Chaiscript Additional Topics

Assignment

Chaiscript knows three different assignments:

- Constant assignment (which creates a constant variable)
- Reference assignment (which creates an alias)
- Clone assignment (which copies the value)

Unfortunately only two assign operator keywords (“:” and “:=”) are used for this:

CODE	ASSIGNMENT	DESCRIPTION
`x=0;`	clone assignment	Assign value 0, other values can be assigned later.
`x:=0;`	constant assignment	Assign value 0 and mark x as constant. It is not allowed to change x any more.
`x=y;`	clone assignment	Assign value of y to x, only if x is not constant.
`x:=y;`	reference assignment	Make x an alias for y. Both x and y will contain the same value. x will not become constant.
`x=clone(0);`	clone assignment	Same as x = 0;
`x:=clone(0);`	clone assignment	Assign value 0. If x is an alias for a different variable, then remove this reference and make x an independet variable.
`x=clone(y);`	clone assignment	Same as x = y;

CODE	ASSIGNMENT	DESCRIPTION
`x:=clone(y);`	clone assignment	Assign value of y. If x is an alias for a different variable, then remove this reference and make x an independet variable.

In general: Once a type is assigned, then only values of the same type can be assigned. Automatic conversion applies if available.

The builtin function `clone()` will create a copy of the provided argument. For nested data objects only the root object is copied (`clone()` will not do a deep copy).

Conclusion

- Clone assignment will be the most often used assignment.
- Constant assignment might be useful, but could be replaced by clone assignment.
- Reference assignment may lead to strange side effects and should be avoided. However it might be faster than clone assginment in some special cases (see the quicksort example below).

Details about Variables

As noted above, a variable might be in a certain state and of a certain type. There are some helper functions which provide some more detailed information about the variable. Here are examples along with the return values of `type_name()`, `is_var_undef()` and `is_var_const()`.

EXAMPLE	TYPE_NAME(X)	IS_VAR_UNDEF(X)	IS_VAR_CONST(X)
`var x;`	`""`	`true`	`false`
`var x:=5;`	`"int"`	`false`	`true`
`var x=5;`	`"int"`	`false`	`false`
`var x:=clone(5);`	`"int"`	`false`	`false`
`var x=int64_t(5);`	`"int64"`	`false`	`false`
`var x:="abc";`	`"string"`	`false`	`true`
`var x="abc";`	`"string"`	`false`	`false`
`var x:=[];`	`"Vector"`	`false`	`true`
`var x=[];`	`"Vector"`	`false`	`false`
`var x:=Vector();`	`"Vector"`	`false`	`false`
`var x=Vector();`	`"Vector"`	`false`	`false`
`var x:=y;`	depends on y	`false`	depends on y
`var x=y;`	depends on y	`false`	`false`

EXAMPLE	TYPE_NAME(X)	IS_VAR_UNDEF(X)	IS_VAR_CONST(X)
`var x=fun() {};`	`"Function"`	`false`	`false`

Conclusion

- Using a constructor (`int64_t`, `Vector`, ...) in a variable declaration will always lead to a none-constant variable.
- The function `type_name()` is useful to figure out how to deal with a variable of unkown type.

Function Arguments

If a (not constant) variable is passed as an argument to a function, then it is passed as reference: Any modification to this variable inside the function will remain after the function call.
An error is generated if the function tries to assign a value to an argument which is not a variable:

```
1  def add_dot(s)
2  {
3    print(is_var_return_value(s));
4    print(is_var_const(s));
5    s = s + "."; // error if argument for s is not a variable
6  }
```

CALL	BEHAVIOR	IS_VAR_RETURN_VALUE(S)	IS_VAR_CONST(S)
`var a = "a";` `add_dot(a);`	`a=="a."`	`false`	`false`
`var a = "a";` `add_dot(a+"");`	error	`true`	`false`
`add_dot("");`	error	`false`	`true`

Conclusion

- If either `is_var_return_value()` or `is_var_const()` return true, then the argument is not a variable and the function must not assign a value to this argument.
- If an assignment to the function arguments is required, then clearly describe this in the documentation for this function.
- It is a good programming pactice to modify only the value of the first function argument.

Lambda Functions

A lambda function is a nameless function. Instead of “`def function-name`” Chaiscript will just use “`fun`” as keyword.

```
var f = fun(x) {return x*x;};
print(f(3));
```

Lambda functions are often used if a function expects another function as argument. In such cases, the use of a lambda function is more compact
(see also the quicksort example below).

Global Variables

There are two different declarations for variables, which define the visibility of the declared variable.

var x;

Declared on file level: x is visible only on file level, but not within any function definition.
Declared inside functions: x is visible only within the function, but not within any function definitions within a function. The following example will not work because x is not visible inside the lambda function:

```
def g(Function f) {
  f();
}
def h() {
  var x = 5;
  g( fun() {print(x);} ); // error: can not find 'x'
}
h();
```

global x;

Declared on file level: x is accessable everywhere, especially inside function definitions.
Declared insided a function: x is accessable inside nested function definitions. This will solve the above error with the lamda function:

```
def g(Function f) {
  f();
}
def h() {
  global x = 5; // x can be used within the lambda function
  g( fun() {print(x);} ); // g will print the value 5
}
h();
```

Conclusion

- Use “`var`” for local variables inside a function
- Use “`global`” for variables which must be accessible everywhere

Quicksort

There is no build-in sort method in Chaiscript, so let’s implement quicksort for Chaiscript. The Quicksort algorithm is based on “https://en.wikipedia.org/wiki/Quicksort (https://en.wikipedia.org/wiki/Quicksort)“. Quicksort will use some of the above mentioned topics.

```
 1  def swap(Vector v, int a, int b) {
 2    var e := v[a];
 3    v[a] := v[b];
 4    v[b] := e;
 5  }
 6
 7  def partition(Vector v, int lo, int hi, Function is_lower_than) {
 8    var i = lo;
 9    var j;
10    for( j = lo; j < hi; ++j ) {
11      if ( is_lower_than(v[j], v[hi]) ) {
12        swap(v, i, j);
13        ++i;
14      }
15    }
16    swap(v, i, hi);
17    return i;
18  }
19
20  def quicksort(Vector v, int lo, int hi, Function is_lower_than) {
21    if ( lo < hi ) {
22      var p = partition(v, lo, hi, is_lower_than);
23      quicksort(v, lo, p-1, is_lower_than);
24      quicksort(v, p+1, hi, is_lower_than);
25    }
26  }
```

The “swap()” function will exchange two elements by using the reference assignment. This is more faster than using a clone assignment. In all other cases the more common clone assignment is used.

The purpose is to sort the elements of vector v, which is passed by reference to swap, partition and quicksort. This elements of vector v are modfied by swap, partition and quicksort.

The last argument of the quicksort function must be the compare function for two elements (return true, if first argument is lower than the second argument). This can be implemented with a lambda function:

```
1 var v = [3, 2, 4, 1, 6, 7, 2, 4, 1, 6, 2, 8, 4, 1, 6, 5, 7, 2, 4, 1, 6
2 quicksort(v, 0, v.size()-1,  fun(a,b) { return a < b; } );
3 print(v);
```