TOWARD AUTOMATIC PROGRAMMING

BY

COSMIN RĂDOI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Doctoral Committee:

Professor Grigore Roșu, Chair and Director of Research
Professor David Padua
Adjunct Professor Danny Dig
Manu Sridharan, Uber

Abstract

Programming, the act of creating and changing software source code, should be a collaborative process between humans and computers. This dissertation shows a general approach and two techniques that bring us closer to this goal. The general approach is inspired by human programmers: they learn how to transform code by looking at similar past transformation instances, then they change new code by stringing together several fine-grained transformations.

First, we give a technique for inferring abstract program transformations from concrete code examples. The transformations are expressed as formal rules in a term rewriting language with contexts, and they are inferred from examples via a novel anti-unification algorithm. For evaluation, we use the technique to successfully infer 15 JavaScript linting rules.

Second, we give a technique for searching through compositions of program transformations to satisfy a goal. The search is an evolutionary algorithm with the fitness function defined over the code and the code transformations as mutations. For evaluation, we apply the technique to the problem of automatically translating imperative, sequential programs to functional MapReduce programs. The algorithm successfully finds efficient MapReduce implementations for programs with complex indirect accesses, such as WordCount.

To Codruța and my parents.

Acknowledgments

I thank my parents for raising me to believe in myself, for nurturing my passion for mathematics, and for encouraging me to pursue computer science.

I thank my wife Codruța for being there for me through thick and thin, wherever our path takes us.

I thank my advisor Grigore Roșu for creating the $\mathbb{K}$ framework, the foundation of this dissertation, for giving me the freedom to explore various ideas, and for working together and encouraging me to develop the ideas into principled structures.

I thank the rest of my dissertation committee, Manu Sridharan, Danny Dig, and David Padua, for their feedback and insightful suggestions for improving my research. I especially thank Manu Sridharan for his trust in my ability to find and implement solutions, and for his gentle guidance throughout.

The seeds of this research have been planted during my internships at IBM Research – T.J. Watson and at Samsung Research America. I thank my other collaborators there, Stephen Fink and Rodric Rabbah from IBM, and Satish Chandra from Samsung, for giving me the opportunity to work on hard problems, and for their valuable suggestions and help over the years.

I thank my colleagues and collaborators for making research fun. In particular, I thank Semih Okur, Darko Marinov, Adrian Nistor, Yu Lin, Alex Gyori, Rajesh Karmani, Mihai Codoban, Caius Brîndescu, Loránd Szakács, Milos Gligoric, Radu Mereuță, Brandon Moore, Daejun Park, Owolabi Legunsen, Yi Zhang, Shijiao Yuwen, Michael D. Adams, Dwight Guth, Traian Șerbănuță, Manasvi Saxena, Ali Kheradmand, Xiaohong Chen, Lucas Peña, and Everett Hildenbrandt.

I especially thank Marius Minea, for showing me that research exists, Danny Dig, for teaching me how to do academic research, and Andrei Ștefănescu, for working together so well, on top of being a good friend.

I would like to thank my friends outside computer science for distracting me from research, thus keeping me sane. In particular, I thank Yulia, Luisa, David, Iulia, Monika, James, Ileana, Adina, Nisha, Maria, and Kim.

I thank IBM for the PhD Fellowship that supported a year of my work, and the National Science Foundation for the grants that supported my research over the years (CCF-1646559, CCF-1218605, and CCF-1318191).

Table of Contents

Chapter 1

Introduction

Programming, the act of creating and changing software source code, is now done almost exclusively by human programmers, with little to no assistance from computers. Computers mainly help with translating programs from high-level programming languages to lower-level machine instructions, and with optimizing these lower-level representations. Translation and low-level optimization are a good fit for computer algorithms because their input and output are well defined.

Program transformation is at the core of software development yet, paradoxically, it remains mostly a manual process. While developers do occasionally write new code from scratch, most development time is spent changing existing code in various ways, from simple bug or code style fixes to extensive refactoring or API changes. In most of these cases, developers transform programs by editing their textual representation, i.e., the code, by hand. Many of these transformations are repetitive and tedious, and thus error-prone.

Ideally, a program transformation system would aid the developer by translating their abstract intent into multiple textual changes that are applied, repetitively, across the code base. Refactoring tools are the first step toward this goal, but they are limited to a small set of predefined transformations, and are often hard to configure.

Automating program transformations is challenging. Even representing a program transformation is not trivial. A patch expressed as a textual *diff*, like those used by source version control tools, is littered with syntactic details, so it is highly unlikely it will fit any new context. Likewise, recording a sequence of AST edits and generalizing from them can be fragile [1, 2, 3]. Regular expression search-and-replace is capable of representing changes applicable in multiple places, but regular expressions have no knowledge of the underlying structure of the code (i.e., they are applied to text, not the AST) or of the context in which they are matched. Furthermore, they are notoriously hard to write, so their use is usually limited to small transformations. Scripts which manipulate the AST directly are powerful but require that the developer understands the concept of an AST, knows how to use an AST-manipulation library, and is willing

to write code in order to make a change to her program.

A principled way to manipulate programs is using term rewrite systems [4]. Term rewrite systems allow expressing powerful abstract syntax tree (AST) transformations via simple, concise rules. Unfortunately, the lack of developer-friendly tools has limited their use outside academia. Furthermore, even with easy-to-use tools, they still require that the developer to learn a term rewriting language and be willing to write rewrite rules by hand.

We propose a general approach for automating source-to-source program transformations and give two techniques for implementing it. Our focus is not on synthesizing or otherwise creating new code from scratch, but on transforming existing code.

Our approach is inspired by how developers transform code. Often the first step is to look at similar transformations (from the same codebase or even external sources, e.g., StackOverflow) in order to form a mental model of the transformation. Then the programmer applies a similar transformation to new code. The programmer strings together multiple such transformations to achieve a higher-level goal. They may skip the learning phase if they already know a particular transformation from previous experience. Finally, it is often the case the programmer does not find the right solution on the first try. In these cases, they might backtrack some transformation steps and try a different path, i.e., the programmer searches for the solution that satisfies their goal (e.g., passing all tests or fixing a bug).

Our approach for automating program transformations follows the above process, and consists of two techniques:

1. **Evolutionary Program Transformation**. In most real-world scenarios, the program that satisfies a particular goal is not reached in a single program transformation step. The goal is usually satisfied by a sequence of several different transformations. For the automated system to discover a sequence of steps that satisfies a goal, we propose a technique that searches through the possibilities using an evolutionary algorithm [5].

2. **Generalization-based Program Transformation Inference**. The evolutionary algorithm above requires, as input, a set of predefined abstract program transformations. The transformations are abstract in the sense that they are general rules applicable to new code. In particular, our transformations are rewrite rules [4] in a variant of matching logic [6, 7]. The difficulty is that writing such rules requires the programmer to learn our rewriting language, something that

most programmers might not be willing to do. To alleviate this issue, we propose a technique for automatically inferring the program transformation rules from concrete code examples. The technique is based on our novel anti-unification algorithm, which extends the approach of Alpuente et al. [8] with contexts [9] and rewrite symbols [4].

In practice, we envision the programmers using the evolutionary system to generate complex concrete program transformations that satisfy various goals, and using the inference algorithm for creating new program transformation steps from previous code examples.

1.1 Evolutionary Program Transformation

We discovered the need for an evolutionary approach to composing program transformation while working on the problem of automatically translating sequential, imperative code to functional, parallel MapReduce [10] (see Section 3.3 for details on MapReduce). Such a transformation helps by reducing costs when re-targeting legacy sequential code for MapReduce. Furthermore, a translator simplifies the process of targeting MapReduce in new programs: a developer can concentrate on sequential code, letting the translator handle the parallelization.

The most challenging part of the translation is finding a sequence of atomic transformation steps that lead to an efficient MapReduce program.

We first attempted to solve this problem in the traditional way, by trying to make a unique "good" choice between transformations. When the system was not making the right transformation choice, we tried to employ more powerful analysis to generate heuristics that work in most cases. We mostly failed on this path because we could not figure out powerful enough heuristics to obtain consistently good results.

Our proposal builds upon techniques developed in the evolutionary algorithms [5] field, and is most akin to genetic programming [11]. The main differentiating aspect from genetic programming is that program transformation mutation operators have strong, specific semantic meaning — see Section 3.4 for an example. Furthermore, there is no crossover operator in the proposed system, though integrating crossover is a possible future extension.

The evolutionary approach arose from dropping the requirement to make a unique choice at each step. We thus allowed our system to explore the space of semantically

equivalent programs obtained by applying multiple program transformation rules at each step.

We have implemented our techniques in MOLD, a tool that transforms Java programs into Scala programs. The resulting programs can be executed either on a single computing node via parallel Scala collections, or in a distributed manner using Spark, a popular MapReduce framework [12, 13]. MOLD leverages the WALA analysis framework [14] to generate Array SSA and implements a custom rewriting engine using the Kiama [15] library. In an experimental evaluation, we tested MOLD on a number of input kernels taken from real-world Java and MapReduce benchmarks. In most cases, MOLD successfully generated the desired MapReduce code, even for code with complex indirect array accesses that cannot be handled by previous techniques. To our knowledge, MOLD is the first to automatically translate sequential implementations of canonical MapReduce programs like `WordCount` into effective MapReduce programs.

1.2 Generalization-based Program Transformation Inference

This work started from the observation that it would be helpful for developers to have a tool that, given several examples, *infers* a program transformation – in this case one or more term rewriting rules. Andersen and Lawall [16, 17] have observed this need as well and they have proposed a tool for generalizing several patches into a transformation represented as a composition of rewriting rules. Unfortunately, their rewriting language is not very expressive. As a consequence, more complex transformations need to be expressed as sequences, leading to fragility (see Section 4.1).

We propose a novel approach (and accompanying system) for inferring, from code examples, program transformations expressed as term rewrite rules in a rich language with *context variables* and *associative function symbols*. Rewrite rules with *context variables* are capable of matching over arbitrary contexts, like in Felleisen and Hieb [9]. They are supported by several modern rewriting/reduction engines specialized for programming language syntax and transformation, such as PLT-Redex [18], K [19], and Spoofax [20]. The contexts allow matching a pattern anywhere within a term which, in many cases, replaces the need to have a transformation expressed as a sequence of transformations, as is the case for other approaches [1, 2, 3, 16, 17, 21, 22]. For instance, an *anywhere* context can be used to match an assignment only when appearing

anywhere within the condition of an `if` statement (see the motivating example in Section 4.1). The associative function symbols allow matching particular instructions in a block, as well as formal and actual parameters for invocations. Without these rewrite-language features, our system would not be able to express the fixes for many warning types reported by ESLint [23], the linting tool we base our benchmark suite on.

Our system takes as input several concrete code transformation example snippets (i.e., *before* and *after* code snippets) and outputs one general, developer-readable rewrite rule that generalizes all input examples. More precisely, when applied to the *before* of any of the input examples, the rule will generate precisely the corresponding *after*. Furthermore, the rule will apply in new, similar contexts.

We evaluate our approach on the problem of learning fixes for linting tool warnings, and show that it effectively infers powerful, general rewrite rules.

1.3 Program Transformations as Matching Logic Formulae

The two techniques presented in this thesis are based on expressing program transformations as term rewrite rules. In particular, the program transformations can be seen as formulae in specialized subsets of Matching Logic [7].

Matching Logic, described in more detail in Section 2, is a solution for using the operational semantics of programming languages for program verification. In Matching Logic formulae are interpreted as sets of values in the model, and the formulae are kept small using configurations and their abstraction.

This work stems from the observation that, while Matching Logic and K are meant to simplify program verification, they also provide an excellent theoretical framework and implementation for source-to-source program transformation. In particular, rewrite rules are just implications in Matching Logic, our contexts are a runtime generalization of configuration abstraction, and anti-unification is simply a disjunction of formulae.

1.4 Contributions

This dissertation proposes a general approach to program transformation and makes the following contributions:

1. **Imperative code to Lambda with `fold` via Array SSA form.** We show a way to convert array-manipulating imperative programs to a functional representation by using their Array SSA form.

2. **Evolutionary Program Transformation.** We present an evolutionary algorithm for exploring the broad space of possible transformations generated by a set of abstract program transformation rules.

3. **Indirect array accesses to `groupBy` operations.** We present a rewrite rule for effectively handling complex indirect array accesses by converting them to `groupBy` operations.

4. **Imperative programs to MapReduce.** We give a tool for translating imperative programs to functional MapReduce programs, and an experimental evaluation showing the tool's ability to handle complex input programs beyond those in previous work.

5. **Generalization algorithm for rewrite rules.** We introduce a technique and algorithm for generalization/learning of rewrite rules from multiple concrete code-change examples. The algorithm generalizes from term patterns with associative symbols, rewrite symbols, and context variables.

6. **Learning linting rules from examples.** We give an optimized implementation of the generalization algorithm together with a thorough evaluation showing the feasibility of the approach on learning fixes for linting tool warnings.

Chapter 2

Background

Matching Logic [7] is a variant of first-order logic specialized to specifying and reasoning about program structure via pattern matching.

Matching Logic emerged as a solution to using the operational semantics of programming languages for program verification. Operational semantics are easy to define and understand, but they have been perceived as too low-level to be used for program verification. The alternate solutions, Hoare [24] and dynamic [25] logics, are higher-level but they need to be defined separately and require additional effort to prove soundness with respect to the language's operational semantics. Matching Logic and its implementation, the K framework [26], overcome these obstacles by interpreting formulae as sets of values in the model, and by, respectively, using configurations and their abstraction to separate concerns and keep formulae small.

2.1 Matching Logic Syntax

We assume the reader is familiar with first-order logic, functions, and sets. Matching Logic is sorted but, as sorts are not relevant to this dissertation, we introduce an unsorted variant here.

Definition 2.1 *Let Σ be a signature of symbols, and let Var be an infinite set of variables. Matching logic Σ-**formulae**, or Σ-**patterns**, are defined inductively as:*

$$
\begin{array}{rll}
\varphi ::= & x \in Var & \textit{// variable} \\
| & \sigma(\varphi_1, ..., \varphi_n) \textit{ with } \sigma \in \Sigma & \textit{// structure} \\
| & \neg\varphi & \textit{// complement} \\
| & \varphi \wedge \varphi & \textit{// intersection} \\
| & \exists x.\varphi \textit{ with } x \in Var & \textit{// binding}
\end{array}
$$

We call Σ-**patterns** just **patterns**, or **formulae**, when Σ is clear from the context.

The syntax allows expressing meaningful patterns, and it is at the same time succinct and easy to read. The patterns are constructed by nesting variables, concrete syntax, and logical connectors. This allows extracting relevant information from deep within structures, as well as constructing new structures. The complement, intersection, and binding patterns allow reasoning using logical propositions and formulae.

Building upon the above, we derive the following FOL-like constructs:

$$\begin{array}{rcll} \top & \equiv & \exists x.x & // \text{ all terms} \\ \bot & \equiv & \neg\top & // \text{ no terms} \\ \varphi_1 \vee \varphi_2 & \equiv & \neg(\neg\varphi_1 \wedge \neg\varphi_2) & // \text{ disjunction} \\ \varphi_1 \rightarrow \varphi_2 & \equiv & \neg\varphi_1 \vee \varphi_2 & // \text{ implication} \\ \varphi_1 \leftrightarrow \varphi_2 & \equiv & (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) & // \text{ equivalence} \\ \forall x.\varphi & \equiv & \neg(\exists x.\neg\varphi) & // \text{ universal quantification} \end{array} \tag{2.1}$$

$\top$ is matched by all terms, $\bot$ is matched by no terms, $\varphi_1 \vee \varphi_2$ is matched by terms matching φ_1 or φ_2, etc. For example, $assignment(x, y \wedge \neg x)$ is matched by all assignment structures where the assigned term is different from the assignee.

2.2 Matching Logic Semantics

Intuitively, patterns are matched by other terms that have more structure. Still, it is often the case that background theories or domains are layered upon the purely structural match. For example, if we layer the theory of integers and consider that addition is commutative, we can say that $1 + 4$ matches $x + 3$ with the substitution $x = 2$.

Definition 2.2 *A matching logic* Σ***-model*** *consists of:*

(1) A non-empty set M*.*

(2) A function $\sigma_M : M^n \rightarrow \mathcal{P}(M)$ *for each symbol* $\sigma \in \Sigma$ *called the* ***interpretation*** *of* σ *in* M*.*

Symbols are interpreted as relations. The usual Σ-algebra models are a special case of matching logic models where $|\sigma_M(m_1, \ldots, m_n)| = 1$ for any $m_1, \ldots, m_n \in M$. Similarly, for partial Σ-algebra models, $|\sigma_M(m_1, \ldots, m_n)| \leqslant 1$. The undefinedness of σ_M is captured with $\sigma_M(m_1, \ldots, m_n) = \varnothing$.

We use the same notation when extending σ_M to argument sets:

$$\sigma_M(A_1, \ldots, A_n) = \bigcup \{\sigma_M(a_1, \ldots, a_n) | a_1 \in A_1, \ldots, a_n \in A_n\} \quad \text{where } A_1 \subseteq M, \ldots, A_n \subseteq M. \tag{2.2}$$

Definition 2.3 *Given a model M and a map $\rho : Var \to M$, called an M-**valuation**, let its extension $\bar{\rho} : \text{PATTERN} \to \mathcal{P}(M)$ be inductively defined as follows:*

- $\bar{\rho}(x) = \{\rho(x)\}$, *for all* $x \in Var$
- $\bar{\rho}(\sigma(\varphi_1, \ldots, \varphi_n)) = \sigma_M(\bar{\rho}(\varphi_1), \ldots, \bar{\rho}(\varphi_n))$, *for all* $\sigma \in \Sigma$, *where "$\setminus$" is set difference*
- $\bar{\rho}(\neg\varphi) = M \setminus \bar{\rho}(\varphi)$, *for all* $\varphi \in \text{PATTERN}$
- $\bar{\rho}(\varphi_1 \wedge \varphi_2) = \bar{\rho}(\varphi_1) \cap \bar{\rho}(\varphi_2)$, *for all* $\varphi_1, \varphi_2 \in \text{PATTERN}$
- $\bar{\rho}(\exists x.\varphi) = \bigcup_{a \in M} \overline{\rho[a/x]}(\varphi)$, *where $\rho[a/x]$ is ρ extended with a mapping of x to a. If $a \in \bar{\rho}(\varphi)$ then we say that a **matches** φ with witness ρ.*

The definition above gives us the equivalent of traditional term matching. Given a pattern (i.e., a term with variables) φ and a ground term model M, then a ground term a matches φ iff there exists a substitution ρ such that $\rho(\phi) = a$. We can regard patterns as predicates (see Section 2.3), where the equivalent of "predicate φ holds in a" is "a matches pattern φ".

The semantics of derived FOL-like constructs follows naturally from the semantics above:

- $\bar{\rho}(\top) = M$
- $\bar{\rho}(\bot) = \varnothing$
- $\bar{\rho}(\varphi_1 \vee \varphi_2) = \bar{\rho}(\varphi_1) \cup \bar{\rho}(\varphi_2)$
- $\bar{\rho}(\varphi_1 \to \varphi_2) = M \setminus (\bar{\rho}(\varphi_1) \setminus \bar{\rho}(\varphi_2))$
- $\bar{\rho}(\varphi_1 \leftrightarrow \varphi_2) = M \setminus (\bar{\rho}(\varphi_1) \ \Delta \ \bar{\rho}(\varphi_2))$
  where "Δ" is the symmetric difference operation on sets
- $\bar{\rho}(\forall x.\varphi) = \bigcap_{a \in M} \overline{\rho[a/x]}(\varphi)$

Of particular interest to us is the implication, which can be seen as a generalization of ground term matching to arbitrary patterns. The interpretation of $\varphi_1 \to \varphi_2$ is the set of all terms that, if matched by φ_1 then are also matched by φ_2.

2.3 Predicate Patterns

Matching logic collapses the function and predicate symbols of first-order logic. Intuitively, matching logic patterns can be seen both as terms and as predicates. When regarded as terms, they express structure; when regarded as predicates, they create constraints. The matching logic models are similar to the first-order logic models, except that the symbols in the signature are interpreted as functions returning sets of values instead of single values. Logical conjunction can be seen as intersection of the sets of valuations (i.e., of the matching terms), negation as the complement of the a set, and the existential quantifier as union over all possible valuations.

To clarify how a pattern can be interpreted as a predicate, we use the following definition:

Definition 2.4 *Given a model M, pattern φ is an M-**predicate**, or a **predicate** in M, iff for any M-valuation $\rho : Var \to M$, $\bar{\rho}(\varphi)$ is either M (it holds, it is "true") or $\emptyset$ (it does not hold, it is "false"). A pattern φ is simply a **predicate** iff it is a predicate in all models M.*

Note that $\top$ and $\bot$ are predicates, and can be seen as the matching logic equivalent of "true" and "false". Furthermore, the logical connectives preserve the predicate nature of patterns, e.g. if φ_1 and φ_2 are predicates then $\varphi_1 \vee \varphi_2$ is also a predicate.

2.4 Configuration Abstraction

Configuration abstraction is a technique for framing-out details which are not relevant for a particular formula. The semantics of a programming language is usually defined as a set of rewrite rules, which are formulae in dynamic matching logic, each interpreting one particular structure in the language. For example, the following K language rule interprets reading the value of a variable from the current state of the program:

$$\texttt{<k>}\ x \Rightarrow i \ldots \texttt{</k>}\ \texttt{<state>}\ \ldots x \mapsto i \ldots \texttt{</state>} \tag{2.3}$$

The rule is interpreted as follows: given that variable x is bound to value i in the program's state, when reading x in the computation cell, replace it with its value i. The rule only mentions relevant information, with $\ldots$ abstracting over the rest of the

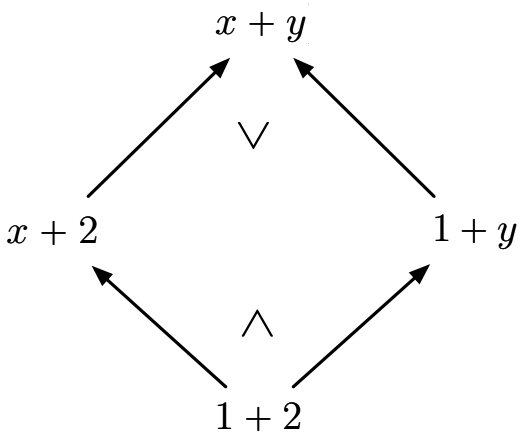


Figure 2.1: Unification and Anti-Unification in Matching Logic

configuration. The rest of the computation is elided by the first set of dots, and other bindings in the `state` cell are ignored. Any other cells that might be relevant (e.g., for keeping track of the stack, threads, etc.) are allowed without being mentioned by this rule.

2.5 Unification and Anti-Unification

Unification and generalization (or anti-unification) can be represented as conjunction and, respectively, disjunction of matching logic patterns [7]. Figure 2.1 shows an example of this representation. The arrows are matching logic implications. $x + 2$ and $1 + y$ are two matching logic patterns, with no other theory or domain in the logic – i.e., the plus sign is interpreted syntactically only. The unifier of the two patterns is their conjunction, $(x + 2) \wedge (1 + y) = (1 + 2)$. The anti-unifier, or generalizer, is their disjunction, $(x + 2) \vee (1 + y) = (x + y)$.

Unification as matching logic conjunction. Let φ_1 and φ_2 be matching logic patterns. We are interested in the concrete terms which match *both* φ_1 and φ_2. The concrete terms can be represented by some pattern φ, which is a **unifier** of φ_1 and φ_2. The pattern φ is more specific than both φ_1 and φ_2, so $\varphi \to \varphi_1$ and $\varphi \to \varphi_2$. Of particular interest is the *most general* unifier, i.e., if $\varphi' \to \varphi_1$ and $\varphi' \to \varphi_2$ then $\varphi' \to \varphi$. In matching logic, by definition, this pattern is the conjunction $\varphi_1 \wedge \varphi_2$.

Anti-unification (generalization) or matching logic disjunction. Pattern anti-unification, or *generalization*, is the opposite of unification. Given two patterns φ_1 and φ_2, we are interested in the concrete terms which match *either* φ_1 or φ_2. The concrete terms can be represented by some pattern φ, which is the **anti-unifier** (or **generalizer**) of φ_1 and φ_2. The pattern φ is more general than both φ_1 and φ_2, so $\varphi_1 \rightarrow \varphi$ and $\varphi_2 \rightarrow \varphi$. Similarly to unification, of particular interest is the *most specific* anti-unifier which, in matching logic, is by definition $\varphi_1 \vee \varphi_2$.

As the anti-unifier φ is a pattern from which both patterns φ_1 and φ_2 can be obtained by some (not necessarily concrete) substitution of the φ's variables, anti-unification can be seen as a mechanism for finding the common structures between the two input patterns.

Alpuente et al. [8] give an order-sorted equational generalization algorithm capable of covering pattern associative and commutative symbols. While powerful, matching with associative and commutative symbols lacks crucial features required for expressing program transformations. In Chapter 4 we describe their approach and how we extend it to infer complex program transformation patterns.

Chapter 3

Evolutionary Program Transformation

Program transformation is at the core of most software tools and processes. Compilers, including just-in-time compilers for interpreted languages such as JavaScript, transform code from a high-level language to lower-level, machine-executable code. Optimizing compilers, as well as automated and manual refactorings, transform code to improve various aspects of its performance, from time, memory, or energy consumption to softer aspects like maintainability or readability.

The work presented in this chapter was done in collaboration with Stephen J. Fink, Rodric Rabbah, and Manu Sridharan [27].

3.1 Problem

Despite many years of extensive research, program transformation systems still have trouble generating high performance code in many cases. For example, automatic parallelization works well for specific cases (e.g., stencil code) but fails to scale to more complex programs involving indirect memory references.

For most real-world problems, the optimal program is not reached through one large transformation, but is the result of applying several finer-grained transformations. Failures to generate optimal programs boil down to program transformation systems having difficulty answering the following very basic question:

> *Given a program, a set of program transformations, and a goal, which transformation gets the program closest to the goal?*

The problem is fundamentally hard and it does not have a unique answer. For example, an answer in the context of sequential program optimization may be the exact opposite to an answer in the context of automatic parallelization. Furthermore, a transformation which may not seem like a good idea at a particular step may be required for enabling a more powerful optimization further down the pipeline (see Section 3.4). While more powerful program analysis techniques, better domain specific languages, and

further specialization will incrementally alleviate the problem, we present an orthogonal approach that sidesteps the problem.

3.2 Our Approach

We bypass the problem of choosing the optimal program transformation by simply not making the choice in the first place. At each step, instead of making a set-in-stone choice which reflects some local optimum, we allow the program transformation system to consider many possible transformations. After one step, instead of having one new program, there are several new program variants to consider. Thus, our proposed technique is to search for the globally-optimal program among the program variants reachable via several fine-grained transformation steps.

Most program transformation systems are restricted to giving a single transformation as the answer to the question above. This restriction arose from the need for compilers to be very efficient, i.e., for the compilation time to be low. In other areas, like refactoring, the restriction is rooted in human and technical difficulties, or inconveniences, in dealing with more than one program variant. But the restriction is no longer necessary. It it now technically feasible and affordable to explore multiple transformation paths. On the one hand, cloud computing has made it easy and affordable to access large computational resources — compiling and optimizing code in the cloud may become the norm in the near future. On the other hand, developers have become accustomed to working with multiple variants of their code using tools such as Git [28] or Mercurial [29].

We call the proposed approach *evolutionary program transformation*, because it enriches program transformation with techniques from the field of *evolutionary algorithms* [5]. Drawing inspiration from the biological mechanisms of evolution, evolutionary algorithms build upon the idea that given a population of individuals (which can be anything from constants that need to be optimized to programs), applying environment pressure on them (i.e., removing individuals which score low according to a fitness function) causes natural selection (i.e., an increase in the fitness of the population) along with the emergence of particularly fit individuals. New individuals are created by either recombination, when two or more individuals (parents) are used to generate a new individual, or mutation, when a single individual is changed directly to create a new individual. For evolutionary program transformation, the idea is to consider program transformations as mutation operators.

The cross-pollination of program transformation and evolutionary algorithms has been mostly untapped, the notable exceptions being superoptimizers [30, 31], (though they are mostly restricted to loop-free, low-level instruction sequences) and genetic programming [11]. Our approach can be seen as a genetic programming approach with only mutation operators which have strong, specific semantic meaning. This dramatically reduces the space of possible program variants, thus increasing the probability of reaching a good solution.

3.3 Case Study: Translating to MapReduce

We discovered the need for an evolutionary approach while working on the problem of automatically translating sequential, imperative code to functional, parallel MapReduce.

MapReduce is a programming model which has gained traction both in research and in practice over the last decade. Mainstream MapReduce frameworks [32, 33] provide significant advantages for large-scale distributed parallel computation. In particular, MapReduce frameworks can transparently support fault-tolerance, elastic scaling, and integration with a distributed file system.

Additionally, MapReduce has attracted interest as a parallel programming model, independent of difficulties of distributed computation [10]. MapReduce has been shown to be capable of expressing important parallel algorithms in a number of domains, while still abstracting away low-level details of parallel communication and coordination.

Automatically translating sequential imperative code into a parallel MapReduce framework would be very useful. An effective translation tool could greatly reduce costs when re-targeting legacy sequential code for MapReduce. Furthermore, a translator could simplify the process of targeting MapReduce in new programs: a developer could concentrate on sequential code, letting the translator handle parallel computation.

Translating an imperative loop to the MapReduce model inherits many of the difficulties faced by parallelizing compilers, such as proving loops free of loop-carried dependencies. However, the MapReduce framework differs substantially from a shared-memory parallel loop execution model. Notably, MapReduce implies a *distributed memory* programming model: each mapper and reducer can operate only on data which is "local" to that function. So, an automatic translator must at least partition memory accesses in order to create local mapper and reducer functions which do not rely on shared memory.

Additionally, the communication model in MapReduce is more limited than traditional distributed memory parallel programming with message-passing. Instead, mappers and reducers communicate via a *shuffle* operation, which routes mapper outputs to reducer inputs based on on key fields in the data. These restrictions allow practical MapReduce frameworks to run relatively efficiently at large scale; however, they also introduce constraints on the programmer and challenges for an automatic translator.

The key challenge in building a translator to MapReduce is the effective handling of the imperative updates in the original code, which must be translated away to fit the program in a functional MapReduce form. Statements performing indirect reads and writes to array elements inside loops pose a particular challenge, due to the difficulty of determining whether such statements can access locations overlapping across loop iterations.

Previous approaches to parallelizing such codes (*e.g.*, [34]) typically rely on being able to prove that loop iterations only interfere with each other in very constrained ways, limiting their applicability. Section 3.4 will illustrate a simple example that existing approaches cannot handle.

3.4 Motivating Example

We assume the reader is familiar with the MapReduce model. Here, we briefly review MapReduce details as presented by Dean and Ghemawat [32] before presenting an overview of our approach using their `WordCount` example.

In the MapReduce framework, the programmer defines a *map* function and a *reduce* function. The functions have the following types:

$$\texttt{map} : \langle k1, v1 \rangle \rightarrow \texttt{List}[k2, v2] \tag{3.1}$$

$$\texttt{reduce} : \langle k2, \texttt{List}[v2] \rangle \rightarrow \texttt{List}[v2] \tag{3.2}$$

The MapReduce framework relies on a built-in, implicit *shuffle* function to route the output of the mappers to the input of the reducers. Logically, the shuffle function performs a *group-by* operation over the map outputs. That is, for each distinct key k of type $k2$ output by a mapper function, the shuffle function collects all the values of type $v2$ associated with k, forms a list l of these values, and sends the resulting pair (k, l) to a reducer.

Dean and Ghemawat use `WordCount` as an example when defining mappers and reducers. In their `WordCount` example, the *map* function takes as input a document name and a String which holds the document contents. For each word w in the contents, the mapper emits a pair $(w, 1)$. The shuffle operation will create a list of the form $[1, 1, \ldots, 1]$ for each word w, grouping the map output values (all ones) by word. Then the reducers simply sum up the number of ones present in the list associated with each word. The resulting sums represent the frequency count for each word.

The built-in *shuffle* or *group-by* operation plays a central role, for at least two reasons. Firstly, the shuffle operation encapsulates all communication between nodes. In traditional distributed-memory, parallel computing models, the programmer must explicitly pass messages or manage remote memory access in order to express communication. Instead, in MapReduce, the programmer simply defines functions which produce and consume tuples, and the framework transparently implements the necessary communication. MapReduce cannot express every possible parallel algorithm and communication pattern – but when MapReduce does apply, it relieves the programmer from the burden of managing communication explicitly, resulting in much simpler parallel programming. Secondly, we note that the shuffle operation can be extremely expensive, and can limit performance in many use cases if not managed carefully. Naïve use of MapReduce can result in all-to-all communication patterns whose overhead can overwhelm any speedups from parallel computation.

In the remainder of this chapter, we focus on MapReduce primarily as a convenient model for expressing parallel computation. In particular, we consider the challenge of automatically translating sequential code into a MapReduce parallel-programming model. We will not address issues specific to large-scale distributed Map-Reduce deployments, such as fault-tolerance, elasticity, and distributed file systems.

Overview of our approach. Consider the challenge of automatically generating effective MapReduce code for `WordCount`. Figure 3.1 shows the sequential Java code, our starting point. The program iterates through a list of documents `docs`, accumulating the word counts into the `m` map.

Parallelizing the Figure 3.1 example is difficult because of the updates to the shared `m` map in different loop iterations — naïvely running loop iterations in parallel would cause a race conditions on `m` if two iterations try to simultaneously update a word's count. Some parallelism might be achieved by splitting the `docs` list of documents into

```
Map<String,Integer> WordCount(List<String> docs) {
  Map<String,Integer> m = new HashMap<>();
  for (int i = 0; i < docs.size(); i++) {
    // simplified word split for clarity
    String[] split = docs.get(i).split(" ");
    for (int j = 0; j < split.length; j++) {
      String w = split[j];
      Integer prev = m.get(w);
      if (prev == null) prev = 0;
      m.put(w, prev + 1);
    }
  }
  return m;
}
```

Figure 3.1: Java word count program.

chunks, and computing word counts for each chunk simultaneously.

However, this transformation still leaves the sequential work of combining the word counts from the different chunks into final, global word counts. In contrast, the standard MapReduce word count program, which MOLD can generate, enables parallel accumulation of final word counts, by assigning the accumulation task for different ranges of words to different reducers.

We first consider generating a MapReduce program for the inner loop of Figure 3.1 (lines 7–12), which computes word counts for an array of words `split`. The first step of our technique is to translate the input program into a functional representation via Array SSA form [35]. Figure 3.2 gives the Array SSA form for our inner loop. Note that here, the `m` map is the "array" being updated by the loop. Our implementation treats Java arrays, `List`s, and `Map`s in a unified manner as mappings from keys to values. This form can be seen as functional if every write to a location of `m` is treated as creating a new copy of the map with a new value for the appropriate word.

After constructing the Array SSA form, MOLD translates the program to a more explicitly functional form. Unlike previous translations from SSA to functional code [36, 37], our technique preserves the structure of loops by translating them using the standard `fold` operation. MOLD transforms each non-ϕ SSA assignment into a `let` statement. E.g., $w = split[j_3]$ is transformed to $let\ w = split[j_3]\ in \ldots$.

Each loop (branching and ϕ instructions) is transformed into a fold going over the domain of the initial loop, with its combining operation taking the tuple of ϕ values as one argument and the loop's index as the other, and returning a new tuple of values. In

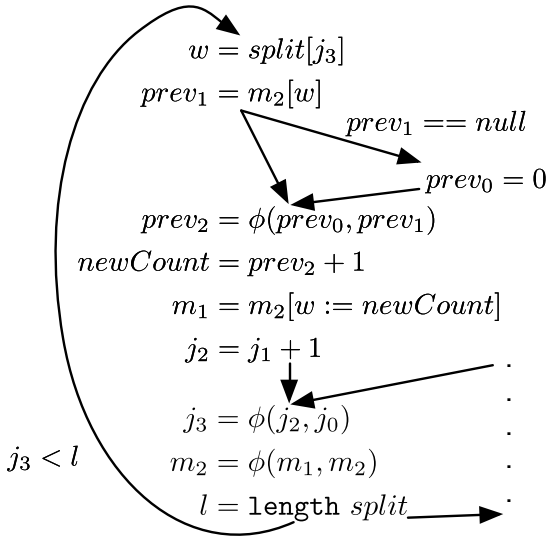


Figure 3.2: Array SSA form for the inner loop of Figure 3.1.

our example, the fold goes over the range of integers from 0 to the length of the `split` array, with a combining operation that takes as arguments m_2 and j_3 and returns m_1, i.e., the in-loop branch of $\phi(m_1, m_2)$.

Each remaining ϕ value with its corresponding `branch` instruction is rewritten into into a predicated `if` assignment. E.g. $\phi(prev_0, prev_1)$ with the corresponding branch condition $prev_1 == null$ is transformed to:

$$\texttt{if } prev1 == null \texttt{ then } prev0 \texttt{ else } prev1 \tag{3.3}$$

Thus, the SSA-form code in Figure 3.2 is converted to:

$$\begin{aligned}
&\texttt{let } updatedCount = \lambda\, m_2\ j_3\,. \\
&\quad \texttt{let } w = split[j3] \texttt{ in} \\
&\quad\quad \texttt{let } prev1 = m_2[w] \texttt{ in} \\
&\quad\quad\quad \texttt{let } prev2 = \texttt{if } prev1 == null \texttt{ then } 0 \texttt{ else } prev1 \texttt{ in} \\
&\quad\quad\quad\quad \texttt{let } newCount = prev2 + 1 \texttt{ in} \\
&\quad\quad\quad\quad\quad \texttt{let } m_1 = m_2[w \texttt{ := } newCount] \texttt{ in} \\
&\quad\quad\quad\quad\quad\quad m_1 \\
&\texttt{in fold } m_0\ updatedCount\ (0...(\texttt{length } split))
\end{aligned} \tag{3.4}$$

Next, MOLD explores the space of possible optimizing transformations that can be applied to the code above. The transformations are expressed as rewrite rules and are detailed in Section 3.8. For now, we will focus on the particular transformations that take the functional, yet sequential, code above and turn it into MapReduce form.

After inlining some of the `let` expressions and renaming variables for readability, we get to:

$$\begin{array}{l}\texttt{let}\ \mathit{updatedCount} = \lambda\, m\ j\,.\\ \quad \texttt{let}\ w = \mathit{split}[j]\ \texttt{in}\\ \quad\quad \texttt{let}\ \mathit{prev} = m[w]\ \texttt{in}\\ \quad\quad\quad m[w\ \texttt{:=}(\texttt{if}\ \mathit{prev} == \mathit{null}\ \texttt{then}\ 0\ \texttt{else}\ \mathit{prev}) + 1]\\ \texttt{in}\ \texttt{fold}\, m\ \mathit{updatedCount}\ (0...(\texttt{length}\ \mathit{split}))\end{array} \tag{3.5}$$

One initial observation is that the fold traverses a range of integers instead of the `split` collection itself. Thus, MOLD transforms the code such that the fold traverses the collection without indirection:

$$\begin{array}{l}\texttt{let}\ \mathit{updatedCount} = \lambda\, m\ w\,.\\ \quad \texttt{let}\ \mathit{prev} = m[w]\ \texttt{in}\\ \quad\quad m[w\ \texttt{:=}(\texttt{if}\ \mathit{prev} == \mathit{null}\ \texttt{then}\ 0\ \texttt{else}\ \mathit{prev}) + 1]\\ \texttt{in}\ \texttt{fold}\, m\ \mathit{updatedCount}\ \mathit{split}\end{array} \tag{3.6}$$

Next, MOLD identifies common update idioms and lifts them to more general functional operations. In our example, the `m` map is lifted to return `zero` for non-existent keys. Thus, the `if` condition returning either zero or the previous value in the map becomes unnecessary, and MOLD replaces it with just the map access. Thus, MOLD transforms the program to:

$$\begin{array}{l}\texttt{let}\ \mathit{updatedCount} = \lambda\, m\ w\,.\, m[w\ \texttt{:=}\, m[w] + 1]\ \texttt{in}\\ \quad \texttt{fold}\, m\ \mathit{updatedCount}\ \mathit{split}\end{array} \tag{3.7}$$

The `fold` call takes the initial `m` map, the `updatedCount` function, and the `split String` array, and computes a new map with updated word counts. The `updateCount` accumulator function takes a map `m` and a word `w` as arguments, and returns a new map that is identical to `m` except that the count for `w` is incremented.

The functional form above is semantically equivalent to the original imperative code, but unfortunately exposes no parallelism, since the `fold` operation is sequential. Furthermore, trying to parallelize the fold directly would not work as the accesses to the `m` collection do not follow a regular pattern, i.e. `w` may have any value, independent of the underlying induction variable of the loop.

A common way to parallelize such code is to take advantage of the commutativity of the updating operation and tile the fold, namely the loop [38]. While this solution does expose parallelism, it is coarse-grained, may not be applicable in the presence of indirect references, and does not match the MapReduce model of computation.

MOLD generates this tiled solution, but it also explores a different parallelization avenue: instead of avoiding the non-linear `w` value, a program can inspect it [39] to reveal parallelism. Parts of computation operating on distinct `w` values are independent so they can be executed in parallel. Thus, our example is also equivalent to:

$$\begin{array}{l} \texttt{let}\ \mathit{grouped} = (\texttt{groupBy}\ \mathit{id}\ \mathit{split})\ \texttt{in} \\ \quad \texttt{map}(\lambda\, k\ v\,.\,\texttt{fold}\, m[k](\lambda\, y\ x\,.\,y+1)\ v)\ \mathit{grouped} \end{array} \tag{3.8}$$

The inner fold is only computing the size of the `v` list of words, and adding it to the previous value in m. Assuming an initially empty map `m`, in the end, the rewrite system produces the following program as part of its output:

$$\begin{array}{l} \texttt{let}\ \mathit{grouped} = (\texttt{groupBy}\ \mathit{id}\ \mathit{split})\ \texttt{in} \\ \quad \texttt{map}(\lambda\, k\ v\,.\,v.\mathit{size})\ \mathit{grouped} \end{array} \tag{3.9}$$

The sequential `fold` operation has been completely eliminated. Instead, we are left with the equivalent of the canonical MapReduce implementation of word counting. The `groupBy` operation yields a map from each word to a list of copies of the word, one for each occurrence; this corresponds to the standard mapper. Then, the `map` operation takes the grouped data and outputs the final word count map, corresponding to the standard reducer.[1] Given a large number of documents spread across several servers, the `groupBy` task can be run on each server separately, and the `map` task in the reducer can also be parallelized across servers, given the standard "shuffle" operation to connect the mappers and reducers. The standard MapReduce implementation does not construct explicit lists with a `groupBy`, but instead sends individual word instances to

[1] The value of the `map` operation is itself a map, with the same keys as the input map; see Section 3.8.1 for details.

the reducers using "shuffle." MOLD further optimizes the above program to generate this more efficient implementation.

In this section we discussed the inner loop of the input `WordCount` program. In Section 3.8 we discuss how this integrates with the `fold` for the outer loop, and how MOLD brings the entire program to an optimized MapReduce form.

Finally, MOLD takes what it considers the best generated code versions and translates them to Scala. The best generated version, exactly as it is output by MOLD, is:

```
docs
  .flatMap({ case (i, v9) => v9.split(" ") })
  .map({ case (j, v8) => (v8, 1) })
  .reduceByKey({ case (v2, v3) => v2 + v3 })
```

The generated code's computation structure is similar to classic MapReduce solutions for the WordCount problem. Each document is split into words, which are then mapped to $(word, 1)$ pairs. The `reduceByKey` groups the "1" values by their associated *word*, and then reduces each group by $+$, effectively counting the number of words. The code above uses custom collection classes which implement interfaces that provide higher-order functional operators (*e.g.*, `map`). The system provides three implementations of these operators. The code can execute using Scala sequential collections, Scala parallel collections, or on the Spark MapReduce framework [12].

3.5 MOLD System Overview

Figure 3.3 illustrates the design of our translator. An input program is first translated into Array SSA form [35] which facilitates the derivation into an lambda-calculus-style functional representation. In contrast with Appel's and Kelsey's work on converting from SSA to functional code [36, 37], our translation uses a `fold` operator to maintain the structure of loops, which is essential for later transformations.

The initial "lambda plus `fold`" representation of a program is still far from an effective MapReduce program. While `fold` operations could be implemented using reducers, the lack of mappers hinders parallelism, and the code still operates on shared data structures. To address these problems, we employ a rewriting system to generate a large space of MapReduce programs. The rewrite rules govern where mapper constructs can be introduced in a semantics-preserving manner. Critically, in more complex cases where loop iterations access overlapping locations, we exploit the MapReduce *shuffle*

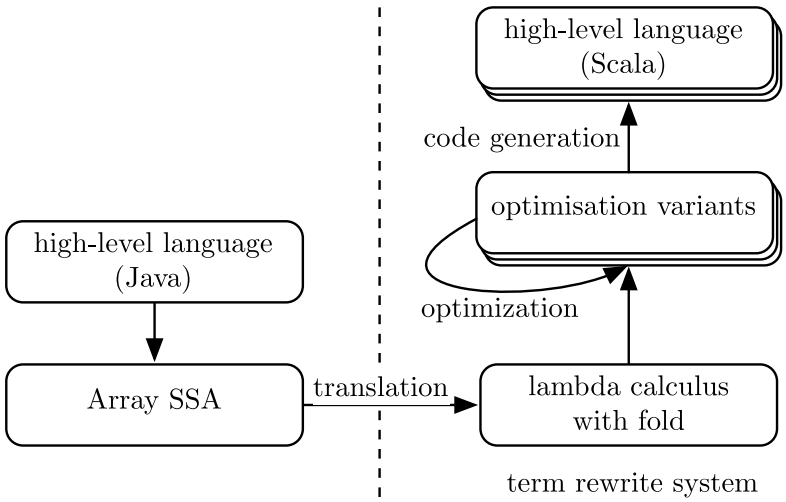


Figure 3.3: Overview of our translation system.

feature to group operations by accessed locations, exposing much more fine-grained parallelism than previous approaches. Given the rewriting rules, our system performs a heuristic search to discover a final output program, using a customizable cost function to rank programs.

In this chapter we give an automatic translation from imperative array-based code to a functional intermediate representation, amenable to generating MapReduce programs. Critically, the translation represents loops as `fold` operations, preserving their structure for further optimization. Then, we present a rewriting system to generate a broad space of equivalent MapReduce programs for a given imperative program from our functional IR. The space is explored via heuristic search based on a cost function that can be customized for different backends. Then, we describe a rewrite rule in our system that introduces `groupBy` operations to effectively handle complex indirect array accesses. This novel technique is critical for handling basic MapReduce examples like "word count.". Finally, we present Mold, an implementation of our techniques, and an experimental evaluation showing the tool's ability to handle complex input programs beyond those in the previous work.

3.6 Generating Functional IR

In this section, we describe the initial stages of our translation system, which convert an input program into a functional intermediate representation via Array SSA form.

3.6.1 Array SSA

The standard SSA representation enables straightforward tracking of def-use relationships by ensuring each variable has a single static definition which reaches all uses. However, standard SSA does not reflect modification to the stack or heap, such as the effects of array writes. Array SSA form [35] extends traditional SSA form with constructs that represent modifications to array contents.

Array SSA form generates a new name for an array variable on each write. Any subsequent reads from the array use the new name, simplifying tracking of data flow though the array. In contrast to the original work on Array SSA, our system treats arrays as being immutable data structures, as we aim to translate the program into a functional representation.

Hence, each array write is modeled as creating a fresh array with a new SSA name. With this functional model, our Array SSA form is simplified compared to the previous work [35]; we do not require a ϕ statement after each array write, and we do not require a special semantics for ϕ statements over array variables.

Mold generalizes the collections covered by Array SSA to include maps and lists. Thus, new SSA names are also introduced when putting an element into a map, or adding or putting an element into a list. This allows Mold to treat arrays, lists, and maps in a unified manner as mappings from keys to values.

The Array SSA form guarantees that, as long as no aliases are introduced through callee or caller functions, and no pointer to an array is retrieved/stored through another object or array, the arrays (and the lists and maps) can be treated as a values. Note that our translation to Array SSA form does not check for aliasing of array variables introduced in unobserved code, e.g., via caller or callee functions. Our implementation currently assumes that unobserved code does not introduce such aliasing.

3.6.2 Functional IR

Previous work by Appel [36] and Kelsey [37] observed that a program in SSA form is essentially a functional program. This observation can be extended in a straightforward manner to the Array SSA form described above, yielding a method for translating to a functional intermediate form.

However, the aforementioned translation techniques are not suitable for our purposes, as they do not preserve the structure of loops (which are translated to general recursive

$$
\begin{array}{rl}
\text{Var} & a|b|c|\ldots \\
& |\langle \text{Var}, \text{Var}, \ldots \rangle \\
\text{Exp} & \lambda\, \text{Var}[:\text{Type}]\,.\,\text{Exp} \\
& |\,\texttt{let}\,\text{Var} = \text{Exp}\,\texttt{in}\,\text{Exp} \\
& |\ \text{Exp}\ \ \text{Exp} \\
& |\langle \text{Exp}, \text{Exp}, \ldots \rangle \\
& |\,\text{Exp}[\text{Exp}] \\
& |\,\text{Exp}[\text{Exp}\ \texttt{:=}\ \text{Exp}] \\
\text{Type} & A \mid B \mid C \mid \ldots \\
& |\ \text{Type} \to \text{Type} \\
& |\,\langle \text{Type}, \text{Type}, \ldots \rangle \\
& |\ \text{Type}[\text{Type}]
\end{array}
$$

Figure 3.4: Lambda calculus IR

function calls). Transformation of loop computations is critical to introducing parallelism, so our system relies on knowledge of loop structure. Here, we give a translation from our Array SSA form to a functional IR that includes a built-in `fold` construct, used to preserve the structure of loops.

Mold's intermediate representation (IR), shown in Figure 3.4, is a typed lambda calculus. For brevity, throughout the rest of the chapter, we omit the types when they are clear from the context. The IR is enriched with a tuple constructor, e.g., $\langle e_1, e_2, \ldots \rangle$. Tuples behave like functions (e.g., $\langle f, g\rangle\ \langle a, b\rangle = \langle f\ a, g\ b\rangle$), and are unwrapped on application (e.g., $(\lambda\, x\, y\,.\,y)\ \langle a, b\rangle = b$).

We sometimes use a tuple notation in the parameter part of lambda expression for highlighting a particular grouping of the parameters, but it can be interpreted in a curried fashion, e.g., $\lambda\langle x, y\rangle z\,.\,e = \lambda\, x\ y\ z\,.\,e$. The IR has `let` expressions which allow simple pattern-matching on tuples (e.g., $\texttt{let}\langle x_1, x_2\rangle = \langle e_2, e_2\rangle\,\texttt{in}\, x_1 + x_2$). $a[b]$ and $a[b\ \texttt{:=}\ c]$ are read and write accesses at index b of map (array) a.

Figure 3.5 shows the built-in data structures along with signatures for the functions and operations operating over them. These constructs are mostly well-known, and we will describe them as needed throughout the paper.

The translation from Array SSA relies on left-`fold`, a higher-order function that takes an initial element (a zero) of type B and a combining operation $B \to A \to B$,

(data structures)

$\mathtt{M}[A]$: bag (multiset) with values of type A

$\mathtt{M}[K, V]$: map with keys of type K and values of type V

$\mathtt{M}[K, V]$ is an $\mathtt{M}[\langle K, V \rangle]$

(higher order functions)

$$\begin{aligned}
\mathtt{fold} &: B \to (\langle B, A \rangle \to B) \to (\mathtt{M}[A] \to B) \\
\mathtt{map} &: (A \to B) \to (\mathtt{M}[A] \to \mathtt{M}[B]) \\
\mathtt{map} &: (\langle K, V \rangle \to W) \to (\mathtt{M}[K, V] \to \mathtt{M}[K, W]) \\
\mathtt{groupBy} &: (A \to K) \to (\mathtt{M}[A] \to \mathtt{M}[K, \mathtt{M}[A]])
\end{aligned}$$

(functions and operations)

$$\begin{aligned}
\mathtt{zip} &: \langle \mathtt{M}[K, A], \mathtt{M}[K, B] \rangle \to \mathtt{M}[K, \langle A, B \rangle] \\
\mathtt{zip} &: \langle \mathtt{M}[A], \mathtt{M}[B] \rangle \to \mathtt{M}[\langle A, B \rangle] \\
\mathtt{zip} &: \langle \mathtt{M}[K], \mathtt{M}[K, B] \rangle \to \mathtt{M}[K, B] \\
\mathtt{++} &: \langle \mathtt{M}[K, A], \mathtt{M}[K, A] \rangle \to \mathtt{M}[K, A] \\
&\quad \text{addition with replacement of matching keys} \\
\oplus \ \boxplus &\quad \text{monoid plus operators} \\
\oplus &: \langle \mathtt{M}[K, A], \mathtt{M}[K, A] \rangle \to \mathtt{M}[K, A] \\
a \oplus b &= (\mathtt{map}\, \lambda\, k \langle x \ y \rangle \,.\, x \boxplus y)(a\, \mathtt{zip}\, b)
\end{aligned}$$

Figure 3.5: Built-in data structures and operators

and returns a function which reduces the elements of a collection of A elements into a value of type B by applying the operation from left to right.

MOLD transforms Array SSA form to the functional intermediate representation (IR) by applying the rules in Figure 3.6. The instructions in the control flow graph (CFG) are visited in topological order. Strongly connected components are considered nodes in the order, and are visited in turn in topological order. We use the $\prec$ operator in Figure 3.6 to reflect this ordering: $s \prec R$ matches a statement s followed by remaining statements R in the topological ordering.

We first discuss the non-loop rules, which are mostly straightforward. SSA assignments $x = E$ are transformed to `let`s. Any branch instruction is skipped, left to be handled when reaching its associated ϕ. The `return` instruction is replaced with the returned variable, which eventually sits at the innermost level of the `let` nest.

An `if` statement in the original code corresponds a branching statement followed by a set of ϕ instructions in SSA form. MOLD transforms each of the ϕ instructions corresponding to an `if` into a functional `if` with the condition coming from the branching instruction, and the branches being the arguments of the ϕ instruction. As the instructions are visited in a topological order, the variables holding the result for each of the two branches are already available in scope.

Computing the results for the `if` before the instruction is not an issue from a semantic perspective because in our representation we have no side effects and no recursion (except for structured `fold` recursion). Also, performance is not hurt as the two branches are inlined by the rewrite system in a later step.

The more complex rule translates loops to applications of the `fold` operator. In Figure 3.6, a loop is specified in terms of its ϕ variables, which include the index variable i and other variables $r_1, r_2, \ldots$ updated in the loop body. These variables characterize all possible effects of the loop visible to subsequent code. For each ϕ variable r_k, we use r'_k to refer to the value coming from outside the loop, and r''_k for the new value produced by the loop.

The loop gets translated to a `fold` over the domain of values for the index variable, from i' to l, the loop bound, in Figure 3.6. The combining function takes as arguments the current r_k values and loop index, runs the body of the loop E once for those values, and returns the new r_k values. The initial value for the `fold` is a tuple of the r'_k values coming from outside the loop.

Any loop with a loop-invariant domain (i.e., the domain does not depend on the

$$\mathcal{L}(x = E \prec R) \quad \downarrow \quad \texttt{let}\, x = \mathcal{L}(E)\, \texttt{in}\, \mathcal{L}(R) \qquad \mathcal{L}(a[x := y]) \quad \downarrow \quad a[x := y]$$

$$\mathcal{L}(\texttt{return}\, x) \quad \downarrow \quad x \qquad \mathcal{L}(\text{branch instruction} \prec R) \quad (\text{handled when reaching its } \phi) \quad \downarrow \quad \mathcal{L}(R)$$

$$\mathcal{L}\left(\begin{array}{l} \texttt{for}\ \ i = \phi(i', i''), \\ \qquad r_1 = \phi(r'_1, r''_1), \ldots, r_n = \phi(r'_n, r''_n) \\ \qquad i < l \\ \{\ E\ \} \prec R \end{array}\right) \quad \downarrow \quad \mathcal{L}\left(\begin{array}{l} \texttt{let}\, f = \lambda\, r_1\ r_2 \ldots i\, .\, \mathcal{L}(E \prec \langle r''_1, \ldots, r''_n \rangle)\, \texttt{in} \\ \quad \texttt{let}\, r = \texttt{fold}\langle r'_1, \ldots, r'_n \rangle\ f\ \ \texttt{Range}(i', l)\, \texttt{in} \\ \qquad \texttt{let}\, r_1, \ldots, r_n = r\, \texttt{in}\, \mathcal{L}(R) \end{array}\right)$$

$$\mathcal{L}\left(\begin{array}{l} x = \phi(x_0, x_1) \\ \text{generated by the } \texttt{if} \\ \text{with branch condition } C \end{array}\right) \prec R \quad \downarrow \quad \texttt{let}\, x = \text{if}\, C\, \text{then}\, x_0\, \text{else}\, x_1\, \texttt{in}\, \mathcal{L}(R) \qquad \mathcal{L}(\ldots) \quad \downarrow \quad \ldots$$

Figure 3.6: Array SSA to Lambda Calculus with `fold`. $\mathcal{L}$ is the translation function, and $\prec$ reflects a topological ordering of statements in the CFG.

loop's ϕ functions) can be translated to a `fold`. Our current implementation only handles indexed collection iteration with a range of numbers with stride 1.

All other SSA instructions (function calls, operations, etc.) are transformed to lambda calculus in the intuitive straightforward manner. Function calls are not inlined.

3.7 Translation System

The transformation described in the previous section generates a lambda calculus representation of the original program but is still far from MapReduce form. The loops in the original program are now sequential `fold`s that do not expose any parallelism. In order to get to MapReduce form, MOLD explores the space of semantically equivalent programs obtained by applying a set of program transformation rules.

3.7.1 Exploration and Refinement

MOLD distinguishes between *refinement* and *exploration* rewrite rules. *Refinement* rules make definite improvements to the input term, e.g., eliminating a redundant operation such as $\texttt{fold}\, r\, \lambda\langle r, \langle k, v\rangle\rangle\, .\, r[k := v]$. *Exploration* rules may either improve the original code or bring it to a form that allows other rules to apply, e.g., loop fission is not necessarily an optimization but may allow another rule to eliminate part of the original loop.

Exploration rules are treated as transitions between states, i.e., applying a transition rule generates a new state in the system. Refinement rules do not generate new states but are applied exhaustively to the output of an exploration rule.

One transition in our rewrite system is comprised of one application of an exploration rule followed by a complete reduction using the set of refinement rules. The set of refinement rules can be seen as a separate confluent rewrite system.

The set of transformation rules is not complete, *i.e.*, they do not generate all possible semantically equivalent programs. The rules are intended to be sound, *i.e.*, they are intended to preserve the semantics of the original program, but we have not formally proven this. More formal characterizations of soundness and completeness are planned for future work.

3.7.2 Optimized Exploration

MOLD's rewrite system implements optimization mechanisms which can be applied according to a number of policies, guided by estimates of code quality. The system is not confluent nor terminating – so, the rewrite engine explores the space of possible rewrites guided by a heuristic driven by a cost function.

The optimization problem reduces to searching through this system for a good solution. The number of states is kept in check by having a single state represent all alpha-equivalent programs that have the same beta-reduced form.

MOLD searches through the state space guided by a cost approximation function over program variants. The cost function approximates the runtime performance of the code on a particular platform. Thus, MOLD allows optimization for different platforms by adopting appropriate cost functions.

Figure 3.7 shows the cost estimation function for generating MapReduce programs. The estimated cost is computed recursively over a given term. The cost of function composition/application and tuples is the sum of the cost of their subexpressions. The cost for collection accesses has an extra weight ($C_{get}^{collection}$ and $C_{set}^{collection}$) to encourage access localization. `map` and `fold` operators have an initial cost meant to model the start of a distributed operation ($C_{init}^{\texttt{map}}$, $C_{init}^{\texttt{fold}}$, and $C_{init}^{\texttt{groupBy}}$), and have their operation cost multiplied by a constant ($C_{op}^{\texttt{map}}$, $C_{op}^{\texttt{fold}}$, and $C_{op}^{\texttt{groupBy}}$) representing an approximation for the size of the array. Binary operations are curried, and their function has a constant cost. All other functions have a predefined, constant, cost.

A unique set of constants is used for generating all programs in our evaluation, i.e., the constants are not program-specific. We determined good values by manually running experiments and refining the constants. This process could be automated to find a more precise and possibly platform-specific set of constants. Furthermore, our rough cost estimation function could be made more precise by applying techniques such as those proposed by [40], but the current approach has proved sufficient for optimizing most programs in our evaluation suite.

3.8 Optimization Rules

In this section we present the main rules of MOLD's rewrite system. We show the rules in a simplified form. The actual rules have additional complexity for updating types, for handling idiosyncrasies of our source and target languages, for piggy-backing arity

$$
\begin{aligned}
\mathcal{C}(F \circ G) &= \mathcal{C}(F) + \mathcal{C}(G) \\
\mathcal{C}(F(G)) &= \mathcal{C}(F) + \mathcal{C}(G) \\
\mathcal{C}(\langle F, G, ... \rangle) &= \mathcal{C}(F) + \mathcal{C}(G) + ... \\
\mathcal{C}(A[I]) &= C_{get}^{collection} + \mathcal{C}(A) + \mathcal{C}(I) \\
\mathcal{C}(A[K \texttt{ := } V]) &= C_{set}^{collection} + \mathcal{C}(A) + \mathcal{C}(K) + \mathcal{C}(V) \\
\mathcal{C}(\texttt{map}\, F) &= C_{init}^{\texttt{map}} + C_{op}^{\texttt{map}} * \mathcal{C}(F) \\
\mathcal{C}(\texttt{fold}\, I\ F) &= \mathcal{C}(I) + C_{init}^{\texttt{fold}} + C_{op}^{\texttt{fold}} * \mathcal{C}(F) \\
\mathcal{C}(\texttt{groupBy}\, F) &= C_{init}^{\texttt{groupBy}} + C_{op}^{\texttt{groupBy}} * \mathcal{C}(F)
\end{aligned}
$$

Figure 3.7: Cost estimation function

information useful for code generation, and for optimizing the exploration. Furthermore, the actual rule set has additional variants and guards for correctly handling non-pure functions like `random`.

Figure 3.8 summarizes the notation and functions we use in the following figures.

3.8.1 Extract `map` from `fold`

The transformation for revealing parallelism which is most commonly applied is the "extract map from fold" rule in Figure 3.9. It transforms a fold by identifying *independent* computations in its combining function f, i.e., operations that do not depend on results from other f invocations during the fold. These independent computations are extracted into a (parallelizable) map operation. For example, $\texttt{fold}\, 0\, \lambda\, r\ k\ v\, .\, r + (\texttt{f}\ k\ v)$ is transformed to $(\texttt{fold}\, 0\, \lambda\, r k\ v_f\, .\, r + v_f) \circ (\texttt{map}\, \lambda\, k\ v\, .\, \texttt{f}\ k\ v)$, as $(\texttt{f}\ k\ v)$ is independent (we shall explain `map` shortly). After the transformation, the purely-functional `map` can be easily parallelized, while the `fold` with the commutative + operation can also be executed very efficiently.

The signatures for our data structures and mapping operators relevant to this transformation are shown in Figure 3.5. Data structures are either a bag of values of type A, or *indexed collections* (e.g., arrays or maps) with key type K and value type V. We often view an indexed collection as a list of key-value pairs. The first `map` version takes a collection of elements of type A into a collection of elements of type B, as is standard. The second `map` version is similar but only applies to indexed collections. It generates a new indexed collection with the same keys as the original and the mapped

E	letters in uppercase are pattern variables
x	letters in lowercase are program variables or patterns matching program variables
$E_1 \subset E_2$	E_1 is a subexpression of E_2
$\text{free}(E)$	is the set of free variables in E
$x \in E$	is shorthand for $x \in \text{free}(E)$
$E[E_1/E_0]$	substitute E_1 for E_0 in E
K	is only used for denoting pattern matches on the parameters binding to the keys of the operator domain

Figure 3.8: Notation for optimization rules.

values. We assume that a mapping function $A \to B$ is implicitly lifted to $\langle K, A\rangle \to B$ if necessary.

The "extract `map` from `fold`" rule, shown in Figure 3.9, matches on any `fold` taking $\langle r_0^0, \ldots, r_n^0\rangle$ as the initial value and a function combining each tuple of keys K and values V of a collection to a tuple $\langle r_0, \ldots, r_n\rangle$. The `fold` operation E is split into the composition of functions $(\lambda\langle v_0^f, \ldots, v_m^f\rangle \,.\, F) \circ G$, such that G is the most expensive computation (according to the cost function $\mathcal{C}$; see Section 3.7) that is *independent* of other "iterations" of the `fold`'s execution. If we see the `fold` as a loop, G does not have any loop carried-dependencies.

How do we reason that G is independent? For a functional `fold` operation, a dependence on other fold "iterations" manifests as an access of an *accumulator* parameter r_i, i.e., a parameter holding the "result thus far" of the fold. Hence, if G makes no reference to any parameter r_i, it is trivially independent. Unfortunately, this simple reasoning is insufficient for providing independence for cases like the following:

$$\texttt{fold}\, r^0\, \lambda\, r\, k \,.\, r[k := f(r[k])] \tag{3.10}$$

This `fold` updates each entry in a collection to a function of its previous value. We would like to extract the computation $f(r[k])$ into a parallel map operation, but it accesses accumulator parameter r and hence is not trivially independent.

To handle cases like the above, we use this more sophisticated independence check

(extract `map` from `fold`)

$$\frac{\texttt{fold}\langle r_0^0, \ldots, r_n^0\rangle\, \lambda\langle r_0, \ldots, r_n\rangle\ K\ V\,.\,E}{\begin{array}{l}(\texttt{fold}\langle r_0^0, \ldots, r_n^0\rangle\, \lambda\langle r_0, \ldots, r_n\rangle\ K\ \langle v_0^f, \ldots, v_m^f\rangle V_{\cap\,\mathrm{free}(F)}\,.\,F) \\ \quad \circ\ (\texttt{map}\,\lambda K\ V\,.\langle G[r_\_^0/r\_], V_{\cap\,\mathrm{free}(F)}\rangle) \\ E = (\lambda\langle v_0^f, \ldots, v_m^f\rangle\,.\,F) \circ G \\ F \text{ is } \arg\max \mathcal{C}(G) \text{ with the condition:} \\ \quad \nexists i \in [0..n]\ .r_i \in G \wedge r_i \in E[r_\_^0/r\_] \text{ where} \\ \quad\quad r_\_^0/r\_ = r_i^0[k]/r_i[k] \text{ applied for all } i \in [1..n]\ k \in K\end{array}}$$

(fold to group by)

$$\frac{\texttt{fold}\, r_0\, \lambda\, r\, V\,.\, r[E := B]}{\begin{array}{l}(\texttt{map}\,\lambda\, k\, l\,.(\texttt{fold}\, r_0[k]\, \lambda\, g\, V\,.\, C)\ l)\ \circ\ (\texttt{groupBy}\,\lambda\, V\,.\, E) \\ \quad C = B[g/r[E]] \\ \quad r \notin C \wedge r \notin E \wedge \exists\ v \in V.v \in E \\ \quad \text{we cannot prove } E \text{ is distinct across the folding}\end{array}}$$

Figure 3.9: Rules revealing parallelism in `fold` operators

for G:

$$\nexists i \in [0..n]\ .r_i \in G \wedge r_i \in E[r_\_^0/r\_] \tag{3.11}$$

As shown in Figure 3.9, the $E[r_\_^0/r\_]$ expression substitutes an access to the initial collection $r_i^0[k]$ for $r_i[k]$ in E, for all possible r_i and k. (We shall discuss the reason for this particular substitution shortly.) So, in essence, the formula ensures that for any $r_i \in G$, *all* appearances of r_i in the enclosing expression E are of the form $r_i[k]$, i.e., they are accesses to a *current* key k. (Any non-conforming access like $r_i[k+1]$ will not be removed by the $r_\_^0/r\_$ substitution.) Checking that all collection accesses in E use a current key ensures that G remains independent in spite of its access of the accumulator collection.

If a non-trivial (i.e., contains computation with a non-zero cost) G is found, it is pulled out into a `map` which is then composed with a `fold` executing F, the remaining computation in E. The signature of the `fold`'s operation is adjusted to account for the change: $v_0^f, \ldots, v_m^f$, the variables linking G to F, are now parameters, and any previous

parameters (V) which are still needed by F are propagated (i.e., $V_{\cap \text{free}(F)}$). As the extracted G no longer has access to the r_i parameters, we place $G[r^0_{\_}/r_{\_}]$ in the map instead, so its accesses are performed on the initial collection r^0.

The rule does not specify how E is decomposed. E is in many cases a tuple of expressions. Our current implementation selects the subexpression with the largest cost for each expression in the tuple E. It uses a recursive function that computes the largest subexpression considering name binding constraints and the cost function.

The "extract `map` from `fold`" rule rewrites the example above that updates all collection values to:

$$(\texttt{fold}\, r^0\, \lambda\, r\langle k, v\rangle\, .\, r[k := v]) \circ (\texttt{map}\, \lambda\, k\, .\, f(r_0[k])) \tag{3.12}$$

The "extract `map` from `fold`" transformation is somewhat analogous to parallelizing a loop with no loop-carried dependencies in imperative code. A key difference is that here, we leverage our functional IR to extract and parallelize sub-computations of the `fold` without worrying about side effects; similar transformations for imperative loops would likely require greater sophistication.

3.8.2 Transform `fold` to `groupBy`

While the "extract `map` from `fold`" rule exposes significant parallelism, it cannot handle cases where distinct loop iterations can update the same array / map location. MapReduce applications like `WordCount` from Section 3.4 often work around such issues by using a *shuffle* operation to group inputs by some key and then process each group in parallel. Here we present a "`fold` to `groupBy`" rule that enables our system to automatically introduce such *shuffle* operations where appropriate, dramatically increasing parallelism for cases like `WordCount`. We are unaware of any similar automatic transformation in previous work.

The transformation we used for grouping by word is an application of the "`fold` to `groupBy`" rule shown in Figure 3.9. As shown in Figure 3.5, `groupBy` clusters the elements of a collection of type $\texttt{M}[A]$ according to the result of the function $A \to K$. It returns a map from keys K to lists $\texttt{M}[A]$ of elements in the original collection that map to a specific key. The rule matches any `fold` with a body which is an update of a collection at an index E that we cannot prove as distinct for each execution of the `fold`'s body. If the index is obviously distinct, MOLD applies the "extract `map` from

`fold`" rule instead — see Section 3.8.1.

The output code first groups the elements of the collection by the index expression ($\texttt{groupBy}\,\lambda V\,.\,E$), and then it folds each of the groups using the update expression B from original body of the loop. *groupBy*'s output is a Map from each distinct value of E to the corresponding subset of the input collection. The `map` operation's parameters are k, which binds to the keys of the grouped collection (i.e., evaluations of E), and l which contains a subset of the input collection. The fold starts from the k value of r_0, and folds l using the operation C, which is original expression B with accesses to index E of the old reducer replaced with with g, the new parameter corresponding only to the k-index of r.

The side condition requires that r does not appear in either the new expression C or the index expression E. Otherwise, the result of the of computation could depend on `fold`'s evaluation order, so the transformation would not be correct. To avoid grouping by an invariant expression, resulting in a single group, the side condition also requires that E is dependent on some parameter in V.

Revisiting the original example, the expression below is the program before applying the rule, with a beta reduction applied to make the match clear:

$$\texttt{fold}\,m\;\;\lambda\,m\;w\,.\,m[w := m[w] + 1] \tag{3.13}$$

The outer m matches r_0, the inner m matches r, w matches E, and `m[w] + 1` matches B. The side conditions are satisfied, so the expression is rewritten to:

$$\texttt{map}(\lambda\,k\;l\,.\,\texttt{fold}\,m[k](\lambda\,g\;w\,.\,g+1)\;l) \circ (\texttt{groupBy}\,\lambda\,w\,.\,w) \tag{3.14}$$

3.8.3 Localizing Accesses

MapReduce platforms often require computations to be *local*, i.e., to be free of accesses to global data structures. Our system contains rules to *localize* computations that do not meet this condition. Consider the following computation, based on an intermediate transformation of the `WordCount` example:

$$(\texttt{map}\,\lambda\,k\;v\,.\,m[k] + \texttt{size}\;v)\;\mathit{grouped} \tag{3.15}$$

Here, *grouped* maps each word to a list of occurrences, and the `map` is summing the size of each list with existing counts in some map m. This code cannot be executed in

(localize-map-accesses)

$$\frac{\texttt{map}\,\lambda\,KV\,.\,E}{\lambda\,a\,.((\texttt{map}\,\lambda\,K\;V v\,.\,E[v/c[i]])\;\circ(\texttt{zip}\,a\;c))} \quad \begin{array}{l} c \in \{c \subset E \mid \\ (\exists! i \in K.c[i] \subset E) \wedge \\ (\text{free}(c) \setminus \text{free}(E) = \varnothing)\;\wedge \\ (\nexists v \in K \cup V\;.\;v \in c)\} \end{array}$$

(localize-group-by-accesses)

$$\frac{\texttt{groupBy}(\lambda\,k\,.\,E)\;D}{\texttt{groupBy}(\lambda\,kv\,.\,E[v/c[k]])c} \quad \begin{array}{l} D \text{ is the domain (set of} \\ \text{keys) of the } c \text{ Map} \end{array}$$

Figure 3.10: Rules for localizing accesses

MapReduce because it accesses the collection m, unless it is localized.

Localization is achieved by explicitly passing global data as a parameter to the relevant operations, using the built-in `zip` operation from Figure 3.5. `zip` is overloaded to allow various input collection types. Its first version takes two maps with the same key type into a map from keys to pairs of values. If one of the maps is missing a value for a certain key, the zero value for the map's value type is used instead. For example: $\texttt{zip}(M(1 \to 8), M(2 \to 9)) = M(1 \to \langle 8, 0\rangle, 2 \to \langle 0, 9\rangle)$. `zip`'s second version takes a bag S and a map M and returns a map that retains only the entries from M with keys from S, e.g., $\texttt{zip}(S(3), M(3 \to 8, 1 \to 9, 2 \to 7)) = M(3 \to 8)$.

Using the `zip` operation, the `map` from the example above can be transformed to:

$$\texttt{map}\;(\lambda\,k\;v\;v_m\,.\,v_m + \texttt{size}\;v)\;\;\texttt{zip}(\mathit{grouped}, m) \tag{3.16}$$

The "localize `map` accesses" rule in Figure 3.10 achieves this transformation. In this form, the `map`'s operation only refers to data provided through its parameters, making it amenable to MapReduce execution. The "localize `groupBy` accesses" (Figure 3.10) and "localize `fold` accesses" (not shown) achieve the same purpose for their respective operators.

3.8.4 Loop Optimizations and Code Motion

Our rewrite system has many rules that are inspired from classic loop optimizations and code motion that permits the application of more rules during rewriting. Figures

(map fusion)

$$\frac{(\texttt{map}\,\lambda\, k_2 v_2\,.\,E_2)\circ(\texttt{map}\,\lambda\, k_1 v_1\,.\,E_1)}{\texttt{map}\,\lambda\, k_1, v_1\,.\,E_2[k_1/k_2][E_1/v_2]}$$

(fold vertical fission)

$$\frac{\texttt{fold}\langle r_0^0,\ldots,r_n^0\rangle\,\lambda\langle\langle r_0,\ldots,r_n\rangle V\rangle\,.\langle E_0,\ldots,E_n\rangle}{\lambda\, c\,.\langle(\texttt{fold}\, r_0^0\,\lambda\, r_0\, V\,.\,E_0)\ c,\ldots,(\texttt{fold}\, r_n^0\,\lambda\, r_n\, V\,.\,E_n)\ c\rangle}$$
$$\text{applied } \forall k\in 0\ldots n \quad \forall k'\neq k.r_{k'}\notin E_k$$

(map vertical fission)

$$\frac{(\texttt{map}\,\lambda\, KV\,.\,E)\circ(\texttt{zip}\, C_0\ldots C_n)}{(\texttt{map}\,\lambda\, KV[z/v]\,.\,E[z/F])\circ(\texttt{zip}\, C_0\ldots(\texttt{map}\,\lambda\, k\ v\,.\,F\ C_k)\ldots C_n)}$$
$$F,G=\underset{E=F\circ G\wedge cond}{\arg\max}\ \mathcal{C}(F)$$
$$cond:(\exists!k.C_k\in F)\wedge v\in V \text{ goes over } C_k\ \wedge\forall v'\in V.v\neq v\Rightarrow v'\notin C_k$$

(map horizontal fission)

$$\frac{\texttt{map}\ \ \lambda\, K\ V\,.\,B}{\texttt{map}\,\lambda\, K(\text{free}(F)\cap V)(\text{free}(G)\setminus\text{free}(B))\,.\,F)\circ(\texttt{map}\,\lambda\, KV\,.\,G)}$$
$$F,G=\underset{B=F\circ G}{\arg\max}\,\mathcal{C}(F)\quad F \text{ is not trivial}$$

Figure 3.11: Fusion-fission rules for merging and splitting `map` and `fold` operators.

3.11, 3.13, and 3.12 detail these transformations.

Figure 3.11 shows rules for merging and splitting `map` and `fold` operators. As loops in original imperative program often update multiple variables, the initial Array SSA to Lambda phase generates `fold` operators reducing over tuples of those variables. Even after `map` operators are revealed, the operators sometimes still involves large tuples. The vertical fission rules in Figure 3.11 split the operators over tuples into multiple operators going over parts of the original tuples. The `fold` vertical fission rule rewrites a `fold` reducing to a tuple into a tuple of `fold` operators reducing to the same tuple. The `map` vertical fission achieves the same purpose for the input domain. The `map` horizontal fission is the counterpart of the traditional loop fission, splitting a `map` with a $F \circ G$ into $(\texttt{map}\, F)\circ(\texttt{map}\, G)$. The `map` fusion rule is its inverse.

Figure 3.13 shows the set of rules used by Mold to enable the application of other

(transpose zipped)

$$\frac{(\texttt{map}\,\lambda\, K\langle v_0,\ldots,v_n\rangle\,.\,E)(\texttt{zip}\,C_0\ldots C_n)}{(\texttt{map}\,\lambda\, K\langle v_0,\ldots,v'_k,\ldots,v_n\rangle\,.\,E[v'_k/v_k[J]])(\texttt{zip}\,C_0\ldots(\texttt{t}(C_k))[J]\ldots C_n)}$$
$$k \in 1\ldots n \wedge \exists J.v_k[J] \subset E \wedge \text{free}(J) \subset \text{free}(ALL)$$

(lower-dimension-for-update)

$$\frac{\texttt{fold}\,r^0\,\lambda\, r\ k\ V\,.\,r[\langle c,k\rangle\,\texttt{:=}\,E]}{\texttt{fold}(r^0[c])\,\lambda\, r'\ k\ V\,.\,r'[k\,\texttt{:=}\,E[r'[k]/r[\langle c,k\rangle]]}$$
c is loop invariant; all accesses to r in E are $r[\langle c,k\rangle]$

$$\frac{\texttt{fold}\,r^0\,\lambda\, r\ k\ V\,.\,r[\langle k,c\rangle\,\texttt{:=}\,E]}{\texttt{fold}(\texttt{t}(r^0)[c])\,\lambda\, r'\ k\ V\,.\,r'[k\,\texttt{:=}\,E[r'[k]/r[\langle k,c\rangle]]}$$
c is loop invariant; all accesses to r in E are $r[\langle k,c\rangle]$

Figure 3.12: Code motion rules for arrays and collections with multiple dimensions.

rules as well as to simplify computation. They also generate code which uses `++`, an operation shown in Figure 3.5. `++` takes two maps with the same type into a new map which has as entries the union of the maps, and any matching keys taking their value from the second map.

Figure 3.12 shows transformations on operators going over collections with multiple dimensions.

3.8.5 Monoid-based Transformations

Various transformation rules rely on viewing the map data structure as a *monoid*, i.e., a set with an associative binary operator and an identity element. Its identity is an empty map, while its "plus" operation (denoted by $\oplus$) is based on the "plus" of its value type parameters.

The sum of two maps, i.e., $a \oplus b$, is another map with the same keys and, as values, the sum of their values. If a value is missing in either map, it is replaced by the value type's *zero*. At the bottom of Figure 3.5 we give a possible implementation for the $\oplus$ based on `zip`.

Identifying computation that can be seen as operating over monoid structures allows further optimizations, since we can exploit associativity to expose more parallelism.

(eliminate empty fold)

$$\frac{\texttt{fold}\, r_0 (\lambda\, r K V\, .\, r[K := V]) d}{r_0 \texttt{ ++ } d}$$

(eliminate empty map)

$$\frac{\texttt{map} (\lambda\, r V\, .\, V) d}{d}$$

(fold → size)

$$\frac{\texttt{fold}\, r_0\, \lambda\, r\ I\, .\, r + E}{\lambda\, x\, .\, E * (\text{size}\, x) + r_0} \quad \forall i \in I . i \notin E$$

(reduce by key)

$$\frac{(\texttt{map}\ \ \lambda\, K\ V\, .\ \ \text{size}\, V) \circ (\texttt{groupBy}\, \lambda\, P\, .\, E)}{(\text{reduceByKey}\, +) \circ (\texttt{map}\, \lambda\, P\, .(E, 1))}$$

(transpose of transpose)

$$\frac{(\texttt{t} \circ \texttt{t})\ a}{a}$$

(factor out store)

$$\frac{\texttt{fold}\, r'\, \lambda\, r\, i\, .\, r[E := B]}{r'[E := \texttt{fold}\, r'[E]\, \lambda\, k\, .\, B[k/r[E]]]} \quad i \notin E \wedge r \notin E$$

(apply store)

$$\frac{r[I := E_0][I := E_1]}{r[I := E_1[E0/(r[I := E_0][I])]]}$$

Figure 3.13: Simplifying and enabling transformation rules.

(eliminate null check)

$$\frac{\texttt{if } a[k] == null \texttt{ then } 0 \texttt{ else } a[k]}{a[k]} \qquad a \text{ is a monoid with 0 as identity}$$

(identify map monoid plus)

$$\frac{\texttt{map}(\lambda\, i\, x\, y\, .\, x \oplus y)(\texttt{zip}\ A\ B)}{A \boxplus B} \qquad \begin{array}{l} A \text{ and } B \text{ are } \texttt{M}[T] \text{ monoids} \\ \oplus \text{ is the plus for } T \\ \boxplus \text{ is the plus for } \texttt{M}[T] \end{array}$$

(swap map and fold)

$$\frac{(\texttt{fold}\, r_0\, \oplus) \circ (\texttt{map}\, f)}{\lambda\, c\, .(r_0 \oplus f(\texttt{fold } 0_{\boxplus} \boxplus)\ c))} \qquad \begin{array}{l} \texttt{map} \text{ over monoid } \boxplus, 0_{\boxplus} \\ \forall ab.f(a \boxplus b) = f(a) \oplus f(b) \end{array}$$

(flatMap)

$$\frac{(\texttt{fold}\, r_0\, \oplus) \circ (\texttt{map}\, f)}{r_0 \oplus \text{flatMap}\, f} \qquad \texttt{fold} \text{ over the monoid } \oplus, 0_{\oplus}$$

Figure 3.14: Monoid-based rules

Figure 3.14 shows our set of monoid-based transformation rules. The first two rules are "enabling" rules that make code more amenable to other optimizations, while the final rule is itself an optimization.

To illustrate the "eliminate `null` check" rule in Figure 3.14, let us revisit an intermediate expression from the motivating example:

$$\begin{array}{l} \texttt{let}\, prev = m[w]\, \texttt{in} \\ \quad m[w\, \texttt{:=}(\texttt{if } prev == null \texttt{ then } 0 \texttt{ else } prev) + 1] \end{array} \tag{3.17}$$

Here, the conditional block can be eliminated by considering m a monoid with 0 as the identity element. Applying the rule yields:

$$\texttt{let}\, prev = m[w]\, \texttt{in}\ m[w\, \texttt{:=} prev + 1] \tag{3.18}$$

This transformation enables other optimizations by giving the code a more uniform structure.

In Section 3.4 we showed how the inner loop of the `WordCount` code of Figure 3.1 is transformed to a MapReduce form. We now explain the last two rules of Figure 3.14 by showing how they are used to optimize the full loop nest. The inner loop of Figure 3.1 iterates over each line in the input. After beta reduction, and without assuming m is initially empty as we did in Section 3.4, its optimized form is:

$$(\texttt{map}\,\lambda\, k\ v\,.\, m[k] + \texttt{size}\ v) \circ (\texttt{groupBy}\, id) \tag{3.19}$$

Placing this code in the context of the IR for the outer loop yields, after applying some non-monoid rules:

$$\begin{array}{l} \texttt{fold}\, m\, \lambda\, m\ \langle i, doc\rangle\,. \\ \quad \texttt{let}\, do\_count = (\texttt{map}\,\lambda\, k\ v\,.\,\texttt{size}\ v) \circ (\texttt{groupBy}\, id)\ \texttt{in} \\ \quad\quad m\ \texttt{++}\ ((\texttt{map}\,\lambda\, k\langle v_1, v_2\rangle\,.\, v_1 + v_2) \\ \quad\quad\quad\quad (\texttt{zip}\ \ m\ \ (do\_count\ \ (\texttt{split}\ \ doc)))) \end{array} \tag{3.20}$$

We can simplify this program using the "identify map monoid plus" rule in Figure 3.14. The m map and the value of (do_count (`split` doc)) are both maps from strings to numbers. Integer numbers are monoids with arithmetic addition as plus, so our maps can be seen as monoids with the $\oplus$ operator defined in Figure 3.5. Zipping two monoids and adding up their corresponding values, as done above, is simply an implementation of the $\oplus$ operator. Thus, applying the rule rewrites the code to:

$$\begin{array}{l} \texttt{fold}\, m\, \lambda\, m\ \langle i, doc\rangle\,. \\ \quad \texttt{let}\, do\_count = (\texttt{map}\,\lambda\, k\ v\,.\,\texttt{size}\ v) \circ (\texttt{groupBy}\, id)\ \texttt{in} \\ \quad\quad m \oplus (do\_count\ \ (\texttt{split}\ \ doc)) \end{array} \tag{3.21}$$

do_count (`split` doc) does not depend on the m parameter, so can be extracted to a `map` using the "extract `map` from `fold` rule" — see Section 3.8.1. Furthermore, the resulting `map` is split into a composition of maps through fission. The computation

reaches this form:

$$
\begin{aligned}
&\texttt{let}\ \mathit{foldDocCount} = \texttt{fold}\, m\, \lambda\, m\ \langle i, \mathit{docCount}\rangle\,.\\
&\quad m \oplus \mathit{docCount}\ \texttt{in}\\
&\qquad \mathit{foldDocCount} \circ\\
&\qquad (\texttt{map}\, \lambda\, i\ \mathit{groups}\,.(\texttt{map}\, \lambda\, k\ v\,.\,\texttt{size}\ v)\ \mathit{groups}) \circ\\
&\qquad (\texttt{map}\, \lambda\, i\ \mathit{split}\,.\,\texttt{groupBy}\, \mathit{id}\ \mathit{split}) \circ\\
&\qquad (\texttt{map}\, \lambda\, i\ \mathit{doc}\,.\,\texttt{split}\ \ \mathit{doc})
\end{aligned}
\tag{3.22}
$$

While the above computation reveals significant parallelism, the final `fold`, which merges the word count map for each document, is inefficient: it repeats the work of grouping results by word and summing counts. Notice that instead of doing all the operations for each *doc* and merging the results at the end, the program could start by merging all the *doc*s and then computing word counts on the result. The "swap `map` with `fold`" shown in Figure 3.14 achieves this transformation.

"swap `map` with `fold`" rewrites a composition of a `fold` over a monoid of $\texttt{M}[B]$ using the monoid's plus ($\oplus$) with a `map` using function $f : A \to B$ into an application of f to the result of `fold`ing over `map`s input using monoid A's plus ($\boxplus$). The original value r_0 is also $\oplus$-added. Notice that the transformation eliminates the `map` operation, replacing it with a single application of f. Depending on the cost of f, this may result in significant speedups. The operation is correct as long as f distributes over the monoid pluses, i.e., $\forall\ a\ b\ .\ f(a \boxplus b) = f(a) \oplus f(b)$.

All three `map` functions in the above programs have distributive operations. Guided by our cost function, MOLD applies the "swap `map` with `fold`" rule two times and does two reductions of operations with identity. After some restructuring for readability, we reach the following program:

$$
\begin{aligned}
&\texttt{let}\ \mathit{foldBagPlus} = \texttt{fold}\, \texttt{0}_{\texttt{Bag}}\, \lambda\, \mathit{allWords}\ \langle i, \mathit{words}\rangle\,.\\
&\quad \mathit{allWords} \boxplus_{\texttt{Bag}} \mathit{words}\ \texttt{in}\\
&\texttt{let}\ \mathit{mapToSize} = \texttt{map}\, \lambda\, k\ v\,.\,\texttt{size}\ v\\
&\texttt{in}\ \lambda\, \mathit{input}\,.\, m \oplus (\mathit{mapToSize} \circ (\texttt{groupBy}\, \mathit{id}) \circ\\
&\quad \mathit{foldBagPlus}\ \circ (\texttt{map}\, \lambda\, i\ \mathit{doc}\,.\,\texttt{split}\ \ \mathit{doc}))\ \mathit{input}
\end{aligned}
\tag{3.23}
$$

$foldBagPlus$ is the counterpart of $foldDocCount$ from the previous code version

but, instead of merging count maps, it now merges `Bag`s of words. A `map` followed by a folding of the `Bag` is equivalent to a Scala `flatMap` operation, as expressed by the "flatMap" rule in Figure 3.14. Applying the rule bring the above program to:

$$\begin{aligned}
&\texttt{let}\ \mathit{mapToSize} = \texttt{map}\,\lambda\, k\ v\,.\,\texttt{size}\ v \\
&\texttt{in}\ \lambda\, \mathit{input}\,.\, m \oplus (\mathit{mapToSize} \circ (\texttt{groupBy}\, \mathit{id}) \circ \\
&\quad (\texttt{flatMap}\,\lambda\, i\ \ \mathit{doc}\,.\,\texttt{split}\ \ \mathit{doc}))\ \mathit{input}
\end{aligned} \tag{3.24}$$

The `groupBy` generates a `Map` from words to `Bag`s of words, which can be expensive in terms of I/O. As we are only interested in the size of the `Bag`, the program is transformed (by the "reduce by key" rule in 3.13 from Section 3.8.4) to:

$$\begin{aligned}
&\texttt{let}\ \mathit{mapToSize} = \texttt{map}\,\lambda\, k\ v\,.\,\texttt{size}\ v \\
&\texttt{in}\ \lambda\, \mathit{input}\,.\, m \oplus (\text{reduceByKey}\ +) \circ \\
&\quad (\texttt{map}\,\lambda\, i\ \ \mathit{word}\,.(\mathit{word}, 1)) \circ \\
&\quad (\texttt{flatMap}\,\lambda\, i\ \ \mathit{doc}\,.\,\texttt{split}\ \ \mathit{doc}))\ \mathit{input}
\end{aligned} \tag{3.25}$$

The `Bag` of words maps to a `Bag` of $(\mathit{word}, 1)$ pairs. `reduceByKey` groups the "1" values by key (*i.e.*, word) and reduces each group using + operation, which is equivalent to the previous counting but does not generate a `Bag` of identical words for each key (*i.e.*, for each word).

3.9 Implementation

We present some details regarding Mold's implementation by following through the transformation phases in Figure 3.3.

The translation from Java to Array SSA is an extension of the SSA implementation in WALA [14] to handle arrays and collections. The translation from Array SSA to the Lambda IR is implemented in Scala.

For the rewrite system we extended Kiama [15], a strategy-based term rewriting library for Scala. We added support for state exploration, name-bindings aware operations (e.g., "subexpression of"), and cost-guided exploration modulo $\alpha\beta$-conversion.

Mold renames variables where necessary to solve name conflicts, and flattens `let`

expressions to improve performance by the following rule:

$$\frac{\texttt{let}\, x = (\texttt{let}\, y = E_y\, \texttt{in}\, E_x)\, \texttt{in} \ldots}{\texttt{let}\, y = E_y\, \texttt{in}\, (\texttt{let}\, x = E_x\, \texttt{in} \ldots)} \tag{3.26}$$

The flattened `let` constructs translate to cleaner Scala code with less block nesting. MOLD generates Scala code from the lambda calculus IR by syntactic pretty-printing rules like ($\mathcal{S}$ is a function translating to Scala):

$$\mathcal{S}(\lambda V\,.\,F) \rightarrow \texttt{\{}\,\mathcal{S}(V)\texttt{=>}\mathcal{S}(F)\texttt{\}} \tag{3.27}$$

The `let` expressions are transformed to value declarations:

$$\texttt{let}\, X = Y\, \texttt{in}\, Z \rightarrow \texttt{val}\ \ \mathcal{S}(X)\ \texttt{=}\ \mathcal{S}(Y)\ \texttt{;}\ \ \mathcal{S}(Z) \tag{3.28}$$

The Scala code is emitted using Kiama's pretty printing library [41] and Ramsey's algorithm [42] for minimal parenthesization.

The built-in data structures (Figure 3.5) roughly follow Scala collection library's conventions. Code generated by our tool can be executed as traditional Scala code by simple implicit conversions [43]. The same code can be executed either sequentially or using the Scala parallel collections [44]. In a similar manner, we also provide a backend based on the Spark MapReduce framework [12]. This allows programs generated by MOLD to be executed on Hadoop YARN clusters [45].

3.10 Evaluation

We now present an experimental evaluation designed to answer the following research questions:

1. **Can MOLD generate *effective* MapReduce code?**

   We define *effective* code as satisfying three conditions:

   (a) It does not have redundant computation.

   (b) It reveals a high level of parallelism.

   (c) Its accesses to large data structures are localized, enabling execution on distributed-memory MapReduce platforms.

Algorithm	*# loop nests*	*# loops*	*Translation time(s)*	`fold` → `fold` ∘ `map`	`fold` → `groupBy`	*Localization*	*Optimization & Enabling*	*Monoid*	***Total***
WordCount	1	1	11	1	1	3	7	3	15
Histogram	1	1	233	0	3	1	11	3	18
Linear Regression	1	1	28	1	0	1	0	0	2
String Match	1	1	68	1	0	1	0	0	2
Matrix Product	1	3	40	4	0	2	14	0	20
PCA	2	5	66	10	0	1	4	0	15
KMeans	2	6	340	5	0	1	4	0	10

Table 3.1: Evaluation programs and applied transformations

2. **Is Mold *efficient*?**

   We measure how long it takes for Mold to find an effective solution.

3. **Is the proposed approach *general*?**

   We show that the core rewrite rules are general and match many code scenarios. Also, we discuss how our cost optimization approach can generate effective solutions for different execution platforms.

To answer these questions, we study the results of using Mold to translate several sequential implementations of the Phoenix benchmarks [10], a well-established MapReduce benchmark suite which provides both MapReduce and corresponding sequential implementations.

The benchmark suite provides C implementations while our system expects Java code as input. We do a manual, straightforward, syntactic, translation of the C sequential implementations to Java. We transform C `struct` to simple Java data classes, functions to static methods, and arrays to Java arrays. Since Java does not have native multi-dimensional arrays, we use a separate Java class implementing two-dimensional array behavior. Also, because Mold takes a single Java method as input, we inline methods that contain significant parts of the computation.

Furthermore, we manually implemented and tuned each of the benchmarks in Scala to provide a baseline against which we compare the performance of the MOLD-generated code variants (see Section 3.9). We derived three hand-written implementions: the first uses sequential Scala collections, the second uses parallel Scala collections, and the third use Spark [12]. We did not directly use third-party implementations since in some cases they do not exist, and in other cases they introduce optimizations orthogonal to the translation to MapReduce (*e.g.*, they use the Breeze [46] linear algebra library).

To gauge the quality of the code generated by MOLD we pass each program through our tool, we execute the resulting code and hand-written implementations, and we measure and evaluate the results. For each program:

- We measure MOLD's execution time and log the sequence of applied rules to reach the optimal solution.
- We check that the transformations preserved the semantics of the original. program by executing both the original program and the generated one on the same input data set and asserting that the results are the same.
- We manually inspect the code to check whether the operators going over large input data sets have been parallelized, and whether data accesses have been localized.
- We execute the generated code with the three backends described at the end of Section 3.9, and then we execute the hand-written implementations on the same data sets.

When comparing with hand-written implementations, we execute each version of a benchmark on the same dataset five times within the same JVM instance. We report the average of the measurements after dropping the highest and lowest time values. We run the experiments on a quad-core Intel Core i7 at 2.6 GHz (3720QM) with 16 GB of RAM. MOLD's rewrite system state exploration is parallelized.

3.10.1 Can MOLD Generate *Effective* MapReduce Code?

We discuss how MOLD generates MapReduce code for each of the subject programs, and how the generated code compares to hand-written implementations. Table 3.1 shows the number of loops and loop nests in the original programs, and gives the

translation time and a summary of the transformations applied in order to reach the optimal version.

WordCount. Figure 3.1 shows the original sequential code, with the outer loop iterating over documents, and the inner loop iterating over each word in each document and updating a shared map (`counts`).

One of the solutions found by MOLD is a map over all documents followed by a fold. The `map` contains a `groupBy` operation which computes a word count for each document (facilitated by the "`fold` to `groupBy`" rule). The `fold` merges the word counts across documents. While this is a good solution, merging word count maps may be expensive. Using the "swap `map` with `fold`" rule, MOLD moves the `fold` operation up the computation chain. In this way it reaches a form similar to the traditional MapReduce solution for the WordCount problem. The documents are split into words, which are then shuffled (`groupBy`) and the numbers of elements in each word bucket is counted.

The generated code exposes the maximal amount of parallelism for this case, and all accesses are localized so the code is distributable. Appendix 5 lists the transformation steps taken by the tool to reach the solutions discussed above.

Histogram. This benchmark poses a similar challenge to WordCount. It computes the frequency with which each RGB color component occurs within a large input data set. MOLD generates a solution similar to WordCount. It first groups each color component by its values, and then maps to the size of the groups. Given a cost function which puts a higher weight on `groupBy` operations, MOLD can also generate a solution where the input data set is tiled and a `map` goes over each tile to compute its histogram, and the resulting histograms are merged by a final `fold`. This is similar to the approach taken by the Phoenix MapReduce solution.

The generated code is parallel and accesses are localized. The Phoenix implementation assumes the input is a single array encoding the RGB channels. The "`fold` vertical fission" rules split the computation by channel but cannot eliminate the direct stride accesses to the input array. To localize the accesses, the tool assumes the MapReduce implementation has a function `selectEveryWithOffset(k,o)` which takes every `k`th element of the input starting with offset `o`.

Linear Regression and String Match. These two benchmarks are less challenging. MOLD brings them both to a parallelized and localized form by an application of "extract `map` from `fold`" followed by an application of "localize `map` accesses".

Matrix Product. Matrix multiplication is a classic problem with many possible parallel and distributed implementations. The solution reached by MOLD under the cost function targeting MapReduce is a nesting of three `map` operators. Considering the input arrays `a` and `b`, the outer `map` goes over `a`'s rows, the middle `map` goes over `b`-transposed's rows, i.e., it goes over b's columns. The inner operation `zip`s `a`'s row with `b`'s column, then it `map`s with the multiplication operation, and finally it sums up the products. The generated code allows fine grained parallelism and it has good localization of accesses.

PCA and KMeans. Generally, for these benchmark the generated code exposes good parallelism and access localization, but the generated code is not optimal. In PCA, there are three redundant `map`s and one redundant `fold`, out of a total of twelve operators. They are left over due to some limitations of our code motion rules in handling complex tuple structures. In both cases the transformation leaves some additional no-op restructuring of the results at the end of the computation pipeline. Considering that the transformation is source-to-source, a programmer could identify and fix these issues, while still benefiting from having most of the algorithm well parallelized.

Backends

For five of the benchmarks, the Scala generated code type-checks. For the remaining two, we had to add types where MOLD could not infer them due to type erasure in the input Java bytecode. Using the backends described in the Section 3.9, we were able to execute all benchmarks using the Scala collections, and we were able to execute five of the benchmarks on Spark. The remaining two benchmarks, KMeans and PCA, were not executable on Spark due to current limitations of the framework (it does not allow nested distributed data structures).

Comparison with Hand-Written Implementations

Table 3.2 compares the execution time of the generated code and hand-written implementations. Columns 2 and 3 show the execution time of the generated code

program	Scala			
	generated		hand-written	
	sequential	parallel	sequential	parallel
WordCount	42.31	17.38	35.58	14.94
Histogram	9.65	7.17	9.98	6.60
LinearRegression	13.77	11.00	13.08	10.60
StringMatch	8.65	3.81	2.58	0.68
MatrixProduct	8.81	2.05	0.41	0.48
PCA	7.02	14.86	3.53	0.77
KMeans	11.90	40.31	0.61	0.89
program	Spark			
	generated		hand-written	
	1-thread	8-thread	1-thread	8-thread
WordCount	5.99	2.41	5.93	2.42
Histogram	8.84	2.42	8.65	2.59
LinearRegression	1.15	0.62	1.03	0.51
StringMatch	4.78	1.92	4.48	1.57
MatrixProduct	9.02	2.29	5.90	1.58
PCA	-	-	-	-
KMeans	-	-	-	-

Table 3.2: Execution results.

using sequential and parallel Scala collections, respectively. Similarly, columns 4 and 5 show the hand-written implementations using sequential and parallel Scala collections, respectively. Columns 6 and 7 show the results of executing the generated code using Spark with either 1 or 8 hardware threads. Columns 8 and 9 show the corresponding results when executing the hand-written Spark implementations.

The generated versions using parallel Scala collections are between 23% and 80% faster than the generated sequential versions, with the exception of PCA and KMeans which exhibit a slowdown when executed in parallel. The generated code for these two benchmarks was found to contain significant redundant computation that severely impacts the performance of the parallel code.

Comparing the executions of the generated and hand-written versions that use parallel Scala collections, we found the former is 4%-16% slower for WordCount, Histogram, and LinearRegression, 4x-6x slower for StringMatch and MatrixProduct, and at least one order of magnitude slower for PCA and KMeans (for the reason explained earlier).

The 8-thread Spark-based versions of WordCount, LinearRegression, and StringMatch are faster than the Scala-based ones. We believe Spark gains this advantage by caching intermediate results and by using memory mapping for IO. The generated Spark codes for WordCount and Histogram are slightly faster (1% and 7% respectively) than the hand-written Spark versions. The remaining benchmarks are 22%-45% slower. As explained earlier, the generated code for KMeans and PCA relies on nested distributed data structures, which are not currently supported in Spark.

3.10.2 Is Mold Efficient?

Table 3.1 reports the Mold execution time in column 4. For all but one program, the tool reaches the solution in under 4 minutes, and in some cases, Mold is fast enough that it could run interactively, with the programmer getting the translated code within tens of seconds. The outlier is KMeans, which is a larger program compared to the others and has separated nested loops and complex control structure.

3.10.3 Is the Proposed Approach General?

We say that a rewrite rule set is general if it can be used to reach effective solutions for multiple programs, and each good solution depends on the application of multiple rules. Columns 5-9 of Table 3.1 show, for each program, the number of applications for each of the rule groups presented in Section 3.8.

The "extract `map` from `fold`" rule (column 5) is required in all but one of the programs. The "`fold` to `groupBy`" rule (column 6) is used for WordCount and Histogram, the two programs with indirect accesses. Upon inspection, we found that the previous two rules parallelized the computationally expensive sections of the programs. Furthermore, the access localization rules are useful in all cases (column 7).

With the exception of Linear Regression and String Match, all other programs also require the application of classic loop optimization and code motion transformations.

3.11 Related Work

We discuss other approaches to taking code expressed as a non-parallel program and either executing it in parallel or transforming it to a form that can be executed in parallel or on a distributed system.

3.11.1 Inspector-Executor

MOLD exposes parallelism using the `fold` to `groupBy` rewrite that introduces shuffle operations to move data to computation. This particularly benefits imperative loops that update a global data structure with irregular access patterns, as in `WordCount`.

The above computation structure is similar to that of inspector-executor frameworks where, at runtime:

- An *inspector* curates data access patterns for a loop body and determines an ordering for retrieving data values.
- An *executor* fetches the values from remote memory locations in the specified order and executes the loop body.

This model is used for irregular reference patterns over sparse data structures to improve spatial locality and hide memory access latency.

Initial inspector-executor transformations were applied manually [39]. Numerous advances have automated the process and introduced advanced data reordering that combine static and runtime analysis (*e.g.*, [47] and [48]). In [47], the authors showed that sequences of loop transformations (*e.g.*, data reordering, iteration reordering, tiling) can be legally composed at compile time to yield better performance for indirect memory references.

MOLD differs from inspector-executor models in a number of ways. By rewriting loops into a functional style with high-level operators, we retain a level of abstraction that is not available when loops are decomposed into explicit inspector and executor loops. Further, MOLD operators may be mapped to more general parallel execution frameworks that include MapReduce, as we have shown in this paper.

3.11.2 Source-to-Source Transformation

MapReduce offers a programming model for parallel computing that is convenient and scalable, for applications that fit the paradigm. It frees the programmer from the burden of orchestrating communication in a distributed or parallel system, leaving such details to the MapReduce framework which may offer other benefits as well. For this reason, it is often the target of compilation from annotated sequential codes or domain-specific languages. In our work, we aimed to apply a source-to-source transformation to MapReduce directly from unmodified sequential (Java) code.

The prevalence of general purpose GPUs has catalyzed the interest in source-to-source transformations from sequential codes to parallel orchestration languages (*e.g.*, OpenMP, OpenCL) or GPU languages (*e.g.*, CUDA).

In [49] for example, a number of these approaches are evaluated and a new skeleton-based compiler is described. A common theme in these efforts is the reliance on a programmer to identify and annotate their source code to aid the compiler in generating a suitable and correct parallel implementation.

In comparison, MOLD automatically discovers if a loop is suitable for translation into a MapReduce style and applies term rewriting rules to enumerate a number of candidate implementations.

In [38], the author describes a compiler analysis for recognizing parallel reductions. This analysis relies on array dataflow analysis [50] to summarize data that is reachable and modified within a loop, and is applicable when memory aliases can be disambiguated. An important differentiator in our work is the use of `groupBy` which affords the ability to resolve data aliases via MapReduce shuffle operations.

The MOLD internal representation is derived from a program in array SSA form, extending previous observations that a program in SSA form is essentially a functional program [36, 37]. This functional representation is the basis for the transformations described in this dissertation to rewrite imperative loops into a MapReduce style. MOLD leverages the power of functional programming [51] and its algebraic properties [52, 53]. We use many of these properties in the optimization rules described in Section 3.8.

There is also work on refactoring toward parallelism or a more functional form. For example, [54] proposes a refactoring tool to parallelize Java loops, and [55] presents an automated refactoring of Java code to use the Java 8 collection operators. Both approaches transform the original program AST directly and are limited to specific access patterns.

3.11.3 Program Synthesis

An extensive body of work concerns the use of program synthesis techniques to generate efficient code. Particularly relevant to our work is superoptimization, where program forms are enumerated and checked against supplied test cases to find a desired code sequence. This may be exhaustive as in the original superoptimizer, goal-oriented [30], or stochastic [31]. In many of these applications, the context is a peephole optimizer that reorders the instructions of a critical inner loop at ISA-level.

This is also the case for component based program synthesis [56]. In contrast, our work is a source-to-source transformation that applies a much larger scale refactoring to loops from an imperative code sequence to a functional MapReduce style.

Chapter 4

Inferring Program Transformations

In this chapter we present a way to infer program transformations by using pattern generalization, which, to our knowledge, is a novel approach. Previous approaches have either looked at pattern generalization from a purely theoretical perspective, or have taken more ad-hoc approaches to inferring program transformations.

We introduce a novel technique and algorithm for generalization/learning of rewrite rules from multiple transformation examples. More precisely, the algorithm generalizes from term patterns with associative symbols and context variables, as explained above. Further, we show how the pattern generalization algorithm can be used to generalize rewrite rules in a straightforward manner. Finally, we give an optimized implementation together with a thorough evaluation showing the feasibility of the approach on learning fixes for linting tool warnings.

Our algorithm for generalizing from rewrite examples to a rewrite rule is based on two key advances. We begin with the technique of Alpuente et al. [8] for generalizing patterns, albeit in a manner that cannot introduce contexts. Our first advance is to extend this technique to introduce contexts during generalization, in a manner that produces readable results and runs efficiently in practice (Section 4.4). Second, we show that our pattern generalization algorithm can be extended to generalize rewrite rules by simply treating a rewrite rule as a *pair* of patterns, with some post-generalization filtering to ensure the final output is a well-formed rewrite rule (Section 4.10).

The work presented in this chapter was done in collaboration with Manu Sridharan, Satish Chandra, Andrei Ștefănescu, and Grigore Roșu.

4.1 Problem

In this section we give an overview of our technique. We show an example of a useful program transformation and explain what would be necessary to automate this transformation (Section 4.1.1), how our approach works on the example (Section 4.1.2), and why other tools cannot learn or even express this transformation (Section 4.1.3).

4.1.1 What Is Necessary

One class of common, repetitive, and tedious program transformations is fixing the problems reported by various code quality checkers, such as linting tools. One such problem, detected by the popular ESLint [23] linter, is making an assignment within a conditional, e.g., within the condition of an `if` statement. This is a bad programming practice as the assignment, denoted with the `=` operator, can easily be confused with an equality check, denoted with the `==` operator.

Figure 4.1 shows two concrete code examples exhibiting the problem above, and their respective fixes. The code above the rewrite arrow is taken from real-world code and has been automatically marked by ESLint as having the problem. Both snippets of code contain an assignment within the conditional of the `if` statement. One way to fix this problem is to hoist the assignment to just before the `if` statement. The code snippets below the rewrite arrow show the respective fixes.

This fix is correct as long as the expressions in the conditional before the assignment do not have side effects which would affect the assigned expression. Checking or proving this correctness property is beyond the scope of this work but, as we will discuss in Section 4.9, our approach does lend itself to verification.

Before discussing inference of transformation rules, let us consider what is needed to even express a transformation rule that handles both of the examples in Figure 4.1:

(i) The rule must be able to match syntactic elements (i.e., parts of the AST), like the "then" branch of the `if` statement, and preserve them after the rewrite.

(ii) The transformation must be able to replace the content of a syntactic element, like the condition of the `if` statement. Crucially, this replacement must be able to preserve immediately-surrounding code not relevant to the transformation; e.g., the conjuncts `hooks && 'set' in hooks && notxml` in the conditional of the left example in Figure 4.1 should be preserved, even though the assignment is extracted.

To express such transformations, we need rules that use *context variables*, capable of matching over arbitrary contexts, like in Felleisen and Hieb [9] and supported by several modern rewriting/reduction engines specialized for programming language syntax and transformation, such as PLT-Redex [18], K [19], and Spoofax [20]. Consequently, our learning algorithm is capable of learning rules with not only associative symbols, but

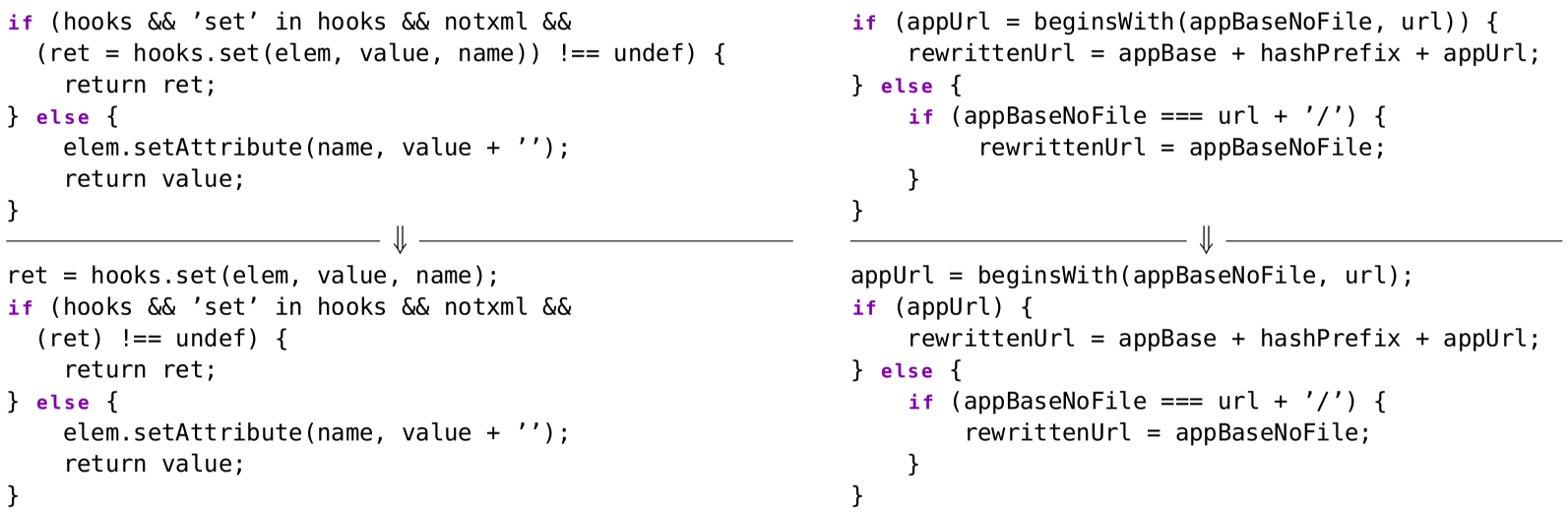


Figure 4.1: Before-and-After Code Examples Of Hoisting an Assignment Out of an `if` Conditional

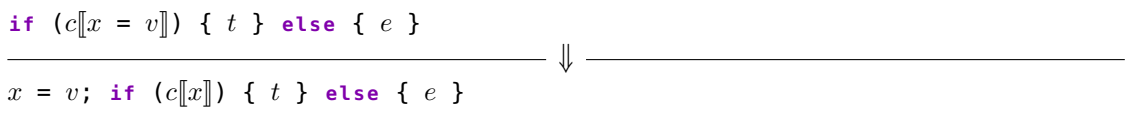


Figure 4.2: Program Transformation Rule Example

also with context variables. Then we show that these features allow us to learn complex and powerful transformations directly from ground transformation examples.

Eelco Visser [57] gives a taxonomy of program transformations, and surveys rewriting strategy approaches. Relevant to our approach are ELAN [58], Maude [59], and Stratego [60], which are the inspiration for our strategy language. Related to the evolutionary approach, Bournez and Kirchner [61] discuss the theoretical aspects of extending ELAN with a probabilistic choice strategy.

4.1.2 How We Express the Transformation

Matching Logic provides a concise and easy-to-read way to express the transformation. The rewrite rule in Figure 4.2 encodes the fix for the above issue. The expression before the rewrite (i.e., above the bar), is the matched pattern. The italicized variables are *pattern variables* that can match any term, where a term is a node in the AST. The expression below the rewrite (i.e., below the bar), is the pattern for the rewritten term. We see the same variables, but the code has been restructured. The body of the `if` is preserved through the t variable, exemplifying how the rewriting language meets the (i) requirement above. Furthermore, as explained below, the assignment is hoisted before the loop while the conditional is changed appropriately, satisfying the (ii) requirement.

Variable c is an "anywhere" context variable and matches any term that contains, at any depth within, at least one match of x `=` v. The assignment matched inside c is hoisted before the loop. The conditional of the `if` remains unchanged, with the exception of the assignment expression which is replaced by just x. The change to the content of $c[\![]\!]$ encodes this transformation: the subterm matched by the pattern inside $c[\![x = v]\!]$ is replaced by whatever is matched by x. Looking at the example on the LHS of Figure 4.1, x matches `ret = hooks.set(elem, value, name)`, and, after the rewrite, the rest of the condition is preserved while the assignment is replaced by just `ret`. Also, the assignment is moved before the `if` statement.

4.1.3 Why Previous Approaches Fall Short

To our best understanding, previous approaches fall short of inferring, or even of expressing, the transformation above.

Several lines of work express transformations as AST manipulating scripts. Kim et al. [62] proposed inferring desired transformation as a composition from a set of predefined transformation scripts. Meng et al. [1, 21, 22] advanced a line of work in which they infer transformations which are represented as an abstraction over a sequence of fine-grained insert/delete/update AST operations, coupled with data-flow conditions. Negara et al. [2, 3] propose a different approach that is also based on sequences of fine-grained AST operations, but the operations are recorded directly from the IDE, and they use a data mining technique to identify interesting sequences. While all of these techniques have shown success, their representation of program transformations as sequences of fine-grained changes has several limitations.

In terms of expressing the transformation, previous approaches representing transformations as fine-grained AST edit sequences [1, 2, 3, 16, 17, 21, 22] do have a way of meeting the (i) requirement described at the end of Section 4.1.1, i.e., they do have a concept of matching part of the AST and preserving it, possibly in a different structure, after the transformation. For Figure 4.1, we believe these other approaches could express the example transformation on the RHS. However, the example on the LHS is trickier, as the assignment is deeper within the conditional. As explained in Section 4.1.1, having a transformation that covers both examples requires a way of traversing over the irrelevant parts of the conditional in a general way.

More specifically, these previous techniques could attempt to express the edits for this transformation as an AST node update x `=` $v \Rightarrow x$ followed by an insertion $\epsilon \Rightarrow x = u$ (we use our rewrite language for exemplification but the other approaches have various different representations). The first fine-grained transformation could possibly be applied correctly, provided other mechanisms ensure x and v only match within a conditional — e.g., data flow techniques in the work of Meng et al. [1, 21, 22]. However, the latter insertion transformation must apply only before the `if` statement where the first transformation was applied. To our understanding, none of the approaches based on transformation sequences can encode this restriction.

In terms of learning, the fine-grained transformation sequence approach is vulnerable to small changes in the input sequences used for learning. Meng et al. [1] note this problem as a threat to both the performance (i.e., can the algorithm generalize from

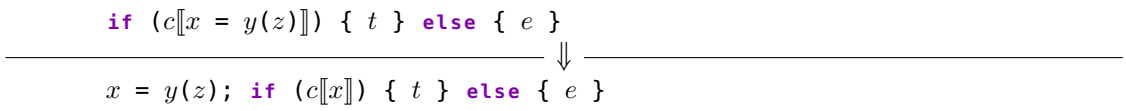


Figure 4.3: Rule Inferred from the Examples in Figure 4.1

the examples?) and correctness (i.e., is the generalization correct?) of their algorithm, and Negara et al. [3] also mention it as a challenge for performance.

Finally, the fine-grained code changes coupled with custom match heuristics are hard to present to the user in a form that is intuitive to understand and easy to adapt. The non-standard representation also makes it difficult to establish any correctness properties in the future.

Alpuente et al. [8] propose a generalization approach capable of inferring term patterns with associative and commutative functions. Their approach is principled and the correctness properties of their algorithm are formally proved. Still, because their language has no equivalent of our contexts, it cannot handle cases such as our motivating example.

4.2 Our Approach: Anti-Unification

The key contribution of our work is an algorithm for inferring powerful rules like that of Figure 4.2 from examples. While developers could write such rewrite rules by hand, having a system suggest them could significantly ease adoption. Writing a rewrite rule by hand requires familiarity with both the rewriting language and the AST structure of the language. For example, the initializer, condition, and step of a `for` loop must be explicitly initialized as `for(`x`, `y`, `z`)` even when they are not relevant for the rewrite and they are simply preserved, as the JavaScript AST structure must be preserved. Furthermore, our system can be used in an interactive environment, e.g., monitoring code edits and proposing a rule on the fly when similar changes are being applied.

Starting with the two examples in Figure 4.1, there are infinitely many way to generalize them to a rewriting rule in our language. Interesting for us are the *most specific* generalizations (abbreviated as "MSGs" from here on), i.e., generalizations which preserve the behavior of the original rewrite rules and transform a minimum of other terms. Given a MSG, it is straightforward to generalize further by collapsing sub-patterns into either contexts or variables. Hence, the MSGs are a compact representation of all

possible generalizations, much like the compact version-space algebra representation in other programming-by-example work [63, 64]. Our ability to cleanly separate the necessary *most-specific* generalization from further *heuristic* generalization also distinguishes our technique from previous work.

One of the possible MSGs for the examples in Figure 4.1 is shown in Figure 4.3. This rule is more specific than the one in Figure 4.2, as it only matches assignments where the right-hand side is a method call, expressed by $y(z)$. It is possible to obtain the more general rule in Figure 4.2 by directly generalizing the Figure 4.3 rule, i.e., by collapsing the method call to a variable. In order to obtain the Figure 4.2 rule, the developer can now either adjust the rewriting rule directly as they see fit, or they can give the tool an additional example to help it generalize further.

Algorithmic Challenges and Contributions

No extant technique can infer the MSGs described above. The closest technique is that of Alpuente et al. [8], which can generalize from pairs of patterns that may include associative operators. Alpuente et al.'s technique lacks two key features that we need:

1. The generalization must be able to introduce contexts, in order to infer rules like that of Figure 4.2.

2. In our scenario, rather than just generalizing patterns used for matching, we must generalize rewrite rules.

To address the first issue, we extend Alpuente et al.'s technique to introduce contexts in a manner that explores all possible nestings between pairs of patterns. Further, we give techniques that simplify away unnecessary patterns introduced during this process. We prove that this extended technique computes the MSGs expressible in our language. We also describe optimizations that improve scalability of this technique in practice.

To extend the generalization to rewrite rules, we make the key observation that a rewrite rule can itself be viewed as a pattern, consisting of two sub-patterns (the left-hand side and right-hand side) connected by the symbol $\Rightarrow$. Treating rewrite rules as patterns allows applying the pattern generalization algorithm directly: it discovers common generalizations computed for the LHS and RHS patterns and ensures the same pattern variable is used in both cases. Further, we describe how to handle cases where the algorithm output does not have an obvious interpretation as a rewrite rule, e.g., if some variables appear only on the RHS of the rule.

$$
\begin{aligned}
\text{all sorts} &::= \text{Variable} \mid \text{Context} \\
\text{Pattern} &::= \text{top sort} \\
\text{Variable} &::= a, b, c, ... \\
\text{Context} &::= \text{Variable}[\![\text{Pattern}]\!] \\
\text{RewritingRule} &::= \text{Pattern} \Rightarrow \text{Pattern}
\end{aligned}
$$

Figure 4.4: Rewriting Language

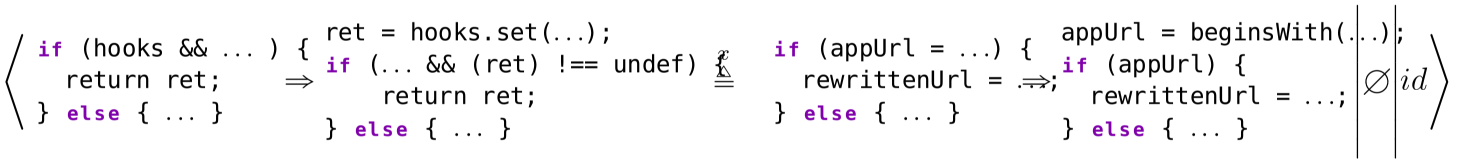
In the following sections we present our rules for computing the MSGs. We first show our rewriting language (Section 4.3), then talk about the structure of our generalization transition system (Section 4.4), then explain the transition system's inference rules (Sections 4.5, 4.6, and 4.7), and then talk about how the final solutions are obtained from the configurations of the transition system (Section 4.8). In Section 4.10 we use these generic rules for the specific task of inferring program transformations.

4.3 Rewriting Language

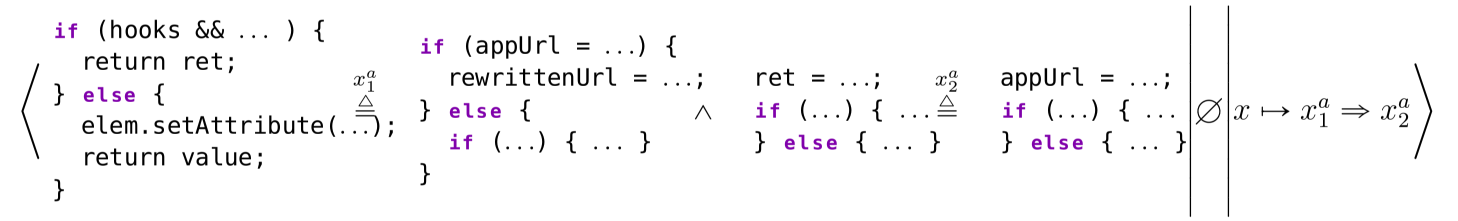
Figure 4.4 sketches our pattern matching and rewriting language. The language extends the syntax of an underlying programming language by allowing pattern-matching Variables (distinguished by italics) or Contexts to appear anywhere in a syntax tree, as indicated by the first production for "all sorts". A pattern is simply the "top sort," i.e., any valid syntactic construct in the programming language extended with Variables and Contexts.

We restrict Contexts [9] to *anywhere* contexts, as explained below. $c[\![p]\!]$ is a context pattern which allows p to match anywhere within the current term. The match is successful if there exists at least one such match. On success, the context variable c is bound to the current term with the inner term replaced by a "hole" marker, denoted by a special variable $\square$. For example, $c[\![\text{bar}(x)]\!]$ matches foo(bar(buz)) with substitution $c \mapsto \text{foo}(\square) \wedge x \mapsto \text{buz}$. When the inner pattern p matches in multiple places, each match generates a distinct solution. For example, $c[\![\text{bar}(x)]\!]$ matches foo(bar(bar(buz))) with two solutions: $c \mapsto \text{foo}(\square) \wedge x \mapsto \text{bar(buz)}$ and $c \mapsto \text{foo(bar}(\square)) \wedge x \mapsto \text{buz}$.

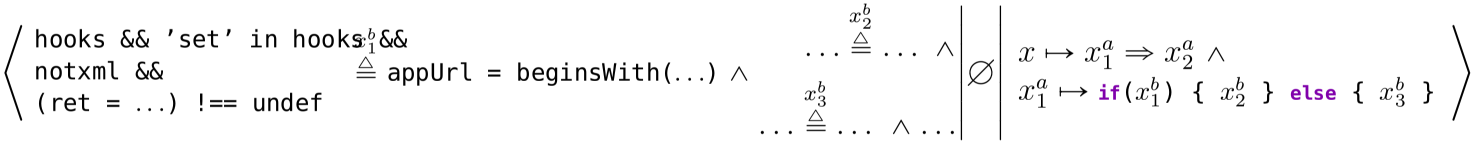
A rewriting rule consists of a left-hand-side and right-hand-side pattern, with the standard meaning: when a term matches the left-hand side pattern, rewrite it to the right-hand side pattern, preserving the variable bindings from the LHS. Rewrite rules



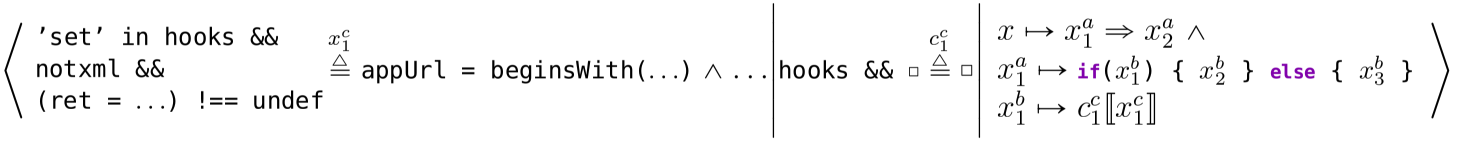
(a) ↓ solve x by *Decomposing* on ⇒ (rule shown in Figure 4.8 and explained in Sec 4.5)



(b) ↓ solve x_1^a by *Decomposing* on the `if` statement (rule shown in Figure 4.8 and explained in Sec 4.5)



(c) ↓ solve x_1^b by *Context (left)* (rule shown in Figure 4.9 and explained in Sec 4.6)



see Figure 4.6 ...

Figure 4.5: Motivating Example Walkthrough. The fresh variable names follow the naming conventions from the inference rules in Figures 4.8, 4.9, 4.10, and, to preserve uniqueness, have as superscripts the letter label of the inference step. We are eliding, via dots, uninteresting parts of the configurations.

. . . from Figure 4.5

(d,e,f,g) ↓ after four more *Context (left)* inference steps (rule shown in Figure 4.9 and explained in Sec 4.6)

$$\left\langle \begin{array}{l} \texttt{ret = hooks.set(...)} \overset{x_1^g}{\triangleq} \\ \texttt{appUrl = beginsWith(...)} \\ \wedge \ldots \end{array} \middle| \begin{array}{l} \texttt{hooks \&\& } \square \overset{c_1^c}{\triangleq} \square \wedge \texttt{'set' in hooks \&\& } \square \overset{c_1^d}{\triangleq} \square \wedge \\ \texttt{notxml \&\& } \square \overset{c_1^e}{\triangleq} \square \wedge \square \texttt{ !== undef} \overset{c_2^f}{\triangleq} \square \wedge (\square) \overset{c_1^g}{\triangleq} \square \end{array} \middle| \begin{array}{l} x \mapsto x_1^a \Rightarrow x_2^a \wedge \\ x_1^a \mapsto \texttt{if(}x_1^b\texttt{) \{ }x_2^b\texttt{ \} else \{ }x_3^b\texttt{ \}} \wedge \\ x_1^b \mapsto c_1^c[\![x_1^c]\!] \wedge x_1^c \mapsto c_1^d[\![x_1^d]\!] \wedge \\ x_1^d \mapsto c_1^e[\![x_1^e]\!] \wedge x_1^e \mapsto c_2^f[\![x_2^f]\!] \wedge \\ x_2^f \mapsto c_1^g[\![x_1^g]\!] \end{array} \right\rangle$$

(h,i) ↓ solve x_1^g by *Decomposing* on the assignment operator (=), followed by applying the *0-arity* rule (Figure 4.6) on x_1^h

$$\left\langle \texttt{hooks.set(...)} \overset{x_2^h}{\triangleq} \texttt{beginsWith(...)} \wedge \ldots \middle| \texttt{ret} \overset{x_1^h}{\triangleq} \texttt{appUrl} \wedge \ldots \middle| \cdots \wedge x_1^g \mapsto x_1^h \texttt{ = } x_2^h \right\rangle$$

(. . . k, . . . , l) ↓ after many similar steps starting on the RHS of the rewrite, i.e., starting with x_2^a

$$\left\langle \ldots \middle| \texttt{ret} \overset{x_1^h}{\triangleq} \texttt{appUrl} \wedge \texttt{ret} \overset{x_1^l}{\triangleq} \texttt{appUrl} \wedge \texttt{hooks \&\& } \square \overset{c_1^c}{\triangleq} \square \wedge \texttt{hooks \&\& } \square \overset{c_1^k}{\triangleq} \square \middle| \ldots \right\rangle$$

(m,n) ↓ identify via *Renaming* (Figure 4.10) regular variables x_1^h with x_1^l and context variables c_1^c with x_1^k
(note that x_1^h and c_1^c come from the LHS of the rewrite rules and x_1^l and x_1^k come from the RHS;
also, the constraints and substitutions on the second line are only interesting for the next inference step)

$$\left\langle \cdots \middle| \begin{array}{l} \texttt{ret} \overset{x_1^h}{\triangleq} \texttt{appUrl} \wedge \texttt{hooks \&\& } \square \overset{c_1^c}{\triangleq} \square \texttt{ } \wedge \texttt{'set' in hooks \&\& } \square \overset{c_1^d}{\triangleq} \square \wedge \ldots \end{array} \middle| \begin{array}{l} \cdots \wedge x_1^l \mapsto x_1^h \wedge x_1^k \mapsto x_1^c \wedge \\ x_1^b \mapsto c_1^c[\![x_1^c]\!] \wedge x_1^c \mapsto c_1^d[\![x_1^d]\!] \end{array} \right\rangle$$

see Figure 4.7 . . .

Figure 4.6: Motivating Example Walkthrough (continuation

... from Figure 4.6

(o) ↓ flattening contexts c_1^c and c_1^d via the *Context of Context* inference rule (shown in Figure 4.10 and explained in Sec 4.7)

$$\left\langle \cdots \;\middle|\; \texttt{hooks \&\& 'set' in hooks \&\& } \square \overset{c_1^c}{\triangleq} \square \wedge \ldots \;\middle|\; \cdots \wedge x_1^b \mapsto c_1^c[\![x_1^d]\!] \right\rangle$$

(...) ↓ many steps

$$\left\langle \varnothing \;\middle|\; \begin{array}{l} \texttt{hooks \&\& 'set' in hooks \&\& notxml \&\& (}\square\texttt{) !== undef} \overset{c_1^c}{\triangleq} \square \wedge \\ \texttt{ret} \overset{x_1^h}{\triangleq} \texttt{appUrl} \wedge \texttt{hooks.set} \overset{x_1^j}{\triangleq} \texttt{beginsWith} \wedge \ldots \end{array} \;\middle|\; \begin{array}{l} x \mapsto x_1^a \Rightarrow x_2^a \wedge \\ x_1^a \mapsto \texttt{if(}x_1^b\texttt{) \{ }x_2^b\texttt{ \} else \{ }x_3^b\texttt{ \}} \wedge \\ x_1^b \mapsto c_1^c[\![x_1^g]\!] \wedge x_1^g \mapsto x_1^h \texttt{ = } x_2^h \wedge x_2^h \mapsto x_1^j\texttt{(}x_2^j\texttt{)} \wedge \\ x_2^a \mapsto x_1^u \texttt{ ; } x_2^u \wedge x_1^u \mapsto x_1^l \texttt{ = } x_2^l \wedge x_1^l \mapsto x_1^h \wedge \ldots \end{array} \right\rangle$$

Figure 4.7: Motivating Example Walkthrough and Final Configuration.

only match and rewrite the entire term, i.e., the rule itself is not applied *anywhere* within a term.

In subsequent sub-sections, we describe an algorithm for generalizing from a pair of patterns. To generalize from a pair of rewrite rules, one can simply treat each rewrite rule as a pattern, with $\Rightarrow$ treated as a regular, non-associative symbol. The final output of the algorithm is then post-processed as explained in Section 4.9 to obtain the final solution.

4.4 Transition System

We propose an algorithm for computing a set of *most-specific generalizations*, *MSGs*, of two patterns from the language of Figure 4.4.

The generalization relation extends naturally to rewrite rules by seeing their LHSs and RHSs as a pair. In general, a pair of patterns may not have a unique most-specific generalization in our language, and hence our algorithm computes the set of *all* such generalizations.

Similar to Alpuente et al. [8], we specify the computation of MSGs using a transition system over configurations, with the transition relation given by a set of inference rules.

Each configuration is a triple of the form $\langle C \mid S \mid \theta \rangle$. C is a conjunction of unsolved *generalization constraints*. A generalization constraint is a triple $t_1 \overset{x}{\triangleq} t_2$ meaning that patterns t_1 and t_2 must be generalized, with pattern variable x mapped to the resulting generalized pattern in θ. S holds the solved generalization constraints, i.e., constraints that have been solved yet are still required for the simplification steps in Section 4.7.

Constraints which have been solved yet are not required for other rules are discarded by the inference rules. θ holds the generalization (partial) solution as an substitution, a mapping from pattern variables to patterns.

To compute generalizations for terms t_1 and t_2, the initial configuration is $\langle t_1 \overset{x}{\triangleq} t_2 \mid \varnothing \mid id \rangle$, where id is the identity, i.e., empty, substitution.

Generalizing from our two examples in Figure 4.1 , t_1 is the entire transformation on the left, *including* the rewrite symbol ($\Rightarrow$), and t_2 is the entire transformation on the right.

The generalization is done via two sets of inference rules. The first set, shown in Figures 4.8 and 4.9 and explained in Sections 4.5 and 4.6, perform the actual generalization. For readability, each configuration is represented on three lines, with

each of its components occupying one line. The side conditions correspond to the assumption of the inference rule.

Each rule advances the system from one configuration to one or more possible subsequent configurations. The rules in this set are confluent, i.e., all application orders lead to the same result. The algorithm terminates when no more rules apply, at which point the rules from the second set are applied.

The second set of inference rules, shown in Figure 4.10, identifies identical subpatterns and eliminates equivalent contexts (see Section 4.7). The final solutions are computed from θ in each final configuration (see Section 4.8).

Throughout the following sections we will refer to Figures 4.5, 4.6, and 4.7, which show part of the inference steps taken by our tool to generalize the rewrite rule in Figure 4.3 from the two transformations in Figure 4.1.

4.5 Preliminary Inference Rules

In this section we explain the inference rules for for generalization that do not introduce contexts. These rules are shown in Figure 4.8. (Rules that introduce contexts are discussed in Section 4.6.) The rules in this subsection are similar to the ones proposed by Alpuente et al. [8], but they are not sorted and the variable inference rule is missing completely, as it is replaced by our rules that introduce contexts (see Section 4.6).

The Decompose rule in Figure 4.8 handles the generalization of two patterns generated by the same non-associative function symbol f. The unsolved constraint is replaced with unsolved constraints for each pair of subterms, in order, and a binding from x to $f(x_1, ..., x_n)$ is added to the substitution. Inference step (b) in Figure 4.5 shows an instance of such a decomposition. The f symbol is the `if` statement, and the rule generates three new constraints marked by variables x_1^b, x_2^b and x_3^b, and adds a substitution to wire them to the solution.

The Associative rule handles the cases where the common symbol generating the terms is associative. We show the rule for left associativity; a symmetric rule for right associativity is omitted for brevity. For JavaScript, lists of expressions and sequences of statements are considered associative.

Given an associative binary operator f, we use $f(x_1, \ldots, x_n)$ as a shorthand representation for all equivalent ways of applying f to x_1 to x_n while maintaining order. The rule

(decompose)

$$C \wedge f(t_1, \ldots, t_n) \stackrel{x}{\triangleq} f(t'_1, \ldots, t'_n) \quad C \wedge t_1 \stackrel{x_1}{\triangleq} t'_1 \wedge \cdots \wedge t_n \stackrel{x_n}{\triangleq} t'_n$$
$$S \rightarrow S$$
$$\theta \quad \theta \wedge x \mapsto f(x_1, \ldots, x_n)$$

f is not associative
$x_1 \ldots x_n$: fresh variables
$n > 0$

(associative, left)

$$C \wedge f(t_1, \ldots, t_n) \stackrel{x}{\triangleq} f(t'_1, \ldots, t'_m) \quad C \wedge f(t_1, \ldots, t_k) \stackrel{x_1}{\triangleq} t'_1 \wedge f(t_{k+1}, \ldots t_n) \stackrel{x_2}{\triangleq} f(t'_2, \ldots t'_m)$$
$$S \rightarrow S$$
$$\theta \quad \theta \wedge x \mapsto f(x_1, x_2)$$

f is associative
$k \in \{1, \ldots, n-1\}$
x_1, x_2: fresh variables
$n \geqslant 2$ and $m \geqslant 2$

(identity expansion, right)

$$C \wedge f(t_1, \ldots, t_n) \stackrel{x}{\triangleq} t \quad C \wedge (f(t_1, \ldots, t_n) \stackrel{x}{\triangleq} f(e, t) \vee f(t_1, \ldots, t_n) \stackrel{x}{\triangleq} f(t, e))$$
$$S \rightarrow S$$
$$\theta \quad \theta$$

f is associative
with identity e
$t \neq f(\ldots)$ and $t \neq e$
$n \geqslant 2$

Figure 4.8: Generalization Inference Rules for Matching Without Contexts. Explained in Section 4.5.

works by splitting generalization constraint in two, one generalizing a prefix of the left term with the head of the right term ($f(t_1, \ldots, t_k) \overset{x_1}{\triangleq} t'_1$), and the other generalizing the postfix of the left term with the tail of the second term ($f(t_{k+1}, \ldots t_n) \overset{x_2}{\triangleq} f(t'_2, \ldots t'_m)$). The constraints are connected through the $x \mapsto f(x_1, x_2)$ substitution. When the associative symbol f has n children, we have $n - 1$ direct subsequent configurations.

To illustrate how the rule works, consider the following configuration involving instruction sequences:

$$\langle \texttt{i1;i2;i3} \overset{x}{\triangleq} \texttt{i1;i3} \mid \varnothing \mid id \rangle \tag{4.1}$$

The `;` operator is associative, so the Associative rule can be applied yielding two subsequent configurations:

$$\langle \texttt{i1} \overset{x_1}{\triangleq} \texttt{i1} \wedge \texttt{i2;i3} \overset{x_2}{\triangleq} \texttt{i3} \mid \varnothing \mid x \mapsto x_1\texttt{;}x_2 \rangle \tag{4.2}$$

$$\langle \texttt{i1;i2} \overset{x_1}{\triangleq} \texttt{i1} \wedge \texttt{i3} \overset{x_2}{\triangleq} \texttt{i3} \mid \varnothing \mid x \mapsto x_1\texttt{;}x_2 \rangle \tag{4.3}$$

The last rule in Figure 4.8, and a symmetrical one not shown for brevity, expands the generalization of terms with associative symbols to handle identity. In conjunction with the Associative rule, this leads, when generalizing from lists of expressions or sequences of statements, to a pattern which can "ignore" a sublist or subsequence.

Looking again at the example above, generalizing from $\texttt{i1;i2;i3} \overset{x}{\triangleq} \texttt{i1;i3}$ by only using the associative rule eventually yields the patterns $\texttt{i1;}e$ and $e\texttt{;i3}$, i.e., a sequence either starting with `i1` or ending with `i3`. A more specific pattern would be $\texttt{i1;}e\texttt{;i3}$, i.e., a sequence of instructions starting with `i1`, ending with `i3`, and allowing any other instructions in the middle.

To obtain the more specific pattern, we apply the Identity rule to the first constraint of the second configuration above, i.e., to $\texttt{i1;i2} \overset{x_1}{\triangleq} \texttt{i1}$. This leads to the following two configurations:

$$\langle \texttt{i1;i2} \overset{x_1}{\triangleq} e\texttt{;i1} \wedge \ldots \mid \varnothing \mid x \mapsto x_1\texttt{;}x_2 \rangle \tag{4.4}$$

$$\langle \texttt{i1;i2} \overset{x_1}{\triangleq} \texttt{i1;}e \wedge \ldots \mid \varnothing \mid x \mapsto x_1\texttt{;}x_2 \rangle \tag{4.5}$$

The second configuration, after a further Associative split, leads to $\texttt{i1;}x_{12}\texttt{;i2}$, where x_{12} is the generalization $\texttt{i2} \overset{x_{12}}{\triangleq} e$. To avoid non-termination, the Identity rule is applied lazily.

(context, right)

$$\begin{array}{lcl} C \wedge f(\dots) \overset{x}{\triangleq} g(t_1, \dots, t_k, \dots, t_n) & & C \wedge f(\dots) \overset{x_k}{\triangleq} t_k \\ & S & \rightarrow \quad S \wedge \square \overset{c_k}{\triangleq} g(t_1, \dots, t_{k-1}, \square, t_{k+1}, \dots, t_n) \\ & \theta & \theta \wedge x \mapsto c_k[\![x_k]\!] \end{array}$$

$k \in \{1, \dots, n\}$
c_k, x_k: fresh variables
$n > 0$

(context, left)

$$\begin{array}{lcl} C \wedge f(t_1, \dots, t_k, \dots, t_n) \overset{x}{\triangleq} g(\dots) & & C \wedge t_k \overset{x_k}{\triangleq} g(\dots) \\ & S & \rightarrow \quad S \wedge f(t_1, \dots, t_{k-1}, \square, t_{k+1}, \dots, t_n) \overset{c_k}{\triangleq} \square \\ & \theta & \theta \wedge x \mapsto c_k[\![x_k]\!] \end{array}$$

$k \in \{1, \dots, n\}$
c_k, x_k: fresh variables
$n > 0$

(0-arity)

$$\begin{array}{lcl} C \wedge t \overset{x}{\triangleq} t' & & C \\ & S & \rightarrow \quad S \wedge t \overset{x}{\triangleq} t' \\ & \theta & \theta \end{array}$$

t and t' are constants
or have arity zero

Figure 4.9: Generalization Inference Rules for Contexts. Explained in Section 4.6.

4.6 Inference Rules for Contexts

In this section we explain the inference rules for generalization via introduction of contexts, shown in Figure 4.9.

The first two rules, which are symmetrical, are the central inference rules of our system. They handle the case where the terms are generated by different symbols f and g, and at least one of them has subterms.

The first rule generalizes the two patterns to a context $c_k[\![x_k]\!]$, where x_k is some generalization of the f pattern. For this generalization to be valid, x_k must also match within some sub-term the g term. Hence, for each sub-term t_k of the g term, the rule creates a distinct configuration with a new generalization constraint $f(\dots) \overset{x_k}{\triangleq} t_k$.

On the RHS, it introduces a context $g(t_1, \ldots, t_{k-1}, \square, t_{k+1}, \ldots, t_n)$, which replaces the subterm with a context hole ($\square$) while leaving the rest of the term intact. This encodes the structure of the term that has been traversed to reach the subterm. The rule does not process the LHS term at all, representing this no-op by an identity context, i.e., by just $\square$.

Consider the following configuration, obtained from example in Figure 4.1 when generalizing, within the `if` on the right, from `(ret = ...) !== undef` on the left and the assignment:

$$\langle \texttt{(ret = ...) !== undef} \overset{x}{\triangleq} \texttt{appUrl = ...} \mid \varnothing \mid id \rangle \tag{4.6}$$

The inequality check on the left has subterms: `(ret = ...)` and `undef`. For each of the subterms, the Context (left) rule creates a distinct new configuration for the case where the new context has a hole for that subterm:

$$\langle \texttt{(ret=...)} \overset{x_1}{\triangleq} \texttt{appUrl=...} \mid \square \texttt{!== undef} \overset{c_1}{\triangleq} \square \mid x \mapsto c_1[\![x_1]\!] \rangle \tag{4.7}$$

$$\langle \texttt{undef} \overset{x_2}{\triangleq} \texttt{appUrl=...} \mid \texttt{(ret=...) !==}\square \overset{c_2}{\triangleq} \square \mid x \mapsto c_2[\![x_2]\!] \rangle \tag{4.8}$$

The second configuration will eventually be pruned (see Section 4.8), as $\texttt{undef} \overset{x_2}{\triangleq} \texttt{appUrl =...}$ leads to a less specific generalization than the first configuration.

Steps (c) to (g) in Figures 4.5 and 4.6 show the generalization from the entire conditions of the two examples by repeatedly applying the context rules. Enough contexts have been generated that the remaining generalization constraint is over the two corresponding assignments. We will show a rule in Section 4.7 that can remove some of the unnecessary contexts in the configuration.

The third rule in Figure 4.9 handles the case where the terms are either constants or have no children. In this case, we simply generalize the terms to an unconstrained variable x.

4.7 Rules for Renaming and Context Flattening

Figure 4.10 shows the rules for variable renaming and for flattening contexts that represent identical patterns in our language.

The first rule identifies variables (both regular and context) matching identical content by introducing a rename, i.e., a substitution from variable to variable, and removing the redundant solved constraint. As explained in Section 4.9, this is crucial for inferring

(renaming)

$$\begin{array}{rcl} C & & C \\ S \wedge t \overset{x_1}{\triangleq} t' \wedge t \overset{x_2}{\triangleq} t' & \rightarrow & S \wedge t \overset{x_2}{\triangleq} t' \\ \theta & & \theta \wedge x_1 \mapsto x_2 \end{array}$$

(context of context)

$$\begin{array}{rcl} C & & C \\ S \wedge t_1 \overset{c_1}{\triangleq} t'_1 \wedge t_2 \overset{c_2}{\triangleq} t'_2 & \rightarrow & S \wedge t_1[t_2/\Box] \overset{c_1}{\triangleq} t'_1[t'_2/\Box] \\ \theta \wedge x_1 \mapsto c_1[\![x_2]\!] \wedge x_2 \mapsto c_2[\![x_3]\!] & & \theta \wedge x_1 \mapsto c_1[\![x_3]\!] \\ \quad \forall s \in \theta \,.\, c_2 \in s \Rightarrow & & \\ \quad\quad \exists x_2, x_3.\ s = x_2 \mapsto c_2[\![x_3]\!] \wedge & & \\ \quad \forall p \in \theta \,.\, p \neq s \wedge x_2 \in p \Rightarrow & & \\ \quad\quad \exists x_1 . p = x_1 \mapsto c_1[\![x_2]\!] & & \end{array}$$

(context of variable)

$$\begin{array}{rcl} C & & C \\ S \wedge t_c \overset{c}{\triangleq} t'_c \wedge t_2 \overset{x_2}{\triangleq} t'_2 & \rightarrow & S \wedge t_c[t_2/\Box] \overset{x_1}{\triangleq} t'_c[t'_2/\Box] \\ \theta \wedge x_1 \mapsto c[\![x_2]\!] & & \theta \\ c \notin \theta \quad x_2 \notin \theta & & \end{array}$$

Figure 4.10: Rules for Renaming and Context Flattening. Explained in Section 4.7.

rewrite rules. Steps (m,n) in Figure 4.6 show how the *Renaming* rule identifies variables on the two sides of the rewrite symbol.

The second rule collapses contexts to create a smaller (and thus both faster and easier to read) rewrite rule. In general, contexts can be nested, with different context nesting orders representing different term sets. Still, as our language is restricted to *anywhere* contexts, we can collapse nested contexts to obtain equivalent smaller patterns when the eliminated context does not appear elsewhere in the pattern. Step (o) in Figure 4.7 shows one such flattening. Intuitively, context c_1^d and variable x_1^d are not used in a different structure (i.e., not nested as $c_1^c[\![c_1^d[\![x_1^d]\!]]\!]$) anywhere else in the pattern so they can be safely eliminated.

Similarly, the third rule removes the context wrapping a variable when neither the context nor the variable appear anywhere else in term, in a different structure.

4.8 Extracting Solutions

The solutions, i.e. generalizing patterns, are obtained by applying the substitution recursively on the variable generalizing the original terms. For example, Applying substitution $x \mapsto f(y), y \mapsto c[\![z]\!]$ on the starting variable x becomes $f(c[\![z]\!])$. As the rules do not introduce cycles (we do not consider the initial identity substitution as introducing cycles), the recursion step is guaranteed to terminate. Figure 4.7 shows part of one final configuration when generalizing from the transformation examples in Figure 4.1. Applying the substitution on variable x will yield the rule in Figure 4.3.

Most specific generalizer pruning The context inference rules allow inference of solutions which are more general than others in the same solution set. We account for this by pruning all such solutions at the end of the algorithm. For example, looking at step (c) in Figure 4.5, the application of the *Context (left)* rule also yields another subsequent configuration which generalizes from `hooks` expression on one side, and the assignment on the other side. This inference path will eventually lead to a configuration that is more general than the one shown in the figure, so the more general configuration will be pruned at the very end of the algorithm. This pruning step ensures the final solution is the set of MSGs of the input patterns.

4.9 Generalizing Rewrite Rules

Generalizing rewrite rules follows naturally from generalizing patterns. Rewrite rule $l \Rightarrow r$ is more general than rewrite rule $l' \Rightarrow r'$ if $l \Rightarrow r$ applies to any term that $l' \Rightarrow r'$ applies to (i.e., l is more general than l') and, in all such instances, the resulting terms are identical.

We lift generalizing patterns to generalizing rewrite rules by considering each rewrite rule as a pair, i.e., the rewrite arrow is seen as a free function symbol, and then applying the standard pattern generalization algorithm. The first step in Figure 4.5 shows how the rewrite arrow is considered a regular symbol and the *Decomposition* step is applied.

The set of solutions will contain rewrite rules that have the same set of variables on both sides of the rewrite, as well as rewrite rules that have extra variables on the LHS or RHS. A rewrite rule that contains a variable on the LHS which does not appear on the RHS can be interpreted as deleting, instead of moving, that part of the term. A

new variable on the RHS can be interpreted as introducing a new term which varies between applications of the rewrite rule. Such a term can either be left as a variable, i.e., be considered symbolic, or it can be synthesized considering other constraints. In our case, we only allow such solutions when the new variable on the RHS has been generalized from JavaScript identifiers. In these cases, we synthesize a new identifier with a fresh, unique name.

The above is useful for the rules in our benchmark suite which introduce a new variable. One such example is the x variable in our motivating example transformation (Fig 4.3). After the rule is applied, the metavariable x is replaced with a JavaScript variable (syntactically, an identifier) that has a fresh, unique name.

Identifying variables on both the LHS and RHS of the rewrite is done via the Renaming inference rule in Figure 4.10. For the example in Figure 4.1, the analysis phase identified a (context) generalization for the functions before the rewrite and a different (context) generalization for the functions after the rewrite. The *Renaming* inference rule recognizes that they generalize identical pairs of patterns, and introduces a rename to the substitution to encode the identity for the final solution.

4.10 Inferring Program Transformations

In this section we explain how we extend the inference algorithm presented above to infer program transformations from multiple examples of concrete code transformations.

Given a set of patterns (that may be either ground terms or may contain variables as well), we find the set of their MSGs, i.e., *most specific generalizations* (defined in Section 4.4), by applying the following algorithm:

function GENERALIZE($inputPatternSet$)
  $oneInputPattern \leftarrow$ head of $inputPatternSet$
  $generalizations \leftarrow \{oneInputPattern\}$
  for each p **in** tail of $inputPatternSet$ **do**
    $newGens \leftarrow \varnothing$
    for each g **in** $generalizations$ **do**
      $newGens \leftarrow newGens \cup$ genPair $\langle g, p \rangle$
    $generalizations \leftarrow newGens$
  return $generalizations$

The algorithm takes as input a set of patterns (in our case, the program transformations), and outputs another set of patterns which are the MSGs of the patterns in the input set. The generalization of a set of one pattern is a set of the pattern itself. Going forward, the result of generalizing between a set of patterns and an additional pattern is the set union of generalizing from each element in the set with the additional pattern. Generalizing from a pair of patterns is denoted as `genPair` above, and it is explained in Section 4.4.

The final result is obtained by adding each input pattern to the generalization set in this manner. As the patterns obtained at each step are the most specific (as explained in Section 4.4), the order in which input patterns (i.e., input transformation examples) are considered does not matter.

Correctness of Inferred Rewrite Rules. Expressing program transformations as rewriting rules enables verifying properties of the transformations. For example, many of the transformations in our benchmark suite are refactorings, and it could be useful to verify that they preserve semantics. However, as the inferred rewrite rule only covers the syntactic part of the transformation, additional side conditions would be needed to ensure that behavior has not changed. Adding such side conditions and performing verification is beyond the scope of this dissertation. Note that in practice, many desired code transformations are *not* semantics preserving, and our system works equally well in such scenarios.

4.11 Optimizations

We applied several optimizations to allow our algorithm to scale to the input code used in our evaluation. With the exception of the heuristic pruning, presented at the end of this section, all other optimizations do not affect the correctness of the algorithm.

Top-down decomposition

For clarity, the rules are presented as inference rules over configurations. In the implementation we avoid maintaining a set of unsolved constraints. The algorithm recursively decompose the initial constraint top-down gathering the set of solutions on returning from the recursion. So the unsolved constraints are implicit in the recursion

step being computed, leaving the working configurations as pairs of the solved constraints and the solution environment.

Caching

The main benefit of the top-town decomposition optimization above is enabling the caching of constraint solutions. The solution for a constraint obtained by only applying the inference rules from Figures 4.8 and 4.9 is independent of its outer constraint. The solution for the `i < 10` $\overset{x}{\triangleq}$ `i < arch.length` constraint is independent of the other `for` loop constraint, i.e., if the constraint would appear again elsewhere in the term, the solution would be the same. This is not true after also applying the "context of context" and "context of variable" rules as they have side-conditions which may depend on the outer constraint. For example, the "context of context" rule checks whether the eliminated context appears anywhere else in the overall solution, so, while the check may be true locally, it could be invalidated by extra bindings in the outer constraint. In order to allow caching, the algorithm only applies the rules from Figure 4.10 at the very end, after all analysis rules have been executed.

Avoiding unfeasible generalization paths

We restrict context rules to only recursively descend on a pair of terms when there is an intersection between either the symbols or the constants that appear in the two terms.

Heuristic pruning of generalization solutions

To improve scalability, we inhibit searching for context generalization in terms appearing on both sides of the rewrite. For example, looking at Figure 4.1, the algorithm will not search for a context generalization between "else" branches because, for each rule, the else branch is identical before and after the transformation. Without this restriction, the algorithm would have found the generalization $c[\![x \texttt{ + } y]\!]$, as the `+` operator appears in the "else" branch on both sides.

This heuristic is the only optimization that affects the a formal property of our algorithm, i.e., it may lead to over-generalizations. In practical terms, when generalizing from a set of example transformations, the optimization may lead our algorithm to converge faster on the desired transformation pattern by ignoring the details in code

which is preserved by the transformations. We could not check the difference in number of steps required for converging with and without the optimization because deactivating the optimization hampers scalability to the point where most transformation patterns cannot be inferred. Finally, in our evaluation, we did not find any transformation type for which the details would have been relevant.

4.12 Correctness of Generalization Algorithm

We discuss the correctness properties of our inference rules. Recall that an inference rule operates over configurations of the form $\langle C \mid S \mid \theta \rangle$. For a configuration c, let $c(x)$ be the term obtained by applying the substitution θ from c recursively on the variable x.

The following theorem formally states that our rules are *sound*, in the sense that any configuration derived by the rules is a generalizer.

Theorem 4.1 *Let $c_0 \equiv \langle t_1 \overset{x}{\triangleq} t_2 \mid \varnothing \mid id \rangle$ be an initial configuration. Then for any configuration c obtained by applying the rules in Figure 4.8 and Figure 4.9 starting with c_0, $c(x)$ is a generalizer of t_1 and t_2.*

To prove the theorem, let c be a configuration such that $c(x)$ is a generalizer of t_1 and t_2, and let c' be a configuration obtained by applying a rule from Figure 4.8 or Figure 4.9 on c. Then, we can show that $c'(x)$ is also a generalizer of t_1 and t_2, by analyzing each inference rule. For example, Decompose replaces a variable with the term $f(x_1, \ldots, x_n)$ if the variable generalizes terms of the form $f(\ldots)$ in both the left and the right, and thus the result is still a generalizer of t_1 and t_2 (a MSG).

The next theorem states that our rules are *complete*, in the sense that any most specific generalizer (MSG) can be derived using the rules.

Theorem 4.2 *Let $c_0 \equiv \langle t_1 \overset{x}{\triangleq} t_2 \mid \varnothing \mid id \rangle$ be an initial configuration. Then any configuration c such that $c(x)$ is the MSG of t_1 and t_2 can be obtained by applying the rules in Figure 4.8, Figure 4.9, and Figure 4.10 starting with c_0.*

In the following, let c be a configuration such that $c(x)$ is a generalizer of t_1 and t_2. The proof relies on two main steps:

1. Let g be a generalizer of t_1 and t_2, such that g is strictly more specific than $c(x)$. Then there exists some configuration c' obtained by applying a rule from from Figure 4.8 or Figure 4.9 on c such that g is more specific than $c'(x)$.

2. If $c(x)$ is not an MSG of t_1 and t_2, then at least one of the rules in Figure 4.8, Figure 4.9, or Figure 4.10 can apply.

To see why the completeness holds, let g be some MSG of t_1 and t_2. We construct a sequence of configurations $c_0, \ldots, c_i, \ldots, g$ starting at c_0 such that each configuration c_i is obtain from c_{i-1} by applying an inference rule and g is more specific than $c_i(x)$. Notice that for the initial configuration c_0, we have that $c_0(x)$ is x (the most general generalizer of t_1 and t_2). Thus, g is more specific than $c_0(x)$. For any c_i in the sequence, step (2) ensures that, if $c_i(x)$ does not equal g modulo the rules in Figure 4.10, then we can extend the sequence by applying an inference rule. Further, step (1) ensures that we can extend the sequence with a configuration c_{i+1} such that g is more specific than $c_{i+1}(x)$. Finally, it can be shown that there are no infinite such sequences, thus we can conclude that we can always construct such a sequence for any MSG g.

It follows as a consequence of the two theorems above that all the MSGs are found by using the rules. Indeed, the set of configurations generated by the rules consists only of generalizers (by Theorem 4.1) and contains all MSGs (by Theorem 4.2). Thus, the MSGs are all the elements in the set that are not more specific than any other elements in the set.

4.13 Evaluation

Learning program transformations from examples can be useful in many scenarios, from interactive tools where the developer shows the tool example transformations one by one until satisfied with the inferred transformation, to fully-automatic systems that learn from large code bases.

We evaluate our approach by using our system to learn linting tool error fixes from examples. A linting tool shows the developer warnings about common stylistic and semantic problems in her code. The developer then has to figure out ways to change to code in order avoid the problem. The fixes are not always obvious, and they are often repetitive, thus tedious to implement. Learning the transformations alleviates this problem by allowing the developer to show the tool a few example fixes, with the tool generalizing from them and suggesting a rule capable of automatically fixing other similar problems in the code base.

The usefulness of an automation tool is defined by the ratio between the amount of effort required without the tool and the amount of effort required to use the tool,

including learning. In our case, the experience of using the tool is similar to the first few steps of the manual approach: the developer implements the first few code changes by hand. The only extra cost is that of activating the tool, i.e., telling the tool to look at the examples. While not implemented in our prototype, similar approaches by other tools have deemed this cost acceptable. Once the tool has learned a rule, it proposes to the developer fixes for similar problems in other parts of the codebase, which the developer can accept or reject on a case by case basis. The tool is useful if it can learn from few examples program transformations that are applicable in many other cases.

4.13.1 Methodology

We use our system to automatically learn fixes for a set of 15 linting tool warning types reported by ESLint [23], a widely-used linting tool for JavaScript. To select the warning types, we run ESLint on the top 20 most popular open-source JavaScript projects on GitHub, the popular Git host. For three of the projects ESLint did not report any warnings because they contained very little JavaScript code. The remaining 17 projects, shown in Figure 4.12, are used for evaluation.

We order the warning types by the number of times they are reported across all projects, and choose the 10 that are encountered most frequently and have a straightforward fix — e.g., it is not straightforward to decide on a fix for a high cyclomatic complexity warning.

5 of the 10 warnings involved syntactically distinct transformations that have been bundled together by ESList because they are similar semantically, e.g., checking for an assignment in a `for` loop condition is different from checking for an assignment in a `do-while` loop condition, and its fix would be different. We modify ESList to report such cases distinctly, creating 5 new warning types. All in all, our benchmark consists of 15 warning types, shown in Table 4.11. See the ESLint documentation [23] for a detailed description of the problems fixed by the rules.

For each of the 15 linting tool warning types, we repeat the following steps:

1. Run the ESList tool and gather all warning reports for this warning type. A warning report contains the source location in the project.
2. Randomly select at most 10 reports from each open-source project, then randomly select 10 reports from the resulting pool. We filter out code snippets larger than 200 characters, which is usually a few lines of code. For the no-loop-func example,

eqeqeq !==

$$x \texttt{ != } y \Downarrow x \texttt{ !== } y$$

eqeqeq ===

$$x \texttt{ == } y \Downarrow x \texttt{ === } y$$

guard-for-in

$$\texttt{for (}e\texttt{ in }l\texttt{) \{ }b\texttt{ \}} \Downarrow \texttt{for (}e\texttt{ in }l\texttt{) \{ if ({}.has(}l\texttt{, }e\texttt{)) \{ }b\texttt{ \} \}}$$

no-cond-assign if

$$\texttt{if (}c[\![a\texttt{ = }b]\!]\texttt{) }t\texttt{ else }e \Downarrow a\texttt{ = }b\texttt{; if (}c[\![a]\!]\texttt{) }t\texttt{ else }e$$

no-cond-assign while

$$\texttt{while (}c[\![a\texttt{ = }e]\!]\texttt{) \{ }b\texttt{ \}} \Downarrow a\texttt{ = }e\texttt{; while (}c[\![a]\!]\texttt{) \{ }b\texttt{; }a\texttt{ = }e\texttt{ \}}$$

no-loop-func

$$\texttt{for(}i\texttt{; }c\texttt{; }s\texttt{) \{ }c[\![\texttt{ function(}a\texttt{) \{ }b\texttt{ \} }]\!]\texttt{ \}} \Downarrow \texttt{n = function(}a\texttt{) \{ }b\texttt{ \}; for(}i\texttt{; }c\texttt{; }s\texttt{) \{ }c[\![n]\!]\texttt{ \}}$$

no-neg-cond ternary

$$\texttt{!}x\texttt{ ? }y\texttt{ : }z \Downarrow x\texttt{ ? }z\texttt{ : }y$$

no-neg-cond if

$$\texttt{if (!}x\texttt{) }y\texttt{ else }z \Downarrow \texttt{if (}x\texttt{) }z\texttt{ else }y$$

no-nested-ternary left

$$c[\![\texttt{ }u\texttt{ ? }x\texttt{ ? }y\texttt{ : }z\texttt{ : }v\texttt{ }]\!] \Downarrow n\texttt{ = }x\texttt{ ? }y\texttt{ : }z\texttt{; }c[\![\texttt{ }u\texttt{ ? }n\texttt{ : }v\texttt{ }]\!]$$

no-nested-ternary right

$$c[\![\texttt{ }u\texttt{ ? }v\texttt{ : }x\texttt{ ? }y\texttt{ : }z\texttt{ }]\!] \Downarrow n\texttt{ = }x\texttt{ ? }y\texttt{ : }z\texttt{; }c[\![\texttt{ }u\texttt{ ? }v\texttt{ : }n\texttt{ }]\!]$$

no-return-assign

$$\texttt{return }x\texttt{ = }y \Downarrow x\texttt{ = }y\texttt{; return }x$$

no-throw-literal

$$\texttt{throw }x \Downarrow \texttt{throw new Error(}x\texttt{)}$$

no-void

$$\texttt{void 0} \Downarrow \texttt{undef}$$

no-unneeded-ternary false

$$x\texttt{ ? false : true} \Downarrow \texttt{!}x$$

no-unneeded-ternary true

$$x\texttt{ ? true : false} \Downarrow x$$

Figure 4.11: Benchmarks

project name	URL (https://github.com/)
Semantic-UI	Semantic-Org/Semantic-UI
angular.js	angular/angular.js
brackets	adobe/brackets
backbone	jashkenas/backbone
d3	mbostock/d3
express	expressjs/express
foundation-sites	zurb/foundation-sites
impress.js	impress/impress.js
ionic	driftyco/ionic
jQuery-File-Upload	blueimp/jQuery-File-Upload
jquery	jquery/jquery
meteor	meteor/meteor
moment	moment/moment
react	facebook/react
resume.github.com	resume/resume.github.com
reveal.js	hakimel/reveal.js
three.js	mrdoob/three.js

Figure 4.12: Benchmark Projects

we have only found 15 instances of any size across all projects, so we select the 10 smallest of them, most of which are around 800 characters, which is about one page of code.

3. Write by hand the rewrite rule that encodes the fix. This is the rule the tool is expected to infer back from examples.
4. Execute the rewrite rule on the code of 10 warning reports obtaining 10 fix example instances.
5. Generate 10 random permutations of the example instances and, for each of them, execute the algorithm presented in Section 4.10. The algorithm takes as input a set of example instances and, at each step, the outer loop processes an additional example instance. The random permutation determines the order in which the initial `head` call and the outer loop obtain example instances from the set. Additionally, during evaluation, at each step we measure the time it takes to generalize and check whether the expected rule has been reached.

	code size (AST nodes)			execution time (ms)			# examples to expected rule		
rule identifier	mean±S.D.	median	max	mean±S.D.	median	max	mean±S.D.	median	max
eqeqeq !==	17±6	22	31	1±0	0	1	2±0	2	3
eqeqeq ===	18±3	20	25	2±1	2	5	3±1	4	5
guard-for-in	68±40	66	180	2366±874	78	4809	6±1	7	10
no-cond-assign if	86±21	96	142	59±51	36	164	5±2	4	10
no-cond-assign while	85±41	105	170	3840±944	8	12706	3±1	3	6
no-negated-cond ternary	49±27	83	102	1±1	2	4	2±0	2	4
no-negated-cond if	82±34	79	154	63±37	30	132	8±1	8	10
no-nested-ternary right	76±27	66	137	12±6	5	26	2±1	2	5
no-nested-ternary left	65±22	88	101	26±14	17	59	7±2	8	10
no-return-assign	45±40	29	165	1±1	1	5	2±0	2	3
no-throw-literal	21±17	25	57	2±1	1	4	3±1	5	7
no-unneeded-ternary false	28±5	30	39	0±0	1	1	2±0	2	2
no-unneeded-ternary true	19±6	20	36	8±19	2	62	2±0	3	3
no-void	9±1	11	11	0±0	0	0	2±0	2	2
no-func-in-loop	307±131	286	508	-	-	-	-	-	-

Table 4.1: Benchmark Results

4.13.2 Results

Figure 4.11 gives the desired rewrite rule for each of the ESLint benchmark rules. Note that of the 15 desired rules, five require use of our anywhere contexts. This gives strong evidence that rules with contexts are useful in practice; to our best understanding, no previous technique for learning transformations from examples can even express these rules, let alone learn them.

Table 4.1 shows the results of our experiment. For each transformation, it shows the mean, standard deviation, median, and maximum for:

- The example instances AST sizes (i.e., number of nodes in the AST).
- The execution time it took a permutation to get to the expected rule. For each permutation, the execution time is the sum of the generalization times up to the point where it converged to the expected rewrite rule.
- The number of examples required to converge on the expected rule.

In most cases, the execution time for generalizing from examples to the desired rule is negligible, with a maximum median execution time of 78ms. The optimizations described in Section 4.11 are key to obtaining these performance results. Deactivating any one of them makes the vast majority of the examples time out.

For the no-loop-func rule, the only available examples were significantly larger than those for the other rules, as explained earlier. Our implementation does not yet scale to these examples; it exceeded a 9-minute timeout in two-thirds of cases, and the cases that could be handled were not sufficient to learn the desired rule (though the rule that was learned was possibly useful). We plan to further improve scalability in the future; in particular, the top-down decomposition optimization (Section 4.11) makes the algorithm highly parallelizable, and it could even be executed in a distributed manner.

In most cases, the number of examples required to converge to the desired rewrite rule is quite low (four or fewer for two thirds of the rules). For the three rules, a median of 7-8 examples were required. In these cases, we found that a quirk of our AST format, ESTree [65], led to identical pieces of code being represented by different AST nodes, complicating generalization. We plan to address this issue in future work by reasoning about equivalence classes of AST nodes. Also, we plan to explore further heuristic generalization, which we expect will yield a significant reduction in the number of examples required.

4.14 Related Work

At a high level, our generalization approach is similar to, and can be seen as an extension of, the work of Alpuente et al. [8]. They present a syntactic order-sorted generalization algorithm for terms with function symbols which can obey any combination of associativity, commutativity, and identity axioms. Our system does not implement commutativity and order-sorting because both aspects would not be particularly helpful in the practical cases we considered, would have complicated both the system and its presentation, and could hurt generalization performance. Still, order-sorting and commutativity are orthogonal to the other generalization rules (i.e. rules specifying the way two terms are generalized, differentiating them from rewriting rule), and the system implementation could be extended to include them. Our extension of Alpuente et al.'s work is the inference of contexts, with rules specializing for context variables [9] and *anywhere* contexts; this capability is crucial for our application domain.

Raychev et al. [66] present a search-based technique for computing a sequence of refactorings consistent with some initial edits performed by the developer. At a high level this work is similar to ours; their technique infers a program transformation from a (partial) example. However, their transformations are limited to compositions of certain core refactorings; our system can learn much more general transformations.

Negara et al. [2, 3] propose a technique for learning refactorings from sequences of very fine-grained code changes, e.g., insert a numeric literal. Their algorithm is based solely on labeled sequences of changes and disregards completely where in the code the change is applied. While this may infer some useful refactorings like "convert element to collection", the applicability is limited.

Closer to our approach, Andersen and Lawall [16] propose an approach for inferring code patches from examples. Their system learns program transformations expressed as a composition of rewrite rules in a restricted language (no contexts or associative matching). They show an example that their approach cannot handle, but which can be expressed in our language, and can be learned by our system. Furthermore, their approach could not handle other cases covered by our system, such as our motivating example.

In a follow-up of the above work, Andersen et al. [17] address some of the limitations, but the short paper does not go into detail about how the algorithm works and its effectiveness is not clear from the limited evaluation.

Kim et al. [62] propose inferring refactorings as a composition from a set of predefined transformations. The search for the right transformation is a greedy heuristic. The approach is limited by the set of predefined transformations, so it cannot scale to new contexts without significant effort.

Meng et al. [21] propose a system for inferring program transformations from one example. The pattern matching condition is not expressed in a clearly-defined language, but has an internal representation in the refactoring tool which includes tracking data dependencies. The transformation itself is represented as a sequence of insert/remove/update AST operations.

Meng et al. [1] extend the above approach to learn from more than one example by computing the diffs for each of the example input-output pairs and taking the common subsequence of the fine-grained AST operations. The drawback is that the algorithm becomes dependent on the order and type of the operation sequence, as explained in Section 4.1.3.

Following on the above line of work, Meng et al. [22] apply the above technique to clone detection and removal via an automatically-generated extract method refactoring.

Chapter 5

Conclusion and Future Directions

Currently, the main obstacle to automatic programming is the development effort required to create even simple automatic program transformations. We give two techniques that help overcome this obstacle by:

(i) discovering complex transformations via searching through possible compositions of rewrite rules, and by

(ii) learning new rewrite rules from concrete code transformation examples.

The techniques introduced by this dissertation are just a step toward automatic programming, and they are only a partial implementation of our proposed approach. There are several directions of research that will bring us even closer to the goal.

The evolutionary approach might provide similar benefits for other optimization problems. In particular, we believe it would provide significant benefits if applied to lower level intermediate representations, e.g., LLVM. Optimizations over low-level intermediate representations have traditionally been crippled by tight constraints on compilation time. With today's cloud computational resources, removing this limitation is feasible. This will allow running algorithms such as ours that explore many optimization paths, which could provide significant performance gains. Another promising application would be improving code maintenance, e.g., by improving code readability.

Applying the evolutionary approach to other problems would also encourage research into a topic this work only touches upon: quickly and efficiently determining how close a program is to an optimization goal. Our approach is a straightforward static, syntax-driven evaluation of the program's AST. This could be significantly improved by actually executing the program as part of the optimization loop, or by employing more advanced static techniques, such as symbolic execution.

Our anti-unification algorithm only infers patterns with *anywhere* contexts and no side-conditions. Our current algorithm constructs complex contexts in the inference process, but discards them along the way in order to keep the patterns compact and in

order to scale to large code inputs. Removing this limitation could be challenging, but it would also allow the technique to be applied to larger classes of transformations.

Allowing the algorithm to also infer side-conditions, in particular side-conditions containing negation, would also be helpful. One approach to inferring negation side-conditions is to anti-unify a negative input pattern (i.e., a pattern that should not be matched by the desired pattern) with a positive pattern (i.e., a pattern that should be matched by the desired pattern). This will yield a pattern that is more general than both the positive pattern and the negative patterns. The desired pattern could be obtained by replacing each variable in the anti-unification pattern with a conjunction of the sub-pattern matched by the variable in the positive pattern and the negation of the sub-pattern matched by the variable in the negative pattern.

The main challenge we encountered to applying anti-unification to program transformation was scaling the algorithm to larger programs. Improving performance, especially considering the above generalizations, is a prerequisite to the mainstream success of this approach.

From a theoretical perspective, we have not formally defined the languages used by the two techniques in terms of Matching Logic. Clarifying this connection is required for verifying and certifying the correctness of the proposed transformations.

References

[1] N. Meng, M. Kim, and K. S. McKinley, "LASE: Locating and applying systematic edits by learning from examples," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 502–511.

[2] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig, "Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?" in *Proceedings of the 26th European Conference on Object-Oriented Programming*, ser. ECOOP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 79–103.

[3] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining Fine-grained Code Changes to Detect Unknown Change Patterns," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 803–813.

[4] F. Baader and T. Nipkow, *Term Rewriting and All That.* New York, NY, USA: Cambridge University Press, 1998.

[5] T. Back, *Evolutionary algorithms in theory and practice.* Oxford Univ. Press, 1996.

[6] G. Rosu, "Matching logic - extended abstract (invited talk)," in *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland*, ser. LIPIcs, M. Fernández, Ed., vol. 36. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 5–21.

[7] G. Roşu, "Matching logic," *Logical Methods in Computer Science (LMCS)*, vol. 13, no. 4, 2017.

[8] M. Alpuente, S. Escobar, J. Espert, and J. Meseguer, "A modular order-sorted equational generalization algorithm," *Information and Computation*, vol. 235, pp. 98–136, 2014.

[9] M. Felleisen and R. Hieb, "The revised report on the syntactic theories of sequential control and state," *Theor. Comput. Sci.*, vol. 103, no. 2, pp. 235–271, 1992.

[10] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, ser. HPCA '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–24.

[11] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992, vol. 1.

[12] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10.

[13] "Apache Spark," https://spark.apache.org, accessed on 03/20/2014.

[14] "T. J. Watson Libraries for Analysis," http://wala.sf.net, accessed on 2013-05-20.

[15] A. M. Sloane, "Lightweight language processing in Kiama," in *Generative and Transformational Techniques in Software Engineering III*, ser. GTTSE III. Springer, 2011, pp. 408–425.

[16] J. Andersen and J. Lawall, "Generic patch inference," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, Sept 2008, pp. 337–346.

[17] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S.-C. Khoo, "Semantic patch inference," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 382–385.

[18] M. Felleisen, R. B. Findler, and M. Flatt, *Semantics engineering with PLT Redex*. Mit Press, 2009.

[19] G. Rosu and T. F. Serbanuta, "An overview of the K semantic framework," *The Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[20] L. C. Kats and E. Visser, *The spoofax language workbench: rules for declarative specification of languages and IDEs*. ACM, 2010, vol. 45, no. 10.

[21] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: Generating program transformations from an example," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 329–342.

[22] N. Meng, L. Hua, M. Kim, and K. S. McKinley, "Does automated refactoring obviate systematic editing?" in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, pp. 392–402.

[23] "ESLint," http://eslint.org, accessed on 15/05/2018.

[24] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.

[25] D. Harel, *Dynamic Logic*. Dordrecht: Springer Netherlands, 1984, pp. 497–604.

[26] "K Framework," http://kframework.org, accessed on 1/10/2015.

[27] C. Radoi, S. J. Fink, R. Rabbah, and M. Sridharan, "Translating imperative code to mapreduce," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 909–927.

[28] "Git distributed version control system," http://git-scm.com, accessed on 1/10/2015.

[29] "Mercurial distributed source control management tool," http://mercurial.selenic.com, accessed on 1/10/2015.

[30] R. Joshi, G. Nelson, and K. Randall, "Denali: A goal-directed superoptimizer," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ser. PLDI '02. New York, NY, USA: ACM, 2002, pp. 304–314.

[31] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 305–316.

[32] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004.

[33] "Apache Hadoop," http://hadoop.apache.org, accessed on 03/05/2014.

[34] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson, "Relooper: Refactoring for loop parallelism in java," in *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. New York, NY, USA: ACM, 2009, pp. 793–794.

[35] K. Knobe and V. Sarkar, "Array SSA form and its use in parallelization," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '98. New York, NY, USA: ACM, 1998, pp. 107–120.

[36] A. W. Appel, "SSA is functional programming," *SIGPLAN Not.*, vol. 33, no. 4, pp. 17–20, Apr. 1998.

[37] R. A. Kelsey, "A correspondence between continuation passing style and static single assignment form," in *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, ser. IR '95. New York, NY, USA: ACM, 1995, pp. 13–22.

[38] S.-w. Liao, "Parallelizing user-defined and implicit reductions globally on multiprocessors," in *Proceedings of the 11th Asia-Pacific Conference on Advances in Computer Systems Architecture*, ser. ACSAC'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 189–202.

[39] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang, "Communication optimizations for irregular scientific computations on distributed memory architectures," *Journal of Parallel and Distributed Computing*, vol. 22, no. 3, pp. 462–478, Sep. 1994.

[40] Y. Klonatos, A. Nötzli, A. Spielmann, C. Koch, and V. Kuncak, "Automatic synthesis of out-of-core algorithms," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 133–144.

[41] S. d. Swierstra and O. Chitil, "Linear, bounded, functional pretty-printing," *J. Funct. Program.*, vol. 19, no. 1, pp. 1–16, Jan. 2009.

[42] N. Ramsey, "Unparsing expressions with prefix and postfix operators," *Software: Practice and Experience*, vol. 28, no. 12, pp. 1327–1356, 1998.

[43] B. C. Oliveira, A. Moors, and M. Odersky, "Type classes as objects and implicits," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 341–360.

[44] "Scala Parallel Collections," http://docs.scala-lang.org/overviews/parallel-collections/overview.html, accessed on 03/20/2014.

[45] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16.

[46] "Breeze," http://www.scalanlp.org, accessed on 03/20/2014.

[47] M. M. Strout, L. Carter, and J. Ferrante, "Compile-time composition of run-time data and iteration reorderings," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, ser. PLDI '03. New York, NY, USA: ACM, 2003, pp. 91–102.

[48] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "Code generation for parallel execution of a class of irregular loops on distributed memory systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 72:1–72:11.

[49] C. Nugteren and H. Corporaal, "Introducing Bones: a parallelizing source-to-source compiler based on algorithmic skeletons," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, ser. GPGPU-5. New York, NY, USA: ACM, 2012, pp. 1–10.

[50] M. H. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam, "Detecting coarse-grain parallelism using an interprocedural parallelizing compiler," in *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '95. New York, NY, USA: ACM, 1995.

[51] R. Lämmel, "Google's MapReduce programming model — revisited," *Science of Computer Programming*, vol. 70, no. 1, pp. 1 – 30, 2008.

[52] R. S. Bird, "Algebraic identities for program calculation," *Comput. J.*, vol. 32, no. 2, pp. 122–126, Apr. 1989.

[53] E. Meijer, M. Fokkinga, and R. Paterson, "Functional programming with bananas, lenses, envelopes and barbed wire," in *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, ser. FPCA '91. New York, NY, USA: Springer-Verlag New York, Inc., 1991, pp. 124–144.

[54] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson, "Relooper: Refactoring for loop parallelism in java," in *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. New York, NY, USA: ACM, 2009, pp. 793–794.

[55] L. Franklin, A. Gyori, J. Lahoda, and D. Dig, "Lambdaficator: From imperative to functional programming through automated refactoring," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1287–1290.

[56] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 62–73.

[57] E. Visser, "A survey of rewriting strategies in program transformation systems," *Electronic Notes in Theoretical Computer Science*, vol. 57, no. 0, pp. 109 – 143, 2001, {WRS} 2001, 1st International Workshop on Reduction Strategies in Rewriting and Programming.

[58] P. Borovanskỳ, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen, "An overview of ELAN," *Electronic Notes in Theoretical Computer Science*, vol. 15, pp. 55–70, 1998.

[59] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott, *All About Maude, A High-Performance Logical Framework*, ser. LNCS. Springer, 2007, vol. 4350.

[60] E. Visser, "Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5," in *Rewriting techniques and applications*. Springer, 2001, pp. 357–361.

[61] O. Bournez and C. Kirchner, "Probabilistic rewrite strategies. Applications to ELAN," in *Rewriting techniques and applications*. Springer, 2002, pp. 252–266.

[62] M. Kim, D. Notkin, and D. Grossman, "Automatic inference of structural changes for matching across program versions," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 333–343.

[63] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld, "Programming by demonstration using version space algebra," *Machine Learning*, vol. 53, no. 1-2, pp. 111–156, 2003.

[64] S. Gulwani, W. R. Harris, and R. Singh, "Spreadsheet data manipulation using examples," *Communications of the ACM*, vol. 55, no. 8, pp. 97–105, 2012.

[65] "ESTree," https://github.com/estree/estree, accessed on 15/05/2018.

[66] V. Raychev, M. Schäfer, M. Sridharan, and M. T. Vechev, "Refactoring with synthesis," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, 2013, pp. 339–354.

WordCount

```
for (int i = 0; i < docs.length; i++) {
  String[] split = docs[i].split(" ");
  for (int j = 0; j < split.length; j++) {
    String word = split[j];
    Integer prev = m.get(word);
    if (prev == null) prev = 0;
    m.put(word, prev + 1);
  }
}
```

⇓ Java to Lambda

```
Range(0, docs.size).fold(m)({
    case (v34, i) =>
      val split = docs(i).split(" ");
      val v10 = split.size;
      Range(0, v10).fold(v34)({
          case (v33, j) =>
            val v13 = v33(split(j));
            val prev = { if (v13 != null) v13 else 0 };
            v33.updated(split(j), prev + 1)
        })
  })
```

⇓ S:eliminate-null-check

```
Range(0, docs.size).fold(m)({
    case (v34, i) =>
      val v6 = docs(i).split(" ");
      val v4 = Range(0, v6.size).groupBy({
          case (j) => v6(j)
        });
      val v5 = v4.map({
          case (v2, v3) =>
            v3.fold(v34(v2))({
                case (v1, j) => v1 + 1
              })
        });
      v34 ++ v5
  })
```

⇓ S:reduce-monoid-identity-op

```
Range(0, docs.size).fold(m)({
    case (v34, i) =>
      val v6 = docs(i).split(" ");
      val v4 = Range(0, v6.size).groupBy({
```

```
        case (j) => v6(j)
      });
    val v5 = v4.map({
        case (v2, v3) =>
          v34(v2) + v3.size
      });
    v34 ++ v5
})
```

⇓ E:localize-group-by-accesses

```
Range(0, docs.size).fold(m)({
    case (v34, i) =>
      val v4 = docs(i).split(" ").groupBy(ID).__2;
      val v5 = v4.map({
          case (v2, v3) =>
            v34(v2) + v3.size
        });
      v34 ++ v5
})
```

⇓ E:localize-map-accesses

```
Range(0, docs.size).fold(m)({
    case (v34, i) =>
      val v4 = docs(i).split(" ").groupBy(ID).__2;
      val v5 = v4.zip(v34).map({
          case (v2, (v3, v7)) =>
            v7 + v3.size
        });
      v34 ++ v5
})
```

⇓ E:localize-fold-accsses

```
docs.fold(m)({
    case (v34, (i, v8)) =>
      val v4 = v8.split(" ").groupBy(ID).__2;
      val v5 = v4.zip(v34).map({
          case (v2, (v3, v7)) =>
            v7 + v3.size
        });
      v34 ++ v5
})
```

⇓ E:map-vertical-fission

```
docs.fold(m)({
    case (v34, (i, v8)) =>
      val v4 = v8.split(" ").groupBy(ID).__2;
      val v5 = v4.map({
          case (v2, v3) => v3.size
        }).zip(v34).map({
          case (v2, (v9, v7)) => v7 + v9
```

```
        });
      v34 ++ v5
  })
```

⇓ E:identify-map-monoid-plus

```
docs.fold(m)({
    case (v34, (i, v8)) =>
      (v8.split(" ").groupBy(ID).__2).map({
          case (v2, v3) => v3.size
        }) |+| v34
  })
```

⇓ E:pull-map-from-fold-by-subexpression-extract

```
docs.map({
    case (i, v8) =>
      (v8.split(" ").groupBy(ID).__2).map({
          case (v2, v3) => v3.size
        })
  }).fold(m)({
    case (v34, (i, v10)) =>
      v10 |+| v34
  })
```

⇓ E:map-horizontal-fission

```
docs.map({
    case (i, v8) =>
      v8.split(" ").groupBy(ID).__2
  }).map({
    case (i, v11) =>
      v11.map({
          case (v2, v3) => v3.size
        })
  }).fold(m)({
    case (v34, (i, v10)) =>
      v10 |+| v34
  })
```

⇓ E:swap-map-with-fold

```
val v11 = docs.map({
    case (i, v8) =>
      v8.split(" ").groupBy(ID).__2
  }).fold("ZERO_TOKEN")({
    case (v12, (i, v13)) =>
      v12 |+| v13
  });
v11.map({
    case (v2, v3) => v3.size
  })
```

⇓ E:map-horizontal-fission

```
val v11 = docs.map({
    case (i, v8) =>
      v8.split(" ").groupBy(ID)
  }).map({
    case (i, v14) => v14.__2
  }).fold("ZERO_TOKEN")({
    case (v12, (i, v13)) =>
      v12 |+| v13
  });
v11.map({
    case (v2, v3) =>
      v3.size
  })
```

⇓ E:map-horizontal-fission

```
val v11 = docs.map({
    case (i, v8) =>
      v8.split(" ")
  }).map({
    case (i, v15) =>
      v15.groupBy(ID)
  }).map({
    case (i, v14) => v14.__2
  }).fold(Map())({
    case (v12, (i, v13)) =>
      v12 |+| v13
  });
v11.map({
    case (v2, v3) =>
      v3.size
  })
```

⇓ E:swap-map-with-fold

```
val v14 = docs.map({
    case (i, v8) =>
      v8.split(" ")
  }).map({
    case (i, v15) =>
      v15.groupBy(ID)
  }).fold(Map())({
    case (v16, (i, v17)) =>
      v16 |+| v17
  });
(v14.__2).map({
    case (v2, v3) =>
      v3.size
  })
```

⇓ E:swap-map-with-fold

```
val v15 = docs.map({
    case (i, v8) =>
```

```
      v8.split(" ")
  }).fold(Map())({
    case (v18, (i, v19)) =>
      v18 |+| v19
  });
val v14 = v15.groupBy(ID);
(v14.__2).map({
    case (v2, v3) =>
      v3.size
  })
```

⇓ E:flatMap

```
val v15 = docs.flatMap({
    case (i, v8) =>
      v8.split(" ")
  });
val v14 = v15.groupBy(ID);
(v14.__2).map({
    case (v2, v3) =>
      v3.size
  })
```

⇓ E:reducebykey

```
docs.flatMap({
    case (i, v8) =>
      v8.split(" ")
  }).map({
    case (j, v6) =>
      (v6, 1)
  }).reduceByKey({
    case (v2, v3) =>
      v2 + v3
  })
```