NORMALIZING-REFACTORINGS: SIMPLIFYING THE CONSTRUCTION OF SOURCE CODE TRANSFORMATIONS

A thesis submitted
to Kent State University in partial fulfillment of the requirements for the degree of Master of Science

by

Christian Newman

Fall 2013
Thesis written by

Christian Newman

B.S., Kent State University, USA 2010

Approved by

___________________________________, Advisor

___________________________________, Chair, Department of Computer Science

___________________________________, Dean, College of Arts and Sciences
# TABLE OF CONTENTS

NORMALIZING-REFACTORINGS: SIMPLIFYING THE CONSTRUCTION OF SOURCE CODE TRANSFORMATIONS ......................................................... I

TABLE OF CONTENTS .................................................................................................................................................. III

LIST OF FIGURES ..................................................................................................................................................... V

LIST OF TABLES ....................................................................................................................................................... VI

ACKNOWLEDGEMENTS ........................................................................................................................................ VII

CHAPTER 1 INTRODUCTION ......................................................................................................................... 8

1.1 Problem Statement ............................................................................................................................................. 10

1.2 Research Contributions ................................................................................................................................. 11

1.3 Organization of the Thesis ............................................................................................................................. 12

CHAPTER 2 INFRASTRUCTURE ...................................................................................................................... 13

CHAPTER 3 MOTIVATIONAL EXAMPLE ..................................................................................................... 16

3.1 Change to Operator `new` .......................................................................................................................... 16

3.2 Breadth of Transformation .......................................................................................................................... 20

CHAPTER 4 SIMPLIFYING TRANSFORMATIONS ....................................................................................... 23

4.1 Syntactic Variation in Expressions ............................................................................................................. 23

4.2 Normalizing-Refactorings .......................................................................................................................... 26

CHAPTER 5 EVALUATION & IMPLEMENTATION .................................................................................... 39

5.1 Implementation .............................................................................................................................................. 39

5.2 Evaluation ..................................................................................................................................................... 41
CHAPTER 6 RELATED WORK .............................................................................................................. 44

CHAPTER 7 CONCLUSIONS AND FUTURE WORK .................................................................... 47

7.1 Future Work ................................................................................................................................. 48

CHAPTER 8 REFERENCES .................................................................................................................. 49
LIST OF FIGURES

Figure 1. Source code example................................................................. 13

Figure 2. Source code from Figure 1 marked up in srcML with all original text preserved.
................................................................................................................. 14

Figure 3. A simplified view of the XSLT used for transformation of expression

statements due to change in behavior of operator new after normalization refactoring.

The full version includes the use of extension functions to match the indentation of

the original. Note that only a single transformation needed to be written, in contrast
to the 10 needed previously due to all the different syntactic constructs. ............... 17

Figure 4. Grammar for Normalizing-Refactorings ........................................ 29

Figure 5. Example of srcML toolset workflow............................................. 40
LIST OF TABLES

Table 1. Results for the second case study ................................................................. 21
Table 2. Generalized technique for normalizing Compound Expressions ................. 30
Table 3. Generalized technique for normalizing Expression Sets ................................. 32
Table 4. Template Metafunction Normalizing Examples ............................................. 33
Table 5. List of Normalizing-Refactorings for Compound Expressions ...................... 36
Table 6. List of Normalizing-Refactorings for Compound Expressions (cont.) ............... 37
Table 7. List of Normalizing-Refactorings for Expression Sets ..................................... 38
Table 8. Total # of refactorings by category ................................................................. 39
ACKNOWLEDGEMENTS

The work I've done here is a result of years of encouragement and support from my family and friends. Without them, none of this would have been possible. I'd like to thank all of them for their support and guidance throughout my life, particularly my mother and father; Phyllis and Dale Newman, and my brother and sister; Jordan and Bethany Newman. You are all a blessing.

Those whom I've worked with in SDML also hold my highest gratitude. Through the projects we've done together as well as time spent talking in both matters serious and for entertainment, it has been a pleasure working with all of you. Your input in this work and help in seeing it through has been of the utmost importance.

I'd like to thank my advisor, Dr. Jonathan I. Maletic for the enormity of effort he's put into making me the student and person that I am today. The skills, knowledge, and experience he's passed on are all priceless and if it were not for him I may not be pursuing my current goals.
CHAPTER 1

Introduction

Many industrial software products have very long life-cycles. These systems represent substantial investments and are valuable assets that support the business of the organization they’re used by. Not only do these systems change as new features are added, but they must also undergo modification to address changes to dependent platforms, APIs, frameworks, and libraries. These types of changes are generally termed adaptive maintenance tasks. These tasks involve changing the software to respond to changes in its environment or platform. Here we are particularly focused on adaptive maintenance tasks.

Adaptive maintenance changes are unique as they are typically outside of an organization’s control. Hardware component obsolescence, changes to underlying operating systems, and new compiler or platform versions are all examples of unexpected changes that occur regularly. As these changes are oftentimes unexpected, organizations are forced to react to these adaptive changes as they are discovered. This has a negative impact on the schedule and cost for projects. Moreover, the changes are primarily repetitive in nature and can be highly fault prone when done manually by a developer.

When an organization is faced with these changes, they must either adapt the system or, if possible, postpone the changes until a later time. While postponement is quite common, especially in operating systems and platforms, there are many practical
reasons to make the investment. These include access to new features and technology that may be included in the new version as well as customer or market requirements. It may also prevent customers from having to perform invasive workarounds. Lastly, vendors eventually stop supporting the old versions and organizations are forced to address the problem.

Ideally, changes to a critical part of a piece of software’s environment will be clearly documented in the change log or release notes. Unfortunately, there is rarely enough detail to clearly direct a developer to the needed changes. Due to this, many developers identify adaptive maintenance problems by experimentation. Once the changes necessary to address the problems are identified, developers can be assigned to manually identify the errors and make corrections in the source code.

Attempts to automate part of this process have been proposed. A transformational approach to automate adaptive maintenance changes using the srcML infrastructure has been developed in [Collard, Maletic, Robinson 2010]. In this work, given the identified adaptive maintenance tasks transformations are developed that can automatically modify and correct the code to support the adaptation. Identifying where in the code these transformation need to be applied can also be automated to a large extent. Unfortunately, constructing the transformations for all the various syntactic situations can be quite complex and in itself error prone.
1.1 Problem Statement

This thesis is focused on addressing the problem of constructing transformations that address all syntactic variations. Specifically, a means of reducing the number of different transformations necessary is presented. The basic idea is relatively simple. I propose to develop a set of refactorings (program transformations that do not modify the behavior of a program) that will convert a set of different syntactic variations in a programming language into one standard version. This leverages the fact that there are often many different ways to express a given statement in a particular programming language. In this thesis I am limiting the discussion to C++, however in practice the same approach can be applied to any programming language. For example, there are many ways to increment an integer variable by one in C++ as seen below:

```cpp
i = i + 1;
i++;
++i;
i += 1;
i = 1 + i;
```

While this is a simple example, one would need to construct one complicated transformation to deal with these variations or build a set of transformation to address each individually. The approach taken here is to convert all these variations into one standard form (e.g., \( i = i + 1; \)) and then build one transformation that works only on that variation. This greatly simplifies the construction of transformations to support
adaptive maintenance tasks and also has the added value of simplifying the testing and verification of such transformation.

The objective of this thesis is to determine if a standard set of forms (*normal forms*) can be identified for C++. These normal forms have the exact same behavior and semantics as the original code. Moreover, I propose to discover a set of refactorings that will convert a given segment of source code into its normalized form.

### 1.2 Research Contributions

A set of refactorings to normalize semantically identical expressions in source code to a single syntax is presented. This is the first time a comprehensive set of refactorings for the purpose of simplifying transformations has been presented in literature. Transformational approaches have traditionally pushed the issue of dealing with syntactic variations in expressions into the need to write multiple transformations (that is, the user must handle them). This work will greatly simplify the manual labor needed by their developers to construct source-code transformations, thereby making it easier for Program Transformation as a technique for adaptive maintenance to be adopted in both research and industry. It’s also worth noting that the application of general program transformation done outside of adaptive maintenance may also be eased by the technique presented in this work.
1.3 Organization of the Thesis

This work is organized as follows: CHAPTER 2 provides an overview of the infrastructure on top of which the Normalizing-Refactorings are currently implemented. CHAPTER 3 motivates the work by giving examples of problems solved in industry using Program Transformation. It talks about how prevalent the problem of isomorphisms is and how difficult the problem is to solve using traditional methods. CHAPTER 4 describes the technique of Normalizing-Refactorings and how it is applied to source code. CHAPTER 5 talks about the evaluation of the work. CHAPTER 6 presents related work. CHAPTER 7 talks about conclusions and future work.
CHAPTER 2

Infrastructure

The transformational approach presented in this thesis is built to work on top of the srcML format and toolkit. srcML (Source-Code Markup Language)[Collard, Maletic 2004] is an XML format used to augment code with syntactic information from the AST. It preserves all text from the source code which includes comments, preprocessing information, and formatting. The primary focus of srcML is to construct an intermediate representation of source code using XML instead of traditional source code documents or data representations. srcML emphasizes a programmer-centric rather than compiler-centric view of the source code.

The format is supported with a toolkit containing two tools: src2srcml and srcml2src with the former converting to srcML and the latter converting from srcML. srcML has been used for lightweight fact extraction[Collard, Kagdi, Maletic 2003] [Collard, Decker, Maletic 2011], source code transformation[Collard, Maletic, Robinson

```
#include "rotate.h"
// rotate three standards
void rotate(int& n1, int& n2, int& n3) {
    // copy original values
    int tn1 = n1, tn2 = n2, tn3 = n3;
    // move
    n1 = tn3;
    n2 = tn1;
    n3 = tn2;
}
```

Figure 1. Source code example.
2010], and pattern matching[Dragan, Collard, Maletic 2006]. Below I’ll list some of the features of srcML to give a better idea of what it is and what it’s used for.

- srcML gives a complete view of the code. That is, all preprocessor statements, template definitions, and more are preserved. As an example, Figure 1 gives a small C++ function and Figure 2 shows the srcML markup for that function. From this example, it’s easy to see that the aforementioned characteristic holds true.

- srcML is robust. Code does not need to compile for srcML to parse it and create a srcML archive. If a source code doesn’t compile but we want to run srcML on it, we merely use src2srcml. The quality of the markup may not be great since the code is structured poorly (it doesn’t compile after all) but there will be no loss of the original text.

```
<cpp:include>
  #<cpp:directive>include</cpp:directive> <cpp:file>"rotate.h"</cpp:file>
</cpp:include>

<comment type="line">  // rotate three values</comment>
<function type="void" name="rotate"> <formal-params>(<param type="int"&amp;" name="n1"/>,
<param type="int"&amp;" name="n2"/>,
<param type="int"&amp;" name="n3"/>)</formal-params>
<block>
  <comment type="line">  // copy original values</comment>
  <decl-stmt>
    <decl>
      <type>int</type> <name>tn1</name> = <name>n1</name>,
      <name>tn2</name> = <name>n2</name>,
      <name>tn3</name> = <name>n3</name>
    </decl>
  </decl-stmt>
  <comment type="line">  // move</comment>
  <expr-stmt>
    <expr><name>n1</name> = <name>tn3</name></expr>;
    <expr><name>n2</name> = <name>tn1</name></expr>;
    <expr><name>n3</name> = <name>tn2</name></expr>;
  </expr-stmt>
</block>
</function>
```

**Figure 2.** Source code from Figure 1 marked up in srcML with all original text preserved.
• The tools that implement the format are very efficient. They translate at a speed of about 25KLOC per second and can handle about 3,000 files per minute. One example cited in their paper is the Linux kernel, which srcML can translate in less than seven minutes. Going from srcml to source code is even faster—250 KLOC per second.

• The srcML translator is not based on the parsers of any particular compiler—it has its own parser and only needs to understand enough about the code to insert proper markup. This means it can allow syntax that compilers may not be able to.

• Since srcML is an XML representation, it can take advantage of current and future XML tools for transformation, validation, and other activities. This means it works with tools like XSLT, XQUERY, DOM, SAX, etc.

The representation and the toolkit are used to support fact extraction, querying, transformation, and validation of source code. Here, I use the srcML format as the intermediate representation of source code to support implementation of the refactorings/transformations that I’ve developed.
CHAPTER 3

Motivational Example

The need for a set of Normalizing-Refactorings for support in automating software maintenance tasks is motivated via a transformational approach presented in [Collard, Maletic, Robinson 2010]. While they were addressing an adaptive maintenance problem using a transformational solution, a large number of syntactic variations to cover each adaptive change were required in their solution. These variations were not obvious and uncovered by trial and error.

Their work focuses on two adaptive changes which involved two separate products; an embedded device and a Windows desktop application. These changes arose due to changes in the VxWorks and Microsoft Visual Studio development platforms. Experienced developers at ABB originally performed the necessary changes manually on the source code. These changes dealt with changes to system APIs, runtime changes in the underlying platform, and changes to the compilers used to build the products. Below I will use part of their case study to motivate and explain the problem.

3.1 Change to Operator new

The operator new transformation was done in response to changes in the compiler, which were made in response to changes in the C++ standard. These changes had an effect on the semantics of operator new. Before these changes, it was common
practice to directly call the operator new in the initialization of a variable declaration. An example for their paper is given below:

```cpp
CNICmdFactory *cmdFactory = new CNICmdFactory;
```

Prior versions of the C++ ISO had operator new return 0 in the case of a memory error.

Error checking was done by checking the result for 0 at which point the developer could choose how they wished to deal with the problem (error handler, etc). In the new version, operator new was made to throw an exception. One fix for the problem is to wrap calls to operator new in a try/catch block, preserving the original behavior and maintaining correctness. An example from their paper is shown below:

```cpp
CNICmdFactory *cmdFactory;
try {
    cmdFactory = new CNICmdFactory;
} catch (...) {
```

![XSLT code]

**Figure 3.** A simplified view of the XSLT used for transformation of expression statements due to change in behavior of operator new after normalization refactoring. The full version includes the use of extension functions to match the indentation of the original. Note that only a single transformation needed to be written, in contrast to the 10 needed previously due to all the different syntactic constructs.
The transformation here, as one may have picked up, is to find specific instances where operator new is called and change them so that they correspond with this new behavior. The result of the transformation must then be applied to the original statement(s). In their work, they used XPath and XSLT to, respectively, match instances of operator new that needed to be transformed and then carry out the transformation. Through XPath they identified twelve different contexts using new that needed to be transformed. These contexts included the use of new inside expression statements, while loops, if statements, and return statements. Each of these distinct contexts may require a different transformation. For example, a function that uses the new operator in a return statement will need to be transformed differently than a new operator used in an assignment statement. Let’s walk step by step through transforming the following line of code in order to deal with new throwing an exception:

```cpp
CNICmdFactory *cmdFactory = new CNICmdFactory;
```

The context we must handle is a new statement occurring within a definition. First, we copy the declaration,

```cpp
CNICmdFactory *cmdFactory;
```

without the initialization,

```cpp
cmdFactory = new CNICmdFactory;
```

and create a new try/catch block. We place the copied declaration above the try block.

```cpp
CNICmdFactory *cmdFactory;
```
try {
    catch (...) {
    }
}

second, we move the initialization into the newly created try block,

    CNICmdFactory *cmdFactory;
    try {
        cmdFactory = new CNICdmFactory;
    } catch (...) {
    }

and then create an initialization to place into the catch block to handle the case where
new throws an exception,

    CNICmdFactory *cmdFactory;
    try {
        cmdFactory = new CNICdmFactory;
    } catch (...) {
        cmdFactory = NULL;
    }

This is a simple example, but there are those that are more complicated. One such
example found their case study follows:

    if ((nioMemCpy=new DataByte)==0) {}

For this transformation, one must move the assignment of nioMemCpy outside of the if-
statement while leaving the comparison with 0 intact inside of the if-statement. It’s
different for a while statement, where, in addition to the assignment being moved out and
the condition being left in, the assignment must be placed at the end of the while’s block
(and a block created if the while did not originally have one). These are only a small number of the problems inherent in writing transformations to make them semantically sound. For this study, the complete transformation for changing operator new involved twelve XSLT templates. A simplified view of the XSLT used for the operator new transformation can be seen in Figure 3.

3.2 Breadth of Transformation

In this section, it will be shown that problems to which transformational solutions are applied are many in number. Data from the case studies[Collard, Maletic, Robinson 2010] summarizing the number of changes made to two separate software systems is given below.

The first system they studied contained approximately 122 KLOC of C and C++ code contained in 405 files. The code studied was composed of a hybrid mix of procedural (54%) and object-oriented code (46%). This system had previously undergone manual adaptive maintenance due to changes in the way the gcc compiler handles the new operator in C++. The compiler adopted the new standard of throwing an exception when memory is not available instead of returning 0. The case study concerns the changes made to the system to adapt it to this new standard and whether the automated approach to fixing the problem was as effective as the manual approach. In order to qualify this, two baselines were used: The first was taken from before the manual changes were made; the second from after and is used to verify the automated transformation was successful.
Their automated approach was able to mimic every change that was previously made manually as well as apply these changes correctly. In addition to these results they actually identified a number of cases missed during the initial manual changes. They later found that these changes were made in future revisions, but the fact that the automated approach covered more of the needed changes is a testament to the usefulness of Program Transformation’s application to this sort of problem. The automated approach was able to identify and apply all changes with no additional cost or effort.

The system in the second study contained approximately 3.9 MLOC of C and C++ code spread among 13,800 source files. This system was composed of mostly object-oriented C++ code (84%) with the remaining 16% being procedurally designed C code. The system had recently undergone adaptive maintenance changes due to a C++ compiler migration (i.e., Visual Studio 2003 to Visual Studio 2005). Developers manually performed 1756 changes themselves.

<table>
<thead>
<tr>
<th>Table 1. Results for the second case study</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>Template Class</td>
</tr>
<tr>
<td>Iterator Variable</td>
</tr>
<tr>
<td>Deprecated Str</td>
</tr>
<tr>
<td>Data Private</td>
</tr>
<tr>
<td>Qualifying Functions</td>
</tr>
<tr>
<td>Totals</td>
</tr>
</tbody>
</table>

Table 1 shows the results of the transformations for the second case study from their paper. The automated approach applied all 1756 changes correctly and identified 304 changes that were missed during the initial manual change to the system but, again,
corrected in future updates to the system. The changes that were missed, just as in the first case study, were not odd or corner-case changes, but merely missed due to the manual effort it takes to undertake adaptive maintenance on this scale.

In order to accomplish these changes, the authors needed to, as stated previously, deal with every variation in syntax manually. They did it successfully, but at that cost of very hard write and understand XSLT. Due to the sheer number of changes that need to be made to the system, minimizing the number of cases that need to be manually written into a transformation is desirable. The less the writer of a transformation needs to worry about syntax variation, the more said writer can focus on fixing the problem the transformation is meant to solve. With this I have given a glimpse into why automatic transformations are desirable as well as why it is pertinent that we make the application of these transformations as easy to conceive of and write as possible. The point of applying transformational approaches to these problems is to make it easier to deal with them as well as more efficient than manually addressing them. If transformations are difficult to write and error-prone (due to being difficult to write or otherwise) then their advantages will not be taken advantage of in software maintenance tasks like adaptive maintenance.
CHAPTER 4

Simplifying Transformations

As a quick note, the terms Isomorphism and Syntactic Variation will be used interchangeably. An isomorphism (Syntactic Variation) refers to the set of semantically identical but syntactically unique ways of writing an expression.

For example,

```java
if(condition1 == condition2){}
```

and

```java
bool var condition1 == condition2;
if(var){}
```

are isomorphic of one another in terms of their if-statements. If condition1 and condition2 have the same state in both examples, then both examples have identical semantics (you can interchange one for the other and you’d get the same behavior).

The goal here is to simplify the construction of program transformations by reducing the number of isomorphisms to which the transformation is applied. We first identify all the syntactic variations and then present a set of refactorings to normalize the various cases into one normal form.

4.1 Syntactic Variation in Expressions

When confronted with an adaptive maintenance task, developers need to identify each particular change and then develop transformations to address each. I have given examples of how this may be done using srcML in previous sections. Ideally, a complete
set of example changes would be on hand to develop the entire set of transformation necessary to undertake the given adaptive maintenance task. Typically developers manually identify the necessary types of adaptive changes from examining systems that have undergone these changes. Additionally, the documentation for the new releases of APIs, compilers, and operating systems typically contain lists of the major changes and some basic suggestions for what needs to be corrected to support new features and interfaces.

However, even given a complete set of necessary items to change to support an adaptive maintenance task will not cover all situations developers will need to be sensitive to within the context of all software systems. That is, it is very difficult to predict all the usages of an API or feature for any given system. Conversely, if given a set of changes that are known to be involved in some adaptive change we must determine if they all address the same problem or not. As such, we must identify all syntactic variations of each item to be changed in the context of the system.

We’ll now revisit the adaptive change involving the new operator. As the details of this change presents a good example of the problems encountered in even this fairly simple adaptive task. The general goal is to transform the code fragments such as the one below:

```
CNICmdFactory *cmdFactory = new CNICmdFactory;
```

To the following:

```
CNICmdFactory *cmdFactory;
try {
```
This is in response to a change in the C++ language standard as described previously. To construct a transformation to address this adaptive change the specific statements that include a call to the operator new must be identified, then the expression that calls new must be isolated in a try/catch block. Finally, the result must be tied into the original statement.

However, this (a declaration statement) is not the only place the new operator can occur. Each situation must be either handled separately, with a separate transformation, or a very complicated single transformation can be developed. In our experience, identifying all the different possible syntactic usages of a given construct and developing the specific transformations can be fairly complicated and error prone. Below is a (incomplete) list of different possible syntactic uses of the new operator.

```cpp
int *ptr;
while ((ptr = new int[n]) != 0) {       //Or in other construct
    ...}
ptr = new int[n];                       //Expression statement
foo (new int[n]);                       //As a parameter to a function/method
return new int[n];                      //In a return
```
typedef int* ptr;
ptr B[2] = {new int[10], new int[10]};  //As a parameter to an array constructor

Given these isomorphisms, one of two things is necessary: A transformation to cover all cases, or a way of handling the variations in this syntax so that a transformation need not worry about them. My approach does the latter of these two and will be discussed next.

4.2 Normalizing-Refactorings

I have developed a set of refactorings, specifically for C++, which can be selectively applied to a part of the code to normalize certain syntactic features of a language. I want to stress that they are selectively applied in order to reduce and localize the impact of code modifications so that they are restricted only to the adaptive maintenance task at hand.

I term these refactorings Normalizing-Refactorings; they are akin to *term rewriting* in compiler terminology, which is used to simplify syntax in order to produce target code. The differences between Normalizing-Refactorings and term rewriting will be discussed in Chapter 6 but, in short, Normalizing-Refactorings have a different goal and are not applied globally to all situations.

When applying Normalizing-Refactorings, many complicated syntactic variations are converted into a series of normal-forms that can in turn be more easily converted into
the final target. The use of the word normal-form refers to a standard syntax for every isomorphism of an expression within the scope of the source to which Normalizing-Refactorings are applied. A normal form is an isomorphism of the original syntax of the expression and was chosen by taking the simplest form of the expression expressible in terms of the language grammar. As an example, in the various different cases of the new operator, we apply my Normalizing-Refactoring techqiue to simplify each syntactic variation to an equivalent, normal-form. That is, one syntax that only needs one transformation rule (for this particular adaptive change).

For example,

```c
int *ptr = new int[n];
```

is be refactored to:

```c
int *ptr;
ptr = new int[n];
```

and

```c
int *ptr;
while ((ptr = new int[n]) != 0) {
    ...
}
```

is be refactored into:

```c
int *ptr;
ptr = new int[n];
while (true) {
    bool var1 = ptr != 0
```
if(!var1){
    break;
}
...
ptr = new int[n];
}

and
return new int[n];
is refactored to:
int *ptr;
ptr = new int[n];
return ptr;

I term this particular Normalizing-Refactoring an *R-Value Normalization* due to the fact that the isomorphisms we are handling all contain R-Values, which are expressions that may only occur on the right hand side of an assignment statement. In fact, most Normalizing-Refactorings are of this category. In the above example, R-Value Normalization simplifies each of the present syntactic variations of the new operator into normal-form, namely an assignment statement. As a consequence of this simplification, one transformation rule can be applied to all the different usages, resulting in a consistent solution with minimal impact on the source code structure and formatting. That is, the code will continue to be understandable and maintainable after the Normalizing-Refactorings are automatically applied.

The statement:
ptr = new int[n];

in all of the above examples would be replaced with

```java
try {
    ptr = new int[n];
} catch (...) {
    ptr = 0;
}
```

via a transformation that takes place after the Normalizing-Refactorings and this transformation would need only to match that single expression instead of the many forms of the expression present in the source code before the refactorings were applied.

---

**Figure 4. Grammar for Normalizing-Refactorings**

---

I believe that I have formulated the complete set of R-Value Normalizations for all appropriate syntactic situations in C++. To define Normalizing-Refactorings more concisely, I will now define terminology for the purpose of explaining how Normalizing-
Refactorings work. These terms generally refer to expressions within a specific context that Normalizing-Refactorings must be able to handle.

Figure 4 gives the BNF definitions of these terms.

An Expression-Atom (EXPRATOM) is an expression that is already in normal-form. That is, it does not need to be further simplified in order to be in normal-form. Expression-Atoms are the building blocks of the concept of Normalizing-Refactorings as they represent the Normal-Forms we seek. The second term is Compound-Expression.

Table 2. Generalized technique for normalizing Compound Expressions

<table>
<thead>
<tr>
<th>Compound Expression</th>
<th>Normalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPR1 op EXPR2 op ... op EXPRn</td>
<td>TYPE temp; Temp = EXPR1 op EXPR2 op ... op EXPRn</td>
</tr>
<tr>
<td>EXPR1(EXPR21(...EXPRn1, EXPRn2),EXPR22)</td>
<td>TYPE temp; temp = EXPRn1; TYPE temp2 temp2 = EXPRn2; TYPE temp3; temp3 = EXPR21; TYPE temp4 temp4 = EXPR22; TYPE temp5; temp5 = EXPR1(temp3(temp, temp2), temp4);</td>
</tr>
</tbody>
</table>

Table 2 presents a generalized view at what composes a Compound-Expression and how they’re normalized. A Compound-Expression is exactly as it sounds; an expression made up of Expression-Atoms. Because it is made up of Expression-Atoms, it make be broken down into those separate Expression-Atoms, which is exactly what we
do to get to the Normal-Form. A Compound-Expression is never an expression that is in normal-form. The final term is Expression-Set. An Expression-Set is a set of expressions. The elements in this set may be Compound-Expressions or Expression-Atoms. Expression-Sets in C++ may correspond to function arguments, for-loop condition arguments, etc. As with Compound-Expressions, an Expression-Set is never (completely) in normal-form, if it were in Normal-Form, it would be an Expression-Atom.

When applying R-Value Normalization, we want to figure out which of these situations we’re dealing with; if the expression we are normalizing is an Expression-Atom, or if it is a Compound-Expression or an Expression-Set. Below are examples of expressions that may have R-Value Normalization applied to them and what form they correspond to.

The following is a Compound-Expression:

```c++
Foo(str1+str2);
```

In this case, there are two Expression-Atoms; the call to ‘Foo’ and the argument ‘str1+str2’. To normalize this, we extract the inner Expression-Atom (str1+str2) and store its value in a variable before placing this new variable into the function call.

```c++
auto temp = str1+str2;
Foo(temp);
```

The function call is now an Expression-Atom because it is in its most simple, behaviorally consistent format in terms of the language grammar. To understand this example, another, slightly more complicated example will be given below.

The following is a Compound-Expression:
Foo(string(str2));

This Compound-Expression must be broken apart similarly to the example above. The result of applying R-Value Normalization follows:

auto temp1 = string(str2);

Foo(temp1);

With both examples, it’s hopefully clear how Normalizing-Refactorings are applied to expressions of this form.

<table>
<thead>
<tr>
<th>Expression Set</th>
<th>Normalization</th>
</tr>
</thead>
</table>
| F(EXPR1, EXPR2, ..., EXPRn) | TYPE1 TEMP1;
|                          | TYPE2 TEMP2;
|                          | ... |
|                          | TYPEn TEMPn;
|                          | TEMP1 = EXPR1;
|                          | TEMP2 = EXPR2;
|                          | ... |
|                          | TEMPn = EXPRn;
|                          | F(TEMP1, TEMP2, ..., TEMPn); |

Notice how both expressions were calls to a function called Foo but had very different syntax for their arguments. This is why Normalizing-Refactorings are useful; now both calls to Foo are of the form Foo(variable) and can be easily transformed using a transformation that only deals with this single syntax instead of both separate syntaxes. All of the above expressions (after Normalizing-Refactorings were applied) are Expression-Atoms; they can no longer be broken apart. Further examples of Compound-Expressions can be found in Table 5. Finally, here is an example of an Expression-Set:
This is an Expression-Set because there may be multiple Expression-Atoms or Compound-Expressions inside of it. In this example, each argument to Foo is an element in the Expression-Set. Notice that one argument (str.substring(str.find(str2))) is a Compound-Expression and the other argument (vector()) is an Expression-Atom. Table 5, Table 6, and Table 7 have many examples of R-Value normalization to help in understanding exactly how these are applied in C++.

Aside from R-Value normalization, there is one other type of Normalizing, called Template Metafunction Normalization. This involves taking complex Metafunctions and placing them into a separate Typedef much the same way expressions were removed from function arguments above and stored in variables. Examples of this are in Table 4.
Normalizing-Refactorings are similar to the Fowler catalog of refactoring [Fowler 1999]. When applied to an initialization expression, R-Value Normalization splits the assignment of the variable from the declaration of a variable. This is the inverse of Fowler’s Replace Assignment, which combines the assignment of a variable with its declaration. The R-Value Normalization for Compound-Expressions introduces a temporary variable for the value of the expression; this is the inverse of Fowler’s Inline Temp Refactoring, which replaces a temporary variable with its value. Fowler’s catalog includes many inverse refactorings, e.g., Extract Method and Inline Method. One reason that inverses to Replace Assignment with Initialization and Inline Temp are not in Fowler’s catalog is that these inverses do not make the code more understandable or readable. However, for purposes of reducing the number of isomorphisms a user must consider before applying transformations, they are applicable. There is no equivalent to Template Metafunction Normalizing in Fowler’s catalog as Fowler’s was derived from Java and does not deal with template refactoring issues.

An important similarity to Fowler’s refactorings is that Normalizing-Refactorings do not change the behavior of the code. This permits the application of a test suite after the Normalizing-Refactorings are performed, but before the general purpose
transformation is applied to ensure that the Normalizing-Refactorings did not impact the behavior of the system.

It is worth noting that since the resulting code from these transformations will be manually maintained by developers going forward, any changes to the overall structure or style may lead to a rejection of the new changes, as described in a number of studies on past real world projects [Cordy 2003; Van De Vanter 2002]. Examples have been given [Cordy 2003] on large projects where any potential changes to the system had to be presented to the programmers in the exact view of the source code that they were familiar with. If not, the proposed changes were rejected. The Normalizing-Refactorings are, therefore, made to make as little impact on the structure of the code as possible by both limiting the scope to which they are applied as well as only normalizing statements that must be normalized in order to ease the construction of a general transformation.
### Table 5. List of Normalizing-Refactorings for Compound Expressions

<table>
<thead>
<tr>
<th>Compound Expression Examples</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>//declaration with initialization</code> aType object = otherObject.getWidth();</td>
<td>aType object; object = otherObject.getWidth();</td>
</tr>
<tr>
<td><code>auto temp = g(); aType object;</code> Try { object = temp+f(); } catch(...) { object = temp+SomeUserDefinedPlaceHolder; }</td>
<td></td>
</tr>
<tr>
<td><code>auto temp; temp = foo(a, b); aType object(temp);</code></td>
<td></td>
</tr>
<tr>
<td><code>aType* foo() { return(new aType()); }</code></td>
<td>aType* foo() { aType* object; object = new aType(); return(object); }</td>
</tr>
<tr>
<td><code>aType(): Memer2(“String”), …, MemerN(“String”)</code></td>
<td>aType(): Memer2(“String”), …, MemerN(“String”)</td>
</tr>
<tr>
<td><code>{ someType *object = new someType(); Member1(object); … }</code></td>
<td></td>
</tr>
</tbody>
</table>
Table 6. List of Normalizing-Refactorings for Compound Expressions (cont.)

<table>
<thead>
<tr>
<th>Compound Expression Examples (cont.)</th>
<th>Normalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>//condition of if-statement if (Object.IsEmpty() &amp;&amp; Go()) {</td>
<td>bool temp;</td>
</tr>
<tr>
<td>}</td>
<td>temp = Object.isEmpty() &amp;&amp; Go();</td>
</tr>
<tr>
<td></td>
<td>if (temp) {</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>//condition of if-else if (condition1.MeetCondition()) {</td>
<td>bool temp1;</td>
</tr>
<tr>
<td>} else if (condition2.MeetCondition()) {</td>
<td>temp = condition1.MeetCondition();</td>
</tr>
<tr>
<td>}</td>
<td>if (temp1) {</td>
</tr>
<tr>
<td></td>
<td>doSomething();</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>else {</td>
</tr>
<tr>
<td></td>
<td>bool temp2;</td>
</tr>
<tr>
<td></td>
<td>temp2 = condition2.MeetCondition();</td>
</tr>
<tr>
<td></td>
<td>if (temp2) {</td>
</tr>
<tr>
<td></td>
<td>doTheOtherThing();</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>//condition of while-loop while (!success() == true) {</td>
<td>while (true) {</td>
</tr>
<tr>
<td>}</td>
<td>bool temp;</td>
</tr>
<tr>
<td></td>
<td>temp = success == true;</td>
</tr>
<tr>
<td></td>
<td>if (!temp){</td>
</tr>
<tr>
<td></td>
<td>break;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>KeepTrying();</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>//condition of while-loop with if-statement while (!success() == true) {</td>
<td>while (true) {</td>
</tr>
<tr>
<td>}</td>
<td>bool temp1;</td>
</tr>
<tr>
<td></td>
<td>templ = success==true;</td>
</tr>
<tr>
<td></td>
<td>if (!temp1) {</td>
</tr>
<tr>
<td></td>
<td>break;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>bool temp2 = hungry&amp;&amp;dinnertime;</td>
</tr>
<tr>
<td></td>
<td>if (!temp2){</td>
</tr>
<tr>
<td></td>
<td>goto label</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>else {</td>
</tr>
<tr>
<td></td>
<td>GoEatDinner();</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>label:</td>
</tr>
</tbody>
</table>

Table 7. List of Normalizing-Refactorings for Expression Sets

<table>
<thead>
<tr>
<th>Expression Set</th>
<th>Normalizing-Refactorings</th>
</tr>
</thead>
<tbody>
<tr>
<td>//function call</td>
<td>foo(new someType(), bar(getSize())); someType *object2; object2 = new someType(); int temp2; temp2 = bar(getSize()); foo(object2, temp2);</td>
</tr>
<tr>
<td>//object construction</td>
<td>aType object; someType *object2; object2 = new someType(); int temp2; temp2 = bar(getSize()); object(object2, temp2);</td>
</tr>
<tr>
<td>//for-loop</td>
<td>int i; i = 0; while (true) { bool temp; temp = i &lt; this.size(); if (!temp) break; doSomething(); ++i; }</td>
</tr>
</tbody>
</table>

```java
//function call
foo(new someType(), bar(getSize()));

//object construction
aType object;
object(new someType(), bar(getSize()));

//for-loop
for (int i = 0; i < this.size(); ++i) {
doSomething();
}
```
CHAPTER 5

Evaluation & Implementation

I now discuss some of the implementation details pertaining to how the Normalizing-Refactorings were developed and then applied to a non-trivial software system for evaluation.

5.1 Implementation

Normalizing-Refactorings are currently implemented using a libxml2 wrapper called libxml++, which wraps the c-like interface for libxml2 in a C++-like interface. The current version of the tool, called nRefac, builds and compiles on linux using the gcc compiler. It does not yet build or run on Windows. I use the srcML format, summarized previously, as the representation of source code to support transformation via Normalizing-Refactorings.

<table>
<thead>
<tr>
<th>Type</th>
<th>Total Normalized</th>
<th>Percentage of Total Refactorings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>5566</td>
<td>35%</td>
</tr>
<tr>
<td>Function Calls</td>
<td>3981</td>
<td>25%</td>
</tr>
<tr>
<td>If</td>
<td>3418</td>
<td>22%</td>
</tr>
<tr>
<td>Return</td>
<td>1577</td>
<td>9%</td>
</tr>
<tr>
<td>For</td>
<td>1134</td>
<td>7%</td>
</tr>
<tr>
<td>While</td>
<td>157</td>
<td>1%</td>
</tr>
<tr>
<td>Else If</td>
<td>154</td>
<td>1%</td>
</tr>
</tbody>
</table>
This is a two-step process just like any transformational approach. First, we must specify the locations at which Normalizing-Refactorings should be applied. With the srcML format, these locations in the code can be specified with an XPath expression that uses the srcML markup. For example, to match all if-statements that have the new operator in a condition (for C++) the XPath for srcML is:

```
//src:if[src:condition/
    op:operator='new']
```

Second, given the particular statement/location to be transformed, we must apply the transformation. Since we are normalizing code, the transformations are already known to nRefac and require, in very few situations, only a small amount of user interaction. That is, nRefac knows how to take syntax within a given node and transform it into the proper Expression-Atom with little to no help. Once the refactorings are applied to the srcML archive, the next logical question is how to apply what has been done to the code base. As stated prior, the toolkits that support srcML allow for

![Diagram](diagram.png)

**Figure 5. Example of srcML toolset workflow**
translation both to and from the format. In this case, the srcml2src tool is used to translate srcML back to source code. A graphic of this workflow is given in Figure 5

5.2 Evaluation

To properly evaluate Normalizing-Refactorings, it was decided that the approach should be run on a system of non-trivial size and the result of the refactorings compiled to ensure the behavior of the system has not been modified by application of the Normalizing-Refactorings. This would ensure that the refactorings correctly modified the target system and that any future transformations to the system may localize any errors resulting post-transformation to the transformation itself. The evaluation of Normalizing-Refactorings is completed using the following five steps:

- Build and compile the system that the tool will be run on.

- Run the system’s provided unit tests to make certain that it is behaviorally sound.

Once it is certain that the system operates correctly without the normalizings, the next three steps are as follows:

- Run the Normalizing-Refactoring tool on the system. This will change the syntax of all supported constructs indiscriminately to their normal form.

- Re-build and re-compile the system after the tool has finished.

- Run the system’s provided unit tests to be certain that, although the syntax of many structures has changed, the system is still behaviorally equivalent to what it was before Normalizing-Refactorings were applied.
The system used to evaluate Normalizing-Refactorings is the open source project called the Bullet Physics Engine\(^1\). Bullet provides a number of features used in games and movies to support physical interaction with simulated environments. The version of Bullet that I used has 75kloc (bullet-2.81-rev2613). It was chosen both for the ease of modifying the build system and the irregular syntax normally found in code-bases that employ a lot of low-level memory manipulation and code optimization.

Bullet’s test cases can be broken down into module-based tests. That is, Bullet’s support for game and animation physics is broken down into a series of modules that handles physics for different situations. The unit tests that Bullet uses, which number fourteen in total, are made to test each of these modules. Despite not having tests for each function in the system, the module-level granularity of the tests does appear cover each class present in the Bullet engine. The execution time for Bullet was about 8.5 seconds and the total number of completed refactorings numbered 15,987. Table 8 shows the total number of refactorings divided up by which structure was normalized. Initialization statements, Function Calls, and If-statements make up the bulk of Normalizing-Refactorings applied to the system. Considering those numbers, a change to the arguments of a function in some adaptive maintenance task could have a fair impact assuming all function calls in the system aren’t unique. All unit tests came back positive; the system post-Normalizing-Refactorings was semantically equivalent to what it was pre-Normalizing-Refactorings.

\(^1\) See http://bulletphysics.org/wordpress/
With a reasonable base of unit tests with which to measure the impact Normalizing-Refactorings have had on the system, it’s not hard to reason that future transformations made to the system on top of Normalizing-Refactorings have a higher chance of causing fewer errors and, also, the errors that are caused by the transformation have little chance of being compounding errors resulting from application of Normalizing-Refactorings and then some transformation—the error would have to come from the transformation itself since the Normalizing-Refactorings were previously proven, via unit tests, to be sound.
CHAPTER 6

Related Work

Normalizing-refactorings were made to operate within a number of constraints important to the maintenance and evolution of large software systems [Cordy 2003; Van De Vanter 2002]. These constraints include preserving the programmer’s view of the code, preserving comments, and having minimal impact on the structure of the code-base. Since the refactorings operate on top of srcML, we are able to take advantage of all XML technologies and guarantee minimal impact on the overall structure of code, as discussed above.

Normalizing-Refactorings are related to Term Rewriting[Baader, Nipkow 1998]. Term Rewriting is a technique used to simplify expressions to a Normal Form. In textbook term rewriting, the primary focus is repeated application of simplification rules to an expression until that expression converges; the result of this convergence being the Normal Form. With Normalizing-Refactorings, instead of changing the expression, we store the meaning of the expression in a variable and replace the expression with that same variable. The effect is akin to Term Rewriting in that we’ve simplified the expression (by replacing it with a single variable) using (rewrite) rules for how different expressions should be handled (rules for expressions within if statements, expressions in functions call arguments, expressions within return values, etc). Where it differs from Term Rewriting is in the way that the expression’s meaning is preserved. In Term Rewriting, the meaning is preserved because the rules used to rewrite it constrain these
rewrites to semantically identical post-rewrite expressions. With Normalizing-Refactorings, we preserve meaning by simply paying close attention to how we extract expressions whose results will ultimately be stored within variables. Again, Normalizing-Refactorings must preserve the programmer’s view of the code (as described above), must impact only specific segments of the code base (as defined by the user), and are catered specifically to address concerns with Program Transformation as it applies to Software Maintenance; all things that Term Rewriting is not meant to accomplish.

Normalizing-Refactoring is a technique that has not been applied anywhere else by any tool readily available open source. However, many Program Transformation tools must either solve the problems that Normalizing-Refactorings addresses or push the problem to the user to solve. Normally, the latter is the case; the user must deal with isomorphisms themselves. This category of Transformation Language would include ASF+SDF[Brand, Heering, Klint, Olivier 2002], RASCAL[Klint, Storm, Vinju 2009], ELAN[Borovanský et al. 1996], STRATEGO[Visser 2001], TXL[Cordy, Dean, Malton, Schneider 2002] and DMS[Baxter, Pidgeon, Mehlich 2004]. ASF+SDF and RASCAL are iterations of one another, with RASCAL being an advancement on ASF+SDF. They both work on ASTrees created by a combination of SDF and SGLR parsing. STRATEGO and ELAN introduce the concept of rewrite strategies with ELAN being the first to explore the idea and Stratego generalizing rewrite strategies to make them more modular than ELAN. While these languages don’t implement Normalizing-Refactorings directly, many of them could be used to implement them and would make their users’ jobs easier if they
were to. Coccinelle[Lawall 2005] uses a language that leverages a technique similar to Normalizing-Refactorings. This language is called Semantic Patch[Padioleau, Hansen, Lawall, Muller 2006]. Semantic Patch shares a number of features with Normalizing-Refactorings in that it handles isomorphisms by storing their result in a variable and allowing the user to operate on that variable, much like tool I created. The difference between Normalizing-Refactorings and Semantic Patch is that Semantic Patch substitutes variables in for isomorphisms so that irrelevant parts of the code are ignored at transformation time. The difference between their techniques and ours is that their variables are defined by the user. That is, the user must know where isomorphisms exist within the code and manually mark them with a variable so that they’re ignored at transformation time. This means that every new transformation requires the user to go through and substitute variables in for isomorphisms by hand. My technique automatically removes isomorphisms within a scope defined by the user, which means that the user need not worry about them at all and may write transformations in terms of the expected normalizings that nRefac will apply to the source code.
CHAPTER 7

Conclusions and Future Work

A set of Normalizing-Refactorings for C++ were presented that standardizes source code syntax of semantically identical expressions. Traditional approaches to program transformation in software maintenance currently require developers deal with isomorphisms in the code. That is, they must write a separate transformation for every isomorphism corresponding to the expression(s) they’re attempting to transform. Normalizing-Refactorings alleviate this problem by creating a single syntax of all of these isomorphisms. This makes the specification of the transformations themselves not only easier, but also reduces the number of transformations one needs to construct to handle every case of varying syntax.

A full set of Normalizing-Refactorings for C++ was presented along with a tool that implements the Normalizing-Refactorings using the srcML. These refactoring were evaluated on a medium sized open source software system. Since a set of test cases were provided for the system I was able to verify that the Normalizing-Refactorings were behavior preserving. That is, the test cases all passed before and after applying the refactorings.

Software Engineering sometimes involves the need to adapt software to a new environment, compiler, or operating system. These changes are costly as programmers will need to update their code to correspond with new APIs or code-structure requirements inside of the new environment. The update could be to millions of lines of
code and even with a professional team of software engineers this is a daunting task. The use of program transformation to automate these structure changes alleviates the need for exhaustive, partially-automated or fully-manual matching and transformation by fully automating the task and freeing engineers to focus on writing the transformations themselves and then verifying their completeness and correctness. It removes one of the largest steps in the entire process.

7.1 Future Work

Normalizing-Refactorings can be implemented for other languages. The building blocks for that are already present in this paper, but applying Normalizing-Refactorings to a language like Python will require a separate implementation and differing requirements from their application to C++. Alongside this, a Program Transformation language that automatically applies Normalizing-refactorings as part of its underlying transformation engine would allow for users to transform code without ever having to worry about the Normalizing-Refactorings themselves. The current implementation of Normalizing-Refactorings has to manipulate the structure of the code in order to normalize syntax; it may be possible to un-normalize code and return it to a form closer to its original, which would further minimize their impact on the code therefore further preserving the programmer’s view of the code.
References


