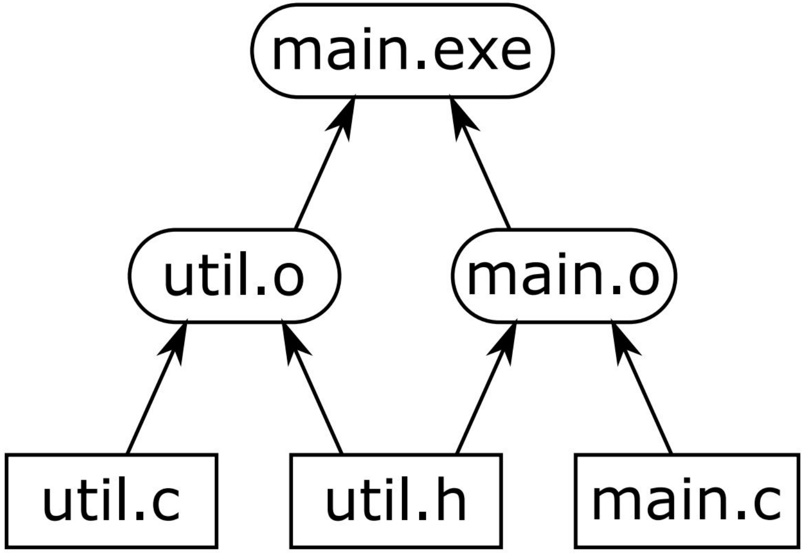
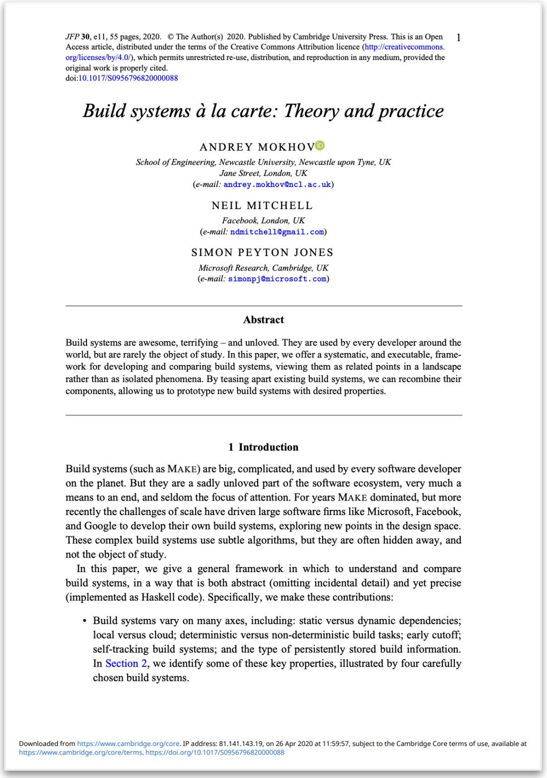
What Makes Buck2 Special?

Neil Mitchell, Meta
https://ndmitchell.com/



(a) Task dependency graph

Build systems à la carte: Theory and practice
Mokhov, Mitchell, Peyton Jones



JFP **30**, e11, 55 pages, 2020. © The Author(s) 2020. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution, and reproduction in any medium, provided the original work is properly cited.
doi:10.1017/S0956796820000088

Build systems à la carte: Theory and practice

ANDREY MOKHOV
School of Engineering, Newcastle University, Newcastle upon Tyne, UK
Jane Street, London, UK
(*e-mail:* andrey.mokhov@ncl.ac.uk)

NEIL MITCHELL
Facebook, London, UK
(*e-mail:* ndmitchell@gmail.com)

SIMON PEYTON JONES
Microsoft Research, Cambridge, UK
(*e-mail:* simonpj@microsoft.com)

Abstract

Build systems are awesome, terrifying – and unloved. They are used by every developer around the world, but are rarely the object of study. In this paper, we offer a systematic, and executable, framework for developing and comparing build systems, viewing them as related points in a landscape rather than as isolated phenomena. By teasing apart existing build systems, we can recombine their components, allowing us to prototype new build systems with desired properties.

1 Introduction

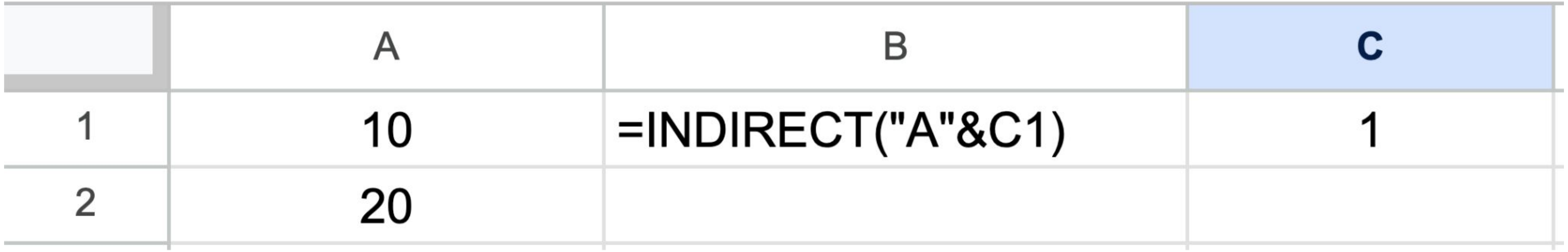
Build systems (such as MAKE) are big, complicated, and used by every software developer on the planet. But they are a sadly unloved part of the software ecosystem, very much a means to an end, and seldom the focus of attention. For years MAKE dominated, but more recently the challenges of scale have driven large software firms like Microsoft, Facebook, and Google to develop their own build systems, exploring new points in the design space. These complex build systems use subtle algorithms, but they are often hidden away, and not the object of study.

In this paper, we give a general framework in which to understand and compare build systems, in a way that is both abstract (omitting incidental detail) and yet precise (implemented as Haskell code). Specifically, we make these contributions:

- Build systems vary on many axes, including: static versus dynamic dependencies; local versus cloud; deterministic versus non-deterministic build tasks; early cutoff; self-tracking build systems; and the type of persistently stored build information. In Section 2, we identify some of these key properties, illustrated by four carefully chosen build systems.

Dynamic (monadic) vs static (applicative)

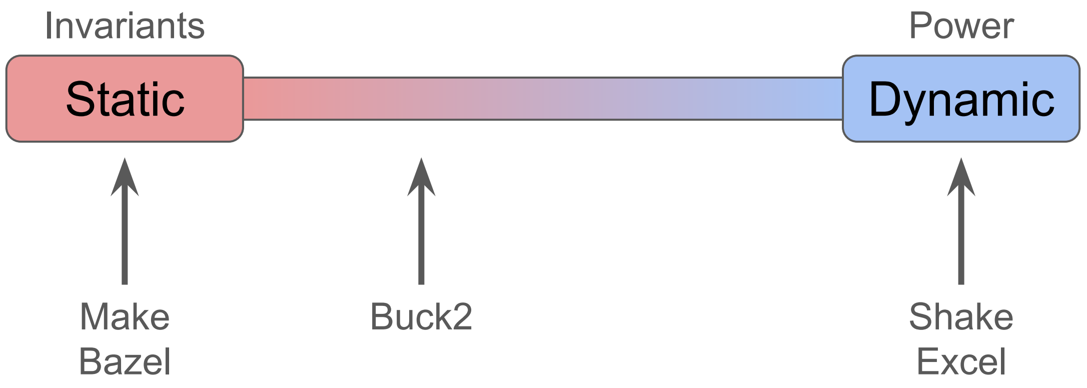
Unlike Make, Excel (the build system) does not need to know all task dependencies upfront. Indeed, some dependencies may change dynamically during computation.

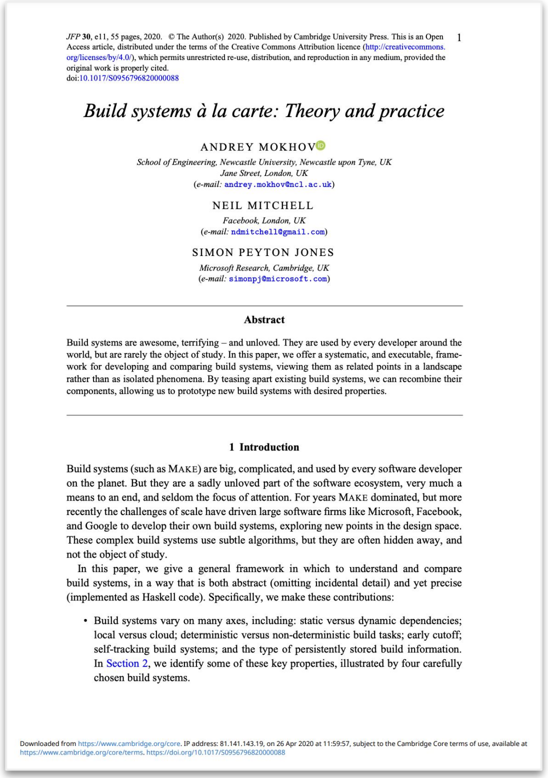


	A	B	C
1	10	=INDIRECT("A"&C1)	1
2	20		

What makes Buck2 special?

- Two build graphs (also true for Buck1, Bazel)
- Somewhere between dynamic and static





JFP **30**, e11, 55 pages, 2020.
doi:10.1017/S0956796820000088

Build systems à la carte: Theory and practice

ANDREY MOKHOV
School of Engineering, Newcastle University, Newcastle upon Tyne, UK
Jane Street, London, UK
(*e-mail:* andrey.mokhov@ncl.ac.uk)

NEIL MITCHELL
Facebook, London, UK
(*e-mail:* ndmitchell@gmail.com)

SIMON PEYTON JONES
Microsoft Research, Cambridge, UK
(*e-mail:* simonpj@microsoft.com)

Abstract

Build systems are awesome, terrifying – and unloved. They are used by every developer around the world, but are rarely the object of study. In this paper, we offer a systematic, and executable, framework for developing and comparing build systems, viewing them as related points in a landscape rather than as isolated phenomena. By teasing apart existing build systems, we can recombine their components, allowing us to prototype new build systems with desired properties.

1 Introduction

Build systems (such as MAKE) are big, complicated, and used by every software developer on the planet. But they are a sadly unloved part of the software ecosystem, very much a means to an end, and seldom the focus of attention. For years MAKE dominated, but more recently the challenges of scale have driven large software firms like Microsoft, Facebook, and Google to develop their own build systems, exploring new points in the design space. These complex build systems use subtle algorithms, but they are often hidden away, and not the object of study.

In this paper, we give a general framework in which to understand and compare build systems, in a way that is both abstract (omitting incidental detail) and yet precise (implemented as Haskell code). Specifically, we make these contributions:

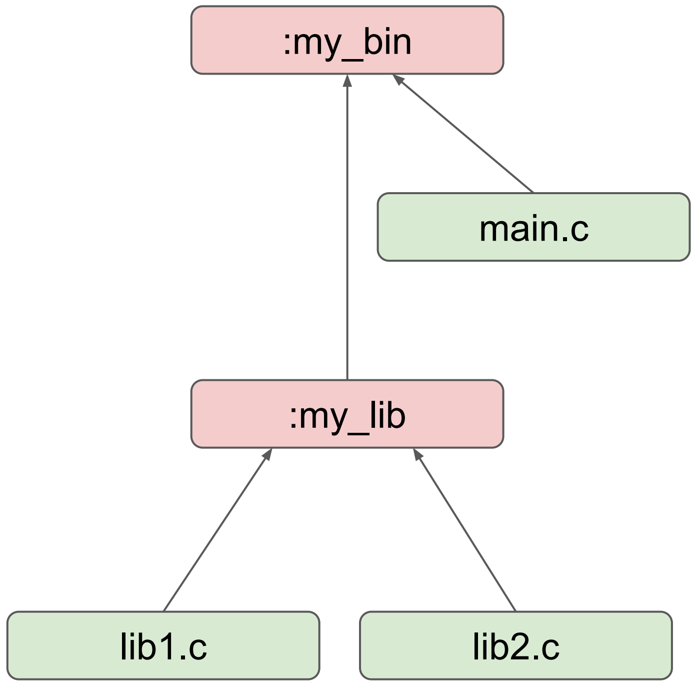
- Build systems vary on many axes, including: static versus dynamic dependencies; local versus cloud; deterministic versus non-deterministic build tasks; early cutoff; self-tracking build systems; and the type of persistently stored build information. In Section 2, we identify some of these key properties, illustrated by four carefully chosen build systems.

A Buck2 example

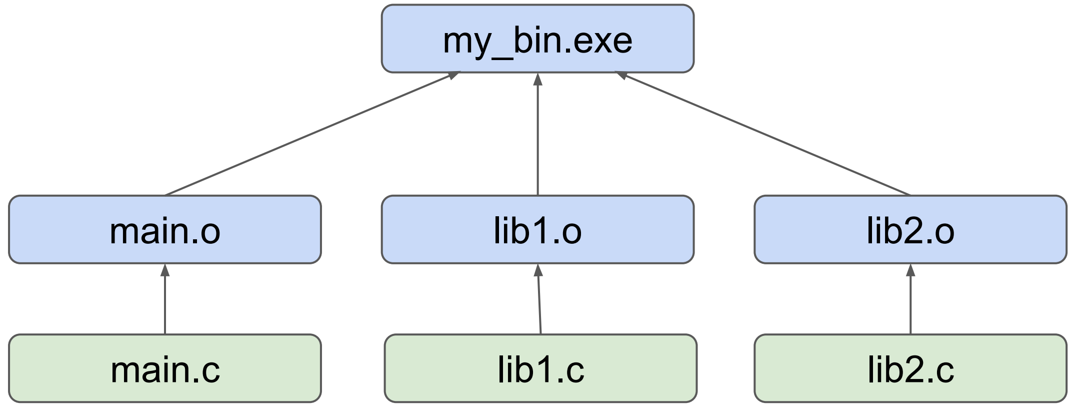
```
c_library(
    name = "my_lib",
    srcs = ["lib1.c", "lib2.c"],
)


c_binary(
    name = "my_bin",
    srcs = ["main.c"],
    deps = [":my_lib"],
)
```

Bazel and Buck1 look very similar.
These are all *large scale* build systems.

Target graph



Action graph



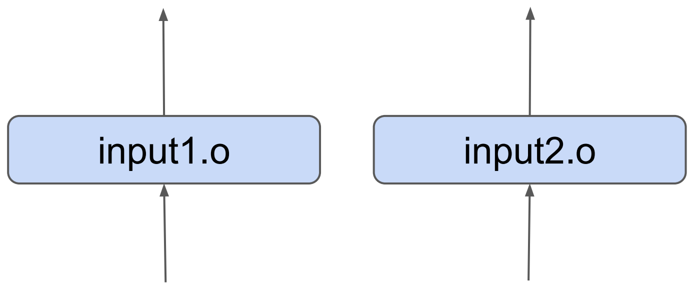
Target graph

- Exactly what the user writes
- In terms of targets
- Entirely static
- Used for computing the action graph (analysis)
- Used for static analysis
  - SBOM
  - CI test selection

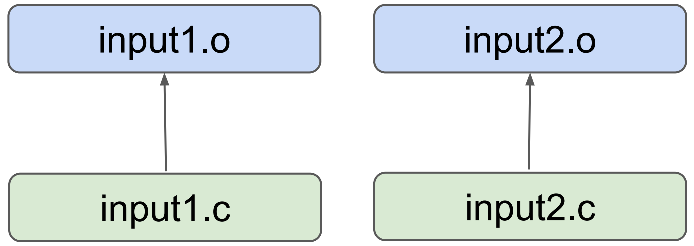
Action graph

- Computed from the target graph
- Files connected by actions (command lines)
- Operational concerns
- Cannot invalidate static analysis done on the target graph

Operational concerns

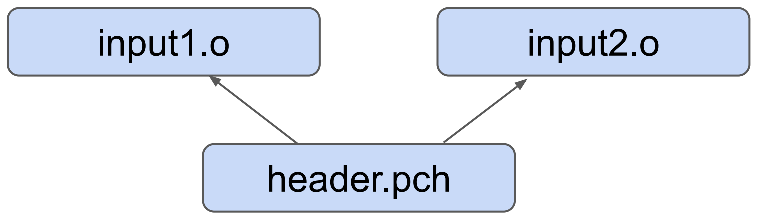


Parallelism

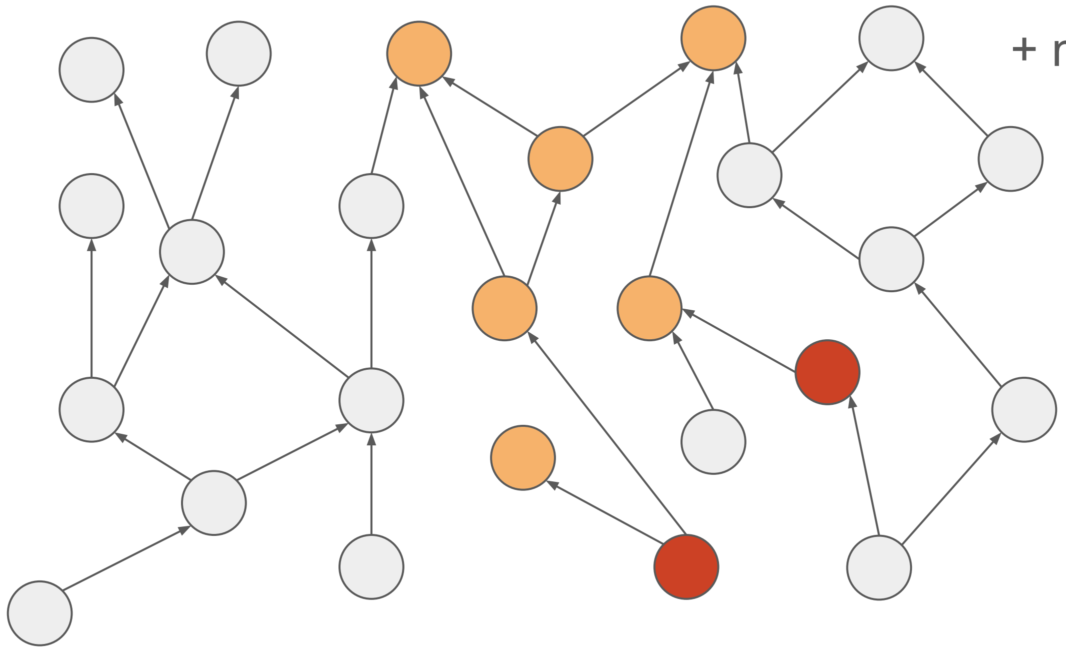


Incrementality

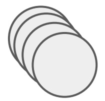
Sharing



Invariants: Target Determination (CI)



+ millions more unchanged



Requires: Zero impact from targets you don’t transitively depend on

https://github.com/facebookincubator/buck2-change-detector

Analysis

Analysis is a function that translates a target graph node to some action graph

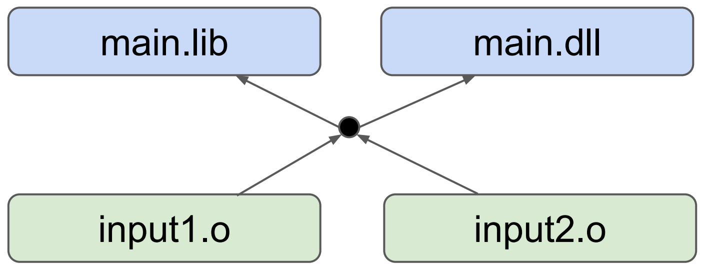
- Output:
  - Artifacts (with actions), plus pure data (metadata)
- Input:
  - Pure data for this target node
  - Artifacts for all source inputs of this target node
  - Outputs of analysis from all target dependencies

- Can only access artifacts bubbled up by your dependencies
  - Assuming analysis is a pure function (which Buck2 and Bazel both enforce with Starlark)

Building the graph: run

```
run(command_line_arguments, [input, files], [output, files])
```

This gives us static dependency graphs



Does more “dynamic” give us more power?

Dynamic: Look at your computed results then ~~grab arbitrary nodes and~~ make parts of the graph.

- Arbitrary node lookup violates target graph static analysis.
- Expressive power? No. A 1:1 mapping to the target graph would be maximally expressive.
- Operationally? Yes! If we have more fine grained nodes, we can do better parallelism, incrementality, sharing.

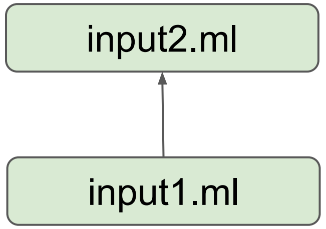
Problem 1: OCaml/Haskell

An OCaml library is a bunch of `.ml` files, that have internal imports.

Must compile each module in dependency order. Three options:

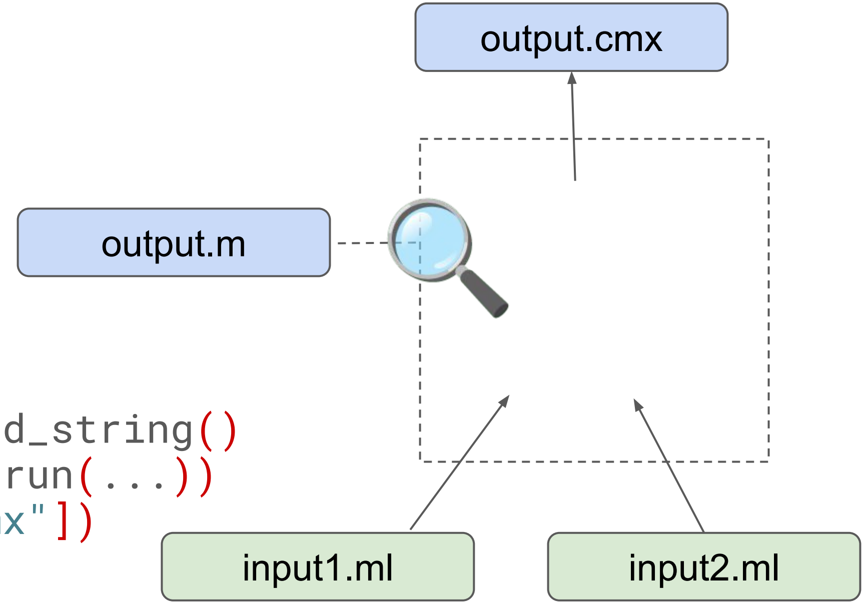
1. Compile every file in the library in one action
   (coarse, low parallelism, incrementality) - how Haskell/Bazel works.
2. Write every file and its internal dependencies in the target graph
   (duplicated information) - how OCaml/Bazel works.
3. Add more powerful features.

```
ocamldep $inputs -o output.m
for x in linearise(output.m)
    ocamlc $x
```



Solution 1: Dynamic outputs

```
run("ocamldep …", srcs, ["output.m"])


def f(ctx, artifacts, outputs):
    makefile = artifacts["output.m"].read_string()
    follow_makefile(makefile, lambda x: run(...))
    run("combine", …, outputs["output.cmx"])


dynamic_output(dynamic = ["output.m"], output = ["output.cmx"], f = f)
```

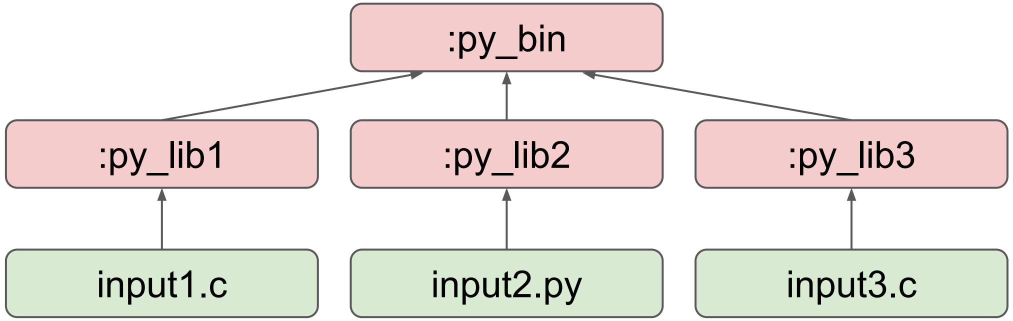


Action graph now has a dynamic fragment, with static boundaries.

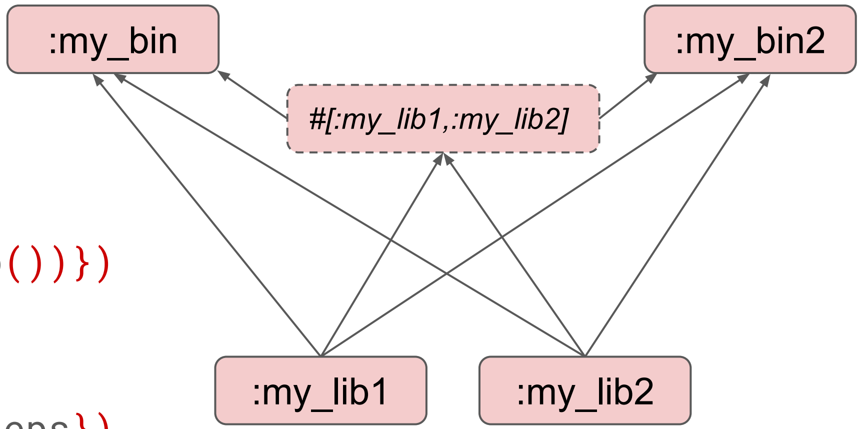
Problem 2: Python Omnibus

Python can depend on C/C++ libraries. Much faster if these libraries are linked together in one lump. The full set of transitively dependent C++ libraries is only visible to the binary. What to do?

1. Link them together separately each time. Lots of duplicated work.
2. Make the user specify sets of libraries to pre-link. Hard to keep correct, requires detailed knowledge of hidden data (what has C++ dependencies).
3. Add more powerful features.



Solution 2: Anon targets



```
omnibus = anon_rule(impl = …,
    attrs = {"deps": attrs.list(attrs.dep())})


out = anon_target(omnibus, {"deps": cpp_deps})
run("python_link …", out.artifact("so"), "output.par")
```

Target graph gains new nodes for additional sharing.
But these nodes don’t expose any additional data.

What we added to the action graph

	Dynamic outputs	Anon targets
Parallelism	✅✅	
Incrementality	✅✅	✅
Sharing	✅	✅✅
Information	New from command results	Hiding target information
Uses in prelude	15 results / 8 files	8 results / 8 files

Problem 3: Is there a third problem?

We haven’t found one.

How would we prove there is not a third problem?

Introducing Buck2 - https://buck2.build

- A build system
- Developed and used by Meta
- Supports many languages (C++, Rust, Python, Go, OCaml, Erlang…)
- Designed for large mono repos
- Open source
- Remote execution
- 2x as fast as Buck1 😎
- Has powerful action graph features

Conclusion

- Large scale build systems (Buck1, Bazel, Buck2) have two graphs - target and action graph
- Target graph is static and user written (constant-static?)
    - Important for CI concerns
- Action graph must refine the target graph with operational concerns
    - Can be dynamic (based on command results) but *not* reach out beyond the target graph
- Can use Build systems à la carte to help design a build system

Perhaps try Buck2: https://buck2.build/