Lvalues and References

C defines the notion of a *lvalue*, essentially an addressable object, as well as a number of type *qualifiers*, `const`, `volatile`, and `restrict`. As these type qualifiers are generally only meaningful to the type system as applied to lvalues, the two concepts are closely related. A const lvalue cannot be modified, the compiler cannot assume that a volatile lvalue will not be concurrently modified by some other part of the system, and a restrict lvalue must have pointer type, and the compiler may assume that no other pointer in scope aliases that pointer (this is solely a performance optimization, and may be ignored by implementers). *Lvalue-to-rvalue conversion*, which takes an lvalue of type `T` and converts it to an expression result of type `T` (commonly called an *rvalue* of type `T`) also strips all the qualifiers from the lvalue, as an expression result is a value, not an addressable object that can have properties like immutability. Though lvalue-to-rvalue conversion strips the qualifiers from lvalues, derived rvalue types such as pointer types may include qualifiers; `const int *` is a distinct type from `int *`, though the latter is safely convertable to the former. In general, any number of qualifiers can be safely added to the pointed-to-type of a pointer type, e.g. `int *` converts safely to `const int *` and `volatile int *`, both of which convert safely to `const volatile int *`.

Since lvalues are precicely "addressable objects", in C, only lvalues can be used as the operand of the `&` address-of operator. Similarly, only modifiable lvalues may be used as the assigned-to operand of the mutating operators: assignment, compound assignment (e.g. `+=`), and increment and decrement; roughly speaking, lvalues without the `const` qualifier are modifiable, but lvalues of incomplete types, array types, and struct or union types with const members are also not modifiable. Lvalues are produced by the following expressions: object identifiers (function identifiers are not considered to be lvalues), the result of the `*` dereference operator applied to an object pointer, the result of a member expression `s.f` if the left argument `s` is an lvalue (note that the preceding two rules imply that the result of indirect member expressions `s->f` are always lvalues, by desugaring to `(*s).f`), and the result of the indexing operator `a[i]` (similarly by its desugaring to `*((a)+(i))`). Somewhat less obviously, parenthesized lvalue expressions, string literals, and compound literals (e.g. `(struct foo){ 'x', 3.14, 42 }`) are also lvalues.

All of the conversions described above are defined in standard C, but Cforall requires further features from its type system. In particular, to allow overloading of the `*?` and `?[?]` dereferencing and indexing operators, Cforall requires a way to declare that the functions defining these operators return lvalues, and since C functions never return lvalues and for syntactic reasons we wish to distinguish functions which return lvalues from functions which return pointers, this is of necessity an extension to standard C. In the current design, an `lvalue` qualifier can be added to function return types (and only to function return types), the effect of which is to return a pointer which is implicitly dereferenced by the caller. C++ includes the more general concept of *references*, which are typically implemented as implicitly dereferenced pointers as well. Another use case which C++ references support is providing a way to pass function parameters by reference (rather than by value) with a natural syntax; Cforall in its current state has no such mechanism. As an example, consider the following (currently typical) copy-constructor signature and call:

```
void ?{}(T *lhs, T rhs);

T x;
T y = { x };
```

Note that the right-hand argument is passed by value, and would in fact be copied twice in the course of the constructor call `T y = { x };` (once into the parameter by C's standard `memcpy` semantics, once again in the body of the copy constructor, though it is possible that return value optimization will elide the `memcpy`-style copy). However, to pass by reference using the existing pointer syntax, the example above would look like this:

```
void ?{}(T *lhs, const T *rhs);

T x;
T y = { &x };
```

This example is not even as bad as it could be; assuming pass-by-reference is the desired semantics for the `?+?` operator, that implies the following design today:

```
T ?+?(const T *lhs, const T *rhs);

T a, b;
T c = &a + &b,
```

In addition to `&a + &b` being unsightly and confusing syntax to add `a` and `b`, it also introduces a possible ambiguity with pointer arithmetic on `T*` which can only be resolved by return-type inference.

Pass-by-reference and marking functions as returning lvalues instead of the usual rvalues are actually closely related concepts, as obtaining a reference to pass depends on the referenced object being addressable, i.e. an lvalue, and lvalue return types are effectively return-by-reference. Cforall should also unify the concepts, with a parameterized type for "reference to `T`", which I will write `T&`.

Firstly, assignment to a function parameter as part of a function call and local variable initialization have almost identical semantics, so should be treated similarly for the reference type too; this implies we should be able to declare local variables of reference type, as in the following:

```
int x = 42;
int& r = x; // r is now an alias for x
```

Unlike in C++, we would like to have the capability to re-bind references after initialization, as this allows the attractive syntax of references to support some further useful code patterns, such as first initializing a reference after its declaration. Constant references to `T` (`T& const`) should not be re-bindable.

One option for re-binding references is to use a dedicated operator, as in the code example below:

```
int i = 42, j = 7;
int& r = i;  // bind r to i
r = j;       // set i (== r) to 7
r := j;      // rebind r to j using the new := rebind operator
i = 42;      // reset i (!= r) to 42
assert( r == 7 );
```

Another option for reference rebind is to modify the semantics of the `&` address-of operator. In standard C, the address-of operator never returns an lvalue, but for an object of type `T`, returns a `T*`. If the address-of operator returned an lvalue for references, this would allow reference rebinding using the usual pointer assignment syntax; that is, if address-of a `T&` returned a `T*&` then the following works:

```
int i = 42; j = 7;
int& r = i;  // bind r to i
r = j;       // set i (== r) to 7
&r = &j;     // rebind r to j using the newly mutable "address-of reference"
i = 42;      // reset i (!= r) to 42
assert( r == 7 );
```

This change (making addresses of references mutable) allows use of existing operators defined over pointers, as well as elegant handling of nested references-to-references.

The semantics and restrictions of `T&` are effectively the semantics of an lvalue of type `T`, and by this analogy there should be a safe, qualifier dropping conversion from `const volatile restrict T&` (and every other qualifier combination on the `T` in `T&`) to `T`. With this conversion, the resolver may type most expressions that C would call "lvalue of type `T`" as `T&`. There's also an obvious argument that lvalues of a (possibly-qualified) type `T` should be convertable to references of type `T`, where `T` is also so-qualified (e.g. lvalue `int` to `int&`, lvalue `const char` to `const char&`). By similar arguments to pointer types, qualifiers should be addable to the referred-to type of a reference (e.g. `int&` to `const int&`). As a note, since pointer arithmetic is explictly not defined on `T&`, `restrict T&` should be allowable and would have alias-analysis rules that are actually comprehensible to mere mortals.

Using pass-by-reference semantics for function calls should not put syntactic constraints on how the function is called; particularly, temporary values should be able to be passed by reference. The mechanism for this pass-by-reference would be to store the value of the temporary expression into a new unnamed temporary, and pass the reference of that temporary to the function. As an example, the following code should all compile and run:

```
void f(int& x) { printf("%d\n", x++); }

int i = 7, j = 11;
const int answer = 42;
```

```
f(i);      // (1)
f(42);     // (2)
f(i + j);  // (3)
f(answer); // (4)
```

The semantics of (1) are just like C++'s, "7" is printed, and `i` has the value 8 afterward. For (2), "42" is printed, and the increment of the unnamed temporary to 43 is not visible to the caller; (3) behaves similarly, printing "19", but not changing `i` or `j`. (4) is a bit of an interesting case; we want to be able to support named constants like `answer` that can be used anywhere the constant expression they're replacing (like `42`) could go; in this sense, (4) and (2) should have the same semantics. However, we don't want the mutation to the `x` parameter to be visible in `answer` afterward, because `answer` is a constant, and thus shouldn't change. The solution to this is to allow chaining of the two lvalue conversions; `answer` has the type `const int&`, which can be converted to `int` by the lvalue-to-rvalue conversion (which drops the qualifiers), then up to `int&` by the temporary-producing rvalue-to-lvalue conversion. Thus, an unnamed temporary is inserted, initialized to `answer` (i.e. 42), mutated by `f`, then discarded; "42" is printed, just as in case (2), and `answer` still equals 42 after the call, because it was the temporary that was mutated, not `answer`. It may be somewhat surprising to C++ programmers that `f(i)` mutates `i` while `f(answer)` does not mutate `answer` (though `f(answer)` would be illegal in C++, leading to the dreaded "const hell"), but the behaviour of this rule can be determined by examining local scope with the simple rule "non-`const` references to `const` variables produce temporaries", which aligns with programmer intuition that `const` variables cannot be mutated.

To bikeshed syntax for `T&`, there are three basic options: language keywords (`lvalue T` is already in Cforall), compiler-supported "special" generic types (e.g. `ref(T)`), or sigils (`T&` is familiar to C++ programmers). Keyword or generic based approaches run the risk of name conflicts with existing code, while any sigil used would have to be carefully chosen to not create parsing conflicts.