Types for 0 and 1 literals

The literals `0` and `1` are treated specially by Cforall, due to their potential uses in operator overloading. Earlier versions of Cforall allowed `0` and `1` to be variable names, allowing multiple interpretations of them according to the existing variable overloading rules, with the following declarations in the prelude:

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

This did, however, create some backward-compatibility problems and potential performance issues, and works poorly for generic types. To start with, this (entirely legal C) code snippet doesn't compile in Cforall:

```
if ( 0 ) {}
```

It desugars to `if ( (int)(0 != 0) ) {}`, and since both `int` and `forall(dtype DT) DT*` have a != operator which returns `int` the resolver can not choose which `0` variable to take, because they're both exact matches.

The general `!=` computation may also be less efficient than a check for a zero value; take the following example of a rational type:

```
struct rational { int32_t num, int32_t den };
rational 0 = { 0, 1 };

int ?!=? (rational a, rational b) {
    return ((int64_t)a.num)*b.den != ((int64_t)b.num)*a.den;
}

int not_zero (rational a) { return a.num != 0; }
```

To check if two rationals are equal we need to do a pair of multiplications to normalize them (the casts in the example are to prevent overflow), but to check if a rational is non-zero we just need to check its numerator, a more efficient operation.

Finally, though polymorphic null-pointer variables can be meaningfully defined, most other polymorphic variables cannot be, which makes it difficult to make generic types "truthy" using the existing system:

```
forall(otype T) struct pair { T x; T y; };
forall(otype T | { T 0; }) pair(T) 0 = { 0, 0 };
```

Now, it seems natural enough to want to define the zero for this pair type as a pair of the zero values of its element type (if they're defined). The declaration of `pair(T) 0` above is actually illegal though, as there is no way to represent the zero values of an infinite number of types in the single memory location available for this polymorphic variable - the polymorphic null-pointer variables defined in the prelude are legal, but that is only because all pointers are the same size and the single zero value is a legal value of all pointer types simultaneously; null pointer is, however, somewhat unique in this respect.

The technical explanation for the problems with polymorphic zero is that `0` is really a rvalue, not a lvalue - an expression, not an object. Drawing from this, the solution we propose is to give `0` a new built-in type, `zero_t`, and similarly give `1` the new built-in type `one_t`. If the prelude defines `!=` over `zero_t` this solves the `if ( 0 )` problem, because now the unambiguous best interpretation of `0 != 0` is to read them both as `zero_t` (and say that this expression is false). Backwards compatibility with C can be served by defining conversions in the prelude from `zero_t` and `one_t` to `int` and the appropriate pointer types, as below:

```
// int 0;
forall(otype T | { void ?{safe}(T*, int); }) void ?{safe} (T*, zero_t);
forall(otype T | { void ?{unsafe}(T*, int); }) void ?{unsafe} (T*, zero_t);

// int 1;
forall(otype T | { void ?{safe}(T*, int); }) void ?{safe} (T*, one_t);
forall(otype T | { void ?{unsafe}(T*, int); }) void ?{unsafe} (T*, one_t);

// forall(dtype DT) const DT* 0;
forall(dtype DT) void ?{safe}(const DT**, zero_t);
// forall(ftype FT) FT* 0;
forall(ftype FT) void ?{safe}(FT**, zero_t);
```

Further, with this change, instead of making `0` and `1` overloadable variables, we can instead allow user-defined constructors (or, more flexibly, safe conversions) from `zero_t`, as below:

```
// rational 0 = { 0, 1 };
void ?{safe} (rational *this, zero_t) { this->num = 0; this->den = 1; }
```

Note that we don't need to name the `zero_t` parameter to this constructor, because its only possible value is a literal zero. This one line allows `0` to be used anywhere a `rational` is required, as well as enabling the same use of rationals in boolean contexts as above (by interpreting the `0` in the desguraring to be a rational by this conversion). Furthermore, while defining a conversion function from literal zero to `rational` makes rational a "truthy" type able to be used in a boolean context, we can optionally further optimize the truth decision on rationals as follows:

```
int ?!=? (rational a, zero_t) { return a.num != 0; }
```

This comparison function will be chosen in preference to the more general rational comparison

function for comparisons against literal zero (like in boolean contexts) because it doesn't require a conversion on the `0` argument. Functions of the form `int ?!=? (T, zero_t)` can acutally be used in general to make a type `T` truthy without making `0` a value which can convert to that type, a capability not available in the current design.

This design also solves the problem of polymorphic zero for generic types, as in the following example:

```
// ERROR: forall(otype T | { T 0; }) pair(T) 0 = { 0, 0 };
forall(otype T | { T 0; }) void ?{safe} (pair(T) *this, zero_t) {
    this->x = 0; this->y = 0;
}
```

The polymorphic variable declaration didn't work, but this constructor is perfectly legal and has the desired semantics.

We can assert that `T` can be used in a boolean context as follows:

```
`forall(otype T | { int ?!=?(T, zero_t); })`
```

Since the C standard (6.5.16.1.1) specifically states that pointers can be assigned into `_Bool` variables (and implies that other artithmetic types can be assigned into `_Bool` variables), it seems natural to say that assignment into a `_Bool` variable effectively constitutes a boolean context. To allow this interpretation, I propose including the following function (or its effective equivalent) in the prelude:

```
forall(otype T | { int ?!=?(T, zero_t); })
void ?{safe}( _Bool *this, T that ) { *this = that != 0; }
```

Note that this conversion is not transitive; that is, for `t` a variable of some "truthy" type `T`, `(_Bool)t;` would use this conversion (in the absence of a lower-cost one), `(int)t;` would not use this conversion (and in fact would not be legal in the absence of another valid way to convert a `T` to an `int`), but `(int)(_Bool)t;` could legally use this conversion.

Similarly giving literal `1` the special type `one_t` allows for more concise and consistent specification of the increment and decrement operators, using the following de-sugaring:

```
++i => i += 1
i++ => (tmp = i, i += 1, tmp)
--i => i -= 1
i-- => (tmp = i, i -= 1, tmp)
```

In the examples above, `tmp` is a fresh temporary with its type inferred from the return type of `i += 1`. Under this proposal, defining a conversion from `one_t` to `T` and a `lvalue T ?+=? (T*, T)`

provides both the pre- and post-increment operators for free in a consistent fashion (similarly for -= and the decrement operators). If a meaningful `1` cannot be defined for a type, both increment operators can still be defined with the signature `lvalue T ?+=? (T*, one_t)`. Similarly, if scalar addition can be performed on a type more efficiently than by repeated increment, `lvalue T ?+=? (T*, int)` will not only define the addition operator, it will simultaneously define consistent implementations of both increment operators (this can also be accomplished by defining a conversion from `int` to `T` and an addition operator `lvalue T ?+=?(T*, T)`).

To allow functions of the form `lvalue T ?+=? (T*, int)` to satisfy "has an increment operator" assertions of the form `lvalue T ?+=? (T*, one_t)`, we also define a non-transitive unsafe conversion from `_Bool` (allowable values `0` and `1`) to `one_t` (and `zero_t`) as follows:

```
void ?{unsafe} (one_t*, _Bool) {}
```

As a note, the desugaring of post-increment above is possibly even more efficient than that of C++ - in C++, the copy to the temporary may be hidden in a separately-compiled module where it can't be elided in cases where it is not used, whereas this approach for Cforall always gives the compiler the opportunity to optimize out the temporary when it is not needed. Furthermore, one could imagine a post-increment operator that returned some type `T2` that was implicitly convertable to `T` but less work than a full copy of `T` to create (this seems like an absurdly niche case) - since the type of `tmp` is inferred from the return type of `i += 1`, you could set up functions with the following signatures to enable an equivalent pattern in Cforall:

```
lvalue T2 ?+=? (T*, one_t);   // increment operator returns T2
void ?{} (T2*, T);            // initialize T2 from T for use in `tmp = i`
void ?{safe} (T*, T2);        // allow T2 to be used as a T when needed to
                              // preserve expected semantics of T x = y++;
```