© 2011 Fredrik Berg Kjolstad
REFACTORING TRANSFORMATIONS FOR MAINTAINABLE, SCALABLE AND EFFICIENT PARALLELISM

BY

FREDRIK BERG KJOLSTAD

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Advisers:

Professor Marc Snir
Professor Torsten Hoeffer
Computing is everywhere and our society depends on it. Increased performance over the last decades has allowed us to solve ever more interesting problems. We long relied on frequency scaling and instruction-level parallelism to give us the headroom we needed without disrupting software development, but this came to an end. The burden has now shifted to the programmer who is told to take advantage of a rapidly increasing host of parallel resources. Focus has been on helping him express parallel tasks, but this is not enough. Once parallelism is expressed, the programmer must still efficiently map it to the target system. Furthermore, the programmer who wants his application to survive the next platform must also care about maintainability and scalability.

This thesis explores the concept of automated refactoring transformations to make parallel applications maintainable, scalable and efficient. To that end, it develops two novel transformations that target very different application domains. Furthermore, it provides a catalogue of refactoring transformations for the HPC programmer, and argues that the proposed transformations require a new refactoring infrastructure.

The first transformation that is developed is targeted at irregular object-oriented shared-memory parallel applications, where too much shared data is a source of bugs and excessive synchronization. The transformation automatically converts mutable Java classes to immutable Java classes, which guarantees that objects instantiated from them will not be altered by other threads.

The second transformation targets regular distributed-memory parallel applications that spend significant time packing non-contiguous data into contiguous communication buffers. The transformation automatically replaces the packing code with a datatype that describes the layout of the packed data. This empowers the runtime to optimize the transfer for the target system, for example by exploiting advanced network features such as non-contiguous sends and receives.

By providing such parallel refactoring transformations we can improve the productivity of the programmer in his struggle to keep his application running on a plethora of platforms.
To my mother and sister, for their love and support.

They encourage me to follow my dreams, even when I get in over my head.
ACKNOWLEDGMENTS

This research was made possible by the support and advice of my advisor Marc Snir. He gave me the freedom to explore, the guidance to not get lost, and helped me find the conviction to continue towards a PhD.

Danny Dig was an invaluable source of learning and energy, and he showed me how to develop and present ideas. He also co-wrote the papers that became chapter 3 and chapter 5 of this thesis. I would like to thank Ralph Johnson for generously sharing his wisdom, and for guiding me to write my first pattern. Furthermore, I am thankful to Torsten Hoefler for teaching me about high performance computing and for being a great running partner.

I would also like to thank my roommates Rajinder Sodhi and Brett Jones, and my friends in Champaign-Urbana for being my friends, listening to my concerns and for making life fun.

Finally, I would like to thank my family and friends abroad, especially my mother, for their support and patience while I study far away from home.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER 1</th>
<th>INTRODUCTION</th>
<th>1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 2</th>
<th>RELATED WORK</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Refactoring</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>Immutability</td>
<td>3</td>
</tr>
<tr>
<td>2.3</td>
<td>Datatypes</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 3</th>
<th>A TRANSFORMATION FOR CLASS IMMUTABILITY$^1$</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>8</td>
</tr>
<tr>
<td>3.2</td>
<td>Motivating Example</td>
<td>11</td>
</tr>
<tr>
<td>3.3</td>
<td>Immutator</td>
<td>13</td>
</tr>
<tr>
<td>3.4</td>
<td>Transformations</td>
<td>14</td>
</tr>
<tr>
<td>3.5</td>
<td>Program Analysis</td>
<td>17</td>
</tr>
<tr>
<td>3.6</td>
<td>Discussion</td>
<td>23</td>
</tr>
<tr>
<td>3.7</td>
<td>Evaluation</td>
<td>24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 4</th>
<th>DATATYPE EXTRACTION</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>Illustrative Example</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>Datatype IR</td>
<td>35</td>
</tr>
<tr>
<td>4.4</td>
<td>Datatype Optimizations</td>
<td>39</td>
</tr>
<tr>
<td>4.5</td>
<td>Evaluation</td>
<td>48</td>
</tr>
<tr>
<td>4.6</td>
<td>Future Datatype Work</td>
<td>51</td>
</tr>
<tr>
<td>4.7</td>
<td>A Datatype Extraction Case Study</td>
<td>54</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 5</th>
<th>THE CASE FOR REFACTORING IN HPC</th>
<th>59</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>59</td>
</tr>
<tr>
<td>5.2</td>
<td>Towards An HPC Refactoring Catalog</td>
<td>60</td>
</tr>
<tr>
<td>5.3</td>
<td>Annotations and Views</td>
<td>64</td>
</tr>
<tr>
<td>5.4</td>
<td>A Framework for Advanced Refactoring</td>
<td>65</td>
</tr>
</tbody>
</table>

| CHAPTER 6 | CONCLUSION | 67 |

| REFERENCES | | 69 |
CHAPTER 1

INTRODUCTION

The end of frequency scaling and the rise of the multi-core processor is causing friction in the programming community. It is no longer possible to ride the frequency wave to high performance and programmers must develop parallel applications to achieve speedups.

However, vast amounts of sequential source code exist and it is not feasible to rewrite these sequential systems to parallel systems. A common approach is therefore to refactor sequential applications to parallel applications.

Refactoring is the process of transforming source code without changing its semantics, in order to achieve some non-functional objective. Traditional refactorings focus on changing the source code to improve readability and maintainability [1]. The earliest research on automated refactoring goes back to the 1990s with Opdyke and Johnson’s work on behavior preserving refactoring [2]. Since then many automated refactorings have been proposed by both academia and industry. More recently researchers have begun exploring the potential for automated refactorings to gradually introduce parallelism in existing applications [3, 4].

However, introducing parallelism is not enough. The programmer must also write his program such that it efficiently exploit the underlying system, and avoid pitfalls that decrease the performance of the software. Furthermore, in a world of constant change, parallel applications must be written with scalability, maintainability and performance portability in mind.

This thesis makes the case for automated refactoring transformations to make parallel applications maintainable, scalable and efficient. To that end, we develop and evaluate two novel transformation algorithms, for two very different application domains. Furthermore, we present the beginnings of a catalogue of refactoring transformations for the HPC programmer, and argue that the proposed transformations require a new and more powerful refactoring infrastructure.

The first transformation we develop targets irregular object-oriented shared-memory parallel applications, where too much shared data is a source of bugs and excessive synchronization. The
transformation automatically converts mutable Java classes to immutable classes. This guarantees that objects instantiated from them will not be altered by other threads, which simplifies the application and reduces the need for synchronization such as locks and mutexes. We implemented this transformation as an Eclipse refactoring plugin that allows the programmer to perform the transformation at the click of a button.

The second transformation targets regular distributed-memory parallel applications, such as scientific computing codes. These codes often spend a lot of time packing non-contiguous application data into contiguous communication buffers. We present a novel algorithm that automatically generates a datatype that describes the data that is packed, and then replaces the packing code with the use of this datatype. We implemented this transformation as an Eclipse CDT refactoring plugin that operates on C and C++ code with MPI. The programmer selects an MPI_Send call as well as the packing code that packs the data to be sent, and the tool automatically transforms the packing code to code that uses MPI Datatypes. It then changes the MPI_Send call to send data directly from the user data structures using the newly created datatype. Furthermore, unpacking code that unpacks data received from a MPI_Recv call is totally symmetric and can be handled with the same technique.

The main contributions of this thesis are:

1. A technique and a tool for transforming mutable Java classes to immutable Java classes
2. An algorithm for the Extract Datatype refactoring. The algorithm automatically analyzes and transform packing code to datatype code, and is implemented in a tool that converts packing code written in C or C++ with MPI to code that uses MPI Datatypes.
3. The beginning of a parallel performance refactoring catalogue for HPC
4. A survey of analysis frameworks that can satisfy the needs of HPC refactorings

The rest of this thesis is organized as follows. Chapter 2 describes the related work on refactoring as well as the related work for the analysis techniques used for each refactoring transformation. Chapter 3 presents and evaluates the first transformation, which converts mutable Java classes to immutable Java classes, while chapter 4 presents the second transformation which converts packing code to datatype code. Chapter 5 argues the case for refactoring in High Performance Computing, provides the beginnings of a refactoring catalogue for HPC programmers and argues that the proposed refactorings require a new refactoring infrastructure. Finally, chapter 6 concludes.
CHAPTER 2

RELATED WORK

2.1 Refactoring

The earliest refactoring research focused on achieving behavior-preservation through the use of pre- and post-conditions [2] and program dependence graphs [5]. Traditionally, refactoring tools have been used to improve the design of sequential programs. The more recent work has expanded the area with new usages. Dig, et al. have used refactoring [3, 4] to retrofit parallelism into sequential applications via concurrent libraries. In the same spirit, Wloka et al. [6] present a refactoring for replacing global state with thread local state. Schäfer et al. [7] present Relocker, a refactoring tool that lets programmers replace usages of Java built-in locks with more flexible locks. Our transformations for class immutability makes code easier to reason about and enables parallelism by prohibiting changes to shared state. Our transformation to convert packing code to datatype code makes distributed message-passing applications more readable and maintainable. Moreover, it enables advanced networks to send non-contiguous data without a copy pass at each end, thus potentially increasing the performance substantially.

2.2 Immutability

Specifying and checking immutability There is a large body of work in the area of specifying or checking immutability [8, 9, 10].

Pechtchanski and Sarkar [8] present a framework for specifying immutability constraints along three dimensions: lifetime (e.g., the whole lifetime of an object, or only during a method call), reachability (e.g., shallow or deep immutability), and context. Immutator enforces deep immutability for the whole lifetime of an object, on all method contexts.

Tschantz and Ernst [9] present Javari, a type-system extension to Java for specifying reference immutability. Reference immutability means that an object can not be mutated through a partic-
ular reference, though the object could be mutated through other references. In contrast, object immutability specifies that an object can not be mutated through any reference, even if other instances of the same class can be. Zibin et al. [10] build upon the Javari work and present IGJ that allows both reference and object immutability to be specified. Class immutability specifies that no instance of an immutable class may be mutated. Reference immutability is more flexible, but weaker than object immutability, which in turn is weaker than class immutability. Immutator enforces class immutability.

These systems are very useful to document the intended usage and to detect violations of the immutability constraints. But they leave to the programmer the tedious task of removing the mutable access. In contrast, Immutator performs the tedious task of getting rid of mutable access, by converting mutators into factory method, and cloning the state that would otherwise escape.

Supporting program analyses Components of our program analyses have previously been published: detecting side-effect free methods [11, 12, 13, 14] and escape analysis [15, 16]. Our analyses detect side effects and escapes only on state that is reachable from the target class.

Side-effect analysis [11, 12, 13, 14] uses inter-procedural alias analysis and dataflow propagation algorithms to compute the side effects of functions. There are two major differences between these algorithms and Immutator’s analysis for detecting mutators. First, the scope is different. Our algorithm detects side-effects to variables that are part of the transitive state of the target class, whereas previous work determines all side-effects (including side effects to method arguments that do not belong to the transitive state). Consider the method drawFrame from TextFigure in JHotDraw:

```java
public void drawFrame(Graphics g) {
    g.setFont(fFont);
    g.setColor((Color)getAttribute("TextColor"));
    g.drawString(fText, ...);
}
```

The previous algorithms would determine that drawFrame is a mutator method, because it has side effects on the graphics device argument, g.

However, if Immutator transforms TextFigure then drawFrame will not mutate the transitive state of the target class, thus eliminating the need to clone the graphics device.

Second, our algorithm distinguishes between (i) methods in the target class that directly or
indirectly assign to the fields of the target class and (ii) methods outside the target class (potentially in libraries) that do not assign to target class' fields, but mutate these fields transitivity. 

Immutator converts the former mutators into factory methods, and rewrites the calls to the latter methods into calls dispatched to a copy of \texttt{this} (e.g., see the \texttt{.this} receiver in Fig. 3.1, lines 28–29). This enables Immutator to correctly transform code that invokes library methods.

Escape analysis [15, 16] determines if an object escapes the current context. So far, the primary applications of this analysis has been to determine whether (i) an object allocated inside a function does not escape and thus can be allocated on the stack, and (ii) an object is only accessed by a single thread, thus any synchronizations on that object can be removed. There are three major differences between these algorithms and Immutator’s escape analysis. First, our algorithm detects escaped objects that belong to the transitive state of the target class. Second, our algorithm is designed to be used in an interactive environment. Thus, it does not perform an expensive whole program analysis, but only analyzes the boundary methods of the target class. Third, in addition to escaping objects, our algorithm also detects entering objects.

\section{Datatypes}

\subsection{Generating Datatypes}

Gojun et al., developed a pre-processor tool called AutoMap that automatically generate datatypes for user-annotated C struct types [17]. Furthermore, Tansey and Tilevich developed a GUI tool that generates datatypes for C++ classes [18]. These tools automates the generation of datatypes representing struct and class definitions, but do not look for opportunities to use these datatypes in client code, nor do they find array accesses that can be replaced with vector or contiguous types. In contrast, our algorithm is the first to generate datatypes based on access patterns in packing code, to find indexed and vector accesses in arrays, and to automatically rewrite the client code to use these datatypes.

\subsection{Optimizing Datatype Performance}

There is a large body of research on optimizing the performance of datatypes. Gropp, Lusk and Swider provide a taxonomy of MPI datatypes according to their memory reference patterns, and demonstrate how to efficiently implement these patterns using a variety of techniques [19].

One line of research on datatype processing aims to improve the performance of datatype packing in MPI implementations over client packing code by using efficient internal data structures, runtime
and machine information. Bynna et al. present a technique to improve the performance of derived datatypes by automatically choosing a packing algorithm that is optimized for the memory-access cost of the target machine [20]. Ross, Miller and Gropp describe an efficient internal representation of datatypes called dataloops that aids MPI implementation that performs datatype packing in maintaining high performance during datatype processing [21].

A second line of research describes techniques to take advantage of datatypes to exploit advanced network features that allows moving non-contiguous data without any packing. Wu, Wyckoff and Panda compare the performance of an MPI implementation that performs datatype packing and an implementation that uses the Remote Direct Memory Access (RDMA) feature of InfiniBand [22] to avoid either the packing or the unpacking involved in transmitting non-contiguous [23]. Santhanaraman, Wu and Panda presents a technique they call Send Gather Receive Scatter (SGRS) that uses InfiniBand channels to avoid both the packing and unpacking involved in sending/receiving non-contiguous data [24].

All of these techniques require datatype to be specified explicitly. Our technique converts packing code to datatype code, thereby enabling these optimizations.

**Supporting program analyses** Key steps in our algorithm for converting packing code to datatype code is to convert indexed types to vector types, and vector types to contiguous types. These steps rely heavily on algebraic expression simplification and loop induction variable summarization. Both of these problems have been extensively studied in the literature.

Algebraic expression simplification is widely used in mathematical packages such as Matlab and Mathematica. Moses presents various techniques to reduce the size of expressions with the dual goals of making them more intelligible to the user and to allow the designer to construct useful and efficient systems [25]. Furthermore, Buchberger and Loos formally describe the problem of canonical algebraic simplification and then present two major groups of simplification techniques [26]. Our implementation of the datatype extraction algorithm uses the Open Source Symja library [27] to simplify expressions.

Loop induction variable detection and summarization has most commonly been studied in the context of operator strength reduction. Induction variables are variables whose values change between iterations of a loop. Linear induction variables are induction variables whose values follow an arithmetic sequence. That is, they increase or decrease by a fixed amount from one loop iteration to the next. Cocke and Kennedy developed an early algorithm for operator strength reduction based
on induction variables [28]. Cooper et al. later improved on this algorithm by taking advantage of SSA form to perform efficient sparse induction variable detection [29]. Our implementation of loop induction variable summarization is based on the technique described by Muchnic [30].
CHAPTER 3

A TRANSFORMATION FOR CLASS IMmutABILITY

3.1 Introduction

An immutable object is one whose state can not be mutated after the object has been initialized and returned to a client. By object state we mean the transitively reachable state. That is, the state of the object and all state reachable from that object by following references. Immutability has long been touted as a feature that makes functional programming an excellent choice for both sequential and parallel software [32].

Immutability makes sequential programs simpler. An immutable object, sometimes known as a value object [33], is easier to reason about because there are no side-effects [34]. Applications that use immutable objects are therefore simpler to debug. Immutable objects facilitate persistent storage [35], they are good hash-table keys [36], they can be compared very efficiently by comparing identities [35], they can reduce memory footprint (through interning/memoization [36, 37] or flyweight [38]). They also enable compiler optimizations such as reducing the number of dynamic reads [8]. In fact, some argue that we should always use immutable classes unless we explicitly need mutability [39].

In addition, immutability makes distributed programming simpler [35]. With current middleware technologies like Java RMI, EJB, and Corba, a client can send messages to a distributed object via a local proxy. The proxy implements an update protocol, so if the distributed object is immutable then there is no need for the proxy.

Moreover, as parallel programming becomes ubiquitous in the multicore era, immutability makes parallel programming simpler [40, 41]. Since threads can not change the state of an immutable object, they can share it without synchronization. An immutable object is embarrassingly thread-safe.

However, mainstream languages like Java, C#, and C++ do not support deep, transitive im-
mutability. Instead, they only support shallow immutability through the final, readonly, and const keywords. This is not enough, as these keywords only make references immutable, not the objects referenced by them. Thus, the transitive state of the object can still be mutated.

To get the full benefits of immutability, deep immutability must therefore be built into the class. If a class is class immutable, none of its instances can be transitorily mutated. Examples in Java include String and the classes in the Number class hierarchy.

It is common for OO programs to contain both mutable and immutable classes. For example, the JDigraph open-source library contains MapBag and ImmutableBag. MapBag is intended for cases where mutation is frequent, and ImmutableBag where mutations are rare. The programmer, an expert on the problem domain, uses MapBag when mutations are frequent, and ImmutableBag when mutations are rare.

Sometimes programmers write an immutable class from scratch, other times they refactor a mutable class into an immutable class. The refactoring can be viewed as two related technical problems:

1. The conversion problem consists of generating an immutable class from an existing mutable class.

2. The usage problem consists of modifying client code to use the new immutable class in an immutable fashion.

In this work we solve the conversion problem. To create an immutable class from a mutable class (from here on referred as the target class), the programmer needs to perform several tasks. The programmer must search through the methods of the target class and find all the places where the transitive state is mutated. This task is further complicated by polymorphic methods and mutations nested deep inside call chains that may extend into third party code.

Moreover, the programmer must ensure that objects in the transitive state of the target class do not escape from it, otherwise they can mutated by client code. Such escapes can happen through return statements, parameters, or static fields. Finding objects that escape is non-trivial. For example, an object can be added to a List that is returned from a method, causing the object to escape along with the List.

Furthermore, once the programmer found all mutations, she must rewrite mutator methods, for example by converting them to factory methods. She must also handle objects that enter or escape
the class, for example by cloning them. Common approaches include deleting the mutating method, throwing an exception, or re-implementing it as a factory method that returns a new object.

In 346 cases we studied these code transformations required changing 45 lines of code per target class, which is tedious. Furthermore, it required analyzing 57 methods in the call graph of each target class to find mutators and entering/escaping objects. Because this analysis is inter-procedural and requires reasoning about the heap, it is non-trivial and error-prone. In a controlled experiment where 6 experienced programmers converted JHotDraw classes to immutable counterparts, they took an average of 27 minutes, and introduced 6.37 bugs per class.

To alleviate the programmer’s burden when creating an immutable class from a mutable class, we designed an algorithm and implemented a tool, Immutator, that works on Java classes. We developed Immutator on top of Eclipse’s refactoring engine. Thus, it offers all the convenience of a modern refactoring tool: it enables the user to preview and undo changes and it preserves formatting and comments. To use it the programmer selects a target class and chooses Generate Immutable Class from the refactoring menu. Immutator then verifies that the transformation is safe, and rewrites the code if the preconditions are met. However, if a precondition fails, it warns the programmer and provides useful information that helps the programmer fix the problem.

At the heart of Immutator are two inter-procedural analyses that determine the safety of the transformation. The first analysis determines which methods mutate the transitive state of the target class. The second analysis is a class escape analysis that detects whether objects in the transitive state of the target class state may escape. Although Immutator transforms the source code, the analyses work on bytecode and correctly account for the behavior of third-party Java libraries.

There is a large body of work [11, 12, 13, 14] on detecting whether methods have side effects on program state. Previous analyses were designed to detect any side effect, including changes to objects reachable through method arguments and static variables. In contrast, our analysis intersects the mutated state with the objects reachable through the this reference. Therefore, it only reports methods that have a side effect on the current target object’s state.

Similarly, previous escape analyses [15, 16] report any object that escapes a method, including locally created objects. Our analysis only reports those escaping objects that are also a part of the transitive state of the target class.

This chapter makes the following contributions:
Problem Description While there are many approaches to specifying and checking immutability this is, to the best of our knowledge, the first work that describes the problems and challenges of transforming a mutable class into an immutable class.

Transformations We present the transformations that convert a Java class to an immutable Java class.

Algorithm We have developed an algorithm to automatically convert a mutable class to an immutable class. The algorithm performs two inter-procedural analyses; one that determines the mutating methods, and one that detects objects that enter or escape the target class. Based on information retrieved from these and other analyses our algorithm checks preconditions and performs the mechanical transformations necessary to enforce immutability.

Implementation We have implemented the analyses and code transformations in a tool, Immutator, that is integrated with the Eclipse IDE.

Evaluation We ran Immutator on 346 classes from known open-source projects. We also studied how open-source developers create immutable classes manually. Additionally, we designed a controlled experiment with 6 programmers transforming JHotDraw classes manually. The results show that Immutator is useful. First, the transformation is widely applicable: in 33% of the cases Immutator was able to transform classes with no human intervention. Second, several of the manually-performed transformations are not correct: open-source developers introduced an average of 2.1 errors/class, while participants introduced 6.37 errors/class; in contrast, Immutator is safe. Third, on average, Immutator runs in 2.33 seconds and saves the programmer from analyzing 57 methods and changing 45 lines per transformed class. In contrast, participants took an average of 27 minutes per class. Thus, Immutator dramatically improves programmer productivity.

Immutator as well as the experimental data can be downloaded from: http://refactoring.info/tools/Immutator

3.2 Motivating Example

We describe the problems and challenges of transforming a mutable class into an immutable class using a running example. Class Circle, shown on the left-hand side of Fig. 3.1, has a center, stored
public class Circle {
    private Point c = new Point(0, 0);
    private int r = 1;

    public int getRadius() {
        return r;
    }

    public void setRadius(int r) {
        this.r = r;
    }

    public void moveTo(Point p) {
        this.c = p;
    }

    public void moveTo(int x, int y) {
        c.setLocation(x, y);
    }

    public void moveBy(int dx, int dy) {
        Point center = new Point(c.x+dx, c.y+dy);
        moveTo(center);
    }

    public Point getLocation() {
        return c;
    }
}

public final class ImmutableCircle {
    private final Point c;
    private final int r;

    public ImmutableCircle() {
        this.r = 1;
        this.c = new Point(0, 0);
    }

    private ImmutableCircle(Point c, int r) {
        this.c = c;
        this.r = r;
    }

    public int getRadius() {
        return r;
    }

    public ImmutableCircle setRadius(int r) {
        return new ImmutableCircle(c, r);
    }

    public ImmutableCircle moveTo(Point p) {
        return new ImmutableCircle(p.clone(), this.r);
    }

    public ImmutableCircle moveTo(int x, int y) {
        Point p = new ImmutableCircle(this.c.clone(), this.r);
        p.setLocation(x, y);
        return p;
    }

    public ImmutableCircle moveBy(int dx, int dy) {
        Point center = new Point(c.x+dx, c.y+dy);
        ImmutableCircle p = moveTo(center);
        return p;
    }

    public Point getLocation() {
        return c.clone();
    }
}

Figure 3.1: Immutator converts a mutable Circle (left pane) into an immutable class (right pane).

in field c, and a radius, stored in field r. There are several methods to modify or retrieve the state. The programmer decides to transform this class into an immutable class, since it makes sense to treat mathematical objects as value objects.

Transforming even a simple class like Circle into an immutable class, as shown on the right-hand side of Fig. 3.1, is non-trivial. First, the programmer must find all the mutating methods. Method setRadius on line 19 is a direct mutator, and is easy to spot because it assigns directly to a field. Method moveTo(int, int) on line 27 is a mutator too. However, the code on line 30 does not change the value of c directly, but instead changes the object that c references. Therefore, this method mutates the transitive state of Circle. Method moveBy on line 34 is another mutator that does not mutate the object directly. Instead, it mutates state indirectly by calling moveTo(Point). Finding all mutators (transitive and indirect) is complicated by long call chains, polymorphic methods, aliases, and third-party library code.

Furthermore, the programmer must locate all the places where an object enters or escapes the target class. Consider a client that creates a Point object and passes it to moveTo(Point). Since
the client holds a reference to the point, it can still mutate the object through the retained reference. The programmer may not have access to all existing and future client code so she must conservatively assume that the target class can be mutated through entering and escaping objects. Therefore, to enforce deep immutability, the programmer must find all the places where objects enter the target class (line 23–24) or escape (line 41), and clone them. However, the programmer should avoid excessive cloning and only clone where absolutely required.

Even for this simple example, the transformation requires inter-procedural analysis (line 30 and 36), which must take pointers into account (line 30). Our approach combines the strength of the programmer (the higher-level understanding of where immutability should be employed) with the strengths of a tool (analyzing many methods and making mechanical transformations).

**Immutator** automatically handles the rewriting (Section 3.4) and analysis (Section 3.5) required to make a class immutable.

### 3.3 Immutator

We implemented our algorithm for **Generate Immutable Class** as a plugin in the Eclipse IDE. To use **Immutator**, the programmer selects a class and then chooses the **Generate Immutable Class** option from the refactoring menu. Before applying the changes, **Immutator** gives the programmer the option to preview them in a before-and-after pane. Then **Immutator** makes the class deeply immutable.

Our algorithm transforms the target class in-place. However, the tool makes a copy of the target class and then transforms this copy. This provides the programmer with two variants of the same class: a mutable and an immutable one. The programmer decides where it makes sense to use one over the other.

However, the programmer can not use the deeply immutable version if the class is to be used in client code that relies on structural sharing of mutable state. Consider a **Graph** that contains mutable **Node** objects. The semantics of the **Graph** class ensure that several nodes can share the same successor node. If the programmer made **Graph** immutable, **Immutator** would change mutator methods like `addEdge(n1, n2)` to clone the entering nodes, thus transforming the graph into a tree. On the other hand, structural sharing of immutable objects does not contradict with deep-copy immutable semantics. If the **Graph** contained immutable **Node** objects, then **Immutator** would not clone **Node** objects, thus preserving the sharing semantics of the original class.

Before transforming the target class, **Immutator** checks that it meets four preconditions, and
reports failed preconditions to the programmer. The programmer can decide to ignore the warnings and proceed, or cancel the operation, fix the root cause of the warnings and then re-run Immutator.

3.3.1 Transformation Preconditions

Immutator checks the following preconditions:

**Precondition #1** The target class can only have superclasses that do not have any mutable state.

**Precondition #2** The target class can not have subclasses as these can add mutable state to the target objects.

**Precondition #3** Mutator methods in the target class must have a \texttt{void} return type and must not override methods in superclasses. This is because Immutator rewrites mutator methods to return new instances of the target class and must use the return type for this. Methods in Java can only return one value and it is not allowed to change the return type when overriding a method.

**Precondition #4** Objects that enter or escape the transitive state of the target class must either already implement \texttt{clone}, or the source code of their classes must be available so that a \texttt{clone} method can be added.

While these preconditions may seem restrictive, we believe that value classes are likely to meet them. For example, software that follows the command-query separation principle (methods either perform an operation, or return a value) will not have mutators with non-void return types, thus meeting precondition 3. Furthermore, preconditions 1 and 2 are limitations of the current implementation, and not inherent to the approach. We leave for future work to refactor a whole class hierarchy.

3.4 Transformations

This section describes the transformations that Immutator applies to the target class. We will use the motivating example introduced in Fig. 3.1 to illustrate the transformations.
Make fields and class final  First, *Immutator* makes all the fields of the class **final**. Final fields in Java can only be initialized once, in constructors or field initializers. *Immutator* also makes the target class **final**. This prevents it from being extended with subclasses that add mutable state.

Generate constructors  *Immutator* adds two new constructors (line 5 and 10). The first constructor is the default constructor and it does not take any arguments. This constructor initializes each field to their initializer value in the original class or to the default value if they had none. The second constructor is a **full** constructor. It takes one initialization argument for each field, and is private as it is only used internally to create instances.

3.4.1  Convert Mutators into Factory Methods

Since the fields are **final**, methods can not assign to them. *Immutator* converts mutator methods into factory methods that create and return new objects with updated state.

We call a method a **mutator** if it (i) assigns to a field in the transitive state of a target class instance, or (ii) invokes a method that is a mutator method.

**Convert direct mutators**  Setters are a common type of mutator in object-oriented programs. Lines 19–21 on the right-hand side of Fig. 3.1 show the transformation of `setRadius` to a factory method. *Immutator* changes (i) the return type to the type of the target class, and (ii) the method body to construct and return a new object, created using the full constructor. The constructor argument that is assigned to the `r` field is set to the right-hand-side of the assignment expression. The arguments for the other fields (e.g., `c`) are copied from the current object. Thus, the factory method returns a new object where the `r` field has the new value, while the other fields remain unchanged.

However, not all mutators are simple setters. Some contain multiple statements, while others mutate fields indirectly by calling other mutators. `moveBy`, on line 34–38, demonstrates both of these traits. It contains two statements, and it mutates `c` indirectly by calling `moveTo`.

The right-hand side shows how *Immutator* transforms `moveBy` into a factory method. It introduces a new local reference, called `_this`, to act as a placeholder for Java’s built-in `this` reference. After `_this` is defined at the first mutation, *Immutator* replaces every explicit and implicit `this` with `_this`.

Furthermore, for every statement that calls a mutator, *Immutator* assigns the return value of the method (which is now a factory method) back to `_this`. Thus, the rest of the method sees and
operates on the object constructed by the factory method. Finally, the this reference is returned.

An interesting property of this technique is that it shifts the mutations from the target object to the this reference. That is, instead of mutating the object pointed to by this, it mutates the state of this. Ideally, Immutator would reassign back to this, but in Java the built-in this reference can not be assigned to. Therefore, Immutator replaces it with the mutable place-holder this.

**Convert transitive mutators** Consider the moveTo(int, int) method on line 27–32. Although this method never assigns to the c field, it still mutates c’s transitive state through the setLocation method. Immutator notices that the method setLocation does not belong to the target class, but to java.awt.Point in the GUI library. Therefore, Immutator can not rewrite setLocation into a factory method.

As before, Immutator creates the this reference, and returns it at the end of the method. Furthermore, Immutator clones c, so that the mutation does not affect the original object referenced by this. The cloned c is passed as an argument to the new Circle, which is assigned to this. Since this.c now refers to a clone of the original this.c, we can allow the mutation through setLocation.

3.4.2 Clone the Entering and Escaping state

Another way the transitive state of the target object can be mutated is if client code gets hold of a reference to an object in its internal state, and then mutates it outside of the target class. This can happen in two ways: (i) through objects that are entering the target class (e.g., Point p on line 24), or (ii) through objects that are escaping the target class (e.g., c on line 41).

An object enters the target class if it is visible from client code, and is assigned to a field in the transitive state of the target class. For example, the client code could call moveTo(Point) and then mutate the point through the retained reference.

We define a target class escape as an escape from any of its methods, including constructors. An escape from a method means that an object that is transitively reachable through a field of the target class is visible to the client code after the method returns. For example, on line 41, the object pointed to by c escapes through the return statement, and can then be mutated by client code. Escapes can also occur through parameters and static fields.

If an object enters or escapes then current or future client code may perform any operations on it, and Immutator must conservatively assume that it will be mutated.
Immutator handles entering and escaping objects by inserting a call to the clone method to perform a deep copy of the object in question, as seen on the right-hand side of line 24 and 41. However, if the entering or escaping object is itself immutable, Immutator does not clone it. The current implementation considers the following classes to be immutable: the target class, String, Java primitive wrapper classes (e.g., Integer), and classes annotated with @Immutable.

When Immutator needs to use a clone method that does not exist, it generates a clone stub and reports this to the user, who must implement the stub.

Immutator avoids excessive cloning. For example, it could have inserted a clone call in the private constructor on line 11, but this would have caused unnecessary cloning. Instead, Immutator calls clone sparsely, at the location where objects enter or escape, or where the target class is transitively mutated (e.g., on line 30).

Moreover, Immutator ensures some structural sharing [42], by not adding calls to clone objects that enter from another instance of the same class. For example, when the transformed setRadius method is called, a new instance of ImmutableCircle is created (line 20 on the right-hand side). However, only the r field is mutated, while the c field (the center) remains the same. Since the old circle will not mutate the center, and since the center is not visible from the outside, the new circle does not have to clone it. The result is that the two circles share a part of their state.

3.5 Program Analysis

In the previous section we discussed the transformations to make an existing class immutable. In order to perform these transformations Immutator first analyzes the source code to establish preconditions and to collect information for the transformation phase.

Immutator does not perform a whole-program analysis, but only analyses the target class and methods invoked from it. Thus, the analysis is fast and can be used interactively.

At the heart of Immutator are two analyses. The first detects mutating methods so that these can be converted to factory methods. The second detects objects that enter or escape the target class so that they can be cloned. Both analyses work on a representation generated from byte code, and can therefore analyze third-party library code.
public void client() {
  Circle circle = new Circle();
  Point center = new Point(5, 5);
  circle.moveTo(center);
}

public class Circle {
  public void moveTo(Point p) {
    this.c = p;
  }
}

Figure 3.2: An example points-to graph

3.5.1 Analysis Data Structures

Immutator creates several data structures that are necessary for the program analyses. It constructs both of these data structures using the WALA analysis library [43] as a starting point.

The first data structure is a call graph (CG) starting from every non-private method of the target class. The call graph is used to find mutators as well as entering/escaping objects. For each node in the callgraph Immutator also constructs a control flow graph (CFG) that is used later to find transitive mutations and to build a points-to graph.

In addition to the control-flow structures, Immutator builds a points-to graph (PTG). Points-to analysis establishes which pointers (or references in Java terminology) point to which storage locations. We model the heap storage locations as object allocation sites.

An example of the points-to graphs that Immutator create is illustrated using a simple client program in Fig. 3.2. The graph contains two types of nodes: references, depicted graphically as ellipses, and heap-allocated objects depicted as rectangles. The explicit formal arguments of a method are placed on the border of its bounding box. Directed edges connect references to the objects they point to. For example, the object created on line 2 is represented by the rectangle Circle:2, and the reference it is assigned to on the same line is represented by the circle ellipse. This object has a field c, which is constructed in the field initializer of class Circle. References are connected to their objects by directed edges. The points-to graph only captures relations between references and objects, and does not include scalar primitives.

Notice that the assignment on line 9 creates an alias between the references c and p. This is represented in the points-to graph as a dashed arrow, and is called a deferred edge. A deferred edge means that c can point to any objects that p can point to. We also use deferred edges to represent the relations between formal and actual arguments since Java is a pass-by-value language where
actuals are copied into the formals.

\textit{Immutator} constructs this points-to graph using an inclusion-based (Andersen-style) points-to analysis. The analysis is partly \textit{flow-sensitive} with respect to local variables as it is computed from an SSA representation of the source code. It is also \textit{context-insensitive} since it does not take the calling context into account.

Note that \textit{Immutator} constructs additional nodes that do not exist in the program when they are needed to complete a method summary. One such example is the \texttt{Circle} allocation site and its \texttt{c} field in the \texttt{moveTo} method. When \textit{Immutator} creates the summary for \texttt{moveTo}, the \texttt{this} reference is not connected to any allocation sites. Therefore, \textit{Immutator} constructs additional object and field nodes in order to add the deferred edge that represents the assignment of \texttt{p} to \texttt{c}.

### 3.5.2 Detecting Transitive Mutators

The goal of this analysis is to find the methods that are mutating the transitive state of the target object, either directly or indirectly by calling another mutator method.

Fig. 3.3 shows the pseudocode of the algorithm for detecting mutator methods. The algorithm takes as input the set $M$ of methods in the call graph, the set $MTC$ of methods declared in the target class, and the points-to graph presented in Section 3.5.1. The output of the algorithm is a set $MUT$ of mutator methods.

In Step 1, the algorithm finds the objects and fields that represent the transitive state of the target class. To do so, the algorithm computes the transitive closure of the \texttt{this} references of the target class, i.e., all nodes in the points-to graph reachable from \texttt{this}. These nodes, called \texttt{TARG}, are the set union of all nodes reachable from the \texttt{this} reference in target class methods.

In Step 2, the algorithm finds all transitive mutating methods. These include mutators inside and outside (e.g., \texttt{setLocation()}, called on line 30) the target class. The algorithm visits every field assignment instruction in all the target class methods, as well as methods invoked from the target class. For each assignment it checks whether the left-hand side of the assignment is a reference node that may point to one of the objects in the transitive state of the target class. If it can, this means that the instruction assigns to the transitive state of the target class, and the algorithm marks the method as a direct mutator.

In Step 3, the algorithm propagates mutation summaries from direct mutators backwards through the call graph. If method $m$ calls $m'$ and $m'$ is a mutator, then $m$ becomes a mutator too. To
do this, the analysis visits, in reverse topological order (post-order), the methods in the call graph and merges the mutation summaries of the callees with the summaries of the callers.

3.5.3 Detecting Escaping and Entering Objects

The goal of this analysis is to find mutable objects that enter or escape the target class. These objects can be mutated by a client, thus mutating the transitive state of the target class. Therefore, the analysis finds and clones them.

The algorithm detects entering/escaping objects that are mutable and assigned/fetched to/from the transitive state of the target class.

Fig. 3.4 shows the pseudocode of the algorithm for detecting entering or escaping objects. The algorithm takes as input the points-to graph presented in Section 3.5.1. The output of the algorithm are two sets: ENT containing objects that enter the target class, and ESC containing objects that escape the target class.

In Step 1, the algorithm finds the nodes that form the transitive state of the target class. The transitive state, denoted by the TARG set, is the transitive closure of the this reference of every method in the target class.
**Input:**  
\[ API \leftarrow \text{Set of non-private methods in Target Class} \]  
\[ MTC \leftarrow \text{Methods in Target Class}, \]  
\[ PTG \leftarrow \text{Points-to Graph}, \]

**Output:**  
\[ ESC \quad // \text{Set of Escaping Objects} \]  
\[ ENT \quad // \text{Set of Entering Objects} \]

// Step 1: Find the transitive state of the target class  
\[ TARG \leftarrow \cup_{m\in MTC}(\text{transitiveClosure}(\text{this}, PTG)) \]

// Step 2: Find the transitive closure of the outside nodes  
\[ OUT \leftarrow \cup_{m\in API}(\text{transitiveClosure}(\text{actuals}, PTG) \]  
\[ \cup \text{transitiveClosure}(\text{returns}, PTG) \]  
\[ \cup \text{transitiveClosure}(\text{statics}, PTG) \]

// Step 3: Find the escaping objects  
for each deferred edge \( e \in PTG \) do  
if \( (e.\text{source} \in OUT) \) && \( (e.\text{sink} \in TARG) \) then  
\[ ESC \leftarrow ESC \cup e.\text{sink} \]

// Step 4: Find the entering objects  
for each deferred edge \( e \in PTG \) do  
if \( (e.\text{source} \in TARG) \) && \( (e.\text{sink} \in OUT) \) then  
\[ ENT \leftarrow ENT \cup e.\text{source} \]

Figure 3.4: Detecting entering and escaping objects

In Step 2, the algorithm finds the nodes that are outside of the target class, but that interface with it. Since these are the nodes through which client code interacts with the target class, they are also the nodes that objects can enter or escape through. We call these nodes the *boundary nodes*, as they are at the boundary of the target class.

The boundary nodes are:

- actual arguments passed to non-private (API) methods
- references returned from non-private methods
- static reference fields.

The algorithm computes the transitive closure of boundary nodes, and labels the resulting set \( OUT \).

In Step 3, the algorithm finds the escaping objects. Escaping objects are the objects in the transitive state of the target class that can be seen from methods outside the target class. To find
public Point getLocation() {
    return c;
}

(a) Example of an object escaping from getLocation

public void moveTo(Point p) {
    this.c = p;
}

(b) Example of an object entering through moveTo

Figure 3.5: Detecting escaping and entering objects

des these objects, the algorithm visits all the deferred edges that start in \textit{OUT} and end in \textit{TARG}. We are only interested in the edges that end in \textit{TARG}, because we only care about escaping objects in the transitive state of the target class. For such edges, the algorithm adds the sink target node to the \textit{ESC} set.

Fig. 3.5(a) shows an example of an escaping object. It shows the points-to graph for the \texttt{getLocation} method, with an additional node representing the return statement. We color the transitive state of the target class (which is the transitive closure of \texttt{this}) with orange. We then color the outside nodes with blue. In this example, the only boundary node is the return node, and its transitive closure includes \texttt{c}. Notice that \texttt{c} is colored with both blue and orange. This means \texttt{c} escapes because it can be seen from the outside (it is blue), and it is part of the transitive state of the class (it is orange).

In Step 4, the algorithm finds the entering objects. Entering objects are objects that are visible outside the target class methods, and that can be \textit{seen from the target class}. To find these objects, the algorithm visits all the deferred edges that start in \textit{TARG} and end in \textit{OUT}. We only visit edges that start in \textit{TARG}, because we only care about the entering objects that are assigned to a field in the transitive state of the target class. For such edges, the algorithm adds the sink outside
node to the $ENT$ set.

Fig. 3.5(b) shows an example of an entering object. It shows the points-to graph for the $\text{moveTo}$ method. As before, the transitive state of the target class is colored orange, and the transitive closure of the boundary nodes (i.e., the actual argument) are blue. The actual parameter is a part of the transitive state of the target class (it is orange), and it can be seen from the outside (it is blue). Therefore, objects may enter through it.

For pedagogical reasons, we chose simple examples to illustrate escaping and entering objects. In the codes illustrated in Fig. 3.5(a) and 3.5(b), it is very easy to spot the entering/escaping objects. However, in many cases they are more difficult to find, especially if objects enter or escape through containers, or escape through parameters. Section 3.7 shows an example of a state object escaping through an iterator container. The open-source developer overlooked this escaping object, but $\text{Immutator}$ correctly finds it.

3.6 Discussion

There are cases when the programmer wants only partial immutability. For example, the programmer wants some fields to be excluded from the immutable state of the class (e.g., a $\text{Logger}$ field), or some fields to be shallowly immutable. Or the programmer does not want to clone the entering/escaping objects (e.g., for performance reasons), but rather to document contracts. These are trivial extensions to $\text{Immutator}$ and require no additional analysis.

Currently, $\text{Immutator}$ handles most of the complexities of an object-oriented language like Java: arrays, aliases, polymorphic methods, and generics. It models arrays as an allocation site with just one field, which represents all the array elements. Although this abstraction does not allow $\text{Immutator}$ to distinguish between array elements, it allows $\text{Immutator}$ to detect objects that enter or escape through arrays. $\text{Immutator}$ disambiguates polymorphic method calls by computing the dynamic type of the receiver object using the results of the points-to analysis described in Section 3.5.1. $\text{Immutator}$ also preserves the generic types during the rewriting.

Limitations Since $\text{Immutator}$ analyzes bytecode, it correctly handles calls to third-party libraries. However, if the program invokes native code, $\text{Immutator}$ can not analyze it. Also, like any practical refactoring tool, $\text{Immutator}$ does not handle uses of dynamic class loaders or reflection.
**Future work**  We plan to solve the *usage problem*, i.e., updating the client code to use the transformed class in an immutable fashion.

Additionally, we will relax some of the constrains imposed by the current preconditions, to allow Immutator to transform more classes. For example, we could completely eliminate the requirement that the target class has no superclass/subclass (P1/P2), by allowing Immutator to transform a whole class inheritance hierarchy at once. Similarly, we could eliminate the requirement that mutators have a void return type (P3). Immutator could, for example, return a Pair object which encapsulates both the old return type, and the newly created object. Immutator would then have to change the callers of such methods to fetch the appropriate fields.

### 3.7 Evaluation

To evaluate the usefulness of Immutator we answer the following research questions:

Q1: How applicable is Immutator?

Q2: Is Immutator safer than manual transformations?

Q3: Does it make the programmer more productive?

All these questions address the higher level question “Is Immutator useful?” from different angles. Applicability measures how many classes in real-world programs can be directly transformed, i.e., they meet the preconditions. Correctness ensures that the runtime behavior is not modified by the transformation. Productivity measures whether automation saves programmer time.

#### 3.7.1 Methodology

We use a combination of three empirical methods, one controlled experiment and two case studies, that complement each other. The experiment allows us to quantify the programmer time and programmer errors, while the case studies give more confidence that the proposed algorithm and experiment findings generalize to real-world situations.

**Case Study #1 (CS1)**  We ran Immutator on all classes in 3 open-source projects, a total of 346 concrete classes. Table 3.1 shows the projects that we used: Jutil Coal 0.3, jpaul 2.5.1 and Apache Commons Collections 3.2.1.
We do not suggest that every class in a project should be immutable. That is not for a tool to decide. Rather, we evaluate how well the transformation works over all classes without imposing a selection criteria that could limit the generalization of the findings.

Case Study #2 (CS2) We also conducted case studies of how open-source programmers implement immutability. To find existing immutable classes in real-world projects we used two code search engines: krugle (www.krugle.org), and Google (www.google.com/codesearch). We searched for Java classes whose name contains the word ‘Immutable’ and classes whose documentation contained the word ‘Immutable’. These are classes that are likely to be immutable, and the documentation of these classes confirmed that the developers intended them to be immutable. We also searched for classes implementing an Immutable interface, a convention used in some open-source projects. In cases when we found errors in their immutable classes, we contacted the developers to ask for clarification.

Controlled Experiment We asked 6 experienced programmers (with an average of 7 years of Java programming) to manually transform for immutability 8 classes from the JHotDraw 5.3 framework. JHotDraw is an open-source 2D graphics framework for structured drawing editors.

We gave each programmer a 1-hour tutorial on making classes immutable, and then we asked them to transform one or two JHotDraw classes and report the time. We used classes from the Figure class hierarchy that made sense to become immutable. Since the Figure classes are part of a deep class inheritance hierarchy, we told the participants to treat the target class as if it was the only class in the hierarchy, i.e., to change only the target class. No programmer got a class larger than 400 LOC. We also used Immutator to transform the same classes (we relaxed the first
two preconditions), and we compared the results against a golden-standard.

To answer the applicability question, we wrote a statistics tool that applied the transformation to all classes in each project from CS1. For classes that did not pass all preconditions, the tool collected the failed preconditions. Since we ran Immutator in automatic mode, it only applied the transformation to classes that passed all preconditions. In interactive mode, Immutator could have transformed more classes, after the programmer addressed failed preconditions.

To answer the correctness question, we ran extensive test suites before and after all transformations from CS1. We only used projects that had extensive tests to help us confirm that the transformation did not break the systems. We also carefully inspected a few classes that we chose randomly.

To be able to run existing test suites, we wrote a tool that generates a mutable adapter between the immutable classes and the tests. The adapter has the same interface as the original class, but contains a reference to an instance of the immutable class. When a test calls a mutator, the adapter invokes the corresponding factory method of the immutable instance, and assigns the returned object to the reference. Our generated adapters were not adequate for 9% of the case study classes, due to not supporting static instance fields. Additionally, due to exceptions raised by our current implementation, we failed to analyze 20 of the classes in CS1. These were excluded from the reported data.

Furthermore, to compare correctness of manual versus tool-assisted transformation, we carefully analyzed the immutable classes that were produced manually in the second case study (CS2) and in the controlled experiment.

To answer the productivity question, we used Immutator to transform all the classes in Table 3.2 that met the preconditions. For each class, we report the number of methods that Immutator analyzed, as well as the number of source changes. We further broke this down into the total number of lines that had edits, the number of mutators that had to be converted to factory methods, and the number of entering or escaping objects that had to be cloned. We also report the time Immutator spent analyzing and transforming the code. For the controlled experiment, we asked each programmer to report the time spent to analyze and transform a class.
3.7.2 Results

To be useful, **Immutator** must be applicable, correct, and must increase programmer productivity.

Applicability

Table 3.2 shows that 33.74% of the classes in CS1 meet the preconditions without requiring any modification from the programmer. Out of the classes that failed preconditions, most are due to superclasses containing mutable state (P2), entering/escaping objects (P4), and mutators with non-void return values (P3).

However, keep in mind that a programmer would not select all classes, but rather the ones that provide benefit. We hypothesize that such classes are more likely to meet the preconditions. Even in cases when classes do not meet all preconditions, **Immutator** enables the programmer to identify issues with the push of a button.

Correctness

For each project in CS1, we ran the full test suite before and after the transformations. The transformations did not cause any new failures.

Table 3.3 shows that even expert programmers make errors when creating immutable classes. The last set of three columns show how many entering or escaping objects the open-source programmers forgot to clone, and how many mutating methods they still left in the immutable class.

<table>
<thead>
<tr>
<th>project</th>
<th>immutable class</th>
<th>programmer errors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>mutator</td>
</tr>
<tr>
<td>JDigraph</td>
<td>ImmutableBag</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>FastNodeDigraph</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>HashDigraph</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>ArrayGrid2D</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>MapGrid2D</td>
<td>-</td>
</tr>
<tr>
<td>WALA</td>
<td>ImmutableByteArray</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>ImmutableStack</td>
<td>-</td>
</tr>
<tr>
<td>java.util.collections</td>
<td>ImmutableEntry</td>
<td>-</td>
</tr>
<tr>
<td>Guava</td>
<td>ImmutableEntry</td>
<td>-</td>
</tr>
<tr>
<td>peaberry</td>
<td>ImmutableAttribute</td>
<td>-</td>
</tr>
<tr>
<td>Spring</td>
<td>ImmutableFlow-AttributeMapper</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3.3: Immutability errors in open-source projects

2 Does not include the adapter class
We confirmed with the open-source developers that our findings indicate genuine immutability errors in their code, and that developers meant those classes to be deeply immutable. Most agreed that their implementation choice was an incorrect design decision or was made for the sake of performance. Furthermore, the JDigraph developers took our patch and fixed the errors.

We illustrate one of the subtleties of detecting escaping objects with a real example from WALA’s ImmutableStack:

```java
public Iterator<T> iterator() {
    if (entries.length == 0) {
        return EmptyIterator.instance();
    }
    return new ArrayIterator<T>(entries);
}
```

This immutable class lets some internal state stored in the `entries` field escape. At first sight, it is not obvious, since no fields are directly returned. However, `entries` is passed to an `ArrayIterator` that is returned, so `entries` escapes too. Client code can now use the returned iterator to fetch any element from `entries` and mutate it directly.

Table 3.4 shows the data for the controlled experiment. Programmers made errors similar with the ones in CS2. However, the density of errors was higher: 6.37 errors/class. The manual inspection of the immutable classes generated by our prototype implementation revealed 4 bugs. None of these were inherent to the algorithm.
Productivity

Table 3.1 shows that Immutator saved the programmer from editing 45 lines of code per target class on average. More important, many of these changes are non-trivial: they require analyzing 57 methods in context to find transitive mutations, entering and escaping objects. In contrast, when using Immutator, the programmer only has to initiate the transformation. On average, Immutator analyzes and transforms a class in 2.33 seconds using a Macbook Pro 4.1 with a 2.4 GHz Core 2 Duo CPU. Compared to the time taken to manually transform a class in the controlled experiment, 27 minutes, this is an improvement of almost 700x.
CHAPTER 4
DATATYPE EXTRACTION

4.1 Introduction

Data movement is a fundamental part of parallel applications on distributed memory machines. It is also often the most costly operation in terms of time and energy, and usually does not scale as well as the rest of the application. It is therefore essential that we move data as few times as possible, and move it efficiently when data movement can not be avoided.

Datatypes are objects that describe data layout. Examples include strided layouts and indexed layouts. In strided layouts each block is located a fixed stride from the previous block. In indexed layouts each block’s location is described by a displacement from the first block. When datatypes are used with a data movement operation such as send, receive, put or file write, non-contiguous data can be transferred with a single call to the underlying runtime.

Datatypes are powerful tools that enable runtimes to optimize data movement in several ways. First, multiple blocks can be packed into a contiguous buffer and sent in one operation. Even with the additional copy pass, this is often a significant improvement over multiple small blocking transfers due to the high latency cost associated with each transfer. Such packing can also be performed by user packing code. This is a common idiom in scientific applications [44]. However, by describing non-contiguous data using datatypes, the runtime is free to determine the best way to transfer the data for a given system. It can optimize data packing [20, 21, 19], or use an alternative communication scheme, such as pipelining multiple small asynchronous messages [45].

Moreover, many network controllers support direct memory access (DMA) and can access user memory without involving the CPU. Some networks, such as InfiniBand [22], also provide support for non-contiguous data transfers. This is a very powerful feature that enables zero-copy transfers [23, 46], which avoids memory-to-memory data copying. This can significantly reduce communication overheads [44, 46], but is only possible if the data layout is provided as an argu-
ment in the communication call.

Finally, since datatypes allow known data layouts to be specified declarative and concisely they increase the readability of the code. This is important to reduce the cost of software maintenance, an activity that may span decades.

MPI is a parallel programming model where a rich set of datatypes are available to the programmer [47]. The programmer can specify contiguous and vector (strided) datatypes, as well as homogeneous and heterogeneous indexed datatypes. Furthermore, the programmer can describe complex data layouts efficiently by composing datatypes hierarchically.

A more restricted set of datatypes are available in several important network layer API’s, such as BSD Sockets, ARMCI [48] and Infiniband OFED [49]. They are also planned for inclusion in GASNet 2.0 [50], which is the target API for the Berkeley UPC Compiler.

However, datatypes are often underused [44, 21]. We believe there are at least two reasons for this. First, runtimes were traditionally not optimized for datatypes and datatypes were therefore often slower than manual packing loops. However, this has changed and runtimes are now better optimized [20, 21, 19], and provide powerful features such as non-contiguous zero-copy transfers that can only be exploited if datatypes are used [23, 46]. Second, datatypes are considered hard to construct. They are declarative, and require programmers to precisely describe exact data layouts in a terse way. Furthermore, they are very hard to debug since no tools are available to track the memory locations accessed by a communication library. However, once constructed their terseness and preciseness makes them easy to reason about. In short, datatypes declare the structure of the data instead of the logic required to pack it.

Moreover, parallel programming models such as UPC [51] and CAF [52] do not expose datatypes to the programmer (although a UPC extension exist that implicitly supports datatypes through indexed and strided copies). Therefore, the programmer has no other choice than to express non-contiguous data transfers using either multiple sends or packing code.

The potential performance, performance portability and readability benefits of using datatypes, combined with the difficulty in creating them for humans and the lack of means to express them in some languages, motivates an automated approach. Since packing code is a common idiom for sending non-contiguous data, a technique for converting packing code to datatype code is needed.

For programming models that expose datatypes to the programmer, a refactoring tool or a source-to-source compiler should be used to port packing code to datatype code. For programming
models where datatypes are not exposed to the programmer, such as UPC and CAF, a compiler pass that converts packing code to datatype code is desirable. This allows the compiler to output code that takes advantage of datatype capabilities in the network or uses pipelined asynchronous messages where this is supported, or that optimizes the packing for the target architecture where it is not.

We present an algorithm that converts packing code to datatype code. The algorithm first converts the packing code to an intermediate representation (IR) called the Datatype IR. The IR captures the essential information required to generate datatypes. The algorithm then performs a number of specialization and compression passes to optimize the IR so that efficient datatypes can be generated. After the algorithm has produced a datatype description of the layout of the packed data, the IR can be used to replace the existing packing code with datatype code. The presentation assumes C and MPI, but the techniques generalize to other environments.

We implemented our algorithm as a refactoring plugin for C with MPI on top of Eclipse CDT. We used this implementation to evaluate the approach on the NAS Parallel LU Benchmark. The evaluation shows that the algorithm is applicable to real world code and that it finds good datatypes.

4.2 Illustrative Example

We present a running example to illustrate some of the challenges of converting packing code to datatype code. The running example is a border-exchange between two nodes that execute a two dimensional iterative stencil computation. The example is similar to one of the simpler packing loops in the NAS LU benchmark. The example is implemented in C99 using MPI. As mentioned before, MPI provides a rich API for constructing datatypes and is therefore an excellent target for our algorithm. We first provide a quick introduction to MPI datatypes in section 4.2.1, before we describe the details of the border exchange in section 4.2.2.

4.2.1 MPI Datatypes

MPI Derived Datatypes are opaque objects that describe data layout [47]. They define a type map, which is an ordered sequence of primitive datatypes and displacements. Primitive datatypes correspond to the primitives in C and Fortran, and include such types as MPI_DOUBLE and MPI_INT.
Datatypes are a central concept in MPI and are used with message sending/receiving, remote direct memory access (RDMA), and file IO to specify the data to send/write.

MPI provides several constructors that can be used to construct derived datatypes. These include `MPI_Type_vector()` and `MPI_Type_create_hindexed()`. The former creates a derived datatype given a displacement list, while the latter creates a derived datatype that describes a vector with a given count, block length and stride.

Even though MPI provides different constructors to produce derived datatypes, it only formally distinguishes between derived and basic datatypes. The constructors merely provide different ways to create such derived datatypes. However, in this work we consider the output of different constructors to be different datatypes. In other words, `MPI_Type_vector()` constructs a vector datatypes, which are different from the contiguous datatypes produced by `MPI_Type_contiguous()`.

### 4.2.2 Border Exchange Example

Iterative stencil computations on structured grids are a common class of algorithms. They are used in weather and atmospheric simulations, fluid dynamics, and many other applications. The input to many stencil computations is a regular, \( N \)-dimensional mesh. The values of the mesh points are updated iteratively based on the values of surrounding points from the previous iteration. All points use surrounding points in the same relative positions. This set of relative positions is called a stencil.

Such computations are implemented on distributed memory systems by partitioning the mesh, and adding halo points at the boundary of each partition. Halo points are replicas of border points from neighboring partitions that are needed to compute the local updates. The halo region is updated between every iteration in a process called a border exchange [53].

![Figure 4.1: 5-point stencil Computation on two processors with ghost cells (grey) and border exchanges.](image-url)
Figure 4.2 illustrates a stencil computation on two processors with a 5-point stencil. Each point contains three double values, and the computation uses border exchanges to communicate border values. The white cells form the domain of the computation. The gray area shows the ghost cell region of each processor.

Figure 4.2(a) illustrates how the left ghost cell region of processor 2 is packed into a communication buffer, and figure 4.2(b) shows the corresponding C99 source code. The code consists of a loop with index i that is used to address both the source and the destination of the packing operations. Inside the loop there are three packing statements that packs the three values of each border cell into the packing buffer.

Figure 4.2(c) shows the equivalent MPI datatype code. The source code constructs a vector with N blocks, 3 elements per block, and an N * 3 element stride between each block. The vector is then used to parameterize the MPI_Send and specifies the layout of the data to send, relative to location &grid[0][0][0].

The left Ghost Cell Halo is packed/copied into the send buffer

```c
double buffer[N*3];
for (int i = 0; i < N; i++) {
    buffer[3*i] = grid[i][0][0];
    buffer[3*i+1] = grid[i][0][1];
    buffer[3*i+2] = grid[i][0][2];
}
MPI_Send(buffer, N*3, MPI_DOUBLE, left, tag, comm);
```

C99 packing code that copies the left ghost cell halo into a send buffer

```c
MPI_Datatype vec_t;
MPI_Type_vector(N, 3, N*3, MPI_DOUBLE, &vec_t);
MPI_Type_commit(&vec_t);
MPI_Send(&grid[0][0][0], 1, vec_t, left, tag, comm);
```

An MPI_TypeVector that describes the layout of the data to be sent
4.3 Datatype IR

The algorithm we describe in this work automates the conversion from packing code to datatype code. That is, given the code in figure 4.2(b) it produces the code in figure 4.2(c). The input to the algorithm is a send call that sends data from a packing buffer and is from here on referred to as the target send call.

For ease of presentation, we only discuss packing code that copies data from user memory to a communication buffer. Unpacking code is totally symmetric and is handled in the same way.

The problem of extracting datatypes from packing statements can be broken down into a sequence of sub-problems that can be solved independently. Central to this process is an intermediate representation we call the Datatype IR. This intermediate representation can be constructed from imperative languages with packing code, such as C, C++, Fortran and UPC, and captures the necessary information to generate efficient datatypes. Once constructed it describes a datatype that is equivalent to the packing code, and can be used to replace the packing code with a code that constructs a datatype. The Datatype IR is the focal point of our algorithm and serves to simplify each stage.

Figure 4.3 outlines the Datatype IR. The IR describes the relationship between packing groups in the source code and datatypes. Packing groups are composed hierarchically, and describe a collection of packing statements.

A Packing Group is one of:

**Packing Statement** A statement that copies a value from user memory to a packing buffer. The IR node for a packing statement contains the memory location the value is retrieved from, as well as the location in the packing buffer it is packed into.

**Packing Sequence** A sequence of packing groups. The IR node for a packing sequence contains the sub-groups, as well as expressions that describe the symbolic *distance* between every pair of consecutive sub-groups in the source data structure. Section 4.3.2 describes how distances are calculated.

**Packing Loop** A loop whose body is a packing group. The IR node for a packing loop contains the body sub-group as well as a summary of loop induction variables and loop constants. Induction variables are discussed in section 4.3.2.
Since the definition is recursive, arbitrary collections of packing statements can be described by a packing group.

Every packing group has a source location associated with it. The location describes the first location in the source data structure that the packing group reads from. Thus, the location of a packing group is identical to the location of its first packing statement. The location of a packing statement is a pair of the form (name, displacement). The first element is a named variable or pointer that is used to access the source data structure. The second element is an expression that describes the offset from the name where the packed element is located. The displacement could either be an index, in the case of arrays, or a field describing the struct element that is accessed. For example, the location of the following packing statement \( b[j] = a[i+1] \), where \( b \) is a packing buffer, is the pair \((a, i + 1)\). This would also be the location of the equivalent packing statement \( b[j] = *(a+i+1) \), where \( a \) is a pointer.

A more complex example is \( b[i] = a[i][j][2] \), where the array \( a \) is defined as \( a[D_1][D_2][D_3] \). In this case the location is determined by symbolically multiplying each array subscript by the size of every inner dimension, and then adding these products together. That is, given the source data structure access expression \( A[s_n][s_1][s_0] \), the location expression is symbolically computed using the formula \( \sum_{j=0}^{n} (s_j \prod_{k=0}^{j-1} \text{dim}_k) \). The location of the example thus becomes \((a, iD_2D_3 + jD_3 + 2)\).

In the IR each packing group is mapped to exactly one datatype, while a datatype can describe multiple packing groups. The datatypes in our IR match the datatypes in MPI. These form a powerful set of datatypes that can be composed to describe any data layout, so the same IR can handle other environments. The supported datatypes are depicted on the right in figure 4.3, where they are ordered from more general (top) to more efficient (bottom). The details of each datatype will be described in section 4.4.1 as the step that specializes the IR to use them is discussed.

Before Datatype IR can be constructed a number of preconditions must be checked. The preconditions are described in section 4.3.1, and must be met before our algorithm can create datatypes and rewrite code. The algorithm can convert any packing code that satisfies these preconditions to datatype code.

Section 4.3.2 describes how the IR can be constructed from packing code written in C. Once the initial IR has been generated, it is optimized by subsequent passes as described in section 4.4.
4.3.1 Preconditions

The algorithm targets packing code blocks: a sequence of statements that copy data from program data structures into a buffer that is consumed by the target send call. The algorithm checks that four preconditions are met before it generates the IR:

1. The code block is “straight-line code”: perfectly nested loops, with buffer assignments, primitive variable assignments and loops, with index values that do not depend on values in the buffer

2. The buffer content that is produced by the code block is only consumed by the MPI library (e.g., by a send)

3. The code block writes into consecutive locations in the buffer

4. The code block must directly proceed the target send call.

The first condition is necessary in order to support a declarative definition of the source locations. The second implies that it is safe to not update the buffer (that is, to remove the packing code block). In packing code blocks where the third condition does not hold it can often be established by an initial step where the packing code is reordered. The reordering involves known techniques such as code motion and loop transformations (loop split and loop reversal). Similarly, in code where the fourth condition fails it can often be established by moving the packing code ahead of the target send call, through code motion and function inlining. However, loop transformations, code motion, and inlining have been studied extensively before and are out of the scope of this work.
4.3.2 Datatype IR Construction

The Datatype IR can be constructed from any imperative language with packing code, but we will discuss it for C with MPI.

The first step in constructing the IR is to find the packing statements that produce the packing buffer values that are sent in the target send call. These can be found by traversing the control flow graph (CFG) backwards from the send call until the packing statement that packs into the first buffer location that is sent is found. The code between the first packing statement and the target send call is checked against the third and fourth preconditions. If both conditions hold, this code contains the packing statements that produces values for the target send call.

Once the packing statements have been found, packing sequences and packing loops are constructed. Every packing group inside a packing loop, or at the top level, is added to the same packing sequence. Additionally, packing groups inside loops are added to a new packing loop node that represents that loop.

For each packing sequence the distance between every consecutive pair of sub-groups is computed. The distance between two packing groups is only defined if their locations are in the same array, and none of the index expressions are conditionally redefined between the packing groups. If these conditions hold, then the distance is the location index expression of the second packing group minus the location index expression of the first group. For example, given two packing statements that pack the variables at location \( a[i] \) and \( a[i+k] \) the distance is \( i+k - i = k \), assuming \( i \) is not redefined between the packing statements.

If it is redefined then the index expression of the second packing statement must be updated to reflect this before the distance is computed. That is, if \( i \) was incremented by one between the
packing statements, then the distance would be \((i + 1 + k) - i = k + 1\). In general this would require reaching definitions analysis. However, precondition 1 require the code to be “straight line code” (perfectly nested loops without conditionals) which makes the analysis straight forward.

For each packing loop, loop constants and loop induction variables are summarized using standard approaches \([28, 30]\). A discussion of these can be found in section 4.4.1.

Finally, to complete the IR the packing groups are assigned datatypes. Structs are the most general datatypes, and can be used to describe the layout of the data packed by any packing code that passes the preconditions in section 4.3.1. The IR construction stage therefore assigns a struct datatype to each of the packing groups. Figure 4.4(a) shows the resulting IR for our border exchange example. The loop is represented by a struct that conceptually contains \(N\) sub-structs, one for each iteration of the loop. However, only one struct node representing the \(N\) sub-structs is actually constructed in the IR. This sub-struct represents the packing sequence in the loop body and have 3 double sub-types, one for each packing statement.

### 4.4 Datatype Optimizations

The previous section described how to construct an IR representing a correct datatype. Code can be emitted to construct a hierarchy of structs, and each packing statement can be replaced with statements that store displacements, block lengths and types.

However, the datatype constructors would require a number of arguments proportional to the number of values in the packing buffer. Furthermore, observe that the code to assemble these arguments would mirror the original packing code, effectively replacing packing code with code that packs displacements. Note that this code may still be preferable to packing code as it only needs to be run once to construct a datatype that can be reused. Moreover, the displacement construction code stores addresses computed on the CPU instead of values loaded from memory, so it is likely to be faster than packing code. Therefore, to improve efficiency and readability, and to ensure we can exploit hardware features such as vector send, we perform a sequence of optimizations to to yield more efficient datatypes.

We consider a datatype to be more efficient than another datatype if it can be described with fewer arguments. A contiguous type is therefore more efficient than a vector, which is more efficient than an indexed type. That is, a contiguous type only requires two arguments (count and subtype), while a vector requires four (count, block length, stride and subtype). Furthermore, an indexed
For each datatype, bottom up

- Specialize to hindexed?
  - yes
  - Specialize to hvector?
    - yes
    - Specialize to vector?
      - yes
      - Specialize to contiguous?
        - yes
        - Compress into parent block length?
          - no
          - yes
          - yes
          - yes
          - yes

- Merge structs and indexed types

- Compress contiguous type into send count

**Figure 4.5: Datatype Specialization**

A datatype requires $2n + 2$ arguments, where $n$ is the number of blocks it describes. Efficient datatypes are likely to result in higher performing code, and they also tend to be more readable.

To generate efficient datatypes the algorithm performs a number of optimizations. There are two types of optimizations: *specializing substitutions* that replace a more general datatype with a more specialized and efficient one, and *datatype compressions* that merges datatypes. Figure 4.5 shows how the optimizations are ordered. The sequence of specialization optimizations are applied to one datatype at a time, bottom up. If any specialization, except from specialize to hindexed, fails then the algorithm continues to the next datatypes. Except from the compress into parent block length, the compression optimizations are performed after all the specializations have been ap-
plied. The compress into block length optimization is performed interleaved with the specialization optimizations, as it may expose additional specialization opportunities.

The resulting intermediate representation describes an efficient datatype that is equivalent to the packing code. Specialization substitutions are described in section 4.4.1 and datatype compressions are described in section 4.4.2.

4.4.1 Specialization Optimizations

The datatypes to the left in figure 4.3 forms a strict order from the most general at the top (structs), to the most efficient at the bottom (contiguous types). Each specialization makes a datatype less general and therefore has preconditions that must be satisfied before the specialization can be applied. As shown in figure 4.5, the algorithm specializes one datatype at a time, bottom up, along the specialization chain until it can’t be specialized further. That is, structs are specialized to hindexed types, which are specialized to hvectors and so on.

Note that the indexed and indexed block datatypes are in a branch of to the side of the main specialization chain. Given an hindexed datatype there are two possible specializations. It can be specialized to a hvector datatype, or to an indexed datatype. Our algorithm attempts to specialize it to an hvector first as this is the most efficient of the two. Only if the hvector specialization fails will the algorithm attempt specialization to indexed datatypes.

The following sections describe each specialization transformation as well as their preconditions.

Struct to hindexed specialization

After IR construction every datatype is a struct. The first specialization therefore transforms struct datatypes to hindexed datatypes.

Hindexed datatypes are similar to structs as both specify a byte displacement for each block. However, they add the constraint that all subtypes must be the same datatype. While struct datatypes allow a different subtype to be specified for each block, hindexed types only allow a single subtype to be specified for all the blocks. This transformation therefore requires that every struct block has the same subtype. If they do then the struct is specialized to an hindexed datatype.

In our running example the struct representing the packing sequence can be specialized to an hindexed type as every subtype is an `MPI_DOUBLE`. The struct representing the packing loop can also be specialized to an hindexed type as all of its subtypes are the same.
Hindexed to hvector specialization

Hvector datatypes describe data layouts as a sequence of blocks with a fixed byte stride between each block. This is a significant constraint over hindexed datatypes, where each block is described by independent byte displacement. However, it also leads to a significant reduction in the number of argument required to specify the datatype (down to four from linear), and hence a significant improvement in its efficiency. Three preconditions must be met before hindexed datatypes can be specialized to hvector types:

Every packing statement must access the same source array, struct or scalar variable
C, like most imperative languages, does not provide any guarantees for the location of arrays, structs or scalar variables in memory. Since hvectors require the same stride between the source data structure locations of each pair of consecutive packing groups, these locations must be in the same array, struct or scalar variable.

All block lengths must be the same size
Hvector datatypes only allow one block length to be specified, which applies to every block. The block length of each block must therefore be the same, and this can be established by symbolically comparing the block length expressions of each hindexed block.

There must be a fixed distance between the locations in the source data structure accessed by every pair of consecutive packing statements
As mentioned above, hvectors describe data blocks that are located a fixed stride apart. The distance between the source data structure locations of each consecutive packing group described by the hvector must therefore be fixed. There are two cases to consider when determining whether this is the case, namely packing sequences and packing loops.

Packing Sequences
For a packing sequence to be specialized to an hvector the distance between each consecutive pair of packing groups must be the same. As described in section 4.3.2 the IR construction phase computes these distances. The hindexed to hvector specialization pass can therefore simply compare the distance between every consecutive pair of packing groups symbolically.

Consider the packing sequence from our running example in figure 4.2(b), which consists of three
packing statements. The location index expression of the first statement is $i \times N \times 3$, the index of the second statement $i \times N \times 3 + 1$, and the index expression of the third statement $i \times N \times 3 + 2$. The distances between consecutive packing statements are therefore:

$$((i \times N \times 3 + 1) - (i \times N \times 3), (i \times N \times 3 + 2) - (i \times N \times 3 + 1))$$

Determining that the expressions describing two distances are the same requires expression simplification, which is key to the effectiveness of our algorithm. However, expression simplification is widely used in mathematical software, and some techniques have been discussed in the literature [25, 26]. One simple method is to convert each expression into a polynomial and then put the polynomial into a canonical form. This can be done efficiently, and allows comparison in linear time by comparing every term independently. After symbolic simplification the above expression becomes $(1, 1)$, and since the distances are the same the packing statements can be represented by an hvector. The hvector stride is the expression describing the distances, multiplied with the size of the elements in source data structure. In the above example the stride therefore becomes `sizeof(double)`.

**Packing Loops**  As defined earlier, packing loops only contain one sub-group. However, the nature of loops means this sub-group may be executed many times. Loop induction variables are variables whose value changes from one loop iteration to the next. If the location displacement expression of the loop’s sub-group is an index expression that contains an induction variable, then it will describe different memory locations in each iteration.

Linear induction variables are a sub-class of induction variables whose values increase or decrease by a fixed amount in each loop iteration. That is, their progression follows an arithmetic sequence. A linear induction expression is an expression that only contains loop constant expressions and linear induction variables, and the linear induction variables are only combined through addition and subtraction. The values of such an expression also follow an arithmetic sequence. If the location index expression of the loop’s sub-group is a linear induction expression, then it can be represented by an hvector. The stride of the arithmetic sequence described by the linear induction expression becomes the hvector’s stride.

Non-linear induction variables are induction variables whose progression follows some non-arithmetic sequence. One example is an induction variable that is multiplied by itself in each loop iteration,
thus following a geometric sequence. If non-linear induction variables are used in the location index expression of a loop’s sub-group, then that loop can not be described by an hvecto.

Linear induction variables can be summarized using standard approaches [28, 30], and a simple adaptation of these approaches can be used to summarize induction expressions. These algorithms summarize every linear induction variable as an affine function of the form \( f(i) = c_1 biv + c_2 \), where \( biv \) is a basic induction variable, and \( c_1 \) and \( c_2 \) are both loop constant expressions. \( c_1 \) describes the induction variables stride relative to the stride of the basic induction variable, and \( c_2 \) describes the induction variables value in the first iteration of the loop.

A basic induction variable is a variable that is incremented or decremented by a constant expression in each loop iteration. The variable \( i \) in figure 4.2(b) is an example of a basic induction variable, and is described with the affine function \( 1 \times i + 0 \), i.e. in terms of itself. Note that \( c_2 \) is 0 as the induction variable has the value 0 in the first iteration of the loop. This can be determined using a standard reaching definitions analysis [30]. Variables that are not described in terms of themselves are called dependent induction variables. An example of a dependent induction variable would be \( j = i \times 2 \). The affine function of \( j \) would be \( 2i + 0 \).

Given a list of the affine expressions describing linear induction variables, we compute their stride. The stride of a basic induction variable is simply the constant expression \( c_1 \), while the stride of a dependent induction variable is the stride of \( c_1 \) multiplied with the stride of its basic induction variable. Induction expressions are summarized the same way as dependent induction variables, and their stride are also computed the same way.

In the example in figure 4.2(b) the loop is a packing loop that contains a packing sequence. The location of the packing sequence is the same as the location of its first packing statement, namely \((grid, i \times N \times 3)\). As mentioned above, \( i \) is a basic induction variable. The affine function describing the stride of the location’s index expression is therefore \( N \times 3 \times i + 0 \). Since the stride of \( i \) is 1, the stride of the location’s stride expression is \( N \times 3 \).

Since the packing sequence location is a linear induction expression the packing loop can be described by a hvecto datatype, with a stride of \( N \times 3 \times \text{sizeof(double)} \).

**Hvecto to vector specialization**

Vectors are like hvectors, with the exception that their strides are given in number of elements (multiples of the subtype extent) instead of bytes. That is, a vector’s stride can be multiplied by
its subtype’s extent to yield its stride in bytes.

This places an additional constraint on the packing code. For an hvector to be specialized to a vector, the hvector’s stride must be divisible by the extent of it’s subtype. One test for this is to evaluate whether each term of the hvector’s stride contains a one or more factors that are equal to the subtype’s extent.

If the hvector stride is divisible by the subtype extent, then the hvector can be specialized to a vector and the vector’s stride is the quotient of the division.

In our running example the stride of the hvector describing the packing sequence is $\text{sizeof(double)}$. Furthermore, the extent of the MPI\_DOUBLE subtype is also $\text{sizeof(double)}$. The hvector stride is clearly divisible by the subtype extent, and the hvector can therefore be specialized to a vector with a stride of 1.

The running example also contains a packing loop that has so far been specialized to a hvector. The hvector’s subtype is the datatype describing the packing sequence, and its extent is $3 \times \text{sizeof(double)}$. This can clearly divide the stride of the packing loop’s hvector, which has a stride of $N \times 3 \times \text{sizeof(double)}$, and the quotient of the division is $N$. The packing loop’s hvector can therefore be specialized to a vector with stride $N$.

Vector to contiguous specialization

The last IR transformation along the main specialization chain is to specialize vectors to contiguous datatypes. A contiguous datatype describes elements that are laid out contiguously in memory. The precondition for this transformation is therefore that the stride of the vector, which describes the distance between the start of each block, is equal to its block length.

In our example, the stride and block length of the vector describing the packing sequence are both 1. It can therefore be specialized to a contiguous type. However, the vector describing the packing loop has a stride of $N$ and a block length of 1, and since these differ, no further specialization is possible.

Figure 4.4(b) shows the Datatype IR after the specializations described in sections 4.4.1–4.4.1 has been applied. The packing loop is described by a vector with count $N$, block length 1, and stride $N$. Its subtype is the packing sequence, which is described by a contiguous datatype with count 3 and whose subtype is MPI\_DOUBLE.
Hindexed to indexed specialization

If the hindexed to hvector specialization fails then the algorithm attempts to specialize hindexed
types to indexed types. Indexed types are similar to hindexed types, but the displacements are
given in elements (multiples of the subtype extent) instead of bytes.

This imposes two preconditions. First, the every packing statement in the packing group that is
described by the hindexed type must pack elements from the same source array. This is necessary
because indexed types are element strided, and most imperative languages provide no guarantees
for the relative locations or alignment of independent arrays. Second, every displacement must be
shown to be divisible by the subtype’s extent. This is similar to the precondition for specializing
an hvector to a vector.

If both of these preconditions are met, then the hindexed type is specialized to an indexed type.
This involves dividing each displacement by the extent of the subtype.

Indexed to indexed_block specialization

The last specialization that we will discuss is the specialization from indexed types to indexed_block
types. An indexed_block type is an indexed type where the length of every block is the same. Thus,
if the block length of every block is the same expression, then the indexed type is specialized to an
indexed_block type.

4.4.2 Compression Optimizations

The last class of transformations is the compression transformations, which remove superfluous
datatypes from the IR. This results in a datatype hierarchy that can be created with less code,
that is more readable, and that convey more precise information to the runtime.

Figure 4.4 shows the compression transformations as taking place after all the specializations have
been completed. This is a small simplification that was made for ease of presentation. In practice
some of the compression optimizations must be interleaved with specialization transformation as
they expose additional specialization opportunities.

In section 4.4.1 the hvector representing the packing loop in our running example could be
specialized to a vector because the hvector stride, $N \times 3 \times \text{sizeof(double)}$, was divisible by the
subtype extent, $3 \times \text{sizeof(double)}$. However, say we were sending only two of the three doubles
such that the subtype’s extent is $2 \times \text{sizeof}(\text{double})$. This is not divisible by the hvector stride and would therefore prevent specialization to a vector. However, if we fold the contiguous type that describes the subtype into the block length of the hvector, as described below, then the vector’s subtype would become an MPI.DOUBLE. This would divide any hvector stride given in doubles, thus allowing the packing loop hvector to be specialized.

The *compress contiguous type into parent block lengths* compression is therefore applied after the subtype has been fully specialized, but before the parent datatype is specialized. The following sections present the compression transformations. For each transformation we state where in our algorithm it is applied.

Figure 4.4(c) shows the final IR for our running example after all the compressions have been applied. The only compression that was applicable in this case was the compress contiguous types parent into block length compression. This compression replaced the contiguous type with a corresponding increase in vector block length. Note that since the stride of vector datatypes is given in multiples of the subtype, the stride has increased by a factor equal to the count of the contiguous type (that is, from $n$ to $n \times 3$).

Compress contiguous types into parent block length

Contiguous types describe elements laid out contiguously in memory. However, struct datatypes provide a block length argument that specifies the number of contiguous elements in each block.

If a struct has a contiguous subtype, then that subtype can be folded into the struct’s block length argument. This involves three operations. First, the subtype of the struct is replaced with the subtype of the contiguous type. Second, the block length argument of the struct is symbolically multiplied by the count of the contiguous type. Third, the contiguous type is deleted.

This transformation is applied to a struct datatype and its subtype after the subtype has been fully specialized, but *before* the parent struct is specialized. It is therefore unnecessary to fold contiguous types into the block length of other datatypes, as folding will have taken place before specialization.

Merge structs and indexed types

If a struct, hindexed, indexed or indexed_block datatype has a subtype of the same type, then the subtype can be merged with the parent datatype. This involves making the subtype’s subtype the
subtype of the parent, symbolically multiplying the subtype’s count with the parent’s count, and deleting the subtype. This transformation is applied after all datatypes have been fully specialized. For example, consider the case where the IR in figure 4.4(a) could not be specialized at all. In that case the merge struct compression would merge the two structs, resulting in one struct of size \( N \times 3 \).

Compress contiguous types into send count

Send functions provide a count argument specifying the number of layouts described by the datatype to send. Thus, if the top datatype is a contiguous type then it can be folded into the send count. This involves multiplying the send count by the count of the contiguous type, replacing the send call’s datatype with the contiguous type’s subtype, and deleting the contiguous type. This compression transformation is applied after all specialization has been completed.

4.5 Evaluation

Previous work has established that message bandwidth can be increased significantly if network support for non-contiguous memory transfers is exploited. Taking advantage of such advanced network features with MPI requires datatypes to be specified. Wu, et al. showed vector bandwidth improvements up to a factor of 3.6 for large messages compared to manual packing code when using their Multi-W technique for InfiniBand [23]. Similarly, Santhanaraman et al. showed vector bandwidth improvements up to a factor of 4.75 for large messages using their SGRS technique for InfiniBand [46]. They also report low CPU utilization as communication workload is offloaded to the network, which indicates increased potential for exploiting communication-computation overlap. Furthermore, Worringen, et al. report performance improvements up to a factor of 2.1, from transferring non-contiguous data using their direct_pack_ff algorithm instead of a generic pack/unpack implementation [54]. The direct_pack_ff algorithm takes advantage of features in the SCI interconnect [55] to copy non-contiguous data directly to global memory. Finally, Tanabe and Nakajo demonstrated a performance improvement up to a factor of 6.8 from accelerating MPI Datatypes using their DIMMnet-2 RDMA system, as compared with a software implementation [56]. All of these experiments were conducted using synthetic micro-benchmarks.

Furthermore, multiple research groups have demonstrated that datatypes can have comparable or
better performance than manual packing code, even if the network does not support non-contiguous transfers. Ross et al. showed that an optimized MPI implementation can have comparable performance to manual packing code when transmitting common data structures such as vectors and 3D faces [21]. Moreover, Byna, et al. provide a technique that outperforms manual packing code by as much as 205% for a matrix transpose by taking advantage of knowledge of the memory system to improve memory access cost [20]. Finally, Hoefler and Gottlieb demonstrate speedups up to a factor of 3.8 and 18% for a Fast Fourier Transform and a conjugate gradient solver respectively by expressing communication using datatypes [44].

This firmly establishes datatypes as a useful way of specifying non-contiguous transfers that can provide speedups given the right hardware. Furthermore, we expect network capabilities to improve in the future making datatypes increasingly relevant.

To evaluate the usefulness of our algorithm we therefore answer the following research questions:

1. Is the algorithm applicable?

2. Does the algorithm produce efficient datatypes?

4.5.1 Methodology

To evaluate our algorithm we implemented it in a refactoring tool that targets C with MPI. We then applied the tool to every send call in the NAS LU benchmark, as they all pack the data that is sent.

Implementation

The refactoring tool was implemented as a refactoring plugin on top of the Eclipse CDT IDE. The user marks the target send call as well as the packing code in the text editor and selects Convert Packing Code to Datatype from the refactoring menu. The tool then checks the preconditions before it generates, specializes and compresses the IR. The tool then presents a difference view showing the code changes going from packing code to datatype code and asks the user to accept or reject the changes. If the user accepts the changes then the tool rewrites the packing code to equivalent datatype code.

The tool is implemented in Java and consist of roughly 5000 SLOC. It operates on the Eclipse CDT AST and performs expression simplification using the Symja Open Source Library [27].
NAS LU

We applied the tool to the LU application from the NAS Parallel Benchmarks [57]. The LU application solves a block lower triangular-block upper triangular system of equations, resulting from an unfactored implicit finite-difference discretization of the Navier-Stokes equations in three dimensions. It does this by performing symmetric successive over-relaxation (SSOR).

LU contains 12 send calls, and each sends packed data. The data structures that are packed are 2- or 4-dimensional arrays. The packing code range from simple loops containing one packing statement, to two sets of double-nested loops containing five packing statements each. In two cases, data from two different arrays are packed and sent together.

In every case the send calls are directly proceeded by the packing code. However, in 6 cases the data is not packed contiguously into the packing buffer, thus breaking the third precondition. In all of these cases one loop split was sufficient to establish the precondition.

The LU benchmark is written in Fortran with MPI. Since our tool targets C with MPI we ported the LU communication kernels to C. The kernels were located in the files: exchange1.f, exchange3.f, exchange4.f, exchange5.f and exchange6.f. We then invoked the C communication kernels from LU so that we could run the application and verify the results, thus giving us confidence that our ports were correct.

4.5.2 Results

In every case the packing code could be specialized to hvectors, vector or contiguous datatypes.

In total 42 packing groups were specialized. 10 groups were specialized to hvectors, 13 groups were specialized to vectors and 19 groups were specialized to contiguous types. All the contiguous types were compressed into block length and send count arguments. Furthermore, two of the hvectors and four of the vectors were merged as their parent type was specialized from an hindexed to an hvectors type. This 8 hvectors and 9 vectors sufficient to replace the packing codes of the 12 send calls.

Note that the packing code of two send calls packed fully contiguous data, and could therefore be described without any datatypes by increasing the count arguments of the send calls.

The transformations took on average 32 ms, with the fastest transformation taking 6 ms and the slowest transformation taking 49 ms. Transformation time was correlated with the number of packing groups that had to be specialized and their complexity.
We manually inspected the packing code produced by our tool and confirmed that it could not be further improved.

4.6 Future Datatype Work

Numerous lines of research could be inspired by this work. We will first discuss future work that are natural extensions of this work. We will then suggest research projects that are related to datatypes, but that are not direct extensions of this work.

This work presents a general algorithm, that would fit as well into a compiler as a refactoring tool. A natural next step is therefore to implement our algorithm as a proof-of-concept in a compiler. We have provided an implementation in the form of a refactoring tool. As such, we could rely on the programmer to establish preconditions, clean up dead code, and ensure that datatypes are not re-constructed every time the communication code is invoked. However, a compiler can not rely as much on the user and should perform enabling transformations, such as loop split, code motion and inlining, and post-transformations such aggressive dead code elimination and lazy initialization code insertion.

The author’s experience and anecdotal evidence suggests that datatypes are hard to construct. This motivated an automated approach, even for an environment such as MPI that expose datatypes to the programmer. However, a user study to quantify the benefits of automation has not yet been conducted, and would provide a useful contribution to the datatype literature.

Datatype IR construction, and hence our algorithm, has four preconditions. Our evaluation suggests that these preconditions are not too strict for the tool to be applicable, and that they can easily be established by the programmer. However, future work could extend the algorithm to handle more cases, and thereby relax these preconditions. The first precondition can be relaxed to handle more complex packing and unpacking code as long as the unpacking does not depend on unpacked data. For complex packing code where some of the packing is decided by if statements the algorithm would only be able to specialize all or some of the datatypes to indexed types. That is, the precondition would be moved to one or more of the specialization stages. Furthermore, the third precondition can be relaxed to handle packing code that is packed into the packing buffer in arbitrary order. However, it would not necessarily be possible to specialize the datatypes generated from such code to hvectors, or to only vectorize sub-sets of them. Finally, the fourth precondition can be relaxed to allow packing code that does not directly proceed the send call. This would
require dataflow analysis to find the packing statements that produce values for the target send call, and may require additional code insertion if multiple sets of packing statements on different paths produce such values. Furthermore, determining whether a statement affects the packing buffer requires inter-procedural analysis and alias analysis, or conservative assumptions.

Our algorithm can handle packing code that packs data that is transferred through point-to-point communication calls, moved through RDMA operations, and written to files using an IO operations. However, additional work is required to handle code that packs data that is then transferred by certain collectives such as MPI's AllToAll. MPI_AllToAll takes a datatype and sends one of these data layouts to each rank that participate in the exchange. These data layouts must be laid out contiguously in memory, and the AllToAll therefore contains an implicit contiguous type. However, in certain cases, such as matrix transpose, it is necessary to send overlapping non-contiguous data to the different ranks. Since the current version of MPI_AllToAll implicitly exchanges contiguous layouts, the only way to handle this case that is known to the authors is to tweak the extent of the top datatype [44]. An extension to our algorithm would detect this case and convert the top hvector type resulting from this use case into a contiguous type by tweaking the extent of its subtype. Knowing that the contiguous type is implicit in the AllToAll, it could then be removed before code is emitted. Future work also include investigating whether extensions to our algorithm is needed for other collectives.

Our algorithm targets packing code and converts it to datatype code. Packing code is the most common way to transfer non-contiguous data if datatypes are not used or available. The reason for this is that packing code reduces the number of messages to one, thus minimizing latency. However, an alternative approach to transfer non-contiguous data is to send one message per non-contiguous block, perhaps using asynchronous pipelined messages. Future work could extend our algorithm to detect this pattern in user code, and convert the non-contiguous sends to a single send that uses datatypes to describe the layout. This would up free the runtime to use pipelined messages where this makes sense, to pack data where this is more efficient (e.g., on a system without support for asynchronous sends), or non-contiguous transfer support in hardware where this is available.

The work presented in this paper converts packing or, by extension, unpacking code to datatype code by only considering one side of the communication. This imposes a strict requirement on the ordering of the elements in the layout described by the datatype. That is, the datatype must describe a layout where every element are sent or received in the same order as they appeared in
the original packing buffer. However, if sends and receives could be matched then both sides of the communication could be changed simultaneously. This would free the algorithm from the ordering constraint and turn it into an optimization process. Datatypes element ordering could then be optimized both for our notion of datatype efficiency (least number of arguments), and for improved memory access patterns.

Datatypes are hard to construct or change, and the programmer has little help if the first attempt was incorrect. There are no effective way to debug datatypes, and the programmer is left guessing what might have gone wrong. A useful future work is therefore to build a datatype debugger. Such a debugger could print the memory locations that are sent. Moreover, with sufficient knowledge of the user data structure, it could even show the layout of the data described by the datatype textually or graphically. Information about the user data structure could either be automatically extracted from the program, if it is regular enough, or provided by the user. Consider code that sends the south face of a three dimensional mesh. A datatype debugger could present a 3D visualization showing the mesh with the face in question colored red.

Many current systems with MPI support do not provide hardware facilities for non-contiguous transfers. Furthermore, systems that do provide such capabilities often do not exploit them. This is likely because datatypes have traditionally not been much used by HPC programmers. Such systems instead pack the data inside the MPI library by interpreting the datatype. Such an interpreted solution often performs slightly worse than user provided C packing code that has been compiled to machine code. There are two promising ways to get around this. The first is to provide an MPI-aware offline compiler that can compile datatypes to optimized machine packing code. The second approach is to embed an online datatype compiler into the MPI_Type Commit function call. This approach is preferable as it is transparent to the programmer, easily deployable, and can provide increased performance without recompiling application code. The online compiler would be able to produce code that matches compiled user packing code, or that even outperforms it if it can take advantage of runtime information to produce more optimal machine code than is possible at compile time. Furthermore, the cost of compilation over the lifetime of an execution is unlikely to be a bottleneck, as datatypes are typically generated once and then re-used in every communication.
4.7 A Datatype Extraction Case Study

This section provides a case study of how our tool analyzes and converts one packing code block from the NAS LU application [57] to datatype code.

The exchange.3.f communication kernel in the NAS LU application contains four point-to-point send/receive pairs that perform border exchanges in the north-south and east-west directions. The borders are 2-dimensional faces of a 3-dimensional structured mesh. Each cell in the mesh has five values, so the resulting array is 4-dimensional.

Figure 4.6 shows the original Fortran code for the south border send. The loops pack two 2-dimensional faces into the packing buffer. The first face is packed into the buf packing buffer starting at location 1, while the second face is packed into the same buffer starting at location ny*nz. The first face is packed by the statements on line 60–64, while the second face is packed by the statements on line 65–69.

Figure 4.7 shows the code ported to C. The original Fortran code used Fortran’s convenient array syntax to pass a 4-dimensional array into the communication kernel. However, when this array is passed to C code it becomes a 1-dimensional pointer. In the ported code this pointer is accessed using the standard approach for treating 1-dimensional arrays as multi-dimensional arrays, which is to multiply each index with the inner dimensions. In practice a programmer might hide this indexing logic behind a macro or a function, in which case our tool would need to either expand the macro/in-line the function, or analyze the macro/function to figure out the index expression. However, macro expansion and function inlining are known techniques and are not covered here. Furthermore, since our tool is a refactoring tool we can rely on the programmer to perform these operations.

As each iteration of the loops packs into two different parts of the packing buffer, this code does not satisfy the third precondition from section 4.3.1. To establish this precondition the programmer must therefore perform one loop split to produce the code in figure 4.8.

Once the third precondition has been established our tool can apply our algorithm to the packing code to construct an equivalent datatype. Our tool first summarizes the packing groups in the code, shown in figure 4.9(a). Once the packing groups have been constructed the tool assigns a struct datatype to every packing loop and packing sequence. These are then specialized and compressed as described in section 4.4, yielding the datatypes in 4.9(b). The inner packing sequences are described with contiguous datatypes that are compressed into the block length of the parent datatype (vec.t).
The inner loop is specialized to a vector, while the outer loop can only be specialized to an hvector. Since both loops describe the same layout they can be described with the same datatypes (vec_t and hvec1_t). Finally, the hvector hvec_t, with a count of 2, describes the packing sequence containing the two loops.

The datatypes in the IR are used by our tool to emit the code in figure 4.10. Note that the tool does not currently perform dead-code elimination of the packing loops (lines 51–70 in figure 4.8), but for presentation purposes we have manually removed these. For the same reasons, we also added newlines between the construction of each datatype, although the tool does not do this for technical reasons.

One issue with the code in figure 4.10 is that it constructs a new datatype every time the south border is sent. This is costly and unnecessary as the datatype can often be reused. Since our tool is a refactoring tool we can leave to the user the task of ensuring the datatypes are not unnecessarily re-created. Such tasks are typically easy for users. However, a compiler that implements our algorithm can not rely on the user to modify its output, and should therefore address this problem. We therefore present two solutions to this problem. Both solutions employ lazy initialization to prevent unnecessary re-construction of datatypes.

Figure 4.11 shows the first solution. The hvec_t datatype is added to the static code segment through the use of the static keyword. Furthermore, a static decision variable is used to only construct the datatype the first time the code is executed. Note that this solution requires that the variables that are used to construct the datatype, ny, nz, isiz1 and isiz2, are not changed before any subsequent execution of the communication code. Using this solution the user, or an automated tool that insert this check, would have to verify that this is the case. This is usually easy for a user, who knows a lot about the usage patterns of her variables. However, it would be hard for a tool to determine this, as it would require whole program analysis.

The second solution requires no additional analysis. This is possible by adding additional code to dynamically check at runtime that none of these variables have been changed between executions of the communication code. Figure 4.12 shows these checks for the south border send example. The value of the variables that are used to construct the datatype are stored in static variables when the datatype is constructed. If the communication code is executed again with different values for these variables, then the datatype is re-constructed. Adding this scaffolding is trivial for a tool, and we plan to add this feature as an option in our refactoring tool.
do k = 1, nz
  do j = 1, ny
    ipos1 = (k−1)*ny + j
    ipos2 = ipos1 + ny* nz
    buf(1, ipos1) = g(1, nx−1, j, k)
    buf(2, ipos1) = g(2, nx−1, j, k)
    buf(3, ipos1) = g(3, nx−1, j, k)
    buf(4, ipos1) = g(4, nx−1, j, k)
    buf(5, ipos1) = g(5, nx−1, j, k)
    buf(1, ipos2) = g(1, nx, j, k)
    buf(2, ipos2) = g(2, nx, j, k)
    buf(3, ipos2) = g(3, nx, j, k)
    buf(4, ipos2) = g(4, nx, j, k)
    buf(5, ipos2) = g(5, nx, j, k)
  end do
end do

MPI_Send( buf, 10*ny*nz, MPI_DOUBLE, south, from_n, MPI_COMM_WORLD, IERROR )

Figure 4.6: South border exchange packing code case study (Fortran)

for (k = 1; k <= nz; k++) {
  for (j = 1; j <= ny; j++) {
    ipos1 = (k−1)*ny + j
    ipos2 = ipos1 + ny*nz
    buf[(ipos1−1)+5] = g[(k−1)*(i siz2+4)*(i siz1+4)*5 + (j+1)*(i siz1+4)*5 + (nx)*5 ];
    buf[(ipos1−1)+5 + 1] = g[(k−1)*(i siz2+4)*(i siz1+4)*5 + (j+1)*(i siz1+4)*5 + (nx)*5 + 1];
    buf[(ipos1−1)+5 + 2] = g[(k−1)*(i siz2+4)*(i siz1+4)*5 + (j+1)*(i siz1+4)*5 + (nx)*5 + 2];
    buf[(ipos1−1)+5 + 3] = g[(k−1)*(i siz2+4)*(i siz1+4)*5 + (j+1)*(i siz1+4)*5 + (nx)*5 + 3];
    buf[(ipos1−1)+5 + 4] = g[(k−1)*(i siz2+4)*(i siz1+4)*5 + (j+1)*(i siz1+4)*5 + (nx)*5 + 4];
    buf[(ipos2−1)+5] = g[(k−1)*(i siz2+4)*(i siz1+4)*5 + (j+1)*(i siz1+4)*5 + (nx+1)*5 ];
    buf[(ipos2−1)+5 + 2] = g[(k−1)*(i siz2+4)*(i siz1+4)*5 + (j+1)*(i siz1+4)*5 + (nx+1)*5 + 2];
    buf[(ipos2−1)+5 + 3] = g[(k−1)*(i siz2+4)*(i siz1+4)*5 + (j+1)*(i siz1+4)*5 + (nx+1)*5 + 3];
    buf[(ipos2−1)+5 + 4] = g[(k−1)*(i siz2+4)*(i siz1+4)*5 + (j+1)*(i siz1+4)*5 + (nx+1)*5 + 4];
  }
}

MPI_Send( buf, 10*ny*nz, MPI_DOUBLE, south, from_n, MPI_COMM_WORLD);
for (k = 1; k <= nz; k++) {
    for (j = 1; j <= ny; j++) {
        ipos1 = (k-1)*ny + j;
        buf[(ipos1-1)*5] = g[(k-1)*(isz1+4)*isz1+4]*isz1+4 + (j+1)*isz1+4 + nx*5 + (nx+1)*5 + j*1;
    }
}

for (j = 1; j <= ny; j++) {
    ipos2 = (k-1)*ny + j + ny*nz;
    buf[(ipos2-1)*5] = g[(k-1)*(isz2+4)*isz2+4]*isz2+4 + (j+1)*isz2+4 + nx*5 + (nx+1)*5 + j*1;
}

MPI_Send (buf, 10+ny*nz, MPI_DOUBLE, south, from_n, MPI_COMM_WORLD);

Figure 4.8: Case study after loop split

Figure 4.9: Case study IR

Figure 4.10: Case study after tool conversion to datatypes, and user dead code elimination of the packing loops
static MPI_Datatype hvec_t;
static int init = 0;
if (!init) {
    MPI_Datatype vec_t;
    MPI_Type_vector(ny, 5, (isz1+4)*5, MPI_DOUBLE, &vec_t);
    MPI_Datatype hvec1_t;
    MPI_Type_create_hvector(nz, 1, (isz2+4)*(isz1+4)*5 + sizeof(double), vec_t, &hvec1_t);
    MPI_Type_create_hvector(2, 1, 5 + sizeof(double), hvec1_t, &hvec_t);
    MPI_Type_commit(&hvec_t);
}
MPI_Send(&g[10*(isz1+4) + (nx)*5], 1, hvec_t, south, from_n, MPI_COMM_WORLD);

Figure 4.11: Case Study with the first lazy initialization strategy (assumes ny, nz, isiz1 and isiz2 are not redefined between executions)

static MPI_Datatype hvec_t;
static int init = 0, ny_v, isiz1_v, ns_v, isiz2_v;
if (!init || ny != ny_v || isiz1 != isiz1_v || ns != ns_v || isiz2 != isiz2_v) {
    init = 1;
    ny_v = ny;
    isiz1_v = isiz1;
    ns_v = ns;
    isiz2_v = isiz2;
    MPI_Datatype vec_t;
    MPI_Type_vector(ny, 5, (isz1+4)*5, MPI_DOUBLE, &vec_t);
    MPI_Datatype hvec1_t;
    MPI_Type_create_hvector(nz, 1, (isz2+4)*(isz1+4)*5 + sizeof(double), vec_t, &hvec1_t);
    MPI_Type_create_hvector(2, 1, 5 + sizeof(double), hvec1_t, &hvec_t);
    MPI_Type_commit(&hvec_t);
}
MPI_Send(&g[10*(isz1+4) + (nx)*5], 1, hvec_t, south, from_n, MPI_COMM_WORLD);

Figure 4.12: Case Study with the second lazy initialization strategy
5.1 Introduction

In the last ten years great strides have been made in increasing the productivity of the desktop application programmer. Powerful IDEs such as Eclipse, Microsoft Visual Studio, NetBeans, and IntelliJ IDEA have become the norm and programmers expect automated support for code refactoring.

Before refactoring tools appeared, programmers often over-designed, because it was expensive and error-prone to change the design of large systems once they were implemented. Refactoring tools allow programmers to continuously explore the design space of multi-million line codebases, without the fear of introducing unintended behavioral changes. Modern IDEs incorporate refactoring in their top menu, and often compete on the basis of refactoring support.

So far, automated refactoring support in mainstream IDEs has mainly been geared towards improving code design and readability. However, refactoring is the process of changing source code without changing its semantics, in order to improve some non-functional aspect. Aside from readability and maintainability, there are many such aspects that can benefit from refactoring technology.

Recently we have used refactoring to retrofit sequential Java applications to use parallelism for performance [3, 4, 31]. The goal is to decrease the running time of an application, or to increase the amount of data that can be processed in the same amount of time, without changing the functionality. It is therefore a good fit for refactoring technology.

The HPC (High Performance Computing) programmer is, of course, concerned with getting the most performance he can from the available hardware. However, there are other concerns that are equally, and often more, important and that conflict with this goal. Examples include numerical stability, scalability, maintainability, and portability [58]. Codes must be correct and they typically
have a life-span of decades. The latter implies that they must be maintained for a long time and that they will run on machines that are wildly different, and that may have orders of magnitudes more processor cores than the machine they were originally designed for.

The HPC programmer must carefully balance all of these concerns and has very little effective development tool support to do so. While the domain of application programming is ripe with powerful IDEs, HPC programming typically involves a programmer with VIM, or an X-forwarded text browser, and a command line shell. While HPC programmers have created many great codes using these trusted tools, we believe it is time to bring their tool support into the 21st Century through a powerful IDE with support for the kinds of refactorings that they need.

One promising effort at providing an IDE for parallel and HPC programmers is the Eclipse PTP project [59]. PTP is a set of plugins that extend Eclipse with tools to develop parallel software for remote machines and clusters. It simulates a local development environment even though the code may be run on, and even located at, a remote cluster. Furthermore, it provides an integrated environment in which to interact with the plethora of debuggers, profilers, and job launchers that exist for these systems.

PTP is built on top of CDT, which is a set of Eclipse plugins for C and C++ development. It therefore provides some refactorings such as Rename and Extract Method. However, PTP does not provide the other kinds of refactorings we believe HPC programmers need in order to achieve higher productivity in the face of increasing complexity and a plethora of parallel programming models.

5.2 Towards An HPC Refactoring Catalog

In this section we begin a catalog of refactorings for the HPC programmer. The catalogue is incomplete and we expect it to grow and mature over time. Some of the refactorings in the catalog introduce parallelism, others prepare the code for parallelism, while others still improve the performance of already parallel code.

As the number of cores per processor continues to increase, more cores are contending for the same shared memory resources. It is therefore increasingly important to manage memory resources so that they do not become bottlenecks that limit an application’s performance and prevent it from scaling. Consider for example an application where threads frequently update memory on the same cache line. The threads have to get ownership of this cache line before writing to it, thus forcing
them to write to it one at a time.

Furthermore, a parallel application consists of several units of execution that sequentially manipulate a chunk of data or execute a flow of control. As such, a parallel application can be viewed as an array of cooperating sequential components. If the sequential components suffer from poor performance then the parallel application will too. Also, due to the effect demonstrated by Amdahl’s law, the inherently sequential parts of an application will dominate when the number of parallel execution units becomes sufficiently large. The HPC programmer must therefore not only extract enough parallelism to keep available cores busy, but must also ensure that the sequential parts perform adequately.

We therefore also include refactorings that are intended to help the HPC programmer speed up the sequential components of the parallel application. Since structured programming is dominant in the HPC community [58], the sequential refactorings we propose are for the C language.

5.2.1 Parallel MPI Refactorings

Make Communication Asynchronous This refactoring replaces a synchronous MPI_Send or MPI_Recv with an asynchronous MPI_Isend or MPI_Irecv and an MPI_Wait. It then attempts to move the MPI_Wait as far apart from the send as possible with the constraint that it is not moved past code that writes to memory areas that are being sent, or reads from memory areas that are being received. This allows useful computation to be performed while messages are sent, which decreases running time and makes the application less sensitive to communication jitter.

Split Computation Into Communication Dependent and Independent Parts This refactoring finds the subset of the iterations of a loop that compute results that are subsequently sent to another process, and peals these off from the rest of the loop. This enables the programmer to overlap communication with computation by asynchronously transferring communication-dependent results while the rest are computed. Many iterative algorithms follow the pattern of repeatedly updating chunks of a data structure on different nodes and then having each node exchange the borders of its chunk with those of its neighbors. Iterative structured grid computations are examples of this, where an n-dimensional grid is split into chunks and divided amongst the compute nodes. Since only the borders of the chunks are exchanged between each iteration it is useful to compute this part first so that the exchange can be overlapped with the computation of the interior [53].
**Remove Superfluous Barriers**  This refactoring analyzes an application and attempts to remove superfluous barriers. That is, if it can detect that other MPI functions provide sufficient synchronization then it can remove redundant barriers.

**Replace Barrier with Local Synchronization**  This refactoring attempts to replace an MPI Barrier with synchronous calls to the `MPI_Ssend` and `MPI_Recv` functions, so that only some processes are synchronized in cases where this is safe. This can be useful to relax synchronization in MPI programs where a barrier is used to force all processes to synchronize, when in reality only some of the processes need to wait for each other.

**Gather Data**  This refactoring helps the programmer create communication code to retrieve the whole or parts of a shared data structure maintained on a master process. This can be useful when porting a sequential or shared-memory parallel program that uses a shared data structure to MPI. As a first step it is reasonable to maintain this data structure on a master process, before considering whether to distribute it between the processes.

**Create MPI Datatype from Struct**  This refactoring creates an MPI Datatype for a C struct or an array of structs. Previous work on marshaling data structures as MPI Types include Tansey and Tilevich’s MPI Serializer [18].

**Extract Datatype**  This refactoring replaces a flattening loop with the construction of an MPI Datatype that describes the mapping between the local data structures and a message. A flattening loop is a loop that copies the parts of the local data structures that should be sent to another process into a buffer. This buffer is then sent using a simple MPI datatype. Flattening loops are very common in MPI applications and replacing them with the use of MPI datatypes yields code that is simpler to understand, and faster if a copy pass can be avoided [44].

5.2.2  Sequential C Refactorings

**Restrict Pointer**  The ISO C99 standard added support for a new type qualifier, called *restrict*, that “[…] allows programs to be written so that translators can produce significantly faster executables” [60]. Intuitively, *restrict*, when applied to a pointer $p$, specifies that the object pointed to by $p$ will not be accessed through another pointer in a block where it is updated through $p$. The
Restrict Pointer refactoring performs a whole-program analysis of the source code and restricts a given pointer if it is safe. If there are dependencies preventing the pointer from being restricted the refactoring provides information about the dependency to the programmer so that he can break it.

**Fuse Loops**  Loop Fusion is a technique where two or more loops are fused into one loop. This is useful in some cases because it can give the compiler and the hardware a longer sequence of instructions without branches, which can improve their ability to schedule code. The Fuse Loops refactoring fuses two or more loops if this is safe.

**Split Struct into Hot and Cold Fields**  It is common for some fields in a struct to be accessed more often than others. An example is a linked data structure where the key and next fields are frequently accessed when searching for a key, while the value fields are only accessed when the key is found [61]. Furthermore, some data structures such as heaps and k-d trees are often stored in an array. This refactoring splits a struct in two, where one part contains the fields that are accessed often and the other contains the fields that are accessed less often. The structs are then stored in different arrays so that the hot array can be traversed quickly in cache.

**Organize Block as Load/Compute/Store**  This refactoring organizes a code block so that all input data are read into local variables at the beginning, and the results stored back at the end. This can improve the compiler’s ability to analyze dependencies, optimize code and schedule code and can be useful in kernels that need heavy optimization.

**Split Loop**  This refactoring finds independent code blocks in a loop and lets the programmer decide which code blocks to split of into new loops. This serves as an enabling step when the programmer wants to incrementally parallelize a large loop using OpenMP or CUDA. It is also helpful when he wants to split of parts of a loop that does not contain loop carried dependencies and that can therefore be parallelized.

**Unroll Loop**  This refactoring checks whether it is safe to unroll a loop and then unrolls (or re-rolls) it by a given factor. Unrolling can be useful to reduce branch overhead and to improve the compiler’s ability to schedule code, but excessive unrolling can end up causing ICache misses (and even hurt scheduling). This refactoring allows the programmer to quickly test different unroll degrees to find the one that yields the best performance. See Section 5.3 for a discussion on how to
maintain readability when expressing performance transformations like Unroll Loop in the source code.

**Block Loop** This refactoring establishes whether it is safe to convert a linear loop to a blocked/tiled loop and then, with input from the programmer, performs the transformations. This can drastically improve the cache hit ratio since the program only operates on a small block of data at a time.

### 5.3 Annotations and Views

We realize that some of the loop-related refactorings in the previous section, such as Unroll Loop and Block Loop, are traditionally performed by optimizing compilers. However, the compilers knowledge about the performance of the final code is very limited, due to its static nature and the complexity of modern hardware. The former means that it can not know which paths the program will take at runtime. The latter means that even if it could, it would need a full model of the hardware to predict the performance. It is therefore forced to make educated guesses and it often guesses wrong.

The programmer, on the other hand, does not need to guess. He can use a profiler, or manually instrument the code with timers, and then explore the space of optimizations effectively using his intuition. For this reason, performance programmers still manually implement these optimizations when they need near-optimal performance. The refactorings in the previous section help them do this safely.

Furthermore, compilers often cannot perform loop transformations due to dependencies that break transformation preconditions. Sometimes these dependencies are real, but in many cases they are artifacts of the conservative nature of compiler analysis. In fact, compilers are forced to be overly conservative, due to their static nature, the NP-Completeness of the dependency testing problem, and the tradeoff between compile time and powerful pointer analysis.

With the programmer in the loop more transformations are possible, as demonstrated by the Parascope project [62, 63]. The programmer can tell the refactoring engine to ignore dependencies he knows to be incorrect, or rewrite the code once the refactoring environment has pointed them out.

However, it would be better if the performance programmer had the best of both worlds. He keeps his source code readable and can take control of loop optimization when needed. The programmer
decides when he should be in the driver seat and has the tools to help him drive fast and safe.

To aid readability, we therefore propose that refactoring engines insert source annotations instead of transforming loops directly. The refactoring engines will also analyze the code to determine that transformations are safe. By keeping the source code free of performance constructs that are unrelated to the algorithm the programmer wants to express, he can more easily focus on solving the problem at hand.

To further tune performance, the programmer can expand the annotations in a performance view to understand the performance characteristics of the code, and use this understanding to improve the annotations.

The expanded source code lets the programmer reason about the performance of the code, while the original source code allows him to reason about its meaning. For example, when profiling he would want to use the performance view, while the normal view would be more suitable for debugging. Finally, the IDE tracks the annotations and indicates an error if the programmer later changes the source in a way that is inconsistent with the annotation preconditions.

When the programmer wants to build a project, the IDE first passes the code to an internal source-to-source annotation compiler. This compiler expands annotations before it passes the code to the target compiler. The annotation compiler is also available as a command line application for programmers who need to build the project outside of the IDE.

The annotations are in the form of pragma directives that are ignored by C preprocessors. Therefore, the program will still work without the annotation compiler.

5.4 A Framework for Advanced Refactoring

Users have come to expect certain features from a modern refactoring framework. This includes invoking refactorings directly from the editor, previewing changes before they are applied, and the ability to undo and redo changes. Eclipse provides a rich infrastructure that has these features, in addition to an AST program representation, visitors, and rewrite capability. CDT and PTP extend Eclipse with support for C, C++ and MPI. Moreover, PTP provides numerous other useful features for HPC programmers. We therefore believe the best option is to base the performance refactoring infrastructure we envision on Eclipse with CDT and PTP.

However, the refactorings we discussed in the previous section require more sophisticated analysis than traditional refactorings. It is therefore necessary to augment the Eclipse infrastructure with
Table 5.1: Candidates for a Refactoring Analysis Backend

<table>
<thead>
<tr>
<th>Project</th>
<th>Target Lang.</th>
<th>Impl. Lang.</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cetus</td>
<td>C</td>
<td>Java</td>
<td>AST, CFG, Callgraph, Inter-Procedural Pointer Analysis (Steensgaard), Loop Dependence Analysis</td>
</tr>
<tr>
<td>clang/LLVM</td>
<td>C, C++, Objective-C</td>
<td>C++</td>
<td>AST, CFG, Incremental Compilation, SSA IR, Callgraph, Inter-Procedural Pointer Analysis (DSA, Anderson, Steensgaard), Loop Dependence Analysis</td>
</tr>
<tr>
<td>Elsa/Pork/Dehydra/Treehydra</td>
<td>C, C++</td>
<td>C</td>
<td>AST, Scriptable AST Analysis</td>
</tr>
<tr>
<td>LLNL Rose</td>
<td>C, C++, Fortran, OpenMP, UPC</td>
<td>C++</td>
<td>AST, CFG, Callgraph, Inter-Procedural Pointer Analysis (Steensgaard), Loop Dependence Analysis</td>
</tr>
</tbody>
</table>

a powerful analysis framework. To demonstrate this we discuss two refactorings and the analyses that are necessary to implement them.

Consider the Restrict Pointer refactoring. To safely add the restrict qualifier to a pointer the refactoring must verify that no other pointers accessed in the same block refer to any of the memory areas that are modified through the restricted pointer. This requires inter-procedural alias analysis to check for aliases and array disambiguation to find out if array accesses through pointers refer to the same parts of an array.

Now consider the Split Computation Into Communication Dependent and Independent Parts refactoring. It requires dataflow analysis, array disambiguation and possibly shape analysis to find out which parts of a data structure are sent to other processes. It must also do loop dependence analysis and pointer analysis to find if there are loop carried dependencies that prevent it from peeling of loop iterations.

Table 5.1 contains four analysis framework candidates that we surveyed. Given the need for a mature framework with powerful analysis data structures we believe LLNL Rose [64] and clang/LLVM [65] are the most promising candidates. In order to investigate their suitability we have created an Eclipse plugin that allows refactoring plugins to access the Rose compiler APIs. We have also started to implement the Restrict Pointer refactoring on top of this plugin. When this has been completed, we are planning to do the same using clang/LLVM.
CHAPTER 6

CONCLUSION

Not long ago, refactoring to improve the design of code was impractical and only done by a select group of hero programmers. Refactoring tools empowered the average programmer to explore the design space like a pro. A similar situation exists today for parallel refactoring. However, these require more in-depth analysis than design refactoring, which means that they can be even more useful to automate. In this decade, refactorings to simplify parallel applications and to improve their performance can become as transformative as design refactorings was in the previous decade.

This thesis demonstrate the feasibility of automated refactorings to increase the maintainability, scalability and efficiency of parallel applications, by presenting the algorithms to automate two such transformations for very different domains. Furthermore, it presents the beginnings of a refactoring catalogue for high performance computing.

The first of the two automated refactorings addresses the need for immutable classes in irregular shared-memory parallel applications. Programmers use immutability to simplify both sequential and parallel applications, and to reduce the need for synchronization in shared-memory parallel applications. Although some classes are designed from the beginning to be immutable, other classes are retrofitted with immutability. Transforming mutable to immutable classes is tedious and error-prone. Our tool, Immutator, automates the analysis and transformations required to make a class immutable. Experiments and case studies of manual transformations, as well as running Immutator on 346 open-source classes, show that Immutator is useful. It is applicable in more than 33% of the studied classes. It is safer than manual transformations which introduced between 2 and 6 errors/class. It can save the programmer significant work (analyzing 57 methods and editing 45 lines) and time (27 minutes) per transformed class.

The second transformation provides a solution to the difficulty of constructing datatypes for regular distributed memory applications. The upwards scalability limitations of shared memory programming will cause more systems with distributed memory to be deployed in the future.
The increasing relative cost of communication will cause hardware vendors to move towards more advanced network subsystems. The prevalence of non-contiguous transfers, especially in scientific applications, will make hardware capable of non-contiguous zero-copy transfers common. Such features can only be used if datatypes are provided. Datatypes can simplify distributed applications and can, with the right hardware, drastically reduce the cost of non-contiguous communication over that of packing code. We have presented an algorithm that automates the process of converting packing code to datatype code. We have implemented this algorithm as a refactoring tool and evaluated it by transforming the packing code in the NAS LU application. The evaluation shows that the algorithm is applicable, and that it generates good datatypes.

These transformations demonstrate that the feasibility of refactoring for maintainable, scalable and efficient parallelism. In the future we hope that many more such refactoring transformations will be developed to improve the productivity of the parallel programmer.
REFERENCES


[38] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1994.


