Implementing Overloading and Polymorphism in Cforall

by

Richard C. Bilson

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada 2003

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The programming language Cforall extends the C language with, among other things, overloading, parametric polymorphism, and functions that can return multiple values from a single call. This thesis presents an outline of the first implementation of the core Cforall language. An effective implementation of Cforall requires complete support for new language constructs while preserving the behaviour and efficiency of existing C programs. Analyzing the meaning of Cforall programs requires significantly more sophisticated techniques than are necessary for C programs; existing techniques for the analysis of overloading and polymorphism are adapted and extended to apply to Cforall. Three strategies for generating code for polymorphic programs are compared, using plain C as an intermediate representation. Finally, a realistic Cforall program is presented and characteristics of the generated C code are examined.

Acknowledgements

I would like to thank my readers, Professors Gordon Cormack and Andrew Malton, for their time and assistance; Dr. Glen Ditchfield, for being a constant source of ideas and new challenges, and for reading a draft of this thesis; Rodolfo Esteves, for his help in implementing the Cforall translator; my supervisor, Professor Peter Buhr, for his experience, encouragement, and attention to detail; and my wife, Jean Knetsch, for her love and support. I also gratefully acknowledge funding support from the Ontario Graduate Scholarships in Science and Technology and the Natural Sciences and Engineering Research Council.

For Claire, my Mother

Contents

List of Figures

Chapter 1

Introduction

This thesis describes an implementation of the programming language Cforall, an extension of the C language [19] that features overloading and parametric polymorphism. This implementation includes algorithms to analyze Cforall programs and to convert these programs into an executable form.

Cforall was first described by Glen Ditchfield [11], who also wrote the definitive specification of the language [12]. However, this document has not yet been updated to reflect some clarifications and changes that have been made to the language as a result of the work done in this thesis. Furthermore, the Cforall language has been extended in other ways since Ditchfield's thesis; in particular, it now incorporates many of the features of the KWC project [6, 35].

1.1 The Philosophy of Cforall

The C language was originally designed by Dennis Ritchie in the period from 1969–1973 as an implementation language for the UNIX operating system [29]. In the ensuing years it has become as notable for its deficiencies as for its merits — its lack of high-level abstraction mechanisms, reliance on error-prone pointer manipulations, and cryptic syntax being some of the most common complaints. Notwithstanding these faults, it has become one of the pre-eminent implementation languages for computer systems and applications, and much of its competition consists of the languages C++ and Java, which are themselves attempts to improve upon, rather than replace, the syntax and semantics of C.

While its many faults may make improving the C language seem like an easy task, the popularity and versatility of C makes any attempt fraught with controversy. The C language has been used for a wide variety of tasks, from the largest applications to the smallest embedded systems, from computationally-intense mathematical programming to time-critical control systems. C has proven itself versatile enough to support these varied application domains largely because of its simplicity, efficiency, and portability. Any proposed improvements will be judged by these same standards.

Preserving the characteristics that have made C a success imposes important constraints on both the design and the implementation of extensions. This thesis describes the implementation of a number of extensions; these implementations have been guided by the following principles:

- Standard C code must have the same behaviour when translated by a Cforall compiler as when translated by a C compiler. This condition is crucial.
- Standard C code must not be penalized, either in execution speed or executable code size, when translated by a Cforall compiler as compared to its translation by a C compiler.
- Cforall code must be portable to at least as many computer systems as C code.
- Subject to any limitations imposed by the first three principles, the translation of code using Cforall extensions must be done in the most efficient way possible, in terms of execution speed and executable code size.

This thesis addresses portability concerns by describing a translation of Cforall constructs to C – the translated code can then be compiled and optimized on any system with a C compiler. Unfortunately, it is not possible to translate all Cforall constructs into ISO Standard C while still preserving the performance and behaviour of existing code, but the extensions to Standard C necessary to support Cforall are themselves portable and widely-implemented. A translation system also makes it easier to judge the performance of the system according to the first and second principles: the translation of a standard C program should be essentially the same program. The limited access to the underlying computer available through a high-level language may, however, prevent some optimizations that would be possible with a full compiler.

1.2 Characteristics of the Cforall Language

Cforall extends the C language in many ways. This thesis deals only with a few of the major extensions, and the interactions among them.

1.2.1 Overloading

The basic principle of overloading is that many different entities (functions or data objects) can be defined with the same name in the same program without one definition hiding the others. Overload resolution is the process by which the use of an overloaded name is matched with one particular definition of that name, based on the context in which the name is used. In the simplest overloading systems, multiple functions can be defined with the same name but a different number of parameters, or with different parameter types. When the compiler encounters a call to a function with an overloaded name, it uses the number and types of the parameters to determine the appropriate function to invoke. Overloading of this form is found in the Java and C++ languages. Since the number and types of the parameters must uniquely determine the proper overload, this approach allows for a simple overload resolution algorithm.

In Cforall, functions that overload the same name may be defined with identical parameter lists but a different number of return values, or returning values of different types. To select the proper function for a call, the compiler must take into account the context in which the result of an expression is used, as well as the types of its parameters. Overloads can still be resolved efficiently, but it requires a more sophisticated algorithm for expression analysis, such as the ones devised by Cormack [8] or Baker [3]. Once an algorithm is in place to resolve return-value overloading, it is straightforward to extend the system even further. Variables as well as functions can have overloaded names; that is, multiple variables with the same name but different types can be accessible within the same scope, with the appropriate variable selected based on the context in which the name is used. Finally, the constants 0 and 1 can be overloaded (because of their special meaning to the C language and in many application domains).

Operator Overloading

Cforall also allows user-defined overloading of built-in operators, such as arithmetic and relational operators. These are specified by defining functions using special names. When the

compiler attempts to determine the particular operation to use to implement an expression, it implicitly re-writes expressions involving operators into a corresponding function-call form, e.g.:

- `?-?` is the name of the function that implements the binary subtraction operator. That is, an expression of the form `a - b`, where `a` and `b` are expressions, is re-written as `?-?(a,b)`.
- `-?` is the name of the function that implements the unary negation operator. An expression of the form `-a` is re-written as `-?(a)`.
- `?()` is the name of the function that implements the function call operator. An expression of the form `a(b,c,d)` is re-written as `?()(a,b,c,d)`.

Certain operators cannot be re-written in function-call form, since they require semantics that are different from that of a function call. These exceptions are:

- The cast operator, `(int)a`
- The `sizeof` operator, `sizeof a` and `sizeof(int)`
- The address operator, `&a`
- The short-circuit logical operators, `a && b` and `c || d`
- The conditional operator, `a ? b : c`
- The sequence (comma) operator, `a, b, c`
- The member selection operators, `a.b` and `a->b`

1.2.2 Polymorphism

One of Cforall's major enhancements to C is the ability to write functions using parametric polymorphism. Parametric polymorphism allows the code of a function to be independent of the types of the function parameters, enabling a single function to be used for many different argument types. As an example, consider two C functions:

```
int int_identity( int x ) {
  return x;
}

double double_identity( double x ) {
  return x;
}
```

where the code in each case is the same, but the types of the parameter and return value are different. The same code could apply to any type, but it becomes the programmer's responsibility to repeat the code explicitly for each type (and maintain them separately in the future). In contrast, this potentially infinite set of C functions can be represented by one polymorphic Cforall function:

```
forall( type T )
T identity( T x ) {
  return x;
}
```

Here, the function's dependence on its argument type has been removed by giving the function another parameter, the type parameter `T`. The set of possible values for `T` is the set of all types known to the compiler. For a specific use of the identity function, the value of this type parameter can be used by the compiler to determine if the argument to the function is well typed, and to determine the type of the value returned. In practice it is not necessary to specify the values of type parameters explicitly; they are inferred from the argument types. For example, given the function call `identity(1)`, the compiler can infer that the type `int` is the appropriate type to use in place of `T`, whereas in the call `identity("hello")` the appropriate type would be `char*`.

Kinds of Types

There are actually three kinds of type parameters in Cforall, corresponding to the three kinds of types in C: object types, incomplete types, and function types. The object types are the set of all non-function types in the language — including arithmetic types, pointers, arrays, unions, structures, and enumerations — where the size of the type is known. An incomplete type is a type that

designates an object but lacks the information necessary to deduce the object's size. In particular, the type `void`, an array type with unspecified length, or a structure or union type with unspecified members are all incomplete. A type that is incomplete may not be instantiated, nor can its instances be manipulated or assigned, but pointers to objects of its type can exist.[1] C places the same restrictions on function types, but function and data pointers are mutually incompatible. For this reason, there are three different kinds of type parameters. Type parameters designated by the keyword `type` are restricted to complete object types, so polymorphic functions can accept and return parameters of these types, instantiate new objects, and assign from one object to another. The keyword `dtype` designates type parameters that can represent any object or incomplete type, so a polymorphic function can only accept and return pointers to these types and instantiation and assignment are forbidden. The keyword `ftype` designates function type parameters, which are incompatible with other types and must be passed and returned through pointers.

Assertions

The utility of type parameters is limited by their universality. It is difficult to write useful polymorphic functions, since the code for such a function cannot assume the existence of any properties or functions related to values of a parameterized type (except that objects of types designated by the keyword `type` can be instantiated and assigned). It is possible to pass relevant operations along with function parameters, but this style of programming becomes cumbersome in large systems. As an example, here is a function that can be used to compute the square of its input:

```
forall( type T )
T square( T x, T (*multiply_T)( T, T ) ) {
  return multiply_T( x, x );
}
```

The major disadvantage with this style of programming is that a user of `square` must supply not only the number to be squared but also a multiplication operation; in general, the programmer may have to supply a large number of operations at every call to a polymorphic function, and the set of operations to be passed depends on implementation details of the function. To avoid

[1] Presumably these pointers point to objects instantiated in another part of the program, where the type is complete.

these problems, Cforall allows the specification of a set of properties or operations that must be available for a type parameter. As a result, the polymorphic function is reduced in its generality, but it can then make use of this additional information about the type to perform useful work. For example, in

```
forall( type T | { T ?*?( T, T ); } )
T square( T x ) {
  return x * x;
}
```

the body of the `square` function does not know the precise type of its parameter, but it can assume that there exists a multiplication operation that can be applied to a pair of values of this type, and it can use that operation to compute its result. As with type parameters, the compiler can automatically select the appropriate definition of the overloaded name for a particular combination of type parameters and pass that function as an implicit argument. Thus, a compiler interpreting the call `square(3)` would infer that the type parameter T is int, and that the integer multiplication operation must also be provided. In the call `square(1.0)`, the compiler selects the type double for T and multiplication on doubles as the operation. This selection is made based on the operations in scope at the point where the function call occurs.

This system of polymorphism has two properties that make it useful for large scale software development. From the perspective of a user of `square`, the interface betrays no hint of polymorphism — the function is called in exactly the same way as a monomorphic `square` function would be called. This enables the writer of `square` to substitute a more general function without breaking existing code. Secondly, the generic `square` can be used with an infinite family of types, including types that were unknown at the time `square` was written. Since Cforall supports general overloading, it is possible to implement a multiplication operation for the new type and have the compiler use this new operation as the assertion argument to `square` when applied to values of the new type. In this way, Cforall embodies the principles of *generalizability* and *incrementality*, as described by Cormack and Wright [9], enabling both the implementation and the uses of an abstraction to evolve independently.

Contexts

Cforall adds one more mechanism to this form of polymorphism: the notion of a context, a way of encapsulating a set of assertion parameters in order to describe the abstract properties of types at a higher level, e.g.:

```
context group( type T ) {
  T ?+?( T, T );  // binary operation
  T -?( T );      // inverse operation
  T 0;            // identity element
};
```

A context can be used to constrain a type parameter in a `forall` clause. Semantically, stating that a type parameter conforms to the context `group` is the same as stating that the operations specified by that context are assertion parameters in the manner described above. Contexts can constrain multiple type parameters simultaneously, and they can extend other contexts, e.g.:

```
context ring( type T | group( T ) ) {
  T ?*?( T, T );  // another operation
};
```

Here, a type constrained by context `ring` must support the `*` operation specified by the context definition, as well as the operations of context `group`.

Specialization

Specialization refers to the conversion of a polymorphic function or object into one that is less polymorphic, or not polymorphic at all. To promote generalizability, Cforall allows for specialization to happen implicitly — that is, any polymorphic entity can be used where a less polymorphic entity is required, provided that there exists a set of actual types and inferred parameters that can satisfy the assertions of the specialization candidate and give it a type compatible with the required type, e.g.:

```
void f( int (*p)( int, int ) );
forall( type U, type V | { U ?+?( U, V ); } ) U g( U, V ) );

f( g );
```

Here, the polymorphic function `g` is used in a place where a monomorphic function is needed, since a substitution of `int` for `U` and `V` gives the two functions compatible types. However, this is a legitimate specialization only if a `+` operation can be found in the current scope. Integer addition is a predefined operation, so it is guaranteed to exist. It is the responsibility of the Cforall compiler to infer this set of type and assertion arguments, which it can do at the same time that it infers other arguments.

1.2.3 Multiple Return Values

In Cforall, functions can return more than one result in the same way that functions in C can return either no results or one result. And, just as the result of a C function can become the argument of another function call (function composition), the multiple results of a Cforall function can provide multiple arguments for another function call, e.g.:

```
[ int, double ] f( void );  // returns 2 values
void g( int, double );

void h() {
  g( f() );                 // both arguments provided by call to f
}
```

Although the topic of expression analysis and code generation for functions returning multiple values has been discussed previously [35], the combination of multiple return values with overloaded names leads to significant new complexity in the analysis of expressions that has not been previously discussed.

1.3 Related Work

1.3.1 Overloading

The basic technique of overloaded names has a long history in programming languages, dating back to PL/I and Algol-68 and implemented in many languages including C++, Java, and Ada. In the first four of these languages, multiple functions can be defined with the same name but different number and types of parameters. The language Ada [17] broke new ground in allowing

functions with the same name to be defined with identical parameter types but different return types. Ada also allows the names of enumeration constants to be overloaded, but not variables or literal constants (such as Cforall's 0 and 1).

The language Haskell supports overloading as well, but in a highly constrained form. There is a discussion of the relationship between Cforall and Haskell in the polymorphism section.

1.3.2 Multiple Return Values

While many languages have some facility to return multiple values from a single function invocation, such as tuples in ML, the semantics of function composition in Cforall is unique. In ML, a tuple result can only be passed to a function that expects a tuple of the same dimension as an argument. In Cforall, multiple results are combined together so that a function can receive more than one of its arguments from a single argument expression. While this kind of implicit flattening of recursive lists is not unknown (for instance, the lists of the rc shell language [13]), its effects on a statically-typed language have not otherwise been explored.

1.3.3 Polymorphism

Parametric polymorphism has been implemented in many languages, including both research-oriented languages and those used more generally. This section does not attempt to be a comprehensive summary of all of these languages, but rather it focuses on those languages whose approaches to polymorphism most strongly influenced or most closely resemble the Cforall approach. Other languages that have attempted to combine parametric polymorphism with the C language include C++ [18] and Cyclone [22]. As well, parametric polymorphism has been proposed in a number of different forms as an extension to the Java language.

The type system of Cforall is based on $F^{\ni}_{\omega}$ [11], an extension of System F [28] that supports contexts and inferred parameters in addition to second-order lambda calculus. The problem of expression analysis for untyped terms in System F is known to be undecidable [38]. Cforall sidesteps this problem by requiring all terms to be explicitly typed; the only type inference required is for type parameters.

ML

ML [26] was the first language to support parametric polymorphism. Like Cforall, it supports universal type parameters, but not the use of assertions and contexts to constrain type arguments. Instead, it supports polymorphic data types, which can be used to constrain the *structure* of arguments, rather than the operations available. It is possible, for instance, to define a list type that is parameterized on the type of list element, and then to write a function that operates on lists with any element type. That function can manipulate the list (for instance, by adding or removing elements), but not the elements contained in it (since it can assume nothing about the element type). An example of this uses ML's pre-defined polymorphic list type:

```
fun concatenate [] list2 = list2
  | concatenate (elem::rest) list2 = elem::(concatenate rest list2);
```

This function has type `fn : 'a list -> 'a list -> 'a list`, a function taking two lists having any type of element (so long as both lists have the same element type), and returning a list of this element type. The output list is the concatenation of the two input lists, computed recursively.

The concept of parameterized data types, and their use to constrain polymorphic arguments, is orthogonal to the use of assertions in Cforall, and could be a reasonable and useful extension to the language.

ML also places some restrictions on the use of polymorphic functions that are not present in Cforall. In particular, polymorphic functions cannot be passed as arguments to other functions. The following Cforall function has no equivalent in ML:

```
void f( forall( type T ) void (*op)( T ), int a, char *b ) {
  op( a );
  op( b );
}
```

Here, a polymorphic function (`op`) is passed to the function `f`; this operation is applied to two values of unrelated type. This restriction is a result of ML's use of type inference; it has been proven that allowing polymorphic parameters makes ML-style type inference an undecidable problem [23].

Ada

Ada, like Cforall, allows the definition of polymorphic functions with assertion parameters. Unlike Cforall, Ada does not allow these polymorphic functions to be used as if they were monomorphic functions: the programmer must instantiate them, specifying type arguments explicitly. Cormack and Wright have shown that the need to explicitly instantiate polymorphic functions imposes a significant burden on the programmer [9], and Shen and Cormack later proposed an extension to Ada that allows implicit instantiation [33].

Haskell

The Haskell language attempts to combine ML-style polymorphism, polymorphic data types, and type inference with the notion of *type classes*, collections of overloadable methods that correspond in intent to contexts in Cforall. The example contexts of Section 1 can be written in Haskell as follows:

```
class Group t where
  (+++) :: t -> t -> t
  neg   :: t -> t
  zero  :: t

class (Group t) => Ring t where
  (***) :: t -> t -> t
```

Unlike Cforall, Haskell requires an explicit association between types and their classes that specifies the implementation of operations. To be able to use the built-in `Int` type with these classes, instance declarations must be provided:

```
instance Group Int where
  (+++) = (+)
  neg   = negate
  zero  = 0

instance Ring Int where
  (***) = (*)
```

These associations determine the functions that are assertion arguments for particular combinations of class and type, in contrast to Cforall where the assertion arguments are selected at

function call sites based upon the set of operations in scope at that point. Haskell also severely restricts the use of overloading: an overloaded name can only be associated with a single class, and methods with overloaded names can only be defined as part of instance declarations. These restrictions prevent the use of the traditional symbols `+`, `-` and `*` to represent the group and ring operations, since these operators are already overloaded as part of the predefined `Num` class that defines operations for arithmetic types.[2] Section 2.4 describes the restrictions on overloading in Haskell in more detail.

ForceOne and Sea

The style of polymorphism in Cforall gets its most direct inspiration from the research language ForceOne [39]. ForceOne pioneered polymorphic functions with inferred parameters that, as in Cforall but not Ada, can be used without requiring the user to explicitly instantiate them. It lacks the idea of a context as a higher-level abstraction of inferred parameters, but in other respects embodies the same style of polymorphism as Cforall.

The programming language Sea [40] combined ForceOne-style polymorphism with the C language, adding a number of other features as well including ML-style polymorphic data and garbage collection. The most striking difference between Sea and Cforall from an implementation perspective is in their respective approaches to code generation, which is discussed in Chapter 3.

[2] Indeed, `++` and `**` are also used by the standard library for list concatenation and exponentiation, respectively.

Chapter 2

Expression Analysis

Expression analysis refers to determining, for each expression in a program, whether that expression is valid and what the expression means — that is, what operations are required in order to implement the computation specified by the expression.

2.1 Expression Analysis in Cforall

At the same time that a Cforall compiler must permit the use of the language's advanced features, it must still assign the same meaning to C expressions as a C compiler would, both for expressions involving intrinsic operations and for calls to user-defined functions. This requirement means that a Cforall compiler must implement all of the ad-hoc rules that govern the treatment of the arguments of intrinsic functions, as well as the more general rules that apply to the arguments of function calls.

One goal of Cforall is to unify the treatment of intrinsic and user-defined operations. In Cforall, it is possible to define new functions that are invoked using arithmetic operators (for instance, to define a new operator + that applies to objects of a new type). As well, the set of rules that resolve overloading implement the same set of implicit conversions that are applied to the arguments of intrinsic arithmetic operations in C. These capabilities make it possible to provide type signatures for many of C's intrinsic operations in Cforall and to allow the expression analysis process to ignore the distinction between user-defined operations and these intrinsic operations.

2.1.1 Overloading

Unlike in C, multiple functions or objects with the same name can be accessible at the same point in the program and the compiler must choose from among them based upon the context in which the name is used: the types of subexpressions (if any), and the type expected by any expression using the result. Overloading can result in a large number of possibilities, since the subexpressions and the containing expression could involve overloaded names as well.

An overload resolution algorithm takes as input an untyped expression tree and produces as output a set of typed expression trees, each one representing a way of matching names in the expression with the declarations of objects or functions having those names. The size of this output set reflects how successful the algorithm is at matching names with declarations in such a way as to make the entire expression well-typed. If the output set is empty, there is no combination of matches that is well-typed; this corresponds to a failure of type-checking in a compiler for a language without overloading. A single result in the output set represents a successful resolution. More than one result means that the expression is ambiguous: there are multiple ways of matching names and declarations in a well-typed way, and no way to choose among the alternatives, e.g.:

```
void f( int );
void f( char* );
int g();
char* g();

f( g() );
```

In this example, there are two well-typed interpretations of the expression: the first `f` could be applied to the result of the first `g`, or the second `f` could be applied to the result of the second `g`. In Cforall, this program is ambiguous since there is no reason to prefer one interpretation over another.

Ranking Conversions

Given the large number of implicit conversions that can be applied to arguments in C and Cforall, there is a great potential for ambiguity. Even for expressions with more than one valid interpretation, however, some interpretations may be preferred to other ones. In particular, interpretations

that require fewer implicit conversions are generally preferred to those requiring more conversions. For this reason, the following example is allowed despite a possible ambiguity:

```
void f( int );
void f( double );
int g();

f( g() );
```

There are two well-typed interpretations of this expression: the result of the call to `g` is a valid input to either version of `f`. For the variant of `f` accepting a double, however, the result of `g` must undergo a conversion, so the interpretation of the expression involving the first `f` is preferred since it does not require conversion.

The allowed conversions have been listed by Ditchfield [12], together with a formula that defines which conversions are preferred over others. Conversions that decrease the range or precision of a type ("unsafe" or "narrowing" conversions) are the least desirable. Replacing a type parameter with a type argument is preferred to an unsafe conversion; the fewer type parameters that need replacement the better. Conversions that increase the range or precision of the type ("safe" or "widening" conversions) are the most preferred; these are ranked according to the amount of widening involved. This formula has been devised to be as general as possible while still implementing the classic C behaviour for conversions and promotions.

```
void f( char );                 // 1
forall( type T ) void f( T );   // 2
void f( double );               // 3
void f( int );                  // 4
int g();

f( g() );   // apply f to int value
```

In this example, applying the first `f` to an `int` requires an unsafe conversion, since the range of `char` is less than the range of `int`. Applying the second `f` requires the replacement of the type parameter `T` with the type `int`, so this alternative is preferred to the first. Applying the third `f` requires only a widening conversion, since the range of `double` is greater than that of `int`, so this alternative is preferred over either of the first two. The alternative involving the fourth `f` is the best, however, since it requires no conversion at all.

The total cost of a set of independent conversions (such as the conversions of a set of argument types to the corresponding parameter types) can be expressed as a triple, containing the number of unsafe conversions in the set, the total number of type parameters replaced, and the total cost of the safe conversions. When comparing two sets of conversions, the one involving the least number of unsafe conversions is preferred; if the sets involve the same number of unsafe conversions, the set involving replacement of the fewest type parameters is preferred; should the sets involve the same number of unsafe conversions and type parameters, the set having the least safe conversion cost is preferred.

2.1.2 Multiple Return Values

The fact that functions may return more than one value adds significant complexity to expression analysis since there is no fixed correspondence between formal and actual parameters, e.g.:

```
void f( int, int, int, int );
int g();              // return one int
[int, int] g();       // return two ints
[int, int, int] g();  // return three ints

f( g(), g() );
```

In this case, there are three different interpretations of the expression; in each case, the two calls to `g` account for different numbers of arguments, and hence correspond to different parameters of `f`.

The effect of this breakdown in the traditional correspondence of parameters and arguments is pervasive, since this correspondence is crucial throughout expression analysis: both overload resolution and type parameter inference operate by comparing the types of corresponding parameters and arguments. The classic algorithms to solve these problems assume implicitly that arguments can be considered independently. In Cforall, the consideration of a particular argument depends on how the overloading of other arguments are resolved.

2.1.3 Polymorphism

In order to validate and implement the use of polymorphism in a function call expression, the compiler must infer a type argument for each type parameter and determine an appropriate set of

arguments to satisfy all assertions. Since the inference depends on the type of the function being called as well as the types of the arguments to the function, it must be repeated for each way in which the overloading of sub-expressions can be resolved.

Type parameters are inferred by *unifying* the types of parameters with the types of their corresponding arguments, in the manner of Robinson [30]. The unification algorithm takes two types as input and produces a *substitution*, a set of pairs of type parameters and type arguments that can be *applied* to a type to replace type parameters with their corresponding arguments. More specifically, the output of the unification algorithm is a substitution that replaces type parameters in such a way as to make the two types the same; the algorithm returns failure if no such substitution exists. For example, the types

```
forall( type T ) T (*)( T, int )
```

and

```
forall( type U ) double (*)( double, U )
```

can be unified by the substitution { (`T`, `double`), (`U`, `int`) }.

The unification algorithm presented here is similar to the algorithm found in Aho *et al.* [1], although it uses a different representation for equivalences among type parameters. Instead of annotating the type graphs directly, this algorithm maintains a separate set of equivalence classes of type parameters.[1] Each equivalence class may in addition have a *representative type*, which is the most specific type that is known to be a valid substitution for the type parameters in the class. This description refers to an equivalence class of type parameters with a possible representative type as a *binding*. Since many pairs of types may need to be unified by the same substitution, this algorithm takes as an additional input a set of bindings that is augmented to produce a set of bindings as a result. At the end of a series of unifications, this set of bindings can be converted into a substitution using a simple algorithm that is described later.

Checking Specializations

In order to check the types of expressions involving implicit specializations, it is necessary to precisely define the necessary conditions under which specialization can occur. For two types σ

[1] Care must be taken to ensure that distinct type parameters in an expression that happen to have the same name are treated distinctly. The translator renames each type parameter uniquely in order to avoid confusion.

and τ, define σ to be a *subtype* of τ (denoted $\sigma \preceq \tau$) if any of the following conditions are true:

- σ and τ are the same type
- σ is of the form

  $$\texttt{forall}(\texttt{type}\ T_1, \ldots, \texttt{type}\ T_m)\ \sigma'$$

  for $m \geq 1$, τ is of the form

  $$\texttt{forall}(\texttt{type}\ U_1, \ldots, \texttt{type}\ U_n)\ \tau'$$

  for $n \geq 0$ (that is, τ' could be a type with no `forall` clause at the outer level), and there is some substitution for the type parameters $T_1 \ldots T_m$ that, when applied to σ', makes the resulting type a subtype of τ'
- σ and τ are both pointer types to σ' and τ', respectively, where σ' and τ' are both function types and $\sigma' \preceq \tau'$ [2]
- σ is a function type with m parameters and n return types and τ is a function type with p parameters and q return types, $m = p$, $n = q$, the type of each return value of σ is a subtype of the corresponding return value of τ, and the type of each parameter of τ is a subtype of the corresponding parameter of σ.

An implicit specialization of a type σ to a type τ is only possible if $\sigma \preceq \tau$ (of course, if $\sigma = \tau$ the types are compatible and there is no need of specialization). Expression analysis must determine, for each pair of argument and parameter types, whether the argument type is a subtype of the parameter type. The basic unification algorithm of Aho *et al.* [1] can be extended to find the substitution referred to in the second rule above.

Some examples demonstrate how the subtyping relationship correctly defines the set of well-typed specializations.

[2] This rule constrains the subtype relation in a way that is arguably artificial — it would be natural to extend it to all pointer types, and to array types as well. In this way, a function expecting an array of function pointers could be passed an array of pointers to more general functions. Unfortunately, it is not possible to implement this with sufficient efficiency to satisfy the basic design goals of the translator, for reasons discussed in Section 3.2.2.

```
void f( forall( type T ) T (*p)( T, T ) );
forall( type U, type V ) U g( U, V ) );

f( g );
```

In this case, the parameter p specifies that a function is required that is polymorphic in one type parameter. The type of the argument function g is polymorphic in two parameters, but the substitution { (U, T), (V, T) } unifies its type with the type pointed to by the parameter p, so the type of g after its decay to a pointer is a subtype of the type of p and valid specialization can be created. Considering an implementation of the function f provides an intuition that this is well-typed:

```
void f( forall( type T ) T (*p)( T, T ) )
{
  p( "Hello", "Goodbye" );
}
```

In this example, p is applied to two arguments, both of type char*. The definition of p constrains its two arguments to have the same type, even though the actual function being called (g) can accept arguments of different types.

```
void f( forall( type T ) T (*p)( T, T ) );
int h( int, int );

f( h );
```

Here, there is no substitution that could be applied to the type of h to make its type match the type pointed to by p, so allowing this example would break strong typing. To see why, consider again the implementation of f — it passes objects of type char* to its argument, but the actual function being called (h) accepts only integers. Since the implementation of the function f is entitled to depend on as many degrees of polymorphism in the function p as is specified by its type, the type of the corresponding argument must be at least as polymorphic.

```
void i( void (*r)( forall( type T ) T (*p)( T, T ) ) );
void j( forall( type U, type V ) U (*q)( U, V ) );

i( j );
```

According to the final rule in the definition of the subtyping relationship, to determine if the type of `j` is a subtype of the type pointed to by `r` it is necessary to determine if the type of `p` is a subtype of the type of `q`. Since there can be no substitution that replaces `T` with `U` in one case and with `V` in another, the subtype relation does not hold and so this example is not validly-typed. To understand why this example would break strong typing, it is again helpful to consider the implementation of the called function, in this case `i`:

```
void j( void (*r)( forall( type T ) T (*p)( T, T ) ) ) {
  // some function that is polymorphic in a single variable
  forall( type T ) T (*s)( T, T );

  r( s );
}
```

Here, the implementation passes a function that is polymorphic in a single variable to `r`, but the parameter `r` is bound to the function argument `j`, which expects an argument that is polymorphic in two type parameters. It is possible that `j` will apply the function it receives to arguments of two different types (corresponding to the type parameters `U` and `V`), but the function `s` that it receives requires that its arguments have the same type, so the strong typing of the language has been violated. In contrast, the following example is valid:

```
void i( void (*r)( forall( type T ) T (*p)( T, T ) ) );
void k( int (*q)( int, int ) );

i( k );
```

Here, the substitution { (`T`, `int`) } allows the type of `p` to match the type of `q`, so the former is a subtype of the latter and hence the type of `k` is a subtype of the type of `r` and the function application is well-typed.

Open and Closed Parameters

To handle the subtype relationship properly, I have extended the basic unification algorithm to keep track of whether type parameters are eligible for substitution or not; the former are *open* type parameters while the latter are *closed*. A closed type parameter unifies only with other occurrences of the same type parameter. As a result, the equivalence classes of type parameters

only contain open type parameters (although the representative type for an equivalence class may be a closed type parameter).

The algorithm categorizes type parameters into these two classes depending on how many function types enclose a parameter's `forall` clause, e.g.:

```
void f( forall( type T ) T (*p)( T, T ) );
forall( type U, type V ) U g( U, V ) );

f( g );
```

In this example, `T` is enclosed within a single function type and so is closed; `U` and `V` are outside of any function type and so they are open. This ensures that no substitution replaces the parameter `T`, but permits both `U` and `V` to be bound (both to `T`, in this case).

Inexact Unification

The unification algorithms referenced previously are *exact* in the sense that the substitution produced must make the two input types the same; if no such substitution exists, the unification fails. In Cforall there are cases where a more permissive unification is desirable, e.g.:

```
forall( type T ) T f( T, T );
int x;
double y;

f( x, y );
```

The type parameter `T` must unify both `int` and `double`; this cannot be done exactly, since there is no substitution that can make the types `int` and `double` the same. However, by choosing the type `double` as a replacement for `T`, both argument types can be safely converted to the types of the corresponding parameters. This property may hold for other types as well (the type `long double`, for instance), but `double` is the "narrowest" type with this property — that is, `double` can be safely converted to any other such type, whereas no other such type can be safely converted to `double`. Define the narrowest type reachable by safe conversion from two types σ and τ to be the *common type* of σ and τ. Often the common type is either σ or τ, as in the

case of `int` and `double`, but it is possible that the common type would be a third type.[3] It is conceivable that there could be more than one type satisfying the common type property for some pair of types, but given the set of types and conversions in Cforall it is possible to verify that the common type of two types is unique when it exists.

Inexact unification is not appropriate in every situation:

```
forall( type T ) T f( T*, T );
int *x;
double y;

f( x, y );
```

Here, `T` is first bound to `int`, and inexact unification allows this to be widened to `double`. Widening the base type of a pointer does not produce a convertible type, however — there is no way to implicitly convert `int*` to `double*`. This restriction holds for the base types of arrays as well, and for the parameter and return types of function types. For this reason, no types within these type constructors are allowed to be widened by inexact unification, and any representative types chosen from types within type constructors (such as the representative type `int` for `T` in the example) must be flagged to prevent them from being widened by later inexact unifications.

Given a basic unification algorithm (presented in the next section), two types σ and τ, an input substitution Γ and two booleans w_σ and w_τ, I define an algorithm for inexact unification $\text{unify\_inexact}(\sigma, \tau, \Gamma, w_\sigma, w_\tau)$ as follows. If σ and τ can be unified by the basic unification algorithm, the result of the inexact unification is the substitution resulting from that unification. If basic unification fails but σ and τ have a common type, then the result depends on w_σ and w_τ, which indicate whether it is permissible to accept a common type that is wider than σ or τ, respectively. If w_σ is false, the unification can only succeed if the common type is σ; similarly, if w_τ is false, the unification can only succeed if the common type is τ. If the choice of common type is acceptable given w_σ and w_τ, the result of the inexact unification is the input substitution. If basic unification fails and there is no common type, the inexact unification fails.

[3] On a typical 32-bit architecture, safe conversions are not defined between the types `long int` and `unsigned int`, but they can both be safely converted to the type `long unsigned int`.

Basic Unification

If σ and τ are arbitrary types and Γ is a (possibly empty) set of bindings, then $\mathrm{unify}(\sigma, \tau, \Gamma)$ is calculated as follows:

- If σ and τ are both pointer types to elements of types σ' and τ', respectively, then the result is $\mathrm{unify}(\sigma', \tau', \Gamma)$.
- If σ is an array of m elements of type σ', and τ is an array of n elements of type τ', then if $m \neq n$ the unification fails; otherwise, the result is $\mathrm{unify}(\sigma', \tau', \Gamma)$.
- If σ is a function type with m parameters and n return types and τ is a function type with p parameters and q return types, then if $m \neq p$ or $n \neq q$ the unification fails; otherwise, the result is the result of the successive unification of corresponding return values and parameters, with Γ used as the input set of bindings for the first sub-unification, the result set of bindings of each sub-unification used as the input set of bindings for the next sub-unification, and the result set of bindings of the last sub-unification used as the result of the unification as a whole. If any sub-unification fails, the unification as a whole fails.
- If σ and τ are both arithmetic, structure, union, or enumerated types, then if they are different types the unification fails; otherwise the result is Γ.
- If σ is a type defined using `typedef` to be a name for some other type σ', then the result is $\mathrm{unify}(\sigma', \tau, \Gamma)$; a similar rule applies for τ.
- If σ and τ are closed type parameters, then if they are different types the unification fails; otherwise the result is Γ.
- If σ and τ are open type parameters in the same equivalence class, the result is Γ.
- If σ is an open type parameter and τ is not an open type parameter, then:
  - If σ is bound to a representative type α, then if τ occurs in α, the unification fails; otherwise the result is $\mathrm{unify\_inexact}(\alpha, \tau, \Gamma, w_\alpha, w_\tau)$, where w_α is false if and only if widening of α has been flagged as prohibited and w_τ is true if and only if this unification is not a recursive unification within a type constructor.

  - If σ is not bound to a representative type then τ becomes its representative type.

  The widening of the resulting representative type for σ is flagged as prohibited if a previous representative type for σ was so flagged, or if this unification is a recursive unification within a type constructor.

  A similar rule applies when τ is an open type parameter but σ is not.

- If σ and τ are both open type parameters (and are not in the same equivalence class) then the result depends on the state of Γ. A new set of bindings Γ' is formed that contains the same bindings as Γ except that bindings involving σ and τ are removed and a new binding is added whose set of type parameters consists of the union of the equivalence classes of σ and τ in Γ.

  - If σ is bound to the representative type α in Γ and τ is bound to the representative type β in Γ, then if τ occurs in α or σ occurs in β the unification fails; otherwise the resulting substitution is the result of $\mathrm{unify\_inexact}(\alpha, \beta, \Gamma', w_\alpha, w_\beta)$, where w_α is false if and only if the widening of α is flagged as prohibited, and w_β is false if and only if the widening of β is flagged as prohibited. If the inexact unification succeeds, the new binding takes the common type of α and β as its representative type. The widening of the resulting representative type is flagged as prohibited if either α or β is so flagged, or if this unification is a recursive unification within a type constructor.
  - If σ is bound to a representative type α and τ is not, then if τ occurs in α the unification fails; otherwise the new set of type parameters is bound to α. The widening of the representative type of the new set is flagged as prohibited if α is so flagged or if this unification is a recursive unification within a type constructor. The result is Γ'. A similar rule applies when τ is bound to a representative type and σ is not.
  - If neither is bound to a representative type, the new set of type parameters is not bound to a representative type. The result is Γ'.

- If the two types are not covered by any of the above-mentioned cases, then the unification fails.

This algorithm is almost an exact unification algorithm in the style of Robinson, except that it calls upon inexact unification to unify the representative types of type parameter equivalence classes.

A type parameter σ *occurs* in the type α if σ, or any equivalent type parameter, plays any role in defining α — for instance, if α is a pointer to σ or a function taking a parameter of type parameter τ, where σ occurs in the representative type of τ. The intent of this "occurs check" is to prevent infinitely recursive types from forming, which might otherwise happen in a case such as:

```
forall( type T ) void f( void (*)( T, T* ) );
forall( type U ) void g( U*, U );
f( g );
```

Here, the type `T` is inferred to be a pointer to type `U`, which in turn is inferred to be a pointer to type `T`. This second inference results in both types being infinitely recursive, except that it fails the occurs check — `U` cannot be unified with `T*`, since `U` occurs in the representative type of `T`. Since recursive types cannot form, each instance of the occurs check terminates.

The basic procedure to convert a set of bindings into a substitution is to create a pair for each type parameter that replaces it with its representative type. At the end of the algorithm, however, a type parameter may have a representative type that involves other type parameters, each with its own representative type. These references to type parameters must be recursively expanded in order to produce an appropriate substitution. Once again, since the occurs check prevents infinitely recursive types from forming, this recursive expansion terminates. If any binding has no representative type, the expression is invalid. This could happen in a case such as:

```
forall( type T ) void f( void (*)( T ) );
forall( type U ) void g( U );

f( g );
```

Here, `T` and `U` are open type parameters that are only unified with each other.

Assertions

Assertion arguments are found by searching the accessible scopes for definitions corresponding to assertion names, and choosing the ones whose types correspond most closely to the asser-

tion types. This involves both overload resolution and type parameter inference, so it can be implemented as a special case of the more general expression resolution algorithm.

Since the use of a context to constrain type parameters is the same as specifying each of the context members separately as assertions (with an appropriate substitution of type parameters), the compiler can replace context constraints with ordinary assertions and treat those assertions uniformly along with any others. For instance, given the context definition

```
context group( type T ) {
  T ?+?( T, T );  // binary operation
  T -?( T );      // inverse operation
  T 0;            // identity element
};
```

the `forall` declaration

```
forall( type U | group( U ) )
```

is replaced with

```
forall( type U | { U ?+?( U, U ); U -?( U ); U 0; } )
```

Note that occurrences of the formal type parameter `T` in the context `group` have been replaced by instances of the actual type parameter `U` in the rewritten declaration.

The set of assertions required by an expression is formed as the combined set of all assertions from `forall` clauses that introduce open type parameters, for all types in the expression that are unified as a part of type parameter inference. The set of declarations of functions and objects that can be used to satisfy those assertions is the set of declarations that are lexically in scope at the point in the program where the expression occurs, plus the combined set of all assertions from `forall` clauses that introduce closed type parameters, for all types in the expression that are unified as a part of type parameter inference. An example motivates this distinction:

```
void f( forall( type T | { T -?( T ); } ) T (*p)( T, T ) );
forall( type U, type V | { U -?( U ); V -?( V ); } ) U g( U, V ) );

f( g );
```

Here the formal parameter `p` requires a function that can be applied to any type for which a negation operation is available. The actual argument `g` is more general than `f` in the sense that its two arguments may have different types, but each of those types must have a negation operation available. Since `U` and `V` are open type parameters that become bound to the closed type parameter `T`, this is equivalent to requiring that `T` have a negation operation defined. It is safe to pass `g` in the place of `p` because the formal parameter type of `p` guarantees that the assertions of `g` are satisfied when it is called.

The set of assertions considered is taken only from the forall clauses of types that actually participate in unification. This restriction avoids problems in situations such as:

```
void f( forall( type T ) T id( T ) );
forall( type U | { U g( U ); } ) U f( U ) );

id( f );
```

Here there is no need to satisfy the assertion `g` of the function `f`; the function `id` cannot call or specialize the function, so the assertion will never be needed.

2.2 Resolution Algorithm

The algorithm proposed here takes as its base the algorithm for overload resolution proposed by Baker [3] for the Ada language. This algorithm by itself is insufficient, however; it must be extended to take into account the use of polymorphic functions, functions returning multiple values, and the implicit coercions available in the Cforall language.

Baker's algorithm traverses expression trees from the bottom up. For each sub-expression, it creates a set of expression trees that represent different interpretations of the sub-expression, one for each type for which an unambiguous and well-typed combination of interpretations of its sub-expressions can be found.

As described in section 1.2.1, most kinds of expressions are treated as function-call expressions to facilitate the overloading of operators. Those operators that cannot be expressed as function calls are treated by the algorithm as special cases; treatment of these expressions is described in Section 2.2.5.

The analysis of a function call expression has four stages:

- determine the possible interpretations for the sub-expression that designates the function to be called, and the possible interpretations for the sub-expressions that compute the arguments of the call
- for each combination of sub-expression interpretations, unify the parameter and argument types
- determine the assertions that constrain the type parameters within the subexpressions
- choose from among valid interpretations of the function call expression by considering the kinds of implicit conversions involved in each

The ultimate goal is to determine a single interpretation for the expression as a whole. The compiler can then generate code for the expression that invokes the particular functions chosen, with appropriate arguments corresponding to the assertions. There may be no type-consistent interpretation of the expression. This situation can occur because of an attempt to call a function when there is no available definition of a function by that name with compatible arguments; or because there is a type parameter in the expression that remains unbound; or because some subexpression has an assertion that cannot be satisfied. Any of these cases is an error, since the compiler cannot generate any code. There may be more than one type-consistent interpretation of the expression. This situation can occur because some function call sub-expression has more than one possible function that is compatible with its parameters; or because some assertion has more than one way of being satisfied. Any of these cases is an error as well: the expression is ambiguous, since it has more than one possible meaning.

An interpretation, for the purposes of this algorithm, is a data structure consisting of a typed expression tree, a set of bindings for type parameters in that expression tree, and a triple that represents the total cost of conversions involved in that interpretation, as defined in Section 2.1.1.

2.2.1 Selection of Candidates

A function-call expression is composed of two parts: an expression that designates the function to be invoked and a list of expressions whose values are the arguments to be passed to the function. Since the algorithm traverses the expression tree from the bottom up, it first determines the set of possible interpretations for each of these sub-expressions.

```
forall( type T | { T ?*?( T, T ); } ) T square( T );

forall( type U | { U square( U ); } ) U f( U );
forall( type V ) V f( V* );
double f( double );

f( 7 );
```

Figure 2.1: Fragment of a Cforall program with an overloaded name

Candidate	Bindings	Assertions Needed
`forall( type U ...)  U f( U )`		
`forall( type V ) V f( V* )`		
`double f( double )`		

Figure 2.2: Result of candidate selection for the example in Figure 2.1

Once the set of interpretations of each sub-expression is determined, a set of candidate interpretations for the function call expression as a whole is formed by taking every combination of sub-expression interpretations. This potentially large set is winnowed down by later phases of the algorithm.

The most natural interpretations of the expression designating the function to be invoked are those that have function type[4], but an interpretation with another type could represent an invocation of a function that overloads the function-call operator. For this reason, a call of the form `a(b, c, d)`, for those interpretations of `a` that do not have a function type, is treated as if it were a call of the form `?()(a, b, c, d)` and candidate interpretations are generated as for other function call expressions. This step is not performed recursively — that is, candidates are only generated for interpretations of the expression `?()` that have function type.

Figure 2.1 shows part of a Cforall program involving the use of the overloaded name `f` in a function call. The expression `7` has only one interpretation, which has type `int` and involves no conversions. The candidate selection phase for the expression `f(7)` chooses all of the possible definitions of `f`, as shown in Figure 2.2; at this stage, no attempt has been made to determine type parameter bindings or to satisfy any assertions.

[4]Technically pointer-to-function, because of the implicit conversion of function values into pointers.

Candidate	Bindings	Assertions Needed
`forall( type U ...) U f( U )` `double f( double )`	`{ (U, int) }`	`int square( int )`

Figure 2.3: Result of type parameter inference for the example in Figure 2.1

Candidate	Bindings	Assertions Needed
`forall( type T ...) T square( T )`	`{ (T, int) }`	`int ?*?( int )`

Figure 2.4: Result of assertion satisfaction for the first candidate of Figure 2.3

2.2.2 Type Parameter Inference

To compute a set of bindings for each candidate, the types of corresponding parameters and arguments are unified together to create a complete set of bindings for the candidate expression. If the unification fails for any parameter, the candidate is discarded as it cannot be well-typed.

Figure 2.3 shows the set of candidates that results from type parameter inference on the candidate set from Figure 2.2. For the first candidate in Figure 2.2, the type `int` is inferred for the parameter `U` (since the interpretation of the sub-expression `7` has type `int`), and an assertion is found that must be satisfied. The second candidate of Figure 2.2 is eliminated, since there is no way to unify the types `int` and `V*` (`int` is not a pointer type). The third candidate is carried forward since it is possible to unify the types `int` and `double` (using the inexact unification rule).

2.2.3 Satisfaction of Assertions

For each candidate interpretation, the assertions to be satisfied (those introduced alongside open type parameters) are extracted during the unification process, as well as any additional declarations that can be used to satisfy them (assertions introduced alongside closed type parameters). Each assertion is satisfied by searching through the set of available declarations (declarations in scope, plus any additional ones extracted from the expression) for declarations having the name specified by the assertion — these are now candidates to satisfy the assertion. For each of these candidates, a unification is attempted between the type of the declaration and the type of the assertion, which may result in the binding of more type parameters, and the introduction of more

assertions, which are satisfied in the same way. This phase generates a new set of candidate interpretations of the expression: for every incoming candidate, a candidate is created for each combination of declarations that satisfies the set of assertions of the original candidate.

Figure 2.4 shows the result of attempting to satisfy the assertion of the first candidate in Figure 2.3. In this case, there is only one definition of square; unification results in inferring the type `int` for the parameter `T` and also in the discovery of the assertion `?*?`, which induces another round of assertion satisfaction. Since there is only one definition of `?*?` for type `int`, the assertion satisfaction process produces one candidate for each of the candidates in Figure 2.3.

2.2.4 Pruning the Candidate Set

From the set of candidates whose parameter and argument types have been unified and whose assertions have been satisfied, those whose sub-expression interpretations have the smallest total cost of conversion are selected (total conversion costs are compared according to the rules of Section 2.1.1). The total cost of conversion for each of these candidates is then calculated based on the implicit conversions and polymorphism involved in adapting the types of the sub-expression interpretations to the formal parameter types.

This set of interpretations is then culled one more time, since it is possible to detect some obvious ambiguities in the candidate set even without taking the expression's context into account. In a classic Baker algorithm, if two candidates result in values of identical type there is no way that any expression context can discriminate between them — if a sub-expression is ambiguous, any containing expression is ambiguous. In Cforall, a candidate is preferred over others returning values of the same types if the candidate has a lower total cost of conversion than the others. In this case, the candidate involving the fewest conversions is kept and others returning the same type are discarded. If two candidates have the same type and the minimal amount of conversions for that type, however, the candidates are ambiguous and all candidates with that type are discarded.

After this stage, the set of candidates represents the only reasonable interpretations of the expression.

For each of the candidates of Figure 2.3, there are no conversions involved in their sub-expression interpretations, so all of the candidates pass through the first pruning. Each of the candidates returns different types (`int` versus `double`), so they pass through the second pruning

as well. Thus the expression `f(7)` has two valid interpretations. Since this expression appears in an expression statement, the interpretation with the lowest total cost of conversion is selected (according to the rules of Sections 2.2.5 and 2.2.6). Since the first interpretation involves the replacement of a type parameter while the second involves a safe conversion, the second interpretation is chosen.

2.2.5 Other Expression Types

Variable references have as many interpretations as there are declarations of functions or objects with the referenced name; each interpretation has the same type as the corresponding declaration.

Member selection expressions (of the form `x.y`) have an interpretation for each interpretation of `x` that has a structure or union type with a member named `y`; each interpretation has the same type as the corresponding member.

```
struct { int a } x;     // 1
struct { double a } x;  // 2
x.a;                    // two interpretations, types int and double
```

Indirect member selection expressions (of the form `x->y`) have an interpretation for each interpretation of `x` that has a pointer-to-structure or pointer-to-union type whose underlying structure or union type has a member named `y`; each interpretation has the same type as the corresponding member.

Constants have one interpretation, whose type is determined by the C rules for the types of constants. The only exceptions to this rule are the special constants 0 and 1, which are treated as variable references.

Cast expressions (of the form `(type)x`) seek, from among the interpretations of `x` with the lowest total cost, the one that most closely corresponds to the specified type. This involves the same ranking of conversions described in Section 2.1.1, although with an expanded set of allowable conversions (since cast conversions are explicit, rather than implicit). Resolution of the cast operator results in zero or one interpretations, depending on whether a unique closest match to the specified type can be found from among the interpretations.

```
int f();      // 1
double f();   // 2
(double)f();  // select 2, since it requires fewer conversions
```

`sizeof` expressions can have either an expression or a type as its argument. If the argument is an expression, that expression must have a single interpretation. The type of this interpretation, or the specified type, is checked to make sure that it is not an incomplete or function type. The sizeof expression has one interpretation, which has type `size_t`.

```
int f();      // 1
double f();   // 2
sizeof f();   // invalid: no way to choose
```

Address expressions (of the form `&x`) have as many interpretations as `x` has interpretations that are lvalues; each interpretation has a type that is a pointer to the type of the corresponding interpretation of `x`. Lvalues arise as a result of variable references or member selections, or as a result of invoking a function that returns an lvalue-qualified type (the built in operators `*?` and `?[?]` are examples).

Short-circuit logical expressions (`&&` and `||`) are implicitly rewritten to compare their arguments unequal to 0, and the result cast to the type `int`. If the resolution of these rewritten sub-expressions fails, the resolution as a whole fails. Otherwise, the expression has a single interpretation of type `int`.

```
a && b;     →     ( (int)( a != 0 ) ) && ( (int)( b != 0 ) );
```

Conditional expressions (of the form `x?y:z`) are implicitly rewritten to compare `x` unequal to 0 and cast the result to `int`, as with the arguments of the logical operators. For each combination of interpretations of `y` and `z`, the types of the two interpretations are unified inexactly together, with widening allowed for either type; if unification is successful, an interpretation is generated whose type is the common type of `y` and `z`, e.g.:

```
int f();
double g();
x == y ? f() : g();  // result has type double, result of f() converted
```

Sequence (comma) expressions (of the form `x,y`) select an interpretation of `x` as if `x` had been cast to the type `void` (thus preferring expressions that return no value). If `x` can be resolved, the comma expression has an interpretation for each interpretation of `y`.

```
void f();   // 1: no results
int f();    // 2: one result
char *g();
f(),g();    // select 1; result type is char*
```

2.2.6 Invoking Resolution

Expressions occur in many different contexts within the language. Expressions that occur as sub-expressions of other expressions are resolved as part of the resolution of the expression as a whole. Expressions that occur in other contexts are resolved using the same algorithm, but the criteria used to select a single interpretation to be used in code generation varies:

Expression statements select an interpretation of the expression as if it had been cast to the type `void`.

```
void f();          // 1: no results
int f();           // 2: one result
f();               // select 1
```

`if`, `while` and `do` statements have their conditions implicitly rewritten to be compared unequal to 0, and the result cast to the type `int`. Resolution is performed as normal and results in either zero or one interpretations of the expression.

```
if( a ) ...     →     if( (int)( a != 0 ) ) ...
```

`switch` statements select the interpretation of their controlling expression that has an integer type; if there is not a single such interpretation the resolution fails.

```
char f();          // 1: an integer type
struct x *f();     // 2: not an integer
switch( f() ) ...  // select 1
```

`return` statements are implicitly rewritten to cast the expression to the function's return type; resolution is performed as normal and results in either zero or one interpretations of the expression.

```
int f();       // 1
double f();    // 2
int g() {
  return f();  // select 1 due to implicit cast
}
```

Initializers are implicitly rewritten to cast their expressions to the types of the corresponding objects or sub-objects; resolution is performed as normal and results in either zero or one interpretations of the expression.

2.3 Efficiency

The algorithm presented here has not been optimized in any sense, either for theoretical complexity or practical efficiency. The complexity and performance of the current implementation of this algorithm is poor; this section discusses some of the reasons for this and some ways to improve the performance. Assessing efficiency is difficult in the absence of empirical data, since there are many independent dimensions that define the size of the input and it is unclear where it is appropriate to approximate these dimensions by constants. For instance, the size of function parameter lists, the nesting depth of expressions in the program, and the number of definitions for each overloaded name are all independent of each other and all influence the complexity of the algorithm. As a result of this profusion of parameters, this section does not attempt to derive the algorithmic complexity of the entire algorithm, but instead focuses its analysis on certain obviously problematic sections of the algorithm.

2.3.1 Overload Resolution

For each sub-expression in an expression tree, Baker's original overload resolution algorithm compares, for each parameter, the type of that parameter with the types of all interpretations of the corresponding argument. For this reason, if there are m formal parameters for each function

and n interpretations for each sub-expression, the amount of work done is $O(mn)$.[5] Because of the possibility in Cforall that a single sub-expression interpretation might correspond to multiple parameters, it is impossible to do a simple matching of parameters and arguments in the manner of Baker. The algorithm presented here considers each combination of argument interpretations separately, matching parameter types with argument types. Given n interpretations for each sub-expression and p sub-expressions (since the number of arguments is now independent of the number of parameters), there are $O(n^p)$ combinations of argument interpretations. The algorithm compares m types for each combination of interpretations, so the complexity is $O(mn^p)$.

This complexity is far from the best that can be done, however. The correspondence between formal and actual may change depending on the choice of interpretations for other actuals, but there are only a limited number of possible correspondences. This is illustrated by an earlier example:

```
void f( int, int, int, int );
int g();
[int, int] g();
[int, int, int] g();

f( g(), g() );
```

Regardless of how many interpretations there are for `g`, there are only four possible ways in which each interpretation can correspond with formal parameters — one way starting at each formal parameter. Some of these ways are invalid; for instance, it is impossible for an interpretation returning two values (e.g., the second instance of `g`) to correspond to formal parameters beginning with the last one (since the second value could have no possible correspondent from among the formals). Thus there are at most mp correspondences between actual interpretations and formal arguments. While this insight does not eliminate the need to consider all n^p combinations of interpretations, it does allow the pre-computation of all possible correspondences in $O(mp)$ time, making the total complexity $O(mp + n^p)$. Since type comparisons are relatively expensive, this is a significant practical savings. A memoization approach is appropriate here, to prevent the pre-computation of correspondences that do not actually occur.

Another approach, based on a suggestion by G. V. Cormack, is to process each argument in turn, keeping track of a set of overload alternatives and, for each alternative, the list of parameter

[5] Assuming that types can be compared in constant time.

types that must be matched. At the start of the algorithm, the set of alternatives is initialized with one alternative for each function having the overloaded name. For each argument, the algorithm attempts to match argument interpretations with alternatives. The types of each argument interpretation (assume there are r types) are unified with the first r types from the alternative's list of types. If these unifications succeed, a new alternative is created based on the old alternative, but with the first r types dropped. After all combinations of argument interpretation and overload alternatives have been considered, the algorithm moves on to the next argument using the set of new alternatives.

This approach still suffers from exponential complexity in the worst case, where each attempt to match an argument interpretation and overload alternative succeeds. Given q interpretations for the overloaded name, n arguments and p interpretations for each argument, for the first argument the amount of work required is $O(pq)$. If each of the matchings succeeds, the size of the set of alternatives when processing the second argument is pq, which must be compared with p argument interpretations, meaning that the amount of work required is $O(p^2q)$. Similarly, when processing argument n, the amount of work required is $O(p^nq)$, so the total amount of work required is $O(\sum_{i=1}^{n} p^iq)$. The advantage with this approach is that the worst case is unlikely — that is, many matchings will likely fail. When a matching fails, all alternative interpretations that depend on that match succeeding are immediately eliminated from consideration.

The exponential nature is fundamental to the algorithm — indeed, given the presence of polymorphic types there could be $O(n^p)$ valid interpretations for each sub-expression. It has been observed in Ada that the number of sub-expression interpretations is bounded in practice by a small constant [33]; it seems reasonable to expect that this is true for Cforall as well.

2.3.2 Type Parameter Inference

Inferring type arguments for type parameters is handled primarily by the unification algorithm. Although unification algorithms are linear in common cases, without care they can be exponential in the worst case — indeed, the *size* of the resulting types can be exponential in the size of the input types. As an example, consider a case with open type parameters $f_1, f_2, \ldots, f_n$ on the formal side and $a_1, a_2, \ldots, a_n$ on the actual side, and with corresponding formal and actual

parameters

$$\begin{array}{cccccccccc} f_2, & f_3, & \ldots, & f_n, & \texttt{void}(*)(f_1, f_1), & \texttt{void}(*)(f_2, f_2), & \ldots, & \texttt{void}(*)(f_n, f_n) \\ \updownarrow & \updownarrow & & \updownarrow & \updownarrow & \updownarrow & & \updownarrow \\ a_1, & a_2, & \ldots, & a_{n-1}, & a_1, & a_2, & \ldots, & a_n \end{array}$$

The first few parameters make the type parameter a_m equivalent to f_{m+1}, for $1 \leq m < n$. The remaining parameters make a_m equivalent to a type involving the pair of types $f_m = a_{m-1}$, for $1 \leq m \leq n$. Thus the size of the type a_n, once all recursive expansions have been done, is $O(2^n)$.

Using a clever choice of data structures it is possible to guarantee linear space and time complexity (Martelli [25] describes a way of doing this). One way to achieve this is to represent types as directed acyclic graphs, where common components are not repeated within the graph. The unification algorithm discussed here has exponential worst-case behaviour as currently implemented, but it could be modified to use a more sophisticated representation for types.

Since the unification algorithm does many operations on sets of type parameters, its efficiency is dependent on the efficiency with which set operations can be implemented. Once again, the current translator implementation is naïve and inefficient, but it could be improved — nearly constant-time set operations are well known [4].

Type parameter inference is also affected by the exponential complexity of the overload resolution. In order to pre-compute correspondences between formal parameters and argument interpretations (as is suggested in the previous section), it is also necessary to pre-compute the set of bindings that can be inferred from the potential correspondences. When considering a particular combination of argument interpretations, the sets of bindings for the appropriate correspondences must be combined to create a complete set of bindings. Combining these sets of bindings may require additional unifications (where a type parameter is bound in two different sets of bindings).

2.3.3 Satisfaction of Assertions

Finding a particular definition to satisfy an assertion is essentially a special case of the general expression resolution algorithm, so it has similar efficiency. What is more troubling is that resolving a particular assertion may result in the introduction of more assertions to be resolved

```
forall( type T | { void f( T* ); } ) void f( T );
```

Figure 2.5: A declaration that could generate an infinite set of assertions.

with no guarantee that the introduction of assertions ever terminates. A simple example of a declaration that could generate an infinite assertion set is shown in Figure 2.5 (from Ziegler [40], converted to Cforall syntax). With this definition, any application of `f` to a type `T` requires the existence of a function `f` that applies to objects of type `T*`. Such a function can be obtained by instantiating the given function for type `T*`, but only if a version of `f` exists that applies to `T**`. In the absence of any other definitions of `f`, this instantiation continues indefinitely.

This defect is not specific to the particular algorithm chosen however — satisfaction of assertions in the Cforall type system is undecidable. Smith [34] shows that an arbitrary Horn-clause logic program can be embedded in a similar type system, and presents a reduction of the Post Correspondence Problem to the problem of satisfying assertions .

The formation of infinite assertion sets can be prevented simply by setting a limit on the number of recursive instantiations permitted. While this inevitably catches certain valid programs that cause a finite but large number of instantiations, there is undoubtedly some level that allows almost all practical programs through. This approach has been used to limit recursive template instantiations in the C++ language [18] and has proven to be successful in practice.

2.4 Related Work

The algorithm presented here is inspired by the one presented by Cormack and Wright [9], although it is structured differently and adapted to the Cforall language. Its treatment of polymorphic function parameters is different, and it accounts for the possibility that functions may return multiple values and that values can be implicitly converted to other types. As well, it expands context uses into the corresponding assertions. Other adaptations of this algorithm to different languages are discussed by Shen and Cormack [33] and Ziegler [40].

The classic type inference algorithm is Damas and Milner's Algorithm W [10], which implements type inference for ML using the standard unification algorithm. An ML type checker must infer types for all values across an entire program; in contrast, Cforall only infers type arguments,

and only at points of function application and specialization.

Although Cforall is similar to Haskell in that it combines overloading and parametric polymorphism, expression analysis is very different in the two languages. As in ML, Haskell types are usually inferred across an entire program. The "satisfaction of assertions" phase of Cforall's analysis also plays out quite differently in a Haskell type checker. Since types must be explicitly associated with type classes and with functions that implement the operations required by that class (using an *instance declaration*), the Haskell system does not search for in-scope declarations for each assertion. Instead, it simply ensures that an instance declaration has already supplied a set of operations for the particular combination of type and class. Wadler and Blott [37] show how resolving uses of type classes can be implemented as a preprocessor before an application of Algorithm W to infer principal types, although their overloading rules result in the same sort of undecidability as is found in the assertion satisfaction algorithm presented here [34].

Expression analysis is decidable in Haskell, however, because the set of allowable overloadings are restricted compared to Cforall. In particular:

- the types of all overloads must be instances of the same generic type — this type is defined in the class declaration using a type parameter. The instance declaration specifies a type argument to be substituted for that type parameter.
- if the type argument specified in the instance declaration is a constructed type, the arguments to the type constructor must be type parameters
- a type class can only constrain a plain type parameter — never a constructed type.
- a type class can not constrain a type parameter that does not appear in the type of the overloaded functions (even if that type parameter could otherwise become associated with a type argument through recursive assertion satisfaction).

Figure 2.6 shows a translation of Figure 2.5 into Haskell. A "pointer" data type must be defined, since Haskell has no built-in type constructor for pointers. The example does not type check, however, since the constraint `(has_f Pointer a)` violates the third restriction above. Intuitively, these restrictions ensure that as recursive assertions are satisfied the types involved become "less complex" and hence the computation eventually converges.

```
data Pointer a = Pointer a

class has_f a where
  f : a -> a

instance (has_f Pointer a) => has_f a where
  f x = f (Pointer x)
```

Figure 2.6: A translation of Figure 2.5 into pseudo-Haskell.

These rules guarantee decidability, but they are not minimally restrictive. Duggan, Cormack, and Ophel [14] show that an instantiating type constructor can have any sort of arguments, so long as the constructor used is unique across all overload instances and no type parameter appears multiple times within its arguments.

A different approach to type inference in the presence of implicit conversions is described by Mitchell [27]. This work is targeted at the different but related problem of general type inference in ML-like languages. It is fundamentally different from the algorithm presented here, however, because it assumes that the set of coercions defines a subtyping relationship that extends to type constructors (e.g., if `int` can be coerced to `double`, then function returning `int` can be coerced to function returning `double`). The Cforall coercions do not define subtypes in this way, so two varieties of unification (exact and inexact) are necessary.

Chapter 3

Code Generation

This chapter describes a translation of Cforall constructs into the C language. All uses of overloading, polymorphism, and multiple return values must be converted into another form for compilation by a C compiler that supports none of these features.

Converting functions and objects with overloaded names to C is a well-known problem, with a standard solution: give the entity a new name that encodes both its original name and its type (or a part of its type). This new name is known as a *decorated name*. Since the combination of name and type must be unique, this process is guaranteed to rename the entity uniquely. As well, all uses of these names must be updated to reflect the renaming. An encoding of this kind is described by Ellis and Stroustrup [15] for an implementation of the C++ language. The current Cforall translator uses a similar scheme, extended to allow variables with overloaded names, to allow functions to be distinguished based only on their return type, and to allow for polymorphism.

Converting functions returning multiple values into standard C form has previously been discussed by Till [35] and is not discussed further in this thesis.

Converting polymorphic functions and objects into a monomorphic form is a complicated problem that allows a number of different approaches with different trade-offs. This conversion is the primary subject of this chapter. There are actually two different transformations involved, corresponding to the two cases in which polymorphic type parameters become associated with arguments: through specialization and at function calls. First, specializations are eliminated by converting the program to a form in which type parameters become associated with type

arguments only at function calls, and then the function calls are converted into a monomorphic form. This chapter discusses these two transformations in reverse order.

3.1 Converting to Monomorphic Form

The fundamental challenge of generating code for polymorphic functions is that the generated code must depend on certain properties of its type arguments, even if only to pass arguments of these types to other functions and accept return values. Appel [2] divides implementation approaches for polymorphic functions into four categories:

Expansion in which a function's code is duplicated for each combination of type parameters on which it is used, and type parameters within the function body are replaced by the corresponding actual type arguments.

Type passing in which arguments representing the actual values of the type parameters are passed to polymorphic functions. These parameters can be used by the function when it must rely on the layout of polymorphic data.

Boxing in which the layout of data objects are made uniform so that polymorphic code can then treat all types of data in the same way.

Coercion in which data values are converted into a boxed representation as they are passed to polymorphic code, and boxed values are "unboxed" as they are passed to monomorphic code.

Although polymorphic objects can exist in the language, they are relatively useless (the intrinsic null pointer constants being the only useful examples). Since polymorphic objects can be treated as if they are functions taking no arguments and returning a single value, this discussion only deals with the problem of implementing functions.

3.1.1 Expansion

The process of expansion involves replacing a polymorphic function with a set of functions, one for each combination of type arguments of the function. For each copy, occurrences of type pa-

rameters are replaced with their corresponding arguments, making the copy monomorphic. Calls to the polymorphic function are replaced with calls to the appropriate expanded variant. Expansion generally produces the fastest implementation, since the overhead of the polymorphism can be resolved at compile time. The added cost of polymorphism occurs in the duplication of code, commonly referred to as "code bloat" — expansion improves code speed at the expense of increasing code size, which may not be an appropriate trade-off for all applications (such as a memory-constrained embedded system).

Expansion is unsuitable as a primary method of implementation since a polymorphic function in Cforall has an identity independent of the types to which it is applied, and expansion provides no way to represent this identity — all uses of polymorphism must be resolved statically, typically by the compiler. This requirement has a number of practical implications. The definition of a polymorphic function cannot be compiled independently of the uses of the function, which violates the traditional model of separate compilation in C. This limitation is not a trivial concern, since separate compilation provides the only real form of information hiding in the C (and Cforall) languages. As well, using pointers to polymorphic functions becomes largely impossible, because the choice of pointed-at functions may not be predictable at compile time.

In addition to these concerns, there are certain types of polymorphic functions that have no finite expansion in terms of monomorphic functions. Cforall permits the definition of functions exhibiting *polymorphic recursion*, where the type domain over which the function is applied varies at run time. Figure 3.1 shows a function that may call itself recursively on a pointer that is one level more indirect than the pointer passed in. The number of levels of indirection, and hence the number of distinct types on which the function can be invoked, is governed by the run-time parameter `n`, so the only way to implement such a function using expansion would be to generate expansions for all possible values of `n`, which is impractical.

In short, the expansion approach imposes significant constraints on the way in which polymorphic functions can be coded and used, in exchange for maximizing the speed of the generated code. For this reason, it is unsuitable as a translator's only implementation strategy, but it is an important optimization. In this way, it is similar to the process of in-line expansion for functions: not every function can be in-lined completely, and some functions should not be in-lined even where it is possible to do so, but the technique is still important in optimizing commonly-used functions.

```
forall( type T ) void *multiptr( T *base, int n ) {
  if( n == 0 ) {
    return base;
  } else {
    T** next = malloc( sizeof( T* ) );
    *next = base;
    return multiptr( next, n - 1 );
  }
}
```

Figure 3.1: Function demonstrating polymorphic recursion.

Implementation

In order to create an expansion of a polymorphic function, the translator simply creates a new copy of the function and substitutes occurrences of type parameters for the corresponding arguments, according to the substitution created during the type parameter inference phase of expression analysis. The expression analysis guarantees that the resulting function is valid.

Figure 3.2 shows a simple example of a polymorphic function, and Figure 3.3 shows its translation according to an expansion algorithm that replaces both type parameters and assertion parameters with their inferred values. Note that the function `?*?` is predefined for integers to represent the intrinsic multiplication operator, and a function version is provided by the system to be passed as a parameter. Since `?*?` is not a valid function name in C, the function is renamed to `__multiply_int` in the generated code. In the output from the real translator, all names are transformed further to prevent conflicts due to overloading.

The monomorphic function generated as a result of expansion may call a polymorphic function, requiring an expansion of this second function. This process is repeated until no calls to polymorphic functions remain. To ensure that this process of repeated expansion eventually terminates, it is necessary to detect functions like the one shown in Figure 3.1. It is unnecessary to avoid all recursive functions: as long as the function is called with the same set of type arguments, the recursive call can be replaced with a recursive call to the existing expansion. For calls where the type parameters have different values, however, a full expansion may be impossible. As an optimization, the expansion could be done for a finite number of levels to attempt to

```
forall( type T | { T ?*?( T p1, T p2 ); } )
T square( T t )
{
  return t * t;
}

int main( int argc, char **argv )
{
  printf( "%d\n", square( atoi( argv[1] ) ) );
  return 0;
}
```

Figure 3.2: A simple example of a polymorphic function and its use.

```
extern int __multiply_int( int, int );

int square_int( int (*multiply)( int, int ), int t )
{
  return __multiply( t, t );
}

int main( int argc, char **argv )
{
  printf( "%d\n", square_int( __multiply_int, atoi( argv[1] ) ) );
  return 0;
}
```

Figure 3.3: A translation of Figure 3.2 using expansion.

account for some common cases.

3.1.2 Type Passing

Type passing is perhaps the most literal interpretation of parametric polymorphism: each type parameter is translated into a data parameter in the real code. This type descriptor describes the characteristics of the data type, enabling the polymorphic code to deal appropriately with values of the type. Potentially, the polymorphic code must accept parameters of that type, pass parameters of that type to other functions, accept return values of that type from those functions, and finally return a value of that type to its caller. It may also need to allocate local variables of polymorphic type.

In order to interoperate with monomorphic code, the polymorphic code must obey the same *calling conventions* — the rules that define how the low-level mechanisms of a particular processor architecture (e.g., stacks and registers) are used to implement parameter passing and function return for values of different types. For instance, integer values could be passed in one kind of register, floating-point values in another, and structure values on the processor stack. Since types in polymorphic code may be unknown at compile-time, the translator must generate code that is able to pass and return values of any type using the low-level mechanism appropriate to that type.

On one hand, type passing can be an improvement over coercion because the values can be passed between monomorphic and polymorphic code without an intermediate boxing step. However there could be a real cost involved in implementing calling conventions at run-time, since the conditional branches involved could defeat important optimizations and instruction pipelining. Whether type passing is a net improvement over coercion probably depends on how often the input program crosses the polymorphism/monomorphism boundary and how complicated the calling conventions are on the target architecture.

From the perspective of the translator implementation, this approach is dependent upon the specific calling conventions of the target architecture and compiler, making it inherently nonportable. A translator from Cforall to C using type passing must augment its output with assembly code or compiler hooks that allow it to precisely control the process of parameter passing and function return. On many compilers the calling convention can be changed using compile-time options, so such a translator must be aware of these options as well and adjust its behaviour

accordingly. These factors make type passing a better candidate for a true compiler, rather than a translator.

As an example, consider the calling convention of the GNU C compiler on the Intel 386 architecture. This calling convention is simple compared to many conventions, since all function arguments are (by default) pushed on the processor stack and most return values are placed in the EAX register. However, there are a number of subtle twists that complicate the implementation. For example:

- The stack must be word-aligned, so objects whose size is an uneven number of words must be padded as they are pushed.
- When structures are returned, the caller allocates memory for the structure and passes the address of this memory as an additional argument.
- When floating-point values are returned they are placed at the top of the floating-point unit's register stack.

Code to implement all these possible behaviours must be inserted around each call within a polymorphic function. There are certain cases in which the compiler can statically rule out some types (for instance, a function that deals only with pointers), but for common examples like the square function in Figure 3.2 the return value from the `?*?` function must be placed in EAX or on the floating-point stack or in caller-allocated storage, depending on the type T. In addition, GNU C has command-line options that change this calling convention (for instance, to pass some parameters in registers), which would necessitate changes in the output of the Cforall translator. Accounting for this variability, combined with the inherent machine dependency and the obvious inefficiency of complicating the call sequence in this way makes a type-passing style impractical.

Alternatively, Cforall could define its own calling convention that makes a type-passing style reasonable. On virtually all platforms, however, the calling convention for C code is optimized for maximum speed across a range of practical applications, so it is inevitable that simplifying this convention for the benefit of polymorphic calls would be less than optimal for monomorphic calls. As a result, the performance of standard C code would degrade when compiled by the Cforall compiler, which violates a basic design goal.

3.1.3 Boxing

The idea of boxing has its origins in dynamically-typed languages such as Lisp, where any variable can potentially hold any data value. The essential idea is to put data values of different types into "boxes", where the characteristics of the boxes are uniform and independent of the type of the data inside. Typically, this is done by using pointers to data values rather than the data values themselves; on most computers, pointers have the same structure regardless of the type of the pointed-at data. The C standard mandates that all pointers to data can be coerced to the type `void*` and back without loss of information, so `void*` is an appropriate box type for data values[1].

The disadvantage to boxing is when the data value must be removed from its box, or "unboxed", because it is needed in a type-dependent operation, such as arithmetic. Using a pointer-style boxing scheme, this makes virtually all value accesses in a program indirect accesses into memory. The resulting overhead is too much for many computationally intensive applications. Since restricting Cforall's application domain in this way is unacceptable, boxing is unacceptable as an implementation method.

3.1.4 Coercion

A refinement of boxing is the idea of coercion, where values are only boxed when necessary, and left unboxed otherwise. In particular, values are boxed when they are passed from monomorphic code to polymorphic code, and unboxed when passed from polymorphic code to monomorphic code. This isolates the cost of the boxing to those places where polymorphic and monomorphic code are mixed. The transition occurs when values are passed as the arguments of functions and returned as results. The essence of the coercive transformation presented here is the insertion of code at these transition points to handle the boxing and unboxing. For simplicity, this discussion refers to functions where the type of a particular value is independent of that function's type parameters as "monomorphic code", although that function could manipulate other values of variable type as well; similarly, "polymorphic code" refers to a function where the type of the

[1]The C standard does *not* require that function pointers be representable by void*. It does say that any function pointer can be cast to another function pointer type and back without loss of information, so `void(*)(void)`, the type of pointers to functions that take no parameters and return no result, is an appropriate box type.

value is known only by the name of a type parameter, although there could be other values in the same function whose type is independent of the type parameters.

Boxing Transformation

Boxing turns every value into a pointer. Those values that have an address (for instance, variables or members of structures and unions) have their address taken and used as the boxed representation. For those values that do not have an address (constants and the results of most expressions), the value is assigned to a temporary variable and the address of the temporary is used as the boxed representation. The unboxing transformation is a dereference.

The boxing transformation has one unfortunate side effect: for values that have an address, their address is used as a box, which means that the recipient of the box can potentially modify the value inside and thus change the actual argument. This situation could change the semantics of C programs that assume their input parameters are copies of the corresponding actual values. One solution is to make a temporary copy of the value before passing it (as is done for values without an address), but this results in needless copying for the many functions that never modify their arguments. For this reason, any polymorphic function that may cause the modification of one of its parameters must make a temporary copy of that parameter. The only exception to this rule is for input parameters that are known to have pointer type, since no pointer values are ever boxed. Interestingly, a polymorphic function can never change the values of its parameters directly (since all operations, including assignment, are performed by calling functions), so the only case in which it could cause the change of a value is when it passes the address of one of its parameters to another function.

It is important to note that not every parameter with a polymorphic type requires that the corresponding argument be boxed. In particular, pointer parameters do not require boxed arguments, e.g.:

```
forall( type T ) T *f( T * );
```

Since data pointers can be coerced back and forth from the box type `void*`, there is no need to add another level of boxing. Function and array parameters are implicitly passed through pointers as well, so no additional boxing is needed for these parameters. As well, it is a common C programming practice to pass and return structure instances through pointers. As a result of

these considerations, there is a large class of useful function parameters that do not require boxed arguments and the consequent overhead.

Adding Coercions

Implementing boxing and unboxing for calls from monomorphic code to polymorphic code is straightforward. Any code necessary to box parameters can be in-lined just ahead of the function call. To receive a return value, a temporary is allocated and the address of the temporary is passed as an extra parameter, removing any need for the polymorphic function to dynamically allocate storage. These transformations are straightforward because the translator knows the arguments corresponding to all type parameters, and hence can generate the appropriate code to perform the boxing and unboxing.

Implementing boxing and unboxing for calls from polymorphic code to monomorphic code is difficult, however, because the type arguments are unknown to the polymorphic code. This situation precludes any simple in-lining of boxing and unboxing actions within the polymorphic code, since the details of boxing and unboxing vary depending on the type arguments. It is only at the point where the polymorphic code was originally called from monomorphic code that the type arguments are known, so the monomorphic code must pass additional information to enable polymorphic code to box and unbox values.

One important observation about calls from polymorphic to monomorphic functions is that virtually all monomorphic functions called by a polymorphic function must have been passed as arguments to that function (whether explicitly or through an assertion).[2] For example, the parameter `?*?` to the `square` function in Figure 3.2 is a monomorphic function passed as a parameter to a polymorphic function. If a polymorphic function calls any other monomorphic function, that function would have to be declared outside of the polymorphic function, which means that it could not involve the type parameters of the polymorphic function; there is no way for any value of unknown type to be compatible with a parameter of fixed type.

The Cforall translator uses this observation to exert control over the calls from polymorphic functions to monomorphic ones. For each monomorphic function that is a parameter to a

[2]The only exception to this rule is monomorphic functions obtained as specializations of polymorphic functions. Since all polymorphic types have the same (boxed) representation, no extra boxing or unboxing is necessary in this case.

```
extern int __multiply_int( int, int );

void square( void (*__multiply_thunk)( void *result, void *p1, void *p2 ),
             void *result,
             void *t )
{
  __multiply_thunk( result, t, t );
}

void
__multiply_thunk_for_int( void *result, void *p1, void *p2 )
{
  *(int*)result = __multiply_int( *(int*)p1, *(int*)p2 );
}

int main( int argc, char **argv )
{
  int temp = atoi( argv[1] );
  int ret;
  square( __multiply_thunk_for_int, &ret, &temp );
  printf( "%d\n", ret );
  return 0;
}
```

Figure 3.4: A translation of Figure 3.2 using coercion.

polymorphic function, the translator creates a “thunk” function, whose purpose is to unbox parameters appropriately, call the original function, and then box the return value. [3] Figure 3.4 shows a translation of Figure 3.2 using a thunk to perform and appropriate unboxing and boxing in the call to `?*?`.

These simple thunks are insufficient in cases where the monomorphic function to be called is not statically determined, e.g.:

[3]The term “thunk” was coined by P. Z. Ingerman in 1961 to refer to the encapsulation of an expression passed as a function parameter in the language Algol 60 [16]. This usage is similar, although the “expression” in this case simply forwards function parameter and return values to a target using a different calling convention.

```
forall(type T)
T select_and_apply( T (*array[])(T), T arg, int selector )
{
  return array[selector](arg);
}
```

At the call site, a pointer to an array of monomorphic functions is passed to a polymorphic function. It is impossible for a monomorphic function calling `select_and_apply` to generate thunks for every function in the array because the contents and size of the array may vary from one invocation of the function to the next. Instead, the translator uses an "adapter" function that takes as its parameters the function to be called and the parameters to be unboxed and passed. After transformation, the polymorphic function has both the adapter and the original function array as parameters:

```
void select_and_apply( void (*adapter)( void (*func)(),
                                        void *ret,
                                        void *parm ),
                       void (*array[])(),
                       void *ret,
                       void *arg,
                       int selector )
{
    adapter( array[selector], ret, arg );
    return;
}
```

The monomorphic code that calls `select_and_apply` generates an appropriate adapter that unboxes the arguments, passes them to the function passed in, and boxes up the return value. If `select_and_apply` is changed to pass the array, or any element thereof, to another polymorphic function, the generated code must pass the adapter to that function as well.

In fact, it is necessary to use adapters rather than thunks in all cases, even in cases where enough information is available to create thunks. Consider the following example (courtesy of Glen Ditchfield):

```
int f(int v) { ... };      // 1
float f(float v) { ... };  // 2

int (*pfi)(int) = f;       // pfi points to 1

forall(type T | T f(T) )
int rtti(T) {
    return f == pfi;
}

rtti(7);   // Must return 1.
rtti(0.5); // Must return 0.
```

In this case the actual identity of the passed function is important; replacing this function with a newly-created thunk would cause the pointer comparison to fail in all cases. Thus, the original function must be passed, along with an adapter to be used when the function is actually called.

Generating Adapters

The most obvious way to generate adapter functions is to emit their definitions into the generated code immediately before their use. This approach is complicated by the standard C restriction that function definitions may not be nested. The current translator implementation requires that the target C compiler support nested functions, which makes its generated code potentially invalid C, although nested functions are supported by the popular GNU C compiler. The adapter functions do not depend on the values of local variables in the containing function, so often they can be trivially “hoisted” out of the containing function to the top level, making nested functions unnecessary. In some cases, however, they may depend on local declarations of types and function prototypes, e.g.:

```
forall( type T ) void f( T (*)( T ) );

void test() {
  union q { int r; float s; };
  struct x { struct q a, b; };
  struct x g( struct x );

  f( g );
}
```

In this example, any adapter that adapts the function `g` must mention the structure `x`, which in turn mentions the union `q`. As a result, any attempt to hoist the adapter out of the `test` function also requires hoisting `x` and `q`. While this is possible, it adds complexity to the implementation and requires renaming to avoid name clashes in the global scope. For these reasons, this hoisting transformation has not yet been implemented.

It is also possible to re-use adapter functions. Since the function to be invoked is passed as a parameter to the adapter, the same adapter can be re-used for any function having the same parameter and return types. When the adapter functions are nested within a function, their definition is only available within that function, and hence the possibility for re-use is limited — the same adapter must be emitted within every function that needs it. If adapter definitions can be hoisted to the top level, however, the adapters can be re-used across an entire translation unit.

Type Descriptors

By using a boxed representation it is possible to avoid any kind of explicit representation of types for `dtype` and `ftype` type parameters. Type parameters declared using the keyword `type`, however, can be instantiated and assigned, e.g.:

```
forall( type T ) void swap( T *a, T *b ) {
  T temp = *a;
  *a = *b;
  *b = temp;
}
```

This case means that a polymorphic function must know the size of the type, so that it can allocate new instances using a facility such as `malloc` or `alloca`.[4] As well, an assignment operation must be provided, even if no assertion requires one. In this sense, a type descriptor is necessary in addition to boxing.

3.1.5 Representing Assertion Arguments

One obvious way to represent assertion arguments is to convert them into explicit arguments, which is the approach taken in Figures 3.4 and 3.3. The advantages of this method are its simplicity and the low overhead for using the assertion arguments. The primary disadvantage is a potential explosion in the number of function arguments in a call; on many architectures a certain number of arguments can be passed efficiently (i.e., in registers), but exceeding a small fixed limit can make passing the rest of the arguments substantially more expensive. The possibility of many assertion arguments is real, given that contexts allow many assertion arguments to be grouped together and that they can include other contexts.

Another approach is to group the assertion arguments into an array and pass a pointer to the array as a single argument. This approach is similar to the concept of a virtual function table in C++. While parameter list explosion is avoided, the overhead is two-fold: the array must be created, and each use of an assertion parameter requires an additional lookup in the array. By allowing programmers to group assertion arguments into contexts, a Cforall translator can use that grouping information to build arrays for various combinations of types and contexts that can potentially be reused in multiple calls. The creation of such arrays is straightforward in languages, such as Haskell, where types must be explicitly associated with sets of assertion arguments, since the array of arguments never changes. In Cforall, it is possible for an assertion argument to be different for each invocation of a function, even if all of the invocations are applied to the same type, e.g.:

[4] Alternatively, an allocation function specific to the type could be passed as a parameter.

```
forall( type T | { T f( T ); } ) void callee( T );

void caller( int (*array_of_func)( int ), int n ) {
  for( int i = 0; i < x; i++ ) {
    int (*f)( int ) = array_of_func[ i ];
    callee( i );
  }
}
```

Here, assertion satisfaction matches the assertion of `callee` with the function pointer `f` declared just inside the `for` loop. But the value of this pointer is different each time through the loop, so any array of the assertion arguments to `callee` must be updated each time `f` is given a new value. In this case, any array building undoubtedly wastes more time than it saves. Such perverse examples are unlikely in practice, however, so this could be a legitimate optimization in general. This example shows, however, that some form of analysis is necessary to determine at what points in the program argument arrays should be rebuilt.

A third approach is possible when the polymorphic function is expanded for a particular combination of types: replace references to assertion parameters with references to the corresponding arguments. This may result in multiple expansions of the same function for the same set of type arguments, if the set of assertion arguments changes from one use of the function to another. While this approach removes any overhead associated with assertion parameters, the previous example demonstrates circumstances in which such a replacement is impossible: the choice of a particular assertion argument is deferred until run-time.

3.1.6 Efficiency

Clearly, expansion is the most efficient implementation method in situations where it can be applied. The coercion method is more generally applicable than expansion and more portable than type-passing so it has been chosen as the implementation method in the current translator, but it pays a significant price in efficiency. Many data values incur the overhead of boxing (although many do not), sometimes requiring the allocation of temporaries and copying of the values. Many calls from polymorphic functions to monomorphic functions must be done through an adapter, which adds the overhead of another function call.

These facts seem to suggest a combination of two approaches: expand where possible, and coerce otherwise. Computationally-intensive polymorphic functions can be written to permit expansion, while code that is executed less often can be written with more freedom, taking advantage of additional capabilities such as separate compilation and the use of pointers to polymorphic functions. Expansion is most beneficial to code that manipulates arithmetic types, since boxing imposes a large overhead relative to object size; for objects passed as pointers, the additional cost of boxing is small.

For new programs making heavy use of polymorphism, it may be advantageous to adopt a different translation strategy, one that redefines the standard calling convention to make type passing feasible. In comparison with the hybrid approach just described, it is not clear that there is any real advantage to this — if critical sections of code are expanded for maximum speed, the only possible improvement would come on non-critical code, so the potential for increased speed is small.

3.2 Implementing Specializations

Specialization is the use of a polymorphic function or object in a context where a function or object having fewer type parameters is required. For any representation of polymorphism, some additional work is needed when a specialized function or object is used. Consider a specialization of the square function from Figure 3.2:

```
int apply_to_seven( int (*func)( int ) ) {
  return func( 7 );
}

void test() {
  apply_to_seven( square );
}
```

The function `apply_to_seven` is entirely monomorphic, and yet in this example it calls the polymorphic function square. With a coercive boxing approach, the unboxed `7` must be boxed before being passed, and the return value unboxed. With a type passing approach a type descriptor must be supplied to the square function in addition to its argument. With any form of

representation, the assertion argument (the multiplication operation) must also be supplied to `square`.

One approach is to pass some kind of closure that contains a pointer to a polymorphic function as well as a representation of the type arguments corresponding to some of its type parameters and any assertion arguments inferred as the result of specialization. This auxiliary data structure is filled in at run time as the polymorphic function is specialized. The structure must allow the same function to have more than one type parameter specialized at different times and in different orders, e.g.:

```
forall( type T, type U ) void multi_spec( T, U );
forall( type V ) void (*spec1)( V, int ) = multi_spec;
forall( type W ) void (*spec2)( int, W ) = multi_spec;

void g( void (*)( int, int ) );
g( spec1 );
g( spec2 );
```

Here, two specializations of the function `multi_spec` are created, one binding the first type parameter and one binding the second. Both specializations have their remaining unbound parameter bound as a result of the implicit specialization that occurs when they are passed as arguments to `g`.

When the specialized entity is used, the generated code uses the extra information to determine how to properly call the function. Since a polymorphic function with all type parameters specialized must be indistinguishable from a monomorphic function (consider the application of the `square` function from within `apply_to_seven`), this approach requires closures (and the corresponding overhead on use) for all function types. As a result, this approach adds overhead to all programs, even existing C programs that make no use of polymorphism, which violates one of the primary design goals for the Cforall translator.

3.2.1 Thunk Functions

Another approach is to replace a passed polymorphic function with a thunk function that has the same type as the parameter (so that no specialization is required); the only action performed by

the thunk function is to call the original function and, if necessary, return its result. In a case such as:

```
void f( int (*p)( int, int ) );
forall( type U, type V | { U ?+?( U, V ); } ) U g( U, V ) );

f( g );
```

a suitable thunk would be:

```
int g_thunk( int p1, int p2 ) {
  g( p1, p2 );
}
```

The call `f(g)` is then changed to be `f(g_thunk)`. Note that the type of the thunk is the same as the type of the parameter to `f` (except that `g_thunk` is a function while `p` is a pointer-to-function). As a consequence of this transformation, type parameters in a function are only associated with type arguments at the point where the function is called. The results of this transformation can then be used as input to any of the transformations described earlier in this chapter, which convert calls to polymorphic functions into a monomorphic form, producing an entirely monomorphic program.

For specializations of functions that take polymorphic functions as parameters (and functions that take functions that take polymorphic functions as parameters, and so on), the same method of thunk generation can be used, but the generated thunk may itself involve a specialization.

```
void f( void (*p)( forall( type U ) U (*)( U ) ) );
void g( int (*q)( int ) );

f( g );
```

Here, the thunk function is:

```
void g_thunk( forall( type U ) U (*p)( U ) ) ) {
  g( p );
}
```

The thunk is valid — it has a type that matches the formal parameter of `f` — but the call `g(p)` still involves a specialization, which means another thunk is necessary:

```
int p_thunk( int x ) {
  p( x );
}
```

For a heavily-nested function type, many thunk functions may be necessary. Since the type of the thunk becomes less heavily-nested for each repetition of the process, this repeated thunk creation must eventually terminate.

3.2.2 Generating Thunks

Practical considerations apply to the generation of thunk functions that are similar to those which apply to the generation of adapter functions. However, in the case of thunk functions the object of specialization may involve local variables of the calling function, so in general it is impossible to simply hoist the thunks. For example, the thunk `p_thunk` of the previous section refers to the local variable `p` of the function `g_thunk`, so `p_thunk` must be nested within `g_thunk`. This could be solved by changing the representation of function pointers as previously described, or by using adapters and passing both the adapter and the function to be specialized. Adopting one of these approaches would have the additional advantage of allowing specialization to be applied to an entire array of function pointers. However, both of these approaches impose an additional level of indirection on all *users* of function pointers, including monomorphic functions, so these approaches are unsuitable for the translator. Another option is to use assembly code, as is done in the GNU C compiler's implementation of nested functions [5].

Although the current translator fails to achieve the goal of converting all programs into Standard C, the wide portability of the GNU C compiler indicates that the implementation of nested functions does not significantly impair portability, so even with this limitation it is still possible to implement Cforall compilers for a wide variety of architectures.

3.2.3 A Complete Example

Figure 3.5 shows the previous example in the context of a complete program. `main` invokes the function `f` with the function `g` as argument, `f` invokes the passed in function with the polymorphic function `id` as argument. This call specializes the polymorphic function `id` to the type of `q`. `g` applies the specialized function to the argument `7`.

```
forall( type T ) T id( T t ) {
  return t;
}

void f( void (*p)( forall( type U ) U (*)( U ) ) ) {
  p( id );
}

void g( int (*q)( int ) ) {
  q( 7 );
}

int main() {
  f( g );
}
```

Figure 3.5: Program requiring a nested specialization thunk.

Figure 3.6 shows the result of expanding specializations in Figure 3.5. The program still uses polymorphism, but all specializations have been replaced by thunk functions. `f` does not call `g` directly, but instead through a thunk. The thunk calls `g`, but passes it another thunk instead of the function argument passed by `f`. When this thunk is called from within `g`, it applies the function passed by `f` to its parameter. As a result, `id` is called on the value `7`, so the meaning of the original program is preserved by the transformation.

The conversion of Figure 3.6 into a completely monomorphic form is shown in Figures 3.7 and 3.8. This is the exact output from the current translator, except that name decoration has been suppressed here to make the output easier to read. All occurrences of the polymorphic type

```
forall( type T ) T (*)( T )
```

are replaced by the monomorphic type

```
void (*)(void (*_adapter)(void (*)(), void *, void *, void *),
        long unsigned int T,
        void (*assign)(),
        void *_retparm,
        void *t)
```

```
forall( type T ) T id(T t) {
    return ((T )t);
}

void f(forall( type U ) void (*p)(U (*)(U ))) {
    p(id);
}

void g(int (*q)(int )) {
    q(7);
}

int main() {
    void _thunk0(forall( type U ) U (*_p0)(U )) {
        int _thunk1(int _pp0) {
            return _p0(_pp0);
        }

        g((&_thunk1));
    }

    f((&_thunk0));
}
```

Figure 3.6: Result of expanding specializations in Figure 3.5.

The parameters of this type are used as follows:

- `_adapter` is an adapter function used to call the assignment function for type T
- `T` holds the size of type T
- `assign` is the assignment function for type T
- `_retparm` is the box that holds the return value from the function
- `t` is the box that holds the argument to the function

The translation of function `id` makes use of all of these parameters. It allocates space for the value returned by the assignment operator (`_temp0`) and then applies the assignment operator (using the adapter) to copy the argument value from its box to the box holding the return value for `id` (`_retparm`). Since the adapter places the result of the assignment operator into a box, the generated code replaces the original assignment operation with a comma expression that contains a call to the adapter, and then returns the value of `_temp0`, where the adapter placed the return value from the assignment. In this example, the value returned by the assignment operator is not used.

The function `id` performs as specified, but its interface is different from the one expected by function `g` — `g` does not box the operand to `q`, pass the size of the argument type, or pass the assignment operation. These tasks are handled by the thunk function.

The thunk functions must be defined in function `main` (Figure 3.8), since `main` is the only function that can infer the type argument in the call to `f`, and this type argument is the same one that is eventually passed to `id`. The function `_thunk0` allows a new thunk to be created from within `f` for each application of `f`'s parameter, which is necessary since `main` does not know the functions to which `f` applies its parameter, and `f` does not know the type arguments. When `f` calls `_thunk0`, `_thunk0` calls `g` with a new thunk, `_thunk1`. `_thunk1` allocates space for the return value (`_temp1`) and then calls the function passed by `f`, passing an adapter for the assignment operator, the size of type `int`, the predefined integer assignment function, and the addresses of the return value and argument. Finally, it passes the value returned by the called function back to its caller.

```
void id(void (*_adapter)(void (*)(), void *, void *, void *),
        long unsigned int T,
        void (*assign)(),
        void *_retparm,
        void *t) {
    void *_temp0;
    (_temp0=alloca(T));
    (_adapter(assign, _temp0, _retparm, t) , _temp0);
    return ;
}

void f(void (*p)(void (*)(void (*_adapter)(void (*)(), void *,
                                           void *, void *),
                          long unsigned int U,
                          void (*assign)(),
                          void *,
                          void *))) {
    p(id);
}

void g(int (*q)(int )) {
    q(7);
}
```

Figure 3.7: Result of adding boxing coercions to functions `id`, `f`, and `g` of Figure 3.6.

```
int main() {
    void _thunk0(void (*_p0)(void (*_adapter)(void (*)(), void *,
                                              void *, void *),
                             long unsigned int U,
                             void (*assign)(),
                             void *,
                             void *)) {
        int _thunk1(int _pp0) {
            void _adapter(void (*_adaptee)(), void *_ret,
                          void *_p0, void *_p1) {
                ((*((int *)_ret))
                  =((int (*)(int *, int ))_adaptee)(((int *)_p0),
                                                    (*((int *)_p1))));
            }

            int _temp1;
            return (_p0(_adapter,
                        sizeof(int ),
                        ((void (*)())(&assign_int)),
                        (&_temp1),
                        (&_pp0)),
                    _temp1);
        }

        g((&_thunk1));
    }

    f((&_thunk0));
}
```

Figure 3.8: Result of adding boxing coercions to function `main` of Figure 3.6.

3.2.4 Efficiency

The thunk approach adds the overhead of one polymorphic function call to each call to a function that has undergone specialization. If a function undergoes specialization multiple times before it is used (as with the function `multi_spec`), there is an additional function call for each specialization. Each thunk created for a nested function type also has the overhead of one polymorphic function call, but this overhead is distributed: in the example from the last section, there is an extra call on entry to the function `g`, and an extra call any time that `g` uses its parameter `p`. How expensive these calls are depends on how efficiently polymorphic function calls are implemented; at best, they are no more expensive than a monomorphic function call.

3.3 Related Work

The expansion technique is popular in imperative languages, and for good reason — it is (relatively) simple to implement and very efficient. The semantics of polymorphic functions in C++ [18], known as *template* functions, and of generics in Modula-3 [20], require expansion as their implementation. These languages prohibit by fiat any constructs that cannot be implemented using expansion, including passing polymorphic functions to procedures and polymorphic recursion. Ada [17], while not mandating a particular implementation strategy for generics, does prohibit programs that can not be implemented by expansion. In C++, this strategy has the consequence of effectively precluding the separate compilation of template code.[5]

Boxing has been used successfully in the implementation of many languages, such as Lisp, Scheme, Smalltalk, and ML. It is a natural choice for these languages since it can facilitate garbage collection as well, but it has also contributed to their reputation for poor efficiency. The idea of coercion originates with Leroy [24], who proposed it as a way of improving the performance of monomorphic code in ML. The boxing transformation is more complicated in ML, since it is possible for a function to receive, for instance, a tuple of polymorphic values from which it extracts one; for this reason, the boxing must often be done recursively, although Shao and Appel [32] discuss a way of minimizing the amount of recursive boxing. In Cforall,

[5]The C++ standard does define an “export” qualifier that attempts to allow separate compilation, but the semantics of templates prevent the compiler from doing much compilation without knowing the uses of the function.

any indirect access to polymorphic values must be done through functions, so adapting functions properly provides a complete solution to the problem.

A different approach to boxing that has been used in some imperative programming languages is to define a box representation and then make the actual boxing the responsibility of the programmer. This approach has the advantage of simplifying the implementation and putting the cost of the boxing up-front, with the disadvantage of increasing the burden on the programmer for those types that do not naturally have a boxed representation. It is most advantageous in situations where nearly all types naturally have an appropriate boxed representation, which is the case in many object-oriented languages — typically implementations of these languages use pointers internally to represent all types except a few arithmetic ones. Implementations of this kind have been described for the Oberon language [31] and for the Java language [7]. The Cyclone language [22] adds parametric polymorphism to C, but polymorphic parameters can only have pointer types. The disadvantages of this approach are magnified for C, since it requires the programmer to self-box not only arithmetic types but also structure, union, and enumerated values. A Cforall program with polymorphic functions written in the Cyclone style and compiled using the coercion transformation has no boxing overhead compared to Cyclone, since the arguments corresponding to pointer parameters are never boxed.

Type passing style has been popular as a low-overhead implementation method for polymorphic functional languages [21, 36]. These implementations typically do not have to conform to an established calling convention that favours monomorphic code. Notably, Ziegler [40] describes a type-passing implementation of his Sea language; he does so by imposing a calling sequence for all functions that makes polymorphic calls more efficient than the coercion-based approach described here, but monomorphic calls less efficient than they otherwise could be.

Chapter 4

Translator Examples

This chapter presents an application of Cforall to a simple problem: read in a sequence of integers (with no fixed bound on the size of the input); when input is complete, output the numbers in reverse order. This simple problem provides an opportunity to demonstrate the polymorphic type system of Cforall and the operation of the translator.

The Cforall translator in its current state implements the complete expression analysis algorithm described in Chapter 2. It converts polymorphic specializations into applications using the technique described in Section 3.2, and converts polymorphic functions into a monomorphic form using the coercion technique of Section 3.1.4. It does not support any of the alternative implementations described in Section 3.1 (expansion or type passing).

The names of variables and functions generated by the translator have various decorations attached in order to allow overloading (Chapter 3). Excerpts from the translator output presented in this chapter have these decorations removed to make their presentation clearer. The complete source code for the examples discussed in this chapter, as well as the output from the translator (with decorations), is found in Appendix A.

4.1 Stream Library

Since Cforall supports the C standard library, the file input and output facilities defined in `stdio.h` can be used in any Cforall program. Since these functions are restricted to a fixed set of stream types and make heavy use of the type-unsafe variable-argument mechanism (used in

functions such as `printf` and `scanf`), it is worth considering how the interfaces could be made more flexible and safe in Cforall. The library presented here draws on the C++ streams library [18] for inspiration, but uses polymorphism and contexts where C++ uses class inheritance.

One context definition describes an `ostream`: a type satisfying this context can be used to write character data to a file, and to signal the occurrence of an error on a previous write operation.

```
typedef unsigned long streamsize_type;

context ostream( dtype os_type ) {
  os_type *write( os_type *, const char *, streamsize_type );
  int fail( os_type * );
};
```

Given this context, it is possible to define C++-style << operators that write values of different types to any sort of stream.

```
forall( dtype os_type | ostream( os_type ) )
        os_type * ?<<?( os_type *, char );
forall( dtype os_type | ostream( os_type ) )
        os_type * ?<<?( os_type *, int );
forall( dtype os_type | ostream( os_type ) )
        os_type * ?<<?( os_type *, const char * );
```

Note that the type satisfying context `ostream` is specified to be a `dtype`. This means that instances of `os_type` type can only be passed and returned through pointers, and that the implementations of these functions cannot instantiate new variables of type `os_type`. For an abstract data type these restrictions do not significantly constrain the implementation, and they permit an efficient translation. For instance, the function

```
forall( dtype os_type | ostream( os_type ) )
os_type * ?<<?( os_type *os, char c ) {
  return write( os, &c, 1 );
}
```

is translated into

```
void *_operator_shiftleft(void *(*write)(void *, const char *,
                                         long unsigned int ),
                          int (*fail)(void *),
                          void *os,
                          char c) {
    return write(os, (&c), 1);
}
```

The translated function is identical to the original except that assertion parameters defined in the context `ostream` have become explicit parameters. The function does not require any of the special mechanisms of Section 3.1.4 (type descriptors, boxing, or adapter functions), since it cannot manipulate `os_type` values directly, but only through pointers.

The profusion of << operators suggests another context, one that describes types that can be written to streams.

```
context writeable(type T) {
  forall(dtype os_type | ostream(os_type)) os_type * ?<<?(os_type *, T);
};
```

A type satisfying this context must provide a << operation that can be applied to *any* `ostream` type.

In a similar way, contexts `istream` and `readable` are defined to describe input streams and types that can be read from streams. A small but complete set of definitions of these contexts with << and >> operators is found in Section A.1.1. An implementation that uses the C `stdio.h` functions to provide data types `ofstream` and `ifstream` that conform to the contexts `ostream` and `istream`, respectively, is presented in Section A.1.2.

4.2 Arrays

An *array* is defined here to be a sequence of data elements all having the same type, stored contiguously in memory, where an individual element may be accessed by using an integer representing its position in the array.

```
context array( type array_type, type elt_type ) {
  lvalue elt_type ?[?]( array_type, int );
};
```

This operation overloads the indexing operator, so that for an array `a` and an integer `i`, the expression `a[i]` returns the element of the array at index `i`. The qualifier `lvalue` means that the element of the array designated by the expression can be modified; returning an value with `lvalue` qualification is similar to returning a value by reference in C++.[1] The first element of the array is at index 0.

Defining the context `array` in this way means that, for any object type `T`, the type `T*` can be treated as an array of `T`. This is because the index operation is pre-defined for all pointer types; the pointer points to the first element of the array.

Given a pointer to a C array, there is no way to know the maximum valid index for the array. Define a *bounded* array type to be an array that can provide its maximum index upon request.

```
context bounded_array( type array_type, type elt_type
                     | array( array_type, elt_type ) ) {
  int last( array_type );
};
```

The context `bounded_array` places the same constraints on the array and element types as the `array` context, but also requires that the `last` function exists.

Note that in the contexts `array` and `bounded_array`, the `array_type` and `elt_type` parameters are of kind `type`, meaning that they must be complete types. This specification means that values of these types must be assignable and instantiable, and that the values must be boxed and unboxed.

The structure `vector_int` and its associated functions implements a bounded array of `int` that dynamically resizes itself as elements are added to the end.

```
typedef struct vector_int {
  int last;
  int capacity;
  int *data;
} vector_int;
```

Since `vector_int` is a complete type, it requires an assignment operation. The translator automatically generates an assignment operator for each structure it encounters.

[1] In translation, functions returning `lvalue` types return a pointer to the actual result, which is implicitly dereferenced in the calling function.

```
struct vector_int {
    int last;
    int capacity;
    int *data;
};
static struct vector_int _operator_assign(struct vector_int *_dst,
                                          struct vector_int _src) {
    ((*_dst).last=_src.last);
    ((*_dst).capacity=_src.capacity);
    ((*_dst).data=_src.data);
    return _src;
}
```

The automatically-generated assignment operator invokes assignment for each of the members of the structure. Here, the members have intrinsic types, so the intrinsic assignment operators are used; if the structure contained a member having a user defined type with an overloaded assignment operator, that operator would be invoked here. Note that the copy is a "shallow copy", in that the pointer to the vector's data is copied but not the vector itself — the destination vector shares the data of the source vector. If a complete copy of the vector is desired, the assignment operator can be redefined by the user to give this behaviour.

The remaining code for `vector_int` is found in Section A.2.2. Aside from the overloaded functions `?[?]` and `last` that implement the context `bound_array`, `vector_int` is implemented in pure C.

4.3 Iterators

When dealing with "collection" data structures such as arrays, lists, and sets, it is often useful to write functions that process elements of a collection one at a time, regardless of the specific type of the container. This inspiration gives rise to the iterator concept.

```
context iterator( type iterator_type, type elt_type ) {
  iterator_type ++?( iterator_type* );
  iterator_type --?( iterator_type* );
  int ?==?( iterator_type, iterator_type );
  int ?!=?( iterator_type, iterator_type );
  lvalue elt_type *?( iterator_type );
};
```

An iterator is a generic "pointer" that references some element within a collection. The iterator can be adjusted to point at the "next" element (however "next" is defined for the particular collection) using the `++` operator, or at the "previous" element using the `--` operator. The `==` and `!=` operators allow for equality comparisons among iterators (two iterators are equal if they reference the same element of the same collection). The `*` operator provides the ability to read and change the element referenced by the iterator.

Typically, iterator algorithms operate over a range of elements. This chapter adopts similar conventions for iterator ranges as the C++ standard library: given two iterators `i1` and `i2`, the range of elements defined by those iterators includes the element referenced by `i1` and all successors of that element up to but not including the element referenced by `i2`. A collection has two special iterators, `begin` and `end`: `begin` references the first element in the collection, and `end` references an imaginary element that is located after the last element in the collection.

Since arrays are stored in contiguous memory, it is possible to use pointers as iterators. Pointers have all of the `iterator` operations defined for them automatically. For a `bounded_array`, it is possible to write generic functions that return the begin and end iterators:

```
forall( type array_type, type elt_type
        | bounded_array( array_type, elt_type ) )
elt_type *begin( array_type array ) {
  return &array[ 0 ];
}

forall( type array_type, type elt_type
        | bounded_array( array_type, elt_type ) )
elt_type *end( array_type array ) {
  return &array[ last( array ) ] + 1;
}
```

These functions return, respectively, a pointer to the first element of the array and a pointer to one past the last element. Since these functions are polymorphic, they require adapters and boxing. For example, the `end` function becomes:

```
void *end(void *(*_adapter1)(void (*)(), void *, int ),
          int (*_adapter2)(void (*)(), void *),
          void (*_adapter3)(void (*)(), void *, void *, void *),
          void (*_adapter4)(void (*)(), void *, void *, void *),
          long unsigned int array_type,
          long unsigned int elt_type,
          void (*_operator_assign_array_type)(),
          void (*_operator_assign_elt_type)(),
          void (*last)(),
          void (*_operator_index)(),
          void *array) {
    return (_adapter1(_operator_index, array, _adapter2(last, array))
            +(1*elt_type));
}
```

The parameters to this function are:

- an adapter for `_operator_index`
- an adapter for `last`
- an adapter for `_operator_assign_array_type`
- an adapter for `_operator_assign_elt_type`
- the size of an instance of `array_type`
- the size of an instance of `elt_type`
- the assignment operator for `array_type`
- the assignment operator for `elt_type`
- the `last` function from context `bounded_array`

- the index operator from context `array`
- the array itself (boxed)

The evaluation of the function first calls the `last` function through an adapter; the adapter unboxes the array, calls `last`, and returns the integer result. This result is then used in the call to the index operator, which once again unboxes the array and performs the indexing operation. The result of the index is a `lvalue elt_type`, which (as previously described) is returned as a pointer, so no further boxing is necessary. Finally, the returned pointer must be incremented to point at the next array element; normally, adding one to a pointer has this effect, but here the size of the data elements is only known at run-time, so it is necessary to specify the size of the increment explicitly.

Since `vector_int` implements the context `bounded_array`, these functions can be used to get the beginning and ending iterators for an object of type `vector_int`.

Using iterators, it is possible to write generic algorithms that can operate over any kind of collection using iterators:

```
forall( type elt_type | writeable( elt_type ),
        type iterator_type | iterator( iterator_type, elt_type ),
        dtype os_type | ostream( os_type ) )
void
write_reverse( iterator_type begin, iterator_type end, os_type *os ) {
  iterator_type i = end;
  do {
    --i;
    os << *i << ' ';
  } while( i != begin );
}
```

In the output from the translator, this seemingly-simple function is translated into a function with 21 parameters; these convey:

- six adapters
- the sizes of types `elt_type` and `iterator_type`
- assignment operators for `elt_type` and `iterator_type`

- the << operator of context `writeable`
- the five operators of context `iterator`
- the two functions of context `ostream`
- the three explicit parameters to the function

Note in particular that the << operation, coming from context `writeable`, is a polymorphic function passed as a parameter (as described in Section 2.1.3):

```
forall( dtype os_type | ostream( os_type ) ) os_type * ?<<?( os_type *, T );
```

Since one of its parameters is a pointer to a polymorphic type while the other parameter is monomorphic (the type of the value to be written), the adapter for the << operator must unbox the monomorphic parameter while passing along the `os_type` pointer and the `ostream` operations. For the function

```
forall(dtype os_type | ostream(os_type)) os_type * ?<<?(os_type *, int);
```

an appropriate adapter is

```
void *_adapter(void *(*write)(void *, const char *, long unsigned int ),
               int (*fail)(void *),
               void (*_adaptee)(),
               void *_p0,
               void *_p1) {
    return ((void *(*)(void *(*)(void *, const char *, long unsigned int ),
                       int (*)(void *), void *, int ))_adaptee)
                   (write, fail, _p0, (*((int *)_p1)));
}
```

Here, the function to be adapted (`_adaptee`) is cast to the proper type (accepting four arguments, the two `ostream` operations and the two explicit parameters) before being applied to the set of parameters passed in (where the last of the parameters is unboxed).

The complete translator output for the `write_reverse` function begins on page 109.

```
int main() {
  ofstream *sout = ofstream_stdout();
  ifstream *sin = ifstream_stdin();
  vector_int vec = vector_int_allocate();

  // read in numbers until EOF or error
  int num;
  for(;;) {
    sin >> &num;
    if( fail( sin ) || eof( sin ) ) break;
    append( &vec, num );
  }

  // write out the numbers
  sout << "Array elements: ";
  write_reverse( begin( vec ), end( vec ), sout );
  sout << "\n";
  return 0;
}
```

Figure 4.1: Program to read in numbers and write them out

4.4 Test Program

Figure 4.1 shows a program that solves the problem posed at the beginning of the chapter, using the data structures and subroutines presented. The first block of statements allocates input and output streams corresponding to standard input and standard output, respectively, and allocates a vector to store the numbers that are read. The next block reads numbers and adds them to the end of the vector until no more remain to be read. The final block uses the `write_reverse` function to write the numbers.

Most of the complexity of the translated output centres around the single line

```
write_reverse( begin( vec ), end( vec ), sout );
```

The complete translation of this statement produces three pages of output (this output is found beginning on page 113). The translation must accomplish a number of tasks:

- The functions `begin` and `end` are polymorphic in the array type (`vector_int`) and the element type (`int`), so the input array (`vec`) is be boxed and the appropriate array operations are be passed, along with adapters. Since `begin` and `end` have the same assertions and type arguments, the adapters generated for one are re-used for the other. The results of `begin` and `end` are assigned to temporaries so that they can be boxed to be passed to `write_reverse`.

- The function `write_reverse` is polymorphic in the iterator type (`int*`), so the five operations of context `iterator` must be passed. The operations are pre-defined polymorphic functions defined over all pointer types; for instance, the equality function is prototyped as:

  ```
  forall(dtype DT) int ?==?(DT*, DT*);
  ```

  Since the equality operation required by context `iterator` is not polymorphic, a specialization is necessary (Section 3.2). The specialization thunk created in this case is:

  ```
  int _thunk3(int *_p0, int *_p1) {
      return _operator_equal_ptr(_p0, _p1);
  }
  ```

  As an additional operation, the translator recognizes that the operation invoked by the thunk simply implements the intrinsic pointer equality operation, so it expands the operation in-line:

  ```
  int _thunk3(int *_p0, int *_p1) {
      return (_p0==_p1);
  }
  ```

  Similar thunks are created for the other four operations of context `iterator`, as well as for the pointer assignment operation. Adapters are created as well, so that the `int*` iterators are properly unboxed before being passed to the thunks. In addition, the size of the iterator type is passed.

- The function `write_reverse` is also polymorphic in the type of element returned by the iterator (`int`, in this case), which must support the polymorphic `<<` operation of context

`writeable`. This operation must be passed, along with an adapter (code for the adapter is presented in the previous section). In addition, the size of the element type and an assignment operation are passed.

- The function `write_reverse` is also polymorphic in the type of the output stream (in this case, `ofstream`). Since the stream type is a `dtype`, no extra adapters, assignment operations, or type sizes need to be passed, but the two operations of context `ostream` must be passed.

The complete output of the translator for the above program is found starting on page 112.

Chapter 5

Conclusions

This thesis describes the basic algorithms and data representations involved in implementing a compilation system for Cforall. This implementation is constrained by the need to support traditional C code in a way that preserves its original meaning and efficiency, while still providing efficient and unrestricted use of the new language features. This constraint has resulted in an implementation that is in some ways quite different from implementations of comparable languages.

5.1 Expression Analysis

The expression analysis algorithm essentially adapts the algorithm of Cormack and Wright [9], with extensions to allow for implicit conversions, the use of contexts, and the composition of functions returning multiple values. While the problem is solvable, some questions remain about the overall efficiency of the algorithm. Dealing with functions returning multiple values adds an exponential factor to the complexity of the algorithm, but it remains to be seen what effect this theoretical complexity has on practical efficiency. The implementation of the translator must be optimized and empirical data must be gathered from experimentation with the translator in order to determine what the real characteristics of the algorithm are.

5.2 Code Generation

This thesis proposes converting polymorphic functions into a monomorphic form using a hybrid approach of boxing and expansion, while at the same time conceding that the choice is controversial. The choice is strongly influenced by the goals of the translation: to preserve the performance of monomorphic programs while at the same time supporting polymorphism completely and uniformly. Typically, language implementations trade off one of these factors against the other to improve the performance of polymorphic code: sacrificing performance on monomorphic code, or restricting the use of polymorphism. It seems that combining both expansion and coercion makes this trade-off unnecessary, but the proof of this will be in the performance of real Cforall programs.

5.3 Future Work

The outstanding questions related to this work are performance issues, both of the translator itself and of the generated code. In order to assess performance, and to quantify the effect of any improvements on the system, there must be a significant body of real code to use in experimentation. Since Cforall is a (near) superset of C, it is natural to experiment on the vast collection of freely-available C programs. These programs are insufficient, however, since they do not make use of overloading, polymorphism, or multiple return values. For this reason, new programs must be written or old programs adapted to use the new features to provide a basis for experimentation with the translator. At this point, it is not even clear how these features will be used to write practical applications — new programming idioms will undoubtedly evolve, and supporting these idioms efficiently may motivate the evolution of the compilation system.

Also, the limitations imposed by the need to generate C code should be considered. It may be possible to improve the performance of the generated code by using a full compiler (or by integrating a Cforall front end with an existing compiler back end).

Appendix A

Source Code for Examples

This appendix contains the complete source for the example Cforall program described in Chapter 4, as well as the C source code that is generated by the Cforall translator for each module. The translator output has been annotated with comments showing the original Cforall code in bold-face. The output of the translator also contains the contents of any included header files and a set of prototypes for functions that implement the C intrinsic operations; these have been edited out of the output to conserve space.

A.1 Stream Library

A.1.1 iostream

iostream.h

```
#ifndef IOSTREAM_H
#define IOSTREAM_H

typedef unsigned long streamsize_type;

// an ostream is any place to which we can write charaters

context ostream( dtype os_type )
{
```

```
  // send some characters to the stream
  os_type *write( os_type *, const char *, streamsize_type );

  // returns 1 if an error has occurred, 0 otherwise
  int fail( os_type * );
};

// the "<<" operation represents writing an object to a stream
// a type is writeable if it supports this operation

context writeable( type T )
{
  forall( dtype os_type | ostream( os_type ) )
    os_type * ?<<?( os_type *, T );
};

// implement writable for some intrinsic types

forall( dtype os_type | ostream( os_type ) )
  os_type * ?<<?( os_type *, char );
forall( dtype os_type | ostream( os_type ) )
  os_type * ?<<?( os_type *, int );
forall( dtype os_type | ostream( os_type ) )
  os_type * ?<<?( os_type *, const char * );


// an istream is any place from which we can read charaters

context istream( dtype is_type )
{
  // read some characters from the stream
  is_type *read( is_type *, char *, streamsize_type );

  // put a character back
  is_type *unread( is_type *, char );

  // returns 1 if an error has occurred, 0 otherwise
  int fail( is_type * );
```

```
  // returns 1 if the last read encountered the end of file, 0 otherwise
  int eof( is_type * );
};

// the ">>" operation represents reading an object from a stream
// a type is readable if it supports this operation

context readable( type T )
{
  forall( dtype is_type | istream( is_type ) )
    is_type * ?<<?( is_type *, T* );
};

// implement readable for some intrinsic types

forall( dtype is_type | istream( is_type ) )
  is_type * ?>>?( is_type *, char* );
forall( dtype is_type | istream( is_type ) )
  is_type * ?>>?( is_type *, int* );

#endif /* #ifndef IOSTREAM_H */
```

iostream.c

```
#include "iostream.h"
#include <stdio.h>
#include <string.h>

// these functions implement writable and readable for some intrinsic types

forall( dtype os_type | ostream( os_type ) )
os_type *
?<<?( os_type *os, char c )
{
  return write( os, &c, 1 );
}
```

```
forall( dtype os_type | ostream( os_type ) )
os_type *
?<<?( os_type *os, int i )
{
  char buffer[20];      // can hold any integer
  sprintf( buffer, "%d", i );
  return write( os, buffer, strlen( buffer ) );
}

forall( dtype os_type | ostream( os_type ) )
os_type *
?<<?( os_type *os, const char *cp )
{
  return write( os, cp, strlen( cp ) );
}

forall( dtype is_type | istream( is_type ) )
is_type *
?>>?( is_type *is, char *cp )
{
  return read( is, cp, 1 );
}

forall( dtype is_type | istream( is_type ) )
is_type *
?>>?( is_type *is, int *ip )
{
  char cur;

  // skip some non-digits
  do {
    is >> &cur;
    if( fail( is ) || eof( is ) ) return is;
  } while( !( cur >= '0' && cur <= '9' ) );

  // accumulate digits
  *ip = 0;
  while( cur >= '0' && cur <= '9' ) {
```

```
    *ip = *ip * 10 + ( cur - '0' );
    is >> &cur;
    if( fail( is ) || eof( is ) ) return is;
  }

  // push back the last non digit
  unread( is, cur );
  return is;
}
```

Translator Output

```
// forall( dtype os_type | ostream( os_type ) )
// os_type *
// ?<<?( os_type *os, char c )
void *___operator_shiftleft__A0_1_0___write__PFP2d0_P2d0PCcUl___fail__PFi_P2d0__
FP2d0_P2d0c_(void *(*__write__PFP8tos_type_P8tos_typePCcUl_)(void *, const char
*, long unsigned int ), int (*__fail__PFi_P8tos_type_)(void *), void *__os__P8to
s_type, char __c__c)
{
    // return write( os, &c, 1 );
    return __write__PFP8tos_type_P8tos_typePCcUl_(__os__P8tos_type, (&__c__c), 1
);
}

// forall( dtype os_type | ostream( os_type ) )
// os_type *
// ?<<?( os_type *os, int i )
void *___operator_shiftleft__A0_1_0___write__PFP2d0_P2d0PCcUl___fail__PFi_P2d0__
FP2d0_P2d0i_(void *(*__write__PFP8tos_type_P8tos_typePCcUl_)(void *, const char
*, long unsigned int ), int (*__fail__PFi_P8tos_type_)(void *), void *__os__P8to
s_type, int __i__i)
{
    // char buffer[20];
    char __buffer__A0c[20];
    // sprintf( buffer, "%d", i );
    sprintf(__buffer__A0c, "%d", __i__i);
    // return write( os, buffer, strlen( buffer ) );
```

```
    return __write__PFP8tos_type_P8tos_typePCcUl_(__os__P8tos_type, __buffer__A0
c, strlen(__buffer__A0c));
}

// forall( dtype os_type | ostream( os_type ) )
// os_type *
// ?<<?( os_type *os, const char *cp )
void *___operator_shiftleft__A0_1_0___write__PFP2d0_P2d0PCcUl___fail__PFi_P2d0__
FP2d0_P2d0PCc_(void *(*__write__PFP8tos_type_P8tos_typePCcUl_)(void *, const cha
r *, long unsigned int ), int (*__fail__PFi_P8tos_type_)(void *), void *__os__P8
tos_type, const char *__cp__PCc)
{
    // return write( os, cp, strlen( cp ) );
    return __write__PFP8tos_type_P8tos_typePCcUl_(__os__P8tos_type, __cp__PCc, s
trlen(__cp__PCc));
}

// forall( dtype is_type | istream( is_type ) )
// is_type *
// ?>>?( is_type *is, char *cp )
void *___operator_shiftright__A0_1_0___read__PFP2d0_P2d0PcUl___unread__PFP2d0_P2
d0c___fail__PFi_P2d0___eof__PFi_P2d0__FP2d0_P2d0Pc_(void *(*__read__PFP8tis_type
_P8tis_typePcUl_)(void *, char *, long unsigned int ), void *(*__unread__PFP8tis
_type_P8tis_typec_)(void *, char ), int (*__fail__PFi_P8tis_type_)(void *), int
(*__eof__PFi_P8tis_type_)(void *), void *__is__P8tis_type, char *__cp__Pc)
{
    // return read( is, cp, 1 );
    return __read__PFP8tis_type_P8tis_typePcUl_(__is__P8tis_type, __cp__Pc, 1);
}

// forall( dtype is_type | istream( is_type ) )
// is_type *
// ?>>?( is_type *is, int *ip )
void *___operator_shiftright__A0_1_0___read__PFP2d0_P2d0PcUl___unread__PFP2d0_P2
d0c___fail__PFi_P2d0___eof__PFi_P2d0__FP2d0_P2d0Pi_(void *(*__read__PFP8tis_type
_P8tis_typePcUl_)(void *, char *, long unsigned int ), void *(*__unread__PFP8tis
_type_P8tis_typec_)(void *, char ), int (*__fail__PFi_P8tis_type_)(void *), int
(*__eof__PFi_P8tis_type_)(void *), void *__is__P8tis_type, int *__ip__Pi)
```

```
{
    // char cur;
    char __cur__c;
    // do {
    do{

    {
        // is >> &cur;
        ___operator_shiftright__A0_1_0___read__PFP2d0_P2d0PcUl___unread__PFP2d0_
P2d0c___fail__PFi_P2d0___eof__PFi_P2d0__FP2d0_P2d0Pc_(__read__PFP8tis_type_P8tis
_typePcUl_, __unread__PFP8tis_type_P8tis_typec_, __fail__PFi_P8tis_type_, __eof_
_PFi_P8tis_type_, __is__P8tis_type, (&__cur__c));
        // if( fail( is ) || eof( is ) ) return is;
        if (((int )((((int )(__fail__PFi_P8tis_type_(__is__P8tis_type)!=0)) || (
(int )(__eof__PFi_P8tis_type_(__is__P8tis_type)!=0)))!=0)))
            return __is__P8tis_type;

    // } while( !( cur >= '0' && cur <= '9' ) );
    }
    } while(((int )((!(((int )((__cur__c>='0')!=0)) && ((int )((__cur__c<='9')!=
0))))!=0)));

    // *ip = 0;
    ((*__ip__Pi)=0);
    // while( cur >= '0' && cur <= '9' ) {
    while(((int )((((int )((__cur__c>='0')!=0)) && ((int )((__cur__c<='9')!=0)))
!=0))){

    {
        // *ip = *ip * 10 + ( cur - '0' );
        ((*__ip__Pi)=(((*__ip__Pi)*10)+(__cur__c-'0')));
        // is >> &cur;
        ___operator_shiftright__A0_1_0___read__PFP2d0_P2d0PcUl___unread__PFP2d0_
P2d0c___fail__PFi_P2d0___eof__PFi_P2d0__FP2d0_P2d0Pc_(__read__PFP8tis_type_P8tis
_typePcUl_, __unread__PFP8tis_type_P8tis_typec_, __fail__PFi_P8tis_type_, __eof_
_PFi_P8tis_type_, __is__P8tis_type, (&__cur__c));
        // if( fail( is ) || eof( is ) ) return is;
        if (((int )((((int )(__fail__PFi_P8tis_type_(__is__P8tis_type)!=0)) || (
```

```
(int )(__eof__PFi_P8tis_type_(__is__P8tis_type)!=0)))!=0)))
            return __is__P8tis_type;

    }
    }

    // unread( is, cur );
    __unread__PFP8tis_type_P8tis_typec_(__is__P8tis_type, __cur__c);
    // return is;
    return __is__P8tis_type;
}
```

A.1.2 fstream

fstream.h

```
#ifndef FSTREAM_H
#define FSTREAM_H

#include "iostream.h"

typedef struct ofstream ofstream;

// implement context ostream
ofstream *write( ofstream *, const char *, streamsize_type );
int fail( ofstream * );

// stream corresponding to standard output
ofstream *ofstream_stdout();

typedef struct ifstream ifstream;

// implement context istream
ifstream *read( ifstream *, char *, streamsize_type );
ifstream *unread( ifstream *, char );
int fail( ifstream * );
int eof( ifstream * );
```

```
// stream corresponding to standard input
ifstream *ifstream_stdin();

#endif /* #ifndef FSTREAM_H */
```

fstream.c

```
#include "fstream.h"
#include <stdio.h>
#include <stdlib.h>

// ofstream and ifstream are simply a wrapper around ANSI stdio

struct ofstream
{
  FILE *file;
  int fail;
};

ofstream *
write( ofstream *os, const char *data, streamsize_type size )
{
  if( !os->fail ) {
    fwrite( data, size, 1, os->file );
    os->fail = ferror( os->file );
  }
  return os;
}

int
fail( ofstream *os )
{
  return os->fail;
}

static ofstream*
make_ofstream()
```

```
{
  ofstream *new_stream = malloc( sizeof( ofstream ) );
  new_stream->fail = 0;
  return new_stream;
}

ofstream *
ofstream_stdout()
{
  ofstream *stdout_stream = make_ofstream();
  stdout_stream->file = stdout;
  return stdout_stream;
}

struct ifstream
{
  FILE *file;
  int fail;
  int eof;
};

ifstream *
read( ifstream *is, char *data, streamsize_type size )
{
  if( !is->fail && !is->eof ) {
    fread( data, size, 1, is->file );
    is->fail = ferror( is->file );
    is->eof = feof( is->file );
  }
  return is;
}

ifstream *unread( ifstream *is, char c )
{
  if( !is->fail ) {
    if( EOF == ungetc( c, is->file ) ) {
      is->fail = 1;
    }
```

```
  }
  return is;
}

int fail( ifstream *is )
{
  return is->fail;
}

int eof( ifstream *is )
{
  return is->eof;
}

static ifstream*
make_ifstream()
{
  ifstream *new_stream = malloc( sizeof( ifstream ) );
  new_stream->fail = 0;
  new_stream->eof = 0;
  return new_stream;
}

ifstream *ifstream_stdin()
{
  ifstream *stdin_stream = make_ifstream();
  stdin_stream->file = stdin;
  return stdin_stream;
}
```

Translator Output

```
// struct ofstream
// {
//   FILE *file;
//   int fail;
// };
struct ofstream
```

```
{
    struct __FILE *__file__P7s__FILE;
    int __fail__i;
};
static struct ofstream ___operator_assign__F9sofstream_P9sofstream9sofstream_(st
ruct ofstream *___dst__P9sofstream, struct ofstream ___src__9sofstream)
{
    ((*___dst__P9sofstream).__file__P7s__FILE=___src__9sofstream.__file__P7s__FI
LE);
    ((*___dst__P9sofstream).__fail__i=___src__9sofstream.__fail__i);
    return ___src__9sofstream;
}

// ofstream *
// write( ofstream *os, const char *data, streamsize_type size )
struct ofstream *__write__FP9sofstream_P9sofstreamPCcUl_(struct ofstream *__os__
P9sofstream, const char *__data__PCc, long unsigned int __size__Ul)
{
    // if( !os->fail ) {
    if (((int )((!(*__os__P9sofstream).__fail__i)!=0)))

        {
            // fwrite( data, size, 1, os->file );
            fwrite(__data__PCc, __size__Ul, 1, (*__os__P9sofstream).__file__P7s_
_FILE);
            // os->fail = ferror( os->file );
            ((*__os__P9sofstream).__fail__i=((*(*__os__P9sofstream).__file__P7s_
_FILE)._flag&0040));
        }


    // return os;
    return __os__P9sofstream;
}

// int
// fail( ofstream *os )
int __fail__Fi_P9sofstream_(struct ofstream *__os__P9sofstream)
```

```
{
    // return os->fail;
    return (*__os__P9sofstream).__fail__i;
}

// static ofstream*
// make_ofstream()
static struct ofstream *__make_ofstream__FP9sofstream__()
{
    // ofstream *new_stream = malloc( sizeof( ofstream ) );
    struct ofstream *__new_stream__P9sofstream;
    (__new_stream__P9sofstream=malloc(sizeof(struct ofstream )));
    // new_stream->fail = 0;
    ((*__new_stream__P9sofstream).__fail__i=0);
    // return new_stream;
    return __new_stream__P9sofstream;
}

// ofstream *
// ofstream_stdout()
struct ofstream *__ofstream_stdout__FP9sofstream__()
{
    // ofstream *stdout_stream = make_ofstream();
    struct ofstream *__stdout_stream__P9sofstream;
    (__stdout_stream__P9sofstream=__make_ofstream__FP9sofstream__());
    // stdout_stream->file = stdout;
    ((*__stdout_stream__P9sofstream).__file__P7s__FILE=(&__iob[1]));
    // return stdout_stream;
    return __stdout_stream__P9sofstream;
}

// struct ifstream
// {
//   FILE *file;
//   int fail;
//   int eof;
// };
struct ifstream
```

```
{
    struct __FILE *__file__P7s__FILE;
    int __fail__i;
    int __eof__i;
};
static struct ifstream ___operator_assign__F9sifstream_P9sifstream9sifstream_(st
ruct ifstream *___dst__P9sifstream, struct ifstream ___src__9sifstream)
{
    ((*___dst__P9sifstream).__file__P7s__FILE=___src__9sifstream.__file__P7s__FI
LE);
    ((*___dst__P9sifstream).__fail__i=___src__9sifstream.__fail__i);
    ((*___dst__P9sifstream).__eof__i=___src__9sifstream.__eof__i);
    return ___src__9sifstream;
}

// ifstream *
// read( ifstream *is, char *data, streamsize_type size )
struct ifstream *__read__FP9sifstream_P9sifstreamPcUl_(struct ifstream *__is__P9
sifstream, char *__data__Pc, long unsigned int __size__Ul)
{
    // if( !is->fail && !is->eof ) {
    if (((int )((((int )((!(*__is__P9sifstream).__fail__i)!=0)) && ((int )((!(*_
_is__P9sifstream).__eof__i)!=0)))!=0)))

        {
            // fread( data, size, 1, is->file );
            fread(__data__Pc, __size__Ul, 1, (*__is__P9sifstream).__file__P7s__F
ILE);
            // is->fail = ferror( is->file );
            ((*__is__P9sifstream).__fail__i=((*(*__is__P9sifstream).__file__P7s_
_FILE)._flag&0040));
            // is->eof = feof( is->file );
            ((*__is__P9sifstream).__eof__i=((*(*__is__P9sifstream).__file__P7s__
FILE)._flag&0020));
        }


    // return is;
```

```
    return __is__P9sifstream;
}

// ifstream *unread( ifstream *is, char c )
struct ifstream *__unread__FP9sifstream_P9sifstreamc_(struct ifstream *__is__P9s
ifstream, char __c__c)
{
    // if( !is->fail ) {
    if (((int )((!(*__is__P9sifstream).__fail__i)!=0)))

        {
            // if( EOF == ungetc( c, is->file ) ) {
            if (((int )(((-1)==ungetc(__c__c, (*__is__P9sifstream).__file__P7s__
FILE))!=0)))

                {
                    // is->fail = 1;
                    ((*__is__P9sifstream).__fail__i=1);
                }


        }


    // return is;
    return __is__P9sifstream;
}

// int fail( ifstream *is )
int __fail__Fi_P9sifstream_(struct ifstream *__is__P9sifstream)
{
    // return is->fail;
    return (*__is__P9sifstream).__fail__i;
}

// int eof( ifstream *is )
int __eof__Fi_P9sifstream_(struct ifstream *__is__P9sifstream)
{
```

```
    // return is->eof;
    return (*__is__P9sifstream).__eof__i;
}

// static ifstream*
// make_ifstream()
static struct ifstream *__make_ifstream__FP9sifstream__()
{
    // ifstream *new_stream = malloc( sizeof( ifstream ) );
    struct ifstream *__new_stream__P9sifstream;
    (__new_stream__P9sifstream=malloc(sizeof(struct ifstream )));
    // new_stream->fail = 0;
    ((*__new_stream__P9sifstream).__fail__i=0);
    // new_stream->eof = 0;
    ((*__new_stream__P9sifstream).__eof__i=0);
    // return new_stream;
    return __new_stream__P9sifstream;
}

// ifstream *ifstream_stdin()
struct ifstream *__ifstream_stdin__FP9sifstream__()
{
    // ifstream *stdin_stream = make_ifstream();
    struct ifstream *__stdin_stream__P9sifstream;
    (__stdin_stream__P9sifstream=__make_ifstream__FP9sifstream__());
    // stdin_stream->file = stdin;
    ((*__stdin_stream__P9sifstream).__file__P7s__FILE=(&__iob[0]));
    // return stdin_stream;
    return __stdin_stream__P9sifstream;
}
```

A.2 Arrays

A.2.1 array

array.h

```
#ifndef ARRAY_H
#define ARRAY_H

// An array has contiguous elements accessible in any order using integer
// indicies.  The first element has index 0.
context array( type array_type, type elt_type )
{
  lvalue elt_type ?[?]( array_type, int );
};

// a bounded array is an array that carries its maximum index with it
context bounded_array( type array_type, type elt_type
                       | array( array_type, elt_type ) )
{
  int last( array_type );
};

// A bounded array can be iterated over by using a pointer to the element
// type.  These functions return iterators corresponding to the first
// element and the one-past-the-end element, STL-style.
forall( type array_type, type elt_type
        | bounded_array( array_type, elt_type ) )
elt_type *begin( array_type );
forall( type array_type, type elt_type
        | bounded_array( array_type, elt_type ) )
elt_type *end( array_type );

#endif /* #ifndef ARRAY_H */
```

array.c

```
#include "array.h"

// the first element is always at index 0
forall( type array_type, type elt_type
        | bounded_array( array_type, elt_type ) )
elt_type *
begin( array_type array )
{
  return &array[ 0 ];
}

// the end iterator should point one past the last element
forall( type array_type, type elt_type
        | bounded_array( array_type, elt_type ) )
elt_type *
end( array_type array )
{
  return &array[ last( array ) ] + 1;
}
```

Translator Output

```
// forall( type array_type, type elt_type
//         | bounded_array( array_type, elt_type ) )
// elt_type *
// begin( array_type array )
void *__begin__A2_0_0____operator_assign__PF2t0_P2t02t0____operator_assign__PF2t
1_P2t12t1___last__PFi_2t0____operator_index__PFL2t1_2t0i__FP2t1_2t0_(void *(*_ad
apterFP9telt_type_11tarray_typei_)(void (*)(), void *, int ), int (*_adapterFi_1
1tarray_type_)(void (*)(), void *), void (*_adapterF9telt_type_P9telt_type9telt_
type_)(void (*)(), void *, void *, void *), void (*_adapterF11tarray_type_P11tar
ray_type11tarray_type_)(void (*)(), void *, void *, void *), long unsigned int a
rray_type, long unsigned int elt_type, void (*___operator_assign__PF11tarray_typ
e_P11tarray_type11tarray_type_)(), void (*___operator_assign__PF9telt_type_P9tel
t_type9telt_type_)(), void (*__last__PFi_11tarray_type_)(), void (*___operator_i
```

```
ndex__PFL9telt_type_11tarray_typei_)(), void *__array__11tarray_type)
{
    // return &array[ 0 ];
    return _adapterFP9telt_type_11tarray_typei_(___operator_index__PFL9telt_type
_11tarray_typei_, __array__11tarray_type, 0);
}

// forall( type array_type, type elt_type
//         | bounded_array( array_type, elt_type ) )
// elt_type *
// end( array_type array )
void *__end__A2_0_0____operator_assign__PF2t0_P2t02t0____operator_assign__PF2t1_
P2t12t1___last__PFi_2t0____operator_index__PFL2t1_2t0i__FP2t1_2t0_(void *(*_adap
terFP9telt_type_11tarray_typei_)(void (*)(), void *, int ), int (*_adapterFi_11t
array_type_)(void (*)(), void *), void (*_adapterF9telt_type_P9telt_type9telt_ty
pe_)(void (*)(), void *, void *, void *), void (*_adapterF11tarray_type_P11tarra
y_type11tarray_type_)(void (*)(), void *, void *, void *), long unsigned int arr
ay_type, long unsigned int elt_type, void (*___operator_assign__PF11tarray_type_
P11tarray_type11tarray_type_)(), void (*___operator_assign__PF9telt_type_P9telt_
type9telt_type_)(), void (*__last__PFi_11tarray_type_)(), void (*___operator_ind
ex__PFL9telt_type_11tarray_typei_)(), void *__array__11tarray_type)
{
    // return &array[ last( array ) ] + 1;
    return (_adapterFP9telt_type_11tarray_typei_(___operator_index__PFL9telt_typ
e_11tarray_typei_, __array__11tarray_type, _adapterFi_11tarray_type_(__last__PFi
_11tarray_type_, __array__11tarray_type))+(1*elt_type));
}
```

A.2.2 vector_int

vector_int.h

```
#ifndef VECTOR_INT_H
#define VECTOR_INT_H

// a "flexible array", similar to a C++ vector, that holds integers
// and can be resized dynamically
```

```
typedef struct vector_int
{
  // the last used index
  int last;

  // the last possible index before reallocation is necessary
  int capacity;

  // the array itself
  int *data;
} vector_int;

// allocate a vector with default capacity
vector_int vector_int_allocate();

// allocate a vector with specified capacity
vector_int vector_int_allocate( int reserve );

// deallocate the vector's storage
void vector_int_deallocate( vector_int );

// reserve more capacity
void reserve( vector_int *vec, int reserve );

// add an element to the end of the vector, resizing as necessary
void append( vector_int *vec, int element );

// implement bounded_array (array.h)

// access to an arbitrary element (will not resize)
lvalue int ?[?]( vector_int vec, int index );

// return the current last element
int last( vector_int vec );

#endif /* #ifndef VECTOR_INT_H */
```

vector_int.c

```
#include "vector_int.h"
#include <stdlib.h>

#define DEFAULT_CAPACITY 20

vector_int
vector_int_allocate()
{
  return vector_int_allocate( DEFAULT_CAPACITY );
}

vector_int
vector_int_allocate( int reserve )
{
  vector_int new_vector;
  new_vector.last = -1;
  new_vector.capacity = reserve;
  new_vector.data = malloc( sizeof( int ) * reserve );
  return new_vector;
}

void
vector_int_deallocate( vector_int vec )
{
  free( vec.data );
}

void append( vector_int *vec, int element )
{
  vec->last++;
  if( vec->last == vec->capacity ) {
    vec->capacity *= 2;
    vec->data = realloc( vec->data, sizeof( int ) * vec->capacity );
  }
  vec->data[ vec->last ] = element;
}
```

```
// implement bounded_array

lvalue int
?[?]( vector_int vec, int index )
{
  return vec.data[ index ];
}

int
last( vector_int vec )
{
  return vec.last;
}
```

Translator Output

```
// typedef struct vector_int
// {
//   int last;
//   int capacity;
//   int *data;
// } vector_int;
struct vector_int
{
    int __last__i;
    int __capacity__i;
    int *__data__Pi;
};
static struct vector_int ___operator_assign__F11svector_int_P11svector_int11svec
tor_int_(struct vector_int *___dst__P11svector_int, struct vector_int ___src__11
svector_int)
{
    ((*___dst__P11svector_int).__last__i=___src__11svector_int.__last__i);
    ((*___dst__P11svector_int).__capacity__i=___src__11svector_int.__capacity__i
);
    ((*___dst__P11svector_int).__data__Pi=___src__11svector_int.__data__Pi);
```

```
    return ___src__11svector_int;
}

// vector_int
// vector_int_allocate()
struct vector_int __vector_int_allocate__F11svector_int__()
{
    // return vector_int_allocate( DEFAULT_CAPACITY );
    return __vector_int_allocate__F11svector_int_i_(20);
}

// vector_int
// vector_int_allocate( int reserve )
struct vector_int __vector_int_allocate__F11svector_int_i_(int __reserve__i)
{
    // vector_int new_vector;
    struct vector_int __new_vector__11svector_int;
    // new_vector.last = -1;
    (__new_vector__11svector_int.__last__i=(-1));
    // new_vector.capacity = reserve;
    (__new_vector__11svector_int.__capacity__i=__reserve__i);
    // new_vector.data = malloc( sizeof( int ) * reserve );
    (__new_vector__11svector_int.__data__Pi=malloc((sizeof(int )*__reserve__i)))
;
    // return new_vector;
    return __new_vector__11svector_int;
}

// void
// vector_int_deallocate( vector_int vec )
void __vector_int_deallocate__F_11svector_int_(struct vector_int __vec__11svecto
r_int)
{
    // free( vec.data );
    free(__vec__11svector_int.__data__Pi);
}

// void append( vector_int *vec, int element )
```

```
void __append__F_P11svector_inti_(struct vector_int *__vec__P11svector_int, int
__element__i)
{
    // vec->last++;
    (*__vec__P11svector_int).__last__i++;
    // if( vec->last == vec->capacity ) {
    if (((int )(((*__vec__P11svector_int).__last__i==(*__vec__P11svector_int).__
capacity__i)!=0)))

        {
            // vec->capacity *= 2;
            ((*__vec__P11svector_int).__capacity__i*=2);
            // vec->data = realloc( vec->data, sizeof( int ) * vec->capacity );
            ((*__vec__P11svector_int).__data__Pi=realloc((*__vec__P11svector_int
).__data__Pi, (sizeof(int )*(*__vec__P11svector_int).__capacity__i)));
        }


    // vec->data[ vec->last ] = element;
    ((*__vec__P11svector_int).__data__Pi[(*__vec__P11svector_int).__last__i]=__e
lement__i);
}

// lvalue int
// ?[?]( vector_int vec, int index )
int *___operator_index__FLi_11svector_inti_(struct vector_int __vec__11svector_i
nt, int __index__i)
{
    // return vec.data[ index ];
    return (&__vec__11svector_int.__data__Pi[__index__i]);
}

// int
// last( vector_int vec )
int __last__Fi_11svector_int_(struct vector_int __vec__11svector_int)
{
    // return vec.last;
    return __vec__11svector_int.__last__i;
```

```
}
```

A.3 Iterators

A.3.1 iterator

iterator.h

```
#ifndef ITERATOR_H
#define ITERATOR_H

#include "iostream.h"

// an iterator can be used to traverse a data structure
context iterator( type iterator_type, type elt_type )
{
  // point to the next element
  iterator_type ++?( iterator_type* );
  iterator_type --?( iterator_type* );

  // can be tested for equality with other iterators
  int ?==?( iterator_type, iterator_type );
  int ?!=?( iterator_type, iterator_type );

  // dereference to get the pointed-at element
  lvalue elt_type *?( iterator_type );
};

// writes the range [begin, end) to the given stream

forall( type elt_type | writeable( elt_type ),
        type iterator_type | iterator( iterator_type, elt_type ),
        dtype os_type | ostream( os_type ) )
void write_reverse( iterator_type begin, iterator_type end, os_type *os );

#endif /* #ifndef ITERATOR_H */
```

iterator.c

```
#include "iterator.h"

forall( type elt_type | writeable( elt_type ),
        type iterator_type | iterator( iterator_type, elt_type ),
        dtype os_type | ostream( os_type ) )
void
write_reverse( iterator_type begin, iterator_type end, os_type *os )
{
  iterator_type i = end;
  do {
    --i;
    os << *i << ' ';
  } while( i != begin );
}
```

Translator Output

```
// forall( type elt_type | writeable( elt_type ),
//         type iterator_type | iterator( iterator_type, elt_type ),
//         dtype os_type | ostream( os_type ) )
// void
// write_reverse( iterator_type begin, iterator_type end, os_type *os )
void __write_reverse__A2_1_0____operator_assign__PF2t0_P2t02t0____operator_shift
left__PA0_1_0___write__PFP2d1_P2d1PCcUl___fail__PFi_P2d1__FP2d1_P2d12t0____opera
tor_assign__PF2t1_P2t12t1____operator_preincr__PF2t1_P2t1____operator_predecr__P
F2t1_P2t1____operator_equal__PFi_2t12t1____operator_notequal__PFi_2t12t1____oper
ator_deref__PFL2t0_2t1___write__PFP2d2_P2d2PCcUl___fail__PFi_P2d2__F_2t12t1P2d2_
(void *(*_adapterFP9telt_type_14titerator_type_)(void (*)(), void *), int (*_ada
pterFi_14titerator_type14titerator_type_)(void (*)(), void *, void *), void (*_a
dapterF14titerator_type_P14titerator_type_)(void (*)(), void *, void *), void (*
_adapterF14titerator_type_P14titerator_type14titerator_type_)(void (*)(), void *
, void *, void *), void *(*_adapterA0_1_0___write__PFP2d0_P2d0PCcUl___fail__PFi_
P2d0__FP2d0_P2d09telt_type_)(void *(*__write__PFP8tos_type_P8tos_typePCcUl_)(voi
d *, const char *, long unsigned int ), int (*__fail__PFi_P8tos_type_)(void *),
void (*)(), void *, void *), void (*_adapterF9telt_type_P9telt_type9telt_type_)(
```

```
void (*)(), void *, void *, void *), long unsigned int elt_type, long unsigned i
nt iterator_type, void (*___operator_assign__PF9telt_type_P9telt_type9telt_type_
)(), void (*___operator_shiftleft__PA0_1_0___write__PFP2d0_P2d0PCcUl___fail__PFi
_P2d0__FP2d0_P2d09telt_type_)(), void (*___operator_assign__PF14titerator_type_P
14titerator_type14titerator_type_)(), void (*___operator_preincr__PF14titerator_
type_P14titerator_type_)(), void (*___operator_predecr__PF14titerator_type_P14ti
terator_type_)(), void (*___operator_equal__PFi_14titerator_type14titerator_type
_)(), void (*___operator_notequal__PFi_14titerator_type14titerator_type_)(), voi
d (*___operator_deref__PFL9telt_type_14titerator_type_)(), void *(*__write__PFP8
tos_type_P8tos_typePCcUl_)(void *, const char *, long unsigned int ), int (*__fa
il__PFi_P8tos_type_)(void *), void *__begin__14titerator_type, void *__end__14ti
terator_type, void *__os__P8tos_type)
{
    // iterator_type i = end;
    void *__i__14titerator_type;
    (__i__14titerator_type=alloca(iterator_type));
    void *_temp0;
    (_temp0=alloca(iterator_type));
    (_adapterF14titerator_type_P14titerator_type14titerator_type_(___operator_as
sign__PF14titerator_type_P14titerator_type14titerator_type_, _temp0, __i__14tite
rator_type, __end__14titerator_type) , _temp0);
    // do {
    do{

    {
        // --i;
        void *_temp1;
        (_temp1=alloca(iterator_type));
        (_adapterF14titerator_type_P14titerator_type_(___operator_predecr__PF14t
iterator_type_P14titerator_type_, _temp1, __i__14titerator_type) , _temp1);
        // os << *i << ' ';
        ___operator_shiftleft__A0_1_0___write__PFP2d0_P2d0PCcUl___fail__PFi_P2d0
__FP2d0_P2d0c_(__write__PFP8tos_type_P8tos_typePCcUl_, __fail__PFi_P8tos_type_,
_adapterA0_1_0___write__PFP2d0_P2d0PCcUl___fail__PFi_P2d0__FP2d0_P2d09telt_type_
(__write__PFP8tos_type_P8tos_typePCcUl_, __fail__PFi_P8tos_type_, ___operator_sh
iftleft__PA0_1_0___write__PFP2d0_P2d0PCcUl___fail__PFi_P2d0__FP2d0_P2d09telt_typ
e_, __os__P8tos_type, _adapterFP9telt_type_14titerator_type_(___operator_deref__
PFL9telt_type_14titerator_type_, __i__14titerator_type)), ' ');
```

```
    }
    // } while( i != begin );
    } while(((int )(_adapterFi_14titerator_type14titerator_type_(___operator_not
equal__PFi_14titerator_type14titerator_type_, __i__14titerator_type, __begin__14
titerator_type)!=0)));

}
```

A.4 Test Program

A.4.1 vector_test

vector_test.c

```
#include "fstream.h"
#include "vector_int.h"
#include "array.h"
#include "iterator.h"

int
main()
{
  ofstream *sout = ofstream_stdout();
  ifstream *sin = ifstream_stdin();
  vector_int vec = vector_int_allocate();

  // read in numbers until EOF or error
  int num;
  for(;;) {
    sin >> &num;
    if( fail( sin ) || eof( sin ) ) break;
    append( &vec, num );
  }

  // write out the numbers
  sout << "Array elements: ";
  write_reverse( begin( vec ), end( vec ), sout );
```

```
  sout << "\n";
  return 0;
}
```

Translator Output

```
// int
// main()
int main()
{
    // ofstream *sout = ofstream_stdout();
    // ifstream *sin = ifstream_stdin();
    // vector_int vec = vector_int_allocate();
    // int num;
    struct ofstream *__sout__P9sofstream;
    struct ifstream *__sin__P9sifstream;
    struct vector_int __vec__11svector_int;
    int __num__i;
    (__sout__P9sofstream=__ofstream_stdout__FP9sofstream__());
    (__sin__P9sifstream=__ifstream_stdin__FP9sifstream__());
    ___operator_assign__F11svector_int_P11svector_int11svector_int_((&__vec__11s
vector_int), __vector_int_allocate__F11svector_int__());
    // for(;;) {
    for (;;)

        {
            // sin >> &num;
            ___operator_shiftright__A0_1_0___read__PFP2d0_P2d0PcUl___unread__PFP
2d0_P2d0c___fail__PFi_P2d0___eof__PFi_P2d0__FP2d0_P2d0Pi_(__read__FP9sifstream_P
9sifstreamPcUl_, __unread__FP9sifstream_P9sifstreamc_, __fail__Fi_P9sifstream_,
__eof__Fi_P9sifstream_, __sin__P9sifstream, (&__num__i));
            // if( fail( sin ) || eof( sin ) ) break;
            if (((int )((((int )(__fail__Fi_P9sifstream_(__sin__P9sifstream)!=0)
) || ((int )(__eof__Fi_P9sifstream_(__sin__P9sifstream)!=0)))!=0)))
                break;

            // append( &vec, num );
            __append__F_P11svector_inti_((&__vec__11svector_int), __num__i);
```

```
        }

    // sout << "Array elements: ";
    ___operator_shiftleft__A0_1_0___write__PFP2d0_P2d0PCcUl___fail__PFi_P2d0__FP
2d0_P2d0PCc_(__write__FP9sofstream_P9sofstreamPCcUl_, __fail__Fi_P9sofstream_, _
_sout__P9sofstream, "Array elements: ");
    // write_reverse( begin( vec ), end( vec ), sout );
    int *_thunk0(int **_p0, int *_p1)
    {
        return ((*_p0)=_p1);
    }

    int *_thunk1(int **_p0)
    {
        return (++(*_p0));
    }

    int *_thunk2(int **_p0)
    {
        return (--(*_p0));
    }

    int _thunk3(int *_p0, int *_p1)
    {
        return (_p0==_p1);
    }

    int _thunk4(int *_p0, int *_p1)
    {
        return (_p0!=_p1);
    }

    int *_thunk5(int *_p0)
    {
        return (&(*_p0));
    }

    void _adapterF11svector_int_P11svector_int11svector_int_(void (*_adaptee)(),
```

```
 void *_ret, void *_p0, void *_p1)
    {
        ((*((struct vector_int *)_ret))=((struct vector_int (*)(struct vector_in
t *, struct vector_int ))_adaptee)(_p0, (*((struct vector_int *)_p1))));
    }

    void _adapterFi_Pii_(void (*_adaptee)(), void *_ret, void *_p0, void *_p1)
    {
        ((*((int *)_ret))=((int (*)(int *, int ))_adaptee)(_p0, (*((int *)_p1)))
);
    }

    int _adapterFi_11svector_int_(void (*_adaptee)(), void *_p0)
    {
        return ((int (*)(struct vector_int ))_adaptee)((*((struct vector_int *)_
p0)));
    }

    void *_adapterFPi_11svector_inti_(void (*_adaptee)(), void *_p0, int _p1)
    {
        return ((int *(*)(struct vector_int , int ))_adaptee)((*((struct vector_
int *)_p0)), _p1);
    }

    int *_temp0;
    (_temp0=__begin__A2_0_0____operator_assign__PF2t0_P2t02t0____operator_assign
__PF2t1_P2t12t1___last__PFi_2t0____operator_index__PFL2t1_2t0i__FP2t1_2t0_(_adap
terFPi_11svector_inti_, _adapterFi_11svector_int_, _adapterFi_Pii_, _adapterF11s
vector_int_P11svector_int11svector_int_, sizeof(struct vector_int ), sizeof(int
), ((void (*)())___operator_assign__F11svector_int_P11svector_int11svector_int_)
, ((void (*)())___operator_assign__Fi_Pii_), ((void (*)())__last__Fi_11svector_i
nt_), ((void (*)())___operator_index__FLi_11svector_inti_), (&__vec__11svector_i
nt)));
    int *_temp1;
    (_temp1=__end__A2_0_0____operator_assign__PF2t0_P2t02t0____operator_assign__
PF2t1_P2t12t1___last__PFi_2t0____operator_index__PFL2t1_2t0i__FP2t1_2t0_(_adapte
rFPi_11svector_inti_, _adapterFi_11svector_int_, _adapterFi_Pii_, _adapterF11sve
ctor_int_P11svector_int11svector_int_, sizeof(struct vector_int ), sizeof(int ),
```

```
 ((void (*)())___operator_assign__F11svector_int_P11svector_int11svector_int_),
((void (*)())___operator_assign__Fi_Pii_), ((void (*)())__last__Fi_11svector_int
_), ((void (*)())___operator_index__FLi_11svector_inti_), (&__vec__11svector_int
)));
    void *_adapterA0_1_0___write__PFP2d0_P2d0PCcUl___fail__PFi_P2d0__FP2d0_P2d0i
_(void *(*__write__PFP8tos_type_P8tos_typePCcUl_)(void *, const char *, long uns
igned int ), int (*__fail__PFi_P8tos_type_)(void *), void (*_adaptee)(), void *_
p0, void *_p1)
    {
        return ((void *(*)(void *(*__write__PFP8tos_type_P8tos_typePCcUl_)(void
*, const char *, long unsigned int ), int (*__fail__PFi_P8tos_type_)(void *), vo
id *, int ))_adaptee)(__write__PFP8tos_type_P8tos_typePCcUl_, __fail__PFi_P8tos_
type_, _p0, (*((int *)_p1)));
    }

    void _adapterFPi_PPiPi_(void (*_adaptee)(), void *_ret, void *_p0, void *_p1
)
    {
        ((*((int **)_ret))=((int *(*)(int **, int *))_adaptee)(_p0, (*((int **)_
p1))));
    }

    void _adapterFPi_PPi_(void (*_adaptee)(), void *_ret, void *_p0)
    {
        ((*((int **)_ret))=((int *(*)(int **))_adaptee)(_p0));
    }

    int _adapterFi_PiPi_(void (*_adaptee)(), void *_p0, void *_p1)
    {
        return ((int (*)(int *, int *))_adaptee)((*((int **)_p0)), (*((int **)_p
1)));
    }

    void *_adapterFPi_Pi_(void (*_adaptee)(), void *_p0)
    {
        return ((int *(*)(int *))_adaptee)((*((int **)_p0)));
    }
```

```
    __write_reverse__A2_1_0____operator_assign__PF2t0_P2t02t0____operator_shiftl
eft__PA0_1_0___write__PFP2d1_P2d1PCcUl___fail__PFi_P2d1__FP2d1_P2d12t0____operat
or_assign__PF2t1_P2t12t1____operator_preincr__PF2t1_P2t1____operator_predecr__PF
2t1_P2t1____operator_equal__PFi_2t12t1____operator_notequal__PFi_2t12t1____opera
tor_deref__PFL2t0_2t1___write__PFP2d2_P2d2PCcUl___fail__PFi_P2d2__F_2t12t1P2d2_(
_adapterFPi_Pi_, _adapterFi_PiPi_, _adapterFPi_PPi_, _adapterFPi_PPiPi_, _adapte
rA0_1_0___write__PFP2d0_P2d0PCcUl___fail__PFi_P2d0__FP2d0_P2d0i_, _adapterFi_Pii
_, sizeof(int ), sizeof(int *), ((void (*)())___operator_assign__Fi_Pii_), ((voi
d (*)())___operator_shiftleft__A0_1_0___write__PFP2d0_P2d0PCcUl___fail__PFi_P2d0
__FP2d0_P2d0i_), ((void (*)())(&_thunk0)), ((void (*)())(&_thunk1)), ((void (*)(
))(&_thunk2)), ((void (*)())(&_thunk3)), ((void (*)())(&_thunk4)), ((void (*)())
(&_thunk5)), __write__FP9sofstream_P9sofstreamPCcUl_, __fail__Fi_P9sofstream_, (
&_temp0), (&_temp1), __sout__P9sofstream);
    // sout << "\n";
    ___operator_shiftleft__A0_1_0___write__PFP2d0_P2d0PCcUl___fail__PFi_P2d0__FP
2d0_P2d0PCc_(__write__FP9sofstream_P9sofstreamPCcUl_, __fail__Fi_P9sofstream_, _
_sout__P9sofstream, "\n");
    // return 0;
    return 0;
}
```

Bibliography

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[3] T. P. Baker. A one-pass algorithm for overload resolution in Ada. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(4):601–614, 1982.

[4] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice Hall, 1996.

[5] Thomas M. Breuel. Lexical closures for C++. In *Proceedings of the 1988 USENIX C++ Conference*, 1988.

[6] P. A. Buhr, David Till, and C. R. Zarnke. Assignment as the sole means of updating objects. *Software—Practice and Experience*, 24(9):835–870, 1994.

[7] Robert Cartwright and Guy L. Steele Jr. Compatible genericity with run-time types for the java programming language. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 201–215. ACM Press, 1998.

[8] G. V. Cormack. An algorithm for the selection of overloaded functions in Ada. *SIGPLAN Notices*, 16(2):48–52, 1981.

[9] G. V. Cormack and A. K. Wright. Type-dependent parameter inference. *ACM SIGPLAN Notices*, 25(6):127–136, 1990. 1990 Conference on Programming Language Design and Implementation.

[10] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM Press, 1982.

[11] Glen Ditchfield. *Contextual Polymorphism*. PhD thesis, University of Waterloo, 1994.

[12] Glen Ditchfield. Cforall Reference Manual and Rationale, 1998. ftp://plg.uwaterloo.ca/pub/Cforall/refrat.ps.gz.

[13] Tom Duff. rc - a shell for Plan 9 and UNIX systems. In *Proceedings of the Summer 1990 UKUUG Conference*, pages 21–33, July 1990.

[14] D. Duggan, G. Cormack, and J. Ophel. Kinded type inference for parametric overloading. *Acta Informatica*, 33(1):21–68, 1996.

[15] Margaret Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual (Revised)*. Addison-Wesley, 1991.

[16] thunk. *The Free On-line Dictionary of Computing*. Denis Howe, Ed. http://www.foldoc.org.

[17] International Organization for Standardization. International Standard — Programming Languages — Ada, 1995. ISO/IEC 8652:1995.

[18] International Organization for Standardization. International Standard — Programming Languages — C++, 1998. ISO/IEC 14882:1998.

[19] International Organization for Standardization. International Standard — Programming Languages — C, 1999. ISO/IEC 9899:1999.

[20] Samuel P. Harbison. *Modula-3*. Prentice Hall, 1992.

[21] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 130–141. ACM Press, 1995.

[22] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, Yanling Wang, and James Cheney. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, June 2002. To appear.

[23] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 468–476. ACM Press, 1990.

[24] Xavier Leroy. Unboxed objects and polymorphic typing. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 177–188. ACM Press, 1992.

[25] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

[26] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[27] John C. Mitchell. Coercion and type inference. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 175–185, 1984.

[28] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Colloque sur la Programmation*, pages 408–423. Springer-Verlag, 1974. Lecture Notes in Computer Science, v. 19.

[29] Dennis M. Ritchie. The development of the C language. In *The second ACM SIGPLAN conference on History of programming languages*, pages 201–208. ACM Press, 1993.

[30] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[31] Paul Roe and Clemens Szyperski. Lightweight parametric polymorphism for Oberon. In *Proceedings of the Joint Modular Languages Conference*, 1997.

[32] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proceedings of the conference on Programming language design and implementation*, pages 116–129. ACM Press, 1995.

[33] Jun Shen and Gordon V. Cormack. Automatic instantiation in Ada. In *Proceedings of the conference on TRI-Ada '91*, pages 338–346. ACM Press, 1991.

[34] Geoffrey S. Smith. *Polymorphic Type Inference for Languages with Overloading and Subtyping*. PhD thesis, Cornell University, 1991. Also available as TR 91–1230.

[35] David Till. Tuples in imperative programming languages. Master's thesis, University of Waterloo, 1989.

[36] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 1–11. ACM Press, 1994.

[37] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM Press, 1989.

[38] J.B. Wells. Typability and type checking in the second order λ-calculus are equivalent and undecidable. In *Proceedings of the 9th IEEE Symposium on Logic in Computer Science*, pages 176–185, July 1994.

[39] A. K. Wright. Design of the programming language ForceOne. Technical Report CS-87-10, University of Waterloo, 1987.

[40] David Ziegler. Adding "Overloading Polymorphism" to "C". Independent Studies thesis, University of Waterloo, 1992.