO_H
#define HELLO_H

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`)

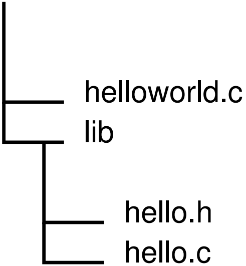
```
#include "hello.h"
#include <stdio.h>

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



helloworld.c
lib
hello.h
hello.c

An Example

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

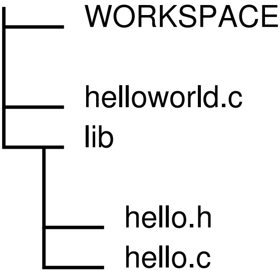


helloworld.c
lib
hello.h
hello.c

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



WORKSPACE

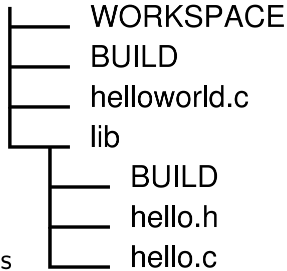
helloworld.c

lib

hello.h

hello.c

An Example



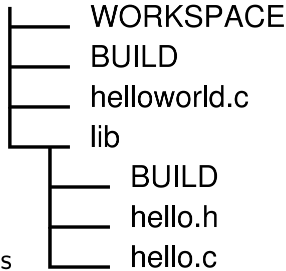
WORKSPACE
BUILD
helloworld.c
lib
BUILD
hello.h
hello.c

- 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

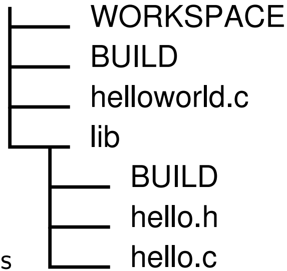
```
cc_binary(
  name="helloworld",
  srcs=["helloworld.c"],
  deps=["//lib:hello"],
)
```

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

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

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



WORKSPACE
BUILD
helloworld.c
lib
BUILD
hello.h
hello.c

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

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

Overview of a `bazel build`

Have declarative descriptions. What happens at `bazel build`?

Overview of a `bazel build`

Have declarative descriptions. What happens at `bazel build`?

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

Overview of a `bazel build`

Have declarative descriptions. What happens at `bazel build`?

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

Overview of a `bazel build`

Have declarative descriptions. What happens at `bazel build`?

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

Overview of a `bazel build`

Have declarative descriptions. What happens at `bazel 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`

Have declarative descriptions. What happens at `bazel 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)

Example cont'd: Dependencies

build //:helloworld

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

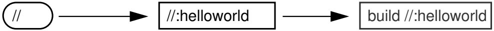
command

Example cont'd: Dependencies

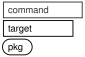
//:helloworld
build //:helloworld
command
target

We look at the target `:helloworld`

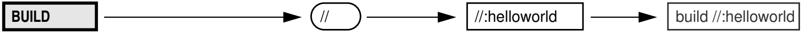
Example cont'd: Dependencies



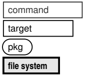
We look at the target `:helloworld`, in package //



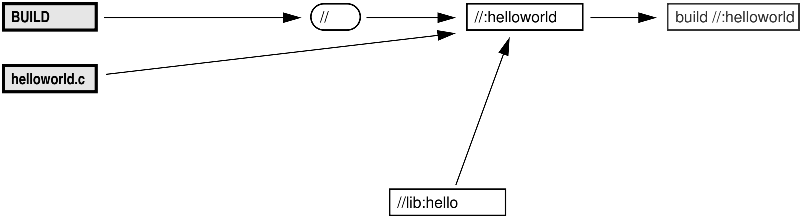
Example cont'd: Dependencies



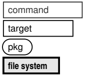
We look at the target `:helloworld`, in package `//`, in file `BUILD`



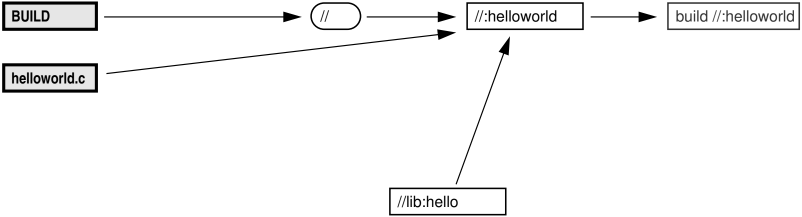
Example cont'd: Dependencies



Two declared dependencies

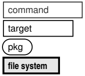


Example cont'd: Dependencies

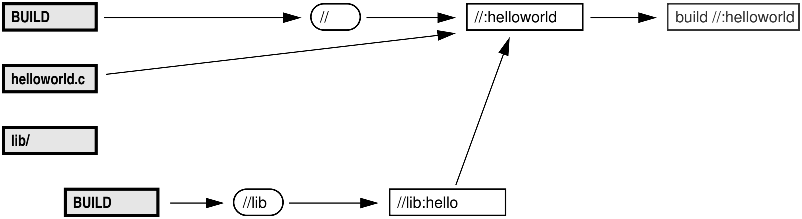


Two declared dependencies

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



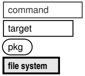
Example cont'd: Dependencies



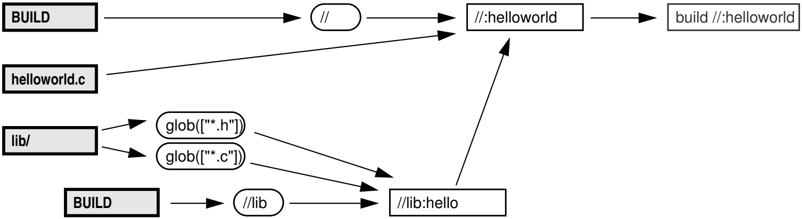
Two declared dependencies, one in a different package

Note: We construct dependency graph over package boundaries!

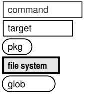
(no recursive calling)



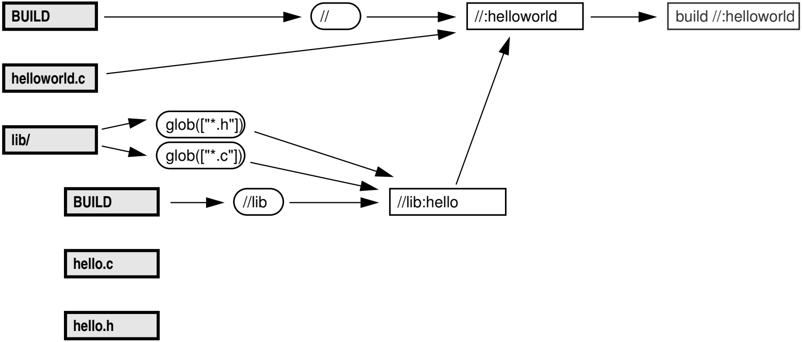
Example cont'd: Dependencies



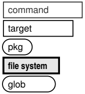
We discover `glob` expressions



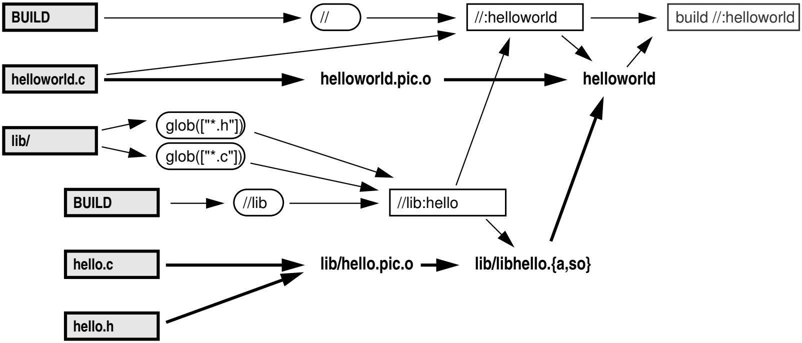
Example cont'd: Dependencies



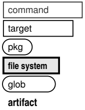
We discover `glob` expressions, and read the directory.



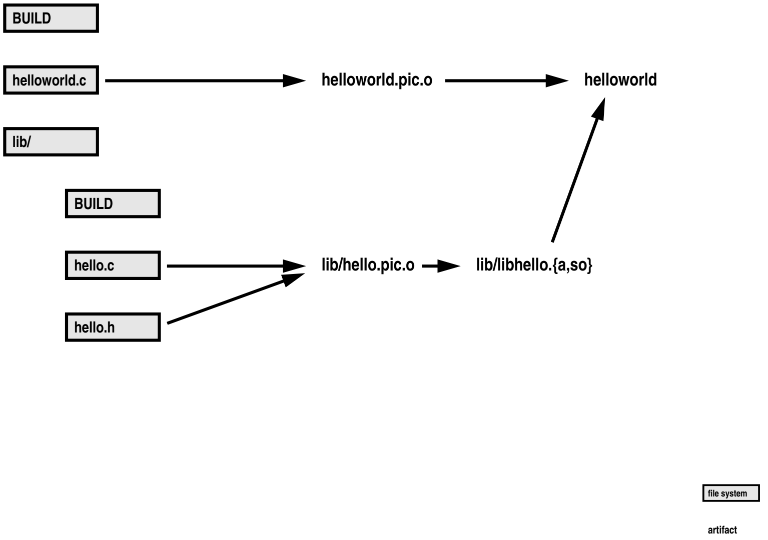
Example cont'd: Dependencies



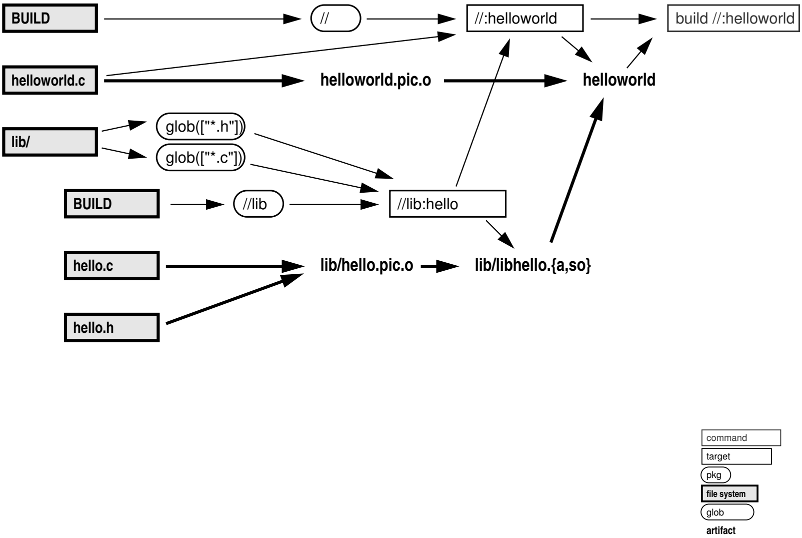
The rules tell us, which artifacts to build.



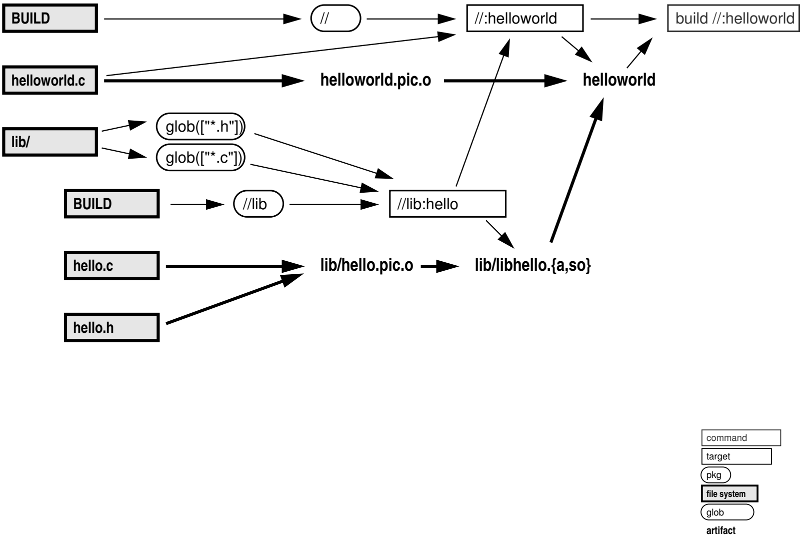
Example cont'd: Dependencies



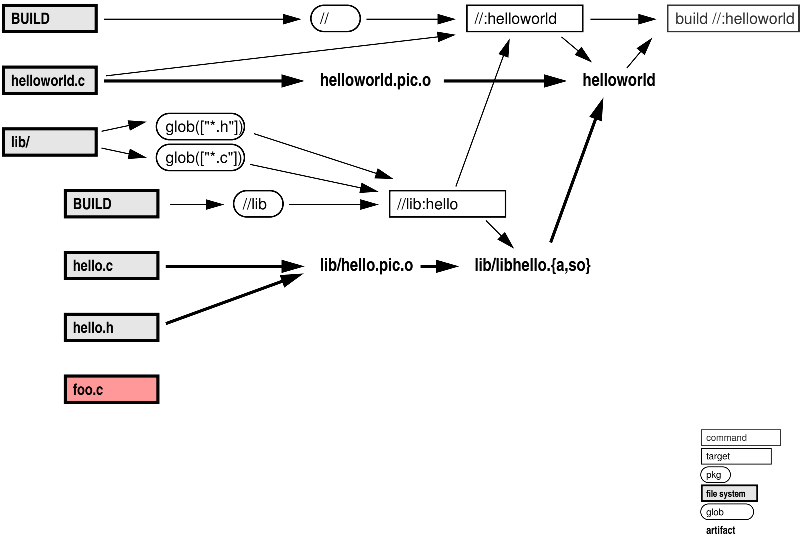
Example cont'd: Dependencies



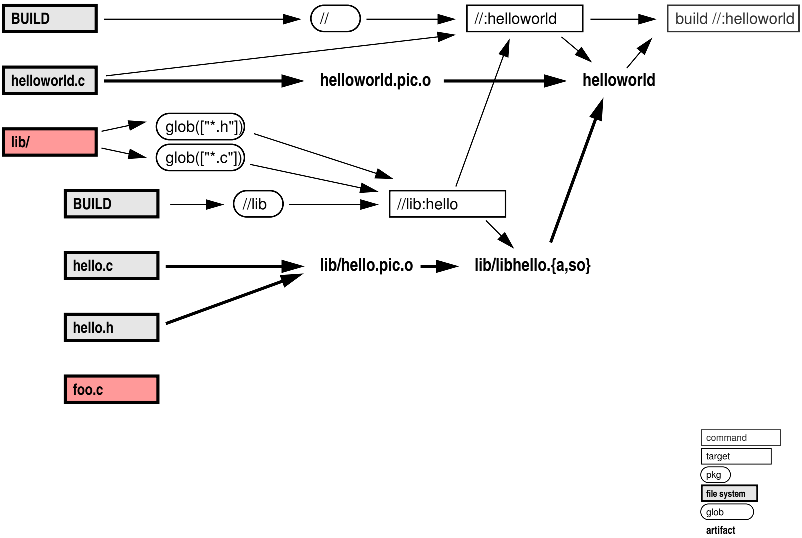
Example cont'd: Adding a File



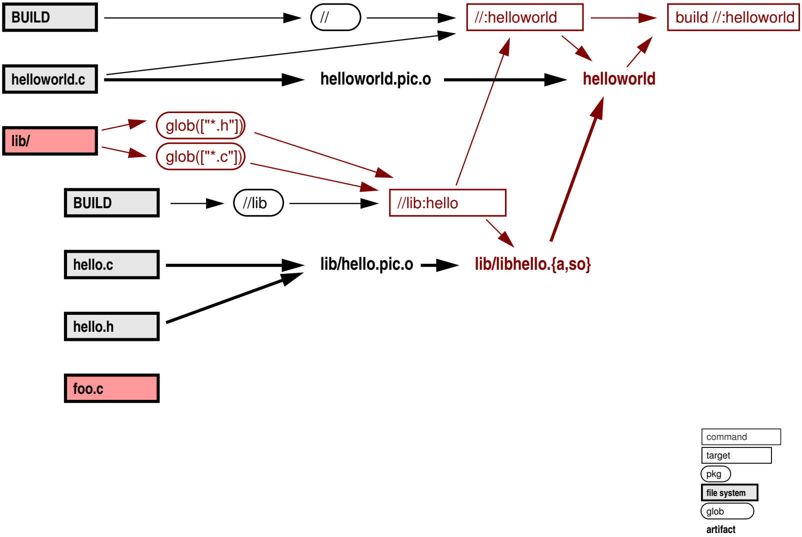
Example cont'd: Adding a File



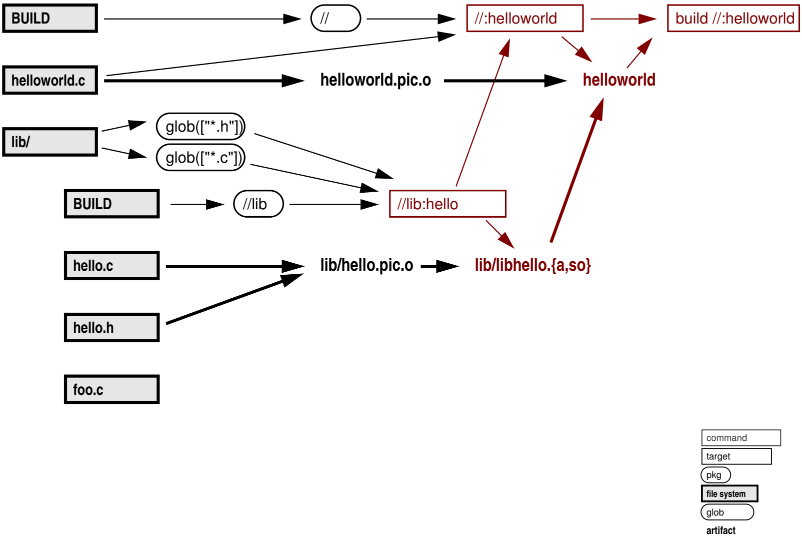
Example cont'd: Adding a File



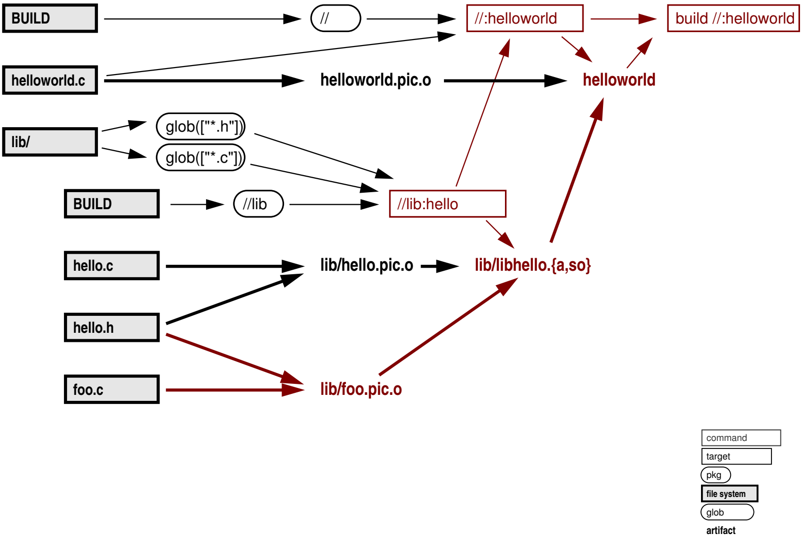
Example cont'd: Adding a File



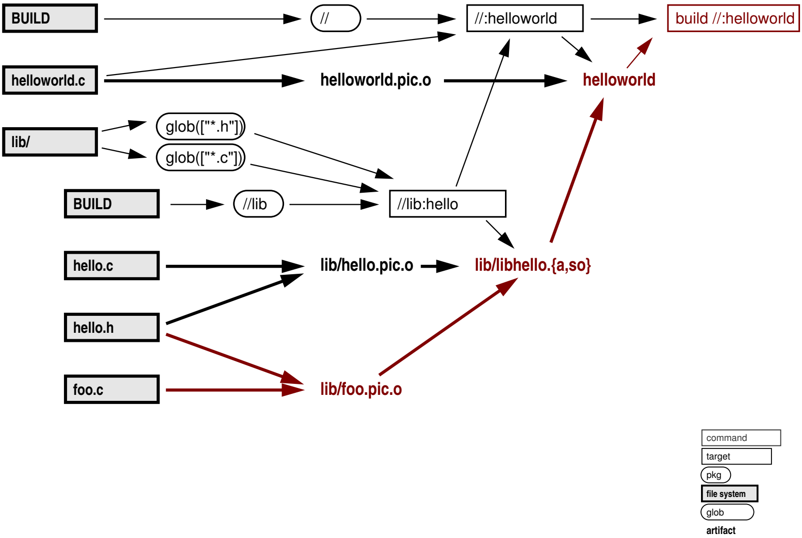
Example cont'd: Adding a File



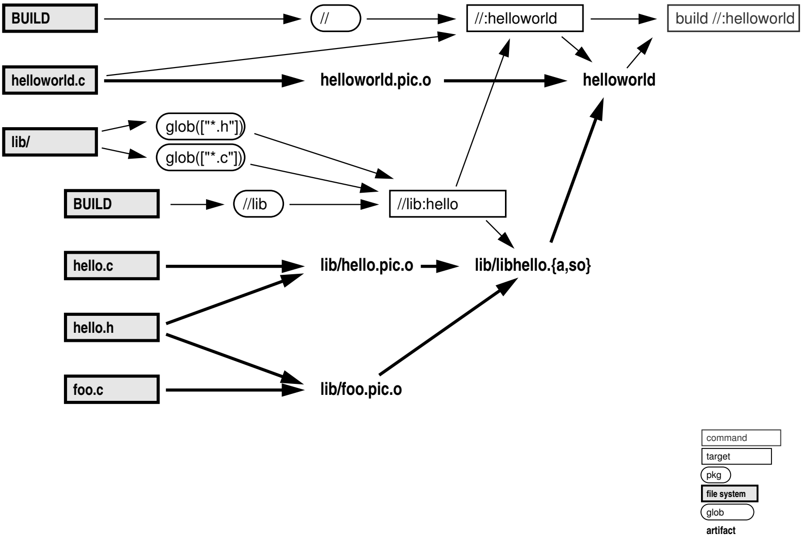
Example cont'd: Adding a File



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 (none, temp dir, `chroot`, . . .)

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.

  (Several close-by engineers working on the same code base!)

Skylark

Skylark

- Bazel has built-in rules

Skylark

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

Skylark

- 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`*)*

Skylark

- Bazel has built-in rules

Skylark

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

Skylark

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

⇝ need ways to expend `BUILD` language: Skylark

Skylark

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

⇝ need ways to expend `BUILD` language: Skylark

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

  ⇝ deterministic, hermetic evaluation

Skylark

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

⇝ need ways to expend `BUILD` language: Skylark

Skylark

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

⇝ need ways to expend `BUILD` language: Skylark

- To get a feeling for the language, let's do an example

Skylark

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

⇝ need ways to expend `BUILD` language: Skylark

- To get a feeling for the language, let's do an example

  . . . and step by step develop rules for LaTeX

Skylark

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

⇝ need ways to expend BUILD language: Skylark

- To get a feeling for the language, let's do an example

  . . . and step by step develop rules for LaTeX
  - typeset pdf files from textual description (*.tex files)

Skylark

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

⇝ need ways to expend `BUILD` language: Skylark

- To get a feeling for the language, let's do an example

  . . . and step by step develop rules for LaTeX

  - typeset `pdf` files from textual description (`*.tex` files)
  - the `*.tex` files can pull in other files (`.sty`, images, diagrams, `\input` other `.tex`-files)

Skylark

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

⇝ need ways to expend `BUILD` language: Skylark

- To get a feeling for the language, let's do an example

  ... and step by step develop rules for LaTeX
  - typeset `pdf` files from textual description (`*.tex` files)
  - the `*.tex` files can pull in other files (`.sty`, images, diagrams, `\input` other `.tex`-files)
  - `pdflatex main.tex && ...`

Skylark

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

⇝ need ways to expend `BUILD` language: Skylark

- To get a feeling for the language, let's do an example

  . . . and step by step develop rules for LaTeX

Macros

Macros

- First approach

Macros

- First approach
    - `latex`-rule is given by an entry point and a list of source files

Macros

- First approach
  - `latex`-rule is given by an entry point and a list of source files
  - have a script to typeset this
    (tmpdir, correct number of `pdflatex` *runs, …)*

Macros

- First approach (entry + files; script)

Macros

- First approach (entry + files; script)

⇝ write a macro in `rules/latex/latex.bzl`

```
def latex(name="", main="", srcs=[]):
  run = str(Label("//rules/latex:runlatex.sh"))
  native.genrule(
    name = name + "_pdf",
    srcs = srcs,
    cmd = ("sh $(location " + run +") $@"
           + " $(location " + main + ") $(SRCS)",
    outs = [name + ".pdf"],
    tools = [run],
  )
```

Macros

- First approach (entry + files; script)

⇝ write a macro in `rules/latex/latex.bzl`

```
def latex(name="", main="", srcs=[]):
    ...
    native.genrule(...)
```

Macros

- First approach (entry + files; script)

⇝ write a macro in `rules/latex/latex.bzl`

```
def latex(name="", main="", srcs=[]):
  ...
  native.genrule(...)
```

- can be loaded in `BUILD` files

Macros

- First approach (entry + files; script)

⇝ write a macro in `rules/latex/latex.bzl`

```
def latex(name="", main="", srcs=[]):
  ...
  native.genrule(...)
```

- can be loaded in `BUILD` files

```
load("//rules/latex/latex.bzl", "latex")
```

Macros

- First approach (entry + files; script)

⇝ write a macro in `rules/latex/latex.bzl`

```
def latex(name="", main="", srcs=[]):
  ...
  native.genrule(...)
```

- can be loaded in `BUILD` files

```
load("//rules/latex/latex.bzl", "latex")
latex(
  name = "slides",
  main = "main.tex",
  srcs = ["diagram.ps"],
)
```

Macros

- First approach (entry + files; script)

$\leadsto$ write a macro in `rules/latex/latex.bzl`

```
def latex(name="", main="", srcs=[]):
  ...
  native.genrule(...)
```

- can be loaded in `BUILD` files

```
load("//rules/latex/latex.bzl", "latex")
latex(
  name = "slides",
  main = "main.tex",
  srcs = ["diagram.ps"],
)
```

$\leadsto$ central maintenance; convenience-targets (`xpdf`, `pdfnup`, ...)

File Groups

File Groups

- Start thinking in groups of files

File Groups

- Start thinking in groups of files
  "That slide with all the diagrams belonging to it"

File Groups

- Start thinking in groups of files

File Groups

- Start thinking in groups of files
- Built-in rule: `filegroup`

```
filegroup(name = "foosection",
          srcs = ["foosection.tex", ":diagram"])
...
filegroup(
  name = "barchapter",
  srcs = ["barchapter.tex", ":foosection", ...])
```

File Groups

- Start thinking in groups of files
- Built-in rule: `filegroup`

```
filegroup(name = "foosection",
          srcs = ["foosection.tex", ":diagram"])
...
filegroup(
  name = "barchapter",
  srcs = ["barchapter.tex", ":foosection", ...])
```

- Gives a label to a set of files (with traversal order)
  ⇝ single maintenance point

File Groups

- Start thinking in groups of files
- Built-in rule: `filegroup`

```
filegroup(name = "foosection",
          srcs = ["foosection.tex", ":diagram"])
...
filegroup(
  name = "barchapter",
  srcs = ["barchapter.tex", ":foosection", ...])
```

- Gives a label to a set of files (with traversal order) ⇝ single maintenance point
- Can be nested, inserting the entries *(but implemented in a memory-efficient way!)*

Rules

Rules

- Next: missing argument checking, `argv` limits

Rules

- Next: missing argument checking, `argv` limits ⇝ Rules
  (also changing the script, now expecting an arguments file)

Rules

- Next: missing argument checking, argv limits ⇝ Rules
  (also changing the script, now expecting an arguments file)

```
latex = rule(
  attrs = {
    "main" : attr.label(allow_files=True),
    "srcs" : attr.label_list(allow_files=True),
    "_runlatex":  attr.label(
      cfg="host", allow_files=True,
      default = Label("//rules/latex:runlatex.sh")),
  },
  outputs = {"pdf" : "%{name}.pdf"},
  implementation = _latex_impl,
  )
```

Rules

- Next: missing argument checking, argv limits ⇝ Rules *(also changing the script, now expecting an arguments file)*

```
def _latex_impl(ctx):
    inputs = depset(ctx.files.srcs) \
             | depset(ctx.files.main)
    inputs_file = ctx.new_file(
        ctx.label.name + ".allinputs")
    ctx.file_action(
        inputs_file,
        "\n".join([f.path for f in inputs])
    )
    ...
```

Rules

- Next: missing argument checking, `argv` limits ⇝ Rules *(also changing the script, now expecting an arguments file)*

```
def _latex_impl(ctx):
    ...
    ctx.file_action(...)
    ...
```

Rules

- Next: missing argument checking, argv limits ⇝ Rules
  (also changing the script, now expecting an arguments file)

```
def _latex_impl(ctx):
    ...
    ctx.file_action(...)
    output = ctx.new_file(ctx.label.name + ".pdf")
    args = [f.path for f in ctx.files._runlatex] \
           + [output.path] \
           + [f.path for f in ctx.files.main[:1]] \
           + [inputs_file.path]
    ...
```

Rules

- Next: missing argument checking, `argv` limits ⇝ Rules
  (also changing the script, now expecting an arguments file)

```
def _latex_impl(ctx):
    ...
    ctx.file_action(...)
    output = ctx.new_file(ctx.label.name + ".pdf")
    args = ...
    ...
```

Rules

- Next: missing argument checking, `argv` limits ⇝ Rules
  (also changing the script, now expecting an arguments file)

```
def _latex_impl(ctx):
    ...
    args = ...
    ...
```

Rules

- Next: missing argument checking, argv limits ⇝ Rules *(also changing the script, now expecting an arguments file)*

```
def _latex_impl(ctx):
    ...
    args = ...
    ctx.action(
        inputs = list(inputs | depset([inputs_file])
                      | depset(ctx.files._runpdflatex))
        outputs = [output],
        command = ["/bin/sh"] + args,
        mnemonic = "PdfLatex",
        progress_message = "Typesetting %s as pdf" \
                           % ctx.label,
    )
```

Rules

- Next: missing argument checking, `argv` limits ⇝ Rules
  (also changing the script, now expecting an arguments file)

```
def _latex_impl(ctx):
    ...
    args = ...
    ctx.action(...)
```

Rules

- Next: missing argument checking, `argv` limits ⇝ Rules
  (also changing the script, now expecting an arguments file)

```
def _latex_impl(ctx):
    ...
    args = ...
    ctx.action(...)
```

- Additional benefits
  - Proper quoting for free
  - Meaningful progress messages

Providers

Providers

- Start to collect macro definitions

Providers

- Start to collect macro definitions, organized in file groups

Providers

- Start to collect macro definitions, organized in file groups
- Want to `\input` such a file group. . .

Providers

- Start to collect macro definitions, organized in file groups
- Want to \input such a file group. . .
- file_action is simple

```
includefile = rule(...)
def _includefile_impl(ctx):
  output = ctx.new_file(ctx.label.name + ".tex")
  deps = depset(ctx.files.srcs)
  includes = ["\\input{%s}\n" % f.short_path
                for f in deps]
  ctx.file_action(output = output,
                  content = "".join(includes))
```

Providers

- Start to collect macro definitions, organized in file groups
- Want to \input such a file group…
- file_action

```
includefile = rule(...)
def _includefile_impl(ctx):
  output = ctx.new_file(ctx.label.name + ".tex")
  deps = depset(ctx.files.srcs)
  includes = ["\\input{%s}\n" % f.short_path
              for f in deps]
  ctx.file_action(output = output,
                  content = "".join(includes))
```

Using this new file implicitly depends on the sources!

Providers

- Start to collect macro definitions, organized in file groups
- Want to `\input` such a file group. . .
- `file_action` plus provider

```
LtxInfo = provider()
includefile = rule(...)
def _includefile_impl(ctx):
  output = ctx.new_file(ctx.label.name + ".tex")
  deps = depset(ctx.files.srcs)
  includes = ["\\input{%s}\n" % f.short_path
              for f in deps]
  ctx.file_action(output = output,
                   content = "".join(includes))
  return [LtxInfo(refd = depset([output])|deps)]
```

Using this new file implicitly depends on the sources!

Providers

- Start to collect macro definitions, organized in file groups
- Want to `\input` such a file group. . .
- `file_action` plus provider

```
LtxInfo = provider()
includefile = rule(...)
def _includefile_impl(ctx):
  output = ctx.new_file(ctx.label.name + ".tex")
  deps = depset(ctx.files.srcs)
  includes = ["\\input{%s}\n" % f.short_path
              for f in deps]
  ctx.file_action(output = output,
                  content = "".join(includes))
  return [LtxInfo(refd = depset([output])|deps)]
```

Using this new file implicitly depends on the sources!

Providers

- Start to collect macro definitions, organized in file groups
- Want to \input such a file group. . .
- file_action plus provider

```
LtxInfo = provider()
includefile = rule(...)
def _includefile_impl(ctx):
  output = ctx.new_file(ctx.label.name + ".tex")
  deps = depset(ctx.files.srcs)
  includes = ["\\input{%s}\n" % f.short_path
              for f in deps]
  ctx.file_action(output = output,
                  content = "".join(includes))
  return [LtxInfo(refd = depset([output])|deps)]
```

Providers

- Start to collect macro definitions, organized in file groups
- Want to \input such a file group...
- file_action plus provider

```
...
def _includefile_impl(ctx):
    ...
    return [LtxInfo(refd = depset([output])|deps)]
```

Providers

- Start to collect macro definitions, organized in file groups
- Want to `\input` such a file group...
- `file_action` plus provider

```
def _includefile_impl(ctx):
    ...
    return [LtxInfo(refd = depset([output])|deps)]
```

Providers

- Start to collect macro definitions, organized in file groups
- Want to `\input` such a file group. . .
- `file_action` plus provider

```
def _includefile_impl(ctx):
    ...
    return [LtxInfo(refd = depset([output])|deps)]
```

- Consuming rules can use it

```
def _latex_impl(ctx):
      inputs = depset(ctx.files.srcs) \
                   | depset(ctx.files.main)




      ...
```

Providers

- Start to collect macro definitions, organized in file groups
- Want to `\input` such a file group…
- `file_action` plus provider

```
def _includefile_impl(ctx):
  ...
  return [LtxInfo(refd = depset([output])|deps)]
```

- Consuming rules can use it

```
def _latex_impl(ctx):
    inputs = depset(ctx.files.srcs) \
             | depset(ctx.files.main)
    for i in ctx.attr.srcs:
        if LtxInfo in i:
            inputs = inputs | i[LtxInfo].refd
    ...
```

Providers

- Start to collect macro definitions, organized in file groups
- Want to `\input` such a file group. . .
- `file_action` plus provider

```
def _includefile_impl(ctx):
  ...
  return [LtxInfo(refd = depset([output])|deps)]
```

- Consuming rules can use it

```
def _latex_impl(ctx):
    inputs = depset(ctx.files.srcs) \
             | depset(ctx.files.main)
    for i in ctx.attr.srcs:
        if LtxInfo in i:
            inputs = inputs | i[LtxInfo].refd
    ...
```

Summary

- declarative `BUILD` files
- generic tool: can bring your own rules
  (Python-like extension language; can start easy)
- *all* dependencies tracked ⇝ 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?