



Package Managers à la Carte

A Formal Model of Dependency Resolution

RYAN T. GIBB, University of Cambridge, United Kingdom

PATRICK FERRIS, University of Cambridge, United Kingdom

DAVID ALLSOPP, Jane Street, United Kingdom

THOMAS GAZAGNAIRE, Tarides, France

ANIL MADHAVAPEDDY, University of Cambridge, United Kingdom

Package managers are legion. Every programming language and operating system has its own solution, each with subtly different semantics for dependency resolution. This fragmentation prevents multilingual projects from expressing precise dependencies across language ecosystems; it leaves external system dependencies implicit and unversioned; and it obscures the full dependency graph that supply-chain analysis depends on. We present the *Package Calculus*, a formalism for dependency resolution that unifies the core semantics of package managers. Through a series of formal reductions, we show how this core is expressive enough to model the diversity of real-world dependency expression languages. The calculus provides the theoretical foundation for future cross-ecosystem tooling, as a *lingua franca* of dependency expression.

CCS Concepts: • **Software and its engineering** → **Software libraries and repositories**; • **Theory of computation** → *Constraint and logic programming*; Problems, reductions and completeness.

Additional Key Words and Phrases: package managers, dependency resolution, software ecosystems

ACM Reference Format:

Ryan T. Gibb, Patrick Ferris, David Allsopp, Thomas Gazagnaire, and Anil Madhavapeddy. 2026. Package Managers à la Carte: A Formal Model of Dependency Resolution. *Proc. ACM Program. Lang.* 10, ICFP, Article 301 (August 2026), 35 pages. <https://doi.org/10.1145/3828699>

1 Introduction

Package managers are numerous and diverse; despite all fundamentally solving the same problem, they remain ad hoc and non-interoperable. Consider a machine-learning project on Debian Linux, using a combination of C, Rust, and OCaml code for a static binary, with dynamic Python bindings that use GPU drivers, which in turn depend on a particular kernel driver. Four separate package managers are required to deploy this program: opam (OCaml), Cargo (Rust), pip (Python), and APT (Debian). If the program is to be portable beyond Debian, even more are required: APK (Alpine) and DNF (Red Hat-based Linux). Dependencies between these ecosystems are managed ad hoc: they are typically unversioned, duplicated across ecosystems, and consequently invisible to security tooling. Yet all these package managers share a common core: from dependencies written in a domain-specific language (DSL) [10], they *resolve* a set of package versions that transitively satisfies them – a *resolution* – and deploy it as a filesystem subtree.

Authors' Contact Information: [Ryan T. Gibb](#), University of Cambridge, Cambridge, United Kingdom, rtg24@cam.ac.uk; [Patrick Ferris](#), University of Cambridge, Cambridge, United Kingdom, pf341@cam.ac.uk; [David Allsopp](#), Jane Street, London, United Kingdom, dallsopp@janestreet.com; [Thomas Gazagnaire](#), Tarides, Paris, France, thomas@tarides.com; [Anil Madhavapeddy](#), University of Cambridge, Cambridge, United Kingdom, avsm2@cam.ac.uk.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2475-1421/2026/8-ART301

<https://doi.org/10.1145/3828699>

In this paper, we define a surprisingly small core calculus as a *lingua franca* of dependency expression, and extensions for ecosystem-specific functionality that reduce back to the core. This makes cross-ecosystem comparison precise, so that superficially different mechanisms can be identified as the same construct, or distinguished where they differ. Each reduction from an extension to the core is a mechanism for emulating the extension in an ecosystem that lacks it natively, formalising workarounds practitioners already apply by hand. We provide the foundation that principled cross-ecosystem tooling – such as polyglot resolvers and translators between ecosystems – has lacked to date. We mechanise in Lean 4 the theorems presented in the paper [33].

We make these specific contributions:

- A survey of over thirty package managers across operating system and programming language ecosystems, identifying a shared core of packages, dependency relations, dependency resolution, and deployment; and axes along which they diverge (§2).
- The *Package Calculus*, a minimal formal system which captures dependency resolution in three conditions – root inclusion, dependency closure, and version uniqueness. We show how the NP-completeness of dependency resolution can be avoided, but is ultimately inherent in capturing the complexity of real-world package managers (§3).
- A model for each axis of divergence in dependency expression as an extension to the core calculus, with a sound and complete reduction from each extension back to the core (§4). We are primarily concerned with satisfiability: the reductions preserve which resolutions are valid, not which are optimal.
- A demonstration that we can compose extensions to model package managers combining several at once, with general considerations for doing so. This opens the path to a *polyglot resolver*: each ecosystem compiles its dependencies into the core and is returned a resolution to deploy – so a project spanning several ecosystems is resolved as one. We further show how the core could serve as an intermediate representation for cross-ecosystem translation (§5).

Package management to date has not unified the semantics of different systems; existing interoperability efforts have focused on sharing solving mechanisms between ecosystems with the same semantics (§6). Instead of seeing package managers as monolithic and unrelated, we tease apart the commonality to show how we can unify the semantics of dependency expression (§7).

2 The Package Management Problem

Package managers all face the same problem: given a project’s dependencies, choose a consistent set of package versions to install – a resolution – and deploy it (§2.1). They underwent a Cambrian explosion in the mid-2000s (Appendix A), and now diverge in the semantics of dependency expression (§2.2). Fig. 1 collects the terminology used throughout the paper.

2.1 A Common Core

We identify concerns common to package managers. The Package Calculus formalises packages, dependencies, and resolutions (§3); deployment affects the semantics of dependency expression (§2.2).

2.1.1 Package. A *package* is a unit of software – source code, a compiled binary, or data – distributed with metadata that declares the other packages it depends on. Packages are identified by a *name*, which refers to a package across its releases, and a *version*, distinguishing one release from another.

2.1.2 Dependency. Package managers express *dependency* relations from a *depender* package to a set of *dependee* packages, meaning the depender requires one of the dependees. Dependees are often software libraries, while packages that are not dependees are typically applications intended for use by an end user. Dependees are typically expressed as a name and set of compatible versions.

GLOSSARY

Package: a unit of software – code, binaries, or data – identified by a *name* and *version* §2.1.1

Dependency: a relation from a *depender* package, requiring one of a set of *dependee* packages §2.1.2

Resolution: the set of packages that satisfies a package’s dependencies transitively, or the act of selecting such a set, computed by a *resolver* §2.1.3

Deployment: the provision of a resolution, typically as a filesystem subtree of installed packages §2.1.4

Ecosystem: the programming language or operating system a package manager serves §2.2.1

Toolchain integration: the coupling of a package manager with build systems and compilers §2.2.2

Packaging language: the DSL in which package metadata and dependencies are expressed §2.2.3

Version constraint: a dependency on a set of versions of a package name, e.g. $\geq 1 \wedge \leq 3$ §2.2.4

Conflict: an ‘anti-dependency’, preventing co-installation, e.g. of rival forks §2.2.5

Concurrent versions: multiple versions of the same package name coexisting in a resolution §2.2.6

Peer dependency: a child’s constraint on a sibling’s version, both dependees of one parent §2.2.7

Feature: optional functionality of a package that may add dependencies, unified across dependers §2.2.8

Package formula: a Boolean (and/or/not) expression over package dependencies §2.2.9

Variable formula: a package formula parameterised by variables, e.g. the operating system §2.2.10

Virtual package: a package name that multiple packages can provide, e.g. an SSH server §2.2.11

Optional dependency: a use-if-present dependency where the dependee is never required, but used for build ordering if it is in the resolution §2.2.12

Fig. 1. A glossary of package management terminology as used in this paper.

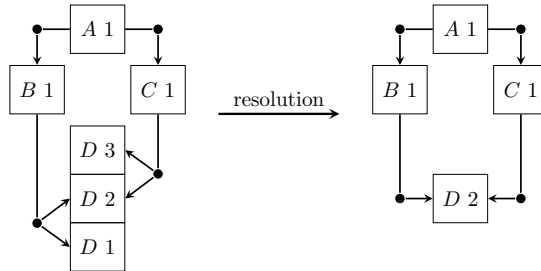


Fig. 2. Resolving a set of dependencies (left) with root package name *A* and version 1 yields a *resolution* – the set of packages that satisfies them – by selecting a single version of each package name (right).

2.1.3 Resolution. Dependency *resolution* is the problem of computing, given a package, the set of package names and versions needed to satisfy its dependencies transitively. A *resolver* performs dependency resolution. A *resolution* is the set of packages selected by *resolving* dependencies. Fig. 2 illustrates a resolution that satisfies three conditions: the root package is included (root inclusion); every package’s dependencies are met by a package in the resolution (dependency closure); and at most one version of each name is chosen (version uniqueness). We formalise these conditions in §3.

2.1.4 Deployment. Given a *resolution*, package managers provide packages through *deployment*. Interpreted-language package managers can place source code in a directory, like npm (JavaScript). Others build packages from metadata instructions, as Cargo, opam, or Cabal (Haskell) do. System package managers might unpack binary archives into the filesystem hierarchy. For embedded deployments, as with BusyBox or Yocto, the result may be a whole firmware image. Deployment is orthogonal to resolution, but its constraints determine the semantics of dependency expression. For example, requiring unique paths or linker symbols per package name forces version uniqueness.

Table 1. A comparison of representative package managers.

Description					Dependency Expression														
Package Manager	Year	Ecosystem	Toolchain ¹	Packaging Language	Version	Conflicts	Concurrent	Peer	Features	Package	Variable	Virtual	Optional						
		§2.2.1	§2.2.2	§2.2.3	Constraints		§2.2.4	Versions		§2.2.6	Deps	§2.2.7	Formulae	§2.2.9	Formulae	§2.2.10	Packages	§2.2.11	Deps
CPAN	1995	Perl	(P)	cpanfile	✓ ²														
pkg_add	1996	OpenBSD	(P)	BSD Make	✓	✓				✓									
CRAN	1997	R	(P)	DESCRIPTION	✓														
APT	1998	Debian Linux	(P)	.deb control	✓	✓				✓	✓	✓							
Portage	2000	Gentoo Linux	(P)	ebuild (Bash)	✓	✓		✓ ³		✓	✓	✓	✓						
pacman	2002	Arch Linux	(P)	PKGBUILD (Bash)	✓	✓											✓		
Nix [26] ⁴	2003	Various	(P) (B)	Nix expressions / .drv				✓											
Zero Install	2003	Various	(P)	feed.xml	✓			✓				✓							
Maven	2004	Java	(P) (B)	pom.xml	✓ ⁵							✓							
Gem	2004	Ruby	(P)	Ruby	✓														
Cabal [13]	2005	Haskell	(P) (B)	.cabal	✓			✓		✓		✓							
APK	2005	Alpine Linux	(P)	APKBUILD (shell)	✓	✓											✓		
tlmgr	2008	T _E X Live	(P)	.tlpsrc															
pip ⁶	2008	Python	(P)	pyproject.toml	✓					✓		✓							
Homebrew	2009	macOS	(P)	Ruby		✓		✓ ⁷				✓							
npm	2010	JavaScript	(P)	package.json	✓			✓	✓			✓							
NuGet	2010	.NET	(P) (B)	.csproj	✓ ⁸							✓							
Chocolatey	2011	Windows	(P)	.nuspec	✓ ⁹														
Composer	2012	PHP	(P)	composer.json	✓	✓											✓		
Conda	2012	Python	(P)	meta.yaml	✓	✓						✓							
pkg ¹⁰	2012	FreeBSD	(P)	BSD Make		✓											✓		
Pub	2012	Dart	(P)	pubspec.yaml	✓														
DNF ¹¹	2013	Red Hat-based Linux	(P)	.spec	✓	✓				✓							✓		
opam [61]	2013	OCaml	(P) ¹²	.opam	✓	✓				✓		✓							✓
pkgman	2013	Haiku OS	(P)	.PackageInfo	✓	✓											✓		
Spack [31]	2014	HPC ¹³	(P)	Python	✓	✓		✓				✓					✓		
Cargo [56]	2014	Rust	(P) (B)	Cargo.toml	✓			✓		✓		✓							
FuseSoC	2014	HDL ¹⁴	(P) (B)	.core (YAML)	✓							✓					✓		
Bazel	2015	Multi-language	(P) (B) ¹⁵	Starlark	MVS			✓											
Go modules [60]	2018	Go	(P) (B) (C)	go.mod	MVS			✓											
Unison [15]	2019	Unison	(P) (B) (C)	Unison				✓											
pack	2022	Idris 2	(P) (B)	TOML + .ipkg															

¹ (P)ackage manager, (B)uild system, and/or (C)ompiler.² CPAN installs the latest version greedily without backtracking.³ Limited support with ‘Slotting’.⁴ Nix uses singular dependencies (Appendix D) and resolves no version constraints, so most columns do not apply.⁵ Maven uses a ‘nearest wins’ heuristic over constraint solving.⁶ Poetry and uv use pyproject.toml’s format and dependencies.⁷ Emulated by hand with per-version formulae such as python@3.11 – the name mangling of §4.2 – for ~2% of packages.⁸ NuGet defaults to lowest version without backtracking.⁹ Chocolatey uses NuGet’s resolution infrastructure.¹⁰ pkg (pkgng) replaced the earlier FreeBSD pkg_* tools.¹¹ DNF replaced YUM as the standard package manager frontend for RPM-based distributions including Fedora.¹² opam supports language-agnostic build scripts.¹³ High-performance computing.¹⁴ Hardware description languages, such as Verilog and VHDL.¹⁵ Bazel is a build system with package management functionality.

2.2 A Survey of Package Managers

Table 1 compares a selection of package managers, chosen to cover the most widely used systems as well as those with distinctive semantics, and assembled from reading official documentation, inspecting source code, and consulting with contributors. We define the table’s columns below.

2.2.1 Ecosystems. Package managers fall into two broad categories: language package managers distribute libraries and tools to developers in a specific programming language; system package managers administer a full operating system. The lines between these often blur; system package managers do distribute language libraries, and language package managers can facilitate setting up a development environment with system dependencies. Almost all package managers group packages in a centralised repository for discoverability and curation, which ranges from validating version constraints (§2.2.4) to maintaining a coherent package set, a single mutually compatible version per name replacing version constraints, like Homebrew (macOS), Nix, T_EX Live, and Stackage (Haskell).

2.2.2 Toolchain Integration. Some package managers, especially language package managers, are integrated with other parts of the toolchain. *opam* includes build scripts in package metadata which can invoke a build system or compiler, but is language-agnostic. Other ecosystems have tighter integration; Cargo is Rust’s package manager *and* build system. This enables functionality impossible without a well-defined API between these parts of the toolchain. For example, Cargo mangles symbols with package versions to support concurrent versions (§2.2.6).

2.2.3 Packaging Language. Packaging DSLs are normally static data structures for expressing dependencies (§2.1.2) – e.g. TOML, .deb control files, or .opam files. Sometimes DSLs are expressions embedded in Turing-complete languages [36] – e.g. Homebrew’s Ruby, Spack’s (high-performance computing) Python, or Portage’s (Gentoo Linux) ebuild scripts – that compile to a static structure. We next cover the varied functionality these DSLs support in expressing dependencies.

2.2.4 Version Constraints. A *version formula* such as $\geq 1 \wedge \leq 3$ enumerates to a set $\{1, 2, 3\}$. Some package managers, such as Nix (Various), can only express a dependency on a single version of a package name. Go’s Minimal Version Selection (MVS) specifies only a minimum version bound for each dependency, and has a linear-time dependency resolution algorithm (§3.3).

2.2.5 Conflicts. A conflict is an ‘anti-dependency’ on the absence of a package name and version set, used, for example, for forks of a project that are not co-installable. Portage distinguishes *weak blocks* (!pkg) from *strong blocks* (!!pkg): weak blocks are non-binding preferences for the resolver, analogous to Debian’s Recommends (§2.2.12), while strong blocks enforce hard mutual exclusion.

2.2.6 Concurrent Versions. Many package managers allow only one version of a package name in a resolution due to deployment constraints such as requiring unique symbols when linking objects, or storing packages at a unique path in the filesystem. Package managers can use isolated environments to work around this, such as *opam* switches or Python virtual environments.

In contrast, package managers with concurrent versions allow multiple versions of a name to exist within a resolution. Cargo uses name mangling to link multiple versions of a library into a binary. Cargo restricts this to packages with different major versions under the semantic versioning scheme: co-linking minor versions would create duplicate, incompatible types in the binary, while different major versions are expected to have distinct APIs [49, 57]. Concurrent versions are therefore isolated – the types of one major version cannot interoperate with another. Nix [26] diverges from the Linux Filesystem Hierarchy Standard to place packages at paths containing unique cryptographic hashes, allowing the installation of multiple versions of a package name on the same system. npm allows duplication of dependee packages by storing them in a subdirectory of their depender.

2.2.7 Peer Dependencies. Given a depender parent and a dependee child, a peer dependency is a constraint that the child places on the version of its ‘peer’ sibling: another dependee of the parent. This is typically used by plugins; for example, a React component library (the child) declares a peer dependency on react (the sibling). It does not provide its own copy, but instead requires that its parent provide a compatible version. This functionality is only required with concurrent versions (§2.2.6), as otherwise the child could simply depend on the peer directly.

npm supports peer dependencies, but their behaviour has changed multiple times. The `--legacy-peer-deps` behaviour is that a peer dependee is only included when the parent also depends on it. The contemporary behaviour is that peer dependees that are not marked as optional are included regardless of the parent’s dependencies.

Peer dependencies appear only in npm’s row of Table 1; npm stands in for a family of JavaScript package managers (pnpm, Bun, Yarn, etc.), omitted as they share its semantics. The mechanism itself is general – any concurrent-version ecosystem (e.g. Cargo) could adopt it – while single-version ecosystems enforce this implicitly, since every depender sees the same version.

2.2.8 Features. In Cargo, dependency relations are parameterised by features – optional functionality of a package that can introduce additional dependencies. For example,

```
image = { version = "0.25", features = ["png"] }
```

selects the image crate’s png feature, causing image to additionally depend on the png decoder crate. The resolver unifies features, so a package will be provided with the union of all features requested by its dependers. In the Python world, dependency specifiers define ‘extras’ which can similarly add dependencies. Portage’s *USE flags* serve the same role: a dependency can request USE flags with `dev-libs/foo[bar]`, and USE flags can trigger additional dependencies via `bar? ( dev-libs/baz )`. USE flags additionally allow a package to depend on the *absence* of a flag – `dev-libs/foo[-bar]` in Portage – combining features with conflicts (§2.2.5). Cargo features are only additive: a depender can require a feature but not block its presence.

Ecosystems without native feature support sometimes encode features manually. opam’s `ocaml-variants` package encodes compiler options in version strings such as `4.02.1+fp`: each combination is a separate version, of which version uniqueness (§3.1) permits one; without unification, every union of options is its own version, so their number grows combinatorially. Newer `ocaml-option-*` packages instead encode options as distinct package names, e.g. `ocaml-option-flambda` – structurally similar to the feature reduction of Def. 4.4.4 – with the `ocaml-options-only-*` family enforcing exclusive combinations via explicit conflicts (§2.2.5).

2.2.9 Package Formulae. opam allows a package to declare that it requires either `lwt` or `async` as a concurrency monad, e.g. `"lwt" {>= "6.0"} | "async"`. Similarly, APT supports disjunction in `Depends` fields, for example, `Depends: libssl-dev | libgnutls-dev` to accept either TLS implementation.

2.2.10 Variable Formulae. Variable formulae extend package formulae (§2.2.9). For example, in opam, the variable `os-distribution` makes a formula conditional on the operating system on which the package manager is deploying the package. In APT, dependencies can similarly be parameterised by computer architecture. The opam package `local-with-test` denotes dependencies that are only required when tests have been enabled for the package. Python’s dependency groups serve a similar role, having groups such as `docs` or `test` in `pyproject.toml` labelling sets of dependencies that are conditional on the development task. Variables can also be used to distinguish build-time from run-time dependencies. Instead of an explicit Boolean formula, these can also be implicit labels on dependencies, making them conditional on the platform or architecture.

2.2.11 Virtual Packages. Virtual packages let a depender express a need for some capability without naming a specific provider; capable packages declare themselves providers of the shared virtual name. For example, in APT [59], both `openssh-server` and `dropbear-bin` provide `ssh-server`.

2.2.12 Optional Dependencies. `opam` supports optional dependencies, where dependees are used by the depender if present in a resolution, determining build ordering and triggering rebuilds. Like features (§2.2.8), they enable functionality, but their effect is resolution-wide rather than scoped to individual dependencies. While such use-if-present behaviour occurs throughout toolchains – with Gentoo going so far as to outlaw unrecorded *automagic dependencies*, and sandboxed builds used to avoid them – only `opam` in Table 1 makes it explicit. Optional dependencies are a purely post-resolution concern, so we omit them from the model (Appendix E).

`opam`’s optional dependencies are distinct from `npm`’s `optionalDependencies` (like Debian’s `Recommends`), which ask the resolver to include a dependee if possible but do not fail otherwise. An extension with resolver preferences could capture this best-effort behaviour.

3 The Package Calculus

We define a minimal formal system for dependency resolution in package management systems (§3.1), extend it with version formulae (§3.2), and show how the NP-completeness of resolution can be avoided, but is ultimately inherent in capturing the complexity of real-world package managers (§3.3).

3.1 The Core Calculus

We define the core calculus, the minimal kernel of the Package Calculus.

DEFINITION 3.1.1 (Packages). We define:

- (a) N as the set of possible package names.
- (b) V as the set of possible package versions.
- (c) $N \times V$ as the set of possible packages, each a name-version pair.
- (d) $R \subseteq N \times V$ as the set of *real* packages – those that exist.

Throughout, any variable left unbound is implicitly universally quantified.

DEFINITION 3.1.2 (Dependency). We define dependencies $\Delta \subseteq (N \times V) \times (N \times \mathcal{P}(V))$ as a relation from packages to a name and set of versions. As is standard for a relation, $p\Delta(n, vs)$ abbreviates $(p, (n, vs)) \in \Delta$, where a depender $p \in N \times V$ expresses a dependency on dependees of name $n \in N$ with versions $vs \subseteq V$. We assume that dependencies are *functional in name*, where $p\Delta(n, vs) \wedge p\Delta(n, vs') \implies vs = vs'$. Dependencies that are not functional in name can be normalised by merging same-name entries per depender, intersecting their version sets.

DEFINITION 3.1.3 (Resolution). Given dependencies Δ and root $r \in R$, a resolution $S \subseteq R$ is valid if the following conditions hold:

- (a) **Root Inclusion:** $r \in S$
- (b) **Dependency Closure:** $\forall p \in S. p\Delta(n, vs) \implies \exists v \in vs. (n, v) \in S$
Every dependency of a package in the resolution must be satisfied by a compatible version.
- (c) **Version Uniqueness:** $\forall (n, v), (n, v') \in S. v = v'$

An installation request is a *query* $Q \subseteq N \times \mathcal{P}(V)$: a set of package names, each with a set of acceptable versions. As the formalism takes only a single root package as input, we realise a query as a synthetic root $r \in R$ whose dependencies are exactly Q , that is, $r\Delta(m, vs) \iff (m, vs) \in Q$.

We write $\mathcal{S}(\Delta, r)$ for the set of all resolutions of r in Δ . Read $S \in \mathcal{S}(\Delta, r)$ as ‘ S is a valid resolution’. `DEPENDENCYRESOLUTION` is the decision problem: given R, Δ , and r , is $\mathcal{S}(\Delta, r)$ non-empty?

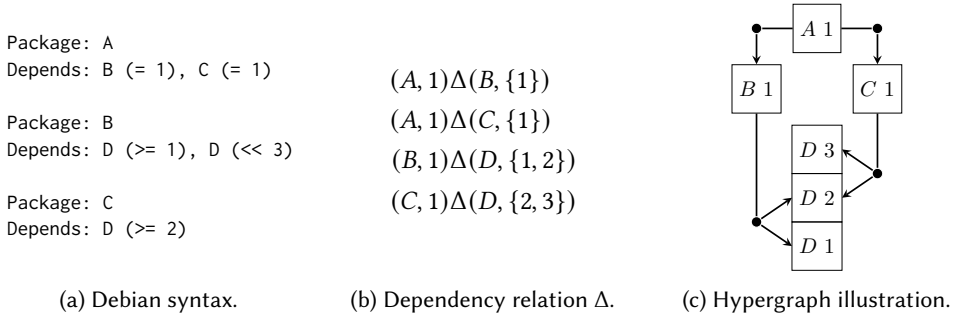


Fig. 3. An instance of the Package Calculus.

We illustrate these definitions with an example. Fig. 3a shows a snippet of Debian syntax for expressing the dependencies in Fig. 3b. We visualise these dependencies in Fig. 3c as a directed hypergraph – a generalisation of a directed graph in which each edge connects a set of source vertices to a set of target vertices [11]. Each dependency is an edge from the depender (the domain) to a set of dependees. Formally, we define vertices R and edges

$$E = \{(\{(n, v)\}, \{(m, u) \mid u \in vs\}) \mid (n, v)\Delta(m, vs)\}$$

Note that we restrict the domain to a cardinality of one – we can only express a dependency *from* one package. The only valid resolution¹ for query $Q = \{(A, \{1\})\}$ is

$$S = \{r, (A, 1), (B, 1), (C, 1), (D, 2)\}$$

THEOREM 3.1.4. **DEPENDENCYRESOLUTION** (Def. 3.1.3) is NP-complete.

PROOF. By a reduction from 3-SAT: a variable becomes a package name with versions $\top$ and $\perp$, and a clause becomes a package name, required by the root, with one version per literal – each depending on the assignment that satisfies that literal. See Appendix B. $\square$

3.2 Version Formulae

The core calculus represents each dependency as an explicit set of compatible versions (Def. 3.1.2). Real package managers, however, do not enumerate these sets: they describe them with *version formulae* (§2.2.4) interpreted over a total ordering of versions that is typically non-lexicographical. We show that the core’s abstract treatment of versions loses nothing by working with sets: a version formula reduces to the core by evaluating it to its set of admitted versions. This is also the simplest instance of the extend-and-reduce pattern of §4 – extend the calculus, define resolution over the extension, then reduce back to the core.

DEFINITION 3.2.1 (Version Ordering). A *version ordering* is a total order $\leq_v$ on V (Def. 3.1.1.b).

Different ecosystems define different version orderings and formulae [49, 58]. The VERS specification [45] defines a universal version range notation, implemented by the univers library [4], which wraps ecosystem-specific version comparison logic. We define the Version Formula Package Calculus to be as expressive as the VERS specification according to the following definitions.

¹An alternative core allows multiple versions and encodes uniqueness via conflicts, but many package managers do not support conflicts (Table 1); we model the majority case directly, and concurrent-version systems extend it (§4.2).

DEFINITION 3.2.2 (Version Formula). We define:

(a) Version formulae Φ :

$$\phi ::= \top \mid \perp \mid \phi \wedge \phi \mid \phi \vee \phi \mid \omega \ c \quad \omega ::= \geq \mid > \mid \leq \mid < \mid = \mid \neq \quad c \in V$$

(b) R_Φ , defined precisely as R in Def. 3.1.1.d, the subscript serving merely to differentiate it from the R produced by the reduction to the core.

(c) $V_n = \{v \mid (n, v) \in R_\Phi\}$, the set of versions of $n \in N$ that exist.

(d) A version ordering $\leq_v$ (Def. 3.2.1).

(e) Semantics function $\llbracket \cdot \rrbracket_n : \Phi \rightarrow \mathcal{P}(V)$ for Φ under $n \in N$, where ω_v interprets ω under $\leq_v$:

$$\begin{aligned} \llbracket \top \rrbracket_n &= V_n & \llbracket \phi_1 \wedge \phi_2 \rrbracket_n &= \llbracket \phi_1 \rrbracket_n \cap \llbracket \phi_2 \rrbracket_n & \llbracket \omega \ c \rrbracket_n &= \{v \in V_n \mid v \omega_v c\} \\ \llbracket \perp \rrbracket_n &= \emptyset & \llbracket \phi_1 \vee \phi_2 \rrbracket_n &= \llbracket \phi_1 \rrbracket_n \cup \llbracket \phi_2 \rrbracket_n \end{aligned}$$

These version formulae can express common version constraints:

$$\begin{aligned} ">1.0.0" &\mapsto \llbracket > 1.0.0 \rrbracket_n & "!=1.4.*" &\mapsto \llbracket < 1.4.0 \vee \geq 1.5.0 \rrbracket_n \\ "<1.2.3" &\mapsto \llbracket \geq 1.2.3 \wedge < 2.0.0 \rrbracket_n & "~1.2.3" &\mapsto \llbracket \geq 1.2.3 \wedge < 1.3.0 \rrbracket_n \end{aligned}$$

We extend the calculus to use version formulae as the representation of versions in dependencies (Def. 3.2.3), define what a valid resolution looks like over this representation (Def. 3.2.4), and reduce the extension back to the core calculus by evaluating each formula to its version set (Def. 3.2.5).

DEFINITION 3.2.3 (Version Formula Dependency). We define version formula dependencies $\Delta_\Phi \subseteq (N \times V) \times (N \times \Phi)$ where an element $p\Delta_\Phi(n, \phi)$ has formula ϕ representing the set of compatible versions of n .

DEFINITION 3.2.4 (Version Formula Resolution). Given dependencies Δ_Φ and root $r_\Phi \in R_\Phi$, a resolution $S_\Phi \subseteq R_\Phi$ is valid if:

- (a) **Root Inclusion:** $r_\Phi \in S_\Phi$, as in Def. 3.1.3.a.
- (b) **Dependency Closure:** $\forall p \in S_\Phi. p\Delta_\Phi(n, \phi) \implies \exists v \in \llbracket \phi \rrbracket_n. (n, v) \in S_\Phi$
- (c) **Version Uniqueness:** $\forall (n, v), (n, v') \in S_\Phi. v = v'$, as in Def. 3.1.3.c.

We write $S_\Phi(\Delta_\Phi, r_\Phi)$ for the set of all resolutions of r_Φ in Δ_Φ .

DEFINITION 3.2.5 (Version Formula Reduction). Given R_Φ , Δ_Φ , and r_Φ , we define a reduction to the core calculus with R , Δ , and r as follows:

- (a) **Packages:** $r = r_\Phi$ and $R = R_\Phi$
- (b) **Dependencies:** $\Delta = \{((n, v), (m, \llbracket \phi \rrbracket_m)) \mid (n, v)\Delta_\Phi(m, \phi)\}$

THEOREM 3.2.6 (Correctness). $S \in \mathcal{S}(\Delta, r) \iff S \in S_\Phi(\Delta_\Phi, r_\Phi)$

3.3 Resolution Complexity

Package managers navigate the NP-completeness of DEPENDENCYRESOLUTION (Thm. 3.1.4) in different ways, forming a spectrum from restricted-but-tractable resolution to full NP-completeness. At the most restricted end, Go's MVS (§2.2.4) [20] achieves deterministic, linear-time resolution. MVS relies on strict adherence to semantic versioning [49]: only major version changes can break APIs. Under this assumption, dependencies specify only a minimum version bound, since any version above the minimum within the same major version is guaranteed to be compatible. Different major versions are treated as different package names, allowing multiple major versions to coexist in the same resolution. Rather than selecting the latest available version, MVS selects the *minimum* version satisfying all bounds – the maximum of all minimum bounds. Since what constitutes a breaking change is ambiguous in practice, selecting the minimum reduces exposure to unintended breakage. Upgrades are an explicit developer action that rewrites the minimum bounds.

Cargo also allows multiple versions of a package name to exist in a resolution, but it falls short of linear-time resolution because its dependency formulae allow upper bounds within a major version – minor version bumps sometimes include breaking changes. Its dependency formulae can also span semver-incompatible [49] version sets (e.g. " ≥ 1 ", " < 3 ", " ≥ 0.1 ", " < 0.5 "), whereas one can only depend on *one* major version of a Go module.

At the other end, in ecosystems like OCaml, any API change is potentially breaking due to features such as module inclusion, where a package's API is implicitly dependent on the APIs of its dependees. There is no meaningful distinction between major and minor versions, making semantic versioning inapplicable and MVS unviable. Were opam to adopt MVS, it would be unable to express compatibility with multiple major versions, and so could depend on only single, exact versions. For this reason, opam specifies no upper bounds by default; instead, when a new release is proposed, the opam-repository continuous integration (CI) builds and tests all packages that depend on it (its reverse dependencies), and failures result in a retroactive upper bound on the new release [39].

Semantic versioning is a useful heuristic but not a substitute for expressive version constraints. Regardless of ecosystem, versioning is determined by the dependee, but incompatibility is a property of the dependency relation; one dependency's compatible change is another's breaking change.

Nix sidesteps the spectrum by treating every name-version pair as a distinct package and requiring the exact version of a dependee to be specified – singular dependencies (Appendix D). Derivations are parameterised by their dependencies in the Nix DSL, so resolution is performed manually rather than by a constraint solver. In the Nixpkgs package repository, typically one version of a package name is included at a time, and breakages are often due to incompatibilities introduced by updates.

We can formalise the approaches underlying this spectrum [19] as two independent ways to reduce the complexity of resolution:

- (1) **Restrict version constraints to lower bounds.** With a total ordering $\leq_v$ of versions (Def. 3.2.1), we redefine a dependency relation to only have a minimum bound, with a dependency as $p\Delta_{\min}(n, v)$ where $p \in N \times V$, $n \in N$, $v \in V$, and v is the minimum bound. We replace the dependency closure condition (Def. 3.1.3.b) with

$$\forall p \in S. p\Delta_{\min}(n, v) \implies \exists v' \in V. (n, v') \in S \wedge v \leq_v v'$$

An incompatibility between a depender and a dependee satisfying this relationship is considered a specification error; the depender and dependee authors must collaborate to fix it. Without upper bounds, any version above the maximum of all minimum bounds is valid, so a greedy algorithm suffices – selecting the minimum satisfying version for stability, as MVS does, or the latest for freshness. This reduces the problem to a graph traversal and version selection in $O(|\Delta_{\min}| + |R|)$ time.

- (2) **Remove version uniqueness.** Remove the version uniqueness condition (Def. 3.1.3.c), allowing multiple versions of a package name. Each dependency can be resolved independently, allowing a greedy algorithm to select any compatible version for each dependency. This reduces the problem to graph traversal in $O(|\Delta|)$ time.

Go's MVS combines both approaches – the version uniqueness constraint is removed for different major versions (approach 2), and within a major version, version constraints are restricted to a lower bound (approach 1). Cargo also removes the version uniqueness constraint for different major versions, but the practical need for upper bounds within a major version reintroduces NP-completeness. opam exploits neither, accepting NP-complete resolution in exchange for expressiveness, at the cost of extensive compatibility testing. Nix adopts approach 2 as well as a restriction stronger than approach 1: singular dependencies, where each dependency specifies exactly one version, delegating version selection from the constraint solver to a Turing-complete packaging language.

Where a package manager sits on this spectrum determines the sophistication of its resolver: greedy algorithms suffice at the restricted end, while those accepting full NP-completeness employ SAT solvers (Appendix C) or implement conflict-driven clause learning (CDCL) natively for performance and error reporting. Similarly, lock files [29] – snapshots of a resolution – are unnecessary under MVS, where the resolution is determined by the minimum bounds alone, but are necessary for reproducibility where the resolution changes as new versions are released.

Our theory was informed by the goal of practical resolution across ecosystems [22, 25]. To this end, singular dependencies and Go’s MVS are not expressive enough to model general dependency resolution, while the Turing-complete Nix DSL is too expressive to model in other dependency resolvers and does not (yet) support resolving version constraints. We argue that the Package Calculus is the *minimum* degree of expressiveness needed to model the ecosystems surveyed.

4 Package Managers, Mise en Place

We model each axis of divergence in dependency expression (§2) by extending the core calculus (§3.1) and reducing each extension back to the core. We define the Conflict Package Calculus (§4.1), Concurrent Package Calculus (§4.2), Peer Package Calculus (§4.3), Feature Package Calculus (§4.4), Package Formula Calculus (§4.5), Variable Formula Calculus (§4.6), and Virtual Package Calculus (§4.7). We cover singular dependencies, which are less expressive than the core, in Appendix D; and optional dependencies, which have no resolution-time effect, in Appendix E. All reductions are polynomial in the size of the input instance. The reductions serve as a compilation pass from the extended calculus to the core. The soundness theorems of the reductions construct a resolution in the extended calculus from a resolution of the reduction. We prepare each extension in isolation – a *mise en place* – before composing them to order in §5.

4.1 Conflicts

The core calculus has no mechanism for one package to forbid another (§2.2.5). We extend it with a conflict relation (Def. 4.1.1), define what a valid resolution looks like over the extended calculus (Def. 4.1.2), and reduce back to the core by encoding each conflict as a synthetic package (Def. 4.1.3).

DEFINITION 4.1.1 (Conflict). We define conflicts $\Gamma \subseteq (N \times V) \times (N \times \mathcal{P}(V))$ as a relation where an element $p\Gamma(n, vs)$ denotes that package $p \in N \times V$ conflicts with package name $n \in N$ with versions $vs \subseteq V$.

We use conflicts Γ as the defining symbol of the calculus and introduce Δ_Γ as Δ in Def. 3.1.2, and R_Γ as R in Def. 3.1.1.d, in order to differentiate them from the use of Δ and R in the reduction to the core. In subsequent extensions, unless we explicitly redefine a subscript of a core calculus definition – as we do with version formula dependency Δ_Φ in Def. 3.2.3 – read it as defined in the core.

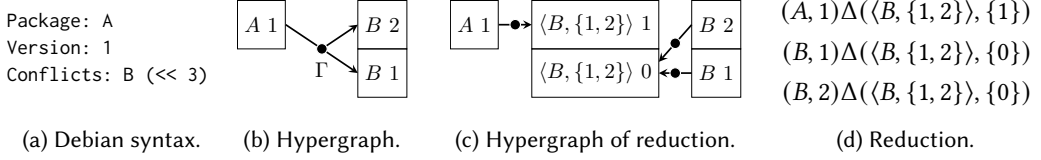
DEFINITION 4.1.2 (Conflict Resolution). Given dependencies Δ_Γ , conflicts Γ , and root r_Γ , a resolution $S_\Gamma \subseteq R_\Gamma$ is valid if:

- (a) **Core Resolution:** $S_\Gamma \in \mathcal{S}(\Delta_\Gamma, r_\Gamma)$ (Def. 3.1.3).
- (b) **Conflict Avoidance:** $\forall p \in S_\Gamma. p\Gamma(n, vs) \implies \nexists v \in vs. (n, v) \in S_\Gamma$

For every package in the resolution, a conflicting package must not be in the resolution.

We write $S_\Gamma(\Delta_\Gamma, \Gamma, r_\Gamma)$ for the set of all resolutions of r_Γ in Δ_Γ and Γ .

We reduce the Conflict Package Calculus to the core calculus by encoding each conflict as a synthetic conflict package name with two versions, 1 and 0. The package declaring the conflict depends on version 1, while each conflicting package depends on version 0. Since both versions cannot coexist by version uniqueness, the two sides of the conflict are mutually exclusive. Fig. 4 shows an instance of the Conflict Package Calculus reduced to the core.

Fig. 4. Conflict Package Calculus $(A, 1)\Gamma(B, \{1, 2\})$ reduced to the core calculus.

DEFINITION 4.1.3 (Conflict Reduction). Given R_Γ , Δ_Γ , Γ , and r_Γ , we define a reduction to the core calculus with R , Δ , and r as follows:

(a) **Packages:** $r = r_\Gamma$ and, with $\langle n, vs \rangle \in N$ and $0, 1 \in V$,

$$R = R_\Gamma \cup \bigcup_{p\Gamma(n,vs)} \{(\langle n, vs \rangle, 0), (\langle n, vs \rangle, 1)\}$$

(b) **Dependencies:**

$$\Delta = \Delta_\Gamma \cup \bigcup_{p\Gamma(n,vs)} \{(p, (\langle n, vs \rangle, \{1\}))\} \cup \bigcup_{p\Gamma(n,vs)} \{((n, u), (\langle n, vs \rangle, \{0\})) \mid u \in vs\}$$

We define soundness and completeness theorems for each reduction. Given an instance of the extended calculus, soundness gives a construction that extracts a resolution of the extended instance from a resolution of its core reduction. Completeness goes the other way: a construction of a resolution of the core reduction from a resolution of the extended instance.

THEOREM 4.1.4 (Soundness). If $S \in \mathcal{S}(\Delta, r)$, then $S_\Gamma = S \cap R_\Gamma$ is a valid resolution in $\mathcal{S}_\Gamma(\Delta_\Gamma, \Gamma, r_\Gamma)$.

THEOREM 4.1.5 (Completeness). If $S_\Gamma \in \mathcal{S}_\Gamma(\Delta_\Gamma, \Gamma, r_\Gamma)$, then

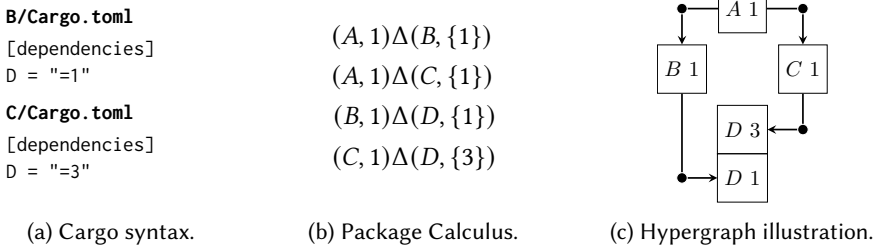
$$S = S_\Gamma \cup \bigcup_{p\Gamma(n,vs)} \{(\langle n, vs \rangle, 1) \mid p \in S_\Gamma\} \cup \bigcup_{p\Gamma(n,vs)} \{(\langle n, vs \rangle, 0) \mid u \in vs, (n, u) \in S_\Gamma\}$$

is a valid resolution in $\mathcal{S}(\Delta, r)$.

This mechanism generalises to a mutual-exclusion *conflict class* by giving the synthetic name one version per member, encoding the n -way exclusion in $O(n)$ instead of $O(n^2)$ pairwise conflicts.

4.2 Concurrent Versions

Fig. 5 illustrates an example of the ‘diamond dependency problem’; $\{(A, B), (A, C), (B, D), (C, D)\}$ as edges form a diamond. We define the Concurrent Package Calculus to model package managers that address this by allowing multiple versions of a package name to exist in the resolution (§2.2.6).

Fig. 5. The ‘diamond dependency problem’, which with $Q = \{(A, \{1\})\}$ has no valid resolution.

DEFINITION 4.2.1 (Granularity Function). We define a set of granularities G and a function $g : V \rightarrow G$ mapping each version to a granularity such that versions that cannot coexist share a granularity. For example, in Cargo, we can define $g(x.y.z) = x$ to extract the major version of a package, and for npm or Nix $g(v) = v$ as they allow duplication regardless of version. We can emulate the core with $g(v) = \epsilon$.

DEFINITION 4.2.2 (Concurrent Resolution). Given dependencies Δ_C , granularity g , and root r_C , a resolution $S_C \subseteq R_C$ and parent relation $\pi \subseteq S_C \times S_C$ are valid if:

- (a) **Root Inclusion:** $r_C \in S_C$, as in Def. 3.1.3.a.
- (b) **Parent Closure:** $\forall p \in S_C. p \Delta_C(n, vs) \implies \exists! v \in vs. (n, v) \in S_C \wedge ((n, v), p) \in \pi$
A parent of a package is the depender in the dependency that caused the child's inclusion, and each parent has exactly one child per dependency.
- (c) **Version Granularity:** $\forall (n, v), (n, v') \in S_C. v \neq v' \implies g(v) \neq g(v')$
Different versions of a name in S_C must differ under g .

We write $S_C(\Delta_C, g, r_C) \ni (S_C, \pi)$ for all resolution-parent pairs of r_C in Δ_C under g .

To reduce the Concurrent Package Calculus to the core calculus, we push the granular version into the package name so that version uniqueness in the core calculus enforces version granularity. We introduce intermediate packages that allow for depending on multiple granular versions of now-distinct package names [50]. The parent relation π and the intermediate packages of the reduction below determine one another (Thms 4.2.4 and 4.2.5); peer dependencies (§4.3) then constrain precisely this selection. npm represents π in its lock file as nested `node_modules` paths. Fig. 6 shows an instance of the Concurrent Package Calculus and its reduction to the core.

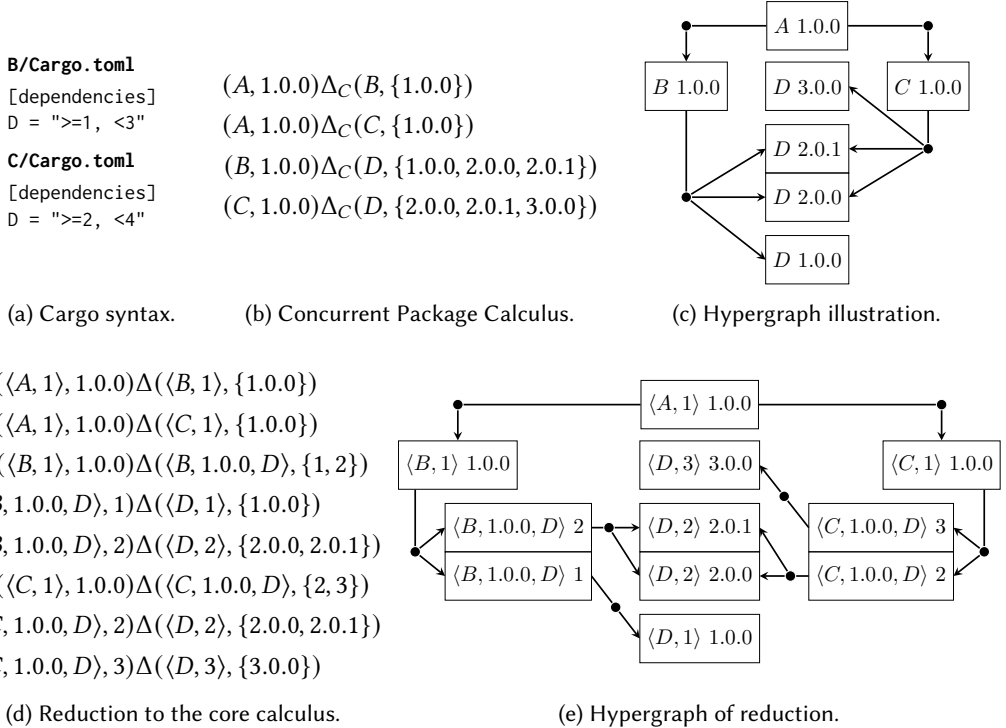


Fig. 6. Concurrent Package Calculus instance where $g(x.y.z) = x$, reduced to the core calculus.

DEFINITION 4.2.3 (Concurrent Reduction). Given R_C , Δ_C , g , and r_C , we reduce to the core with R , Δ , and r . For each $(n, v) \Delta_C(m, vs)$ let $W = \{g(u) \mid u \in vs\}$, with $\langle n, g(v) \rangle, \langle n, v, m \rangle \in N$ and $W \subseteq V$.

(a) **Packages:** $r = \langle \langle n, g(v) \rangle, v \rangle$ where $r_C = (n, v)$, and

$$R = \bigcup_{(n,v) \in R_C} \{(\langle \langle n, g(v) \rangle, v \rangle)\} \cup \bigcup_{(n,v) \Delta_C(m, vs)} \{(\langle \langle n, v, m \rangle, w \rangle \mid |W| > 1, w \in W)\}$$

(b) **Dependencies:** each dependency reduces to a dependency on single-granularity dependees, or to an intermediate $\langle n, v, m \rangle$ with each version depending on a granularity group:

$$\Delta = \bigcup_{(n,v) \Delta_C(m, vs)} \begin{cases} \{(\langle \langle \langle n, g(v) \rangle, v \rangle, (\langle m, w \rangle, vs) \rangle)\} & \text{if } W = \{w\} \\ \left(\begin{aligned} &\{(\langle \langle \langle n, g(v) \rangle, v \rangle, (\langle n, v, m \rangle, W) \rangle)\} \\ &\cup \{(\langle \langle \langle n, v, m \rangle, w \rangle, (\langle m, w \rangle, \{u \in vs \mid g(u) = w\}) \rangle) \mid w \in W\} \end{aligned} \right) & \text{if } |W| > 1 \\ \{(\langle \langle \langle n, g(v) \rangle, v \rangle, (\langle n, v, m \rangle, \emptyset) \rangle)\} & \text{if } W = \emptyset \end{cases}$$

THEOREM 4.2.4 (Soundness). If $S \in \mathcal{S}(\Delta, r)$, then $S_C = \{(n, v) \mid (\langle n, g(v) \rangle, v) \in S\}$ and

$$\pi = \bigcup_{(n,v) \Delta_C(m, vs)} \left\{ ((m, u), (n, v)) \mid \begin{array}{l} u \in vs, (n, v) \in S_C, (m, u) \in S_C, \\ |W| > 1 \implies (\langle n, v, m \rangle, g(u)) \in S \end{array} \right\}$$

are a valid resolution and parent relation in $\mathcal{S}_C(\Delta_C, g, r_C)$.

THEOREM 4.2.5 (Completeness). If $(S_C, \pi) \in \mathcal{S}_C(\Delta_C, g, r_C)$, then

$$S = \{(\langle \langle n, g(v) \rangle, v \rangle) \mid (n, v) \in S_C\} \cup \bigcup_{(n,v) \Delta_C(m, vs)} \left\{ (\langle \langle n, v, m \rangle, g(u) \rangle) \mid \begin{array}{l} (n, v) \in S_C, (m, u) \in S_C, u \in vs, \\ ((m, u), (n, v)) \in \pi, |W| > 1 \end{array} \right\}$$

is a valid resolution in $\mathcal{S}(\Delta, r)$.

The Concurrent Package Calculus allows multiple versions of the same package name to be selected even if it would be possible to use just one, consistent with Cargo's implementation. Future work could restrict versions for *public* dependencies exposed in a depender's interface, which must share one version across their subgraph – a compatibility question that type interfaces make precise [28] – as in Cargo's proposed public/private dependencies [55].

4.3 Peer Dependencies

A peer dependency (§2.2.7) is a child's constraint on a peer the parent must also depend on – meaningful only when multiple versions can coexist, so we define the Peer Package Calculus atop the Concurrent Package Calculus (§4.2). Our formalisation captures npm's `--legacy-peer-deps` behaviour, though the contemporary behaviour can also be modelled with minor modifications.

DEFINITION 4.3.1 (Peer Dependency). We define peer dependencies $\Theta \subseteq (N \times V) \times (N \times \mathcal{P}(V))$ as a relation where an element $p\Theta(n, vs)$ denotes that a parent of package p can only depend on the peer package name n with version in vs .

DEFINITION 4.3.2 (Peer Dependency Resolution). Given dependencies Δ_C , peer dependencies Θ , granularity g , and root r_C , a resolution $S_\Theta \subseteq R_C$ and parent relation $\pi \subseteq S_\Theta \times S_\Theta$ are valid if:

- (a) **Concurrent Resolution:** (S_Θ, π) is valid in $\mathcal{S}_C(\Delta_C, g, r_C)$ (Def. 4.2.2).
- (b) **Peer Satisfaction:** The version of n that p 's parent q selected must satisfy p 's peer constraint.

$$\forall p \in S_\Theta. p\Theta(n, vs) \wedge (p, q) \in \pi \wedge q\Delta_C(n, us) \wedge v \in us \wedge ((n, v), q) \in \pi \implies v \in vs$$

We write $\mathcal{S}_\Theta(\Delta_C, \Theta, g, r_C) \ni (S_\Theta, \pi)$ for all resolution-parent pairs of r_C in Δ_C and Θ under g .

A/package.json

"dependencies":

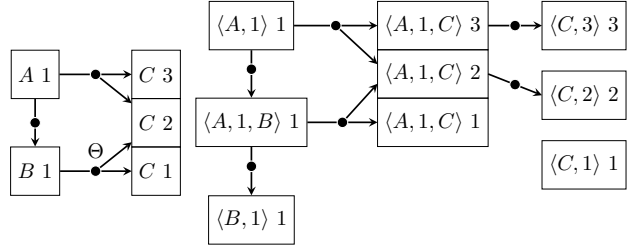
```
{
  "B": "1",
  "C": ">=2 <4"
}
```

B/package.json

"peerDependencies":

```
{
  "C": ">=1 <3"
}
```

$(A, 1)\Delta_C(B, \{1\})$
 $(A, 1)\Delta_C(C, \{2, 3\})$
 $(B, 1)\Theta(C, \{1, 2\})$



(a) npm syntax. (b) Peer Package Calculus. (c) Hypergraph. (d) Hypergraph of reduction.

Fig. 7. Peer Package Calculus instance where $g(v) = v$, reduced to the core calculus.
 $(\langle A, 1, C \rangle, 1)$ can never be selected so it has no dependency on $(\langle C, 1 \rangle, \{1\})$.

We reduce the Peer Package Calculus to the core calculus by modifying the Concurrent Reduction (Def. 4.2.3), using intermediate package dependencies to constrain the peer package version selection. Fig. 7 shows an instance of the Peer Package Calculus reduced to the core.

DEFINITION 4.3.3 (Peer Dependency Reduction). Given R_C , Δ_C , Θ , g , and r_C , we define a reduction to the core calculus with R , Δ , and r . For each dependency $(n, v)\Delta_C(m, vs)$ let $i = \langle n, v, m \rangle$, with $\langle n, g(v) \rangle, i \in N$, and let $Peers = \{(o, u, ws) \mid (n, v)\Delta_C(o, us), u \in us, (o, u)\Theta(m, ws)\}$, the sibling instances constraining m .

(a) **Packages:** $r = (\langle n, g(v) \rangle, v)$ where $r_C = (n, v)$, and

$$R = \bigcup_{(n,v) \in R_C} \{(\langle n, g(v) \rangle, v)\} \cup \bigcup_{(n,v) \Delta_C(m, vs)} (\{(i, u) \mid u \in vs\} \cup \{(i, w) \mid (o, u, ws) \in Peers, w \in ws\})$$

(b) **Dependencies:** each dependency reduces to a dependency on an intermediate name i with versions vs , each of which depends on the corresponding dependee. A peer dependency on m from a sibling o extends i with the peer versions and makes o 's intermediate packages depend on i . We use the full version in the intermediate package to allow selection of an exact peer package version as the intersection of constraints, regardless of g .

$$\Delta = \bigcup_{(n,v) \Delta_C(m, vs)} \left(\begin{aligned} &\{(\langle n, g(v) \rangle, v), (i, vs)\} \\ &\cup \{(\langle i, u \rangle, \langle m, g(u) \rangle, \{u\}) \mid u \in vs\} \\ &\cup \{(\langle n, v, o \rangle, u), (i, ws) \mid (o, u, ws) \in Peers\} \end{aligned} \right)$$

THEOREM 4.3.4 (Soundness). If $S \in \mathcal{S}(\Delta, r)$, then $S_\Theta = \{(n, v) \mid (\langle n, g(v) \rangle, v) \in S\}$ and

$$\pi = \bigcup_{(n,v) \Delta_C(m, vs)} \{(\langle m, u \rangle, (n, v)) \mid u \in vs, (n, v) \in S_\Theta, (\langle n, v, m \rangle, u) \in S\}$$

are a valid resolution and parent relation in $\mathcal{S}_\Theta(\Delta_C, \Theta, g, r_C)$.

THEOREM 4.3.5 (Completeness). If $(S_\Theta, \pi) \in \mathcal{S}_\Theta(\Delta_C, \Theta, g, r_C)$, then

$$S = \bigcup_{(n,v) \in S_\Theta} \{(\langle n, g(v) \rangle, v)\} \cup \bigcup_{(n,v) \Delta_C(m, vs)} \left\{ (\langle n, v, m \rangle, u) \mid \begin{array}{l} (n, v) \in S_\Theta, (m, u) \in S_\Theta, \\ u \in vs, ((m, u), (n, v)) \in \pi \end{array} \right\}$$

is a valid resolution in $\mathcal{S}(\Delta, r)$.

4.4 Features

Features (§2.2.8) are optional functionality of a package that may add dependencies, with selections unified across all dependers. The core has no notion of parameterised dependency or unification; we extend it with both.

DEFINITION 4.4.1 (Feature). We define:

- (a) F as the finite set of possible features.
- (b) $support \subseteq (N \times V) \times F$ as a relation where $(p, f) \in support$ denotes that package p supports feature f .

DEFINITION 4.4.2 (Feature Dependency). We define:

- (a) Parameterised dependencies $\Delta_f \subseteq (N \times V) \times (N \times \mathcal{P}(V) \times \mathcal{P}(F))$ as a relation where an element $p\Delta_f(n, vs, fs)$ denotes that package $p \in N \times V$ depends on package name $n \in N$, with a set of compatible versions $vs \subseteq V$, and a set of required features $fs \subseteq F$.
- (b) Additional dependencies $\Delta_a \subseteq ((N \times V) \times F) \times (N \times \mathcal{P}(V) \times \mathcal{P}(F))$ as a relation where an element $(p, f)\Delta_a(n, vs, fs)$ denotes that package $p \in N \times V$ with feature $f \in F$ depends on (n, vs, fs) as in Δ_f .

DEFINITION 4.4.3 (Feature Resolution). Given parameterised dependencies Δ_f , additional dependencies Δ_a , and root r_f with no features ($\forall f. (r_f, f) \notin support$), a resolution $S_f \subseteq R_f \times \mathcal{P}(F)$ – where an element $(p, fs) \in S_f$ represents package p selected with features $fs \subseteq F$ – is valid if:

- (a) **Root Inclusion:** $(r_f, \emptyset) \in S_f$
- (b) **Parameterised Dependency Closure:**

$$\forall (p, fs_p) \in S_f. p\Delta_f(n, vs, fs) \implies \exists v \in vs, fs' \supseteq fs. ((n, v), fs') \in S_f$$

A dependency parameterised by features must be satisfied by a package with a compatible version and selected features.

- (c) **Additional Dependency Closure:**

$$\forall (p, fs_p) \in S_f, f \in fs_p. (p, f)\Delta_a(n, vs, fs) \implies \exists v \in vs, fs' \supseteq fs. ((n, v), fs') \in S_f$$

A dependency added by a selected feature of the depender must be satisfied as with parameterised dependencies.

- (d) **Feature Unification:** $\forall ((n, v), fs), ((n, v'), fs') \in S_f. fs = fs'$
- (e) **Version Uniqueness:** $\forall ((n, v), fs), ((n, v'), fs') \in S_f. v = v'$
- (f) **Support:** $\forall (p, fs) \in S_f, f \in fs. (p, f) \in support$

Every selected feature of a package must be in the support relation.

We write $S_f(\Delta_f, \Delta_a, r_f)$ for the set of all resolutions of r_f in Δ_f and Δ_a .

To reduce the Feature Package Calculus to the core calculus, we create packages representing features selected for a package [50]. Fig. 8 shows an instance of the Feature Package Calculus and its reduction to the core.

DEFINITION 4.4.4 (Feature Reduction). Given $R_f, \Delta_f, \Delta_a, support$, and r_f , we define a reduction to the core calculus with R, Δ , and r . A dependee (m, vs, fs) reduces to a set of *targets* – the base name when it requires no features, or its feature packages otherwise:

$$t(m, vs, fs) = \begin{cases} \{(m, vs)\} & \text{if } fs = \emptyset \\ \{(\langle m, f' \rangle, vs) \mid f' \in fs\} & \text{if } fs \neq \emptyset \end{cases}$$

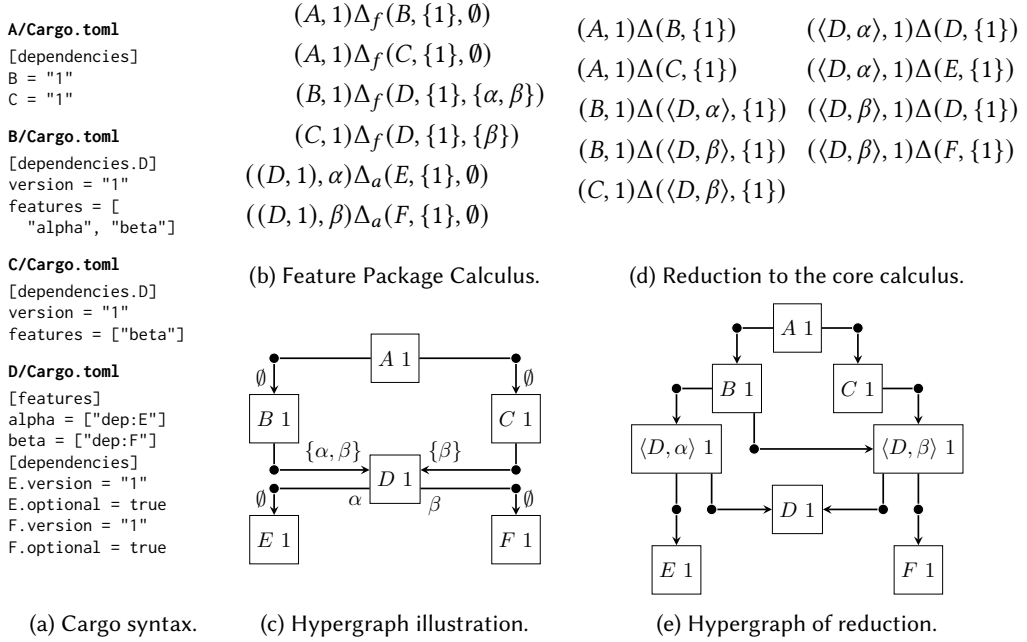


Fig. 8. Feature Package Calculus instance reduced to the core calculus.

(a) **Packages:** $r = r_f$, and each supported feature gives a feature package name $\langle n, f \rangle \in N$:

$$R = R_f \cup \bigcup_{(n,v) \in R_f} \{(\langle n, f \rangle, v) \mid ((n, v), f) \in \text{support}\}$$

(b) **Dependencies:** each synthetic feature package depends on its base package; each parameterised or additional dependency reduces to one dependency per target:

$$\begin{aligned} \Delta = & \bigcup_{(n,v) \in R_f} \{(\langle n, f \rangle, v), (n, \{v\}) \mid ((n, v), f) \in \text{support}\} \\ & \cup \bigcup_{p \Delta_f(n, vs, fs)} \{(p, d) \mid d \in t(n, vs, fs)\} \\ & \cup \bigcup_{((n,v), f) \Delta_a(m, vs, fs)} \{(\langle n, f \rangle, v), d \mid d \in t(m, vs, fs)\} \end{aligned}$$

THEOREM 4.4.5 (Soundness). If $S \in \mathcal{S}(\Delta, r)$, then

$$S_f = \{((n, v), \{f \mid (\langle n, f \rangle, v) \in S\}) \mid (n, v) \in S \cap R_f\}$$

is a valid resolution in $\mathcal{S}_f(\Delta_f, \Delta_a, r_f)$.

THEOREM 4.4.6 (Completeness). If $S_f \in \mathcal{S}_f(\Delta_f, \Delta_a, r_f)$, then

$$S = \bigcup_{((n,v), fs) \in S_f} \left(\{(n, v)\} \cup \{(\langle n, f \rangle, v) \mid f \in fs\} \right)$$

is a valid resolution in $\mathcal{S}(\Delta, r)$.

4.5 Package Formulae

The core's dependency relation is conjunctive: every dependency must be satisfied. Some packaging DSLs allow Boolean formulae (§2.2.9) which a conjunction of dependencies alone cannot express.

DEFINITION 4.5.1 (Package Formula). We define:

(a) The set of package formulae Ψ is given by the grammar:

$$\psi ::= (n, vs) \mid \psi \wedge \psi \mid \psi \vee \psi \mid \neg \psi \quad n \in N, vs \subseteq V$$

(b) The satisfaction relation $S_\Psi \models \psi$ (resolution S_Ψ satisfies formula ψ) is defined:

$$\begin{array}{c} \frac{\exists v \in vs. (n, v) \in S_\Psi}{S_\Psi \models (n, vs)} \text{ DEP} \quad \frac{S_\Psi \models \psi_L \quad S_\Psi \models \psi_R}{S_\Psi \models \psi_L \wedge \psi_R} \text{ AND} \quad \frac{S_\Psi \not\models \psi}{S_\Psi \models \neg \psi} \text{ NOT} \\[10pt] \frac{S_\Psi \models \psi_L}{S_\Psi \models \psi_L \vee \psi_R} \text{ OR-L} \quad \frac{S_\Psi \models \psi_R}{S_\Psi \models \psi_L \vee \psi_R} \text{ OR-R} \end{array}$$

DEFINITION 4.5.2 (Package Formula Dependency). We define package formula dependencies $\Delta_\Psi \subseteq (N \times V) \times \Psi$ as a relation where an element $p\Delta_\Psi\psi$ denotes that package p depends on formula ψ .

DEFINITION 4.5.3 (Package Formula Resolution). Given dependencies Δ_Ψ and root r_Ψ , a resolution $S_\Psi \subseteq R_\Psi$ is valid if:

- (a) **Root Inclusion:** $r_\Psi \in S_\Psi$, as in Def. 3.1.3.a.
- (b) **Formula Closure:** $\forall p \in S_\Psi. p\Delta_\Psi\psi \implies S_\Psi \models \psi$
- (c) **Version Uniqueness:** $\forall (n, v), (n, v') \in S_\Psi. v = v'$, as in Def. 3.1.3.c.

We write $S_\Psi(\Delta_\Psi, r_\Psi)$ for the set of all resolutions of r_Ψ in Δ_Ψ .

We reduce the Package Formula Calculus to the core calculus using synthetic packages for disjunction and the conflict encoding from Def. 4.1.3 for negation. Each disjunction node introduces one synthetic package name with two versions (one per disjunct), analogous to a Tseitin auxiliary variable [64], keeping the reduction linear in the formula size and avoiding the exponential blowup of naïve conversion to conjunctive normal form (CNF). Fig. 9 shows an instance reduced to the core.

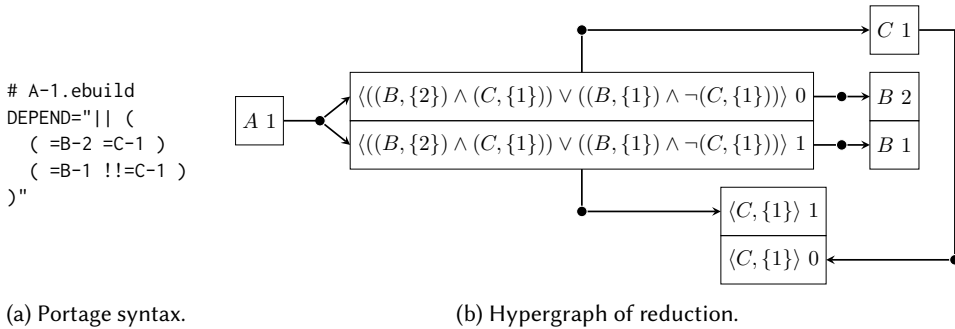


Fig. 9. Package Formula Calculus instance $(A, 1)\Delta_\Psi(\langle (B, \{2\}) \wedge (C, \{1\}) \rangle \vee \langle (B, \{1\}) \wedge \neg(C, \{1\}) \rangle)$ reduced to the core calculus.

DEFINITION 4.5.4 (Package Formula Reduction). Given R_Ψ , Δ_Ψ , and r_Ψ , we define a reduction to the core calculus with R , Δ , and r as follows:

- (a) **Packages:** $r = r_\Psi$, and synthetic packages for each disjunction and negated atom arising in the encoding $\mathcal{E}$ (Def. 4.5.4.b), including those introduced by its De Morgan rewriting:

$$R = R_\Psi \cup \bigcup_{\substack{\psi_L \vee \psi_R \\ \text{arising in } \mathcal{E}}} \{(\langle \psi_L \vee \psi_R \rangle, 0), (\langle \psi_L \vee \psi_R \rangle, 1)\} \cup \bigcup_{\substack{\neg(n, vs) \\ \text{arising in } \mathcal{E}}} \{(\langle n, vs \rangle, 0), (\langle n, vs \rangle, 1)\}$$

where $\langle \psi_L \vee \psi_R \rangle, \langle n, vs \rangle \in N$ and $0, 1 \in V$.

- (b) **Formula:** we define the encoding function $\mathcal{E} : \Delta_\Psi \rightarrow \mathcal{P}(\Delta)$:

$$\mathcal{E}(p\Delta_\Psi(n, vs)) = \{(p, (n, vs))\}$$

$$\mathcal{E}(p\Delta_\Psi(\psi_L \wedge \psi_R)) = \mathcal{E}(p\Delta_\Psi\psi_L) \cup \mathcal{E}(p\Delta_\Psi\psi_R)$$

$$\mathcal{E}(p\Delta_\Psi(\psi_L \vee \psi_R)) = \{(p, (i, \{0, 1\}))\} \cup \mathcal{E}((i, 0)\Delta_\Psi\psi_L) \cup \mathcal{E}((i, 1)\Delta_\Psi\psi_R) \text{ with } i := \langle \psi_L \vee \psi_R \rangle$$

$$\mathcal{E}(p\Delta_\Psi\neg(n, vs)) = \{(p, (\langle n, vs \rangle, \{1\}))\} \cup \{(n, u), (\langle n, vs \rangle, \{0\}) \mid u \in vs\}$$

$$\mathcal{E}(p\Delta_\Psi\neg(\psi_L \wedge \psi_R)) = \mathcal{E}(p\Delta_\Psi(\neg\psi_L \vee \neg\psi_R))$$

$$\mathcal{E}(p\Delta_\Psi\neg(\psi_L \vee \psi_R)) = \mathcal{E}(p\Delta_\Psi(\neg\psi_L \wedge \neg\psi_R))$$

$$\mathcal{E}(p\Delta_\Psi\neg\neg\psi) = \mathcal{E}(p\Delta_\Psi\psi)$$

- (c) **Dependencies:** we apply the encoding function to each dependency $\Delta = \bigcup_{p\Delta_\Psi\psi} \mathcal{E}(p\Delta_\Psi\psi)$.

THEOREM 4.5.5 (Soundness). If $S \in \mathcal{S}(\Delta, r)$, then $S_\Psi = S \cap R_\Psi$ is a valid resolution in $\mathcal{S}_\Psi(\Delta_\Psi, r_\Psi)$.

THEOREM 4.5.6 (Completeness). If $S_\Psi \in \mathcal{S}_\Psi(\Delta_\Psi, r_\Psi)$, define the witness set $W^c(\psi)$ for $c \in \{\top, \perp\}$, where $\top$ marks subformulae on a *taken* path, and $\perp$ on an *untaken* path:

$$W^c((n, vs)) = \emptyset$$

$$W^c(\psi_L \wedge \psi_R) = W^c(\psi_L) \cup W^c(\psi_R)$$

$$W^\top(\neg(n, vs)) = \{(\langle n, vs \rangle, 1)\}$$

$$W^\perp(\neg(n, vs)) = \begin{cases} \{(\langle n, vs \rangle, 0)\} & \text{if } \exists u \in vs. (n, u) \in S_\Psi \\ \emptyset & \text{otherwise} \end{cases}$$

$$W^\top(\psi_L \vee \psi_R) = \begin{cases} \{(\langle \psi_L \vee \psi_R \rangle, 0)\} \cup W^\top(\psi_L) \cup W^\perp(\psi_R) & \text{if } S_\Psi \models \psi_L \\ \{(\langle \psi_L \vee \psi_R \rangle, 1)\} \cup W^\perp(\psi_L) \cup W^\top(\psi_R) & \text{otherwise} \end{cases}$$

$$W^\perp(\psi_L \vee \psi_R) = W^\perp(\psi_L) \cup W^\perp(\psi_R)$$

$$W^c(\neg(\psi_L \wedge \psi_R)) = W^c(\neg\psi_L \vee \neg\psi_R)$$

$$W^c(\neg(\psi_L \vee \psi_R)) = W^c(\neg\psi_L \wedge \neg\psi_R)$$

$$W^c(\neg\neg\psi) = W^c(\psi)$$

Then $S = S_\Psi \cup \bigcup_{p\Delta_\Psi\psi} \begin{cases} W^\top(\psi) & \text{if } p \in S_\Psi \\ W^\perp(\psi) & \text{otherwise} \end{cases}$ is a valid resolution in $\mathcal{S}(\Delta, r)$.

4.6 Variable Formulae

Packaging DSLs sometimes allow variables in package formulae (§2.2.10); we extend the Boolean package formula with variables, values, and comparisons.

DEFINITION 4.6.1 (Variable Formula). Extending the Package Formula (Def. 4.5.1), we define:

- (a) The finite set of variables X , the non-empty set of variable values Y totally ordered by $\leq_Y$, the non-empty set Y_x of possible values for each $x \in X$, and assignment function $\sigma : (x : X) \rightarrow Y_x$.
- (b) The set of variable formulae Ψ is given by the grammar:

$$\psi ::= (n, vs) \mid \psi \wedge \psi \mid \psi \vee \psi \mid \neg \psi \mid x \omega y \quad \omega ::= \geq \mid > \mid \leq \mid < \mid = \mid \neq \quad x \in X, y \in Y$$

- (c) Variable formula dependencies $\Delta_\Psi \subseteq (N \times V) \times \Psi$ as in Def. 4.5.2.
- (d) The satisfaction relation $(S_\Psi, \sigma) \models \psi$ (resolution S_Ψ and assignment σ satisfy formula ψ) extends Def. 4.5.1.b with the judgement form carrying assignment σ . Writing ω_Y for the interpretation of ω under $\leq_Y$, we define:

$$\frac{\sigma(x) \omega_Y y}{(S_\Psi, \sigma) \models (x \omega y)}$$

DEFINITION 4.6.2 (Variable Formula Resolution). Given dependencies Δ_Ψ and root r_Ψ , a resolution $S_\Psi \subseteq R_\Psi$ and assignment σ are valid if:

- (a) **Root Inclusion:** $r_\Psi \in S_\Psi$, as in Def. 3.1.3.a.
- (b) **Formula Closure:** $\forall p \in S_\Psi. p \Delta_\Psi \psi \implies (S_\Psi, \sigma) \models \psi$
- (c) **Version Uniqueness:** $\forall (n, v), (n, v') \in S_\Psi. v = v'$, as in Def. 3.1.3.c.

We write $S_\Psi(\Delta_\Psi, r_\Psi) \ni (S_\Psi, \sigma)$ for all resolution-assignment pairs of r_Ψ in Δ_Ψ .

Variable formulae can capture package-level platform constraints. For example, in npm, "os": ["linux"] depends on variable 'os' with value 'linux'. A filtered dependency in opam, depends: ["foo" {os = "linux"}], becomes $\neg(os = \text{linux}) \vee (foo, V)$. Package-local variables, such as opam's with-test, are namespaced by the package. Some package managers perform this expansion as pre-processing, but lifting it into resolution lets variables be solved for directly – for example, choosing the Linux distribution with the freshest system dependencies.

To reduce to the core, we extend the Package Formula Reduction (Def. 4.5.4).

DEFINITION 4.6.3 (Variable Formula Reduction). Given R_Ψ , Δ_Ψ , and r_Ψ , we define a reduction to the core calculus with R , Δ , and r as follows:

- (a) **Packages:** as Def. 4.5.4.a, with $R \supseteq \{(\langle x \rangle, y) \mid x \in X, y \in Y_x\}$ where $\langle x \rangle \in N$ and $y \in V$.
- (b) **Formula:** we extend the encoding function $\mathcal{E}$ (Def. 4.5.4.b) with:

$$\mathcal{E}(p \Delta_\Psi x \omega y) = \{(p, (\langle x \rangle, \{v \in Y_x \mid v \omega_Y y\}))\} \quad \mathcal{E}(p \Delta_\Psi \neg(x \omega y)) = \mathcal{E}(p \Delta_\Psi x \bar{\omega} y)$$

where $\bar{\omega}$ denotes the complement: $\bar{\geq}$ is $<$, $\bar{>}$ is $\leq$, $\bar{=}$ is $\neq$, etc.

- (c) **Dependencies:** $\Delta = \bigcup_{p \Delta_\Psi \psi} \mathcal{E}(p \Delta_\Psi \psi)$ as in Def. 4.5.4.c.

THEOREM 4.6.4 (Soundness). If $S \in \mathcal{S}(\Delta, r)$, then $S_\Psi = S \cap R_\Psi$ and $\sigma(x) = y$ where $(\langle x \rangle, y) \in S$, arbitrary otherwise, are a valid resolution and assignment in $S_\Psi(\Delta_\Psi, r_\Psi)$.

THEOREM 4.6.5 (Completeness). If $(S_\Psi, \sigma) \in S_\Psi(\Delta_\Psi, r_\Psi)$, define the witness set $W^c(\psi)$ for $c \in \{\top, \perp\}$ as in Thm. 4.5.6, extended with $W^c(x \omega y) = W^c(\neg(x \omega y)) = \emptyset$ and using $(S_\Psi, \sigma) \models \psi$ for satisfaction tests.

Then $S = S_\Psi \cup \bigcup_{p \Delta_\Psi \psi} \begin{cases} W^\top(\psi) & \text{if } p \in S_\Psi \\ W^\perp(\psi) & \text{otherwise} \end{cases} \cup \{(\langle x \rangle, \sigma(x)) \mid x \in X\}$ is a valid resolution in $\mathcal{S}(\Delta, r)$.

4.7 Virtual Packages

Some package managers support dependencies on *virtual packages* (§2.2.11) that do not exist as real packages, but are instead provided by a real package. We extend the core with a provides relation.

DEFINITION 4.7.1 (Virtual Package Provides). We define provides $\Pi \subseteq (N \times V) \times (N \times (V \cup \{\top\}))$ as a relation where an element $p\Pi(n, v)$ denotes that package p provides a package name $n \in N$ with version $v \in V \cup \{\top\}$, where $\top$ matches any version. We define $v \in_{\top} vs \iff v \in vs \cup \{\top\}$. The name n is *real* if it appears in R_{Π} , and is otherwise *virtual*.

We write Δ_{Π} for dependencies (Def. 3.1.2); a dependee name need not be real, since a virtual name is satisfied by a provider rather than a real package.

DEFINITION 4.7.2 (Virtual Package Resolution). Given dependencies Δ_{Π} , provides Π , and root r_{Π} , a resolution $S_{\Pi} \subseteq R_{\Pi}$ and provider relation $\rho \subseteq S_{\Pi} \times N \times S_{\Pi} - \text{read } ((m, u), n, p) \in \rho \text{ as } (m, u) \text{ is } p\text{'s provider for name } n -$ are valid if:

- (a) **Root Inclusion:** $r_{\Pi} \in S_{\Pi}$, as in Def. 3.1.3.a.
- (b) **Virtual Dependency Closure:**

$$\begin{aligned} \forall p \in S_{\Pi}. p\Delta_{\Pi}(n, vs) \implies & (\exists v \in vs. (n, v) \in S_{\Pi}) \\ & \vee (\exists! (m, u) \in S_{\Pi}. \exists v \in_{\top} vs. (m, u)\Pi(n, v) \wedge ((m, u), n, p) \in \rho) \end{aligned}$$

Each dependency is satisfied by a compatible package or a unique provider, witnessed by ρ .

- (c) **Version Uniqueness:** $\forall (n, v), (n, v') \in S_{\Pi}. v = v'$, as in Def. 3.1.3.c.

We write $S_{\Pi}(\Delta_{\Pi}, \Pi, r_{\Pi}) \ni (S_{\Pi}, \rho)$ for all resolution-provider pairs of r_{Π} in Δ_{Π} and Π .

We reduce the Virtual Package Calculus to the core calculus using intermediate package names with versions that encode which provider is selected. As with the parent relation (§4.2), the provider relation ρ and the intermediate packages of the reduction below determine one another (Thms 4.7.4 and 4.7.5). Fig. 10 shows an instance of the Virtual Package Calculus reduced to the core.

DEFINITION 4.7.3 (Virtual Package Reduction). Given R_{Π} , Δ_{Π} , Π , and r_{Π} , we define a reduction to the core calculus with R , Δ , and r . For each dependency $p\Delta_{\Pi}(n, vs)$, define a predicate for whether the dependee has a provider as

$$\text{prov}(n, vs) \iff \exists q, v \in_{\top} vs. q\Pi(n, v)$$

and collect versions of the intermediate name $\langle p, n \rangle$ as

$$us = \{\langle m, w \rangle \mid (m, w)\Pi(n, v), v \in_{\top} vs\} \cup \{\langle n, u \rangle \mid u \in vs, (n, u) \in R_{\Pi}\}$$

with $\langle p, n \rangle \in N$ and $\langle m, w \rangle, \langle n, u \rangle \in V$.

- (a) **Packages:** $r = r_{\Pi}$, and each package that can satisfy a dependency on n – a provider of n , or a real package named n – is exposed as a version of the intermediate name $\langle p, n \rangle$:

$$R = R_{\Pi} \cup \bigcup_{\substack{p\Delta_{\Pi}(n, vs) \\ \text{prov}(n, vs)}} \{(\langle p, n \rangle, w) \mid w \in us\}$$

- (b) **Dependencies:** a dependency with no provider reduces to a direct core dependency; otherwise to a dependency on the intermediate name $\langle p, n \rangle$, whose versions each depend on the corresponding provider or real package:

$$\Delta = \bigcup_{p\Delta_{\Pi}(n, vs)} \begin{cases} \{(p, (n, vs))\} & \text{if } \neg \text{prov}(n, vs) \\ \{(p, (\langle p, n \rangle, us))\} \cup \{((\langle p, n \rangle, \langle m, w \rangle), (m, \{w\})) \mid \langle m, w \rangle \in us\} & \text{otherwise} \end{cases}$$

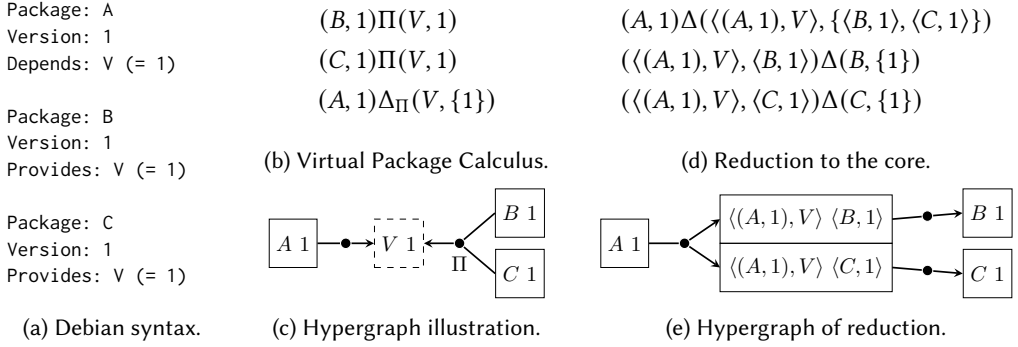


Fig. 10. Virtual Package Calculus instance reduced to the core calculus.

THEOREM 4.7.4 (Soundness). If $S \in \mathcal{S}(\Delta, r)$, then $S_\Pi = S \cap R_\Pi$ and

$$\rho = \bigcup_{p \in \Delta_\Pi(n, vs)} \left\{ ((m, w), n, p) \mid \begin{array}{l} p \in S_\Pi, (m, w) \Pi(n, v), v \in_\top vs, \\ ((p, n), \langle m, w \rangle) \in S \end{array} \right\}$$

are a valid resolution and provider relation in $\mathcal{S}_\Pi(\Delta_\Pi, \Pi, r_\Pi)$.

THEOREM 4.7.5 (Completeness). If $(S_\Pi, \rho) \in \mathcal{S}_\Pi(\Delta_\Pi, \Pi, r_\Pi)$, then

$$S = S_\Pi \cup \bigcup_{\substack{p \in \Delta_\Pi(n, vs) \\ p \in S_\Pi, \text{ prov}(n, vs)}} \{(\langle p, n \rangle, c(p, n))\}$$

is a valid resolution in $\mathcal{S}(\Delta, r)$, where, given $p \in \Delta_\Pi(n, vs)$, $c(p, n)$ selects a provider through ρ if possible and the direct version otherwise:

$$c(p, n) = \begin{cases} \langle m, w \rangle & \text{for some } (m, w) \in S_\Pi \text{ with } \exists v \in_\top vs. (m, w) \Pi(n, v), ((m, w), n, p) \in \rho \\ \langle n, u \rangle & \text{otherwise, with } u \in vs, (n, u) \in S_\Pi \end{cases}$$

5 Package Managers, à la Carte

We return to the motivating example of §1, using four package managers for a single project. Dependencies between these ecosystems are unversioned; the binary will fail if APT has not been used to install the required C libraries and drivers from the Debian repository at compatible versions. For instance, the GPU bindings pip installs must match the kernel driver APT provides – a constraint neither resolver can express. Build sandboxing prevents one package manager from invoking another, so opam cannot call Cargo during a sandboxed build. This leads to ad hoc solutions, such as opam invoking a system package manager to install dependencies via its depext mechanism [61]. Another workaround is vendoring dependees across ecosystems – copying the source code into a project’s version control system – so that versions are fixed and dependencies are available inside sandboxed build environments. Similarly, containerisation technologies like Docker [40, 41] are used to sidestep the problem by bundling all dependencies together. To avoid vendoring or bundling in containers, packages can be duplicated between ecosystem-specific package formats, such as Rust projects packaged in opam. But these workarounds carry serious costs: vendored and duplicated packages require ongoing maintenance when new versions are released; and the dependencies they introduce are hidden from package managers, rendering security vulnerability tracking incomplete.

The Package Calculus offers an alternative. As the core calculus (§3.1) is expressive enough to model each axis of divergence in dependency expression (§4), a project spanning multiple ecosystems can be resolved by reducing each ecosystem's dependencies to the core and solving the combined instance with a single *polyglot resolver*. A dependency that crosses between ecosystems is then an ordinary core dependency. Realising this requires modelling real package managers, which combine several extensions rather than using them in isolation; we show how to compose calculi and their reductions (§5.1).

More ambitiously, the core could serve as an intermediate representation for *translating* dependencies from one ecosystem's language into another, routing n^2 direct translators through $2n$ reductions and liftings (§5.2). This direction is more speculative: it requires inverting the reductions. The reduction of a single-extension instance to the core can be inverted: decoding the synthetic names recovers an equivalent extended instance. Inverting the encoding of composed extensions, whose synthetic structures interleave, follows no known general principle.

5.1 Composition of Extensions

The reductions defined in §4 form a toolkit for constructing the calculi and reductions of combined extensions, to model real package managers. Cargo, for example, uses both concurrent versions (§4.2) and features (§4.4).

5.1.1 Composing Features and Concurrent Versions. Composing the two gives the *Concurrent Feature Package Calculus*. As in the Feature Package Calculus (§4.4), each selected package is paired with a set of features, and as in the Concurrent Package Calculus (§4.2), distinct granularity classes of a name may coexist. The conditions below merge the two – a parameterised (Δ_f) or additional (Δ_a) dependency must select a unique granular version of its dependee, while the feature and granularity invariants of both extensions are preserved.

DEFINITION 5.1.1 (Concurrent Feature Resolution). Given parameterised dependencies Δ_f , additional dependencies Δ_a , granularity g , and root r_{CF} with no features, a resolution $S_{CF} \subseteq R_{CF} \times \mathcal{P}(F)$ and parent relation $\pi \subseteq (N \times V) \times (N \times V)$ are valid if:

(a) **Root Inclusion:** $(r_{CF}, \emptyset) \in S_{CF}$

(b) **Parameterised and Additional Parent Closure:** define

$$sel(p, n, vs, fs) \iff \exists! v \in vs. \exists fs' \supseteq fs. ((n, v), fs') \in S_{CF} \wedge ((n, v), p) \in \pi$$

a predicate for whether, when p depends on n with versions vs and features fs , exactly one version of n in vs is selected with at least the required features, with the parent recorded in π .

$$\begin{aligned} \forall (p, fs_p) \in S_{CF}. p\Delta_f(n, vs, fs) &\implies sel(p, n, vs, fs) \\ \forall (p, fs_p) \in S_{CF}, f \in fs_p. (p, f)\Delta_a(n, vs, fs) &\implies sel(p, n, vs, fs) \end{aligned}$$

(c) **Parent Relation Functionality:**

$$\forall ((n, v), p), ((n, v'), p) \in \pi. v = v'$$

A depender's Δ_f and Δ_a dependencies must agree on a single version of a name.

(d) **Version Granularity:** $\forall ((n, v), fs), ((n, v'), fs') \in S_{CF}. v \neq v' \implies g(v) \neq g(v')$

(e) **Feature Unification:** $\forall ((n, v), fs), ((n, v), fs') \in S_{CF}. fs = fs'$

(f) **Support:** $\forall (p, fs) \in S_{CF}, f \in fs. (p, f) \in support$

We write $S_{CF}(\Delta_f, \Delta_a, g, r_{CF}) \ni (S_{CF}, \pi)$ for all resolution-parent pairs of r_{CF} in Δ_f and Δ_a under g .

DEFINITION 5.1.2 (Concurrent Feature Reduction). Given R_{CF} , Δ_f , Δ_a , $support$, g , and r_{CF} , we define a reduction to the core calculus with R , Δ , and r by combining the Feature Reduction (Def. 4.4.4) and Concurrent Reduction (Def. 4.2.3). For each $(n, v)\Delta_f(m, vs, fs)$ and $((n, v), f)\Delta_a(m, vs, fs)$, write $i = \langle n, v, m \rangle$ for the intermediate name; in the helpers below, the argument j is a per-feature intermediate name in which f' is bound by each $f' \in fs$ comprehension.

(a) **Packages:** $r = \langle \langle n, g(v) \rangle, v \rangle$ where $r_{CF} = (n, v)$. Writing

$$P(j) = \{(i, u) \mid u \in vs\} \cup \{(j, u) \mid f' \in fs, u \in vs\},$$

the real packages are

$$\begin{aligned} R = & \bigcup_{(n,v) \in R_{CF}} \{(\langle n, g(v) \rangle, v)\} \cup \bigcup_{\substack{(n,v) \in R_{CF} \\ ((n,v), f) \in support}} \{(\langle \langle n, f \rangle, g(v) \rangle, v)\} \\ & \cup \bigcup_{(n,v)\Delta_f(m,vs,fs)} P(\langle n, v, m, f' \rangle) \cup \bigcup_{((n,v),f)\Delta_a(m,vs,fs)} P(\langle n, v, f, m, f' \rangle) \end{aligned}$$

(b) **Dependencies:** for a granular dependner p and per-feature intermediate name j ,

$$\begin{aligned} D(p, j) = & \{(p, (i, vs))\} \cup \{((i, u), (\langle m, g(u) \rangle, \{u\})) \mid u \in vs\} \\ & \cup \bigcup_{f' \in fs} \left(\{(p, (j, vs))\} \cup \{((j, u), (\langle \langle m, f' \rangle, g(u) \rangle, \{u\})), ((j, u), (i, \{u\}))) \mid u \in vs\} \right) \end{aligned}$$

the dependencies are

$$\begin{aligned} \Delta = & \bigcup_{\substack{(n,v) \in R_{CF} \\ ((n,v), f) \in support}} \{(\langle \langle \langle n, f \rangle, g(v) \rangle, v \rangle, (\langle n, g(v) \rangle, \{v\})))\} \\ & \cup \bigcup_{(n,v)\Delta_f(m,vs,fs)} D(\langle \langle n, g(v) \rangle, v \rangle, \langle n, v, m, f' \rangle) \\ & \cup \bigcup_{((n,v),f)\Delta_a(m,vs,fs)} D(\langle \langle \langle n, f \rangle, g(v) \rangle, v \rangle, \langle n, v, f, m, f' \rangle) \end{aligned}$$

THEOREM 5.1.3 (Soundness). If $S \in \mathcal{S}(\Delta, r)$ for (R, Δ, r) produced by Def. 5.1.2, then

$$\begin{aligned} S_{CF} = & \{((n, v), \{f \mid (\langle \langle n, f \rangle, g(v) \rangle, v) \in S\}) \mid (\langle n, g(v) \rangle, v) \in S\} \\ \pi = & \{((m, u), (n, v)) \mid (\langle n, v, m \rangle, u) \in S\} \end{aligned}$$

are a valid resolution and parent relation in $\mathcal{S}_{CF}(\Delta_f, \Delta_a, g, r_{CF})$.

THEOREM 5.1.4 (Completeness). If $(S_{CF}, \pi) \in \mathcal{S}_{CF}(\Delta_f, \Delta_a, g, r_{CF})$, define

$$active = ((n, v), _) \in S_{CF}, u \in vs, fs \subseteq fs', ((m, u), fs') \in S_{CF}, ((m, u), (n, v)) \in \pi$$

a predicate for whether (n, v) selects dependee (m, u) , and collect its intermediate packages as

$$witness(j) = \{(i, u) \mid active\} \cup \{(j, u) \mid f' \in fs, active\}$$

then

$$\begin{aligned} S = & \bigcup_{((n,v),fs) \in S_{CF}} \{(\langle \langle n, g(v) \rangle, v \rangle) \cup \{(\langle \langle n, f \rangle, g(v) \rangle, v) \mid f \in fs\}\} \\ & \cup \bigcup_{(n,v)\Delta_f(m,vs,fs)} witness(\langle n, v, m, f' \rangle) \cup \bigcup_{((n,v),f)\Delta_a(m,vs,fs)} witness(\langle n, v, f, m, f' \rangle) \end{aligned}$$

is a valid resolution in $\mathcal{S}(\Delta, r)$.

B-1/Cargo.toml

```
[dependencies.D]
version = ">=1, <3"
features = ["a"]
```

C-1/Cargo.toml

```
[dependencies.D]
version = ">=2, <4"
features = ["b"]
```

D-1/Cargo.toml

```
[features]
a = ["F/c"]
b = ["F/d"]
[dependencies]
F.version = "1"
F.optional = true
```

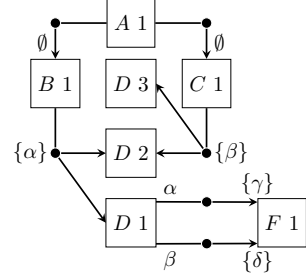
F-1/Cargo.toml

```
[features]
c = []
d = []
```

(a) Cargo syntax.

$$\begin{aligned}
&(A, 1)\Delta_f(B, \{1\}, \emptyset) \\
&(A, 1)\Delta_f(C, \{1\}, \emptyset) \\
&(B, 1)\Delta_f(D, \{1, 2\}, \{\alpha\}) \\
&(C, 1)\Delta_f(D, \{2, 3\}, \{\beta\}) \\
&((D, 1), \alpha)\Delta_a(F, \{1\}, \{\gamma\}) \\
&((D, 1), \beta)\Delta_a(F, \{1\}, \{\delta\})
\end{aligned}$$

(b) Concurrent Feature Package Calculus.



(c) Hypergraph illustration.

Fig. 11. Concurrent Feature Package Calculus instance where $g(v) = v$; D supports feature α at versions 1–2 and β at versions 1–3, and F supports γ and δ at version 1.

Fig. 11 shows a Concurrent Feature Package Calculus instance reduced to the core in Fig. 12 (Appendix F).

5.1.2 Composition Considerations. Reductions satisfy no general composition law: each composition must be defined explicitly, accounting for how the component reductions interact. The concurrent-feature composition shows the method: defining the semantics of the composed extension and then combining reductions. We identify three considerations for composing extensions to model real-world package managers.

Invariant Violation. The conflict reduction (Def. 4.1.3) and the negation encoding of the package formula reduction (Def. 4.5.4) encode mutual exclusion via version uniqueness, which the Concurrent Package Calculus relaxes – breaking the encoding. A correct composition must make the reductions mutually aware: either apply the concurrent reduction first and thread its $\langle n, g(v) \rangle$ renaming through, or assign ϵ granularity to the synthetic packages so they conflict in Def. 4.2.2.c.

Interacting Extensions. Even when invariants are preserved, extensions whose dependency types interact require reductions that are aware of each other's constructs. For example, Portage combines features (§4.4) with package formulae (§4.5), and a formula atom can carry feature annotations: $| | (\text{foo}[\text{bar}] \text{baz})$. A correct composition requires the formula atom (n, vs) (Def. 4.5.1.a) to support features (e.g. (n, vs, fs)), and the feature additional dependency target (n, vs, fs) (Def. 4.4.2.b) to support formulae (e.g. ψ).

Ecosystem Variation. The semantics of individual extensions can also differ subtly between ecosystems. For example, npm supports two different behaviours for peer dependencies depending on the resolver version (§4.3); Cargo features are additive only, while Portage's USE flags can additionally require the absence of a flag (§2.2.8) – effectively a conflict on a feature. In each case the core calculus is the fixed target; composition reconciles the component reductions without extending the core.

5.2 Transpiling Packaging Languages

Polyglot resolution (§5.1) resolves dependencies across ecosystems but leaves them expressed in the core. A more ambitious goal is to *translate* dependencies from one ecosystem’s packaging language into another’s. The core makes this concrete by serving as an intermediate representation: rather than n^2 translators that each understand a pair of dependency languages, one routes through the core, which reduces the problem to a per-ecosystem frontend and backend – $2n$ translators – via the pipeline:

$$L_A \xrightarrow{\text{parse}} \Delta_A \xrightarrow{\text{reduce}} \Delta \xrightarrow{\text{lift}} \Delta_B \xrightarrow{\text{emit}} L_B$$

where L_A and L_B denote the source and target DSLs, and Δ_A , Δ_B the source and target extended-calculus instances.

Casting translation this way isolates where the difficulty lies. Parsing and emitting are the ecosystem-specific frontend and backend, converting between a DSL and a calculus instance; reduction is as in §4. The crux is *lifting*: an operation lift that reconstructs an extended instance from the core by decoding the synthetic names $\langle \cdot \rangle \in N$ that each reduction introduces. For a single extension, lifting is a *retraction* of the reduction, $\text{lift} \circ \text{reduce} = \text{id}$, recovering the original instance up to normalisation; single-extension translation therefore follows from the reductions of §4.

The open problem is lifting a *composed* reduction. A real ecosystem combines several extensions, so its target instance Δ_B must be reconstructed by a composed lifting, and the per-extension liftings do not compose: a composed reduction interleaves the synthetic structures of its constituents – a concurrent-feature instance nests feature names inside granular names (Def. 5.1.2) – so one cannot lift a single extension out in isolation. Like a composed reduction, a composed lifting must be defined explicitly, subject to the considerations of §5.1.2. The calculus therefore reduces cross-ecosystem translation to a single, well-defined problem – inverting composed reductions – which we leave, with the *parse* and *emit* stages, to future work.

6 Related Work

Dependency Resolution. The idea of using a formalism for package versioning dates back to the Optimal Package Install/Uninstall Manager (Opium) [65] in 2007, which adapted Debian’s apt-get to use a SAT solver for dependency resolution. Opium’s formalism is close to our Conflict Package Calculus; it encodes Debian’s dependencies and conflicts in a SAT problem, expanding version formulae to sets. We describe how a resolver can prioritise fresher resolutions (Def. C.5); Opium goes further by optimising an objective function over resolutions via pseudo-Boolean and integer linear programming solvers, which it uses to define the install and uninstall problems.

Later work on the Common Upgradeability Description Format (CUDF) [63] generalised the solver interface into a file format that could be consumed by specialised package resolvers. One such early resolver used answer-set programming (ASP) [32], which also found use in high-performance computing (HPC) software infrastructure for solving dependencies with support for a rich set of user-specified resolver preferences [30]. CUDF provides a fixed vocabulary of dependency constructs – depends, conflicts, and provides – with integer versions, rich optimisation criteria, and support for upgrade semantics (modelling the currently installed state). This line of work also proposed modular package-manager architectures that delegate constraint solving to pluggable external solvers [2], and developed a formal model of package installation [24] whose conditions – abundance, peace, and flatness – correspond to our dependency closure, conflict avoidance, and version uniqueness. The model notes that the presence of flatness varies between ecosystems – the divergence which our concurrent-version extension (§4.2) formalises. However, the fixed vocabulary of CUDF forces package managers to pre-evaluate or expand constructs that do not fit, and CUDF has not seen wide adoption beyond the Debian and opam ecosystems.

Abate et al. [1] observed in 2020 that while SAT-based dependency solving did find traction among some package manager implementations, reusable components for version solving had not seen wider adoption. The Ecosystems project [27] catalogues resolver algorithms across ecosystems, classifying resolution strategies for over thirty package managers. In recent years, some of the larger package ecosystems, such as npm, have come under severe scaling pressure, and it is becoming increasingly important to revisit the question of how to reuse knowledge across ecosystems. For example, PacSolve’s MaxSMT-based resolver is a drop-in replacement for npm that reduced vulnerabilities in resolved packages and picked fresher and fewer dependees [47, 48]. Approximate solving has also been useful for package selection in bounded-latency situations [37]. However, these systems are all tied to a single ecosystem: none can resolve dependencies that cross ecosystem boundaries. PubGrub [66] is a CDCL-based dependency resolution algorithm created for Pub (Dart’s package manager) that has since been adopted by Bundler (Ruby), Poetry (Python), and Swift Package Manager, among others. The pubgrub-rs Rust implementation [50] encodes concurrent versions and features with the same techniques we formalise in §4.2 and §4.4.

More abstract treatments model dependency metadata itself as a mathematical object – a general event structure [8], or a *dependency structure with choice* isomorphic to an antimatroid [9] – characterising the space of dependency-closed states rather than the resolution problem we address.

Build Systems. Package management is closely tied to build systems, which are also surprisingly poorly specified [42]. They are sometimes unified in newer toolchains (Table 1), but doing so makes it difficult to compose projects that span ecosystems (consider the difficulty of depending on a Rust library from OCaml code with just two package managers). The Package Calculus specifies dependency resolution as a formalism independent of the underlying build systems, preserving across ecosystems the separation that unified toolchains lose. In the future, creating a composable formal theory of build systems and package managers [5] would let us combine cross-ecosystem codebases to conduct correctness testing across the vast amount of source code published [38].

Software Supply Chain Security. The increasing complexity of cross-ecosystem dependency chains puts pressure on the security of software supply chains, both in the choice of dependees selected for a software project [6, 7, 67] and in vulnerabilities in the many package managers available [12, 14]. Software Bills of Materials (SBOMs) are being rapidly adopted [16] to specify the full set of versioned dependees that go into a given application. However, the same problem of uneven quality across ecosystems applies here, with the JavaScript ecosystem being particularly unpredictable [51]. The Package Calculus could be used to generate SBOMs systematically, reduce transitive trust dependencies across language and OS ecosystems [53], drop resource usage by debloating unnecessary packages [46], and reduce the latency of applying security updates in third-party dependees [52, 54].

7 Conclusion

We surveyed package managers, identifying a shared core and the axes along which they diverge (§2). We formalised this core as the Package Calculus (§3). We modelled each axis of divergence as an extension with a reduction back to the core (§4). We showed how extensions compose to model package managers that combine several of them, providing the foundation for polyglot resolution across ecosystems, and discussed how the calculus might further serve as an intermediate representation for translation between ecosystems, reducing n^2 translators to $2n$ (§5). Building on this foundation, future work can develop principled cross-ecosystem tooling (§1) via translators between ecosystem DSLs, polyglot resolvers, and ecosystem-specific resolvers verified against the calculus’s specifications.

Acknowledgments

We thank Raito Bezarius, Michael Winston Dales, Kate Deplaix, Roberto Di Cosmo, Mark Elvers, Arjun Guha, Sadiq Jaffer, Ranjit Jhala, Artsiom Karakin, Shriram Krishnamurthi, Thomas Leonard, Jon Ludlam, Andrew Nesbitt, Simon Peyton Jones, Swapnil Raj, James Rowbottom, Jacob Trevor, and Garrett Wollman for their contributions and comments. Simon Peyton Jones in particular spent much time giving us layout and notation suggestions that greatly improved the presentation. We also thank the anonymous ICFP reviewers for their feedback.

Funding

This research was supported by unrestricted donations from the Huawei HiSilicon Scholarship, Tarides, and Jane Street.

Data-Availability Statement

The Lean 4 mechanisation of the theorems in this paper is available as an archive on Zenodo [33]. The latest version is maintained at <https://github.com/RyanGibb/package-calculus/>.

A A Short History of Package Managers

The original Unix and pre-Unix operating systems had no concept of automated package management; the operating system was distributed with libraries and utilities, and third-party software had to be built and installed manually. System V Release 4.0, announced in 1988, included the ‘packaging tool’ `pkgadd` and related commands to automate the process of installing packages onto the system from a source such as a tape, CD-ROM, or networked filesystem, as well as updating and removing them [62]. It verified that dependees were installed, but did not automatically select and install compatible versions. Other commercial Unix vendors created similar systems in the following years, as did early Linux (1991) distributions: Slackware’s `pkgtool` (1993), Debian’s `dpkg` (1994), and Red Hat’s `rpm` (1995). FreeBSD’s Ports (1994) introduced automatic dependee installation, relying on regular expressions on package names to ensure application binary interface (ABI) compatibility.

Early programming-language package managers emerged in the form of archive networks for distributing software over the Internet with the Comprehensive T_EX Archive Network (CTAN) in 1992 [34], the Comprehensive Perl Archive Network (CPAN) in 1995 [21], and the Comprehensive R Archive Network (CRAN) in 1997 [35]. The Perl module `CPAN.pm` and CRAN’s `install.packages()` allowed for the automatic installation of dependees, but always selected the latest indexed version of each, performing no version solving. CTAN had no native automated installer; package management was later provided by T_EX Live’s `tlmgr` (2008), which likewise installed only the latest versions, though T_EX Live mitigated incoherence by distributing packages as a curated annual snapshot rather than a continuously updated live index.

Debian’s Advanced Packaging Tool (APT) (1998), building atop the existing `dpkg`, resolved compatible versions of dependencies based on version constraints and automatically installed them. The Yellowdog Updater, Modified (YUM) (2002), building on `rpm`, brought similar capabilities to Red Hat and other RPM-based distributions, and was later superseded by Dandified YUM (DNF) (2013). Dependency resolution in these systems has been shown to be NP-complete [23], including for upgrades [3]. Slackware is a notable exception to the norm in dependency resolution; official mechanisms do not provide any support for dependency tracking, instead relying on the user to manually install dependees of a package. At the other extreme, Nix (2003) [26] opted out of version solving by requiring exact versions of dependees, deploying packages at paths containing cryptographic hashes (Appendix D).

By the mid-2000s, automatic dependency resolution with version constraints had become standard practice. Package managers began to be built as monolithic systems expected of any new operating system or programming language. This evolutionary pattern extends beyond operating systems and programming languages: plugin ecosystems for web browsers, text editors, and applications have similarly progressed through centralised archives, automatic installation, and dependency resolution. More recently, tools originally designed for CI, chart templating, and infrastructure-as-code – GitHub Actions, Helm, and Terraform – have grown transitive dependency trees and inherited the problems of package management whether or not they were designed for it [43]. As these package managers have developed, the semantics of their dependencies have diverged (§2.2), each introducing functionality to address its ecosystem’s specific needs.

B Thm. 3.1.4: DEPENDENCYRESOLUTION Complexity

PROOF. We establish both membership and hardness:

Membership in NP: a candidate resolution S can be verified in polynomial time by checking root inclusion (Def. 3.1.3.a) in $O(|S|)$ time, dependency closure (Def. 3.1.3.b) in $O(|S| \cdot |\Delta|)$ time, and version uniqueness (Def. 3.1.3.c) in $O(|S|^2)$ time.

NP-Hardness: we give a polynomial-time reduction from 3-SAT, which is well-known to be NP-complete [17]. Given a 3-CNF formula with variables $x_1, \dots, x_n$ and clauses $c_1, \dots, c_m$ (each $c_j = l_1 \vee l_2 \vee l_3$), we build a resolution instance with root $r = (q, \epsilon)$ and packages

$$R = \{(q, \epsilon)\} \cup \bigcup_i \{(x_i, \top), (x_i, \perp)\} \cup \bigcup_j \{(c_j, l_1), (c_j, l_2), (c_j, l_3)\},$$

whose dependencies are, for every clause c_j ,

$$(q, \epsilon) \Delta (c_j, \{l_1, l_2, l_3\}) \quad \text{and, for each } k \in \{1, 2, 3\}, \quad (c_j, l_k) \Delta (\text{var}(l_k), \{\text{pol}(l_k)\}),$$

where $\text{var}(l)$ is the variable of literal l , and $\text{pol}(l)$ is $\top$ for positive literals and $\perp$ for negative.

The construction runs in $O(n + m)$ time. We prove the equivalence:

($\Rightarrow$) Given a satisfying assignment σ , for each clause $c_j = l_1 \vee l_2 \vee l_3$ let $k_j = \min\{k \in \{1, 2, 3\} \mid \sigma(l_k) = \top\}$. Then $S = \{r\} \cup \{(c_j, l_{k_j}) \mid 1 \leq j \leq m\} \cup \{(x_i, \sigma(x_i)) \mid 1 \leq i \leq n\}$ satisfies all dependencies by construction and maintains version uniqueness.

($\Leftarrow$) Any resolution $S \in \mathcal{S}(\Delta, r)$ induces a satisfying assignment $\sigma(x_i) = v$ where $(x_i, v) \in S$ (arbitrary otherwise), since each clause package c_j requires at least one of its literals to be satisfied.

As DEPENDENCYRESOLUTION is in NP and is NP-hard, we conclude that it is NP-complete. $\square$

C SAT-Based Dependency Resolution

Modern package managers [63] often employ SAT-based resolution to solve dependency problems. We encode DEPENDENCYRESOLUTION as SAT as follows.

DEFINITION C.1 (Package Calculus SAT Encoding). Given real packages R , dependency relation Δ , and root r , where X_p denotes inclusion of package p , we define an encoding to the SAT instance $\Sigma := (X_r) \wedge \Sigma_\Delta \wedge \Sigma_\neq$ where:

$$\Sigma_\Delta := \bigwedge_{p \Delta (n, vs)} \left(\neg X_p \vee \bigvee_{\substack{v \in vs \\ (n, v) \in R}} X_{(n, v)} \right) \quad \Sigma_\neq := \bigwedge_{\substack{(n, v), (n, v') \in R \\ v <_v v'}} (\neg X_{(n, v)} \vee \neg X_{(n, v')})$$

Note that for a package name $n \in N$ with $k = |\{v \mid (n, v) \in R\}|$ versions, this adds $O(k^2)$ clauses, but a SAT solver with an at-most-one primitive can reduce this to $O(k)$ [44].

THEOREM C.2 (Soundness). For any assignment $\sigma : \{X_p \mid p \in N \times V\} \rightarrow \{\top, \perp\}$ that satisfies Σ , the set S defined by $\{p \in R \mid \sigma(X_p) = \top\}$ is a valid resolution in $\mathcal{S}(\Delta, r)$.

THEOREM C.3 (Completeness). For any $S \in \mathcal{S}(\Delta, r)$, the assignment $\sigma(X_p) = \top \iff p \in S$ satisfies Σ .

Example C.4. Consider the dependencies from Fig. 3 with query $Q = \{(A, \{1\})\}$. The SAT encoding produces $\Sigma = X_r \wedge \Sigma_\Delta \wedge \Sigma_\#$ where:

$$\begin{aligned}\Sigma_\Delta &:= (\neg X_r \vee X_{A1}) \wedge (\neg X_{A1} \vee X_{B1}) \wedge (\neg X_{A1} \vee X_{C1}) \wedge (\neg X_{B1} \vee X_{D1} \vee X_{D2}) \wedge (\neg X_{C1} \vee X_{D2} \vee X_{D3}) \\ \Sigma_\# &:= (\neg X_{D1} \vee \neg X_{D2}) \wedge (\neg X_{D1} \vee \neg X_{D3}) \wedge (\neg X_{D2} \vee \neg X_{D3})\end{aligned}$$

DEFINITION C.5 (Resolution Ordering). A *resolution ordering* is a partial order $\leq_S$ on $\mathcal{S}(\Delta, r)$ where for $S_1, S_2 \in \mathcal{S}(\Delta, r)$, we say $S_1 \leq_S S_2 \iff \forall (n, v_2) \in S_2. \exists v_1 \in V. (n, v_1) \in S_1 \wedge v_1 \leq_v v_2$. Every package name in S_2 has a version at least as great as that in S_1 according to $\leq_v$ (Def. 3.2.1).

Maximal elements of $\leq_S$ represent the ‘freshest’ possible resolutions, but a maximum resolution may not exist. For example, given $(A, 1)\Delta(B, \{1, 2\})$, $(A, 1)\Delta(C, \{1, 2\})$, and $(B, 2)\Delta(C, \{1\})$, the query $Q = \{(A, \{1\})\}$ has maximal resolutions $\{r, (A, 1), (B, 1), (C, 2)\}$ and $\{r, (A, 1), (B, 2), (C, 1)\}$ but no maximum resolution. Finding a valid resolution (Def. 3.1.3) does not depend on version ordering, but an algorithm can prioritise higher versions (Def. C.6).

DEFINITION C.6 (Ordered SAT Encoding). Given the Package Calculus SAT encoding (Def. C.1), we refine the dependency implication clauses using version ordering $\leq_v$.

$$\Sigma_\Delta := \bigwedge_{p\Delta(n, vs)} (\neg X_p \vee X_{(n, v_1)} \vee \dots \vee X_{(n, v_k)}) \text{ where } \{v_1, \dots, v_k\} = \{v \in vs \mid (n, v) \in R\}, v_1 \geq_v \dots \geq_v v_k$$

This encoding ensures that in each dependency clause, higher versions appear first. SAT solvers prioritising leftmost literals will preferentially assign true to earlier (higher-version) literals. The ordering transforms version preference into a search heuristic without altering satisfiability.

D Singular Dependencies

Singular dependencies only restrict the core calculus, rather than extending it. Nix, Guix (Various) [18], Unison, and Lake (Lean 4) support only a single, exact version of a package name as a dependee, delegating the responsibility of version solving from the package manager to the packager. Where the core calculus expresses a dependency on a *set* of compatible versions and leaves it to the resolver to select among them, singular dependencies require the packager to commit to a specific version at packaging time. The reduction to the core is trivial, but because singular dependencies are *less* expressive than the core, we cannot reduce an instance of the core calculus to singular dependencies.

DEFINITION D.1 (Singular Dependency). We define singular dependencies $\beta \subseteq (N \times V) \times (N \times V)$ as a relation where an element $p\beta d$ denotes a package p expressing a dependency on a package d .

DEFINITION D.2 (Singular Dependency Resolution). Given singular dependencies β and root r_β , a resolution $S_\beta \subseteq R_\beta$ is valid if:

- (a) **Root Inclusion:** $r_\beta \in S_\beta$, as in Def. 3.1.3.a.
- (b) **Dependency Closure:** $\forall p \in S_\beta. p\beta d \implies d \in S_\beta$
- (c) **Version Uniqueness:** $\forall (n, v), (n, v') \in S_\beta. v = v'$, as in Def. 3.1.3.c.

Nix DSL expressions contain conditional logic and feature-like parameters as function arguments when creating derivations. Nix could add support for resolving version constraints as the Package Calculus does, either in the DSL or in the store derivation format, but the semantics of this would need to be defined.

For example, if we can say our Nix derivation depends on a package name with version constraints, can we parameterise this by a feature? Since this feature selection can pull in different dependencies depending on the version of the package selected, this has to be part of the dependency resolution rather than simply a function parameter in the Nix DSL. Even with version-constraint resolution added, such a Nix would need a mechanism to restrict this for ecosystems where it is not possible, for example, when libraries need to link together in a binary.

This explains the proliferation of tools such as `cargo2nix`, `opam-nix`, and `poetry2nix`, doing dependency resolution in ecosystem-specific tooling and locking versions in Nix derivations without any way to express version constraints across ecosystems. In Table 1, Nix is not listed as supporting most extensions, as it does not support them as part of version-constrained dependency resolution.

E Optional Dependencies in Build Graphs

We promised in §2.2.12 that optional dependencies are a build-ordering concern, not a resolution concern, which we now make precise. A valid resolution (Def. 3.1.3) determines *which* packages to install, but not the *order* in which to install them. Package managers like `opam`, as part of their deployment (§2.1.4), must linearise the resolution into a sequence of build actions such that all the dependees of a package are built before the depender itself. This requires a topological sort of the dependency graph induced by the resolution – which is only possible when that graph is acyclic.

In practice, dependency graphs can contain cycles. In Debian, for example, `libgcc-s1` depends on `libc6` (it requires the C library at run time), while `libc6` depends on `libgcc-s1` (it requires the GCC run-time support library). Both packages are valid in a resolution, but no linear installation order can satisfy both ordering constraints simultaneously. APT handles this by breaking cycles at an arbitrary point and relying on `dpkg`'s ability to partially unpack packages before their dependees are fully configured [59]. In `opam`, packages can be marked with the `post` variable, which indicates that the dependee is not required at build time, allowing the cycle to be broken explicitly. Other package managers avoid the problem entirely: `npm` lays out files in nested directories and relies on Node.js's module loader to handle cycles between modules.

DEFINITION E.1 (Build Graph). We formalise this installation ordering as the *build graph*. Given dependencies Δ (Def. 3.1.2) and a resolution S (Def. 3.1.3), the build graph is a directed graph with vertices S and edges $E = \{(p, (n, v)) \mid p \in S, p\Delta(n, vs), (n, v) \in S\}$.

DEFINITION E.2 (Optional Dependency). We define optional dependencies as a relation $O \subseteq (N \times V) \times (N \times \mathcal{P}(V))$. We denote an element of O as $pO(n, vs)$ where package p will use package name n with version $v \in vs$ if (n, v) is in the resolution.

Optional dependees are required before their depender in the build graph (Def. E.1).

$$E = \{(p, (n, v)) \mid p \in S, p\Delta(n, vs), (n, v) \in S\} \cup \{(p, (n, v)) \mid p \in S, pO(n, vs), v \in vs, (n, v) \in S\}$$

Example E.3. Consider,

$$(A, 1)\Delta(B, \{1\}) \quad (A, 1)\Delta(D, \{1\}) \quad (B, 1)\Delta(C, \{1\}) \quad (B, 1)O(D, \{1\})$$

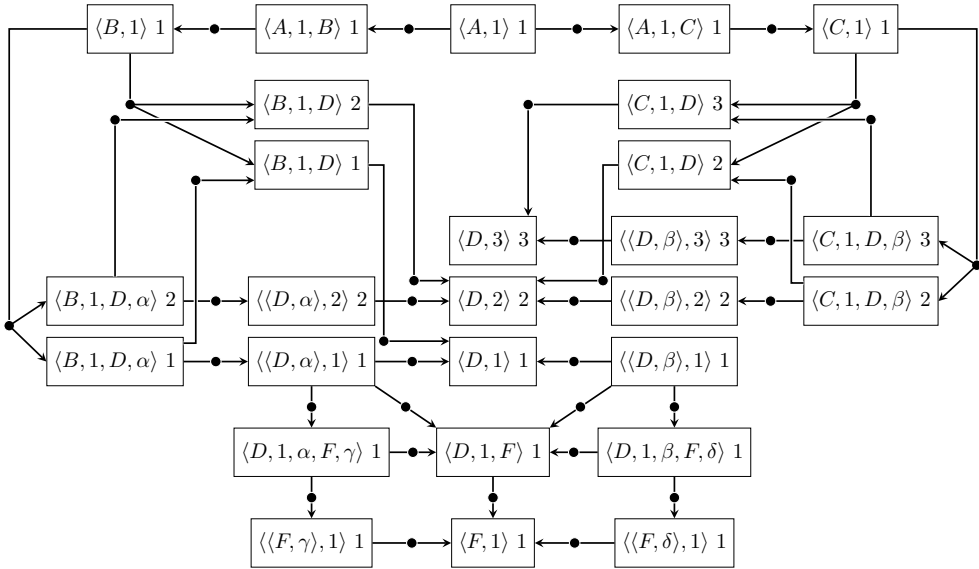
If $Q = \{(B, \{1\})\}$, then $\{r, (B, 1), (C, 1)\} \in \mathcal{S}(\Delta, r)$, and its build graph is $\{(r, (B, 1)), ((B, 1), (C, 1))\}$. If $Q = \{(A, \{1\})\}$, then $\{r, (A, 1), (B, 1), (C, 1), (D, 1)\} \in \mathcal{S}(\Delta, r)$ and $E \ni ((B, 1), (D, 1))$.

We can trivially reduce optional dependencies to the core calculus by omitting them; they have no effect on the set of resolutions $\mathcal{S}(\Delta, r)$. Package B can be resolved regardless of D 's presence. When assembling the build graph, if $pO(n, vs)$ and $(n, v) \in S$ with $v \in vs$, then $(p, (n, v)) \in E$ *always* and there is no mechanism to disable this edge. This model captures `opam`'s behaviour where optional dependencies are purely a build-ordering mechanism with no resolution-time effects.

F Concurrent Feature Package Calculus Reduction

$$\begin{array}{ll}
 \langle\langle A, 1 \rangle, 1 \rangle \Delta \langle\langle A, 1, B \rangle, \{1\}\rangle & \langle\langle C, 1, D, \beta \rangle, 2 \rangle \Delta \langle\langle C, 1, D \rangle, \{2\}\rangle \\
 \langle\langle A, 1, B \rangle, 1 \rangle \Delta \langle\langle B, 1 \rangle, \{1\}\rangle & \langle\langle C, 1, D, \beta \rangle, 3 \rangle \Delta \langle\langle C, 1, D \rangle, \{3\}\rangle \\
 \langle\langle A, 1 \rangle, 1 \rangle \Delta \langle\langle A, 1, C \rangle, \{1\}\rangle & \langle\langle D, \alpha \rangle, 1 \rangle \Delta \langle\langle D, 1 \rangle, \{1\}\rangle \\
 \langle\langle A, 1, C \rangle, 1 \rangle \Delta \langle\langle C, 1 \rangle, \{1\}\rangle & \langle\langle D, \alpha \rangle, 2 \rangle \Delta \langle\langle D, 2 \rangle, \{2\}\rangle \\
 \langle\langle B, 1 \rangle, 1 \rangle \Delta \langle\langle B, 1, D \rangle, \{1, 2\}\rangle & \langle\langle D, \beta \rangle, 1 \rangle \Delta \langle\langle D, 1 \rangle, \{1\}\rangle \\
 \langle\langle B, 1 \rangle, 1 \rangle \Delta \langle\langle B, 1, D, \alpha \rangle, \{1, 2\}\rangle & \langle\langle D, \beta \rangle, 2 \rangle \Delta \langle\langle D, 2 \rangle, \{2\}\rangle \\
 \langle\langle C, 1 \rangle, 1 \rangle \Delta \langle\langle C, 1, D \rangle, \{2, 3\}\rangle & \langle\langle D, \beta \rangle, 3 \rangle \Delta \langle\langle D, 3 \rangle, \{3\}\rangle \\
 \langle\langle C, 1 \rangle, 1 \rangle \Delta \langle\langle C, 1, D, \beta \rangle, \{2, 3\}\rangle & \langle\langle D, \alpha \rangle, 1 \rangle \Delta \langle\langle D, 1, F \rangle, \{1\}\rangle \\
 \langle\langle B, 1, D \rangle, 1 \rangle \Delta \langle\langle D, 1 \rangle, \{1\}\rangle & \langle\langle D, \beta \rangle, 1 \rangle \Delta \langle\langle D, 1, F \rangle, \{1\}\rangle \\
 \langle\langle B, 1, D \rangle, 2 \rangle \Delta \langle\langle D, 2 \rangle, \{2\}\rangle & \langle\langle D, 1, F \rangle, 1 \rangle \Delta \langle\langle F, 1 \rangle, \{1\}\rangle \\
 \langle\langle B, 1, D, \alpha \rangle, 1 \rangle \Delta \langle\langle D, \alpha \rangle, 1 \rangle, \{1\}\rangle & \langle\langle D, \alpha \rangle, 1 \rangle \Delta \langle\langle D, 1, \alpha, F, \gamma \rangle, \{1\}\rangle \\
 \langle\langle B, 1, D, \alpha \rangle, 2 \rangle \Delta \langle\langle D, \alpha \rangle, 2 \rangle, \{2\}\rangle & \langle\langle D, 1, \alpha, F, \gamma \rangle, 1 \rangle \Delta \langle\langle F, \gamma \rangle, 1 \rangle, \{1\}\rangle \\
 \langle\langle B, 1, D, \alpha \rangle, 1 \rangle \Delta \langle\langle B, 1, D \rangle, \{1\}\rangle & \langle\langle D, 1, \alpha, F, \gamma \rangle, 1 \rangle \Delta \langle\langle D, 1, F \rangle, \{1\}\rangle \\
 \langle\langle B, 1, D, \alpha \rangle, 2 \rangle \Delta \langle\langle B, 1, D \rangle, \{2\}\rangle & \langle\langle D, \beta \rangle, 1 \rangle \Delta \langle\langle D, 1, \beta, F, \delta \rangle, \{1\}\rangle \\
 \langle\langle C, 1, D \rangle, 2 \rangle \Delta \langle\langle D, 2 \rangle, \{2\}\rangle & \langle\langle D, 1, \beta, F, \delta \rangle, 1 \rangle \Delta \langle\langle F, \delta \rangle, 1 \rangle, \{1\}\rangle \\
 \langle\langle C, 1, D \rangle, 3 \rangle \Delta \langle\langle D, 3 \rangle, \{3\}\rangle & \langle\langle D, 1, \beta, F, \delta \rangle, 1 \rangle \Delta \langle\langle D, 1, F \rangle, \{1\}\rangle \\
 \langle\langle C, 1, D, \beta \rangle, 2 \rangle \Delta \langle\langle D, \beta \rangle, 2 \rangle, \{2\}\rangle & \langle\langle F, \gamma \rangle, 1 \rangle \Delta \langle\langle F, 1 \rangle, \{1\}\rangle \\
 \langle\langle C, 1, D, \beta \rangle, 3 \rangle \Delta \langle\langle D, \beta \rangle, 3 \rangle, \{3\}\rangle & \langle\langle F, \delta \rangle, 1 \rangle \Delta \langle\langle F, 1 \rangle, \{1\}\rangle
 \end{array}$$

(a) Reduction to the core calculus.



(b) Hypergraph of reduction.

Fig. 12. Reduction of Fig. 11 to the core calculus.

References

- [1] Pietro Abate, Roberto Di Cosmo, Georgios Gousios, and Stefano Zacchiroli. 2020. Dependency Solving Is Still Hard, but We Are Getting Better at It. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, London, ON, Canada, 547–551. doi:10.1109/SANER48275.2020.9054837
- [2] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. 2011. MPM: a modular package manager. In *Proceedings of the 14th International ACM SIGSOFT Symposium on Component Based Software Engineering* (Boulder, Colorado, USA) (CBSE '11). Association for Computing Machinery, New York, NY, USA, 179–188. doi:10.1145/2000229.2000255
- [3] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. 2012. Dependency solving: A separate concern in component evolution management. *Journal of Systems and Software* 85, 10 (Oct. 2012), 2228–2240. doi:10.1016/j.jss.2012.02.018
- [4] aboutcode-org. 2024. univers: a library for parsing and comparing software package versions and version ranges. <https://github.com/aboutcode-org/univers>
- [5] Snorri Agnarsson and M. S. Krishnamoorthy. 1985. Towards a theory of packages. *ACM SIGPLAN Notices* 20, 7 (June 1985), 117–130. doi:10.1145/17919.806833
- [6] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2023. Empirical analysis of security vulnerabilities in Python packages. *Empirical Software Engineering* 28, 3, Article 59 (March 2023), 29 pages. doi:10.1007/s10664-022-10278-4
- [7] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Bram Adams. 2023. On the Discoverability of npm Vulnerabilities in Node.js Projects. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 91 (May 2023), 27 pages. doi:10.1145/3571848
- [8] Gershon Bazerman. 2021. The Semantics of Package Management via Event Structures. arXiv:2107.01542 [cs.LO] <https://arxiv.org/abs/2107.01542>
- [9] Gershon Bazerman, Emilio Minichiello, and Raymond Puzio. 2024. A Mathematical Model of Package Management Systems. arXiv:2302.05417 [math.CT] <https://arxiv.org/abs/2302.05417>
- [10] Jon Bentley. 1986. Programming pearls: little languages. *Commun. ACM* 29, 8 (Aug. 1986), 711–721. doi:10.1145/6424.315691
- [11] Claude Berge. 1970. *Graphes et hypergraphes*. Dunod, Paris, France.
- [12] Aarnav M. Bos. 2023. A Review of Attacks Against Language-Based Package Managers. arXiv:2302.08959 [cs.SE] <https://arxiv.org/abs/2302.08959>
- [13] Cabal Development Team. 2024. The Haskell Cabal. <https://www.haskell.org/cabal/>
- [14] Justin Cappos, Justin Samuel, Scott Baker, and John H. Hartman. 2008. A look in the mirror: attacks on package managers. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) (CCS '08). Association for Computing Machinery, New York, NY, USA, 565–574. doi:10.1145/1455770.1455841
- [15] Paul Chiusano and Rúnar Bjarnason. 2020. Unison: A friendly programming language from the future. <https://www.unison-lang.org/>
- [16] Serena Cofano, Giacomo Benedetti, and Matteo Dell'Amico. 2024. SBOM Generation Tools in the Python Ecosystem: an In-Detail Analysis. arXiv:2409.01214 [cs.CR] <https://arxiv.org/abs/2409.01214>
- [17] Stephen A. Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (Shaker Heights, Ohio, USA) (STOC '71). Association for Computing Machinery, New York, NY, USA, 151–158. doi:10.1145/800157.805047
- [18] Ludovic Courtès. 2013. Functional Package Management with Guix. arXiv:1305.4584 [cs.PL] <https://arxiv.org/abs/1305.4584>
- [19] Russ Cox. 2016. Version SAT. <https://research.swtch.com/version-sat>
- [20] Russ Cox. 2018. Go & Versioning: Minimal Version Selection. <https://research.swtch.com/vgo-mvs>
- [21] CPAN. 1995. Comprehensive Perl Archive Network. <https://www.cpan.org/>
- [22] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2018. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24, 1 (Feb. 2018), 381–416. doi:10.1007/s10664-017-9589-y
- [23] Roberto Di Cosmo. 2005. *EDOS Deliverable WP2-D2.1: Report on Formal Management of Software Dependencies*. Technical Report hal-00697463. HAL / INRIA. <https://hal.science/hal-00697463>
- [24] Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. 2013. Formal Aspects of Free and Open Source Software Components. In *Formal Methods for Components and Objects (FMCO 2012)*, Elena Giachino, Reiner Hähnle, Frank S. de Boer, and Marcello M. Bonsangue (Eds.). Lecture Notes in Computer Science, Vol. 7866. Springer, Berlin, Heidelberg, 216–239. doi:10.1007/978-3-642-40615-7_8
- [25] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. 2019. Dependency Versioning in the Wild. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, Montreal, Canada, 349–359. doi:10.1109/msr.2019.00061

- [26] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. 2004. Nix: A Safe and Policy-Free System for Software Deployment. In *Proceedings of the 18th USENIX Conference on System Administration* (Atlanta, GA) (LISA '04). USENIX Association, USA, 79–92. <https://www.usenix.org/conference/lisa-04/nix-safe-and-policy-free-system-software-deployment>
- [27] Ecosyste.ms. 2025. Package Manager Resolvers. Ecosyste.ms project. <https://github.com/ecosyste-ms/package-manager-resolvers>
- [28] Mark Florisson and Alan Mycroft. 2016. Towards a Theory of Packages. Unpublished draft, University of Cambridge.
- [29] Yogya Gamage, Deepika Tiwari, Martin Monperrus, and Benoit Baudry. 2026. The design space of lockfiles across package managers. *Empirical Software Engineering* 31, 3, Article 63 (Jan. 2026), 32 pages. doi:10.1007/s10664-025-10789-w
- [30] Todd Gamblin, Massimiliano Culp, Gregory Becker, and Sergei Shudler. 2022. Using Answer Set Programming for HPC Dependency Solving. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Dallas, TX, USA, 1–15. doi:10.1109/SC41404.2022.00040
- [31] Todd Gamblin, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral. 2015. The Spack package manager: bringing order to HPC software chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) (SC '15). Association for Computing Machinery, New York, NY, USA, Article 40, 12 pages. doi:10.1145/2807591.2807623
- [32] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2011. Multi-Criteria Optimization in Answer Set Programming. In *Technical Communications of the 27th International Conference on Logic Programming (ICLP'11) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 11)*, John P. Gallagher and Michael Gelfond (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1–10. doi:10.4230/LIPIcs.ICLP.2011.1
- [33] Ryan T. Gibb. 2026. Package Managers à la Carte: Lean 4 Mechanisation. Zenodo. doi:10.5281/zenodo.21373302
- [34] George D. Greenwade. 1993. The Comprehensive T_EX Archive Network (CTAN). *TUGboat* 14, 3 (1993), 342–351.
- [35] Kurt Hornik. 2012. The Comprehensive R Archive Network. *Wiley Interdisciplinary Reviews: Computational Statistics* 4, 4 (2012), 394–398. doi:10.1002/wics.1212
- [36] Paul Hudak. 1996. Building domain-specific embedded languages. *ACM Comput. Surv.* 28, 4es (Dec. 1996), 196–es. doi:10.1145/242224.242477
- [37] Alexey Ignatiev, Mikoláš Janota, and Joao Marques-Silva. 2014. Towards efficient optimization in package management systems. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) (ICSE 2014). Association for Computing Machinery, New York, NY, USA, 745–755. doi:10.1145/2568225.2568306
- [38] Christian Macho, Fabian Oraz, and Martin Pinzger. 2024. DValidator: An approach for validating dependencies in build configurations. *Journal of Systems and Software* 209, Article 111916 (2024), 9 pages. doi:10.1016/j.jss.2023.111916
- [39] Anil Madhavapeddy. 2013. The Year in OCaml Labs. doi:10.59350/dxt23-hxf18
- [40] Anil Madhavapeddy, David J. Scott, and Justin Cormack. 2026. A Decade of Docker Containers. *Commun. ACM* 69, 3 (March 2026), 56–65. doi:10.1145/3761803
- [41] Anil Madhavapeddy, David J. Scott, Patrick Ferris, Ryan T. Gibb, and Thomas Gazagnaire. 2025. Functional Networking for Millions of Docker Desktops (Experience Report). *Proc. ACM Program. Lang.* 9, ICFP, Article 256 (Aug. 2025), 19 pages. doi:10.1145/3747525
- [42] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build systems à la carte. *Proc. ACM Program. Lang.* 2, ICFP, Article 79 (July 2018), 29 pages. doi:10.1145/3236774
- [43] Andrew Nesbitt. 2026. If It Quacks Like a Package Manager. <https://nesbitt.io/2026/03/08/if-it-quacks-like-a-package-manager.html>
- [44] Van-Hau Nguyen, Van-Quyet Nguyen, Kyungbaek Kim, and Pedro Barahona. 2021. Empirical Study on SAT-Encodings of the At-Most-One Constraint. In *The 9th International Conference on Smart Media and Applications* (Jeju, Republic of Korea) (SMA 2020). Association for Computing Machinery, New York, NY, USA, 470–475. doi:10.1145/3426020.3426170
- [45] package-url. 2024. vers: a mostly universal version range specifier. <https://github.com/package-url/vers-spec>
- [46] Pardis Pashakhanloo, Aravind Machiry, Hyonyoung Choi, Anthony Canino, Kihong Heo, Insup Lee, and Mayur Naik. 2022. PacJam: Securing Dependencies Continuously via Package-Oriented Debloating. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security* (Nagasaki, Japan) (ASIA CCS '22). Association for Computing Machinery, New York, NY, USA, 903–916. doi:10.1145/3488932.3524054
- [47] Donald Pinckney. 2024. *Improving Dependency Management via Formal Semantics*. Ph.D. Dissertation. Northeastern University. doi:10.17760/D20698938
- [48] Donald Pinckney, Federico Cassano, Arjun Guha, Jonathan Bell, Massimiliano Culp, and Todd Gamblin. 2023. Flexible and Optimal Dependency Management via Max-SMT. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, Melbourne, Australia, 1418–1429. doi:10.1109/ICSE48619.2023.00124
- [49] Tom Preston-Werner. 2013. Semantic Versioning 2.0.0. <https://semver.org/>
- [50] pubgrub-rs Contributors. 2020. pubgrub-rs: PubGrub version solving algorithm implemented in Rust. <https://github.com/pubgrub-rs/pubgrub>

- [51] Md Fazle Rabbi, Arifa Islam Champa, Costain Nachuma, and Minhaz Fahim Zibran. 2024. SBOM Generation Tools Under Microscope: A Focus on The npm Ecosystem. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing* (Avila, Spain) (SAC '24). Association for Computing Machinery, New York, NY, USA, 1233–1241. doi:10.1145/3605098.3635927
- [52] Kristiina Rahkema and Dietmar Pfahl. 2022. Analysing the Relationship Between Dependency Definition and Updating Practice When Using Third-Party Libraries. In *Product-Focused Software Process Improvement*, Davide Taibi, Marco Kuhrmann, Tommi Mikkonen, Jil Klünder, and Pekka Abrahamsson (Eds.). Springer International Publishing, Cham, 90–107. doi:10.1007/978-3-031-21388-5_7
- [53] Martin Schwaighofer, Michael Roland, and René Mayrhofer. 2024. Extending Cloud Build Systems to Eliminate Transitive Trust. In *Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '24)*. ACM, Salt Lake City, UT, USA, 11 pages. doi:10.1145/3689944.3696169
- [54] Jacob Stringer, Amjed Tahir, Kelly Blincoe, and Jens Dietrich. 2020. Technical Lag of Dependencies in Major Package Managers. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Singapore, 228–237. doi:10.1109/APSEC51365.2020.00031
- [55] The Cargo Project. 2023. Cargo Public/Private Dependencies. <https://rust-lang.github.io/rfcs/3516-public-private-dependencies.html>
- [56] The Cargo Project. 2024. The Cargo Book. <https://doc.rust-lang.org/cargo/>
- [57] The Cargo Project. 2024. Cargo Resolver. <https://github.com/rust-lang/cargo/blob/15fbd2f/src/cargo/core/resolver/mod.rs>
- [58] The Debian Project. 2024. Debian Policy Manual: Chapter 5 - Control files and their fields. <https://www.debian.org/doc/debian-policy/ch-controlfields.html>
- [59] The Debian Project. 2024. Debian Policy Manual: Chapter 7 - Declaring relationships between packages. <https://www.debian.org/doc/debian-policy/ch-relationships.html>
- [60] The Go Project. 2024. Go Modules Reference. <https://go.dev/ref/mod>
- [61] The opam Project. 2013. The opam Manual. <https://opam.ocaml.org/doc/Manual.html>
- [62] The UNIX System Group. 1990. *UNIX System V Release 4 Programmer's Guide: System Services and Application Packaging Tools*. Prentice-Hall, Englewood Cliffs, NJ, USA.
- [63] Ralf Treinen and Stefano Zacchiroli. 2009. *Common Upgradeability Description Format (CUDF) 2.0*. Technical Report TR3. The Mancoosi project (FP7). <https://www.mancoosi.org/reports/tr3.pdf>
- [64] G. S. Tseitin. 1968. On the Complexity of Derivation in Propositional Calculus. In *Studies in Constructive Mathematics and Mathematical Logic, Part II*, A. O. Slisenko (Ed.). Steklov Mathematical Institute, Leningrad, 115–125. English translation published 1970; original in Russian: *Zapiski Nauchnykh Seminarov LOMI* 8 (1968), pp. 234–259.
- [65] Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. 2007. OPIUM: Optimal package install/uninstall manager. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE Computer Society, Minneapolis, MN, USA, 178–188. doi:10.1109/ICSE.2007.59
- [66] Natalie Weizenbaum. 2018. PubGrub: Next-Generation Version Solving. <https://nex3.medium.com/pubgrub-2fb6470504f>
- [67] Lyuye Zhang, Chengwei Liu, Sen Chen, Zhengzi Xu, Lingling Fan, Lida Zhao, Yiran Zhang, and Yang Liu. 2023. Mitigating Persistence of Open-Source Vulnerabilities in Maven Ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM/IEEE, Luxembourg, 191–203. doi:10.1109/ASE56229.2023.00058

Received 2026-02-20; accepted 2026-05-13