Higher Order Functions in Moto

Design Goals

We should be able to define functions which can take functions as arguments and return functions. We should be able to define variables with functional types and call them as functions. We should be able to partially apply existing functions and methods creating new functions inline

Design Overview

Grammar Changes

Declarations of 'function' typed variables will look like those that follow. The function variable name will follow the type. Since functional types are indistinguishable at parse time from function calls we will need to determine expression validity in motov.

int () func
int (String) func
int (String,char) func

int () () func
int (String) () func
int () (String) func
int (String) (char) func
int (String,char) () func
int (String,char) (float) func

int (String ()) func
int (String (char)) func

Just like all other reference types we will be able to cast functional types to and from objects

<int () >
<int (String) >
<int (String,char) >

<int () () >
<int (String) () >
<int () (String) >
<int (String) (char) >
<int (String,char) () >
<int (String,char) (float) >

<int (String ()) >
<int (String (char)) >

Like C we will use the & operator to specify that we are interested in a function's address, not in its evaluation. The necessity for this extra token is clear since there isn't another good way to differentiate a no-arg function address from a call to that function. The other important thing to notice here is that since unlike C, functions in moto are uniquely identified by the function name AND arguments, those argument types must be specified when referencing

the function definition.

&f ()
&f (<String> ?)
&f (<String> ?, <int> ?)

&f (<int () > ?)
&f (<int (String) > ?)
&f (<int (String,char) > ?)

&f (<int () () > ?)
&f (<int (String) () > ?)

Anonymous Function Calls

{ f(&1,&2); }
{ f(&1,&2); return &1; }

Partially evaluated HOFs

&f ("funk")
&f (<String> ?, 27)
&f (&g (<Object> ?))

Design Issues

Everything is about to be 'callable'

Most all expressions in HOF world can result in something that is 'callable'. As a result the pattern '(' expression_list ')' is about to be usable is a slew of new ways e.g.

foo(bar)(maka)
fnarr[27](crazy,wacky)
bust.out("all over")

Thus we need to make some fairly drastic changes to the way the parser deals with functions and methods to allow for this

Fist off we need to un-limit the expressions '(' expression_list ')' can follow since it should now be up to the verifier to decide if the expression its following is 'callable' or not

```
postfix_expression
      : postfix_expression INC
      | postfix_expression DEC
      | postfix_expression functional_expression
      | dereference_expression
      | array_subscript_expression
      | primary_expression
;

functional_expression
      : OPENPAREN CLOSEPAREN
      | OPENPAREN expression_list CLOSEPAREN
```

```
;

dereference_expression
     : postfix_expression DOT NAME
;
```

Unfotunately as soon as we do that we get a shift/reduce conflict with embedded constructor definitions ... time to fix a long standing bug with the parser design:

```
embedded_function_definition_statement
     : function_declaration OPENPAREN declaration_list CLOSEPAREN
OPENCURLY embedded_statement_list CLOSECURLY
     | function_declaration OPENPAREN CLOSEPAREN OPENCURLY
        embedded_statement_list CLOSECURLY
     | function_declaration OPENPAREN declaration_list CLOSEPAREN
OPENCURLY CLOSECURLY
     | function_declaration OPENPAREN CLOSEPAREN OPENCURLY
CLOSECURLY
;

embedded_constructor_definition_statement
     : NAME OPENPAREN declaration_list CLOSEPAREN OPENCURLY
        embedded_statement_list CLOSECURLY
     | NAME OPENPAREN CLOSEPAREN OPENCURLY
        embedded_statement_list CLOSECURLY
     | NAME OPENPAREN declaration_list CLOSEPAREN OPENCURLY
CLOSECURLY
     | NAME OPENPAREN CLOSEPAREN OPENCURLY CLOSECURLY
;


embedded_class_statement_list
     : embedded_class_statement
     | embedded_class_statement_list embedded_class_statement
;

embedded_class_statement
     : embedded_constructor_definition_statement
     | embedded_definition_statement
     | embedded_declare_statement SC
;
```

The above grammar changes not only fix the shift reduce but prohibit statements other than declarations or definitions immediately inside of embedded class definitions

So now the question becomes HOW THE HELL DO WE GET THE FUNCTION OR METHOD NAME IN MOTOX!

We know we want to evaluate the first operand.

Right now there are only two possibilities for it:
1) Its a dereference opcell (holding the class instance and method name)
2) Its an id cell (holding the function name)

In the future there will be many more. These first two possibilites will always be special cases because we cannot determine the function to call by the value of the first operand alone ... we need the arguments. In fact I'd argue that the first operand has no value in isolation because it has no definite type!

All other expression used as the first operand better return something 'functional' though

Function Addressing

We need to be able to get the addresses of functions. We will do this by preceding the function call with an & like in C

```
unary_expression
      : MINUS postfix_expression
      | INC postfix_expression
      | DEC postfix_expression
      | NOT postfix_expression
      | BITWISE_NOT postfix_expression
      | BITWISE_AND function_identifier
      | allocation_expression
      | free_expression
      | postfix_expression

function_identifier
      : NAME OPENPAREN CLOSEPAREN
      | NAME OPENPAREN argument_identifier_list CLOSEPAREN

argument_identifier_list
      : argument_identifier
      | argument_identifier_list COMMA argument_identifier

argument_identifier
      : REL_LT_M type REL_GT_M QUESTION
```

Parser Changes for Functional Types

We need to be able to specify functional types.

```
type
      : basic_type
      | DEF functional_type
;

basic_type
      : NAME
      | NAME array_declaration_list
;

functional_type
      : basic_type OPENPAREN CLOSEPAREN
      | basic_type OPENPAREN type_list CLOSEPAREN
      | functional_type OPENPAREN CLOSEPAREN
      | functional_type OPENPAREN type_list CLOSEPAREN
```

```
;

type_list
     : functional_type
     | basic_type
     | type_list COMMA functional_type
     | type_list COMMA basic_type
;
```

Instantiating arrays of functional types

```
allocation_expression
     : NEW NAME OPENPAREN CLOSEPAREN
     | NEW NAME OPENPAREN expression_list CLOSEPAREN
     | NEW NAME array_index_list
     | NEW NAME array_index_list array_declaration_list
     | NEW DEF functional_type array_index_list
     | NEW DEF functional_type array_index_list
array_declaration_list
;
```

Verifier Changes

So types aren't as simple as they used to be. In HOF world types can actually be recursively defined structures ... so simply retrieving dim and typen won't suffice. Instead we will need type extractor functions for MotoX

```
MotoType* motov_extractType(const UnionCell *p)

     if the cell type == TYPE
          Extract the type name and dimension
          Retrieve the MotoType

          if its not defined
               Throw an error
               recover by instantiating an Object

     if the cell type == DEF

          Extract the return type
          Extract the type arguments
          For each type argument, extract the argument

          Retrieve (construct) the HOF MotoType

     return type;

MotoType* motoi_extractType(const UnionCell *p)

     if the cell type == TYPE
          Extract the type name and dimension
          Retrieve the MotoType
```

```
     if the cell type == DEF

          Extract the return type
          Extract the type arguments
          For each type argument, extract the argument

          Retrieve (construct) the HOF MotoType

     return type;

MotoType* motoc_extractType(const UnionCell *p)

     if the cell type == TYPE
          Extract the type name and dimension
          Retrieve the MotoType

     if the cell type == DEF

          Extract the return type
          Extract the type arguments
          For each type argument, extract the argument

          Retrieve (construct) the HOF MotoType

     return type;
```

Thus in motox the following functions need to be changed to get the types passed to them using the recursive type extraction method

motox_array_new - modify this function to get the type of the array being instantiated by calling motox_extractType

motox_declare - modify this function to get the type of the variable being declared by calling motox_extractType

motox_cast - modfiy this function to get the type of the cast by calling motox_extractType

motox_define - modify this function to get the return type and argument types by calling motox_extractType

motox_fn - modify this function to get the argument types by calling motox_extractType

Function Addressing

The verifier, interpreter and compiler will need new handlers for the FUNCTION_IDENTIFIER opcode.

motov_function_identifier(UnionCell* p) -
- extract the function name
- for each argument
    - extract the argument type and put it in the args array
- see if the function for this name and arguments is defined (exact match only ?)

- if its not, throw a no such function error
- construct the functional type
- instantate and push onto the stack a motoval with the functional type

motoi_function_identifier(UnionCell* p) -
- extract the function name
- for each argument
  - extract the argument type and put it in the args array
- retrieve the function for this name and arguments
- construct the functional type
- instantate a motoval with the functional type
- set its refval.value to the moto function record :)
- push the motoval onto the stack

motoc_function_identifier(UnionCell* p) -
- extract the function name
- for each argument
  - extract the argument type and put it in the args array
- retrieve the function for this name and arguments
- construct the functional type
- instantate a motoval with the functional type
- set its codeval.value to '&'+ function cname
- push the motoval onto the stack

Calls to function typed expressions

So now that everything is callable we need to be able to perform function calls on pretty much anything. The hard part about this is the base case when we are trying to call <exp>.NAME(...) or NAME(...) since these could be true functions or methods.

First off, what should happen is that if a function foo() is defined, but so is the variable foo, the variable should be used. This makes the scoping rules like C's

motov_fn
- Evaluate self
- if this is an FN operand and callee is an id value type
  - see if its a global function or method
- if this is a METHOD operand and callee is an id value type
  - see if its a global method
- if this method or function was NOT global
  - call motoi_fcall
- otherwise
  - call the global method or function

motov_fcall
- motov the expression being called
- Is it a functional type ?
  - If not generate error: "cannot call non functional expression"
- motov the arguments
- get the expression type
- For each type argument,
  - Verify the passed argument is implicitly castable
- return the type's atype

motoi_fn
- Evaluate self
- if this is an FN operand and callee is an id value type
    - see if its a global function or method
- if this is a METHOD operand and callee is an id value type
    - see if its a global method
- if this method or function was NOT global
    - call motoi_fcall
- otherwise
    - call the global method or function

motoi_fcall
- Motoi the expression being called
- Motoi the arguments
- If the function being called was externally defined
    - call it
- If the function being called was defined in moto
    - call it

Assignments of function typed expressions

We actually shouldn't need to do anything special here

Casting Rules

Function calling rulles:

Rule 1: If a function takes an argument of type void(Y) we should be able to pass it an argument of the form X(Y) since wherever it is used the return value is ignored

Rule 2 : If a function takes an argument of the form X(Y) we should be able to pass it an argument of the form Z(Y) where is of type X or a descendant of type X

Rule 3 : if a function takes an argument of the form X(Y) we should be able to pass it an argument of the form X(Z) where Z is of type Y or Z is an ancestor of type Y

Assignment / Implicit Casting Rules:

Rule 1 : If a variable has type void(Y) we should be able to assign it a function of type X(Y) since wherever it is used the return value is ignored

C Code Generation

C Code for HOF Types

Variables that store function pointers that return other function pointers are hard to define in C. They require typedefs for the functional return type. We can however typedef any type we would like in C recursively

A function that takes an int and returns an int:

typedef int (*_iFoie_)(int);

Afunction that takes a float and returns a function that takes an int and returns an int

typedef _iFoie_ (*_ioieFofe_)(float);

We should probably recursively typedef any functional declarations.

This would work like

0) if this type has a C typedef already return it
1) r = typedef for the return type
2) for each argument
    2.1) a_i = typedef for the argument type
3) return _(r-'F')Fo(a_x)e

Functional typedefs must be output prior to class declarations since class declarations may depend on them

Fuctional typedefs should be added to both a vector and a symbol table. The vector will retain the order in which they were defined (least complex to most complex)

Functions that return arrays of functions are even more hairy :)

The above algorithm could probably work ... but its not necessary since type checking is done in motov there is really nothing stopping us from making the C type for any function type a void* . Then in motoc fcall we generate the appropriate cast needed to call the function.

MDF C Symbols for functions that take functions as arguments

Functional arguments to functions cause a new headache for C symbol generation. Since foo(int(String)) and foo(int(int))are both allowed under the HOF rules of function overloading, each must have a different C symbol. The previous algorithm for type differentiation can work here with some modification. The necessary modification involved recording of the functional return types because they do matter.

foo(int()) => _foo_Fioe
foo(string()) => _foo_FSoe

foo(int(String)) => _foo_FioSe
foo(int(int)) => _foo_Fioie
foo(int,int(int)) => _foo_iFioie

The basic idea is that if an argument to a function is a functional type, output 'F', then the return type for that for that argument, the o (for open paren), then the argument types, then e (for the close paren)

F <return type> o <argument types> e contains the functional argument

When an argument is encountered that starts with F, capture to the 'e' that closes it and recurse on the captured string.

This algorithm should be implemented in both **moto** and **mxc** for C symbol name

generation.

UnAnswered Questions

Q. How do we know if the return val to a moto defined function should be tracked ?

Q. How do we know the base name of a moto defined function to push onto the stacktrace stack ?

To be able to answer questions like those above at runtime in compiled code it is clear that what gets stored with function typed values must be more than just a function pointer. It must be a structure that contains both the function pointer and relevant meta information

```
typedef struct function {
        int isTracked;
        void* freefn;
        String fname;
        void* fn;
} Function;
```

Passing functions to externally defined classes

So three key features must be implemented to make this possible
1) Functional types must be parsable by mxc
2) MXC should output the fully qualified string canonical type as the type for both the return type and the argument types
3) We need to define the mechanism by which external functions can call functions passed to them:

There are really four possibilities for this case

3.1) An external function wants to call a compiled mdf
3.2) An external function called by compiled code wants to call another external function
3.3) An external function called by interpreted code wants to call another external function
3.4) An external function wants to call an interpreted function

3.1) An external function wants to call a compiled mdf

f(x,y,z)

3.2) An external function called by compiled code wants to call another external function

f(x,y,z)

3.3) An external function called by interpreted code wants to call another external function

f(x,y,z) interpreter passes C symbol (which we may or may not have!)

3.4) An external function wants to call an interpreted function

Changes to mx.y

```
m_type_specifier
      :  VOID
      |  TYPE_NAME
      ;

m_type
      :  m_type_specifier
      |  m_type  m_array_declaration
      |  m_type  PAREN_L  PAREN_R
      |  m_type  PAREN_L  m_type_list  PAREN_R
;

m_type_list
      :  m_type
      |  m_type_list  COMMA  m_type
;
```

remove m_array_declaration_list

major simplifications need to be done to mx.l and mx.y (including the removal of the TYPE_NAME token and the CT and CN scopes) to remove shift reduce conflicts

Recognizing Functional Types

Functional types can occur in two very ambiguous circumstances :

1) <Functional Type> <Name> - corresponds to declaration
2) new <Functional Type> '[' <expression> ']' - corresponds to array instantiation

The problem is that the above patterns depend on the right side of an ***indeterminate number of tokens*** to identify the functional type (infinite lookahead) where as when YACC or bison see the start of the pattern

<NAME> '(' <NAME>

A function call expression is matched. Thus the above cannot be used directly in most places to match a functional type

What we need to do is do the look ahead in the lexer wherever we MIGHT be seeing functional types and keep looking ahead until we're SURE we're seeing functional types

```
<EMB,D>"new"{S}{T}{S}"("({T}|{S}|[,()]|"[]")*")"({S}"[") {

	/* FIXME: I feel so dirty :( */

	int i=0,j=0, opens = 0, matchedHOF = 0;
	char* mtext = estrdup( yytext+3);
	char word[255];

	for(i=0;i<strlen(yytext+3);i++){
		char c = yytext[i+3];

		/* Make sure there are no reserved words being used */
		if(c==',' || c=='['  || c=='('  || c==')' || c==']' ||
			c==' ' || c=='\n' || c=='\r' || c=='\t') {
			word[j] = '\0';
			if(   isReserved(word) ) {matchedHOF=0; break;}
			else if(matchedHOF == 1 && j>0) { break; }
			else j=0;

		}

		if(c == '(') {
			opens++; matchedHOF=0;
		} else if(c == ')') {
			if(opens == 0) {matchedHOF=0; break;}
			else opens--;
			if(opens == 0) matchedHOF=1;
		} else if(c!=',' && c!='['  && c!='('  && c!=')' && c!=']' &&
					c!=' ' && c!='\n' && c!='\r' && c!='\t')
			{word[j] = c; j++;}
	}

	for(i=strlen(mtext)-1; i>=0;i--)
```

```
            unput(mtext[i]);
      free(mtext);

      if(matchedHOF){
            lvalp->ivs = createIVS(env);
            return HOF_NEW;
      }else{
            lvalp->ivs = createIVS(env);
            return NEW;
      }
}

<EMB,D>{T}{S}"("({T}|{S}|[,()]|"[]")*")"({S}"[]")*{S}{T} {
      int i=0,j=0, opens = 0, matchedHOF = 0;
      char word[255];

      for(i=0;i<strlen(yytext);i++){
            char c = yytext[i];

            /* Make sure there are no reserved words being used */
            if(c==',' || c=='['  || c=='('  || c==')' || c==']' ||
                  c==' ' || c=='\n' || c=='\r' || c=='\t') {
                  word[j] = '\0';
                  if(   isReserved(word) ) {matchedHOF=0; break;}
                  else if(matchedHOF == 1 && j>0) { break; }
                  else j=0;

            }

            if(c == '(') {
                  opens++; matchedHOF=0;
            } else if(c == ')') {
                  if(opens == 0) {matchedHOF=0; break;}
                  else opens--;
                  if(opens == 0) matchedHOF=1;
            } else if(c!=',' && c!='['  && c!='('  && c!=')' && c!=']' &&
                              c!=' ' && c!='\n' && c!='\r' && c!='\t')
                   {word[j] = c; j++;}
      }

   word[j]=0; i-=strlen(word);

      if(   isReserved(word) ) {matchedHOF=0;}

      if(matchedHOF) {

            char* mtext = estrdup( yytext);
            int len = strlen(yytext), ws = 0;
            // printf("### word '%s'\n",word);
            // printf("### matched '%s'\n",mtext);
            if(i<len){
                  for(j=len-1; j>=i;j--)
                        unput(mtext[j]);
            }
```

```
            if(i<len) {
            //     printf("### put back '%s'\n",mtext+i);
            }

            mtext[i]='\0';

            for(i=0;mtext[i+ws];i++){
                  char c = mtext[i+ws];
               if(c == ' ' || c == '\t' || c == '\r' || c == '\n') {ws++;i--;}
               else {mtext[i]= c;}
            }

            mtext[i]='\0';

            lvalp->svs=createSVS(env,mtext);
            countLines(env, mtext);
            free(mtext);

            // printf("### generate '%s'\n",mtext);

            return NAME;
      } else {
            int len = strlen(yytext);
            char* mtext = estrdup( yytext);
            // printf("### matched '%s'\n",mtext);
         for(i=0;i<len;i++) {
            char c = yytext[i];
            if(c == ' ' || c == '\t' || c == '\r' || c == '\n' ||
                  c == '[' || c == ']' || c == '(' || c == ')' || c == ',')
                  break;
         }
            for(j=len-1; j>=i;j--)
                  unput(mtext[j]);
            // printf("### put back '%s'\n",mtext+i);
            mtext[i]=0;
            // printf("### generate '%s'\n",mtext);

            if(strcmp("while",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return EWHILE;}
            else if(strcmp("for",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return EFOR;}
            else if(strcmp("switch",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return ESWITCH;}
            else if(strcmp("case",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return ECASE;}
            else if(strcmp("default",mtext) == 0) {lvalp->ivs =
createIVS(env); free(mtext); return EDEFAULT;}
            else if(strcmp("if",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return EIF;}
            else if(strcmp("else",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return EELSE;}
            else if(strcmp("try",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return EXCP_ETRY;}
            else if(strcmp("throw",mtext) == 0) {lvalp->ivs = createIVS(env);
```

```
free(mtext); return EXCP_ETHROW;}
            else if(strcmp("catch",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return EXCP_ECATCH;}
            else if(strcmp("finally",mtext) == 0) {lvalp->ivs =
createIVS(env); free(mtext); return EXCP_EFINALLY;}
            else if(strcmp("print",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return EPRINT;}
            else if(strcmp("return",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return ERETURN;}
            else if(strcmp("use",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return EUSE;}
            else if(strcmp("break",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return EBREAK;}
            else if(strcmp("continue",mtext) == 0) {lvalp->ivs =
createIVS(env); free(mtext); return ECONTINUE;}
            else if(strcmp("class",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return ECLASS;}

            else if(strcmp("eq",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return REL_EQ_S;}
            else if(strcmp("ne",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return REL_NE_S;}
            else if(strcmp("lt",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return REL_LT_S;}
            else if(strcmp("gt",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return REL_GT_S;}
            else if(strcmp("lte",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return REL_LTE_S;}
            else if(strcmp("gte",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return REL_GTE_S;}

            else if(strcmp("this",mtext) == 0) {lvalp-
>svs=createSVS(env,mtext); free(mtext); return THIS;}
            else if(strcmp("delete",mtext) == 0) {lvalp->ivs=createIVS(env);
free(mtext); return DELETE;}
            else if(strcmp("new",mtext) == 0) {lvalp->ivs=createIVS(env);
free(mtext); return NEW;}
            else if(strcmp("global",mtext) == 0) {lvalp->ivs=createIVS(env);
free(mtext); return GLOBAL;}

            else if(strcmp("true",mtext) == 0) {lvalp-
>svs=createSVS(env,mtext); free(mtext); return BOOLEAN;}
            else if(strcmp("false",mtext) == 0) {lvalp-
>svs=createSVS(env,mtext); free(mtext); return BOOLEAN;}
            else if(strcmp("null",mtext) == 0) {lvalp->ivs = createIVS(env);
free(mtext); return MOTONULL;}

            else {lvalp->svs=createSVS(env,mtext); free(mtext); return NAME;}
      };
}
```

Implementing Short Form

The goal of short form is as follows

1) If there is only one function with a given name than we should be able to get an identifier to that function by simply using the name

2) If there is only one function with a given name and n arguments we should be able to get an identifier to that function by simply saying

fn(?,?,?)

where n = the number of question mark arguments passed

If there is more than one function with the same name and arguments than the above identifies an **Ambiguous Function**

3) If there are multiple functions with the same name than the name alone identifies the 'no-arg' variant if there is one ... otherwise it identifies an **Ambiguous Function**

To implement short form function identification for Higher Order Functions we need to be able to retrieve:

1) All functions with a given name
2) All methods in a given class with a given name

This means that the ftable needs to be re-designed as follows

- Class + Name must be the key to the ftable
- Motoname should be changed to no longer include the arguments
- ftab_getMatch and ftab_getExactMatch must match up the argument count

We will be forced to return function identifier typed valuecells from at least 3 functions in motoX

motoX_id - when only the function / method name is specified
motoX_function_identifier - &
motoX_fn - for partially filled functions

Partial Application

To implement partial function application in the interpreter we will need to be able to store and retrieve the applied arguments dynamically. We will also need to be able to store and retrieve the self pointer (for methods).

At definition time the applied arguments (including the self pointer) need to be stored by some method in some object (a closure)

At function call time we need to be able to retrieve the closure arguments

With EDFs we have no idea of the names of the arguments, only the argument order so it seems the storage for partially filled arguments needs to be based on the arguments

ordinal value

Right now for EDFs motoi fills different arg arrays with types and vals
For MDFs motoi reads the types and vals right off of the op stack

If we actually had an args array everywhere we could use the following algorithm

for i=0;i<real_fn_argc;i++)
    if colosure arg [i] is defined, move args >= i up one and insert closure arg [i] at argv [i]

So Step 1) Everybody should make use of an args array

What sort of structure can we use to determine
1) If closure arg [i] is defined
2) what is the value of closure arg i

Option 1) Int Hashtable
    Use an int hashtable to map arg number to objects wrapping arguments. This means that the arguments will require dynamically allocated object wrappers in addition to Int hashtable storage

Option 2) Use a class like structure to map

The args array is filled in motoi.c by motoi_fillArgs. It is however an awful function!

It puts the address of the val->value into argv for all non reference types (meaning we definitely cannot free the motovals)

The call to moti_fillArgs is often followed by a call to motoi_convertArgs which attempts to cast the elements of argv.

MotoDefined functions don't deal with this sort of bullshit. Instead they get the MotoVal that was passed, then call moto_setVarVal to cast it into the appropriate typed argument

setVarVal calls moto_castVal which is much smarter than motoi_convertArgs

We do of course need argv in the end

so it seems that what would be much smarter would be to have

1) motoi_fillArgs put the MotoVals themselves into an array (popping them off the stack at the same time!)
2) motoi_convertArgs call

Partial Function Verification

Methods

The absolute simplest form of partially applied functions are methods. The only applied variable we need to store with them is the self pointer. We do not need a new closure object to implement them.

motox_dereference_rval - If the member var is not defined see if the mathod is

motox_fcall

motox_function_identifier

motox_ifcall

Compiled Code for anonymous functions

When a partially applied function is identified we need to do a couple things

1) Generate the code for building the appropriate function structure

2) Tell the system to generate an anonymous function which takes a function object as well as the appropriate unfilled arguments and calls the original function

On fcall we will need to

3) pass f to the anonymous function on fcall

Creating the function structure will be done by way of a call to func_createPartial with arrays built inline for pargi and pargt

The anonymous function itself should look like

```
__ANON_X(Function* f,...){
        origfn(real arg, (cast)fakearg );
}
```

In order to generate the anonymous function
fakearg

Algorithms

motox

int motox_calleeIsDDF(const UnionCell*p, MotoVal* self) - Returns true if the callee specified is a dynamically defined function (identified function or partially applied function or method)

A callee is considered dynamically defined iff:

1) The opcode of the callee is FN but the function name being called is a variable
2) The opcode of the callee is FN but we are inside a class definition and the function name being called is a class member variable
3) The opcode of the callee is METHOD but the method name being called is a class member variable
4) The opcode of the callee is FCALL

int motox_lookupMethodOrFn(MotoFunction **f, MotoVal* self,char* fbasename, int argc, char types, int opcode) -** Try to locate a moto function or method with the specified base name, arguments, and cononical argument types.

1) Try finding a method using the type of 'self' as the classname if self is != null.
2) If no method is found and the opcode allows for the possiblity that we are interested in functions as well than try looking for one of those.
3) In the end return 'code' and set f if we find anything.