The Design and Implementation of Implicit Parameters to Support Function Polymorphism

by

Jason O. Dueck

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba

0-612-23289-1

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

COPYRIGHT PERMISSION

THE DESIGN AND IMPLEMENTATION OF IMPLICIT PARAMETERS TO SUPPORT FUNCTION POLYMORPHISM

BY

JASON O. DUECK

A Thesis/Practicum submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Jason O. Dueck © 1997

Abstract

In a programming language, functions that can be invoked with different numbers of arguments, different types of arguments, or can return results of different types are called polymorphic functions. An overview of polymorphism in modern languages is given.

Partial evaluation is a program optimization process which exploits values known at compile-time with the goal of producing a faster and sometimes smaller program. In the Safer_C language, variables, parameters, and functions can be assigned an evaluation time by the programmer. When one or more parameters of a function have an evaluation time of compile-time, the function may be specialized for their values. Function specialization is one of the means through which Safer_C will support function polymorphism. This thesis describes three new developments in this area: 1) Syntax for declaring implicit formal parameters, 2) Boolean conditions for type inference, and 3) Type manipulation functions.

An extension of function specialization has been developed which allows functions to be specialized for types, as well as for values. Compile-time type and value information may be provided explicitly by the programmer, or it may be described abstractly in an implicit formal parameter list. An implicit parameter passing mechanism obtains the types and values described in such lists from the site of each function invocation. The programmer may place constraints on the specialization process by defining a conditional type matching expression for any explicit parameter. Such constraints provide a means to validate type-specific aspects of each function invocation. They also give the compiler a mechanism to support function overloading. Conditional type matching expressions and the implicit parameter passing mechanism were made possible through the creation of a set of compile-time type manipulation functions. These functions accept types as parameters and return a type or value as their result.

Acknowledgments

The last two years have proven to be a very productive and rewarding time for me. Many people have given me advice and help over my years at this university. While I cannot begin to name them all, I would like to thank the following individuals:

Dr. Peter King, for serving on my thesis committee and entrusting this young punk with four teaching assignments.

Dr. James Peters, for serving on my thesis committee.

Dr. Peter Graham and Dr. Brian d'Aurial for their advice and encouragement.

Lynne Romuld, the All Knowing, All Seeing One for answering my endless questions.

And

My supervisor, Dr. Dan Salomon, whose work ethic and dedication to his students has proven to be an inspiration to me.

Finally, C.B. for her patience and understanding.

I love you, dear.

Contents

List of Tables

Chapter 1

Introduction

The evolution of computer technology, as we know it, would not have been possible without the development of high-level programming languages. These languages allow programmers to think in terms of abstract structures instead of machine instructions. This makes the program development process faster and easier, and the resulting programs are usually easier to understand than their assembly language equivalents.

High-level languages have themselves undergone an evolution. Experience and insight have prompted the creation of a wide array of language features, such as control abstraction and data abstraction. Indeed, typical programs written in current "state of the art" languages bear little resemblance to those written in early FORTRAN.

Polymorphism is another of these language features. In conventional typed languages, every variable or value is assigned one and only one type. These languages are said to be *monomorphic*. By contrast, *polymorphic* languages allow some variables or values to have more than one type. Polymorphic functions, then, are functions whose parameters and results may have more than one type [Car 85].

Built-in polymorphic functions and procedures have existed for quite some time. A good example of a polymorphic procedure is Pascal's *write* statement. The procedure *write* can take any number of parameters; these parameters may be any of the built-in types. User-defined polymorphic functions are a more recent development.

In monomorphic programming languages, programmers are often required to create multiple versions of common algorithms (such as sort or search); one for each data type to which they will be applied. Function polymorphism would allow programmers to create one "generic" version of an algorithm, and apply it to many different data types. As such, their code would be easier to reuse and modify [Cor 88].

This paper will discuss an implementation of function polymorphism which makes use of implicit parameter passing. This method has been implemented as part of the Safer_C/2 programming language developed by Salomon [Sal 95a]. Earlier methods will also be examined, and their limitations discussed.

1.1. Safer_C

The C language, introduced in 1972, has become one of the most popular general-purpose languages in the world. This is largely due to the efficiency and flexibility of the language, as well as its expressive power. The C++ language, introduced in 1986 , has become extremely popular as well. C++ extends the power of C by adding support for object-oriented programming, operator and function overloading, as well as other features.

The term *Safer_C* refers to a series of languages, of which the first two are Safer_C/1 and Safer_C/2. Safer_C/1 is semantically identical to C, but corrects most of its major syntactic flaws. Safer_C/1 has been designed to be more resistant to compile-time and run-time errors than C is, without sacrificing its flexibility and power.

This language has been implemented as a translator (written in ANSI C) that accepts Safer_C/1 code and produces ANSI C code. C was also the target language when C++ was introduced, and helped to facilitate its distribution. In addition, work has begun on a second translator that will permit programmer-aided conversion of C code to Safer_C/1. This will allow programmers to switch to using Safer_C/1 without sacrificing their existing C code.

Safer_C/2 is the second stage of the Safer_C project. This language, which is still under development, will use Safer_C/1 as a base to which a number of modern language features will be added. The goal of the Safer_C/2 designers is to create a language that is at least as powerful as C++, but lacks the syntactic flaws that C++ inherited from C.

Safer_C/2, much like Safer_C/1, is being implemented as a translator (written in ANSI C) which produces ANSI C code. The function-polymorphism method presented in this paper has been applied to this translator. Many of the examples used in this paper are written in Safer_C. For an introduction to this language, see the paper by Salomon [Sal 95a].

1.2. Polymorphism

In general, polymorphic languages allow some variables, values, functions, or function parameters to have more than one type. There are a number of kinds of polymorphism; most programming languages support at least one of these. The following is a summary of the most common classifications of polymorphism, as described by Cardelli and Wegner [Car 85]. For our purposes, these classifications will be described as they apply to function polymorphism.

- *ad-hoc* — A function has more than one interface defined for it. These interfaces allow the function to work with different types and/or numbers of parameters. The types of the parameters do not have to share a common structure. There are two major kinds of ad-hoc polymorphism:
  1. *overloading* — The same name is used by the programmer to refer to a number of different function definitions. The compiler will determine which function to call from the program context. An example of this is the `abs` (absolute value) function in Pascal, where the same name is used to refer to two similar functions that manipulate different types of data (integers and reals).
  2. *coercion* — The same function is called for every invocation. If the parameters or result of an invocation are not of the type expected, they are converted to that

type. This can be done statically (by inserting calls to conversion functions into the invocation) or dynamically (by testing the arguments at run-time).

- *universal* — A function is defined with a single interface that could be applied to an unlimited number of types. This function will accept a fixed number of parameters which share a common structure. Cardelli and Wegner state that a universally polymorphic function will execute the same code for arguments of any admissible type [Car 85]. This requires the ability to resolve function and operator overloading at run-time (using dynamic-binding) and is not often done. Instead, monomorphic versions of such a function will be instantiated at compile-time (either as specified by the programmer or automatically by the compiler). Ideally, the compiler should manage the instantiations and invocations of the monomorphic versions. There are two major kinds of universal polymorphism:
  1. *parametric* — The function has at least one implicit or explicit type parameter. This parameter is used by the compiler to determine the types of the other parameters, and may also be used to determine the types of local variables and to determine which instance of overloaded operators and functions to call inside of the function. This paper describes the design and implementation of a form of parametric polymorphism which makes use of partial evaluation techniques to implicitly obtain the required type information.
  2. *inclusion* — In some programming languages, one or more types (or classes) may be defined that are subtypes of another type. In such a language, when a type is specified in a parameter list, the actual parameter may be a value of that type or any of its subtypes. In object-oriented programming (OOP) languages, this is demonstrated by the supertype-subtype relationship. In "pure" OOP environments, inclusive polymorphic functions are expected to use the object methods of their parameters to perform operations (such as comparison) on them. In such an environment, the dynamic-binding of object methods is a viable option.

This summary is not intended to be a complete guide to this subject, as only the most basic classifications of polymorphism have been included. A number of other classifications have been proposed since the paper by Cardelli and Wegner was published. Two recent developments include *contextual* polymorphism [Dit 94] and *extensional* polymorphism [Dub 95].

Chapter 2

Survey of Function Polymorphism

This chapter will examine a number of languages which support function polymorphism. All but one of these languages are actually in use. The remaining language, ForceTwo, has been implemented in prototype form. Their approaches to function polymorphism are discussed, and the advantages and disadvantages of these approaches presented.

2.1. Ada

Ada was developed by CII Honeywell Bull for the United States Department of Defense [Bar 89]. The first version of Ada was completed in 1980 [DoD 80]. An ANSI standard version (called Ada 83) was established in 1983, and an ISO version in 1987. A new version of this language was released in 1995 [Int 94]. This version is referred to as Ada 95 (or Ada 9X). Function polymorphism is achieved in Ada by use of the *generic* and *attribute* mechanisms and function and operator overloading. As the syntax and usage of these mechanisms has not changed significantly from Ada 83 to Ada 95, both versions are simply referred to as "Ada" in this thesis.

Ada Generics

In a generic subprogram or package definition, some parts of the definition are specified to be generic parameters. Generic parameters can be values, types, or even subprograms. A programmer can then use the definition as a template, and instantiate different versions of the subprogram or package by explicitly supplying the generic parameters. Consider the following generic subprogram (from [Bar 89]):

```
generic
   type ITEM is private;
procedure EXCHANGE (X, Y : in out ITEM);

procedure EXCHANGE (X, Y : in out ITEM) is
   T : ITEM;
begin
   T := X;  X := Y;  Y := T;
end;
```

In this example, the type ITEM is a generic parameter. It should be noted that while the function has been declared, it has not yet been instantiated. To do so, one would have to code statements like the following:

```
procedure SWAP is new EXCHANGE( REAL );
procedure SWAP is new EXCHANGE( INTEGER );
procedure SWAP is new EXCHANGE( DATE );
```

These statements will instantiate three different versions of EXCHANGE; one each for Real, Integer, and Date-type variables. Each of these instantiations will be called SWAP. This duplication of names is valid as Ada permits overloading of procedure names, provided the types or number of parameters are different.

Consider the following generic package (also from [Bar 89]):

```
generic
   MAX : POSITIVE;
   type ITEM is private;
package STACK is
   procedure PUSH (X : ITEM);
   function POP return ITEM;
end STACK;
-- package body not shown
```

In this example, a generic Stack package is declared with two generic parameters. MAX is the maximum size of the stack, and ITEM is the type of the values that it will store. The following statements will instantiate two different versions of this package:

```
package REAL_STACK is new STACK( 100, REAL );
package INTEGER_STACK is new STACK( 50, INTEGER );
```

In addition to values and types, Ada will also permit generic subprogram parameters. This is extremely useful, especially when used in conjunction with overloaded, user-defined operators. The next example discussed will make use of a generic subprogram parameter.

Ada Attributes

All variables, types, and subprograms in a programming language have a number of characteristics which distinguish them. Different data types may have very different characteristics. For example, scalar types have minimum and maximum values, discrete types have a certain order, and array types can contain a certain number of values.

In Ada, these characteristics are known as *attributes*, and they may be referred to in programs. The syntax used by Ada is *T'Attribute*, where *T* is the name of a variable, type, or subprogram, and *Attribute* is the name of the attribute to which one wishes to refer. The following are some simple examples of Ada attributes:

`X'ADDRESS`	- First memory address occupied by object X
`T'FIRST`	- Minimum value of type T
`T'PRED(X)`	- Value preceding X of type T
`T'SUCC(X)`	- Value succeeding X of type T
`A'FIRST`	- Lower bound of first index of array type A
`A'LAST`	- Upper bound of first index of array type A

For a complete list of Ada attributes, refer to [DoD 80] or [Bar 89].

The value of an attribute will be determined in one of two ways. If the value is known at compile time, it will be substituted directly into the object code. If it is not, the attribute will be represented by a call to a function which will determine the value at run time.

Attributes can be extremely useful when designing generic subprograms, as they provide a means for the programmer to refer to distinct characteristics of generic data items. This is demonstrated by the following example (derived from one in [Bar 89]):

```
generic
   type INDEX is (<>);
   type ITEM is private;
   type COLLECTION is array (INDEX range <>) of ITEM;
   with function "<" (X, Y : ITEM) return BOOLEAN;
procedure SORT (C : in out COLLECTION);

procedure SORT (C : in out COLLECTION) is
   MIN : INDEX;
   TEMP : ITEM;
begin
   for I in C'FIRST .. INDEX'PRED(C'LAST) loop
      MIN := I;
      for J in INDEX'SUCC(I) .. C'LAST loop
         if C(J) < C(MIN) then MIN := J; end if;
      end for;
      TEMP := C(I); C(I) := C(MIN); C(MIN) := TEMP;
   end for;
end SORT;
```

This example makes use of both attributes and function parameters. The subprogram `SORT` will receive an unconstrained array of type `COLLECTION`, whose elements will be of type `ITEM`. The range of this array will be the discrete type `INDEX`, but is unknown. `SORT` will also receive a function referred to as "<" which it will use to compare two values of type `ITEM`.

In this case, the range of the array is unknown, as is its type. As such, the boundaries of the two loops are referred to by attributes. These are as follows:

`C'FIRST`	- Lower bound of array
`C'LAST`	- Upper bound of array
`INDEX'PRED(C'LAST)`	- Value preceding upper bound of array
`INDEX'SUCC(I)`	- Value succeeding I

The latter two attributes were used to allow for a variety of range types. If expressions such as `C'LAST-1` and `I+1` had been used instead, this subprogram would only be able to sort arrays with a range of type Integer.

In order for this subprogram to work, it must be able to compare different values of type `ITEM`. To allow for a wide variety of types (including user-defined types), a function which does so must be supplied when this subprogram is instantiated. This function, which will actually be an overloaded operator, is referred to as "<" in this subprogram. Consider the following instantiation:

```
type DATE_ARRAY is array (POSITIVE range <>) of DATE;

function "<" (X, Y : DATE) return BOOLEAN is
begin
   if X.YEAR /= Y.YEAR then
      return X.YEAR < Y.YEAR;
   elsif X.MONTH /= Y.MONTH then
      return X.MONTH < Y.MONTH;
   else
      return X.DAY < Y.DAY;
   end if;
end "<";

procedure SORTER is
   new SORT (POSITIVE, DATE, DATE_ARRAY, "<");
```

Procedure `SORTER` will be capable of sorting unconstrained array variables of type `DATE_ARRAY`. The `POSITIVE` ranges of these array variables will be provided when they are actually declared, and may vary.

The function created to compare two `DATE` values is called "<", and is actually a user-overloaded operator. In the generic function body of `SORT`, the operator used to compare two `ITEM` values is also called "<". When `SORTER` is instantiated, the compiler will determine which version of "<" it should use. In our example, it will do this by comparing the specifications of the available versions of "<" with the one given in the generic procedure declaration.

Ada gives the programmer some flexibility when defining a function parameter specification in a generic subprogram. It is possible to specify a default function to use if one is not provided in the instantiation statement. In fact, there are two ways to do so. Consider the following two function parameter specifications:

```
with function "<" (X, Y : ITEM) return BOOLEAN is <>;
with function NEXT (X : T) return T is T'SUCC;
```

The first statement is similar to the one used in the `SORT` subprogram definition, but has `"is <>"` at the end. If a function is not specified in the instantiation statement, the compiler will automatically bind one with a matching designator and specification, if one is declared at that point in the program. In other words, the compiler will determine the actual type of ITEM (in our case, DATE), and will bind the version of "<" which is declared for DATE at that point.

The second specification is somewhat different, as the ``is T'SUCC" part contains an explicit name for the default parameter. Unlike in the first specification, the binding will occur at the point of the generic subprogram declaration. For this reason, this method will work only if the default function is an attribute, has no parameters depending on formal types, or is itself another formal parameter. The following statement would not work, because ITEM is not known until an instantiation is performed.

```
with function "<" (X,Y:ITEM) return BOOLEAN is LESSTHAN;
```

Discussion

It cannot be said that the generic subprogram mechanism is not expressive or lacks flexibility. It is quite capable of supporting function polymorphism, as has been demonstrated. This method, while effective, has a number of shortcomings.

The generic mechanism is a separate program construct which requires its own syntax. This increases the complexity of programs which use this mechanism. The syntax in question can be quite verbose, as was demonstrated in the SORT generic subprogram declaration. Each generic parameter must be described in terms of its general type, limitations, range, range type, and so on. In fairness, it appears to have been designed this way to aid error checking.

Another shortcoming is the amount of unnecessary work that a programmer must do in order to instantiate different versions of generic subprograms. In the SORT example, the programmer had to specify the types of the range and elements of the array in the instantiation statement. It would have been a trivial task for the compiler to discover these types for itself, from the DATE_ARRAY type declaration.

One can carry this argument even further, and declare that the entire instantiation statement is unnecessary. As an alternative, one could call the generic subprogram directly in the program, and let the compiler create and keep track of the instantiations. Again, it would be a trivial task for the compiler to determine the type of the array parameter, and therefore the types of its range and elements.

Attributes seem to be an effective way of referring to instantiation-specific type characteristics. The syntax of the attribute mechanism is terse but intuitive, which is quite desirable.

2.2. C++

The C++ language [Str 91] evolved from "C with Classes," a series of languages developed at Bell Labs by Stroustrup beginning in 1980. The first C++ translator was implemented by Stroustrup in 1983, although a number of features have been added since then. Work is currently underway on a joint ANSI-ISO standard. The working draft of this standard can be viewed at www.cygnus.com/~mrs/wp-draft. Parametric polymorphism is achieved in C++ through the use of the *template* feature. This feature makes it possible to define generic classes and functions.

C++ Template Definitions

A C++ template consists of a *template* header followed by a class or function definition. Part of the header is a parameter list; this is used to identify the "generic" elements of the template. The parameters of a class template may be classes, character strings, function names, and constant values. The parameters of a function template, though, may only be classes. The following is an example of a class template header:

```
   template <class T, int n>
```

In this template header, parameter `T` is a class and parameter `n` is an integer constant.

C++ Class Templates

Generic classes in C++ are often referred to as *container classes*, as they provide the means to store and manipulate data of a particular type in an abstract way. Stacks, lists, and trees are examples of typical container classes. When objects of a container class are instantiated, the programmer must explicitly specify the template parameters. Consider the following Stack container class, adapted from examples given by Stroustrup [Str 91] and Pohl [Poh 94]:

```
template <class T>  // header (with parameter list)
class stack {
private:
  T*  data;     // array containing data
  int top;      // "top of stack" offset in array
  int maxsize;  // maximum size of stack
public:
  stack(int size)    // constructor
        { data = new T[size];
          top = -1;
          maxsize = size; }
  ~stack()
        { delete []data; }
  void push(T value)
        { data[++top] = value; }
  T pop()
        { return data[top--]; }
  int full() const
        { return (top == maxsize-1); }
};
```

The class parameter list for this template contains a single identifier, `T`. The constructor of this class also contains a parameter, `size`. When objects of class `stack` are defined, classes and values must be explicitly supplied for all template and constructor parameters. For example:

```
stack<char> sc(100);  // 100 element char stack
stack<int> si(500);   // 500 element int stack
```

This will define two stacks from the single template. Stack `sc` is a char stack of size 100, and `si` is an int stack of size 500. These objects will behave as if they belong to two individual classes, and may be used normally. The following function (adapted from [Poh 94]) makes use of our `stack` template to reverse an array of `n` integers:

```
void reverse(int list[], int n)
{
  stack<int> stk(n);       // stack of n integers
  int i;                   // loop counter
  for (i=0; i<n; ++i)
    stk.push( list[i] ); // push elements onto stack
  for (i=0; i<n; ++i)
    list[i] = stk.pop(); // pop off in reverse order
}
```

C++ Function Templates

Function template definitions are constructed in a manner similar to that of class templates. Unlike class templates, though, function templates will be instantiated automatically by the compiler when they are called. For this reason, the types of the function parameters must involve the classes in the template.

A function template definition will actually contain two parameter lists; one for the template and one for the function itself. Values for the template parameters will be determined from the program context and passed implicitly. Values for the function parameters will have to be specified explicitly, as usual.

Function templates allow programmers to create "generic" versions of common algorithms, such as sort or search. The following generic sort function, adapted from [Str 91], demonstrates the use of function templates in conjunction with class templates.

```
// Vector class template
template<class T>
class Vector{
private:
  T*  v;                    // array of type T
  int sz;                   // size of the array
public:
  Vector(int s)
       { v = new T[sz = s]; }
  ~Vector()
       { delete []v; }
  T& operator[] (int i)     // overload subscript operator
       { return v[i]; }
  int size()                // return the size of the vector
       { return sz; }
};

// Bubble Sort function template
template<class T>          // template parameter list
void sort(Vector<T>& v)    // function parameter list
{
  int n = v.size();         // size of vector
  int i,j;                  // loop vars
  T   temp;                 // used for swapping
  for (i=0; i<n-1; i++)
    for (j=n-1; i<j; j--)
      if (v[j] < v[j-1]) // swap
        { temp = v[j];
          v[j] = v[j-1];
          v[j-1] = temp; }
}
```

Objects of class `Vector` contain two data items. The first is an array of type `T`, and the second is an integer variable containing the size of this array. These data items have been encapsulated to simplify the use of searching and sorting algorithms. Such algorithms can call the object method `size` to determine the size of, and therefore the upper bound of, the array in question.

The template parameter list of `sort` contains a single class, `T`. The function parameter list of `sort` tells us that this function will accept a `Vector` as a parameter. When this function is called, the compiler will determine what `T` is by finding the base type of the `Vector` parameter, and will instantiate a version that is capable of sorting items of class `T`. The following block of code will instantiate, initialize, and sort the elements of two vectors.

```
// vector of integers
Vector<int> vec1(3);  // vector with three elements
// array initialization omitted
sort(vec1);           // sort vector

// vector of chars
Vector<char> vec2(4); // vector with four elements
// array initialization omitted
sort(vec2);           // sort vector
```

As with the generic SORT procedure defined in Ada, this function template must be able to compare different values of class T. The vectors defined above were of classes int and char, for which the built-in operator < is defined. The intended meaning of < may differ for other classes, such as char*, and will have no meaning at all for user-defined classes. It is possible to overload operators in C++, but the operands must be classes or enumerated types. Hence, it is not possible to overload < to perform a char-by-char comparison of strings of type char*. There are a number of ways to get around this problem.

The first of these is to define a subclass of Vector (called NewVector) which contains an object method, lessthan, that will perform the comparison operation. In order to deal with unique and user-defined types, the actual definition of lessthan will first be placed inside the Comparator class. This allows us to define a generic version of lessthan for built-in types (such as int), and a special version of lessthan for each unique or user-defined type (such as char*) that we wish to use. In essence, we will be using inclusion polymorphism. Consider the following code:

```
// contains a generic version of lessthan
template<class T>
class Comparator{
public:
  inline static lessthan(T& a, T& b)
    { return a < b; }  // use the built-in operator
};
```

```
// contains a special version of lessthan for char*
class Comparator<char*> {
public:
  inline static lessthan(const char* a, const char* b)
    { return strcmp(a,b) < 0; }  // call strcmp instead
};

// class NewVector is a subclass of Vector and Comparator
template<class T>
class NewVector : public Vector<T>,public Comparator<T> {
public:
  NewVector(int size) : Vector<T>(size) {}
};

// Bubble Sort - lessthan function encapsulated
template<class T>
void sort(NewVector<T>& v)       // changed to NewVector
{
  int n = v.size();
  int i,j;
  T   temp;
  for (i=0; i<n-1; i++)
    for (j=n-1; i<j; j--)
      if (v.lessthan(v[j],v[j-1])) // changed to lessthan
        { temp = v[j];
          v[j] = v[j-1];
          v[j-1] = temp; }
}

// create and sort vectors
main()
{
  NewVector<int> v1(3);   // vector of class int
  NewVector<char*> v2(4); // vector of class char*
  sort(v1);               // uses generic lessthan
  sort(v2);               // uses char* lessthan
}
```

This example encapsulated the comparison operation inside the `NewVector` class, which allowed `sort` to access it. When a `NewVector` object was created, it inherited the methods defined in both `Vector` and `Comparator`. For `v1`, the `lessthan` method defined in the generic `Comparator` template was inherited. In the case of `v2`, the compiler determined that a special `Comparator` class had been created for the `char*` type, so the `lessthan` method defined within it was inherited instead.

If one does not wish to encapsulate the comparison operation inside the class of the parameter passed to `sort`, there are alternatives which use parametric polymorphism.

One of these is to explicitly pass an object of class `Comparator` to the `sort` function, as the following example does.

```
// Bubble Sort - lessthan function passed explicitly
template<class T>
void sort(Vector<T>& v, Comparator<T>& cmp)
                        // changed to Vector, Comparator
{
  int n = v.size();
  int i,j;
  T   temp;
  for (i=0; i<n-1; i++)
    for (j=n-1; i<j; j--)
      if (cmp.lessthan(v[j],v[j-1])) // changed to cmp
        { temp = v[j];
          v[j] = v[j-1];
          v[j-1] = temp; }
}

// create and sort vectors
main()
{
  Vector<int> v1(3);      // vector of class int
  Vector<char*> v2(4);    // vector of class char*
  Comparator<int> c1;     // lessthan of class int
  Comparator<char*> c2;   // lessthan of class char*
  sort(v1,c1);            // pass int objects
  sort(v2,c2);            // pass char* objects
}
```

This approach works, but is rather inelegant. Objects `c1` and `c2` of this example are merely "dummy" objects, used to satisfy the type system. In cases like this, where objects passed to a function contain methods but no data, we may pass their methods implicitly instead. The following version of the `sort` function does just that.

```
// Bubble Sort - lessthan function passed implicitly
template<class T>
void sort(Vector<T>& v)  // changed to Vector only
{
  int n = v.size();
  int i,j;
  T   temp;
  for (i=0; i<n-1; i++)
    for (j=n-1; i<j; j--)
      if (Comparator<T>::lessthan(v[j],v[j-1]))
                            // changed to Comparator<T>
        { temp = v[j];
          v[j] = v[j-1];
          v[j-1] = temp; }
}

// create and sort vectors
main()
{
  Vector<int> v1(3);     // vector of class int
  Vector<char*> v2(4);   // vector of class char*
  sort(v1);              // pass int vector only
  sort(v2);              // pass char* vector only
}
```

Discussion

The template feature of C++ is a reasonably concise and effective method for implementing parametric polymorphism. It is not without its flaws, however.

One problem stems from the fact that function template parameters are limited to classes only. In our Bubble Sort example, we had to encapsulate an array along with its size in the `Vector` class in order to pass both to the `sort` function. It would have been simpler if we could have just passed the "bare" array to the function, and have had the size of the array passed implicitly. For example:

```
template <class T, int n>
void sort(T v[n])
{ // code omitted }
main()
{
  int a[3] = {9, 7, 8};
  sort(a);
}
```

This is not possible in C++, though, because arrays are passed by reference automatically, without any size information. We must pass the size explicitly, as shown below.

```
template <class T>
void sort(T v[], int n)
{ // code omitted }
main()
{
  int a[3] = {9, 7, 8};
  sort(a,3);
}
```

This is unfortunate, because the compiler could easily find the size of an array when it finds the type. Once the size is determined, it could be sent to the function instantiation in a number of ways. Conformant arrays in Pascal, for example, pass the size of an array as a hidden runtime parameter.

In the three Bubble Sort examples, different methods were used to pass an overloaded function as a parameter to a function template. In the first version, which used inclusion polymorphism, the `lessthan` function was encapsulated inside one of the parent classes of the `NewVector` object passed. This was somewhat verbose but reasonable, as doing so is standard practice in object-oriented programming.

The second and third versions used parametric polymorphism. In the second version, the function was passed explicitly inside a "dummy" object. This probably would not be done in practice, as this method is more complex but less convenient than the one used in the third version.

In the third version, the `lessthan` function was passed as an implicit parameter. The syntax required to do so could be improved, as it is non-intuitive and misleading. In fact, it does not appear that we are passing a parameter at all. It can be argued that accessing a method of a "foreign" class is bad practice in object-oriented programming, but that is exactly what we seem to be doing in this case.

The syntax of the template feature may also be problematic. The same syntax is used in both function and class template headers, but the template parameter lists differ (classes only vs. classes, strings, functions, and constants) and the instantiations are done quite differently (automatically vs. manually). This can cause a considerable amount of confusion for novice users, unless a great deal of care is taken to read the "fine print" of the template feature description in a book such as [Str 91] or [Poh 94].

2.3. ML

ML is a functional programming language that was designed for use in theorem proving. It was introduced by Gordon, Milner, and Wadsworth in 1977, and used to create their Edinburgh LCF theorem prover. At present, the version that is most widely used is Standard ML, developed by Milner in 1984 [Pau 91]. Compilers for this language are available from a number of sources.

Cardelli and Wegner have described ML as being "the paradigmatic language for parametric polymorphism" [Car 85]. In this language, it is possible to write functions without specifying the types of the parameters and result. The only difference between a monomorphic function definition and its polymorphic equivalent is that the type information will be omitted from the latter one. In ML, most polymorphic functions involve pairs, lists, and other data structures. Consider the following two function definitions:

```
fun pairself (x : real) = (x,x);
fun pairself x = (x,x);
```

Function `pairself` will accept a value, x, and pair it with itself. The first definition is monomorphic; it accepts a value of type `real`. The type of `x` is unspecified in the second definition, though, so that this version will be able to handle more than one type.

ML Type Schemes

Polymorphism in ML is based on *type schemes*, which serve as a form of template for type checking and type inference. The type scheme of a function is determined and printed when the function is entered. The type scheme of a monomorphic function is quite trivial, as is demonstrated by the following definition and response:

```
fun pairself (x : real) = (x,x);
> val pairself = fn : real -> real * real
```

The response tells us that function `pairself` accepts a parameter of type `real` and produces a pair of values, determined to be of type `real` as well.

If ML is unable to determine the type of a parameter and/or the result, then the type scheme of the function will contain a *type variable* in place of that type. A type variable is denoted as a string of characters starting with a single quote. Consider the following definition and response:

```
fun pairself x = (x,x);
> val pairself = fn : 'a -> 'a * 'a
```

In this case, the types of the parameter and resulting pair is represented by the type variable `'a`.

A type scheme may contain more than one type variable. Each of the following functions contain two and three type variables, respectively.

```
fun fst (x,y) = x;
> val fst = fn : 'a * 'b -> 'a
fun fstfst z = fst(fst z);
> val fstfst = fn : ('a * 'b) * 'c -> 'a
```

Function `fst` will return the first of a pair of values, possibly of different types. Function `fstfst` will return the first value in the first pair of pairs.

ML Type Inference

Standard ML uses a *type inference* mechanism to determine the types of the arguments and result in a function definition. When a function is defined, ML will infer unspecified type information from the program context by following a logical series of steps [Pau 91]. In essence, it will break a function definition down into expressions, and repeatedly apply type-checking rules to resolve ambiguities. As this process runs, types are assigned to arguments, overloaded operators, and expressions. Each argument must have the same type everywhere in the definition. The theory behind the ML type inference mechanism is discussed by Milner in [Mil 78]. Consider the following example, from [Pau 91]:

```
fun facti (n,p) =
   if n=0 then p
   else facti(n-1, n*p);
```

The expressions n=0 and n-1 both contain integer constants, so ML deduces that n is of type int and integer subtraction is to be used in n-1. This identifier also appears in the expression n*p, so p is also of type int and integer multiplication will be used. As p is returned in the base case of facti, its result type must be int as well. Hence, ML will respond with the following statement:

```
> val facti = fn : int * int -> int
```

In the case of the monomorphic version of pairself, ML would know that x has already been assigned a type, real. The result of this function would therefore be a pair of real numbers.

As previously stated, if ML cannot determine the type of an argument or the result, a type variable will appear in the type scheme. When such a function is called, ML will substitute the type of the actual argument(s), perform some more type checking, and instantiate an appropriate version [Car 85]. For this reason, our polymorphic version of pairself could accept a parameter and return a pair of almost any type. For example:

```
pairself 4.0;
> (4.0, 4.0) : real * real
pairself 7;
> (7, 7) : int * int
pairself ("Ozzy", 123);
> (("Ozzy", 123), ("Ozzy", 123))
> : (string * int) * (string * int)
```

In these three function calls, pairself accepted a real, an int, and a pair as arguments and returned the expected results.

Discussion

Type schemes, as discussed, seem to work very well. The creation of polymorphic functions in ML is really a trivial task. This is to be expected, as type schemes are an integral part of the ML language and the compilation process. As is the case in most languages, though, it does not do as well when confronted with odd situations. I will discuss two of these at this time.

The first problematic situation is the declaration and use of references (pointers) to polymorphic functions. It is possible, in theory, to declare a reference to a polymorphic function, assign the reference to another, monomorphic function, and then use that reference to call the monomorphic function indirectly with incompatible parameters. The following code, which is no longer valid in Standard ML, demonstrates this problem.

```
fun I x = x;
> val I = fn : 'a -> 'a
val fp = ref I;
> val fp = ref fn : ('a -> 'a) ref
!fp 5;
> 5 : int
```

Function `I` is the identity function; it returns whatever value (of any type) we send it. The reference `fp` is a pointer to `I`. In ML, the "`!`" character is used to dereference a pointer. By calling `!fp` with the parameter `5` (an integer), we are actually calling `I`; this is a valid call and `5` will be returned. In the next code segment, the assignment of `fp` is changed to `not`. When we do so, we limit the type of the argument and result to `bool`. Hence, when we now call `!fp` with an integer argument, we cause a run-time type error.

```
fp := not;
!fp 5;
```

This problem has been studied by a number of people, including Tofte [Tof 90]. His solution, which has been adopted in Standard ML, is to outlaw the creation of unconstrained polymorphic references. Standard ML contains a special class of *weak type variables* which are used by the type inference mechanism to detect such situations. Each weak type variable may only be assigned, explicitly or implicitly, to a single type in the same part of the program. The following declaration of `fp` contains an explicit type assignment. In this part of the program, `fp` may only refer to functions with a boolean parameter and result. When we call `!fp` with an integer parameter, a compile-time error is caused.

```
val fp = ref (I: bool -> bool);
> val fp = ref fn : (bool -> bool) ref
fp := not
!fp 5;
> Error
```

In the following `let` statement, the type of the weak type variable is determined implicitly to be `bool` when `not` is assigned to `fp`. Hence, calling `!fp` with the parameter `true` is valid and will return the value `false`.

```
let val fp = ref I
in fp := not; !fp true end;
> false : bool
```

Another situation which can cause problems involves the use of overloaded functions (such as + and *). As previously stated, the type inference mechanism will determine the type of an expression from the program context. If an expression contains an overloaded function, and the parameters are type variables, then ML will be unable to determine which version of the function to use and will therefore reject the code containing the usage. Consider the following function definition, from [Pau 91]:

```
fun square x = x*x
> Error - Unable to resolve overloading for *
```

To get around this problem, it is necessary to provide type information explicitly. For example:

```
fun square(x : real) = x*x
> val square = fn : real -> real
```

In this case, we have inserted a type constraint on the argument, limiting it to values of type `real`. ML is now able to select the `real` version of function *. Function `square` can now be used, albeit in monomorphic form only.

2.4. ForceTwo

ForceTwo is part of a family of imperative languages developed by Cormack and Wright [Cor 88]. Other languages in this family include Zephyr [Cor 85] and ForceOne [Cor 87]; ForceTwo was created to address some of the limitations in the design and implementation of these. This language, like the others, was used as a "test bed" for Cormack's and Wright's ideas on polymorphism and type systems. These ideas were later expressed in [Cor 90].

Polymorphism is supported in ForceTwo through facilities such as modules, functions, type generators and type converters, function and operator overloading, and parameters. Of these, the latter is of the most interest to us. ForceTwo supports four kinds of parameters:

- *monomorphic* — the traditional kind, whose type is specified in the function header and whose value is provided in the function invocation.
- *type* — the parameter is a type, and is specified explicitly in the function invocation. The syntax of these parameters is as follows:

  `ident : type`

  The parameter name is `ident`, and `type` is a keyword.
- *query* — the parameter is a type, but it is not specified in the function invocation. Instead, it is obtained from one of the actual parameters. The syntax is as follows:

  `ident_1 : ? ident_2`

  This indicates that type `ident_2` will be obtained from the actual parameter bound to `ident_1`.
- *automatic* — the name of the actual parameter is specified in the function definition, instead of in the invocation. The value (if a variable) or version (if a function or operator) of an automatic parameter will be determined from the program context and bound when the function is called. Automatic parameters were designed to permit the implicit passing of function and operator parameters. The syntax is:

  `auto ident :` **`type_specification`**

  The keyword `auto` precedes the parameter identifier. The `type_specification` can be a simple type (for a variable), or the types of the parameters and result (for a function). An example of an automatic parameter is presented later.

Cormack and Wright refer to query and automatic parameters collectively as *implicit* parameters, as their corresponding actual parameters are not specified in the function invocation. Instead, the actual parameters are obtained by the compiler, through its type inference mechanism [Cor 90].

Monomorphic parameters

The following example demonstrates how different kinds of parameters can be used together to create a polymorphic function. Consider the recursive power function, which has two parameters. The first (x) is a value of some type, and the second (i) is an integer exponent. This function will return x to the power of i; this value will be of the same type as x. To create and call a monomorphic version of power (where x is of type real), we could write the following:

```
power: [x: real, i: integer] real ==
        if i = 1 then x else power[x, i-1] * x

power[2.5, 2] -- returns 6.25
```

Type parameters

The definition above contains two monomorphic parameters. If we wanted to create a more general version of this function, we could add a type parameter, t. This parameter would have to be specified explicitly in the function invocations. For example:

```
power: [t: type, x: t, i: integer] t ==
        if i = 1 then x else power[t, x, i-1] * x

power[real, 2.5, 2]   -- returns 6.25
power[integer, 5, 3] -- returns 125
```

Query parameters

This example now has two problems. The first, and most obvious, is that our function invocations now have an extra parameter. They have become more tedious to write, and look awkward. We can eliminate this by converting t from a type parameter to a query parameter:

```
power: [x: ?t, i: integer] t ==
       if i = 1 then x else power[x, i-1] * x

power[2.5, 2] -- returns 6.25
power[5, 3]   -- returns 125
```

The second, less obvious problem, involves the overloaded operator `*`. ForceTwo allows operator overloading, so * may refer to integer multiplication, real multiplication, or some user-defined operation. The compiler simply will not know which version of this operator it should use in each instantiation of `power`. In order to allow the static binding of the appropriate version at compile time, we could make * a parameter:

```
power: [x: ?t, i: integer, *: [t,t]t ] t ==
       if i = 1 then x else power[x, i-1, *] * x

power[2.5, 2, *] -- returns 6.25
power[5, 3, *]   -- returns 125
```

The type specification of parameter * indicates that this operator will take two operands of type `t` and return a result of that type as well. Unfortunately, we now have to explicitly specify * in our function invocations.

Automatic parameters

In order to pass this operator implicitly, we should convert it to an automatic parameter:

```
power: [x: ?t, i: integer, auto *: [t,t]t ] t ==
       if i = 1 then x else power[x, i-1] * x

power[2.5, 2] -- returns 6.25
power[5, 3]   -- returns 125
```

In this final version, the type of the function and the operator used are both passed as implicit parameters. The actual version of * used in the instantiation will be bound at the site of the function invocation. For this technique to work, a version of * matching the type specification in the formal parameter list must exist in the scope of the invocation; this version will be selected by the overloading resolution mechanism. Consider the following code sequence:

```
(
  *: [a: string, b: string] string == concat[a,b]
  power["abc", 3] -- returns "abcabcabc"
)
power["abc", 3]    -- not valid
```

In ForceTwo, the symbols (and) are used to start and end a new scope, respectively. Inside this new scope, we overload * to perform string concatenation. When we call `power` inside this scope with a string parameter, the new version of * will be bound to its formal parameter and `power` will return the result shown. Once the scope ends, though, this version of * will no longer be available.

Discussion

Cormack and Wright feel that polymorphism is best supported through the use of separate facilities such as modules, functions, parameterized types, overloading, and implicit type and function parameters. This appears to work; we used a combination of implicit parameters and overloading to implement the polymorphic `power` function.

The syntax used in formal parameter lists is terse, but is adequate for declaring implicit type, function, and value parameters. Different versions of polymorphic functions will be instantiated by the compiler as needed, as it done in ML, with the implicit actual parameters determined from the program context. It should be noted that the effort required to use an overloaded `string` operator in a polymorphic function was minimal, especially when compared to the contortions that this author had to perform to do a similar task with a `char*` operator in C++.

Unfortunately, this language is not well documented, and development on it has ceased. Hence, there remain a number of unanswered questions concerning the flexibility of both the syntax and the type inference mechanism used.

- What syntax is required to pass an array to a subprogram? Is it possible to pass the size and type of an array implicitly? Is it necessary to encapsulate the array in a module or define it as a parameterized type in order to do so?

- What syntax is required to pass a record to a subprogram? Is it possible to pass the types and ranges used in the record implicitly? If so, how is this done?
- Is it possible to pass a data item of a parameterized type by reference? If so, what syntax is required, and how is the unspecified type information determined by the type inference mechanism?

2.5. Eiffel

Eiffel [Mey 92] was developed by Meyer at Interactive Software Engineering (ISE) in the late 1980s [Wie 95]. This language was designed as a vehicle for advancing Meyer's ideas about the construction of robust object-oriented programs [Mey 88], and is intended for use in large-scale applications. Compilers for this language are available from at least three sources [Wie 95]. Eiffel is an "almost pure" object-oriented language which uses generic classes, inheritance, and dynamic-binding to support inclusion polymorphism. By doing so, it also supports a form of function polymorphism.

Eiffel has many of the same characteristics as other object-oriented languages, but some of the terminology used (and the reasoning behind it) bears closer examination. It is said that a class in Eiffel contains a number of *features*. There are two types of features:

- *attributes* — Data items defined in a class. Attributes will, as expected, contain the data that the programmer wishes to store in objects of the class. An attribute may also contain data related to other attributes or abstract data types implemented in a class. For example, Eiffel has a pre-defined `ARRAY` class. This class has three attributes (`lower`, `upper`, and `count`) which contain the lower and upper bounds of the array, as well as its size. It should be noted that attributes in Eiffel and Ada are different program mechanisms, although it is possible to use them in similar ways. For example, the pre-defined class `PLATFORM` contains attributes related to platform-specific properties (such as the number of bits used to represent an integer).

- *routines* — Subprograms used to perform some kind of a computation on the attributes of an object. A routine may be either a function or a procedure. A routine call may be in the form of an *instruction* (such as `object.print(1)`) or an *expression* (such as `4-3` where "`-`" is a routine called on object "`4`" with "`3`" as the parameter). A routine call in expression form may use either prefix or infix notation. The only difference between a call in expression and instruction form is the syntax used to denote it in the program.

It should be noted that a routine may be *effective* or *deferred*. An effective routine is the usual type; it is implemented in the same class in which it is defined. For a deferred routine, only the declaration is provided. It will be up to the descendants of the class to provide the implementation details.

In a *generic class*, one or more parameters are used in place of a class name in some feature definitions. These parameters are supplied explicitly by the programmer whenever an object of such a class is declared. One commonly-used generic class is the `ARRAY` class. It is possible to define arrays that may contain data items of virtually any class. For example:

```
int_array   : ARRAY[ INTEGER ];
real_array  : ARRAY[ REAL ];
2d_matrix   : ARRAY[ ARRAY[ INTEGER ] ];
```

As mentioned previously, three of the attributes of this class are `lower`, `upper`, and `count`. Two important routines of this class are `item` and `put`. These routines are used to refer to and assign values to elements of an array. For example:

```
int_array.put(99,1); -- assign 99 to element 1
real_array.item(5);  -- return value of element 5
```

In some situations, it may be desirable to limit a parameter to a member of a particular family of classes. This is usually done to ensure that the class passed contains certain features that may only occur in that family. To do so, one must specify the base class of that family in the formal parameter list of the generic class. Doing so will ensure that only descendants of that base class will be accepted as parameters. This is known as *constrained* genericity.

The Eiffel type system is based on inheritance and *conformance*. As with other object-oriented languages, a subclass will inherit all of the features of its base class. It is possible to redefine these features, and implement any deferred routines that may be defined in the base class. A class is said to *conform* to another class if it is a descendent of that class. In general, class `Y` may be used wherever class `X` is specified as long as `Y` conforms to `X`. For a full explanation of Eiffel's conformance rules, refer to chapter 13 of [Mey 92].

Eiffel uses a combination of generic classes, conformance, and dynamic-binding to support inclusion polymorphism. Generic classes allow the programmer to create program constructs that could be used to contain and manipulate data items of more than one class. In order to manipulate these data items, the routines of a generic class must be capable of performing class-dependent operations on them (such as comparisons). To ensure that these operations (actually routines) have been defined, the parameters of a generic class are generally constrained. The compiler will use the rules of conformance to perform static type checking of these parameters.

When the routine of a parameter is called, an Eiffel program will use dynamic-binding to do so. This is necessary because the version of the routine required will depend on the class (or subclass) of the actual parameter. Because of dynamic-binding, the compiler will only have to generate one version of each routine defined in a generic class. It should be noted that an Eiffel compiler may perform static-binding of a routine call if it determines that only one version of that routine will exist at run-time.

Polymorphic routines are not created explicitly in this language. It is possible to create a polymorphic function implicitly, though, by defining a routine inside of a generic class. Consider the following generic class, `SORTABLE_ARRAY`. This class contains a routine, `sort`, which uses the Bubble Sort algorithm to sort an array of an unspecified type. This example is very similar to the Bubble Sort example discussed in section 2.2.

```
class SORTABLE_ARRAY[ T -> COMPARABLE ]
  creation
    make

  feature -- Public
    data : ARRAY[ T ];
    size : INTEGER;

    make( anArray : ARRAY[ T ] ) is
      do
        data := deep_clone( anArray );
        size := data.count;
      end; -- make

    sort is
      local
        i, j : INTEGER;
        temp : T;
      do
        from i := 1
        until i = size
        loop
          from j := size
          until j = i
          loop
            if data.item(j) < data.item(j-1) then
              temp := data.item(j);
              data.put(data.item(j-1), j);
              data.put(temp, j-1);
            end;
            j := j - 1;
          end;
          i := i + 1;
        end;
      end; -- sort

end -- SORTABLE_ARRAY
```

The `SORTABLE_ARRAY` class has four features; two attributes and two routines. Attribute `data` is an array of type `T`, and `size` will contain the size of the array (an integer value). Routine `make` is invoked when the programmer wishes to instantiate an object of this class; an array is passed to it as a parameter. Routine `sort` will, of course, sort the array.

This generic class has one parameter, T, which is constrained to be a descendant of the COMPARABLE class. The COMPARABLE class contains deferred declarations for comparison routines, such as "<" and ">". This constraint ensures that the "<" routine will be available for use in the sorting routine. Many pre-defined classes, like INTEGER, REAL, and STRING, are descendants of the COMPARABLE class. The following lines will declare three objects, A, B, and C, of the SORTABLE_ARRAY class:

```
A : SORTABLE_ARRAY[ INTEGER ];
B : SORTABLE_ARRAY[ REAL ];
C : SORTABLE_ARRAY[ STRING ];
```

The following lines will instantiate and sort objects A, B, and C.

```
!!A.make( << 1, 2, 3 >> );
!!B.make( << 2.5, 6.7, 1.1, 0.5, 7.8 >> );
!!C.make( << "Axl", "Slash", "Duff", "Matt" >> );
A.sort;
B.sort;
C.sort;
```

A user-defined class may be a parameter for the SORTABLE_ARRAY class as long as it is a descendant of the COMPARABLE class and contains an implementation for the "<" routine. The following incomplete COMPLEX definition is an example of such a class.

```
class COMPLEX
  inherit
    COMPARABLE
  creation
    make
  feature
    real_part : REAL;
    imag_part : REAL;
    make ( re: REAL; im : REAL ) is
    -- code omitted
    infix "<" ( other : like Current ) : BOOLEAN is
    -- code omitted
end -- COMPLEX
```

Discussion

It was previously stated that Eiffel is an "almost pure" object-oriented language. The difference between Eiffel and "pure" OOP languages (like Smalltalk-80) is that Eiffel uses static type checking instead of dynamic typing. By performing type-checking at compile time, the run-time cost of sending a message is reduced to that of an indirect procedure call. This combination of static type checking and dynamic binding results in a language that is both flexible and type-safe. While Eiffel still does not have the run-time efficiency of statically-bound languages, it is considerably more efficient than Smalltalk-80 [Cha 89].

Eiffel supports function polymorphism in the context of inclusion polymorphism, while the other languages examined used variations of parametric polymorphism. As such, this author will not attempt to compare these methods directly. Instead, this author will make note of three language characteristics that have relevance to the work presented in this thesis.

The syntax used in Eiffel is noteworthy in that it employs very few cryptic operators and symbols. The reserved words of this language tend to be descriptive and meaningful. In general, this author has found that code written in Eiffel tends to be both readable and concise, especially when compared to similar programs written in C++.

Eiffel allows the programmer access to information about the data types being used (much like Ada does through its attributes). High-level information is generally stored as attributes in the same class as the data type in question. Examples of such high-level information include the size and bounds of an array. Low-level (and machine-specific) information is obtained by accessing attributes defined in the `PLATFORM` class (which is a base class of all other classes). Through these attributes, the programmer can determine the number of bits required to store an object, the highest supported character code, and so on.

In Eiffel, polymorphism has been integrated into the "core" of the language, much like it was in ML. As such, it requires no special constructs or unnatural syntax. This author feels that such integration is important, because it allows the programmer to use polymorphic features without learning new constructs or having to temporarily adopt a new programming philosophy.

Chapter 3

Related Research

The research described in this paper is fairly unique, as it involves the implementation of a form of parametric polymorphism in a language which supports evaluation-time independence [Sal 92]. A number of partial evaluation techniques (most notably function specialization) are used in the implementation of this language. As such, this research has been influenced and inspired by recent work in both the areas of polymorphism and partial evaluation.

Polymorphism

Much of the current work in polymorphism originates from ideas expressed in a paper by Cardelli and Wegner [Car 85]. In that paper, the authors discuss types, type systems, and polymorphism in great detail. Most publications in this area still refer to that paper, as it contains extensive descriptions of most of the basic polymorphism concepts.

A great deal of work has been done on other classifications of polymorphism. One such classification is contextual polymorphism, developed recently by Ditchfield [Dit 94]. This method uses type-related declarations and assertions to provide a form of parametric polymorphism in a modified version of C.

Another recent development is extensional polymorphism, by Dubois, Rouaix, and Weis [Dub 95]. Extensional polymorphism allows the definition of fully (ad-hoc) polymorphic generic functions in ML by providing a framework for type-checking them.

A good deal of effort has been put into extending and improving the ML type-inference system [Mil 78], especially in regards to how it handles problems caused by imperative programming mechanisms (such as polymorphic references). One such extension proposed by Tofte [Tof 90] has been included in Standard ML. Leroy and Weis

[Ler 91] and Wright [Wri 95] have developed more powerful extensions which provide improved support for imperative programming. Laufer and Odersky [Lau 94] have extended this language so that abstract data types may be treated as first-class values. Harper and Morrisett [Har 95] have recently completed work on a modification of ML in which run-time type analysis may be used in polymorphic functions to determine type information. Finally, Ohori [Oho 95] has developed a new ML-style type inference system based on a second-order record calculus which allows labeled records and labeled variants.

Smith and Volpano have recently addressed the problem of applying ML-style type-inference systems to existing imperative languages [Smi 96a]. They have used this work as a basis for providing polymorphic typing in C [Smi 96b].

Baumgartner and Russo [Bau 95] have developed an interesting language extension for C++ in which abstract type hierarchies may be defined independently of class hierarchies. They believe that, by separating a type definition from its implementation, programmers will have more flexibility when using subtype (i.e. inclusion) polymorphism.

Partial Evaluation

An overview of partial evaluation may be found in papers by Jones [Jon 96], Consel and Danvy [Con 93], and Meyer [Mey 91]. These papers provide a good introduction to the principles behind partial evaluation, as well as many of the problems that may be encountered.

The work described in this paper builds upon ideas developed by Salomon [Sal 92] [Sal 95b] [Sal 96] and implemented in the Safer_C/1 translator. By evaluating source code at compile-time, the translator is able to eliminate the need for preprocessor statements, as well as improve the efficiency of the object code produced.

Previous work by this author [Bal 96] has also influenced the research described here. That report describes an implementation of function specialization for Safer_C. This involved modifying the parse tree representation of function definitions to propagate known values and then perform other optimizations.

This author has found the work of Andersen [And 92] [Jon 93] to be particularly useful. He has created an off-line partial evaluator for a subset of C which is capable of reducing or evaluating expressions, unrolling loops, and specializing functions.

Kleinrubatscher, Kriegshaber, Zochling, and Gluck [Kle 95] have since created a partial evaluator for Fortran programs. Their partial evaluator works by translating a Fortran 77 program into an intermediate form, performing a binding-time analysis and optimizations on it, and then translating it back to Fortran 77 again.

Danvy [Dan 96] has recently published a paper which contains a description of a type-directed partial evaluator. The concepts behind this partial evaluator are discussed in detail, and have a basis in lambda-calculus.

Chapter 4

Function Polymorphism in Safer_C/2

In Chapter 2 of this thesis, it was shown that function polymorphism is currently supported in a number of languages. Parametric polymorphism was used in Ada, ML, and ForceTwo, inclusion polymorphism in Eiffel, and a mixture of both in C++. The syntax used to declare and instantiate polymorphic functions varied considerably between these languages.

This chapter describes some of the mechanisms used by Safer_C/2 to support function polymorphism. These program mechanisms were developed in consultation with my supervisor, D.J.Salomon, and have evolved over time. This chapter also describes many of the design issues that were considered during the development process. Some of the ideas expressed in this chapter were derived from earlier versions of the design; these versions are described in Appendix A.

4.1. Parametric or Inclusion Polymorphism?

Before designing the polymorphic features, an elementary question had to be answered: what types of polymorphism should Safer_C support? As was explained in section 1.2, there are two types of universal polymorphism. They tend to be used in different programming environments, and can require very different implementations.

Parametric polymorphism is used in both imperative and functional languages. Static (i.e. compile-time) type-checking is used to validate the type parameters passed, as well as all operator and function calls involving variables of these types. In most languages, different versions of polymorphic functions are instantiated at compile-time as well, using static-binding. There are languages which use some form of dynamic (i.e. run-time) binding [Har 95] [Mor 91], but this is not often done for parametric polymorphism.

Using static-binding ensures that all of the overhead involved with polymorphic functions will occur at compile-time. The compiler will determine which type parameters are being passed and will then instantiate a number of type-specific versions of these functions. At run-time, these type-specific versions are called where appropriate, in the same way that monomorphic functions are. The main disadvantage of this method is that it is costly in terms of size, as the object code generated may contain a number of separate type-specific versions of each polymorphic function.

Inclusion polymorphism, on the other hand, is used in object-oriented programming environments. As with parametric polymorphism, static type-checking is used to validate each of the function calls. With this method, though, the compiler will only create one version of each polymorphic function in the object code. When such a function is called, the program will use dynamic-binding to determine type information and to resolve function and operator overloading.

While this method is efficient in terms of the size of the object code created, the dynamic-binding process generally imposes some run-time overhead. As such, programs which make use of this method can be slower than equivalent programs which use parametric polymorphism [Cha 89].

Is it necessary to choose between these methods? It appears that both parametric and inclusion polymorphism are useful, but in different contexts. Inclusion polymorphism seems to be a logical and elegant method for providing genericity in object-oriented environments. Parametric polymorphism, as described above, may be used to simplify the creation of ad-hoc polymorphic functions, both at the application and operating system level. Hence, this author believes that both should be supported by Safer_C.

As these methods are quite different, they will be implemented independently. The package of object-oriented features that is currently under development will include support for inclusion polymorphism. This thesis describes the design and implementation of a form of parametric polymorphism.

4.2. Objectives

From the examination of function polymorphism in other languages, it was determined that the design produced for Safer_C should satisfy certain criteria. These are presented below, along with explanations of why each is desirable.

1. *Polymorphic functions should be called and used just like monomorphic functions, with type, range, and subprogram parameters passed implicitly. As such, the task of managing their instantiations should fall to the compiler rather than the programmer.*

   This is the method used in C++, ML, and ForceTwo. Doing so should simplify the use of such procedures, as all of the "extra" programming work that they require will be done once, when they are defined.

   Allowing the compiler to manage the instantiations could produce more efficient code, as it may be able to perform optimizations that the programmer cannot. For instance, some function invocations could involve related parameter types. Instead of instantiating a version for each type, the compiler could create only one and insert type-conversion routines into the invocation statements (as is done in ad-hoc polymorphism). Doing so would reduce the size of the code generated.

   Finally, this method will eliminate the minor but nagging problem of having to create and remember different names for each of the function instantiations. It will be up to the compiler to determine internal names for the different versions. Such names will be created by using an encoding ("name mangling") mechanism.

2. *The syntax used to define polymorphic functions should be reasonably terse, intuitive, and unambiguous.*

   The syntax used by Ada's generic subprogram mechanism is fairly complex and verbose; this increases the effort required to define such subprograms. The Safer_C syntax should be kept reasonably terse to minimize the work needed to define polymorphic functions.

Some of the syntax used by Ada (e.g. "`range <>`") and C++ (e.g. "`(Vector<T>& v)`") is quite cryptic. This author strongly feels that the syntax used by Safer_C should be unambiguous and intuitive. This will increase the "readability" of polymorphic functions, and help to avoid confusion.

3. *Syntactic differences between monomorphic and polymorphic functions should be limited to their headers.*

   In Ada and C++, polymorphic functions are defined using a separate program construct. This increases the complexity of such definitions. It was decided that the Safer_C syntax should be designed so that a separate program construct is not required. Instead, the existing function definition statement should be extended to allow for polymorphic function definitions.

   Even with such an extension, there will have to be some syntactic differences between monomorphic and polymorphic function definitions. Limiting these differences to the function headers should further simplify the definitions.

4. *The syntax of a polymorphic function should support type and function parameters, including user-defined types and overloaded operator parameters.*

   To support parametric polymorphism, the programmer should be able to implicitly pass both "simple" types (such as integer or boolean) and user-defined types (such as arrays, structures, or pointers) to user-defined functions.

   Functions almost always have to manipulate their parameters in some type-specific way. If a function or operator is called from within a polymorphic function, the compiler must type-check the call and determine which version of a function or operator is being called. This could be simplified by passing the function or operator in question as an implicit parameter.

Many functions that have parameters of a user-defined type will refer to some basic element of that type at some point. There should be some way for the compiler to determine what these elements are, and there should be a mechanism which would allow the programmer to access them. This is permitted in Ada through the attribute mechanism. Examples of such elements are the base type of an array or pointer, the index type of an array, and the index range of an array.

5. *The instantiation method used to implement this design should support separate compilation.*

   In large programming projects, the source code is often divided between a number of different text files. These files are compiled separately and the resulting object modules are linked together to form the complete program. The benefits to doing so are fairly well known and will not be discussed here.

   As has been mentioned, managing the instantiations of polymorphic functions should be the responsibility of the compiler. This should be done in a way that allows for separate compilation. In other words, it should be possible to define a polymorphic function in one source file and call it from others.

   For the time being, Safer_C has been implemented as a translator that produces ANSI C code. This code is then compiled and linked using existing tools. The goal, then, is not separate compilation but separate translation. Hence, the solution to this problem must take place at the code generation phase and not at the linking phase.

6. *The overall style of the design should be consistent with the style of the existing features of Safer_C.*

   The syntax used in this design should be similar in style to the existing syntax of this language. Safer_C has been designed to be unambiguous and easy to read. The design produced here should use keywords that are distinct and easy to remember. Most importantly, this design should not require radical changes in the existing language.

Safer_C is a unique language, as it allows the programmer to specify the evaluation time of most of the elements of a program. At present, Safer_C supports both translation-time and run-time execution of source code. The design produced should coexist with, and if possible, exploit this feature of the language.

4.3. Overview

As has been mentioned, Safer_C will allow statements and functions to be evaluated at either compile-time (also known as translation-time) or at run-time. This is known as evaluation-time independence [Sal 92].

This unique language feature has a number of benefits. It has been shown that translation-time statements in Safer_C can be used in the same way that preprocessor statements are in C; this eliminates the need for a separate preprocessor meta-language [Sal 95b]. The evaluation of such statements are carried out by the compiler in a partial evaluation phase. In that phase, a number of partial evaluation techniques (such as constant folding, loop unrolling, and function residualization) are applied to the parse tree with the goal of producing a faster and sometimes smaller program [Sal 96].

One of the techniques applied by the partial evaluator is function specialization [Bal 96]. When the values of one or more of the parameters in a function invocation are known at translation-time, it is possible to create a specialized version of the function in which such values are propagated throughout the function body. After performing constant folding and loop unrolling on the specialized function, it will generally execute faster than the original version.

Function polymorphism has been supported in Safer_C by extending function specialization so that functions may be specialized for types, as well as values. A polymorphic function may be defined through the use of type parameters. A type parameter differs from a normal parameter in that it is a type that is passed at translation-time, instead of a value passed at run-time. When provided with type parameters, the partial evaluator will perform type substitution and manipulation operations on the function to create a specialized version.

Type parameters may be passed to a function explicitly or implicitly. In both cases, the compiler will use this type information to create a specialized (monomorphic) version of the polymorphic function. This is known as *instantiating* a function. It is these specialized versions which will actually be included and called in the resulting program; the original function is used only as a template.

Normally. the keyword `tran` is used in the type specification of a formal parameter to indicate that its value should be passed at translation-time (e.g. `x :: tran int`). In this design, all `type` parameters are, by default, translation-time parameters. For this reason, the `tran` keyword may be omitted from formal type parameter specifications.

By default, all of the parameters in a conventional parameter list are explicit. The following function, therefore, should be passed a type parameter explicitly at translation-time:

```
<<swap>> :: func ( T :: type
                   x :: ->T
                   y :: ->T
                 ) void
block
  temp :: T  !! used for swapping
  temp := x@
  x@ := y@
  y@ := temp
end
```

This function will swap the values referenced by two parameters, `x` and `y`. Both of these parameters are pointers to type `T`, which is also a parameter. This function would be used in the following manner:

```
<<main>> :: func () int
block
  a, b :: int
  c, d :: float
  !! variable initializations omitted
  swap( int, &a, &b )
  swap( float, &c, &d )
end
```

In this case, the compiler would instantiate two versions of `swap`; one for type `int` and one for type `float`.

Ideally, the programmer should not have to pass type information explicitly. Doing so is both inconvenient and unnecessary, as the compiler could easily determine the types of the actual parameters at translation-time. For this reason, an implicit parameter passing mechanism has been developed for Safer_C.

A number of different mechanisms were actually considered. The design described here was the third developed; the previous two are discussed in Appendix A. To gain a deeper understanding of the design issues faced during the development process, first-time readers are encouraged to read the appendix before proceeding further. To allow the reader to compare these designs, the following sections and the appendix share some common examples.

4.4. Implicit Parameters

As has been stated, every parameter in a conventional parameter list is passed explicitly. To allow for the declaration of implicit parameters, it has been decided that programmers should be able to partition a formal parameter list into an "implicit" and an "explicit" section. This is done by preceding each section with the keywords `impl` and `expl`, respectively. If a function requires no implicit parameters, then both keywords should be omitted.

When such a function is called, every parameter in the `expl` section must be provided in the actual parameter list. The compiler will use a type unification algorithm [Aho 86] to determine what the actual implicit parameters are and then instantiate an appropriate version of the function. Consider the following version of the `swap` function:

```
<<swap>> :: func ( impl T :: type
                   expl x :: ->T
                        y :: ->T
                 ) void
block
  temp :: T  !! used for swapping
  temp := x@
  x@ := y@
  y@ := temp
end

<<main>> :: func () int
block
  a, b :: int
  c, d :: float
  !! variable initializations omitted
  swap( &a, &b )
  swap( &c, &d )
end
```

In this version, only the explicit parameters are listed in the actual parameter lists. The type parameter, `T`, is now an implicit parameter. As before, the compiler will instantiate two different versions of this function. It should be noted that implicit parameters are not limited to types; values may be passed implicitly as well.

4.5. Type Manipulation Functions

The `swap` function examined previously is unique, as the two explicit parameters are of the same type. In addition, the only operation that is performed on these parameters is assignment.

In real programming situations, polymorphic functions will almost never be this simple. These functions may be complicated by any number of the following factors:

- They may contain type-specific operators and function calls. The programmer may wish to ensure that these operators and functions are defined for the actual type parameters.
- Type conversions may be required which involve at least one of the type parameters. The programmer may want to ensure that such conversions are possible.

- The programmer may wish to place constraints on the types of the parameters. For example, she may want to ensure that a parameter is an ordinal, an array, a struct, etc.
- If one of the actual parameters is an array, the programmer may wish to have the index range and base type passed implicitly.

To deal with these situations, the developers of Safer_C have created a number of translation-time type manipulation functions. These functions will be evaluated as the function in which they are used is being instantiated. These functions may be used in a number of ways:

- Functions returning a type may be used to initialize an implicit type parameter or a type variable. For example, they may be used to determine the type of a variable or the return type of the function. Two such functions are `WidestType` and `BaseType`.
- Functions returning a boolean may be used as conditions in type matching expressions. These expressions are described in section 4.6. Two such functions are `IsArrayType` and `IsTypeConsistent`.
- Some functions may return values. These are generally used to determine information about one of the parameters. Two such functions are `SizeOf` and `HighBound`.

Table 1 contains the names, type signatures, and meanings of all of the type manipulation functions currently proposed for Safer_C. Some of these functions are based upon Ada attributes, while others are derived from type manipulation functions used by compilers for type-checking. A number of these functions will be examined at length in section 4.6.

Table 1. List of translation-time type manipulation functions

Function name	**Type signature**	**Meaning**
StructCompat(T1,T2)	(Type x Type → Boolean)	Are types T1 and T2 structurally compatible?
NameCompat(T1,T2)	(Type x Type → Boolean)	Do types T1 and T2 have the same name?
SameTypes(T1,T2)	(Type x Type → Boolean)	Are the argument types T1 and T2 the same type or synonyms for the same type?
AssignableTypes(T1,T2)	(Type x Type → Boolean)	Is a variable of type T2 assignable to a variable of type T1?
PromotableTypes(T1,T2)	(Type x Type → Boolean)	Is a value of type T2 promotable to a value of type T1? A value of a numeric type can be promoted to a value of a wider type.
ConvertibleTypes(T1,T2)	(Type x Type → Boolean)	Is a value of type T2 convertible to a value of type T1? A value of one type is convertible to a value of another type if a conversion function is built-in or supplied by the user.
IsOrdinalType(T)	(Type → Boolean)	Is the argument type an ordinal type?
IsNumericType(T)	(Type → Boolean)	Is the argument type a numeric type?
IsRangeType(T)	(Type → Boolean)	Is the argument type a range type?
IsPointerType(T)	(Type → Boolean)	Is the argument type a pointer type?
IsArrayType(T)	(Type → Boolean)	Is the argument type an array type?
IsStructType(T)	(Type → Boolean)	Is the argument type a structure type?
IsUnionType(T)	(Type → Boolean)	Is the argument type a union type?
IsEnumType(T)	(Type → Boolean)	Is the argument type an enumerated type?
IsSignedType(T)	(Type → Boolean)	Is the argument type a signed, as opposed to unsigned, numeric type?

IsTypeConsistent(expr)	(Any → Boolean)	This function evaluates to true if the argument expression is type consistent. This function is usually used to test whether an operator or function exists for certain types. Rather than providing a complex description of the types of the parameters and the desired result, it simply requests that the type-consistency checker be run on its argument, and returns true or false depending on the success or failure of that check.
WidestType(T1,T2)	(Type x Type → Type)	Return the widest type of T1 and T2. Types T1 and T2 must be numeric types. The numeric types are ranked by wideness according to the precision of the numeric value they can hold.
NarrowestType(T1,T2)	(Type x Type → Type)	Return the narrowest type of T1 and T2. Types T1 and T2 must be numeric types. This function mirrors the function WidestType.
SizeOf(T)	(Type → Integer)	Return the size in bytes needed to store a variable of type T.
TypeOf(V)	(Any → Type)	Return the type of the variable V.
BaseType(T)	(Type → Type)	Return the base type of type T. Type T must be a compound type such as an array type or a pointer type.
IndexType(T)	(Type → Type)	Return the index type of an array type.
FieldType(T,I)	(Type x Integer → Type)	Return the type of the I-th field of T, a structure type.
Field(I)	(Integer → Field)	Refers to the I-th field of a structure variable. This would be used in place of the field name when the actual name is unknown.
LowBound(T)	(Type → T)	Return the lower bound of a range type.
HighBound(T)	(Type → T)	Return the upper bound of a range type.

While the merits of each function will not be discussed individually, two design decisions were important enough to deserve an explanation.

4.5.1. IsTypeConsistent

While function `IsTypeConsistent` may seem somewhat awkward to use in practice, this function plays a very important role in polymorphic function definitions. `IsTypeConsistent` is used to ensure that an operator or function has been defined for certain parameter and result types.

As currently designed, `IsTypeConsistent` requires an expression as its parameter. It will type-check the expression to determine if the operator/function used in it is valid for the types of its operands/parameters and result. Two other designs were considered for this function:

1. One version would require a standard parameter list consisting of an operator/function, some parameter types, and a result type.
2. Another version would also require an expression for its parameter, but the expression would contain type names instead of variable names.

In the end, it was decided that the current design would require the least effort to implement and understand, and would require approximately the same effort to use.

Ensuring the Existence of Required Functions and Operators

Two other languages have mechanisms which perform a role similar to that of `IsTypeConsistent`. In an Ada generic subprogram definition, the type signature of required functions should be specified explicitly [DoD 80]. Ada generic subprograms were examined in section 2.1 of this thesis. For the following generic function to be instantiated, the < function must have been declared for the type parameter, `T`, and have a return type of `BOOLEAN`:

```
generic
  type T is private;
  with function "<" (U, V : T) return BOOLEAN is <>;
function MAX (A, B : T) return T;
```

In CLU, a generic procedure or cluster definition must explicitly state the name and type signature of each procedure that the type parameter will provide [Ghe 87]. The following generic cluster definition ensures that the `lessthan` procedure has been defined for type parameter `T` and returns a boolean result:

```
set=cluster [T: type] is create, insert, delete
    where T has lessthan: proctype(T,T) returns (bool)
```

A different approach was taken when designing `IsTypeConsistent`. Both of the above mechanisms used specialised syntax; `IsTypeConsistent` uses none. Instead of explicitly stating the type signature of an operator or function, a Safer_C programmer would use the operator or function in an expression. The type signature would be obtained and tested implicitly.

In practice, `IsTypeConsistent` has turned out to be both flexible and easy to use. Assume that x is a formal (value) parameter of a polymorphic function. The following invocation of `IsTypeConsistent` will ensure that the < operator has been defined for the type of x:

```
IsTypeConsistent( x < x )
```

Additionally, by using the result of the < operation in a boolean expression (formed with the `&&` operator), we may indicate that the result type should be boolean:

```
IsTypeConsistent( (x < x) && 1 )
```

4.5.2. Field and FieldType

For a time, the Safer_C designers did not provide a means to access or determine the (unknown) internal characteristics of a structure parameter. It was considered unlikely that a programmer would want to "blindly" pass a structure to a function and expect the function to deal with it properly (with the possible exception of output functions). For this reason, neither of the earlier designs (described in Appendix A) were given this capability.

There could be situations, though, when a programmer may find such a capability useful. For this reason, functions `FieldType` and `Field` have been defined. `FieldType` may be used to determine the type of one of the fields of a structure, while `Field` may be used to refer to the field itself. `FieldType` requires a structure type and a field number as parameters, while `Field` only requires the field number. In this context, "field numbers" refer to the order in which fields appear in a structure definition. Consider the following definitions:

```
B :: type := struct { a :: int          !! field 1
                      b :: float        !! field 2
                      c :: long int }   !! field 3
A :: B  !! A is a variable of type B
```

Assume that A is passed to a function, and that B itself is passed as a type parameter. If the formal parameter names of A and B are X and Y, respectively, then the fields of this structure could be referred to in the following manner:

```
temp1 :: FieldType( Y, 1 )    !! type int
temp2 :: FieldType( Y, 2 )    !! type float
temp3 :: FieldType( Y, 3 )    !! type long int
X.Field( 1 ) := 1
X.Field( 2 ) := 3.14
X.Field( 3 ) := 40000
```

It should be noted that a number of other designs have been considered for this purpose. While `Field` and `FieldType` have been included here, the designers still consider their syntax to be experimental. As such, these functions may not appear in the final release of Safer_C/2.

4.6. Conditional Type Matching

Many of the type manipulation functions were designed to allow the programmer to place constraints on the types of the actual parameters. This may be done for two reasons:

1. The programmer may want to ensure that a function can only be called if the actual parameters are certain kinds of types (such as numeric types). This may be useful if the function contains operator/function calls which are only defined (or have the same intended meaning or usage) for those types.
2. The programmer may want to overload the name of a polymorphic function by creating several versions which manipulate different types in different ways. Conditional type matching would give the compiler a means to choose between the different versions when an invocation is made.

To facilitate this, each explicit parameter may be accompanied by a boolean type matching expression. Such expressions will be evaluated during the function instantiation process; this process will only succeed if each expression evaluates to true. The syntax used is:

```
type_expression where boolean_expression
```

For example:

```
root :: T where IsPointerType(T) and
             ~~ IsNumericType(BaseType(T))
```

This boolean expression will evaluate to true if `root` is a pointer to a numeric data item. While the boolean expression in this example was somewhat complex, it has been found that in practice, such expressions will often consist of a single function call. It should be noted that "~~" is a statement continuation marker in Safer_C.

When designing this feature, the Safer_C designers considered using the more precise phrase "`such that`" instead of "`where`". In the end, it was decided that "`where`" would probably be less cumbersome to use.

A number of examples will now be presented to illustrate how type manipulation functions are used, in conjunction with conditional type matching, to support function polymorphism in Safer_C.

Example 1

Function `square` will return the square of its parameter, x. The type manipulation function `IsTypeConsistent` is used to ensure that the operator "*" is defined for the type of the parameter.

```
<<square>> :: func ( impl
                          T :: type
                     expl
                          x :: T where
                            ~~ IsTypeConsistent(x * x)
                   ) T
block
  return (x * x)
end
```

Function invocations would take the following form:

```
square( 2 )      !! returns 4
square( 1.5 )    !! returns 2.25
```

If the operator "*" were overloaded to perform string concatenation, the following function call would be valid as well:

```
square( "ab" )   !! returns "abab"
```

Example 2

This example is somewhat more complex, as `max` will accept two parameters of (potentially) different types and return the greatest of them. Function `IsNumericType` is called two times to ensure that the actual parameters in the invocation are of numeric types. It is not necessary to check that ">" is defined for the parameters, because this operator is valid for all numeric types. As the precision of the types of the parameters may vary, `WidestType` is used to determine the result type of this function.

Note that since the evaluation time of `max` has been specified as being translation-time (`tran`), its function body would actually be substituted in place of the function invocation by the partial evaluator. The rationale for doing so is discussed in section A.1.

```
<<max>> :: tran func ( impl
                          T1 :: type
                          T2 :: type
                          W  :: type := WidestType(T1,T2)
                       expl
                          a  :: T1 where
                             ~~ IsNumericType(T1)
                          b  :: T2 where
                             ~~ IsNumericType(T2)
                     ) W
body
  return (a > b ? a : b)
end
```

The following are valid function invocations:

```
max( 11, 99 )      !! returns 99, an int
max( 3.14, 5.0 )   !! returns 5.0, a float
max( 11, 40000 )   !! returns 40000, a long int
```

The following invocation would not be valid:

```
max( 11, "abc" )   !! "abc" is not numeric
```

Example 3

Function `sort` will perform a Bubble Sort operation on a one-dimensional array of any size and base type. This function uses `IsArrayType` and `IsTypeConsistent` to ensure that `data` is indeed an array, and that the "<" operator has been defined for its base type. This function also employs `LowBound`, `HighBound`, and `IndexType` to determine the upper and lower boundaries of its index range. Finally, `BaseType` is used to determine the base type of the array.

It should be noted that `LowBound` was used in this example for the sake of completeness; the lower bound of all arrays in Safer_C (as in C) is `0`. This may, however, change in future versions of the language.

```
<<sort>> :: func (
        impl
            T  :: type
            I  :: type     := IndexType(T)
            lo :: tran int := LowBound(I)
            hi :: tran int := HighBound(I)
            B  :: type     := BaseType(T)
        expl
            data :: T where
                 ~~ IsArrayType(T) and
                 ~~ IsTypeConsistent(data[lo] < data[hi])
        ) void
block
  i, j :: I                        !! loop vars
  temp :: B                        !! used for swapping
  for (i := lo; i < hi; i++)
    for (j := hi; i < j; j--)
      if (data[j] < data[j-1])     !! swap
        temp := data[j]
        data[j] := data[j-1]
        data[j-1] := temp
      endif
    endfor
  endfor
end
```

This function would be used in the following manner:

```
a1 :: [0..99] double      !! declare first array
a2 :: [0..999] ->char     !! declare second array
!! array initializations omitted
!! overloading of "<" for ->char omitted
sort( a1 )                !! sort first array
sort( a2 )                !! sort second array
```

Example 4

In the example below, the previous function has been overloaded so that the base type of the array is a structure, not a simple type. The array must be sorted on a "key" field; the field number (described previously) is passed as an explicit translation-time parameter, `F`. `BaseType` is used to determine the structure type (`S`), and `Field` is used to refer to the key field in both the function header and body. `IsStructType` is used to ensure that the base type of the array is a structure; this will help the compiler to distinguish between the two versions of `sort` during the instantiation process.

```
<<sort>> :: func (
        impl
            T  :: type
            I  :: type      := IndexType(T)
            lo :: tran int  := LowBound(I)
            hi :: tran int  := HighBound(I)
            S  :: type      := BaseType(T)
        expl
            data :: T where
                 ~~ IsArrayType(T) and
                 ~~ IsStructType(S) and
                 ~~ IsTypeConsistent(data[lo].Field(F)
                              ~~   < data[hi].Field(F))
            F    :: tran int
        ) void
block
  i, j :: I                       !! loop vars
  temp :: S                       !! used for swapping
  for (i := lo; i < hi; i++)
    for (j := hi; i < j; j--)
      if (data[j].Field(F) < data[j-1].Field(F))
        temp := data[j]
        data[j] := data[j-1]
        data[j-1] := temp
      endif
    endfor
  endfor
end
```

If both versions of `sort` were defined in the same program, the following code would be valid:

```
!! structure type s1 has four fields
s1 :: type := struct { key   :: int      !! key is field 1
                       a,b,c :: char }
!! structure type s2 has two fields
s2 :: type := struct { d     :: float
                       key   :: float } !! key is field 2
a1 :: [0..99]  double  !! array of double
a2 :: [0..999] int     !! array of int
a3 :: [0..5]   s1      !! array of struct s1
a4 :: [0..19]  s2      !! array of struct s2
!! array initializations omitted
sort( a1 )             !! call first version
sort( a2 )             !! call first version
sort( a3, 1 )          !! call second version
sort( a4, 2 )          !! call second version
```

Example 5

The function in this example will perform matrix addition on a pair of two-dimensional arrays, `A` and `B`. Their sum will be stored in another two-dimensional array, `C`. These arrays must be of the same type, `T1`. They may have any size and base type; as before, `IsTypeConsistent` is used to ensure that the "+" operator is defined for the base type.

In Safer_C (as in C), a two-dimensional array is actually an array of arrays. By applying the `BaseType` function to `T1` (which is actually a one-dimensional array type), the compiler will obtain another array type, `T2`. At this point, `LowBound`, `HighBound`, and `IndexType` may be applied to `T1` and `T2` to obtain the dimensions of the matrices. As before, `LowBound` is used to obtain the low bounds of each "dimension" for the sake of completeness.

```
<<add_matrices>> :: func (
     impl
         T1       :: type
         lo1      :: tran int   := LowBound(IndexType(T1))
         hi1      :: tran int   := HighBound(IndexType(T1))
         T2       :: type       := BaseType(T1)
         lo2      :: tran int   := LowBound(IndexType(T2))
         hi2      :: tran int   := HighBound(IndexType(T2))
     expl
         A, B, C :: T1 where
           ~~ IsArrayType(T1) and
           ~~ IsArrayType(T2) and
           ~~ IsTypeConsistent(A[lo1][lo2] + B[lo1][lo2])
        ) void
block
  x, y :: int
  for (x := lo1; x <= hi1; x++)
    for (y := lo2; y <= hi2; y++)
      C[x][y] := A[x][y] + B[x][y]
    endfor
  endfor
end
```

The following code will declare and then add two different pairs of matrices:

```
m1 :: type := [0..9][0..12] int
m2 :: type := [0..5][0..99] float
a, b, c :: m1
x, y, z :: m2
!! array initializations of a, b, x, y omitted
add_matrices( a, b, c )
add_matrices( x, y, z )
```

Chapter 5

Implementation

This chapter describes the implementation of the mechanisms described in sections 4.3 to 4.6. The implementation process was divided into a number of steps; these are described here in the order in which they were performed. While the implementation was not trivial, it was fairly straightforward and, for the most part, presented no serious technical challenges.

One of the objectives of this project was to implement the design in such a way that separate compilation will be supported. Doing so in the context of the Safer_C language (and translation process) raised a number of implementation issues. These issues are presented separately in section 5.2.

The implementation steps were:

1. **Modify the scanner and the parser**

   A number of changes were required to the scanner and parser to support the syntax described in sections 4.4 and 4.6. Three new keywords (`impl`, `expl`, and `where`) were added to the language. The grammar was modified to allow partitioned formal parameter lists, implicit parameter initializations, and to accept conditional type matching expressions.

2. **Modify partial evaluator**

   As stated in section 4.3, function polymorphism was to be implemented as a form of function specialization using translation-time type substitution and manipulation. The partial evaluator was modified to permit such specializations. Section 5.1 describes how the instantiations are managed.

As part of this step, a version of the type unification algorithm described by Aho, Sethi, and Ullman [Aho 86] was implemented. This algorithm was adapted to work with the type tree structure used by the Safer_C translator, and extended to work with values of ordinal types.

When this step was completed, the Safer_C translator was able to reliably process simple polymorphic functions, such as the two versions of `swap` from sections 4.3 and 4.4. A small test suite of polymorphic functions (employing both explicit and implicit type parameters) were developed to ensure that the specialization process was working properly.

3. **Implement a subset of the type manipulation functions**

   Once step 2 was completed, a subset of the type manipulation functions listed in section 4.5 was implemented. This subset consisted of `IsTypeConsistent`, `WidestType`, `IsNumericType`, `IsArrayType`, `BaseType`, `IndexType`, `LowBound`, and `HighBound`.

   Another small test suite of functions was written; these were used to further test the modifications made in step 2 as well as the functions listed here. At the completion of this step, the translator was able to reliably deal with conditional type matching as illustrated by examples 1, 2, 3, and 5 in section 4.6.

4. **Implement the remaining type manipulation functions**

   The next step was to implement the remaining type manipulation functions. At this point, a comprehensive test suite was developed which tested all of the functions and ensured that the function specializer is capable of dealing with more advanced definitions than those examined previously. The test suite included instances of overloaded function definitions as well.

5. **Extend to support separate compilation**

   The final step in the implementation was to extend the instantiation process so that a polymorphic function defined in one source file can be invoked (and possibly instantiated) in another. The issues involved in separate compilation are examined in section 5.2.

5.1. Managing instantiations in a single source file

Safer_C has been implemented as a translator which produces ANSI C code. The translator works by reading Safer_C code into a parse tree, performing partial evaluation operations on it, and then generating ANSI C code from the modified tree. All polymorphic invocations and instantiations must therefore be resolved before the ANSI C code is generated.

When the implementation process began, function specialization had not been "officially" implemented in Safer_C. This author had previously implemented a version of it [Bal 96], but this implementation only performed specializations for translation-time value parameters. In addition, the programmer was required to instantiate the function versions manually.

To validate the ideas expressed in this thesis, an extended version of the "official" function specialization method, as described by Salomon [Sal 96], has been implemented. This description states that a function should be automatically specialized if:

1. Its evaluation time is declared to be run-time (the default), and
2. One or more of its formal parameters are declared to be translation-time.

The extended version of the specializer had to be capable of dealing with both translation-time value and type parameters, as well as implicit parameters and conditional type matching expressions.

As the partial evaluator processes the parse tree, a symbol table is built. Each identifier (i.e. variable, type, function, or label name) in the program will be entered into the table when it is encountered. An entry is removed when the scope in which it is defined ends. The symbol table was extended so that the following information may be stored for each function entry:

- A pointer to the root node of the function in the parse tree. If the function is polymorphic, it is eventually removed from the parse tree and replaced by its instantiations.
- The "class" rating of the function. There are three classes of user-defined functions: those with no translation-time parameters (i.e. monomorphic), those with one or more translation-time parameters in a standard parameter list, and those with implicit and explicit parameters.
- If the function is polymorphic, a linked list of instantiation prototypes. These prototypes are used as a basis for creating the monomorphic instantiations of the function.
- If the function name has been overloaded, a pointer to the next symbol table entry of a function with the same name. Function overloading has not been "officially" implemented in Safer_C; this is merely an ad-hoc implementation.

The specialization process works as follows:

1. When a function definition is encountered in the parse tree, it is entered into the symbol table. The class of the function is determined and stored in the table entry, along with a pointer to the root node and (if the function name has been overloaded) a pointer to the previous function entry with the same name.
2. When a function invocation is encountered, the partial evaluator will check the symbol table entry of the function to determine if it is polymorphic. If so, a type unification algorithm will determine the values of the translation-time parameters and ensure that the invocation is valid. If it is valid:
    - A "name mangling" mechanism is used to obtain the distinct name of an instantiation. This name is created by combining the original name with encoded type signatures of the parameters (both explicit and implicit).
    - The translator will search for an instantiation prototype with this name in the linked list attached to the table entry. If such a prototype is not found, it will be

created and added to the linked list. This prototype will contain the values of the translation-time parameters, as determined by the type unification mechanism.

- The function invocation will be replaced by a call to a function with that name. All explicit translation-time parameters will be removed from the actual parameter list.

If the invocation is not valid but the function name has been overloaded, this step will be repeated for the next function with the same name.

3. Every identifier in a program is only valid in a certain scope. When a scope ends, all of the identifiers that were declared in that scope are removed from the symbol table. The translator will examine each symbol table entry before it is removed to determine if it refers to a polymorphic function. If so:

   - A function specializer is called for each of the instantiation prototypes of the function. The specializer will substitute values and types for the translation-time parameters in the function, thereby creating a monomorphic version. At this point, standard partial evaluation techniques (such as constant folding) will be applied to the newly-created function body.
   - The polymorphic function definition will be removed from the parse tree and replaced by the instantiations.

To illustrate how this would work, consider the swap function (and its invocations) from section 4.4:

```
<<swap>> :: func ( impl T :: type
                   expl x :: ->T
                        y :: ->T
                 ) void
block
  temp :: T  !! used for swapping
  temp := x@
  x@ := y@
  y@ := temp
end
```

```
<<main>> :: func () int
block
  a, b :: int
  c, d :: float
  !! variable initializations omitted
  swap( &a, &b )
  swap( &c, &d )
end
```

The specializer will create two instantiations of swap; one for type `int` and one for type `float`. When the partial evaluation process finishes, the parse tree will contain a representation of the following program:

```
<<swap__int>> :: func ( x :: ->int
                        y :: ->int
                      ) void
block
  temp :: int
  temp := x@
  x@ := y@
  y@ := temp
end

<<swap__float>> :: func ( x :: ->float
                          y :: ->float
                        ) void
block
  temp :: float
  temp := x@
  x@ := y@
  y@ := temp
end

<<main>> :: func () int
block
  a, b :: int
  c, d :: float
  swap__int( &a, &b )
  swap__float( &c, &d )
end
```

The translator will then generate an equivalent program in ANSI C. It should be noted that the function names `swap__int` and `swap__float` differ from those created by the name mangling mechanism. The names were changed in this example for the sake of clarity.

5.2. Managing instantiations in multiple source files

When an entire program is contained in a single source file, managing the instantiations of polymorphic functions is a straightforward task. All of the invocation statements in the program will be in memory at the same time, along with the function body. The specializer will have access to both, and will be able to ensure that there are no missing or duplicate instantiations.

This task becomes much more complex when separate compilation is used. Ideally, it should be possible to declare a polymorphic function in one source file and call it from others, as can be done with monomorphic functions. Doing so in the context of the Safer_C language (and translation process) raises three major implementation issues:

1. In order to instantiate a version of the function, the translator will need its parse tree representation. How should the translator gain access to it?
2. Two or more source files may contain invocations that have the same translation-time parameters. This could cause the translator to repeatedly create the same instantiation of the function. How will the translator manage the "duplicate instantiation" problem?
3. If the source code of the function is modified, instantiations based on the old version will still exist in other files. How will the translator ensure that the final program does not contain "mixed" instantiations?

These issues are very similar to the template instantiation problems faced by C++ compiler designers. There are two basic approaches to these problems [FSF 95]:

1. In Borland C++, the programmer will place the entire template definition in a header file, which is loaded and parsed when a source file is compiled. The compiler will create all of the instantiations needed by that source file and place them in the object file. When all of the object files are linked together, the linkage editor will discover and remove any duplicate instantiations. This method has the following advantages:
   - It requires no modifications to the compiler, as all of the extra management work is done by the linkage editor.

- If a template definition is updated, the header file containing it will be modified as well. The project manager should be able to detect this and recompile all of the source files which include this header file.
- Duplicate instantiations will not appear in the final object file.

This method has two disadvantages:

- It is inefficient in terms of time. Many (duplicate) instantiations may be generated, and all but one will be discarded.
- It requires a special linkage editor which is capable of detecting and discarding duplicate instantiations.

2. The AT&T C++ translator uses a special "template repository" to store the locations of template definitions and their invocations. The information in the repository is updated whenever a source file is compiled. A separate instantiation step is performed just before the object files are linked; this step will instantiate all of the versions required in the program. This method has a number of advantages:

   - It is efficient in terms of time, as each instance of a template is only generated once.
   - It ensures that the current versions of the templates will be used for all instantiations.
   - It does not require a special linkage editor.
   - Duplicate instantiations will not appear in the final object file.

   This method has two main disadvantages:

   - The implementation is extremely complex.
   - The compiler will create one repository in each directory containing source files. For this reason, it is difficult to build multiple programs in one directory or one program over multiple directories.

The Safer_C designers have spent a good deal of time considering these issues. The following ideas were proposed for dealing with each one:

Issue 1: Accessing the parse tree representation of the function definition

- A new statement could be added to the language, which would allow the programmer to specify the name of the file that a function definition is located in. This statement would take a form similar to the following:

  ```
  function_name in "file_name"
  ```

  For example:

  ```
  <<swap>> in "a_file.sl"
  ```

  When the translator would encounter such a statement, it would load and parse the file indicated to obtain the function definition. This idea was deemed to be too inefficient and problematic (especially when combined with Issue 3) and was not considered further.
- The programmer could place the function definition inside a translation-time function, and then load the function into memory before the source file is parsed. By invoking the translation-time function at the start of the source file, the function definition would be "imported" into the program. This is the Safer_C equivalent to `#INCLUDE`ing a header file in C and C++. This is essentially the same solution used in the Borland C++ compiler.
- This issue could be avoided entirely by using a template repository, as was done by the AT&T designers. See the first idea for Issue 2.

Issue 2: Managing the "duplicate instantiation" problem

- Use a variation of the "template repository" idea. In the Safer_C version, the programmer would explicitly create and name the repository to be used for a specific project. The name and path of the repository would be provided to the compiler as a command-line option when it is run. Doing so would avoid the "one template per directory" problem of the AT&T implementation. For example:

  ```
  scl -REP="~/stuff/repository.rep" filename.sl
  ```

As with the AT&T version, all of the instantiations would be performed in a single step, after all of the source files have been translated to ANSI C. These instantiations could be performed by a program other than the main Safer_C translator. For example:

```
   instantiate ~/stuff/repository.rep
```

The Safer_C designers have also considered storing the parse tree representation of each function in the repository, instead of just its location. While this would increase the complexity and size of the repository, it may also improve the efficiency of the instantiation step as it would eliminate the need to load and parse multiple source files to obtain these parse trees.

- Expand the "name mangling" mechanism so that the name of the source file is included as part of the instantiation name. For example:

  ```
  swap__filename_int( a, b )
  swap__filename_float( c, d )
  ```

  This would not avoid the duplicate instantiation problem, but it would ensure that no two duplicates have the same name. As such, object files containing these instantiations could be linked without errors using a standard linkage editor.

- Allow the translator to create duplicate versions with the same names. When all of the source files have been translated to ANSI C, call a separate program to examine the resulting files and remove the bodies of any duplicated functions. The headers would remain, as C compilers generally use them to perform error checking during the compilation process.

Issue 3: Ensuring that all instantiations are based on the same function version

- If the function definition was contained in a translation-time function (stored in a separate file) and then imported, this issue would indeed be valid. Fortunately, most project managers (such as `make` in UNIX) are capable of determining if a source file has been modified. If so, they automatically recompile all other source files which depend upon it. This would ensure that every instantiation of a function is built from its most current version.

- Again, this issue would be avoided if a template repository were used.

Conclusion

While the Safer_C designers favor the template repository idea, it is far too complex to implement in the context of this thesis. As such, it has been decided that the following combination of ideas will be implemented:

- Polymorphic function definitions should be placed inside translation-time functions. The definitions will be imported into source files by invoking these functions.
- It will be up to the UNIX `make` utility to ensure that every source file that calls these translation-time functions will be retranslated if the file containing them is modified.
- The name mangling mechanism will be extended so that source file names will be included in the instantiation names.

While this combination of ideas is the most inefficient in terms of both time and program size, its implementation is trivial and would allow for separate compilation.

Chapter 6

Summary

In many imperative programming languages, the types of variables, parameters, constants, and functions must be specified explicitly by the programmer. This provides a means for type-checking code at compile-time, but can be quite restrictive. In the worst case, the programmer may be forced to implement different versions of a program construct for each type of argument to which it will be applied. A number of languages, such as Ada, C++, and ML, allow the definition of functions which can be applied to more than one type or to a set of types. These are known as polymorphic functions. Polymorphism is a central feature of object-oriented languages like Eiffel.

Partial evaluation is a program optimization process which occurs at compile-time. In general, partial evaluation techniques (such as constant folding, loop unrolling, and function specialization) exploit values known at compile-time with the goal of producing a faster and sometimes smaller program. These techniques are used extensively in the Safer_C programming language, developed at the University of Manitoba. In this language, variables, parameters, and functions can be assigned an evaluation time by the programmer. When one or more parameters of a function have an evaluation time of compile-time (also known as translation-time), the function may be specialized for their values.

Function specialization is one of the means through which Safer_C will support function polymorphism. This thesis described three new developments in this area: 1) Syntax for declaring implicit formal parameters, 2) Boolean conditions for type inference, and 3) Translation-time type manipulation functions.

An extension of function specialization has been developed which allows functions to be specialized for types, as well as for values. Translation-time type and value information may be provided explicitly by the programmer, or it may be described abstractly in an implicit formal parameter list. An implicit parameter passing mechanism will obtain the types and values described in such lists from the site of each function invocation.

Implicit parameters can lead to a greater degree of abstraction in programs. For example, generic sorting functions generally require three parameters: an array, the number of elements in the array, and the base type of the array. As the compiler can easily determine the latter two parameters at translation-time, they are in fact redundant. If the range and base type were declared to be implicit parameters, function invocations would only require one explicit parameter: the array itself.

The programmer may place constraints on the specialization process by defining a conditional type matching expression for any explicit parameter. Such constraints provide a means of validating type-specific aspects of each function invocation. For example, it is possible to specify that a parameter must be an array or a numeric type. Conditional type matching expressions also give the compiler a means to choose between different versions of overloaded functions.

Conditional type matching expressions and the implicit parameter passing mechanism were made possible through the creation of a set of translation-time type manipulation functions. These functions accept types as parameters and return a type or value as their result. Some of these functions (e.g. `WidestType`) were based upon internal functions used by compilers, while others (e.g. `IsTypeConsistent`) were invented especially for conditional type matching expressions. These functions may be used in a number of ways:

- Functions returning a type may be used to initialize an implicit type parameter or a type variable. Two such functions are `WidestType` and `BaseType`.

- Functions returning a boolean may be used as conditions in type matching expressions. Two such functions are `IsArrayType` and `IsTypeConsistent`.
- Functions returning values may be used to initialize an implicit value parameter or translation-time variable. Two such functions are `SizeOf` and `HighBound`.

Future directions

Safer_C/2 is intended to be a "state of the art" programming language that is at least as powerful as C++. When completed, it will contain a number of modern language features, such as operator overloading, parameterized types, and object-oriented support (allowing inclusion polymorphism through dynamic binding). The function specialization method (and related mechanisms) described in this thesis will influence and be influenced by the other language features. At this time, it is envisioned that the work described here will be extended in a number of directions:

- In the current design, implicit parameters are limited only to translation-time types and values. There are some situations in which it may be more appropriate to pass an implicit value parameter (such as the size of an array) at run-time, instead of at translation-time. Consider the following code, based on Example 3 in section 4.6:

  ```
  A :: [0..5] int
  B :: [0..6] int
  !! array initializations omitted
  sort(A)
  sort(B)
  ```

  At present, the translator will create two instantiations for `sort`; one each for range `0..5` and `0..6`. If the upper bound were passed as a run-time parameter, only one instantiation would be required.
- While implementing and testing this function specialization method, it became evident that more type manipulation functions would be useful. For example, simple types (such as `int`, `float`, and `char`) must be treated differently when being printed. For this reason, it is not currently possible to construct "purely" polymorphic

output functions. Additional type manipulation functions, such as `IsInt`, `IsFloat`, and `IsChar` would make it possible to define such output functions.

- At this time, a simple function overloading mechanism has been implemented. This mechanism will attempt to validate one version of an overloaded function at a time, until a perfect match is made. A better mechanism should be developed which will examine all function versions and decide between them on the basis of how "close" their type match is.
- At this time, work is underway on a new "universal" mechanism for expressing and manipulating arrays and structures. Should this type mechanism be included in Safer_C/2, new type manipulation functions should be developed to obtain information about it. Such functions would be similar to (and possibly replace) `IsArrayType`, `IsStructType`, `FieldType`, and `Field`.

Appendix A

Evolution of this work

The function specialization method presented in this paper has evolved a great deal since this research was initiated. This section of the thesis describes the two earlier versions of the design, and explains why they were discarded. For the sake of clarity, some background information from Chapter 4 has been repeated here.

Both of the previous versions have a number of things in common. In all cases, a polymorphic function may be defined by using type parameters in place of type names in the function header. A type parameter differs from a normal parameter in that it will be passed a type at compile-time, instead of a value at run-time.

Type parameters may be passed to a function explicitly or implicitly. In both cases, the compiler will use this type information to create a specialized (monomorphic) version of the polymorphic function. It is these specialized versions which will actually be called in the resulting program.

To indicate that a parameter should be passed at translation-time, the keyword `tran` is placed before its type specification in the formal parameter list. When an explicit type parameter is defined, the keyword `type` is used in place of the type specification; for such parameters, the keyword `tran` is optional.. The following function, therefore, should explicitly be passed a type parameter at translation-time:

```
<<square>> :: func ( T :: tran type; x :: T ) T
block
  return (x * x)
end
```

This function will calculate and return the square of a number of the type represented by `T`, a type parameter. Some calls to this function could be:

```
square( int, 2 )      !! returns 4
square( float, 1.5 ) !! returns 2.25
```

The syntax used to denote explicit type parameters did not change throughout the evolution of the function specialization method. What did change, though, were the methods used to declare and use implicit type parameters.

A.1. Version 1

It was initially proposed that implicit type parameters should be declared by prefacing them with the "?" operator at their first occurrence. This was essentially the method used for declaring query parameters in ForceTwo [Cor 88]. This method, by itself, seems adequate for declaring simple polymorphic functions like the following:

```
<<square>> :: func ( x :: ? T ) T
block
  return (x * x)
end
```

This version of `square` is much like the last one, except that the type parameter `T` is passed implicitly. At translation-time, the compiler would determine what `T` is from the actual parameter used and instantiate an appropriate version of this function. When doing so, it would have to perform type-checking to validate all of the overloaded operator and function calls (such as "*") which involve parameters of type `T`. The function invocations would take the following form:

```
square( 2 )     !! returns 4
square( 1.5 )   !! returns 2.25
```

If the "*" operator were user-overloaded to perform string concatenation, the following function call would also be valid:

```
square( "ab" ) !! returns "abab"
```

The following function, `max3`, is somewhat more complex:

```
<<max3>> :: func ( a, b, c :: ? T ) T
body
  return (a > b ? (a > c ? a : c) : (b > c ? b : c))
end
```

This function would accept three parameters of the type represented by `T`, a type parameter. This function would return the largest of these three parameters, a value which is also of type `T`. The conditional operator "`? :`" is used to determine this value, as opposed to using nested `if...else` statements. For example:

```
max3( 11, 99, 88 )      !! returns 99
max3( 3.14, 1.5, 5.0 ) !! returns 5.0
```

While this appears to work well, it could be improved. To see how, one must understand some of the background and objectives of the Safer_C project.

One of the goals of the Safer_C designers is to eliminate the need for a separate preprocessor meta-language. It has been shown that this can be accomplished by allowing statements and functions to be evaluated at translation-time instead of at run-time [Sal 95b]. The functionality of the preprocessor macro statement can also be duplicated by substituting the body of a function in-line wherever it is invoked [Bal 96]. To do so, the programmer must change the evaluation time of a function to translation-time. This is done by including the keyword `tran` in the function header. For example:

```
<<max3>> :: tran func ( a, b, c :: ? T ) T
body
  return (a > b ? (a > c ? a : c) : (b > c ? b : c))
end
```

The body of this function will be expanded in-line wherever it is invoked. The same result may be obtained by using the following preprocessor macro in C:

```
#define max3(a,b,c) ((a)>(b)?((a)>(c)?(a):(c))  \
                                ((b)>(c)?(b):(c)))
```

Unfortunately, this version of the design (as currently described) was somewhat less flexible than the "macro expansion" method. The macro method would permit the programmer to mix the types of the parameters, while the above Safer_C design would not. This flexibility was restored by defining a number of translation-time type manipulation functions. The first function defined was `widest`, which would accept a number of types as parameters and return the type with the most numerical precision. For example, if passed the types `int` and `long int`, `widest` would return `long int`.

Consider the following version of `max3`:

```
<<max3>> :: tran func (a::?T;b::?U;c::?V) widest(T,U,V)
body
  return (a > b ? (a > c ? a : c) : (b > c ? b : c))
end
```

In this version, each parameter may have a different type. The type of the function itself would be the widest of types `T`, `U`, and `V`.

This method was also extended to allow for the implicit or explicit passing of the size and base type of array parameters. For explicit passing, the size and base type would have been defined as being translation-time parameters in the formal parameter list. It was proposed that the following syntactic expression be used to describe the structure of the array parameter being passed:

```
array :: [0..size-1] base_type
```

This is a minor extension of the type expression used in Safer_C for array declarations. The following program, based upon the Bubble Sort example of section 2.2., illustrates how this method would have worked, and how functions using it would have been invoked:

```
<<sort>> :: func ( base :: tran type
                   size :: tran int
                   data :: [0..size-1] base ) void
block
  i, j :: int                       !! loop vars
  temp :: base                      !! used for swapping
  for (i := 0; i < size-1; i++)
    for (j := size-1; i < j; j--)
      if (data[j] < data[j-1])      !! swap
        temp := data[j]
        data[j] := data[j-1]
        data[j-1] := temp
      endif
    endfor
  endfor
end
```

```
<<main>> :: func () int
block
  a1 :: [0..99] double       !! declare first array
  a2 :: [0..999] ->char      !! declare second array
  !! array initializations omitted
  !! overloading of "<" for ->char omitted
  sort( double, 100, a1 )   !! sort first array
  sort( ->char, 1000, a2 ) !! sort second array
end
```

In this example, `sort` is a polymorphic function to which the size and base type of `data` are passed explicitly at translation-time. Both translation-time parameters are used in the function body; `size` as an integer constant, and `base` as the type of `temp`. The `main` function of this program contains two array declarations, `a1` and `a2`. These arrays have different sizes and base types. The compiler would instantiate two different versions of `sort` which conform to the type/size combinations shown in the actual parameter lists.

Passing the size and type implicitly is a somewhat more complex matter. Four different syntactic expressions were considered that built upon features examined previously. In the order that they were developed, they were:

array :: [0..?*size*-1] ?*base_type*

array :: [0..?(*size*::**type_expr**)-1] ?*base_type*::**type_expr**

array :: [?*size*::**type_expr**] ?*base_type*::**type_expr**

array :: [?*size*::**type_expr**] ?*base_type*

The first expression was an extension of the one presented for use with explicit parameters. In this expression, the "?" operator was used to indicate the presence of both implicit size and type parameters. This expression could have been problematic, as it would not have allowed the programmer to specify constraints for these parameters (as is done for explicit parameters).

To eliminate this problem, the second expression was devised. This expression contained sub-expressions which would describe the type and evaluation time of the two implicit parameters. Unfortunately, this expression was rather convoluted. To reduce its complexity, the designers considered replacing the "index range" section with an "array size" section, as shown in the third expression. This would have created no immediate programming problems, as arrays (as in C) always have an index range of type `int`, starting at 0. However, the designers of Safer_C have discussed the possibility of eliminating these limitations at some future date. If this were done, this expression may have had to be modified so that other index types and ranges could be passed implicitly.

The fourth expression was created by eliminating the type expression used to describe the base type. This was done to further reduce the complexity of the expression and to make it more compatible with the standard implicit parameter definition method (where such type expressions are not needed). In reality, the type expression would always have been "`tran type`" anyway. The following version of the `sort` function uses the fourth expression in its formal parameter list:

```
<<sort>> :: func (data :: [?size::tran int] ?base) void
block
  !! function body omitted
end

<<main>> :: func () int
block
  a1 :: [0..99] double    !! declare first array
  a2 :: [0..999] ->char   !! declare second array
  !! array initializations omitted
  !! overloading of '<' for ->char omitted
  sort( a1 )              !! sort first array
  sort( a2 )              !! sort second array
end
```

The compiler would determine what the `base` and `size` parameters are for each instantiation by examining the type structure of the actual parameters used in each function call. From the formal parameter list, the compiler would know that `size` is a translation-time `int` parameter. Given its placement in the expression, the compiler would consider `base` to be a translation-time `type` parameter.

Discussion

It was eventually decided that this version of the design should be discarded and another developed. This decision was reached for the following reasons:

- This version used the "?" operator to indicate the start of an implicit parameter declaration. This character is already used in Safer_C as a part of the conditional operator "? :". It can be argued that overloading an operator with two completely unrelated meanings (as was done with "*" and "&" in C) would have been a very poor design decision [Sal 95a]. As the "?" character was considered to be the most intuitive and obvious choice for this purpose, simply changing it did not appear to be a good solution.
- The Safer_C designers were not fully satisfied with any of the expressions that were devised to support implicit array size and type passing. The fourth expression that was described seemed adequate, but had two flaws:
    1. The first in this series of expressions was just a minor extension to the type expression used to declare arrays. In the third expression, the "index range" section was replaced with an "array size" section in order to reduce its complexity. Unfortunately, the similarity of the resulting expression to the array declaration expression could have actually caused problems, as it is possible that a programmer could confuse the formats of these two expressions. Simply restoring the "index range" section to avoid this problem would have resulted in cryptic formal parameter lists such as the following one:

       ```
       (data :: [0..?(size::tran int)-1] ?base)
       ```
    2. The complexity and length of such expressions would continue to increase if multi-dimensional arrays were involved. The following formal parameter list would have been required if a three-dimensional array were to be passed:

       ```
       (cube::[?x::tran int][?y::tran int][?z::tran int]?base)
       ```

It was decided that another mechanism should be developed which would allow the range or size of each dimension to be specified in an independent expression. The lessons learned during the development of this design have proven to be quite valuable. As it turns out, the single most important development was the idea for translation-time type manipulation functions. In the final design (described in Chapter 4), such functions play a critical role in the declaration and instantiation of polymorphic functions.

A.2. Version 2

After discarding the original design, it was decided that one should be produced which would not require special operators and compound expressions. In this version, every implicit parameter would appear as an independent expression in the formal parameter list. Such parameters would be listed alongside the explicit parameters in the list, as is done when using explicit translation-time parameters.

While compound expressions would not be required per-say, some expressions would certainly refer to other (implicit) parameters by name. For example, the type of an explicit parameter could be one of the implicit parameters. In the case of arrays, the upper bound could be an implicit parameter as well.

Generally, implicit parameters should be declared before they are actually referred to, in order to increase the readability of parameter lists. For a time, the Safer_C designers considered making this a legal requirement of such parameter lists. After some consideration, it was decided that doing so ran contrary to the Safer_C design philosophy.

This version of the design initially allowed for four different kinds of implicit parameters. After some consideration, it was decided that only three of them should be supported. The fourth is included here anyway, for the sake of completeness.

Implicit translation-time type or value parameters

These implicit parameters would have been used as "building blocks" in other expressions in the formal parameter list. The following syntax was proposed for both types and values:

```
identifier :: imp tran type
```

This was actually an extension of the syntax used to declare explicit translation-time parameters, as described previously. If the parameter is a type, then its *type* would simply be `type`. If it is a value, then its *type* could be a built-in type, a user-defined type, or even the name of a translation-time type parameter. For example:

```
A :: imp tran type   !! A is a type
B :: imp tran int    !! B is a value of type int
C :: imp tran A      !! C is a value of type A
```

As in Version 1, an expression was defined to describe the structure of array parameters. In this version, the same expression would have been used for both explicit and implicit type and range parameters. This expression is nearly identical to the one used by Safer_C for array declarations:

```
array :: [low..high] base_type
```

Instead of passing the size of the array, the programmer would pass the index range. At this time, Safer_C only supports index ranges of type `int` and a low bound of `0`. For this reason, the programmer will only have to define one translation-time `int` parameter, the high bound. If these range limitations were removed in some future version of Safer_C, the expression described here would not have had to be modified. The following formal parameter list illustrates how such expressions would have been used:

```
( T   :: imp tran type   !! base type
  H   :: imp tran int    !! high bound of range
  A   :: [0..H] T )      !! array
```

In this example, `T` and `H` are the type and high index of the array, `A`. While `T` and `H` are implicit parameters in this example, they could be passed as explicit parameters as well. To do so, one would omit the keyword `imp` from the first two parameter descriptions.

One of the advantages of this approach was that multi-dimensional array expressions would have been easy to write and understand. Consider the following example:

```
( base   :: imp tran type                  !! base type
  x,y,z  :: imp tran int                   !! high bounds
  cube   :: [0..x][0..y][0..z] base )      !! 3D array
```

This formal parameter list is quite intuitive and readable, especially when compared to its counterpart in section A.1.

Operator and function parameters

In Version 1 of the design, it was to have been the compiler's responsibility to validate every operator and function call made inside a polymorphic function. This would have become complicated if the parameters of such a call were values of some implicit type parameter. If so, for each instantiation (and actual type parameter), the compiler would have had to ensure that the operator or function is defined for that type in the surrounding scope.

In this version, the capabilities of Safer_C were extended to allow for the declaration of implicit and explicit operator and function parameters. Such declarations would have simplified type-checking and increased the flexibility of polymorphic function calls. Operators and functions were, respectively, the second and third kinds of implicit parameters supported by this design.

When developing the syntax for such declarations, the designers attempted to keep the overall style similar to that of translation-time type/value parameters. The syntax described here requires no special operators. The `imp` keyword is used to distinguish between an implicit and an explicit parameter.

The Safer_C designers had given some thought to the possibility of eventually extending this mechanism to allow for run-time operator and function parameter passing. The syntax described here was designed so that the programmer would be able to specify whether an operator or function is a translation-time (`tran`) or a run-time parameter.

This was done so that such an extension could be accomplished without changing the syntax of this mechanism (and thereby requiring that existing Safer_C code be updated).

Implicit and explicit operator/function parameters would have been used and verified in very different ways. For implicit parameters, the compiler would have searched (in the scope surrounding each invocation of the function) for an operator/function matching the pattern and name in the expression. If one were found, it would have been used in an instantiation of the function. For explicit parameters, the compiler would type-check the operator/function named in the actual parameter list against the pattern in the expression. If the operator/function named matched the pattern, it would have been substituted for the formal parameter in the function instantiation.

The syntactic expressions are listed below, along with some simple examples to illustrate their use. In all cases, `[imp]` was used to show that the keyword `imp` was optional. If it was present, the expression described an implicit parameter; if it was absent, an explicit parameter. The expression for a unary operator was as follows:

```
"operator" :: [imp] tran (type) result_type
```

For example:

```
"++" :: imp tran (T1) T1  !! implicit
"--" :: tran (T2) T2      !! explicit
```

The expression for a binary operator was as follows:

```
"operator" :: [imp] tran (type1, type1) result_type
```

For example:

```
"*" :: imp tran (T1, T1) T1   !! implicit
"<" :: tran (T2, T2) int      !! explicit
```

The expression for a function was as follows:

```
function :: [imp] tran (type1, type2, ...) result_type
```

For example:

```
max  :: imp tran (T1, T1) T1          !! implicit
work :: tran (int, char, real) void   !! explicit
```

To illustrate how implicit type, value, operator, and function parameters would have been used together, some of the examples from section A.1. have be re-written to conform to the design presented here. It should be noted that translation-time type manipulation functions were carried over from the initial design, and may still be used in the same manner. Comments are provided to clarify the less obvious points about these function definitions and calls.

```
!! "square" returns the square of x
!! - operator "*" must be defined for type T
<<square>> :: func ( T   :: imp tran type
                     x   :: T
                     "*" :: imp tran (T, T) T
                   ) T
block
  return (x * x)
end

!! "max3" returns the largest of three parameters
!! - the parameters may be of different types
!! - the type of the function will be the widest of
!!   types T, U, and V
!! - operator ">" must be defined for the widest of
!!   types T, U, and V
!! - in Safer_C, "~~" is a statement continuation mark
<<max3>> :: tran func ( T   :: imp tran type
                        U   :: imp tran type
                        V   :: imp tran type
                        a   :: T
                        b   :: U
                        c   :: V
                        ">" :: imp tran ( widest(T,U,V)
                                     ~~   widest(T,U,V)
                                     ~~ ) int
                      ) widest(T,U,V)
body
  return (a > b ? (a > c ? a : c) : (b > c ? b : c))
end
```

```
!! "sort" will perform a Bubble Sort on an array
!! - the array may be of any size and base type
!! - the high bound and base type are implicit parameters
!! - operator "<" must be defined for the base type
<<sort>> :: func ( base :: imp tran type
                   high :: imp tran int
                   data :: [0..high] base
                   "<"  :: imp tran (base, base) int
                 ) void
block
  i, j :: int                         !! loop vars
  temp :: base                        !! used for swapping
  for (i := 0; i < high; i++)
    for (j := high; i < j; j--)
      if (data[j] < data[j-1])        !! swap
        temp := data[j]
        data[j] := data[j-1]
        data[j-1] := temp
      endif
    endfor
  endfor
end

!! main program
<<main>> :: func () int
block
  !! overloading of "*" as string concat omitted
  square( 2 )                 !! returns 4
  square( 1.5 )               !! returns 2.25
  square( "ab" )              !! returns "abab"

  max3( 11, 99, 88 )          !! returns 99
  max3( 3.14, 1.5, 5.0 )      !! returns 5.0

  a1 :: [0..99] double        !! declare first array
  a2 :: [0..999] ->char       !! declare second array
  !! array initializations omitted
  !! overloading of "<" for ->char omitted
  sort( a1 )                  !! sort first array
  sort( a2 )                  !! sort second array
end
```

Implicit variable parameters

In many programs, global variables are accessed directly by subprograms. While this is generally considered to be poor programming practice, it can be extremely convenient when used judiciously. Implicit variable parameters were designed as an attempt to regulate the access to such data items.

Implicit variable parameters would have been passed automatically to functions when they were invoked. The parameter name would be the same as the actual name of the variable, as declared in the scope surrounding the function call. Such parameters would be declared using the following syntax:

```
name :: imp type
```

For example:

```
limit   :: imp int
highest :: imp T1
```

In short, an implicit variable parameter would have appeared in the formal parameter list but not in the actual function invocations. At translation-time, the compiler would have examined the scope of each function invocation to ensure that the parameters listed were indeed defined there. It was envisioned that such parameters would have two advantages:

1. They would provide a means to regulate global references. If a global variable not listed as an implicit variable parameter were to be accessed, the compiler would have generated a warning message.
2. By examining the header of an existing function, a programmer would have been able to determine the names and types of all of the global data items accessed by the function.

It was eventually decided that implicit variable parameters should be excluded from this version of the design. The main reason was that these parameters could have altered the style of what is supposed to be an unrestrictive language. The secondary reason is that both advantages listed above could have been realized by employing good programming style (for example, by placing a list of "global variables required" in the comments at the function declaration).

Discussion

Version 2 of the design was not discarded because there was something wrong with it, per-say. Generally speaking, the parameter expressions described here were reasonably expressive and concise. Formal parameter lists using such expressions would have been more readable and intuitive (though somewhat longer) than equivalent lists created using Version 1 of the design.

This design was discarded because the developers of Safer_C realized that there were other, more natural ways to deal with the issues discussed here. The final design differs from this one in two major ways:

- In Version 2, implicit and explicit parameters were defined side by side in the parameter list, with the keyword `imp` used to distinguish between them. When declaring polymorphic functions, all implicit and all explicit type/value parameters tended to be grouped together naturally. To simplify matters in the final design, it was decided that parameter lists should be partitioned into two sections: implicit and explicit. This is done through the use of the `impl` and `expl` keywords.
- The final design makes extensive use of translation-time type manipulation functions, while this version did not. In the final design, such functions are used to enable conditional type matching during the instantiation process. Conditional type matching expressions both simplify type-checking and allow the programmer to overload polymorphic function names. Type manipulation functions are used to determine the structure of array parameters; they also eliminate the need for operator/function parameters. For this reason, the final design does not need the special parameter expressions found in versions 1 and 2.

Appendix B

Sample programs

This appendix contains a number of programs from the test suites. Each program is accompanied by the ANSI C code produced by the translator, as well as the output generated by compiling and executing it. All of the C programs were compiled by acc running under SunOS Release 4.1.3.

312 - Safer_C code:

```
Safer_C version 1.5

!! 312.sl
!! ------
!! Test of ASSIGN clause with WidestType TMF

<<max>> :: func ( impl A :: type
                       B :: type
                       W :: type := WidestType(A, B)
                  expl x :: A
                       y :: B
                     ) W
block
  return (x > y ? x : y)
end

<<main>> :: func () void
block
  a :: short int   := 1
  b :: short int   := 2
  c :: int         := 3
  d :: int         := 4
  e :: long int    := 5
  f :: long int    := 6
  g :: float       := 7.0
  h :: float       := 7.5
  i :: double      := 8.0
  j :: double      := 8.5
  k :: long double := 9.0
  l :: long double := 9.5
  printf("Max of...\n")                    !! RESULT:
  !! same types
  printf(" a, b = %d\n",  max(a, b)) !! short int
  printf(" c, d = %d\n",  max(c, d)) !! int
  printf(" e, f = %ld\n", max(e, f)) !! long int
  printf(" g, h = %f\n",  max(g, h)) !! float
  printf(" i, j = %f\n",  max(i, j)) !! double
  printf(" k, l = %e\n",  max(k, l)) !! long double
  !! mixed types
  printf(" a, c = %d\n",  max(a, c)) !! int
  printf(" a, e = %ld\n", max(a, e)) !! long int
  printf(" a, g = %f\n",  max(a, g)) !! float
  printf(" a, i = %f\n",  max(a, i)) !! double
  printf(" a, k = %e\n",  max(a, k)) !! long double
end
```

312 - C code produced:

```
/* Target language:  ANSI C                    */
/* Source language:  Safer_C version 1.5       */
/* Source file:      312.sl                    */
/* Translation date: Wed Jan 22 00:12:34 1997 */

long double
max__Fsrrsr_r(short int x, long double y)
  {
  return x>y ? x : y;
  }
double
max__Fsddsd_d(short int x, double y)
  {
  return x>y ? x : y;
  }
float
max__Fsffsf_f(short int x, float y)
  {
  return x>y ? x : y;
  }
long int
max__Fsllsl_l(short int x, long int y)
  {
  return x>y ? x : y;
  }
int
max__Fsiisi_i(short int x, int y)
  {
  return x>y ? x : y;
  }
```

```
long double
max__Frrrrr_r(long double x, long double y)
  {
  return x>y ? x : y;
  }
double
max__Fddddd_d(double x, double y)
  {
  return x>y ? x : y;
  }
float
max__Fffff_f(float x, float y)
  {
  return x>y ? x : y;
  }
long int
max__Fl1111_l(long int x, long int y)
  {
  return x>y ? x : y;
  }
int
max__Fiiii_i(int x, int y)
  {
  return x>y ? x : y;
  }
short int
max__Fsssss_s(short int x, short int y)
  {
  return x>y ? x : y;
  }
void
main()
  {
  short int a = 1;
  short int b = 2;
  int c = 3;
  int d = 4;
  long int e = 5;
  long int f = 6;
  float g = 7.0;
  float h = 7.5;
  double i = 8.0;
  double j = 8.5;
  long double k = 9.0;
  long double l = 9.5;
  printf("Max of...\n");
  printf(" a, b = %d\n", max__Fsssss_s(a, b));
  printf(" c, d = %d\n", max__Fiiiii_i(c, d));
  printf(" e, f = %ld\n", max__Flllll_l(e, f));
  printf(" g, h = %f\n", max__Ffffff_f(g, h));
  printf(" i, j = %f\n", max__Fdddddd_d(i, j));
  printf(" k, l = %e\n", max__Frrrrr_r(k, l));
  printf(" a, c = %d\n", max__Fsiisi_i(a, c));
  printf(" a, e = %ld\n", max__Fsllsl_l(a, e));
  printf(" a, g = %f\n", max__Fsffsf_f(a, g));
  printf(" a, i = %f\n", max__Fsddsd_d(a, i));
  printf(" a, k = %e\n", max__Fsrrsr_r(a, k));
  }
```

312 - Output:

```
Max of...
 a, b = 2
 c, d = 4
 e, f = 6
 g, h = 7.500000
 i, j = 8.500000
 k, l = -3.103615e+231   (error)
 a, c = 3
 a, e = 5
 a, g = 7.000000
 a, i = 8.000000
 a, k = -3.103615e+231   (error)
```

315 - Safer_C code:

```
Safer_C version 1.5

!! 315.s1
!! ------
!! Simple test of WHERE clause with IsArrayType and
overloading.

<<greatest>> :: func ( impl T    :: type
                       expl data :: T where not
                                    IsArrayType(T)
                     ) T
block
  return data
end
```

```
<<greatest>> :: func ( impl
                         T    :: type
                         B    :: type     := BaseType(T)
                         I    :: type     := IndexType(T)
                         hi   :: tran int := HighBound(I)
                         lo   :: tran int := LowBound(I)
                       expl
                         data :: T where IsArrayType(T)
                     ) B
block
  i   :: I                !! var of index type
  max :: B := data[0]     !! var of base type (with init)

  for (i := lo; i <= hi; i++)
    if (data[i] > max)
      max := data[i]
    endif
  endfor

  return max
end

<<main>> :: func () void
block
  a1 :: [0..4] int   := {3, 9, 2, 6, 4}
  a2 :: [0..2] float := {4.6, 7.7, 2.1}
  b  :: int          := 99;
  c  :: float        := 3.14

  printf("Array 1: %d %d %d %d %d\n", a1[0], a1[1],
         a1[2], a1[3], a1[4])
  printf("High:    %d\n", greatest(a1))

  printf("Array 2: %f %f %f\n", a2[0], a2[1], a2[2])
  printf("High:    %f\n", greatest(a2))

  printf("Integer: %d\n", b)
  printf("High:    %d\n", greatest(b))

  printf("Float:   %f\n", c)
  printf("High:    %f\n", greatest(c))
end
```

315 - C code produced:

```
/* Target language:  ANSI C                   */
/* Source language:  Safer_C version 1.5      */
/* Source file:      315.sl                   */
/* Translation date: Wed Jan 22 00:12:37 1997 */
```

```
float
greatest__Fff_f(float data)
  {
  return data;
  }
int
greatest__Fii_i(int data)
  {
  return data;
  }
float
greatest__FA2_ffI0_2_X2XX0XA2_f_f(float data[3])
  {
  int i;
  float max = data[0];
  for (i = 0; i<=2; i++ )
    {
    if (data[i]>max)
      {
      max = data[i];
      }
    }
  return max;
  }
int
greatest__FA4_iiI0_4_X4XX0XA4_i_i(int data[5])
  {
  int i;
  int max = data[0];
  for (i = 0; i<=4; i++ )
    {
    if (data[i]>max)
      {
      max = data[i];
      }
    }
  return max;
  }
void
main()
  {
  int a1[5] = {3, 9, 2, 6, 4};
  float a2[3] = {4.6, 7.7, 2.1};
  int b = 99;
  float c = 3.14;
  printf("Array 1: %d %d %d %d %d\n", a1[0], a1[1],
         a1[2], a1[3], a1[4]);
  printf("High:    %d\n",
         greatest__FA4_iiI0_4_X4XX0XA4_i_i(a1));
  printf("Array 2: %f %f %f\n", a2[0], a2[1], a2[2]);
  printf("High:    %f\n",
         greatest__FA2_ffI0_2_X2XX0XA2_f_f(a2));
  printf("Integer: %d\n", b);
  printf("High:    %d\n", greatest__Fii_i(b));
  printf("Float:   %f\n", c);
  printf("High:    %f\n", greatest__Fff_f(c));
  }
```

315 - Output:

```
Array 1: 3 9 2 6 4
High:    9
Array 2: 4.600000 7.700000 2.100000
High:    7.700000
Integer: 99
High:    99
Float:   3.140000
High:    3.140000
```

318 - Safer_C code:

```
Safer_C version 1.5

!! 318.s1
!! ------
!! Simple test of WHERE clause with IsTypeConsistent.
!!
!! This is Example 1 from section 4.6. of the thesis.

<<square>> :: func (
                impl T :: type
                expl x :: T where IsTypeConsistent(x * x)
                   ) T
block
  return (x * x)
end

<<main>> :: func () void
block
  a :: int    := 2
  b :: float  := 1.5
  c :: ->char := "ab"
```

```
    printf("%-4d squared = %-4d\n",    a, square(a)) !!int
    printf("%4.2f squared = %4.2f\n", b, square(b)) !!float

  !! This invocation is not type consistent:
  !! printf("%s squared = %s\n", c, square(c))

  end
```

318 - C code produced:

```
  /* Target language:  ANSI C                    */
  /* Source language:  Safer_C version 1.5       */
  /* Source file:      318.sl                    */
  /* Translation date: Wed Jan 22 00:12:40 1997 */

  float
  square__Fff_f(float x)
    {
    return x*x;
    }
  int
  square__Fii_i(int x)
    {
    return x*x;
    }
  void
  main()
    {
    int a = 2;
    float b = 1.5;
    char (*c) = "ab";
    printf("%-4d squared = %-4d\n", a, square__Fii_i(a));
    printf("%4.2f squared = %4.2f\n", b, square__Fff_f(b));
    }
```

318 - Output:

```
  2    squared = 4
  1.50 squared = 2.25
```

320 - Safer_C code:

```
Safer_C version 1.5

!! 320.sl
!! ------
!! Tests ASSIGN clause with WidestType, WHERE clause with
!! IsNumericType.
!!
!! This is Example 2 from section 4.6. of the thesis.

<<max>> :: func ( impl T1 :: type
                       T2 :: type
                       W  :: type := WidestType(T1, T2)
                  expl x  :: T1 where IsNumericType(T1)
                       y  :: T2 where IsNumericType(T2)
                     ) W
block
  return (x > y ? x : y)
end

<<main>> :: func () void
block
  a :: int      := 11
  b :: int      := 99
  c :: float    := 3.14
  d :: float    := 5.0
  e :: long int := 40000
  f :: ->char   := "abc"

  printf("Max of...\n")
  !! int
  printf(" %-5d %-5d = %-5d\n",     a, b, max(a, b))
  !! float
  printf(" %4.3f %4.3f = %4.3f\n", c, d, max(c, d))
  !! long int
  printf(" %-5d %-5d = %-5d\n",     a, e, max(a, e))

!! This invocation contains a non-numeric param:
!! printf(" %d %s = %d\n", a, f, max(a, f))

end
```

320 - C code produced:

```
/* Target language:  ANSI C                   */
/* Source language:  Safer_C version 1.5      */
/* Source file:      320.sl                   */
/* Translation date: Wed Jan 22 00:12:42 1997 */
```

```
long int
max__Fillil_l(int x, long int y)
  {
  return x>y ? x : y;
  }
float
max__Ffffff_f(float x, float y)
  {
  return x>y ? x : y;
  }
int
max__Fiiiii_i(int x, int y)
  {
  return x>y ? x : y;
  }
void
main()
  {
  int a = 11;
  int b = 99;
  float c = 3.14;
  float d = 5.0;
  long int e = 40000;
  char (*f) = "abc";
  printf("Max of...\n");
  printf(" %-5d %-5d = %-5d\n", a, b,
         max__Fiiiii_i(a, b));
  printf(" %4.3f %4.3f = %4.3f\n", c, d,
         max__Ffffff_f(c, d));
  printf(" %-5d %-5d = %-5d\n", a, e,
         max__Fillil_l(a, e));
  }
```

320 - Output:

```
Max of...
 11    99    = 99
 3.140 5.000 = 5.000
 11    40000 = 40000
```

321 - Safer_C code:

```
Safer_C version 1.5

!! 321.s1
!! ------
!! Tests ASSIGN clause with BaseType, IndexType,
!! Lowbound, and HighBound, WHERE clause with IsArrayType
!! and IsTypeConsistent
!!
!! This is Example 3 from section 4.6. of the thesis.

<<sort>> :: func (
       impl T    :: type
            I    :: type     := IndexType(T)
            lo   :: tran int := LowBound(I)
            hi   :: tran int := HighBound(I)
            B    :: type     := BaseType(T)
       expl data :: T where IsArrayType(T) and
              ~~    IsTypeConsistent(data[lo] < data[hi])
                 ) void
block
  i, j :: I                         !! loop vars
  temp :: B                         !! used for swapping

  for (i := lo; i < hi; i++)
    for (j := hi; i < j; j--)
      if (data[j] < data[j-1])      !! swap
        temp := data[j]
        data[j] := data[j-1]
        data[j-1] := temp
      endif
    endfor
  endfor
end

<<main>> :: func () void
block
  a1 :: [0..4] int   := {3, 9, 2, 6, 4}
  a2 :: [0..2] float := {4.6, 7.7, 2.1}

  printf("Before: %d %d %d %d %d\n", a1[0], a1[1], a1[2],
         a1[3], a1[4])
  sort(a1)
  printf("After:  %d %d %d %d %d\n", a1[0], a1[1], a1[2],
         a1[3], a1[4])
  printf("Before: %3.1f %3.1f %3.1f\n", a2[0], a2[1],
         a2[2])
  sort(a2)
  printf("After:  %3.1f %3.1f %3.1f\n", a2[0], a2[1],
         a2[2])
end
```

321 - C code produced:

```
/* Target language:  ANSI C                       */
/* Source language:  Safer_C version 1.5          */
/* Source file:      321.s1                       */
/* Translation date: Wed Jan 22 00:12:44 1997 */

void
sort__FA2_fI0_2_X0XX2XfA2_f_v(float data[3])
  {
  int i,j;
  float temp;
  for (i = 0; i<2; i++ )
    {
    for (j = 2; i<j; j-- )
      {
      if (data[j]<data[j-1])
        {
        temp = data[j];
        data[j] = data[j-1];
        data[j-1] = temp;
        }
      }
    }
  }
void
sort__FA4_iI0_4_X0XX4XiA4_i_v(int data[5])
  {
  int i,j;
  int temp;
  for (i = 0; i<4; i++ )
    {
    for (j = 4; i<j; j-- )
      {
      if (data[j]<data[j-1])
        {
        temp = data[j];
        data[j] = data[j-1];
        data[j-1] = temp;
        }
      }
    }
  }
```

```
void
main()
  {
  int a1[5] = {3, 9, 2, 6, 4};
  float a2[3] = {4.6, 7.7, 2.1};
  printf("Before: %d %d %d %d %d\n", a1[0], a1[1], a1[2],
         a1[3], a1[4]);
  sort__FA4_iI0_4_X0XX4XiA4_i_v(a1);
  printf("After:  %d %d %d %d %d\n", a1[0], a1[1], a1[2],
         a1[3], a1[4]);
  printf("Before: %3.1f %3.1f %3.1f\n", a2[0], a2[1],
         a2[2]);
  sort__FA2_fI0_2_X0XX2XfA2_f_v(a2);
  printf("After:  %3.1f %3.1f %3.1f\n", a2[0], a2[1],
         a2[2]);
  }
```

321 - Output:

```
Before: 3 9 2 6 4
After:  2 3 4 6 9
Before: 4.6 7.7 2.1
After:  2.1 4.6 7.7
```

322 - Safer_C code:

```
Safer_C version 1.5

!! 322.s1
!! ------
!! Tests ASSIGN clause with BaseType, IndexType,
!! Lowbound, and HighBound, WHERE clause with IsArrayType
!! and IsTypeConsistent
!!
!! This is Example 5 from section 4.6. of the thesis.
!! add two matrices; store result in the third
<<add_matrices>> :: func (
     impl T1    :: type
          lo1   :: tran int := LowBound(IndexType(T1))
          hi1   :: tran int := HighBound(IndexType(T1))
          T2    :: type     := BaseType(T1)
          lo2   :: tran int := LowBound(IndexType(T2))
          hi2   :: tran int := HighBound(IndexType(T2))
     expl A, B, C :: T1 where
                  ~~ IsArrayType(T1) and
                  ~~ IsArrayType(T2) and
                  ~~ IsTypeConsistent(A[lo1][lo2] +
                                  ~~  B[lo1][lo2])
         ) void
block
  x, y :: int  !! loop vars

  for (x := lo1; x <= hi1; x++)
    for (y := lo2; y <= hi2; y++)
      C[x][y] := A[x][y] + B[x][y]
    endfor
  endfor
end

!! print an integer matrix
<<print_int>> :: func (
     impl T1    :: type
          lo1   :: tran int := LowBound(IndexType(T1))
          hi1   :: tran int := HighBound(IndexType(T1))
          T2    :: type     := BaseType(T1)
          lo2   :: tran int := LowBound(IndexType(T2))
          hi2   :: tran int := HighBound(IndexType(T2))
     expl M     :: T1 where
                  ~~ IsArrayType(T1) and
                  ~~ IsArrayType(T2)
         ) void
block
  x, y :: int  !! loop vars

  for (x := lo1; x <= hi1; x++)
    for (y := lo2; y <= hi2; y++)
      printf("%4d", M[x][y])
    endfor
    printf("\n")
  endfor
  printf("\n")
end
```

```
!! print a float matrix
<<print_float>> :: func (
     impl T1   :: type
          lo1  :: tran int := LowBound(IndexType(T1))
          hi1  :: tran int := HighBound(IndexType(T1))
          T2   :: type     := BaseType(T1)
          lo2  :: tran int := LowBound(IndexType(T2))
          hi2  :: tran int := HighBound(IndexType(T2))
     expl M    :: T1 where
                  ~~ IsArrayType(T1) and
                  ~~ IsArrayType(T2)
         ) void
block
  x, y :: int  !! loop vars

  for (x := lo1; x <= hi1; x++)
    for (y := lo2; y <= hi2; y++)
      printf("%5.1f", M[x][y])
    endfor
    printf("\n")
  endfor
  printf("\n")
end

<<main>> :: func () void
block
  x, y :: int
  m1   :: type := [0..2][0..3] int
  m2   :: type := [0..1][0..2] float

  a :: m1 := { {1,  2,  3,  4},
          ~~   {5,  6,  7,  8},
          ~~   {9, 10, 11, 12} }
  b :: m1 := { {10,  20,  30,  40},
          ~~   {50,  60,  70,  80},
          ~~   {90, 100, 110, 120} }
  c :: m1
  d :: m2 := { {1.5, 2.5, 3.5},
          ~~   {4.5, 5.5, 6.5} }
  e :: m2 := { {10.5, 11.5, 12.5},
          ~~   {13.5, 14.5, 15.5} }
  f :: m2

  add_matrices(a, b, c)
  printf("Int matrices:\n")
  print_int(a)
  print_int(b)
  printf("Sum:\n")
  print_int(c)
```

```
    add_matrices(d, e, f)
    printf("Float matrices:\n")
    print_float(d)
    print_float(e)
    printf("Sum:\n")
    print_float(f)
  end
```

322 - C code produced:

```
  /* Target language:   ANSI C                    */
  /* Source language:   Safer_C version 1.5       */
  /* Source file:       322.sl                    */
  /* Translation date: Wed Jan 22 00:12:49 1997 */

  void
  add_matrices__FA1_A2_fX0XX1XA2_fX0XX2XA1_A2_fA1_A2_fA1_A2
  _f_v(float A[2][3], float B[2][3], float C[2][3])
    {
    int x,y;
    for (x = 0; x<=1; x++ )
      {
      for (y = 0; y<=2; y++ )
        {
        C[x][y] = A[x][y]+B[x][y];
        }
      }
    }
  void
  add_matrices__FA2_A3_iX0XX2XA3_iX0XX3XA2_A3_iA2_A3_iA2_A3
  _i_v(int A[3][4], int B[3][4], int C[3][4])
    {
    int x,y;
    for (x = 0; x<=2; x++ )
      {
      for (y = 0; y<=3; y++ )
        {
        C[x][y] = A[x][y]+B[x][y];
        }
      }
    }
```

```
void
print_int__FA2_A3_iX0XX2XA3_iX0XX3XA2_A3_i_v(int M[3][4])
  {
  int x,y;
  for (x = 0; x<=2; x++ )
    {
    for (y = 0; y<=3; y++ )
      {
      printf("%4d", M[x][y]);
      }
    printf("\n");
    }
  printf("\n");
  }
void
print_float__FA1_A2_fX0XX1XA2_fX0XX2XA1_A2_f_v(float
M[2][3])
  {
  int x,y;
  for (x = 0; x<=1; x++ )
    {
    for (y = 0; y<=2; y++ )
      {
      printf("%5.1f", M[x][y]);
      }
    printf("\n");
    }
  printf("\n");
  }
void
main()
  {
  int x,y;
  typedef int m1[3][4];
  typedef float m2[2][3];
  int a[3][4] = {{1, 2, 3, 4},
                 {5, 6, 7, 8},
                 {9, 10, 11, 12}};
  int b[3][4] = {{10, 20, 30, 40},
                 {50, 60, 70, 80},
                 {90, 100, 110, 120}};
  int c[3][4];
  float d[2][3] = {{1.5, 2.5, 3.5},
                   {4.5, 5.5, 6.5}};
  float e[2][3] = {{10.5, 11.5, 12.5},
                   {13.5, 14.5, 15.5}};
  float f[2][3];
add_matrices__FA2_A3_iX0XX2XA3_iX0XX3XA2_A3_iA2_A3_iA2_
  A3_i_v(a, b, c);
printf("Int matrices:\n");
print_int__FA2_A3_iX0XX2XA3_iX0XX3XA2_A3_i_v(a);
print_int__FA2_A3_iX0XX2XA3_iX0XX3XA2_A3_i_v(b);
printf("Sum:\n");
print_int__FA2_A3_iX0XX2XA3_iX0XX3XA2_A3_i_v(c);
add_matrices__FA1_A2_fX0XX1XA2_fX0XX2XA1_A2_fA1_A2_fA1_
  A2_f_v(d, e, f);
printf("Float matrices:\n");
print_float__FA1_A2_fX0XX1XA2_fX0XX2XA1_A2_f_v(d);
print_float__FA1_A2_fX0XX1XA2_fX0XX2XA1_A2_f_v(e);
printf("Sum:\n");
print_float__FA1_A2_fX0XX1XA2_fX0XX2XA1_A2_f_v(f);
}
```

322 - Output:

```
Int matrices:
   1   2   3   4
   5   6   7   8
   9  10  11  12

  10  20  30  40
  50  60  70  80
  90 100 110 120

Sum:
  11  22  33  44
  55  66  77  88
  99 110 121 132

Float matrices:
  1.5  2.5  3.5
  4.5  5.5  6.5

 10.5 11.5 12.5
 13.5 14.5 15.5

Sum:
 12.0 14.0 16.0
 18.0 20.0 22.0
```

References

[Aho 86] Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Massachusetts, 1986.

[And 92] Andersen, L.O., *Partial Evaluation of C and Automatic Compiler Generation*, Proceedings of the International Conference on Compiler Construction CC '92, in LNCS, Vol. 641, 251-257.

[Bal 96] Baluta, B. and Dueck, J., *Partial Evaluation of Functions in Safer_C*, Graduate course project report for 74.716 (Advanced Programming Language Design, Translation and Implementation), University of Manitoba, 1996.

[Bar 89] Barnes, J.G.P., *Programming In Ada (Third Edition)*, Addison-Wesley, Reading, Massachusetts, 1989.

[Bau 95] Baumgartner, G. and Russo, V.F., *Signatures: A Language Extension for Improving Type Abstraction and Subtype Polymorphism in C++*, Software - Practice and Experience, Vol. 25 No. 8, 863-889.

[Car 85] Cardelli, L. and Wegner, P., *On Understanding Types, Data Abstraction, and Polymorphism*, ACM Computing Surveys, Vol. 17 No. 4, 471-522.

[Cha 89] Chambers, C. and Ungar, D., *Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language*, Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, 146-160.

[Con 93] Consel, C. and Danvy, O., *Tutorial Notes on Partial Evaluation*, Conference Record of POPL '93: 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 493-501.

[Cor 85] Cormack, G.V., *Zephyr*, Unpublished.

[Cor 87] Cormack, G.V. and Wright, A.K., *Polymorphism in the Compiled Language ForceOne*, Proceedings of the Twentieth Annual Hawaii Conference on System Sciences, 284-292.

[Cor 88] Cormack, G.V. and Wright, A.K., *Practical Parametric Polymorphism*, Unpublished.

[Cor 90] Cormack, G.V. and Wright, A.K., *Type-dependent Parameter Inference*, Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, 127-136.

[Dan 96] Danvy, O., *Type-Directed Partial Evaluation*, Conference Record of POPL '96: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 242-257.

[Dit 94] Ditchfield, G.J., *Contextual Polymorphism*, Ph.D. Thesis, University of Waterloo, 1994.

[DoD 80] *Reference Manual for the Ada Programming Language*, United States Department of Defense, 1980.

[Dub 95] Dubois, C., Rouaix, F., and Weis, P., *Extensional Polymorphism*, Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 118-129.

[FSF 95] *Using and Porting GNU CC (for GCC version 2.7.2)*, Free Software Foundation, Inc., Boston, Mass., 1995.

[Ghe 87] Ghezzi, C. and Jazayeri, M., *Programming Language Concepts*, John Wiley & Sons, New York, 1987.

[Har 95] Harper, R. and Morrisett, G., *Compiling Polymorphism Using Intensional Type Analysis*, Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 130-141.

[Int 94] *Changes from Ada 83 to Ada 9X (Version 5.0)*, Intermetrics, Inc., Cambridge, Mass., 1994.

[Jon 93] Jones, N.D., Gomard, C.K., and Sestoft, P., *Partial Evaluation and Automatic Program Generation*, Prentice Hall, New York, 1993.

[Jon 96] Jones, N.D., *An Introduction to Partial Evaluation*, ACM Computing Surveys, Vol. 28 No. 3, 480-503.

[Kle 95] Kleinrubatscher, P., Kriegshaber, A., Zochling, R., and Gluck, R., *Fortran Program Specialization*, ACM SIGPLAN Notices, Vol. 30 No. 4, 61-70.

[Koe 89] Koenig, Andrew., *C Traps and Pitfalls*, Addison-Wesley, Reading, Massachusetts, 1989.

[Lau 94] Laufer, K. and Odersky, M., *Polymorphic Type Inference and Abstract Data Types*, ACM Transactions on Programming Languages and Systems, Vol. 16 No. 5, 1411-1430.

[Ler 91] Leroy, X. and Weis, P., *Polymorphic Type Inference and Assignment*, Conference Record of POPL '91: 18th ACM Symposium on Principles of Programming Languages, 291-302.

[Mey 88] Meyer, Bertrand., *Object-Oriented Software Construction*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[Mey 91] Meyer, U., *Techniques for Partial Evaluation of Imperative Languages*, SIGPLAN Notices, Vol. 26 No. 9, 94-105.

[Mey 92] Meyer, Bertrand., *Eiffel: The Language*, Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[Mil 78] Milner, Robin., *A Theory of Type Polymorphism in Programming*, Journal of Computer and System Sciences, 17, 348-375.

[Mor 91] Morrison, R., Dearle, A., Connor, R.C.H., and Brown, A.L., *An Ad Hoc Approach to the Implementation of Polymorphism*, ACM Transactions on Programming Languages and Systems, Vol. 13 No. 3, 342-371.

[Oho 95] Ohori, A., *A Polymorphic Record Calculus and Its Compilation*, ACM Transactions on Programming Languages and Systems, Vol. 13 No. 6, 844-895.

[Pau 91] Paulson, L.C., *ML for the Working Programmer*, Cambridge University Press, Cambridge, 1991.

[Poh 94] Pohl, Ira., *C++ for C Programmers*, Benjamin/Cummings, Redwood City, California, 1994.

[Sal 92] Salomon, D.J., *Four Dimensions of Programming-Language Independence*, SIGPLAN Notices, Vol. 27 No. 3, 35-53.

[Sal 95a] Salomon, D.J., *Safer_C: Syntactically Improving the C Language for Error Resistance*, Technical Report 95/07, Department of Computer Science, University of Manitoba.

[Sal 95b] Salomon, D.J., *Using Partial Evaluation to Replace the C Preprocessor*, Unpublished.

[Sal 96] Salomon, D.J., *Using Partial Evaluation in Support of Portability, Reusability, and Maintainability*, Proceedings of the International Conference on Compiler Construction CC '96, in LNCS, Vol. 1060, 208-222.

[Smi 96a] Smith, G. and Volpano, D., *Polymorphic Typing of Variables and References*, ACM Transactions on Programming Languages and Systems, Vol. 18 No. 3, 254-267.

[Smi 96b] Smith, G. and Volpano, D., *Towards an ML-Style Polymorphic Type System for C*, Proceedings of the 6th European Symposium on Programming, 341-355.

[Str 91] Stroustrup, B., *The C++ Programming Language (Second Edition)*, Addison-Wesley, Reading, Massachusetts, 1991.

[Tof 90] Tofte, Mads., *Type Inference for Polymorphic References*, Information and Computation, 89:1-34, 1990.

[Wie 95] Wiener, Richard., *Software Development Using Eiffel*, Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[Wri 95] Wright, A.K., *Simple Imperative Polymorphism*, LISP and Symbolic Computation, Vol. 8 No. 4, 343-355.