Abstract of thesis entitled

"From Conventional to Search-focused Programming Environments"

Submitted by

Tomáš Tauber

for the degree of Doctor of Philosophy

at the University of Hong Kong

in August 2017

Conventional programming environments operate with a complex interplay among different tools for accessing different resources related to source code. These resources contain various metadata, such as code dependencies, naming organization or version control. Each kind of metadata carries its specific format, such as binary or semi-structured files of various formats, and corresponding access methods.

Code entities - such as functions, constants or variables - do not have globally unique persistent identifiers in conventional programming environments. Access to external code is name-addressable and resembles hierarchical file systems. User-chosen hierarchical names, however, are not globally unique. Name conflicts may, therefore, arise among definitions from different libraries or even the same library of different versions. The lack of persistent globally unique identifiers prevents code entities from being directly associated with some metadata. The non-unique identifiers depend on an implicit context: user-chosen identifiers may refer to different entities at different times or locations. Additionally, conventional programming environments specify project-level dependencies in which an unused code may need to be included when dependencies are resolved.

To address these issues, this thesis proposes a design for programming environments, named Search-focused Programming. Search-focused Programming environments have three key properties. First, code entities are deterministically assigned globally unique persistent identifiers based on their content. With unique identifiers, relations can associate dependencies among code entities directly. Second, code and related metadata are uniformly stored. The uniformly stored data can be accessed in two ways: by retrieving data using a declarative query language, and by reading and writing individual records in the transaction log. Third, the shared source code is separate from the locally edited code. Two proof-of-concept prototypes demonstrate two different directions in which one can interact with code. The first prototype presents a command-line interface, providing indirect interactions similar to conventional programming environments. The interaction is realized through bulk processing local source code files and generating corresponding queries or transactions. The second direction is in the form of a web-based interactive interface. The interaction happens by directly looking up individual code entities and manipulating them in a structure-driven editor, i.e. without the bulk processing of layers of naming structures. Finally, this thesis examines the effects of definition-level dependency tracking on existing source code bases. A popular community project repository was analysed, and the source code of the top five most depended on projects was found to never be fully utilized by libraries that declared their dependencies on them. A substantial fraction of the code entities in these five projects remained in more than one version of their respective libraries. Lastly, overhead was measured in compilation speed on client code due to unused code, which would be removed with definition-level dependencies. These results suggest the possible merits of employing the ideas from Search-focused Programming in existing programming environment infrastructures.

An abstract of exactly 458 words.

From Conventional to Search-focused Programming Environments

by

Tomáš Tauber

B.Sc. (Hons.) *The University of Edinburgh*

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy at the University of Hong Kong.

August 2017

To the memory of my grandparents, František (†2016) and Marie (†2017).

"In the practical world of computing, it is rather uncommon that a program, once it performs correctly and satisfactorily, remains unchanged forever." [1]

– Niklaus Wirth

Declaration

I declare that this thesis represents my own work, except where due acknowledgement is made, and that it has not been previously included in a thesis, dissertation or report submitted to this University or to any other institutions for a degree, diploma or other qualifications.

Signed [illegible]

Tomáš Tauber

This page is intentionally left blank.

Acknowledgements

I thank, first and foremost, my supervisor Dr. Bruno C. d. S. Oliveira. The discussions with him were always stimulating and encouraging, provided hints and left me with freedom to pursue the research direction that interested me most. I appreciate his patience and positive attitude that helped me to overcome difficult challenges during my studies. I am also grateful to my secondary supervisor Prof. Francis C. M. Lau for his general guidance and encouragement. It was a pleasure to serve as a teaching assistant in his Compiling Techniques course.

I feel fortunate that I met so many brilliant students and colleagues throughout the course of my studies. I would like to thank all former and current members of the Systems Research Group: Zhiquan Lai, Abdel Rahman F.A.K. Hegazi, Dominic Chit Ho Hung, Luwei Cheng, Mingzhe Zhang, Wu Hao, Dengke Zhang, King Tin Lam etc. My life at the University of Hong Kong would not have been as enjoyable without their help with my first steps. Thanks also goes to all students that have worked on PL-related research: Xuan Bi, Ningning Xie, Yanpeng Yang, Zhiyuan Shi, Weixin Zhang, Haoyuan Zhang, Yanlin Wang, Huang Li, João Alpuim, Jinxu Zhao, Xuejing Huang, Boya Peng, Zhenrui Zhang, Johnny Lin, Kevin Chang etc. I enjoyed all our discussions and any work we have done together.

I would like to thank Vadim Zaliva, Dr. Marco Servetto and all people that I met at various academic gatherings and had stimulating discussions with. I would also like to thank Dr. Jonas S. Karlsson, Dr. Kim Batselier, František Polach and many other hackers or scholars outside HKU's Department of Computer Science that I met in Hong Kong. All these exchanges inspired me a lot and shaped my attitude towards research.

I must also thank the technical staff (Patrick T.C. Au, Daniel Hung, etc.) and the general office (Priscilla Chung, Maria Lam, etc.). They are all extremely helpful and thanks to their excellent and responsive work, research and teaching went very smoothly.

Additionally, I would like to express my thanks to Franz Kafka and Joseph Heller. Their work helped me to stay calm and navigate through various procedures which bore a resemblance to their novels.

I am deeply grateful to my family for all their support and encouragement over all these years. Finally, I thank my girlfriend for always being there for me even during difficult times in my life.

Contents

List of Figures

List of Tables

This page is intentionally left blank.

List of Code Listing

This page is intentionally left blank.

Chapter 1

Introduction

In this thesis, we explore an alternative to how we can access and manage external code and metadata associated to external code. Most computer programs we write rely on external third party code. For example, in the C language, we access the external code using the include directive:

```
#include "library.h"

int main (void) {
  library_function(42);
}
```

This code snippet assumes that the symbol `library_function` is exported by the header file `library.h` and that the header file and the corresponding object file can be found on the local file system. One inherent problem of this *name-addressable* access to external code is the possibility of name collisions, i.e. if other used libraries export symbols of the same name. More modern languages, such as C++, try to reduce this problem through the use of *namespaces*. Namespaces are abstract hierarchical structures that provide scoped access to external symbols. Even though namespaces reduce name collisions, we still encounter problems when faced with software development tools for interacting with external code and metadata.

1.1 Conventional Programming Environments

In this section, we describe our problem setting, what we mean by conventional programming environments, and identify key challenges in this setting. A programming environment is a combination of programming language and software project management tools and processes. We briefly overview example tools and their metadata in Section 1.1.1, and show detailed examples in Chapter 3.

1.1.1 External Code and Metadata

When using external third party code, we will most likely retrieve and manage it and its related metadata with a combination of software development tools. Despite the widespread usage of such tools for interacting with external code and metadata, programming languages do not account for them in their abstractions.

Version Control Metadata. Both our client code and the external library code are likely to reside in a version control repository. A popular version control system Git [2] implements a userspace content-addressable filesystem and provides a version control interface to its stored objects (which are assigned unique identifiers using a cryptographic hash function). Internally stored objects include actual physical files and directory trees as well as versioning metadata, such as commits, references or tags.

Package and Build Management Metadata. Usually, we do not manually manage the external library code, but rely on semi-automatic "package managers". The library author "packages" source code with build scripts and files describing various package metadata (dependencies, release version numbers, licenses etc.) and submits this package into a centralized repository. The centralized repository may have additional semi-automated package approval procedures before the package is exposed to "package managers". The policies in this regard vary across different language communities: some prefer immediate publication without any approval, others may enforce policies on naming or versioning conventions, signalling upgrades to dependent packages etc.

We specify what packages we depend on and their version bounds and run the package manager. The package manager may either download all dependent packages and isolate them with their own transitive dependencies, or construct a graph of transitive dependencies and run a constraint solver on version bounds. In the latter case, the solver may fail or find a version-compatible set of packages. If it succeeds, the package manager then downloads all the transitive dependencies and sets up a build procedure (e.g. changes the compiler search path for these packages). The package manager may also provide an environment isolation, i.e. dependent packages of other projects are not seen in the current project (unless they are specified as dependencies).

Test and Performance Metadata. In addition to the actual library code, the library author can also include suites of performance benchmarks and unit, integration or functionality tests. Continuous integration servers routinely compile the library code, run its associated test suites, and record the outcome of each execution. Even though package managers do not utilize these additional forms of metadata, we may be interested in them. In the presence of multiple libraries of the same functionality (or multiple versions of the same library), looking at such metadata can help us decide which library or version to choose. For example, some tests may express the behaviour we expect or we may be interested if the library was tested on a particular operating system. In any case, we usually investigate these forms of metadata manually in the user interface of continuous integration software.

Documentation, Code Review, and Bug Tracking Metadata. Beyond metadata for managing and assessing external code, we expect to find code

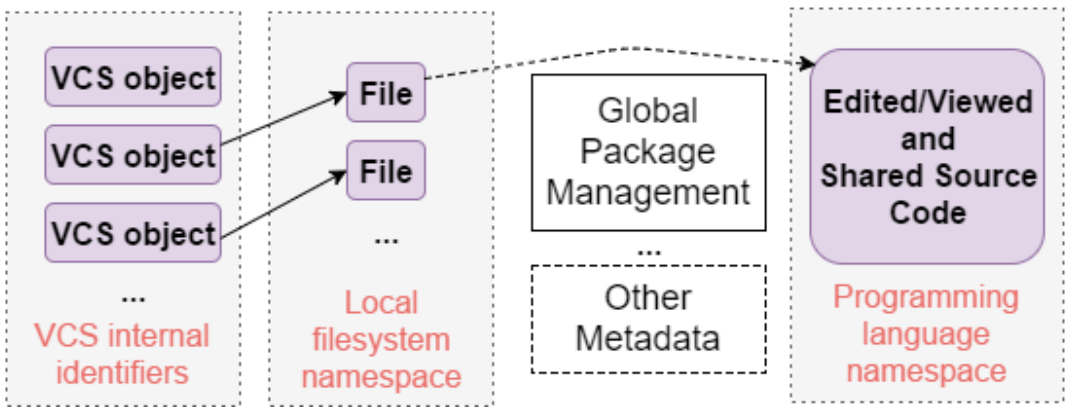

VCS object
VCS object
VCS object
...
VCS internal identifiers
File
File
...
Local filesystem namespace
Global Package Management
...
Other Metadata
Edited/Viewed and Shared Source Code
Programming language namespace

Figure 1.1: Conventional programming environment.

documentation. Code documentation exists in the form of embedded source code comments or external files in various formats. Depending on a scenario, we may either utilize a development environment or a web search engine for finding relevant code documentation. Other than usage and functionality documentation, we may find other specialized documentation: reviews of code changes, feature requests or bug reports. Additional tools usually store and manage this specialized documentation. Again, we may manually investigate these forms of metadata when working with external code.

Other Metadata. The above cases are only examples of what metadata we encounter during development with external code. We can naturally utilize other tools and metadata – for example, we may be interested in results of analysis that computes code coverage of external code. Some metadata may also not be explicitly stated – for instance, what style convention the external code follows. They are, however, all potential factors we may consider when working with external code.

Figure 1.1 summarizes our view of conventional programming environments. There are three layers of naming abstractions between the actual stored external code and its usage in programming language constructs. In this case, external source code originates from a version control system (VCS) repository (i.e. it is assigned an internal identifier in VCS). This stored code is then

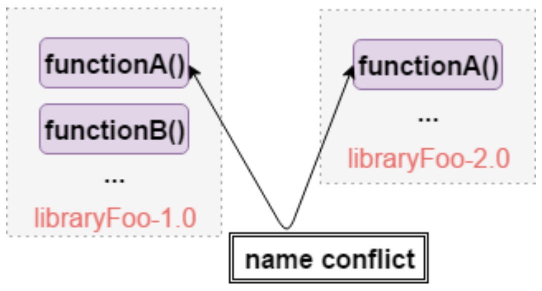


Figure 1.2: Identifier conflicts.

exposed either directly as a source file (for global compilation) or as object and interface files (for separate compilation). These files are actually retrieved based on interactions and versioning information in the package manager's metadata from a global repository. Finally, in the programming language semantics, we have constructs for name-based accessing of external code that vaguely relate to names in the local filesystem.

Generally, the code we edit is the same as the source we share and expose through this layered mechanism. Programming language constructs are often oblivious to the fact that the external code originated from elsewhere. In other words, the programming language abstraction is that source code only resides in certain paths of a local filesystem. Only some metadata is directly available from within the programming language constructs, the rest relies on pre-processing and different tools.

In this section, we described our general assumptions about conventional programming environments. There exist various exceptions to this setting, for example, some languages may offer or require qualifying the names in import statements by URLs (e.g. in Go, "the repository URL and import path are one and the same" [3]). Nonetheless, they may still share features of conventional programming environments we described (e.g. name-based access or no direct access to some metadata).

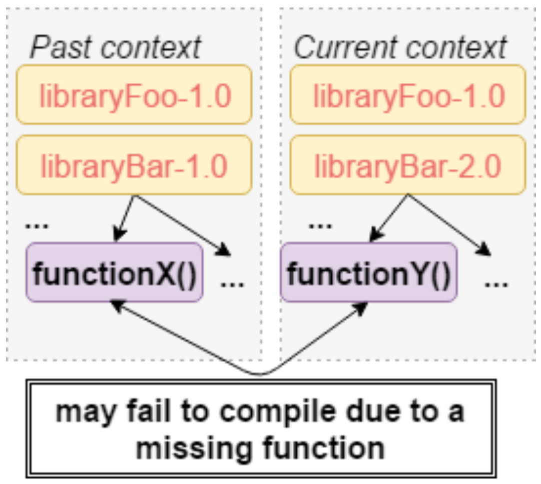


Figure 1.3: Context dependence.

1.1.2 Key Challenges

In this section, we overview issues that arise in the setting we described in Section 1.1.1. Firstly, we look at challenges due to name-addressable access, i.e. a lack of globally unique persistent identifiers.

The profound problem of *identifier collisions* (Figure 1.2) falls into this category. Open source libraries are independently developed by different individuals and organizations. As such, name collisions of symbols exported by different libraries may occur. Centralized package repositories can enforce policies to prevent name collisions across different libraries. Even with that, the same library can have name collisions with itself if we consider its different versions. Package-level dependency trees can contain different versions of the same library on different paths which may or may not be resolved by the package manager.

The other problem in this category is *context dependence* (Figure 1.3). The programming language abstraction of compilation is side-effect free, i.e. given the same input source code, we should obtain the same output target code. This abstraction, however, depends on implicit local and global context. If we compile the same code on two different machines, we may get different outputs, because it was linked with different libraries. If we compile the same code at different times on the same machine, we may get different outputs, because the external code was changed (e.g. updated by a package manager). This situation is common. For example in the CRAN repository of the R programming language community, 41% of the errors in a two year history of packages were caused by incompatible package-level changes [4].

Secondly, challenges arise due to a lack of globally unique persistent identifiers in combination with heterogeneous metadata that we encounter in programming environments. The key challenge here is *directly relating code and metadata*, i.e. without processing or analysing code. For example, at a given time, the continuous integration system stored a successful result of running a set of unit tests for a given function. We would like to know which versions of the library this test set would pass as well. We have different indirect options for how we could find this answer. One possibility is going through the library's version history and repeatedly executing these tests. The other possibility is running an analysis on the repository which identifies revisions where this function and its dependencies are identical with the tested one. Due to refactoring changes, this analysis may be complex and involve a trade-off between time and precision. Nonetheless, these options are not direct.

One related challenge is the *impedance mismatch* between code and related metadata. This issue resembles the traditional OO-relational or multi-tier web programming impedance mismatches [5]. As mentioned, there is a variety of code-related metadata with many relations among different instances of metadata and code. Some of this information fails to be mapped onto a filesystem structure which only provides a naming hierarchy. The challenge of *directly relating code and metadata* is a consequence of code-metadata *impedance mismatch*. One other consequence is the absence of uniform interfaces for retrieving and managing code-related metadata.

Thirdly, challenges arise due to *coarse-grained code distribution* (Figure 1.4). Given our client code may not utilize 100% of the exposed library symbols, we may experience inefficiencies or complications. The compilation times

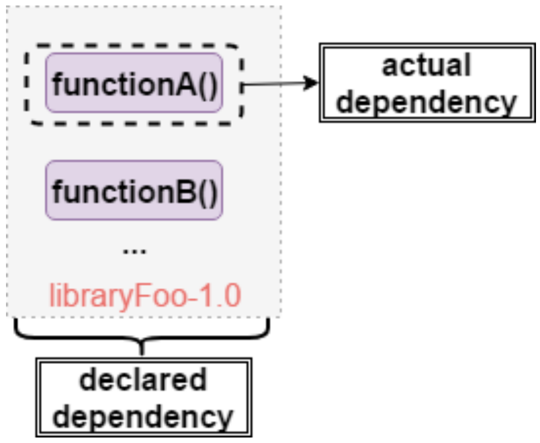


Figure 1.4: Coarse-grained distribution.

can increase due to additional processing of the unused library parts. Furthermore, the unused library parts could fail to compile or cause dependency conflicts. Even though we do not need these unused library parts, coarse-grained dependency specifications bring them in and could cause inefficiencies or complications in our build process.

In this section, we summarized key challenges in conventional programming environments. These challenges occur due to three factors: a lack of globally unique persistent identifiers in name-addressable code access, heterogeneity of code-related metadata, and coarse-grained code distribution.

1.2 Search-focused Programming Environment

In this section, we describe the novel solution that addresses challenges found in conventional programming environments. In particular, we propose Search-focused Programming (SFP) as an alternative to conventional programming environments. Search-focused Programming environments consist of several parts:

1. A separation between source code that is interacted with and source code that is viewed or shared.

2. Identifiers of shared source code and related metadata, such as static

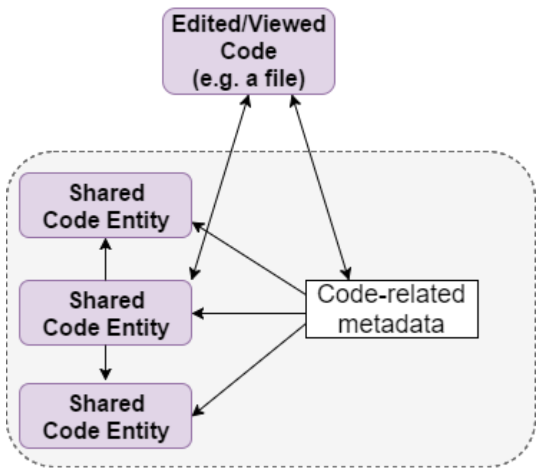


Figure 1.5: Search-focused programming environment.

dependencies, are persistent, globally unique and location-independent. Identifiers are assigned through a content-addressable scheme.

3. Finer granularity of global code dependency tracking is used.

4. Declarative query language for retrieving versioned code declarations and metadata.

5. Interface for reading and writing new shared code and metadata.

Figure 1.5 summarizes relationships and interactions in a Search-focused Programming environment. Code is edited or viewed through a local interface – be it a traditional offline text-oriented editor with files and some explicit synchronization, or a structural online editor. Dependencies among individual code entities (functions, protocols, variables etc.) are stored and captured directly. At the same time, other code-related metadata can be associated with individual code entities through their unique identifiers.

The aim of SFP is to eliminate problems we identified in conventional programming environments as well as enable new applications in a distributed setting. We demonstrate these ideas by adapting an existing programming language to this setting and conducting a study on its community-maintained code repositories.

Name collisions do not occur by design: code definitions are assigned globally unique identifiers based on their content, e.g. two function definitions would have the same global identifier only if they were the same definition. Metadata can directly refer to code using its location-independent identifier and can be inserted and retrieved in a homogeneous way through the respective interface and declarative query language. Context, such as assigned human-readable names at a time, is explicit and can be retrieved using a declarative language. Code distribution happens at a finer granularity and prevents or reduces issues associated with coarse-grained code distribution.

Overall, the vision is to make programming environments "searchable" in a distributed setting, i.e. to be able to access the entire environment in an uniform way and issue queries that produce consistent results everywhere. Besides addressing the challenges described in Section 1.1.2, it opens up new opportunities in programming environments that we describe in Chapter 7. In this section, we briefly described Search-focused Programming and how it addresses challenges present in conventional programming environments.

1.3 Contributions

In this section, we summarize individual contributions of this thesis to alternative programming environments. Programming environments are used in decentralized settings and pose various challenges when it comes to interacting with external shared code. At the same time, with years of work put into programming languages, environment tools, training and existing open-source code, clean-slate solutions may not be viable. In this thesis, we present Search-focused Programming as a distributed environment architecture and its concepts may be useful on their own and backported to existing tools. While Search-focused Programming can facilitate different ways of interaction with shared code and related metadata, we also demonstrate it can simulate existing workflows of conventional programming environments. By adapting an existing language, we examine compatibility of various language features with this new setting.

More concrete contributions of this thesis are as follows:

- This thesis introduces concepts behind Search-focused Programming. Together, they serve as a basis for modular programming environment architecture in a distributed setting. It can accommodate and expose various kinds of metadata present in programming environments, while representing all data (i.e. code and metadata about it) uniformly.

- This thesis studies an existing language, ClojureScript, and shows how it can work within a Search-focused Programming environment. The thesis considers compile time dependencies: the presented work may be applicable to languages with static name resolution (C, Java, Erlang, etc.) or in a limited context in other languages as well. Runtime dependencies, metaprogramming interactions or dynamic name resolution remain as future work.

- Two different usage ways are discussed in contrast to compare textual usage compatibility and more experimental interfaces. Proof-of-concept implementations of both of these approaches show the flexibility of interfaces for transacting and declaratively retrieving code and metadata.

- Given the interaction usage is separate from storage, two usage ways are different but compatible. With that, this thesis shows how existing code could be imported and utilized using a prototype command-line interface.

- This thesis analyses existing source code repositories using modified portions of the prototype implementation and infers their characteristics. In particular, within the community project repository Clojars, about 44% to 78% of shared exposed code of five most depended on ClojureScript libraries was not used by other projects. The unused code of two libraries increased cold start compilation time of their example client code projects 0.23x to 11.04x. This overhead is reduced by using definition-level dependencies in Search-focused Programming environments. The results demonstrate some of the challenges in conventional programming environments. They also suggest possible merits if concepts from Search-focused Programming are employed in existing programming environment infrastructures.

1.4 Thesis Outline

This chapter briefly introduced the setting of programming environments – the layers of naming structures are used when accessing code and kinds of metadata present in programming environments. It identified several key challenges in conventional programming environments and set out to tackle them using Search-focused Programming.

Chapter 2 reviews related work in programming languages, software engineering, and various system implementations. It guides motivation for some decisions in this thesis.

Chapter 3 walks through a workflow for a programming environment around ClojureScript. It shows how projects are set up, versioned, shared, tested and how related metadata can be retrieved.

Chapter 4 gives a conceptual overview of Search-focused Programming. At first, it poses several requirements that motivate further design choices. It then describes issues surrounding the assignment of persistent globally unique identifiers for code distribution and dependency tracking. Following this, it describes mutable and persistent metadata, and two ways for interacting with the programming environment: a declarative query language for retrieving aggregated information at a given time point, and log-based reading and writing of new information.

Chapter 5 then shows two proof-of-concept implementations of interfaces. The first one is a command-line interface that aims to preserve backwards compatibility with the processes of conventional programming environments. The other is a web-based interface that shows a more interactive approach to working with shared data in a programming environment. They both demonstrate the flexibility of the underlying architecture for working with a Search-focused Programming environment.

Chapter 6 examines existing source code repositories. This analysis reveals some of the issues originating from the conventional programming environment design. Experiments with example projects show overhead in compilation time which would be reduced with more fine-grained code distribution, as present in Search-focused Programming. It validates the approach's applicability in existing code structures.

Chapter 7 summarizes the overall findings and discusses different extensions suitable for future work. Future work looks at adopting more expressive query languages, different challenges for the storage layer, granularity of dependency tracking, and related applications.

This page is intentionally left blank.

Chapter 2

Literature Review

2.1 Introduction

In this chapter, we overview different lines of work related to concepts in Search-focused Programming (SFP) environments. They concern different aspects related to different features of SFP environments.

The first aspect is creation of persistent unique identifiers for code entities or code-related artifacts. Section 2.2 describes work that aims to assign identifiers to software for the purpose of scholarly attributions. This line of work may share some goals for identifier assignment in SFP, but has a different focus (citable contributions) and means for assignment. Section 2.4 then looks at package managers, file systems, and distributed systems. Similarly to SFP, some of these systems use content-addressable schemes for assigning globally unique identifiers, but have different aims (e.g. OS-level package management) and respective differences (e.g. different granularity of dependencies).

The second aspect is deterministic code management, i.e. retrieving the same result for the same input at a specified time, through different means. Section 2.4 references work in that direction, such as reproducible build systems. Section 2.5 looks at software engineering work and some of it may help with deterministic code management, e.g. the practice of monolithic source code repositories. Some of this line of work leads to similar effects as SFP, but makes different assumptions (e.g. all within a single organization) from what SFP environments a priori make.

The third aspect is code and related metadata organization. This topic spans across several areas. Section 2.3 covers work around processing code-related metadata for different uses, Section 2.5 then contains software engineering practices and development environments that enable working with code-related metadata. Some of them provide a uniform access to code-related metadata, similarly to SFP environments, but focus on localized environments with different assumptions and may deal with non-persistent identifiers and potential non-determinism.

The final aspect is experimentation with software development and programming environments. Various systems and programming languages are presented in Sections 2.4 and 2.5. Some of their architecture may share common features with the interactive prototype we describe in Chapter 5, such as more fine-grained program manipulation, but may rely on location-dependent identifier assignment.

In addition to the mentioned aspects, this chapter overviews other lines of related work. This line of work may not necessarily fit within the scope of the mentioned aspects, but is still related to SFP, such as various software engineering tools and techniques.

2.2 Persistent Identifiers

Over centuries, libraries and publishers developed and used standard long-lasting references for reliable and efficient finding sources of documents [6]. In the context of digital documents, systems administered by institutions usually create and maintain persistent identifiers. For example, a federation of registration agencies (appointed by the International DOI Foundation) manage the popular Digital Object Identifiers (DOI) [7].

Recently, research communities started to put more emphasis on assigning persistent identifiers to other scholarly materials and datasets. For example, DataCite [8] aims to manage DOI for published research datasets. Software packages could be seen as datasets, but given the continuous software development process and other factors, new approaches are proposed specifically for software. One of such recent efforts is CodeMeta [9] which describes a minimal metadata schema for published scientific software and code. Such schemas, once standardized, could be used as a part of ontologies for capturing project-level metadata in SFP environments.

While SFP shares the goal of reliable and efficient finding of code, it aims to capture general software development. As such, it does have not the primary goal of creating citable contributions for the scholarly record and has other requirements that may be in conflict with some restrictions in the scientific software domain. We expand on these requirements in Chapter 4. In particular, Section 4.2.2 assesses centralized identifier assignment with the respect to posed requirements for SFP environments.

2.3 Source Code and Metadata Processing

Various ways are used to express additional information about source code. We give detailed examples of different metadata sources in Chapter 3. Source code itself may utilize comments, documentation strings, marker interfaces [10], annotations and similar mechanisms. Their usage ranges from extra processable information (e.g. marking code as deprecated) to additional language concepts (e.g. design-by-contract). Using naming conventions to describe concepts beyond original builtin features of a programming language may cause problems. Explicit Programming [11] was proposed to address these problems.

SFP does not focus on extending languages with additional concepts, but on integrating metadata sources and providing a uniform access to them. With these goals, adding and processing metadata about source code can be done outside of language syntax and semantics. We expand on processing metadata for extending languages in Chapter 7.

Some software development tools specify processing of code-related metadata in a declarative way. The most related to our work would be prom [12] and Gerrit [13]. The prom tool was a replacement for the `make` build system and used Prolog to specify build tasks. Gerrit is a tool for code review on top of the Git version control system and allows specifying logical rules for code changes. While the query language in SFP is not as expressive as Prolog, it can facilitate some similar uses. We expand on the possibility of a more expressive query language in Chapter 7.

2.3.1 Linked Data

Structured data exchange and integration appear as frequent topics in the Semantic Web applications. In the domain of software development, software chrestomathies [14] aim to connect code with documentation and other software artifacts, mainly for educational purposes. Nava [15] is an experimental language where software components were discovered and used via semantic queries. For the implementation, they used Cyc, an artificial intelligence project that includes an ontology, a knowledge base, a reasoning engine, and a specification engine. The architecture behind blackboard systems [16] shares some properties with the temporal view of the repository state in SFP environments.

This line of work shares some common goals with SFP, such as uniform programming access to metainformation about code. SFP, however, has other requirements (that we detail in Chapter 4) that motivate other design choices. For the read access at a certain time, we choose a dialect of Datalog. In contrast with semantic queries or artificial intelligence software, Datalog only allows limited domain-specific reasoning. The reason for choosing Datalog over a more powerful or expressive query language is motivated by our requirements: its behaviour is deterministic and has predictable performance. We discuss alternative query languages in Chapter 7.

If we view source code as databases, the database normalization [17] aims to reduce redundancy and improve integrity. These goals are somewhat shared in SFP environments with tracking code dependencies on a finer level. Shared programming environments, however, need some degree of redundancy for preserving the version history as well as due to the nature of software development (e.g. the existence of multiple forks). With that in mind, the expectation of "absence of update anomalies" is in contradiction with the expectations of programming environments.

2.4 System Software

A wide variety of system software has been an inspiration for some of concepts behind SFP. We overview system software areas that are close in their features or goals to some portions of SFP.

2.4.1 Package Management

System and programming language package managers overlap in their functionality. They generally differ in their purpose. Programming language package managers naturally aid software development in a given language. With that goal in mind, their package repository processes (e.g. publishing and using latest package versions) or functionality (e.g. allowing environment isolation) may be tuned to particular language community standards. Nonetheless, even system package managers are related to our work.

Traditional package managers of Linux distributions contain metadata in their package formats, such as RPM or DEB. One relation to our work is their advanced function for managing dependencies through *virtual package* or *feature* specifications. Similar behaviour in SFP environments could be achieved by a custom dependency resolution for queries that return multiple results.

In recent years, open source communities put more emphasis on "reproducible builds" [18] which "aim to provide a verifiable path from software source code to its compiled binary form." Two issues appear in old systems without these reproducible build guarantees: 1) conflicts among packages due to unrestricted pre-install and post-install side-effecting scripts [19], 2) non-determinism in the build process (e.g. due to timestamps in produced outputs) [18]. "Reproducible builds" eliminate or reduce these issues and different directions exist how to realize "reproducible builds".

One direction for achieving "reproducible builds" comes with recent "functional package managers", such as Nix [20] or Guix [21], or even earlier with the Vesta [22] *Software Configuration Management* system. Inspired by functional programming, these package managers aim to provide an immutable abstraction over the system environment and purely functional behaviour of build procedures: given the same inputs, we obtain the same outputs on any system. Inputs include package dependencies, build configurations and steps, while outputs are package binaries assigned with unique identifiers (which are computed as cryptographic hashes from inputs and possibly contents). In addition to that, these package managers manage symbolic links to isolate different package versions (which end up with different identifiers) and rely on community-curated build procedures patched for determinism. Nix has a custom external domain specific language for managing packages, while Guix uses GNU/Scheme as an internal domain specific language. Similarly to Nix, Vesta used an external domain specific language. Vesta focused on specifying build procedures and used a virtual file system that ensured reproducible builds through a distributed append-only namespace.

SFP shares some goals with functional package and software configuration management, such as determinism. On the other hand, it does not focus on general operating system-level build procedures and package distribution where packages may contain unused code, and focuses more on aiding software development and managing code-related metadata.

2.4.2 File Systems

The idea in functional package management for assigning unique identifiers based on content (of inputs and possibly outputs) is itself inspired by work in storage systems on content-addressable data management. We give examples with Git's internal storage in Chapters 1 and 3, so we omit an extensive discussion here. In general, Git could potentially serve as a storage layer for SFP environments. One subtle point is that Git's binary blob objects have identifiers generated purely based on their content without additional inputs (e.g. transitive dependency structure). If Git was used as a storage layer in SFP environments, the Git storage-backed implementation would need to account for this identifier assignment scheme difference (e.g. by creating auxiliary binary objects with embedded code dependency information in their format).

Apart from this line of work that motivated the persistent identifier assignment in SFP, other new file systems aim for more flexible metadata management. For example, TagFS [23] or hFAD [24] allow more flexible file organization through attaching labels to files. Users can then find files by issuing search queries using these labels. Hierarchical organization is a special case in these more general file systems. This motivation is shared with code organization and distribution principles in SFP.

2.4.3 Distributed Systems

Ideas from content-addressable storage expand to distributed systems because of their advantages, such as elimination of duplicate data or implicit integrity checks of stored data. A distributed hash table, such as Content-addressable Networks [25], can serve hash table-like functionality for Internet-scale applications, such as peer-to-peer data sharing networks. One recent example is

Inter-Planetary[1] File System (IPFS) [26] which is a peer-to-peer protocol "designed to create a permanent and decentralized method of storing and sharing files". IPFS is an example of a content-addressable storage: each shared file is assigned a unique persistent identifier based on its content. For mutable references, IPFS has a service named "Inter-Planetary Naming System" (IPNS) where references can be stored and updated under identifiers computed as hashes of authors' public keys. At the time of writing, IPFS is still under development and various projects and initiatives are debated and developed. The most related issue to our work is the idea of a programming language with code stored on IPFS. One experimental language for this purpose is Annah [27] (and its corresponding core language Morte). In Annah, all expressions are encoded in calculus of constructions (using the Boehm-Berarducci encoding of algebraic data types [28]) and stored in a textual representation. After reading an input expression, Annah's compiler downloads and parses external expressions, and super-optimizes the whole program that can then be stored on IPFS. Despite practical issues (e.g. inefficiencies due to the chosen encoding), purely functional languages, such as Annah, can bring foundations for programming languages designed from scratch to fit distributed programming environments. In SFP, the initial focus is on adapting existing languages rather than creating fresh ones. The usage of decentralized storage layer, such as IPFS, is discussed as future work in Chapter 7.

The extent of various distributed systems in related domains is large, so we only list several notable software projects. GitTorrent [29] is a peer-to-peer network of Git repositories and user profiles. Fractalide [30] provides a Flow-based Programming system [31] where nodes are managed by Nix. Similarly to functional package managers, a few novel distributed platforms, such as Tunes OS [32], Awelon Blue [33] or Unison [34], separate location-based access names from linking. Awelon Blue uses cryptographic hashes as unique identi-

[1]The name is a reference to J.C.R. Licklider's "Intergalactic Computer Network", a concept that led to the creation of the Internet.

fiers for linking sequences of bytecode, while Unison assigns such identifiers to high-level language definitions. Just as IPFS, many of these projects are under development, so from their outlook, they share some common techniques with each other as well as with SFP, such as having deterministic persistent identifier assignment.

2.5 Programming Environments and Tools

Software engineering research and practices produce different lines of related work, so we overview the most prominent and relevant ones.

2.5.1 Source Code Searching and Indexing

Wide literature exists on querying source code. CodeQuest [35] and SemmleCode [36] are related to our work in that they used Datalog for querying program code. The query access intended in our SFP design is at a different granularity, but the flexibility and deterministic performance of Datalog motivated a use of its dialect. Google Kythe [37] is a project that describes itself as "pluggable, (mostly) language-agnostic ecosystem for building tools that work with code". It provides a graph storage format and schema that allows tracking cross-references between data in different languages. Presenting a uniform programming access to work with code and related artifacts is a motivation shared in SFP, but our initial focus was in single language ecosystems. We discuss cross-language extensions in Chapter 7.

2.5.2 Dynamic Functionality Resolution

Dependency injection, e.g. Google Guice [38], is finding dependencies dynamically based on the type. Loosely coupled components and services in distributed systems [39, 40] use different discovery mechanisms or middleware to establish mappings. All these techniques are done at runtime, while we focus on static resolution in SFP. We discuss the extension of using the programming interfaces at code runtime in Chapter 7. Keyword programming [41], CodeHint [42], and other IDE tools query a repository and return the correct API call. These tools resemble the interactive web interface prototype we demonstrate in Chapter 5. Apart from details in their experience, they generally work with a single code base (and selected dependencies) rather than all published community code.

2.5.3 Monolithic Codebases

Many large software companies, such as Google [43] or Facebook [44], published reports about their positive experience when using monolithic version control repositories. This is in contrast with common engineering practices with isolated project repositories. Google reported the following benefits [43] of monolithic codebases: "Unified versioning, one source of truth; Extensive code sharing and reuse; Simplified dependency management; Atomic changes; Large-scale refactoring; Collaboration across teams; Flexible team boundaries and code ownership; and Code visibility and clear tree structure providing implicit team namespacing."

Some of these reported benefits motivate SFP environments. The difference is that monolithic repositories assume a consensus on the global repository filesystem structure and semantics enforced within a single organization. Given shared code of community repositories spans across different organizations and individuals, these requirements may not suit this setting. On the other hand, SFP environments only assume a consensus on reading and writing records, as described in Chapter 4. Any filesystem structures are only metadata separate from code, so one can depend on code definitions with or without naming conventions. Each project can follow its own workflow independently of others. For instance, one workflow difference may be in handling changes, i.e. how to divide responsibility between downstream and upstream code (users and maintainers): Automated scripts for some projects may watch for changes in transitive dependencies, actively request evidence and pull in latest results; some may only watch for direct dependencies; others may update dependencies manually, but automatically request all evidence that their dependants need.

2.5.4 Computer-aided Software Engineering

The prototypes we describe in Chapter 5 can be regarded as examples of Computer-aided Software Engineering (CASE) [45]. Traditional CASE environments, even in a distributed setting [46], however, generally operate within a single organization. One related extensive research branch of automated software engineering is tool integration [47]. Processing of multiple sources of data plays a great role in improving code searching (e.g. SEXTANT [48]) where data integration is an important issue [49]. The idea of different tools needing a different access to related data was first explored in Garlan's *views* [50]. Related approaches appeared in, for instance, *virtual source files* in Stellation [51] or *virtual files* in the Desert software engineering environment [52]. The idea is to separate usage of source code from its storage. Some goals of these approaches are similar to the command-line interface prototype we describe in Chapter 5, but they focused on integrated development environments.

OzWeb [53] was an experimental early web-based environment for distributed process-centred code development with artifacts. This environment can be seen as a predecessor to many web-based environments, including the proof-of-concept web-based interaction prototype we describe in Chapter 5. OzWeb's described implementation, however, seems to require strong consistency of the system and its static data model lacks versioning.

2.5.5 Source Code in Database (SCID)

The idea of storing and manipulating code structures directly traces back to early Lisp systems. The reference manual of Interlisp-D [54] describes so-called “file packages” which were database collections of different data objects (function definitions, records, etc.). In addition to them, the system also contained Masterscope, a program analysis tool that had an interactive query interface and kept track of cross-references, variable usage etc.

Later, Intentional Programming (IP) [55] enriched some of these ideas with new concepts, implemented in a new interactive environment for structural editing and generative programming. In such systems, the unique identifier allocation is usually at random or incremental, i.e. independent of its initial definition content. The assumption was that the database server with code objects was shared. IP inspired further generative and metaprogramming systems which moved away from plain textual file storage. An example of these metaprogramming systems is JetBrains MPS [56].

Many Smalltalk systems [57] (similarly to early Lisp systems) store together application state with its source code in an image file and maintain a separate file of all source code changes. This single image-based development improves upon some issues present in conventional programming environments. The management of external code dependencies is on the coarse package level, similarly to conventional programming environments. For SFP environments, we assume the separation between application state and static source code. The extension to runtime code dependencies is discussed in Chapter 7.

Despite operating on top of textual files, Code Bubbles [58] visualizes code in a more fragmented fashion. Code Bubble’s architecture [59] combines Eclipse’s registrations and callbacks plugin mechanism with a message-based one. The message-based mechanism resembles the read and write record-oriented interface to the log we describe in Chapter 4.

Metaprogramming

Logic Metaprogramming [60, 61] stores fragmented program representations as predicates in a deductive database and uses them for flexible and declarative form of metaprogramming. Using a dialect of Prolog, these approaches can express compile-time variability of modules. Similar uses could potentially be implemented within the context of SFP environments.

2.5.6 Dead Code Elimination

Various techniques exist for removing unused code [62]. Selective loading and tree shaking are the most relevant. Selective loading aims to reduce memory usage by loading function objects on-demand into memory (the first they are used). Tree shaking, based on a programmer's specifications, removes statically unused code dependencies. One other technique is "smart linking" [63] where minimal (single unit) object files are produced and only the used ones are linked. These techniques are compile-time, link-time or load-time and results of the underlying analysis are usually not shared and reused, unlike what we assume in Chapter 6. Some of the compile-time and link-time techniques could potentially be reused for processing existing projects in a style of the command-line interface described in Chapter 5.

This page is intentionally left blank.

Chapter 3

Conventional Programming Environments

3.1 Introduction

In this chapter, we describe in detail a workflow example of a conventional programming environment from the perspective of this thesis, i.e. we focus on code and code-related metadata exchange and external code usage. We omit discussing code editors, debuggers and other tools. These tools assist development and may produce shareable metadata, but are orthogonal to the example workflow we describe in this chapter.

Since some of later chapters focus on an existing language, ClojureScript, we describe a process with examples related to this language. ClojureScript [64] is a dialect of Clojure that compiles to JavaScript, and Clojure [65] itself is a modern Lisp dialect that targets the Java Virtual Machine. Apart from the underlying platform, the main differences from Clojure are a lack of some features (e.g. agents) and a different handling of macros. Macros are defined separately from runtime code, accessed via a dedicated namespace primitive *require-macros* and evaluated at compile time. We do not include further discussion of the used language, since it is not the main focus of this chapter.

Section 3.2 starts with presenting an example project structure, as seen

Table 3.1: Example metadata access and interaction summary

Origin	**Storage Format**	**Access**
Project structure	VCS internal binary objects	filesystem commands
Version control	VCS internal binary objects	VCS commands
Project definition	Semi-structured (EDN)	text editor, parsers
Dependencies	Semi-structured (EDN / XML)	text editor, parsers, project/dependency management commands
Source content	Semi-structured (CLJ / CLJS)	text editor, parsers, REPL
Project hosting	Relational database tables	HTTP REST API, web interface
Community repository	Relational database tables	HTTP REST API, web interface

from the file system perspective. In the presence of a version control system (VCS), the actual data source of the project structure originates in its internal binary objects. The file system perspective and interactions are rather just a view over the VCS internal storage. Section 3.3 then expands this example with version control metadata which are accessible through VCS commands.

Section 3.4 looks at the project definition which contains additional metadata serialized in a semi-structured data format. Section 3.4.1 then focuses on dependency metadata which is generated from the project definition. From it, additional data is retrieved and computed through dependency resolution procedures inside a dependency management tool. Section 3.5 looks at the source code content itself which includes more fine-grained dependency information.

Finally, Sections 3.6 and 3.7 look at additional data that is connected with project hosting, continuous integration and a shared community repository. This data internally resides in a relational database, and is exposed in a semi-structured data format through HTTP REST API and in a custom web user interface. Table 3.1 shows the overall summary of example programming environment metadata presented in this chapter.

```
.
├── LICENSE
├── project.clj
├── README.md
├── src
│   └── cljs
│       └── com
│           └── example_utils.cljs
└── test
    └── cljs
        └── com
            └── example_utils_test.cljs
```

Figure 3.1: Example project structure

3.2 Project Structure

While it is possible to directly use ClojureScript's compiler, the common practice is to use automated tasks of project manager software. Various project manager and build automation tools exist. In our example, we describe one of the widespread tools, Leiningen [66]. At first, it can help us to create a project structure from a template. We show a very basic project in Figure 3.1. The purpose of each file is standard and names follow standard conventions: `LICENSE` contains a full software license text, `project.clj` is a project definition manifest (we come back to it in Section 3.4), `README.md` contains introductory documentation, `src/` is a directory tree with the source code files, `test/` is a directory tree with the testing code files.

The project structure shown in Figure 3.1 only contains visible files and directories. In practice, a project would contain additional hidden files and directories defining its environment. For example, with the Git version control system, `.gitignore` file would contain path expressions to exclude from version control (such as build caches) and `.git/` directory with the internal files. Given our example project is under version control, the structure shown in Figure 3.1 is not the "source", but rather a view drawn from the internal binary files present in `.git/objects/` directories. Figure 3.2 shows a list of object

```
39e3638e8fcde49e79eb984bc3c812c93c375966 (initial commit object)
03c2440a1b0da6512432b5812e19ba60dcd7b041 (root tree object)
42d437a17a2acb5779e523edd681935c68677fd4 (.gitignore blob object)
d921d3dffef3939870f5eaa4ce69ac9a308ac2df (LICENSE blob object)
a3ca72a647b9f4b2d5e7f7688cee6e62c152b62c (README.md blob object)
a1dc4fcee8e3bee005a95d7718784118da6f8abe (project.clj blob object)
6c86de6845f2d057d68bfdbc2a790c89b35ce8fe (src tree object)
d5b77fc4f2d94c3c4752382f23795dfe439f33c5 (src/cljs tree object)
752e997580b7805892291eac4296fc8a8904a66e (src/cljs/com tree object)
4eab900ecfb40162ec553bc7f32931949f540b83 (example_utils.cljs blob object)
1cb3689f8f468faa0d61173dac7d5aaa4c810c80 (test tree object)
2cdf1b502be861ccbd93bc971ba1e71077c3941c (test/cljs tree object)
a4ff5c80dab9f51acacaf2bef9cb799323851651 (test/cljs/com tree object)
cc5a078d331f2df8896c56aa972113a5c53c09df (example_utils_test.cljs blob object)
```

Figure 3.2: SHA-1 identifiers of Git internal objects containing the example project structure

```
tree 03c2440a1b0da6512432b5812e19ba60dcd7b041
author Josef K <josefk@example.com> 1486219272 +0800
committer Josef K <josefk@example.com> 1486219272 +0800

initial commit
```

Figure 3.3: Version control metadata example for an initial commit

identifiers that would correspond to the shown example project structure. The first object is a "commit object" which contains version control metadata (author, date, and message) and hash references to a "tree object" and a previous commit object (in this case, none). The content of the first object is shown in Figure 3.3. In this scenario, tree objects contain hash references to other tree objects and blob objects, i.e. they encode the directory structure and more or less correspond to inodes. Blob objects then in this scenario contain the content of files.

The origin of project content is thus in "nameless" binary objects which are shared and exchanged. The host filesystem and Git version control commands only provide a user interface to them.

3.3 Version Control

If we change files and commit them, the number of Git internal objects will grow. Figure 3.4 shows a list of object identifiers that would correspond to the example project after changing some of its source files. The second commit object contains commit metadata, a reference to the first commit object and a

```
e685f8e976cced2a91c8e9537c389204ee09a5bc (second commit object)
39e3638e8fcde49e79eb984bc3c812c93c375966 (first commit object)
800f5e574d4c01998a9eed15b2170e4426d2dd21 (new root tree object)
... (new tree objects)
2b90d2971d87bbff6767255feee5c5a0b804fc63 (new example_utils.cljs blob object)
03c2440a1b0da6512432b5812e19ba60dcd7b041 (old root tree object)
... (old tree and blob objects)
```

Figure 3.4: SHA-1 identifiers of Git internal objects containing the example project structure and changes

reference to the new root tree object. The new root tree object then refers to all new tree and blob objects, such as the changed source code files, after the change. The objects corresponding to the state before the change, such as blob objects for the original source files, are still present in the store. The version control system may also compress the objects in its storage. Nonetheless, all repository content and version control metadata are retrievable using their hash identifiers.

Just as with the project content, version control metadata originates in this data source. Each metadata field references internal objects directly through their unique identifiers.

3.4 Project Definition

In this section, we look at some of additional metadata defined within the content of the `project.clj` file and details of the dependency resolution procedure. We show the corresponding project definition in Code Listing 3.1.

The project definition is a Clojure code that utilizes Leiningen's defproject macro. This macro takes the provided arguments and creates a **def** special form. The **def** form can create a global variable (in a namespace with a given symbol) bound to a value. In this case, the value is a nested map data structure. This data structure then contains both the provided and extra information, such as the full compile path or default repositories.

The first two arguments follow the Maven's namespace convention: `com.example` is the so-called "group-id", `example-utils` is "artifact-id", and `0.1.0` is a specific version string. The following arguments then contain project-wide metadata which may also be duplicated in an unstructured form in the repository (e.g. the license information). Dependencies are specified in a vector that lists dependencies in the mentioned Maven convention. Given differences between compiler versions and their standard libraries, even a simple example project specifies two external dependencies. Finally, the standard compilation with Leiningen for ClojureScript requires a plugin lein-cljsbuild and its configuration. We come back to this in Section 3.6.

Listing 3.1: Example project definition

```
(defproject com.example/example-utils "0.1.0"
  :description "Example library creation"
  :url "http://lpaste.net/352020"
  :license {:name "Eclipse Public License"
      :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.8.0"]
                 [org.clojure/clojurescript "1.9.456"]]
  :plugins [[lein-cljsbuild "1.1.5"]]
  :cljsbuild {
     :test-commands {
         "unit"
         ["phantomjs"
          "resources/private/js/unit-test.js"]}
     :builds {
        :test
        {:source-paths ["src/cljs" "test/cljs"]
         :compiler {
           :output-to "resources/private/js/unit-test.js"
           :optimizations :whitespace
           }}
     }}
```

```
~/.m2/repository/
├── args4j
...
├── cljsbuild
│   └── cljsbuild
│       └── 1.1.5
│           └── cljsbuild-1.1.5.jar
...
├── lein-cljsbuild
│   └── lein-cljsbuild
│       └── 1.1.5
│           └── lein-cljsbuild-1.1.5.jar
└── org
    ├── apache
    ...
    ├── clojure
    │   ├── clojure
    │   ...
    │   │   └── 1.8.0
    │   │       └── clojure-1.8.0.jar
    │   ├── clojurescript
    │   ...
    │   │   └── 1.9.456
    │   │       └── clojurescript-1.9.456.jar
    ...
```

Figure 3.5: Fetched dependencies

)

3.4.1 Dependency Resolution

When we run the "lein deps" build task (which may be implicitly invoked before other tasks), the task will resolve dependencies and fetch corresponding artifacts into a local Maven repository. Figure 3.5 shows some of the dependencies fetched for the example project. Given their large number, we omitted showing all artifacts, e.g. all transitive dependencies of the lein-cljsbuild plugin and ClojureScript compiler, in Figure 3.5.

By default, the dependencies are resolved against two repositories: Maven Central and Clojars (the Clojure community-maintained repository). The dependency resolution procedure follows the standard one from Maven. This procedure is, however, fairly complex [67] and may be augmented by various plugins, so we will only describe its high-level default mechanism.

The tool first fetches additional metadata about each dependency, including their transitive dependencies. These kinds of metadata are usually retrieved via the HTTP(S) protocol in the form of `pom.xml` files. From these additional kinds of metadata, it tries to construct a full dependency tree and fetch corresponding artifacts (e.g. JAR files). When multiple versions of an artifact are encountered in a dependency tree, the "dependency mediation" procedure decides which one is chosen to prevent potential name conflicts between code (e.g. Java classes) contained in these artifacts. The mediation procedure first uses heuristics about the artifact location in the dependency tree. For example, if libA uses libB-2.0 and libC-1.0, and libC-1.0 uses libB-2.5, it will pick libB-2.0, because it is one depth level closer to libA than libB-2.5. If they were on the same level, the procedure would use the order they were defined in. If any of the conflicting versions were specified as fixed or using ranges, the procedure will run a solver that would either find a version that satisfies all constraints or fail the procedure. In our example, if libC-1.0 specified libB-2.5 as a fixed concrete version (denoted by square brackets), the procedure would pick libB-2.5 instead of libB-2.0. If libC-1.0 specified a range and libB-2.0 fitted in that range, the procedure would pick libB-2.0, otherwise it would choose the biggest version in the range.

Lastly, we have more options: versions can be specified with qualifiers; dependencies can be defined with different scopes and can be imported. All these additional options then may play a role in ordering and heuristics used by the dependency mediation procedure.

In this section, we have seen that, unlike project content and version metadata, additional project-related metadata is present in different semi-structured formats in one repository file and in files retrieved from remote repositories. Accessing these kinds of metadata involves using different user or application programming interfaces.

3.5 Source Content

The source files themselves may contain additional metadata. Code Listing 3.2 shows an example of the `example_utils.cljs` file. The **`ns`** macro may contain references that would call respective namespace manipulating functions. In the example listing, the local namespace will contain the `join` name referring to the respective function defined in `clojure.string` (which is loaded with the standard library, but not imported by default). Apart from the local names, the function `right-pad` includes a documentation string. This string, unlike in other programming languages, is not entirely discarded, but can be accessed with other annotations at runtime using the **`meta`** function.

Listing 3.2: Example source code

```
(ns com.example-utils
  (:require [clojure.string :refer [join]]))


(defn right-pad
"If S is shorter than LEN, pad it with CH on the right."
  ([s len] (right-pad s len " "))
  ([s len ch]
  (join (first (partition len len (repeat ch) s)))))
```

In this section, we have seen that other kinds of metadata are present directly in source code files. Code-embedded metadata can be then seen as a semi-structured format and need another interface for their access.

3.6 Project Hosting

The local version control repository would likely be synchronized with a remote version control repository to facilitate collaboration. The remote repository can be using dedicate software or service that provides additional capabilities on top of the plain version control. In our example, we will use the GitLab repository manager [68]. By default, GitLab stores additional metadata about projects (e.g. related issue reports) in a relational database. Typically, we do not have a direct access to this database, so we interact with these kinds of metadata in GitLab's web interface or via REST API. Figure 3.6 shows a shortened output in JSON format from the api/v3/projects/:id REST resource on GitLab 8.16. The resulting nested structure contains various metadata about the project's configuration and overall statistics. For example, the "stars" field refers to the number of users that bookmarked a given project.

3.6.1 Continuous Integration

Projects often also set up a continuous integration service. This service tries to build a given project or run test suites in an isolated environment (usually a virtual machine). For ClojureScript, since the target language is JavaScript, one possible environment is to execute a compiled test suite code in PhantomJS which is a headless web browser (i.e. configured to be used on systems without a monitor and input devices). This is what the project definition's cljsbuild configuration in Code Listing 3.1 specified. In addition to that, Code Listing A.1 shows an example source test file with one unit test and helper code for test execution. GitLab includes its own continuous integration service: Code Listing A.2 then shows an example configuration for ClojureScript test suite execution with PhantomJS. We omit showing this code directly here, because it mostly contains environment-specific configuration details rather than new metadata.

A common configuration is to run build scripts per each commit on one of

```
{"id": 2,
 "description": "Example of creating a CLJS library",
 "default_branch": "master",
 "tag_list": [],
 "public": true,
 "archived": false,
 "visibility_level": 20,
 "ssh_url_to_repo": ...
 ...
 "owner": {...},
 "name": "com.example-utils",
 ...
 "container_registry_enabled": true,
 "issues_enabled": true,
 "merge_requests_enabled": true,
 "wiki_enabled": true,
 "builds_enabled": true,
 "snippets_enabled": false,
 "created_at": ...
 ...
 "namespace": {...},
 "avatar_url": null,
 "star_count": 1,
 "forks_count": 0,
 "open_issues_count": 0,
 ...
 "permissions": {...}}
```

Figure 3.6: GitLab project metadata example

```
[{"id": 8,
  "status": "success",
  "stage": "test",
  "name": "test",
  "ref": "master",
  "tag": false,
  "coverage": null,
  "created_at": ...
  "user": {...},
  "commit": {
    "id": "3c92583519e675e4f95f79b5e8a7489c5ab76a66",
    "short_id": "3c925835",
    "title": "fixed phantomjs",
    "author_name": ...
    ...},
  "runner": {...},
  "pipeline": {...}},
  ...]
```

Figure 3.7: GitLab CI metadata example

the mainline branches of the version control repository. Monitoring of their status can be retrieved through the api/v3/projects/:id/builds REST resource. Figure 3.7 shows a shortened example output from this resource. It contains a variety of metadata about each CI build.

In this section, we have seen examples of many kinds of metadata that project hosting services may provide on top of the plain version control. Accessing these kinds of metadata again involves formats and interfaces different from the ones shown in other sections.

3.7 Deployment

We have different options for sharing bundled artifacts of a project. Different programming language communities [69] have different policies and practices when it comes to publishing libraries on community repositories. Some communities may require an approval procedure from repository maintainers or external tool checking (e.g. that the published library does not break any

```
{:jar_name "example-utils",
 :group_name "com.example",
 :homepage "http://lpaste.net/352020",
 :description "Example library creation",
 :user "josefk",
 :latest_version "0.1.0",
 :latest_release "0.1.0",
 :recent_versions ({:version "0.1.0", :downloads 2}),
 :downloads 2}
```

Figure 3.8: Clojars metadata example

client code), other communities may not impose many restrictions on what library authors can publish. In ClojureScript, the commonly used repositories are the default ones from tools: Maven Central and Clojars. Both of them require a valid `pom.xml` file and a PGP signature of a given library's author. Maven Central imposes additional constraints (packages should only be released versions and depend on other packages on Maven Central).

For Clojars, given one has created an account, Leiningen has a pre-configured task that will generate JAR and `pom.xml` files, sign and upload them. Apart from them, Clojars store additional metadata about each library. They are internally stored in a relational database, but exposed in Clojars' web interface or via REST API. Figure 3.8 shows an example of such metadata in Clojure's EDN format.

We have seen that community repositories contain and expose metadata that were present elsewhere (such as the names and version numbers) as well as its own information, such as the number of downloads of a particular library.

3.8 Example Client Code and Other Metadata

If we were to write client code using the library that we showed how it can be shared on a community repository in Section 3.7, we would follow a similar work flow. We need to include the example library in the dependencies section of a project definition, as shown in Code Listing 3.3. An example client code that could then use the example `right-pad` library function is shown in Code Listing 3.4.

Listing 3.3: Example client project definition

```
(defproject com.example/example-client "0.1.0"
  :description "Example library creation"
  :url "http://lpaste.net/352020"
:license {:name "Eclipse Public License"
     :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.8.0"]
                 [org.clojure/clojurescript "1.9.456"]
                 [com.example/example-utils "0.1.0"]]
  ...
)
```

Listing 3.4: Example client code

```
(ns com.example-client
  (:require [com.example-utils :refer [right-pad]]))


(defn show-progress
  "Assuming loaded is [0,100], show either \"Loading\"
  right padded with dots or \"Complete\""
  ([loaded] (if (= loaded 100)
             "Complete"
             (right-pad "Loading"
               (+ 8 (mod loaded 3)) \.))))
```

The source code content of libraries in shared community repositories may be processed and presented in other ways. For example, a documentation may be extracted from documentation strings in source code and enhanced with additional metadata. In the Clojure ecosystem, ClojureDocs is a popular community repository of documentation where extracted source code documentation may be enhanced by example usage submitted by external users. These additional kinds of metadata are then stored in a document-oriented database (MongoDB) and exposed via a custom web interface or possibly API endpoints.

3.9 Summary

In this chapter, we have gone through various stages associated with the shared code development and described examples of metadata encountered at each stage. Their data source origins were heterogeneous: some metadata reside in a custom binary format of version control internals, some are present in different serialized semi-structured formats locally or remotely, or in remote relational databases exposed via web user interfaces and API.

The disparity between various sources of metadata may result in issues we described in Section 1.1.2. For example, the default dependency resolution procedure we outlined in Section 3.4.1 only concerns naming metadata retrieved from `pom.xml` files. We could still encounter name conflicts later on if the procedure includes two artifacts with distinct group and artifact identifiers, but with their internal files exporting conflicting namespace identifiers.

Individual data sources are accessible often in a straightforward way. If we want to retrieve a project structure and content at a given time point, we can lookup a relevant commit object identifier and use the "git checkout" command. If we want to find out if code at a given commit compiles without any problems, we can retrieve this information from the GitLab CI's interface. If we want to visualize a dependency tree computed by the resolution procedure, we can run the "mvn dependency:tree" command.

In default repositories of ClojureScript, changing existing artifacts is not possible, but other language communities may allow it. Other than that, project changes that cannot be easily signalled through version numbers (such as splitting or merging projects) may cause problems.

Further challenges arise if we require some cross-data source information. For example, our project manifest can specify that we depend on a certain library of a certain version point or a range. We cannot, however, instruct the underlying dependency management tools to fetch only portions (i.e. individual definitions) of transitive dependencies our client code actually depends on. This is one example where combining metadata in source code and a project manifest becomes a challenge.

Similarly, we can retrieve a continuous integration build status information for any library that is hosted publicly. We cannot, however, instruct the dependency management tool to verify that every artifact in our dependency tree passed its associated unit tests or even external tests from other projects. In this example, combing metadata from project manifests and continuous integration services becomes a challenge.

These kinds of scenarios motivated design decisions behind Search-focused Programming. In Chapter 4, we go into details of these design decision and how they address challenges we described in Section 1.1.2 and illustrated with detailed examples in this chapter.

Chapter 4

Search-focused Programming Environments

This chapter describes requirements and design decisions for Search-focused Programming (SFP) environments in terms of code and metadata storage, distribution, and organization. Presented design decisions are judged with regards to properties induced by posed requirements. The discussed issues and principles of SFP environments span from code distribution and assigning global identifiers to code organization, access, and interaction with metadata. Their consideration also includes factors introduced by incorporating existing programming language features.

4.1 Requirements

Before we propose various concepts that build up SFP environments, we outline properties of our key interest that are going to be addressed. The purpose is to specify criteria SFP environments must conform to. These criteria provide a guidance for different alternatives as well as for future extensions.

4.1.1 Determinism

Conventional programming environments may imply context dependence (e.g. local environment state at a certain time) and certain procedures in them can become non-deterministic. For example, the complex dependency resolution procedure relies on the fact everyone follows similar conventions for naming and version incrementation. Some data sources are mutable and may change existing information. So, while at some point in time we can retrieve certain metadata, we may not have access to them in the future. We would be interested in providing deterministic abstractions, i.e. procedures that can be repeated and replicated with the same outcome.

4.1.2 Global Disambiguation

In conventional programming environments, code and metadata may be tied to their physical location. Since there is no equivalent of Uniform Resource Name (URN) for statically linked code definitions, we do not have an ability to associate code with metadata, such that this association is globally unambiguous. We would be interested in global disambiguation of code definitions without enforcing central naming authorities.

We outlined several issues in conventional programming environments due to layers of naming abstractions in Chapter 1. The requirement for global disambiguation would prevent some of these issues "by design".

4.1.3 Flexibility, Extensibility and Expressiveness

A variety of entities in code, metadata and processes exist in conventional programming environments. In order to capture them, the architecture should be flexible enough. It should also allow extensions to accommodate new data and services.

Individual data sources implied a limited range of metadata we had access to. We would be interested in a more expressive language that can be used for accessing different kinds of metadata.

4.1.4 Absence of Inherent Bottlenecks

Since we are interested in accommodating all code and metadata and extending to their new forms, we would be interested in assessing the potential for scalability, i.e. whether there are no inherent bottlenecks in the design. For the purpose of discussing different design aspects, we point out any choices that may prevent horizontal or vertical scaling. One example class of inherent bottlenecks are architectural styles where centralized nodes may be flooded.

4.2 Design of Code Distribution

In this section, we briefly remind ourselves how code distribution works in conventional programming environments (see Chapter 3 for more details). We then overview the concept of global persistent identifiers. We follow it by ideas from persistent structures and content-addressable systems [2], and show how they can provide a way for global persistent identifiers for code. Finally, we present different issues and their solutions when global persistent identifiers are faced with existing programming language features.

4.2.1 Existing Code Distribution

We dedicated the whole Chapter 3 to conventional programming environments, so we only summarize main points. If we want to access external code, we do so by a mutable location-dependent reference to a desired definition in a file bundle with other definitions (both needed and unneeded by client code). This way could lead to name or path conflicts; if they are on the programming language level, they could be limited through scoping and renaming mechanisms of namespace semantics. Other than that, *unneeded code may fail to compile or increase processing time.* Finally, unless it is enforced by repositories, the location we used to reference external code may become invalid. Overall, there is no guarantee of persistent identifiers, and code distribution is a complex interplay of solutions to different technical problems as well as social conventions of different programming language communities.

4.2.2 Persistent Identifiers

The standards for reliable and efficient finding sources of documents through long-lasting references have developed over centuries. With the advent of the World Wide Web, this issue has been addressed in the online context as well. A common approach for creating persistent identifiers relies on systems administered by institutions. For example, the popular Digital Object Identifiers (DOI) [7] are assigned by a federation of registration agencies that were appointed by the International DOI Foundation.

Using standards developed in the context of library information systems would be possible for code, but may be impractical. Unlike scholarly materials and datasets, program code and metadata are generally self-published and managed in ownership and merit-based hierarchies. This practice is widespread and brings its advantages, and it would not be feasible or rational to argue against it. It also remains a question whether the systems meant for document identification would scale up to the level of statically linked code definitions. In summary, traditional persistent identifier approaches would not satisfy some of the requirements we described in Section 4.1.

Computer systems may use one of standardized versions of the universally unique identifier (UUID) [70] scheme for assigning long-lasting references to objects without relying on third party authorities. They, however, do not satisfy the requirements we set out in Section 4.1:

- Versions 1 and 2 use time, so do not satisfy our requirement for determinism.

- Versions 3 and 5 use namespaces, so potentially do not satisfy our requirement for global disambiguation that is independent of naming hierarchies.
- Version 4 is randomized, so does not satisfy our requirement for determinism.

4.2.3 Global Identifiers for Code

In Chapter 3, among other things, we discussed how persistent content-addressable storage works in internals of version control systems. This mechanism guarantees globally unique identifiers with an extremely small probability of collisions (given good hash functions). These identifiers are generated in a deterministic way and their creation and assignment do not require third party naming authorities. In that sense, they would satisfy the requirements of Section 4.1.

We can thus imagine a scheme where program entities (constants, functions, types, etc.) that are declared with a global scope gain unique content-dependent identifiers instead of user-chosen names (i.e. symbols). That is definitions of the form `(def X body)` bind '`body`' to an identifier H(body's external dependencies + body) where H is some hash function. It may seem that we lose human-readable names, but the idea is to store human-readable names for global program entities separately and use them (along with other metadata) for organizing and interacting with program entities. We expand on this side in Section 4.3.

Name collisions happen with user-chosen non-unique identifiers. The global identifier assignment using a content-addressable scheme may have an identifier collision, but assuming hash functions used in practice, even with billions of objects, the probability of collisions is minuscule [71]. With non-cryptographic or weak algorithms, one may possibly construct a malicious input to find a collision. Since it is outside of the scope of our posed requirements, we discuss one potential counter-measure in Section 4.2.8.

4.2.4 Deterministic Assignment of Identifiers

We briefly introduced a content-addressable mechanism for creating identifiers for globally-scoped program entities. We can now discuss several practical issues with regards to this mechanism. The first question is: what exactly should we use as the input to H? For the code entity body's external dependencies, it could be a sorted list or a set of identifiers. For the '`body`' itself, we have different options and each option has its advantages and disadvantages:

- *Surface Syntax*: Using the surface syntax is a straightforward solution. The main drawback is that this option is sensitive to small refactoring or character changes (e.g. removing whitespaces) – one counter-measure is to process the source text with the same auto-formatting style.

- *Surface Abstract Syntax Tree (AST)*: This approach counters the problems of using textual source and is insensitive to character changes. It still may be sensitive to small refactoring changes (e.g. renaming of internal symbols) and parsers of some languages may be non-deterministic (e.g. they would execute certain macros or assign random symbols to unnamed parameters). Since determinism was one of our requirements, this approach may require creating a custom parser or taking steps to restore determinism, i.e. modifying the existing language's parser or processing the parser's output.

- *Intermediate Representations or Target Bytecode*: Using "desugared" core abstract syntax tree or the target bytecode would be more resilient to small internal refactoring changes. The previous two options are linked with the original source – so, assuming they are stored and can be looked up using the assigned identifier, we can view and get an editable representation easily. In this option, it is not the case, so we would either need to store the original sources (there could be many source definitions that are assigned with the same identifier) and let the user choose the preferred one, or employ some "resugaring" techniques [72] if it is even possible. In addition to that, as with the surface AST, the output may be non-deterministic (e.g. bytecode generated with profile-guided optimizations), so we would need to take steps to restore determinism.

In summary, each option has its own advantages and disadvantages. The selection depends on specific goals and conditions one may have in a given language environment. In our case, we could choose to use the source syntax. It may be a potential bottleneck for scalability, but with the increasing physical storage sizes, we do not consider it to be a serious threat.

4.2.5 Name-dependent Programming Language Features

While content-addressable assignment of identifiers works for constants and functions, it becomes a question how to deal with other entities. In ClojureScript, one example is a variable bound to a mutable atom:

```
(def ^:dynamic example-ref (atom 0))
```

In this case, the definition's body is not enough for generating a globally unique identifier. In the standard language, this feature is tied to location-based disambiguation. If we assign identifiers based only on the body content, "distinct" definitions would be assigned the same identifier. One other example is a protocol definition:

```
(defprotocol ReadWrite
  (read [this])
  (write [this message]))
```

This feature is a more dynamic equivalent of interfaces. In languages with expressive type systems, it may be possible to disambiguate such definitions based on their type signatures. Here, we only have names and arities. It is, thus, possible to assign ambiguous identifiers. Multiple distinct protocols could denote different semantics, but would be assigned the same identifier if we assigned them identifiers only based on their body contents. For example, ReadWriteAsync and ReadWriteSync could be protocol definitions that contain the same body as the above shown ReadWrite, but expect different semantics of their methods (which, in a dynamic language, could be specified externally in documentation, tests or pluggable type systems).

Extended Input-based Solution

One approach in design would be to forbid the use of name-dependent language features. This approach would not satisfy our requirement for flexibility. It seems that for these features, the requirements for flexibility and global disambiguation are in opposition. That is partially true. The goal is to have an identifier assignment scheme where we could globally disambiguate between code fragments that have the same content (or even the same name), but should be used in different contexts. For global variables, each identifier corresponds to a memory location. For protocols, each identifier corresponds to a protocol's structure as well as its author's assumptions (given the lack of expressive type annotations). Different contexts that we need to disambiguate can be different global namespaces as well as same namespaces, but different versions or forks.

Our solution for this problem is to assign identifiers using an extended scheme: $H(\text{full qualified name} + \text{timestamp} + \text{author's public key} + \text{signature} + \text{body's external dependencies} + \text{body})$.

Given that we require globally unique names without a centralized name-assigning authority, we need additional input other than the full qualified name:

- *Timestamp*: Name changes only rely on conventions, so it is possible that an author may want to use the same definition at the same location to denote different semantics, for example with different versions. Since in these cases, we need to disambiguate between these definitions, including a timestamp ensures that they are assigned distinct identifiers.

- *Author's public key and signature*: Without a centralized authority that oversees and enforces globally unique names, different individuals may use the same definition at the same location to denote different semantics, for example by a coincidence or in case of forks. We need to disambiguate between definitions in these cases. Since public keys for digital signature schemes can be generated locally and shared through different means, we do not sacrifice global disambiguation. Including an author's public key and signature ensures same definitions with colliding locations from different authors are assigned distinct identifiers.

Overall, this extension still preserves determinism. Given the input information, the same identifier for a given code definition is generated everywhere. One alternative for *Extended Input-based* schemes would be to include a "canonical specification" in additional inputs. These canonical specifications could express *essential* assumptions and metainformation about given code entities. They could range from naming and authorship information to pluggable type system annotations. While this approach is more flexible, guaranteeing determinism and the absence of bottlenecks may become a major challenge.

4.2.6 Hierarchical Identifiers

If we look beyond code identifiers, we can assign persistent unique identifiers to other entities as well. This is different from conventional programming environments where identifiers are non-unique, mutable and may be unrelated to their structural content. The way persistent unique identifiers can be assigned in SFP is in a hierarchical fashion. For example, given that we have code entities with identifiers id1, id2, id3 etc., we can assign an identifier to a file that contains them as $H(id1, id2, id3, ...)$, and possibly include other inputs, similarly to name-dependent language features. The versioned project unique identifiers would then be assigned from these unique file identifiers.

In conventional programming environments, the version control system assigns persistent identifiers to files, but only based on their textual content. The assigned identifiers, thus, do not reflect any structural information, transitive dependencies etc. So, even though we have a deterministic procedure to retrieve files (given VCS's assigned identifiers), we cannot disambiguate them globally from files with the same textual content in different repositories or project versions. For that, we would need additional inputs (physical repository location, reference to a particular repository state, etc.) and processing (potentially non-deterministic retrieval of project-level dependencies). Given that, these persistent identifiers in conventional programming environments do not satisfy our requirement for global disambiguation.

This is assuming that the VCS identifier assignment works, as described in Chapter 3. We could possibly utilize VCS as a storage layer if we created binary VCS objects for individual code fragments and embedded the input for identifier assignment in their content. This idea is, however, related to implementation choices rather than design and principles we intended to discuss in this chapter.

4.2.7 Mutual Recursion

One issue that is worth discussing is mutual recursion, i.e. how we assign identifiers to program entities that are mutually dependent. Similarly to how mutual recursion is handled in programming language semantics, we have several options:

- Group the mutually recursive program entities into a single entity, assign an identifier to that entity, and then create and assign identifiers to projections of that grouped entity. For example, if we have several mutually recursive functions, we can create, using a local let recursive binding, a tuple where its components are the mutually recursive functions. We would then generate an identifier for this tuple, and create and assign an identifier for each component projection.

- Extract the recursive references into parameters and instantiate them at the call site.
- Create forward name references using the extended input-based identifier assignment (with timestamps and authors' public keys and signatures) and use that to bootstrap the recursive definitions.

These options are in a way equivalent, so the usage depends on individual scenarios and language features that are required for their implementation.

4.2.8 Extensibility and Implementation Choices

Finally, commitment to a single hash function for assigning identifiers could limit future extensibility, as new hash functions with better properties may be invented in the future. A potential solution to this limitation could be the use of multihash [73] for identifiers. Multihash is a recent informal proposal to use the first several bytes for encoding what hash function and size the remaining digest represents. Identifiers would as such self-define what hash function was used for their generation. By doing so, multiple hash functions could coexist for the identifier generation and assignment. The only consequence would be that program entities could potentially have multiple globally unique persistent identifiers.

We describe the prototype implementations in Chapter 5. For identifier assignment, the prototypes used the Murmur3 hash algorithm and the surface syntax as the input. These choices were made in the context of a proof of concept prototype. Beyond the prototype, the implementation would need to take into consideration other requirements. For example, if we required a code similarity detection, the assigned identifiers would need to reflect that, as discussed in Section 7.2.3.

4.3 Design of Code Organization, Access and Interaction with Metadata

In Section 4.2, we focused on the issue of assigning persistent identifiers to program entities. We discussed a deterministic scheme that can assign persistent globally unique identifiers to program entities. Its main advantage is that this scheme is location-independent and does not require centralized naming authorities. The drawback is that generated identifiers do not convey any human-readable information, unlike centralized location-based schemes. Centralized location-based schemes tie source code hierarchical organization with human-readable names in identifier assignments.

In this section, we look at how human-readable names or other metadata associated to program entities can be stored, retrieved and searched in a uniform manner. In conventional programming environments, since there are no globally unique persistent identifiers, each data source creates its internal identifiers for managing metadata and exposes them through its custom interface. The approach in conventional programming environments thus limits flexibility and expressiveness requirements we have for interactions with metadata.

Many issues arise with data organization in a distributed environment, so we do not aim to extensively cover every possible issue. The focus of this section is on basic concepts and interfaces that underline proof of concept prototypes we describe in Chapter 5. We point out some limitations and expand on them in Chapter 7.

4.3.1 Organization

We first explain the terminology we use in discussing code and metadata organization and then discuss two categories of metainformation present in programming environments.

Here are common terms we use in our discussion:

- Entities refer to "objects" which were assigned persistent globally unique identifiers. Apart from program entities we described in Section 4.2.3, entities can represent projects or other objects in programming environments.

- Constants refer to "values" which do not have globally unique identifiers, such as strings, booleans or numbers. Examples are names assigned to program entities, license names and texts, return values or standard and error outputs of test suit executions.

- Relations or attributes express a mapping between subjects (entities) and objects (other entities or constants). For consistency, we denote predicate symbols using Clojure's named dictionaries: for example, {`:code/uid id1`, `:code/dependsOn id2`} denotes that code with id1 depends on code with id2. In the predicate logic notation, this fact could be expressed as $dependsOn\ (id1, id2)$. The relations imply names originating from a global ontology or a schema – our used relation names are only examples and other relation names would be equally good. We do not aim to discuss issues, such as ontology or schema mismatches – we assume that relations used in the context of programming environments would rarely change and their total number is small and mostly fixed. In contrast, we assume names assigned to program entities may change frequently and their total number is large and may grow.

- Some data can be stored and retrieved directly ("extensional predicates" in the deductive database terminology) and some data could be computed using other data ("intensional predicates"). One example is that using the direct code dependency data, we can compute the transitive dependencies. We omit the further discussion here and expand on this issue when discussing Code and Metadata Access and Interaction.

- We mention "temporal" facts. By that, we mean that each relation has an implicit time component that tracks provenance, i.e. maps a given fact to the transaction that introduced it. We discuss this concept further in relation to Storage, Access and Interaction.

In summary, we operate with entities and constants: entities are objects with persistent globally unique identifiers (e.g. program entities as we defined in Section 4.2.3), constants are values of different types (e.g. strings or numbers). Relations correspond to facts and associate entities to other entities or constants. We distinguish between two categories of facts:

- *Temporal facts*, also referred to as "tags": these facts may change over time, i.e. be retracted after they are added. They correspond to metadata that varies over time in programming environments: names, project structures, etc.

- *Persistent facts*, also referred to as "evidence": these facts do not vary over time, i.e. once they are added, they are assumed at any time afterwards. They correspond to code, static dependencies, continuous integration results, etc.

Modelling of persistence and mutability is a separate issue and we omit a full discussion here. One possibility is to assume a time component in each fact and express persistent or mutation through rules – this approach is used, for example, in Deadalus [74].

Deciding what facts should be persistent depends on the perceived reproducibility or irreversibility of a particular process. When code is published under a copyleft license, it can be freely distributed and persistence does not pose a problem. Similarly if there was an irreversible approval process, e.g. as a part of a code review, the “approval” can be a persistent fact. With continuous integration results and running test suites, we assume the process is reproducible. That means it produces a consistent result under “normal conditions” (e.g. no hardware failures) at any time. Any such highly deterministic processes, for example code analysers or style checkers, can represent its results as persistent facts.

The example with test suite results depends on a particular fact we want to express. If we want to express that a given function passed a particular test suite (which is a statically known code), we take it as a persistent fact. On the other hand, if we want to express code passed *all its approved associated tests* at a given time, it is a temporal fact, because a new test may be added later and fail. We discuss how one updates information in Section 4.3.3.

Lastly, some processes are non-deterministic in nature – for example, the performance measurements. In such case, it depends on the trust in reproducibility. If it is guaranteed that code profiling or performance measurements are always done under the same stable setting and facts would represent rigorous statistical estimates, they can represent persistent facts.

We explained basic terminology and concepts for code and metadata organization. Using temporal relations is in line with our requirements for flexibility, extensibility and expressiveness. We later discuss potential scalability issues in relation to Storage and Access.

4.3.2 Storage

We discuss the issue of storing code and metadata and present log-based uniform storage. In conventional environments, as we demonstrated in Chapter 3, various sources of code-related metadata exist with a different access to them. That does not satisfy the global disambiguation requirement we set out for this chapter. Challenges arise in various situations. Due to a lack of globally unique persistent identifiers, detecting the related code and metadata across different sources becomes challenging. Similar situations may happen with any source changes or topological changes – for example, if a project grows and is split into sub-projects which are copied to newly created repositories, sub-projects may lose parts of their metadata, e.g. continuous integration results, that were attached to the old repository. In general, synchronizing and managing multiple heterogeneous sources in a distributed environment add additional complexity and may prevent global disambiguation.

This problem setting appears in various forms in other domains. In distributed systems, independent computer nodes work towards a common goal, so the common challenge is designing a protocol where messages are received reliably and in the same order by all network participants. These protocols are referred to as atomic broadcasts [75] and are present as primitives in different distributed system implementations. In software architecture patterns, event reading and committing can be used for tracking internal states of entities. This pattern is known as Event Sourcing [76].

Overall, all of these concepts revolve around providing a uniform reliable source of all data that can be thought as a data storage abstraction: append-only totally-ordered sequence of records. This is akin to logs in various systems, be it journalling in file systems, commit logs in version control or transaction logs in databases. Log implementations differ in details – for example, one difference is whether we store state transforming operations or their results (a part of the state after applying an operation) in the log. For the purposes of our discussion, we assume these details are more or less equivalent with regards to our requirements.

Since the log-centric storage is append-only, it mitigates issues related to incompatible changes. Similarly to version control system, updates or deletions do not remove the original objects, only mark them as not being a part of the latest project or other hierarchical metadata structure. We could, however, always come back and retrieve them. Depending on particular metadata changes, different merge semantics may be needed. This is, however, beyond the scope of our discussion and we assume the "last writer wins" strategy as a default one.

From our perspective, the log-centric storage satisfies the requirements we previously laid out in Section 4.1:

- *Determinism*: we talk more about the actual interfaces in Section 4.3.3, but in general, the log-centric storage provides the history of all changes and a deterministic way to obtain a state (i.e. an aggregated representation of requested data) at any point – be it through replaying operations in the log, or through references to snapshots.

- *Flexibility and Extensibility*: since the log becomes one reliable uniform source of all data, we do not need to worry about issues we face in conventional programming environments (e.g. explicit synchronization and fault tolerance of each source of data). This fact gives us flexibility to capture and store any data related to a programming environment. Similarly, the log can store new or extended forms of metadata. Extending functionality or services is a matter of communicating schema changes and processing these new or extended forms of metadata.

- *Absence of Inherent Bottlenecks*: Given various large-scale log-centric system implementations exist [77, 78] and allow real-time processing with multiple readers and writers, we assume that the log-centric storage does not pose any inherent scalability issues.

One potential issue for global disambiguation could be assuming a centralized infrastructure in the same way "default repositories" exist in conventional programming environments. Persistent facts would work without any problems even if they were inserted at different points at two distinct log-centric infrastructures. On the other hand, temporal facts may pose a challenge if we tried to combine conflicting information from different logs or the same log with itself at different points in time. We assume these situations are not frequent and would be resolved at the client side. Alternative approaches with different trade-offs would be to rely on a single infrastructure with a decentralized consensus on the log content or enabling "rule delegation" on peers that process the log and provide its materialized view at given time points. We expand on these alternative approaches as well as other requirements (such as security) in Chapter 7.

Finally, one potential issue for scalability is the maintenance of a complete log. Even if we assumed an infinite storage, so that we do not run out of storage, traversing the log would take longer and longer. The solution to this issue depends on the log implementation details. If it is a Git-like sequence of snapshots, we compress, archive or filter out data that may not be frequently used or the most recent snapshots do not depend on. If it is a sequence of change operations, the common strategy is "log compaction". The goal there is to remove obsolete records and only store the most recent ones. By doing so, we lose the ability to replay all states, but still have the full "backup" and can replay the most recent states.

We introduced the log-centric storage, explained how it satisfies our requirements for determinism, flexibility and extensibility, and scalability. We also discussed two potential issues and their solution.

4.3.3 Access and Interaction

We previously discussed the log-centric storage. Next, we look at ways how we can access and interact with code and metadata in the log.

In conventional programming environments, access and interaction differ and depend on particular application programming interfaces to given data sources. In general, we observe two types of access and interaction:

- *Read-only view*: in this type, we are interested in retrieving code and metadata as of a given time point. For example, we retrieve answers for queries: *What is the unique persistent identifier of a function with the name "foo" as of the most recent version of the project "bar"? What are its dependencies?* In other words, we are looking at a materialized view over the log at a given time point (with indices or cached values for faster retrieval).

- *Read and write records*: in this type, we are interested in retrieving individual records from the log and inserting new ones to the log. We use this access when building the materialized views with indices over the log, inserting new code and metadata, and reacting to requests or code changes. For example, a continuous integration service would read an execution request, retrieve relevant code, attempt to compile it or run a particular test suite, and write back the result.

These two types of access and interaction follow from what we observed in conventional programming environments. For the read-only usage, we can facilitate this access through using a query language. In our discussion, we assume a query language to have at least an expressive power of Datalog [79]. To coincide with our proof of concept implementation, we show our examples in a notation similar to Datomic's [80] query language, a variant of Datalog, expressed as a Clojure domain specific language. For simplicity, we omit discussing more complex expressions with parametrized bindings and aggregates. Instead, we present each query in the following form: [: **find** variable + :in $ % :where clause+] where variable+ represents one or more symbols starting with '?' that bind a variable in one of more clauses. The dollar symbol represents an external data source (a materialized view at a given time point) and the percentage symbol represents logical rules, i.e. intesional predicates. Clauses then contain expressions with extensional and intensional predicates. Extensional predicates follow a schema obtained from the data source. The example queries we formulated above would be expressed as follows:

```
[:find ?uid :in $ % :where
[?c :code/uid ?uid]
[?c :code/name "foo"]
[?p :project/contains ?c]
[?p :project/name "bar"]
[?p :project/latest true]]


[:find ?dep :in $ % :where
[?c :code/name "foo"]
(trans-dep ?c ?dep)
[?p :project/contains ?c]
[?p :project/name "bar"]
[?p :project/latest true]]
```

The trans−dep symbol here refers to an input rule for computing transitive dependencies defined as:

```
[[(trans-dep ?c1 ?c2) [?c1 :code/uses ?c2]]
 [(trans-dep ?c1 ?c2) (trans-dep ?c1 ?c)
                      (trans-dep ?c ?c2)]]
```

This rule has two parts. The first part says $?c2$ is a transitive dependency of $?c1$ if $?c1$ directly uses $?c2$. The second part says that $?c2$ is a transitive dependency of $?c1$ if there is $?c$, such that $?c$ is a transitive dependency of $?c1$ and $?c2$ is a transitive dependency of $?c$.

For evaluating queries at a given time point, we assume the data source can be obtained and provided as follows: (as−of (connect uri) txtime). Here, uri is the data source universal resource identifier, connect is a function that provides a connection to the data source, txtime is a time reference with the respect to the log, and as−of is a function that provides the view over that data source as of a given time point. The evaluation of each query results in a set of tuples.

For a fast access in certain workloads, different applications may provide an access to a subset of the stored information. These applications may choose to create specialized indices or caches that help with their workloads. For general lookups, the indexing strategy of Datomic[1] (a deductive database used for the prototypes in Chapter 5) can be employed: it uses 4 covering indices with different sort orders to allow efficient navigation in different directions of each relation.

The second type of interaction is on the level of reading and writing individual log entries. We can facilitate this type of interaction through using the publish-subscribe pattern. The log plays the role of a message broker. For example, given `conn` is a connection resulting from `(connect uri)`, Code Listing 4.1 shows how one would publish new data.

Listing 4.1: Example of transacting a record

```
(transact conn [{:db/id uid1
  :code/name "baz"
  :code/source "..."}
  {:db/id uid2
  :project/name "bar"
  :project/contains uid1}])
```

In the example, we are adding a record with code named “baz” into a project named “bar”. The effect of this transaction depends on the previous events in the log: new entities may be created and related metadata attached to them, metadata about existing entities may be updated, or everything stays the same (if a record with the same content was previously committed). The schema attributes are resolved to concrete identifiers at that time.

For retrieving new records as soon as they are added to the log, we would

[1] `http://docs.datomic.com/indexes.html`

use an intermediate structure. Given queue refers to a structure that is obtained from the connection and filled with subscribed records from the log, the following code would forever process newly arrived data:

```
(while true
  (let [data (.take queue)]
    (do-something data)))
```

In this example, we assume the .take method blocks if the queue is empty and the do−something function processes retrieved records. Altogether with the log metadata, the obtained data would contain the transacted record, e.g. what we showed in Code Listing 4.1.

We have seen two types of accessing and interacting with the stored code and metadata: the flatten query-oriented one for temporal read access, and the transaction-oriented one for inserting and processing new records. Both types are in line with our requirements for expressiveness, flexibility and extensibility: the record structure is flexible to capture any form of metadata about code entities and be extended to new ones; the query-oriented access allows us to express queries that utilize different kinds of metadata. Both types enable extensions for new services by providing a flexible and uniform interface.

4.4 Summary

In this chapter, we discussed different issues and explained design concepts behind Search-focused Programming environments. We first set key properties we expect from Search-focused Programming environments: determinism, global disambiguation, flexibility, extensibility, expressiveness, and the absence of inherent bottlenecks. Each of these properties stands on its own and motivated issues we discussed for different areas of Search-focused Programming environments.

The first area we focused on was code distribution. As we identified in

Chapter 3, conventional programming environments distribute code in bulks and rely on complex version numbering procedures to prevent identifier collisions of individual definitions. In addition to that, identifiers in conventional programming environments are mutable, location-dependent and globally non-unique. In SFP, we propose an alternative approach where identifiers for code entities are persistent, globally unique, and deterministically assigned based on code content. We look at different issues with regards to this persistent identifier scheme, such as what exact content to base identifiers on or what to do with programming language features that are intervened with user-assigned names (e.g. protocols or global variables).

We then focused on code organization. Given entities in SFP are assigned globally unique identifiers, they can be associated with additional facts about them. We discussed two forms of such facts: temporal and persistent. Classifying a fact as temporal or persistent depends on the perceived reproducibility or irreversibility of the process for obtaining a given fact.

The data source for all code and metadata is a uniform log-centric storage. The log-centric storage plays a key role as an integration component in enterprise architectures. It is also a basis for fault tolerance and scalability in various system implementations. The storage flexibility allows extensions of stored data. Lastly, the log provides a deterministic access for obtaining changes in temporal facts as well as obtaining an aggregated state at any given point in time.

We proposed two ways for accessing and interacting with code and metadata. The first one uses a materialized flat view at a given time in the form of a deductive database. In this way, we have enough flexibility and expressiveness for queries to retrieve stored code and its corresponding metadata. The second way is a transaction-oriented access for writing and retrieving new records. This access can enable new extensions that process stored facts as soon as they are inserted.

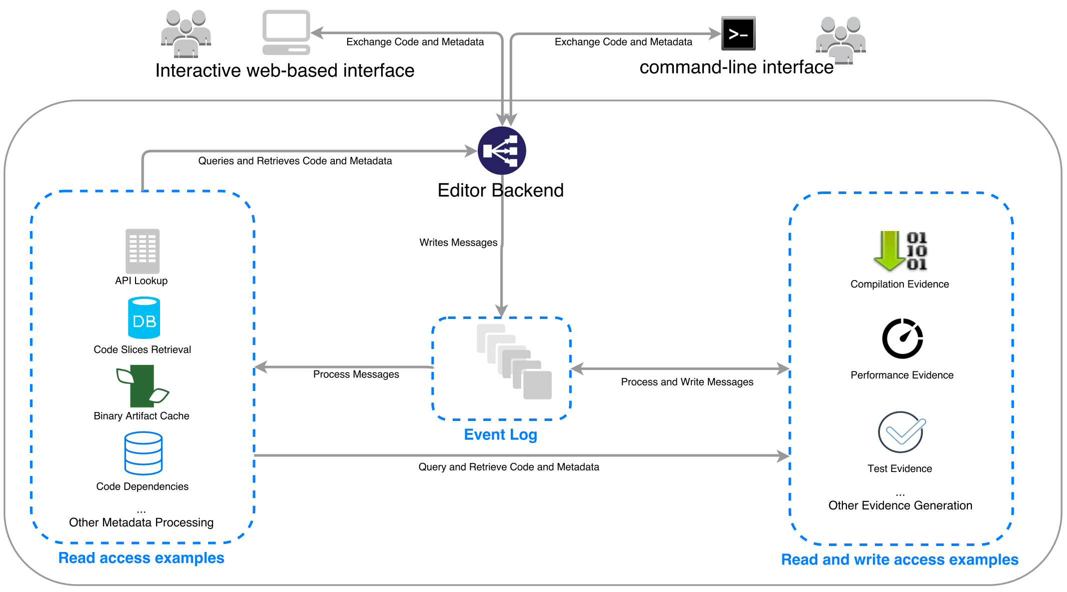

Interactive web-based interface
Exchange Code and Metadata
Exchange Code and Metadata
command-line interface
Queries and Retrieves Code and Metadata
Editor Backend
Writes Messages
API Lookup
Code Slices Retrieval
Binary Artifact Cache
Code Dependencies
...
Other Metadata Processing
Read access examples
Process Messages
Event Log
Process and Write Messages
Query and Retrieve Code and Metadata
Compilation Evidence
Performance Evidence
Test Evidence
...
Other Evidence Generation
Read and write access examples

Figure 4.1: Search-focused programming environment: storage, access and interaction

The summary of Search-focused Programming environments is shown in Figure 4.1. The log collects code and metadata from users as well as automated services (which process the log and write back to it). The records in the log can be materialized in flat views which are used for fast and declarative access.

Chapter 5

Implementation of Prototype Interfaces

5.1 Introduction

This chapter presents two directions in which interaction with code in SFP environments can be realized. The proof of concept implementation of each direction demonstrates the flexibility of SFP. The first direction aims to mimic the commonly present text file editing interface of conventional programming environments. The second direction shows a more interactive approach without intermediate interactions with files and namespaces. Given both of these interfaces are built on top of what we described in Chapter 4, code definitions originating in one interface can be used in the other and vice versa.

5.1.1 Language Choice and Previous Work

Before presenting the proof-of-concept interface implementation, we comment on differences from previous work [81, 82] in early stages and the choice of ClojureScript. ClojureScript was chosen for practical reasons:

- Its sizeable community and usage in industry can be more adequate in our setting than, for example, a purely academic language environment.

- Its uniform syntax and plain namespace resolution rules made implementation and experiments more manageable than it would have been in environments with complex combinations of language versions, extensions or globally scoped language features.

- The presence of existing tools, such as for the storage backend or experimental editors, enabled more code reuse.

Previous work [81, 82] describes work done in the context of Python and Haskell programming environments and briefly outlines some of ideas that are examined in detail in Chapter 4. The focus differed from goals set out in Chapter 1 and mainly centred around modular tool development [82]. SFP may be suitable for modular tool development, but the pursuits in this area remain as future work due to the scope described in Section 7.2.6.

5.2 Command-line Interface

Since our primary goal is recreating the interaction known from conventional programming environments, we follow examples we outlined in Chapter 3. For example, we can take the client project definition and code we showed in Code Listings 3.3 and 3.4. From that example, the namespace facility together with the project definition could be interpreted as the following query:

Listing 5.1: "Namespace as query"

```
[:find ?uid :in $ % :where
[?c :code/uid ?uid]
[?c :code/name "right-pad"]
[?f :file/namespace "com.example-utils"]
[?f :file/contains ?c]
[?p :project/contains ?f]
[?p :project/name "com.example/example-utils"]
[?p :project/version "0.1.0"]]
```

We look in detail how we can generate queries from project definitions and source code namespace directives in Section 5.2.1. Even though the extracted query contains the same information as the original project definition and source code namespace directives, there are several differences to be noted:

1. Extracted queries may seem more verbose and we do not assume they would be written manually. Instead, we can think of the project definition and source code namespace directives as a "frontend" or "surface" language, and the extracted query represents a "backend" language that the surface can be translated to. This is the main idea behind the command-line interface presented in this chapter.

2. Even though the example extracted query encodes the original project-file-code entity hierarchy, it does not need to do so in general (e.g. it could only work with code entities).

3. Even though the example of the extracted query only contains the information in the source project definition and namespace directives, it could in general utilize more metainformation.

4. The original namespace directives refer to a local environment that is assumed to be modified by a dependency manager (which uses the project definition in order to do so). The query is evaluated against a specified database at a given time point (which gives a view over the log as we discussed in Section 4.3).

5. The original namespace directives refer to symbols that can be found in external files. These files are assumed to exist in a local environment (e.g. previously compiled). The query resolves to a globally unique identifier of a code entity that may exist in a local environment or brought to it on-demand.

6. The original namespace directives are assumed to map each symbol to

one definition. If there are multiple definitions, it depends on the implementation strategy – some interpreters or compilers may terminate with an error, others may resolve the conflict in some way (e.g. by an implicit ordering of search paths). On the other hand, the query evaluates into a result set which may have a size different than one. A result set with more than one element may be a desired feature (if we wish to generate different program variants) or not. When it is not a desired feature, we revert to some conflict resolution strategy, similarly to conventional programming environments. We expand on this topic in Section 5.2.1.

What we mentioned relates to using external code locally. Other differences arise when we try to share our local code. Given we assigned an identifier to the shown code as well as the file and project it was in, we can imagine transacting the following record:

Listing 5.2: Generated metadata

```
[{:db/id uid-show-progress
  :code/uid uid-show-progress
  :code/name "show-progress"
  :code/source "(defn show-progress ..."
  :code/dependsOn uid-right-pad}
 {:db/id uid-file-example
  :file/namespace "com.example-client"
  :file/contains uid-show-progress}
 {:db/id uid-project-example
  :project/name "com.example/example-client"
  :project/version "0.1.0"
  :project/contains uid-file-example}]
```

As we discussed in detail in Chapter 4, some code (and metadata) is tied with global names. For that, the user needs to be able to choose whether the identifiers of entities tied with names should be fixed to some existing identifiers or generated as new ones.

5.2.1 Generating Queries

ClojureScript (and Clojure) include many commands for manipulating namespaces. The most common ones are: **require**, **refer**, **use**, and **import**. These commands are generally not used directly. The idiomatic way is to use the namespace (**ns**) macro, as we have seen in our code examples. Each command may also have additional options:

- **require**: This command loads all definitions from a given namespace (and loads any additional namespaces transitively needed) and exposes them via a fully qualified path symbol. It can take additional arguments: :as introduces a shorthand alias symbol for the fully qualified path, :**refer** exposes symbols from a given namespace directly without the qualifying prefix.

- **refer**: This command works the same way as the optional argument of **require** – it exposes symbols from other namespaces without using their fully qualified prefixes. It allows additional arguments for selectively exposing only specified symbols or renaming them.

- **use**: This command is a combination of **require** and **refer**.

- **import**: In Clojure, this command is used for working with Java packages and class names; in ClojureScript, it is only used for working with Google Closure's JavaScript library.

The combination of different commands and different options is fairly complex, so for the purpose of our proof of concept implementation, we decided to focus on a subset of available namespace functionality. Namely, we consider the following constructs:

```
(ns ...
  (:require nsfull))


(ns ...
  (:require [nsfull :refer [...]]))


(ns ...
  (:require [nsfull :as nsshort]))
```

We focused on two areas of interaction in command-line interface implementation: deployment and build processing.

5.2.2 Deployment

By deployment, we mean the procedure of publishing a project in a community-shared repository, similarly to what we described in Chapter 3. In our setting, it would mean processing local files and transacting corresponding code and metadata into the log. We implemented the command-line interface as an extension of Leiningen's cljsbuild plugin in Clojure – the prototype, hence, operates with the standard project structures we presented in Chapter 3.

We show the main routine in Algorithm 3 for deployment: it is a simplified pseudocode that omits various implementation details and presents the procedure more in an imperative style for a wider audience. The routine starts by initialization. It creates a connection to either an implicit repository or to a repository provided in the project manifest. It then tries to retrieve the project's unique identifier: if a project of a given version exists, it will get its unique identifier; otherwise it will transact a record with this project metadata and return its identifier. In the prototype implementation, the identifier assignment was left to the database engine. This is clearly a limitation, and the full implementation will need to include a deterministic identifier assignment for non-code entities as well. In this case, the identifier can be generated from the project metadata, time and author.

To emulate the bulk processing of conventional programming environments, the resolution of dependencies is not on-demand (as in the example Code Listing 5.1), but done altogether in the routine for creating a global environment. This subroutine for retrieving a global environment mapping is shown in Algorithm 1. This subroutine takes listed dependencies from the manifest and composes a disjunctive clause that relates all projects to their contained files. The remaining query is almost the same as in Code Listing 5.1, but without the specified namespaces and name symbols. The evaluation of this query, hence, returns triples which contain the full namespace path, the exported name symbol, and the unique identifier. From this result set of triples, a straightforward nested mapping is constructed. If during the environment construction, a symbol collision is encountered, a conflict handling routine is called. The default behaviour is to print a warning and continue. This behaviour emulates the one in conventional programming environments; we discuss other possibilities in Chapter 7.

After generating the global environment, the routine continues by retrieving and sorting the input files (using internal functions from the compiler). Each input file's content is processed – given ClojureScript is a Lisp-family language, the file content is read as a sequence of form expressions. As with projects, a file namespace entity identifier is retrieved or created and a local environment is initialized.

Each form expression is analysed using the unmodified ClojureScript compiler's analyser (details in the pseudocode are simplified). If the expression is controlling the namespace, the local environment is modified accordingly (any aliases and references are added accordingly). If the expression depends on a global name (as discussed in Chapter 4), different policies may be applied: either (if any) old identifiers can be used or new ones are generated. Since change management is tied with this feature and is not fully implemented in the prototype, the implementation's default behaviour is to follow the same procedure as expressions independent of global names – the difference is that the global name is added as the input for the identifier generation. This is clearly a limitation of the prototype – we expand on the full implementation in Chapter 7.

Processing of forms is shown in Algorithm 2. The symbol resolution is a lookup using the local and global environments. Through that, we get a sorted set of identifiers of dependencies. The prototype implementation assigns identifiers as the default Murmur3 hashes of dependencies and textual source content (obtained from a source logging pushback reader). Chapter 4 provided a throughout discussion about different options for assigning identifiers, so we briefly comment on the prototype implementation choices:

- Using textual source content has a few drawbacks (e.g. identifier assignment sensitive to small changes), as discussed in Chapter 4. This choice in the prototype is mainly due to implementation details: ClojureScript's parser may be non-deterministic due to reader macro expansions. For example, an expression ‘(x#) is implicitly expanded to a new generated symbol (e.g. (x__2804__auto__)). If we wanted to use the abstract syntax tree and enforce determinism, we would need to take additional steps in the implementation, such as using a custom parser, modifying the procedure how these symbols are generated or defining our hash function to ignore such symbols. One other reason for using textual source content was the implicit storage of the form that is suitable for editing.

- Murmur3 is a non-cryptographic hash function, so it would be potentially prone to collision attacks. We did not pose security requirements on the prototype implementation, so the choice of a hash function was not essential in the implementation. The full implementation would need to consider this requirement as well as other issues discussed in Chapter 4 (e.g. using multiple hash functions).

After the identifier is assigned, we modify the local environment (point all defined symbols to this identifier) and prepare and transact metadata – i.e. producing transactions with records similar to the example in Code Listing 5.2. After the whole file is processed, its local environment is added under a corresponding namespace key in the global environment.

```
Input: Project definition, connection, time
Output: Global environment
clauses ← () ;                          // empty list initialization
foreach [project-name, project-version] ∈ proj − deps do
    append to clauses: (and [?p :project/name project−name]
      [?p :project/version project−version])
end
query ←
  [:find ?n ?sym ?fid
                  :where
                  [?p :project/contains ?f]
                  [?f :file/namespace ?n]
                  [?f :file/contains ?d]
                  [?d :code/uid ?fid]
                  [?d :code/name ?sym]
                  (or clauses)];
result ← eval(query, conn, t);
foreach [ns, symbol, uid] ∈ result do
    if genv[ns][symbol] exists then
        handle-conflict(projdef, conn, t);
    end
    else
        genv[ns][symbol] ← uid ;
    end
end
return genv
```

Algorithm 1: Global environment generation

5.2.3 Build Processing

Building ClojureScript code may involve various options for build configurations that we do not aim to replicate in the prototype implementation. We make several assumptions:

- The unmodified version of the original compiler is used. This assumption

```
Input: Used symbols, connection, form expression, a log of
       transactions, file unique id, local and global environments
Output: Local environment
$deps \leftarrow resolve - symbols(used, lenv, genv)$ ;
$uid \leftarrow hash(deps, exp)$ ;
$txout \leftarrow prepare - and - transact(conn, fuid, exp, defined, deps, uid)$ ;
append $txout$ to $txs$ ;
foreach $symbol\ s \in defined$ do
    $lenv[s] \leftarrow uid$ ;
end
return $lenv$
```

Algorithm 2: Process form expression

```
Input: Project definition
Output: A log of transactions
$txs \leftarrow []$ ;                                  // empty vector initialization
$conn \leftarrow create - connection(projdef)$ ;
$puid \leftarrow get - or - create - project - uid(projdef, conn, txs)$ ;
$genv \leftarrow create - global - env(projdef, conn, current - time)$ ;
$inputs \leftarrow find - src - files(projdef)$ ;
$inputs \leftarrow topsort(inputs)$ ;
foreach $input \in inputs$ do
    $lenv \leftarrow \{\}$ ;                                  // empty map initialization
    $fuid \leftarrow get - or - create - file - uid(conn, puid, input, txs)$ ;
    foreach $form\ f \in input$ do
        if $f$ is ns then
            $lenv \leftarrow process - ns(genv, lenv, f)$ ;
        end
        else
            $(defined, used) \leftarrow analyze(f)$ ;
            if $f$ is name-dependent then
                $lenv \leftarrow$
                  $process - nform(used, conn, f, txs, fuid, lenv, genv)$ ;
            end
            else
                $lenv \leftarrow$
                  $process - form(used, conn, f, txs, fuid, lenv, genv)$ ;
            end
        end
    end
    $genv[input - ns] \leftarrow lenv$;
end
return $txs$
```

Algorithm 3: Main routine for deployment

implies we need to provide it with inputs serialized in files.

- The source code was previously processed by a deployment procedure. This assumption implies each definition extracted from the input files was assigned a globally unique identifier.

- We focus on compiling ClojureScript code into a JavaScript file that can then be loaded by a web browser. This focus implies a lack of a traditional single "entry point" in the sense of computer programming (e.g. a single dedicated function named "main"). Instead, we assume we are provided a set of identifiers indicating any code blocks or definitions annotated with `^:export`. From these annotations, we obtain a set of points to all "entry points", i.e. code that should be translated (altogether with all its transitive dependencies) into JavaScript. After the compilation into JavaScript, the translated code "entry points" can be called from within the JavaScript execution environment in a web page that loaded it.

The main routine is shown in Algorithm 4. First, all dependencies are collected altogether with their respective file names. While deployment involves resolving direct dependencies, transitive dependencies are collected for build processing. The collected dependencies are, thus, traced from "entry points" transitively. Transitive dependencies can be captured using the following rule definition in the Datomic query language:

```
[[(transDepends ?t1 ?t2) [?t1 :code/dependsOn ?t2]]
 [(transDepends ?t1 ?t2) (transDepends ?t1 ?tx)
                         (transDepends ?tx ?t2)]]
```

The first part of the rule states that transitive dependencies are direct dependencies. The second part is then recursive, stating given `?tx` where `?t1` transitively depends on `?tx` and `?tx` transitively depends on `?t2`, we conclude that `?t1` transitively depends on `?t2`.

Some definitions may be associated with several file namespaces globally. Since we emulate the semantics of command line interfaces, these conflicts are resolved by restricting files to the ones specified by the project-level dependencies. Then, for each dependency, we output its code to a corresponding file. If the file does not exist, we create it and generate a namespace header for all co-located code fragments.

```
Input: Entry points, connection
$fdeps \leftarrow collect - dependencies(entry - points, conn)$ ;
foreach $[file - name, dep - src] \in fdeps$ do
    if file-name does not exists then
        create $file - name$;
        $header \leftarrow generate - header(file - name, fdeps)$;
        output $header$ to $file - name$;
    end
    output $dep - src$ to $file - name$;
end
compile-all(src-dir);
```

Algorithm 4: Build processing routine

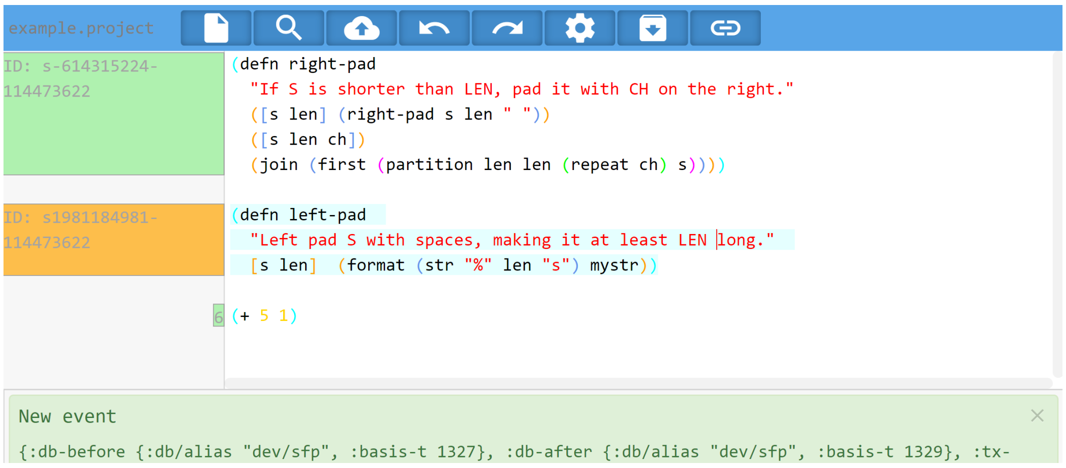
5.3 Interactive Web Interface

The goal of the command-line interface prototype in Section 5.2 was to emulate the bulk processing of conventional programming environments. The goal of the interactive web interface prototype is to facilitate a more direct interaction with the stored code and metadata. Since one of the aims of SFP is the separation between the stored and edited or visualized code, we chose a web interface over extending traditional editors which may be more tied to the file storage.

The interface underwent several iterations of its development. The first iterations utilized the CodeMirror [83] editor, firstly in a custom interface, later in a modified interface of the LightTable editor (which itself utilizes CodeMirror). The CodeMirror editor is flexible and extensible, works across different versions of many web browsers and provides specialized editor modes for over 100 programming languages. Its main focus is, however, plain textual editing and a tremendous amount of engineering has been put into it. With that, it becomes rather non-trivial for structural operations and book keeping we need for working with dependencies on a definition-level.

The current iteration has, thus, moved towards a more structural editor. The underlying editor that has been extended is an experimental editor *paren-soup*. Unlike CodeMirror, it only supports a single language, ClojureScript, which is also its implementation language, and only creates editors out of marked HTML elements with the "contenteditable" attribute. It uses a simple embedding of the abstract syntax tree in the HTML Document Object Model (DOM): each abstract syntax tree node becomes a span element which contains the node type in its class attribute. In addition to this convenience for our prototype implementation, it provides two builtin features desirable for our prototype – we describe them in Sections 5.3.1 and 5.3.2.

The interactive web interface prototype aims to enable working directly with external code. This is in contrast with the command-line interface where external code lies behind two layers of indirection: the project manifest and the namespace commands. We describe two main ways of interaction: one is looking up external code and inserting references to it directly into the abstract syntax tree; the other is editing external code. These interactions are described in Sections 5.3.3 and 5.3.4. All this is done in the context of an interface which allows flexible organization (i.e. fragments of code are not constrained by file boundaries). The editing area (Figure 5.1) can, thus, contain code of different origins.



```
(defn right-pad
  "If S is shorter than LEN, pad it with CH on the right."
  ([s len] (right-pad s len " "))
  ([s len ch])
  (join (first (partition len len (repeat ch) s))))

(defn left-pad
  "Left pad S with spaces, making it at least LEN long."
  [s len]  (format (str "%" len "s") mystr))

(+ 5 1)
```

```
{:db-before {:db/alias "dev/sfp", :basis-t 1327}, :db-after {:db/alias "dev/sfp", :basis-t 1329}, :tx-
```

Figure 5.1: A screenshot of the experimental web-based interface.

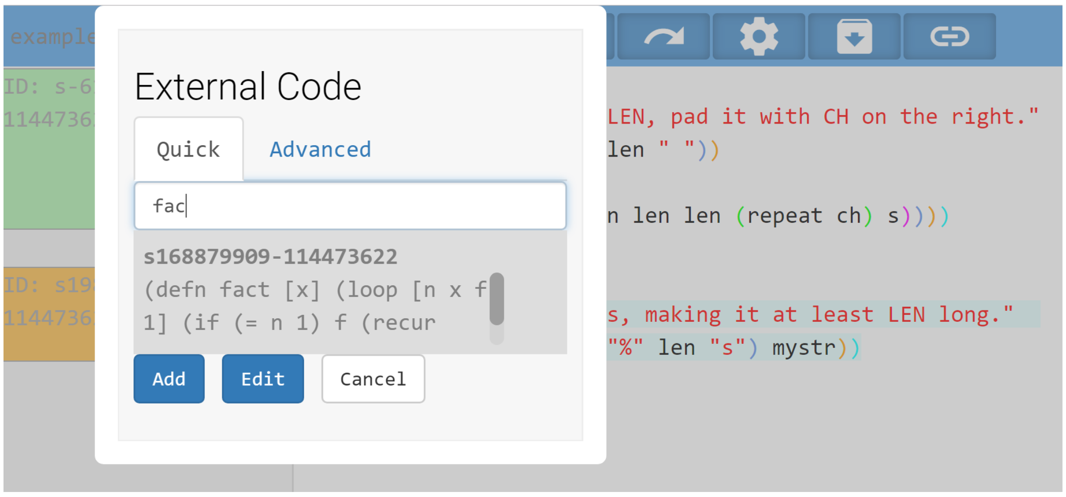
5.3.1 Instant Evaluation Environment

This functionality is better known as “InstaREPL”. The traditional REPL has an interactive command-line interface where expressions are typed and their results are written in a sequence. With InstaREPL, results appear side-by-side with all input. When input changes, results next to it get updated immediately. This concept has been popularized with the mentioned Light Table editor that was used for the earlier iterations of this interface. This functionality was slightly augmented to display information related to SFP concepts. For expressions that define any symbols, it displays their assigned globally unique identifiers and indicates whether code has been shared in the repository by the background colour (green and orange), as shown in Figure 5.1.

5.3.2 Indentation-driven Structural Editing

The Lisp family of languages has the property of homoiconicity where the language syntax mirrors the structure of the abstract syntax tree. With that, it becomes a commonplace to use more structure-driven editing as opposed to plain text editing. The common approach is through using the ParEdit[1] mode extension that exists for various editors. ParEdit allows manipulating

[1]https://www.emacswiki.org/emacs/ParEdit



```
ID: s-6
1144736
ID: s19
1144736
```

```
fac
s168879909-114473622
(defn fact [x] (loop [n x f
1] (if (= n 1) f (recur
```

```
LEN, pad it with CH on the right."
len " "))

n len len (repeat ch) s))))

s, making it at least LEN long."
"%" len "s") mystr))
```

Figure 5.2: A screenshot of the external code search dialogue window in the experimental web-based interface.

Lisp code at the level of abstract syntax tree cells. Cells are manipulated by dedicated key bindings, e.g. for creating new ones or moving around the hierarchy. It is a fairly effective way for writing Lisp code, but it requires memorization of new key bindings.

A more recent alternative, named ParInfer[2], combines structural editing (in the spirit of ParEdit) with plain text editing. ParInfer achieves its editing mode by inferring the intended structure from whitespace indentation and completing or removing corresponding parenthesis. This editing mode has been ported to many other editors, including Code Mirror. For our prototype, it provides the simplicity of plain text editing combined with structural modifications, which we utilize for our further extensions.

5.3.3 Inserting Dependencies

Given some edited code, we would like to insert a reference to an external code entity. The first step is looking up external code. In the prototype, when control and insert keys are pressed, a lookup dialogue window is opened (Figure 5.2). The dialogue window allows either quick lookup using a keyword, or an advanced lookup by specifying a full query. Other options are possible,

[2]https://shaunlebron.github.io/parinfer/

for example displaying and navigating a hierarchical structure and selecting external code from it. Such alternatives are viable and perhaps desirable in the full implementation, but were not essential for the prototype implementation.

The keyword lookup shows the results in the order of insertion in the prototype. The task of selecting external code one wishes to use may involve writing additional keywords in order to narrow down the selection. As with displaying and navigating through external code, alternative options could be employed for the quick lookup functionality. For example, the ranking of quick search results can incorporate additional criteria from the context (projects encompassing other external code, user preferences and recent actions, etc.). Again, these options were not considered for the proof of concept prototype. They are, however, feasible, i.e. could be built on top of the same infrastructure, and can be a part of future work efforts described in Chapter 7.

After selecting the external code, a new variable node with a reference to its unique identifier is inserted into the structure at the cursor's location. Before the edited code is evaluated in InstaREPL, the external code as well as all its transitive dependencies need to be loaded. Transitive dependencies are retrieved in the same way as in the command-line interface. In the prototype, the external code with corresponding aliases is evaluated in the background using the editor's self-hosted REPL.

5.3.4 Editing and Synchronization

The second option is that we would like to edit an existing code. The procedure is the same as with inserting dependencies, but one would choose to edit instead of insert in the dialogue window (Figure 5.2). This appends selected code at the end of the editor window. Before that, just as with inserting dependencies, it loads all transitive dependencies with aliases needed by the selected code in the editor's self-hosted REPL.

For saving any changes from the editor, the prototype only provides a limited functionality. Visible code (and loaded dependencies) in the editor window can be saved locally (using the LocalStorage API of the web browser). For external synchronization, the editor code can be marked next to the editor window and synchronized after pressing the InstaREPL area. The synchronization assigns identifiers to marked code, and transacts them altogether with dependency metadata. The log of corresponding events is then displayed at the bottom of the editor screen (events are retrieved using the EventSource web API).

As a proof of concept prototype, the full metadata synchronization is not yet supported. With edited code, the prototype does not track the "origin" and does not transact additional metadata, such as corresponding file and project namespaces. The full implementation would require this, but the functionality goes beyond transacting original metadata, e.g. users would need to be authorized and permissions would need to be checked. For users synchronizing editing code from non-owned projects, the editor would either need to create their own "fork" or stop the synchronization with an error message. Lastly, for code that was written freshly in the experimental web editor, synchronization may need to present project and file assignment for compatibility with the command-line interface (which only has a restricted access to external code through generated queries, as described in Section 5.2.1).

Similarly, even on the level of code entities, the origin is not fully tracked in the prototype. This would be desirable for generating and transacting version control-related metadata. Again, this functionality is beyond the scope of a proof of concept prototype. Finally, the experimental editor only focuses on ClojureScript. The full programming environment, at least in the web development context (where ClojureScript is used), would need to account for editing additional resources, such as HTML, CSS and JavaScript.

5.4 Summary

This chapter presented two directions in which interaction with external code in SFP environments can be realized. The proof of concept implementations of these interaction approaches demonstrated the flexibility of interfaces for reading and writing data in SFP environments.

The first direction is more "conservative" and follows the interaction from conventional programming environments. It provides a command-line interface similar to what standard tools do and ensures compatibility with traditional file-oriented editors. The interaction is "indirect" in the sense that external code is accessed via two naming layers, as in conventional programming environments. Two naming layers are the project manifest and individual files' namespace control expressions. The access is done by taking these two naming layers and generating "bulk" retrieval queries from them. Sharing any changes is done similarly by "bulk" processing local files and transacting corresponding data. This direction shows that traditional interfaces can be emulated in SFP environments.

The second direction offers a more direct approach for interacting with external code. The external code can be navigated and queried using the query language of Datomic (a deductive database used in the prototype) in an interactive web-based editor. The editor itself combines structural and text editing. The overall organization may resemble traditional editors, but visible edited code may have originated across different projects and files. Sharing of edited code is then done per-individual definition rather than "bulk" as in the command-line interface. This direction shows that potentially new and more flexible interfaces can be realized in SFP environments.

Both directions were implemented as proof of concept prototypes, and as such, they do not provide a full programming environment experience. In particular, the interactive web-based editor only supports ClojureScript code, which would limit its scope. Providing a full programming environment is, however, beyond the scope of proof of concept prototypes.

Given both of these interfaces are implemented on top of the same infrastructure, it hints how interoperability can be achieved. While the first interface needs the naming hierarchy in order to access files, the second one needs results of cross-reference analysis with aliases for external code. This interoperability can be achieved by providing and transacting additional data on sharing code. For example, the interactive web-based editor may need to present a project and file namespace assignment in order to achieve interoperability.

This page is intentionally left blank.

Chapter 6

Existing Repository Analysis

6.1 Introduction

In this chapter, we evaluate how ideas from SFP environments would affect existing code. It demonstrates that SFP environments could scale up to the existing repository sizes. We also analyse in detail the effect of tracking dependencies at a finer level. This analysis shows whether the concepts from SFP could potentially be applied independently of the overall environment as well as how structures in existing code fit this setting.

In particular, Section 6.2 describes the experimental setting, how repositories were processed and the final dataset was obtained. With the dataset, we conducted three experiments regarding definition-level dependencies. For the top five most depended on projects, Section 6.3.1 compares the dependencies declared on the project level with the dependencies on the definition level we obtained from cross-reference program analysis. These results give an idea of how suitable project level dependencies are and what portion of exposed functionality in popular libraries is actually used across all projects in a community repository.

Section 6.3.2 presents the results with the respect to version information, i.e. what portion of definitions is shared across a different number of successive versions. These results indicate, given the observed transitive usage, some version changes may not affect clients even though they require their explicit management. Lastly, Section 6.3.3 takes sample client code in two of the libraries and presents results from measuring compilation time overhead with project-level dependencies. These results suggest potential utility of definition-level dependency tracking in existing programming environments. Section 6.5 discusses and summarizes our findings from the existing repository analysis.

6.1.1 Research Questions

To guide our evaluation presented in this chapter, we formulated three research questions. The overall motivation is to compare conventional and Search-focused Programming (SFP) environments by analysing structures in existing code repositories and by conducting experiments on their effects. In this section, we state each research question and explain its motivation in detail.

RQ1 (Usage). *How do declared project dependencies in conventional programming environments compare with transitively used code (that is taken as a dependency in SFP)?* When introducing our problem setting in Chapter 1, we introduced key challenges inherent to conventional programming environments. We, however, do not know how prevalent some of these challenges are. Are declared project dependencies in existing code repositories optimal? By optimal, we mean that they only contain code that is (transitively) used, e.g. due to a manual curation of code maintainers. Understanding project-level and code-level dependency structures in existing code can give us a comparison on the granularity of code distribution in conventional and SFP environments. Answers to these questions enable us to discuss implications for current programming environments.

RQ2 (Persistence). *How does project content persist across different project versions?* Different projects follow different conventions with regards to annotating their versions. Even with different conventions, version tracking happens at the same (project-level) granularity. Each new version of external project dependency, thus, requires potential attention, maintenance, recompilation etc. on the client side. It may be the case that a client code depends on a part of a project that did not change, yet the client is forced to worry about the project level version change. Does project content change as much as project versions? Looking into these issues can give us insights into change management in conventional programming environments and what it means in the context of SFP.

RQ3 (Effects). *What are effects of conventional and SFP code distribution on the compilation pipeline?* Static characteristics of existing source code repositories give us understanding and comparison between conventional and SFP environments. The other area worth examining are effects on existing tools in the presence of these differences. The compilation pipeline is potentially directly affected by distribution of unused code in conventional programming environments. Comparison with code distribution in SFP can partially give us a sense of practical impact on programming environments. Such estimates help us understand and discuss how some of techniques we presented in the context of SFP could be adopted and incorporated into existing environments.

6.2 Method

In this section, we describe how we collected the dataset, pre-processed it in different steps and stored extracted information.

6.2.1 Context

The example of a conventional programming environment was the ecosystem around ClojureScript. We described it in detail in Chapter 3. In summary, the dependency management of Clojure and ClojureScript piggybacks on Java's

Maven repository infrastructure. Each artifact is published as a JAR file containing ClojureScript source code. Dependencies are specified against these artifacts in configuration files of build automation tools (e.g. Leiningen or Boot) and may be scoped (e.g. only for test time). Information specified in configuration files of build tools then translates into Maven's `pom.xml` files. Maven automatically resolves transitive dependencies from them. Even though dependencies may be on "snapshots" or version ranges, the recommended guideline encourages using specific versions.

6.2.2 Data Collection

We focused on the Clojars repository, since it is the default community repository of open source Clojure and ClojureScript projects. We obtained the offline copy of the entire repository via rsync: the snapshot we used contained 115,917 artifacts. In addition to that, we downloaded all versions of two additional libraries (core.async and transit-cljs) from Maven Central (the default Java ecosystem repository) via HTTP. These two libraries were not present on Clojars, but we found them among the most depended on projects during preprocessing (Section 6.2.3). Lastly, for the compilation pipeline experiments, we collected the sample client code from Github repositories of respective projects (Reagent and om) and retrieved any additional dependencies from Maven Central.

6.2.3 Pre-processing

We pre-processed the dataset in two stages.

Stage I. We first parsed all `pom.xml` files (this metadata is described in Chapter 3) and examined their declared dependencies. We took dependencies marked as provided, compile, or system. We resolved the "snapshot" dependencies to the closest stable or latest nightly release. We selected only those projects that declared a dependency on ClojureScript (any version), i.e. on the compiler and the standard library of ClojureScript. There were 9,582 projects that declared a dependency on ClojureScript. We stored all the parsed project-level information (Section 6.2.4). Based on the declared project-level dependencies, we selected the top five most depended on projects. We excluded Clojure and ClojureScript compilers and standard libraries (which were naturally depended on in every project) and CLJSJS projects (which package JavaScript libraries for easy code distribution in the Clojure ecosystem). The selected projects were the following ones:

- *core.async*: a library that facilitates asynchronous programming and communication.
- *Reagent*: a library that provides a ClojureScript interface to React. React is Facebook's popular JavaScript library for declarative creation of interactive user interfaces.
- *cljs-time*: a library for data and time operations (e.g. time zone calculations).
- *transit-cljs*: a library for the Transit data interchange (serialization) format.
- *om*: a library that provides a ClojureScript interface to React, but has various design and implementation differences from Reagent.

We selected these most depended on projects with regards to our research questions. These projects can aid in different development scenarios, are mature and have been actively developed for a few years. Given these properties, we can explore them and client code that depends on them in order to answer our research questions. If we selected projects that are not so depended on (e.g. newer libraries), our results could be limited by that.

Stage II. In the second stage, we processed the relevant artifacts of these selected projects as well as of all their dependants. For this pre-processing stage, we reused parts of the proof of concept command-line interface implementation we described in Chapter 5. In particular, we reused parts for a tool to extract dependency information from existing source code and extended it to read sources from JAR files. It calls the analyzer from the ClojureScript compiler, intercepts its environment after each top level form and does a custom symbol resolution. This is to perform a limited form of cross-reference analysis based on symbol usage. In particular, the macro definitions are not expanded, hence we only approximate an estimate of static dependencies. For the purpose of experiments, the unique identifiers were generated as hashes of the extracted dependency information and the source text using Clojure's `hash` function (which internally uses MurmurHash3). From each artifact, we extracted information about what files they contained, what top level expressions each file contained as well as details about each expression (if it defines any symbols, its dependencies, its unique identifier, etc.).

6.2.4 Storage

For storing the extracted information, we inserted everything into the Datomic database. We used Datomic in the prototype described in Chapter 5 and we reused parts of the prototype implementation, as mentioned in Section 6.2.3. From the first stage of processing, we stored project name, version, project-level dependencies, and other attributes (such as descriptions). Each project then refers to its files that were extracted in the second stage of processing. Each file then refers to top level expressions it contains. Lastly, there are details about each top level expression. We show the overall simplified schema of extracted information in Figure 6.1 – each rectangle represents an entity type with its attributes.

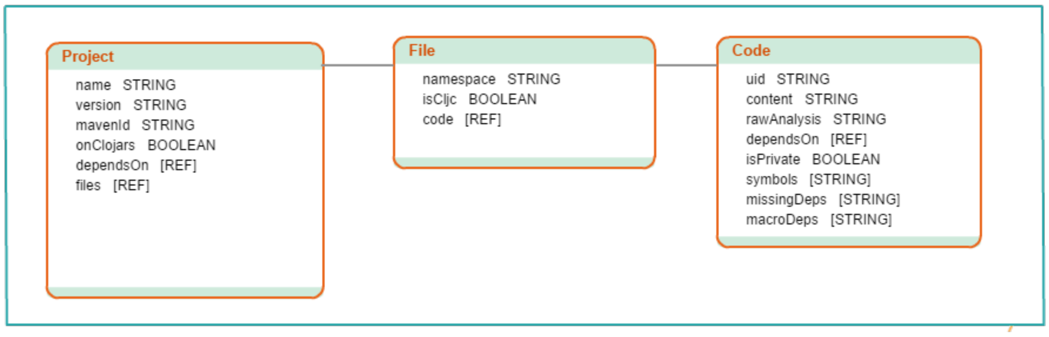


Figure 6.1: Simplified schema of the extracted information

6.2.5 Compilation Pipeline Experiments

In these experiments, we tried to estimate effects of conventional and SFP code distribution on the compilation pipeline. As mentioned in Section 6.2.2, we retrieved sample client code projects in Reagent and om repositories. For each of these projects, we prepared two variants. The first variant was the project with all code of transitive dependencies, as distributed in conventional programming environments. The second variant only contained used code from transitive dependencies (based on the used symbols in the client code), as distributed in SFP environments. The code for dependencies in the SFP pre-processed variant was extracted from the collected dataset into corresponding files and namespaces.

In addition to ClojureScript code, non-ClojureScript code dependencies were manually added to all project variants. They include the JavaScript dependencies (react, react-dom, and react-dom-server) and Clojure macros (be it directly in Reagent or om, or in transitive dependencies, such as core.async). The conventional programming environment variant of om's sample projects also required Java dependencies (transit-java and jackson-core) for successful compilation.

We tried to isolate the effect on the compilation pipeline from other effects (network latency and speed for distributing code, computation or caching of transitive dependencies, etc.), so the mentioned pre-processing was all done offline before the experiments. The experiments measured time for executing the compile function of ClojureScript compiler's build API (version 1.9.293 on Clojure 1.8.0) on each project's variant. We compiled all projects on the following platform with the HotSpot™VM (1.8.0_77): Intel®Core™i5 3570 CPU, 1600MHz DDR3 16GB RAM, Ubuntu 16.04.1. For the time measurement, we used Criterium[1] which takes necessary steps to gain stable results: non-measured warm-up iterations for the JIT compiler and managing GC before and after testing to isolate its impact on timing results. We measured each variant in an isolated REPL instance and every compilation was "cold start" from scratch (i.e. results from previous compilations were deleted). We used the default configuration of Criterium for 60 measured runs from which we report mean and standard deviation.

6.3 Results

6.3.1 Transitive Code Usage (RQ1)

In this experiment, we compared the portion of code that is being distributed in conventional programming environments with what is transitively used (i.e. that is distributed in SFP environments). As mentioned in Section 6.2, we selected the top five most dependent on projects, collected all their version artifacts and analysed each artifact. From this stored information, we report the total number of top level expressions that were not marked as private (e.g. by def−) and the total number of exposed top level expressions that are transitively used. Transitively used expressions include the ones that are indirectly (via other top level expressions) or directly used by their symbol in all client code (of projects that declared a corresponding project dependency). We then report the static usage estimate as this fraction:

$$\frac{\text{total number of exposed expressions transitively used}}{\text{total number of exposed expressions distributed}}$$

The results are shown in Figure 6.2 and Table 6.1: the second column shows

[1]https://github.com/hugoduncan/criterium

Table 6.1: Transitive code usage results

Library	**Version Artifacts**	**Exposed Expressions**	**Transitively Used**	**Usage**
core.async	18	366	206	56.3%
reagent	20	1064	572	53.8%
cljs-time	32	758	198	26.1%
transit-cljs	14	52	29	55.6%
om	56	1120	246	22.0%

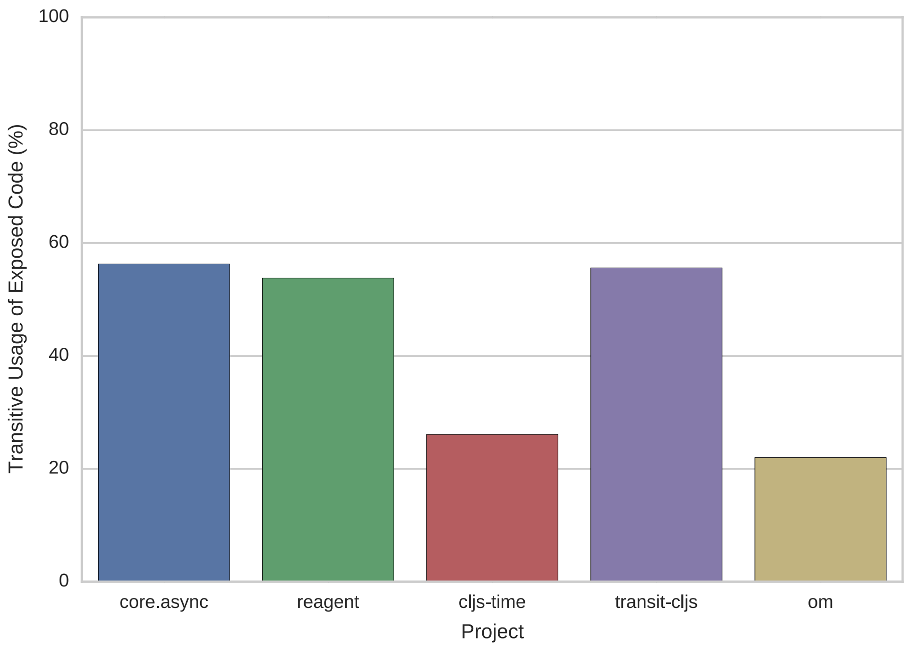


Figure 6.2: Transitive code usage (%)

the number of distinct version artifacts of each project, the third column shows the total number of top level expressions that declared some exposed symbols (from all versions), the fourth column shows the number of top level expressions that are transitively depended on, the last column estimates the overall static usage of top-level expressions declaring some exposed symbols.

We can see that the usage varies from 22% to 56.3%. We further discuss these results in Section 6.4.

6.3.2 Project Content Persistence across Different Versions (RQ2)

In this experiment, we evaluate how many top level expressions are preserved across different versions. We took all top level expressions present in the mentioned five most depended on projects and grouped them according to the number of project versions they were present in. The results are shown in Figure 6.3.

The majority of top level expressions are present in more than two versions. In Reagent, the number of top level expressions present in only one version slightly outweighs the number of expressions present in two or more versions.

6.3.3 Code Distribution Effects on the Compilation Pipeline (RQ3)

In this experiment, we try to estimate the effects of conventional and SFP code distribution on the compilation pipeline. As mentioned in Section 6.2.5, we took sample client code projects of Reagent and om and prepared two variants of their code dependencies. We then measured cold start compilation time for each sample project's variant.

The results are shown in Table 6.2 and Figures 6.4, 6.5 and 6.6: reported times are means and corresponding standard deviations of 60 measured runs.

The conventional programming environment code distribution of unused code caused the compilation to be 0.23x to 11.04x slower on the sample client projects than with only used code in SFP code distributed variants.

6.4 Discussion

In this section, we discuss our results, their practical implications and outline limitations of the experiments we conducted.

Version(s)	1	2+
core.async	106	260
reagent	552	512
cljs-time	143	615
transit-cljs	11	41
om	449	671

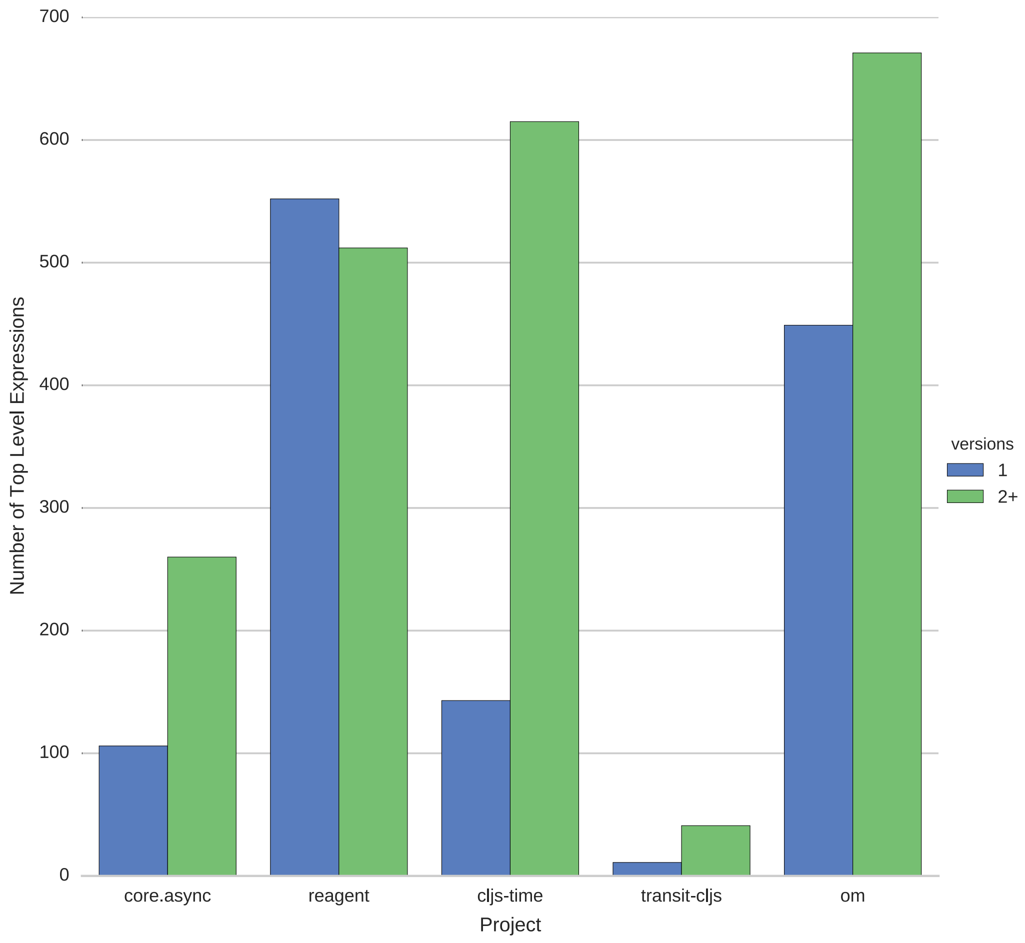


Figure 6.3: The total number of top level expressions of different projects present in one and two or more of their versions

Table 6.2: Sample client projects mean compilation times (in seconds)

Variant	conventional	SFP
Reagent		
simple	1.40 ± 0.06 s	1.06 ± 0.01 s
todomvc	1.46 ± 0.03 s	1.16 ± 0.05 s
geometry	1.50 ± 0.01 s	1.22 ± 0.03 s
om		
animation	18.71 ± 0.06 s	1.92 ± 0.02 s
counters	20.83 ± 1.74 s	2.55 ± 0.02 s
cursor_as_key	18.61 ± 0.04 s	2.13 ± 0.01 s
harmful	18.66 ± 0.53 s	2.08 ± 0.02 s
hello	18.39 ± 0.04 s	1.95 ± 0.01 s
input	18.47 ± 0.73 s	1.94 ± 0.02 s
instrument	18.64 ± 0.05 s	1.97 ± 0.01 s
mixins	18.44 ± 0.53 s	3.77 ± 0.03 s
mouse	21.18 ± 0.04 s	4.39 ± 0.04 s
multi	18.13 ± 0.70 s	1.97 ± 0.01 s
multiroot	18.92 ± 0.05 s	1.94 ± 0.01 s
refs	18.78 ± 0.71 s	2.06 ± 0.02 s
shared	18.75 ± 0.04 s	1.94 ± 0.02 s
sortable	20.54 ± 0.79 s	2.72 ± 0.02 s
state_bug	18.50 ± 0.04 s	2.16 ± 0.02 s
stateful	18.11 ± 0.60 s	1.64 ± 0.02 s
two_lists	18.71 ± 0.04 s	2.01 ± 0.02 s
typeahead	18.08 ± 0.73 s	4.48 ± 0.04 s
unmount	18.69 ± 0.04 s	1.99 ± 0.01 s
update_props	18.40 ± 0.58 s	1.94 ± 0.03 s
verify	18.63 ± 0.05 s	3.85 ± 0.04 s

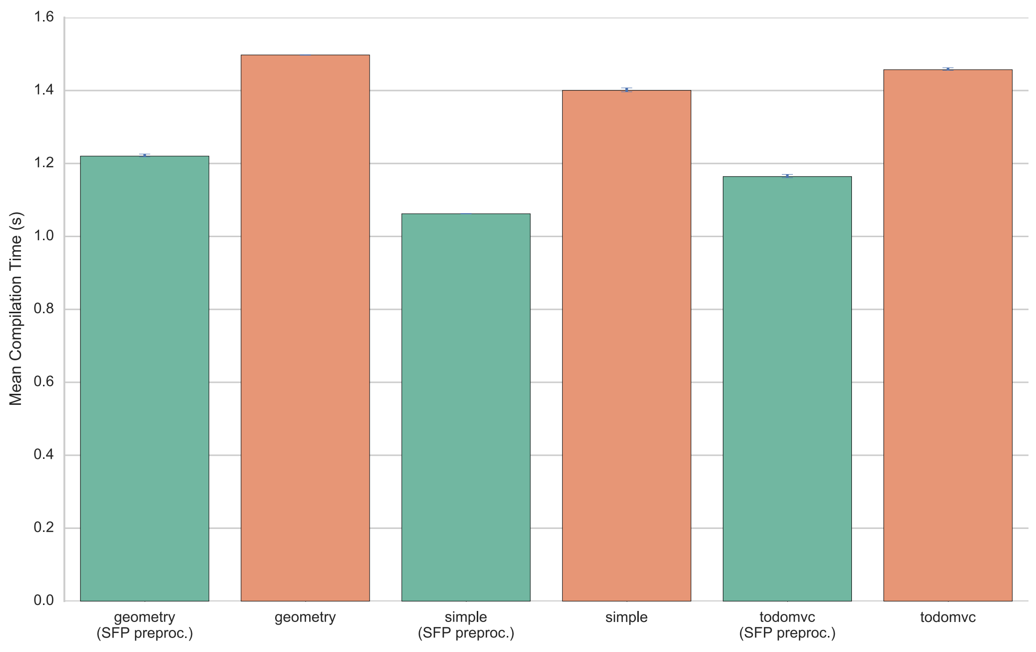


Figure 6.4: Reagent example projects compilation times (in seconds)

6.4.1 Results

Transitive Code Usage. Reported transitive dependencies were only the static ones and did not include dependencies introduced at runtime or via macros. The artifacts we processed only included the source files, i.e. no test or benchmark files. These facts should be taken into account when considering the usage and may explain why portions of code distributed in conventional programming environments were never used.

For example, among all ClojureScript projects we processed, we did not find any usage of `to-chan` function of core.async (which creates a channel out of a collection). This function is, however, used in internal test suites. The reported usage, hence, should not be interpreted as absolute.

Nonetheless, even though some parts of projects are not used (at least in application code), they are still distributed in conventional programming environments to all dependent projects. In our dataset, transitive usage of all exposed code of the five most depended on projects varied from 22% to 56.3%. In SFP environments, we would not need to distribute and process external code that is not used in our client code.

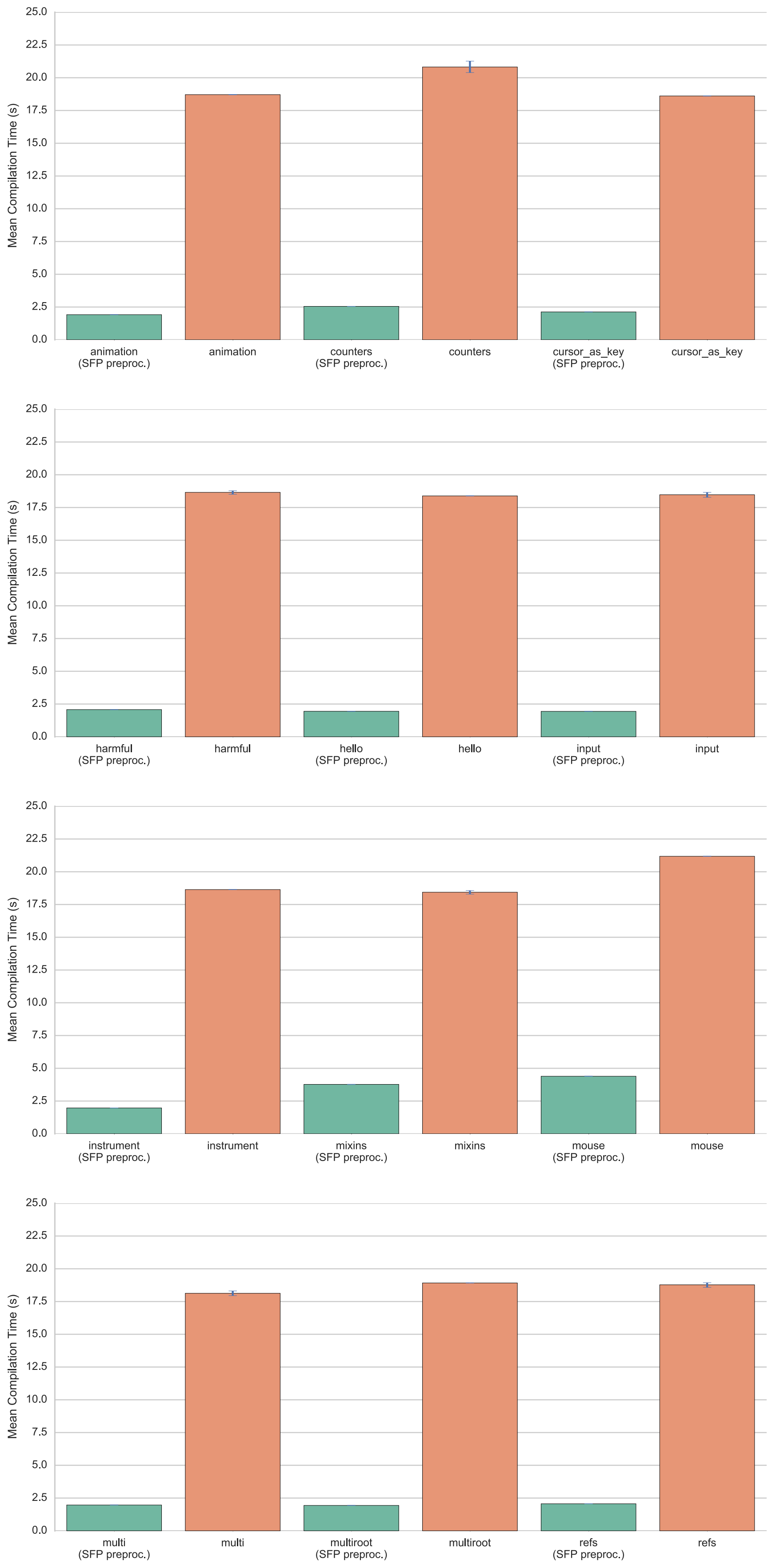
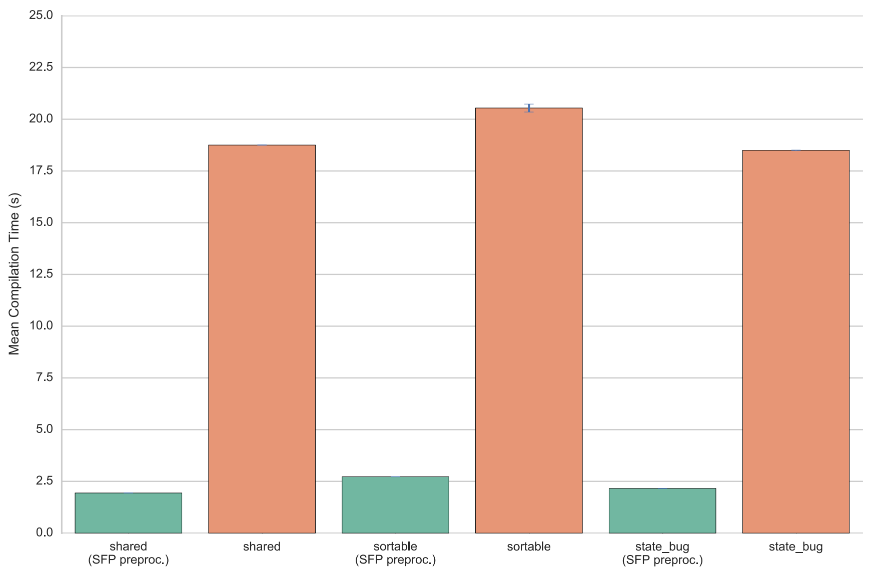
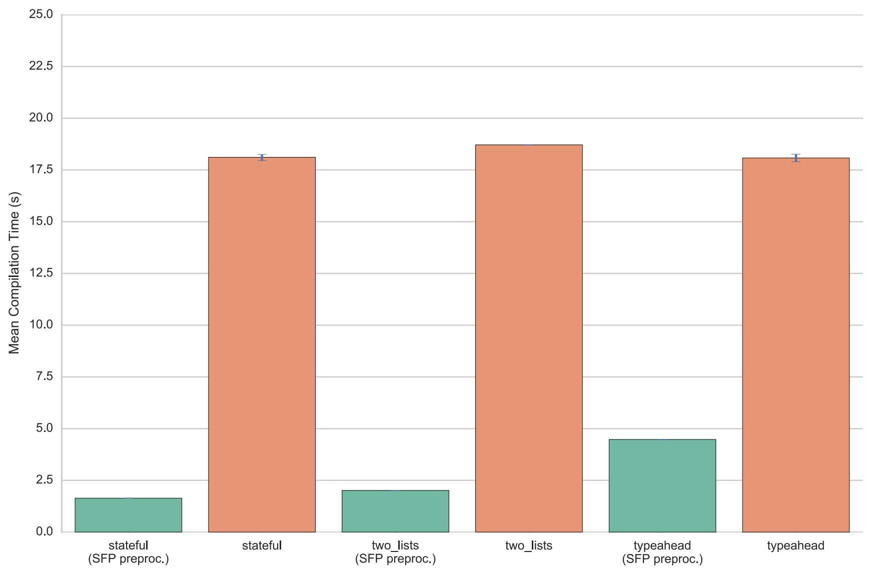


Figure 6.5: Om example projects compilation times (in seconds), part 1





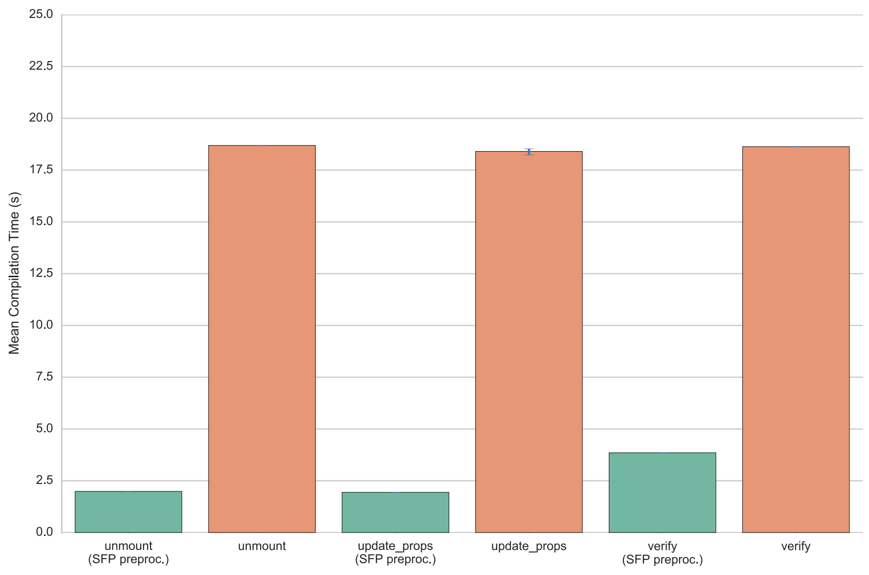


Figure 6.6: Om example projects compilation times (in seconds), part 2

Project Content Persistence across Different Versions. We looked at how project content persists across different project versions and found that a large portion of source code content remains in two or more versions. As project versions change, client code that depends on that particular project needs to manage these changes. Client code may only use parts of the external project that did not change across its versions and still needs to handle external project version changes. In SFP environments, we know precisely content associated with each version and track only what client code depends on.

Code Distribution Effects on the Compilation Pipeline. We found that unused code is distributed in conventional programming environments, so moved to investigate the effects it has on the compilation pipeline. The three sample projects of Reagent were fairly self-contained and only depended on several functions of Reagent (and their transitive dependencies). Given that, the compilation time with conventional programming environment code distribution of unused code was about 0.23-0.32x higher than when only depending on used code in the SFP variants.

The sample projects of om contained more dependencies – for example, projects "mouse", "counters", and "sortable" used core.async. The om project itself brings additional dependencies, such as transit-cljs (and related transit-clj, transit-java, and jackson-core). The transit-cljs related code, however, was never used in the sample projects. The compilation time with conventional programming environment code distribution of unused code was up to 11.04x higher than when only depending on used code in the SFP variants.

The SFP variants were actually including some portions of unused code due to the limitations of our cross-reference static analysis. One example are top level expressions in "om.dom" generated by the gen−react−dom−fns macro (which generates a **defn** expression for every HTML tag). Given that, even though only some of these generated expressions are used, all of them were included even in the SFP variants. Despite this restriction to only static code dependencies, the SFP variants still improved the initial compilation speed in all sample client code programs.

6.4.2 Implications

We compared conventional and Search-focused Programming (SFP) environments with the respect to existing source code repositories. Given that many other existing programming language environments contain similar elements as the one of ClojureScript, we expect they follow similar trends of transitive code usage. The outcomes from our comparison are potentially transferable to other programming language environments.

We gained some insights into what unused code may be distributed in conventional programming environments. One example is helper code used in the content of local test suites. We observed that distributing unused code causes adverse effects on the compilation pipeline.

The outcomes from our study could aid design and development of programming environment infrastructures. One option would be to adopt parts of what SFP proposes. For example, the community repository storage could be changed to track cross-reference dependencies and allow their transitive retrieval. Such change could be done, while the existing user interface is partially preserved, as illustrated by the command-line interface prototype in Chapter 5. The other option would be to develop tools and user interfaces that would aid project maintainers, e.g. by giving a global overview of their exposed code usage and providing hints for project restructuring.

Given that we conducted our experiments on the entire community repository of projects, we indirectly examined that SFP can work at the scale of existing open source code. It was not our primary goal, so our comments in that regard are limited observations. In general, without compressions or optimizations, the total size of all artifacts was in tens of GBs. Given it had duplicate definitions, it was sufficient to use Datomic on top of standard relational database backends at this scale. The only experienced issues were sometimes with recursive rules at a large scale: in the used versions (0.9.5530 and earlier) during experiments, we had to reorder where clauses or split queries to help the underlying database engine. These steps forced the engine not to compute too many unused intermediate results. Overall, these steps did not present any fundamental challenges and experienced issues were orthogonal to our goals. Detailed stress testing experiments that compare different storage backends would be desirable for implementations beyond the proof of concept prototype. Different Datalog engines were used and compared in the area of static code analysis [84] where more complex recursive rules were executed against a dataset recording statement-level metadata as source code relations. These results suggest that mature Datalog engines, such as LogicBlox[2], can optimize and efficiently execute rules for static code analysis, so data management workloads from programming environments should not present non-overcomeable challenges.

6.4.3 Limitations

We outline several limitations that may affect our results. Firstly, the used cross-reference analysis was restricted to static code dependencies. This restriction excluded capturing dependencies that are introduced at runtime, via macro expansions at compile time or via other languages (Clojure, JavaScript, Java). Given this restriction, our results should not be interpreted as absolute.

When looking at project content persistence across different versions, differences in how version changes are expressed may affect the results. For example, there may be new version releases that only update test suites or non-ClojureScript code-related dependencies. Semantics of how such changes are expressed in version number changes may result in different numbers of artifacts.

[2]`http://www.logicblox.com`

Lastly, the dataset was mainly collected from the default community repository. Other sources could potentially provide a more complete view for the existing code comparison in conventional and SFP environments. For the usage and persistence research questions, the community repository, however, provided a sufficiently large dataset for our study. For the effects research question, the sample projects may limit the scope of the results. The purpose of sample projects is to illustrate the usage of a particular library, so their source code and dependency structure may not match those of whole applications. Even with these restrictions, our experiments allowed us to examine the effects of code distribution on the compilation pipeline. As mentioned in our discussion, given all restrictions, our findings should be interpreted as estimates (rather than absolute) of the overall existing code properties in programming environments.

6.5 Summary

Code distribution in SFP environments can be applicable with existing source code. In the context of one large community repository, we found that a substantial portion of exposed code in the five most depended on projects was not statically used in other projects that depended on them. Beyond that, a substantial portion of exposed code in the five most depended on projects persisted across two or more of their versions. Lastly, for the sample projects of Reagent and om, code distribution in conventional programming environments resulted in 0.23x to 11.04x slower initial compilation speed than with SFP variants.

We can conclude that SFP environments could potentially scale to existing code repositories. In addition to that, our results suggest that ideas from SFP environments could bring improvements in different factors, such as initial compilation speed, when employed in existing programming environment infrastructures.

This page is intentionally left blank.

Chapter 7

Conclusions and Future Work

In this chapter, we draw overall conclusions and summarize main findings from the thesis. We then move onto describing potential extensions and future directions that can follow from this work.

7.1 Conclusions

Programming environments encompass the life cycle of software libraries and applications. They involve managing and processing code-related metadata. They include metadata that facilitates access to external code, version control, package and build management, continuous integration, documentation, and many other kinds. In conventional programming environments, each kind of metadata carries its own specific storage format and access method. This non-uniform approach of managing resources in conventional programming environments brings certain challenges.

The first set of challenges comes from the lack of globally unique persistent identifiers for code entities. Historically, the access to external code is name-addressable and resembles the file system hierarchy. User-chosen names, however, do not yield persistent and globally unique identifiers. Naturally, name conflicts may arise among definitions from different libraries or even the same library of different versions. In addition to that, the lack of persistent globally unique identifiers prevents associating code and metadata. Finally, there is an implicit context dependence (user-chosen identifiers may mean different things at different times or locations) and coarse-grained code distribution where unused definitions may be included.

With these challenges in mind, we proposed a design for programming environments, named Search-focused Programming. Search-focused Programming environments are composed of the following features:

1. Shared code and related metadata entities are assigned globally unique persistent identifiers. The identifier assignment is deterministic and only relies on static inputs.

2. Given that individual code fragments have globally unique identifiers, code dependencies can be tracked and distributed at a finer granularity.

3. Code and related metadata are uniformly stored in a deductive database.

4. With the uniform storage, two ways for accessing it are present. One is through a declarative query language for aggregated retrieval; the other is reading and writing records at the level of transactions.

5. Shared code is stored at a fine granularity; any namespace metadata that references it is stored separately from it. Due to this separation, the code that we interact with is separate from the code that we share. The interaction can be achieved in different ways.

In order to demonstrate the last point, we implemented two proof of concept prototypes. These prototypes show two directions in which code interaction can be realized. The first direction presents a command-line interface that is akin to conventional programming environments. The interaction is realized through bulk processing local source code files and generating corresponding queries or transactions. The second direction comes with a web-based interactive interface. The interaction happens by directly looking up external code and manipulating it in a structure-driven editor interface, i.e. with more flexibility and without a bulk processing of layered naming structures. Both of these proof of concept prototypes are built on top of the same infrastructure, and hence could interchange code and metadata.

Lastly, in order to examine potential effects of SFP on existing source code bases, we conducted experiments on a popular community project repository. These experiments provided a comparison of project-level and definition-level dependency tracking. We found out that source code of top five most popular projects was never fully utilized by libraries that declared dependencies on them. A substantial fraction of the same code in these projects remained in several of their versions. From these results, it is possible that we may depend on a fraction of a library that remains the same across different versions, while we need to maintain project-level dependencies and potentially deal with version conflicts. Finally, with sample client code from two of the projects, we examined the effects of project-level and definition-level dependencies on cold start compilation speed. We observed a speed-up in compiling variants where dependencies were taken from definition-level usage. This result suggests one possible merit that could be potentially applied in programming environments.

In summary, the overall contributions of this thesis are as follows:

- It introduces the requirements and design for Search-focused Programming (SFP) environments. Each topic, from issues concerning globally unique identifiers to uniform data access, lays foundations for programming environment interactions in a distributed setting.
- It presents this framework while considering features and programming environment of an existing programming language, ClojureScript. Given similarities in other programming environments, work from this thesis can be applicable in other programming environments.
- It develops a proof-of-concept implementation of a command-line inter-

face which is akin to interactions in conventional programming environments.

- It develops a proof-of-concept implementation of a web-based interactive interface which allows more direct and finer interactions with external code.

- It shows a potential migration path from infrastructures of conventional programming environments. Both of these proof-of-concept implementations show different directions and flexibility in which Search-focused Programming (SFP) environments can be realized. While different from each other, they are compatible and can share code with each other, which presents a potential migration path from existing programming environments.

- It analyses an existing popular community project repository and infers comparisons between different levels of granularity in dependency tracking. Given tracking dependencies on a definition-level may bring merit on its own, conventional programming environments could benefit from adapting some elements of Search-focused Programming (SFP) environments.

We have summarized the outcomes of work presented in this thesis. Overall, SFP aims to store and represent programming environment entities and relationships uniformly and provide a deterministic retrieval mechanism. With this aim in mind, this thesis lays foundations for future work which we describe in Section 7.2.

7.2 Future Work

In this section, we outline several possible directions for future work. Section 7.2.1 describes potential extensions to the query language used for interacting with the stored code and metadata. Section 7.2.2 looks at the possibility of making the environment more resilient in a distributed setting. Section 7.2.3 examines challenges associated with declaring dependencies at the granularity of expressions. Section 7.2.4 describes the potential application in using metadata for generative metaprogramming. Section 7.2.5 looks at some issues related to the runtime setting (rather than the static compile or load time setting). Section 7.2.6 lists several other topics, such as extensions to other programming languages.

7.2.1 Query Language

Datalog as a logical query language offers a predictable performance of its execution, and has been widely studied and used in various domains [85]. One drawback is limited expressiveness, but different extensions exist. Datomic's variant of Datalog is itself an extension that adds aggregation, negation, and other features. The extensions we consider for future work are in different directions. Evita Raced [86] and MetaLogiQL [87] explored metacompilation of Datalog dialects – that itself presents an interesting direction for a declarative lightweight macro system. We expand on several ideas in that direction (not limited to Datalog metacompilation) in Sections 7.2.3 and 7.2.4.

We can expand to query languages that operate in a distributed setting. An example is WebdamLog [88]. It is a distributed Datalog-style language which supports aggregation, access control, and rule delegation, i.e. assigning evaluation tasks to different network nodes (peers). Rule delegation and access control are aspects that may benefit programming environments:

- We can assume multiple "search spaces" where metadata may differ – e.g. labels attached to source code definitions. With this assumption, one requirement for the query language is the ability to express merging of different data sources. WebdamLog may be suitable for this application, as it allows bodies of query rules to refer to relations on remote peers and to use variables in place of relations or peers. This flexibility would, for example, allow expressing automated conflict resolution where labels from one data source are preferred over others.

- For the prototype, we did not consider access control as a requirement. Even with open source code, access control is an important consideration for the full implementation. WebdamLog's formal semantics includes annotations for access control propagation and a set of rules for each peer that enable implementing an access control policy. This feature presents a more principled approach towards access control in the query language.

We can also expand to query languages with expressiveness comparable with general purpose programming languages. These more expressive query languages can facilitate more complex code processing in programming environments:

- They can enable more complex rules related to metaprogramming, such as inheritance, ML-style modules or some applications mentioned in Section 7.2.4.
- They can enable conceptual reasoning with knowledge base ontologies. Some preliminary work of this kind has been done in the experimental Nava language [15].

One uncertainty when it comes to higher expressiveness is the question of scalability. For the sake of large code bases, it may be necessary to have two levels of query languages: one with a predictable performance of its execution that is executed against the entire code base, and one with a higher expressiveness that is executed against a view resulting from the first query's execution.

7.2.2 Federated or Decentralized Source

The question of consensus and infrastructure ownership presents another challenge for future work. With mutable labels, multiple sources of truth could potentially exist: in one search space, we can attach a certain label to shared code; in a different search space, we can remove that label. Two possible approaches can deal with this scenario:

- *Federated*: Federated sources follow what is a commonplace scenario where multiple sources of truth exist. With that assumption, various issues open up, such as discovery and integration of sources. Some of these issues have been addressed in the context of semantic web and biomedical data [89]. Querying federated sources would require a language support, which is discussed in Section 7.2.1.

- *Decentralized*: If we assume a single source of truth, it does not necessarily imply a centralized storage infrastructure. With several distributed systems projects we mentioned in Chapter 2, we could assume a single decentralized network as a storage mechanism. Even with decentralized storage, integration and querying are not solved problems. It is possible that, despite decentralized storage, issues from federated sources would need to be considered in some form, because a global consensus may be hard to reach.

7.2.3 Granularity of Dependencies

One natural area for future work is a different granularity of tracking dependencies on external code. This is mainly at the level of data types, but it could possibly extend to control flow constructs. A finer granularity would enable more precise dependency tracking – e.g. given an object, we could state we only depend on a subset of its methods. Different challenges exist with this proposal.

First, some challenges circle around the implementation. One natural way would be creating abstract syntax trees as Merkle hash trees [90] where each non-leaf node has an associated hash label computed from labels or values of its child nodes. Using abstract syntax trees for assigning identifiers has inherent advantages and disadvantages, as discussed in Chapter 4.

Using more fine-grained tracking becomes more coupled with the used programming language's semantics. With complex data types, the case may be that even seemingly unused portions of code may have load time side-effects that are required for correct behaviour. One other issue is that existing languages may not provide support for structural reasoning (i.e. to state that input requires only a subset of some structure). For dependency tracking, not all identifiers may refer to identifiers of "meaningful" external dependencies.

With the last point, this finer granularity may be more useful for other applications. One example is code coverage computation where the result could refer directly to traces by their identifiers rather than offsets from top-level definitions. Some of these applications may pose additional requirements on the identifier assignment. One application scenario is if we wanted to lookup similar code to our client code and check if similar code had any reported or fixed bugs. With such scenario in mind, we may need to introduce one additional requirement on the identifier assignment which is that similar code should be assigned similar identifiers. This requirement could be met by fingerprinting [91] techniques.

A finer granularity of dependencies may be applied in programming language feature evolution. Metadata could have a form of annotations at given points of abstract syntax trees and annotations processors could be implemented in order to support new features.

7.2.4 Configurations and Generative Metaprogramming

One potential application where SFP may be expanded is with storing program configuration metadata and generating variants based on these annotations. In the Clojure-ClojureScript ecosystem, the natural choice is what is achieved with "reader conditionals". They are syntax annotations for reader-time macros that parse an expression based on the platform, so that one can write the same code for all platforms (JavaScript, Java, or Common Language Runtime) in one place. In SFP environments, plain text files may be used for interaction, but not for the primary storage of the shared source code. With that, variants for different platforms can be stored separately with metadata annotating their respective platforms. Viewing them or automatically generating new variants that use them can be achieved in a similar way to annotative variation management [92].

With more complex build settings, expressiveness of meta-level operations in annotative approaches may not be sufficient. More complex build settings may involve a large number of variables with various relationships among them. These relationship may state that some variables preclude or imply others. In order to capture that only certain configurations are valid, we may need to have program code entities stored at a finer granularity and to use a more complex query language, such as a full Prolog dialect. Both of these requirements present challenges on their own, as discussed in Sections 7.2.3 and 7.2.1.

7.2.5 Change Management, Runtime Lookup and Dependencies

Chapter 5 did not focus on version management and associated metadata. When sharing code, change management has an important role. In conventional programming environments, change management revolves around numbered version releases and responsibilities of upstream and downstream developers. Different communities adapt different standards for change management

[69]: some assume it is a client's responsibility to upgrade libraries his or her code depends on, some require the library authors to fix client code, others may spread the responsibility among multiple parties. All these approaches can co-exist in SFP environments. The difference is the granularity of tracking changes.

This thesis focused on static, i.e. compile or load time, resolution. Similar techniques, however, could be adapted to runtime. In combination with change management, this future work direction can find applications in hot code reloading. After detecting a change (by periodically monitoring the transaction log, as described in Chapter 4), the system may try to upgrade relevant parts. Upgrade may be conditioned by a presence of certain metadata – e.g. that it was flagged as a security fix by several maintainers, or that successful passing of relevant test suites was recorded.

7.2.6 Beyond a Prototype

Some areas offer opportunities for future work in extensions beyond the proof of concept prototypes presented in Chapter 5. The command-line interface prototype focused on processing project manifests, code and its naming structures, but did not work with other metadata. Some types of metadata, given they are bundled with source code, may not present research challenges for processing; others may require additional analysis that connects it with shared source code.

We focused on a single language and a platform in Chapter 5. Supporting multiple languages and platforms can take different forms. One possibility was outlined in Section 7.2.4 where the main parts of the source language are shared, but target platforms are different. When we look beyond that, with different source languages, we may raise various questions, such as how identifiers should be assigned. Should different source languages have different identifier assignment schemes, or provided they target the same platform (such as Clojure and Java), should their identifiers be assigned based on the target abstract syntax or bytecode?

Additionally, as discussed in Chapter 4, globally unique identifier assignment schemes need many considerations when it comes to existing programming languages. One issue in environments with multiple languages arises with their name-dependent features. Presumably, the used language may be one of the extended inputs for identifier assignment. Chapter 4 sketched out the possibility of having "canonical specifications" as a general form of input. If such specifications can be written independently of the underlying programming languages (e.g. using an interface description language), they could potentially serve as an input for these cross-language identifier assignment schemes.

The web-based interface in Chapter 5 centred around displaying and editing code. Extending it for displaying further metainformation may present additional work. If change management is present (as discussed in Section 7.2.5), debugging and ways for interrogating external code lineage may be worth exploring.

Lastly, in terms of extensions beyond the proof of concept prototype, scalability and security are requirements that need to be addressed in a greater depth. The system implementation that addresses such concerns could possibly uncover further research opportunities in this regard – for instance, optimizations of prevalent retrievals and operations in programming environments.

More complete environment implementations would also enable exploring research questions related to development productivity. Randomized controlled trials could contrast development experience in conventional programming environments and in various implementations of SFP (a set of command-line tools, existing IDE plugins, interactive web-based editor with interactive search functionality, etc.). Outside these experiments, more complete implementations could serve as case studies of tool development experience in different programming environment architectures.

This page is intentionally left blank.

Glossary

API	Application Programming Interface. 30, 38, 41, 43, 122
AST	Abstract Syntax Tree. 50, 51, 122
DOI	Digital Object Identifiers. 16, 17, 48, 122
DOM	Document Object Model. 81, 122
EDN	Extensible Data Notation. 30, 41, 122
HTML	HyperText Markup Language. 81, 85, 104, 122
HTTP	Hypertext Transfer Protocol. 30, 36, 92, 122
JSON	JavaScript Object Notation. 38, 122
REPL	Read-Eval-Print-Loop. 30, 82, 84, 96, 122
REST	Representational State Transfer. 30, 38, 40, 41, 122
SFP	Search-focused Programming. 8–13, 15–24, 26, 27, 44, 45, 53, 66–70, 80, 82, 86, 89–91, 95, 96, 98, 100, 101, 104, 105, 107, 110–112, 117–119, 122
URN	Uniform Resource Name. 46, 122
UUID	universally unique identifier. 48, 122
VCS	version control system. 4, 30, 54, 60, 122

XML Extensible Markup Language. 30, 122

Appendix A

Sample User Code

Listing A.1: Example test code

```
(ns com.example-utils-test
  (:require [cljs.test :refer-macros [deftest is testing
      run-tests]]
            [com.example-utils :refer [right-pad]]))

(deftest a-test
  (testing "Hello world padding."
    (is (= (right-pad "hello world" 14 \.) "hello world
       ..."))))

(defn exit [code]
  (js/setTimeout #(.exit js/phantom code) 0)
  (aset js/phantom "onError" (fn [])))

(defmethod cljs.test/report [:cljs.test/default :
   end-run-tests] [m]
  (if (cljs.test/successful? m)
    (exit 0)
    (exit 1)))

(try
  (enable-console-print!)
  (run-tests)
  (catch js/Object e
    (print (.-stack e))
    (exit 1)))
```

Listing A.2: Example continuous integration configuration

```
image: clojure:lein-2.7.0

before_script:
  - apt-get update -y
  - apt-get install -y bzip2 wget libfreetype6
    libfontconfig
  - wget https://bitbucket.org/ariya/phantomjs/downloads
    /phantomjs-2.1.1-linux-x86_64.tar.bz2
  - tar -xjf phantomjs-2.1.1-linux-x86_64.tar.bz2
  - rm *.bz2
  - mv phantomjs-* /usr/share/phantomjs
  - ln -s /usr/share/phantomjs/bin/phantomjs /usr/bin/
    phantomjs
  - lein deps

test:
  script:
  - lein cljsbuild test
```

Bibliography

[1] N. Wirth, "Program Development by Stepwise Refinement," *Communications of the ACM*, vol. 14, no. 4, pp. 221–227, April 1971.

[2] S. Chacon and B. Straub, "Git Internals," in *Pro Git*. Apress, 2014.

[3] A. Gerrand, "Organizing Go Code," 2012. [Online]. Available: https://blog.golang.org/organizing-go-code

[4] A. Decan, T. Mens, M. Claes, and P. Grosjean, "When GitHub Meets CRAN: An Analysis of Inter-Repository Package Dependency Problems," in *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, March 2016, pp. 493–504.

[5] C. Ireland, D. Bowers, M. Newton, and K. Waugh, "A Classification of Object-Relational Impedance Mismatch," in *Proceedings of the 1st International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA)*, March 2009, pp. 36–43.

[6] R. F. Strout, "The Development of the Catalog and Cataloging Codes," *The Library Quarterly*, vol. 26, no. 4, pp. 254–275, October 1956.

[7] N. Paskin, "Toward Unique Identifiers," *Proceedings of the IEEE*, vol. 87, no. 7, pp. 1208–1227, July 1999.

[8] "DataCite," 2009. [Online]. Available: https://www.datacite.org/

[9] M. B. Jones, C. Boettiger, A. Cabunoc Mayes, A. Smith, P. Slaughter, K. Niemeyer, Y. Gil, M. Fenner, K. Nowak, M. Hahnel, L. Coy, A. Allen, M. Crosas, A. Sands, N. C. Hong, P. Cruse, D. Katz, and C. Gobl, "CodeMeta: An Exchange Schema for Software Metadata," 2016. [Online]. Available: https://github.com/codemeta/codemeta

[10] J. Bloch, *Effective Java (Second Edition).* Prentice Hall PTR, 2008.

[11] A. Bryant, A. Catton, K. De Volder, and G. C. Murphy, "Explicit Programming," in *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD)*, ser. AOSD '02. ACM Press, April 2002, pp. 10–18.

[12] T. Kielmann, "prom: A flexible, Prolog-based make tool," TU Darmstadt, Tech. Rep., 1991.

[13] S. Pearce, "Gerrit Code Review," 2009. [Online]. Available: https://www.gerritcodereview.com/

[14] R. Lämmel, "Software Chrestomathies," *Science of Computer Programming*, vol. 97, no. P1, pp. 98–104, January 2015.

[15] H. Samimi, C. Deaton, Y. Ohshima, A. Warth, and T. Millstein, "Call by Meaning," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, ser. Onward! '14. ACM Press, October 2014, pp. 11–28.

[16] I. D. Craig, "Blackboard Systems," *Artificial Intelligence Review*, vol. 2, no. 2, pp. 103–118, June 1988.

[17] C. Beeri, P. A. Bernstein, and N. Goodman, "A Sophisticate's Introduction to Database Normalization Theory," in *Proceedings of the 4th International Conference on Very Large Data Bases (VLDB)*, vol. 4. VLDB Endowment, September 1978, pp. 113–124.

[18] J. Bobbio et al., "reproducible-builds.org," 2015. [Online]. Available: https://reproducible-builds.org/

[19] F. Mancinelli, J. Boender, R. Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen, "Managing the Complexity of Large Free and Open Source Package-Based Software Distributions," in *Proceedings of 21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '06. IEEE, September 2006, pp. 199–208.

[20] E. Dolstra, "The Purely Functional Software Deployment Model," Ph.D. dissertation, Utrecht University, January 2006.

[21] L. Courtès, "Functional Package Management with Guix," in *Proceedings of the 6th European Lisp Symposium (ELS).* European Lisp Scientific Activities Association, June 2013.

[22] A. Heydon, *Software Configuration Management using Vesta.* Springer, 2006.

[23] S. Bloehdorn, O. Görlitz, S. Schenk, and M. Völkel, "Tagfs – Tag Semantics For Hierarchical File Systems," in *Proceedings of the 6th International Conference on Knowledge Management (I-KNOW).* Springer, September 2006, pp. 6–8.

[24] M. Seltzer and N. Murphy, "Hierarchical File Systems are Dead," in *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS)*, ser. HotOS'09. USENIX Association, 2009.

[25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A Scalable Content-addressable Network," in *Proceedings of the 2001 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, vol. 31, no. 4. ACM Press, August 2001, pp. 161–172.

[26] J. Benet, "IPFS - Content Addressed, Versioned, P2P File System," July 2014. [Online]. Available: http://arxiv.org/abs/1407.3561

[27] G. Gonzales, "Annah." [Online]. Available: https://github.com/Gabriel439/Haskell-Annah-Library

[28] C. Böhm and A. Berarducci, "Automatic Synthesis of Typed Λ-programs on Term Algebras," *Theoretical Computer Science*, vol. 39, pp. 135–154, August 1985.

[29] C. Ball, "GitTorrent." [Online]. Available: https://github.com/cjb/GitTorrent

[30] S. Mackenzie and D. Michiels, "Fractalide." [Online]. Available: https://github.com/fractalide/fractalide

[31] J. P. Morrison, *Flow-based programming: A New Approach to Application Development*. J.P. Morrison Enterprises, 2010.

[32] TUNES Project, "TUNES OS/Language Project." [Online]. Available: http://tunes.org/

[33] D. Barbour, "Awelon Blue," 2012. [Online]. Available: https://awelonblue.wordpress.com/

[34] P. Chiusano, "Unison," 2014. [Online]. Available: http://unisonweb.org/

[35] E. Hajiyev, M. Verbaere, and O. de Moor, "CodeQuest: Scalable Source Code Queries with Datalog," in *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*, ser. Lecture Notes in Computer Science, D. Thomas, Ed., vol. 4067. Springer, July 2006, pp. 2–27.

[36] M. Verbaere, E. Hajiyev, and O. De Moor, "Improve Software Quality with SemmleCode," in *Proceedings Companion to the 22nd ACM SIG-*

PLAN Conference on Object Oriented Programming Systems and Applications (OOPSLA). ACM Press, October 2007, pp. 880–881.

[37] S. Yegge et al., "Google Kythe." [Online]. Available: http://www.kythe.io/

[38] R. Vanbrabant, *Google Guice: Agile Lightweight Dependency Injection Framework.* Apress, 2008.

[39] L. D. Ngan and R. Kanagasabai, "Semantic Web Service Discovery: State-of-the-art and Research Challenges," *Personal and Ubiquitous Computing*, vol. 17, no. 8, pp. 1741–1752, September 2012.

[40] R. Anthony, *Systems Programming: Designing and Developing Distributed Applications.* Elsevier Science, 2015.

[41] G. Little and R. C. Miller, "Keyword Programming in Java," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '07. ACM Press, 2007, pp. 84–93.

[42] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen, "CodeHint: Dynamic and Interactive Synthesis of Code Snippets," in *Proceedings of the 36th IEEE/ACM International Conference on Software Engineering (ICSE)*, ser. ICSE 2014. ACM Press, July 2014, pp. 653–663.

[43] R. Potvin and J. Levenberg, "Why Google Stores Billions of Lines of Code in a Single Repository," *Communications of the ACM*, vol. 59, no. 7, pp. 78–87, June 2016.

[44] G. Durham and S. Agarwal, "Scaling Mercurial at Facebook." [Online]. Available: https://code.facebook.com/posts/218678814984400/scaling-mercurial-at-facebook/

[45] A. F. Case, "Computer-aided Software Engineering (CASE): Technology for Improving Software Development Productivity," *SIGMIS Database*, vol. 17, no. 1, pp. 35–43, September 1985.

[46] D. B. Leblang and R. P. Chase Jr., "Computer-Aided Software Engineering in a Distributed Workstation Environment," in *Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE)*, ser. SDE 1. ACM Press, April 1984, pp. 104–112.

[47] F. Asplund and M. Törngren, "The Discourse on Tool Integration Beyond Technology, a Literature Survey," *Journal of Systems and Software*, vol. 106, no. C, pp. 117–131, August 2015.

[48] T. Schafer, M. Eichberg, M. Haupt, and M. Mezini, "The SEXTANT Software Exploration Tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 753–768, September 2006.

[49] B. de Alwis and G. C. Murphy, "Answering Conceptual Queries with Ferret," in *Proceedings of the 30th IEEE/ACM International Conference on Software Engineering (ICSE)*, ser. ICSE '08. ACM Press, May 2008, pp. 21–30.

[50] D. Garlan, "Views for Tools in Integrated Environments," in *An International Workshop on Advanced Programming Environments.* Springer, June 1986, pp. 314–343.

[51] M. C. Chu-Carroll, J. Wright, and D. Shields, "Supporting Aggregation in Fine Grained Software Configuration Management," *SIGSOFT Software Engineering Notes*, vol. 27, no. 6, pp. 99–108, November 2002.

[52] S. P. Reiss, "The Desert Environment," *ACM Transactions on Software Engineering and Methodology*, vol. 8, no. 4, pp. 297–342, October 1999.

[53] G. E. Kaiser, S. E. Dossick, W. Jiang, and J. J. Yang, “An Architecture for WWW-based Hypercode Environments,” in *Proceedings of the 19th IEEE/ACM International Conference on Software Engineering (ICSE)*, ser. ICSE ’97. ACM Press, May 1997, pp. 3–13.

[54] W. Teitelman, J. Goodwin, and D. G. Bobrow, *Interlisp-D Reference Manual Volume II: Environment.* Xerox Palo Alto Research Center, October 1985.

[55] C. Simonyi, “The Death Of Computer Languages, The Birth of Intentional Programming,” Tech. Rep. MSR-TR-95-52, September 1995.

[56] M. Völter and V. Pech, “Language modularity with the MPS language workbench,” in *Proceedings of the 34th IEEE/ACM International Conference on Software Engineering (ICSE)*, ser. ICSE 2012, IEEE. IEEE, June 2012, pp. 1449–1450.

[57] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker, *Pharo by Example.* Square Bracket Associates, 2010.

[58] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola Jr, “Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments,” in *Proceedings of the 32nd IEEE/ACM International Conference on Software Engineering (ICSE)*, ser. ICSE 2010. ACM Press, May 2010, pp. 455–464.

[59] S. P. Reiss, J. N. Bott, and J. J. Laviola, “Plugging In and Into Code Bubbles: The Code Bubbles Architecture,” in *Proceedings of the 2nd International Workshop on Developing Tools as Plug-Ins (TOPI).* IEEE, June 2012, pp. 55–60.

[60] K. De Volder, “Type-Oriented Logic Meta Programming,” Ph.D. dissertation, Vrije Universiteit Brussel, 1998.

[61] K. Klose and K. Ostermann, “Modular Logic Metaprogramming,” in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, ser. OOPSLA ’10. ACM Press, October 2010, pp. 484–503.

[62] D. K. Layer and C. Richardson, “Lisp Systems in the 1990s,” *Communications of the ACM*, vol. 34, no. 9, pp. 48–57, September 1991.

[63] B. De Bus, D. Kästner, D. Chanet, L. Van Put, and B. De Sutter, “Post-pass Compaction Techniques,” *Communications of the ACM*, vol. 46, no. 8, pp. 41–46, August 2003.

[64] M. McGranaghan, “ClojureScript: Functional Programming for JavaScript Platforms,” *IEEE Internet Computing*, vol. 15, no. 6, pp. 97–102, November 2011.

[65] R. Hickey, “The Clojure Programming Language,” in *Proceedings of the 2008 Symposium on Dynamic Languages (DLS)*, ser. DLS ’08. ACM Press, July 2008, p. 1.

[66] P. Hagelberg, “Leiningen.” [Online]. Available: https://leiningen.org/

[67] The Apache Software Foundation, “Maven Introduction to the Dependency Mechanism.” [Online]. Available: https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

[68] GitLab Inc., “GitLab.” [Online]. Available: https://www.gitlab.com/

[69] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, “How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems,” in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, ser. FSE 2016. ACM Press, November 2016, pp. 109–120.

[70] P. Leach, M. Mealling, and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace," RFC 4122, July 2005.

[71] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, 1995.

[72] J. Pombrio and S. Krishnamurthi, "Hygienic Resugaring of Compositional Desugaring," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2015, September 2015, pp. 75–87.

[73] Protocol Labs Inc., "multihash," 2016. [Online]. Available: https://github.com/multiformats/multihash

[74] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears, "Dedalus: Datalog in Time and Space," in *Lecture Notes in Computer Science*, vol. 6702 LNCS. Springer, 2011, pp. 262–281.

[75] F. Cristian, H. Aghili, R. Strong, and D. Dolev, "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement," *Information and Computation*, vol. 118, no. 1, pp. 158 – 179, April 1995.

[76] D. Betts, J. Dominguez, G. Melnik, F. Simonazzi, and M. Subramanian, *Exploring CQRS and Event Sourcing: A Journey into High Scalability, Availability, and Maintainability with Windows Azure.* Microsoft Patterns & Practices, February 2013.

[77] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A Distributed Messaging System for Log Processing," in *Proceedings of ACM SIGMOD 6th International Workshop on Networking Meets Databases (NetDB)*, ser. NeDB'11. ACM Press, June 2011, pp. 1–7.

[78] Event Store LLP, "Event Store." [Online]. Available: https://geteventstore.com/

[79] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases: The Logical Level*. Addison-Wesley, 1995.

[80] Cognitect Inc., "Datomic," 2012. [Online]. Available: http://www.datomic.com/

[81] T. Tauber, "When Importless Becomes Meaningful: A Call for Tag-based Namespaces in Programming Languages," in *Proceedings of the Companion Publication of the 2014 ACM SIGPLAN conference on Systems, Programming, and Applications: Software for Humanity (SPLASH)*, ser. SPLASH '14. ACM Press, October 2014, pp. 29–31.

[82] T. Tauber and B. C. d. S. Oliveira, "Modular Architecture for Code and Metadata Sharing," in *Proceedings of the 15th International Conference on Modularity (MODULARITY)*, ser. MODULARITY 2016. ACM Press, March 2016, pp. 106–117.

[83] M. Haverbeke, "CodeMirror." [Online]. Available: http://codemirror.net/

[84] A. I. Antoniadis, "Declarative Points-To Analysis on Different Datalog Engines," Master's thesis, National and Kapodistrian University of Athens, 2014.

[85] P. Barceló and R. Pichler, Eds., *Datalog in Academia and Industry*, ser. Lecture Notes in Computer Science. Springer, 2012, vol. 7494.

[86] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis, "Evita Raced: Metacompilation for Declarative Networks," in *Proceedings of the VLDB Endowment*, vol. 1, no. 1. VLDB Endowment, August 2008, pp. 1153–1165.

[87] T. J. Green, D. Olteanu, and G. Washburn, "Live Programming in the LogicBlox system," in *Proceedings of the VLDB Endowment*, vol. 8, no. 12. VLDB Endowment, August 2015, pp. 1782–1791.

[88] V. Zaychik Moffitt, J. Stoyanovich, S. Abiteboul, and G. Miklau, "Collaborative Access Control in WebdamLog," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. ACM Press, May 2015, pp. 197–211.

[89] M. D. Wilkinson, B. Vandervalk, and L. McCarthy, "The Semantic Automated Discovery and Integration (SADI) Web service Design-Pattern, API and Reference Implementation," *Journal of Biomedical Semantics*, vol. 2, no. 1, pp. 5–23, October 2011.

[90] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *Advances in Cryptology — CRYPTO '87*, ser. Lecture Notes in Computer Science, C. Pomerance, Ed., vol. 293. Springer, 1988, pp. 369–378.

[91] M. Chilowicz, E. Duris, and G. Roussel, "Syntax Tree Fingerprinting: A Foundation for Source Code Similarity Detection," LIGM Université Paris-Est, Tech. Rep. 2009–03, September 2009.

[92] S. Chen, M. Erwig, and E. Walkingshaw, "An Error-tolerant Type System for Variational Lambda Calculus," in *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, ser. ICFP '12, 2012.