Modeling Compiler Dependencies in Spack

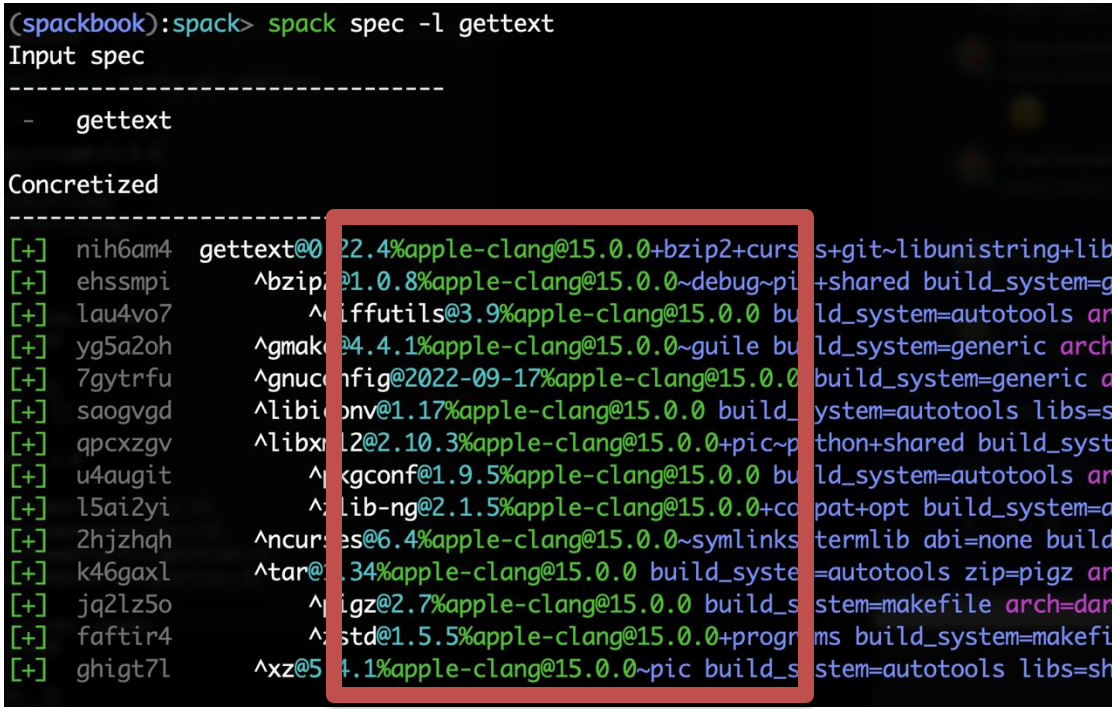
2024 Scalable Tools Workshop

Todd Gamblin
Lawrence Livermore National Laboratory

THE LINUX FOUNDATION

Compilers in Spack are a node attribute, not a node

- Each node is assumed to have a compiler
  - Doesn't make sense for most python/ruby/etc. packages
  - Extra meaningless metadata
- Can't leverage virtual package concept
  - depends_on("c")
  - depends_on("cxx@17")
  - depends_on("fortran@95")
- "Compiler" metadata is just name/version
  - Doesn't include all the needed provenance
  - Mixing of C/C++ and Fortran means you have to lie



Why aren't compilers proper dependencies?

They should be, but…

1. We wanted to mix compilers in one DAG
   - Spack's original dependency model required *only* one version of a package in a DAG
2. We needed to auto-detect vendor compilers
   - Often required for fastest builds
   - Needed an expedient way to use what's available
3. Modeling compiler compatibility is *hard*

```
> spack compilers
==> Available compilers
-- apple-clang sonoma-aarch64 ---------------------
apple-clang@15.0.0

-- gcc sonoma-aarch64 -----------------------------
gcc@13.2.0
```

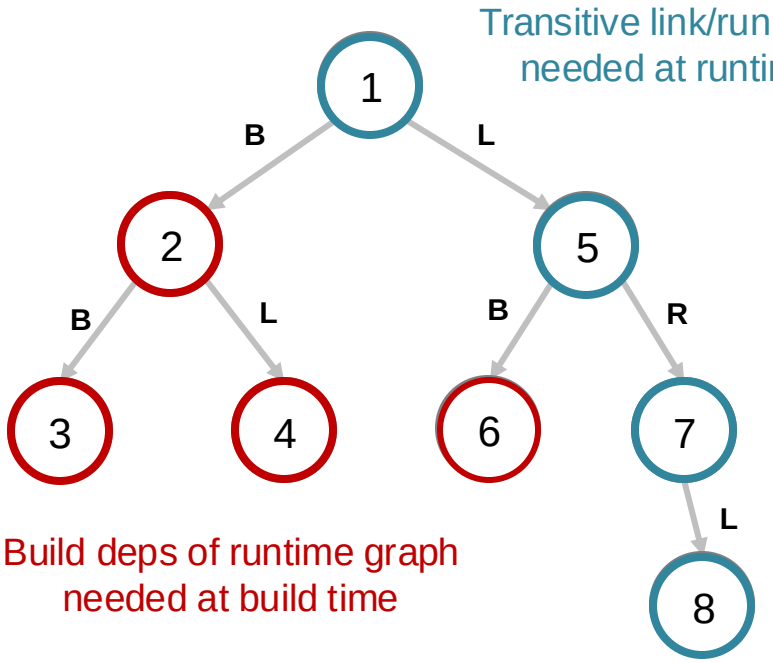
compilers.yaml

```yaml
compilers:
- compiler:
    spec: apple-clang@=15.0.0
    paths:
      cc: /usr/bin/clang
      cxx: /usr/bin/clang++
      f77: null
      fc: null
    flags: {}
    operating_system: sonoma
    target: aarch64
    modules: []
    environment: {}
    extra_rpaths: []
- compiler:
    spec: gcc@=13.2.0
    paths:
      cc: /opt/homebrew/bin/gcc-13
      cxx: /opt/homebrew/bin/g++-13
      f77: /opt/homebrew/bin/gfortran-13
      fc: /opt/homebrew/bin/gfortran-13
    flags: {}
    operating_system: sonoma
    target: aarch64
    modules: []
    environment: {}
    extra_rpaths: []
```

3 kinds of dependencies in Spack

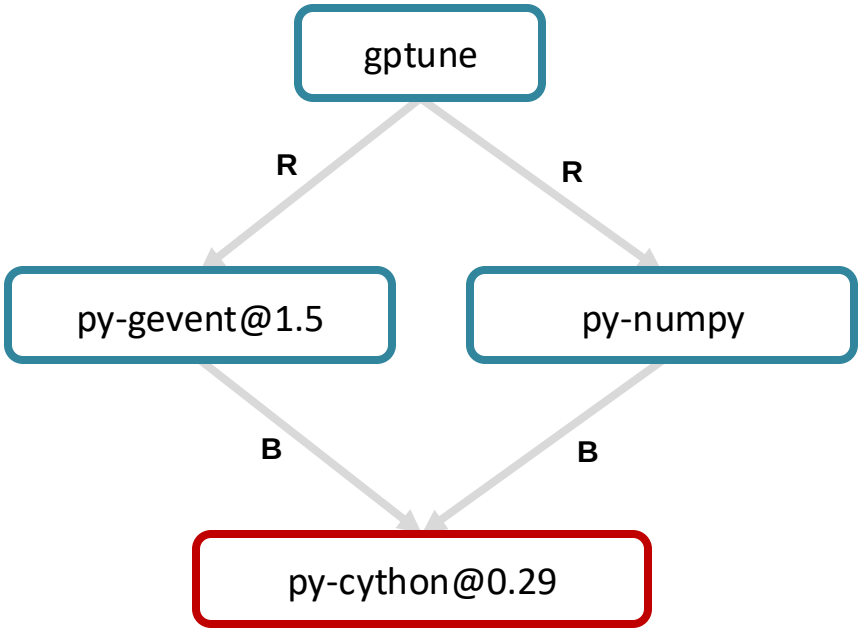
Approximate meanings:

- **build** dependency
  - program needed at build time
  - added to PATH at build time
- **link** dependency
  - library needed at runtime
  - RPATHs are added to dependents
- **run** dependency
  - program needed at runtime
  - added to PATH at runtime



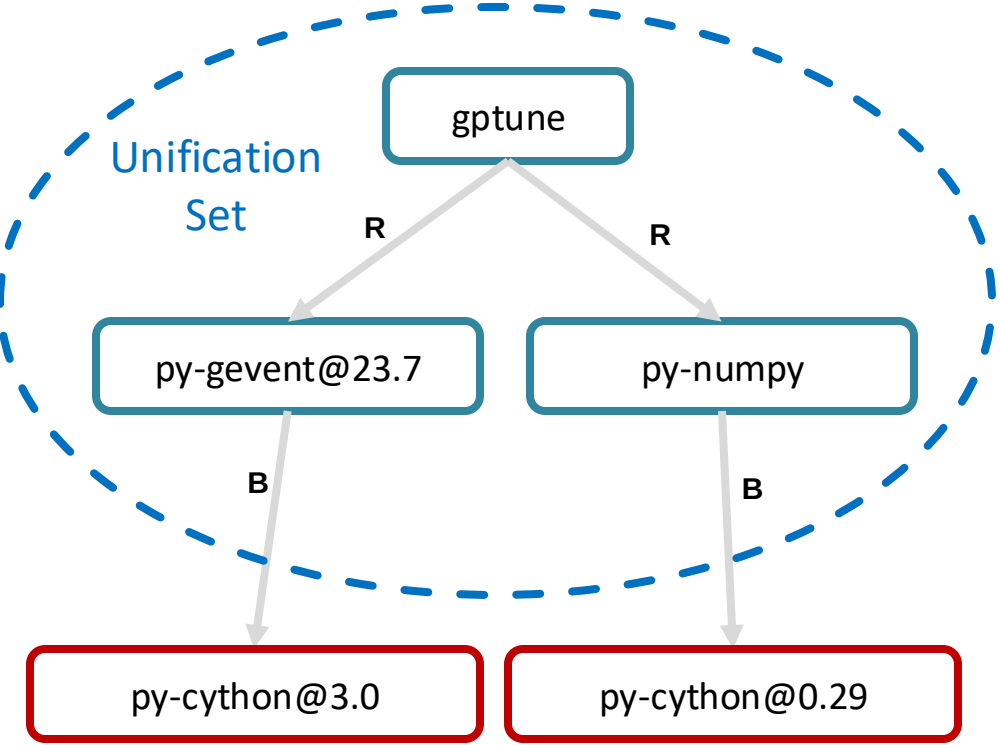
B: build **L**: link **R**: run

Unification can cause issues



- Only one configuration per package allowed in the DAG
  - Ensures ABI compatibility
  - Too restrictive
- In the example py-numpy needs to use py-cython@0.29 as a build tool
- That forces us to use an old py-gevent, because newer versions of py-gevent depend on py-cython@3.0 or greater

We added a notion of unification sets to our solve



- The constraint on build dependencies can be relaxed, without compromising ABI
- Single node constraint now only enforced *within unification sets*
  - These are the set of nodes used together at runtime
- Now we can have two py-cython nodes
  - This allows us to use the latest version of py-gevent

We reworked the solver to create “duplicate” nodes when needed

Original: deduce a single dependency node by name:

```
node(DependencyName)
   :- dependency_holds(PkgName, DependencyName)
```

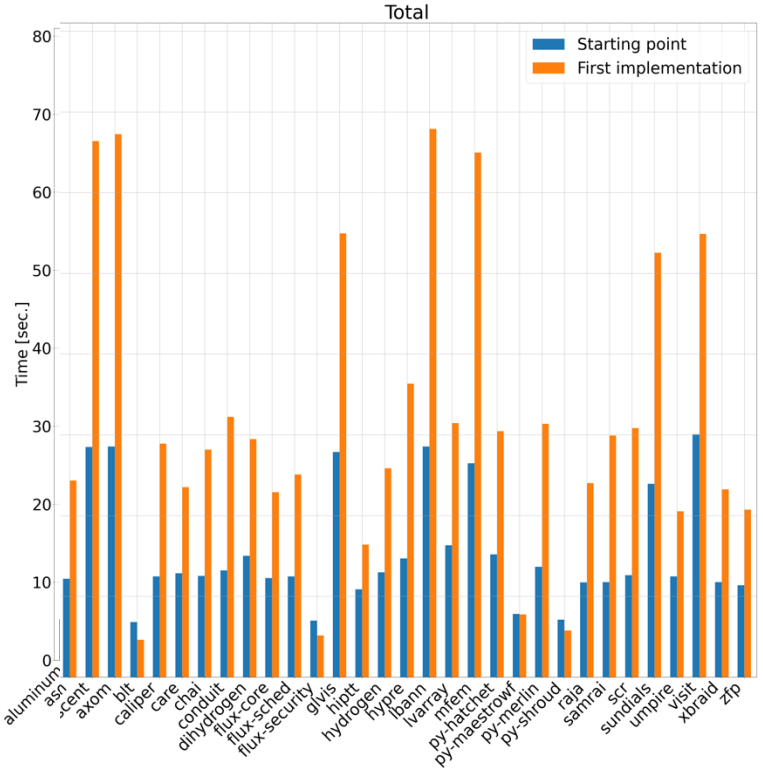
Reworked to **choose** from duplicate nodes:

Convert node identifier from **name** to **(id, name)**

```
1 {
   depends_on(PkgNode, node(0..N-1, DepNode), Type)
   : max_dupes(DepNode, N)
} 1
   :- dependency_holds(PkgNode, DepNode).
```

Limit number of duplicates

First try at allowing duplicates in a single solve



Increased solve times by >> 2x in some cases

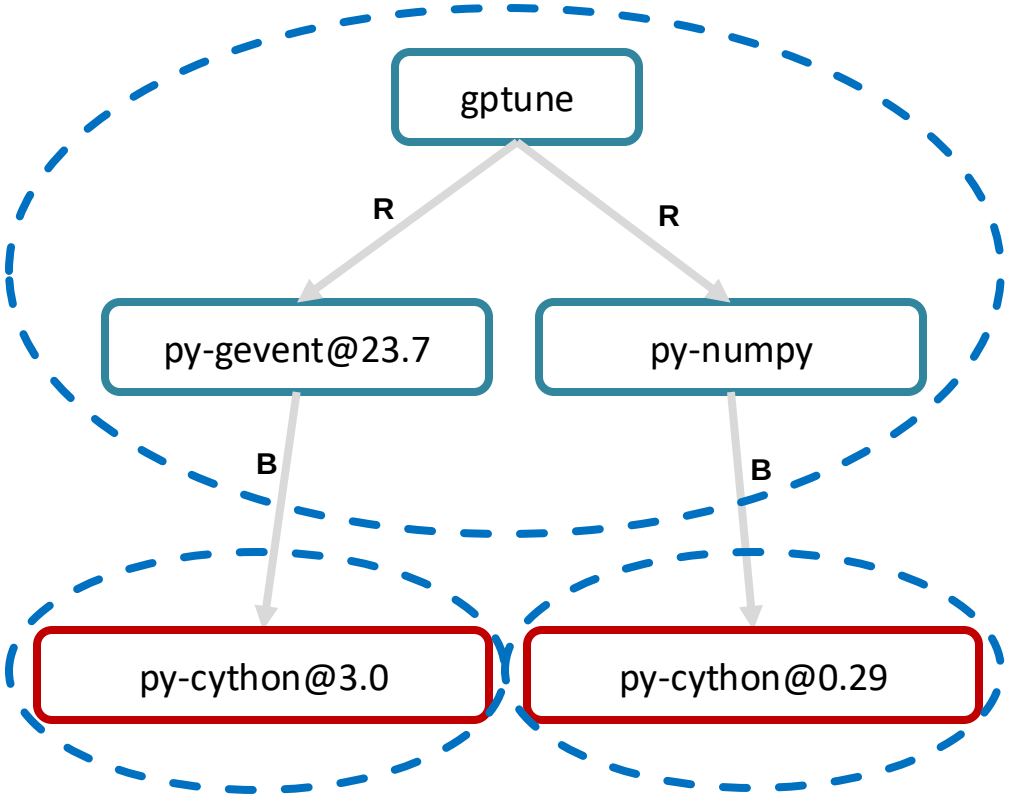
It turns out that cycle detection in the solver is *expensive*

```
path(A, B) :- depends_on(A, B).
path(A, C) :- path(A, B), depends_on(B, C).

:- path(A, B), path(B, A).   % this says "no cycles"
```

- Has to maintain path() predicate representing paths between nodes
- Cycles are actually rare in solutions
  - Switched to post-processing for cycle detection, only pay if there are cycles
  - Later found a #edge feature in clingo – nearly free cycle detection integrated with solver
- Similar issue arose for variant propagation in graph
  - Fixed by reworking variant propagation not to track paths

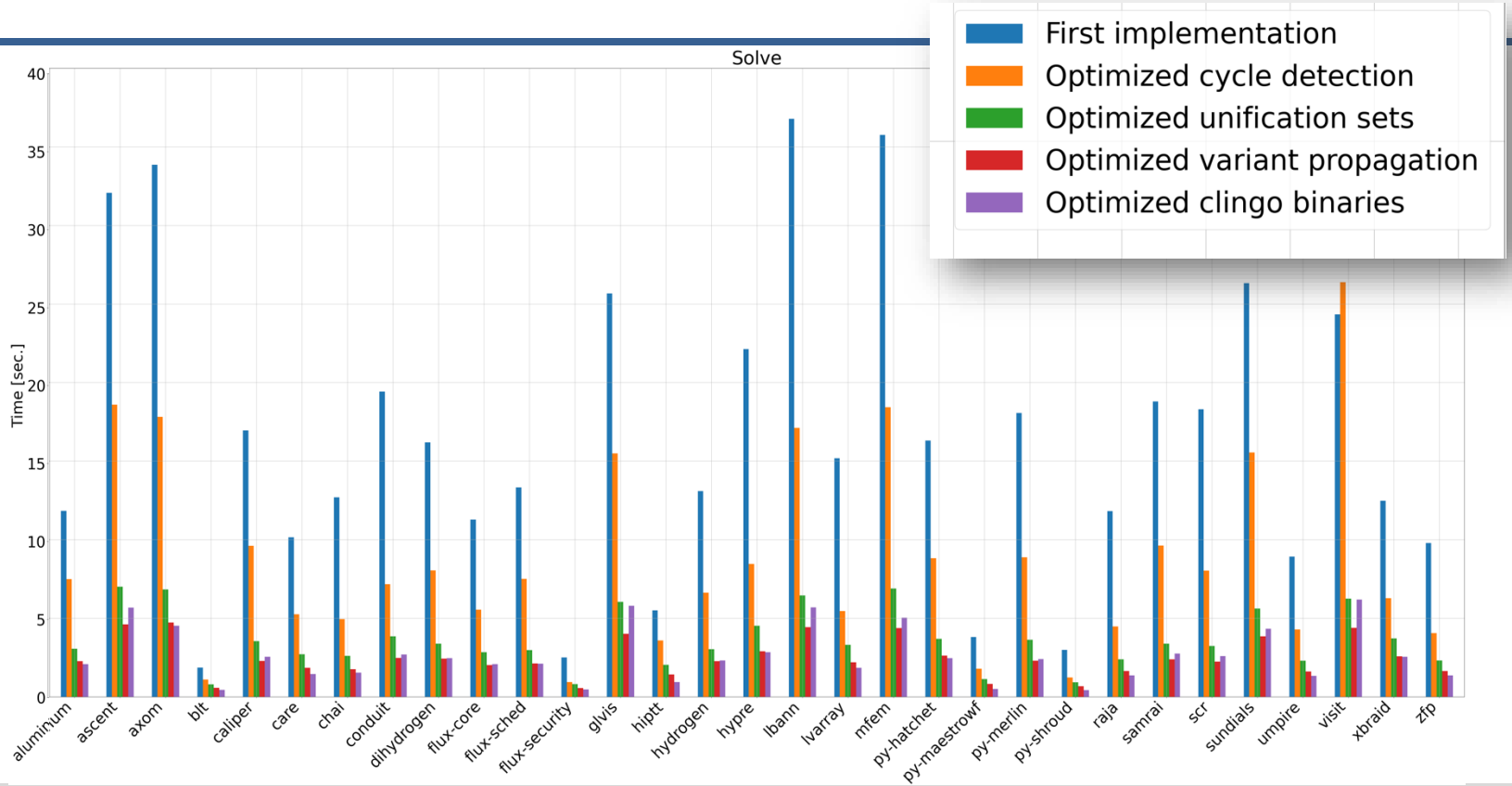
50%+ improvement in solve time

Fully general unification sets can be expensive

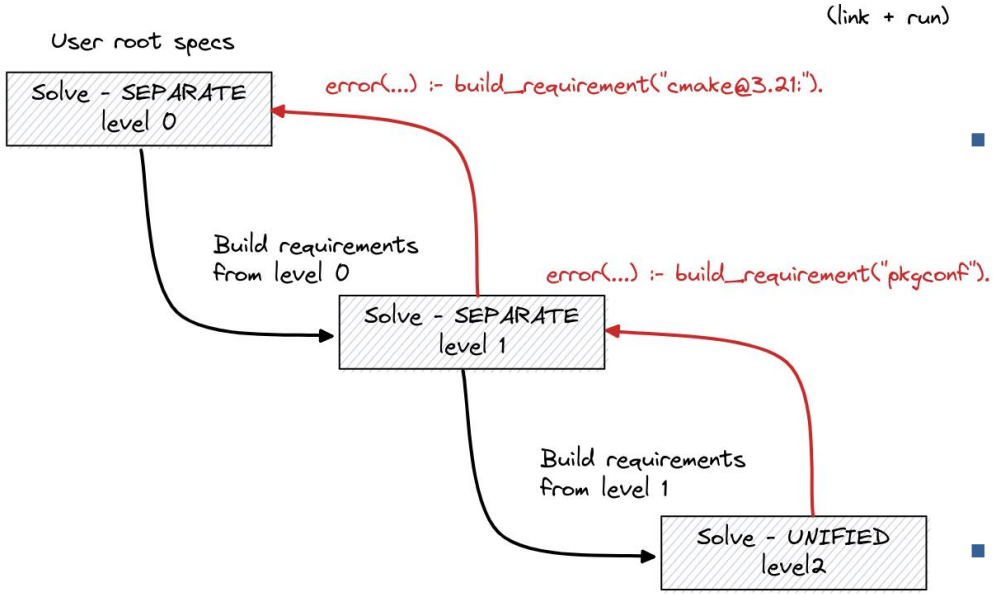


- Unification set creation was originally recursive for *every* build dependency
  - Blows up when grounding
- Mitigation:
  - For now, only create new sets for explicitly *marked* build tools
  - Transitive build dependencies that are not from marked build tools go into a single *common* unification set
  - Eliminate blowup by bounding recursion
- For full generality, need better heuristics to split judiciously

Solve Time Optimizations



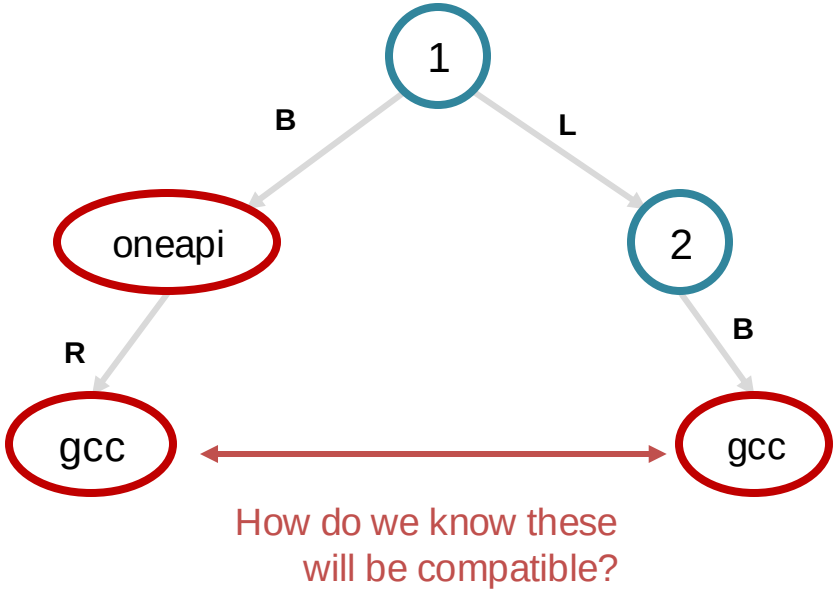
It was not trivial to come up with this model



- In addition to this “coupled” method, we tried an iterative version with multiple solves
- Multiple solves had some disadvantages:
  - Slower due to overhead of multiple solves
  - Not coupled, so feedback from solve to solve was awkward
  - Packagers needed to “help” the solver (impractical)
- Coupling is important for compilers b/c build environment *can* affect run environment

Mixed build dependencies enable us to model compilers as dependencies

- Suppose we build a simple C++ package with the oneapi compiler
  - Model it as a build dependency
  - Now we're allowed to mix compilers
- Build dependencies enable us to model oenapi's dependency on gcc
  - needs gcc to get a libstdc++
  - Wasn't represented with node attribute model
- Suppose we also need to link against another package that uses gcc
  - How do we know the runtime libraries are compatible?

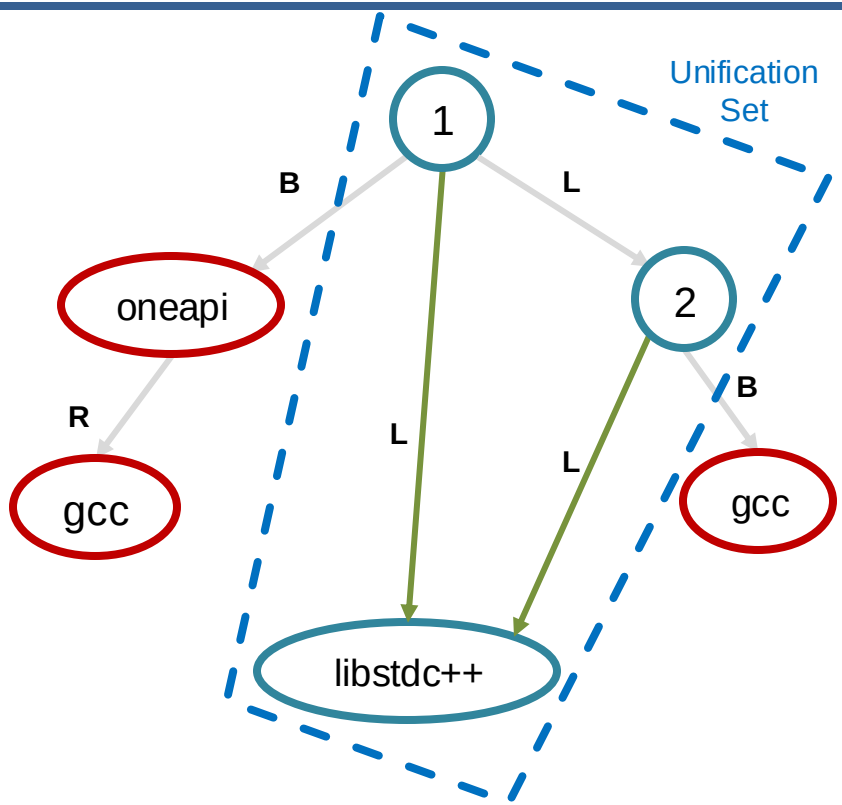


What is a compiler anyway?

- A compiler is **a build dependency**
- A compiler also ***imposes*** **link dependencies on its build-dependent**
  - Compilers actually have *hidden* dependencies
- Runtime libraries have the same runtime semantics as regular libraries
  - Loaded by the same ld.so
  - We need to model them like regular libraries

New compiler dependency model

- Model runtime libraries as link dependencies
- Unification set enforces compatibility
  - For the observant: it's a little more subtle
  - We relax this a bit to allow dependents to use a *newer* libstdc++ than dependencies (v0.22.1)
- This model gets us most of what we want
- Problems:
  - Do 1 and 2 need to know they depend on libstdc++?
  - Isn't that just a compiler implementation detail?
    - e.g., there is libc++, too



Spack v0.22.0 was released in May

Highlights:

1. Compiler runtime dependencies
   - gcc-runtime, intel-oneapi-runtime, libgfortran, libc
   - OS compatibility on linux now uses libc version, not OS tag
2. Improved spack find UI for Environments
3. Improved command-line string quoting
4. Revert default spack install behavior to --reuse
5. More control over reused specs
6. New redistribute() directive
7. New conflict: and prefer: syntax for package preferences
8. include_concrete: in environments
9. python-venv isolation

github.com/spack/spack

Full release notes:
https://github.com/spack/spack/releases/tag/v0.22.0

We've made a lot of progress on compiler dependencies

- Compiler *runtime libraries* represented in the graph
  - C++, Fortran runtimes
- libc is now represented in dependency graphs on Linux
  - No more need to rely on OS tag for compatibility information
- Reuse binaries *without* their compiler needing to be configured locally
- Improved buildcache hit rate using libraries for compatibility

Packages now depend on languages

- Languages are *almost* virtuals
  - HDF5 package depends on cxx and fortran
- Handled specially internally
  - Solver has a bit of hard-coding for language virtuals
  - When compilers are proper nodes we'll make them regular virtuals

```
class Hdf5(CMakePackage):
    """HDF5 is a data model, library, and file format for storing and managi
    data. It supports an unlimited variety of datatypes, and is designed for
    flexible and efficient I/O and for high volume and complex data.
    """

    homepage = "https://portal.hdfgroup.org"
    url = "https://support.hdfgroup.org/ftp/HDF5/releases/hdf5-1.14/hdf5-1.1
    list_url = "https://support.hdfgroup.org/ftp/HDF5/releases"
    list_depth = 3
    git = "https://github.com/HDFGroup/hdf5.git"
    maintainers("lrknox", "brtnfld", "byrnHDF", "gheber", "hyoklee", "lkurz"

    tags = ["e4s", "windows"]
    executables = ["^h5cc$", "^h5pcc$"]

    test_requires_compiler = True

    license("custom")

    depends_on("cxx", type="build", when="+cxx")
    depends_on("fortran", type="build", when="+fortran")
```

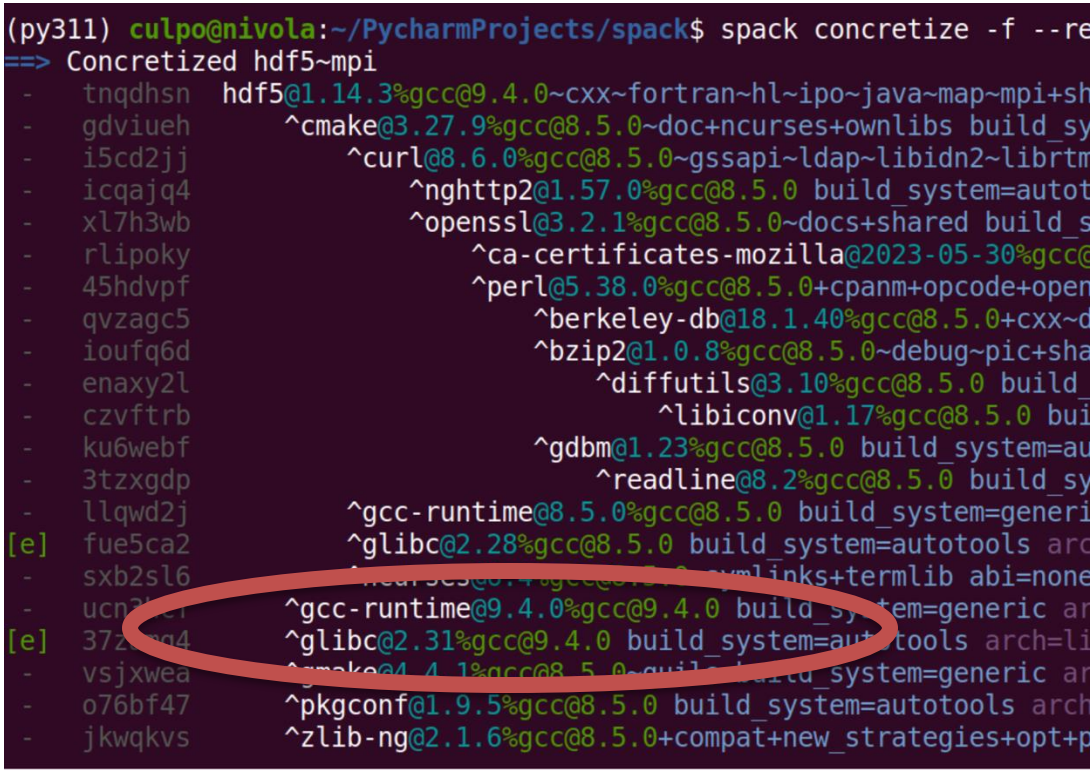
Compilers now model their own runtimes

- Gcc package *provides* cxx virtual
  - Can use this for openmp as well
  - Done for intel-oneapi and gcc
- runtime_constraints method
  - adds global rules to solver
  - pkg object works much like rest of Spack DSL, but allows “*” for “any”
- Right separation of concerns
  - *Compiler* knows about own runtimes and can force dependencies
  - Hidden behind a virtual
  - Packages only depend on virtual
- Any package could be a compiler now
  - Likely useful for tools to inject libs

```
class Gcc(AutotoolsPackage, GNUMirrorPackage, CompilerPackage):

    provides("cxx")

    @classmethod
    def runtime_constraints(cls, *, spec, pkg):
        pkg("*").depends_on(
            "gcc-runtime",
            when="%gcc",
            type="link",
            description="If any package uses %gcc,
            it depends on gcc-runtime",
        )

        pkg("*").depends_on(
            f"gcc-runtime@{str(spec.version)}:",
            when=f"%{str(spec)}",
            type="link",
            description=f"If any package uses %{str(spec)}, "
            f"it depends on gcc-runtime@{str(spec.version)}:",
        )
```

We've (finally) modeled libc as a dependency

- libc is a virtual
  - glibc and musl packages are providers
  - (nearly) every graph has libc in it, via the compiler
  - Can be external or built by Spack
- We are *not* building libc for every install
  - Automatically detect system libc version
  - Add a node to the graph to be used for binary compatibility
- No longer using OS tags for buildcaches
  - Now use libc for this
  - *many* more buildcache hits



Libc modeling makes for a much better buildcache experience

- Currently on develop (emacs 100% from binary):

```
(py311) culpo@nivola:~/PycharmProjects/spack$ spack install emacs
[+] /usr (external glibc-2.17-2hhcy7kzv3wlfqcascwhvup4uysp4hoy)
[+] /home/culpo/PycharmProjects/spack/opt/spack/linux-centos7-x86_64_v3/gcc-10.2.1/gcc-runtime-10.2.1-4gmidou73wvttvhlun564olcopuijl2i
[+] /home/culpo/PycharmProjects/spack/opt/spack/linux-centos7-x86_64_v3/gcc-10.2.1/gmp-6.2.1-nkhm7cmp6samsykyksuqda77xbxnibfn
[+] /home/culpo/PycharmProjects/spack/opt/spack/linux-centos7-x86_64_v3/gcc-10.2.1/berkeley-db-18.1.40-thoi4z7lozgaednxhjd4ojhw2lcsgo5g
[+] /home/culpo/PycharmProjects/spack/opt/spack/linux-centos7-x86_64_v3/gcc-10.2.1/gmake-4.4.1-ucqstlhik7pmv5ijyckmr6mxv46vgeuj
==> Installing nasm-2.15.05-e2kvtqiaava2osr3jbyptdufjgov4dgc [6/39]
==> Fetching https://binaries.spack.io/develop/developer-tools-manylinux2014/build_cache/linux-centos7-x86_64_v3-gcc-10.2.1-nasm-2.15.05-e2kvtqiaava2osr3jbyptdufjgov4dgc.spec.json.sig
gpg: Signature made mar 23 apr 2024, 14:38:11 CEST
gpg:                using RSA key D2C7EB3F2B05FA86590D293C04001B2E3DB0C723
gpg: Good signature from "Spack Project Official Binaries <maintainers@spack.io>" [ultimate]
==> Fetching https://binaries.spack.io/develop/developer-tools-manylinux2014/build_cache/linux-centos7-x86_64_v3/gcc-10.2.1/nasm-2.15.05/linux-centos7-x86_64_v3-gcc-10.2.1-nasm-2.15.05-e2kvtqiaava2osr3jbyptd
==> Extracting nasm-2.15.05-e2kvtqiaava2osr3jbyptdufjgov4dgc from binary cache
==> nasm: Successfully installed nasm-2.15.05-e2kvtqiaava2osr3jbyptdufjgov4dgc
  Search: 0.00s.  Fetch: 2.61s.  Install: 0.17s.  Extract: 0.11s.  Relocate: 0.04s.  Total: 2.78s
[+] /home/culpo/PycharmProjects/spack/opt/spack/linux-centos7-x86_64_v3/gcc-10.2.1/nasm-2.15.05-e2kvtqiaava2osr3jbyptdufjgov4dgc
==> Installing pcre-8.45-xi42ks7asbq2sqmdc7bdsmhzzhlie6m4 [7/39]
==> Fetching https://binaries.spack.io/develop/developer-tools-manylinux2014/build_cache/linux-centos7-x86_64_v3-gcc-10.2.1-pcre-8.45-xi42ks7asbq2sqmdc7bdsmhzzhlie6m4.spec.json.sig
gpg: Signature made mar 23 apr 2024, 14:38:12 CEST
gpg:                using RSA key D2C7EB3F2B05FA86590D293C04001B2E3DB0C723
gpg: Good signature from "Spack Project Official Binaries <maintainers@spack.io>" [ultimate]
==> Fetching https://binaries.spack.io/develop/developer-tools-manylinux2014/build_cache/linux-centos7-x86_64_v3/gcc-10.2.1/pcre-8.45/linux-centos7-x86_64_v3-gcc-10.2.1-pcre-8.45-xi42ks7asbq2sqmdc7bdsmhzzhli
==> Extracting pcre-8.45-xi42ks7asbq2sqmdc7bdsmhzzhlie6m4 from binary cache
==> pcre: Successfully installed pcre-8.45-xi42ks7asbq2sqmdc7bdsmhzzhlie6m4
  Search: 0.00s.  Fetch: 2.34s.  Install: 0.18s.  Extract: 0.14s.  Relocate: 0.03s.  Total: 2.52s
[+] /home/culpo/PycharmProjects/spack/opt/spack/linux-centos7-x86_64_v3/gcc-10.2.1/pcre-8.45-xi42ks7asbq2sqmdc7bdsmhzzhlie6m4
==> Installing tree-sitter-0.22.2-x5wmc6tind5qxo2zrswb7vyj3xuhak2d [8/39]
==> Fetching https://binaries.spack.io/develop/developer-tools-manylinux2014/build_cache/linux-centos7-x86_64_v3-gcc-10.2.1-tree-sitter-0.22.2-x5wmc6tind5qxo2zrswb7vyj3xuhak2d.spec.json.sig
gpg: Signature made mar 23 apr 2024, 14:38:22 CEST
gpg:                using RSA key D2C7EB3F2B05FA86590D293C04001B2E3DB0C723
gpg: Good signature from "Spack Project Official Binaries <maintainers@spack.io>" [ultimate]
==> Fetching https://binaries.spack.io/develop/developer-tools-manylinux2014/build_cache/linux-centos7-x86_64_v3/gcc-10.2.1/tree-sitter-0.22.2/linux-centos7-x86_64_v3-gcc-10.2.1-tree-sitter-0.22.2-x5wmc6tind
==> Extracting tree-sitter-0.22.2-x5wmc6tind5qxo2zrswb7vyj3xuhak2d from binary cache
==> tree-sitter: Successfully installed tree-sitter-0.22.2-x5wmc6tind5qxo2zrswb7vyj3xuhak2d
```

Backward-compatible syntax is a bit tricky

- We generalized semantics for **^** and **%**:

  ^zlib *transitive* link/run dependency on zlib

  %gcc *direct* dependency on gcc

root +foo ^dep @2.0 %gcc@12 +bar

@2.0 and +bar associate with dep

- % can now be used for more than compilers
  - e.g., %cmake@3.28 for cmake versions
- Historically, only **^** starts a new associative context
  - *Cannot* make **%** behave exactly like **^**
  - Users frequently write @version %compiler@version +variants

root +foo ^dep @2.0 %{gcc@12 +bar}

Now +bar associates with gcc

- Introducing **{}** to dependency specs so we can add variants
  - Also helps with ambiguity introduced by node splitting
  - “the c compiler used to build the gcc used to build root”

root %{gcc %{c}}

What's left

1. Remove the old concretizer (finally done)
   - Bootstrap now handled with some simple JSON templates for clingo
2. Semantics of % will change
   - ^gcc will mean "transitive link or run dependency on zlib"
   - %gcc will mean "direct dependency on gcc"
3. compilers.yaml → packages.yaml
4. Make compilers into nodes

Roadmap for this year

1. Finish compiler dependencies
   - Compilers will appear as build dependencies
   - All special compiler logic generified for any build dependency
   - Enable bootstrapping compilers from buildcache
   - Continue to flesh out low-level compiler runtime libraries
2. Harden public build caches and make them more broadly compatible
3. Enable bare-metal MPI/CUDA/ROCm installations
4. Make the build cache on by default; build only what we need from source
5. Automatic Python package generation
6. Continue to improve Windows support
7. Speed improvements for concretization and metadata management

Lots of exciting work ahead!

Disclaimer

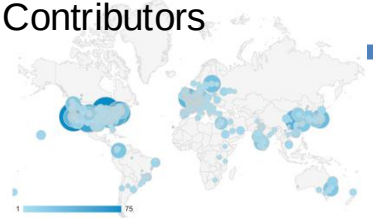
This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

THE LINUX FOUNDATION

The concretizer includes information from packages, configuration, and CLI

Dependency solving is NP-hard

Contributors



- new versions
- new dependencies
- new constraints

package.py repository

concretizer

spack developers

yaml default config packages.yaml

admins, users

yaml local preferences config packages.yaml

users

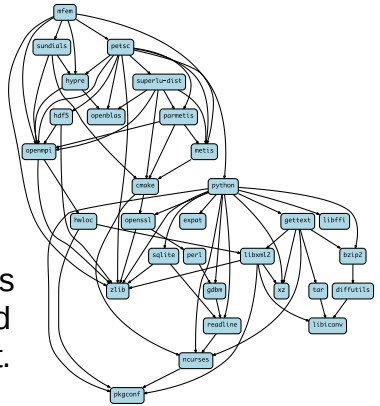
yaml local environment config spack.yaml

users

Command line constraints

spack install hdf5@1.12.0 +debug

Concrete spec is fully constrained and can be built.



Some stats on problem sizes

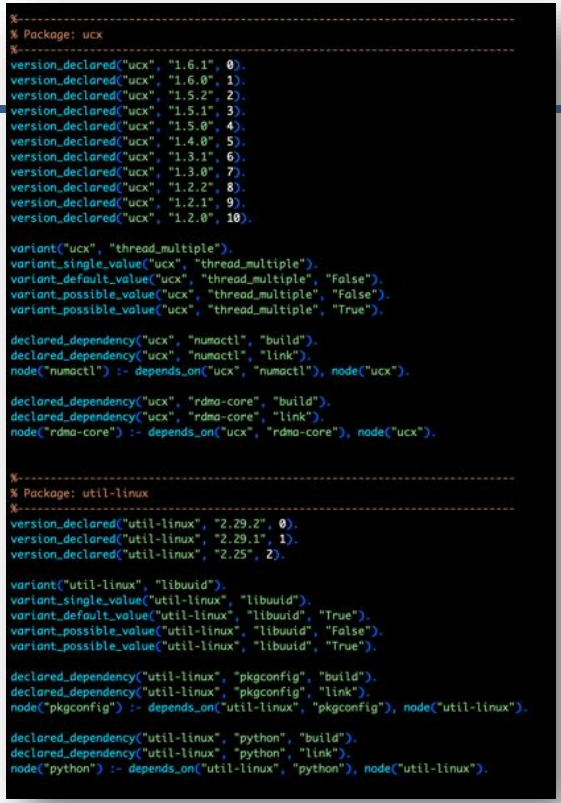
- Main logic program is:
  - ~250 rules
  - 20 optimization criteria
  - 933 lines of ASP code

- Problem instances can vary quite a bit
  - Common dependencies get us some magic numbers
  - gmake's optional dependency on guile makes most solves consider at least 527 packages
  - gnuconfig is notably very simple ☺

Package	Possible dependencies	Facts
gnuconfig	1	150
zlib	527	30,095
gmake	527	30,160
openmpi	527	109,021
qt	527	109,029
trilinos	694	224,142
root	699	146,372
mfem	714	273,078
r-condop	774	142,212
warpx	819	319,374
exawind	820	322,535

We reimplemented Spack's concretizer using *Answer Set Programming*

- Was originally a greedy, custom Python algorithm
- Answer Set Programming is a *declarative* programming paradigm
  - Looks like Prolog
  - Built around modern CDCL SAT solver techniques
  - Designed for combinatorial search problems
- ASP program has 2 parts:
  1. Large list of facts generated from package recipes (problem instance)
     - 60k+ facts is typical – includes dependencies, options, etc.
  2. Small logic program (~700 lines of ASP code)
- Algorithm (the part we write) is conceptually simpler:
  - Generate facts for all possible dependencies
  - Send facts and our logic program to the solver
  - Rebuild a DAG from the results
- We're using **Clingo**, the Potassco grounder/solver package



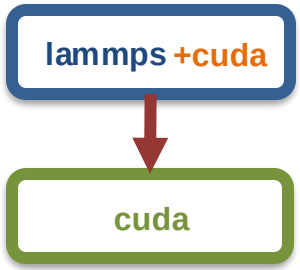
Some facts for HDF5 package

Spack's concretizer is implemented using Answer Set Programming (ASP)

ASP looks like Prolog but is converted to SAT with optimization

Facts describe the graph

```
node("lammps").
node("cuda").
variant_value("lammps", "cuda", "True").
depends_on("lammps", "cuda").
```



First-order rules (with variables) describe how to resolve nodes and metadata

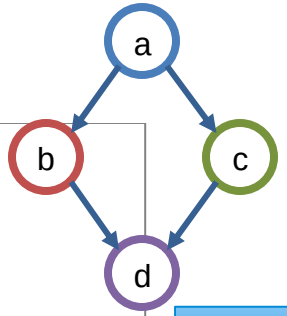
```
node(Dependency) :- node(Package), depends_on(Package, Dependency).
```

```
node("mpi")
```

```
node("hdf5").
depends_on("hdf5", "mpi").
```

Ground Rule

Grounding converts a first-order logic program into a propositional logic program, which can be solved for stable models




```
depends_on(a, b).
depends_on(a, c).
depends_on(b, d).
depends_on(c, d).

node(Dep)
  :- node(Pkg),
     depends_on(Pkg, Dep).

% at least one is true
1 { node(a); node(b) }.
```

First-order Logic Program

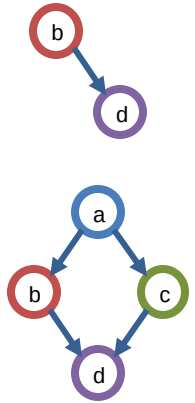
Ground

```
depends_on(a, b).
depends_on(a, c).
depends_on(b, d).
depends_on(c, d).

node(b) :- node(a).
node(c) :- node(a).
node(d) :- node(c).
node(d) :- node(b).

% at least one is true
1 { node(a); node(b) }.
```

Propositional Program

Solve

```
Answer 1:
node(b)
node(d)

Answer 2:
node(a)
node(b)
node(c)
node(d)
```



Stable Models (Answer Sets)

Answer 1: Only node(b) is true
Answer 2: Both node(a) and node(b) are true

ASP searches for *stable models* of the input program

- Stable models are also called ***answer sets***
- A ***stable model*** (loosely) is a set of true atoms that can be deduced from the inputs, where every rule is idempotent.
  - Similar to fixpoints
  - Put more simply: a *set of atoms where all your rules are true!*
- Unlike Prolog:
  - Stable models contain everything that can be derived (vs. just querying values)
  - Good ways to do optimization to select the “best” stable model
  - ASP is guaranteed to complete!

When would we go to “Version 1.0”?

Big things we’ve wanted for 1.0 are:

- New concretizer — **Done!**
- production CI — **Done!**
- production public build cache — **Done!**
- Compilers as dependencies — **Aiming for November**
- Buildcache hardening — **Aiming for November**
- Stable package API — **Aiming for June**
  - Enables separate package repository

We are getting very close!