Cforall Reference Manual and Rationale

Glen Ditchfield

98/01/17

Contents

1 Introduction **1**
1.1 Definitions of Terms 1

2 Lexical Elements **6**
2.1 Keywords 6
2.2 Identifiers 6
2.2.1 Constant Identifiers 6
2.2.2 Operator Identifiers 7
2.2.3 Scopes of Identifiers 8
2.2.4 Linkage of Identifiers 8

3 Conversions **10**
3.1 Other Operands 10
3.1.1 Anonymous Members 10
3.1.2 Specialization 11
3.2 Safe and Unsafe Conversions 11
3.3 Conversion Cost 12

4 Expressions **13**
4.1 Primary Expressions 14
4.2 Postfix Operators 15
4.2.1 Function Calls 16
4.2.2 Postfix Increment and Decrement Operators 19
4.2.3 Other Postfix Operators 19
4.3 Unary Operators 20
4.4 Cast Operators 22
4.5 Multiplicative Operators 23
4.6 Additive Operators 24
4.7 Bitwise Shift Operators 25
4.8 Relational Operators 25
4.9 Equality Operators 26
4.10 Bitwise AND Operator 27
4.11 Bitwise Exclusive OR Operator 28
4.12 Bitwise Inclusive OR Operator 28
4.13 Logical AND Operator 29
4.14 Logical OR Operator 29
4.15 Conditional Operator 30
4.16 Assignment Operators 30

4.17 Comma Operator 34

5 Declarations **35**
5.1 Type Specifiers 35
5.1.1 Structure and Union Specifiers 35
5.1.2 Forall Specifiers 36
5.2 Type Qualifiers 39
5.3 Specification Definitions 40
5.3.1 Assertions 41
5.4 Type Declarations 42
5.4.1 Default Functions and Objects 46
5.5 Initialization 48

6 Statements **49**
6.1 Expression and Null Statements 49
6.2 Jump Statements 49
6.3 Selection Statements 49
6.4 Iteration Statements 49

7 Preprocessing Directives **51**
7.1 Predefined Macro Names 51

A Examples **52**
A.1 C Types 52
A.1.1 Scalar, Arithmetic, and Integral Types 52
A.1.2 Modifiable Types 52
A.1.3 Pointer and Array Types 53
A.2 Relationships Between Operations 55
A.2.1 Relational and Equality Operators 55
A.2.2 Arithmetic and Integer Operations 56

Chapter 1

Introduction

This document is a reference manual and rationale for Cforall, a polymorphic extension of the C programming language. It makes frequent reference to the ANSI C standard [1], and occasionally compares Cforall to C++ [5].

The manual deliberately imitates the ordering of the ANSI C standard (although the section numbering varies). Unfortunately, this means that the manual contains more "forward references" than usual, and that it will be hard to follow if the reader does not have a copy of the ANSI standard near-by. For a gentle introduction to Cforall, see the companion document "An Overview of Cforall" [4].

> □ Commentary (like this) is quoted with quads. Commentary usually deals with subtle points, the rationale behind a rule, and design decisions. □

The syntax notation used in this document is the same as is used in the ANSI C standard, with one exception: ellipsis in the definition of a nonterminal, as in "*declaration: ...*", indicates that these rules extend a previous definition, which occurs in this document or in the ANSI C standard.

1.1 Definitions of Terms

Wherever possible, terms in this document have the same meaning as in the ANSI C standard. In particular, "shall" states a requirement on a program or implementation of Cforall, and "shall not" states a prohibition.

For the convenience of the reader, the following definitions are quoted from various sections of the ANSI C standard.

compatible type (§3.1.2.6) Two types have *compatible type* if their types are the same. Additional rules for determining whether two types are compatible are described [below].

- (§3.5.2.2) Each enumerated type shall be compatible with an integer type; the choice of type is implementation-defined.
- (§3.5.3) For two qualified types to be compatible, both shall have the identically qualified version of a compatible type; the order of type qualifiers within a list of specifiers or qualifiers does not affect the specified type.
- (§3.5.4.1) For two pointer types to be compatible, both shall be identically qualified and both shall be pointers to compatible types.
- (§3.5.4.2) For two array types to be compatible, both shall have compatible element types, and if both size specifiers are present, they shall have the same value.

- (§3.5.4.3) For two function types to be compatible, both shall specify compatible return types. Moreover, the parameter type lists, if both are present, shall agree in the number of parameters and in use of the ellipsis terminator; corresponding parameters shall have compatible types. If one type has a parameter type list and the other type is specified by a function declarator that is not part of a function definition and that contains an empty identifier list, the parameter list shall not have an ellipsis terminator and the type of each parameter shall be compatible with the type that results from the application of the default argument promotions. If one type has a parameter type list and the other type is specified by a function definition that contains a (possibly empty) identifier list, both shall agree in the number of parameters, and the type of each prototype parameter shall be compatible with the type that results from the application of the default argument promotions to the type of the corresponding identifier. (For each parameter declared with function or array type, its type for these comparisons is the one that results from conversion to a pointer type. For each parameter declared with qualified type, its type for these comparisons is the unqualified version of its declared type.)

Moreover, two structure, union, or enumerated types declared in separate translation units are compatible if they have the same number of members, the same member names, and compatible member types; for two structures, the members shall be in the same order; for two structures or unions, the bit-fields shall have the same widths; for two enumerations, the members shall have the same values.

composite type (§3.1.2.6) A *composite type* can be constructed from two types that are compatible; it is a type that is compatible with both of the two types and satisfies the following conditions:

- If one type is an array of known size, the composite type is an array of that size.
- If only one type is a function type with a parameter type list (a function prototype), the composite type is a function type with the parameter type list.
- If both types are function types with parameter type lists, the type of each parameter in the composite parameter list is the composite type of the corresponding parameters.

These rules apply recursively to the types from which the two types are derived.

For an identifier with external or internal linkage declared in the same scope as another declaration for that identifier, the type of the identifier becomes the composite type.

declaration (§3.5) A *declaration* specifies the interpretation and attributes of a set of identifiers.

default argument promotions (§3.3.2.2) If the expression [in a function call] that denotes the called function has a type that does not include a prototype, the integral promotions are performed on each argument and arguments that have type `float` are promoted to `double`. These are called the *default argument promotions* [...]. The ellipsis notation in a function prototype declarator causes argument type conversions to stop after the last declared parameter. The default argument promotions are performed on trailing arguments.

definition (§3.5) A declaration that also causes storage to be reserved for an object or function named by an identifier is a *definition*.

derived declarator type (§3.1.2.5) Array, function and pointer types are collectively called *derived declarator types*.

integral promotions (§3.2.1.1) A `char`, a `short int`, or an `int` bit-field, or their signed or unsigned varieties, or an enumeration type, may be used in an expression whenever an `int` or `unsigned int` may be used. If an `int` can represent all values of the original type, the value is converted to an `int`; otherwise, it is converted to an `unsigned int`. These are called the *integral promotions*.

function prototype (§3.1.2.1) A *function prototype* is a declaration of a function that declares the types of its parameters.

lvalue (§3.2.2.1) An *lvalue* is an expression (with an object type or an incomplete type other than `void`) that designates an object. When an object is said to have a particular type, the type is specified by the lvalue used to designate the object. A *modifiable lvalue* is an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member of all contained structures or unions) with a const-qualified type.

name space (§3.1.2.3) If more than one declaration of a particular identifier is visible at any point in a translation unit, the syntactic context disambiguates uses that refer to different entities. Thus, there are separate *name spaces* for various categories of identifiers, as follows:

- *label names* (disambiguated by the syntax of the label declaration and use);
- the *tags* of structures, unions, and enumerations (disambiguated by following any of the keywords `struct`, `union`, or `enum`);
- the *members* of structures or unions; each structure or union has a separate name space for its members (disambiguated by the type of the expression used to access the member via the `.` or `->` operator);
- all other identifiers, called *ordinary identifiers* (declared in ordinary declarators or as enumeration constants).

object (§1.6) An *object* is a region of data storage in the execution environment, the contents of which can represent values. Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined. When referenced, an object may be interpreted as having a particular type.

qualified type (§3.1.2.5) Each unqualified type has three corresponding *qualified versions* of its type: a *const-qualified* version, a *volatile-qualified* version, and a version having both qualifications. A derived type is not qualified by the qualifiers (if any) of the type from which it is derived.

[Cforall also allows *lvalue-qualified* versions of types, and versions having any combination of the three type qualifiers.]

scope (§3.1.2.1) There are four kinds of *scopes*: function, file, block, and function prototype....

A label name is the only kind of identifier that has *function scope*. It can be used (in a `goto` statement) anywhere in the function in which it appears, and is declared implicitly by its syntactic appearance (followed by a "`:`" and a statement). Label names shall be unique within a function.

Every other identifier has scope determined by the placement of its declaration (in a declarator or type specifier). If the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has *file scope*, which terminates at the end of the translation unit. If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has *block scope*, which terminates at the "}" that closes the associated block. If the declarator or type specifier that declares the identifier appears within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has *function prototype scope*, which terminates at the end of the function declarator.

Two identifiers have the same scope if and only if their scopes terminate at the same point.

Structure, union, and enumeration tags have scope that begins just after the appearance of the tag in a type specifier that declares the tag. Each enumeration constant has scope that begins just after the appearance of its defining enumerator in an enumerator list. Any other identifier has scope that begins just after the completion of its declarator.

[Cforall adds a fifth kind of scope: definition scope.]

sequence point (§2.1.2.3) At certain specified points in the execution sequence called *sequence points*, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place.

side effect (§2.1.2.3) Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all *side effects*, which are changes in the state of the execution environment.

storage-class specifier (§3.5.1) The keywords `auto`, `extern`, `register`, `static`, and `typedef` are *storage-class specifiers*.

storage duration (§3.1.2.4) An object has a *storage duration* that determines its lifetime. There are two storage durations: static and automatic.

An object whose identifier is declared with external or internal linkage, or with the storage-class specifier `static` has *static storage duration*. For such an object, storage is reserved and its stored value is initialized only once, prior to program startup. The object exists and retains its last-stored value throughout the execution of the entire program.

An object whose identifier is declared with no linkage and without the storage-class specifier `extern` has *automatic storage duration*. Storage is guaranteed to be reserved for a new instance of such an object on each normal entry into the block with which it is associated, or on a jump from outside the block to a labeled statement in the block or in an enclosed block. If an initialization is specified for the value stored in the object, it is performed on each normal entry, but not if the block is entered by a jump to a labeled statement.

translation unit (§2.1.1.1) A source file together with all the headers and source files included via the preprocessing directive `#include`, less any source lines skipped by any of the conditional inclusion preprocessing directives, is called a *translation unit*.

type (§3.1.2.5) The meaning of a value stored in an object or returned by a function is determined by the *type* of the expression used to access it. (An identifier declared to be an object is the simplest such expression; the type is specified in the declaration of the identifier.) Types are partitioned into *object types* (types that describe objects), *function types* (types that describe functions), and *incomplete types* (types that describe objects but lack information needed to determine their sizes).

An array type of unknown size is an incomplete type. It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage). A structure or union type of unknown content is an incomplete type. It is completed, for all declarations of that type, by declaring the same structure or union tag with its defining content later in the same scope.

type qualifier (§3.5.3) The keywords `const` and `volatile` are *type qualifiers*.

type specifier (§3.5.2) A *type specifier* is a structure or union specifier, enumeration specifier, typedef name, or any legal combination of the keywords `void`, `char`, `short`, `int`, `long`, `float`, `double`, `signed`, or `unsigned`.

[Cforall adds the `type`, `dtype`, and `ftype` keywords and the `forall` specifier as new type specifiers.]

visible (§3.1.2.1) An identifier is *visible* (i.e. can be used) only within a region of program text called its *scope*.

Chapter 2

Lexical Elements

2.1 Keywords

Syntax

keyword: ...
```
forall
lvalue
spec
dtype
ftype
type
```

2.2 Identifiers

Cforall allows operator overloading by associating operators with special function identifiers. Furthermore, the constants "`0`" and "`1`" have special status for many of C's data types (and for many programmer-defined data types as well), so Cforall treats them as overloadable identifiers as well. Programmers can use these identifiers to declare functions and objects that implement operators and constants for their own types.

2.2.1 Constant Identifiers

Syntax

identifier: ...
```
0
1
```

The tokens "`0`" and "`1`" are identifiers. No other tokens defined by the rules for integer constants are considered to be identifiers.

□ Why "`0`" and "`1`"? Those integers have special status in C. All scalar types can be incremented and decremented, which is defined in terms of adding or subtracting 1. The operations "`&&`", "`||`", and "`!`" can be applied to any scalar arguments, and are defined in terms of comparison against 0. A *constant-expression* that evaluates to 0 is effectively compatible with every pointer type.

In C, the integer constants 0 and 1 suffice because the integral promotion rules can convert them to any arithmetic type, and the rules for pointer expressions treat constant expressions evaluating to 0 as a special case. However, user-defined arithmetic types often need the equivalent of a 1 or 0 for their functions or operators, polymorphic functions often need 0 and 1 constants of a type matching their polymorphic parameters, and user-defined pointer-like types may need a null value. Defining special constants for a user-defined type is more efficient than defining a conversion to the type from `int` that checks that its argument is 0 or 1, and simpler than extending the language with a `bit` type that can only take on those values and allowing the programmer to define a conversion from `bit`.

Why *just* "`0`" and "`1`"? Why not other integers? No other integers have special status in C. A facility that let programmers declare specific constants—"`const Complex 12`", for instance—would not be much of an improvement. Some facility for defining the creation of values of programmer-defined types from arbitrary integer tokens would be needed. The complexity of such a feature doesn't seem worth the gain. □

2.2.2 Operator Identifiers

Table 2.1 lists the programmer-definable operator identifiers and the operations they are associated with. Functions that are declared with (or pointed at by function pointers that are declared with) these identifiers can be called by expressions that use the operator tokens and syntax, or the operator identifiers and "function call" syntax. The relationships between operators and function calls are discussed in descriptions of the operators.

<table>
<tr><td>?[?]</td><td>subscripting</td><td>?%?</td><td>remainder</td><td>?^?</td><td>exclusive OR</td></tr>
<tr><td>?()</td><td>function call</td><td>?+?</td><td>addition</td><td>?|?</td><td>inclusive OR</td></tr>
<tr><td>?++</td><td>postfix increment</td><td>?-?</td><td>subtraction</td><td>?=?</td><td>simple assignment</td></tr>
<tr><td>?--</td><td>postfix decrement</td><td>?<<?</td><td>left shift</td><td>?*=?</td><td>multiplication assignment</td></tr>
<tr><td>++?</td><td>prefix increment</td><td>?>>?</td><td>right shift</td><td>?/=?</td><td>division assignment</td></tr>
<tr><td>--?</td><td>prefix decrement</td><td>?<?</td><td>less than</td><td>?%=?</td><td>remainder assignment</td></tr>
<tr><td>*?</td><td>dereference</td><td>?<=?</td><td>less than or equal</td><td>?+=?</td><td>addition assignment</td></tr>
<tr><td>+?</td><td>unary plus</td><td>?>=?</td><td>greater than or equal</td><td>?-=?</td><td>subtraction assignment</td></tr>
<tr><td>-?</td><td>arithmetic negation</td><td>?>?</td><td>greater than</td><td>?<<=?</td><td>left-shift assignment</td></tr>
<tr><td>~?</td><td>bitwise negation</td><td>?==?</td><td>equality</td><td>?>>=?</td><td>right-shift assignment</td></tr>
<tr><td>!?</td><td>logical complement</td><td>?!=?</td><td>inequality</td><td>?&=?</td><td>bitwise AND assignment</td></tr>
<tr><td>?*?</td><td>multiplication</td><td>?&?</td><td>bitwise AND</td><td>?^=?</td><td>exclusive OR assignment</td></tr>
<tr><td>?/?</td><td>division</td><td></td><td></td><td>?|=?</td><td>inclusive OR assignment</td></tr>
</table>

Table 2.1: Operator Identifiers

□ Operator identifiers are made up of the characters of the operator token, with question marks added to mark the positions of the arguments of operators. The question marks serve as mnemonic devices; programmers can not create new operators by arbitrarily mixing question marks and other non-alphabetic characters. Note that prefix and postfix versions of the increment and decrement operators are distinguished by the position of the question mark. □

□ The use of "`?`" in identifiers means that some C programs are not Cforall programs. For instance, the sequence of characters "`(i < 0)?-i:i`" is legal in a C program, but a Cforall compiler will detect a syntax error because it will treat "`?-`" as an identifier, not as the two tokens "`?`" and "`-`". □

□ Certain operators can *not* be defined by the programmer:

- The logical operators "`&&`" and "`||`", and the conditional operator "`?:`". These operators do not always evaluate their operands, and hence can not be properly defined by functions unless some mechanism like call-by-name is added to the language. Note that the definitions of "`&&`" and "`||`" say that they work by checking that their arguments are unequal to 0, so defining "`!=`" and "`0`" for user-defined types is enough to allow them to be used in logical expressions.
- The comma operator. It is a control-flow operator like those above. Changing its meaning seems pointless and confusing.
- The "address of" operator. It seems useful to be able to define a unary "`&`" operator that returns values of some programmer-defined pointer-like type. The problem lies with the type of the operator. Consider the expression "`p = &x`", where `x` is of type `T` and `p` has the programmer-defined type `T_ptr`. The expression might be treated as a call to the unary function "`&?`". Now what is the type of the function's parameter? It can not be `T`, because then `x` would be passed by value, and there is no way to create a useful pointer-like result from a value. Hence the parameter must have type `T*`. But then the expression must be rewritten as "`p = &?( &x )`"—which doesn't seem like progress!

  The rule for address-of expressions would have to be something like "keep applying address-of functions until you get one that takes a pointer argument, then use the built-in operator and stop". It seems simpler to define a conversion function from `T*` to `T_ptr`.
- The `sizeof` operator. It is already defined for every object type, and intimately tied into the language's storage allocation model. Redefining it seems pointless.
- The "member of" operators "`.`" and "`->`". These are not really infix operators, since their right "operand" is not a value or object.
- Cast operators. Anything that can be done with an explicit cast can be done with a function call. The difference in syntax is small.

□

2.2.3 Scopes of Identifiers

Cforall's scope rules differ from C's in one major respect: a declaration of an identifier may overload outer declarations of lexically identical identifiers in the same name space, instead of hiding them. The outer declaration is hidden if the two declarations have compatible type, or if one declares an array type and the other declares a pointer type and the element type and pointed-at type are compatible, or if one has function type and the other is a pointer to a compatible function type, or if one declaration is a `type` or `typedef` declaration and the other is not. The outer declaration becomes visible when the scope of the inner declaration terminates.

□ Hence, a Cforall program can declare an `int v` and a `float v` in the same scope; a C++ program can not.

Note that enumeration constants all exist in the name space of ordinary identifiers, and all have type `int`, so one identifier can not be used for two enumeration constants in the same scope. In other words, `orange` can not be a constant from `enum color` and `enum fruit` at the same time. □

2.2.4 Linkage of Identifiers

Cforall's linkage rules differ from C's in only one respect: instances of a particular identifier with external or internal linkage do not necessarily denote the same object or function. Instead, in the set of translation units and libraries that constitutes an entire program, any two instances of a particular identifier with external linkage denote the same object or function if they have compatible types, or if one declares an array type and the other declares a pointer type and the element type and pointed-at type are compatible, or if one has function type and the other is a pointer to a compatible function type. Within one translation unit, each instance of an identifier with internal linkage denotes the same object or function in the same circumstances. Identifiers with no linkage always denote unique entities.

> □ A Cforall program can declare an `extern int v` and an `extern float v`; A C program cannot. □

Chapter 3

Conversions

Cforall defines situations where values of one type are automatically converted to another type. These conversions are called *implicit conversions*. The programmer can request *explicit conversions* using cast expressions.

3.1 Other Operands

3.1.1 Anonymous Members

If an expression has a structure or union type that has an anonymous member, it can be converted implicitly or explicitly to the anonymous member's type. The result of the conversion is the anonymous member of the converted expression, and is an lvalue if the converted expression was.

If an expression's type is a pointer to a structure or union type that has an anonymous member, it can be converted implicitly or explicitly to a pointer to the anonymous member's type. The result of the conversion is a pointer to the anonymous member.

Examples

```
struct point {
    int x, y;
};
void move_by(struct point* p1, struct point p2) {
    p1->x += p2.x;
    p1->y += p2.y;
}

struct color_point {
    enum { RED, BLUE, GREEN } color;
    struct point;
} cp1, cp2;
move_to(&cp1, cp2);
```

Thanks to implicit conversion, the two arguments that `move_by()` receives are a pointer to `cp1`'s second member and a copy of `cp2`'s second member.

3.1.2 Specialization

A function or value whose type is polymorphic may be converted to one whose type is less polymorphic by binding values to one or more of its inferred parameters. Any value that is legal for the inferred parameter may be used, including other inferred parameters.

If, after the inferred parameter binding, an assertion parameter has no inferred parameters in its type, then an object or function must be visible at the point of the specialization that has the same identifier as the assertion parameter and has a type that is compatible with or can be specialized to the type of the assertion parameter. The assertion parameter is bound to that object or function.

The type of the specialization is the type of the original with the bound inferred parameters and the bound assertion parameters replaced by their bound values.

Examples
The type

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

can be specialized to (among other things)

```
forall(type T) void (*)(T, T);    /* U bound to T */
forall(type T) void (*)(T, real); /* U bound to real */
forall(type U) void (*)(real, U); /* T bound to real */
void f(real, real);               /* both bound to real */
```

The type

```
forall(type T | T ?+?(T,T)) T (*)(T);
```

can be specialized to (among other things)

```
int (*)(int);                   /* T bound to int, and */
                                /* T ?+?(T,T) bound to int ?+?(int,int) */
```

3.2 Safe and Unsafe Conversions

In C, a pattern of conversions known as the *usual arithmetic conversions* is used with most binary arithmetic operators to convert the operands to a common type and determine the type of the operator's result. In Cforall, these and other conversions play a role in overload resolution, and collectively are called the *safe conversions*.

The following conversions are *direct* safe arithmetic conversions.

- A `char`, a `short int`, an `int` bit-field, or their signed or unsigned varieties, or an enumeration type, may undergo an integral promotion.
- An `int` may be converted to an `unsigned int`, and to a `long int`.
- An `unsigned int` may be converted to a `long unsigned int`. If a `long int` can represent all values of an `unsigned int`, then an `unsigned int` may also be converted to a `long int`.
- A `long int` may be converted to a `long unsigned int`.
- A `long unsigned int` may be converted to a `float`.
- A `float` may be converted to a `double`.

- A `double` may be converted to a `long double`.

Furthermore, if type `T` can be converted to type `U` by a safe arithmetic conversion and type `U` can be converted to type `V` by a safe arithmetic conversion, then the conversion from `T` to type `V` is an *indirect* safe arithmetic conversion.

A *direct safe conversion* is a direct safe arithmetic conversion, or one of the following conversions:

- from any object type or incomplete type to `void`;
- from a pointer to any non-`void` type to a pointer to `void`;
- from a pointer to any type to a pointer to a more qualified version of the type;
- from a structure or union type to the type of an anonymous member of the structure or union;
- from a pointer to a structure or union type to a pointer to the type of an anonymous member of the structure or union;
- within the scope of an initialized type declaration, conversions between a type and its implementation or between a pointer to a type and a pointer to its implementation.

Conversions that are not safe conversions are *unsafe conversions*.

□ As in C, there is an implicit conversion from `void*` to any pointer type. This is clearly dangerous, and C++ does not have this implicit conversion. Cforall keeps it, in the interest of remaining as pure a superset of C as possible, but discourages it by making it unsafe. □

3.3 Conversion Cost

The *conversion cost* of a safe conversion[1] is a measure of how desirable or undesirable it is. It is defined as follows.

- The cost of a conversion from any type to itself is 0.
- The cost of a direct safe conversion is 1.
- The cost of an indirect safe arithmetic conversion is the smallest number of direct conversions needed to make up the conversion.

Examples

The cost of an implicit conversion from `int` to `long` is 1.

The cost of an implicit conversion from `long` to `double` is 3, because it is defined in terms of conversions from `long` to `unsigned long`, then to `float`, and then to `double`.

If `int` can represent all the values of `unsigned short`, then the cost of an implicit conversion from `unsigned short` to `unsigned` is 2: `unsigned short` to `int` to `unsigned`. Otherwise, `unsigned short` is converted directly to `unsigned`, and the cost is 1.

If `long` can represent all the values of `unsigned`, then the conversion cost of `unsigned` to `long` is 1. Otherwise, the conversion is an unsafe conversion, and its conversion cost is undefined.

[1] Unsafe conversions do not have defined conversion costs.

Chapter 4

Expressions

Cforall allows operators and identifiers to be overloaded. Hence, each expression can have a number of *interpretations*, each of which has a different type. The interpretations that are potentially executable are called *valid interpretations*. The set of interpretations depends on the kind of expression and on the interpretations of the subexpressions that it contains. The rules for determining the valid interpretations of an expression are discussed below for each kind of expression. Eventually the context of the outermost expression chooses one interpretation of that expression to be executed.

An *ambiguous interpretation* is an interpretation which does not specify the exact object or function denoted by every identifier in the expression. An expression can have some interpretations that are ambiguous and others that are unambiguous. An expression that is chosen to be executed shall not be ambiguous.

The *best valid interpretations* are the valid interpretations that use the fewest unsafe conversions. Of these, the best are those where the functions and objects involved are the least polymorphic. Of these, the best have the lowest total conversion cost, including all implicit conversions in the argument expressions. Of these, the best have the highest total conversion cost for the implicit conversions (if any) applied to the argument expressions. If there is no single best valid interpretation, or if the best valid interpretation is ambiguous, then the resulting interpretation is ambiguous.

□ Cforall's rules for selecting the best interpretation are designed to allow overload resolution to mimic C's operator semantics. In C, the "usual arithmetic conversions" are applied to the operands of binary operators if necessary to convert the operands to a common type (roughly speaking, the "smallest" type that can hold both arguments). In Cforall, those conversions are "safe". The "fewest unsafe conversions" rule ensures that the usual conversions are done, if possible. The "lowest total expression cost" rule chooses the proper common type. The odd-looking "highest argument conversion cost" rule ensures that, when unary expressions must be converted, conversions of function results are preferred to conversion of function arguments: `(double)-i` will be preferred to `-(double)i`.

The "least polymorphic" rule reduces the number of polymorphic function calls, since such functions are presumably more expensive than monomorphic functions and since the more specific function is presumably more appropriate. It also gives preference to monomorphic values (such as the `int 0`) over polymorphic values (such as the null pointer `0`). However, interpretations that call polymorphic functions are preferred to interpretations that perform unsafe conversions, because those conversions potentially lose accuracy or violate strong typing.

There are two notable differences between Cforall's overload resolution rules and the rules for C++ defined in [5]. First, the result type of a function plays a role. In C++, a function call must be completely resolved based on the arguments to the call in most circumstances. In Cforall, a function call may have several interpretations, each with a different result type, and the interpretations of the containing context choose among them. Second, safe conversions are used to choose among interpretations of all sorts of functions; in C++, the "usual arithmetic conversions" are a separate set of rules that apply only to the built-in operators. □

Expressions involving certain operators are considered to be equivalent to function calls. A transformation from "operator" syntax to "function call" syntax is defined by *rewrite rules*. Each operator has a set of predefined functions that overload its identifier. Overload resolution determines which member of the set is executed in a given expression. The functions have internal linkage and are implicitly declared with file scope. The predefined functions and rewrite rules are discussed below for each of these operators.

□ Predefined functions and constants have internal linkage because that simplifies optimization in traditional compile-and-link environments. For instance, "`an_int + an_int`" is equivalent to "`?+?(an_int, an_int)`". If integer addition has not been redefined in the current scope, a compiler can generate code to perform the addition directly. If predefined functions had external linkage, this optimization would be difficult. □

□ Since each subsection describes the interpretations of an expression in terms of the interpretations of its subexpressions, this chapter can be taken as describing an overload resolution algorithm that uses one bottom-up pass over an expression tree. Such an algorithm was first described (for Ada) by Baker [2]. It is extended here to handle polymorphic functions and arithmetic conversions. The overload resolution rules and the predefined functions have been chosen so that, in programs that do not introduce overloaded declarations, expressions will have the same meaning in C and in Cforall. □

□ Expression syntax is quoted from the ANSI C standard. The syntax itself defines the precedence and associativity of operators. The sections are arranged in decreasing order of precedence, with all operators in a section having the same precedence. □

4.1 Primary Expressions

Syntax

primary-expression:
 identifier
 constant
 string-literal
 (*expression*)

Predefined Identifiers

```
const int                1;
const int                0;
forall(dtype DT) DT *const 0;
forall(ftype FT) FT *const 0;
```

Semantics

The valid interpretations of an *identifier* are given by the visible declarations of the identifier. A *constant* or *string-literal* has one valid interpretation, which has the type and value defined by C. A parenthesised expression has the same interpretations as the contained *expression*.

The predefined integer identifiers "`1`" and "`0`" have the integer values 1 and 0, respectively. The other two predefined "`0`" identifiers are bound to polymorphic pointer values that, when specialized with a data type or function type respectively, produce a null pointer of that type.

Examples

The expression `(void*)0` specializes the (polymorphic) null pointer to a null pointer to `void`. `(const void*)0` does the same, and also uses a safe conversion from `void*` to `const void*`. In each case, the null pointer conversion is better than the unsafe conversion of the integer `0` to a pointer.

□ Note that the predefined identifiers have addresses.

Cforall does not have C's concept of "null pointer constants", which are not typed values but 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. Cforall handles these cases through overload resolution and specialization. □

4.2 Postfix Operators

Syntax

postfix-expression:
  primary-expression
  postfix-expression `[` *expression* `]`
  postfix-expression `(` *argument-expression-list*$_{opt}$ `)`
  postfix-expression `.` *identifier*
  postfix-expression `->` *identifier*
  postfix-expression `++`
  postfix-expression `--`
argument-expression-list:
  assignment-expression
  argument-expression-list `,` *assignment-expression*

Rewrite Rules

`a[b]` ⇒ `?[?](a, b)`
`a(` *arguments* `)` ⇒ `?()(a,` *arguments* `)`
`a++` ⇒ `?++(&(a))`
`a--` ⇒ `?--(&(a))`

□ Note that "`++`" and "`--`" are rewritten as function calls that are given a pointer to that operand. (This is true of all operators that modify an operand.) As Hamish Macdonald has pointed out, this forces the modified operand of such expressions to be an lvalue. This partially enforces the C semantic rule that such operands must be *modifiable* lvalues.

Subscript expressions are rewritten as function calls that pass the first parameter by value. This is somewhat unfortunate, since array-like types tend to be large. The alternative is to use the rewrite rule "`a[b]` ⇒`?[?](&(a), b)`". However, C semantics forbid this approach: the `a` in "`a[b]`" can be an arbitrary pointer value, which does not have an address. □

Predefined Identifiers

```
int           ?++(int*),           ?--(int*);
unsigned      ?++(unsigned*),      ?--(unsigned*);
long          ?++(long*),          ?--(long*);
long unsigned ?++(long unsigned*), ?--(long unsigned*);
float         ?++(float*),         ?--(float*);
double        ?++(double*),        ?--(double*);
long double   ?++(long double*),   ?--(long double*);

forall(type T) T*                ?++(T**),                ?--(T**);
forall(type T) const T*          ?++(const T**),          ?--(const T**);
forall(type T) volatile T*       ?++(volatile T**),       ?--(volatile T**);
forall(type T) const volatile T* ?++(const volatile T**), ?--(const volatile T**);

forall(type T) lvalue T                ?[?](T*,                ptrdiff_t);
forall(type T) const lvalue T          ?[?](const T*,          ptrdiff_t);
forall(type T) volatile lvalue T       ?[?](volatile T*,       ptrdiff_t);
forall(type T) const volatile lvalue T ?[?](const volatile T*, ptrdiff_t);
```

For every complete enumerated type E there exist

E `?++(`E`*), ?--(`E`*);`

□ In C, a semantic rule requires that pointer operands of increment and decrement be pointers to object types. Hence, `void*` objects cannot be incremented. In Cforall, the restriction follows from the use of a `type` parameter in the predefined function definitions, as opposed to `dtype`, since only object types can be inferred arguments corresponding to the type parameter `T`. □

4.2.1 Function Calls

Semantics

A *function designator* is an interpretation of an expression that has function type. The *postfix-expression* in a function call may have some interpretations that are function designators and some that are not.

For those interpretations of the *postfix-expression* that are not function designators, the expression is rewritten and becomes a call of a function named "`?()`". The valid interpretations of the rewritten expression are determined in the manner described below.

Each combination of function designators and argument interpretations is considered. For those interpretations of the *postfix-expression* that are monomorphic function designators, the combination has a valid interpretation if the function designator accepts the number of arguments given, and each argument interpretation matches the corresponding explicit parameter:

- if the argument corresponds to a parameter in the function designator's prototype, the argument interpretation must have the same type as the corresponding parameter, or be implicitly convertible to the parameter's type
- if the function designator's type does not include a prototype or if the argument corresponds to "..." in a prototype, a default argument promotion is applied to it.

The type of the valid interpretation is the return type of the function designator.

For those combinations where the interpretation of the *postfix-expression* is a polymorphic function designator and the function designator accepts the number of arguments given, there shall be at least one set of *implicit arguments* for the implicit parameters such that

- If the declaration of the implicit parameter uses type-class `type`, the implicit argument must be an object type; if it uses `dtype`, the implicit argument must be an object type or an incomplete type; and if it uses `ftype`, the implicit argument must be a function type.
- if an explicit parameter's type uses any implicit parameters, then the corresponding explicit argument must have a type that is (or can be safely converted to) the type produced by substituting the implicit arguments for the implicit parameters in the explicit parameter type.
- the remaining explicit arguments must match the remaining explicit parameters, as described for monomorphic function designators.
- for each assertion parameter in the function designator's type, there must be an object or function with the same identifier that is visible at the call site and whose type is compatible with or can be specialized to the type of the assertion declaration.

There is a valid interpretation for each such set of implicit parameters. The type of each valid interpretation is the return type of the function designator with implicit parameter values substituted for the implicit arguments.

A valid interpretation is ambiguous if the function designator or any of the argument interpretations is ambiguous.

Every valid interpretation whose return type is not compatible with any other valid interpretation's return type is an interpretation of the function call expression.

Every set of valid interpretations that have mutually compatible result types also produces an interpretation of the function call expression. The type of the interpretation is the composite type of the types of the valid interpretations, and the value of the interpretation is that of the best valid interpretation.

□ One desirable property of a polymorphic programming language is *generalizability*: the ability to replace an abstraction with a more general but equivalent abstraction without requiring changes in any of the uses of the original[3]. For instance, it should be possible to replace a function "`int f(int);`" with "`forall(type T) T f(T);`" without affecting any calls of `f`.

Cforall does not fully possess this property, because unsafe conversions are not done when arguments are passed to polymorphic parameters. Consider

```
float  g(float, float);
int    i;
float  f;
double d;
f = g(f, f);  /* (1) */
f = g(i, f);  /* (2) (safe conversion to float) */
f = g(d, f);  /* (3) (unsafe conversion to float) */
```

If `g` was replaced by "`forall(type T) T g(T,T);`", the first and second calls would be unaffected, but the third would change: `f` would be converted to `double`, and the result would be a `double`.

Another example is the function "`void h(int*);`". This function can be passed a `void*` argument, but the generalization "`forall(type T) void h(T*);`" can not. In this case, `void` is not a valid value for `T` because it is not an object type. If unsafe conversions were allowed, `T` could be inferred to be *any* object type, which is undesirable. □

Examples

A function called "`?()`" might be part of a numerical differentiation package.

```
extern type       Derivative;
extern double     ?()(Derivative, double);
extern Derivative derivative_of( double (*f)(double) );
extern double     sin(double);

Derivative sin_dx = derivative_of(sin);
double     d;
d = sin_dx(12.9);
```

Here, the only interpretation of `sin_dx` is as an object of type `Derivative`. For that interpretation, the function call is treated as "`?()(sin_dx, 12.9)`".

```
int f(long);        /* (1) */
int f(int, int);    /* (2) */
int f(int*);        /* (3) */

int i = f(5);       /* calls (1) */
```

Function (1) provides a valid interpretation of "`f(5)`", using an implicit `int` to `long` conversion. The other functions do not, since the second requires two arguments, and since there is no implicit conversion from `int` to `int*` that could be used with the third function.

```
forall(type T) T h(T);
double d = h(1.5);
```

"`1.5`" is a `double` constant, so `T` is inferred to be `double`, and the result of the function call is a `double`.

```
forall(type T, type U) void g(T,U);     /* (4) */
forall(type T)         void g(T,T);     /* (5) */
forall(type T)         void g(T,long);  /* (6) */
void   g(long, long);                   /* (7) */
double d;
int    i;
int*   p;

g(d,d);         /* calls (5) */
g(d,i);         /* calls (6) */
g(i,i);         /* calls (7) */
g(i,p);         /* calls (4) */
```

The first call has valid interpretations for all four versions of `g`. (6) and (7) are discarded because they involve unsafe `double`-to-`long` conversions. (5) is chosen because it is less polymorphic than (4).

For the second call, (7) is again discarded. Of the remaining interpretations for (4), (5), and (6) (with `i` converted to `long`), (6) is chosen because it is the least polymorphic.

The third call has valid interpretations for all of the functions; (7) is chosen since it is not polymorphic at all.

The fourth call has no interpretation for (5), because its arguments must have compatible type. (4) is chosen because it does not involve unsafe conversions.

```
forall(type T) T min(T,T);
double           max(double, double);
```

```
spec min_max(T) {
    T min(T,T);
    T max(T,T);
}
forall(type U | min_max(U)) void shuffle(U,U);
shuffle(9, 10);
```

The only possibility for `U` is `double`, because that is the type used in the only visible `max` function. 9 and 10 must be converted to `double`, and `min` must be specialized with `T` bound to `double`.

```
extern void q(int);      /* (8) */
extern void q(void*);    /* (9) */
extern void r();
q(0);
r(0);
```

The `int 0` could be passed to (8), or the `(void*)` specialization of the null pointer `0` could be passed to (9). The former is chosen because the `int 0` is less polymorphic. For the same reason, `int 0` is passed to `r()`, even though it has *no* declared parameter types.

4.2.2 Postfix Increment and Decrement Operators

Semantics

First, each interpretation of the operand of an increment or decrement expression is considered separately. For each interpretation that is a bit-field, the expression has one valid interpretation, with the type of the operand, and the expression is ambiguous if the operand is.

For each interpretation of the operand that is not a bit-field, the expression is rewritten, and the interpretations of the expression are the interpretations of the corresponding function call. Finally, all interpretations of the expression produced for the different interpretations of the operand are combined to produce the interpretations of the expression as a whole; where interpretations have compatible result types, the best interpretations are selected in the manner described for function call expressions.

> □ Increment and decrement expressions show up two deficiencies of Cforall's type system. First, there is no such thing as a pointer to a bit-field. Therefore, there is no way to define a function that alters a bit field argument, and hence no way to define increment and decrement functions for bit fields. As a result, the semantics of increment and decrement expressions must treat bit-fields specially. This holds true for all of the operators that may modify bit-fields.
>
> Second, type qualifiers are not included in type values, so polymorphic functions that take pointers to arbitrary types often come in four flavors, one for each possible qualification of the pointed-at type. □

4.2.3 Other Postfix Operators

Semantics

The interpretations of subscript expressions are the interpretations of the corresponding function call expressions.

In the member selection expression "$s.m$", there shall be at least one interpretation of s whose type is a structure type or union type containing a member named m. If two or more interpretations of s have members named m with mutually compatible types, then the expression has an ambiguous interpretation whose type is the composite type of the types of the members. If an interpretation of s has a member *m* whose type is not compatible with any other *s*'s *m*, then the expression has an interpretation with the member's type. The expression has no other interpretations.

The expression "*p*->*m*" has the same interpretations as the expression "(**p*).*m*".

4.3 Unary Operators

Syntax

unary-expression:
    postfix-expression
    `++` *unary-expression*
    `--` *unary-expression*
    unary-operator cast-expression
    `sizeof` *unary-expression*
    `sizeof (` *type-name* `)`

unary-operator: one of
    `& * + - ~ !`

Rewrite Rules

```
*a  ⇒ *?(a)
+a  ⇒ +?(a)
-a  ⇒ -?(a)
~a  ⇒ ~?(a)
!a  ⇒ !?(a)
++a ⇒ ++?(&(a))
--a ⇒ --?(&(a))
```

Predefined Identifiers

```
int           ++?(int*),           --?(int*);
unsigned      ++?(unsigned*),      --?(unsigned*);
long          ++?(long*),          --?(long*);
long unsigned ++?(long unsigned*), --?(long unsigned*);
float         ++?(float*),         --?(float*);
double        ++?(double*),        --?(double*);
long double   ++?(long double*),   --?(long double*);
forall(type T)  T*                ++?(T**),                --?(T**);
forall(type T)  const T*          ++?(const T**),          --?(const T**);
forall(type T)  volatile T*       ++?(volatile T**),       --?(volatile T**);
forall(type T)  const volatile T* ++?(const volatile T**), --?(const volatile T**);

forall(type T)  lvalue T                *?(T*);
forall(type T)  const lvalue T          *?(const T*);
forall(type T)  volatile lvalue T       *?(volatile T*);
forall(type T)  const volatile lvalue T *?(const volatile T*);
forall(ftype FT)                     FT *?(FT*);
```

```
int           +?(int),           -?(int),           ~?(int);
unsigned      +?(unsigned),      -?(unsigned),      ~?(unsigned);
unsigned long +?(unsigned long), -?(unsigned long), ~?(unsigned long);
long          +?(long),          -?(long),          ~?(long);
float         +?(float),         -?(float);
double        +?(double),        -?(double);
long double   +?(long double),   -?(long double);

int           !?(int), !?(unsigned), !?(unsigned long), !?(long),
              !?(float), !?(double), !?(long double);
forall(dtype DT) int !?(const volatile DT*);
forall(ftype FT) int !?(FT*);
```

For every complete enumerated type E there exist

```
E ++?(E*), --?(E*);
```

Constraints
The operand of `sizeof` shall not be `type`, `dtype`, or `ftype`.

Semantics
When the `sizeof` operator is applied to an expression, the expression shall have exactly one unambiguous interpretation. The `sizeof` expression has one interpretation, which is of the implementation-defined integral type `size_t` (defined in `<stddef.h>`).

When `sizeof` is applied to an identifier declared by a *type-declaration* or a *type-parameter*, it yields the size in bytes of the type that implements the operand. When the operand is an opaque type or an inferred type parameter, the expression is not a constant expression.

□

```
type   Pair = struct { int first, second; };
size_t p_size = sizeof(Pair);    /* constant expression */

extern type Complex;
size_t c_size = sizeof(Complex); /* non-constant expression */

forall(type T) T f(T p1, T p2) {
    size_t t_size = sizeof(T);   /* non-constant expression */
    /* ... */
    }
```

"`sizeof Complex`", although not statically known, is fixed. Within `f()`, "`sizeof(T)`" is fixed for each call of `f()`, but may vary from call to call. □

When the "`&`" operator is applied to an expression, the operand shall have exactly one unambiguous interpretation. The "`&`" expression has one interpretation which is of type `T*`, where `T` is the type of the operand.

The interpretations of prefix increment and decrement expressions are determined in the same way as the interpretations of postfix increment and decrement expressions.

The interpretations of other unary expression are the interpretations of the corresponding function call.

Examples

```
long li;
void eat_double(double);

eat_double(-li);   /* ⇒ eat_double( -?(li) ); */
```

The valid interpretations of "`-li`" are

interpretation	result type	expression conversion cost
`-?((int)li)`	`int`	(unsafe)
`-?((unsigned)li)`	`unsigned`	(unsafe)
`-?(li)`	`long`	0
`-?((long unsigned)li)`	`long unsigned`	1
`-?((float)li)`	`float`	2
`-?((double)li)`	`double`	3
`-?((long double)li)`	`long double`	4

The valid interpretations of the `eat_double` call, with the cost of the argument conversion and the cost of the entire expression, are

interpretation	argument cost	expression cost
`eat_double((double)-?((int)li))`	4	(unsafe)
`eat_double((double)-?((unsigned)li))`	3	(unsafe)
`eat_double((double)-?(li))`	3	$0 + 3 = 3$
`eat_double((double)-?((long unsigned)li))`	2	$1 + 2 = 3$
`eat_double((double)-?((float)li))`	1	$2 + 1 = 3$
`eat_double( -?((double)li))`	0	$3 + 0 = 3$
`eat_double((double)-?((long double)li))`	(unsafe)	(unsafe)

Each has result type `void`, so the best must be selected. The interpretations involving unsafe conversions are discarded. The remainder have equal expression conversion costs, so the "highest argument conversion cost" rule is invoked, and the chosen interpretation is `eat_double((double)-?(li))`.

4.4 Cast Operators

Syntax

cast-expression:
  unary-expression
  (*type-name*) *cast-expression*

Constraints
The *type-name* in a *cast-expression* shall not be `type`, `dtype`, or `ftype`.

Semantics
In a cast expression "`(`*type-name*`)`e", if *type-name* is the type of an interpretation of e, then that interpretation is the only interpretation of the cast expression; otherwise, e shall have some interpretation that can be converted to *type-name*, and the interpretation of the cast expression is the cast of the interpretation that can be converted at the lowest cost. The cast expression's interpretation is ambiguous if more than one interpretation can be converted at the lowest cost or if the selected interpretation is ambiguous.

□ Casts can be used to eliminate ambiguity in expressions by selecting interpretations of subexpressions, and to specialize polymorphic functions and values. □

4.5 Multiplicative Operators

Syntax

multiplicative-expression:
    cast-expression
    multiplicative-expression `*` *cast-expression*
    multiplicative-expression `/` *cast-expression*
    multiplicative-expression `%` *cast-expression*

Rewrite Rules

```
a * b ⇒ ?*?(a,b)
a / b ⇒ ?/?(a,b)
a % b ⇒ ?%?(a,b)
```

Predefined Identifiers

```
int           ?*?(int, int),                    ?/?(int, int),
              ?%?(int, int);
unsigned      ?*?(unsigned, unsigned),          ?/?(unsigned, unsigned),
              ?%?(unsigned, unsigned);
unsigned long ?*?(unsigned long, unsigned long), ?/?(unsigned long, unsigned long),
              ?%?(unsigned long, unsigned long);
long          ?*?(long, long),                  ?/?(long, long),
              ?%?(long, long);
float         ?*?(float, float),                ?/?(float, float);
double        ?*?(double, double),              ?/?(double, double);
long double   ?*?(long double, long double),    ?/?(long double, long double);
```

Semantics

The interpretations of multiplicative expressions are the interpretations of the corresponding function call.

Examples

```
int         i;
long        li;
void        eat_double(double);
eat_double( li % i );
```

"`li % i`" is rewritten as "`?%?(li,i)`". The valid interpretations of `?%?(li, i)`, their result types, and their conversion costs for the operators given above are

interpretation	result type	expression conversion cost
`?%?( (int)li, i)`	`int`	(unsafe)
`?%?( li, (long)i)`	`long`	1
`?%?( (unsigned)li, (unsigned)i)`	`unsigned`	(unsafe)
`?%?( (long unsigned)li, (long unsigned)i)`	`long unsigned`	3

The interpretations involving unsafe conversions are discarded. The costs of converting the others to `double` are 4 and 5, so the best interpretation of `eat_double(li, i)` is `eat_double((double)?%?(li, (long)i))`.

4.6 Additive Operators

Syntax

additive-expression:
multiplicative-expression
additive-expression `+` *multiplicative-expression*
additive-expression `-` *multiplicative-expression*

Rewrite Rules

```
a + b ⇒ ?+?(a,b)
a - b ⇒ ?-?(a,b)
```

Predefined Identifiers

```
int           ?+?(int,           int),           ?-?(int,           int);
long          ?+?(long,          long),          ?-?(long,          long);
unsigned      ?+?(unsigned,      unsigned),      ?-?(unsigned,      unsigned);
long unsigned ?+?(long unsigned, long unsigned), ?-?(long unsigned, long unsigned);
float         ?+?(float,         float),         ?-?(float,         float);
double        ?+?(double,        double),        ?-?(double,        double);
long double   ?+?(long double,   long double),   ?-?(long double,   long double);

forall(type T)  T*                 ?+?(T*,                 ptrdiff_t),
                                   ?+?(ptrdiff_t,          T*),
                                   ?-?(T*,                 ptrdiff_t);
forall(type T)  const T*           ?+?(const T*,           ptrdiff_t),
                                   ?+?(ptrdiff_t,          const T*),
                                   ?-?(const T*,           ptrdiff_t);
forall(type T)  volatile T*        ?+?(volatile T*,        ptrdiff_t),
                                   ?+?(ptrdiff_t,          volatile T*),
                                   ?-?(volatile T*,        ptrdiff_t);
forall(type T)  const volatile T*  ?+?(const volatile T*, ptrdiff_t),
                                   ?+?(ptrdiff_t, const volatile T*),
                                   ?-?(const volatile T*, ptrdiff_t);

forall(type T)  ptrdiff_t          ?-?(const volatile T*, const volatile T*);
```

Semantics
The interpretations of additive expressions are the interpretations of the corresponding function calls.

> □ `ptrdiff_t` is an implementation-defined identifier defined in <stddef.h> that is synonymous with a signed integral type that is large enough to hold the difference between two pointers. It seems reasonable to use it for pointer addition as well. (This is technically a difference between Cforall and C, which only specifies that pointer addition uses an *integral* argument.) Hence it is also used for subscripting, which is defined in terms of pointer addition. The ANSI C standard uses `size_t` in several cases where a library function takes an argument that is used as a subscript, but `size_t` is unsuitable here because it is an unsigned type. □

4.7 Bitwise Shift Operators

Syntax

shift-expression:
  additive-expression
  shift-expression `<<` *additive-expression*
  shift-expression `>>` *additive-expression*

Rewrite Rules

```
a << b ⇒ ?<<?(a,b)
a >> b ⇒ ?>>?(a,b)
```

Predefined Identifiers

```
int           ?<<?(int,           int),           ?>>?(int,           int);
long          ?<<?(long,          long),          ?>>?(long,          long);
unsigned      ?<<?(unsigned,      unsigned),      ?>>?(unsigned,      unsigned);
long unsigned ?<<?(long unsigned, long unsigned), ?>>?(long unsigned,long unsigned);
```

Semantics
The interpretations of a bitwise shift expression are the interpretations of the corresponding function calls.

4.8 Relational Operators

Syntax

relational-expression:
  shift-expression
  relational-expression `<` *shift-expression*
  relational-expression `>` *shift-expression*
  relational-expression `<=` *shift-expression*
  relational-expression `>=` *shift-expression*

Rewrite Rules

```
a < b  ⇒ ?<?(a,b)
a > b  ⇒ ?>?(a,b)
a <= b ⇒ ?<=?(a,b)
a >= b ⇒ ?>=?(a,b)
```

Predefined Identifiers

```
int ?<?(int,           int),           ?<=?(int,           int),
    ?>?(int,           int),           ?>=?(int,           int);
int ?<?(long,          long),          ?<=?(long,          long),
    ?>?(long,          long),          ?>=?(long,          long);
int ?<?(unsigned,      unsigned),      ?<=?(unsigned,      unsigned),
    ?>?(unsigned,      unsigned),      ?>=?(unsigned,      unsigned);
int ?<?(long unsigned, long unsigned), ?<=?(long unsigned, long unsigned),
    ?>?(long unsigned, long unsigned), ?>=?(long unsigned, long unsigned);
int ?<?(float,         float),         ?<=?(float,         float),
    ?>?(float,         float),         ?>=?(float,         float);
int ?<?(double,        double),        ?<=?(double,        double),
    ?>?(double,        double),        ?>=?(double,        double);
int ?<?(long double,   long double),   ?<=?(long double,   long double),
    ?>?(long double,   long double),   ?>=?(long double,   long double);

forall(dtype DT) int ?<?(const volatile DT*, const volatile DT*);
forall(dtype DT) int ?>?(const volatile DT*, const volatile DT*);
forall(dtype DT) int ?<=?(const volatile DT*, const volatile DT*);
forall(dtype DT) int ?>=?(const volatile DT*, const volatile DT*);
```

Semantics

The interpretations of a relational expression are the interpretations of the corresponding function call.

□ The type parameter DT is used for both parameters of the pointer comparison functions, and the use of **`dtype`** restricts the argument types to object types and incomplete types. This replaces C's semantic rules that the arguments of a pointer comparison must have the same object type or incomplete type. □

4.9 Equality Operators

Syntax

equality-expression:
 relational-expression
 equality-expression `==` *relational-expression*
 equality-expression `!=` *relational-expression*

Rewrite Rules

```
a == b ⇒ ?==?(a,b)
a != b ⇒ ?!=?(a,b)
```

Predefined Identifiers

```
int ?==?(int,           int),           ?!=?(int,           int);
int ?==?(long,          long),          ?!=?(long,          long);
int ?==?(unsigned,      unsigned),      ?!=?(unsigned,      unsigned);
int ?==?(long unsigned, long unsigned), ?!=?(long unsigned, long unsigned);
int ?==?(float,         float),         ?!=?(float,         float);
int ?==?(double,        double),        ?!=?(double,        double);
int ?==?(long double,   long double),   ?!=?(long double,   long double);

forall(dtype DT) int ?==?(const volatile DT*, const volatile DT*);
forall(dtype DT) int ?!=?(const volatile DT*, const volatile DT*);
forall(ftype FT) int ?==?(FT*, FT*);
forall(ftype FT) int ?!=?(FT*, FT*);

forall(dtype DT) int ?==?(const volatile DT*, const volatile void*);
forall(dtype DT) int ?==?(const volatile void*, const volatile DT*);
forall(dtype DT) int ?!=?(const volatile DT*, const volatile void*);
forall(dtype DT) int ?!=?(const volatile void*, const volatile DT*);

forall(dtype DT) int ?==?(const volatile DT*, forall(dtype DT2) const DT2*);
forall(dtype DT) int ?==?(forall(dtype DT2) const DT2*, const volatile DT*);
forall(dtype DT) int ?!=?(const volatile DT*, forall(dtype DT2) const DT2*);
forall(dtype DT) int ?!=?(forall(dtype DT2) const DT2*, const volatile DT*);
forall(ftype FT) int ?==?(FT*, forall(ftype FT2) FT2*);
forall(ftype FT) int ?==?(forall(ftype FT2) FT2*, FT*);
forall(ftype FT) int ?!=?(FT*, forall(ftype FT2) FT2*);
forall(ftype FT) int ?!=?(forall(ftype FT2) FT2*, FT*);
```

□ The three groups of polymorphic equality operations provide comparisons between any two pointers of the same type, between pointers to `void` and pointers to object types or incomplete types, and between the null pointer constant and pointers to any type. In the last case, a special C rule for null pointer constant operands has been replaced by a consequence of the Cforall type system. □

Semantics

The interpretations of an equality expression are the interpretations of the corresponding function call.

The result of an equality comparison between two pointers to predefined functions or predefined values is implementation-defined.

□ The implementation-defined status of equality comparisons allows implementations to use one library routine to implement many predefined functions. These optimization are particularly important when the predefined functions are polymorphic, as is the case for most pointer operations □

4.10 Bitwise AND Operator

Syntax

AND-expression:
  equality-expression
  AND-expression `&` *equality-expression*

Rewrite Rules

```
a & b ⇒ ?&?(a,b)
```

Predefined Identifiers

```
int           ?&?(int,           int);
long          ?&?(long,          long);
unsigned      ?&?(unsigned,      unsigned);
long unsigned ?&?(long unsigned, long unsigned);
```

Semantics
The interpretations of a bitwise AND expression are the interpretations of the corresponding function call.

4.11 Bitwise Exclusive OR Operator

Syntax

exclusive-OR-expression:
  AND-expression
  exclusive-OR-expression `^` *AND-expression*

Rewrite Rules

```
a ^ b ⇒ ?^?(a,b)
```

Predefined Identifiers

```
int           ?^?(int,           int);
long          ?^?(long,          long);
unsigned      ?^?(unsigned,      unsigned);
long unsigned ?^?(long unsigned, long unsigned);
```

Semantics
The interpretations of a bitwise exclusive OR expression are the interpretations of the corresponding function call.

4.12 Bitwise Inclusive OR Operator

Syntax

inclusive-OR-expression:
  exclusive-OR-expression
  inclusive-OR-expression `|` *exclusive-OR-expression*

Rewrite Rules

a | b ⇒ ?|?(a,b)

Predefined Identifiers

```
int           ?|?(int,           int);
long          ?|?(long,          long);
unsigned      ?|?(unsigned,      unsigned);
long unsigned ?|?(long unsigned, long unsigned);
```

Semantics
The interpretations of a bitwise inclusive OR expression are the interpretations of the corresponding function call.

4.13 Logical AND Operator

Syntax

logical-AND-expression:
 inclusive-OR-expression
 logical-AND-expression `&&` *inclusive-OR-expression*

Semantics
The operands of the expression "*a* `&&` *b*" are treated as "`(int)((`*a*`)!=0)`" and "`(int)((`*b*`)!=0)`", which shall both be unambiguous. The expression has only one interpretation, which is of type `int`.

□ When the operands of a logical expression are values of built-in types, and "`!=`" has not been redefined for those types, the compiler can optimize away the function calls.

A common C idiom omits comparisons to `0` in the controlling expressions of loops and `if` statements. For instance, the loop below iterates as long as `cp` points at a `Complex` value that is non-zero.

```
extern type          Complex;
extern const Complex 0;
extern int           ?!=?(Complex, Complex);
Complex *cp;

while (cp && *cp) { /* ... */ }
```

The logical expression calls the `Complex` inequality operator, passing it `*cp` and the `Complex 0`, and getting a 1 or 0 as a result. In contrast, C++ would apply a programmer-defined `Complex`-to-`int` conversion to `*cp` in the equivalent situation. The conversion to `int` would produce a general integer value, which is unfortunate, and possibly dangerous if the conversion was not written with this situation in mind. □

4.14 Logical OR Operator

Syntax

logical-OR-expression:
logical-AND-expression
logical-OR-expression `||` *logical-AND-expression*

Semantics
The operands of the expression "*a* `||` *b*" are treated as "`(int)((`*a*`)!=0)`" and "`(int)((`*b*`)!=0)`", which shall both be unambiguous. The expression has only one interpretation, which is of type `int`.

4.15 Conditional Operator

Syntax

conditional-expression:
logical-OR-expression
logical-OR-expression `?` *expression* `:` *conditional-expression*

Semantics
The conditional expression "*a*`?`*b*`:`*c*" is first treated as if it were the call "`cond((`*a*`)!=0,` *b*`,` *c*`)`", with `cond` declared as

```
long double      cond(int, long double, long double);
double           cond(int, double, double);
float            cond(int, float, float);
unsigned long    cond(int, unsigned long, unsigned long);
long             cond(int, long, long);
unsigned         cond(int, unsigned, unsigned);
int              cond(int, int, int);
forall(type T) T cond(int, T, T);
```

An interpretation of the conditional expression is ambiguous if the corresponding interpretation of the function call would be ambiguous.

If such a call would not have at least one interpretation, then the expression has one interpretation, with type `void`, and is interpreted as

```
(void)((int)((a)!=0) ?(void)(b) :(void)(c))
```

The interpretation is ambiguous if any of the rewritten argument expressions are ambiguous.

□ The object of the above is to bring the second and third operands to a common type. The monomorphic `cond` functions are defined for each of the arithmetic types so that conversion costs will be taken into account in selecting that type. The polymorphic version allows conditional expressions to have pointer, union, or structure types. □

4.16 Assignment Operators

Syntax

assignment-expression:
conditional-expression
unary-expression assignment-operator assignment-expression
assignment-operator: one of
`= *= /= %= += -= <<= >>= &= ^= |=`

Rewrite Rules

Let "←" be any of the assignment operators. Then

```
a ← b ⇒ ?←?( &(a), b)
```

Predefined Identifiers

```
char           ?=?(char*, char),
               ?=?(volatile char*, char);
unsigned char  ?=?(unsigned char*, unsigned char),
               ?=?(volatile unsigned char*, unsigned char);
signed char    ?=?(signed char*, signed char),
               ?=?(volatile signed char*, signed char);
short          ?=?(int*, int),
               ?=?(volatile int*, int);
unsigned short ?=?(unsigned int*, unsigned int),
               ?=?(volatile unsigned int*, unsigned int);
int            ?=?(int*, int),
               ?=?(volatile int*, int);
unsigned int   ?=?(unsigned int*, unsigned int),
               ?=?(volatile unsigned int*, unsigned int);
long           ?=?(long*, long),
               ?=?(volatile long*, long);
unsigned long  ?=?(unsigned long*, unsigned long),
               ?=?(volatile unsigned long*, unsigned long);
float          ?=?(float*, float),
               ?=?(volatile float*, float);
double         ?=?(double*, double),
               ?=?(volatile double*, double);
long double    ?=?(long double*, long double),
               ?=?(volatile long double*, long double);

/* Pointer assignment where the type pointed at by the left operand has all
   of the qualifiers or more of the type pointed at by the right operand. */
forall(ftype FT) FT*            ?=?(FT**, FT*),
                                ?=?(FT* volatile*, FT*);
forall(dtype DT) DT*            ?=?(DT**, DT*),
                                ?=?(DT* volatile*, DT*);
forall(dtype DT) const DT*      ?=?(const DT**, DT*),
                                ?=?(const DT* volatile*, DT*);
forall(dtype DT) volatile DT*   ?=?(volatile DT**, DT*),
                                ?=?(volatile DT* volatile*, DT*);
forall(dtype DT) const volatile DT*
                                ?=?(const volatile DT**, DT*),
                                ?=?(const volatile DT* volatile*, DT*),
                                ?=?(const volatile DT**, volatile DT*),
                                ?=?(const volatile DT* volatile*, volatile DT*),
                                ?=?(const volatile DT**, const DT*),
                                ?=?(const volatile DT* volatile*, const DT*);
```

```
/* Assignment between pointers to void and pointers to object types or
   incomplete types. */
forall(dtype DT) DT*
  ?=?(DT**, void*),                 ?=?(DT* volatile*, void*);
forall(dtype DT) const DT*
  ?=?(const DT**, void*),           ?=?(const DT* volatile*, void*),
  ?=?(const DT**, const void*),     ?=?(const DT* volatile*, const void*);
forall(dtype DT) volatile DT*
  ?=?(volatile DT**, void*),        ?=?(volatile DT* volatile*, void*),
  ?=?(volatile DT**, volatile void*), ?=?(volatile DT* volatile*, volatile void*);
forall(dtype DT) const volatile DT*
  ?=?(const volatile DT**,          void*),
  ?=?(const volatile DT* volatile*, void*),
  ?=?(const volatile DT**,          const void*),
  ?=?(const volatile DT* volatile*, const void*),
  ?=?(const volatile DT**,          volatile void*),
  ?=?(const volatile DT* volatile*, volatile void*),
  ?=?(const volatile DT**,          const volatile void*),
  ?=?(const volatile DT* volatile*, const volatile void*);
forall(dtype DT) void*
  ?=?(void**, DT*), ?=?(void* volatile*, DT*);
forall(dtype DT) const void*
  ?=?(const void**, DT*), ?=?(const void* volatile*, DT*),
  ?=?(const void**, const DT*), ?=?(const void* volatile*, const DT*);
forall(dtype DT) volatile void*
  ?=?(volatile void**, DT*), ?=?(volatile void* volatile*, DT*),
  ?=?(volatile void**, volatile DT*), ?=?(volatile void* volatile*, volatile DT*);
forall(dtype DT) const volatile void*
  ?=?(const volatile void**,          DT*),
  ?=?(const volatile void* volatile*, DT*),
  ?=?(const volatile void**,          const DT*),
  ?=?(const volatile void* volatile*, const DT*),
  ?=?(const volatile void**,          volatile DT*),
  ?=?(const volatile void* volatile*, volatile DT*),
  ?=?(const volatile void**,          const volatile DT*),
  ?=?(const volatile void* volatile*, const volatile DT*);

/* Assignment from null pointers to other pointer types. */
forall(dtype DT) DT*
  ?=?(DT**,                         forall(dtype DT2) const DT2*),
  ?=?(DT* volatile*,                forall(dtype DT2) const DT2*);
forall(dtype DT) const DT*
  ?=?(const DT**,                   forall(dtype DT2) const DT2*),
  ?=?(const DT* volatile*,          forall(dtype DT2) const DT2*);
forall(dtype DT) volatile DT*
  ?=?(volatile DT**,                forall(dtype DT2) const DT2*),
  ?=?(volatile DT* volatile*,       forall(dtype DT2) const DT2*);
forall(dtype DT) const volatile DT*
  ?=?(const volatile DT**,          forall(dtype DT2) const DT2*),
```

```
  ?=?(const volatile DT* volatile*, forall(dtype DT2) const DT2*);
forall(ftype FT) FT*
  ?=?(FT**,                         forall(ftype FT2) FT2*),
  ?=?(FT* volatile*,                forall(ftype FT2) FT2*);

forall(type T) T*
  ?+=?(T**, ptrdiff_t),                ?+=?(T* volatile*, ptrdiff_t),
  ?-=?(T**, ptrdiff_t),                ?-=?(T* volatile*, ptrdiff_t);
forall(type T) const T*
  ?+=?(const T**, ptrdiff_t),          ?+=?(const T* volatile*, ptrdiff_t),
  ?-=?(const T**, ptrdiff_t),          ?-=?(const T* volatile*, ptrdiff_t);
forall(type T) volatile T*
  ?+=?(volatile T**, ptrdiff_t),       ?+=?(volatile T* volatile*, ptrdiff_t),
  ?-=?(volatile T**, ptrdiff_t),       ?-=?(volatile T* volatile*, ptrdiff_t);
forall(type T) const volatile T*
  ?+=?(const volatile T**, ptrdiff_t), ?+=?(const volatile T* volatile*, ptrdiff_t),
  ?-=?(const volatile T**, ptrdiff_t), ?-=?(const volatile T* volatile*, ptrdiff_t);
```

For every complete structure or union type S there exist

```
S ?=?(S*, S), ?=?(S volatile*, S);
```

For every complete enumerated type E there exist

```
E ?=?(E*, int), ?=?(E volatile*, int);
```

□ The right-hand argument is `int` because the integral promotions convert enumeration constants and variables to `int` whenever they are used as expressions. □

Let "o=" be any of the compound assignment operators, and E be a complete enumerated type. Then there exist functions

```
E              ?o=?(E*,                       int),
               ?o=?(volatile E*,              int);
char           ?o=?(char*,                    char),
               ?o=?(volatile char*,           char);
signed char    ?o=?(signed char*,             signed char),
               ?o=?(volatile signed char*,    signed char);
unsigned char  ?o=?(unsigned char*,           unsigned char),
               ?o=?(volatile unsigned char*,  unsigned char);
short          ?o=?(int*,                     int),
               ?o=?(volatile int*,            int);
unsigned short ?o=?(unsigned*,                unsigned),
               ?o=?(volatile unsigned*,       unsigned);
int            ?o=?(int*,                     int),
               ?o=?(volatile int*,            int);
unsigned       ?o=?(unsigned*,                unsigned),
               ?o=?(volatile unsigned*,       unsigned);
long           ?o=?(long*,                    long),
               ?o=?(volatile long*,           long);
unsigned long  ?o=?(unsigned long*,           unsigned long),
               ?o=?(volatile unsigned long*,  unsigned long);
```

There also exist

```
float       ?*=?(float*,      float),       ?*=?(volatile float*,      float),
            ?/=?(float*,      float),       ?/=?(volatile float*,      float),
            ?+=?(float*,      float),       ?+=?(volatile float*,      float),
            ?-=?(float*,      float),       ?-=?(volatile float*,      float);
double      ?*=?(double*,     double),      ?*=?(volatile double*,     double),
            ?/=?(double*,     double),      ?/=?(volatile double*,     double),
            ?+=?(double*,     double),      ?+=?(volatile double*,     double),
            ?-=?(double*,     double),      ?-=?(volatile double*,     double);
long double ?*=?(long double*,long double), ?*=?(volatile long double*,long double),
            ?/=?(long double*,long double), ?/=?(volatile long double*,long double),
            ?+=?(long double*,long double), ?+=?(volatile long double*,long double),
            ?-=?(long double*,long double), ?-=?(volatile long double*,long double);
```

Semantics
The structure assignment functions provide member-wise assignment; each non-array member and each element of each array member of the right argument is assigned to the corresponding member or element of the left argument using the assignment function defined for its type. All other assignment functions have the same effect as the corresponding C assignment expression.

□ Note that, by default, union assignment uses C semantics—that is, bitwise copy—even if some of the union members have programmer-defined assignment functions. □

Each interpretation of the left operand of an assignment expression is considered separately. For each interpretation that is a bit-field, the expression has one valid interpretation, with the type of the left operand. The right operand is cast to that type, and the assignment expression is ambiguous if either operand is. For each interpretation of the left operand that is not a bit-field, the expression is rewritten, and the interpretations of the assignment expression are the interpretations of the corresponding function call. Finally, all interpretations of the expression produced for the different interpretations of the left operand are combined to produce the interpretations of the expression as a whole; where interpretations have compatible result types, the best interpretations are selected in the manner described for function call expressions.

4.17 Comma Operator

Syntax

expression:
  assignment-expression
  expression , *assignment-expression*

Semantics
In the comma expression "a, b", the first operand is interpreted as "`(void)(`a`)`", which shall be unambiguous. The interpretations of the expression are the interpretations of the second operand.

Chapter 5

Declarations

Syntax

> *declaration:* ...
> *type-declaration*
> *spec-definition*

Constraints
There shall be at most one declaration of an identifier with no linkage and compatible type in the same scope and in the same name space, except for tags (as specified in section 3.5.2.3 of the ANSI C standard).

An identifier declared by a type declaration shall not be redeclared as a parameter in a function definition whose declarator includes an identifier list.

□ This restriction echos ANSI C's ban on the redeclaration of typedef names as parameters (section 3.7). This avoids an ambiguity between old-style function declarations and new-style function prototypes:

```
void f(Complex, /* ... 3000 characters ... */);
void g(Complex, /* ... 3000 characters ... */)
    int Complex; {/* ... */}
```

Without the rule, `Complex` would be a type in the first case, and a parameter name in the second. □

5.1 Type Specifiers

Syntax

> *type-specifier:* ...
> *forall-specifier*

A *forall specifier* introduces implicit type parameterization into a function declaration.

5.1.1 Structure and Union Specifiers

Syntax

> *struct-declaration:* ...
> *specifier-qualifier-list*

Semantics

A *struct-declaration* with no *struct-declarator-list* declares an *anonymous member*.

If an anonymous member has a structure or union type, its members are accessible as if they were members of the containing structure or union.

Examples

```
struct point {
    int x, y;
};
struct color_point {
    enum { RED, BLUE, GREEN } color;
    struct point;
};
struct color_point cp;
cp.x = 0;
cp.color = RED;

struct literal {
    enum { NUMBER, STRING } tag;
    union {
        double n;
        char  *s;
    };
};
struct literal *next;
int            length;
extern int     strlen(const char*);
/* ... */
if (next->tag == STRING) length = strlen(next->s);
```

5.1.2 Forall Specifiers

Syntax

forall-specifier:
    `forall (` *type-parameter-list* `)`

Constraints

If the *declaration-specifiers* of a declaration that contains a *forall-specifier* declares a structure or union tag, the types of the members of the structure or union shall not use any of the type identifiers declared by the *type-parameter-list*.

□ This sort of declaration is illegal because the scope of the type identifiers ends at the end of the declaration, but the scope of the structure tag does not.

```
forall(type T) struct Pair { T a,b; };     /* illegal */
```

If an instance of `struct Pair` was declared later in the current scope, what would the members' type be? □

Semantics

The *type-parameter-lists* and assertions of the *forall-specifiers* declare a collection of type identifiers, function and object identifiers with no linkage.

If, in the declaration "`T D1`", `T` contains *forall-specifiers* and `D1` has the form

`D(`*parameter-type-list*`)`

then a type identifier declared by one of the *forall-specifiers* is an *inferred parameter* of the function declarator if and only if it is not an inferred parameter of a function declarator in `D`, and it is used in the type of a parameter in the following *type-parameter-list* or it and an inferred parameter are used as arguments of a specification in one of the *forall-specifiers*. The identifiers declared by assertions that use an inferred parameter of a function declarator are assertion parameters of that function declarator.

□ Since every inferred parameter is used by some parameter, inference can be done bottom-up. I could also argue that this constraint leads to more easily understood programs. □

If a function declarator is part of a function definition, its inferred parameters and assertion parameters have block scope; otherwise, identifiers declared by assertions have a *declaration scope*, which terminates at the end of the *declaration*.

A function type that has at least one inferred parameter is a *polymorphic function* type. Function types with no inferred parameters are *monomorphic function* types. One function type is *less polymorphic* than another if it has fewer inferred parameters, or if it has the same number of inferred parameters and fewer of its explicit parameters have types that depend on an inferred parameter.

The names of inferred parameters and the order of identifiers in forall specifiers are not relevant to polymorphic function type compatibility. Let f and g be two polymorphic function types with the same number of inferred parameters, and let f_i and g_i be the inferred parameters of f and g in their order of occurance in the function types' *parameter-type-lists*. Let f' be f with every occurrence of f_i replaced by g_i, for all i. Then f and g are compatible types if f''s and g's return types and parameter lists are compatible, and if for every assertion parameter of f' there is an assertion parameter in g with the same identifier and compatible type, and vice versa.

Examples

Consider these analogous monomorphic and polymorphic declarations.

```
int fi(int);
forall(type T) T fT(T);
```

`fi()` takes an `int` and returns an `int`. `fT()` takes a `T` and returns a `T`, for any type `T`.

```
int (*pfi)(int) = fi;
forall(type T) T (*pfT)(T) = fT;
```

`pfi` and `pfT` are pointers to functions. `pfT` is not polymorphic, but the function it points at is.

```
int (*fvpfi(void))(int) {
    return pfi;
}
forall(type T) T (*fvpfT(void))(T) {
    return pfT;
}
```

`fvpfi()` and `fvpfT()` are functions taking no arguments and returning pointers to functions. `fvpfT()` is monomorphic, but the function that its return value points at is polymorphic.

```
forall(type T)         int (*fTpfi(T))(int);
forall(type T)         T   (*fTpfT(T))(T);
forall(type T, type U) U   (*fTpfU(T))(U);
```

`fTpfi()` is a polymorphic function that returns a pointer to a monomorphic function taking an integer and returning an integer. It could return `pfi`. `fTpfT()` is subtle: it is a polymorphic function returning a *monomorphic* function taking and returning `T`, where `T` is an inferred parameter of `fTpfT()`. For instance, in the expression "`fTpfT(17)`", `T` is inferred to be `int`, and the returned value would have type `int (*)(int)`. "`fTpfT(17)(13)`" and "`fTpfT("yes")("no")`" are legal, but "`fTpfT(17)("no")`" is illegal. `fTpfU()` is polymorphic (in type `T`), and returns a pointer to a function that is polymorphic (in type `U`). "`f5(17)("no")`" is a legal expression of type `char*`.

```
forall(type T, type U, type V) U* f(T*, U, V* const);
forall(type U, type V, type W) U* g(V*, U, W* const);
```

The functions `f()` and `g()` have compatible types. Let f and g be their types; then $f_1 =$ `T`, $f_2 =$ `U`, $f_3 =$ `V`, $g_1 =$ `V`, $g_2 =$ `U`,, and $g_3 =$ `W`. Replacing every f_i by g_i in f gives

```
forall(type V, type U, type W) U* f(V*, U, W* const);
```

which has a return type and parameter list that is compatible with g.

□ The word "`type`" in a forall specifier is redundant at the moment, but I want to leave room for inferred parameters of ordinary types in case parameterized types get added one day.

Even without parameterized types, I might try to allow

```
forall(int n) int sum(int vector[n]);
```

but C currently rewrites array parameters as pointer parameters, so the effects of such a change require more thought. □

□ A polymorphic declaration must do two things: it must introduce type parameters, and it must apply assertions to those types. Adding this to existing C declaration syntax and semantics was delicate, and not entirely successful.

C depends on declaration-before-use, so a forall specifier must introduce type names before they can be used in the declaration specifiers. This could be done by making the forall specifier part of the declaration specifiers, or by making it a new introductory clause of declarations.

Assertions are also part of polymorphic function types, because it must be clear which functions have access to the assertion parameters declared by the assertions. All attempts to put assertions inside an introductory clause produced complex semantics and confusing code. Building them into the declaration specifiers could be done by placing them in the function's parameter list, or in a forall specifier that is a declaration specifier. Assertions are also used with type parameters of specifications, and by type declarations. For consistency's sake it seems best to attach assertions to the type declarations in forall specifiers, which means that forall specifiers must be declaration specifiers.

The chosen syntax led to the following question: what is the meaning of

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

There are two possibilities.

- `f` is polymorphic in `T`. This interpretation would allow the following.

```
forall(type T) T* alloc(void);
int *p = alloc();
```

Here `alloc()` would receive `int` as an inferred argument, and return an `int*`.

- `f` is monomorphic, and returns a polymorphic value. This facility is more limited than it seems at first. Consider `f()` above; the value returned must have type `T` for every object type `T`. Where is `f()` to get such a value? In practice, the only useful polymorphic return types are pointers to polymorphic function types and `forall(type T) const T*`—and in that case, the only returnable value is `0`.

Cforall took the second tack, so that inferred arguments would be apparent from the function's explicit arguments. Consequently, functions like `alloc` must be written as

```
forall(type T) T* alloc(T initial_value);
```

□

5.2 Type Qualifiers

Cforall defines a new type qualifier `lvalue`.

Syntax

type-qualifier: ...
    `lvalue`

Semantics

`lvalue` may be used to qualify the return type of a function type. Let `T` be an unqualified version of a type; then the result of calling a function with return type `lvalue T` is a modifiable lvalue of type `T`. `const` and `volatile` qualifiers may also be added to indicate that the function result is a constant or volatile lvalue.

□ The `const` and `volatile` qualifiers can only be sensibly used to qualify the return type of a function if the `lvalue` qualifier is also used. □

An lvalue-qualified type may be used in a cast expression if the operand is an lvalue; the result of the expression is an lvalue.

□ `lvalue` provides some of the functionality of C++'s "`T&`" (reference to object of type `T`) type. Reference types have four uses in C++.

- They are necessary for user-defined operators that return lvalues, such as "subscript" and "dereference".
- A reference can be used to define an alias for a complicated lvalue expression, as a way of getting some of the functionality of the Pascal `with` statement. The following C++ code gives an example.

  ```
  {   char& code = long_name.some_field[i].data->code;
      code = toupper(code);
  }
  ```

  This is not very useful.
- A reference parameter can be used to allow a function to modify an argument without forcing the caller to pass the address of the argument. This is most useful for user-defined assignment operators. In C++, plain assignment is done by a function called "`operator=`", and the two expressions

```
a = b;
operator=(a,b);
```

are equivalent. If `a` and `b` are of type `T`, then the first parameter of `operator=` must have type "`T&`". It cannot have type `T`, because then assignment couldn't alter the variable, and it can't have type "`T*`", because the assignment would have to be written "`&a = b;`".

In the case of user-defined operators, this could just as well be handled by using pointer types and by changing the rewrite rules so that "`a = b;`" is equivalent to "`operator=(&(a),b)`". Reference parameters of "normal" functions are Bad Things, because they remove a useful property of C function calls: an argument can only be modified by a function if it is preceded by "`&`".

- References to const-qualified types can be used instead of value parameters. Given the C++ function call "`fiddle(a_thing)`", where the type of `a_thing` is `Thing`, the type of `fiddle` could be either of

```
void fiddle(Thing);
void fiddle(const Thing&);
```

If the second form is used, then constructors and destructors are not invoked to create a temporary variable at the call site (and it is bad style for the caller to make any assumptions about such things), and within `fiddle` the parameter is subject to the usual problems caused by aliases. The reference form might be chosen for efficiency's sake if `Thing`s are too large or their constructors or destructors are too expensive. An implementation may switch between them without causing trouble for well-behaved clients. This leaves the implementor to define "too large" and "too expensive".

I propose to push this job onto the compiler by allowing it to implement

```
void fiddle(const volatile Thing);
```

with call-by-reference. Since it knows all about the size of `Thing`s and the parameter passing mechanism, it should be able to come up with a better definition of "too large", and may be able to make a good guess at "too expensive".

In summary, since references are only really necessary for returning lvalues, I'll only provide lvalue functions. □

5.3 Specification Definitions

Syntax

spec-definition:
  `spec` *identifier* `(` *type-parameter-list* `)` `{` *spec-declaration-list*$_{opt}$ `}`
spec-declaration-list:
  spec-declaration `;`
  spec-declaration-list spec-declaration `;`
spec-declaration:
  specifier-qualifier-list declarator-list
declarator-list:
  declarator
  declarator-list `,` *declarator*

□ The declarations allowed in a specification are much the same as those allowed in a structure, except that bit fields are not allowed, and incomplete types and function types are allowed. □

Semantics
A *specification definition* defines a name for a *specification*: a parameterized collection of object and function declarations.

The declarations in a specification consist of the declarations in the *spec-declaration-list* and declarations produced by any assertions in the *spec-parameter-list*. If the collection contains two declarations that declare the same identifier and have compatible types, they are combined into one declaration with the composite type constructed from the two types.

5.3.1 Assertions

Syntax

assertion-list:
  assertion
  assertion-list assertion
assertion:
  | *identifier* (*type-name-list*)
  | *spec-declaration*
type-name-list:
  type-name
  type-name-list , *type-name*

Constraints
The *identifier* in an assertion that is not a *spec-declaration* shall be the name of a specification. The *type-name-list* shall contain one *type-name* argument for each *type-parameter* in that specification's *spec-parameter-list*. If the *type-parameter* uses type-class `type`, the argument shall be the type name of an object type; if it uses `dtype`, the argument shall be the type name of an object type or an incomplete type; and if it uses `ftype`, the argument shall be the type name of a function type.

Semantics
An *assertion* is a declaration of a collection of objects and functions, called *assertion parameters*.

The assertion parameters produced by an assertion that applies the name of a specification to type arguments are found by taking the declarations specified in the specification and treating each of the specification's parameters as a synonym for the corresponding *type-name* argument.

The collection of assertion parameters produced by the *assertion-list* are found by combining the declarations produced by each assertion. If the collection contains two declarations that declare the same identifier and have compatible types, they are combined into one declaration with the composite type constructed from the two types.

Examples

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

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

```
};
spec list_of(type List, type Element) {
    Element car(List);
    List    cdr(List);
    List    cons(Element, List);
    List    nil;
    int     is_nil(List);
};
spec sum_list(type List,
              type Element  | summable(Element) | list_of(List, Element)) {};
```

`sum_list` contains seven declarations, which describe a list whose elements can be added up. The assertion "`|sum_list(i_list, int)`" produces the assertion parameters

```
int       ?+=?(int*, int);
const int 0;
int       car(i_list);
i_list    cdr(i_list);
i_list    cons(int, i_list);
i_list    nil;
int       is_nil;
```

5.4 Type Declarations

Syntax

type-parameter-list:
  type-parameter
  type-parameter-list , *type-parameter*
type-parameter:
  type-class identifier assertion-list$_{opt}$
type-class:
  `type`
  `dtype`
  `ftype`
type-declaration:
  storage-class-specifier$_{opt}$ `type` *type-declarator-list* ;
type-declarator-list:
  type-declarator
  type-declarator-list , *type-declarator*
type-declarator:
  identifier assertion-list$_{opt}$ = *type-name*
  identifier assertion-list$_{opt}$

Constraints
If a type declaration has block scope, and the declared identifier has external or internal linkage, the declaration shall have no initializer for the identifier.

Semantics

A *type-parameter* or a *type-declarator* declares an identifier to be a type name for a type distinct from all other types.

An identifier declared by a *type-parameter* has no linkage. Identifiers declared with type-class `type` are object types; those declared with type-class `dtype` are incomplete types; and those declared with type-class `ftype` are function types. The identifier has block scope that terminates at the end of the *spec-declaration-list* or polymorphic function that contains the *type-parameter*.

A *type-declarator* with an initializer is a *type definition*. The declared identifier is an incomplete type within the initializer, and an object type after the end of the initializer. The type in the initializer is called the *implementation type*. Within the scope of the declaration, implicit conversions can be performed between the defined type and the implementation type, and between pointers to the defined type and pointers to the implementation type.

A type declaration without an initializer and without a storage-class specifier or with storage-class specifier `static` defines an incomplete type. If a translation unit or block contains one or more such declarations for an identifier, it must contain exactly one definition of the identifier (but not in an enclosed block, which would define a new type known only within that block).

□ Incomplete type declarations allow compact mutually-recursive types.

```
type t1;                      /* Incomplete type declaration. */
type t2 = struct { t1* p; /* ... */ };
type t1 = struct { t2* p; /* ... */ };
```

Without them, mutual recursion could be handled by declaring mutually recursive structures, then initializing the types to those structures.

```
struct s1;
type t2 = struct s2 { struct s1* p; /* ... */ };
type t1 = struct s1 { struct s2* p; /* ... */ };
```

This introduces extra names, and may force the programmer to cast between the types and their implementations. □

A type declaration without an initializer and with storage-class specifier `extern` is an *opaque type declaration*. Opaque types are object types. An opaque type is not a *constant-expression*; neither is a structure or union that has a member whose type is not a *constant-expression*. Every other object type is a *constant-expression*. Objects with static storage duration shall be declared with a type that is a *constant-expression*.

□ Type declarations can declare identifiers with external linkage, whereas typedef declarations declare identifiers that only exist within a translation unit. These opaque types can be used in declarations, but the implementation of the type is not visible.

Static objects can not have opaque types because space for them would have to be allocated at program start-up. This is a deficiency, but I don't want to deal with "module initialization" code just now. □

An incomplete type which is not a qualified version of a type is a value of type-class `dtype`. An object type which is not a qualified version of a type is a value of type-classes `type` and `dtype`. A function type is a value of type-class `ftype`.

□ Syntactically, a type value is a *type-name*, which is a declaration for an object which omits the identifier being declared.

Object types are precisely the types that can be instantiated. Type qualifiers are not included in type values because the compiler needs the information they provide at compile time to detect illegal statements or to produce efficient machine instructions. For instance, the code that a compiler must generate to manipulate an object that has volatile-qualified type may be different from the code to manipulate an ordinary object.

Type qualifiers are a weak point of C's type system. Consider the standard library function `strchr()` which, given a string and a character, returns a pointer to the first occurrence of the character in the string.

```
char *strchr(const char *s, int c) {
    char real_c = c;  /* done because c was declared as int. */
    for (; *s != real_c; s++)
        if (*s == '\0') return NULL;
    return (char*)s;
}
```

The parameter `s` must be `const char*`, because `strchr()` might be used to search a constant string, but the return type must be `char*`, because the result might be used to modify a non-constant string. Hence the body must perform a cast, and (even worse) `strchr()` provides a type-safe way to attempt to modify constant strings. What is needed is some way to say that `s`'s type might contain qualifiers, and the result type has exactly the same qualifiers. Polymorphic functions do not provide a fix for this deficiency, because type qualifiers are not part of type values. Instead, overloading can be used to define `strchr()` for each combination of qualifiers. □

□ Since incomplete types are not type values, they can not be used as the initializer in a type declaration, or as the type of a structure or union member. This prevents the declaration of types that contain each other.

```
type t1;
type t2 = t1;                    /* illegal: incomplete type 't1'. */
type t1 = t2;
```

The initializer in a file-scope declaration must be a constant expression. This means type declarations can not build on opaque types, which is a deficiency.

```
extern type Huge;  /* extended-precision integer type. */
type Rational = struct {
    Huge numerator, denominator;   /* illegal */
    };
struct Pair {
    Huge first, second;            /* legal */
    };
```

Without this restriction, Cforall might require "module initialization" code (since `Rational` has external linkage, it must be created before any other translation unit instantiates it), and would force an ordering on the initialization of the translation unit that defines `Huge` and the translation that declares `Rational`.

A benefit of the restriction is that it prevents the declaration in separate translation units of types that contain each other, which would be hard to prevent otherwise.

File a.c:

```
extern type t1;
type t2 = struct { t1 f1; /* ... */ } /* illegal */
```

File b.c:

```
extern type t2;
type t1 = struct { t2 f2; /* ... */ } /* illegal */
```

□

□ Since a *type-declaration* is a *declaration* and not a *struct-declaration*, type declarations can not be structure members. The form of *type-declaration* forbids arrays of, pointers to, and functions returning `type`. Hence the syntax of *type-specifier* does not have to be extended to allow type-valued expressions. It also side-steps the problem of type-valued expressions producing different values in different declarations.

Since a type declaration is not a *parameter-declaration*, functions can not have explicit type parameters. This may be too restrictive, but it attempts to make compilation simpler. Recall that when traditional C scanners read in an identifier, they look it up in the symbol table to determine whether or not it is a typedef name, and return a "type" or "identifier" token depending on what they find. A type parameter would add a type name to the current scope. The scope manipulations involved in parsing the declaration of a function that takes function pointer parameters and returns a function pointer may just be too complicated.

Explicit type parameters don't seem to be very useful, anyway, because their scope would not include the return type of the function. Consider the following attempt to define a type-safe memory allocation function.

```
#include <stdlib.h>
T* new(type T) { return (T*) malloc(sizeof(T)); };
...
int* ip = new(int);
```

This looks sensible, but Cforall's declaration-before-use rules mean that "`T`" in the function body refers to the parameter, but the "`T`" in the return type refers to the meaning of `T` in the scope that contains `new`; it could be undefined, or a type name, or a function or variable name. Nothing good can result from such a situation. □

Examples

Since type declarations create new types, instances of types are always passed by value.

```
type A1 = int[2];
void f1(A1 a) { a[0] = 0; };
typedef int A2[2];
void f2(A2 a) { a[0] = 0; };
A1 v1;
A2 v2;
f1(v1);
f2(v2);
```

`V1` is passed by value, so `f1()`'s assignment to `a[0]` does not modify v1. `V2` is converted to a pointer, so `f2()` modifies `v2[0]`.

A translation unit containing the declarations

```
extern type Complex;  /* opaque type declaration. */
extern float abs(Complex);
```

can contain declarations of complex numbers, which can be passed to `abs`. Some other translation unit must implement `Complex` and `abs`. That unit might contain the declarations

```
type Complex = struct { float re, im; };
Complex cplx_i = {0.0, 1.0};
float abs(Complex c) {
```

```
        return sqrt(c.re*c.re + c.im*c.im);
    }
```

Note that `c` is implicitly converted to a `struct` so that its components can be retrieved.

```
    type Time_of_day = int;         /* seconds since midnight. */
    Time_of_day ?+?(Time_of_day t1, int seconds) {
        return ((int)t1 + seconds)%86400;
    }
```

`t1` must be cast to its implementation type to prevent infinite recursion.

□ Within the scope of a type definition, an instance of the type can be viewed as having that type or as having the implementation type. In the `Time_of_day` example, the difference is important. Different languages have treated the distinction between the abstraction and the implementation in different ways.

- Inside a Clu cluster [6], the declaration of an instance states which view applies. Two primitives called `up` and `down` can be used to convert between the views.
- The Simula class [7] is essentially a record type. Since the only operations on a record are member selection and assignment, which can not be overloaded, there is never any ambiguity as to whether the abstraction or the implementation view is being used. In C++ [5], operations on class instances include assignment and "`&`", which can be overloaded. A "scope resolution" operator can be used inside the class to specify whether the abstract or implementation version of the operation should be used.
- An Ada derived type definition [8] creates a new type from an old type, and also implicitly declares derived subprograms that correspond to the existing subprograms that use the old type as a parameter type or result type. The derived subprograms are clones of the existing subprograms with the old type replaced by the derived type. Literals and aggregates of the old type are also cloned. In other words, the abstract view provides exactly the same operations as the implementation view. This allows the abstract view to be used in all cases.

  The derived subprograms can be replaced by programmer-specified subprograms. This is an exception to the normal scope rules, which forbid duplicate definitions of a subprogram in a scope. In this case, explicit conversions between the derived type and the old type can be used.

Cforall's rules are like Clu's, except that implicit conversions and conversion costs allow it to do away with most uses of `up` and `down`. □

5.4.1 Default Functions and Objects

A declaration of a type identifier `T` with type-class `type` implicitly declares a *default assignment* function `T ?=?(T*, T)`, with the same scope and linkage as the identifier `T`.

□ Assignment is central to C's imperative programming style, and every existing C object type has assignment defined for it (except for array types, which are treated as pointer types for purposes of assignment). Without this rule, nearly every inferred type parameter would need an accompanying assignment assertion parameter. If a type parameter should not have an assignment operation, `dtype` should be used. If a type should not have assignment defined, the user can define an assignment function that causes a run-time error, or provide an external declaration but no definition and thus cause a link-time error. □

A definition of a type identifier `T` with implementation type `I` and type-class `type` implicitly defines a default assignment function. A definition of a type identifier `T` with implementation type `I` and an assertion list implicitly defines *default functions* and *default objects* as declared by the assertion declarations. The default objects and functions have the same scope and linkage as the identifier `T`. Their values are determined as follows:

- If at the definition of `T` there is visible a declaration of an object with the same name as the default object, and if the type of that object with all occurrence of `I` replaced by `T` is compatible with the type of the default object, then the default object is initialized with that object. Otherwise the scope of the declaration of `T` must contain a definition of the default object.

- If at the definition of `T` there is visible a declaration of a function with the same name as the default function, and if the type of that function with all occurrence of `I` replaced by `T` is compatible with the type of the default function, then the default function calls that function after converting its arguments and returns the converted result.

  Otherwise, if `I` contains exactly one anonymous member such that at the definition of `T` there is visible a declaration of a function with the same name as the default function, and the type of that function with all occurrences of the anonymous member's type in its parameter list replaced by `T` is compatible with the type of the default function, then the default function calls that function after converting its arguments and returns the result.

  Otherwise the scope of the declaration of `T` must contain a definition of the default function.

  □ Note that a pointer to a default function will not compare as equal to a pointer to the inherited function. □

A function or object with the same type and name as a default function or object that is declared within the scope of the definition of `T` replaces the default function or object.

Examples

```
spec s(type T) {
    T a, b;
}
struct impl { int left, right; } a = { 0, 0 };
type Pair | s(Pair) = struct impl;
Pair b = { 1, 1 };
```

The definition of `Pair` implicitly defines two objects `a` and `b`. `Pair a` inherits its value from the `struct impl a`. The definition of `Pair b` is compulsory because there is no `struct impl b` to construct a value from.

```
spec ss(type T) {
    T    clone(T);
    void munge(T*);
}
type Whatsit | ss(Whatsit);
type Doodad | ss(Doodad) = struct doodad {
    Whatsit;    /* anonymous member */
    int extra;
};
Doodad clone(Doodad) { /* ... */ }
```

The definition of `Doodad` implicitly defines three functions:

```
Doodad ?=?(Doodad*, Doodad);
Doodad clone(Doodad);
void   munge(Doodad*);
```

The assignment function inherits `struct doodad`'s assignment function because the types match when `struct doodad` is replaced by `Doodad` throughout. `munge()` inherits `Whatsit`'s `munge()` because the types match when `Whatsit` is replaced by `Doodad` in the parameter list. `clone()` does *not* inherit `Whatsit`'s `clone()`: replacement in the parameter list yields "`Whatsit clone(Doodad)`", which is not compatible with `Doodad`'s `clone()`'s type. Hence the definition of "`Doodad clone(Doodad)`" is necessary.

Default functions and objects are subject to the normal scope rules.

```
type T = ...;
T a_T = ...;        /* Default assignment used. */
T ?=?(T*, T);
T a_T = ...;        /* Programmer-defined assignment called. */
```

□ A compiler warning would be helpful in this situation. □

□ The *class* construct of object-oriented programming languages performs three independent functions. It *encapsulates* a data structure; it defines a *subtype* relationship, whereby instances of one class may be used in contexts that require instances of another; and it allows one class to *inherit* the implementation of another.

In Cforall, encapsulation is provided by opaque types and the scope rules, and subtyping is provided by specifications and assertions. Inheritance is provided by default functions and objects. □

5.5 Initialization

An expression that is used as an *initializer* is treated as being cast to the type of the object being initialized. An expression used in an *initializer-list* is treated as being cast to the type of the aggregate member that it initializes. In either case the cast must have a single unambiguous interpretation.

Chapter 6

Statements

Many statements contain expressions, which may have more than one interpretation. The following sections describe how the Cforall translator selects an interpretation. In all cases the result of the selection shall be a single unambiguous interpretation.

6.1 Expression and Null Statements

The expression in an expression statement is treated as being cast to `void`.

6.2 Jump Statements

An expression in a `return` statement is treated as being cast to the result type of the function.

6.3 Selection Statements

The controlling expression *e* in the switch statement

```
switch ( e ) ...
```

may have more than one interpretation, but it shall have only one interpretation with an integral type. An integral promotion is performed on the expression if necessary. The constant expressions in `case` statements with the switch are converted to the promoted type.

6.4 Iteration Statements

The controlling expression *e* in the loops

```
if ( e ) ...
while ( e ) ...
do ... while ( e );
```

is treated as "`(int)((e)!=0)`".

The statement

```
for (a; b; c) ...
```

is treated as

```
for ((void)(a); (int)((b)!=0); (void)(c)) ...
```

Chapter 7

Preprocessing Directives

7.1 Predefined Macro Names

The implementation shall define the macro names `__LINE__`, `__FILE__`, `__DATE__`, and `__TIME__`, as in the ANSI C standard. It shall not define the macro name `__STDC__`.

In addition, the implementation shall define the macro name `__CFORALL__` to be the decimal constant 1.

Appendix A

Examples

A.1 C Types

This section gives example specifications for some groups of types that are important in the C language, in terms of the predefined operations that can be applied to those types.

A.1.1 Scalar, Arithmetic, and Integral Types

The pointer, integral, and floating-point types are all *scalar types*. All of these types can be logically negated and compared. The assertion "`scalar(Complex)`" should be read as "type `Complex` is scalar".

```
spec scalar(type T) {
    int !?(T);
    int ?<?(T, T), ?<=?(T, T), ?==?(T, T), ?>=?(T, T), ?>?(T, T), ?!=?(T, T);
};
```

The integral and floating-point types are *arithmetic types*, which support the basic arithmetic operators. The use of an assertion in the *spec-parameter-list* declares that, in order to be arithmetic, a type must also be scalar (and hence that scalar operations are available). This is equivalent to inheritance of specifications.

```
spec arithmetic(type T | scalar(T) ) {
    T +?(T),     -?(T);
    T ?*?(T, T), ?/?(T, T),  ?+?(T, T),  ?-?(T, T);
};
```

The various flavors of `char` and `int` and the enumerated types make up the *integral types*.

```
spec integral(type T | arithmetic(T) ) {
    T ~?(T);
    T ?&?(T, T),  ?|?(T, T), ?^?(T, T);
    T ?%?(T, T);
    T ?<<?(T, T), ?>>?(T, T);
};
```

A.1.2 Modifiable Types

The only operation that can be applied to all modifiable lvalues is simple assignment.

```
spec m_lvalue(type T) {
    T ?=?(T*, T);
};
```

Modifiable scalar lvalues are scalars and are modifiable lvalues, and assertions in the *spec-parameter-list* reflect those relationships. This is equivalent to multiple inheritance of specifications. Scalars can also be incremented and decremented.

```
spec m_l_scalar(type T | scalar(T) | m_lvalue(T) ) {
    T ?++(T*), ?--(T*);
    T ++?(T*), --?(T*);
};
```

Modifiable arithmetic lvalues are both modifiable scalar lvalues and arithmetic. Note that this results in the "inheritance" of `scalar` along both paths.

```
spec m_l_arithmetic(type T | m_l_scalar(T) | arithmetic(T) ) {
    T ?/=?(T*, T), ?*=?(T*, T);
    T ?+=?(T*, T), ?-=?(T*, T);
};

spec m_l_integral(type T | m_l_arithmetic(T) | integral(T) ) {
    T ?&=?(T*, T), ?|=?(T*, T),  ?^=?(T*, T);
    T ?%=?(T*, T), ?<<=?(T*, T), ?>>=?(T*, T);
};
```

A.1.3 Pointer and Array Types

Array types can barely be said to exist in ANSI C, since in most cases an array name is treated as a constant pointer to the first element of the array, and the subscript expression "`a[i]`" is equivalent to the dereferencing expression "`(*(a+(i)))`". Technically, pointer arithmetic and pointer comparisons other than "`==`" and "`!=`" are only defined for pointers to array elements, but the type system does not enforce those restrictions. Consequently, there is no need for a separate "array type" specification.

Pointer types are scalar types. Like other scalar types, they have "`+`" and "`-`" operators, but the types do not match the types of the operations in `arithmetic`, so these operators cannot be consolidated in `scalar`.

```
spec pointer(type P | scalar(P) ) {
    P         ?+?(P, long int), ?+?(long int, P), ?-?(P, long int);
    ptrdiff_t ?-?(P, P);
};

spec m_l_pointer(type P | pointer(P) | m_l_scalar(P) ) {
    P     ?+=?(P*, long int), ?-=?(P*, long int);
    P     ?=?(P*, void*);
    void* ?=?(void**, P);
};
```

Specifications that define the dereference operator (or subscript operator) require two parameters, one for the pointer type and one for the pointed-at (or element) type. Different specifications are needed for each set of type qualifiers, because qualifiers are not included in types. The assertion "`|ptr_to(Safe_pointer, int)`" should be read as "`Safe_pointer` acts like a pointer to `int`".

```
spec ptr_to(type P | pointer(P), type T) {
    lvalue T *?(P);  lvalue T ?[?](P, long int);
};

spec ptr_to_const(type P | pointer(P), type T) {
    const lvalue T *?(P);  const lvalue T ?[?](P, long int);
};

spec ptr_to_volatile(type P | pointer(P), type T) }
    volatile lvalue T *?(P);  volatile lvalue T ?[?](P, long int);
};

spec ptr_to_const_volatile(type P | pointer(P), type T) }
    const volatile lvalue T *?(P);
    const volatile lvalue T ?[?](P, long int);
};
```

Assignment to pointers is more complicated than is the case with other types, because the target's type can have extra type qualifiers in the pointed-at type: a "`T*`" can be assigned to a "`const T*`", a "`volatile T*`", and a "`const volatile T*`". Again, the pointed-at type is passed in, so that assertions can connect these specifications to the "`ptr_to`" specifications.

```
spec m_l_ptr_to(type P | m_l_pointer(P),
                type T | ptr_to(P,T) {
    P  ?=?(P*, T*);
    T* ?=?(T**, P);
};

spec m_l_ptr_to_const(type P | m_l_pointer(P),
                      type T | ptr_to_const(P,T)) {
    P        ?=?(P*, const T*);
    const T* ?=?(const T**, P);
};

spec m_l_ptr_to_volatile(type P | m_l_pointer(P),
                         type T | ptr_to_volatile(P,T)) {
    P           ?=?(P*, volatile T*);
    volatile T* ?=?(volatile T**, P);
};

spec m_l_ptr_to_const_volatile(
       type P | ptr_to_const_volatile(P),
       type T | m_l_ptr_to_volatile(P,T) | m_l_ptr_to_const(P)) {
    P                 ?=?(P*, const volatile T*);
    const volatile T* ?=?(const volatile T**, P);
};
```

Note the regular manner in which type qualifiers appear in those specifications. An alternative specification can make use of the fact that qualification of the pointed-at type is part of a pointer type to capture that regularity.

```
spec m_l_ptr_like(type MyP | m_l_pointer(MyP),
                  type CP | m_l_pointer(CP) ) {
    MyP ?=?(MyP*, CP);
    CP  ?=?(CP*, MyP);
};
```

The assertion "`| m_l_ptr_like(Safe_ptr, const int*)`" should be read as "`Safe_ptr` is a pointer type like `const int*`". This specification has two defects, compared to the original four: there is no automatic assertion that dereferencing a `MyP` produces an lvalue of the type that `CP` points at, and the "`|m_l_pointer(CP)`" assertion provides only a weak assurance that the argument passed to `CP` really is a pointer type.

A.2 Relationships Between Operations

Different operators often have related meanings; for instance, in C, "+", "+=", and the two versions of "++" perform variations of addition. Languages like C++ and Ada allow programmers to define operators for new types, but do not require that these relationships be preserved, or even that all of the operators be implemented. Completeness and consistency is left to the good taste and discretion of the programmer. It is possible to encourage these attributes by providing generic operator functions, or member functions of abstract classes, that are defined in terms of other, related operators.

In Cforall, polymorphic functions provide the equivalent of these generic operators, and specifications explicitly define the minimal implementation that a programmer should provide. This section shows a few examples.

A.2.1 Relational and Equality Operators

The different comparison operators have obvious relationships, but there is no obvious subset of the operations to use in the implementation of the others. However, it is usually convenient to implement a single comparison function that returns a negative integer, 0, or a positive integer if its first argument is respectively less than, equal to, or greater than its second argument; the library function `strcmp` is an example.

C and Cforall have an extra, non-obvious comparison operator: "`!`", logical negation, returns 1 if its operand compares equal to 0, and 0 otherwise.

```
spec comparable(type T) {
    const T 0;
    int     compare(T, T);
}

forall(type T | comparable(T)) int ?<?(T l, T r) {
    return compare(l,r) < 0;
}
/* ... similarly for <=, ==, >=, >,
   and !=. */

forall(type T | comparable(T)) int !?(T operand) {
    return !compare(operand, 0);
}
```

A.2.2 Arithmetic and Integer Operations

A complete arithmetic type would provide the arithmetic operators and the corresponding assignment operators. Of these, the assignment operators are more likely to be implemented directly, because it is usually more efficient to alter the contents of an existing object than to create and return a new one. Similarly, a complete integral type would provide integral operations based on integral assignment operations.

```
spec arith_base(type T) {
    const T 1;
    T       ?+=?(T*,T), ?-=?(T*,T), ?*=?(T*,T), ?/=?(T*,T);
}

forall(type T | arith_base(T)) T ?+?(T l, T r) {
    return l += r;
}

forall(type T | arith_base(T)) T ?++(T* operand) {
    T temporary = *operand;
    *operand += 1;
    return temporary;
}

forall(type T | arith_base(T)) T ++?(T* operand) {
    return *operand += 1;
}
/* ... similarly for -, --, *,and /. */

spec int_base(type T) {
    T ?&=?(T*, T), ?|=?(T*, T),  ?^=?(T*, T);
    T ?%=?(T*, T), ?<<=?(T*, T), ?>>=?(T*, T);
}

forall(type T | int_base(T)) T ?&?(T l, T r) {
    return l &= r;
}
/* ... similarly for |, ^, %, <<,and >>. */
```

Note that, although an arithmetic type would certainly provide comparison functions, and an integral type would provide arithmetic operations, there does not have to be any relationship among `int_base`, `arith_base` and `comparable`. Note also that these declarations provide guidance and assistance, but they do not define an absolutely minimal set of requirements. A truly minimal implementation of an arithmetic type might only provide `0`, `1`, and `?-=?`, which would be used by polymorphic `?+=?`, `?*=?`, and `?/=?` functions.

Bibliography

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

[3] G. V. Cormack and A. K. Wright. Type-dependent parameter inference. *SIGPLAN Notices*, 25(6):127–136, June 1990. Proceedings of the ACM Sigplan'90 Conference on Programming Language Design and Implementation June 20-22, 1990, White Plains, New York, U.S.A.

[4] Glen Ditchfield. An overview of cforall. in preparation, 1996.

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

[6] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[7] Standardiseringskommissionen i Sverige. *Databehandling – Programspråk – SIMULA*, 1987. Svensk Standard SS 63 61 14.

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

Index

Italic page numbers give the location of the main entry for the referenced term. Plain page numbers denote uses of the indexed term. Entries for grammar non-terminals are italicized. A typewriter font is used for grammar terminals and program identifiers.

`!?`, *7*, 20
`*?`, *7*, 20
`++?`, *7*, 20
`+?`, *7*, 20
`--?`, *7*, 20
`-?`, *7*, 20
`?!=?`, *7*, 26
`?%=?`, *7*, 31
`?%?`, *7*, 23
`?&=?`, *7*, 31
`?&?`, *7*, 28
`?()`, *7*, 15
`?*=?`, *7*, 31
`?*?`, *7*, 23, 41
`?++`, *7*, 15
`?+=?`, *7*, 31, 41
`?+?`, *7*, 24, *46*
`?--`, *7*, 15
`?-=?`, *7*, 31
`?-?`, *7*, 24
`?/=?`, *7*, 31
`?/?`, *7*, 23
`?:`, 30
`?<<=?`, *7*, 31
`?<<?`, *7*, 25
`?<=?`, *7*, 26
`?<?`, *7*, 26
`?==?`, *7*, 26
`?=?`, *7*, 31, 46
`?>=?`, *7*, 26
`?>>=?`, *7*, 31
`?>>?`, *7*, 25
`?>?`, *7*, 26
`?[?]`, *7*, 15
`?^=?`, *7*, 31
`?^?`, *7*, 28
`?|=?`, *7*, 31
`?|?`, *7*, 29
`~?`, *7*, 20

`0`, *6*, 13–15, 19, 29, 41
`1`, *6*, 14

`abs`, 45
`abs(Complex)`, *45*
additive-expression, 24, *24*, 25
`alloc`, *38*, 39
ambiguous interpretation, 13, *13*, 17, 19, 21, 22, 30, 34
AND-expression, 28, *28*
anonymous member, 10, 12, *36*, 47
argument-expression-list, 15, *15*
`arithmetic`, 52, *52*, 53
arithmetic types, *52*
assertion, *41*
assertion, 41, *41*
assertion parameters, 11, 17, 37, *41*
assertion-list, 41, *41*, 42
assignment-expression, 15, 30, *30*, 34
assignment-operator, 30, *30*
automatic storage duration, *4*

best valid interpretations, *13*, 15, 17
block, 3, 4, 43
block scope, *3*, 37, 43

cast expression, 8, 22, 39
cast-expression, 20, 22, *22*, 23
`color_point`, *36*
comma expression, 8
compatible type, *1*, 8, 9, 11, 17, 37
`Complex`, 21, 29, 45, *45*
composite type, *2*, 17, 41
conditional-expression, 30, *30*
`const`, *4*, 39

const-qualified, 3, *3*, 40
constant, 14
constant identifiers, 6
constant-expression, 6, 43
conversion cost, *12*, 13, 23
`cplx_i`, *45*

declaration, *2*
declaration, *35*, 37, 45
declaration scope, *37*
declaration-specifiers, 36
declarator, 40
declarator-list, 40, *40*
default argument promotions, 2, *2*, 16
default assignment, *46*
default functions, *47*
default objects, *47*
deficiencies
  generalizability, 17
  nesting opaque types, 44
  pointers to bit-fields, 19
  pointers to qualified types, 19, 44
  static opaque objects, 43
  union assignment, 34
  void* conversion, 12
definition, *2*
definition scope, 4
derived declarator types, *2*
direct safe conversion, *12*
`Doodad`, 47
`dtype`, 43

`eat_double`, 22, 23
equality-expression, 26, *26*, 28
exclusive-OR-expression, 28, *28*
explicit conversions, *10*
expression, 14, 15, 30, 34, *34*
`extern`, *4*, 43
external linkage, 9

file scope, *3*, 14
forall specifier, *35*
forall-specifier, 35, 36, *36*, 37
`ftype`, 43
function designator, *16*
function prototype, *3*
function prototype scope, *3*
function scope, *3*
function types, *4*, 41, 43

generalizability, *17*

identifier, *6*, 14, 15, 40–42
identifiers
  for constants, 6
  for operators, 7
implementation type, *43*, 47
implicit arguments, *16*
implicit conversions, 10, *10*, 43
inclusive-OR-expression, 28, *28*, 29
incomplete types, 3, *4*, 40, 41, 43, 44
inferred parameter, 11, 21, *37*
initializer, 43
initializer, 48
initializer-list, 48
`integral`, *52*, 53
integral promotions, 2, *2*, 11, 33
integral types, *52*
internal linkage, 9, 14
interpretations, *13*, 21, 48, 49

keyword, *6*

less polymorphic, 11, 13, 19, *37*
linkage, 8, 46, 47
`list_of`, *42*
`literal`, *36*
logical-AND-expression, 29, *29*, 30
logical-OR-expression, 30, *30*
lvalue, *3*, 10, 39
`lvalue`, *39*
lvalue-qualified, *3*

`m_l_arithmetic`, 53, *53*
`m_l_integral`, *53*
`m_l_pointer`, *53*, 54, 55
`m_l_ptr_like`, *55*
`m_l_ptr_to`, *54*
`m_l_ptr_to_const`, 54, *54*
`m_l_ptr_to_const_volatile`, *54*
`m_l_ptr_to_volatile`, 54, *54*
`m_l_scalar`, 53, *53*
`m_lvalue`, 53, *53*
`min_max`, *19*
modifiable lvalue, *3*, 39, 52
monomorphic function, 16, *37*
`move_by`, *10*
multiplicative-expression, 23, *23*, 24

name spaces, *3*, 8

no linkage, 9, 35, 37, 43
null pointer, 13, 19, 27

object, *3*
object types, 3, *4*, 41, 43
opaque type declaration, *43*
operator identifiers, 7, 14
overloading, 6, 8

parameter-declaration, 45
parameter-type-list, 37
`point`, *36*
`pointer`, *53*, 54
polymorphic function, 16, *37*
postfix-expression, 15, *15*, 16, 20
primary-expression, *14*, 15
`ptr_to`, 54, *54*
`ptr_to_const`, 54, *54*
`ptr_to_const_volatile`, 54, *54*
`ptr_to_volatile`, 54, *54*
`ptrdiff_t`, 16

qualified type, 1, 3, 12, 43
qualified versions, 2, *3*

relational-expression, 25, *25*, 26
rewrite rules, *14*

safe conversions, *11*, 12, 17
`scalar`, 52, *52*, 53
scalar types, *52*
scopes, *3*, 8, 46, 47
sequence points, *4*
shift-expression, 25, *25*
side effects, *4*
`sizeof`, 21
spec-declaration, 40, *40*, 41
spec-declaration-list, 40, *40*, 41, 43
spec-definition, 35, *40*
spec-parameter-list, 41, 52, 53
specialization, 14, 19
specification, 37, *41*
specification definition, *41*
specifier-qualifier-list, 35, 40
`square`, *41*
`static`, *4*, 43
static storage duration, *4*
storage duration, *4*
storage-class specifiers, *4*, 43
storage-class-specifier, 42

`strchr`, *44*
string-literal, 14
struct-declaration, *35*, 36, 45
struct-declarator-list, 36
`sum_list`, 42
`summable`, *41*

`Time_of_day`, *46*
translation unit, *4*, 43
type, *4*
`type`, 8, 17, 41, 43
type declaration, 12, 46
type definition, *43*, 47
type names, 43
type qualifiers, 3, *4*, 53
type specifier, *4*
type-class, 17, 43
type-class, 42, *42*
type-declaration, 21, 35, *42*, 45
type-declarator, 42, *42*, 43
type-declarator-list, 42, *42*
type-name, 20, 22, 41–43
type-name-list, 41, *41*
type-parameter, 21, 41, 42, *42*, 43
type-parameter-list, 36, 37, 40, 42, *42*
type-qualifier, *39*
type-specifier, *35*, 45
`typedef`, 8

unary-expression, 20, *20*, 22, 30
unary-operator, 20, *20*
unsafe conversions, 12, *12*, 13, 17
usual arithmetic conversions, *11*

valid interpretations, *13*, 14, 16
visible, 3, *5*, 8, 14
`volatile`, *4*, 39
volatile-qualified, *3*

`Whatsit`, 47