An Overview of Cforall

Glen Ditchfield

96/08/02

Cforall is an extension of the C programming language [1] that provides opaque types and parametric polymorphism. It allows the definition of polymorphic functions, which have parameters whose types are not statically known. These "unknown" types, and any operations on instances of the types that the function needs, are part of the function's interface. When polymorphic functions are called, parameter inference is used to determine the parameter types from the types of the arguments, and the operations that are needed are selected from the calling environment.

Cforall's most distinctive feature, compared to other parametrically polymorphic languages, is the use of *specifications* as a name for the collection of declarations that define the interfaces of types.

1 Type Declarations

Type declarations introduce new type names into the current scope. The simplest type declaration is

```
type T;
```

Within the scope of that declaration, `T` is the name of an type distinct from all other types. It can be instantiated, and instances can be assigned and passed to the `sizeof` and the "address-of" operators. The scope of `T` may contain declarations of other functions that manipulate `T` objects[1].

A type declaration with an initializer is a *type definition.*

```
type Complex = struct { double re, im; };
```

`Complex` is a new type, implemented as a structure containing two `doubles`. Within the scope of the definition, implicit conversions exist between the new type and the implementation type. By default, assignment for the new type is the same as assignment for the implementation type. (Assignment between `structs` copies each non-array member and each element of each array member, using the appropriate assignment operator for each data item.)

```
Complex i = { 0.0, 1.0 };
double abs(Complex c) {
    return sqrt(c.re*c.re + c.im*c.im);
}
```

`type` is not itself a type; type declarations just *look* like ordinary declarations. Type declarations are allowed in the declaration lists of blocks and translation units, and in `forall` specifiers and specification definitions (see below). They are not allowed in structs or function parameter lists, and there are no pointers to types, functions returning types, or arrays of types.

[1] Here "object" has the same sense as it does in the C standard: a region of memory containing a value.

1.1 Opaque Types

`type` differs from `typedef` in two ways: a type name is a new type, not a synonym for an existing type, and a type name is subject to linkage, just like a variable or function name. Hence the normal C scope and linkage rules provide opaque types. A type can be defined in one translation unit and declared with external linkage in another unit, where its name is known but its implementation is hidden.

```
extern type Complex;
extern Complex i;
extern double abs(Complex);
double d = abs(i);
```

2 Overloading

Cforall allows identifiers to be overloaded. Multiple declarations of an identifier are allowed if they specify incompatible types. Each declaration refers to a distinct object or function. Declarations in inner scopes only hide declarations of the same identifier in outer scopes if the specified types are compatible. In the case of function types, two overloadings of an identifier might differ only in their return types.

"`0`" and "`1`" are special identifiers that can be overloaded to define the two most useful constants for new types. "`?+?`" is a special identifier denoting the binary addition operator. Most operators can be overloaded by declaring functions whose names contain question marks where the operands in an expression would go[2], and a set of rewrite rules defines the relationship between "expression" syntax and "function call" syntax. The standard C operators are treated as predefined functions. A `0` constant and a `+=` assignment operator could be defined for the type `Complex` like this:

```
const Complex 0 = { 0.0, 0.0 };
Complex ?+=?(Complex* target, Complex source) {
    target->re += source.re;
    target->im += source.im;
    return *target;
}
```

Given two `Complex` variables called `c1` and `c2`, "`c1 += c2`" is treated as the function call "`?+=?(&c1,c2)`".

Cforall does not allow the definition of new operators. Neither does it allow overloading of the "flow of control" operators "`&&`", "`||`", and "`?:`". However, these operators, and the controlling expressions of loops and `if` statements, are defined to work by checking that their operands are not equal to `0`, so overloading `0` and "`!=`" for new types is sufficient.

3 Polymorphic Functions

`Polymorphic functions` are functions that have parameters whose types are not statically known. The functions can be called with arguments of many different types. A `forall` specifier is used to introduce names for the actual types of the arguments.

Here is a function that interchanges the values of its arguments.

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

[2] Cforall's prefix and postfix increment operators have distinct names: `++?` and `?++`.

The parameters `a` and `b` must be pointers to objects of exactly the same type. That type is not known statically, but is passed to `swap()` as parameter `T`. Within `swap()`, instances of `T` can be declared, initialized, and assigned (and also returned, although `swap()` has no need to do so). `T` is called an *inferred parameter*.

Given a call to `swap()`, a Cforall translator must deduce the value to pass to `T` and the assignment function to use.

```
int main(void) {
    int m, n;
    Complex q, r;
    /* ... */
    swap(&m, &n);
    swap(&q, &r);
    swap(&m, &r);           /* compile-time error! */
    /* ... */
}
```

For the first call, the translator must infer from the `a` and `b` arguments that the pointed-at type is `int`, and it must locate the integer assignment function[3]. Both are passed to `swap()` along with the addresses of `m` and `n`. During the call, `temp` will have type `int`. For the second call, much the same happens: `T` is `Complex`, and the assignment function is complex assignment.

The inference process is entirely static. If the translator cannot determine the value of a type parameter at compile time, the call is invalid.

Polymorphic functions really are functions, not templates or macros. They have addresses and types, just like ordinary "monomorphic" functions.

3.1 Assertions

Practically all polymorphic functions require more from their arguments than `swap()` does. A function that doubles its argument might require that the argument have an addition operation. Cforall provides *assertions* in type declarations so that programmers can assert that certain functions exist.

```
forall(type T | T ?+?(T,T) )
T twice(T v) {
    return v + v;
}
/* ... */
int four = twice(2);
```

The assertion "`| T ?+?(T,T)`"[4] declares that, whatever type `T` is, there must be a function named "`?+?`" that takes two values of that type and returns a value of that type; "`?+?`" is an *assertion parameter* of `twice`. The declaration has the same scope as `T`, so a translator can determine statically that the use of "`+`" in the body of `twice()` is legal. At the call site the translator must locate an appropriate addition operator and pass it to `twice()` along with the `int` and assignment arguments. If no appropriate "`?+?`" function is visible in the scope of the call, then the call is invalid.

4 Specifications

Most polymorphic functions need many assertion parameters, and many related polymorphic functions need the same assertion parameters. Those facts suggest a need for a language facility for abstracting out asser-

[3] Integer assignment is a predefined function in every translation unit.

[4] Read "`|`" as "such that".

tions. Hence Cforall provides the *specification*: a collection of object and function declarations, parameterized by one or more types. An assertion can apply a specification to argument types; it asserts that for those arguments, all functions and objects declared in the specification exist.

The simplest use of specifications and assertions defines a property of a type. For instance, a type might be considered to have the property of "summability" if it has an increment operator and a "0" value. A specification can express that requirement:

```
spec summable(type T) {
    T ?+=?(T*, T);
    const T 0;
}
```

The `Complex` type defined above meets those requirements (it *satisfies* the specification), and an assertion on its declaration could announce that fact.

```
extern type Complex | summable(Complex);
```

This single declaration is equivalent to the three declarations

```
extern type Complex;
extern Complex ?+=?(Complex*, Complex);
extern const Complex 0;
```

Specifications can also describe relationships *between* types. Consider a type whose values represent points along a dimension, and a companion type whose values represent the distance between points. Certain arithmetic combinations of the types are meaningful:

```
spec dimension(type Point, type Distance) {
    Distance ?-?(Point, Point);
    Point    ?+?(Point, Distance);
    Point    ?+?(Distance, Point);
    Point    ?-?(Point, Distance);
}
```

Cforall allows pointer arithmetic, and the standard C header `<stddef.h>` defines the type `ptrdiff_t` to hold the result of subtracting pointers. Hence the assertion "`| dimension(char*, ptrdiff_t)`" holds. The specification might also be satisfied by programmer-defined types representing instants in time and time intervals.

A type can satisfy many specifications simultaneously. Each specification selects a different portion of the type's interface. Different assertions can be used to select the appropriate interface. For instance, a program could use different specifications for the reader's and writer's views of a buffer.

It is important to note that a specification is not a type, an abstract type, or a supertype. The `dimension` example shows that a specification does not even have to describe the interface of a *single* type.

4.1 Specification Hierarchies

Since the parameter list of a specification is made up of type declarations, it can contain assertions. Programmers can use this to define new specifications that extend old ones. For instance, a programmer might want to describe types with a binary addition operator as well as the facilities that `summable` describes.

```
spec addable(type T | summable(T)) {
    T ?+?(T, T);
}
```

The `int` type satisfies `addable`. To verify the assertion "`| addable(int)`", a Cforall translator first checks that `int` satisfies `summable`, then locates the addition operator for `int`s. `Complex` as defined above does not satisfy `addable` because it lacks an addition operator, but if the translation unit that declares "`extern type Complex`" defines its own complex addition operator, `Complex` would satisfy `addable` within that translation unit.

This mixture of specifications and assertions lets programmers define hierarchies of related specifications. In cases like the example above, the hierarchy resembles a tree of subclasses. They can also be used to imitate the nesting of `private`/`protected`/`public` interfaces of C++ classes.

4.2 Specifications and Polymorphic Functions

Here is a function that adds up the elements of an array of unspecified type.

```
forall(type Elt | summable(Elt))
Elt sum(const Elt list[], int n) {
    Elt total = 0;
    int i;
    for (i = 0; i != n; i++) {
        total += list[i];
    }
    return total;
}
```

Within `sum`, `Elt` is a type with a `0` value and assignment and "`+=`" operators, and `total` is an instance of it. (Note the use of `Elt`'s `0` to initialize `total`.) The constant `0` and the operator "`+=`" can be used because they are declared by the assertion that `Elt` satisfies `summable`.

```
extern type Complex | summable(Complex);
int main(void) {
    Complex clist[10];
    int     ilist[10];
    /* ... */
    Complex cs = sum(clist, 10);
    int     is = sum(ilist, 10);
}
```

For the first call, the Cforall translator must infer from the `clist` argument that the array element type is `Complex`, and must locate `0` and "`+=`" in the current scope. (The assertion `summable(Complex)` declared them.) For the second call, much the same happens: `Elt` is `int`, and the integer `0` and "`+=`" are predefined functions.

4.3 Kinds of Types

In the terminology of the C standard, `sum`'s `Elt` type is an "object type". It has a size, an assignment operator, and can be instantiated. Some polymorphic functions do not require that their implicit parameters have these properties. Cforall provides two variants of the `type` declaration for these situations. `dtype` declarations declare names for "data" types (incomplete types as well as object types). `ftype` declarations declare names for function types. These are mostly used to declare polymorphic functions that manipulate pointers, but do not have to manipulate instances of the pointed-at type. The predefined pointer operators make heavy use of them: one of the predefined overloadings of the equality operator is

```
forall(ftype FT) int ?==?(FT*, FT*);
```

and hence a Cforall program can compare any two pointers that point at the same type of function.

4.4 Null Pointers

One of the finer points of the C language is its treatment of "null pointer constants". These are not literal values that belong to some pointer type; they are special strings of tokens. The C token "`0`" is an expression of type `int` with the value "zero", and it *also* is a null pointer constant. Similarly, "`(void*)0`" is an expression of type `(void*)` whose value is a null pointer, and it is also a null pointer constant. However, in C, "`(void*)(void*)0`" is *not* a null pointer constant, even though it is null-valued, a pointer, and constant! The semantics of C expressions contain many special cases to deal with subexpressions that are null pointer constants.

In Cforall, this concept does not exist. Instead, one of the predefined overloadings of the "`0`" identifier is a polymorphic object that can point at any type of data.

```
forall(dtype DT) const DT *const 0;
```

Given the expression "`(void*)0`", a Cforall translator infers that `DT` must be `void`, and produces a constant, null-valued pointer to `void`. Cforall's parameter inference rules replace C's special cases.

5 Overload Resolution

Since identifiers can be overloaded, expressions can have many interpretations, each of a different type. Identifiers have one valid interpretation for each declaration that is visible in the current scope. Function calls and expressions involving overloadable operators have one valid interpretation for each return type[5] produced by a valid combination of function (or operator) interpretations and argument (or operand) interpretations. Given two declarations "`int f(int);`" and "`double f(double)`", the expression "`f`" has two valid interpretations, one for each function declaration. "`f(12)`" has two interpretations, one of type `int`, and one of type `double` that implicitly converts `12` to type `double`. "`(int)f(12)`" has two integer interpretations, one of which converts a `double` function result to `int`.

When an expression has more than one interpretation for a type, overload resolution chooses the best of the valid interpretations, based on three concepts: safe conversions, conversion cost, and degree of polymorphism. The safe conversions are those shown in figure 1. The conversion cost of a safe conversion is the path length in the graph of safe conversions. One polymorphic function is less polymorphic than another if it has fewer inferred parameters, or if it has the same number of inferred parameters and fewer explicit parameters use them in their types.

There are four tie-breaking rules to select the best valid interpretation.

1. The best valid interpretations use the fewest unsafe conversions.

2. Of these, the best are those where the functions being called are the least polymorphic functions.

3. Of these, the best have the lowest total conversion cost, including all implicit conversions in the argument expressions.

4. Of these, the best have the highest total conversion cost for the implicit conversions (if any) used to convert the argument expressions to the corresponding parameter types.

An interpretation for a given type is ambiguous if there is no single best valid interpretation, or if the best valid interpretation uses an ambiguous function, operator, or argument interpretation.

At the outermost level of an expression, the context of the expression forces it to have a certain type (`void` in the case of a statement context, for instance), and the best interpretation of that type is the "meaning" of the expression.

[5] Technically, for each composite type formed by combining compatible return types.

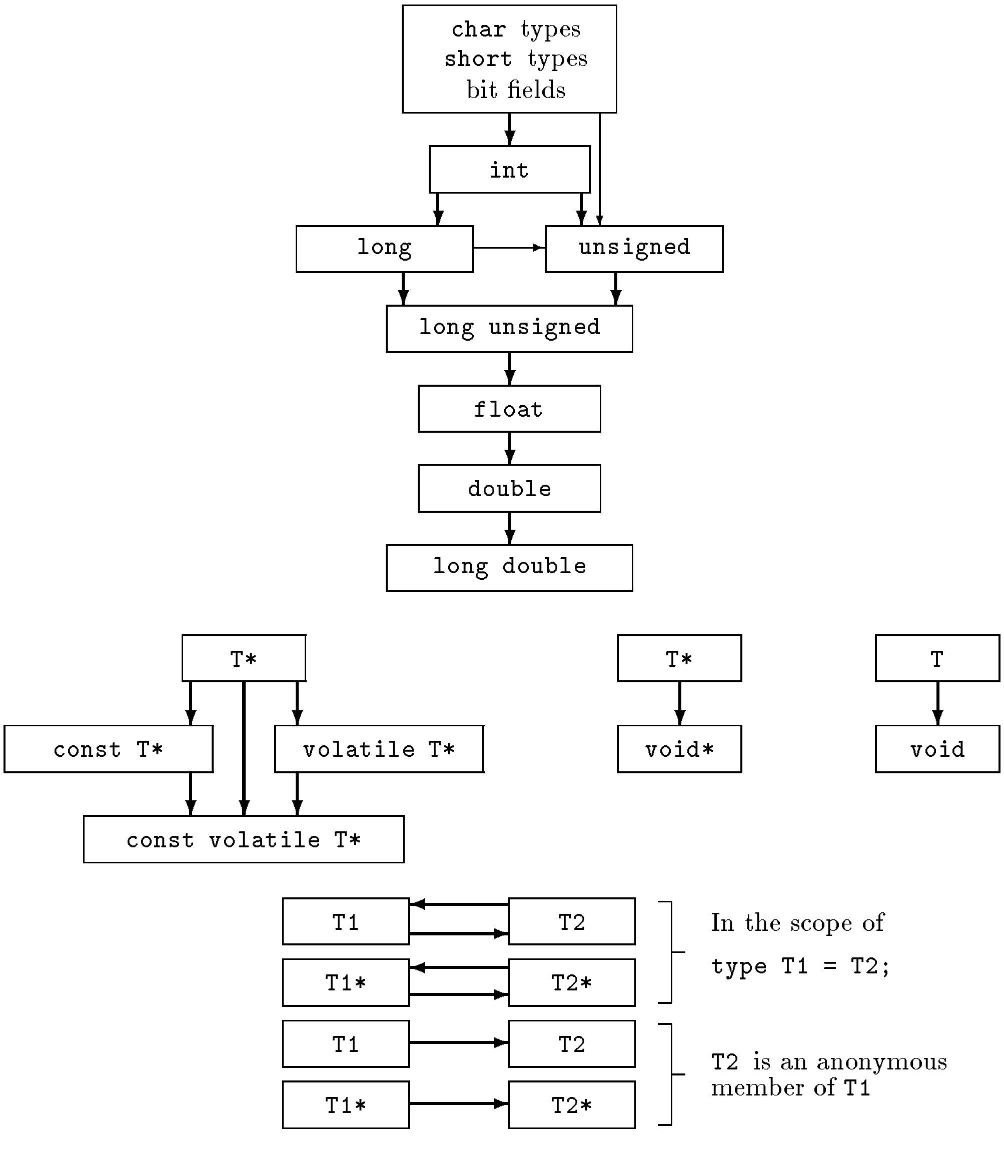

Figure 1: Safe conversions. Thin arrows represent implementation-dependent conversions.

5.1 Predefined Identifiers

The overload resolution rules given above apply equally to "operator" functions and "ordinary" functions. (This contrasts to C++[2, §13.2], where the ambiguity rules differ for binary operators.) The rules were designed so that a set of predefined operator identifier overloadings would mimic C's expression semantics. The "unsafe conversion" rule blocks selection of inappropriate operators. The "lowest total conversion cost" rule has the same effect as C's "usual arithmetic conversions". The "highest total conversion cost" rule places implicit conversions at the operands of unary operators, instead of at their results.

6 Inheritance

Specifications and assertions allow polymorphic functions to define the interfaces they require of their arguments. In object-oriented languages, superclasses provide the same facility. But superclasses also provide *inheritance*: the ability to implement a new type by modifying the implementation of an existing type.

For inheritance, Cforall borrows the *anonymous member* syntax from Plan 9 C[3]. A structure or union may contain members with no names. If the anonymous member is a structure or union, its members are treated as members of the containing structure or union. In the case of structures, new member fields can be added to an existing structure.

```
struct point {
    int x, y;
};
struct color_point {
    struct point;
    Color color;
};
struct color_point new(int x, int y, Color c) {
    struct color_point result;
    result.x = x; result.y = y; result.color = c;
    return result;
}
```

In the case of unions, new alternative members can be added.

```
union house_pet {
    Dog fido;
    Cat felix;
};
union pet {
    union house_pet;
    Pony  flicka;
};
```

In the mixed case, the programmer can declare tagged unions without having to make up a member name for the collection of alternative members.

```
struct literal {
    enum {an_int, a_float} tag;
    union {
        int i;
        double d;
```

```
    };
};
void print(struct literal l) {
    if (l.tag == an_int) printf("integer %d", l.i);
    else printf("float %g", l.d);
}
```

The containing type may be converted to the anonymous member's type, and pointers to the containing type may be converted to pointers to the anonymous member's type, so the containing type inherits functions defined for the anonymous member.

```
void move(struct point *p, int x, int y) {
    p->x = x;
    p->y = y;
}
int main(void) {
    struct color_point p;
    move(&p, 0, 0);
    /* ... */
}
```

In the case of opaque types, inheritance must work differently because the implementation of the type is hidden from client programs and because the type's definer may not want to inherit all of the implementation type's functions. Recall that by default the opaque type inherits its assignment operator from its implementation. If the inherited version is inappropriate, the programmer can provide an explicit definition to override it. Cforall extends this behavior to all functions by allowing the programmer to place an assertion on the definition of the opaque type. Default versions of functions and objects declared by the assertions are created from those of the implementation type by substituting the opaque type for the implementation type. This is much like Ada's "derived subprograms" [4, §3.4].

```
spec pointlike(type T) {
    int ?==?(T, T); /* equality operator */
    void move(T*, int, int);
}

int ?==?(struct point a, struct point b) {
    return a.x == b.x && a.y == b.y;
}

type Chess_Piece | pointlike(Chess_Piece) = struct {
    struct point;
    enum {pawn, rook, bishop, knight, queen, king} kind;
};

int ?==?(Chess_Piece a, Chess_Piece b) {
    return (struct point)a == (struct point)b && a.kind == b.kind;
}
```

Here `Chess_Piece` inherits the structure's assignment operator as its assignment operator, inherits the structure's `move()` function (which is the same as its anonymous member's `move()` function) as its move function, and defines its own equality operator to override the inherited equality operator.

References

[1] American National Standards Institute, 1430 Broadway, New York, New York 10018. *American National Standard for Information Systems – Programming Language – C*, December 1989. X3.159-1989.

[2] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, first edition, 1990.

[3] Ken Thompson. Plan 9 c compilers. In *Proceedings of the Summer 1990 UKUUG Conference*, pages 41–51, 1990. http://plan9.att.com/plan9/doc/compilers.html.

[4] United States Department of Defense. *The Programming Language Ada: Reference Manual*, ANSI/MIL-STD-1815A-1983 edition, February 1983. Published by Springer-Verlag.