Notes on Quantifiers

These are some notes on the basics of quantifiers and how to implement them. Sorry if it's is too rudimentary in places.

Forms of Quantifier Expressions

Quantifiers are boolean-valued expressions that evaluate a quantified sub-expression multiple times. The number of times the evaluation happens depends on the kind of quantification, which upcoming examples illustrate.

As in normal predicate logic, there are two forms of quantification -- *universal* and *existential*. Universal quantification is expressed with a `forall` expression, existential with `exists`.

The basic form of universal quantification is:

```
forall(x:t) predicate
```

This is read "for all values *x* of type *t*, *predicate* is true", where the variable *x* must appear one or more times in the *predicate*. For example,

```
forall (rec:PersonRecord) rec.name != nil
```

This expression says that the `name` field of all `PersonRecord`s is not nil, assuming an appropriate definition of `PersonRecord`, e.g.,

```
obj PersonRecord = name:Name and age:Age and ...
obj Name = string;
obj Age = integer;
```

There are two extended forms of universal quantification:

Extended Form	Reading	Equivalent To
`forall (x:t | p1) p2`	For all *x* of type *t*, such that *p1* is true, *p2* is true.	`forall (x:t) if p1 then p2`
`forall (x in l) p`	For all *x* in *l*, *p* is true.	`forall (x:basetype(l)) if x in l then p`

The "Equivalent To" column means that the extended forms are not any more powerful than the standard form, just more convenient in some cases. The list form is particularly convenient in postconditions of operations that return lists, e.g.,

```
op CreateNonEmptyList()->(l: integer*)
    post: forall (item in l) item != nil;
end;
```

This postcondition says that all of the items in the output list must be non-nil.

The value of a quantifier expression is true if *all* sub-expression evaluations are true. E.g.,

```
forall (item in l) item != nil
```

is true if and only if every item in the list is not nil. If one or more items is `nil`, then the `forall` is false. In this sense, universal quantification is the collected `and` of all sub-expression values.

Existential quantification is logically complementary to universal quantification. That is, existential quantification is the logical `or` of all sub-expression values. E.g., the following expression is true if at least one element of the list `l` is nil:

```
exists (item in l) item = nil
```

While it's not critical to formalize the relationship between universal and existential quantification, it's helpful to understand their similarity as operators. The relationship is based on the fact that `forall` is repeated `and`, and `exists` is repeated `or`. This leads to an extended form of DeMorgan's law

```
x and y <=> not (not x or not y)
x or y <=> not (not x and not y)
```

for quantifiers

```
forall (x:t) p  <=> not (exists (x:t) not p)
exists (x:t) p  <=> not (forall (x:t) not p)
```

Again, this isn't a big deal, but it helps illustrate that both quantifiers are doing the same thing. Namely, they're applying a boolean operator multiple times to get a single boolean result.

Quantifier Evaluation

One can think of quantifiers as a form of programming loop. The important difference between a quantifier and a loop is that the quantifier only produces a single boolean value. In contrast, a program loop typically does not produce a value itself. Rather, the loop executes a body of statements multiple times, with the statements producing value(s) stored in persistent variable(s).

Here's a side-by-side example that illustrates the difference between a `forall` expression and a `for` loop. The idea in both examples is to check that there's no 0 value in a list of integers. The quantifier version is

```
forall (i in l) i != 0
```

The loop version is

```
bool result = false;
for (i = 0; i < length(l); i++) {
    if (l[i] == 0) {
        result = false;
        break;
    }
}
```

In this example, the quantifier expression looks a lot simpler. But this is because the problem is to compute a boolean value that checks all of the elements of a list. If the problem involves some other computation than a boolean value, then a `forall` quantifier can't be used. E.g., there is no quantifier expression that can do the list-summing computation done by the following loop

```
int sum = 0;
for (i = 0; i < length(l); i++ {
    sum += l[i];
}
```

The recurring important point is that quantifiers evaluate to a single boolean value. A quantifier expression cannot be used to produce values other than boolean. (Aside -- things like integer summing are done in a functional language using recursion, but that's off-topic as far as quantifiers go.)

Implementing Quantifiers

Since quantifiers act like a form of loop, the interpreter implementation will naturally use a loop. The list form of quantifiers is the easy case. E.g.,

```
forall (x in l) expression
```

loops through each element of the list `l`, and evaluates `expression` in each iteration. Prior to each iteration, the interpreter binds the value of each successive list element to the quantifier variable `x`.

The quantifier defines its own scope, with the quantifier variable `x` defined locally in that scope. The type checker currently allocates a symbol table in the `chkQuant` function, but the offset computation probably needs to be upgraded in order to work. (It's opportune that you just did the update of the tuple symbol table offset computation.)

When the quantified `expression` is evaluated in the `forall` scope, it will do a Lookup on all the variables it the expression, including the local quantifier variable. This means that `doForall` needs to descend into the quantifier's scope. In this way, Lookup will find the local quantifier variable in the normal way.

Interpreting the general form of quantifiers is where the trick comes in. For a case like the very first example,

```
forall (rec:PersonRecord) rec.name != nil
```

the universe of quantified values is unbounded. This is because the number of possible values of `PersonRecord` is unbounded, e.g., because the number of strings and integers is unbounded.

The trick we're going to use is to evaluate an unbounded quantifier with a universe of values that have been created since the beginning of an interpreter session. A "session" will eventually be interactive, but for now it means during the interpretation of a single file.

What I have in mind for keeping track of values is to keep separate (hash) tables of each value type. E.g., whenever a value of type `PersonRecord` is created, a pointer to that value will be stored in the `PersonRecord` value table. Then when a quantifier over `PersonRecord` is evaluated, the value universe will be all values in the `PersonRecord` value table.

There's a bit of thinking that has to go into when values of a particular type are created. Think about the following example:

```
var pr: PersonRecord;
pr := {"Jones", 25};
```

The raw value "`{"Jones", 25}`" is not known to be of type PersonRecord in `doTupleConstructor`. It only gets that type when it's bound to the `pr` variable that's been explicitly declared as `PersonRecord`. So, determining that a new value of type `T` has come into existence can go something like this:

a. check for new value creation in the `Bind` function
b. do so by looking up the value in the value table for the type of identifier the value is being bound to
c. if the value is not already in the table, put it there

Another context in which a new type value is potentially created is as the return of an operation. I'm not 100% sure, but I think this case will probably be caught by a value binding of a variable in a post condition. This requires a bit of thought.

There are probably some tricks that can make the table search more efficient, What comes to mind is the way the string.{h,c} abstraction works, where any two or more literal strings that are lexically equal will always be represented by the same value in the string table. And this means that string equality can be performed with just pointer equality. In a similar way, any lexically equal tuple literals could be stored in a table such that pointer equality could be used instead of deep equality.

For starters, you can leave efficiency out of the picture, until the concept works.

How this Fits into the Slightly Grander Scheme of Testing

The way that a tester will "populate" a value universe goes something like this:

a. If there are some initial "canned" values, the tester can do a bunch of global expressions evaluations at the beginning of a session.
b. As test cases are executed, any test function that returns a value will cause that value to be added to the type-specific value universe.
c. The evaluation of unbounded quantifiers will get stronger as the value universe expands.