Program Generalization for Software Reuse: From C to C++

Michael Siff and Thomas Reps
University of Wisconsin-Madison
1210 West Dayton Street
Madison, WI 53706
{siff, reps}@cs.wisc.edu

Abstract

We consider the problem of software generalization: Given a program component C, create 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 this paper, we present an algorithm that generalizes C functions via type inference. The original functions operate on specific data types; the result of generalization is a collection of C++ function templates that operate on parameterized types. This version of the generalization problem is useful in the context of converting existing C programs to C++.

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 paper 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 our 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.

We believe that one 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 this paper, we present an algorithm 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 the paper 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 in this paper 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

To appear in the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, October 16-18, 1996, San Francisco, CA.

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 and 5, we discuss the integer-exponentiation function shown in Figure 1 and show how it can be generalized to the exponentiation template shown in Figure 2, 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 1, and C' is the function template shown in Figure 2.[1]

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

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

Figure 1: 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 2: A C++ `power` function template.

The main contribution of this paper 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 paper describes the generalization algorithm, as well as the type-inference system for C on which it is based.

Section 2 discusses the opportunities for generalization that the algorithm is capable of identifying. Section 3 discusses the ways in which the generalization algorithm makes use of the declarations in the functions being generalized. Section 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 5 presents the function-generalization algorithm and illustrates it on the `power` example. Section 6 discusses the implementation of the algorithm. Section 7 concerns related work. Section 8 presents some suggestions for extending our ideas.

[1] We actually depart slightly from the above definition of the generalization problem. For example, with Figures 1 and 2, 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.

2 Using Type Inference to Enable Generalization: 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 [10], 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 structures.* Again, in keeping with the need for a "looser" type system for C, the standard notion of subtype between structures is adopted (i.e., a subtype of a structure type s has

all the fields of s and possibly more). For the C-to-C++ function-generalization problem, this is needed in order to identify opportunities to create function templates with structure arguments. (A C++ function template can be called with structure 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 [6, 10, 11], are discussed in Section 3.)

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 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 3 to sort an array of integers. An examination of the code reveals

```
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 3: A function to sort an array of integers.

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 4.

```
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 4: A function template for `sort`.

The type system treats the standard C operators as operators with polymorphic types. In this case, `<` has type $\forall\alpha.\alpha \times \alpha \rightarrow \iota$.[2] At each occurrence of a standard operator, the generic type is instantiated in the usual way [10]; 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 must be of a class that is equipped with a particular overloaded operator. In the case of Figure 4, 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 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 structure subtyping as the basis for inferring types does not yield a powerful enough generalization algorithm. In particular,

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

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 1 (see page 2). 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 methods are 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 structure 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 2 (page 2) 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 4.)

2.3 Structure Subtyping

Structure subtyping allows us to generalize functions that deal with container structures such as sets, stacks, and queues. We adopt the standard notion of subtype between C structures (based on the presentation of record subtyping in [2]): 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 structures;[3] a structure type s is a subtype of another structure 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 structure:

```
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):

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

In C++, a function template that has a parameter with a structure type can be called with structure arguments that contain more fields than just those accessed from within the template's body. The C++ compiler uses a structure-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 structure subtyping.

By using structure subtyping, the generalizer deduces for the example code given above that the argument type can be a pointer to any structure 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 structure that has a `next` field that is a pointer to a structure that has an `i` field of type `int`. Because `struct IntList` is a subtype of the (implicit) structure type, a `struct IntList *` can be used as an argument to `getNextVal`.

3 Polymorphism, Monomorphism, Declarations, 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 [6, 10, 11]:

[3] As mentioned earlier, and discussed in detail in Section 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.

> 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 [8, 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` $\to$ `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` $\to$ `int`, `int` $\to$ `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` $\to$ `float`. The type-inference algorithm might deduce that f has a polymorphic type, say $\forall\alpha\beta.\alpha \to \beta$; however, if it deduces instead that f has type ι, the generalizer treats f as having the functional type `int` $\to$ `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 $\forall\alpha.\alpha \times \iota \to \alpha$; if `y` is a pointer and `x` numeric, then `+` is treated as $\forall\alpha.\iota \times \alpha \to \alpha$; if `x` and `y` are both numeric, then `+` is treated as $\forall\alpha.\alpha \times \alpha \to \alpha$.
- 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 2.3. 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 [16, 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 will disappear when we extend our techniques 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 (i) polymorphic and (ii) 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 beyond the scope of this paper; however, the following example, adapted from [16, pages 256–257], demonstrates how the generalization paradigm could be used to transform C++ classes into class templates: A class representing a bounded stack of integers is defined in Figure 5. (For simplicity, the stack ignores such issues as underflow and overflow.)

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

Figure 5: `class stack`: A simple stack of integers.

The constructor `stack` creates an empty stack with space allocated for `size` integer elements. The class template `stack`, shown in Figure 6, is what would be created via class generalization. Notice that the sole template argument, `T`, is the type of the element stored in the stack. The fact that the `size` argument of the `stack` constructor is used to allocate an array in the `new` expression is what restricts `size`'s type to be `int`.

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

Figure 6: `class stack`: A class template for stacks.

4 Restraints on Polymorphism and the "Over-Generalization" Problem

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 $\forall\alpha.\alpha \to \alpha$ or $\iota \to \iota$, rather than $\forall\alpha\beta.\alpha \to \beta$.
- Binary operators are given either the type $\iota \times \iota \to \iota$, $\forall\alpha.\alpha \times \alpha \to \iota$, or $\forall\alpha.\alpha \times \alpha \to \alpha$, rather than $\forall\alpha\beta.\alpha \times \beta \to \alpha$, $\forall\alpha\beta.\alpha \times \beta \to \beta$, or $\forall\alpha\beta\gamma.\alpha \times \beta \to \gamma$.

The decision to limit constructor introduction to constants occurring alone on the right-hand side of assignment expressions is, admittedly, somewhat arbitrary. Our feeling was that there are expressions in which we do not wish to introduce constructors, such as `n > 0` in `power`. It is clear that some kind of rule to limit constructor introduction is needed. For example, it is not clear 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 they have the type, numeric × numeric → numeric).

4. *Unary operators*: `++`, `--`, `-`, `~` are given the type: $\forall\alpha.\alpha \to \alpha$. The unary logical operator `!` is given the type $\iota \to \iota$. The address-of operator `&` and the dereference operator `*` are given the types, $\forall\alpha.\alpha \to \alpha$ `ptr` and $\forall\alpha.\alpha$ `ptr` $\to \alpha$, 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 7 [8, page 25]. This version is equivalent to the function given in Figure 1, 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 1, the generalization of Figure 7 results in a template with two type parameters, as shown in Figure 8.

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 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.

We are currently investigating techniques to prevent overgeneralization. One possibility is 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)`.

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

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

Figure 7: The `power` function with two local variables.

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

5 An Algorithm to Generalize C Functions

This section concerns the algorithm for function generalization. The steps of the algorithm are depicted in Figure 9.

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:

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

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 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, examine each function on the worklist in round-robin fashion and perform the following steps:
   (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).

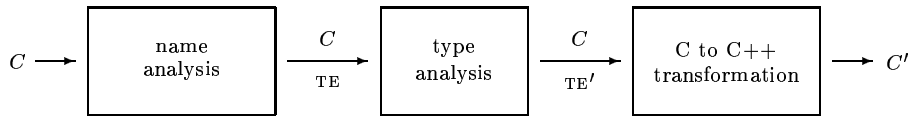

Figure 9: In this process, TE and TE′ are type environments — mappings of program variables to type expressions. The input C need not be a complete C program; it can be any number of C functions contained in files. All we assume is that C's files compile without error according to the ANSI standard.

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

The algorithm terminates because type expressions are regular terms, and there are only a finite number of such terms that can be constructed. We believe the worst-case complexity of the running time of the algorithm to be polynomial, but do not have a proof as yet.

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 3. α is a member of an unbounded set of type variables. The unary type constructor `ptr` represents pointer types. There is no tuple type (since there is no tuple type in C). 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 structure type with k fields. The type is tagged with a type variable α. The tag is necessary for inferring subtypes of the structure type and for representing self-referential structure types. A type expression of the form $\texttt{Typedef}(\nu, \tau)$ is much like a `typedef` in C. It assigns a new type name ν to the type τ. These "named" types allow nested structure 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 [17], 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 cases to consider:

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. If a program contains casts, we make no guarantees about the type safety of the resulting template (but the template will be no more unsafe that the original function).

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 [4]. For example, the rule for function application is:

$$\frac{\text{TE} \vdash f : (\tau_1, \ldots, \tau_n) \rightarrow \tau, \ \text{TE} \vdash e_1 : \tau_1, \ldots, \text{TE} \vdash e_n : \tau_n}{\text{TE} \vdash f(e_1, \ldots, e_n) : \tau}$$

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)`.

  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. As mentioned above, such type distinctions do not affect generalization. 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 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 $\forall\alpha.\alpha \times \alpha \to \alpha$. 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 $\forall\alpha.\alpha \times \iota \to \iota$ 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, array expressions of the form e_0`[`e_1`]` are translated into the expression `*(`e_0 `+` e_1`)`. Suppose `a` is declared an array and `i` is declared an `int`. Because of the translation, `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 structures: 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 structure 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 structure type σ that does not have an l-field, then σ is constrained to be a structure type that has the fields of σ plus an l-field of type β, where β is a fresh type variable. The expression e`.`l has type β.
  3. If e's type is the type variable α, then α is constrained to be a structure type that has an l-field of type β, where β is a fresh type variable. The expression e`.`l has type β.
  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 10: The `MatchAB` function.

As an example, consider the `matchAB` function in Figure 10. The function takes two structures as arguments and returns "true" if the structures 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:` $\forall\alpha_0, \alpha_1, \alpha_2.\alpha_1 \times \alpha_2 \to \alpha_0$`,`
   `rec0:` α_1`,`
   `rec1:` α_2`,`
   `i:` α_3`]`

2. The expression `rec0->a` constrains α_1 to be a structure with an `a`-field that has type α_4, a fresh type variable.
3. `rec1->a` constrains α_2 to be a structure 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 structure 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 structure 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 $\iota \times \iota \to \iota$.
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:

$$\begin{array}{ll} \texttt{[matchAB:} & \forall \alpha_4, \alpha_5, \sigma_0 \preceq \{\texttt{a} : \alpha_4, \texttt{b} : \alpha_5\}, \\ & \sigma_1 \preceq \{\texttt{a} : \alpha_4, \texttt{b} : \alpha_5\}.\sigma_0 \times \sigma_1 \to \iota, \\ \texttt{rec0:} & \sigma_0 \preceq \{\texttt{a} : \alpha_4, \texttt{b} : \alpha_5\}, \\ \texttt{rec1:} & \sigma_1 \preceq \{\texttt{a} : \alpha_4, \texttt{b} : \alpha_5\}, \\ \texttt{i:} & \iota\texttt{]} \end{array}$$

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

- Unions are typed just as structures.
- 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 [16, 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`

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, 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)`.

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 $T \times T \to T$. 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 of 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
```

6 Implementation and Results

We have implemented the C-to-C++ function-generalization algorithm in Standard ML of New Jersey (version 108.5) on a Sun Sparc running SunOS 4.1.3. The current implementation is a tool that requests a file name from the user and outputs templatized C++ code. The input is a C program component that (i) has been pre-processed so that all include files are incorporated and all macros expanded, and (ii) compiles without error or warning according to the ANSI 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 generalization examples used throughout this paper have been produced by the implementation. Preliminary results on larger examples appear promising. One test involved an input file that provides library routines to support the use of binary-decision diagrams (bdds). It consisted 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 bdd structures that have been augmented with additional fields.

A major inhibitor of generalization appears to be the use of the standard I/O functions. 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 I/O library.

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

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 preprocessed. The tool should be able to restore macros so that resultant programs remain "clean."

7 Related Work

Although much has been written about the problem of software reuse (for example, see [19, 9, 14]), including work on identifying reusable components [3, 1], 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 [5]. 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 [20]. 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 paper. 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 [5, 15, 12]. [15] 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. [12] uses polymorphic type inference on existing C programs, but for determining information about the transfer of values, as opposed to producing reusable code.

8 Conclusions and Future Work

This paper 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. However, there are a number of other possible varieties of generalization problems, including the following ones:

- *C structure generalization.* The goal would be to transform one or more structures in a C program or program component into C++ class templates. Some fields would become private members while others would become public. Some functions that operate on these structures would be transformed into member functions, while others would become function templates.

  Because C structure generalization would involve the introduction of encapsulation, this version of the generalization problem raises some additional issues (and is potentially a much more difficult problem).

- *C++ class generalization.* The goal would be to transform a C++ class to a class template. Template arguments would be determined by type inference on the member data fields, both public and private. Member functions would remain essentially the same, except for the types of arguments, local variables, and return values, which would also be determined by type inference.

The potential for combining program-generalization operations with other program-transformation operations, such as program slicing [18, 13, 7], has not escaped our attention. In the scenario we envision, program slicing and generalization would be used in concert to "mine" existing C software for useful components. Slicing would be used to extract an instantiated "proto-component", which in general would be made up of parts of several modules of the original system. Generalization would then be used to convert the slice into template functions or a class template.

Acknowledgements

The comments of Susan Horwitz on this work are greatly appreciated.

This work was supported in part by the National Science Foundation under grant CCR-9100424 and by the Defense Advanced Research Projects Agency under ARPA Order No. 8856 (monitored by the Office of Naval Research under contract N00014-92-J-1937).

References

[1] G. Caldiera and V. R. Basili. Identifying and analyzing reusable software components. *IEEE Comp.*, 24:61–70, 1991.

[2] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[3] A. Cimitile and G. Visaggio. Software salvaging and the call dominance tree. *J. Systems Software*, 28:117–127, 1995.

[4] 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.

[5] 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.

[6] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. AMS*, 146:29–60, 1969.

[7] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[8] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[9] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6), June 1995.

[10] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[11] John C. Mitchell. Type systems for programming languages. In *Handbook of Theoretical Computer Science, Volume B*, pages 365–458. The M.I.T. Press/Elsevier, 1990.

[12] Robert O'Callahan and Daniel Jackson. Detecting shared representations using type inference. Technical Report CMU-CS-95-202, Carnegie Mellon University, September 1995.

[13] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, 1984.

[14] Rubén Prieto-Díaz and William B. Frakes, editors. *Advances in Software Reuse*. IEEE Computer Society Press, March 1993.

[15] Geoffrey Smith and Dennis Volpano. Towards an ML-style polymorphic type system for C. In *1996 European Symposium on Programming*, April 1996. to appear.

[16] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1991.

[17] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.

[18] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[19] Mansour Zand and Mansur Samadzadeh. Special issue on software reuse. *J. Systems Software*, 30(3), September 1995.

[20] Amy Moorman Zaremski and Jeannette M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, April 1995.