Modus: A Datalog Dialect for Building Container Images

Anonymous Author(s)*

Abstract

Containers help share and deploy software by packaging it with all its dependencies. Tools, like Docker or Kubernetes, spawn containers from images as specified by a build system's language, such as Dockerfile. A build system takes many parameters to build an image, including OS and application versions. These build parameters can interact: setting one can restrict another. Dockerfile lacks support for reifying and constraining these interactions, thus forcing developers to write a build script per workflow. As a result, developers have resorted to creating *ad hoc* solutions such as templates or domain-specific frameworks that harm performance and complicate maintenance because they are verbose and mix languages.

To address this problem, we introduce Modus, a Datalog dialect for building container images. Modus' key insight is that container definitions naturally map to proof trees of Horn clauses. In these trees, container configurations correspond to logical facts, build instructions correspond to logic rules, and the build tree is computed as the minimal proof of the Datalog query specifying the target image. Modus relies on Datalog's expressivity to specify complex workflows with concision and facilitate automatic parallelisation.

We evaluated Modus by porting build systems of 6 popular Docker Hub images to Modus. Modus reduced the code size by 20.1% compared to the used *ad hoc* solutions, while imposing a negligible performance overhead, preserving the original image size and image efficiency. We also provide a detailed analysis of porting OpenJDK image build system to Modus.

1 Introduction

Software sharing and deployment are hard because they impose the necessity of managing versions, dependencies, and execution environments. Containers, such as Docker containers [28], can package software with all its dependencies, simplifying sharing and deployment without considerable performance overhead. Containers are widely used in cloud computing [30], continuous integration/delivery [28], and reproducible research [12]. Containers are spawned from images, filesystem snapshots accompanied by configuration files. Images consist of layers that store changes to the underlying filesystem, such as file additions, modifications and deletions, that are combined in runtime using a union mount filesystem.

Container images are constructed by executing instructions written in a build system's language, the most popular of which is Dockerfile [4]. Dockerfiles describe a sequence of instructions, or a directed acyclic graph (DAG) of instructions in the case of multi-stage builds [3], that run shell commands, copy local files or files from other images into the constructed image, or set image properties defined in the OCI specification such as the working directory. The layered design of container images enables software reuse in the form of building images on top of other images. Public container registries, such as Docker Hub [2], store more than 500 000 public image repositories comprising over 2 million layers [40].

Since most software is configurable and evolving, container images are intrinsically parameterised. Image parameters include software versions, configuration parameters, and compilation flags. For example, the official image `python:3.9.6-alpine3.13` is parameterised with the version 3.9.6 of Python interpreter, as well the name and the version of the Linux distribution, Alpine 3.13. These parameters can, and often do, depend on and interact with each other and these interactions determine how images are built. For example, installing an older version of Python on a newer Linux distribution may require executing extra instructions to add appropriate software repositories. Dockerfiles do not express dependencies between parameters and the build logic. For example, Dockerfile specifies build parameters only as global variables that are used as arguments for build instructions, but does not permit using parameter values to control the sequence of executed instructions.

Dockerfile's lack of expressiveness forces developers to create *ad hoc* solutions specific to their domains. In this paper, we consider two such domains: packaging popular software into containers, and reproducing automated program repair experiments. OpenJDK [9], the most popular implementation of Java, is distributed in multiple versions that are updated independently, with different sets of features enabled, and is built for different platforms. Because of the limitations of Dockerfiles, the official OpenJDK Docker images [24] use a templating approach that generates Dockerfile via a combination of shell scripts and templates written in `awk` [11] and `jq` [7]. The reliance on *ad hoc* solutions, such as `awk`/`jq` templates causes maintenance problems. First, developers must learn multiple languages or frameworks to maintain the build definitions. Second, such build definitions are verbose.

To address the limitations of Dockerfiles, we propose Modus, a logic programming language for building container images. Its key insight is that an image build can be reduced to the problem of solving a set of Horn clauses, logical formulas in the form $\texttt{f}(x_1, ..., x_n)$ `:-` $\texttt{g}(y_1, ..., y_m), ..., \texttt{h}(z_1, ..., z_l)$. In Modus, images and layers are represented as logical facts, *e.g.* the image `python:3.9.6-alpine3.13` is represented as `python("3.9.6","alpine","3.13")`. Build instructions correspond to logic rules; the build DAG is computed as the minimal proof of the fact representing the target image from truthful facts representing existing images. Expressing build rules as Horn clauses leads to concise build definitions, allows users to define the same build workflows as defined in *ad hoc* Dockerfile templates, and enables automatic build parallelisation.

Horn clauses admit various semantics. Modus is a dialect of Datalog [17], a specific kind of Horn clauses, in which the computation of minimal proofs is decidable. When designing Modus, we judiciously chose a set of Datalog extensions that better model the domain of container builds. Broadly, these extensions fall into four categories. First, we extended Datalog with operators that implement container-related operations and domain-specific builtin predicates. Second, string manipulations, such as variable expansion, are crucial for defining container builds. However, a naïve addition of string manipulations to Datalog makes it undecidable.

To address this problem, we adapted stratified construction [13] to tractably support string operations. Third, Dockerfiles allow users to specify arbitrary parameters when launching a target's build. In Modus, the target is specified as a goal of a Datalog program, however the standard Datalog requires that all constants used in the goal are explicitly defined in the Datalog program. This requirement is impractical because of the large or even unknown number of possible configuration options in complex builds. To address this usability limitation, we implemented an approach that permits passing arbitrary build parameters through the goal of a Datalog program without compromising the program's safety.

To evaluate Modus, we ported build systems of 6 popular Docker Hub projects from Dockerfile templates into *Modusfiles*. Modus captured all Linux build scenarios, while reducing the code size by 20.1% with a negligible performance overhead. We also provide a detailed analysis of porting OpenJDK build system into Modus.

The contributions of this work are the following:

- We propose a novel application of Datalog, the definition of container image builds, where images and layers are represented as facts, build instruction as clauses, and build trees are minimal proofs of facts representing target images;
- We design and implement Modus, a dialect of Datalog for container build definitions with a set of judiciously chosen Datalog extensions;
- Our evaluation of Modus on popular Docker Hub images shows that it concisely expresses build scenarios for real-world software with negligible overhead.

All code, scripts, and data necessary to reproduce this work are available at https://zenodo.org/record/6366487.

2 Overview

This section explains Dockerfile's lack of expressiveness, and the disadvantages of this problem's popular workaround, the templating approach. It also shows how Modus expresses container image builds in Datalog, and how it reduces code size and build time when packaging real world projects.

2.1 Inexpressivity of Dockerfiles

Assume we would like to package an application compiled with GCC and GNU Make. A typical Dockerfile follows:

```
FROM gcc:bullseye AS bullseye_dev_release
COPY . /app/
RUN cd /app/ && make
```

It defines a build stage called `bullseye_dev_release` that represents a container image. This image is built on top of a public image `gcc:bullseye` with GCC installed on Debian Bullseye, by copying `app`'s source into the container via `COPY`, then compiling it with `RUN`. These instructions add filesystem layers on top of `gcc:bullseye`.

During development, it is typical to build a debug *target* of an application since it provides additional information about failures. Assume that the command `make debug` compiles a debug target of our application. There are two approaches to add a debug target into Dockerfile that we refer to as the duplication approach and the scripting approach. The duplication approach copies the code of `bullseye_dev_release` to create a new stage `bullseye_dev_debug` and replaces `make` with `make debug`. The scripting approach invokes an embedded shell script that processes a Dockerfile argument controlling the target:

```
FROM gcc:bullseye AS bullseye_dev_combined
ARG TARGET
COPY . /app/
RUN if [ "$TARGET" = "debug" ] ; then \
      cd /app/ && make debug ; \
    else \
      cd /app/ && make ; fi
```

In this example, the shell script dynamically chooses the command to execute based on the value of the variable `TARGET`.

Although the second approach might appear to be superior since it avoids duplication, it has several disadvantages. First, it cannot be applied to any instructions other than `RUN`, since other instructions, such as `COPY`, cannot be expressed as shell scripts. Second, using embedded scripts makes re-building images slower because of ineffective caching: the build system has to invalidate all cached layers following an `ARG` command if the value of `ARG` changes, even if the commands executed and the data copied do not change. Finally, embedded scripts interfere with layer management: they may enforce a particular placement of `RUN`s inducing a granularity of layers that does not coincide with one intended by the developer.

Software is often packaged for different *base* Linux distributions. Adding another base, say, Alpine, may require executing additional commands, *e.g.* installing GCC and GNU Make with `apk add gcc make`. Considering the disadvantages of the scripting approach, we resort to the duplication approach and add two build stages:

```
FROM alpine AS alpine_dev_release
RUN apk add gcc make
COPY . /app/
RUN cd /app/ && make
```

and `alpine_dev_debug` defined in a similar fashion.

The images `*_dev_*` are not designed for production since they contain redundant files such as the installed compiler and build system. Redundant files increase both image size and attack surface [1, 36]. A solution to this problem is multi-stage builds [3], *i.e.* copying the compiled program into a slim image from an auxiliary container using the `COPY --from` command, to build a production *mode* as shown below for Debian slim:

```
FROM debian:bullseye-slim AS bullseye_prod_release
COPY --from=bullseye_dev_release /app/ /app/
```

When building `bullseye_prod_release`, Docker constructs and executes a build directed acyclic graph (build DAG) in Figure 2a.

Since Alpine and Debian use different implementations of the C standard library (libc), binaries cannot be copied across them, which introduces a dependency between the base of the production image, *e.g.* `debian:bullseye-slim`, and the stage used for compilation, *e.g.* `bullseye_dev_release`. Dockerfile cannot express this dependency, which forces us to duplicate `bullseye_prod_release` to create a stage `alpine_prod_release` that copies binaries from `alpine_dev_release` into the base image `alpine`.

In summary, Dockerfile's lack of expressiveness forced us to create four duplicates of the development stages and two duplicates of the production stages, which is a code smell [35].

```
app(base, "dev", target) :-
  dev_image(base),
  copy(".", "/app/"),
  make(target).

dev_image("alpine") :- from("alpine").
dev_image("bullseye") :- from("gcc:bullseye").

app(base, "prod", "release") :-
  prod_image(base),
  app(base, "dev", "release")::copy("/app", "/app").

prod_image("alpine") :- from("alpine").
prod_image("bullseye") :- from("debian:bullseye-slim").

make("debug") :- run("cd /app/ && make debug").
make("release") :- run("cd /app/ && make").
```

Figure 1: Modusfile defining the image app parameterised with base image, build mode, and compilation target.

2.2 Build Instructions as Datalog Rules

To address the limited expressiveness of Dockerfiles, we introduce a new approach to define container image builds. The key intuition of this approach is that build instructions can be represented as a set of particular Horn clauses, formulas in the form

$$\texttt{img}(x_1,...,x_m) \texttt{ :- } \texttt{img}_{base}(y_1,...,y_k),\ \texttt{l}_1(t_1,...,t_h),...,\ \texttt{l}_n(z_1,...,z_g)$$

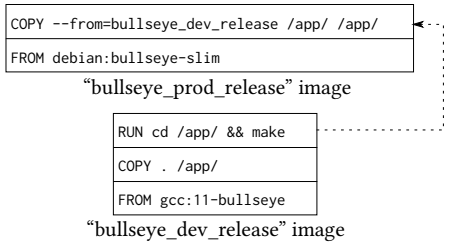
that can be interpreted as "the image `img` is constructed from the base image $\texttt{img}_{base}$ by adding the layers $\texttt{l}_1$,..., $\texttt{l}_n$".

Figure 1 shows a program written in our implementation of Horn clauses, a Datalog dialect Modus, that concisely expresses all build scenarios from Section 2.1. In this example, the *image predicate* `app(base, mode, target)` represents the target image. This predicate is defined using two rules, for the development mode `"dev"` and for the production mode `"prod"`. The rule for the development mode selects the base image using the image predicate `dev_image`, copies the files into the container using the *builtin predicate* `copy`, and builds the program using the *layer predicate* `make`. The `make` predicate is a layer predicate, because it is defined using the builtin layer predicate `run`. The rule for the production mode copies binaries from the development image using the *operator* `::copy`. In Modus, operators are not part of the logical inference. Instead, they implement container-specific functionality such as copying files between containers or setting image properties.

Compared to the Dockerfile (Section 2.1), the Modusfile (Figure 1) involves no code duplication — each instruction, such as copying files to the container or running compilation, appears exactly once.

To build an image from a Modusfile, the user has to specify a *goal*. For example, the goal `app("bullseye","prod","release")` would build a production image of the app on Debian Bullseye. The build DAG for this image is computed as a minimal proof of the fact representing the goal from facts representing existing images. Figure 2a shows such tree computed for `app("bullseye","prod","release")`. Notice that, modulo auxiliary predicates `make` and `prod_image`, this proof tree contains exactly the instructions from the build DAG of the image `bullseye_prod_release` in Figure 2a.

Modus can build multiple images at once, in parallel. For example, if the user specifies the goal `app(base,"prod","release")`, where



(a) Build DAG for the target `bullseye_prod_release`.

```
app("bullseye", "prod", "release")
├─ prod_image("bullseye")
│  └─ from("debian:bullseye-slim")
├─ (
│  └─ app("bullseye", "prod", "release")
│     ├─ dev_image("bullseye")
│     │  └─ from("gcc:bullseye")
│     ├─ copy(".", "/app/")
│     └─ make("release")
│        └─ run("cd /app/ && make")
└─ )::copy("/app/", "/app/")
```

(b) Proof tree for the goal `app("bullseye", "prod", "release")`

Figure 2: Build DAG and corresponding proof tree.

`base` is a variable, Modus automatically deduces that it needs to build the images `app("bullseye","prod","release")` and `app("alpine","prod","release")`, and builds them in parallel.

2.3 Code Size and Build Efficiency

The lack of Dockerfile's expressiveness described in Section 2.1 has motivated developers to adapt an alternative solution, to generate Dockerfiles from templates, which we refer to as the *templating approach*. As of March 2022, 17 out of 25 most popular projects in Docker Hub [2] use the templating approach.

We illustrate the templating approach using the build system for the widely-used official OpenJDK images. OpenJDK uses a combination of several string processing tools, `jq` for the JSON processing and `gawk` for advanced string processing [11], to define Dockerfile templates, from which it generates 40 Dockerfile instances for different OpenJDK configurations. Each configuration is identified by the tuple of the Java version, the Java type (JDK or JRE), and the base Linux distribution, *e.g.* `("8", "jdk", "oraclelinux7")`.

Each OpenJDK configuration requires executing a different set of instructions to build an image. The instructions are selected based on configuration parameters in a Dockerfile template, a fragment of which is shown in Figure 3. This template mixes three languages: `{{` and `-}}` are handled by a `gawk` script, expressions such as `if oracle_version == "7" then` are from `jq`'s query language, and `RUN` and `FROM` instructions are from Dockerfile's syntax.

The templating approach as used by OpenJDK has several disadvantages. First, it complicates maintenance, since it requires supporting an *ad hoc* templating frameworks that mix several languages. Second, string processing tools are inefficient, and Dockerfile generation can take a noticable portion of the build time. Finally, string processing scripts are error-prone and hard to debug.

```
{{
        if is_alpine then (
-}}
FROM alpine:{{ alpine_version }}

RUN apk add --no-cache java-cacerts

ENV JAVA_HOME /opt/openjdk-{{ env.version }}
{{
        ) elif is_oracle then (
-}}
FROM oraclelinux:{{ oracle_version }}-slim

RUN set -eux; \
{{ if oracle_version == "7" then ( -}}
        yum install -y \
{{ ) else ( -}}
        microdnf install \
  {{ ) end -}}
```

Figure 3: A fragment of Dockerfile template for OpenJDK.

Modus concisely describes OpenJDK's build scenarios without using external tools. OpenJDK images are represented via image predicate `openjdk(MAJOR_VERSION, JAVA_TYPE, VARIANT)`. Figure 4a shows a fragment of Modusfile corresponding to the template in Figure 3. It shows that Modus enables *abstraction*, since it allows extracting reusable parts of the build logic, such as selecting the base image via the predicate `base_image`. When launching a build, Modus generates proof trees for the specified goals. Figure 4b shows a fragment of the proof tree for the goal `openjdk("8", "jdk", "oraclelinux7")`. This proof tree corresponds to the concrete Dockerfile generated by the template, and builds an identical image.

Using Modus enabled us to reduce the code size of OpenJDK container build system by 47.6% and reduce build time by 40.6% compared to the templating approach. Our case study on OpenJDK images is described in more details in Section 6.

3 Background

Modus is a Datalog dialect specialised for container builds via a set of extensions. It uses BuildKit [8] to build images from proof trees.

3.1 Datalog

A Datalog program consists of a finite set of rules and facts. Rules are Horn clauses in the form L_0`:-`$\mathsf{L}_1, ..., \mathsf{L}_n$, where each L_i is a literal `p(`$\mathtt{t}_1, ..., \mathtt{t}_n$`)` such that `p` is a predicate symbol and $\mathtt{t}_i$ are terms. A term is either a constant or a variable. We denote variables and predicate symbols using alphanumeric strings with optional underscores, *e.g.* `foo_bar`, and constants using quoted strings, *e.g.* `"foobar"`. The left side of a clause is its head; the right-hand side is its body.

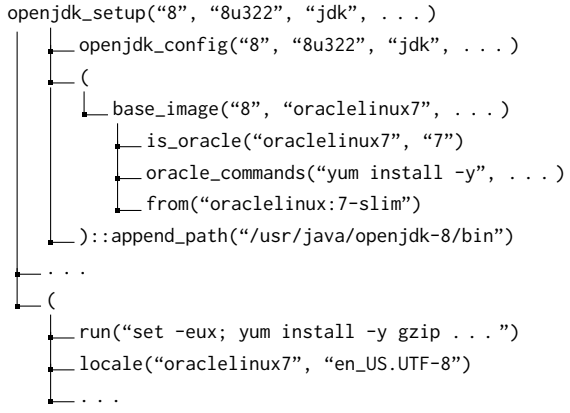
Datalog imposes structural restrictions on its rules. First, each predicate symbol has to be applied to the same number of arguments, which defines its arity. Second, each Datalog program has to satisfy the following safety conditions that ensure the set of all facts that can be derived from a Datalog program is finite:

Definition 3.1 (Safety Conditions). The following structural constraints should hold: (1) each fact is ground (has no variables) and (2) a variable that occurs in the rule's head must occur in its body.

```
base_image(MAJOR_VERSION, VARIANT, JAVA_HOME, ...) :-
    (
        is_alpine(VARIANT, ALPINE_VERSION),
        from(f"alpine:${ALPINE_VERSION}"),
        ...
    ;
        is_oracle(VARIANT, ORACLE_VERSION),
        oracle_commands(INSTALLER, CLEANER,
             ORACLE_VERSION),
        from(f"oraclelinux:${ORACLE_VERSION}-slim"),
        ...

openjdk_setup(MAJOR_VERSION, VERSION, ...) :-
    openjdk_config(MAJOR_VERSION, VERSION, ...),
    base_image(MAJOR_VERSION, VARIANT, ...),
    (
        run(BASE_SETUP_COMMAND),
        locale(VARIANT, LANG),
```

(a) A fragment of Modusfile for building OpenJDK.



(b) A proof tree fragment for `openjdk("8", "jdk", "oraclelinux7")`

Figure 4: A build DAG and the corresponding proof tree.

Variables that occur in the head of a rule and also in the body of the same rule are called grounded. Variables that occur only in the head of a rule but not in the body are called non-grounded. A ground substitution $\theta \stackrel{\text{def}}{=} \{x_1/c_1, ..., x_n/c_n\}$ is a mapping from variables to constants. We denote an application of the substitution θ to the term $\mathtt{t}$ as $\mathtt{t}\theta$, and to the literal L as $\mathsf{L}\theta$.

In the proof-theoretic interpretation of Datalog, the meaning of a program is defined as the set of all facts that can be inferred from the program. For a given rule L_0 `:-` $\mathsf{L}_1, ..., \mathsf{L}_n$ and a set of ground facts $\mathsf{F}_1, ..., \mathsf{F}_n$, we say that F_0 can be inferred in one step from $\mathsf{F}_1, ..., \mathsf{F}_n$ if there is a substitution θ such that for any $i \in 0..n$, $\mathsf{F}_i = \mathsf{L}_i\theta$. We refer to this as elementary production principle (EPP). A ground fact F can be inferred from a program P if either $F \in P$ or F can be inferred by applying EPP a finite number of times. The sequence of applications of EPP used to infer a fact F from P forms a proof of F from P. A proof can be represented using a proof tree.

Definition 3.2 (Proof Tree). For a given Datalog program P and a ground fact F, a proof tree of F from P is the tree $(F, R, \theta, \{t_1, ..., t_n\})$ such that $n = 0$ iff $F \in P$, otherwise $R \stackrel{\text{def}}{=} \mathsf{L}_0$ `:-` $\mathsf{L}_1, ..., \mathsf{L}_n$ is a rule from P, $F = \mathsf{L}_0\theta$, and t_i are proof trees of $\mathsf{L}_i\theta$.

Consider a Datalog program defining graph reachability:

```
reach(n, n) :- node(n).
reach(n1, n2) :- arc(n1, n3), reach(n3, n2).
```

and truthful facts: `node("1")`, `node("2")`, `node("3")`, `arc("1","2")` and `arc("2","3")`. Then, a proof tree of the fact `reach("1","3")` can be visualised as follows:

```
reach("1", "3")
├─ arc("1", "2")
└─ reach("2", "3")
   ├─ arc("2", "3")
   └─ reach("3", "3")
      └─ node("3")
```

Datalog allows specifying a goal to select a subset of its outputs that are subsumed by the goal. For example, the fact `f("a","a")` is subsumed by the goal `f(x,x)`, but `f("a","b")` is not.

3.2 Container Images and Dockerfiles

Containers, a form of OS-level virtualisation, are spawned from images, filesystem snapshots accompanied by configuration files. Let $\mathcal{F}$ be a set of all filesystems, trees of files with their contents. Filesystems are defined recursively, that is a subtree of a filesystem F under the path p denoted as $subtree(F, p)$ is also a filesystem. $\mathbb{F}_{host}$ is the filesystem of the host OS. Container images, as standardised by The Open Container Initiative (OCI) [21] have layered design. A layer is typically stored as a delta for two filesystems, but to abstract over filesystem implementations we define a layer L non-constructively as a pair of filesystems (F_1, F_2). A container image $I \stackrel{\text{def}}{=} [L_1, ..., L_n]$ is a stack of layers such that for each i, if $L_i = (F_1, F_2)$ and $L_{i+1} = (F_3, F_4)$, then $F_2 = F_3$. We denote the set of all images as $\mathcal{I}$. We assume that images can be referenced by their identifiers using the function $image\_by\_ref : ID \to I$. The runtime semantics of images is a function $[[\ldots]] : \mathcal{I} \to \mathcal{F}$ that is usually computed dynamically using a union mount filesystem, such as OverlayFS [15]. To abstract away irrelevant implementation details, we define this semantics as simply the top filesystem on the stack, that is $[[I]] \stackrel{\text{def}}{=} snd(peek(I))$.

Two common operations performed on images are $\overline{run}$ and $\overline{copy}$. The operation $\overline{run} : Scripts \to \mathcal{I} \to \mathcal{I}$ takes a script s and an image I, executes s on the filesystem $[[I]]$ resulting in a new filesystem F, and returns an image that is obtained by appending the layer $([[I]], F)$ to I. The operation $\overline{copy} : \mathcal{F} \times Paths \to \mathcal{I} \to \mathcal{I}$ takes a filesystem F, a path p and an image I, and returns an image obtained by adding the layer containing F under the path p on top of I.

Dockerfile is a widely-used format for specifying container image builds. Dockerfiles define *build stages*, each of which consists of a single `FROM` *id* instruction specifying the base image $I_{base} \stackrel{\text{def}}{=} image\_by\_ref(id)$, followed by a sequence of n instructions. Each i-th instruction corresponds to a function $f_i : \mathcal{I} \to \mathcal{I}$. If this instruction is `RUN` *script*, then $f_i \stackrel{\text{def}}{=} \overline{run}(script)$. If this instruction is `COPY` *src dst*, then $f_i \stackrel{\text{def}}{=} \overline{copy}(subtree(\mathbb{F}_{host}, src), dst)$. The semantics of the build stage is the image $(f_n \circ ... \circ f_1)(I_{base})$. Dockerfiles also support instructions that set image properties such as `WORKDIR` that sets the current working directory, but such instructions do not create new layers.

$build\_image(F, \_, \_, \{t_1, ..., t_n\}) =$
  if $n = 0$ then
    match F with
    | `from(id)` $\to$ $image\_by\_ref($`id`$)$
  else
    let $I_{base} = build\_image(t_1)$ in
    let $f_i = build\_layer(t_i)$ for $i \in [2, n]$ in
    $(f_n \circ ... \circ f_2)(I_{base})$

$build\_layer(F, \_, \_, \{t_1, ..., t_n\}) =$
  if $n = 0$ then
    match F with
    | `run(s)` $\to$ $\overline{run}(s)$
    | `copy(src, dest)` $\to$ $\overline{copy}(subtree(\mathbb{F}_{host},$ `src`$),$ `dest`$)$
  else
    let $f_i = build\_layer(t_i)$ for $i \in [1, n]$ in
    $f_n \circ ... \circ f_1$

Figure 5: Building images from proof trees.

4 The Modus Language

This section introduces the syntax and semantics of Modus, describes and motivates Datalog's extensions of that it implements.

4.1 Syntax & Semantics

Modus' syntax is based on Datalog's syntax given in Section 3.1. Modus extends this syntax via the notion of *predicate kind*. Modus has three kinds of predicates: image predicates, layer predicates, and logic predicates. Image predicates correspond to container images. An image predicate is either the built-in predicate `from` that refers to an existing image by name or it is defined in a rule

$$\texttt{i}_1\texttt{(x}_1\texttt{,...,x}_m\texttt{)} \texttt{ :- } \texttt{i}_2\texttt{(y}_1\texttt{,...,y}_k\texttt{), } \texttt{l}_1\texttt{(t}_1\texttt{,...,t}_h\texttt{),..., } \texttt{l}_n\texttt{(z}_1\texttt{,...,z}_g\texttt{)}$$

where $\texttt{i}_1$ and $\texttt{i}_2$ are image predicates, and $\texttt{l}_i$ are zero or more layer predicates. Layer predicates describe image layers. A layer predicate is either `run` and `copy` or a predicate defined in a rule

$$\texttt{l}_1\texttt{(x}_1\texttt{,...,x}_m\texttt{) :- l}_2\texttt{(t}_1\texttt{,...,t}_h\texttt{),..., l}_n\texttt{(z}_1\texttt{,...,z}_g\texttt{)}$$

where $\texttt{l}_i$ are layer predicates. Logic predicates are predicates that do not represent any container-related entities; they define build logic. Logic predicate can be defined through other logic predicates, and can appear in any part of rules' bodies. For example, the program in Figure 1 defines image predicates `app`, `dev_image`, `prod_image` and a layer predicate `make`.

The semantics of a Modus program is a mapping from goals to sets of images. For a given program P and a goal G, Modus computes all facts that can be inferred from P and are subsumed by G, then constructs a proof tree for each fact. A fact corresponds to the built image, and its proof tree is the recipe to build the image. Since there can be multiple proof trees for a given fact, we choose the minimal tree w.r.t. the number of layers, as it naturally optimises the resulting image size and build time. Given a proof tree $(F, R, \theta, \{t_1, ..., t_n\})$, Modus builds an image that can be defined using the function $build\_image : ProofTrees \to \mathcal{I}$ given in Figure 5. Note that this algorithm assumes that the proof tree is stripped of logic predicates. An efficient implementation of *build_image* is discussed in Section 4.3.

4.2 Extensions

Implementations of Datalog often contain domain-specific extensions, *e.g.* to facilitate implementation of program analysers [26]. Modus is the first Datalog implementation designed for container image builds, and we judiciously chose a set of extension to help modeling this new domain.

4.2.1 Built-in Predicates and Operators To facilitate definition of image builds, we introduced a library of built-in predicates. Apart from the predicates `from`, `run`, and `copy` described above, we introduce predicates for defining conditions on the input parameters. Notably, the `semver_*` predicates define software version comparison as per the SemVer specification [10]. For example, the following fact is true: `semver_lt("1.0.3","1.1.0")`, where `lt` means $<$.

The key difficulty of adding built-in predicates to Datalog is that built-in predicates such as `semver_lt` are infinite relations, which require special handling to retain Datalog's decidability. We used the standard approach [16] to support built-in predicates: we defer the evaluation of a built-in predicate until all arguments of this predicate are bound to constants.

Some container-specific operations are inconvenient to express using predicates. For example, in multi-stage builds [3], files are copied from a temporary image to the current image. Since images are identified with literals, an operation expressing multi-stage build would need to take a literal as an argument. However, Datalog does not permit applying predicates to literals. To address this, we introduce operators that use the syntax `literal::operator(`t_1`,...,`t_n`)`. A notable example of an operator is `::copy` that is needed to implement multi-stage builds. Formally, the semantics of `::copy` are defined by adding the following rule in the `match` statement of the function *build_layer* in Figure 5:

$$\texttt{| literal::copy(src, dest)} \rightarrow \overline{copy}(subtree([[build\_image(\texttt{literal}, \_, \_, \emptyset)]], \texttt{src}), \texttt{dest})$$

An example usage of `::copy` is given in Section 2.2.

4.2.2 String Manipulation & Stratified Construction String manipulations are often used in container builds systems to parse/construct configuration options. String manipulations are enabled in Modus via the built-in predicate `string_concat(A, B, C)` that states that C is the concatenation of A and B. Modus also introduces Python-like formatted strings that are defined through `string_concat`. For example, `foo(f"a${x}")` is equivalent to `string_concat("a", x, y), foo(y)`, where `y` is a fresh variable.

A naïve incorporation of `string_concat(A, B, C)` into Datalog makes it possible to generate strings of arbitrary length, making evaluating Datalog programs intractable. To address this problem, we adapted stratified construction [13] to forbid Modus programs that involve recursive predicates that depend on arguments of `string_concat`, in the spirit of stratified negation [16]. As a result, each Modus program can apply `string_concat` only up to a predefined number of times, which is independent of the input. This limitation did not impose any obstacles when porting realistic build systems presented in Section 5.

4.2.3 Non-grounded Variables Datalog's safety conditions (Definition 3.1) require that each variable that occurs in the rule's head must occur in its body. When using built-in predicates, this variable should also be an argument of a non-built-in predicate in the body. This condition can be naturally satisfied in Datalog's traditional applications such as program analysis, where all constants are present in the database. In contrast, this restriction in inconvenient for container builds as the following example demonstrates:

```
app(cflags) :-
  from("gcc:latest"),
  copy(".", "."),
  run(f"gcc ${cflags} test.c -o test").
```

The variable `cflags` is not grounded, as it only appears as an argument of a built-in predicate, so this is an invalid Datalog program. Dockerfiles allow users to specify arbitrary parameters when launching a target's build, such as the `-g` flag. However, GCC accepts a large number of flags, so it would impractical to list all accepted flags in the Modus program. Instead, it would be natural to allow users to use this program to build the goal `app("-g")`, since the argument of `run` can then be inferred from the goal.

To enable such usage scenarios, we relaxed the safety conditions by allowing user-defined predicates with non-grounded variables. At the same time, we introduced the new restriction that, during evaluation, we defer the evaluation of these predicates until all of their arguments are bound to constants, following the handling of built-in predicates in Section 4.2.1. Doing so enabled us to support the usage scenario above without sacrificing Datalog's safety.

4.3 Implementation

To generate proof trees for a given goal, we implemented a custom, top-down Datalog solver based on SLD resolution [16] that 1) supports our library of built-in predicates described in Section 4.2.1, 2) handles non-grounded variables in a non-standard way as explained in Section 4.2.3, and 3) generates proofs that minimise the number of layers required to build a given image.

To build images from proof trees, Modus uses BuildKit [8] as the backend. BuildKit provides an intermediate representation LLB, which is described by BuildKit's developers as "LLB is to Dockerfile what LLVM IR is to C". LLB is a DAG, where a node either refers to existing image, or corresponds to $\overline{run}$ or $\overline{copy}$ operation. To implement the function *build_image*, Modus translates the proof tree into LLB and executes it with BuildKit. This approach has several advantages. First, BuildKit automatically parallelises the build. Second, it provides automatic caching, *i.e.* it tracks changes to the filesystem and shell commands, and only re-builds layers and images that are affected by these changes.

Currently, BuildKit does not support build graphs with multiple output. To overcome this problem, when building multiple images, we construct a dummy image and attach the target images as children to this dummy image. Although this workaround enables us to build multiple images in parallel, BuildKit's API does not currently give us direct access these child images after they are built. Instead, we must redundantly call the build system. This cost is especially noticeable, since these calls can only start after all builds are finished, preventing Modus from maximally exploiting parallelism. We refer to this redundant step as *exporting*. This step is not an essential part of Modus' build process, but a workaround over the current limitations of BuildKit's API.

Project	Templating Method	Outputs
Ubuntu	bash	6
Redis	sed + awk	9
Nginx	sed	8
NodeJS	sed + awk	32
MySQL	awk + jq	4
Traefik	envsubst	4

Table 1: Corpus Descriptive Statistics.

5 Evaluation on Docker Hub Images

We now show that, for a corpus of six popular Docker projects, Modus reduces the overall size of code used to build images by an average of 20.1%, with a negligible performance overhead.

To build our corpus, we considered images as ranked by Docker Hub "Suggested" filter, which closely follows downloads, and selected the first six which used different sequences of commands for templating. Under this rule, selecting MySQL filtered out Python and PostgreSQL. We also intended to rule out projects that either do not use parameters in their tags or build only one container image, as the focus of Modus is on parameterised builds, but we did not need to employ this filter. We have no reason to believe that this selection process introduces bias with respect to the object of our study — namely, builds conditioned on parameters. Using popular examples is common practice in empirical work. Any project considering Modus would do so to adopt Modus' feature set precisely because Modus promises to help them speed or ease the maintenance of their build.

Porting each project to Modus requires understanding the existing build and, in turn, substantial manual effort. The six projects we selected collectively cover an interesting subdomain of the container build system space. Table 1 summarises our corpus.

5.1 Modus' Code Reduction

We now quantify and compare the size of the build systems of our corpus to their Modus ports. Our key finding is that Modus reduces build code size by 20.1%, on average, over our corpus in lines of code, by reducing repetition and avoiding scripting. We measure code size after normalising the code by stripping comments, whitespace, and "stop words", *i.e.* the Cooked_2 normalisation described in Section 6.2.

Table 2 shows our code reduction results, calculated in two ways. Table 2a only counts the Dockerfile templates, not scripts that use them to generate Dockerfiles, in the Templating columns and only Modusfiles in the Modus columns. To the counts in Table 2a, Table 2b adds the sizes of all templates and scripts used to build the images in the Templating columns, and any scripts needed to generate version lists (but not templating) in the Modus columns. We present both tables because the definition of build code is not well established. One view is that a build system consists only of the Dockerfile and Modufiles. Another view is that a build system includes all the version fetching and templating scripts. Further complicating matters is the fact that some of these scripts may additionally perform tasks unrelated to building their project. Thus, we present these two tables to establish lower and upper bounds on the true size of each project's build system.

In Table 2a, the Ubuntu project does not use templates, but a bash script that prints its Dockerfiles, so we did not report its template sizes. The Nginx project's templating script contains most of its build logic, written in bash, and it simply writes results into its templates. The script manipulates strings and constructs sets of packages to install, inserted into the Dockerfile in the form of a space-separated string that get passed to `apt`. Porting this logic into a Modusfile substantially reduces the build code needed from 169 lines of bash script to 58 lines of Modusfile (ignoring docker build instructions in either case). This accounts for the swing in the results for Nginx across Table 2a and Table 2b.

> Modus reduces build system code size by 20.1% on average in lines and 21.5% reduction in words.

Modus owes this result to its expressivity. In Section 6, we use a case study to illustrate how it achieves these savings in detail.

5.2 Modus Build Time

We now compare our corpus' build systems to their Modus ports. These experiments were performed on AWS' **c5.2xlarge**, which, at the time of writing, had 8 CPUs, 16 GiB RAM, 10 Gbps bandwidth, and an SSD disk with 8000 IOPS. We used Modus *v0.1.11* to build our Modusfile, and used the BuildKit mode of `docker build` and *GNU parallel* [38] to build our corpus' Dockerfiles. Some projects build images for platforms other than x86_64, but we only ran our tests on a x86_64 VM, not for other architectures. Despite this, our Modusfiles include code to build other architectures, when a project in our corpus does, to ensure a fair code size comparison. We exclude the redundant export step discussed in Section 4.3 from our results, since it is a workaround for BuildKit API limitations. Exporting averages 4.78 s across our corpus. When BuildKit exposes an appropriate API, this overhead will disappear.

Table 3 shows our results. Each of the μ column is an average of 64 runs. The CI columns is the 95% confidence interval for the true average. In both table, we did not include time taken to pull base images, *e.g.* `alpine` or `debian`, because (1) Docker Hub request limits meant that we could not do a fresh pull for each run; and 2) the time taken to pull images depends on network conditions and CPU speed (for extraction), two problems Modus does not address.

> Modus increases average total build times by 0.6%, showing that Modus reduces build code size with negligible overhead.

Modus' implementation is not yet optimised, but still it achieves this negligible overhead. Even including the low and input-independent overhead of the BuildKit workaround (Section 4.3), developers can safely adopt Modus today to take advantage of its concision.

5.3 Validating Modus-Built Images

Since our Modusfiles are direct translation of our corpus' existing build systems, we expect Modus images to closely match the Dockerfile images in size and space efficiency, as measured by `dive` [22]. Our results shows that, on average, our image efficiency score is 0.15% higher than the Dockerfile counterpart. In the worse case, our efficiency score is not more than 0.05% lower. The average difference in image size is 1.01 MiB (0.4%) and the maximum difference is 3.96 MiB (3.6%). We also ran simple smoke tests that runs the entry

	Templating			Modus		
Project	**CR**	**Words**	**Bytes**	**CR**	**Words**	**Bytes**
mysql	162	600	6398	176 (+08.6%)	586 (-02.3%)	7041 (+10.1%)
nginx	170	668	6155	209 (+22.9%)	786 (+17.7%)	8168 (+32.7%)
node	202	843	8080	165 (-18.3%)	584 (-30.7%)	6480 (-19.8%)
redis	161	645	6232	161 (+00.0%)	590 (-08.5%)	6187 (-00.7%)
traefik	72	194	2993	52 (-27.8%)	142 (-26.8%)	2335 (-22.0%)
ubuntu		n/a		61 (n/a)	191 (n/a)	2553 (n/a)

(a) Modusfile and Dockerfile templates only.

	Templating			Modus		
Project	**CR**	**Words**	**Bytes**	**CR**	**Words**	**Bytes**
mysql	318	1081	11022	316 (-00.6%)	1012 (-06.4%)	11357 (+03.0%)
nginx	292	907	9069	209 (-28.4%)	786 (-13.3%)	8168 (-09.9%)
node	664	2134	19122	569 (-14.3%)	1683 (-21.1%)	15414 (-19.4%)
redis	244	952	8577	207 (-15.2%)	773 (-18.8%)	7657 (-10.7%)
traefik	105	302	3938	52 (-50.5%)	142 (-53.0%)	2335 (-40.7%)
ubuntu	78	254	2533	69 (-11.5%)	212 (-16.5%)	2688 (+06.1%)

(b) Including version fetching (for both) and templating scripts.

Table 2: Normalised code size for our corpus (Table 1).

Project	**Dockerfiles (s)**		**Modus (s)**	
	μ	CI	μ	CI
mysql	66.57	66.02–67.12	68.34	68.10–68.57
nginx	23.08	23.04–23.12	26.45	26.40–26.51
node	108.95	108.51–109.39	83.30	82.26–84.33
redis	199.85	199.77–199.94	202.19	199.77–204.61
traefik	9.02	8.73–9.32	8.91	8.43–9.38
ubuntu	7.86	7.81–7.91	12.25	12.16–12.34

(a) `modus build` and parallel `docker build` only.

Project	**Dockerfiles (s)**		**Modus (s)**	
	μ	CI	μ	CI
mysql	69.52	68.96–70.07	69.67	69.42–69.92
nginx	23.15	23.11–23.18	26.45	26.40–26.51
node	109.99	109.55–110.44	88.20	87.14–89.25
redis	200.87	200.71–201.04	203.35	200.92–205.78
traefik	9.06	8.76–9.35	8.91	8.43–9.38
ubuntu	11.23	11.12–11.34	12.28	12.19–12.37

(b) Including version fetching (for both) and templating scripts.

Table 3: Build time for each project in our corpus (Table 1).

executable with flags to print its version on each image produced: all Modus images passed this test.

6 Case Study: OpenJDK

We conducted a case study to detail how we port build systems to Modus and to explain how Modus' expressivity allows it to define build systems with concision. We chose to study the official image packaging for OpenJDK because 1) Docker maintains it and 2) it outputs 40 images [24], at the time of writing, which is above 32.1, the average number of images produced by the top five (by GitHub stars) build systems maintained by Docker. We first describe OpenJDK's current build system and how we manually ported it, while taking care to produce an equivalent Modus build. The Modus port is concise, using 47.6% fewer words than OpenJDK's official build, and fast, achieving a total build-time speedup of 40.6%.

6.1 Porting OpenJDK's Build to Modus

The existing OpenJDK packaging uses a templating approach to handle parameter interactions. Figure 6 presents the workflow of Docker's OpenJDK build system (DOBS). The build system accepts two inputs: Dockerfile templates and OpenJDK source URLs. Step 1 updates the version data by checking known sources of OpenJDK and generates the file `versions.json`. In step 2, `awk`, an advanced string processing tool [11], processes the Dockerfile templates, generating an appropriate `jq` [7] expression according to the code within template delimiters `'{{'` and `'}}'`. In step 3, `jq` generates Dockerfile instances by applying the expression constructed at step 2 to `versions.json` (in step 3). A CI service performs step 4, taking the Dockerfiles from step 3 and building 40 OCI images.

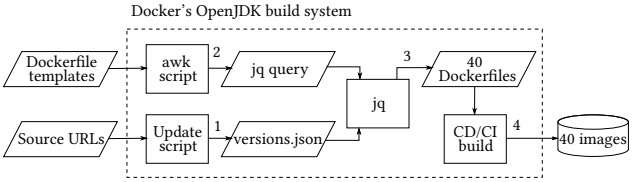


Figure 6: DOBS build steps to produce OpenJDK images.

To port a build system to Modus, one needs to identify its parameters, re-encode into Modus' data format and translate its conditional logic into rules. We ported the subset of DOBS that generates Linux images. Examining DOBS' source, we learned that the directory structure[1] containing DOBS' Dockerfiles encodes the parameters that DOBS uses to identify an image: major application version, Java type, and base image variant. These triples, in turn, index DOBS' JSON versions file [24] where we found the additional parameters needed to build the image[2]: full version, AMD64 binary URL, and ARM64 binary URL. Modus' primary data format is a set of Modus facts, which are essentially tuples. We manually inspected both the existing JSON version data [24] and templated Dockerfile [24] to learn how to convert the JSON data into Modus facts. For simplicity, each fact that we generate for our build system includes all of the parameters. We manually examined each conditional block in DOBS and wrote an equivalent Modus rule.

To ensure correctness of our port, after each implementation of a Modus rule, we built images that correspond to those new branches of logic and used dive [22] to compare the operations

[1]For example, consider https://github.com/docker-library/openjdk/tree/master/19/jdk/alpine3.14.

[2]We omit parameters that do not relate to the image build itself, such as checksums.

	Templating			Modus		
Variant	CR	Words	Bytes	CR	Words	Bytes
Raw	549	2209	16109	267 (-51.4%)	869 (-60.7%)	10389 (-35.5%)
Cooked_1	441	1556	10626	246 (-44.2%)	750 (-51.8%)	9607 (-9.6%)
Cooked_2	403	1326	9642	244 (-39.5%)	695 (-47.6%)	7645 (-20.7%)

Table 4: Modus builds OpenJDK images with 47.6% less words.

performed in each layer with the corresponding DOBS image. We contend that this development process established a close mapping between our *linux.Modusfile* and their *Dockerfile-linux.template*; the interested reader can build and examine the relevant artefacts using https://zenodo.org/record/6366487 to confirm.

A simple metric that can act as a heuristic for correctness is the *difference in image size*, so we set up a CI that computes this for every OpenJDK image we build. The average difference in size is **0.38MB**. For comparison, the largest OpenJDK image is currently **675MB**[3]. We spot-checked five uniformly chosen images by tarballing the filesystems and then running `pkgdiff` [31] on them to determine whether these differences are data or executables. All the differences we inspected were in data, like log files.

6.2 Modus Concisely Builds OpenJDK

Modus' expressivity makes it more concise than Docker's OpenJDK build system (DOBS). We quantify its concision, show that this concision and readability comes for free: the efficiency of the images that Modus builds match those produced by DOBS.

Table 4 displays the statistics of raw, unedited files, as well as two variants, Cooked_1 and Cooked_2[4]. Cooked_1 removes comments and empty lines to de-emphasise formatting conventions. Cooked_2 removes further tokens that are analogous to stop words, like delimiters. In particular, we remove templating delimiters such as `}}`, and we replace whitespace chains defined by `[ \t]\+` with a single space to balance out different indentation choices. As with whitespace, we remove these tokens in an effort to make the comparison more fair and centered on tokens with more semantic content.

> On OpenJDK, Modus achieves a reduction of 47.6% words and 20.7% bytes, using the Cooked_2 code normalisation.

These size reductions provide evidence that Modus can make build systems more maintainable. Much of Modus' concision is a result of its constructs and built-in operators to handle conditionally selecting instructions. In addition to concision, Modus constructs enhance readability. With templating, readability often breaks down when using nested conditional syntax like in the DOBS templated Dockerfile (Figure 3). In this case, it is often tricky to work out scopes. Such cases force developers to skim through an entire templated Dockerfile to understand its behaviour. The use of a rule, such as in Figure 4a, explicitly restricts the scope of variables and nested expressions provide a convenient yet readable alternative to the nested conditionals found in DOBS' templated Dockerfile. In this way, Modus lends itself to more readable and reusable code.

[3]This is openjdk:19-jdk-bullseye [24].

[4]The script that implements these variants can be found here, https://zenodo.org/record/6366487 at openjdk-images-case-study/benchmark-scripts/code_size.sh

	Approach	μ (s)	$\mu + \mu_t$ (s)
OpenJDK Dockerfiles	Sequential	516.3	637.4
	Parallel	119.8	240.9
	Manual Optim.	276.7	397.8
Modus	Total	143.1	143.1
	Exporting	18.0	N/A

Table 5: Average build time results for OpenJDK images over 10 runs. μ_t is the average time to perform template processing to construct parameter-specific Dockerfiles.

Another Modus feature that promotes concision is its `::merge` operator. Applied to a group of literals, *e.g.* (l_1, l_2)`::merge`, this operator squashes all `run` and `copy` commands into a single command during build DAG construction, which results in a single layer built. We used `dive` [22] (Section 6.1) to estimate the efficiency of an image. DOBS' images achieve image efficiency (above 95%), but at a cost to their readability and separability. To avoid recording redundant modifications in the layer diffs and bloating the image size, nearly a half of the DOBS Dockerfile is a single `RUN` layer [24]. Using `::merge` as in our `linux.Modusfile` [5], meant that we did not need to pack operations into a single command. We extracted the steps of the build into logical rules and merged them, resulting in the following efficiency scores:

- DOBS OpenJDK images: average efficiency of **98.8%**.
- Modus OpenJDK images: average efficiency of **98.9%**.

Modus' `::merge` operator facilitates the best of both worlds: the readability of separating code sections without the inefficiency of more layers recording more diffs.

6.3 Modus Quickly Builds OpenJDK

This section compares the build time of Docker's Official build system (DOBS) with our port using Modus. It shows that Modus scales to builds that output a large number of images. We used the same experiment setup as in Section 5.2, with docker-library/openjdk and modus-continens/openjdk-images-case-study revisions as found in https://zenodo.org/record/6366487. DOBS uses separate GitHub CI/CD workflows [24] to build the image, while in our experiment we (again) used GNU Parallel to build all Dockerfiles at once.

Table 5 presents the results. We included the build time with and without template processing since this can vary with different template engines. DOBS performs some duplicate copying of binaries. The manual optimisation approach is our attempt to optimise DOBS to fix this problem using Docker's builder pattern [3], where a shell script separately builds and copies binaries [6]. The exporting step in the final row of Table 5 is a subset of the time required to perform the image builds that could be reduced with future optimizations. We performed $n = 10$ runs of each approach and computed the sample mean. Template processing (steps 2 and 3 in Figure 6) was run separately with $n = 10$ runs leading to a sample mean of $\mu_t = 121.1$s. Since template processing is necessary to output images, Modus outperforms every DOBS approach when

[5]Found at lines 246–252 of openjdk-images-case-study/linux.Modusfile, in our reproduction package, https://zenodo.org/record/6366487.

[6]Found at openjdk-images-case-study/openjdk/build.sh in our reproduction package, https://zenodo.org/record/6366487.

accounting for template processing time. Specifically, Modus's total build time is lower than the expected total time with templating, $\mu + \mu_t$, as 143.1s is less than Sequential (637.4s), Parallel (240.9s), and our Manual Optimization (397.8s).

> Modus achieves a 40.6% speed-up over OpenJDK's official build system, when including Dockerfile generation.

This impressive speed-up runs counter to the results reported in Section 5, where we report a negligible slow-down. This speed-up is partly a result of Modus being able to quickly perform SLD resolution (Section 3), so it avoids slow template processing. DOBS' template processing is notably slow; this is due to their awk script (step 2 of Figure 6) relying on inefficient user-defined functions [25]. In fact, $\mu_t > \mu_p$ where μ_p is the mean time to solely build the images in parallel so DOBS' template processing setup is a significant factor in their total build time. Furthermore, the expected time without the exporting step performed by Modus is $143.1 - 18.0 = 125.1$ seconds, which indicates potential for Modus to outperform even step 4 (Figure 6) in isolation. The results for our manual optimization from Table 5 suggest that this copying incurs a further penalty that outweighs any benefit from avoiding the extra network calls fetching duplicate binaries. This motivates the use of a system like Modus over shell scripts or *ad hoc* optimizations.

7 Related Work

Works relevant to Modus include container build systems, build systems based on logic programming, and Datalog implementations and extensions.

Container Build Systems The most popular container build system, which is distributed with Docker, uses Dockerfiles [4] as the build definition language. In Dockerfile, build parameters must either be hard-coded or set as global variables; users have limited ability to describe how interactions of these parameters affect the build workflow. This leads to substantial code duplication. As a result, developers resort to developing *ad hoc* Dockerfile templating frameworks, as in Official OpenJDK images [24] maintained by Docker. These frameworks are cumbersome, as they typically mix several string processing languages, and still verbose. Earthly [5] extends Dockerfiles with additional constructs. Since its conditional statements are evaluated during the build process, the branches taken are not known in advance and therefore Earthly does not automatically parallelise builds with such conditional instructions. Buildah [20] embeds build instructions into a shell script and therefore it does not parallelise builds, since shell scripts, as Turing-complete programs, cannot be automatically parallelised. Nix [19] and Guix [6, 18] are functional package managers that can also be used to build containers. In order to package software with Nix or Guix, all dependencies must also be packaged by these managers. Compared to these systems, Modus substantially reduced the code size, enables automatic build parallelisation due to its static build DAG construction with Datalog, and is package manager agnostic.

Build Systems Based on Logic Programming Biomake [23] and Prom [27] are build systems [29] based on Prolog [14]. They both extend the capabilities of make [37] and therefore are 1) designed to model file dependencies, 2) Turing complete, *i.e.* allow expressing non-terminating computations, and 3) non-declarative, *i.e.* the success of evaluation depends on the order of literals in clause bodies. Modus is designed to model a new domain of container images, which differ from a traditional filesystem by their layered design (Section 3.2). Since Modus is a Datalog dialect, it is 1) not Turing-complete, which guarantees that computations of the build DAG for each target terminate, and 2) declarative, which allows the developer to arbitrarily order image layers without being constrained by the Horn clause evaluation strategy.

Datalog Implementations and Extensions Although Datalog is a precisely defined fragment of Horn clauses, its numerous implementations [26, 32–34] implement different sets of non-standard features, called extensions, that are chosen to better model the application domains. To the best of our knowledge, our work is the first that applies Datalog, and logic programming in general, to the domain of container image builds. Modus, as many other Datalog implementation, supports built-in predicates. Although we used the standard approach [16] to support them, the set of predicates is different. Specifically, Modus implements container-related predicates, such as `from` and `run` (Section 4). As explained in Section 4.2.3, Modus allows using rules with non-grounded variables. DES [33] also allows rules with non-grounded variables, but, since it is implemented in Prolog, the semantics of such rules is different. DES may return literals with uninitialised variables as outputs. In Modus, all variables must be initialised, since all build parameters must be concrete to build an image. Thus, we implemented a custom approach for handling non-grounded variables, discussed in Section 4.2.3. Most Datalog implementations are designed as query languages, so their semantics is defined via the facts they infer. In contrast, Modus' semantics is defined via proof trees, and it searches not for arbitrary, but minimal, proof trees. Previously, minimal proof trees were used to assist in debugging Datalog programs [39]. Modus implements advanced support for string manipulation using the predicate `string_concat` and adapts stratified construction [13] to ensure the tractability of solving string constraints. Although stratified construction limits the number of concatenation applications during evaluation, this restriction can potentially be lifted by, for example, implementing a generalized sequence transducer [13].

8 Conclusion

This paper introduces Modus, a novel application of logic programming to building container images. The key intuition of Modus is that proofs of Horn clauses naturally map to container image builds and therefore Horn clauses, more specifically a decidable fragment of Horn clauses called Datalog, is a suitable formalism for container image build definitions. When designing Modus, we identified several Datalog's extensions, including domain-specific built-in predicates and operators, non-grounded variables, and advanced string manipulations, that facilitate image build definitions. An evaluation on popular Docker Hub projects revealed that Modus substantially reduced the code size of container image build systems while introducing only a negligible performance overhead, preserved the original image size and image efficiency.

References

[1] [n.d.]. Best practices for scanning images. https://docs.docker.com/develop/scan-images/. Accessed: 2022-03-17.

[2] [n.d.]. The Docker Hub. https://hub.docker.com/. Accessed: 2021-08-19.

[3] [n.d.]. Docker Multi-stage Build. https://docs.docker.com/develop/develop-images/multistage-build/. Accessed: 2021-08-19.

[4] [n.d.]. The Dockerfile Reference. https://docs.docker.com/engine/reference/builder/. Accessed: 2021-08-19.

[5] [n.d.]. The Earthly build system. https://earthly.dev/. Accessed: 2021-08-19.

[6] [n.d.]. The GNU Guix package manager. https://guix.gnu.org/. Accessed: 2021-08-19.

[7] [n.d.]. JQ command-line JSON processor. https://stedolan.github.io/jq/. Accessed: 2021-08-19.

[8] [n.d.]. moby/buildkit: Concurrent, cache-efficient, and Dockerfile-agnostic builder toolkit. https://github.com/moby/buildkit. Accessed: 2021-11-18.

[9] [n.d.]. The OpenJDK implementation of Java. https://openjdk.java.net/. Accessed: 2021-08-19.

[10] [n.d.]. SemVer specification. https://semver.org/. Accessed: 2022-03-17.

[11] Alfred V Aho, Brian W Kernighan, and Peter J Weinberger. 1979. Awk—a pattern scanning and processing language. *Software: Practice and Experience* 9, 4 (1979), 267–279.

[12] Carl Boettiger. 2015. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review* 49, 1 (2015), 71–79.

[13] Anthony Bonner and Giansalvatore Mecca. 1998. Sequences, datalog, and transducers. *J. Comput. System Sci.* 57, 3 (1998), 234–259.

[14] Ivan Bratko. 2001. *Prolog programming for artificial intelligence*. Pearson education.

[15] Neil Brown and collaborators. [n.d.]. The OverlayFS filesystem. https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html. Accessed: 2021-08-19.

[16] Stefano Ceri, Georg Gottlob, and Letizia Tanca. 2012. *Logic programming and databases*. Springer Science & Business Media.

[17] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. 1989. What you always wanted to know about Datalog(and never dared to ask). *IEEE transactions on knowledge and data engineering* 1, 1 (1989), 146–166.

[18] Ludovic Courtès. 2013. Functional package management with guix. *arXiv preprint arXiv:1305.4584* (2013).

[19] Eelco Dolstra and Andres Löh. 2008. NixOS: A purely functional Linux distribution. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. 367–378.

[20] Open Repository for Container Tools. [n.d.]. The Buildah build system. https://buildah.io/. Accessed: 2021-08-19.

[21] The Linux Foundation. [n.d.]. Open Container Initiative. https://opencontainers.org/. Accessed: 2021-08-19.

[22] Alex Goodman and collaborators. [n.d.]. Dive, a tool for exploring each layer in an image. https://github.com/wagoodman/dive. Accessed: 2022-03-04.

[23] Ian H Holmes and Christopher J Mungall. 2017. BioMake: a GNU make-compatible utility for declarative workflow management. *Bioinformatics* 33, 21 (2017), 3502–3504.

[24] Docker Official Images. [n.d.]. Docker Official Image packaging for Java. https://github.com/docker-library/openjdk. Accessed: 2021-08-19.

[25] Docker Official Images. [n.d.]. jq-template.awk. https://github.com/docker-library/bashbrew/blob/b2cbaa3678ffbce7c0d90be5a518c1ec068011b5/scripts/jq-template.awk. Accessed: 2022-02-25.

[26] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*. Springer, 422–430.

[27] Thilo Kielmann. 1991. *PROM: A flexible, PROLOG-based make tool*. Technical Report. Citeseer.

[28] Dirk Merkel et al. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.

[29] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build systems à la carte. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–29.

[30] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. 2017. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing* 7, 3 (2017), 677–692.

[31] Andrey Ponomarenko. [n.d.]. pkgdiff. https://lvc.github.io/pkgdiff/. Accessed: 2022-03-16.

[32] Leonid Ryzhyk and Mihai Budiu. 2019. Differential Datalog. *Datalog* 2 (2019), 4–5.

[33] Fernando Sáenz-Pérez. 2011. DES: A deductive database system. *Electronic notes in theoretical computer science* 271 (2011), 63–78.

[34] Jiwon Seo, Stephen Guo, and Monica S Lam. 2013. SociaLite: Datalog extensions for efficient social network analysis. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 278–289.

[35] Abdullah Sheneamer and Jugal Kalita. 2016. A survey of software clone detection techniques. *International Journal of Computer Applications* 137, 10 (2016), 1–21.

[36] Rui Shu, Xiaohui Gu, and William Enck. 2017. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. 269–280.

[37] Richard M Stallman and Roland McGrath. 1991. GNU Make-A Program for Directing Recompilation. (1991).

[38] Ole Tange et al. 2011. Gnu parallel — the command-line power tool. *The USENIX Magazine* 36 (2011), 42–47.

[39] David Zhao, Pavle Subotić, and Bernhard Scholz. 2020. Debugging large-scale datalog: A scalable provenance evaluation strategy. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42, 2 (2020), 1–35.

[40] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S Warke, Mohamed Mohamed, and Ali R Butt. 2019. Large-scale analysis of the docker hub dataset. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–10.