TECHNIQUES FOR SOFTWARE RENOVATION

By

Michael Benjamin Siff

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

(Computer Science)

at the

UNIVERSITY OF WISCONSIN – MADISON

1998

Abstract

Software renovation is the process of introducing new features—including polymorphism, objects, and encapsulation—into existing software systems while preserving the original functionality of the system. The goal of software renovation is to improve the efficiency of development, maintenance, and comprehension. The research described in this thesis focuses on three software-renovation techniques:

1. *Generalization*: The identification and subsequent transformation of program components that operate on a particular type of input into *polymorphic* program components that operate on a wide array of inputs.

2. *Modularization*: The clustering of associated data types and functions with the intent of *encapsulating* the types and function into distinct classes or modules.

3. *Physical subtyping*: The identification of relationships among data types based on the representation of the types in memory with the intent of generating inheritance hierarchies.

The techniques described in the thesis are aimed particularly at the problem of transforming legacy C programs into C++ programs that make use of C++'s advanced features—most notably classes, templates, inheritance, and virtual functions. Some aspects of this work apply specifically to the C-to-C++ problem; however, most aspects apply to almost any language.

Acknowledgements

First and foremost, I would like to thank my advisor, Tom Reps, for his guidance and support over the last four years. I particularly appreciated Tom's patience for the long period during which I was searching for a thesis topic. I also would like to thank him for his open mindedness in allowing me to pursue a topic significantly different from the main avenues of research our group had been investigating.

I would like to thank Charles Fischer and Marvin Solomon for their comments on this thesis and for participating in my thesis defense. I would also like to thank Anne Condon and Ken Kunen for sitting on the thesis committee. Special thanks to Anne, Susan Horwitz, and Eric Bach for their guidance and encouragement over the last several years.

Thanks to my longtime officemate Manuvir Das for making coming to work entertaining even when research was difficult. Thanks also to Manuvir for doing me the service of completing his thesis six months prior to me, and, by doing so, showing me the ropes.

Thanks to Tom Ball for being a friend and mentor, rather than just a mentor.

Thanks to Larry and Jean Landweber for all the support (and food!) they have offered me over the last five years.

Thanks to all the other graduate students in Computer Sciences who have made the last five years more palatable. Particular thanks to Mark Callaghan, John Watrous, Bruce Irvin, Doug Burger, Babak Falsafi, and T. Vijaykumar.

I should not thank my high-school friends, because if it was not for them I would have more gladly gone away from home to college (I did anyway). I should not thank my college friends, because if it was not for them I would have more gladly moved to the Midwest upon finishing college (I did anyway). I should not thank my friends in Madison, because if it was not for them I might have finished sooner and I would not be sad about leaving here now (I have to anyway). But I thank all of these friends, nevertheless. In some cases they have been an inspiration to work; in all cases they have been an inspiration to smile.

I fear that by listing names of friends who have made my life better over the last few years I will accidentally leave someone's name off, or, worse, run out of acid-free paper. Instead, I'll just list a few: To Bill Raich, Himelblau, Tim Hintz, Jermaine Jones, and Pete Jacobs for putting up with me as a housemate and keeping me well fed. To Greg Ford for knowing what I've been through. To James Markham, Marc Zampetti, and Paul Cappelluzzo for putting up with me and Greg over the last five years. To Sherry Kuchma for just being Sherry.

I would like to thank my entire family for giving me the gift of rambling which has allowed me to complete this thesis as well as this prolonged Acknowledgements section. In particularly, I would like to thank my parents, Daniel and Joan, to whom this thesis is dedicated. Without their love and kindness—I don't even want to speculate. I would also like to thank my brother Andrew for being not only a great brother, but also a great friend, and for instilling in me a love of math at an early age.

Finally, I wish to thank Krishna Kunchithapadam whose friendship and guidance has simply been invaluable in writing this thesis.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Software renovation is the process of introducing new features—including polymorphism, objects, encapsulation, and parallelism—into existing software systems while preserving the original functionality of the system. The goal of software renovation is to improve the efficiency of development, maintenance, and comprehension. The research described in this thesis focuses on three renovation techniques:

1. *Generalization*: The identification and subsequent transformation of program components that operate on a particular type of input into *polymorphic* program components that operate on a wide array of inputs.

2. *Modularization*: The clustering of associated data types and functions with the intent of *encapsulating* the types and function into distinct classes or modules.

3. *Physical subtyping*: The identification of relationships among data types based on the representation of the types in memory with the intent of generating inheritance hierarchies.

The techniques described herein are aimed particularly at the problem of transforming legacy C programs into C++ programs that make use of C++'s advanced features—most notably classes, templates, inheritance, and virtual functions. Some aspects of this work apply specifically to the C-to-C++ problem; however, most aspects apply to almost any language. The motivations for focusing on the C-to-C++ conversion problem can be summarized as follows:

- *Practical needs.* The conversion of existing C code to C++ is a practical problem of current interest and economic importance. The last decade or so has seen the gradual transition from C to C++ as the standard programming language for systems development. While companies recognize the superiority of object-oriented languages, particularly C++, for large system implementation and maintenance [7], there is a plethora of large-scale legacy systems written in C that are actively used. These are difficult to maintain because of their size and complexity. The C-to-C++ conversion problem is also important because:
  - C++ is easier to maintain because it facilitates reuse and encapsulation.
  - Maintenance tasks are often carried out by new software engineers, who these days are trained primarily in C++ and not in C.
- *Encapsulation.* Information hiding (or encapsulation) is the separation of the external aspects (the *interface*) of a data type, from its internal implementation details [45, 56]. Encapsulation prevents a program from becoming so interdependent that small changes have large ripple effects. C++ supports private and protected members of classes, whereas C has no mechanism for encapsulation.
- *Type safety.* C++ offers templates, inheritance, and virtual functions, all of which can be employed in a type-safe manner. C can emulate many of C++'s advanced features via the use of void pointers and type casts, both of which

sacrifice type safety.

- *Ease of transformation.* C-to-C++ conversion problem is eased by the fact that C++ is (essentially) an upwards-compatible extension of C. The final step involved in the conversion process can often be carried out by a straightforward syntax-directed translation. The transformation need only make changes to certain portions of the C code: some base types are replaced by template-argument types; constructors are inserted in certain assignment expressions; some function calls are replaced by method invocation, etc. A large portion of the code can remain unchanged.

We now illustrate, with a few small examples, the three software renovation techniques described in this thesis as applied to the C-to-C++ conversion problem:

1. *Generalization.* Formally, generalization is a transformation of a program component C into a parameterized program component C' such that C' is usable in a wider variety of syntactic contexts than C. Furthermore, C' should be a semantically meaningful generalization of C; namely, there must exist an instantiation of C' that is equivalent in functionality to C. In the context of the C-to-C++ conversion problem, generalization refers to the transformation of C functions that operate on a particular type of input to C++ function templates that operate on a wide array of inputs.

   For example, `power`, the integer exponentiation function shown in Figure 1, expects two inputs: an integer base and an integer exponent. A power function template, such as the one shown in Figure 2, can take as the base a value of any

```
int power(int base, int n)
{
  int p;

  p = 1;
  while(n > 0) {
    p = p * base;
    n--;
  }
  return p;
}
```

Figure 1: The `power` function.

type for which a multiplication operator is defined and for which there is a constructor from integers (so that `p` can be initialized from the value `1`). Since C++ allows for user-defined constructors and operator overloading, this template can now be applied to a potentially unbounded collection of types. Figure 3 illustrates the difference between the monomorphic `power` function and the polymorphic `power` template. In Chapter 2, we describe how this form of generalization can be achieved automatically using polymorphic type inference.

2. *Modularization.* The goal of modularization is to encapsulate data types with related functions into distinct classes or modules. Ideally, modules have a clear-cut purpose (they are *maximally cohesive*) and are independent of each other (they are *minimally coupled*) [52]. In the C-to-C++ domain, modularization refers to the identification of `struct` types and functions that manipulate values of those types that can be sensibly organized into C++ classes.

   As an example, consider the C implementation of stacks and queues shown in

```
template<class T> T power(T base, int n)
{
  T p;

  p = T(1);
  while(n > 0) {
    p = p * base;
    n--;
  }
  return p;
}
```

Figure 2: A C++ `power` function template.

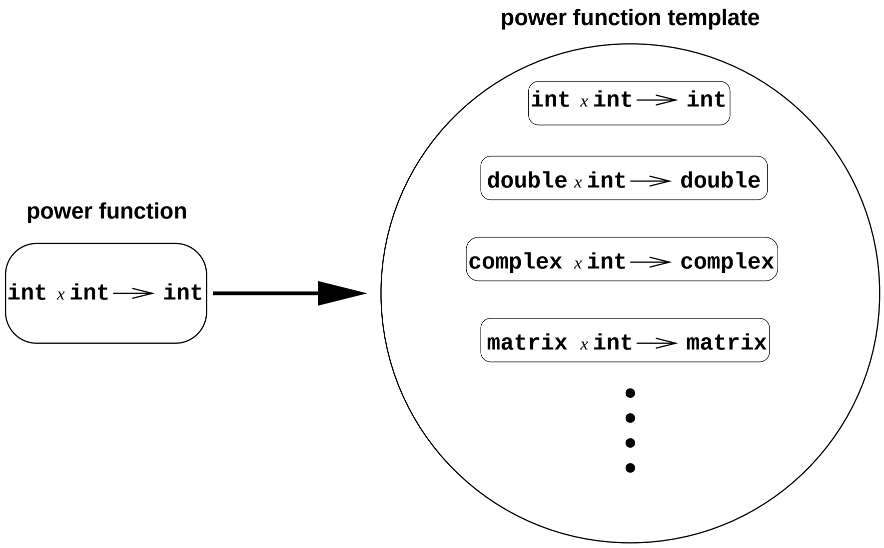

Figure 3: The generalization process: The type of the power function versus the possible types of a power function template.

Figure 4. Queues are represented by two stacks, one for the front and one for the back; information is shifted from the front stack to the back stack when the back stack is empty. The queue functions only make use of the stack fields indirectly—by calling the stack functions. Although the stack and queue functions are written in an interleaved order, we would like to be able to tease the two components apart and make them separate classes, one a client of the other, as in the C++ code given in Figure 5. In Chapter 3, we explain how a technique known as *concept analysis* can be used to identify potential modules (in this case C++ classes) in legacy C code. The resulting information can then be supplied to a suitable transformation tool that maps C code to C++ code, as in the queue and stack example.

3. *Physical subtyping.* Although C does not support subtyping (in the style of object-oriented languages), programmers often take advantage of the physical layout of program data in memory to simulate subtyping. For the C-to-C++ conversion problem, the identification of physical subtypes in C programs provides a mechanism that can be used to identify base classes, subclasses, and virtual functions.

   As an example, consider the C code shown in Figure 6. The function `translateX` is declared to take two arguments: a pointer to a `Point`, `p`, and an integer, `x`. The function translates `p`'s horizontal component by `x` units. If `pt` is declared to be a `Point`, then the expression `translateX(&pt,1)` is legal C. We might also wish to apply `translateX` to a variable `cp` of type `ColorPoint`, but the statement `translateX(&cp,1)` is *not* legal in C. We can *cast* a pointer to a `ColorPoint` to

```
#define QUEUE_SIZE 10
struct stack { int *base, *sp, size; };
struct queue { struct stack *front, *back; };

struct stack* initStack(int sz)
{ struct stack* s = (struct stack*) malloc(sizeof(struct stack));
  s->base = s->sp = (int*)malloc(sz * (sizeof(int)));
  s->size = sz;
  return s; }

struct queue* initQ()
{ struct queue* q = (struct queue*) malloc(sizeof(struct queue));
  q->front = initStack(QUEUE_SIZE);
  q->back  = initStack(QUEUE_SIZE);
  return q; }

int isEmptyS(struct stack* s)
{ return (s->sp == s->base); }

int isEmptyQ(struct queue* q)
{ return (isEmptyS(q->front) && isEmptyS(q->back)); }

void push(struct stack* s, int i)
{ *(s->sp) = i; s->sp++; }    /* no overflow check */

void enq(struct queue* q, int i)
{ push(q->front, i); }

int pop(struct stack* s)
{ if (isEmptyS(s)) return -1;
  s->sp--;
  return (*(s->sp)); }

int deq(struct queue* q)
{ if (isEmptyQ(q)) return -1;
  if (isEmptyS(q->back))
    while(!isEmptyS(q->front)) push(q->back, pop(q->front));
  return pop(q->back); }
```

Figure 4: A queue using two stacks in C.

```
const int QUEUE_SIZE = 10;

class stack {
private:
  int* base;
  int* sp;
  int size;
public:
  stack(int sz) {
    base = sp = new int[sz];
    size = sz; }
  int isEmpty() {
    return (sp == base); }
  int pop() {
    if (isEmpty()) return -1;
    sp--;
    return (*sp); }
  void push(int i) {
    // no overflow check
    *sp = i; sp++; }
};

class queue {
private:
  stack *front, *back;
public:
  queue() {
    front = new stack(QUEUE_SIZE);
    back = new stack(QUEUE_SIZE); }
  int isEmpty() { return (front->isEmpty() && back->isEmpty()); }
  int deq() {
    if (isEmpty()) return -1;
    if (back->isEmpty())
      while(!front->isEmpty()) back->push(front->pop());
    return back->pop(); }
  void enq(int i) { front->push(i); }
};
```

Figure 5: Queue and stack classes in C++.

```
typedef struct {
  int x,y;
} Point;

typedef enum { RED, BLUE } color;

typedef struct {
  int x,y;
  color c;
} ColorPoint;

void translateX(Point *p, int dx)
{
  p->x += dx;
}
```

Figure 6: A simple example of subtypes in C: `ColorPoint` can be thought of as a subtype as `Point`.

be a pointer to a `Point` as in:

```
translateX((Point *)(&cp), 1)
```

Because of the way values of the types `Point` and `ColorPoint` are laid out in memory, the cast of actual parameter `&cp` in this call on `translateX` causes one type (i.e., `ColorPoint`) to be treated as a *subtype* of another type (i.e., `Point`).

In Chapter 4, we make precise the notion of physical subtyping in C and describe an algorithm by which physical subtypes can be recognized automatically. This information can then be used to generate inheritance hierarchies in C++ (with subtypes corresponding to subclasses). For example, the C code shown in Figure 6 might correspond to the C++ code shown in Figure 7.

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

```
class Point {
protected:
  int x,y;
public:
  void translateX(int dx) {
    x += dx;
  }
};

typedef enum { RED, BLUE } color;

class ColorPoint : public Point {
protected:
  color c;
};
```

Figure 7: A simple example of subclasses in C++: subclass `ColorPoint` is derived from base class `Point`.

- We lay the theoretical foundations for several new approaches to software renovation:
  - application of polymorphic type inference to C-to-C++ conversion
  - application of concept analysis to the modularization problem
  - definition of concept partition and its application to modularization
  - definitions of physical subtyping and physical type safety
  - application of physical subtyping to program comprehension and type checking
- We develop algorithms for:
  - transforming C functions to C++ function templates

  - generating module partitions from a concept lattice
  - discovering physical-subtype relationships in C programs

- We have developed prototype tools for:

  - automatically generalizing C functions to C++ function templates
  - applying concept analysis to identify potential modules in C programs
  - identifying physical subtypes and checking physical type safety

  We have applied these tools to several large benchmarks, and the results of these studies are reported upon in the thesis.

1.1 Related Work

Before proceeding to the core of the thesis, it is worthwhile to describe how the contributions of this work relate to three subareas of software engineering and programming languages that have been studied previously in great depth, specifically software reuse, software comprehension, and type theory. Although the primary thrust of the thesis is software renovation as applied to the C-to-C++ conversion problem, the machinery developed in the thesis both makes use of results from, as well as makes contributions to, each of the three areas. This section articulates some of these relationships.

1.1.1 Software Reuse

In general, reuse is the idea of applying that which is already at hand—whether it be physical resources or conceptual resources—to avoid duplication of effort over time and space [20]. Software reuse enables software engineers to save time (and often space) by using program components that are already implemented [5, 6]. Furthermore, using components and algorithms that have been tested before and are well understood not only save time in the present, but also save time *later* as there is less need to debug.

There are several ways in which to achieve effective software reuse. Probably the best way to attain high levels of reuse is to *design for reuse* [20]. However, it is too late to rebuild and redesign from the ground up when dealing with large legacy systems. One of the goals of my work on software renovation has been to make legacy code reusable. The three renovation techniques described in this thesis can aid in the recovery of reusable components from legacy code in several ways, including the following:

- Generalized program components promote reusability since they can be used in a wider variety of contexts.
- Encapsulated code has fewer dependencies on other code and therefore can be reused in other contexts more easily.
- Physical subtypes can correspond to subclasses—this promotes reuse through inheritance.

1.1.2 Program Comprehension

The work in this thesis may also be contrasted with research in the field of program comprehension. In particular, this work may be contrasted with the problem of *cliché* recognition [55, 66]. Clichés are widely used algorithms and data structures (such as stacks and stack operations) as well as commonly employed computational idioms such as iteration and filtering. Much of the work on cliché recognition (and its cousin *plan* recognition) has been based on searching libraries of commonly used structures. The work in this thesis is independent of such a priori knowledge about how programs are written; it may be useful in its own right as a program-comprehension tool for building initial libraries of clichés. The software-renovation techniques described in the thesis can aid in program comprehension in several ways, including the following:

- Generalization results in program components that are often instances of or related to *container* structures, such as lists, stacks, and queues.

- The modularization process results in collections of related program components which by their very nature—smaller and less tangled with other components—may be easier to recognize as instances of plans or clichés.

- Analysis of physical subtypes can help the software engineer recognize type hierarchies and object-oriented idioms. The application of physical subtypes to program comprehension is discussed further in Chapter 4.

1.1.3 Type Theory

Much of the research presented in this thesis has its foundation in type theory, particularly in the area of type inference. The most significant links between the software-renovation techniques presented here and type theory are as follows:

- One of the central ideas behind the generalization process described in Chapter 2 is to impose a parametric polymorphic type system on C programs in place of

C's traditional monomorphic types. The type-inference algorithm is based on Hindley-Milner type inference [31, 47], but with the augmentations specific to the C-to-C++ conversion problem: operator overloading, constructors, casts, etc.

The idea of mixing polymorphism with C appears in several places, among them [59, 50]. [59] concerns a new dialect of C that is polymorphic and type safe. This differs from our approach in that it is not aimed at adding polymorphism to existing code. [50] uses polymorphic type inference on existing C programs, but for determining information about the transfer of values, as opposed to producing reusable code.

- The modularization process as applied to the C-to-C++ conversion problem can be thought of as a type-inference problem: how to infer classes in C program. However, this departs from traditional type-inference problems in that there are almost always multiple different classes that can be inferred from the same code and thus there is no notion of a *principal* type. The machinery that we develop in Chapter 3 is also very different from the machinery used in traditional work on type inference. It is based on *concept analysis*, a branch of lattice theory that can be used to identify similarities among a set of *objects* based on their *attributes*. Our work compares favorably with contemporaneous work on on concept analysis and modularization by Lindig and Snelting [40] and Sahraoui et al. [57]. (See Section 3.7.)

- Physical-subtype detection is also based on the imposition of a new type system on C programs. This type systems differs from the one employed in the generalization problem by allowing subtype polymorphism instead of parametric polymorphism. The type system focuses on the storage representation of aggregate data types and in that sense is similar in spirit to the work on pointer aliasing in the presence of `struct` and `union` types discussed in [61, 70].

1.2 Organization of the Thesis

The remaining chapters of this thesis are organized as follows:

In Chapter 2, we consider the problem of identifying functions that can be generalized to C++ function templates that can operate on a wide variety of types. We describe an implementation of our algorithm and present results of applying it on several large benchmarks. We discuss the problem of transforming C++ classes into C++ function templates. Both techniques rely on a type-inference algorithm, which is explained in detail therein. We also show how a similar technique can assist in the generalization of Standard ML programs.

In Chapter 3, we consider the problem of identifying modules in legacy code by using concept analysis—a branch of lattice theory that can be used to identify similarities among a set of *objects* based on their *attributes*. We define the notion of a concept partition and present an algorithm for discovering the possible partitions of a concept lattice. We discuss an implementation and present results. We illustrate how concept analysis might be applied to related problems in software architecture—such as understanding how modules interact—by applying concept analysis to a collection of Standard ML functors.

In Chapter 4, we describe an algorithm for identifying implicit subtype relationships among C data types. Specifically we define the notion of a physical subtype and use this definition to devise a type-safety analysis for cast expressions in C programs. We also discuss how object-oriented idioms can be discovered in C program and how they can seed the process of transforming legacy C code into C++ code that makes use of inheritance and virtual functions.

Chapter 5 presents some conclusions.

Chapter 2

Generalization

2.1 Introduction

"Software reuse" has often been touted as the key to improving both the quality of software and the productivity of software engineers. Reuse can take many forms—reuse of specifications, designs, architecture, and code. However, reuse in any of these forms has, at best, only partially lived up to its promise.

This chapter focuses on a form of *code-oriented* software reuse. A difficulty with achieving practical code-oriented software reuse is that since each new application has slightly different requirements, existing code is often too specific to be efficiently adapted. Thus, the premise of this work is that to support more effective code reuse what is needed are tools to aid programmers with the task of adapting code to new contexts.

One possible method for supporting software reuse is *program generalization*. Generalization is a transformation of a program component C into a parameterized program component C' such that C' is usable in a wider variety of syntactic contexts than C. Furthermore, C' should be a semantically meaningful generalization of C; namely, there must exist an instantiation of C' that is equivalent in functionality to C.

In the following sections, an algorithm is presented that discovers possibilities for reuse in C functions: Given a collection of C functions that operate on arguments of specific data types, the result of generalization is a collection of C++ function templates that operate on arguments that have parameterized types. The main idea behind the generalization algorithm is the use of type inference to discover possibilities for reuse by identifying code that is "type independent". To identify type-independent code, we discard the standard C type system and replace it with a parametric polymorphic type system. Although the types inferred by this system are "unsound" for the original C functions—after all, C has no notion of polymorphic type—the inferred types indicate what variables of the functions can have their types "lifted" to depend on function-template arguments. This exploits the fact that C++ performs type-checking at template-instantiation time and does not permit a template to be instantiated with a parameter of an inappropriate type.

The principles discussed in this chapter can, for the most part, be applied to any programming language. We focus on a method by which C programs can be generalized to "templatized" C++ programs. The choice to concentrate on generalization from C to C++ is motivated by several factors:

- *C++ genericity features.* While C's lack of polymorphism makes it difficult to specify reusable code, C++'s templates, class constructors, and operator overloading make it suitable for creating reusable code.

- *Practical needs.* The conversion of existing C code to C++ is a practical problem of current interest and great economic importance. Although the C-to-C++ conversion problem is eased by the fact that C++ is (essentially) an upwards-compatible extension of C, there are still interesting issues that arise in such a conversion process, in particular, how to discover places in the code where the improved features of C++ can be exploited. A particularly important instance of this involves C++ templates: It would be desirable to have a tool that converts existing C code to "templatized" C++. The generalization technique described herein offers a way to introduce templates, classes, and constructors when C programs are ported to C++.

- *Ease of transformation.* The final step involved in creating the generalized C++ code is a straightforward syntax-directed translation. The transformation need only make changes to certain portions of the C code being generalized: some base types are replaced by template-argument types; constructors are inserted in certain assignment expressions, etc. The remainder of the code is unchanged.

The program-generalization problem can be characterized as follows:

> A *generalization* of a program component C in language L is a parameterized program component C' in language L' such that C' can be instantiated to an L'-program component that is equivalent in meaning to C.

For instance, in Sections 2.2.2 and 2.5, we discuss the integer-exponentiation function shown in Figure 8 and show how it can be generalized to the exponentiation template shown in Figure 9, in which a repeated "multiplication" step is used to perform an "exponentiation" operation over domains other than the integers. In this example, L is C, L' is C++, component C is the function shown in Figure 8, and C' is the function template shown in Figure 9.[1]

[1]We actually depart slightly from the above definition of the generalization problem. For example,

```
int power(int base, int n)
{
  int p;

  p = 1;
  while(n > 0) {
    p = p * base;
    n--;
  }
  return p;
}
```

Figure 8: The `power` function.

```
template<class T> T power(T base, int n)
{
  T p;

  p = T(1);
  while(n > 0) {
    p = p * base;
    n--;
  }
  return p;
}
```

Figure 9: A C++ `power` function template.

The main contribution of this chapter is an algorithm that transforms C functions to C++ function templates. The algorithm has been implemented and tested on a variety of C programs. The chapter describes the generalization algorithm, as well as the type-inference system for C on which it is based.

---

with Figures 8 and 9, template C' can be instantiated to C by mapping template argument `T` to type `int`. However, in C++, instantiation of function templates is a "hidden operation." It is carried out automatically by the compiler without an explicit directive furnished by the user. In contrast, because class templates are instantiated explicitly, a generalization algorithm that created class templates would match the above model exactly.

Section 2.2 discusses the opportunities for generalization that the algorithm is capable of identifying. Section 2.3 discusses the ways in which the generalization algorithm makes use of the declarations in the functions being generalized. Section 2.4 addresses the problem of over-generalization and describes some of the choices made in the generalization algorithm to try to avoid creating overly general templates. Section 2.5 presents the function-generalization algorithm and illustrates it on the `power` example. Section 2.6 discusses the implementation of the algorithm. Section 2.7 discusses some limitations of our generalization algorithm. Section 2.8 extends the generalization process to apply to C++ classes. Section 2.9 discusses the generalization process applied to Standard ML programs. Section 2.10 concerns related work.

2.2 Sources of Polymorphism

The basic idea behind the program-generalization algorithm is as follows:

> We discard the standard C type system and replace it with a parametric polymorphic type system. Type inference is carried out in a relatively standard fashion [47], and then generalization is performed by mapping free type variables in the resulting type expressions to template parameters.

As mentioned in the introduction, the types inferred during this process are unsound with respect to C. However, our goal is not to determine types for the original C code but to *generalize* the code: We wish to determine which variables in the C code are polymorphic and can therefore have their types "lifted" to depend on function-template arguments.

The function-generalization algorithm takes advantage of three main sources of polymorphism:

- *Parametric polymorphism via operator overloading.* C does not support type-safe polymorphism. (Although, a limited form of polymorphism can be achieved through the use of void pointers, it is obtained at the loss of type safety.) In order to have a "looser" type system for C, the standard C operators are treated by our system as operators with polymorphic types (i.e., universally quantified, or "generic", types). This takes advantage of the fact that the goal of function generalization is to produce a function template in C++, which is a language that supports operator overloading. By "freeing" the standard C operators from their monomorphic types, we are (sometimes) able to assign user-defined functions polymorphic types. (The type system can also deduce constraints that a certain template argument must be of a type that is equipped with a particular overloaded operator.)

- *Constructor introduction.* In keeping with the need for a "looser" type system for C, assignments of the form $x = c$, where c is a constant, are given special treatment. Such statements are treated as an opportunity to introduce a constructor, turning the statement into $x = T(c)$. The idea is that assignments of the form $x = c$ indicate that x should be given a value that is *based* on c, rather than a strict requirement that the value *be* c.

- *Subtyping relationships between `struct` types.* Again, in keeping with the need for a "looser" type system for C, the standard notion of subtype between records is adopted for `struct`s (i.e., a subtype of a `struct` type s has all the fields of s

and possibly more)[2]. For the C-to-C++ function-generalization problem, this is needed in order to identify opportunities to create function templates with `struct` arguments. (A C++ function template can be called with `struct` arguments that contain more fields than just those accessed from within the template's body.)

These sources of polymorphism are discussed in greater detail in the remainder of this section. Although all three are related to issues that have been examined in previous studies of type inference, there are various details that concern the application of type inference to the problem of program generalization. (Some related issues, concerning differences between our approach to type inference and the ways in which type inference has been traditionally formulated [31, 47, 48], are discussed in Section 2.3.)

2.2.1 Parametric Polymorphism Via Operator Overloading

Parametric polymorphism captures certain kinds of commonalities among similar operations on different types. This is a powerful mechanism for code-oriented software reuse. In C, however, although a limited form of polymorphism can be achieved through the use of void pointers, this is obtained at the loss of type safety.

Our approach to the function-generalization problem exploits the fact that the goal of function generalization is to produce a function template in C++, which is a language that supports operator overloading. Although operator overloading is ordinarily synonymous with *ad hoc* polymorphism, in our work we make use of the C++ features that support *ad hoc* polymorphism in a disciplined way: Operator overloading

[2] By "standard notion of subtype between records" we refer to the subtyping between records as discussed by Cardelli in [12, 13]. This is a substantially different form of subtyping than *physical subtyping*, which is the subject of Chapter 4. A discussion between the differences of these two kinds of subtyping can be found on page 174.

is one of the mechanisms we use for expressing the commonalities deduced from the original C code via a *parametric* polymorphic type system. In particular, our system treats the standard C operators as operators with polymorphic types (i.e., universally quantified, or "generic", types). By "freeing" the standard C operators from their (mostly) monomorphic types, we are sometimes able to assign user-defined functions polymorphic types (which in turn allows us to generalize the functions to C++ function templates).

As an example, consider the code in Figure 10 to sort an array of integers. An

```
void sort(int *list, int size)
{
  int i, j, tmp;

  for(i = 0; i < size; i++) {
    for(j = i + 1; j < size; j++) {
      if (list[j] < list[i]) {
        tmp = list[j];
        list[j] = list[i];
        list[i] = tmp;
      }
    }
  }
}
```

Figure 10: A function to sort an array of integers.

examination of the code reveals that there is only one feature, other than the declarations, that makes it specialized for sorting integers; namely, the `<` operator expects the values it receives when it compares two elements of the array to be of type `int`. Ideally, we would like the function-generalization algorithm to report that this function can be generalized to the C++ function template shown in Figure 11.

The type system treats the standard C operators as operators with polymorphic

```
template <class T>
void sort(T *list, int size)
{
  int i, j;
  T tmp;

  for(i = 0; i < size; i++) {
    for(j = i + 1; j < size; j++) {
      if (list[j] < list[i]) {
        tmp = list[j];
        list[j] = list[i];
        list[i] = tmp;
      }
    }
  }
}
```

Figure 11: A function template for `sort`.

types. In this case, `<` has type $\forall\alpha.\alpha \times \alpha \rightarrow \iota$.[3] At each occurrence of a standard operator, the generic type is instantiated in the usual way [47]; that is, the quantifiers are stripped off, and the body of the type is instantiated with fresh type variables different from all other type variables used elsewhere. Unification of type expressions allows the system to deduce how certain types are related to other types. For instance, the expression `i < size` causes the generic type $\forall\alpha.\alpha \times \alpha \rightarrow \iota$ to be instantiated to, say, $\beta \times \beta \rightarrow \iota$, and unification deduces that `i` and `size` must have the same type; however, `i` and `size` are not required to have a specific monomorphic type, such as `int`. (Because `i` is used in an array-index expression in the `sort` function (i.e., `list[i]`), as analysis of the `sort` function progresses, `i` and `size` are ultimately discovered to be of type ι.)

The type system can also deduce constraints that a certain template argument

[3] The symbol ι denotes "monomorphic type". For the moment, think of ι as `int`; ι is discussed further in Section 2.3.

must be of a class that is equipped with a particular overloaded operator. In the case of Figure 11, class `T` must have a `<` operator. This captures the notion that the sort algorithm requires only that a comparison operator `<` be defined for the type of the array's elements. (It also exploits the fact that C++ performs type-checking at template-instantiation time; the C++ compiler will not permit the template to be instantiated unless the class is equipped with an appropriate overloaded `<` operator.)

2.2.2 Constructor Introduction

The type system gives special treatment to assignments of the form $x = c$, where c is a constant. This is motivated by the fact that using just operator overloading and `struct` subtyping as the basis for inferring types does not yield a powerful enough generalization algorithm. In particular, a reason why many functions cannot be adequately generalized using such a type-inference system is because they contain expressions that assign numerical constants to variables.

To understand the issue, consider the `power` function shown in Figure 8 (see page 20). The function takes two integer arguments, `base` and `n`. The result is `base` raised to the `n`th power. The same repeated-"multiplication" algorithm could be used to perform an "exponentiation" operation over domains other than the integers. Not only are floating-point numbers a possibility, but so are more complex data types, such as matrices and complex numbers.

Unfortunately, if generalization is based solely on traditional type-inference methods, we are unable to generalize `power`. Traditional type-inference systems would use the expression `p = 1` as grounds for deducing (via unification) that `p`'s type is `int` (and the expression `n > 0` as grounds for deducing that `n`'s type is `int`). The upshot is that `power`, and functions like it, would not be generalized using a type-inference system based solely on operator overloading and `struct` subtyping.

For this example, a more desirable outcome would be for generalization to produce a function template that works on bases of any type that supports a "multiplication" operator (where "multiplication" does not necessarily have to be a numeric operation) and that have a suitable "unit" value (i.e., an element corresponding to `1`).

Our approach is to introduce some additional flexibility into the way type inference is performed for assignment expressions: If the value being assigned (i.e., the right-hand side) is a constant expression, then a constructor of a C++ class can be introduced. In essence, this says that the variable being assigned to (i.e., the left-hand side) is of a type that is either the originally declared type (in which case the constructor may be thought of as the identity function) or of a class that has a constructor that maps c to some value of the class. This approach captures the notion that assignment statements of the form $x = c$ indicate that x should be given an initial value that has some value *based* on c rather than a strict requirement that the value *be* c.

Figure 9 (page 20) shows a C++ function template derived from the `power` function via generalization.

In the case that assignment statements are of a more complicated form, such as $x = e$, where e is not a constant expression, constructors are not introduced; instead the types of the left-hand and right-hand sides are constrained to be the same. Constructors could be introduced here, but this might cause *over-generalization.* That is, if too few type constraints are imposed by assignments, then almost every argument of a function would be generalized into a template argument, and the resulting template function is likely to be incomprehensible. (Over-generalization is discussed further in Section 2.4.)

2.2.3 `Struct` Subtyping

`Struct` subtyping allows us to generalize functions that deal with container `struct`s such as sets, stacks, and queues. We adopt the standard notion of subtype between C `struct` types (based on the presentation of record subtyping in [13])[4] : The subtype relation is the trivial relation (i.e., t is a subtype of t' if and only if t and t' are the same type) for all types except `struct`s;[5] a `struct` type s is a subtype of another `struct` type s' if, for every field l of type t' in s', l is a field of type t in s and t is a subtype of t'. Type s may contain additional fields that do not occur in s' and still be a subtype of s'. Thus, an instance of s has at least as much information, and perhaps more information, than an instance of s'. An s value can always be thought of as an s' value by projecting on the common fields.

Consider the following linked-list `struct`:

```
struct IntList {
  int i;
  struct IntList *next;
};
```

Now consider a function that, given a `struct IntList *`, returns the value of the `i` field of the next element in the list (ignoring checking for null pointers):

[4]This definition is contrasted with physical subtyping on page 174. See footnote on page 23 for more details.

[5]As mentioned earlier, and discussed in detail in Section 2.3, the type system uses a single symbol ι to represent monomorphic types, and thus it need not consider any of the various arithmetic types (i.e. `char, int, long, double`, etc.) to be related as subtypes of one another.

```
int getNextVal(struct IntList *node)
{ return (node->next->i); }
```

In C++, a function template that has a parameter with a `struct` type can be called with `struct` arguments that contain more fields than just those accessed from within the template's body. The C++ compiler uses a `struct`-subtyping rule at template-instantiation time when it checks whether a template is being instantiated with arguments of the appropriate types. Thus, to be able to identify opportunities to create such function templates, the function generalizer also needs to use `struct` subtyping.

By using `struct` subtyping, the generalizer deduces for the example code given above that the argument type can be a pointer to any `struct` that has a `next` field and an `i` field. Function `getNextVal` is then generalized to the function template shown below:

```
template <class T>
int getNextVal(T *node)
{ return (node->next->i); }
```

Implicit in the template is that `T` is a `struct` that has a `next` field that is a pointer to a `struct` that has an `i` field of type `int`. Because `struct IntList` is a subtype of the (implicit) `struct` type, a `struct IntList *` can be used as an argument to `getNextVal`.

2.3 Polymorphism and the "Valid-Code Assumption"

Because in our context generalization involves *two* typed languages, and a translation from one to the other, the goal of type inference in our work is somewhat different from the usual one. Ordinarily, type inference is treated as a problem of showing that type annotations are completely superfluous. Specifically, many type-inference problems can be cast in the following framework, in which a typed language is related to an untyped language [31, 47, 48]:

> Suppose L is a typed language, L' is a related untyped language, and "erasure" function Erase : $L \to L'$ removes type annotations from terms of L. Given L, L', Erase, and a term $t' \in L'$, the type-inference problem is the problem of discovering a term $t \in L$ such that $\text{Erase}(t) = t'$.

In other words, type inference is traditionally a problem of recovering types when all information in the declarations is ignored.

However, for a language like C, in order to distinguish among multiple uses of the same identifier in different scopes, a type-inference system *does* need to consider the declarations. In addition, it needs to examine the declarations to obtain information that cannot be obtained in any other way, such as storage-class, type-qualifier, and arithmetic-precision information, which in general cannot be determined by context. Consequently, in contrast to the way type inference is traditionally formulated, our type-inference algorithm relies on what we will call the "Valid-Code" Assumption:

> We assume that the input files containing the program fragment to be

generalized compile without error according to the ANSI standard.

The Valid-Code Assumption changes the character of the type-inference problem somewhat. In particular, *the type-inference algorithm need only be concerned with the question of whether an expression has monomorphic type or polymorphic type.* Because inconsistencies between monomorphic types have already been checked for by the C compiler, there is no need for them to be rechecked by the type-inference algorithm (and there is also no need for the actual monomorphic types to be tracked during type inference). The Valid-Code Assumption also allows us to ignore issues about implicit type conversions and promotions that can occur among arithmetic types [38, pages 197–202]. For this reason, the type system uses a single symbol, ι, to represent monomorphic types. (Most other type-inference systems have collection of different monomorphic types, e.g., `int`, `float`, `int` $\rightarrow$ `float`, etc.)

Type ι is the one "base type" of the type system. In most polymorphic type systems, all base types are non-functional types (e.g., `int`, `float`, etc.). In contrast, in our type system ι also represents monomorphic functional types (e.g., `int` $\rightarrow$ `int`, `int` $\rightarrow$ `float`, etc.).

The Valid-Code Assumption has two important consequences:

- After type inference has been performed, any expression whose type is ι can be given the type that the C compiler would have assigned to the expression. For example, suppose that f is a function declared to be of type `int` $\rightarrow$ `float`. The type-inference algorithm might deduce that f has a polymorphic type, say $\forall \alpha\beta.\alpha \rightarrow \beta$; however, if it deduces instead that f has type ι, the generalizer treats f as having the functional type `int` $\rightarrow$ `float`.

- It is always safe for the type system to fall back on ι because we know that the C compiler was able to assign *some* type to each subexpression.

The Valid-Code Assumption also helps with some other issues that arise in performing type inference on C programs:

- Without using type information from declarations, it would be difficult to resolve the types of operators that are overloaded in C. In particular, pointer arithmetic poses a problem. For example, the expression `x + y` could refer to pointer addition or numeric addition. In the case of pointer addition, the result and exactly one of the arguments should be pointers, and the other argument should be `int`. In the case of numeric addition, the result and both arguments should be of numeric types. In this situation, the part of the system that generates initial type assignments for the type-inference algorithm consults the original declarations of `x` and `y`: If `x` is a pointer and `y` numeric, then `+` is treated as $\forall\alpha.\alpha \times \iota \rightarrow \alpha$; if `y` is a pointer and `x` numeric, then `+` is treated as $\forall\alpha.\iota \times \alpha \rightarrow \alpha$; if `x` and `y` are both numeric, then `+` is treated as $\forall\alpha.\alpha \times \alpha \rightarrow \alpha$.

- The Valid-Code Assumption also helps us to deal with a few quirks of the C++ function-template mechanism. The issue is that inferred types are sometimes more general than the C++ template mechanism can handle. Because of the Valid-Code Assumption, in these cases (which all involve local variables and function return types), we can assign these entities the types they had in the original C code.

  This issue is illustrated by the `getNextVal` example from Section 2.2.3. Type inference deduces that the type of the `i` field of `T` is polymorphic. Unfortunately, due to the limitations of the C++ function-template mechanism, we cannot generalize `getNextVal` to have a template parameter that stands for the return type. In particular, C++ function templates are subject to the following restriction:

> Each template argument must affect the type of at least one of the function arguments [62, page 280].

Nor does C++ offer any way of expressing "the type of the `i` field in class T". We must therefore design the generalization system so that in cases when the return type of a function is determined to be a polymorphic type that is independent of the types inferred for the function arguments, the return type in the function template is restricted to be the return type given in the original declaration.

Similarly, type inference may sometimes deduce that local variables have polymorphic types that are independent of the function's argument types. Again, because of the restriction cited above, the generalizer cannot give such variables polymorphic types. Instead, the generalization algorithm restricts the types of these local variables to the types they had in their original declarations.

The inability to handle polymorphic return types is a limitation that, in some cases, can hamper the ability of our algorithm to create appropriate generalizations of functions. The inability to handle polymorphic local variables is less of a limitation: Local variables whose types are independent of the argument types are often an indication that the variables are being misused or perhaps not used at all.

Some of these problems disappear when our techniques are extended to handle a related generalization problem, that of creating class templates rather than function templates. For example, it is not a problem to have a member function whose return type is both polymorphic and independent of the types inferred for the function's arguments: The return type can be an additional argument of the class template. The topic of generalizing programs to create class templates is discussed in Section 2.8.

2.4 Restraints on Polymorphism

This section concerns the problem of over-generalization, and describes some of the choices made in the generalization algorithm to try to avoid creating overly general templates. These choices are aimed at introducing *restraints* on the amount of polymorphism that is identified. If too many opportunities are provided for generalization, then almost every argument of a function will be generalized into a template argument, which has the danger that the results will be difficult to understand. The chief restraints on polymorphism that we impose are as follows:

- Constructor introduction is limited to constants occurring alone on the right-hand side of assignment expressions.
- Unary operators are given either the type $\forall\alpha.\alpha \rightarrow \alpha$ or $\iota \rightarrow \iota$, rather than $\forall\alpha\beta.\alpha \rightarrow \beta$.
- Binary operators are given either the type $\iota \times \iota \rightarrow \iota$, $\forall\alpha.\alpha \times \alpha \rightarrow \iota$, or $\forall\alpha.\alpha \times \alpha \rightarrow \alpha$, rather than $\forall\alpha\beta.\alpha \times \beta \rightarrow \alpha$, $\forall\alpha\beta.\alpha \times \beta \rightarrow \beta$, or $\forall\alpha\beta\gamma.\alpha \times \beta \rightarrow \gamma$.

The decision to limit constructor introduction to constants occurring alone on the right-hand side of assignment expressions is, admittedly, somewhat arbitrary. Our motivation is that there are expressions in which we do not wish to introduce constructors, such as `n > 0` in `power`. It is apparent that some kind of rule to limit constructor introduction is needed. For example, it is unclear whether the generalization of `x + (1 + 2)` should be `x + (T(1) + T(2))`, `x + T(1 + 2)`, or `x + T(U(1) + U(2))`. To prevent such ambiguities, we conservatively limit constructor introduction to just the one kind of context.

We assign the following types to the standard C operators:

1. *Comparison operators*: `>`, `<`, `==`, `!=`, `>=`, `<=` are given the type: $\forall\alpha.\alpha \times \alpha \to \iota$. The decision that the result type of comparison operators is ι was motivated by the fact these expressions are often used as "booleans" in control expressions of `if`, `while` and `for` statements. The decision to restrict these binary operators to operands of a single type variable stemmed from the desire to capture the constraint that like quantities be compared. It is also an effective way to prevent over-generalization. For example, in `power`, `n` is constrained to be of type `int` because it is compared with `0`.

2. *Binary logical operators*: `&&` and `||` are given the type $\iota \times \iota \to \iota$. The motivation behind the restraints on the type of the logical operators is similar to that for the restraint on the result type of comparison operators. Logical operators are frequently used in control expressions and their operands are often the results of comparisons.

3. *Binary arithmetic operators*: `*`, `+`, `-`, `/`, `%`, `|`, `&`, `^`, `<<`, `>>` are given the type $\forall\alpha.\alpha \times \alpha \to \alpha$. The decision to restrict these binary operators to a type quantified over a *single* type variable was motivated by the desire for their generalizations to have the same type homogeneity that the operators have in C (where

```
int power(int base, int n)
{
  int i, p;

  p = 1;
  for(i = 1; i <= n; i++)
    p = p * base;
  return p;
}
```

Figure 12: The `power` function with two local variables.

they have the type, numeric $\times$ numeric $\rightarrow$ numeric).

4. *Unary operators*: `++`, `--`, `-`, `~` are given the type: $\forall\alpha.\alpha \rightarrow \alpha$. The unary logical operator ! is given the type $\iota \rightarrow \iota$. The address-of operator `&` and the dereference operator `*` are given the types, $\forall\alpha.\alpha \rightarrow \alpha$ `ptr` and $\forall\alpha.\alpha$ `ptr` $\rightarrow \alpha$, respectively.

These measures all help to prevent the creation of overly general templates. However, it is still possible for over-generalization to occur. For instance, consider the version of the `power` function shown in Figure 12 [38, page 25]. This version is equivalent to the function given in Figure 8, but has a second local variable, `i`, which is used as the iteration variable in the loop. Because the comparator `<=` is applied to two *variables* in this version (namely, `i` and `n`), rather than a variable and a *constant*, as in Figure 8, the generalization of Figure 12 results in a template with two type parameters, as shown in Figure 13.

When instantiated with integer arguments, this template behaves as expected. However, the second template parameter appears to be superfluous. The disadvantage of having `i` and `n` be of type other than `int` is that it makes it harder to understand what the function template accomplishes, and it is not clear that there are any

```
template <class T0, class T1>
power(T0 base, T1 n)
{
  T1 i;
  T0 p;

  p = T0(1);
  for(i = T1(1); i <= n; i++)
    p = p * base;
  return p;
}
```

Figure 13: A C++ template for `power` function with two local variables.

offsetting advantages to having this general a template.

This kind of problem could be rather serious when attempting to create libraries of templates using generalization. If function templates have too many parameters, it may become difficult to understand how they are intended to be used. One way to avoid over-generalization would be to allow the user to supply directives to "anchor" a variable's type. For example, the parameter `n` in `power` might be declared as `$ANCHOR int n`, resulting in a one-parameter template that has the header `template <class T> power(T base, int n)`.

2.5 An Algorithm to Generalize C Functions

This section concerns the algorithm for function generalization. The steps of the algorithm are depicted in Figure 14.

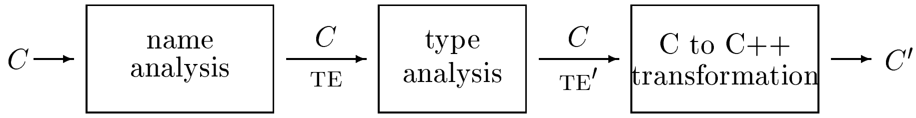

Figure 14: In this process, TE and TE′ are type environments—mappings of program variables to type expressions. The input C need not be a complete C program; it can be any number of C functions contained in files. All we assume is that C's files compile without error according to the ANSI standard.

2.5.1 Name Analysis and Initial Type Assignment

After the program fragment's abstract syntax tree is constructed, it is first subjected to *name analysis:* Each name used in the program fragment is resolved to the appropriate declaration. A type environment is then created in which each name is assigned an initial type. Each declared variable is assigned a unique type variable. Functions declared without definition (function prototypes) are assigned their declared types. User-defined functions are assigned polymorphic function types, quantified over the type variables occurring in their argument and return types. The type environment produced for the `power` example is shown below:

$$
\begin{array}{ll}
[\texttt{power:} & \forall \alpha_0, \alpha_1, \alpha_2 . \alpha_1 \times \alpha_2 \to \alpha_0\texttt{,} \\
\ \texttt{base:} & \alpha_1\texttt{,} \\
\ \texttt{n:} & \alpha_2\texttt{,} \\
\ \texttt{p:} & \alpha_3]
\end{array}
$$

2.5.2 Type Analysis

Type inference is employed to determine the relationships among the type variables introduced during name analysis. The type-inference system involves satisfying constraints among types and type variables. As discussed in Section 2.3, the Valid-Code Assumption simplifies the type-inference task in the following ways:

- There is only one monomorphic type in the type system, denoted by ι, and, because we know that the C compiler was able to assign *some* type to each subexpression, it is always safe for the type system to fall back on ι.

- After type inference has been performed, any expression whose type is ι can be given the type that the C compiler would have assigned to the expression.

The goal of the type-inference phase of function generalization is to infer appropriate types for every function in the input, some of which may be polymorphic. The type-inference algorithm is a "worklist" algorithm. A function stays on the the worklist until a most-precise type (i.e., most polymorphic type, consistent with the context given by the rest of the program component) has been inferred. A sketch of the algorithm is as follows:

1. Put every function on the worklist.

2. Until the worklist is empty or an entire round of processing is completed that does not introduce any changes[6], examine each function on the worklist in round-robin fashion and perform the following steps:

[6] See Section 2.7.2 for an explanation of what we mean by "changes".

(a) Solve the type constraints of the body of f to sharpen the estimated types for the names used in f.

(b) If f makes use of any function (including itself) that is on the worklist, keep f on the worklist (at the "tail"). Otherwise, remove f from the worklist.

3. For each function f:

   (a) If f's return type has been inferred to be polymorphic over one or more type variables that do not occur in the types inferred for f's arguments, then constrain the return type to be ι.

   (b) For each local variable x of f, if x's type has been inferred to be polymorphic over one or more type variables that do not occur in the types inferred for f's arguments, then constrain x's type to be ι.

4. If any constraints were added in Step 3, reinitialize the worklist with all of the functions, and perform an additional phase of round-robin iteration (as in Step 2).

Initially, every function is placed on the worklist. When the algorithm enters the body of function f, f is removed from the worklist. If, in processing the body of f, it is determined that f's type is dependent upon the type of a function g that has "incomplete" type (i.e., g is on the worklist) then f is put back on the worklist. The algorithm proceeds until either the worklist is empty or, after one complete iteration, the worklist and the types of all functions on the worklist remain the same[7]. This

[7] See Section 2.7.2 for an explanation of what we mean by "same".

allows type inference to be carried out on program components that include forward references and mutual recursion.

Section 2.7.2 explains why the algorithm terminates in polynomial time.

The type system

Type expressions are of the form:

$$\begin{aligned} \tau \quad ::= \quad & \iota|\alpha|\tau\ \texttt{ptr}|(\tau_1,\ldots,\tau_n)\rightarrow\tau| \\ & [\alpha]\{l_1:\tau_1,\ldots,l_k:\tau_k\}|\texttt{Typedef}(\nu,\tau) \end{aligned}$$

The monotype, ι, is discussed in Section 2.3. α is a member of an unbounded set of type variables. The unary type constructor `ptr` represents pointer types. The arrow ($\rightarrow$) represents a class of type constructors, one for each arity n. A type expression of the form $[\alpha]\{l_1:\tau_1,\ldots,l_k:\tau_k\}$ represents a `struct` type with k fields. The same construct is used to represent `union` types (see page 48). The type is tagged with a type variable α. The tag is necessary for inferring subtypes of `struct` types and for representing self-referential `struct` types. A type expression of the form `Typedef`(ν,τ) is much like a `typedef` in C. It assigns a new type name ν to the type τ. These "named" types allow nested `struct` types to be represented in a compact form. Through the use of named types, it can be shown that any type formed during type inference can be represented by a type expression that is polynomial in the length of the program. Type schemes are of the form:

$$\sigma ::= \tau|\forall\alpha.\sigma$$

A type environment, TE, is a map from program variables to type schemes.

As demonstrated in [63], polymorphic type inference in the presence of imperative programming features can make a Milner-style type system unsound. The type unsoundness arises in the use of polymorphic references—it is unsafe to treat the same memory cell as two different types. For example, in order to maintain type safety in ML, type variables must be divided into two classes, applicative and imperative.

It is unnecessary to complicate the type system used for the generalization of C programs with a distinction between imperative and applicative types. There are two reasons:

1. C has no explicit polymorphism, and the only way a program can cause a memory cell to hold values of different types is via a type cast or by use of a `union` type. If a program contains casts, we make no guarantees about the type safety of the resulting template.

2. If a function contains no casts then the generated C++ template also contains no casts. Because function templates are instantiated separately for each use of distinct monomorphic types, memory cells can never hold values of multiple types. Thus, if P is a type-safe C program, then P', its generalization, is a type-safe C++ program.

Type-inference rules

We have developed a set of type-inference rules in the style of [21]. For example, the rule for function application is:

$$\frac{\textsc{te} \vdash f : (\tau_1, \ldots, \tau_n) \to \tau, \ \ \textsc{te} \vdash e_1 : \tau_1, \ldots, \textsc{te} \vdash e_n : \tau_n}{\textsc{te} \vdash f(e_1, \ldots, e_n) : \tau}$$

The rules can be used to formalize the argument that the type-inference system is sound: The generalization of a type-safe C program is a type-safe C++ program. Rather than bog down the reader with pages of type-inference rules (and to maintain the spirit of the definitions of C and C++), we present the type-inference rules in words rather than formulas.

The type system incorporates the following constraints:

- Control statements impose the restriction that the type of the control expression be ι. Control expressions are found in `if`, `while`, `for`, `switch`, and `case` statements. For example, the statement `if (flag) break;` restricts `flag`'s type to be ι.

- Statements of the form `return` e require that the type of e coincide with the formal return type of the enclosing function. In the `power` example, the statement `return p` restricts `p`'s type be the same as the return type of `power` (i.e., $\alpha_0 = \alpha_3$).

- If an assignment expression is of the form $x = c$, where c is a constant expression and x is a variable, then a constraint is recorded in the environment that indicates that the type of x either matches the type of c or has a constructor that takes a single argument whose type is the original declared type of x. For example, in `power`, the expression `p = 1` constrains the type of `p` to be either an integer or a class `C` that has a constructor with signature `C(int)`. The `int` comes from `p`'s declared type. At first glance, this may seem wrong: Shouldn't the constructor take an argument that is the type of c? This is not possible because the type of c is ambiguous. For example, in the expression $x = 100$, 100 could be a `char`, `int`, `short int`, `unsigned short int`, etc. However, because of the Valid-Code

Assumption, it must be that the type of 100 is compatible with the type of x.

- Assignment expressions of the form $e = e'$, where e' is not a constant, impose the requirement that the type of e be the same as the type of e'. Implicit type conversions and promotions that can occur among arithmetic types are ignored—they are addressed in Section 2.7.1. For instance, even if `i` is declared to be of type `int` and `f` is declared to be of type `float`, the expression `f = i` causes the type of `f` and the type of `i` to be the same type.

- Primitive operators have polymorphic function types, as discussed in Section 2.4. In each expression involving a primitive operator, the polymorphic type is instantiated by stripping off the quantifiers and instantiating the body of the type with fresh type variables that are different from all other type variables used elsewhere. For example, the `*` operator has type $\forall\alpha.\alpha \times \alpha \rightarrow \alpha$. Thus, in `power`, the expression `p * base` requires that `p` and `base` be of the same type, but does not restrict that type to be monomorphic. Similarly, the expression `n > 0` forces `n` and `0` to be of the same type, and because `0` has type ι this forces `n` to have type ι.

- An indirection expression of the form `*`e constrains the type of e to be τ `ptr`, for some type τ. The result of the expression is τ. Likewise, if e is of type τ then the result of an expression of the form `&`e is τ `ptr`.

- Function calls are treated much like primitive operators. The types of actual arguments are unified with the types of the arguments of the function type. If the function type is universally quantified, it is instantiated with fresh type variables prior to unification with the actuals. For example, if `f` has type $\forall\alpha.\alpha \times \iota \rightarrow \iota$ then the expression `f(a, b)` has type ι, the type of `a` is unified with a new type variable, and the type of `b` is constrained to be ι.

- For an expression of the form `(*`e`)(`$x_1, \ldots, x_n$`)`, the type of e is constrained to be a pointer to a function of arity n.

- The type of a function with variable length argument list is taken to be ι.

- In C, an array expression of the form e_0`[`e_1`]` is equivalent to the expression `*(`e_0 `+` e_1`)`. Our typing rule is based on this equivalence. Suppose `a` is declared an array and `i` is declared an `int`. Because of the above rule, `a[i]` and `i[a]` are equivalent expressions. Because of the Valid-Code Assumption, it is always possible to distinguish between which expression is being used as the index and which as the array. For example, `a[i]` restricts the type of `i` to ι (since it is being used as the index) and type of `a` to be a pointer type.

- The types of expressions involving void pointers are constrained to be ι. This includes variables and functions declared to be of type `void*` as well as casts to `void*`. The use of `void*` is an indication that despite the Valid-Code Assumption, the program may have run-time type errors. As a trivial example, suppose x is declared to be a void pointer. Then the expression `x = &x` is "valid", but cannot be typed generically in our system.

- In C, there are two kinds of expressions that access fields of `struct` types: e`.`l and e`->`l. The latter form is just syntactic sugar for `*(`e`).`l and so we focus on the former. For each occurrence of an expression e`.`l, there are four possibilities:

1. If e's type has been inferred to be a `struct` type with an l-field of type τ, then $e.l$ has type τ.

2. If e's type has been inferred to be a `struct` type α that does not have an l-field, then α is constrained to be a `struct` type that has the fields of α plus an l-field of type α', where α' is a fresh type variable. The expression $e.l$ has type α'.

3. If e's type is the type variable α, then α is constrained to be a `struct` type that has an l-field of type α', where α' is a fresh type variable. The expression $e.l$ has type α'.

4. If e's type is ι, then the Valid-Code Assumption ensures that this is a legal use of $e.l$. The expression $e.l$ has type ι.

```
int matchAB(struct Record *rec0,
            struct Record *rec1)
{
  int i;

  i = (rec0->a == rec1->a) &&
      (rec0->b == rec1->b);
  return i;
}
```

Figure 15: The `MatchAB` function.

As an example, consider the `matchAB` function in Figure 15. The function takes two `struct` types as arguments and returns "true" if the `struct` types agree on both their `a`-fields and `b`-fields. Type inference on this function proceeds as follows:

1. Initially, we have the type environment:

[`matchAB`: $\forall\alpha_0, \alpha_1, \alpha_2.\alpha_1 \times \alpha_2 \rightarrow \alpha_0$,
`rec0`: α_1,
`rec1`: α_2,
`i`: α_3]

2. The expression `rec0->a` constrains α_1 to be a pointer to a `struct` with an `a`-field that has type α_4, a fresh type variable.

3. `rec1->a` constrains α_2 to be a `struct` with an `a`-field that has type α_5, a fresh type variable.

4. `rec0->a == rec1->a` constrains α_4 and α_5 to be equal. (The result type of the equality operator is ι.)

5. The expression `rec0->b` constrains α_1 to be a `struct` with not only an `a`-field, but also a `b`-field that has type α_6, a fresh type variable.

6. `rec1->b` constrains α_2 to be a `struct` with not only an an `a`-field, but also a `b`-field that has type α_7, a fresh type variable.

7. `rec0->b == rec1->b` constrains α_6 and α_7 to be equal.

8. The assignment to `i` constrains α_3 to be ι, because `&&` is considered to have type $\iota \times \iota \rightarrow \iota$.

9. `return i` constrains α_0 to be ι, since `i`'s type has been constrained that way.

10. The final type environment is as follows:

$$
\begin{array}{ll}
[\texttt{matchAB:} & \forall \alpha_4, \alpha_6, \alpha_1 \preceq [\alpha_8]\{\texttt{a} : \alpha_4, \texttt{b} : \alpha_6\}, \\
 & \alpha_2 \preceq [\alpha_9]\{\texttt{a} : \alpha_4, \texttt{b} : \alpha_6\}.\alpha_1 \times \alpha_2 \to \iota, \\
\texttt{rec0:} & \alpha_1 \preceq [\alpha_8]\{\texttt{a} : \alpha_4, \texttt{b} : \alpha_6\}, \\
\texttt{rec1:} & \alpha_2 \preceq [\alpha_9]\{\texttt{a} : \alpha_4, \texttt{b} : \alpha_6\} \\
\texttt{i:} & \iota]
\end{array}
$$

Thus, despite the declaration of both `a` and `b` as `struct Record *`, this function generalizes to allow two `struct`s of different types to be called by `matchAB`, assuming that they both have `a` and `b` fields with compatible types.

- `union` types are typed just as `struct` types. This is because the type of an expression `x.a` or `x->a` is independent of whether `x` is a `union` or `struct`. `Union` types differ from `struct` types in storage representation, but that does not effect the generalization process. (For an alternate treatment of `union` types based on their storage representation, see Chapter 4.)

- Conditional expressions of the form e_0 `?` e_1 `:` e_2 impose the restrictions that the type of e_0 be ι, the type of e_1 be the same as the type of e_2, and that the result type be the same as the type of e_1 and e_2.

- In a cast expression of the form `(`τ`)`e (where τ is a type other than `void*`), the cast expression itself is given type ι, but e is allowed to be any type. Type safety is not lost by this because the C++ compiler checks that the type cast makes sense at template-instantiation time. The template imposes an implicit constraint that e's value be one that is able to be converted "legally" to a value of type τ. For example, a function template containing a cast `(int)`e can only be called with arguments that have an `int` conversion, either predefined, as one has for arithmetic and pointer types, or explicitly defined (see [62, page 232]).

The result of type inference on `power` is summarized as follows:

- α_3 can be constructed from `int`
- $\alpha_0 = \alpha_1 = \alpha_3$
- $\alpha_2 =$ `int`

2.5.3 C-to-C++ Transformation

A C function is transformed into a C++ function template as follows: For each type variable occurring in the types inferred for the arguments, a fresh type identifier is generated. If no type variables occur, then the function remains as is. Otherwise, the function is made into a template with a template argument for each of the new type identifiers. In the `power` example, `base` has type α_0 and `n` has type `int`, so one new type identifier is created, `T`. The function is given the template header `template <class T> power(T base, int n)`.

For each variable declaration occurring within the function body, the variable's inferred type is considered. If that type contains any of the type variables occurring in the argument list, then it is converted to a C++ type with each such type variable replaced by the corresponding type identifier. If no such type variables occur in the inferred type (i.e., the inferred type is ι), the declaration remains as is.

Statements and expressions are transformed recursively via a straightforward syntax-directed translation. All statements and expressions remain the same in the template as in the original function body except for expressions of the form $x = c$, where c is a constant expression and x's type is a type variable α. If T is the type identifier associated with α as described above, then the assignment expression is converted to $x = T(c)$, where T is a C++ class constructor. For instance, in the `power` example, `p = 1` is transformed into `p = T(1)`.

2.5.4 Signatures

The generalization process produces information about restrictions on the types of template arguments. For example, in the `power` function template, the type of `base` cannot be just any type. It must have a constructor on integers and have a binary `*` operator of type $T \times T \rightarrow T$. Although the C++ compiler infers such restrictions on template arguments when checking each instantiation of the function template, C++ does not provide a mechanism to allow us to state such restrictions explicitly.

In addition to performing generalization *per se* (and creating appropriate function templates), we can also arrange for the system to produce documentation about constraints on the conditions under which the template can be instantiated (e.g., by creating descriptive signatures in the form of C++ comments about the template arguments). Signatures describing type restrictions on template arguments can be generated during the transformation phase of generalization by keeping track of constructors that have been inserted and operators that have been overloaded. In a case in which generalization is employed to create a template library from existing code, the signatures can be placed in the library's header file along with the function-template prototypes for easy reference.

The signature of the power function template appears below:

```
// power :  T * int -> T
// T :
//    operator * :  T * T -> T
//    constructor T() :  int -> T
```

Section 2.7.2 describes a limitation to our generalization algorithm's ability to accurately infer signatures.

2.6 Implementation and Results

We have implemented the C-to-C++ function-generalization algorithm, described in the previous section, as a tool that takes as input a C file and outputs templatized C++ code. The input is a C program component that has been pre-processed and compiles without error or warning according to the C standard. The tool features an option to only produce prototypes for templates that would be created, essentially summarizing the work that can be done, which allows the user to decide if it is worthwhile generalizing the input file. The tool also produces statistics describing how many templates are generated out of how many possible functions and the average number of template arguments. The tool is written is Standard ML.

The generalization examples used throughout this chapter have been produced by the implementation. We have also run our generalization tool on several medium-to-large programs. Table 1 summarizes the results produced by the generalization tool on the SPEC benchmark.

An example that generalized quite well is a representation of queues used in a thread

program	LOC	functions	templates	avg. function arity	avg. template arity	max. template arity	time
compress	1,934	24	4	1.4	2.0	2	91.8
gcc	205,085	1,289	403	2.0	2.7	6	180.2
go	29,246	372	71	2.4	3.3	4	69.8
ijpeg	31,211	134	46	2.3	2.6	4	37.8
li	7,597	367	61	2.1	1.9	3	76.5
m88ksim	19,915	128	23	1.9	1.5	3	48.3
perl	26,871	205	78	2.0	2.3	4	31.1
vortex	67,219	768	254	5.5	3.1	8	105.7

Table 1: Summary of results from applying generalization to C programs from the SPEC benchmark. Timings are in seconds using our generalization tool compiled with Standard ML of New Jersey version 110.7 on a Sun UltraSPARC running Solaris 2.6. (Timings inlcude parsing C files into abstract syntax trees.)

library. The functions are written in a modular style and this allows generalization to proceed successfully. Templates functions were produced that allow for queues to be formed with elements of any type.

Another interesting example is a library of routines to support the use of binary-decision diagrams. It consists of roughly one thousand lines of code and thirty-eight function definitions. Twelve templates were generated, with an average of 1.6 template arguments. The templates allow the same functions to be used on modified binary-decision diagram `struct`s that have been augmented with additional fields.

The use of the standard input and output functions can be a major inhibitor of generalization. Such functions cannot be generalized because the source is not visible. Often times the use of `printf` calls for debugging purposes prevent variables from being generalized. For one example we tried — a program to determine whether or not a point is inside a polygon — the program did not generalize at all because of type casts and the use of the standard input and output library.

One way around this limitation is to first transform the program to C++ by simply changing standard C input and output functions such as `printf` and `scanf` to C++ stream-based input and output functions such as `cin` and `cout`. Generalization can now proceed uninhibited since the `<<` and `>>` operators may be overloaded. While stream-based input and output is quite similar to standard C input and output, it is not identical and it may be difficult to automatically convert from one form to the other in a meaning-preserving way.

Generic memory allocation and handling functions such as `malloc` and `memcpy` can also be a major inhibitor of generalization. These functions can cause problem because they are frequently used in association with type casts, as in the following:

```
p = (int *)malloc(sizeof(int));
```

As with the case of input and output, this limitation can be handled in part by first converting the C code to C++, mapping `malloc` to `new` (and `memcpy` to appropriate copy constructors) where possible. (See Section 2.8 for a discussion of how generalization works in the presence of `new`). As is the case with input and output, it may be difficult to automatically convert from one form of memory allocation to another in a meaning-preserving fashion.

On the smaller examples tested, the generated function templates were tested by calling them with arguments of the originally declared types. The results were consistent with the results of calling the original C functions with the same arguments.

Some observations:

- The code fragments that produced the least generalization tended to make heavy use of global variables, standard I/O functions, and type casts.
- Functions that modify the elements of structures generalize nicely, as do functions written in a modular style.
- Over-generalization does not seem to be a big problem.
- Future versions of the generalization tool will have to drop the requirement that input programs be pre-processed. The tool should be able to restore macros so that resultant programs remain "clean."

2.7 Limitations and Complexity

In this section we describe a few limitations of our generalization algorithm.

2.7.1 Coping with Overflow and Casts

We say a C++ function template f' is a *meaning-preserving generalization* of a function f iff for every legal context in which f is employed, f' may be substituted with identical effect.

In order to guarantee that the generalization of a function is meaning preserving, the generalization process must be augmented slightly. There are sometimes implicit type casts in C programs that, if ignored, can result in incorrect generalization. Consider the simple multiplication program in Figure 16.

```
long mult(short m, short n)
{
  long a;

  a = m * n;
  return a;
}

void foo()
{
  short i,j;
  long l;

  i = j = 32767;
  l = mult(i,j); /* result: l = 1073676289 */
}
```

Figure 16: Two short integers multiplied to a long integer in C.

In the generalization algorithm described in Section 2.5, the `*` operator has type $\forall\alpha.\alpha \times \alpha \rightarrow \alpha$ and thus generalization produces the code shown in Figure 17.

The problem is that because we ignore the implicit promotion from a short integer to a long integer in the original C function, the template version assumes that the

```
template <class T1>
T1 mult(T1 m,T1 n)
{
  T1 a;

  a = m * n;
  return a;
}

void foo()
{
  short i,j;
  long l;

  i = j = 32767;
  l = mult(i,j); // result is still short, so l = 1
}
```

Figure 17: Two short integers multiplied in C++.

result must also be a short integer. Observe that if `i` and `j` are declared to be long integers in the C++ version of `foo`, as in Figure 18, then the result of the call to `mult` is as intended.

```
void foo()
{
  long i,j,l;

  i = j = 32767;
  l = mult(i,j); // result is l = 1073676289
}
```

Figure 18: Two long integers multiplied in C++.

A solution to this problem is to treat such implicit casts as if they are explicit as in Figure 19.

```
long mult(short m, short n)
{
  long a;

  a = (long)(m * n); /* think of cast as function shortToLong... */
  return a;
}
```

Figure 19: Two short integers multiplied in C, explicitly promoted to `long`.

Generalization can now proceed as before, using the type-inference rule for cast expressions previously mentioned on page 48:

> In a cast expression of the form $(\tau)e$ (where τ is a type other than `void *`), the cast expression itself is given type ι, but e is allowed to be any type. Type safety is not lost by this because the C++ compiler checks that the type cast makes sense at template-instantiation time. The template imposes an implicit constraint that e's value be one that is able to be converted "legally" to a value of type τ. For example, a function template containing a cast `(int)`e can only be called with arguments that have an `int` conversion, either predefined, as one has for arithmetic and pointer types, or explicitly defined (see [62, page 232]).

The intuition is that the type cast to `long` is really like a function that takes an argument from the source type (`short` in Figure 19) and returns a value of type `long`. In the case of `mult`, the `(long)` cast will not affect the generalization on the types of `m` and `n`, but it will keep the type of `a` as `long`. The result of this generalization is shown in Figure 20. This function template works on short integers as well as any type that has a conversion to long integer defined.

```
template <class T>
long mult(T m, T n)
{
  long a;

  a = m * n;

  return a;
}
```

Figure 20: An alternate multiplication template.

2.7.2 Complexity Issues

There is a subtlety to the fixed-point computation of the type-inference algorithm described in Section 2.5.2. When we stated (on page 39) that iteration continues "until the worklist is empty or an entire round of processing is completed that *does not introduce any changes*" and (on page 40) that "the algorithm proceeds until either the worklist is empty or, after one complete iteration, the worklist and *the types of all functions on the worklist remain the same*," we mean the types of the functions, thought of as potential template functions, remain the same. As an example consider the recursive function `f`, declared to take an argument that is a pointer to a linked-list, shown in Figure 21. After one iteration of type inference, the type inferred for `f`'s argument `x` is:

$$([\alpha_1]\{\texttt{i} : \iota, \texttt{next} : \alpha_2\ \texttt{ptr}\})\ \texttt{ptr}$$

This implies that our generalization algorithm would generate the following template declaration for `f`:

```
template<class T> int f(T *x) { ... }
```

```
struct S {
  int i;
  struct S *next;
};

int f(struct S *x)
{
  if (x->i)
    f(x->next);
  return 0;
}
```

Figure 21: A function illustrating a subtlety in our type-inference algorithm.

After another iteration, the type of `x` is inferred to be:

$$([\alpha_1]\{\texttt{i} : \iota, \texttt{next} : ([\alpha_2]\{\texttt{i} : \iota, \texttt{next} : \alpha_3\ \texttt{ptr}\})\ \texttt{ptr}\})\ \texttt{ptr}$$

However, given this type for `x`, our generalization algorithm would generate the *same* template declaration for `f`.

If we require the algorithm to wait for all the inferred types to reach a fixed point, the algorithm need not terminate. Indeed, this is the case with the above example. It is sufficient for the algorithm to halt when the potential template declarations have reached a fixed point. This is because C++ guarantees that templates will type check when instantiated.

The C++ template facility cam implicitly capture what our type system is not quite powerful enough to express: seemingly infinite types. In the case of the above example, the C++ template declaration is essentially saying "this function operates on arguments that are pointers to a class that has a `next` member that is a pointer to a class that has a `next` member that is a pointer to a class ..."

Our reliance on the C++ type system limits our ability to accurately determine signatures for functions (see Section 2.5.4) as illustrated by the above example. On the other hand, our reliance on C++'s type-checking abilities allows our type-inference algorithm to terminate in a polynomial number of iterations. Below we sketch a proof of this fact:

For a program of size n there are initially $O(n)$ functions defined. In the worst case all the functions are all mutually recursive. The complexity of one iteration through all functions is linear in the number of unification operations required. Unification is itself a linear-time operation in terms of the size of the type expressions being unified[37]. Initially there are are $O(n)$ type variables (type expressions of size $O(1)$). In the worst case, the complexity of the type expressions can be linear. So, for each iteration, the number of operations is bounded above by $O(n^2)$.

How many iterations occur in the worst case? There are at most $O(n)$ potential template parameters. Iterations continue only if this number decreases, so there are at most $O(n)$ iterations. This results in a crude upper bound of $O(n^3)$ for the worst-case running-time complexity of our type-inference algorithm. Our experimental data from Section 2.6 suggests that in practice the algorithm terminates in linear time.

The polynomial worst-cast running-time complexity of our type inference algorithm is an interesting contrast with the complexity of other polymorphic type inference algorithms. Type inference on the polymorphic recursive lambda calculus, which has a type system similar to ours, is an undecidable problem [30]. The complexity of type inference for ML is exponential-time complete [37].

2.8 Generalization of C++ Classes

A process similar to C-to-C++ function generalization can be applied in the case where C++ is both the source and target language. There are several motivations for developing a tool to automatically generalize C++ programs:

1. There is an abundance of C++ code that does not make use of templates because many C++ compilers did not correctly support templates in early versions.

2. It is often easier to develop and test code written in terms of specific types, even if the code is general. Stroustrup, [62, p. 257], argues that "it is usually a good idea to debug a particular class before turning it into a template."

3. It is not always easy to identify opportunities for generalization in classes and functions by casual inspection. This is particularly true in cases where constants can be lifted to template arguments (see below).

Functions in C++ can be generalized much as functions in C. A few language constructs, specific to C++, require the type-inference system to be augmented. These are primarily keywords pertaining to storage allocation, exception handling, operator overloading, and the visibility of class members. For example, an expression of the form `new T[i]` restricts the type of `i` to be monomorphic (namely `int`). The expression has type pointer to the type `T`.

Of greater interest is the problem of class generalization. In this case, the same basic technique used for function generalization is applied, except that the unit on which we operate is a class, and the restriction on type variables in the return types (see page 33) of functions (in this case member functions) no longer holds. A fresh type

```
class stack {
private:
  int *sp;
  int size;
public:
  stack(int sz) { size = sz; sp = new int[size]; }
  void push(int i) { *sp = i; sp++; }
  int pop() { sp--; return (*sp); }
};
```

Figure 22: `class stack`: A simple stack of integers.

variable is assigned to each member as well as to arguments, return values, and local variables of member functions. Type inference is then used to determine which types can be generalized and made into template arguments. The result of the generalization is a class template.

An example: Stack template

The following example, adapted from [62, pages 256–257], demonstrates how the generalization paradigm can be used to transform C++ classes into template classes.

A class representing a stack of integers is defined in Figure 22. For simplicity, the stack ignores such issues as underflow and overflow. The constructor `stack` creates an empty stack with space allocated for `size` integer elements. The `push` and `pop` functions are defined as expected.

The class template `stack`, shown in Figure 23, is created via class generalization. Notice that the sole template argument, `T`, is the type of the element stored in the stack. The definition of the C++ `new` command restricts the type of the `size` field to

```
template <class T>
class stack {
private:
  T *sp;
  int size;
public:
  stack(int sz) { size = sz; sp = new T[size]; }
  void push(T i) { *sp = i; sp++; }
  T pop() { sp--; return (*sp); }
};
```

Figure 23: `class stack`: A class template for stacks.

be `int`. So, the type of a program variable used as the size of a `new` expression must be bound to ι in the type system used for C++-to-C++ generalization.

Fixed-type template parameters

For a class, a template argument need not be a type name. Constant expressions and function arguments can also be used. For example,

```
template <int size>
class intArray {
  int a[size];
}
```

is a class template parameterized by an integer that determines the size of the member array. A class with a ten-element array can be specified as `intArray<10>`. Thus, another generalization opportunity that exists in C++ class generalization is the identification of constants that can be "lifted" to fixed-type template arguments. For instance, `class ten { int a[10]; };` could be generalized to the class template `intArray` in the example above. Existing occurrences of `ten` are then replaced by `intArray<10>`.

It is also possible to lift function names. For example, suppose we have a class that makes use of a random number generator. We might wish to compare how the class behaves with respect to different generators. We would like to be able to generalize such a class to the form:

```
template <int rand()>
class C {
  ⋮
  // some occurrences of rand
  ⋮
};
```

Discretion should be used when lifting constants to fixed-type template arguments. For example, the constant string `"\n"` representing a carriage return is an unlikely candidate for lifting. Similar discretion should be taken in the related lifting problem that occurs in generalization of ML programs, discussed in Section 2.9.

Lifting constructor parameters

In some instances, a constructor parameter can be replaced by a template parameter. This might be the case with the `size` argument of the of the `stack` constructor. For example, Figure 24 shows a different stack template. In this case, existing calls to the constructor of the form `stack(n)` are replaced by `stack<n>()`.

This is not truly a generalization, since C++ has no type-safe notion of polymorphic container objects. (That is, class templates can be instantiated over any number of different types, but each instance becomes a monomorphic class in its own right.) Thus, `stack<int,5>` and `stack<int,10>` are incompatible types. (In this sense, lifting constructor parameters can be thought of as producing *less* general code.) In some

```
template <class T, int size>
class stack {
private:
  T *sp;
public:
  stack() { sp = new T[size]; }
  void push(T i) { *sp = i; sp++; }
  T pop() { sp--; return (*sp); }
};
```

Figure 24: `class stack`: Another class template for stacks.

contexts, this separation of types can add type safety to programs. An example is a matrix class template, shown in Figure 25, for which it is illegal to multiply matrices of different sizes.

Circumventing function-template limitations

As discussed in Section 2.3, for function templates each template argument must affect the type of at least one of the function arguments. This limitation restricts generalizations that could otherwise be made through structure subtyping. This expressibility can be regained, at least in part, through the use of class templates. For example, a `getNextVal` function (see Section 2.2.3, page 28) that is more general in its return value can be written as a member function, as shown in Figure 26.

A member function of a class template is very similar to a function template. A difference is that a member function of a class template can refer to the class-template parameters in its return type (as in `getNextVal` in Figure 26) regardless of whether it refers to the class-template parameters in its argument types. This is because a member function implicitly takes its class as an argument, so while the `getNextVal`

```
// two-dimensional, n-by-n matrices
template <class T, int n>
class Matrix {
private:
  T M[n][n];
public:
  ...
  Matrix<T,n> operator *(const Matrix<T,n> &B) {
    Matrix<T,n> C;
    for(int i=0; i < n; i++)
      for(int j=0; j < n; j++) {
        int sum=0;
        for(int k=0; k < n; k++)
          sum += M[i][k] * B.M[k][j];
        C.M[i][j] = sum;
      }
    return C;
  }
};

int main()
{
   Matrix<int,2> a,b,c;
   Matrix<int,4> d;

   ...

   c = a * b; // okay
   c = a * m; // type error

   ...
}
```

Figure 25: Matrix multiplication: type safety in numbers.

```
template <class T>
class Simple {
private:
  T x_;
  Simple<T> *next_;
  T x() {return x_;}
  Simple<T> *next(){return next_;}
public:
  Simple(int i){x_ = i; next_ = NULL;}
  void link(Simple<T> *s){next_ = s;}
  T getNextVal() {return next()->x();}
};
```

Figure 26: A class template alternative to the `getNext` problem.

member function is written as:

```
T getNextVal() {return next()->x();}
```

it is very much like if it were written:

```
T getNextVal(Simple<T> *this) {return this->next()->x();}
```

The `getNextVal` function can also be used by classes derived from `Simple`.

Another way to avoid the problem is to introduce trivial class templates in conjunction with function templates. This allows for function templates to be explicitly parameterized by type. As an example, consider the program in Figure 27. Here, the function template `getNextVal` takes two arguments. The return type of the function is also a parameter to the template class that is the type of the first argument. The second argument is the an object from which the next `x` field is to be extracted. The second argument can be any object of a class that has a `next` field which is a pointer to an object of a class that has an `x` field that has the type indicated by the first argument. While this is not as succinct as we might like, it does allow for the template functions to operate on a broad array of classes in terms of both the names and types of the fields. In fact, this mechanism is widely used in the C++ Standard Template Library.

2.9 Generalization of Standard ML

As mentioned in Section 2.1, generalization can apply to a wide variety of programming languages. In this section, as a case study, we describe how techniques similar to those described above can be employed to generalize Standard ML programs. The work here demonstrates that it is useful to consider generalization even in the case of a language that already uses polymorphic type inference.

An elegant aspect of generalizing ML programs is that the rigorously defined semantics of functions and functors allow for straightforward proofs that generalization transformations are meaning preserving.

Polymorphic type inference is already present in Standard ML. To achieve function generalization akin to that described for C in Section 2.5, we need only remove explicit type constraints and let the type-inference engine do the rest. However, unlike C++, ML does not allow the user to overload operators. To compensate for the lack of overloading, additional parameters can be inserted in function definitions, in effect "lifting" operators, function names, and constant values. Further generalization in ML can be achieved by transforming structures to functors by lifting value and type members of structures to functor parameters. These mechanisms are described below.

```
struct IntList {
  int x;
  struct IntList *next;
} *s;

struct IntListList {
  struct IntList x;
  struct IntListList *next;
} *t;

template <class T> class dummy {};

template <class T0, class T1>
T0 getNextVal(dummy<T0>, T1 t)
{
  return t->next->x;
}

void foo()
{
  dummy<int> dummy_int;
  dummy<struct IntList> dummy_IntList;

  getNextVal(dummy_int, s);
  // getNextVal(dummy_IntList, s); <-- error
   getNextVal(dummy_IntList, t);
  // getNextVal(dummy_int, t); <-- error
}
```

Figure 27: Using class templates with function templates.

```
fun intsort [] = []
  | intsort (x :: xs) =
      let
        fun insert (x, []) = [x]
          | insert (x, Y as y :: ys) =
              if x < y then x :: Y
              else y :: insert (x, ys)
      in
        insert (x, intsort xs)
      end

val intsort = fn : int list -> int list
```

Figure 28: Insertion sort for integers in Standard ML.

Generalizing ML functions by lifting

As an example of the lifting, consider the insertion-sort function given in Figure 28. The function sorts a list of integers. The only type-dependent operation in the function is the comparison operator `<`. This function cannot be generalized simply by defining a new `<` operator for some other type, since, unlike in C++, the user cannot overload operators. However, it can be generalized by lifting the operator out of the function and making it a parameter as shown in function `insort` given in Figure 29. Since `insort` is curried, occurrences of `intsort` can be replaced by `(insort (op < ))`. The semantics of function application can be used to prove that this is a meaning-preserving transformation.

Using type constraints to test for generality

Type constraints are used in ML programs for two main reasons:

1. Type constraints give the reader (and the programmer) hints as to the intention

```
fun insort _ [] = []
  | insort (op <) (x :: xs) =
      let
        fun insert (x, []) = [x]
          | insert (x, Y as y :: ys) =
               if x < y then x :: Y
               else y :: insert (x, ys)
      in
        insert (x, insort (op <) xs)
      end

val insort = fn : ('a * 'a -> bool) -> 'a list -> 'a list
```

Figure 29: Generalized insertion sort.

of the code.

2. Constraints assist type inference. Generic polymorphism is disallowed for certain uses of reference types. For example, the expression `Array.array(10,nil)` is intended to create an array of lists with each element initialized to the empty list. The expression is only valid if a monomorphic list type can be inferred. Type constraints can be used to inform the compiler which type is intended, as in `Array.array(10,nil:int list)`. Type constraints may also be needed to inform the compiler of what fields a record type has. For example:

   ```
   type person = name : string, yearOfBirth : int
   fun age thisYear (p : person) = thisYear - (#yearOfBirth p)
   ```

As a simple test for generality, type constraints can be erased and type inference can be performed without them. In the case that an error (or warning) results[8], the original

[8]Typical messages would be: "Warning: type vars not generalized because of value restriction are instantiated to dummy types" or "Error: unresolved flex record"

constraints would be restored. Otherwise, the inferred types would be compared with the annotated types. If the inferred types are more general, that fact would be reported and the more general types are output as constraints.

The utility of such a tool would stem from the fact that programmers, as part of the design phase, often write the type constraint for a function prior to writing the function body. The discovery that a function is more general than its type constraint indicates a potential conflict between design and implementation. On one hand, the programmer may be able to make use of the added generality of the function. On the other hand, the information can assist in identifying errors at an early stage in program development.

Generalizing structures

Another opportunity for generalization of ML programs can be found in abstract datatypes that are unnecessarily specific in their underlying types. For example, consider the declaration for the type `stack` in the stack structure shown in Figure 30. The structure is a simplistic representation of a stack of integers. Suppose the type that `Stack` constructs from, `int list * int`, is temporarily "erased" and replaced by a type variable. Type inference on the function defined within the `IntStack` structure reveals that the constructor could be generalized to be from type `'a list * int`. This type of generalization can be performed on any structure by considering its datatype declarations. The generalization for `IntStack` (and its signature) appears in Figure 31.

```
signature INT_STACK =
  sig
    exception OVERFLOW
    exception UNDERFLOW
    type stack
    val mkStack : unit -> stack
    val push    : stack * int -> stack
    val pop     : stack -> stack * int
  end
structure IntStack : INT_STACK =
  struct
    exception OVERFLOW
    exception UNDERFLOW
    datatype stack = Stack of int list * int
    val stackSize = 100
    fun mkStack () = Stack([], 0)
    fun push (Stack(L, sp), i) =
          if sp < stackSize then Stack(i::L, sp+1)
          else raise OVERFLOW
    fun pop (Stack([], _)) = raise UNDERFLOW
      | pop (Stack(i::is, sp)) = (Stack(is, sp-1), i)
  end
```

Figure 30: `IntStack`: A simple stack of integers in Standard ML

```
signature STACK =
  sig
    exception OVERFLOW
    exception UNDERFLOW
    type stack
    val mkStack : unit -> 'a stack
    val push    : stack * 'a -> stack
    val pop     : stack -> stack * 'a
  end
structure Stack : STACK =
  struct
    exception OVERFLOW
    exception UNDERFLOW
    datatype 'a stack = Stack of 'a list * int
    val stackSize = 100
    fun mkStack () = Stack([], 0)
    fun push (Stack(L, sp), i) =
          if sp < stackSize then Stack(i::L, sp+1)
          else raise OVERFLOW
    fun pop (Stack([], _)) = raise UNDERFLOW
      | pop (Stack(i::is, sp)) = (Stack(is, sp-1), i)
  end
```

Figure 31: `Stack`: A generalized stack structure (and signature) in Standard ML

Generalizing structures to functors

The value of `stackSize` in the `IntStack` structure is arbitrary. A generalization might be a functor that is parameterized by the size of the stack. For example, a stack functor based on `IntStack` might be of the following form:

```
functor Stack (S : sig val size : int end) : STACK =
  struct
    ...
    val stackSize = S.size
    ...
  end
```

The original stack implementation is then provably equivalent to:

```
structure IntStack100 = Stack (struct val size = 100 end)
```

A constant value within a structure may be lifted to a function parameter. This lifting process is much like the corresponding process of lifting constant expressions to fixed-type parameters in C++ class templates, discussed in Section 2.8.

Sometimes generalizing a structure requires more subtlety. Consider the binary-search-tree structure in Figure 32 (adapted from [53, p. 127]). `Tree` is a `btree` constructor from triples of type `'a btree * (int * 'a) * 'a btree`. The type of the key is required to be `int` because of the use of the `<` operator to compare keys. However, we would like for the binary-search tree to work with other ordered types as well. We can generalize the `IntBTree` structure to a functor parameterized by an order structure, as shown in Figure 33. The structure `IntBTree` can be instantiated from this

```
structure IntBTree =
  struct
    datatype 'a btree =
      Empty |
      Tree of 'a btree * (int * 'a) * 'a btree
    val empty = Empty
    fun lookup (Empty, _) = NONE
      | lookup (Tree(L, (i, a), R), key) =
          if key < i then lookup (L, key)
          else if i < key then lookup (R, key)
               else SOME(a)
    fun insert (Empty, key, a) = Tree(Empty, (key, a), Empty)
      | insert (Tree(L, p as (i, _), R), key, a) =
          if key < i then Tree(insert (L, key, a), p, R)
          else if i < key then Tree(L, p, insert (R, key, a))
               else Tree(L, (i, a), R)
  end
```

Figure 32: `IntBTree`: A binary search tree structure with integer keys.

functor as shown in Figure 34. This structure is provably equivalent to the one defined in Figure 32.

The `power` function revisited

Suppose we wish to generalize the power function written in ML:

```
fun power b n =
  if n = 0 then 1
  else b * (power b (n - 1))
```

Imitating the C-to-C++ power generalization discussed earlier, we might create a function like:

```
fun powgen (c, op * ) b n =
```

```
functor BTree (structure Order :
                 sig
                   type key
                   val cmpKey : key * key -> bool
                 end) =
  struct
    datatype 'a btree =
      Empty |
      Tree of 'a btree * (Order.key * 'a) * 'a btree
    infix <
    fun a < b = Order.cmpKey (a, b)
    ⋮
end
```

Figure 33: `BTree`: A binary search tree functor.

```
structure IntBTree = BTree (structure Order =
                              struct
                                type key = int
                                val cmpKey = op <
                              end)
```

Figure 34: `IntBTree`: An instantiation of the `BTree` functor.

```
functor Power (structure S :
                 sig
                   type t
                   val c : int -> t
                   val binOp : t * t -> t
                 end) =
  struct
    infix *
    val op * = S.binOp
    fun power b n =
      if n = 0 then S.c(1)
      else b * (power b (n - 1))
  end
```

Figure 35: `Power`: A functor for computing generalized power.

```
  if n = 0 then c(1)

  else b * (powgen (c, op * ) b (n-1))
```

A provably equivalent integer power function can then be formed as:

```
val power = powgen (fn x => x, op * )
```

However, lifting operators to function parameters as such may quickly get messy. A more elegant solution is to wrap the power function inside a functor, as in Figure 35. From this functor, a power function can be created for any type, given a binary "multiplication" operator defined for that type and function that builds values of the type from integers. A provably equivalent integer power function can be built as in Figure 36.

```
structure P = Power (structure S =
                       struct
                         type t = int
                         val c = fn x => x
                         val binOp = (op * ) : int * int -> int
                       end)
val power = P.power
```

Figure 36: `power`: An instantiation of the `Power` functor.

2.10 Related Work

Many of the ideas of this chapter are also discussed in [58].

Although much has been written about the problem of software reuse (for example, see [68, 46, 54]), including work on identifying reusable components [18, 8], we are unaware of previous work on the problem of automatically creating polymorphic functions from monomorphic functions.

Our work may be contrasted with what is provided by the NORA system, which also makes use of type inference to support polymorphic components [27]. The paradigm in NORA is to extend a base language (e.g., C, Pascal, Modula-2) with a more powerful type system that permits fragments to be given types and units containing unbound names to be given types. In contrast, generalization involves elevating a fully fleshed-out fragment into a polymorphic, reusable component. In other words, whereas NORA supports the *use* of polymorphic components, our goal is to provide support for *extracting* such components from existing code.

Our work may also be contrasted with the signature-matching tool described in [69]. While that work also employs type inference to facilitate software reuse, it does so by assisting the user in finding suitable code from software libraries rather than automatically producing generalized code. A signature matching tool might work nicely in conjunction with the work described in this chapter. A generalizer could be used to create code to be placed into libraries and the signature matcher could be used to retrieve generalized code.

The idea of mixing polymorphism with C appears in several places, among them [27, 59, 50]. [59] concerns a new dialect of C that is polymorphic and type safe. This differs from our approach in that it is not aimed at adding polymorphism to existing code. [50] uses polymorphic type inference on existing C programs, but for determining information about the transfer of values, as opposed to producing reusable code.

2.11 Summary

This chapter has discussed the problem of C function generalization and given an algorithm that provides a way to transform one or more C functions into C++ function templates. In the next chapter, we focus on the problem of transforming C structures into C++ classes. Along with the C++ class generalization process described in Section 2.8, this can be viewed as furthering the C-to-C++ generalization effort, offering the possibility of transforming C structs into C++ class templates.

Chapter 3

Modularization

3.1 Introduction

Many existing software systems were developed using programming languages and paradigms that do not incorporate object-oriented features and design principles. In particular, these systems often lack a modular style, making maintenance and further enhancement an arduous task. The software engineer's job would be less difficult if there were tools that could transform code that does not make explicit use of modules into functionally equivalent object-oriented code that does make use of modules (or classes). Given a tool to (partially) automate such a transformation, legacy systems could be modernized, making them easier to maintain. The modularization of programs offers the added benefit of increased opportunity for code reuse.

A major difficulty with software modularization is the accurate identification of potential modules and classes. This chapter describes how a technique known as *concept analysis* can help automate modularization. The main contributions of this chapter are:

- We show how to apply concept analysis to the modularization problem. We focus on one variant of the modularization problem—the conversion of a C program to a C++ program, where the C program's `struct` types are the starting point for the C++ program's classes.

- Previous work on the modularization problem has made use only of "positive" information: Modules are identified based on properties such as "function `f` uses variable `x`" or "`f` has an argument of type `t`". It is sometimes the case that a module can be identified by what values or types it does *not* depend upon—for example, "function `f` uses the fields of `struct queue`, but not the fields of `struct stack`". Concept analysis allows both positive and negative information to be incorporated into a modularization criterion. (See Section 3.3.2.)

- Unlike several previously proposed techniques, the concept-analysis approach offers the ability to "stay within the system" (as opposed to applying ad hoc methods) when the first suggested modularization is judged to be inadequate:

  - If the proposed modularization is on too fine a scale, the user can "move up" the partition lattice. (See Section 3.4.)

  - If the proposed modularization is too coarse, the user can add additional attributes to identify finer-granularity concepts. (See Section 3.3.)

- We demonstrate how concept analysis can be used to identify distinct modules amid "tangled" code. (See Section 3.3.2.)

- We define the notion of a *concept partition* and present an algorithm to generate partitions from a concept lattice. (See Section 3.4.)

- We have implemented a prototype tool that uses concept analysis to find potential modularizations of C programs. The implementation has been tested on several medium-to-large-sized examples. (See Section 3.5.)

As an example, consider the C implementation of stacks and queues shown in Figure 37. Queues are represented by two stacks, one for the front and one for the back; information is shifted from the front stack to the back stack when the back stack is empty. The queue functions only make use of the stack fields indirectly—by calling the stack functions. Although the stack and queue functions are written in an interleaved order, we would like to be able to tease the two components apart and make them separate classes, one a client of the other, as in the C++ code given in Figure 38.

This chapter discusses a technique by which modules (in this case C++ classes) can be identified in legacy C code. The resulting information can then be supplied to a suitable transformation tool that maps C code to C++ code, as in the aforementioned example. Although other modularization algorithms are able to identify the same decomposition [10, 67], they are unable to handle a variant of this example in which stacks and queues are more tightly intertwined (see Section 3.3.2). In Section 3.3.2, we show that concept analysis *is* able to group the code from the latter example into separate queue and stack modules.

The remainder of this chapter is structured as follows: Section 3.2 introduces contexts and concept analysis, and an algorithm for building concept lattices from contexts. Section 3.3 discusses a process for identifying modules in C programs based on concept analysis. Section 3.4 defines the notion of a concept partition and presents an algorithm for finding the partitions of a concept lattice. Section 3.5 discusses the implementation. Section 3.7 concerns related work.

```
#define QUEUE_SIZE 10
struct stack { int *base, *sp, size; };
struct queue { struct stack *front, *back; };

struct stack* initStack(int sz)
{ struct stack* s = (struct stack*) malloc(sizeof(struct stack));
  s->base = s->sp = (int*)malloc(sz * (sizeof(int)));
  s->size = sz;
  return s; }

struct queue* initQ()
{ struct queue* q = (struct queue*) malloc(sizeof(struct queue));
  q->front = initStack(QUEUE_SIZE);
  q->back  = initStack(QUEUE_SIZE);
  return q; }

int isEmptyS(struct stack* s)
{ return (s->sp == s->base); }

int isEmptyQ(struct queue* q)
{ return (isEmptyS(q->front) && isEmptyS(q->back)); }

void push(struct stack* s, int i)
{ *(s->sp) = i; s->sp++; }    /* no overflow check */

void enq(struct queue* q, int i)
{ push(q->front, i); }

int pop(struct stack* s)
{ if (isEmptyS(s)) return -1;
  s->sp--;
  return (*(s->sp)); }

int deq(struct queue* q)
{ if (isEmptyQ(q)) return -1;
  if (isEmptyS(q->back))
    while(!isEmptyS(q->front)) push(q->back, pop(q->front));
  return pop(q->back); }
```

Figure 37: A queue using two stacks in C.

```
const int QUEUE_SIZE = 10;

class stack {
private:
  int* base;
  int* sp;
  int size;
public:
  stack(int sz) {
    base = sp = new int[sz];
    size = sz; }
  int isEmpty() {
    return (sp == base); }
  int pop() {
    if (isEmpty()) return -1;
    sp--;
    return (*sp); }
  void push(int i) {
    // no overflow check
    *sp = i; sp++; }
};

class queue {
private:
  stack *front, *back;
public:
  queue() {
    front = new stack(QUEUE_SIZE);
    back = new stack(QUEUE_SIZE); }
  int isEmpty() { return (front->isEmpty() && back->isEmpty()); }
  int deq() {
    if (isEmpty()) return -1;
    if (back->isEmpty())
      while(!front->isEmpty()) back->push(front->pop());
    return back->pop(); }
  void enq(int i) { front->push(i); }
};
```

Figure 38: Queue and stack classes in C++.

		attributes				
		four-legged	hair-covered	intelligent	marine	thumbed
objects	cats	√	√			
	dogs	√	√			
	dolphins			√	√	
	gibbons		√	√		√
	humans			√		√
	whales			√	√	

Table 2: A crude characterization of mammals.

3.2 A Concept Analysis Primer

Concept analysis provides a way to identify sensible groupings of *objects* that have common *attributes* [65].

To illustrate concept analysis, we consider the example of a crude classification of a group of mammals: cats, gibbons, dogs, dolphins, humans, and whales. Suppose we consider five attributes: four-legged, hair-covered, intelligent, marine, and thumbed. Table 2 shows which animals are considered to have which attributes. We can interpret this data in a variety of ways. For example, we might observe that whales, dolphins, humans, and gibbons are all intelligent. On the other hand, gibbons, dogs, and cats are all hair-covered, but only the latter two are four-legged.

In order to understand the basics of concept analysis, a few definitions are required. A *context* is a triple $\mathcal{C} = (\mathcal{O}, \mathcal{A}, \mathcal{R})$, where $\mathcal{O}$ and $\mathcal{A}$ are finite sets (the objects and attributes, respectively), and $\mathcal{R}$ is a binary relation between $\mathcal{O}$ and $\mathcal{A}$. In the mammal example, the objects are the different kinds of mammals, the attributes are the characteristics four-legged, hair-covered, etc. The binary relation $\mathcal{R}$ is given in Table 2. For example, the tuple (whales, marine) is in $\mathcal{R}$, but (cats, intelligent) is not.

Let $X \subseteq \mathcal{O}$ and $Y \subseteq \mathcal{A}$. The mappings $\sigma(X) = \{a \in \mathcal{A} | \forall o \in X : (o,a) \in \mathcal{R}\}$ (the *common attributes* of X) and $\tau(Y) = \{o \in \mathcal{O} | \forall a \in Y : (o,a) \in \mathcal{R}\}$ (the *common objects* of Y) form a *Galois connection.* That is, the mappings are *antimonotone*:

$$X_1 \subseteq X_2 \Rightarrow \sigma(X_2) \subseteq \sigma(X_1)$$

$$Y_1 \subseteq Y_2 \Rightarrow \tau(Y_2) \subseteq \tau(Y_1)$$

and *extensive*:

$$X \subseteq \tau(\sigma(X)) \quad \text{and} \quad Y \subseteq \sigma(\tau(Y)).$$

In the mammal example, $\sigma(\{\text{cats}, \text{gibbons}\}) = \{\text{hair-covered}\}$ and $\tau(\{\text{marine}\}) = \{\text{dolphins}, \text{whales}\}$.

A *concept* is a pair of sets—a set of objects (the *extent*) and a set of attributes (the *intent*) (X, Y)—such that $Y = \sigma(X)$ and $X = \tau(Y)$. That is, a concept is a maximal collection of objects sharing common attributes. In the example, ({cats, dogs}, {four-legged, hair-covered}) is a concept, whereas ({cats, gibbons}, {hair-covered}) is not a concept. A concept (X_0, Y_0) is a *subconcept* of concept (X_1, Y_1), denoted by $(X_0, Y_0) \leq (X_1, Y_1)$, if $X_0 \subseteq X_1$ (or, equivalently, $Y_1 \subseteq Y_0$). For instance, ({dolphins, whales}, {intelligent, marine}) is a subconcept of ({gibbons, dolphins, humans, whales}, {intelligent}). The subconcept relation forms a complete partial order (the *concept lattice*) over the set of concepts. The concept lattice for the mammal example is shown in Figure 39. Each node in the lattice represents a concept. A key indicating the extent and intent of each concept is shown in Table 3.

The fundamental theorem for concept lattices [65] relates subconcepts and superconcepts as follows:

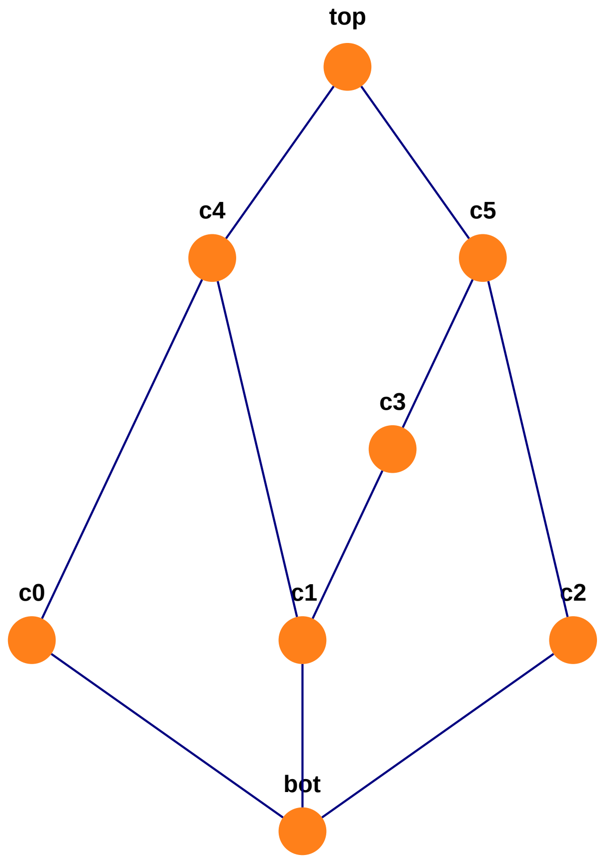

Figure 39: The concept lattice for the mammal example.

top	({cats, gibbons, dogs, dolphins, humans, whales}, $\emptyset$)
c_5	({gibbons, dolphins, humans, whales}, {intelligent})
c_4	({cats, gibbons, dogs}, {hair-covered})
c_3	({gibbons, humans}, {intelligent, thumbed})
c_2	({dolphins, whales}, {intelligent, marine})
c_1	({gibbons}, {hair-covered, intelligent, thumbed})
c_0	({cats, dogs}, {hair-covered, four-legged})
bot	($\emptyset$, {four-legged, hair-covered, intelligent, marine, thumbed})

Table 3: The extent and intent of the concepts for the mammal example.

$$\begin{array}{rcl} c_1 \sqcup c_2 & = & \\ (\{\text{gibbons}\}, \{\text{hair, intell., thumbed}\}) & \sqcup & (\{\text{dolphins, whales}\}, \{\text{intell., marine}\}) \\ & = & (\tau(\{\text{intelligent}\}), \{\text{intelligent}\}) \\ & = & (\{\text{gibb., humans, dolph., whales}\}, \{\text{intell.}\}) \\ & = & c_5 \end{array}$$

Figure 40: An example use of the fundamental theorem of concept lattices: This computation corresponds to the fact that $c_1 \sqcup c_2 = c_5$ in the lattice shown in Figure 39.

$$\bigsqcup_{i \in I} (X_i, Y_i) = \left(\tau \left(\bigcap_{i \in I} Y_i \right), \bigcap_{i \in I} Y_i \right).$$

The significance of the theorem is that the least common superconcept of a set of concepts can be computed by intersecting their intents, and by finding the common objects of the resulting intersection. An example of the application of the fundamental theorem is shown in Figure 40.

There are several algorithms for computing the concept lattice for a given context [26, 60]. We describe a simple bottom-up algorithm here.

An important fact about concepts and contexts used in the algorithm is that, given a set of objects X, the smallest concept with extent containing X is $(\tau(\sigma(X)), \sigma(X))$. Thus, the bottom element of the concept lattice is $(\tau(\sigma(\emptyset), \sigma(\emptyset))$—the concept consisting of all objects (often the empty set, as in our example) that have all the attributes in the context relation.

The initial step of the algorithm is to compute the bottom element of the concept lattice. The next step is to compute *atomic* concepts—smallest concepts with extent containing each of the objects treated as a singleton set[1]. Computation of the atomic

[1]Atomic concepts should be distinguished from the *atoms* of a lattice—the nodes reachable from

$$
\begin{aligned}
\tau(\sigma(\{\text{cats}\})) &= \tau(\{\text{four-legged}, \text{hair-covered}\}) \\
&= \{\text{cats}, \text{dogs}\} \\
\tau(\sigma(\{\text{dogs}\})) &= \tau(\{\text{four-legged}, \text{hair-covered}\}) \\
&= \{\text{cats}, \text{dogs}\} \\
\tau(\sigma(\{\text{dolphins}\})) &= \tau(\{\text{intelligent}, \text{marine}\}) \\
&= \{\text{dolphins, whales}\} \\
\tau(\sigma(\{\text{gibbons}\})) &= \tau(\{\text{hair-covered}, \text{intelligent}, \text{thumbed}\}) \\
&= \{\text{gibbons}\} \\
\tau(\sigma(\{\text{humans}\})) &= \tau(\{\text{intelligent}, \text{thumbed}\}) \\
&= \{\text{humans, gibbons}\} \\
\tau(\sigma(\{\text{whales}\})) &= \tau(\{\text{intelligent}, \text{marine}\}) \\
&= \{\text{dolphins, whales}\}
\end{aligned}
$$

Figure 41: Computing atomic concepts in the mammal example.

concepts for the mammal example is shown in Figure 41.

The algorithm then closes the set of atomic concepts under join: Initially, a worklist is formed containing all pairs of atomic concepts (c', c) such that $c \not\leq c'$ and $c' \not\leq c$. While the worklist is not empty, remove an element of the worklist (c_0, c_1) and compute $c'' = c_0 \sqcup c_1$. If c'' is a concept that is yet to be discovered then add all pairs of concepts (c'', c) such that $c \not\leq c''$ and $c'' \not\leq c$ to the worklist. The process is repeated until the worklist is empty.

The iterative step of the concept-building algorithm is illustrated in Figure 42.

the bottom element in one step. In Section 3.4, we define the notion of a well-formed context—a context in which the atomic concepts correspond to the atoms of the concept lattice.

$$
\begin{aligned}
c_0 &= (\{\text{cats}, \text{dogs}\}, \{\text{hair-covered}, \text{four-legged}\}) \\
c_1 &= (\{\text{gibbons}\}, \{\text{hair-covered}, \text{intelligent}, \text{thumbed}\}) \\
c_2 &= (\{\text{dolphins, whales}\}, \{\text{intelligent}, \text{marine}\}) \\
c_3 &= (\{\text{gibbons}, \text{humans}\}, \{\text{intelligent}, \text{thumbed}\}) \\
Worklist &= [(c_0, c_1), (c_0, c_2), (c_0, c_3), (c_1, c_2), (c_2, c_3)] \\
c_4 &= c_0 \sqcup c_1 = (\{\text{cats}, \text{gibbons}, \text{dogs}\}, \{\text{hair-covered}\}) \\
Worklist &= [(c_0, c_2), (c_0, c_3), (c_1, c_2), (c_2, c_3), (c_2, c_4), (c_3, c_4)] \\
c_0 \sqcup c_2 &= \top = (\{\text{cats}, \text{gibbons}, \text{dogs}, \text{dolphins}, \text{humans}, \text{whales}\}, \emptyset) \\
Worklist &= [(c_0, c_3), (c_1, c_2), (c_2, c_3), (c_2, c_4), (c_3, c_4)] \\
c_0 \sqcup c_3 &= \top \\
Worklist &= [(c_1, c_2), (c_2, c_3), (c_2, c_4), (c_3, c_4)] \\
c_5 &= c_1 \sqcup c_2 = (\{\text{gibbons}, \text{dolphins}, \text{humans}, \text{whales}\}, \{\text{intelligent}\}) \\
Worklist &= [(c_2, c_3), (c_2, c_4), (c_3, c_4), (c_0, c_5), (c_4, c_5)] \\
c_2 \sqcup c_3 &= c_5 \\
Worklist &= [(c_2, c_4), (c_3, c_4), (c_0, c_5), (c_4, c_5)] \\
c_2 \sqcup c_4 &= \top \\
Worklist &= [(c_3, c_4), (c_0, c_5), (c_4, c_5)] \\
c_3 \sqcup c_4 &= \top \\
Worklist &= [(c_0, c_5), (c_4, c_5)] \\
c_0 \sqcup c_5 &= \top \\
Worklist &= [(c_4, c_5)] \\
c_4 \sqcup c_5 &= \top \\
Worklist &= \emptyset
\end{aligned}
$$

Figure 42: Bottom-up computation of concepts for the mammals example.

3.3 Using Concept Analysis to Identify Modules

The main idea of this chapter is to apply concept analysis to the problem of identifying potential modules in legacy code. An outline of the process is as follows:

1. Build a context, where objects are functions defined in the input program and attributes are properties of those functions. The attributes could be any of several properties relating the functions to data structures. Attributes are discussed in more detail below.

2. Construct the concept lattice from the context, as described in Section 3.2.

3. Identify *concept partitions*—collections of concepts whose extents partition the set of objects. Each concept partition corresponds to a possible modularization of the input program. Concept partitions are discussed in Section 3.4.

3.3.1 Concept analysis and the stack and queue example

Consider the stack and queue example from the introduction. In this section, we will demonstrate how concept analysis can be used to identify the module partition indicated by the C++ code in Figure 38 on page 85.

First, we define a context as shown in Table 4. The next step is to build the concept lattice from the context, as described in Section 3.2. The concept lattice for the stack and queue example is shown in Figure 43. Table 5 shows the extent and intent of the concepts corresponding to the nodes in the lattice.

One of the advantages of using concept analysis is that multiple possibilities for

	returns `stack`	*returns* `queue`	*has* `stack` *arg.*	*has* `queue` *arg.*	*uses* `stack` *fields*	*uses* `queue` *fields*
`initStack`	√				√	
`initQ`		√				√
`isEmptyS`			√		√	
`isEmptyQ`				√		√
`push`			√		√	
`enq`				√		√
`pop`			√		√	
`deq`				√		√

Table 4: The context for the stack and queue example.

top	(*all objects*, $\emptyset$)
c_5	({`initQ`, `isEmptyQ`, `enq`, `deq`}, {uses `queue` fields})
c_4	({`initStack`, `isEmptyS`, `push`, `pop`}, {uses `stack` fields})
c_3	({`isEmptyQ`, `enq`, `deq`}, {has `queue` argument, uses `queue` fields})
c_2	({`isEmptyS`, `push`, `pop`}, {has `stack` argument, uses `stack fields`})
c_1	({`initQ`}, {returns `queue`})
c_0	({`initStack`}, {returns `stack`})
bot	($\emptyset$, *all attributes*)

Table 5: The extent and intent of the concepts for the stack/queue example.

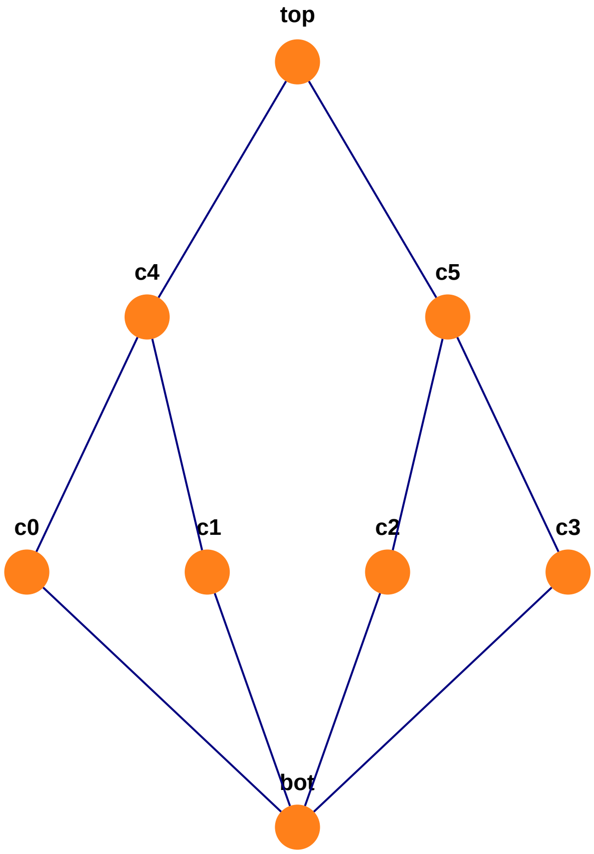

Figure 43: The concept lattice for the stack and queue example.

modularization are offered. In addition, the relationships among concepts in the concept lattice also offers insight into the structure within proposed modules. For example, at the atomic level, initialization functions (concepts c_0 and c_1) are distinct concepts from other functions (concepts c_2 and c_3). The former two concepts correspond to constructors and the latter two to sets of member functions. Concept c_4 corresponds to a stack module and c_5 corresponds to a queue module. The subconcept relationships $c_0 \subseteq c_4$ and $c_2 \subseteq c_4$ indicate that the stack concept consists of a constructor concept and a member-function concept.

3.3.2 Adding complementary attributes

The stack and queue example, as considered thus far, has not demonstrated the full power that concept analysis brings to the modularization problem. It is relatively straightforward to separate the code shown in Figure 37 into two modules, and techniques such as those described in [10, 67] will also create the same grouping. We now show that concept analysis offers the possibility to go beyond previously defined methods: It offers the ability to tease apart code that is, in some sense, more "tangled".

To illustrate what we mean by more tangled code, consider a slightly modified stack and queue example. Suppose the functions `isEmptyQ` and `enq` have been written so that they modify the stack fields directly, as in Figure 44, rather than calling `isEmptyS` and `push`. While this may be more efficient, it makes the code more difficult to maintain—simple changes in the stack implementation may require changes in the queue code. Furthermore, it complicates the process of identifying separate modules. If we apply concept analysis using the same set of attributes as we did above, attribute "uses `stack`

```
int isEmptyQ(struct queue* q) {
  return (q->front->sp == q->front->base
    && q->back->sp == q->back->base); }

void enq(struct queue* q, int i) {
  *(q->front->sp) = i;
  q->front->sp++; }
```

Figure 44: "Tangled" `isEmptyQ` and `enq` functions.

fields" now applies to `isEmptyQ` and `enq`. Table 6 shows the context relation for the tangled stack and queue code with the original sets of objects and attributes. The resulting concept lattice is shown in Figure 45. (The extent and intent of the concepts corresponding to the nodes in the lattice are shown in Table 7.) Observe that concept c_5 can still be identified with a queue module, but none of the concepts coincide with a stack module. In particular, even though the extent of c_0 is $\{\texttt{initStack}\}$ and the extent of c_2 is $\{\texttt{isEmptyS}, \texttt{push}, \texttt{pop}\}$, the concept $c_0 \sqcup c_2 = c_7$ is not the stack concept:

$$\tau(\sigma(\{\texttt{initStack}, \texttt{isEmptyS}, \texttt{push}, \texttt{pop}\})) =$$

$$\{\texttt{initStack}, \texttt{isEmptyS}, \texttt{isEmptyQ}, \texttt{push}, \texttt{enq}, \texttt{pop}\}$$

In other words, the extent of c_7 mixes the stack operations with some, but not all, of the queue operations.

The problem is that the attributes in this context reflect only "positive" information. A distinguishing characteristic of the stack operations is that they depend on the fields of `struct stack` but *not* on the fields of `struct queue`. To "untangle" these components, we need to augment the set of attributes with "negative" information—in this case, we add a new attribute—the complement of "uses `queue` fields" (i.e.,

	returns `stack`	*returns* `queue`	*has* `stack` *arg.*	*has* `queue` *arg.*	*uses* `stack` *fields*	*uses* `queue` *fields*
`initStack`	√				√	
`initQ`		√				√
`isEmptyS`			√		√	
`isEmptyQ`				√	√	√
`push`			√		√	
`enq`				√	√	√
`pop`			√		√	
`deq`				√		√

Table 6: The context for the "tangled" stack and queue example.

top	(*all objects*, $\emptyset$)
c_7	({`initStack`, `isEmptyS`, `isEmptyQ`, `push`, `enq`, `pop`}, {uses `stack` fields})
c_5	({`initQ`, `isEmptyQ`, `enq`, `deq`}, {uses `queue` fields})
c_3	({`isEmptyQ`, `enq`, `deq`}, {has `queue` argument, uses `queue` fields})
c_6	({`isEmptyQ`, `enq`}, {has `queue` arg., uses `queue` fields, uses `stack` fields})
c_2	({`isEmptyS`, `push`, `pop`}, {has `stack` argument, uses `stack fields`})
c_1	({`initQ`}, {returns `queue`})
c_0	({`initStack`}, {returns `stack`})
bot	($\emptyset$, *all attributes*)

Table 7: The extent and intent of the concepts for the tangled stack/queue example.

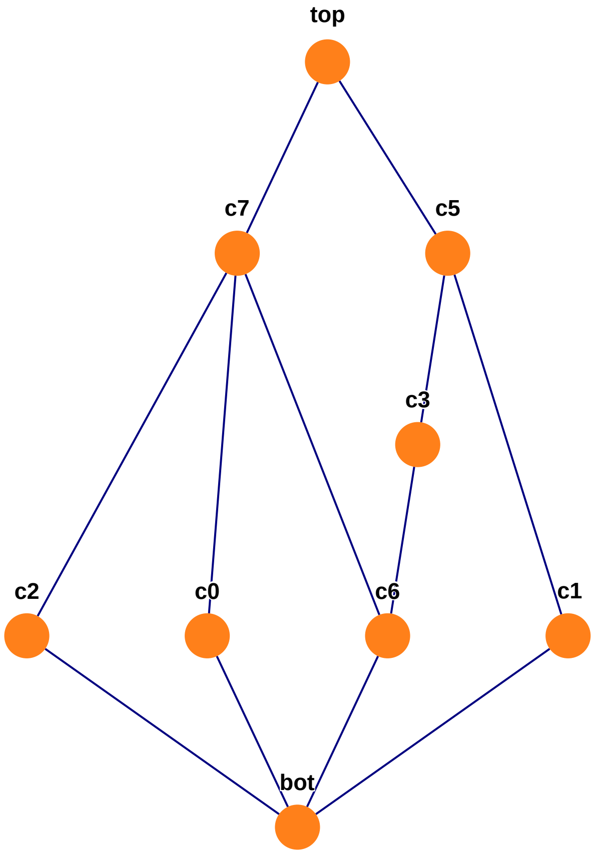

Figure 45: The concept lattice for the tangled stack and queue example.

	returns `stack`	*returns* `queue`	*has* `stack` *arg.*	*has* `queue` *arg.*	*uses* `stack` *fields*	*uses* `queue` *fields*	*not* `queue` *fields*
`initStack`	√				√		√
`initQ`		√				√	
`isEmptyS`			√		√		√
`isEmptyQ`				√	√	√	
`push`			√		√		√
`enq`				√	√	√	
`pop`			√		√		√
`deq`				√		√	

Table 8: The context for the "untangled" stack and queue example.

"does *not* use `queue` fields"). The corresponding context is shown in Table 8. The resulting concept lattice is shown in Figure 46. (The extent and intent of the concepts corresponding to the nodes in the lattice are shown in Table 9.) This concept lattice contains all of the concepts in the concept lattice from Figure 45, as well as an additional concept, c_4, which corresponds to a stack module. This modularization identifies `isEmptyQ` and `enq` as being part of a queue module that is separate from a stack module, even though these two operations make direct use of stack fields.

This example raises some issues for the subsequent C-to-C++ code-transformation phase. Although one might be able to devise transformations to remove these dependences of queue operations on the private members of the stack class (e.g., by introducing appropriate calls on member functions of the stack class), a more straightforward C-to-C++ transformation would simply use the C++ friend mechanism, as shown in Figure 47.

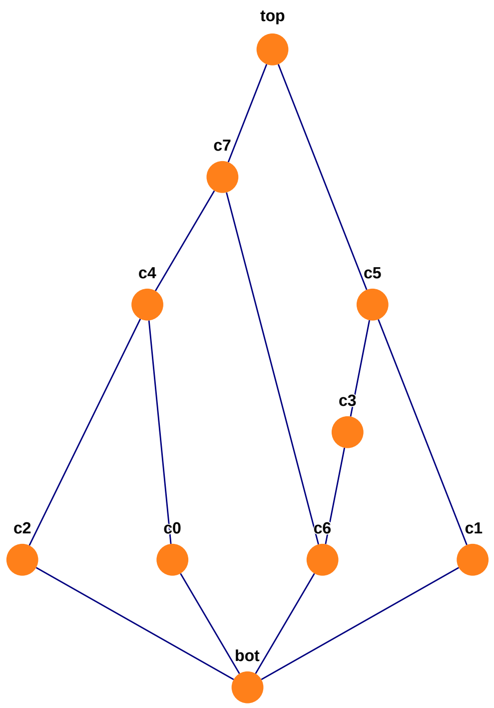

Figure 46: The concept lattice for the untangled stack and queue example.

top	(*all objects*, $\emptyset$)
c_7	({`initStack`, `isEmptyS`, `isEmptyQ`, `push`, `enq`, `pop`}, {uses `stack` fields})
c_5	({`initQ`, `isEmptyQ`, `enq`, `deq`}, {uses `queue` fields})
c_4	({`initStack`, `isEmptyS`, `push`, `pop`}, {uses `stack` fields, *not* `queue` fields})
c_3	({`isEmptyQ`, `enq`, `deq`}, {has `queue` argument, uses `queue` fields})
c_6	({`isEmptyQ`, `enq`}, {has `queue` arg., uses `queue` fields, uses `stack` fields})
c_2	({`isEmptyS`, `push`, `pop`}, {has `stack` arg., uses `stack` fields, *not* `queue` fields})
c_1	({`initQ`}, {returns `queue`})
c_0	({`initStack`}, {returns `stack`, does *not* use `queue` fields})
bot	($\emptyset$, *all attributes*)

Table 9: The extent and intent of the concepts for the untangled stack/queue example.

```
const int QUEUE_SIZE = 10

class queue;

class stack {
  friend class queue;
private:
  int* base;
  int* sp;
  int size;
public:
  stack(int sz) { base = sp = new int[sz]; size = sz; }
  int isEmpty() { return (sp == base); }
  int pop() {
    if (isEmpty()) return -1;
    sp--;
    return (*sp); }
  void push(int i) { *sp = i; sp++; } // no overflow check
};

class queue {
private:
  stack *front, *back;
public:
 queue() {
   front = new stack(QUEUE_SIZE); back = new stack(QUEUE_SIZE);
}
  int isEmptyQ() {
    return (front->sp == front->base && back->sp == back->base); }
  void enq(int i) {
    *(front->sp) = i;
    front->sp++; }
  int deq() {
    if (isEmpty()) return -1;
    if (back->isEmpty())
      while(!front->isEmpty()) back->push(front->pop());
    return back->pop(); }
};
```

Figure 47: Queue and stack classes in C++ with friends.

3.3.3 Other choices for attributes

A concept is a maximal collection of objects having common properties. A cohesive module is a collection of functions (perhaps along with a data structure) having common properties. Therefore, when employing concept analysis to the modularization problem, it is reasonable to have objects correspond to functions.[2] However, we have more flexibility when it comes to attributes. There are a wide variety of attributes we might choose in an effort to identify concepts (modules) in a program. Our examples have used attributes that reflect the way `struct` data types are used. But in some instances, it may be useful to use attributes that capture other properties. Other possibilities for attributes include the following:

- *Variable-usage information*: Related functions can sometimes be identified by their use of common global variables. An attribute capturing this information might be of the form "uses global variable `x`" [40, 57].
- *Dataflow and slicing information* can be useful in identifying modules. Attributes capturing this information might be of the form "may use a value that flows from statement s" or "is part of the slice with respect to statement s".
- *Information obtained from type inferencing*: Type inference can be used to uncover distinctions between seemingly identical types (see [58, 51]). For example, if f is a function declared to be of type $\texttt{int} \times \texttt{int} \rightarrow \texttt{bool}$, type inference might discover that f's most general type is of the form $\alpha \times \beta \rightarrow \texttt{bool}$. This reveals

[2] Some legacy code is monolithic—multiple tasks are contained within one function. In such cases, it may be preferable to have objects correspond to slices [64, 32] rather than functions.

that the type of f's first argument is distinct from the type of its second argument (even though they had the same declared type). Attributes might then be of the form "has argument of type α" rather than simply "has argument of type `int`". This would prevent functions from being grouped together merely because of superficial similarities in the declared types of their arguments.

- *Disjunctions of attributes*: The user may be aware of certain properties of the input program, perhaps the similarity of two data structures. Disjunctive attributes allow the user to specify properties of the form "π_1 or π_2". For example, "uses fields of stack or uses fields of queue".

Any or all of these attributes could be used together in one context. This highlights one of the advantages of the concept-analysis approach to modularization: It represents not just a single *algorithm* for modularization; rather, it provides a *framework* for obtaining a collection of different modularization algorithms.

3.4 Concept Partitions

Thus far, we have discussed how a concept lattice can be built from a program in such a way that concepts represent potential modules. However, because of overlaps between concepts, not every group of concepts represents a potential modularization.

For the above examples involving stacks and queues, it is straightforward to look at the small collection of concepts and find the desired modules by hand. In practice, programs to be modularized will be much larger. Modularization is unlikely to ever be a fully automated process, but it would be helpful to have a tool that suggests a palatable number of potential modularizations from which a software engineer might reasonably choose. Not all combinations of potential modules work together. Some concepts *overlap*. That is, a function may appear in multiple concepts, but it should only reside in one module.

Feasible modularizations are partitions: collections of modules that are disjoint, but include all the functions in the input code. To limit the number of choices that a software engineer would be presented with, it is helpful to identify such partitions.

Below, the notion of a *concept partition* is made more formal and an algorithm to identify such partitions from a concept lattice is presented.

3.4.1 Concept Partitions

Given a context $(\mathcal{O}, \mathcal{A}, \mathcal{R})$, a *concept partition* is a set of concepts whose extents form a partition of $\mathcal{O}$. That is, $P = \{(X_0, Y_0), \ldots, (X_{k-1}, Y_{k-1})\}$ is a concept partition iff the extents of the concepts *cover* the object set (i.e. $\bigcup X_i = \mathcal{O}$) and are pairwise disjoint ($X_i \cap X_j = \emptyset$ for $i \neq j$ and $X_i, X_j \in P$). In terms of modularizing a program, a concept partition corresponds to a collection of modules such that every function in the program is associated with exactly one module.

As a simple example, consider the concept lattice shown in Figure 46 on page 100. The concept partitions for that context are listed below:

P_1	$\{c_0, c_1, c_2, c_3\}$
P_2	$\{c_0, c_2, c_5\}$
P_3	$\{c_1, c_3, c_4\}$
P_4	$\{c_4, c_5\}$
P_5	$\{\text{top}\}$

P_1 is the *atomic partition.* P_2 and P_3 are combinations of atomic concepts and larger concepts. P_4 consists of one stack module and one queue module. P_5 is the trivial partition: All functions are placed in one module.

By looking at concept partitions, the software engineer can eliminate nonsensical possibilities. In the preceding example, c_7 does not appear in any partition—if it did, then to what module (i.e., nonoverlapping concept) would `deq` belong?

An *atomic partition* of a concept lattice is a concept partition consisting of exactly the atomic concepts. (Recall that the atomic concepts are the concepts with smallest extent containing each of the objects treated as a singleton set. For instance, see the atomic concepts in the mammal example on page 90 in Section 3.2.) A concept lattice need not have an atomic partition. For example, the lattice in Figure 39 on page 88 does not have an atomic partition: The atomic concepts are c_0, c_1, c_2, and c_3; however, c_1 and c_3 overlap—the object "gibbons" is in the extent of both concepts.

When it exists, the atomic partition of a concept lattice is often a good starting point for choosing a modularization of a program. In order to develop tools to work with concept partitions, it is useful to be able to guarantee the existence of atomic partitions. Contexts that result in atomic concepts that, in turn, form a concept partition can be characterized precisely by the following definition: a context $(\mathcal{O}, \mathcal{A}, \mathcal{R})$

is *well-formed* if and only if, for every pair of elements $x, y \in \mathcal{O}$, $\sigma(\{x\}) \subseteq \sigma(\{y\})$ implies $\sigma(\{x\}) = \sigma(\{y\})$.

While not every context results in a concept lattice that has an atomic partition, we can *extend* any context—by adding additional attributes—to make it well-formed. Informally, a context extension is another context over the same set of objects (but a possibly augmented set of attributes) whose concept lattice offers as least as many ways of grouping the objects as did the lattice derivable from the original context. More formally, a context $(\mathcal{O}, \mathcal{A}', \mathcal{R}')$ is an *extension* of context $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ if and only if $\mathcal{A} \subseteq \mathcal{A}'$ and $\mathcal{R} \subseteq \mathcal{R}'$.

There are several ways in which a non-well-formed context can be extended into a well-formed context. The important step in any such process is to identify the 'offending' pairs of objects x and y for which $\sigma(\{x\}) \subsetneq \sigma(\{y\})$. This inequity may be counterbalanced by the addition of an attribute such that, in the resulting context, $\sigma(\{x\}) \not\subseteq \sigma(\{y\})$. Two such ways to extend a context to well-formed context are described below:

1. A context can be extended via the addition of *unique identification attributes*: for each pair of objects, x, y such that $\sigma(\{x\}) \subsetneq \sigma(\{y\})$, a new attribute a_x that uniquely identifies x is added to the extended attribute set. x becomes the *only* object that has attribute a_x in the extended context relation (i.e., $\tau(\{a_x\}) = \{x\}$).

   As an example, consider the mammal context shown in Table 2 on page 86. The context is not well-formed because the attributes of "human" are a proper subset of the attributes of "gibbon". To make a *uniquely attributed* extension, we augment the attribute set to include the attribute a_{human}. The resulting

	four-legged	hair-covered	intelligent	marine	thumbed	a_{human}
cats	$\surd$	$\surd$				
dogs	$\surd$	$\surd$				
dolphins			$\surd$	$\surd$		
gibbons		$\surd$	$\surd$		$\surd$	
humans			$\surd$		$\surd$	$\surd$
whales			$\surd$	$\surd$		

Table 10: The uniquely-attributed extension of the mammal context shown in Table 2 on page 86.

context is shown in Table 10. The resulting concept lattice shown in Figure 48. Table 11 shows the extent and intent corresponding to the nodes in the lattice. Table 12 shows the partitions of the lattice. Partition P_1 is the atomic partition.

2. A context can be extended to a well-formed context by augmenting a context with negative information (similar to what is done in Section 3.3.2).

   Given a context $(\mathcal{O}, \mathcal{A}, \mathcal{R})$, a *complement* of an attribute $a \in \mathcal{A}$ is an attribute $\overline{a}$ such that $\tau(\{\overline{a}\}) = \{x \in \mathcal{O} | (x, a) \notin \mathcal{R}\}$. That is, $\overline{a}$ is an attribute of exactly the objects that do not have property a. For example, the attribute "does *not* use `queue` fields" from the tangled stack and queue example (page 99) is the complement of the attribute "uses `queue` fields".

   The *complemented extension* of a context $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ is a new context $(\mathcal{O}, \mathcal{A}', \mathcal{R}')$ formed by the algorithm in Figure 50.

   For example, we can form a different well-formed extension of the mammal context shown in Table 2 (page 86) by creating the complemented extension. To make the complemented extension, we augment the attribute set to include the

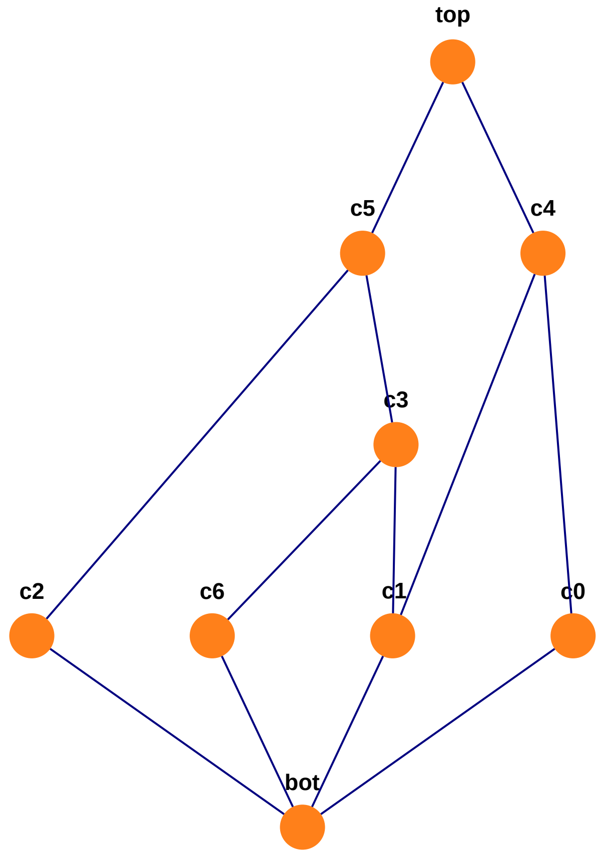

Figure 48: The concept lattice for the uniquely-attributed mammal example.

top	$(\{\text{cats}, \text{gibbons}, \text{dogs}, \text{dolphins}, \text{humans}, \text{whales}\}, \emptyset)$
c_5	$(\{\text{gibbons}, \text{dolphins}, \text{humans}, \text{whales}\}, \{\text{intelligent}\})$
c_4	$(\{\text{cats}, \text{dogs}, \text{gibbons}\}, \{\text{hair-covered}\})$
c_3	$(\{\text{gibbons}, \text{humans}\}, \{\text{intelligent}, \text{thumbed}\})$
c_2	$(\{\text{dolphins}, \text{whales}\}, \{\text{intelligent}, \text{marine}\})$
c_6	$(\{\text{humans}\}, \{\text{intelligent}, \text{thumbed}, a_{\text{human}}\})$
c_1	$(\{\text{gibbons}\}, \{\text{hair-covered}, \text{intelligent}, \text{thumbed}\})$
c_0	$(\{\text{cats}, \text{dogs}\}, \{\text{hair-covered}, \text{four-legged}\})$
bot	$(\emptyset, \{\text{four-legged}, \text{hair-covered}, \text{intelligent}, \text{marine}, \text{thumbed}, a_{\text{human}}\})$

Table 11: The extent and intent of the concepts for the uniquely-attributed mammal example.

P_1	$\{c_0, c_1, c_2, c_6\}$
P_2	$\{c_2, c_6, c_4\}$
P_3	$\{c_0, c_2, c_3\}$
P_4	$\{c_0, c_5\}$
P_5	$\{\text{top}\}$

Table 12: Concept partitions corresponding to the concept lattice in Figure 48. P_1 is the atomic partition.

	four-legged	hair-covered	intelligent	marine	thumbed	not hair-covered
cats	$\surd$	$\surd$				
dogs	$\surd$	$\surd$				
dolphins			$\surd$	$\surd$		$\surd$
gibbons		$\surd$	$\surd$		$\surd$	
humans			$\surd$		$\surd$	$\surd$
whales			$\surd$	$\surd$		$\surd$

Table 13: The complemented extension of the mammal context shown in Table 2 on page 86.

> complementary attribute "not hair-covered". The resulting context is shown in Table 13. The resulting concept lattice shown in Figure 49. Table 14 shows the extent and intent corresponding to the nodes in the lattice. Table 15 shows the partitions of the lattice. Partition P_1 is the atomic partition.

It should be clear that both forms of extension result in well-formed contexts.

Both uniquely-attributed extensions and complemented extensions result in a concept lattice with at least as many (and frequently many more) nodes than the lattice derived from the original context. We say that a concept lattice $\mathcal{L}'$ derived from a $\mathcal{C}' = (\mathcal{O}, \mathcal{A}', \mathcal{R}')$ is an *extension* of a concept lattice $\mathcal{L}$ derived from a $\mathcal{C} = (\mathcal{O}, \mathcal{A}, \mathcal{R})$ if $\mathcal{A} \subseteq \mathcal{A}'$, and for every concept c in $\mathcal{L}$, there is a concept c' in $\mathcal{L}'$ with the same extent. More formally, if $X \subseteq \mathcal{O}$ such that $\tau(\sigma(X)) = X$, then $\tau'(\sigma'(X)) = X$ where τ' and σ' are the common-object and common-attribute relations, respectively, for context $\mathcal{C}'$.

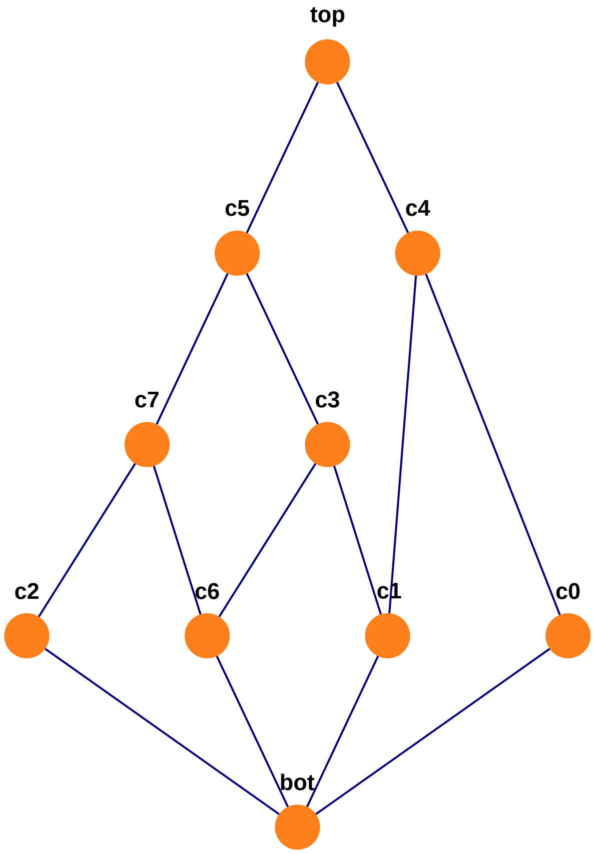

Figure 49: The concept lattice for the complemented mammal example.

$\mathcal{A}' \leftarrow \mathcal{A}$
$\mathcal{R}' \leftarrow \mathcal{R}$
`while` $(\mathcal{O}, \mathcal{A}', \mathcal{R}')$ `is not well formed do`
    `let` $x, y \in \mathcal{O}$ `be such that` $\sigma(\{x\}) \subsetneq \sigma(\{y\})$
    `let` $a \in \mathcal{A}'$ `be such that` $a \notin \sigma(\{x\})$`,` $a \in \sigma(\{y\})$
    $\mathcal{A}' \leftarrow \mathcal{A}' \cup \{\overline{a}\}$`, where` $\overline{a}$ `is a new attribute`
    $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{(x, \overline{a}) | (x, a) \notin \mathcal{R}'\}$
`endwhile`

Figure 50: An algorithm to compute the complemented extension of a context.

top	$(\{\text{cats, gibbons, dogs, dolphins, humans, whales}\}, \emptyset)$
c_5	$(\{\text{gibbons, dolphins, humans, whales}\}, \{\text{intelligent}\})$
c_4	$(\{\text{cats, dogs, gibbons}\}, \{\text{hair-covered}\})$
c_7	$(\{\text{dolphins, humans, whales}\}, \{\text{not hair-covered}\})$
c_3	$(\{\text{gibbons, humans}, \{\text{intelligent, thumbed}\})$
c_2	$(\{\text{dolphins, whales}\}, \{\text{intelligent, marine, not hair-covered}\})$
c_6	$(\{\text{humans}\}, \{\text{intelligent, thumbed, not hair-covered}\})$
c_1	$(\{\text{gibbons}\}, \{\text{hair-covered, intelligent, thumbed}\})$
c_0	$(\{\text{cats, dogs}\}, \{\text{hair-covered, four-legged}\})$
bot	$(\emptyset, \{\text{four-legged, hair-covered, intelligent, marine, thumbed, not hair-covered}\})$

Table 14: Extents and intents for the complemented mammal example.

P_1	$\{c_0, c_1, c_2, c_6\}$
P_2	$\{c_2, c_6, c_4\}$
P_3	$\{c_0, c_1, c_7\}$
P_4	$\{c_0, c_2, c_3\}$
P_5	$\{c_7, c_4\}$
P_6	$\{\text{top}\}$

Table 15: Concept partitions corresponding to the concept lattice in Figure 49. P_1 is the atomic partition.

Given a context $\mathcal{C}$, both uniquely-attributed extensions and complemented extensions of $\mathcal{C}$ result in concept lattices that are extensions of the lattice derived from $\mathcal{C}$. In both cases, attributes are added to the extended context to make in easier to distinguish objects. For example, if $\sigma\{x\} \subsetneq \sigma\{y\}$ then there is at least one attribute which is a property of y that is not a property of x. Whether adding unique attributes or complementary attributes, negative information is represented in a positive form in the extended context to help distinguish such x and y.

3.4.2 Finding Partitions from A Concept Lattice

We say that concept lattice derived from a well-formed context is a well-formed concept lattice. Given a well-formed concept lattice, we define the following relations on its elements:

A concept d *covers* concept c if $c < d$ and there is no concept e such that $c < e < d$. If d covers c, we say "c is covered by d". The set of covers of concept c, denoted by $\mathrm{covs}(c)$, is the set of lattice elements d such that d covers c. The set of elements *subordinate* to d, denoted by $\mathrm{subs}(c)$, is the set of lattice elements c such that $c < d$.

The algorithm builds up a collection of all the partitions of a concept lattice. Let P be the collection of partitions that we are forming. Let W be a worklist of partitions. We begin with the atomic partition, which is the covers of the bottom element of the concept lattice. P and W are both initialized to the singleton set containing the atomic partition.

The algorithm works by considering partitions from worklist W until W is empty. For each partition removed from W, new partitions are formed (when possible) by

```
[1]   A ← covs(⊥)                  // the atomic partition
[2]   P ← {A}
[3]   W ← {A}
[4]   while W ≠ ∅ do
[5]     remove some p from W
[6]     for each c ∈ p
[7]       for each c′ ∈ covs(c)
[8]         p′ ← p − subs(c′)
[9]         if (⋃p′) ∩ c′ = ∅       // if p′ and c′ are disjoint
[10]          p″ ← p′ ∪ {c′}
[11]          if p″ ∉ P
[12]            P ← P ∪ {p″}
[13]            W ← W ∪ {p″}
[14]          endif
[15]        endif
[16]      endfor
[17]    endfor
[18]  endwhile
```

Figure 51: An algorithm to find the partitions of a well-formed concept lattice.

selecting a concept of the partition, choosing a cover of that concept, adding it to the partition, and removing overlapping concepts. The algorithm is given in Figure 51.

As an example, consider the atomic partition of the concept lattice derived from the uniquely-attributed mammal context (see Figure 48). The algorithm begins with the atomic partition (consisting of concepts, c_0, c_1, c_2, and c_6) as the sole member of the worklist. The algorithm removes the atomic partition from the worklist, as p in line [5] of Figure 51. Suppose that in the first iteration of the for loop in line [6], c refers to c_0. The covering set of c_0 is the singleton set consisting of c_4, so c' is assigned c_4 in line [7]. In line [8], p' is assigned the value of p minus the subordinate concepts of c_4 (i.e., c_1, c_0, and bottom), so p' is $\{c_2, c_6\}$. In line [9], the union of the extents of c_2 and

c_6 is disjoint with the extent of c_4; thus, in line [10], the partition $p'' = \{c_2, c_6\} \cup \{c_4\}$ is formed. p'' is added to the set of partitions and to the worklist in line [12] and line [13].

In the worst case, the number of partitions can be exponential in the number of concepts. Furthermore, the techniques for making contexts well-formed, discussed in Section 3.4.1, only exacerbate the problem: More precise means of distinguishing sets of objects translates to more concepts. This is turns leads to more possibilities for partitions.

If the number of concepts in a concept lattice is large, it may be impractical to consider every possible partition of the concepts. In such a case, it is possible to adapt the algorithm to work interactively, with guidance from the user. Before attempting to find a new partition, the algorithm would pause for the user to specify *seed sets* of concepts, which would be used to force the algorithm to find only coarser partitions than the seed sets (i.e., partitions that do not subdivide the seed sets).

A proof of the correctness of the algorithm can be found in Appendix A.1.

3.5 Implementation and Results

We have implemented a prototype tool that employs concept analysis to discover potential modularizations of C programs. Our implementation is actually a collection of tools, working together as depicted in Figure 52. As the diagram indicates, the tool consists of five components:

1. The AST builder ("FrontEndC") takes a compilable, preprocessed C program and generates an abstract syntax tree that is annotated with type information.

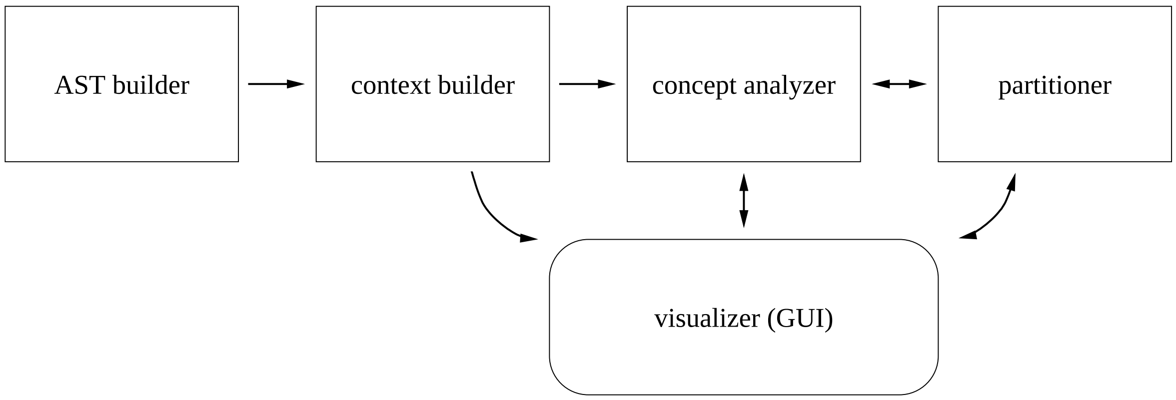

Figure 52: A C concept-analysis tool: the different components of our implementation.

2. The context builder takes as input a typed abstract syntax tree and generates formal contexts by traversing the tree. This component consists of a collection of functions allowing for contexts to be constructed based on a wide assortment of objects and attributes, including, but not limited to, the following criteria:

   - Objects are functions, attributes are types. (f, t) is in the context relation if and only if an expression of type t occurs in function f.
   - Objects are functions, attributes are aggregate types (i.e., `struct` or `union` types). (f, t) is in the context relation if and only if a field selector (i.e., `.` or `->`) is applied to an expression of type t in f. (This is the kind of context used to produce the data in Table 16 on page 119.)
   - Objects are functions, attributes are aggregate types. (f, t) is in the context relation if and only if f has a parameter type or return type that uses t. (This is the kind of context used to produce the data in Table 17 on page 120.)
   - Objects are functions, attributes are global variables. (f, v) is in the context

relation if and only if f uses v.

- Objects are labels of case statements. The attributes are functions. (c, f) is in the context relation if and only if a statement of the form "`case` c" occurs in function f. Labels are only counted as objects when they are textual labels (i.e. enumerated values)—integer and character constants are ignored, as is the `default` label.

The context builder also includes the ability to merge a collection of contexts into one context. This feature is quite useful for generating one context from many input C files. The context merger assumes that the attributes and objects across files are represented in a consistent fashion. For example, if an attribute represented by the string `"Stack"` is present in more than one context, it is assumed to refer to the same property (whether it means "uses field of type `Stack`", "returns type `Stack`, etc.) in each context in which it appears.

3. The concept analyzer takes as input a formal context and builds the concept lattice bottom up, as described in Section 3.2. The transitively reduced graph representation of the lattice is then computed. The concept analyzer also provides functions to compute common-attribute and common-object sets.

4. The partitioner takes as input a concept lattice and generates the set of all partitions of that lattice, as described in Section 3.4. In order to construct partitions, the concept lattice must be derived from a well-formed context. The partitioner offers two ways to transform an arbitrary context into a well-formed context: by adding unique attributes or by computing the complemented extension. Since the

number of partitions can be very large, the partitioner also provides the ability to compute partitions incrementally.

5. The visualizer has two main functions: to display context and concept lattice information in an interconnected fashion and to provide a graphical user interface through which the user can manipulate the other components of the C concept-analysis system.

The entire system is written in Standard ML except for the visualization component, which is written in Standard ML in conjunction with the SmlTK interface to Tcl/Tk.

The simple examples mentioned thus far in this chapter have been analyzed using our implementation. We have also applied the prototype tool on the integer portion of the SPEC 95 benchmark suite, as well as several other programs of comparable size.

As an example, consider the SPEC 95 benchmark *go* ("The Many Faces of Go"). The program consists of roughly 28,000 lines of C code, 372 functions, and 8 user-defined aggregate types. The concept lattice for the fully complemented context (i.e., the context including all the complements of the original attributes) associated with these functions and data types consists of thirty-four concepts and was constructed in less than thirty seconds of user time (using Standard ML of New Jersey version 110 on a SPARCstation 20 with 256MB of RAM running Solaris 2.5.1). The partitioner identified 63 possible partitions of the lattice in roughly the same amount of time.

Table 16 summarizes results from contexts where objects are functions, attributes are aggregate types (`struct` or `union`), and (f, t) is in the context relation if and only if a field selector (i.e., `.` or `->`) is applied to an expression of type t in f. The programs listed are a variety of programs that make significant use of `struct` types,

program	KLOC	objects	attributes	concepts
chull	1.0	26	3	9
bdd	2.5	51	13	20
go	28.0	372	16	34
Lucent code	160.0	45	44	95

Table 16: Results from applying concept analysis to various C programs. Objects are functions and attributes are of the form "uses the fields of `struct t`".

but do not necessarily take the `struct` types as arguments to the functions. (For example, in *go*, the variables of `struct` types are all global.) The programs include *go*, *chull* (computes the convex hull of a set of points), *bdd* (manipulates and manages binary decision diagrams) and some switching-configuration control code from Lucent Technologies.

Table 17 summarizes results from contexts where objects are functions, attributes are aggregate types, and (f, t) is in the context relation if and only if f has a parameter type or return type that uses t. In this case, a type t' 'uses' type t, if one of the following cases holds:

- $t' = t$
- t' is equivalent to t via a sequence of `typedef` associations
- t' is a pointer to a type that 'uses' t.

The programs listed are those from the integer portion of the SPEC 95 benchmark that make moderate to heavy use of aggregate types by passing them into and returning them from functions, as well as *bash*, a Unix shell, which also makes heavy use of `struct` types. The data indicates that from a *quantitative* point of view, concept

program	KLOC	objects	attributes	concepts
li	7.6	299	3	7
m88ksim	19.9	24	14	14
perl	26.9	189	17	42
ijpeg	31.2	407	29	48
vortex	67.2	654	48	93
bash	73.6	266	32	50
gcc	205.1	1,358	38	61

Table 17: Results from applying concept analysis to C programs from the SPEC benchmark (and *bash*). Objects are functions and attributes are of the form "uses `struct t` in parameter list or return type".

analysis is practical. In the worst case, the number of concepts can be exponential in the number of objects; however, in the cases we have studied, there are actually fewer concepts than objects.

The data shown in Table 16 and Table 17 is generated from contexts that are not well-formed, so the partition algorithm cannot be applied to the concept lattices. Instead, we use the two techniques for extending contexts to well-formed contexts described in Section 3.4. Table 18 summarizes the results of applying concept analysis to the well-formed extensions of the contexts used in Table 17, where the extensions are produced by adding unique identification attributes. Table 19 summarizes the results of applying concept analysis to the complemented extensions of the contexts used in Table 17.

The data from the well-formed extensions indicate that it is more tractable to form concept lattices (and to generate partitions from the lattices) from well-formed contexts made by adding unique identification attributes than it is from complemented extensions. The reason for this is that while the number of attributes rises sharply

program	KLOC	objects	attributes	concepts	partitions
li	7.6	299	299	303	6
m88ksim	19.9	24	26	26	9
perl	26.9	189	172	197	5,760
ijpeg	31.2	407	303	322	5,713
vortex	67.2	654	575	620	?
bash	73.6	266	211	229	231
gcc	205.1	1,358	1,288	1,311	?

Table 18: Results from applying concept analysis to C programs from the SPEC benchmark (and *bash*). Objects are functions and attributes are of the form "uses `struct t` in parameter list or return type" plus unique identification attributes. "?" indicates that the partitioning algorithm never terminated due to exponential behavior.

program	KLOC	objects	attributes	concepts	partitions
li	7.6	299	6	15	18
m88ksim	19.9	24	17	24	16
perl	26.9	189	32	13,826	?
ijpeg	31.2	407	50	?	?
vortex	67.2	654	91	?	?
bash	73.6	266	44	1,942	?
gcc	205.1	1,358	65	?	?

Table 19: Results from applying concept analysis to C programs from the SPEC benchmark (and *bash*). Objects are functions and attributes are of the form "uses `struct t` in parameter list or return type" plus complemented extensions. "?" indicates that the concept-lattice generator or the partitioning algorithm never terminated due to exponential behavior.

by adding unique identification attributes, the density of the context does not really change. On the other hand, complemented extensions have a much smaller increase in the number of attributes, but a much sharper rise in the density of the context. This is reflected in the increased number of concepts in the generated lattices. In some cases (e.g. *gcc*, *vortex*, *ijpeg*), the concept-lattice generation component did not complete even after running for several days.

In the worst case, the number of partitions of a given concept lattice can be exponential in the number of nodes (i.e., concepts) in the lattice. As the tables indicate, in several instances exponential behavior is apparently exhibited.

We now describe a case study.

Case study: *chull*

Chull is a program taken from a computational-geometry library that computes the convex hull of a set of vertices in the plane. The program consists of roughly one thousand lines of C code. It has twenty-six functions and three user-defined `struct` data types: `tVertex`, `tEdge`, and `tFace`, representing vertices, edges, and faces, respectively. Thus, one might hope that three modules—one for each `struct` type—would fall out naturally.

Initially, we formed the context consisting of the twenty-six functions as the object set and three attributes ("uses fields of `tVertex`", "uses fields of `tEdge`", and "uses fields of `tFace`"). The context is not well formed: For example, the attribute set of the `MakeEdge` function is the singleton set consisting of "uses fields of `struct tEdge`", which is a proper subset of the attributes of the `CleanEdges` function ("uses fields of `struct tEdge`" and "uses fields of `struct tFace`"). To extend the context to a well-formed context, we added unique attributes, such as $a_{\texttt{MakeEdge}}$. The resulting context consisted of the twenty-six functions as objects and a total of twenty-four attributes (the original three attributes plus twenty-one unique identification attributes). The resulting concept lattice had thirty concepts and is shown in Figure 53. The partitioning algorithm discovered eight partitions, of which one seemed particularly interesting: It consisted of eleven concepts, ten of which had singleton extents (corresponding to individual functions) and one concept that consisted of the sixteen functions that use the fields of `struct tVertex`. A possible interpretation of this partition as a modularization would be one vertex class (with sixteen member functions), and the remaining functions standing on their own, some of them being friend functions.

We also extended the original *chull* context to a well-formed context by adding complementary attributes. The resulting context consisted of the twenty-six functions as objects and a total of six attributes (the original three attributes plus their complements: "does not use fields of `struct tEdge`", etc.) The resulting concept lattice had twenty-eight nodes and is shown in Figure 54. The partitioning algorithm discovered 154 possible partitions of the concept lattice.

The atomic partition groups the functions into the eight concepts listed in Table 20. Thus, this partition does not break the code directly into three modules (e.g., one for each `struct` type). However, assuming that the goal is to transform *chull* into an equivalent C++ program, the eight concepts do suggest two possible modularizations: Concepts 2, 3, and 4 would correspond to three classes, for vertex, edge, and face, respectively; concept 1 would correspond to a "driver" module; and the functions in concepts 5 through 8 would form four "friend" modules, where each of the functions

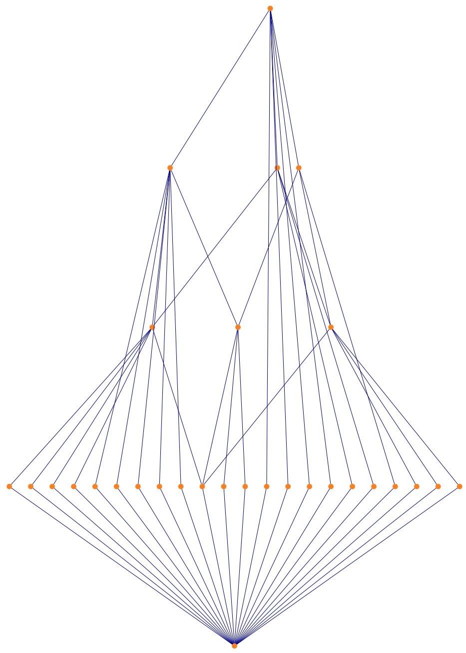

Figure 53: A concept lattice for *chull* after unique identification attributes were added.

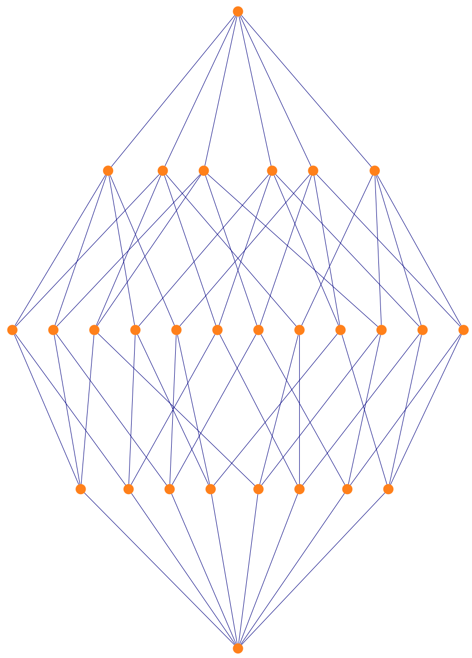

Figure 54: A concept lattice for *chull* after three complementary attributes were added.

concept number	user-defined struct types	functions
1	none	main, CleanUp, CheckEuler, PrintOut
2	tVertex	MakeVertex, ReadVertices, Collinear, ConstructHull, PrintVertices
3	tEdge	MakeEdge
4	tFace	CleanFaces, MakeFace
5	tVertex, tEdge	CleanVertices, PrintEdges
6	tVertex, tFace	Volume6, Volumed, Convexity, PrintFaces
7	tEdge, tFace	MakeCcw, CleanEdges, Consistency
8	tVertex, tEdge, tFace	Print, Tetrahedron, AddOne, MakeStructs, Checks

Table 20: The atomic partition of the concept lattice for *chull* shown in Figure 54.

would be declared to be a `friend` of the appropriate classes. Alternatively, one could group concepts 2–8 into a polyhedron class with nested vertex, edge, and face classes. Concept 1 would still represent a "driver" module. This possibility corresponds to one of the non-atomic partitions.

Another interesting partition is summarized in Table 21. It consists of four concepts. The first concept identifies the four functions not associated with any of the `struct` types. The second, third, and fourth concepts identify possible face, edge, and vertex classes, respectively. This partition comes close to our expectation (i.e., one module for each `struct` type); however, several functions that, from their names, we might have expected to end up in the edge and face modules ended up in the vertex module (e.g., `Tetrahedron`, `Volume6`, `Volumed`, `PrintEdges`, and `PrintFaces`). These member functions would need to be declared as `friends` of the other classes.

concept	user-defined `struct` types	functions
1	none	`main`, `CleanUp`, `CheckEuler`, `PrintOut`
4	`tFace`	`CleanFaces`, `MakeFace`
10	`tEdge`	`MakeCcw`, `MakeEdge`, `CleanEdges`, `Consistency`
26	`tVertex`	`MakeVertex`, `ReadVertices`, `Print`, `Tetrahedron`, `ConstructHull`, `AddOne`, `Volume6`, `Volumed`, `MakeStructs`, `CleanVertices`, `Collinear`, `Convexity`, `Checks`, `PrintVertices`, `PrintEdges`, `PrintFaces`

Table 21: A four-concept partition of the concept lattice for *chull* shown in Figure 54.

3.6 Concept Analysis and Software Architecture

Object-oriented principles such as encapsulation and modularity, recurring themes in this chapter, are examples of principles of software design at the level of computation. Software architecture ([25, 24]) is also an aspect of software design, but 'one-level further removed' from the detail. It is concerned with the big picture of large software systems: the ways in which various system components interact, rather than which datatypes and algorithms are affiliated with which classes. Concept analysis provides a natural way for attempting to identify facts about the underlying software architecture of a large system.

There are numerous directions in which one might apply concept analysis to software architecture. For example, objects could be software components (for example: parser, lexer, etc.) and attributes could be other components that are referred to either in design or in implementation by these components. Alternatively, attributes could be descriptors such as "I/O-dependent" or "numerical".

In this section, we describe how concept analysis can be applied to Standard ML programs where objects are ML functors and attributes are the signatures of functors and structures that are used by a given functor. In Standard ML, a *structure* is a collection of datatypes and functions, often used to represent an abstraction such as a stack. It can be compared with a class in C++, but is frequently used on a coarser scale to implement something resembling a collection of classes. A *functor* is a parameterized structure: given other structures it forms a new structure. A *signature* defines an interface for a structure or functor. Every structure and functor has a signature either explicitly or implicitly. An example of a structure, a signature, and a functor is shown in Figure 55. Observe how the structure `IntOrdKey` is explicitly declared to have signature `ORD_KEY`, while the functor `StackFn` is not given an explicit signature. Its implicit signature (determined by the Standard ML type-inference mechanism) is as follows:

```
sig
  type item
  type stack
  val empty : stack
  val push  : stack * item -> item
  val pop   : stack -> stack * item
end
```

Concept analysis can be applied to Standard ML to gain insight into common usage patterns of signatures and functors. We are interested in discovering relationships such as “functors X, Y, and Z always use structures with signatures A and B”’. Given that

```
signature ORD_KEY =
  sig
    type ord_key
    val compare : ord_key * ord_key -> order
  end

structure IntOrdKey : ORD_KEY =
  struct
    type ord_key = int
    fun compare (i, j) =
      if i < j then LESS
      else if i > j then GREATER
           else EQUAL
  end

functor StackFn (structure OrdKey : ORD_KEY) =
  struct
    type item = OrdKey.ord_key
    datatype stack = STACK of item list
    val empty = STACK(nil)
    fun push (STACK(l), i) = STACK(i::l)
    fun pop (STACK(l)) = (STACK(tl l), hd l)
  end
```

Figure 55: Examples of structures, signatures, and functors in Standard ML.

```
functor BuildAstFn(structure CModel : C_MODEL
                   structure Symbol : SYMBOL
                   ... ) : BUILD_AST =
  struct
    ...
    (* using unbound functor CTypeFn of signature C_TYPE *)
    structure CType = CTypeFn(structure Symbol = Symbol)
    ...
  end
```

Figure 56: The Standard ML functor `BuildAstFn`.

information, a programmer might reorganize a program, perhaps making use of a new structure or functor that ties A and B together.

As a case study, we describe how concept analysis can be employed to study the software architecture of a Standard ML system called *FrontEndC*, used to parse C programs into abstract syntax trees. For example, analysis of the functor `BuildAstFn` sketched in Figure 56 reveals that the object `BuildAstFn` has attributes `C_MODEL` and `SYMBOL` by the explicit mention of those signatures in the functor's parameter list as well as the attribute `C_TYPE` by the implicit mention of that signature by the instantiation of the functor `CTypeFn` within the body of `BuildAstFn`. Every functor is considered to use its signature, in this case indicated by the declaration `: BUILD_AST`.

FrontEndC consists of nineteen user-defined functors and signatures. The context formed has only twelve objects (i.e. functors) since we ignore those functors that use only their own signature. (We also ignore dependencies on signatures of structures and functors from the Standard ML of New Jersey Standard Library.) The resulting context is shown in Table 22. The concept lattice derived from this context is shown in Figure 57. The extents of the concepts can be identified as follows: The bottom node has empty extent and the extent of any other node is the union of the extents of its predecessors plus its label, if any. For example, the extent of the node labeled **CType** has extent {**BuildAst**, **Ast**, **TypeUtil**, **Sizeof**, **CType**}.

The lattice reveals some aspects of the software architecture of *FrontEndC*:

- There are clearly two major components: the lexing and parsing component that produces the parse tree and the abstract syntax tree (AST) building component that converts a parse tree to an AST.

- Each component has a 'driver' module: `ParseCFn` for the lexer/parser and `BuildAstFn` for the AST builder. These are the only two atomic concepts. If we were to have known in advance which functors and signatures deal with which half of the process we could have formed two concept lattices (they would look like the left and right halves of this lattice) and the two atomic concepts would be the bottom elements in those lattices.

- The overlap between these two major components is found at the middle of the lattice: both `ParseCFn` and `BuildAstFn` use a generic error module (signature `ERROR`) and both use the parse tree datatype given in the `C_MODEL` signature.

3.7 Related Work

Because modularization reflects a design decision that is inherently subjective, it is unlikely that the modularization process can ever be fully automated. Given that some user interaction will be required, the concept-analysis approach offers certain advantages over other previously proposed techniques (e.g., [42, 17, 49, 43, 9]), namely,

	Ast	*BuildAst*	*CLex*	*CLrVals*	*CType*	*CompileHelp*	*ParseC*	*Sizeof*	*SymbolTable*	*TDefTable*	*TokenTable*	*TypeUtil*
ADORNMENT	$\surd$	$\surd$										
AST	$\surd$	$\surd$										
BUILD_AST		$\surd$										
COMP_HELP			$\surd$	$\surd$		$\surd$	$\surd$				$\surd$	
C_LEX			$\surd$				$\surd$					
C_LR_VALS				$\surd$			$\surd$					
C_MODEL		$\surd$		$\surd$		$\surd$	$\surd$					
C_TOKENS			$\surd$				$\surd$				$\surd$	
C_TYPE	$\surd$	$\surd$			$\surd$			$\surd$				$\surd$
C_VARIANT			$\surd$	$\surd$			$\surd$			$\surd$		
ERROR		$\surd$					$\surd$					
GRND_SIZES		$\surd$						$\surd$				
PARSE_C							$\surd$					
SIZEOF		$\surd$						$\surd$				
SYMBOL	$\surd$	$\surd$			$\surd$				$\surd$			$\surd$
SYM_TAB	$\surd$	$\surd$							$\surd$			
TOKE_TABLE			$\surd$				$\surd$				$\surd$	
TYPE_DEFS						$\surd$	$\surd$			$\surd$		
TYPE_UTIL		$\surd$										$\surd$

Table 22: A context for *FrontEndC*. (For aesthetic reasons, attributes and objects are transposed from their traditional positions: attributes appear as row headings and objects appear as column headings.)

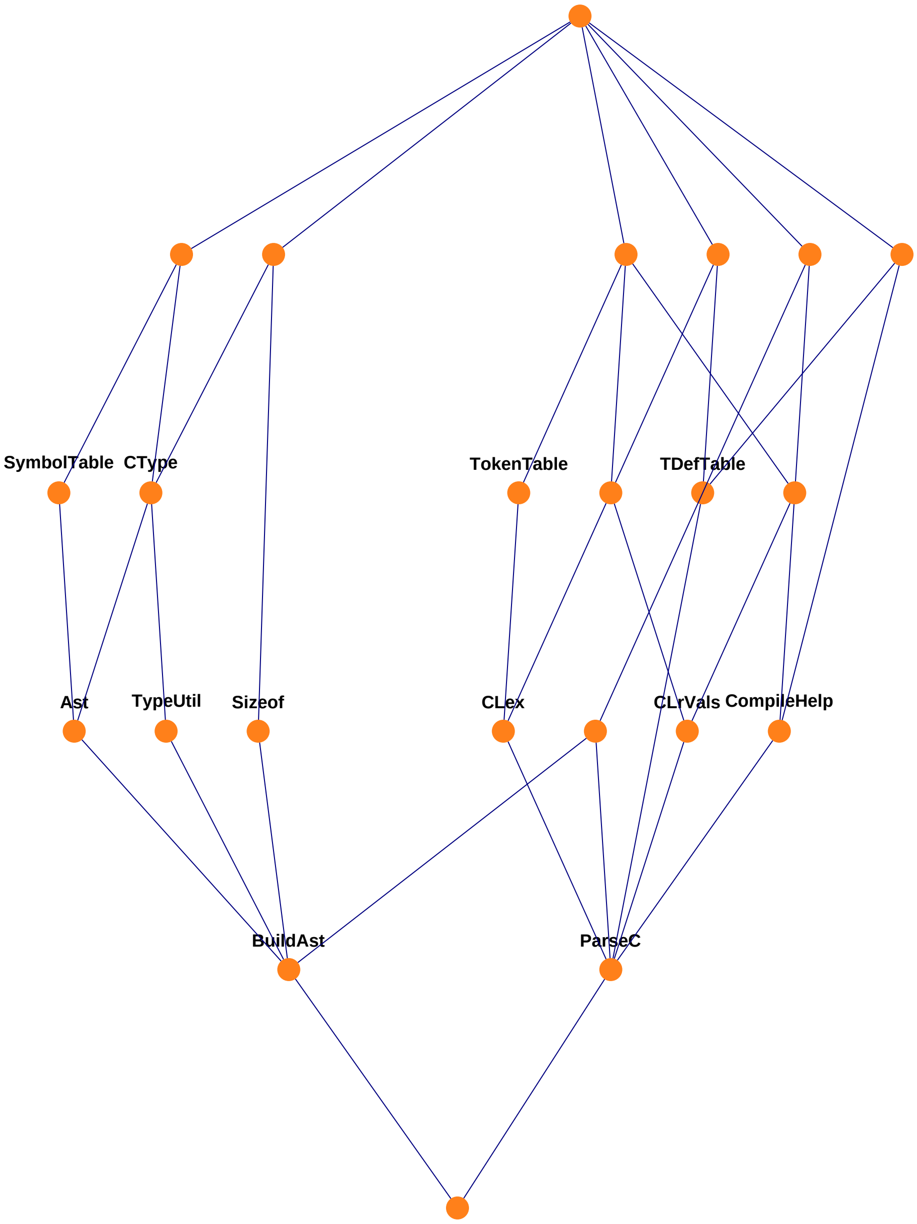

Figure 57: The concept lattice for the *FrontEndC* example.

the ability to "stay within the system" (as opposed to applying ad hoc methods) when the user judges that the modularization that the system suggests is unsatisfactory. If the proposed modularization is on too fine a scale, the user can "move up" the partition lattice. (See Section 3.4.) If the proposed modularization is too coarse, the user can add additional attributes to generate more concepts. (See Section 3.3.) Furthermore, concept analysis really provides a family of modularization algorithms: Rather than offering one fixed technique, different attributes can be chosen for different situations.

The reader is referred to [9, pp. 27–32] for an extensive discussion of the literature on the modularization problem. In the remainder of this section, we discuss only the work that is most relevant to the approach we have taken.

Liu and Wilde [42] make use of a table that is very much like the object-attribute relation of a context. However, whereas our work uses concept analysis to analyze such tables, Liu and Wilde propose a less powerful analysis. They also propose that the user intervene with ad hoc adjustments if the results of modularization are unsatisfactory. As explained above, the concept-analysis approach can naturally generate a variety of possible decompositions (i.e., different collections of concepts that partition the set of objects).

The concept-analysis approach is more general than that of Canfora et al. [10], which identifies abstract data types by analyzing a graph that links functions to their argument types and return types. The same information can be captured using a context, where the objects are the functions, and the attributes are the possible argument and return types (for example, the first four attributes in the context shown in Table 4 on page 93). By adding attributes that indicate whether fields of compound data types are used in a function, as is done in the example used in Section 3.3, concept-analysis becomes a more powerful tool for identifying potential modules than the technique described in [10].

The work described in [11] and [17] expands on the abstract-data-type identification technique described in [10]: Call and dominance information is used to introduce a hierarchical nesting structure to modules. It may be possible to combine the techniques from [11] and [17] with the concept-analysis approach of this chapter.

Canfora et al. discuss two types of links that cause undesirable clustering of functions [9]. The first type, "coincidental links", caused by routines that implement more than one function, can be overcome by program slicing [64, 32]. The second type, "spurious links", is caused by functions that access supporting data structures of more than one object type. In most of the approaches mentioned above, spurious links arise from a function that accesses several global variables of different types. The work described in [42, 17, 43, 67, 9] will all stumble on examples that exhibit spurious links. In our approach, an analogous kind of spurious link arises due to functions that access internal fields of more than one `struct`. An example is found in the tangled-code example discussed in Section 3.3.2, where the `enq` function uses the fields of both `struct stack` and `struct queue`. The additional discriminatory power of the concept-analysis approach is due to the fact that it is able to exploit both positive and negative information.

In contrast with the approach to identifying *objects* described in [2], our technique is aimed at analyzing relationships among functions and types to identify *classes*. In [2], the aim is to identify objects that link functions to specific variables. A similar effect can be achieved via concept analysis by introducing one attribute for each actual parameter.

There has been a certain amount of work involving the use of cluster analysis to identify potential modules (e.g., [33, 2, 39, 9]). This work (implicitly or explicitly) involves the identification of potential modules by determining a similarity measure among pairs of functions. We are currently investigating the link between concept analysis and cluster analysis.

[22] offers background on lattice theory and an introduction to concept analysis. [65] formalizes the notions of concept analysis and provides a proof of the fundamental theorem.

Concept analysis has been applied to many kinds of problems. Concept analysis was first applied to software engineering in the NORA/RECS tool, where it was used to identify conflicts in software-configuration information [60].

Contemporaneously with our own work, Lindig and Snelting [40] and Sahraoui et al. [57] independently explored the idea of applying concept analysis to the modularization problem. In both of these studies, the context relations used for concept analysis relate each function of the program to the global variables accessed by the function.

The results reported by Lindig and Snelting on two case studies of small to medium-sized Fortran and Cobol programs are not encouraging. In both cases, the concept lattice that resulted did not identify any useful ways to decompose the program into modules. However, we believe that the results achieved by our approach to using concept analysis are more promising than those of Lindig and Snelting and Sahraoui et al. This is due to several factors:

- The languages on which the techniques were applied—i.e., Fortran and Cobol (in the case of Lindig and Snelting) versus C. The C-to-C++ conversion problem is a variant of the modularization problem that has more structure than Fortran-to-X and Cobol-to-X conversion/modularization problems. In particular, the C program's `struct` types serve as a natural starting point for the C++ program's classes.

- Lindig and Snelting and Sahraoui et al. use context relations that relate each function of a program to the global variables accessed by the function. In our work, context relations relate each function of a program to (i) the fields of user-defined `struct` types that the function accesses, (ii) the types of sub-expressions that occur within the function, and (iii) the complements of (i) and (ii).
- In our work, we employ negative information (e.g., "attributes of the form `f` does not use fields of `struct t`"). This allows the concepts identified to be based not only on the similarities between functions, but also on their differences.

3.8 Summary

This chapter has shown how to apply concept analysis to the problem of identifying modules in programs that do not specify them explicitly. We have discussed two major advantages concept analysis offers to the solution of this problem as compared to previous work:

1. The inclusion of both positive and negative information
2. The ability to "stay within the system"

Furthermore, this chapter has described an implementation of a prototype tool for using concept analysis to analyze C programs. We have also defined the notion of a concept partition and presented an algorithm to discover all such partitions given a concept lattice.

Chapter 4

Physical Subtyping in C

4.1 Introduction

Although C does not support subtyping (in the style of object-oriented languages), programmers often take advantage of the physical layout of program data in memory to simulate subtyping. For the C-to-C++ conversion problem, the identification of physical subtypes in C programs provides a mechanism that can be used to identify base classes, subclasses, and virtual functions.

As an example, consider the C code shown in Figure 58. The function `translateX` is declared to take two arguments: `p` (a pointer to a `Point`) and `x` (an integer). The function translates `p`'s horizontal component by `x` units. If `pt` is declared to be a `Point`, then the expression `translateX(&pt,1)` is legal C. We might also wish to apply `translateX` to a variable `cp` of type `ColorPoint`, but the statement `translateX(&cp,1)` is *not* (strictly speaking) legal in C. We can *cast* a pointer to a `ColorPoint` to be a pointer to a `Point` as in:

```
translateX((Point *)(&cp), 1)
```

Because of the way values of the types `Point` and `ColorPoint` are laid out in memory, the cast of actual parameter `&cp` in this call on `translateX` causes one type (i.e., `ColorPoint`) to be treated as a *subtype* of another type (i.e., `Point`).

```
typedef struct {
  int x,y;
} Point;

typedef enum { RED, BLUE } color;

typedef struct {
  int x,y;
  color c;
} ColorPoint;

void translateX(Point *p, int dx)
{
  p->x += dx;
}
```

Figure 58: A simple example of subtypes in C: `ColorPoint` can be thought of as a subtype as `Point`.

In this chapter, we make precise the notion of physical subtyping in C and describe an algorithm by which physical subtypes can be recognized automatically. This information can then be used to generate inheritance hierarchies in C++ (with subtypes corresponding to subclasses). For example, the C code shown in Figure 58 might correspond to the C++ code shown in Figure 59.

Contributions of this chapter include:

- Defining the notion of *physical subtypes*.
- Defining a memory model for C and defining a notion of *physical type safety* based on that model.
- Identifying several *idioms* in C programs that represent C++-style object-oriented constructs and discusses which of these idioms correspond to physical subtypes,

```
class Point {
protected:
  int x,y;
public:
  void translateX(Point *p, int dx) {
    p->x += dx;
  }
};

typedef enum { RED, BLUE } color;

class ColorPoint : public Point {
protected:
  color c;
};
```

Figure 59: A simple example of subclasses in C++: subclass `ColorPoint` is derived from base class `Point`.

and which require more complex techniques to be used to identify them automatically.

- Presenting formal rules by which the physical-subtype relationship may be inferred.
- Defining the class of *physical-subtype correct* programs and shows that this class of programs is necessarily physically type safe.
- Describing two complementary tools (that identify physical subtypes) to detect subtype hierarchies in large software systems.
- Describing how physical subtyping can be used to identify type errors and portability errors that escape the notice of standard C compilers.

Section 4.2 reviews what it means to be a subtype and explains the difference between subtypes in C and in C++. Section 4.3 reviews storage-allocation requirements for data structures in C and discusses how the size and *alignment* of data types relate to physical subtypes. Section 4.4 defines a memory model for C and defines physical type safety based on that model. Section 4.5 explains several common idioms by which C programmers emulate object-oriented programming. Section 4.6 presents a type system for C and formalizes physical subtypes by presenting a collection of inference rules. Section 4.6 also defines the class of *physical-subtype correct* programs and shows that this class of programs is necessarily physically type safe. Section 4.7 describes how some object-oriented idioms (such as *downcasts*), are an instance of physical subtyping, but are sometimes still physically type safe. Section 4.8 describes our implementation of two complementary tools that identify physical subtypes. Section 4.8 also presents some results of applying these tools to detect subtype hierarchies and to augment the type checking facility of standard C compilers. Section 4.9 concerns related work.

4.2 Subtypes in C and C++

Informally, a type t is a *subtype* of type t' when any instance of type t can be sensibly interpreted as being of type t'. More formally, t is a subtype of t' if it conforms to the Liskov Substitution Principle:

> For every object x of type t there is an object x' of type t', such that for all programs P defined in terms of t', the behavior of P is unchanged when x is substituted for x' [41].

We say that a programming language L *supports subtyping* if a user may define types t and t' in a program P in L such that:

1. If P contains an expression e' of type t' and if expression e is of type t, the program resulting from replacing e' with e in P is still a legal L program.[1]

2. t is a subtype of t' according to the Liskov Substitution Principle.

A programming language can support subtyping in two ways: explicitly (by declaration) or implicitly (by structure). C++ supports explicit subtyping because for any class defined, a subtype of that class can be created via public inheritance. For example,

```
class ColorPoint : public Point
```

is an explicit declaration that `ColorPoint` is a subtype of `Point`. The rules of C++ guarantee that an object of class `ColorPoint` can be substituted in a context expecting an object of class `Point` in a sensible way. In other words, explicitly named subtypes in C++ satisfy the Liskov Substitution Principle.

C does not support explicit subtyping, but it does support implicit (structural) subtyping. There is no facility for explicitly declaring one type to be a subtype of another. Any pointer type can be thought of as a subtype of the type `void*`, and because of implicit coercions and promotions, arithmetic types can be thought of as subtypes of each other. C does not allow all substitutions. For example, the following is illegal in C:

[1] "P is a legal L program" means only that P will compile successfully. It says nothing about whether P is sensible in any other sense—e.g., whether P is guaranteed to halt, or whether P is guaranteed not to crash.

```
Point p;
ColorPoint cp;
...
p = cp;  /* error: incompatible types */
```

However, by the use type-cast expressions, any pointer of one type can be used in a context where a pointer of another type is expected. So the following is legal (and reasonable) in C:

```
Point *pp;
ColorPoint *cpp;
...
pp = (Point *)cpp;  /* okay: explicit type cast */
```

Using pointer-type casts in conjunction with the address-of operator (`&`) and the indirection operator ($*$), any type can be cast to any other. For example:

```
Point p;
ColorPoint cp;
...
p = *((Point *)(&cp));
```

Most C compilers do not require explicit casts from one pointer type to another. A pointer of one type may be *implicitly type cast* to a pointer of another type (at most causing the compiler to issue a warning). Implicit type casts can occur in two places:

- Assignments:

```
Point *pp;
ColorPoint *cpp;

pp = cpp;   /*  implicit type cast  */
```

- Function calls:

```
ColorPoint *cpp;
void translateX(Point *, int);

translateX(cpp, 1);    /*  implicit type cast  */
```

Implicit casts cause one commonly used compiler (`gcc`) to issue the warning "assignment from incompatible pointer type," but the code compiles nevertheless.

Member selection in a `union` type can also be thought of as a form of type cast. For example, if `u` is a `union` and `a` is a member of `u` of type t, then `u.a` can be thought of as `*((`t`*)(&u))`, assuming `a` is not a bit-field member. We will sometimes refer to non bit-field member selection from `union`s as *implicit union casts*.

Substitution is a weaker notion than subtyping since it comes with no guarantee of expected behavior. One of the fundamental contributions of this chapter is to provide a notion subtyping for C—*physical subtyping*—that satisfies the Liskov Substitution Principle.

The intuitive idea behind physical subtyping is that an expression of one type may be substituted for an expression of another type if when the two are laid out in memory, the values stored in corresponding locations "make sense". Consider the following code:

```
Point pt;
ColorPoint cp;


pt.x = 3;  pt.y = 41;
cp.x = 5;  pt.y = 17;  pt.c = RED;
```

A picture of how `pt` and `cp` are represented in memory might look like:

`pt`	3	41	
`cp`	5	17	`RED`

Intuitively, `cp` can be thought of as being of the same type as `pt` simply by *ignoring its last field*. In some sense, the data "lines up."

Subtypes and type safety

Subtypes are often associated with type safety—for good reason. Sound definitions of subtypes and type safety enforce the principle that *substitution of subtypes preserves type safety*:

> Given two types t and t' such that t is a subtype of t', type-safe program P that makes use of an expression e' of type t', and any expression e of type t, then the program P' resulting from the *substitution* of e for e' in P is also a type-safe program.

(Observe the similarity between this principle and the Liskov Substitution Principle.)

In some programming languages, subtyping can be decoupled from type safety. For example, in the programming language *Eiffel*, no such safety-preserving principle holds for subtypes [45]. However, the goal of this chapter is to develop a definition of subtyping in C that jibes with the definition of subtyping in C++. Because in C++ substitution of subtypes preserves type safety, we formally define type safety in C as *motivation* for the subsequent definition of physical subtypes.

Section 4.4 defines physical type safety. Section 4.6.2 defines physical subtyping. Section 4.6.4 shows that *substitution of physical subtypes preserves physical type safety.*

4.3 Sizes, Alignments, and Offsets of C Types

The crucial aspect of physical subtyping is how data types are stored (or laid out) in memory. In this section, we briefly review the storage model for C data structures. For a more detailed account, the reader is directed to [29, 38, 34].

All data objects in C are represented by an integral number of bytes in memory. The *size* of a data object is the number of bytes occupied by the data object [29]. A character type (`char`, `signed char`, `unsigned char`) is defined to occupy one byte of memory. The sizes of other C types are implementation dependent, but must conform to the following guidelines:

- `signed`, `unsigned`, `const`, and `volatile` qualifiers do not affect the size of a type. For example, an `unsigned int` is of the same size as a `const int`.
- `short`s and `int`s require at least 16 bits.
- `short`s are no longer than `int`s.
- `long`s require at least 32 bits.

- `int`s are no longer than `long`s.
- `float`s, `double`s, and `long double`s require at least 32 bits.
- a `union` requires at least as much storage as its largest member.
- a `struct` requires at least as much storage as the sum of the storage of its members, respecting the alignments of its members (see below).

Some computers allow an object to be stored at any address in memory, regardless of the type of the object. However, many computers impose *alignment restrictions* on certain data types. Some types are required to be stored at addresses that are integral multiples of bytes. Suppose we are working on a computer with a word size of four bytes and that an `int` occupies four bytes of memory. Further suppose that the machine-level instruction for multiplying two integers requires that the integers be *aligned* in memory at addresses that are multiples of the word (or `int`) size. For example, imagine an integer stored in memory location 3–6 (out of possible locations 0–7) as in the diagram below:

0	1	2	3	4	5	6	7
			x	x	x	x	

If the multiplication instruction requires the `int`s to be stored at word multiples, then one of the following cases may occur, depending on the computer architecture:

- The execution of the multiplication instruction on a non-word aligned address results in a run-time error (halting the program).

- The machine adjusts the address to be on a word boundary, say addresses 4–7, resulting in an incorrect perceived value for the `int` in question.
- The machine copies the four bytes to a word-aligned location at a significant loss of efficiency.

If a data object is stored in accordance with its type's alignment restriction, expressions (other than casts) involving that object are *portable* across all C compilers compliant with the C standard. However, because of the ability of C programmers to cast an expression of one type to be of another type, it is possible to make an end-run around alignment restrictions resulting in *non-portable code*. One of the advantages of physical subtypes (formalized in Section 4.6) is: *a cast expression from type t to type t' where t is a physical subtype of t' can be used without loss of portability.*

Character types have no alignment restrictions since they occupy only one byte of memory. The alignment restrictions of other C types are implementation dependent, but conform to the following guidelines:

- `signed`, `unsigned`, `const`, and `volatile` qualifiers do not affect the alignment of a type. For example, an `unsigned int` has the same alignment as a `const int`.
- The alignment of a `struct` or `union` is no less than the maximum alignment of its members.

The storage rules of `struct` types dictate that the first member be stored at the beginning of the `struct`. All subsequent members are stored in the order they are declared within the `struct`. The alignments of the types of the members of a `struct` may require that unused space be placed between members. For example, suppose a machine requires integers to be stored at four-byte multiples. Consider the following `struct`:

```
struct {
  char a;
  int b;
  char c;
} x;
```

If the `struct` is stored at address 0, then `x.a` is at address 0, `x.b` is at address 4 (not address 1), and `x.c` is at address 8. Furthermore, the entire `struct` is padded to be a multiple of its largest alignment, resulting in a total size of 12 bytes.

We say that the number of bytes between the address of a field of a `struct` and the `struct` itself is the *offset* of the field. C guarantees that the offset of the first member (assuming it is not a bit field) of a `struct` is 0 and the offset of any other member (again, assuming it is not a bit field) is a multiple of the alignment of the type of that member. C guarantees that all non-bit-field members of unions are placed at offset 0.

The storage of bit-field members is almost entirely implementation dependent. In particular, the presence of bit fields in a `struct` make it nearly impossible to predict how a compiler will choose the offsets of other members of a `struct`. In our type system (see Section 4.6) we assume that offsets are supplied explicitly. In our experiments (see Section 4.8) we assume that for `struct` types without bit fields, the members are stored as close together as possible without violating alignment restrictions.

4.4 Physical Type Safety

In this section, we define a memory model for C programs and a notion of physical type safety based on that model.

4.4.1 Scalar Dereferences

Before explaining our memory model for C and how it pertains to type safety, a little more background is needed. A *scalar type* is an arithmetic type, a pointer type, or an enumerated type [29]. A *dereference* (or indirection) is an expression that takes an address (a value of pointer type) and returns the value to which the address points. We define a *scalar dereference* to be a dereference of an address pointing to a scalar type.

A C program with arbitrary dereferences is equivalent to a C program with only scalar dereferences. To see this, consider any legal[2] dereference of a pointer to a non-scalar type. The possibilities include pointers to functions, pointers to arrays, pointers to `struct`s, and pointers to `union`s. We can normalize the prorgam so it has no dereferences of pointers to functions. Dereferencing an array of elements of type t is equivalent to dereferencing the first element of the array and is therefore equivalent to dereferencing an object of type t. Dereferencing a pointer to an array results in an array type, but as mentioned, an array of elements of type t can be thought of as a pointer to the first element of the array and, thus, as being of type pointer to t, which is a scalar type. This leaves pointers to `struct`s and pointers to `union`s to consider.

The result of a dereference of a pointer to a `struct` or a pointer to a `union` can

[2]In C, it is illegal to dereference an expression of type `void*`.

only be used in three contexts: an address-of expression, an assignment expression, or a member selection. Obviously, the address of a dereference of a pointer is the original pointer itself.

C has a strict rules concerning assignments between `struct`s and `union`s: both the left-hand and right-hand sides of the assignment must be of the same type. Because of this, an assignment, between `struct`s is equivalent to member-by-member assignment[3] For example:

```
struct {
  int i,j;
} *x, *y;


*x = *y;


/* equivalent to:
  x->i = y->i;
  x->j = y->j;   */
```

An assignment between `union`s is equivalent to an assignment between the largest member of the `union`s, as in the following:

```
union {
  char c;
  double d;
```

[3]The C Standard dictates that implementations of the assignment operator can assume that the left-hand object and right-hand value do not overlap in memory (unless they exactly overlap, as in `S=S`). If overlap does occur, the behavior of the assignment is implementation dependent [29].

```
} *u, *v;

*u = *v;

/* equivalent to:
  u->d = v->d;    */
```

The remaining case is member selection of a dereference of a pointer to a `struct` or a pointer to a `union`. If the member selected is a scalar type, we are done. If not, we can apply this argument recursively until the only dereferences remaining in our program result in scalar types. Therefore, a C program with arbitrary dereferences is equivalent to a C program with only scalar dereferences.

4.4.2 A Memory Model for C

We consider a simple memory model in which, at any given point during a program's execution, every memory address has a unique *allocation type* associated with it. We define the allocation type of a memory address to be one of the of the following:

- *Unallocated*: Addresses that have not been allocated or are no longer allocated in the program are classified as unallocated. Uninitialized pointer variables point to unallocated addresses.
- *Ready*: A memory address is ready if it has been allocated but has not yet received a value.

- *Pad*: A memory address is a pad if it refers to an unused portion of a `struct` or `union` type as sometimes needed to ensure proper alignment. (See Section 4.3.)
- *Middle*: A memory address is "in the middle" if it refers to a byte other than the first byte of an object of a scalar type that has been allocated to the program.
- *Scalar type t*: A memory address is of a scalar type t if it is the address of the first byte of an object of type t that is currently stored there.

We illustrate the memory model with a simple example: Suppose our machine consists of sixteen bytes of memory, addressed from 0 to 15. Suppose further that `int`s occupy four bytes and have a four-byte alignment and pointer types occupy two bytes with no alignment restriction. Suppose a program begins with the following statically allocated variables:

```
char c;

struct S {
  int *a;
  int b;
} s;

union U {
  char c;
  int i;
} u;
```

At the outset of program execution, one possible view of the allocation types of each memory address is the following, where "u" represents "unallocated", "r" represents "ready", "m" represents "in the middle", and "p" represents "pad":

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
u	u	r	u	r	m	p	p	r	m	m	m	r	m	m	m

```
          c     — s.a —        ———— s.b ————    ———— u ————
```

Now suppose the following statements are executed:

```
c = 'A';         /* address  2 assigned scalar type char */


s.b = 23;        /* address 10 assigned scalar type int  */
s.a = &(s.b);    /* address  4 assigned scalar type int* */


u.c = c;         /* address 12 assigned scalar type char */
                 /* addresses 13-15 assigned pad         */
```

The allocation types of each memory address should now look like the following, where "c" represents `char`, "i" represents `int`, and "pi" represents pointer to `int`:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
u	u	c	u	pi	m	p	p	i	m	m	m	c	p	p	p

```
          c     — s.a —        ———— s.b ————  u.c
```

Observe that the union type has been instantiated as a `char`, and so the remaining three bytes of the four allocated for the union (to accommodate the larger `int`) are now tagged as padding rather than as being in the middle of a type.

4.4.3 Physical Type Safety Defined

C is a very flexible language and allows its users to cast any pointer type to any other pointer type arbitrarily. We have already seen how casts can be used for subtyping. Now we consider how type casts can be used in dangerous ways.

We illustrate the point by continuing the previous example: Suppose the next two statements reassign `s.a` to point to a `char` instead of an integer, and then store a value at the address now pointer to by `s.a`:

```
s.a = (int *)&c;
*(s.a) = 42;
```

According to the C language, this is perfectly acceptable code, but let's consider what happens at the last assignment expression. As far as a C compiler knows, `s.a` still points to an `int`, so an attempt will be made to copy *four bytes* to the the four bytes beginning at the address of `c`. If the underlying architecture is strict with how it manipulates integers, the program will likely crash, since the address of `c` (i.e., 2) is not a multiple of the alignment of `int` (i.e., 4). On the other hand, suppose the copy does take place. The last two bytes of the copy will overwrite memory addresses 3, 4, and 5. The address 3 is not allocated to the program, and this in itself might cause the program to crash. Worse yet, addresses 4 and 5 are where `s.a` is stored in the first place! A subsequent dereference of `s.a` could point anywhere in memory. This illustrates that legal C programs need not be type safe. We call such an error a *physical type error*.

We say that the *expected type* of a C expression is the type associated with the expression according to the propagation and promotion rules prescribed by the C language [34, 38, 29]. In other words, the expected type of an expression is the type the compiler expects the expression to be.

We say that a program P is *physically type safe* if for every scalar dereference in P, the address dereferenced is "ready" or has allocation type scalar type t and the expected type of the scalar dereference is scalar type t' such that either $t = t'$ or else t and t' are both pointer types.

Returning to the example, the assignment: `*(s.a) = 42` violates physical type safety since the expected type of the dereference is an `int` but the allocation type of the address pointed to by `s.a` is `char`.

4.5 Object-Oriented Idioms in C

In this section, we consider several object-oriented *idioms* that can be found with perhaps surprising frequency in C programs. These idioms emulate C++ features, such as *inheritance*, *class hierarchies*, *multiple inheritance*, and *virtual functions*.

4.5.1 Inheritance

Redundant declarations

C programmers can emulate public inheritance in a variety of ways. Perhaps the most common (at least for data types with a small number of members), is by declaring one `struct` type's member list to have another `struct` type's member list as a *prefix*. For example:

```
typedef struct { int x, y; } Point;
```

```
typedef struct { int x, y;
                 Color c;  } ColorPoint;
```

Instances of `ColorPoint` can be used in any context that allows the use of an instance of `Point`. Any valid context expecting a `Point` can, at most, refer to the `Point`'s `x` and `y` members. Any instance of `ColorPoint` has such `x` and `y` members and at the same relative offsets as for an instance of `Point`.

First members

The use of redundant declarations is perhaps the simplest method of implementing subclassing in C. However, making a textual copy of the members of a *base* class in the body of each *derived* class is both cumbersome and error-prone. The *first-member idiom* represents an improvement that alleviates both of these problems.

Subtype relationships often characterize *is-a relationships*, as in "a color point *is a* point". Members of `struct` types often characterize *has-a* relationships. For example, a `Person` *has a* `name`:

```
typedef struct {
  ...
  char *name;
  ...
} Person;
```

However, because C guarantees that the first member of an object of a `struct` type begins at the same address at which the object itself begins, the first member can also reflect an is-a relationship. For example, consider this alternative definition of `ColorPoint`:

```
typedef struct {
  Point p;
  Color c;
} ColorPoint;
```

Now a ColorPoint can be used where a Point is expected in two equivalent ways:

```
  ColorPoint cp;
  void translateX(Point *, int);


  translateX((Point *)&cp, 1);
  translateX(&(cp.p), 1);
```

In the second call to `translateX`, the reference to the `Point` component of `cp` is made more explicit (at the cost of having the programmer remember the names of the first member and modifying such code if and when the member names change).

Array padding

The first-member idiom may be implemented in a slightly different manner—in which the allocation of storage space for the members of the *base* class is separated from the access to those members. Consider another definition for `ColorPoint`:

```
typedef struct {
  char base[sizeof (Point)];  /* storage space for a point */
  Color c;
} ColorPoint;
```

In this definition of `ColorPoint`, sufficient space is allocated to hold an entire `Point` rather than explicitly declaring a member of type `Point` (as in the first-member idiom) or using all the members of `Point` (as in the redundant-declaration idiom). The space is allocated by using a byte (`char`) array of the same size as `Point`.

While this idiom seems quite similar to the first-member idiom, it poses problems for our formal model of physical subtypes and physical type safety (described in Section 4.4 and Section 4.6, respectively). The problems are discussed in Section 4.7.3.

This idiom is prevalent in several large systems that we have analyzed with our tools (described in Section 4.8), most notably call-processing code from Lucent Technologies and `gcc`.

Flattening

We say that `struct` type s is a flattening of a `struct` type s' if the list of members of s is the result of replacing occurrences of `struct` members m within the list of members of s' by the list of the members of m. Sometimes subtyping relationships are implied by flattening. For example:

```
typedef struct {
  Point p;
  Color c;
```

```
} ColorPoint;
```

flattens to

```
typedef struct {
  int x,y;
  Color c;
} ColorPoint;
```

In this example, flattening can be seen as a transformation from the first-member idiom to the redundant-declaration idiom. However, flattening need not be confined to predefined `struct`s or to first members. For example:

```
typedef struct {
  int x;
  struct {
    int y;
    Color c;
  } s;
} ColorPoint;
```

also flattens to:

```
typedef struct {
  int x,y;
  Color c;
} ColorPoint;
```

Although this technique is not a very good way to implement subtypes or polymorphism (the idiom is complicated to understand and is rather error-prone), it has been used in some software projects (where, perhaps, two structures may have different logical groupings of the same list of fields). Identifying this idiom allows us to issue a diagnostic to the programmer (perhaps suggesting better ways to reorganize the data structures involved to make the object-oriented idiom more explicit and less error-prone).

Flattening is discussed further in Section 4.6.3.

4.5.2 Class hierarchies

It is not uncommon to find implicit class hierarchies in C programs using one or more of the inheritance idioms discussed above. One interesting combination is to use the first-member idiom for the top level of inheritance and then to use something like the redundant declaration idiom for deeper levels of inheritance. For example:

```
typedef struct { .... } Point;


typedef struct {
  Color c;
  } AuxColor;
typedef struct {
  Point p;
  AuxColor aux;
} ColorPoint;
```

```
typedef struct {
  char name[10];
} AuxName;
typedef struct {
  Point p;
  AuxColor aux;
  AuxName aux2;
} NamedColorPoint;
```

Notice that `NamedColorPoint` can be thought of a subclass of `Point` by the first-member idiom and as a subclass of `ColorPoint` by the redundant-declaration idiom.

Using the tools described in Section 4.8.1, we found that this idiom is prevalent in *xemacs* (a graphical-user-interface version of the text editor *emacs*).

4.5.3 Multiple Inheritance

An extension of all of the above idioms can be used to implement multiple inheritance in C. The C++ language specifies that the member list of a multiply derived class is the concatenation of the member lists of each of its base classes (in order of specification) followed by the member list of the derived class itself. The *first-member* idiom is just a special case of the above layout rule (where there is only one *base* class). There are several ways in which programmers may emulate multiple inheritance in C. Here we illustrate one such method, which we call *the `+1` idiom*. The following example is based on an example in [14]:

```
typedef struct { ... } Clock;
typedef struct { ... } Radio;

typedef struct {
  Clock clock;
  Radio radio;
} ClockRadio;
```

The type `ClockRadio` can be thought of as a class that multiply inherits from the types (classes) `Clock` and `Radio`. Suppose we define a pointer to a `Clock` and set it to point to the `Clock` in a `ClockRadio`:

```
  ClockRadio cr;
  Clock *c = &(cr.clock);
```

We can access the `Radio` of `cr` directly as in:

```
  Radio *r = &(cr.radio);
```

Alternatively, we can refer to the radio through the clock:

```
  r = (Radio *)(c + 1);
```

This makes sense because in C, `c + 1` is a pointer-arithmetic expression meaning: the address indicated by `c` plus (one times) the size of the object pointed to by `c`. In other words, `c + 1` says "point to the the next member in the `struct` containing what `c` points to". An advantage to using the `+1` idiom is that the programmer need only remember the types of members, not member names.

By C's type rules, the expression `c + 1` has the same type as `c`. So the cast is from `Clock*` to `Radio*`. The two types need not have anything in common, so the cast, on the surface seems ill advised. However, because of the way C lays out data in memory, the cast is type safe[4].

It is one thing to identify an instance of the `+1` idiom, it is entirely another thing to determine if such an instance makes sense in terms of subtypes (or in terms of type safety). The problem of making sense of casts such as these is outside the scope of this thesis; however, Section 4.5.4 mentions some possible avenues for dealing with it.

4.5.4 Downcasts

It is common object-oriented practice to allow objects of a derived class to be treated as if they are of a base class. This notion is referred to as an *upcast*—as in casting up from a subclass (subtype) to a superclass (supertype). The complementary notion of a *downcast* is not as common, but still very useful in object-oriented programming. A downcast is when an object of a base class is treated as an object of a derived class, or in C when a cast expression is from a supposed supertype down to a supposed subtype. This is not always type safe (see Section 4.7). In this and the next section, we describe how and why downcasts are used.

The following is a simple example of a downcast from type `Point` to type `ColorPoint`:

```
Point* pp;
void make_red(ColorPoint* cp) {
  cp->c = RED;
```

[4] This assumes that the sizes and alignments of the `Clock` and `Radio` types are such that no padding is required between the `clock` and `radio` fields.

```
}
...
make_red((ColorPoint *) pp);  /* downcast from Point to ColorPoint */
```

The substitution is not necessarily type safe since `make_red` accesses the `c` member of `ColorPoint`, which is not a member of `Point`. However, suppose `pp` was assigned as follows:

```
ColorPoint cp;
Point *pp;
...
pp = &cp;                     /* upcast from ColorPoint to Point */
...
make_red((ColorPoint *) pp);  /* downcast okay */
```

In this case the downcast is type safe.

4.5.5 Virtual functions

Downcasts are necessary in order to implement *virtual functions*, which are one of the most powerful aspects of object-oriented programming.

Suppose we redefine `Point` and `ColorPoint` so that each has a member `print`:[5]

```
typedef struct _Point {
```

[5]According to the definition of physical subtyping presented in Section 4.6.2, this definition of `ColorPoint` is not a subtype of `Point` because the type of the `ColorPoint`'s member `print` is not a subtype of `Point`'s member `print`. Our definition of physical subtyping assumes that a function pointer is a subtype of another function pointer only if the two types are the same. This is to avoid type-safety problems with *covariant* function subtypes. See [19, 15] for more details. In Section 4.7.3 we consider relaxing the rule to allow all function pointers to be subtypes of all other function pointers.

```
  int x, y;
  void (*print)(struct _Point *);  /* argument type is really Point* */
} Point;

typedef struct {
  int x, y;
  void (*print)(ColorPoint *);
  Color c;
} ColorPoint;
```

This is an example of how class-member functions can be simulated in C.

Suppose we initialize an array of pointers to `Points`:

```
  Point p;
  ColorPoint cp;
  ...
  Point *A[2];
  A[0] = &p;
  A[1] = (Point *)(&cp); /* upcast */
```

We can then use the following loop to print out each the contents of each point in the array:

```
  int i;
  int n; /* number of elements in A */
  for (i=0; i < n; i++)
    A[i]->print(A[i]);  /* simulated virtual function call */
```

Each call of `A[i]->print` is like a virtual function call—the actual function being called depends on the value stored at `A[i]`. When `A[i]` actually points to a `colorPoint` the call `A[i]->print(A[i])` is implicitly "downcasting" the argument. The execution of this loop is type safe, since the downcast is only performed when the pointer to the `Point` at `A[1]` is really pointing to a `ColorPoint` (that was assigned to `A[1]` via an upcast).

4.6 Formalizing Physical Subtypes

In this section, we present our non-standard type system for C and formally define physical subtyping. We also define what it means for a program to be physical-subtype correct and present, as a theorem, one of the fundamental contributions of this chapter, namely that physical-subtype-correct C programs are physically type safe.

We $t \prec t'$ to denote that t is a *physical subtype* of type t'. The intuition behind physical subtypes can be summarized as follows:

- The size of a type is no larger than the size of any of its subtypes.
- Ground types are physical subtypes of themselves and not of other ground types. For example:
  - $\texttt{int} \prec \texttt{int}$
  - $\texttt{int} \not\prec \texttt{double}$
  - $\texttt{double} \not\prec \texttt{char}$
  - an enumerated type is not a physical subtype of a different enumerated type (or any other ground type)

- A pointer to type t is a physical subtype of a pointer to type t' if and only if t is a physical subtype of t'.
- If an aggregate type is a physical subtype of another aggregate type then the members of two types 'line up' in some sensible fashion.

4.6.1 A Type System for C

Our work addresses a slightly simplified version of the C type system:

- We ignore type qualifiers (e.g., `const int` and `volatile int` are treated as `int`).
- We consider typedefs to be synonyms for the types they redefine.

Types are described by the following language of type expressions:

t ::

	ground	
$\mid$	$t[n]$	// array of type t of size n
$\mid$	t `ptr`	// pointer to t
$\mid$	$s\{m_1, \ldots, m_k\}$	// `struct`
$\mid$	$u\{\mid m_1, \ldots, m_k \mid\}$	// `union`
$\mid$	$(t_1, \ldots, t_k) \rightarrow t$	// function

m ::

	(t, l, i)	// member labeled l of type t at offset i
$\mid$	$(l : n)$	// bit field labeled l of size n

ground ::

$e\{id_1, \ldots, id_k\}$ // `enum`

| `void*` | `char` | `unsigned char`

| `short` | `int` | `long` | `double` | ...

Non-bit-field members `struct` and `union` types are annotated with an *offset*. In a `struct` the offset of a member indicates the difference in bytes between the storage location of this member and the first member of the `struct`. The first member is, by definition, at offset 0. All members of `union` types are considered to have offset 0. (See Section 4.3)

4.6.2 Physical Subtypes

We say that t *matches* t' if t is a physical subtype of t' and t' is a physical subtype of t.

Figure 60 presents rules in the style of [28] for inferring that one type is a physical subtype of another. Below, we consider each of the physical-subtype rules individually:

- *Reflexivity*: Any type is a physical subtype of itself.
- *Pointers*: If t is a physical subtype of t' then a pointer to type t is a physical subtype of a pointer to type t'.
- *Void pointers*: A pointer to type t is a physical subtype of `void*`. Void pointers are generic: they can, by definition, only be used in contexts where any other pointer can be used. It is illegal to dereference an object of type `void*`. In fact,

[Reflexivity] $$\overline{t \prec t}$$

[Pointers] $$\frac{t \prec t'}{t\ \mathtt{ptr} \prec t'\ \mathtt{ptr}}$$

[Void pointers] $$\overline{t\ \mathtt{ptr} \prec \mathtt{void*}}$$

[Arrays] $$\frac{n' \leq n}{t[n] \prec t[n']}$$

[First members] $$\frac{t \prec t' \quad m_1 = (l, t, 0)}{\{m_1, \ldots, m_k\} \prec t'}$$

[Structures] $$\frac{k' \leq k \quad m_1 \prec m'_1 \ldots m_{k'} \prec m'_{k'}}{\{m_1, \ldots, m_k\} \prec \{m'_1, \ldots, m'_{k'}\}}$$

[Left union] $$\frac{\forall i \leq k : (m_i = (l, t, 0) \quad t \prec t')}{\{|m_1, \ldots, m_k|\} \prec t'}$$

[Right union] $$\frac{\forall i \leq k : (m_i = (l, t', 0) \quad t \prec t')}{t \prec \{|m_1, \ldots, m_k|\}}$$

[Member subtype] $$\frac{m = (l, t, i) \quad m' = (l', t', i') \quad l = l' \quad i = i' \quad t \prec t'}{m \prec m'}$$

Figure 60: Inference rules for physical subtypes.

the only legal operations on a void pointer that are cause for concern are type casts. For example:

```
Bar *b
Foo *f;
void *vp;
...
vp = (void *)b; /* upcast: Bar* is a subtype of void* */
...
f = (Foo *)vp;  /* downcast: Foo* is a supertype of void */
```

The cast from `void*`to `Foo*` is an example of a downcast, discussed in Section 4.5.4 and Section 4.7.2.

- *Arrays*: An array of elements of type t is a subtype of a smaller array of elements of type t. (Note the *contravariance* between the direction of the subtype relation and the direction of the inequality between the sizes of the two arrays.)

- *First members*: If t is a physical subtype of t' then a `struct` with a first member (the member at offset 0) of type t is a physical subtype of t'. This captures the first-member idiom described in Section 4.5. For example, assuming `ColorPoint` is a physical subtype of `Point`, then:

```
typedef struct {
  ColorPoint cp;
  char *name;
} NamedColorPoint;
```

is a physical subtype of `ColorPoint` as well as a physical subtype of `Point`. (This example also illustrates the *transitivity* of the physical-subtype relation.)

- *Structures*: `struct` s with k members is a physical subtype of `struct` s' with k' members if:
  - s has no fewer fields than s' (i.e., $k' \leq k$). (Again, note the contravariance of size versus subtype.)
  - Each member of s' has the same label, and the same offset as each of the corresponding members of s, and the types of members of s' are physical supertypes of the types of the corresponding members of s.

For example:

```
struct {
  int a;
  struct { double d1,d2,d3; } b;
  char c;
}
```

is a physical subtype of:

```
struct {
  int a;
  struct { double d1,d2; } b;
}
```

This is in contrast with Cardelli-style structural subtyping between *record* types ([13, 1]). A record type is like a `struct` type, but the order of members (and therefore the layout in memory) is unimportant. A record type $\{l_1 : t_1, \ldots, l_k : t_k\}$ is a subtype of record type $\{l'_1 : t'_1, \ldots, l'_{k'} : t'_{k'}\}$ iff for each label l_i, there is a j such that $l_i = l'_j$ and t_i is a subtype of t_j.

- *Left unions*: `union` u is a physical subtype of t' if the type of every member of u is a physical subtype of t'. For example:

```
union {
  ColorPoint cp;
  NamedColorPoint ncp;
}
```

  is a physical subtype of `Point`.

- *Right unions*: t is a physical subtype of `union` u if t is a physical subtype of the type of every member of u. For example, `NamedColorPoint` is a physical subtype of:

```
union {
  Point p;
  ColorPoint cp;
}
```

- *Unions*: The Left-Union and Right-Union rules together imply the following rule: `union` u is a physical subtype of `union` u' if the type of every member of u is of a physical subtype of the type of every member of u'. For example:

```
union {
  struct { double d1,d2,d3,d4; } a;
  struct { double d1,d2,d3; } b;
}
```

is a physical subtype of:

```
union {
  struct { double d1,d2; } aa;
  struct { double d1,d2,d3; } bb;
}
```

The reason the rule for unions is so stringent will become apparent when we consider physical type safety in Section 4.4. We consider more relaxed rules for unions in Section 4.7.1.

4.6.3 Relaxing the Rules

The definition of physical subtypes in the previous section is not quite broad enough to capture some of the more complicated object-oriented idioms that arise in practice (see Section 4.5). In this section, we consider a somewhat relaxed definition of physical subtypes. The motivation for the difference between the original definition of physical subtypes and the relaxed definition presented here is twofold:

1. *Complexity.* By the original definition, the algorithm for determining if a type t is a physical subtype of t' has worst-case running-time that is polynomial in the maximum of the sizes of t and t'. (Here, the "size of the type" refers to the

size of representation of the type in our type system, not the amount of memory required to store the type.) For the more relaxed definition, for some cases (albeit rare ones), the running-time complexity can be exponential.

2. *Type safety.* If the rules are relaxed too much, then it is possible that certain instances of implied subtyping (i.e., type casts) that should be flagged as dangerous might not be caught. (Type safety is discussed in Section 4.4. Relaxations that compromise type safety are discussed in Sections 4.7 and Sections 4.8.)

Member labels

It is sometimes the case that two `struct` types look very similar but differ in one or more member labels, as in the following definitions of `ColorPoint` and `MyColorPoint`:

```
typedef struct {
  int x,y;
  color c;
} ColorPoint;
typedef struct {
  int x,y;
  color color;
} MyColorPoint;
```

This case can arise when more than one programmer is responsible for creating similar data types. However, by relaxing the restriction that member labels must match, a `struct` type with a small number of members may *accidentally* be found to be a subtype of another:

```
typedef struct {
  int x,y;
} Point;
typedef struct {
  int quot,rem;
} div_t;
```

One possible way of avoiding such problems is to use a heuristic, perhaps requiring that a minimum number or percentage of member labels must match.

Flattening

Recall from Section 4.5.1 that a `struct` type s is a flattening of a `struct` type s' if the list of members of s is the result of replacing occurrences of `struct` members m within the list of members of s' by the list of the members of m, as in the following:

```
typedef struct {
  Point p;
  Color c;
} ColorPoint;
```

flattens to

```
typedef struct {
  int x,y;
  Color c;
} ColorPoint;
```

$$\text{[Flatten left]}\quad \frac{\begin{array}{l} \exists i \leq k : (\quad m_i = (l_i, \{m'_1, \ldots, m'_{k'}\}, d_i) \;\; \wedge \\ \qquad \forall j \leq k' : (m'_j = (l'_j, t'_j, d'_j) \;\; \wedge \;\; m''_j = (l'_j, t'_j, d'_j + d_i)) \;\; \wedge \\ \qquad \{m_1, \ldots, m_{i-1}, m''_1, \ldots, m''_{k'}, m_{i+1}, \ldots, m_k\} \prec t) \end{array}}{\{m_1, \ldots, m_k\} \prec t}$$

$$\text{[Flatten right]}\quad \frac{\begin{array}{l} \exists i \leq k : (\quad m_i = (l_i, \{m'_1, \ldots, m'_{k'}\}, d_i) \;\; \wedge \\ \qquad \forall j \leq k' : (m'_j = (l'_j, t'_j, d'_j) \;\; \wedge \;\; m''_j = (l'_j, t'_j, d'_j + d_i)) \;\; \wedge \\ \qquad t \prec \{m_1, \ldots, m_{i-1}, m''_1, \ldots, m''_{k'}, m_{i+1}, \ldots, m_k\}) \end{array}}{t \prec \{m_1, \ldots, m_k\}}$$

Figure 61: Inference rules for physical subtypes with flattening.

There are numerous ways to flatten a `struct`. To name a few: flatten all member `struct`s, no matter how deeply nested within the outer `struct`; flatten the first member that is a `struct`; or flatten all member `struct`s that are not nested. There is also a question of what to do with members that are `union`s that themselves contain member `struct`s. In Section 4.7.1 and Section 4.8.1, we consider relaxations of `union`-subtyping rules that can be used in conjunction with the flattening relaxation to permit flattening of `struct`s within `union`s within `struct`s. Here, we consider two relaxed physical-subtyping rules for `struct`s that incorporate flattening of member `struct`s (but not member `union`s). Formally, we add the pair of subtype rules shown in Figure 61. Informally, the rule "Flatten left" states that a `struct` type s is a physical subtype of t if the flattening of s is a physical subtype of t. "Flatten right" states that a type t is a physical subtype of a `struct` type s if t is a physical subtype of the flattening of s.

For example:

```
struct S {
  int i;
  struct {
    int j,k;
  };
};
```

is a physical subtype of

```
struct T {
  int i, j;
};
```

because the flattening of `struct S`, namely

```
struct S {
  int i, j, k;
};
```

is a physical subtype of

```
struct T {
  int i, j;
};
```

4.6.4 Physical Subtypes and Type Safety

Whole classes of errors in C programs can be avoided by making sure that programs are physically type safe. Unfortunately, physical type safety, as defined, is a *dynamic* notion: It is undecidable to statically determine the allocation type of every address dereferenced in a program. (In order to statically determine the allocation type of an address, we must know which statements will be executed, which is an undecidable problem.) Instead, we can take a more conservative route and show that for an easily recognizable class of C programs, physical type safety is guaranteed.

We define a C program to be *physical-subtype correct* if there are no expressions involving bit-field members of union types and if every cast—explicit, implicit, and implicit-union (see page 145)—is from a type t to a type t' where t is a physical subtype of t'.

Theorem 4.1 *If P is a physical-subtype-correct C program, then P is physically type safe.*

Sketch of proof: Suppose this is not the case. There must be some expression in P that dereferences an address whose allocation type is a scalar type t that differs from the expected type of the dereference (and at least one of t and t' is not a pointer type). But the only way in which that can happen is if at some point a pointer to the address in question is type cast from type t to t'. Since P is physical-subtype correct, then t is a physical subtype of t'. By a case-by-case analysis of the subtype-inference rules in Section 4.6.2, it is apparent that the only case in which a scalar type is ever considered to be a physical subtype of another scalar type is if they are both pointers. Since this is not the case, we have a contradiction. Therefore, P is physically type safe.

4.7 Limitations of Physical Subtyping

The converse of the theorem presented in the previous section does not hold: If program P is physically type safe it does *not* imply that P is physically subtype correct. In this section, we discuss the discrepancy between physical-subtype correct programs and physically type safe programs. We also discuss limitations of physical type safety due to the simplicity of our memory model. In particular, we discuss programs that are not physically type safe, but yet are still sensible.

4.7.1 Subtyping and `Unions`

The strict subtyping rule for `unions` presented in Section 4.6.2, is motivated out of a desire to guarantee that substitutions preserve type safety. With such a strict definition, there are cases where substitution of one `union` for another is physically type safe that are not considered as instances of physical subtyping. For example:

```
union U {
  char   c;
  double d;
  int    i;
} u;

union V {
  char   c;
  int    i;
};
```

```
void usesInt(union V *v); /* only uses the i field of v */
...
u.i = 17;
...
usesInt(&u);    /* this call is physically type safe */
```

The function `usesInt` is assumed to only access the `i` field of its `union V*` parameter. Because, in this case, we know that the actual argument is a pointer to a `union` that has been assigned an `int` value, this code is physically type safe. However, because unions can be used inconsistently, our rules for physical subtyping say that `union U` is not a physical subtype of `union V`.

It is tempting, because of the name "union", to consider C `union` types as if they are Cardelli-style union types (also known as *variant types* or *variant-record types*) [12]. A variant is a labeled disjoint sum, that looks like a record: it is an unordered set of label-type pairs, often denoted with square brackets as in the following simple example taken from [28]:

```
type Vehicle = [Air   : AirVehicle,
                Land  : LandVehicle,
                Water : WaterVehicle]
```

The idea here is that an object of type Vehicle is one of the three member types. There is usually only one operation that can be applied to variants—case selection. Case selection prescribes a different action for each member of the variant and then executes the action corresponding to the form the variant takes.

In the type system presented in [28], a variant v

$$[l_1 : t_1, l_2 : t_2, \ldots, l_k : t_k]$$

is a subtype of a variant v'

$$[l_1 : t'_1, l_2 : t'_2, \ldots, l_k : t'_k, \ldots, l_{k'} : t'_{k'}]$$

if for each $i \leq k$, t_i is a subtype of t'_i. In other words, the set of labels occuring in v is a subset of the labels occurring in v' and each of the types of the members of v is a subtype of the corresponding type in v'. (This is the *dual* of the record subtyping rule mentioned on page 174.)

A C `union` type does, on the surface, appear quite similar to Cardelli-style unions, because an object of `union` type can be thought of as being one of its members at any given time. However, C provides no way of enforcing that a `union` will be treated consistently, and so a physical-subtyping rule for `union` types equivalent to the above subtyping rule for variants is not physically type safe.

It is possible, however, to use a combination of `struct`s, `union`s, and `enum`s to simulate variants in C. For example:

```
typedef struct { ... } AirVehicle;
typedef struct { ... } LandVehicle;
typedef struct { ... } WaterVehicle;
typedef enum { Air, Land, Water } VehicleKind;
typedef struct {
  VehicleKind kind;
  union {
```

```
    AirVehicle   a;
    LandVehicle  l;
    WaterVehicle w;
  } u;
} Vehicle;
```

The idea here is that the enumerated member `kind` is used to *discriminate* among the members of the `union` member `u`. Case selection is emulated via a `switch` statement, as in the following example:

```
void drive(LandVehicle *);
void float(WaterVehicle *);
void fly(AirVehicle *);
void navigate(Vehicle *v)
{
  switch(v->kind) {
  case Air:
    fly(v->u.a);
    break;
  case Land:
    drive(v->u.l);
    break;
  case Water:
    float(v->u.w);
    break;
```

```
  };
}
```

At this point, we can consider *relaxing* the physical-subtype rules to emulate variant subtyping. Suppose types `Automobile` and `LandVehicle` have been declared so that the former is a physical subtype of the latter. Suppose we define a type `AirOrCar`, intended as a subtype of `Vehicle`, as follows:

```
typedef struct { ... } AirVehicle;
typedef struct { ... } LandVehicle;
typedef struct { ... } WaterVehicle;
typedef struct { ... } Automobile;  /* phys. subtype of LandVehicle */
typedef struct { ... } Vehicle;
typedef struct {
  VehicleKind kind;
  union {
    AirVehicle a;
    Automobile l;
  } u;
} AirOrCar;
```

Assuming that we maintain the same physical-subtyping rules for `struct` types, then we must adapt rules for unions such that

```
union {
  AirVehicle a;
```

```
    Automobile l;
  } u;
```

is a physical subtype of

```
  union {
    AirVehicle   a;
    LandVehicle  l;
    WaterVehicle w;
  }
```

The variant subtyping rule given above can be adapted for `union`s as follows: a `union` u is a "relaxed" physical subtype of a `union` u' if the labels of members of u is a subset of the labels of members of u' and each of the types of the members of u is a subtype of the corresponding type in u'.

Assuming that the member `kind` (or whatever a user chooses to call it) associated with the union is used *consistently*, then substituting a union of type t where a union of type t' is expected and where t is a relaxed physical subtype of t' *makes sense.* For example, the function `navigate` should behave correctly if it is called with an argument of type pointer to an `AirOrCar` instead of pointer to a `Vehicle`.

The problem, of course, is that the unions must be used in a consistent way. Without even presenting a formal definition of what it means for a union to be used consistently, we illustrate the apparent undecidability of this problem:

```
  union {
     char c;
```

```
  int i;
} u;
...
if (cond) /* cannot statically determine value of cond */
  u.c = 'A';
else
  u.i = 3;
/* u could contain either an int or char here */
```

It is still useful to know when C programs try to emulate variant types, even if we can make no claims about the type safety of such emulations. Such uses of `union`s are another way of simulating subclasses, where each member of the union is implicitly a subclass of the class represented by the struct enclosing the union.

4.7.2 Downcasts, Virtual Functions, and Type Safety

One of the foundational concepts of this chapter is that a type cast from a pointer type t to a pointer type t' can be thought of as a substitution of a value of type t in a context expecting a value of type t' and hence that t is being treated as subtype of t'. As discussed in Section 4.5.4, not all type casts between pointers are from a subtype to a supertype (i.e., upcasts). Recall from Section 4.5.4, a downcast is a type cast from a supertype to a subtype. In this section, we show how downcasts are not necessarily type safe, but can be in some instances, and how this further illustrates the discrepancy between physical-subtype correct programs and physically type safe programs.

The following is an example of a downcast (from Section 4.5.4) from `Point` to

`ColorPoint`:

```
Point* pp;
void make_red(ColorPoint* cp) {
  cp->c = RED;
}
...
make_red((ColorPoint *) pp);  /* downcast from Point to ColorPoint */
```

The downcast is not necessarily physically type safe since `cp->c` may refer to an offset of `*pp` that is not of the scalar type `Color` (and could in fact refer to unallocated memory). However, if `pp` has been assigned to point to a `ColorPoint` (by an upcast) then the code is type safe. `Point` is not a physical subtype of `ColorPoint`, so this program is not physical-subtype correct.

Recall the virtual-function example given in Section 4.5.5:

```
typedef struct _Point {
  int x, y;
  void (*draw)(struct _Point *);
} Point;
typedef struct _ColorPoint {
  int x, y;
  void (*draw)(Point *);
  Color c;
} ColorPoint;
void printPoint(Point *p) { ... }
```

```
void printColorPoint(Point *p)
{
  ColorPoint *cp = (ColorPoint*)p;  /* downcast */
  ...
}
int i;
int n; /* number of elements in A */
Point *A[n];
...
for (i=0; i < n; i++)
  A[i]->print(A[i]);  /* simulated virtual function call */
```

Each call of `A[i]->print` is like a virtual function call—the actual function being called depends on the value stored at `A[i]`. The execution of this loop is type safe, since the downcast is only performed when the pointer to a `Point` at `A[1]` is really pointing to a `ColorPoint` via an upcast.

The example illustrates the utility of identifying downcasts, even if their type safety cannot be determined. It is another clue as to how classes, subclasses, and virtual functions might be extracted out of C programs. In fact, this example is a simplified version of emulated classes, subclasses, and virtual functions found throughout the SPEC benchmark program *ijpeg*. The presence of simulated virtual functions in the *ijpeg* code was identified from the occurrences of downcasts. (See Section 4.8.5.)

Observe that because all pointer types are considered to be subtypes of `void*`, then `void*` being cast to another pointer type is an instance of a downcast. In the general case, identifying safe uses of downcasts is like identifying safe uses of void pointers. This also has applications in C programs that do not make use of `struct` and `union` types (and so may not be emulating classes at all).

It is undecidable to statically determine when a downcast is used in a physically type-safe fashion. However, there are several techniques (not addressed in this thesis) that might be used to provide a "safety analysis" on downcasts as well as to justify a more relaxed definition of physical-subtype correctness and thus narrow the gap between physical-subtype-correct programs and physically type-safe programs. Such techniques might include:

- Dataflow analysis in conjunction with context-free-language reachability. Context-free-language reachability (CFL-reachability) is a generalization of ordinary graph reachability in which each edge of a graph is labeled with a letter from an alphabet [44]. A path p from s to t is considered to connect s to t only if the word spelled out by the letters on the edges of p is in a given context-free language. The language used for the downcast-analysis problem could be a matched-parentheses language in which the open parentheses are upcasts and the closed parentheses are downcasts.

- Constraint-based-type analyses [4, 3, 16]. These techniques might also be used to check for physically type-safe instances of the `+1` idiom (see Section 4.5.3) and other instances of code that is not physically subtype correct, but may still be type safe.

4.7.3 Limitations of Physical Type Safety

In this section, we consider limitations of physical type safety due to the simplicity of our memory model. In particular, we discuss programs that are not physically type safe, but yet are still sensible.

Generic pointers

Before the establishment of the ANSI C Standard, programmers often used a `char*` to represent a generic pointer instead of `void*`[6]. Programmers also sometimes use pointers to other scalar types (particularly to `int`) to represent generic pointers. It is easy to adapt the rules for physical subtyping to reflect these uses, although at a potential loss of type safety: because pointers to scalar types can be dereferenced (unlike `void*`), allowing one pointer to be used in place of a pointer to `char`, for example, can result in the same address in memory being used as two different scalar types.

To avoid the sticky issue of contravariance of functional subtypes [19, 15], the definition of physical subtypes given in Section 4.6.2 states that two pointers to functions are only subtypes of each other if they have the same type. Experience has shown that there are cases where it is useful to consider two function pointer types as subtypes, again at a loss of type safety.

Bit fields

According to the ANSI C Standard, the rules for storing bit fields are implementation dependent [38, 29], thus in order to maintain physical type safety (as well as portability), types with bit-field members are not considered to be physical subtypes or

[6] `void*` has not always been available in C.

physical supertypes of any other types. In practice, we sometimes may wish to relax the definition, and allow two bit-field members to be considered subtypes of each other if they are of the same length. Again, this sacrifices physical type safety.

Array padding

The array-padding idiom described in Section 4.5.1 poses problems for our formal model of physical subtypes and physical type safety. Recall the simple example:

```
typedef struct {
  char base[sizeof (Point)];  /* storage space for a point */
  Color c;
} ColorPoint;
```

`ColorPoint` is intended to be a subtype of `Point` and can be treated as such as in the following:

```
  Point *p;
  ColorPoint cp;

  p = (Point *)(&cp);  /* or equivalent, &(cp.base) */
  p->x = 2;
  p->y = 3;
  translateX(p, 1);
```

Suppose we wish to change our subtyping rules to reflect this. We might add a rule saying that a byte array of size n is a subtype of a type t that also has size n. This new rule plus the first-member rule would establish that this version of `ColorPoint` is a physical subtype of `Point`.

However, such a change implies that the theorem from Section 4.6.4—physical-subtype correctness implies physical type safety—is no longer true. This is because we are allowing one scalar type to be treated as another (`char` as `int` in this case).

This highlights another potential limitation with our notion of type safety and our memory model for C. On the other hand, it is not clear that such a subtyping rule can be guaranteed to be safe for any reasonable definition of physical type safety. The problem is one of portability. On some computer architectures this subtyping rule is intuitively type safe, but on others it is most certainly not. On some architectures, `double`s are required to be aligned in memory at double-word boundaries (eight-bytes on many machines), while `char`s (and therefore `char` arrays) have no alignment requirement. Consider the following code:

```
struct A {
    char d[8];
} a;
struct B {
    double d;
};
void set_double(struct B *b)
{
    b->d = 3.14;
}
```

It is *system dependent* as to where the `a` is aligned, but any object of type `struct B` must be double-word aligned. If `struct A` is a subtype of `struct B`, then we can pass a pointer to `struct A` into the `set_double` function. If `a` happens not to be aligned on a double-word boundary then any of the dangerous problems described in Section 4.3 might arise. For example, on a SPARC architecture, this provokes a bus error, which causes the program to crash.

Notice that portability and type safety can be thought of as two different issues. On machines where there are no alignment restrictions, the above code can be considered to be type safe. Even when such a cast is type safe, it is desirable to warn users that it is potentially non-portable code.

4.8 Implementation and Results

In this section, we describe several tools we have developed based on physical-subtype analysis, and present some results from applying these tools to several large software systems.

4.8.1 Implementation and Tools

In this section we describe the basic implementation of the physical-subtype-analysis algorithm, as well as several tools that use the algorithm to analyze C programs. All of the implementations discussed have been written in Standard ML. The tools act on data structures representing C types and abstract syntax trees. The abstract syntax trees can be generated from any preprocessed ANSI C program.[7]

[7] The implementation of the C front end in ML, which generates abstract syntax trees from C programs, has been joint work with Satish Chandra of Lucent Technologies.

Physical-subtype-analysis algorithm

The core physical-subtype-analysis algorithm takes as input two types, t and t', and compares them to determine *to what extent* t is a physical subtype of t'. If t is not a physical subtype of t' according to the rules presented in Section 4.6.2, then several *relaxations* of the subtyping rules are considered:

- *Label mismatches*: The members of `struct`s "line up" as far as offsets are concerned, but have non-matching labels. (See Section 4.6.3.)

- *Flattening*: One `struct` can be identified as a subtype of another `struct` by flattening members that are themselves `struct`s. (See Section 4.5.1 and Section 4.6.3.)

- *Variant-style `union`s*: One `union` can be identified as a subtype of another by applying the rule that treats `union`s as Cardelli-style variants. (See Section 4.7.1.)

- *Pointers to scalars and functions*: Programmers sometimes use pointers to scalar types (such as `char` and `int`) as generic pointers (instead of `void*`). It can also be useful to allow function pointers of different types to be substituted for each other. (See Section 4.7.3.)

- *Bit-field members*: Bit-field members are considered equivalent when they are of the same size. This allows `struct`s having bit-field members to be physical subtypes or supertypes. (See Section 4.7.3.)

- *Array pads*: Byte (`char`) arrays of size n are considered to be subtypes or supertypes (depending on the context) of other types also of size n. Array pads are discussed in Section 4.5.1 and Section 4.7.3.

The algorithm returns a result in one of two forms:

1. *t is a physical subtype of t'* as well as six numbers indicating how many of each of the above relaxations have been invoked in order to identify the subtype relationship. If any of the numbers corresponding to the last four of these relaxations are non-zero, a substitution of an expression of type t for an expression of type t' may compromise type safety (and portability) as explained in Section 4.4.

2. *t is not a physical subtype of t'.*

Cast checker

Given two types, t and t', the *physical-type-cast checker* returns one of three possible results:

1. *Upcast*: If the core physical-subtype-analysis algorithm returns that t is a subtype of t' then the checker returns "upcast" along with the six numbers indicating how many of each of the relaxations were invoked in determining the subtype relationship.

2. *Downcast*: If the core algorithm returns that t is not a subtype of t' then the core algorithm is applied to see to what extent t' is a subtype of t. If the algorithm returns that t' is a subtype of t then the checker returns "downcast" along with the six numbers indicating how many of each of the relaxations were invoked in determining the subtype relationship.

3. *Mismatch*: If the core algorithm determines that t is not a physical subtype of t' and t' is not a physical subtype of t then the checker returns "mismatch".

Type-cast analyzer

Given an abstract syntax tree representation of a C program, the *physical-subtype-cast analyzer* proceeds by traversing the abstract syntax tree and collecting the pairs of types associated with every implicit and explicit cast. Each cast is annotated as being of one of the following:

- *Assignment*: For each assignment expression, an implicit cast is identified if the type of expression on the left side of the assignment differs from that on the right side.
- *Call*: For each call site, an implicit cast is identified for each actual argument whose type differs from its corresponding formal.
- *+1*: For each explicit type cast that is an instance of the `+1` idiom (see Section 4.5.3).
- *Cast*: For each explicit type cast that is not an instance of the `+1` idiom.

After generating the complete list of cast information, the physical-type-cast checker is applied to each pair of types. The result of a type-cast analysis is a list consisting of the following, for each occurrence of a cast:

- The location in the file where the cast occurred.
- The type being cast from.
- The type being cast to.
- The kind of the cast: assignment, call, `+1`, or cast.

- The result of the physical-type-cast checker: upcast, downcast, or mismatch.
- A list of six numbers indicating the number of relaxations needed to identify the subtype relationship. If the result of the cast checker is "mismatch", all six numbers are zero.

`Struct`-subtype analyzer

Given an abstract syntax tree representation of a C program, the ***`struct`**-subtype analyzer* proceeds by traversing the abstract syntax tree and collecting a list of every `struct` type defined in the program. For each pair of `struct` types, t and t', the physcial-subtype algorithm is used to determine to what extent, if any, t is a subtype of t'. The result of a `struct`-subtype analysis is a list consisting of the following, for each pair of `struct` types for which some subtype relation has been identified:

- The subtype.
- The supertype.
- The size (in bytes) of the subtype.
- The size (in bytes) of the supertype.
- The difference between the two sizes.
- A list of six numbers indicating the number of relaxations needed to identify the subtype relationship.

For the C-to-C++ conversion problem, the `struct`-subtype analyzer can help identify potential class hierarchies. The `struct`-subtype analyzer is also a nice complement to the type-cast analyzer. Sometimes, implied subtype relationships are obfuscated by casts to and from generic types (usually `void*`[8]). `Struct`-subtype analysis can assist the manual tracking of such relationships. For example, given the following program:

```
typedef struct {
  int x,y;
} Point;
typedef enum BLUE, GREEN, RED Color;
typedef struct {
  int xx,yy;
  Color c;
} ColorPoint;

void foo()
{
  Point *pp;
  ColorPoint *cpp;
  void *vp;
  ...
  vp = cpp;  /*  upcast from ColorPoint* to void*  */
  ...
  pp = vp;   /*  downcast from void* to Point*     */
}
```

[8]Analysis of `void` pointers will be considered in future work. A brief discussion of the problems involved can be found in Section 4.7.2.

the type-cast anayzer produces the following output:

```
                                                                Relaxations
Line#  From Type        To Type      Kind    Result     L  F  U  P  B  A
------------------------------------------------------------------------
  16   ptr(ColorPoint) ptr(void)   assign  upcast     0  0  0  0  0  0
  18   ptr(void)        ptr(Point)  assign  downcast   0  0  0  0  0  0
```

This leaves us with two questions: Is there a more interesting subtype relationship hiding beneath the `void` pointers? Is the downcast safe? The result of applying the `struct`-subtype analyzer to the same program can shed some light on things:

```
                              Sizes                 Relaxations
Subtype       Supertype    Sub  Super  Diff      L  F  U  P  B  A
-----------------------------------------------------------------
ColorPoint    Point         12    8      4       2  0  0  0  0  0
```

This indicates that `ColorPoint` is a subtype of `Point` assuming two label-mismatch relaxations. So, in this case, the downcast can be considered to be type safe.

4.8.2 Visualizing the Results

For analzying larger systems, it is often useful to visualize the results of physical-subtype analysis graphically. The output of the `struct`-subtype analyzer can displayed as a graph where vertices represent `struct`s and there is an edge from t to t' if t is a physcial subtype of t'. Figure 62 shows a small example of such a graph from the SPEC95 benchmark *vortex*. This graph shows a small “class hierarchy”: The class

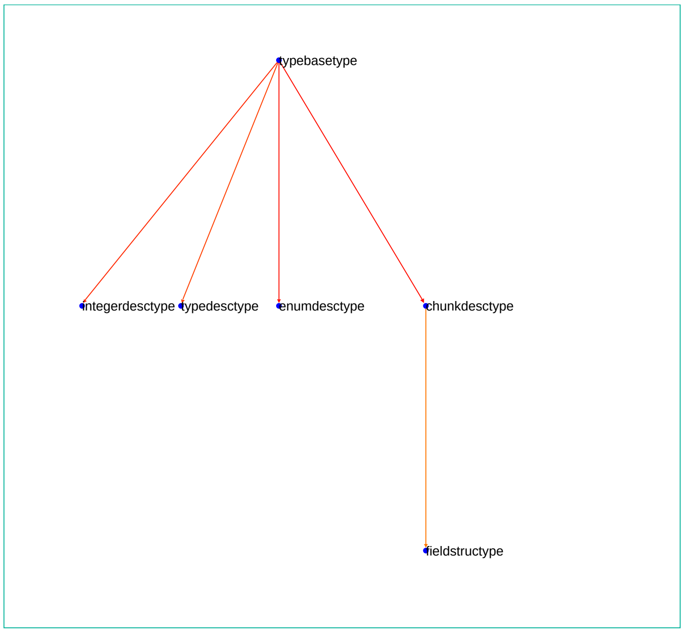

Figure 62: An example of the physical-subtype relation for *vortex*.

hierarchy is a tree with base class `typebasetype` and derived classes `integerdesctype`, `typedesctype`, `enumdesctype`, and `chunkdesctype`, which has as a physical subtype the `struct fieldstructype`.

The output of the type-cast analyzer is also suitable for visualization as a graph. In this case, the vertices might represent types and edges upcast and downcast relationships. Figure 63 shows a set of upcasts found in the *vortex* benchmark. In this graph, a number of pointer types are cast to `Ptr(DrawObj)`, which is a pointer to `struct DrawObj`.

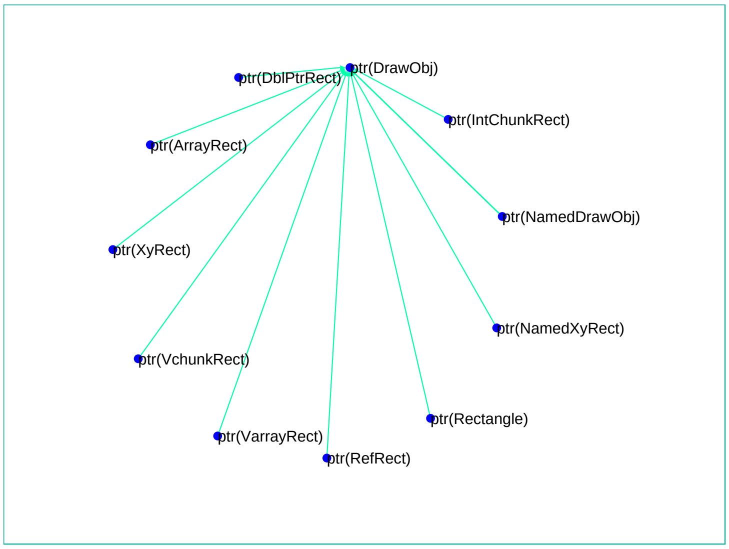

Figure 63: An example of a set of upcasts found in *vortex*.

Bench.	kloc	Casts	sCasts	Up	Down	Mismatch
bash	76	2,202 (309)	180 (65)	147 (50)	22 (4)	11 (11)
binutils	516	3,713 (581)	701 (212)	516 (179)	166 (17)	19 (16)
gcc	208	10,197 (552)	947 (128)	905 (124)	12 (2)	30 (2)
ijpeg	31	1,643 (252)	827 (119)	688 (87)	139 (32)	0 (0)
perl	27	2,526 (231)	806 (77)	791 (73)	3 (1)	12 (3)
vortex	67	3,933 (252)	674 (152)	650 (144)	9 (2)	15 (6)
xemacs	288	5,746 (476)	827 (176)	770 (153)	38 (11)	19 (12)
phone	388	27,364 (2047)	5,559 (488)	1,886 (335)	844 (11)	2,869 (142)

Table 23: Summary of type-cast analysis on several benchmarks. **kloc** is the number of source lines (in thousands) the program, including comments. **Casts** is the total number of casts, implicit and explicit, in the program. The number in parenthesis is the number of such casts between unique pairs of types. **sCasts** is the number of casts, implicit and explicit, between pointers to `struct`s. **Up** is the number of **sCasts** that were classified as upcasts by the physical-subtype analysis, while **Down** is the number of downcasts. **Mismatch** is the number of **sCasts** that are neither downcasts or upcasts.

4.8.3 Benchmark Summary

We applied our tools to a number of C programs from the SPEC95 benchmarks (*gcc*, *ijpeg*, *perl*, *vortex*), as well as GNU's *bash*, *binutils*, and *xemacs*, and a large amount of telephone call-processing code from a Lucent Technologies' product (denoted by *phone*).

Table 23 summarizes the various benchmarks analyzed, in terms of their size, number of casts, and types of casts, as classified by the type-cast analyzer. The number of casts in these programs, which represents a wide-variety of application domains, is non-trivial. Furthermore, we see that a large number of the casts are between pointers to `struct`s, evidence that programmers must reason about the physical-type relationships between `struct`s. Of these casts, the majority are upcasts but a substantial number are downcasts or mismatches (i.e., there is no physical-subtype relationship between the two types at a cast between pointer-to-struct types), which could indicate potential errors.

After running both the type-cast analyzer and the `struct`-subtype analyzer on the benchmarks, we examined some of the cases for which the type-cast analyzer reported a downcast or mismatch. This found a number of potential errors and interesting idioms (including the "`+1`" idiom described in Section 4.5.3), some of which we report on below. Most downcasts we have examined appear to be sensible (at least by our manual examination), but require a more extensive type-checking algorithm, such as the constraint-based algorithm of [16] to determine if the code that uses the downcast is physically type safe.

We note that most of the benchmarks we applied our analyses to are time-tested and robust C programs, so it is perhaps not unsurprising that we found few physical type errors in them. However, the abundance of upcasts and downcasts, as reported in Table 23, indicates that the potential for modifications to introduce physical type errors is significant.

4.8.4 Telephone Call-Processing Code

This section presents an example of a potential type error found by the type-cast tool when applied to a large software subsystem for telephone call processing. The code presented here is not the actual code analyzed.

Message-passing is the common communication mechanism for telephone switching systems, which are massive distributed systems. Such a system may contain over a thousand different kinds of messages. Message format in these systems generally follows the standard header-body paradigm: a header contains meta-information about the message; the body contains the contents of the message. The body itself may consist of another message, and so on. Messages are specified using `struct`s and `union`s.

Typically, a "dispatch" procedure receives a message from the operating system. Depending on the contents of the header, the dispatcher will call other procedures that deal with specialized sets of messages and expect a pointer to a particular message to be passed as an argument. Often, the dispatcher will "look ahead" into the body of message to find a commonly occurring case that requires immediate handling. For example, we found such a dispatcher procedure that declared its view of messages as:

```
struct {
  header hdr;
```

```
  union {
    Msg1 m1;
    Msg2 m2;
    struct { int x; int y; } m3;
  } body;
} M;
```

There are three messages nested inside `Message`, represented by `M.body.m1`, `M.body.m2`, and `M.body.m3`. The first and second messages reference typedef'd `struct`s. Message `m3` is declared inline. Now, the dispatcher contains the following code:

```
if (M.hdr.tag == 3 && M.body.m3.x == 1)
   process_m3(&M);      /* implicit cast */
```

where the function `process_m3` expects a pointer to the following structure:

```
typedef struct { int x; char c; int y; } Msg3;
```

The type-cast analyzer flagged the implicit cast at the call as a "mismatch" because the second field of `Msg3` does not match the type of the second field of the anonymous `struct` represented by `M.body.m3`. Clearly, the code establishes that these two types represent the same message, yet they are incompatible. If the dispatcher were to access field `m3.y` and the procedure `process_m3` had accessed `(&M)->c`, a physical type error would occur. A programmer simply examining the dispatcher may easily insert a reference to `m3.y`, oblivious to this problem.

4.8.5 Identifying Virtual Functions in *ijpeg*

The *ijpeg* benchmark suite (taken from SPECINT95) provides a good illustration of the use of abstract base classes and virtual functions (i.e. type-based code dispatch).

ijpeg provides a set of generic image-manipulation routines that convert an image from any one of a set of input formats to any one of a set of output formats (although the JPEG file format is the usual input or output type). Each image format is represented internally as an abstraction—images have a size, a colormap, a pixel array, etc. The image-manipulation routines are written in a fairly generic fashion, without reference to any specific image format. Components of an image are accessed or changed via calls through function pointers that are associated with each image object. The program initially sets up the input-image and the output-image objects with functions that are format-specific, and then passes pointers to the image objects to the generic image-manipulation routines.

The relevant code fragments (with portions elided for the sake of brevity) are:

```
cjpeg_source_ptr select_file_type(j_compress_ptr cinfo, FILE* infile)
{
  int c = /* get image type */;
  switch (c) {
  case BMP:
      return jinit_read_bmp(cinfo); break;
  case GIF:
      return jinit_read_gif(cinfo); break;
  case TARGA:
```

```
    return jinit_read_targa(cinfo); break;
  case PPM:
    return jinit_read_ppm(cinfo); break;
  /* ... */
  }
  return 0;
}
/* ... */
void start_input_gif(j_compress_ptr cinfo, /* ... */)
{
  gif_source_ptr src = (gif_source_ptr) cinfo; /* DOWNCAST */
  /* ... */
}
void start_input_ppm(j_compress_ptr cinfo, /* ... */)
{
  ppm_source_ptr src = (ppm_source_ptr) cinfo; /* DOWNCAST */
  /* ... */
}
int main()
{
  struct jpeg_compress_struct cinfo;
  FILE* infile;
  j_compress_ptr src_mgr;
```

```
    /* open infile */
    /* process command line arguments and options */

    src_mgr = select_file_type(&cinfo, infile); /* image now loaded */

    /* manipulate image using generic functions */
    (*src_mgr->start_input)(&cinfo, src_mgr);
    jpeg_default_colorspace(&cinfo);
    jpeg_stdio_dest(&cinfo, outfile);
    jpeg_start_compress(&cinfo, TRUE);

    while (cinfo.next_scanline < cinfo.image_height) {
        num_scanlines = (*src_mgr->get_pixel_rows)(&cinfo, src_mgr);
        jpeg_write_scanlines(&cinfo, src_mgr->buffer, num_scanlines);
    }

    (*src_mgr->finish_input)(&cinfo, src_mgr);
    jpeg_finish_compress(&cinfo);
    jpeg_destroy_compress(&cinfo);
    /* ... */
}
```

The `main` routine above and the various `jpeg_` functions that it calls have no notion of the specific input-image type they are dealing with. The selection of the input image type and the initialization of the relevant function pointers and data structures of `jpeg_compress_struct` are done during the call to `select_file_type`. This separation simulates the object-oriented idiom of using abstract base classes and virtual functions to build extensible software libraries.

Each of the image-format-specific functions performs a *downcast* when the function is entered. By examining these downcasts (which were identified by the physical-type-cast analyzer) we were able to track down the virtual-function idiom in *ijpeg*. After the downcast, the format-specific functions execute code that is particular to a given format.

4.9 Related Work

Many of the ideas of this chapter are also discussed in [16].

The identification of classes in the context of the C-to-C++ problem is discussed in Chapter 3. Section 3.7 describes related work in that area. This chapter has examined the related problem of determining how the (user-defined) types in a C program might be organized into a C++ class hierarchy.

The idea of applying alternate type systems to C appears in several places, among them [27, 59, 50, 58, 61]. Most of these references discuss the application of *parametric polymorphism* to C (see Chapter 2), while in this Chapter we discuss the application of *subtype polymorphism* to C. Section 2.10 describes related work pertaining to the application of parametric polymorphism to C.

[59] concerns a new dialect of C that is polymorphic and type safe. This differs from our approach in that it is not aimed at adding polymorphism to existing code. [50] uses

polymorphic type inference on existing C programs, but for determining information about the transfer of values, as opposed to producing reusable code.

The type system developed in this chapter has similarities with several type systems proposed by Cardelli [12, 13, 1]. The primary difference is that we take into account the physical layout of data types when determining subtype relationships, while in Cardelli's work the notion of physical layout does not apply. In particular, there are substantial differences between our notions of `struct` and `union` subtyping and Cardelli's notion of record and variant subtyping. In Cardelli's formulation, a record r is a subtype of a record r' if the set of labels occurring in r' is a subset of those occurring in r and if the type of the members of r' are supertypes of their corresponding members in r. In our system, a `struct` s is a subtype of a `struct` s' if the set of labels *and their offsets* of the members of s' is a subset of those occuring in s, the types of all but the last member of s' *match* the corresponding types in s (i.e., are supertypes and subtypes of the corresponding types in s), and the type of the last member of s' is a supertype of the corresponding member of s. (See Section 4.6.2.) In Cardelli's work, a variant v is a subtype of a variant v' if the set of labels occurring in v is a subset of the labels occurring in v' and each of the types of the members of v is a subtype of the corresponding type in v'. In our system, a `union` u is a subtype of a `union` u', if *each* type of each member of u is a subtype of each type of each member of u'. (See Section 4.6.2 and Section 4.7.1.)

Our work may be contrasted with the pointer-alias analyses described by Steensgaard [61] and Zhang et al. [70]. The common use of pointer-alias analysis is for improving the precision of dataflow analyses for compile-time optimizations. However, pointer-alias detection can also be useful for program-comprehension tools and software-engineering tools. These applications of pointer-alias analysis resemble some of the uses of physical subtyping described in Section 4.8.

In [70], a subtype-like relationship called a *weakly right-regular relation* is defined for pairs of C expressions. The definition of the relation has some of the flavor of the physical-subtyping rules discussed in this chapter. For example, if x is related to x' and both are pointers, then $*x$ is related to $*x'$. However, the system described in [70] has no provision for handling type casts and considers two `struct`s to be related only if they are of equal type, whereas in our system we can relate one `struct` as a physical subtype of another.

Steensgaard, in [61], imposes a "non-standard" type system upon C programs and then uses type inference for the purpose of improving the efficiency and accuracy of points-to analysis. Steensgaard's type system does allow for two `struct`s of differing types to be related, but makes no guarantee of the type safety of this relation. Furthermore, this memory model differs from ours in that it is a *static* model while ours is a *dynamic* model (see Section 4.4).

The tools we have developed based on physical-subtype analysis (see Section 4.8) are related to, but complementary to, such tools as *lint* [36, 35] and *LCLint* [23]. Our tools, as well as *lint* and *LCLint*, can be used to assist in the static detection of type errors that escape the notice of many C compilers. Like *lint*, our type-cast analyzer can be used to identify potential portability problems (see Section 4.7.3), although we recognize a different set of such problems (namely, those arising from alignment restrictions). *LCLint* defines a storage model similar to ours, except it is less precise: no distinction is made among scalar types. *LCLint* can identify problems and constructs that our system cannot—for example, problems with dereferencing null pointers—but only by requiring the user to add explicit annotations to the source code. On the other hand, neither *lint* nor *LCLint* has any notion of subtyping. *Lint* and *LCLint* can improve *cleanliness* of programs. The tools described herein can not only improve cleanliness, but can *guarantee* physical type safety.

4.10 Summary

This chapter has discussed the problem of identifying implicit subtype and subclass hierarchies in C programs. We have formally defined a notion of physical subtypes and built two tools using the definition that can automatically uncover physical-subtype relationships. We have also defined a memory model for C and a notion of physical type safety based on that model. We have proved that for a robust class of C programs, if all type casts are from physical subtypes to physical supertypes then a program is physically type safe. We have also identified several idioms in C programs that represent C++-style object-oriented constructs.

Chapter 5

Conclusions

This thesis describes three techniques that assist with the software-renovation process. The three techniques are aimed particularly at the problem of transforming legacy C programs into C++ programs that make use of C++'s advanced features, including classes, templates, and inheritance. Some aspects of this work apply specifically to the C-to-C++ problem; however, most aspects apply to almost any language.

In Chapter 2, we present an algorithm to generalize C functions that operate on particular types of inputs to C++ function templates that operate on wider varieties of inputs. At the heart of our algorithm is polymorphic type inference. The type-inference mechanism we have developed is based on imposing a non-standard type system on C. Our type system and type-inference algorithm differ from other polymorphic type-inference systems in that we include operator overloading, `struct` subtyping, and constructor introduction. We make use of the standard C type system to recover type information that cannot be inferred from context. The Valid Code Assumption assures that the declared C types are compatible with our inferred types. We have implemented our algorithm in the form of a tool that given a collection C functions as input, returns C++ function templates where possible. We have applied the tool to several medium-to-large sized systems, including the SPEC benchmark.

In Chapter 2, we also discuss how our techniques can be extended to the problem of generalizing C++ classes to C++ class templates. We also show how our generalization techniques can be applied to Standard ML programs. This is particularly interesting because Standard ML already uses polymorphic type inference to assign type to expressions, yet generalization can still be useful for identifying functors from structures and checking explicit type specifications against inferred types.

In Chapter 3, we shift the focus from functions to classes. We show how concept analysis can be applied to the modularization problem, specifically the problem of identifying potential C++ classes from C code. We demonstrate how concept analysis can take advantage of negative information to tease apart "tangled" software components. We define concept partitions, present an algorithm for generating all possible partitions of a concept lattice, and describe how concept partitions can aid the modularization process. We have implemented a tool that uses concept analysis to identify potential modularizations of C programs. As with the generalization tool, we have applied the concept-analysis tool to the SPEC benchmark, as well to as *bash*. We discuss how our techniques are more flexible and powerful than several other modularization techniques. We contrast our work to that of Lindig and Snelting [40] and that of Sahraoui et al. [57], both which also apply concept analysis to the modularization problem. Our results appear to be more successful than the results reported in their work because the concept lattices our tool generates remain of tractable size, even for medium-to-large C programs.

In Chapter 3, we also describe how concept analysis can be used in software-architecture recovery by identifying software components of a coarser grain than classes. We illustrate this by applying concept analysis to Standard ML functors and signatures.

In Chapter 4, we again shift the focus—this time from classes to subclasses. We describe how implicit subtyping relationships can be discovered in C programs and how these relationships are used to emulate C++-style class hierarchies. We define a memory model for C, and define physical type safety based on that model. We identify several idioms in C programs that represent C++-style object-oriented constructs; we discuss which of these idioms correspond to physical subtypes, and which require more complex techniques to be applied to identify them automatically. We define physical subtyping and present formal rules by which the physical-subtype relationship may be inferred. We define the class of physically subtype-correct programs and show that this class of programs is necessarily physically type safe. We describe two complementary tools that identify physical subtypes and use this information to detect subtype hierarchies in large software systems. We also describe how physical subtyping can be used to identify type errors and portability errors that escape the notice of standard C compilers.

A theme that runs through all three chapters is the significance of type theory to the software-renovation process. In particular, all three renovation techniques discussed make use of an analysis of the way types are used in C programs. One of the central ideas behind the generalization process described in Chapter 2 is to impose a parametric polymorphic type system on C programs in place of C's traditional monomorphic types. The type-inference algorithm is based on Hindley-Milner type inference [31, 47], but with the augmentations such as operator overloading, constructor introduction and `struct` subtyping. Physical-subtype detection is also based on the imposition of a new type system on C programs. This type system differs from the one employed in the generalization problem by allowing subtype polymorphism instead of parametric polymorphism. The type system is in some ways similar to other subtyping systems, such as those in [12, 12, 1], but is unique in that it focuses on the storage representation of `struct` and `union` types. The modularization process as applied to the C-to-C++ conversion problem can also be thought of as a type-inference problem: how to infer classes in C program. However, this departs from traditional type-inference problems in that there are almost always multiple different classes that can be inferred from the same code, and thus there is no notion of a *principal* type. The machinery we used to perform this kind of type inference—concept analysis—is very different from the machinery used in traditional work on type inference.

It is difficult to measure accurately the success of the three software-renovation techniques presented in this thesis. It is one thing to demonstrate the feasibility of such techniques—which we have certainly done by applying our implementations to large benchmarks—but it is quite another to verify that they really help with the task of improving the efficiency of development, maintenance, and comprehension of existing software systems. That being said, one clear path for testing the ideas described in this thesis is to apply these ideas to a large software-renovationproject *over time.*

In relation to verifying the utility of the three software-renovationtechniques described in this thesis, it is also desirable to build a robust renovation tool that draws on aspects of all three areas. It is unlikely that the software-renovation process will be fully automated anytime in the near future because there is so much subjectivity involved. A practical tool would allow user input to direct the actions—whether it be generalization, modularization, subtype identification, or some other techniques—to make the renovationprocess more efficient.

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[2] B. L. Achee and Doris L. Carver. A greedy approach to object identification in imperative code. In *Third Workshop on Program Comprehension*, pages 4–11, 1994.

[3] Alex Aiken. Set constraints: Results, applications, and future directions, 1994. http://http.cs.berkeley.edu/ aiken/ftp/ppcp94.ps.

[4] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference, November 1993. http://http.cs.berkeley.edu/ aiken/ftp/fpca93.ps.

[5] Ted. J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability Volume I: Concepts and Models*. ACM Press, 1989.

[6] Ted. J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability Volume II: Applications and Experience*. ACM Press, 1989.

[7] Frederick P. Brooks, Jr. *The Mythical Man Month*. Addison-Wesley, 1995.

[8] G. Caldiera and V.R. Basili. Identifying and analyzing reusable software components. *IEEE Computer*, 24:61–70, 1991.

[9] G. Canfora, A. Cimitile, and M. Munro. An improved algorithm for identifying objects in code. *Software — Practice and Experience*, 26(1):25–48, January 1996.

[10] G. Canfora, A. Cimitile, M. Tortorella, and M. Munro. Experiments in identifying reusable abstract data types in program code. In *Second Workshop on Program Comprehension*, pages 36–45, 1993.

[11] G. Canfora, A. De Lucia, G. A. Di Lucca, and A. R. Fasolino. Recovering the architectural design for software comprehension. In *Third Workshop on Program Comprehension*, pages 30–38, 1994.

[12] Luca Cardelli. A semantics of multiple inheritance. In G.Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, number 173 in Lecture Notes in Computer Science, pages 51–68. Springer-Verlag, 1984.

[13] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[14] Frank M. Carrano, Paul Helman, and Robert Veroff. *Data Abstraction and Problem Solving with C++: Walls and Mirrors*. Addison-Wesley, 1998.

[15] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):341–447, May 1995.

[16] Satish Chandra, Thomas Ball, Krishna Kunchithapadam, Thomas Reps, and Michael Siff. Physical type checking in c via physical subtyping. submitted to the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages, July 1998.

[17] A. Cimitile, M. Tortorella, and M. Munro. Program comprehension through the

identification of abstract data types. In *Third Workshop on Program Comprehension*, pages 12–19, 1994.

[18] A. Cimitile and G. Visaggio. Software salvaging and the call dominance tree. *Journal of Systems Software*, 28:117–127, 1995.

[19] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, 1990.

[20] James O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison Wesley, 1992.

[21] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, January 1982.

[22] B.A. Davey and H.A. Priestley. *Introduction to lattices and order.* Cambridge University Press, 1990.

[23] David Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 44–53, May 1996.

[24] David Garlan and Dewayne Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, April 1995.

[25] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, 1994.

[26] R. Godin and R. Missaoui H. Alaoui. Incremental concept formation algorithms based on Galois (concept) lattices. *Computational Intelligence*, 11(2):246–267, 1995.

[27] F.-J. Grosch and G. Snelting. Polymorphic components for monomorphic languages. In R. Prieto-Diaz and W.B. Frakes, editors, *Advances in software reuse: Selected papers from the Second International Workshop on Software Reusability*, pages 47–55, Lucca, Italy, March 1993. IEEE Computer Society Press.

[28] Carl A. Gunter. *Semantics of Programming Languages*. The MIT Press, 1992.

[29] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual*. Prentice Hall, 1991.

[30] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 1992.

[31] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the AMS*, 146:29–60, 1969.

[32] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[33] David H. Hutchens and Victor R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, SE-11(8):749–757, August 1985.

[34] ISO/IEC. *Programming langauges—C*. Number 9899. ISO/IEC, 1990.

[35] S. C. Johnson. Lint, a C program checker, July 1978.

[36] S. C. Johnson and D. M. Ritchie. UNIX time-sharing system: Portability of C programs and the UNIX system. *Bell Systems Technical Journal*, 57(6):2021–2048, 1978.

[37] Paris C. Kannellakis, Harry G. Mairson, and John C. Mitchell. *Computational Logic: Essays in Honor of Alan Robinson*, chapter Unification and ML Type Reconstruction, pages 444–478. MIT Press, 1991.

[38] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[39] Thomas Kunz. Evaluating process clusters to support automatic program understanding. In *Fourth Workshop on Program Comprehension*, pages 198–207, 1996.

[40] Christian Lindig and Gregor Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th International Conference on Software Engineering*, pages 349–359, 1997.

[41] Barbara Liskov. Data abstraction and hierarchy. *SIGPLAN Notices*, 23(5), May 1988.

[42] Sying-Syang Liu and Norman Wilde. Identifying objects in a conventional procedural language: An example of data design recovery. In *Conference on Software Maintenance*, pages 266–271. IEEE Computer Society Press, November 1990.

[43] Panos E. Livadas and Theodore Johnson. A new approach to finding objects in programs. *Software Maintenance: Research and Practice*, 6:249–260, 1994.

[44] David Melski and Thomas Reps. Interconvertibility of set constraints and context-free language reachability. In *PEPM '97: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 74–89, New York, June 1997. ACM.

[45] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall International, 1988.

[46] Hafedh Mili, Fatma Mili, and Ali Mili. Resuing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6), June 1995.

[47] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[48] John C. Mitchell. *Handbook of Theoretical Computer Science*, volume B, chapter Type systems for programming languages, pages 365–458. The M.I.T. Press/Esevier, 1990.

[49] Philip Newcomb. Reengineering procedural into object-oriented systems. In *Second Working Conference on Reverse Engineering*, pages 237–249, July 1995.

[50] Robert O'Callahan and Daniel Jackson. Detecting shared representations using type inference. Technical Report CMU-CS-95-202, Carnegie Mellon University, September 1995.

[51] Robert O'Callahan and Daniel Jackson. Practical program understanding with type inference. Technical Report CMU-CS-96-130, Carnegie Mellon University, May 1996.

[52] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[53] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[54] Rubén Prieto-Díaz and editors William B. Frakes. *Advances in Software Reuse*. IEEE Computer Society Press, 1993.

[55] Charles Rich and Richard C. Waters. Formalizing reusable software components in the Programmer's Apprentice. In Ted. J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability Volume II: Applications and Experience*, chapter 15, pages 313–343. ACM Press, 1989.

[56] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[57] Houari A. Sahraoui, Walcélio Melo, Hakim Lounis, and François Dumont. Applying concept formation methods to object identification in procedural code. Technical Report CRIM-97/05-77, CRIM, 1997.

[58] Michael Siff and Thomas Reps. Program generalization for software reuse: From C to C++. In *Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 135–146, San Francisco, October 1996.

[59] Geoffrey Smith and Dennis Volpano. Towards an ML-style polymorphic type system for C. In *1996 European Symposium on Programming*, April 1996. to appear.

[60] Gregor Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Transactions on Software Engineering and Methodology*, 5(2):146–189, April 1996.

[61] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Proceedings of the 1996 International Conference on Compiler Construction*, number 1060 in Lecture Notes in Computer Science, pages 136–150. Springer-Verlag, April 1996.

[62] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1991.

[63] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.

[64] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[65] Rudolf Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. In Ivan Rival, editor, *Ordered Sets*, pages 445–470. NATO Advanced Study Institute, September 1981.

[66] Linda Wills. *Automated Program Recognition by Graph Parsing*. PhD thesis, Massachusetts Institute of Technology, July 1992.

[67] Alexander Yeh, David R. Harris, and Howard B. Reubenstein. Recovering abstract data types and object instances from a conventional procedural language. In *Second Working Conference on Reverse Engineering*, pages 227–236, 1995.

[68] Mansour Zand and Mansur Samadzadeh. Special issue on software reuse. *Journal of Systems Software*, 30(3), September 1995.

[69] Amy Moorman Zaremski and Jeannette M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, April 1995.

[70] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 81–92, San Francisco, October 1996.

Appendix A

A.1 Correctness of the concept-partition algorithm

We assume familiarity with the basic definitions for contexts, concepts, and concept lattices given in Section 3.2, and concept partitions given in Section 3.4. Recall that a *concept* is a pair of sets—a set of objects (the *extent*) and a set of attributes (the *intent*) (X, Y)—such that $Y = \sigma(X)$ and $X = \tau(Y)$. A concept is uniquely identified by its extent. In the following, we abuse the language slightly by referring to a concept as a set when really we mean to treat the extent of the concept as a set, which it is. For example, "the union of two concepts" refers to the concept identified by taking the smallest concept which has an extent containing the union of the extents of the two concepts.

Recall the following definitions:

- *Well-formed context*: A context $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ is *well-formed* if and only if, for every pair of elements $x, y \in \mathcal{O}$, $\sigma(\{x\}) \subseteq \sigma(\{y\})$ implies $\sigma(\{x\}) = \sigma(\{y\})$. (See page 106.)

- *Atomic partition*: An *atomic partition* of a concept lattice is a concept partition consisting of exactly the concepts with smallest extent containing each of the objects treated as a singleton set (the atomic concepts). (See page 105.)

We say that a set of concepts q *blankets* a concept c if the extent of c is a subset of the union of the extents of q.

Lemma A.1 *The concept lattice derived from a well-formed context has an atomic partition.*

***Proof*:**

For any context, the collection of atomic concepts is $\{(\tau(\sigma(\{x\})), \sigma(\{x\}))|x \in \mathcal{O}\}$

By extensivity, $x \in \tau(\sigma(\{x\}))$ and thus the union of the extents of atomic concepts is equal to the set of all objects.

If $\sigma(\{x\}) = \sigma(\{y\})$ then x, y are in the extent of the same atomic concept.

Suppose $\sigma(\{x\}) \neq \sigma(\{y\})$.

For any $z \in \tau(\sigma(\{x\}))$, $\sigma(\{x\}) \subseteq \sigma(\{z\})$.

By well-formedness, $\sigma(\{x\}) = \sigma(\{z\})$.

So, $z \notin \tau(\sigma(\{y\}))$.

Likewise, if $z \in \tau(\sigma(\{y\}))$ then $z \notin \tau(\sigma(\{x\}))$

Thus, either $\tau(\sigma(\{x\})) = \tau(\sigma(\{y\}))$ or $\tau(\sigma(\{x\})) \cap \tau(\sigma(\{y\})) = \emptyset$

$\therefore$ The concept lattice derived from a well-formed context has an atomic partition.

$\square$

Lemma A.2 *If c is a non-atomic concept in a concept lattice derived from a well-formed context, then the set of atomic concepts that are subordinate to c in the lattice form a partition of c.*

Proof:

Let q be the set of atomic concepts that are subordinate to c. By the definition of an atomic concept, the concepts in q are pairwise disjoint.

Consider any x in the extent of c. The atomic concept containing x (i.e., $\tau(\sigma\{x\})$ must be a member of q, so x is in the union of the extents of q. So q blankets c.

$\therefore$ Concept set q is a partition of c.

$\square$

Lemma A.3 *If c is a non-atomic concept in a concept lattice derived from a well-formed context, then there exists a $q \subseteq subs(c)$ and a $c' \in q$ that is covered by c such that the set of extents of the elements of q partition the extent of c.*

Proof:

Let c' be any concept that is covered by c (i.e., $c \in \text{covs}(c')$). Let q' be the set of atomic concepts that are subordinate to c'. By Lemma A.2, q' partitions c'.

Let q be the set consisting of c' and the atomic concepts that are subordinate to c except for those atomic concepts in q'. By definition, q is disjoint.

Consider any x in the extent of c. The atomic concept containing x (i.e., $\tau(\sigma\{x\})$ must either be in q or its extent must be a subset of c'. Either way, x is a member of the union of the extents of q. So q blankets c.

$\square$

Definition 1 (Level order for a concept lattice) *Given a concept lattice, a **level order** for the lattice is a linear order induced by a breadth-first navigation through the lattice starting at the bottom element, not including the top element. The top element is defined to be greater than all the other elements of the lattice.*

Definition 2 (Partition order) *Given any two partitions p, p' of a concept lattice and a level order for the lattice $<_c$, suppose the concepts in p are $c_{\alpha_0}, \ldots, c_{\alpha_{j-1}}$ such that $c_{\alpha_{j-1}} <_c \ldots <_c c_{\alpha_1} <_c c_{\alpha_0}$ and the concepts of p' are $c_{\beta_0}, \ldots, c_{\beta_{k-1}}$ such that $c_{\beta_{k-1}} <_c \ldots <_c c_{\beta_1} <_c c_{\beta_0}$. We then can think of p as the "word" $c_{\alpha_0} c_{\alpha_1} \cdots c_{\alpha_{j-1}}$ and the word p' as the "word" $c_{\beta_0} c_{\beta_1} \cdots c_{\beta_{k-1}}$. We define the **partition order** on the set of partitions of the concept lattice to be the lexicographic ordering of these "words."*

Lemma A.4 *In the partitioning algorithm (Figure 51, page 114), every $p \in P$ is a partition.*

Proof: By induction on the number of iterations in the algorithm.

Base case: $p = \text{covs}(\perp)$, so p is the atomic partition (i.e. the partition consisting of all the atomic concepts). The partition algorithm assumes the context to be well-formed and hence p is a partition by Lemma A.1.

Induction hypothesis: For iterations less than i, all p inserted into P at line [12] are partitions.

Induction step: At iteration i, if p'' is inserted into P at line [12], then $p'' = p - \text{subs}(c') \cup \{c'\}$. The union of the extents of p'' is equal to $\mathcal{O}$ because the union of the extents of p is equal to $\mathcal{O}$ and the union of the extents of the concepts subordinate to c' is a subset of the extent of c'. Line [9] ensures that the objects in the extent of c' are disjoint from $p - \text{subs}(c')$. Thus p'' is also a partition.

$\square$

Lemma A.5 *If p is any partition of a concept lattice derived from a well-formed context, then the partitioning algorithm (Figure 51, page 114) finds p.*

Proof: By Lemma A.1, there is at least one partition, namely the atomic partition. Suppose the claim is false. Fix a partition order for the partitions of the lattice. Suppose p_* is the least such partition (according to the partition order) that is not found by the algorithm. We now show that this assumption leads to a contradiction.

The atomic partition is discovered by the algorithm, so p_* must contain a non-atomic concept; call it c_*. By Lemma A.3, there exists a $q \subseteq \text{subs}(c_*)$ and a $c_0 \in q$ that is covered by c_* (i.e., $c_* \in \text{covs}(c_0)$) in the concept lattice such that the set of extents of the elements of q partitions the extent of c_*. Let $p_0 = p_* - \{c_*\} \cup q$. p_0 is a partition because $c_* = \bigcup q$ and the elements of q are pairwise disjoint. $p_0 <_p p_*$ because for all $c \in q$, $c <_c c_*$. By hypothesis, p_0 is discovered by the algorithm. Because $c_* \in \text{covs}(c_0)$, at some iteration of the while loop beginning at line[4] the p in line [5] is p_0; at some iteration of the for loop beginning at line [6], c is c_0; for some iteration of the for loop beginning at line [7], c' is c_*; and finally, at line [12], p'' is p_*. Therefore, partition p_* is discovered by the algorithm, which contradicts our assumption. That is, there is no such least partition, and hence all partitions are discovered by the partitioning algorithm.

□

Theorem A.6 *Given a concept lattice derived from a well-formed context, the partition algorithm (Figure 51, page 114) finds exactly all partitions of the lattice.*

***Proof*:** Immediate from Lemma A.4 and Lemma A.5.

□

Index