Design & Implementation of the MeldC Translator

Esther Yeuk Kay Woo

Technical Report
CUCS-045-92

Columbia University
Department of Computer Science
New York, NY 10027

March 1, 1992

The Programming Systems Laboratory is supported by National Science Foundation grants CCR-9106368 and CCR-8858029, by grants and fellowships from AT&T, BNR, Bull, DEC, IBM and SRA, by the New York State Center for Advanced Technology in Computers and Information Systems and by the NSF Engineering Research Center for Telecommunications Research.

Contents

Chapter 1

Overview

The MeldC language is a "meld" of conventional C and support for object-oriented programming, concurrent programming and distributed programming (to allow access to objects and data stored in remote locations). This report describes the compiler support for the language. The compiler translates MeldC programs into C code and links with the runtime library to produce object code. Classes in MeldC are translated into C structures that describe objects, and methods are translated into C structures that describe frames in runtime. The compiler also implements static strong typing and static binding of methods.

The source code of the MeldC compiler lives in the following files: active.c, build_classes.c, codegen.c, decl_tab.c, extern.c, generic.c, header.c, list.c, main.c, support.c, and machdep.c. Each of these files more or less implements a particular functionality of the compiler or a particular feature of the language. The compiler is implemented in the C language with approximately 19424 lines of code of which 7074 lines are lex/yacc code. Yacc generates approximately 417 rules for the MeldC grammar. The executable of the MeldC compiler is 638976 bytes compiled on sun4 machine and 546208 bytes compiled on a mips machine.

1.1 Design Overview

To avoid enforcing forward declaration of classes, the MeldC compiler is forced to be a two pass compiler. An external file, `.info`, enables exchange of information between the two passes. Since the sole purpose of the first pass is to parse the input file and output relevant information to the external file, the first pass is simply a dumb parser. The second pass of the compiler involves more intricacies.

The second pass centers around three main data structures, the definition table, `defn_tab`, which stores types and class definitions, the declaration table, `decl_tab`, which keeps scoping and stores current local variables, and the class hierarchy, `class_hierarchy` which keeps all inheritance information about classes. There are redundancy between the definition table and the class hierarchy tree since they both keep information about instance variables and methods. It was designed this way just because it was easier to build a new data (tree) structure for inheritance than to modify the structure of the definition table.

Some well defined modules implement particular language features and others perform particular functionalities in the compilation. The file `active.c` implements *active values*, `generic.c` implements *generic object*, and `extern.c` implements the `import/export` capability of MeldC. `build_classes.c` implements multiple inheritance as well as maintaining the class hierarchy. `decl_tab.c` maintains both the definition table and the declaration table, and also implements functions relevant to typing. The file `list.c` supports the data structure used to store expressions and `codegen.c` performs code generation for function calls. The file `support.c` is a buffer management module and `machdep.c` contains code which are machine dependent.

1.2 Implementation Overview

This section describes sequentially what happens during the compilation process. The MeldC compiler is a two pass compiler. In its first pass, detailed information about classes and methods are gathered and written to an external file, `featurename.info`. The format of the `.info` file is described section X. If multiple files are being compiled together, the compiler parses through all input files with the first pass before going into the second pass to resolve any circular dependency which might occur with the `import/export` construct in the language or which have been caused by inheritance. After the first pass, the function `handle_exports()` is called to write `export` information to the `.info` file.

The second pass of the compiler starts by reading in global data declarations which usually resides in include files. The definitions are built into the compiler's data structure but NO code generation is done at this point. It is assumed that it suffices to include the header files in the cfile generated. What this implies is that any explicit data declarations appearing before a feature will be lost!

Before parsing the feature in the second pass, the compiler reads the `.info` file corresponding to the feature and builds the class definitions into its two main data structures: the definition table, `defn_tab` and the class hierarchy, `class_hierarchy`. Classes specifies by `import` statements are also brought in. MetaClass is `implicitly imported` at this point. Then the class hierarchy tree is checked for redundant parent links by `trim_inheritance_tree` (not implemented), for duplicating instance variables by `check_duplicate_instance_var()` and for non-existent classes by `check_non_existent()`.

As the parsing continues in the second pass, expressions are built up recursively enforcing strong type checking, and code is generated at the end of each statement.

At the end of the second pass, a types table which contains information about all types used in the feature is generated in the cfile. This data structure is used in runtime for linearizing structures and objects.

1.3 Auxiliary Files

Several intermediate files are used to aid the transfer of information between compilation stages. The MeldC compiler is a two pass compiler and two files, a `featurename.info` file and a `filename.act` file are generated in the first pass for each input `filename.m` file.

The `.info` file contains detailed information about classes in the feature. The second pass will read this file and build the classes accordingly. It has the following format:

```
FEATURE <featurename>
CLASS <classname> MP_TYPE <"meldtype" | "ctype">
```

```
[ PARENT <<featurename>[<classname>]>, <classname> ]
   <instance vars>
   ...
METHOD <return type> <method name>
   <type> <parameter name>
   ...
END METHOD
METHOD <return type> <method name>
   <type> <parameter name>
   ...
[ MERGE_DECL <classname> (<pname> ...), ...]
END METHOD
END CLASS
 *
CLASS ...
END CLASS
END FEATURE
EXPORT_INFO
feature "fname" class "cname"
feature "fname" class "cname" object "objectname"
...
END EXPORT_INFO
```

The `.act` file contains information about active values in each class. See section N for details.

The second pass generates a corresponding `filename.h`, `filename.c`, for each input file. These are C code translated from the programmer's MeldC source code.

Several files are generated to provide information to the MeldC linker. A `featurename.i` file is generated for each input file, and it contains initialization expressions for class objects as well as global objects. The MeldC linker uses this file together with the file called `linker_info`, which is generated for each compilation and contains the names of the features which need to be linked. The MeldC linker then generates a file `meldC.out.c` and together with the files `filename.c & filename.h`, object code is produced by compiling these files with the C compiler and linking with the MeldC runtime library.

A symbolic link `featurename.h` is created to the file `filename.h`.

Chapter 2

The MeldC Syntax

2.1 Keywords

MeldC has fifty-two reserved keywords that cannot be used for variables or changed by the user. In addition to these keywords, identifiers created by the user may not include the dollar-sign symbol ("$"), the at-sign ("@"), or begin with two adjacent underlines ("__").

Keywords marked below with a dagger ("†") are C keywords as well as MeldC keywords.

$selector	delayuntil	if †	short †
$self	do †	implementation	signed †
$sender	double †	imports	sizeof †
$thread	else †	int †	static †
auto †	end	interface	struct †
begin	enum †	long †	switch †
break †	exports	merges	to
case †	extern †	methods	typedef †
char †	feature	object	union †
class	float †	register †	unsigned †
const †	for †	respond	void †
continue †	from	return †	volatile †
default †	goto †	send	while †

2.2 BNF

In the BNF that follows, constructs for methods, class definitions, and other components of objects are shown, as well as extensions of the C language for special use by MeldC.

Readers who are not familiar with standard C syntax (either Kernighan and Ritchie C or ANSI C) are referred to Appendix A13 of "The C Programming Language"(Second Edition) by Kernighan and Ritchie, where a complete BNF for conventional C can be found. Since MeldC has adopted many of the elements of conventional C, only those grammatical elements that differ between the two languages are listed here. For example, the following non-terminals, although used in MeldC, are not defined here; their definitions can be found in K&R:

compound-statement
declaration-list
declarator
expression
identifier-list
init-declarator-list
pointer
statement-list
storage-class-specifier

likewise,

declaration-specifiers	is modified (see the definition below)
type-specifier	is extended (see the definition below)

The complete MeldC BNF follows:

program ::=
    feature *feature-name*
    interface *externals**
    implementation
        body
    end feature $feature\text{-}name_{opt}$

externals ::=
    exports *port-list*
  | **imports** *port-list*

port-list ::=
    port
  | *port-list* , *port*

port ::=
    class-name
  | *object-name*
  | *feature-name* **[** *class-name* **]**
  | *feature-name* **[** *object-name* **]**

body ::=
    *meldc-decl**
meldc-decl ::=
    *data-declaration-list**
  | *class-decl**

data-declaration-list ::=
    data-declaration+

data-declaration ::=
    declaration-specifiers $init\text{-}declarator\text{-}list_{opt}$;

class-decl ::=
    `class` *class-name ::=*
        class-body
    `end class` *class-name*$_{opt}$
  | `class` *class-name* `merges` *super-class-list ::=*
        class-body
    `end class` *class-name*$_{opt}$

super-class-list ::=
    super-class
  | *super-class-list* **,** *super-class*

super-class ::=
    class-name
  | *featurename* `[` **class-name** `]`

class-body ::=
    inst-variables `methods` *method-part-list*

inst-variables ::=
    *data-declaration-list**

method-part-list ::=
    *method-part**
method-part ::=
    method-decl
  | *active-value-rules*

active-value-rules ::=
    (active-value-list) `-->` *compound-statement*

active-value-list ::=
    active-value
  | *active-value-list , active-value*

active-value ::=
    identifier
  | *identifier @*

method-decl ::=

*type-specifier func-dcltr para-declarations merge-parameter-list**
method-body

func-dcltr ::=
  func-dcltr-1
| *pointer func-dcltr*

func-dcltr-1 ::=
  func-dcltr-2
| *func-decl* ( )
| *func-decl* [ ]

func-dcltr-2 ::=
  (*func-dcltr*)
| *identifier* (*func-formals-list*$_{opt}$)
| **string** ( )

func-formals-list ::=
  func-formal-declaration
| *func-formals-list* , *func-formal-declaration*

func-formal-declaration ::=
  type-specifier declarator
| *identifier*

para-declarations ::=
  para-decl+

para-decl ::=
  type-specifier declarator

merge-parameter-list ::=
  merge-parameter
| *merge-parameter-list merge-parameter*

merge-parameter ::=
  , *class-name* (*identifier-list*$_{opt}$)

method-body ::=
  decl-stat-list

decl-stat-list ::=

```
            decl-stat+
decl-stat ::=
            decl-stat-list declaration-list
        |   decl-stat-list stmt-list

stmt-list ::=
            statement-list ;
        |   send-stmt ;
        |   delayuntil-stmt ;
        |   respond-stmt ;

send-stmt ::=
            obj-expression . message
        |   send message to obj-expression

message ::=
            identifier ( argument-expression-list_opt )
        |   string ( )

delayuntil-stmt ::=
            delayuntil message
        |   delayuntil message from obj-expression

respond-stmt ::=
            respond message
        |   respond message to obj-expression

obj-expression ::=
            expression

declaration-specifiers ::=
            storage-class-specifier declaration-specifiers_opt
        |   type-specifier declaration-specifiers_opt

primary-expression ::=
            identifier
        |   constant
        |   string
        |   (expression)
        |   $sender
        |   $selector
        |   $self
```

| *$thread*

type-specifier ::=
`void`
| `char`
| `short`
| `int`
| `long`
| `float`
| `double`
| `signed`
| `unsigned`
| *enum-specifier*
| *struct-or-union-specifier*
| *typedef-name*
| **class-name**
| `object`

feature-name ::= identifier

class-name ::= identifier

object-name ::= identifier

2.3 The Translation

Two distinct data structures are used in MeldC runtime: the object structure and the frame structure. One object structure is allocated for one object, while one frame struct is allocated for one method (since this version of MeldC is a message based context switch, rather than the traditional stack based context switch.). Other C structures are also defined for local blocks. These definitions appear in the hfile which is generated in the second pass of the compilation process.

2.3.1 Object Structures

A unique C structure definition is generated for each class definition and for each object (instance of the class), an object structure is allocated. The

object structure contains the class' instance variable as well as some system variables (e.g. a lock for each object). The object structure's identifier will be in the form of
`$featurename$classname`.
For example:

```
feature MY_FEATURE
class foo ::=
      int x;
      char y;
methods:
      . . .
end class foo
```

translates to the object structure:

```
struct $MY_FEATURE$foo {OBJ_STRUCT_PREFIX
  int x;                               /* instance variable */
  char y;                               /* instance variable */
  struct _frame *$P;
  char *$S;
  struct _method *$init;
  struct _method *$$init;
  struct _method *$$term;
};
```

`OBJ_STRUCT_PREFIX`, defined in `meldc_user.h` is a macro which expands into a C structure containing system variables relevant to all objects.

The object structure is generated in pass 2 after parsing the class. The function `print_decl_structure()` does this.

2.3.2 Frame Structures

A unique C structure definition is generated for each method definition and for each each function/method invocation, a frame structure is allocated. The frame structure contains the method's input parameters (call by value), local variables, a return parameter (`$RET`) (return by reference) and some system data (used for context switching). These frames are used in runtime to implement MeldC threads, currently a message based design.

The frame structure's identifier will be in the form of
`$featurename$classname$methodname`.
For example: A method `hello()` in the class `foo` above

```
void hello(char *p)
{
    int a;
    printf("hello\n");
}
```

translates to the frame structure

```
struct $MY_FEATURE$foo$hello {FRAME_STRUCT_PREFIX
  char *p;                                  /* input parameter */
  int *$RET;                                /* return value pointer */
  struct _frame *$P;                        /* tmp frame pointer */
  char *$S;                                 /* tmp string pointer */
  int a;                                    /* local variable */
};
```

`FRAME_STRUCT_PREFIX`, defined in `meldc_user.h` is a macro which expands into a C structure containing system variables relevant to all method definitions.

Nested blocks definition are also represented by a C structure in the form of `$featurename$classname$methodname$int` where `int` is an integer starting from 0. This C structure will contained the local variables declared inside the block.

For example:

```
int hello()
{
    {
    int i;
    char *s;
    ...
    }
}
```

A C struct will be defined as:

```
struct $featurename$classname$hello$0 {
  int i;
  char *s;
  ...
}
```

The above naming convention is applied recursively for nested blocks definition. This local variables in nested blocks are accessed in a uniform manner. For example:

```
int hello(int x, int y)
{
  int z;

  {
    int i; char *s; int x;
    ...;
  }
  ...
  {
    int j; char *k; int y;
    ...;
  }

  {
    int i; char *s; int x;
    ...;
  }
}
```

The C struct for method `hello` (data part) will be

```
struct $featurename$classname$hello {
    int x;                                /* formal parameter x */
    int y;                                /* formal parameter y */
    int z;                                /* local variable z   */
    int *$RET;                          /* return value */
    union {
          struct $featurename$classname$hello$0  $0;
          struct $featurename$classname$hello$1  $1;
```

```
          struct $featurename$classname$hello$2  $2;
    } $$;
    ...
};
```

The compiler also generates temporary variables in the form of `$TMP{int}` to hold the return values of function calls in expressions. These temporary variables are generated in the C structure representing the local block/scope. If the method call is bind statically, the the temporary variable will be declared as the return type of the method. However, if the call cannot be bind until runtime (i.e. dynamic binding), by default, the temporary variable is declared to be of type `int *`, as a pointer to a location where the return value is actually stored; it acts as a pointer in this case. The memory is allocated in runtime.

For example, if we have the expression : `x = foo_obj.hello(1) + obj.hello(2);` where `foo_obj` is an instance of a known class (`foo`) and `obj` is a *generic object*. The first call of the method `hello()` can be bind statically and thus `$TMP0` will be declared as an `int`. The second call of `hello()` is a dynamic binding call, thus `$TMP1` is declared as an `int *`.

The frame structure as well as C structs representing local blocks is generated in pass 2 after parsing the body of the block. The function `print_decl_structure()` does this.

Chapter 3

Data Structures

3.1 Definition & Declaration Tables

The definition table, `defn_sp` stores information about classes, methods, parameters of methods, and user defined types. The declaration table, `decl_sp` stores variables and objects which are active during parsing. During parsing, the system often looks into these tables for information, *e.g.* for type checking of parameters. Both tables are implemented in a stack like fashion to enforce scoping. Whenever a new scope is entered, another level is *pushed* onto the definition/declaration stack; when the program exits a scope, a level is *popped*. A *frame* in each table is implemented by a header node, which has a `head` pointer and a `tail` pointer, which references the definitions/declarations in this frame. The next frame, the immediate scope enclosing the current scope, is accessed by the `next` pointer.

The functions `print_defn_tabs()` and `print_decl_tabs()` helps with debugging. A call of these two functions can be inserted anywhere in the compiler code and during execution, the contents of the table will be printed to screen.

`Note:` Since the definitions of the classes are loaded into the definition table by reading the .info file, if the type of a variable is a class, instead of storing `"class <classname>"` into the table, only the string `"<classname>"` is stored. This causes much grieve in type checking, and hacks have been placed in a few places to *temporarily* solve this problem.

The following are functions that interfaces with the tables; please refer to the documentation in the source code (`decl_tab.c`) for more details:

- add_class_decl
- add_defn
- add_feature_decl
- add_formals
- add_method_decl
- add_method_decl_decl
- append_defn
- enq_new_defn
- find_all_member_defns
- find_class_defn
- find_defn
- find_enum_type
- find_member_defn
- find_method_defn
- find_method_index
- find_struct_element
- find_str_msg_defn
- find_union_element
- if_class
- if_enum
- if_struct
- if_typedef
- if_union
- init_struct_level
- is_a_basic_type

- is_a_class_type
- is_a_func_decl
- is_a_func_init
- is_c_typename
- last_defn
- method_has_optional_args
- name_in_defn
- new_defn
- pop_defn_frame
- print_class_info
- print_class_info_ifile
- print_decl_structure
- print_decl_tabs
- print_defn_tabs
- print_instance_tab
- print_last_n_data_decls
- print_last_n_init_stmts
- print_last_n_objects
- print_struct_elem_tab
- print_struct_union_enum_typedef
- push_defn_frame

3.2 The Class Hierarchy Tree

The compiler uses a tree structure to store information about the class hierarchy, `class_hierarchy`. Each class is represented by a class node which has pointers to nodes representing its methods, and instance variables. Its `parent` pointer points to a list of class nodes which it immediately inherits, (or `merges`) from. Its `children` pointer points to a list of class nodes of which it is the immediate parent. For efficiency reason, all lists are implemented with a header node which has `head` and `tail` pointers. The class hierarchy tree is built at the beginning of the second pass in the compilation. It is virtually a read-only-access once it is built. The same tree is used regardless of the inheritance semantics (merging or overriding) specified by the program.

The function `traverse_all()` is useful for debugging. It prints all defined classes in the system showing their inheritance relationships with indentations.

`Note:` Information about methods, and parameters in the class hierarchy tree may not always be consistent with those in the definition tables. It would be nice if the these two data structures could somehow merge together.

The following functions interfaces with the class hierarchy tree; please refer to documentation in the source code (`build_classes.c`) for more details:

- add_child
- add_class_node
- add_instance_node
- add_method_node
- add_parent
- add_to_topological_list
- check_circular_inheritance
- check_conflict_inst_var
- check_duplicate_instance_var
- check_duplicate_instance_var_parent

- check_inode
- check_non_existent
- check_non_existent_parent
- clean_sync_with_field
- collect_inst_list
- delete_class_node
- duplicate
- find_ancester
- find_ClassNode
- find_InstanceNode
- find_instance_var_MERGE
- find_instance_var_OVERRIDE
- find_instance_var_type
- find_MethodNode
- find_override_Method
- get_parent_obj
- init_build_hierarchy
- make_CharListNodeHeader
- make_ClassNode
- make_ClassNodeHeader
- make_InstanceNode
- make_ListNodeHeader
- make_MergeParamList
- make_merge_arg
- make_MethodNode
- make_ParamList
- map_para

- merge_inherit_inst_var
- process_parents
- set_para_values
- sync_inherited_members
- topological_traverse
- traverse_all
- traverse_parent
- trim_inheritance_tree

Chapter 4

Classes

Classes is the equivalent of an abstract data type in object oriented programming. The definition of a MeldC class includes instance variables, active values declarations and the set of symbolic methods and string selectors. All instance variables are *private* and all methods/selectors are *public* in C++ terminology. Classes can inherit characteristics of other classes and this is discussed in chapter 6.

4.1 Meldtype versus Ctype Classes

The MeldC compiler supports two types of classes: the default type, `meldtype` and `ctype`. `ctype` classes are designed to achieve efficiency since methods in these classes would be invoked just like calling a C function, without having to queue up its frame and go through the MeldC scheduling process. However, problems arise when `ctype` classes is coupled with inheritance. And generally, it just causes lots of unclear semantics issues as well as implementation problems with other language features. However, the class DynamicBinding in runtime (`metaClass.m`) heavily relies and takes advantage of this design. As a last word, the compiler does not have strong support for ctype classes.

4.2 Loading

The compiler builds the interface definition of classes, before it starts parsing the feature. The class' interface definition includes its instance variable declarations, the method declarations, the methods' return type and their formal parameters, as well as the class' immediate parents (if inheriting). This is implemented by the function `build_classes()`. The function reads the `.info` file which contains these information and stores them into the definition table and the class hierarchy tree. The following describes the sequence of events when loading in the definition of a particular class.

`build_classes()` parses the `.info` file and scans for the marker `CLASS`; this begins the definition of a class. The function `add_class()` is a wrapper function which calls `add_class_decl()` to add the class definition to the definition table and `add_class_node()` to add the class to the class hierarchy tree. Similarly, `add_method()` calls `add_method_decl()` and `add_method_node()` respectively. However, the formal parameter declarations are only added to the definition table by `add_formals()`. The function `add_instance()` stores the instance variable declarations into the class hierarchy tree only. (As they are being parsed later, the declarations are added to the declaration table.)

4.3 Parsing

After parsing the class name declaration, some internal data structure is initialized; for processing active values. New scope is created both in the definition and the declaration table. Then as the instance variables are parsed, their declarations are added in the declaration table. After all instance variables are parsed, an `element table` for this class is generated in the cfile for linearization purpose in runtime. (see section 11.3 for details about support for linearization)

Next the methods definition and the active value declarations are parsed. (see chapter 8 for details about active values) Symbolic methods in MeldC are declared just like C functions in ANSI C. (We actually support both traditional C and ANSI C declarations, but would like to eliminate traditional C eventually.) The C function declaration corresponding to the method is generated to both the cfile and the hfile at this time; it is slightly more involved for `ctype` methods. For each method, three system variables: `$RET, $P, $S` are added to the frame structure.

The compiler continues to parse the `method_body`. At this point, C code for any initialization of local variables declared in the method is generated. For `meldtype` methods by `print_last_n_init_stmts()`, for `ctype` methods, by `print_last_n_data_decls()`. Code generation then continues as `decl_stat_list` is parsed (see chapter 9 for details about code generation at the statement level). After parsing the method, the frame structure corresponding to the method is generated in the cfile. Though the frame structures for `ctype` methods is *never* used, it is still generated in the hfile.

After parsing the class, the compiler generates two system methods in the cfile: `$featurename$classname$$init()` and `$featurename$classname$$term()` by `gen_initializer() & gen_terminator()` respectively. `$$init()` initialize any initialized instance variables, and this method is invoked in runtime whenever an object of this class is created. Then the object structure corresponding to this class is generated in the hfile. `$$term()` is an empty function currently.

4.4 Objects

Objects are instances of classes. The MeldC objects are implemented as pointers to an object structure, thus objects must be created explicitly. Not all objects are *equal* though. All objects of primitive types has storage space already. The MeldC language have the concepts *remote object* and *persistent object*, but they are completely transparent to the compiler.

4.4.1 Global Objects

Global objects are objects declared at the feature scope and are visible for the whole feature. The first pass keeps all global objects' information in an array `global_objects` to be retrieved when writing out *export* information to the `.info` file. In the second pass, the function `print_last_n_objects()` prints information about this object to the file `featurename.i` which is used by the MeldC linker.

4.4.2 Class Objects

Class objects are instances of the Metaclass. For each class, C code is generated to initialize the class object by the function `print_class_init()`. The initialization code for the class `foo` in feature `MY_FEATURE` follows:

```
class _class_init $MY_FEATURE$foo_class_init = {
  "foo",
  "MY_FEATURE",
  0,
  $MY_FEATURE$foo$$init,
  sizeof(struct $MY_FEATURE$foo),
  3,
  3,
  $MY_FEATURE$foo$sym_method_tab,
  0,
  $MY_FEATURE$foo$str_method_tab,
  $MY_FEATURE$foo$instance_var_tab,
  NULL,
  NULL,
  (char *) 0
};
```

This piece of initialization code is actually only used for initializing the Metaclass class object. Other class objects are initialized with input arguments when they are created. The function `print_class_info_ifile()` generates the C code to create all class objects in the feature and writes it to the file ifile. The C code generated corresponds to sending the `Create()` message to the Metaclass object with the relevant arguments (same as the fields in the `_class_init` structure).

Chapter 5

Imports and Exports

The MeldC programming enables code reuse through the its `imports/exports` construct. The compiler supports this feature by bringing in the class definitions into the program. Note that a single feature resides in one file.

5.1 Exporting

In the first pass, we simply record the exportlist from the MeldC source file into the end of the .info file by calling `handle_exports()` which calls `write_exports_file()`. It writes the classes, objects the program is exporting in the following format:

```
EXPORT\_INFO
feature, class, object, ...
...
END EXPORT\_INFO
```

5.2 Importing

`imports` is handled in both passes. In the first pass, the compiler simply writes out the import statement to the .info file as it is parsed. In the second pass of the compilation, `handle_imports()` which reads the import statement in the .info file calls `read_imports_file()` to bring in the objects the program is importing. Class definitions are imported by the function `implicit_import()` which builds a class at a time. `implicit_import()` recognizes which feature the class resides in and finds the corresponding info file. The info file usually resides in the current working directory. Otherwise, when compiling, the `-E` switch specified on the compile line tells the compiler the directories where the info files are located. The directory paths are stored in a global variable `export_path` and the function `find_export_file()` in `extern.c` searches through the directories whose names are in the `export_path`. Note that all this is done after the classes defined in the current feature is built into the data structures and C code has been generated for them in the cfile. *No* C code (method tables, parameter tables, etc) is generated for imported classes.

5.3 Im/Ex-porting Objects

In the first pass, all global object(*only* object) declarations are collected (in `global_objects` and `global_objects_decl`). When exporting the object, it checked whether the object has been declared, if not, an error is flagged. Also, if a name is both a classname *and* an objectname, the *object* is exported.

When importing the object, the declaration of the object is added into the declaration table, and it is declared as an extern variable in the hfile generated for the current feature.

The following functions supports the implementation of `imports/exports`: (file `extern.c`)

- break_out_exports
- break_out_imports
- break_out_methods
- break_out_params
- find_export_file
- implicit_import

- params_into_string
- read_extern_file
- read_extern_file0
- reset_imported_classes
- write_extern_file

Chapter 6

Inheritance

6.1 Introduction

MeldC, as an object oriented language, supports the notion of multiple inheritance. A MeldC class can inherit from multiple parent classes and possess the characteristic of its ancestor up in the hierarchy tree as well as defined specific characteristics of its own. The semantics of the inherited methods with a class follows *merge* semantics; methods inherited are all merged when invoked. Instance variables are all inherited if they do not conflict.

Another kind of semantics, *override* semantics is also supported. if two methods have the same name in the class hierarchy, the method lower in the hierarchy has precedence. The same applies to instance variables. A scope operator, `::` is provided to access methods and instance variables specifically. The programmer specifies which semantics to use when compiling the MeldC programme.

Inheritance is supported across features also. A class can inherit from a class which is defined in another feature.

If a circular dependency in the class hierarchy is detected, meldc will give an error message.

6.2 Syntax

```
CLASS classname MERGES parent_classes ::=
  instance_variables_declaration
METHODS:

  return_type methodname (formal_parameter_declarations)
      [merge_parameters_declarations]
  {
     ...
  }

END CLASS


parent_classes :
          classname
        | featurename[classname]
        | parent_classes classname
        | parent_classes featurename[classname]
        ;

merge_parameters_declarations:
          merge_param
        | merge_parameters_declarations merge_param
        ;

merge_param:
          , classname (merge_decl_list)
        ;

merge_decl_list :
          merge_decl
        | merge_decl_list , merge_decl
        ;

merge_decl:
          formal_parameter
        | instance_variable
        | global_variable
        ;
```

```
e.g.

class Circle merges Shape ::=
      int radius;
      int center;
methods:
      void draw(), Shape(radius, center)
      {
         /* code to draw a circle */
      }
end class
```

This is a class `Circle` whose immediate parent is the class `Shape`. An object of `Circle` has instance variables `radius` and `center`. When a circle object invoked its `draw` method, the method `draw` in `Shape` is invoked with the parameters `radius` and `center`.

6.3 Design

In this implementation, the object structure of a class contains pointers to the object structures of its parent classes.

```
e.g. in feature FF if class C inherits from class A and class B
(assume both class A & B is local to feature FF)

                +-------+           +-----+
                |       |           |     |
                |  A    |           |  B  |
                |       |           |     |
                +-------+            +-----+
                |   y   |           |  x  |
                +-------+           +-----+
                    ^     +------+     ^
                    |     |      |     |
                    |     |  C   |     |
          __$FF$A   |     +------+     | __$FF$B
                    +-----+--    |     |
                          +------+     |
```

```
|  ----+-----+
+------+
|  x   | (C's own instance var)
+------+
```

It is as if class C has 2 other instance variables, __FFA and __FFB, besides the ones defined in class C. The naming of these internal (to the programmer) instance variables follows the convention : __$<featurename>$<classname>, where <featurename> is the name of the feature and where <classname> is defined.

6.3.1 Initialization of an Object

When an object of class C is created, its internal instance variables is initialized accordingly with a call of Create() with the appropriate class by the runtime system.

6.3.2 Synchronization of Inherited Instance Variables

All instance variables inherited and defined by the current class are checked against each other, if they conflict, and error message will be displayed.

2 instance variables conflict if they have the same name but different types. e.g.

```
      class B
        int x;
      /
   class C
        char x;
```

However, 2 instance variables which have identical names and types do not conflict. They do need to synchronize with each other though. Only one copy of the duplicated instance variables is accessed, the first occurrence of that instance variable when traversing the up hierarchy tree in reverse topological order (essentially depth first search up its parent link). All the other duplicating instance variables will be synchronized to this copy. e.g.

```
            class B
              int x;                <--- this needs to be sync
            /
        class C
            | int x;                <--- this copy of x is accessed
            |
            |
          +-------+
          | c_obj |
          +-------+
```

There is a need to synchronize them because when an inherited method is invoked, it is invoked with a 'sub-object' (e.g. `c_obj->_$FF$B`) within the calling object (e.g. c_obj), we want the sub-object's instance variable to have the same value as the calling object's instance variables, otherwise the inherited method would be working with different values as the calling methods.

Thus code is generated in the class's $$init() to synchronize all these duplicate copies of inherited instance variables. e.g.
`((struct $FF$C *)_obj)->_$FF$B->x = ((struct $FF$C *)_obj)->x;`

This is implemented in `build_classes.c` by the functions:

- sync_inherited_members()
- merge_inherit_inst_var()
- collect_inst_list()
- check_conflict_inst_var()
- check_inode()

6.3.3 Invoking Inherited Methods - Merging

- Order of method invocation

  MeldC's inheritance scheme follows a *merge* semantics when a method of a class which is inheriting is invoked. Inherited methods are invoked in *reverse topological order*, all of a class's parent's methods are called before its own method is called. The order of its parent classes as it appears in the class declaration determines which parent's class's methods are invoked first.

The inherited methods are invoked *synchronously*. This is a policy issue. It seems more appropriate to invoke them synchronously because the programmer really treat his one invocation as one call, rather than a merge of several different calls. e.g.

```
    class A                                    class A
    class B merges A                           /     \
    class C merges A                      class B    class C
    class D merges B, C                        \     /
                                                class D
                                                  |
                                                  |
                                              +-------+
                                              | d_obj |
                                              +-------+
```

If all of them has a method `hello()`, then the call `d_obj.hello()` will invoke `A's hello(), B's hello(), C's hello()` then D's hello() respectively and synchronously. If class D also inherits from class A, `A's hello()` will *not* be invoked again.

- Invoking the inherited methods with

  The inherited methods are invoked with the appropriate *sub-objects* within the calling object. If we have the above example, the method invocation will be as follows:

```
       ((struct $FF$D)d_obj)->__$FF$B->__$FF$A.hello();
 sync -->
       ((struct $FF$D)d_obj)->__$FF$B.hello();
 sync -->
       ((struct $FF$D)d_obj)->__$FF$C.hello();
 sync -->
       ((struct $FF$D)d_obj).hello();
```

  Again, synchronization of inherited instance variables are needed between the invocations. This is implemented in `build_classes.c` by `merge_inherit_methods(), and topological_traverse()`.

- Parameter mapping of inherited methods merge_parameters_declarations of a method specifies the arguments to call an inherited method of a class's immediate parent class. Currently, the parent's class method (of the same name) can be invoked with the calling object's method's formal parameters, the calling object's instance variables (+any inherited ones), and any global variables. e.g.

```
  int w = 99;                                 /* global var */

class A ::=
  int z = 0;                                /* instance var */
  hello(int x, int y, int a, int b)

class B merges A ::=
  hello(int x, int y) , A(y, x, w, z)


b_obj.hello(1, 100) will invoke

   ((struct $FF$B *)b_obj)->__$FF$A.hello(100, 1, 99, 0);
   ((struct $FF$B *)b_obj.hello(1, 100);
```

However if no specification of how to call an inherited method is given, the inherited method is called with the arguments given to the calling method. meldc will give warnings if parameter types do not match or the number of arguments given is not what is expected. This is implemented in build_classes.c by get_para_values(), make_merge_arg(). With the above example:

```
class B merges A ::=
  hello(int x, int y)

 b_obj.hello(1, 100) will invoke

    ((struct $FF$B *)b_obj)->__$FF$A.hello(1, 100);
    ((struct $FF$B *)b_obj.hello(1, 100);
```

6.4 Override Semantics

Inheritance using override semantics behaves very much like that of C++. If methods or instance variables in the class hierarchy have the same name, the method or instance variable lower in the hierarchy takes precedence. However, in the case of multiple inheritance, conflicts arise when methods/instance variables having the same name are defined in the same level of the hierarchy.

The main data structure supporting this inheritance scheme is still the class hierarchy tree. Most of the functions supporting the merging scheme can also be used. Slight modifications is needed to relax the checking for conflicting instance variables and methods. For merge semantics, a conflict arises when anywhere in the hierarchy tree, two instance variables/methods have the same name; for override semantics, a conflict only arise when they are defined in classes at the same level of the hierarchy tree.

With the override scheme, methods are not merged anymore when invoked. Instead, after `topological_traverse()` is invoked, the system finds the last method returned, checks that there is no conflicting method, and generates C code to bind this method (if static binding). The relevant functions in `build_classes.c` are:

- `check_conflict_inst_var`
- `find_instance_var_OVERRIDE`
- `find_instance_var_type`
- `find_override_Method`

6.5 Scope Operator

The scope operator in MeldC provides a way to access a method or an instance variable hidden by inheritance with override semantics. With override semantics, since the method or instance variable lower in the hierarchy always takes precedence, the programmer cannot access another method or instance variable with the same name higher in the hierarchy; with the scope operator, this is possible. To invoke a method higher in the class hierarchy of an object: `<object>.<class>::<method>()`; to access an instance variable higher in the class hierarchy: `<class>::<instance var>`, where ¡class¿ is an ancestor of the class which the object belongs to in the former case, and of the current class in the latter case. e.g.

```
a_circle::Shape.draw();        /* invoke draw() in Shape */
```

```
str = Shape::name;        /* access name in Shape */
```

Code generation for invoking a message with scope operator requires some string mangling with the message, the object and the classname. See details in the file `codegen.c` in `proc_send_stmt()`. Accessing of a variable with the scope operator is supported by `find_instance_var_OVERRIDE()` in `build_classes.c`. It searches through the hierarchy and tries to find the instance variable in the class specified with the scope operator.

6.6 Runtime Support

6.6.1 Class Object and Object Creation

Since MeldC supports dynamic binding, the class hierarchy information must be preserved at runtime as well. In the second pass of the compilation, the compiler generates code to support this. For each class, C code is generated to create the class object at runtime and the class hierarchy information is embedded in this code. At runtime, the class objects will have pointers to class objects which are their immediate parents; thus the hierarchy is preserved. The relevant functions in the file `decl_tab.c` are:

- print_class_info_ifile()
- make_create_para()
- proc_send_stmt()

When an object is created, the `Create()` function in the meta-class recursively searches through the class hierarchy and creates the *sub-objects*. Note that though a class may appear more than once in the hierarchy, only one copy of the object is created, and this object is referenced through multiple pointers.

6.6.2 Dynamic Binding

As in static binding, a reverse topological sort is first performed on the class's hierarchy tree to find all the methods applicable. Then if *merge* semantics is specified, for all the methods found, a frame would be created and queued to be executed. Before and after its execution, two C functions are called to synchronize the multiple copies of instance variables of the object (multiple copies of instance variables exist because with merging semantics, the hierarchy tree can have instance variables having the same name and type). These ancestor's method frames are always executed synchronously, which is actually *not* the correct behaviour. The binding class's method frame, however, is executed synchronously or asynchronously accordingly. The relevant functions defined in the class `DynamicBinding` in `runtime/meldc/metaClass.m` are :

- internal_binding()
- binding()
- topological_sort()

6.7 Other Issues

6.7.1 Ctype Classes

The current implementation does not support inheritance of ctype classes. All code generated are `meldtype`. It would not be too difficult to support ctype inheritance though, we can check what type the class is and called `meldtype_gen()` or `ctype_gen()` as appropriate in `merge_inherit_methods()`. We should also check that a meldtype class does not inherit from a ctype class or vice versa. It does not seem to make any sense.

6.7.2 `init()` not merged

The invocation of `init()` of each class is not merged. Since the initialization, and creation of *sub-objects* (the representation of it's parent) is done here, it's essentially *merged*. The creation of the sub-objects will invoke the corresponding `init()` in the parent class.

6.7.3 `merge_decl`

The current implementation only allows the merge declaration of a method to be either a formal parameter, an instance variable, or a global variable.

It would be nice if we can allow arbitrary expressions as well. The difficulty is in evaluating the expression when invoking the inherited method. How hard would this be? However, we should also evaluate how useful this feature is. The implementation is involved and may not be worth it.

6.7.4 No Code Copying

This implementation of inheritance does no code copying. The previous design is forced to copy and rename inherited methods in its current class to make sure that the object structure that is invoking the inherited methods is compatible. This approach, however, has a serious problem with MeldC's **`imports/exports`** notion. If a class is inheriting from a class which is imported into the current feature, it is difficult, if not impossible to copy the parent class's methods.

In this implementation, the inherited method is invoked by the sub-objects (the representation of it's parents) that the object carries. Thus object structure is compatible when invoking an inherited method. However, there is much code devoted to synchronize, the multiple copies of duplicating inherited instance variables (if any).

6.7.5 Inheriting an External Class

A class can inherit from a parent class which is defined in another feature. The parent class will be silently *imported*. Or should the programmer be required to explicitly import the class before defining a class to inherit from it.

Chapter 7

Generic Objects

7.1 The usefulness of generic objects

Generic objects is designed and implemented by James Show.

In C, a programmer can write "a= b + c" as long as a, b, and c are integer or floating point variables. The operator "+" has different meanings depends on the operands it is applied (the operations to add real to real, int to int, or int to real are all different). Such operator is called *overloaded*, *polymorphic*, or *generic*. Memorizing symbols for separate operators for adding integers and adding floating point numbers is a not-so-productive effort for programmers since the concepts of adding integer and floating point numbers are so similar. As we can see in this case, a judicious use of overloading can contribute to the simplicity and usability of a programming language. On the contrary, excessive use of polymorphism can results in programs that are hard to understand because a name or operator might represent unexpected entities or operations.

Similar to polymorphic operators, generic objects are objects which might have different meanings. They have the same effect as virtual functions in C++. In Meld C, a class object can be declared with type qualifier *generic*. A generic object can be assigned to and acts as it is an object of its derived class (its children), and it is expected to be bounded dynamically at run time. As a result, a method being passed to a generic object might behave differently depends on which object receives the message at the time the message is sent. Both virtual functions and generic objects require dynamic binding which takes place at run time, they are costly. But similarly to polymorphic operators which increase a little bit of compile time, dynamic binding encourages data encapsulation and increases flexibility for the language. Without generic objects, strongly typed language such as Meld C is too rigid to support information hiding and data encapsulation adequately.

Let's say we have whole bunch of objects polygon:

```
/*****************************************************
 *
 *  Graph of the classes, with functions defined in
 *  each class in parenthesis (i= init(), a= area(),
 *  c= cost()).
 *
 *               polygon(i,a,c)
 *                /   \
 *               /     \
 *   (i,a) triangle     rectangle (i,a)
 *                        \
 *                         \
 *                        square (i,a)
 *
 *****************************************************/

extern int printf();
extern int exit();

feature Gons

interface:

implementation:
  meldC_main main_obj= meldC_main.Create();

class meldC_main ::=
methods:
void init()
{
  generic polygon P1= polygon.Create();
  triangle T1= triangle.Create();
  rectangle R1= rectangle.Create();
  square S1= square.Create();

  P1.cost(P1);
  P1= T1;
  P1.cost(P1);
  P1= R1;
  P1.cost(P1);
  P1= S1;
  P1.cost(P1);
```

```
  exit(0);
}
end class meldC_main

class polygon ::=
double width,height;
methods:
void init()
{ }

void cost(generic polygon p)
{  printf("the cost is %lf\n",p.area() * 12.0);  }

double area()
{  return(0.0);  }
end class polygon


class triangle merges polygon ::=
methods:
void init()
{  width= 2.0;  height= 3.0;  }

double area()
{  return(width*height/2);  }
end class triangle


class rectangle merges triangle ::=
methods:
void init()
{  width= 2.0;  height= 3.0; }

double area()
{  return(width*height);  }
end class rectangle


class square merges rectangle ::=
methods:
void init()
{  width= 2.0;  }

double area()
{  return(width*width);  }
end class square

end feature
```

Generic object serves as a placeholder holder for as-yet-undetermined child objects. In ANSI C, to keep program modular is considered a good software engineering practice. Since function cost() from previous example is used by all other classes, it is a good idea to make it more powerful to handle all the classes so everybody can share the function. For example,

```
void cost(int type)
{
  switch (type) {
    triangle: printf(``the cost is %lf\n'',width * height / 2);
              break;
    rectangle: printf(``the cost is %lf\n'',width * height);
              break;
    square: printf(``the cost is %lf\n'',width * width);
              break;
    default: printf(``the cost is %lf\n'',0.0);
              break;
  }
}
```

If we later on add *parallelogram* into our polygon group, we will have to modify procedure cost() to take care of the *parallelogram*. This makes the source code harder to maintain and verify. For a programmer to add more functionalities to the source code, he must know every detail of the rest of code unrelated to his work. This is counter-productive.

As we can see from the example above, dynamic binding provides facility for programmer to practice data encapsulation. Stanley B. Lippman summarized the advantages of virtual functions in his book "C++ Primer" which also apply to generic objects: "Users of the hierarchy and its applications who wish either to extend the hierarchy or customize the applications must have access to the source code. This make both distribution and support more expensive. Run-time type resolution encapsulates the implementation details of the derived hierarchy from user. Conditional test on the class type are no longer necessary. This simplify user code and make it less volatile. User code, no longer subject to change with each alteration to hierarchy, is simpler to program and maintain."

In original Meld C, each class of objects has its own function area().

To calculate the area of an object of class square S1, we simply use the statement "S1.area()". Similarly, to calculate the area of a triangular object T1, we use "T1.area()". Generic objects can make managing the objects easier. From the previous Meld C example, we can assign P1 of class polygon to be any one of its derived (child) objects. To find out the cost of a polygon object, programmer can simply assign the object to P1 and calls P1.cost(P1). The same procedure cost() will work with additional derived class as long as area() method is defined in that derived class. When more derived classes are added to class polygon, all the previous code are still reusable without any change. In case of ANSI C example, even though we designed the function cost() with modularity and data encapsulation in mind, we still need extensive alteration to the source to add extra functionality into the program. Omission of any such alteration of the code might results in mysterious run-time error.

In this implementation of the Meld C, the compiler will do the following services for generic objects.

- Key word "*generic*" can only apply to class objects. For example, "generic int;" is not allowed, and "generic polygon P1;" (polygon is a class name) is a legal declaration.
- The assignment of an object to a generic object; example: "P1= T1;". Polygon is the class of generic object P1. T1 must be be an object of class polygon or derived class of polygon.
- Generic object can be passed as an argument to a function which expects the parameter to be an object of one of its children class; example: "void cost(generic polygon p);".
- Meld C is a strong typed language but we also want it to be flexible

and not limiting the creative process of programmers, we decided to allow assignment of generic object of parent class to object of its children. For example, "g_b= g_a;", g_b is a generic object of class B, a derived class of class A, and g_a is a generic object of class A. g_a might be assigned to object of class A at runtime, then the statement would be illegal. But, g_a might be assigned to object of derived class of class B or B itself, then the statement would be legal. We simply print out a warning during compilation. The responsibility is rested on the programmer.

- Meld C does NOT check whether a message exists for the generic object, such as area() in P1.area(). Checking to see if area() is declared for P1 or at least one of P1's children is costly in compile time. Such an error will cause program to halt at runtime. The responsibility is again rested on the programmer.

- In the very first original implementation of generic object, there is no way to determine the return type of a message passed to generic object. In the primitive version, generic objects are implemented as global variables which can be assigned to any other object in the program. There is no way to predict what a generic object might be, even less the return type of the messages been sent. A reasonable guess would be assume all of them return int. Unfortunately, it not hard to come up with examples which will cause the program to behave unexpectedly. In this implementation of Meld C, generic object can only be assigned to objects of same class or of its derived classes. Because of this limitation, sometimes we have information to determine the return type of messages. If the message of a generic object is defined for the original class of the object, such as "P1.area();"

which returns *double*, it will assume the all “P1.area();” returns *double*. If the message is not defined in the original class of the generic object, the compiler assumes it returns *int* as the returned type, as in the primitive version. It also gives user a warning about the return type of such method is defaulted to int at compile time. Hopefully user will try to eliminate those warnings because they might cause program to behave unexpectedly. If the user defines different return type for the same message in different classes, the program will return unpredictable result because different type causes the result to be interpreted differently. It is hard if not impossible to make sure the return type of all the functions belong to derived classes with the same name to be the same. In Meld C, functions belong to different classes can have method with same name but returns different types. Only in the case that there is an generic object of that class declared, then it is desirable to make sure all of those functions returns the same type. It is hard and complicate to create a compiler that determines when all method name should return same type and produces correct code depends on whether there is an generic object of a class declared.

7.2 Implementation Decisions

In the original implementation of Meld C, generic object is implemented already. The problem with it is that the only generic objects that can be defined is too “generic” – any object can be assigned to it. This ruins the current Meld C’s claim that it is a strongly typed language. Any hacker can defeat the type checking mechanism in Meld C by declaring his object generic. A more restrictive implementation is needed.

Two new different implementations of the generic object were implemented. The compiler must handle declaration such as "extern generic unsigned triangle T1;". In the first implementation, to avoid erasing existing information in the symbol table, another field called "type_qualifier" was added to the symbol table. The most annoying part of the implementation is that the changes are everywhere whenever there is insertion into the symbol table and whenever there is type checking. Because this extra field wasn't there before, all the function declarations in the above categories need to be modified to handle the generic information. For simple assignment and parameter passing, this implementation works fine; for example, "g_a= b", *g_a* is a generic object of class A and *b* is a object of class B which inherits from class A. The compiler simply accesses the symbol table and figures out which object is generic and does the type checking accordingly. But there is a problem when the compiler encounters this expression: "*(g_a_p + 2)= b;", b is an object of class B which inherits from class A, and g_a_p is a pointer to generic object of class A. Meld C compiler must know that the expression results in a pointer which points to a generic object. Inside Meld C compiler, all expressions are represented by a structure called DUAL_STR, which is comprised of two parts: the expression itself (the name) and its type (type definition). From the example above, generic information is necessary for doing type-checking. The logical way to do so would be add another field at the end of DUAL_STR and creates a new structure called TRIO_STR – a simple solution to record the generic information. The problem with this implementation is DUAL_STR is used in hundreds of locations in the compiler code. Because DUAL_STR is defined as a macro inside the compiler, the overhead of TRIO_STR will be high at compile time. Thus the complete implementation was abandoned after a group discussion.

The second version simply inserts generic information into the type definition part of the DUAL_STR. Similarly, generic information is inserted in to the existing type definition field in the symbol table. A few rules were added to the YACC rules to handle such insertion. The troublesome part of this implementation is to make sure that the keyword *generic* is never output to the code prepared for the cc compiler (which will cause cc compiler error). This makes the modification much messier because the modification are sparsely located throughout the code generating functions. This version provides the facilities for doing complete strong type-checking for the problem encountered in the first implementation. To a casual user, the first version (if completed) and second version are the same. But from a implementor's point of view, the second version is much cheaper. Using the second implementation, Meld C compiler can take constant (const in C++ and ANSI C) declarations in addition to generic objects.

7.3 Implementation Details

- Meld C compiler is a two pass compiler, thus the first pass must pass generic information to the second pass.
- Most of the type checking is done at assignment and parameter passing.
- typecmp() in decl_tab.c, which allows binary operators to perform operations on regular and generic operands. It was used to both check assignment and parameter matching.

- proc_send_stmt() in codegen.c, which relax the type checking for sending a message to generic object. If not relaxed, compiler will complain such message is not defined.

- dynamic_send()in codegen.c, which decides the return type of the methods for generic object. It also transform all void declaration into int, to eliminate cc complaints

- output functions in codegen.c and decl_tab.c. Make sure that no generic information is output to the cc.

7.4 Known Bugs

- Dynamic checking is done by inserting extra code into the program to watch for impending errors. This will slows down the execution time of the running program. Comparing to static checking at compile time, dynamic checking is less efficient in terms of execution time and program space. The more serious problem of dynamics checking is that errors can lurk in a program until they are reached during execution. For large programs, some parts of the code are rarely executed. Even after the program is used for a long period of time, type errors might still occur.

- Compiler does not check a message passed to a generic object is declared at all, either in the class itself, or its children class.

- When a message is send to a generic object, the message might not be defined for that particular class. The compiler should be able to go up in the class hierarchy, find the method if the message has been inherit from object's parents and figures out the return type of the message. The current version simply assumes the return type is *int*. For example, cost() is not defined in class *square*. When compiler encounters statements "P1= S1; P1.cost();", it assumes the return type of P1.cost() to be *int*. The compiler should look up cost() in the class hierarchy and figure out the return type is *void*. We are just lucky that *void* in Meld C is converted to *int* anyway, and we did not use the value returned at all. We might have the problem that several parents have the same message we want, but they return different types. In such case, a conflict resolution policy is required. Using signature for function in C++ might solve the problem.

- The declaration for generic parameter to a method is not a complete name of a declaration (not desired "$feature$class", but "class" only). A more complete and thorough declaration is desired because there can be class with same name but in the different branches of hierarchy.

- Even though this implementation does support generic information for doing strong type checking, but type checking facilities itself is not complete. Even the generic pointer can be taken care of, but type checking cannot handle them yet.

7.5 Future Work

A thorough comparison should be done between generic objects in Meld C and virtual functions in C++. The author's hunch is that both features have equal power, but one of them might be more expressive for programmers which is yet to be determined. Currently Meld C does not support signature to distinguish functions. Such a language feature let C++ resolve conflicts which Meld C has problems. This might make comparison harder.

Chapter 8

Active Values

8.1 The Picture

Having active values in a programming language provides the user an easy construct to associate some actions to a variable which will be executes the variable's value is changed. In MELDC, active values is implemented by inserting comma expressions into the code where an active value is encountered. This report describes in detail the design and implementation issues of active values in MELDC. Active values is designed and implemented by Larry Katzman.

My project was to assist the Programming Systems group at Columbia University by implementing active values for the MELDC language. MELDC is an object oriented language similar to the C language with multiple inheritance, remote objects, and persistent objects (coming soon). In addition, it supports multiple threads (light-weight processes) and now *active values*.

Active values make programming easier for the user. The programmer can declare a variable as an active value, and spefify code in a rule associated to the variable. Whenever the value of an active variable gets assigned a value in an expression/statement, the MeldC compiler inserts the code which invoke that active variable's particular rules into the expression/statement. The code is inserted as a function call, and the return value of the active value assignment is the original intended value—it is as if the code was never inserted.

An example of the use of active values would be to maintain the relationship between fahrenheit and celsius variables. Whenever the celsius value changed, the code would be inserted to update fahrenheit and vice-versa.

The code required to implement active values in MeldC is dispersed throughout the files `parser.h, decl_tab.c, main.c, s1.l, s2.l, p1.y, p2.y,` and `active.c`, with the major part in the last two files.

8.2 I Declare Active

To give the programmer more flexibility, I incorporated two variations of active values. One variation in which rules will be triggered whenever an active value is assigned to a value, and a second variation in which the value must actually *change* before it can trigger its rules. The default variation for an active value is the kind where a rule will be triggered whenever the a value is assigned to the variable. The kind of active value which requires the variable to really change is declared with the added syntax of the '@' symbol.

```
class class_name ::=
    int a,b,c;
    char d;
methods:
    (a)-->{ /*meld code for this rule */ printf("HELLO NEW YORK\n"); }
    (b,d,a@)-->{printf("HELLO WORLD\n");}
    (d@,a,c)-->{printf("HELLO LONG ISLAND\n");}
    (c,c)-->{printf("I will be printed twice\n");}
    (ted)-->{printf("Ted\n");}
end class class_name
```

Active variables declared here must be declared instance variables. So in the above example, the compiler will generate a warning that the active value *ted* has not been declared. It should be noted that an active value can be declared *anywhere* bewteen **methods:** and **end class**. That means that active values can be declared before, after, and in between method declarations.

However, the following active value declaration is not supported :

```
    (*a)-->{rule} or
    (a.b->c)--> {rule}
```

They will both cause parse errors since they are both dereferenced pointers.

Active values can share rules, have more than one rule, and can cause a particular rule to be executed more than once. Thus given the code below :

```
    a=b=3;
```

```
d='g';
c=a=3;
```

The output will be the following :

```
HELLO WORLD
HELLO NEW YORK
HELLO WORLD
HELLO LONG ISLAND
HELLO WORLD
HELLO LONG ISLAND
HELLO NEW YORK
HELLO LONG ISLAND
HELLO LONG ISLAND
I will be printed twice
I will be printed twice
```

The rule associated with any active value declaration can be as complex as any MeldC method. That means that it can have local variables, blocks, loops, function calls, etc. However rules do not have any parameters, while methods indeed do. For example,

```
(a)-->{
 int i;
        num = 1;
        for(i=0;i<z; i++)
            num = num*a;
        printf("Num is now %d\n",num);
     }
```

The order of invocations of the rules associated with an active value is preserved; i.e. they are invoked in the order that they are declared.

In order to guarantee that the rules triggered by the active value is executed *immediately* and *atomically* after the active value is assigned or changed, the rules are generated as *ctype* code. This means that a rule call is a synchronous call, hence, it cannot be preempted by any threads.

8.3 The Art

Much of the ideas of design of active values is inspired from the implementation of active values in `C++` by Chris Wood, Adam Woodbury, and Arthur Cruz in the Programming Languages & Translator class in Fall 1990.

At every point where an active value is encountered, a comma expression which contains code to invoke the rules associated with that particular active value is inserted into the code. The active value is first updated as specified in the program, then the rules associated with it is invoked, finally, the value of the active value is returned as the last expression inside the comma expression. In the case of '@' active values, a conditional expression `(<if its value has changed> ? <invoke rules> : (void) 1)` is generated to ensure rules are not invoked if the value of the active variable remains the same. Rules are implemented as a functions, and thus invoked by making a function call.

8.3.1 What's Nice and Not

This approach preserves the structure of the original code as much as possible. The MELDC programmer will find it easier to debug, and the implementation is easier—than it would be to insert action statements into the code.

8.4 The Mechanics

8.4.1 The First Coming

The data structures for implementing active values is set up in the first pass of the compilation and is accessed and used for code generation in the second pass. For every MELDC source file *filename.m*, a corresponding file *filename.act* will generated to store information of active values. The file descriptor for this file is `active_fp`. In addition, it has two global variables `numactive` and `max_dim`.

In order to parse active value declarations, the non_terminals are added to the grammar in `p1.y` :

- active

- active_value_rule
- other_type
- literal_exp
- literal_exp1
- uorstr_type

For every rule that is parsed by the production `active_value_rule`, `numactive` is incremented and a marker *"$ruledone$"* is printed to the `.act` file. For every active value `LABEL` parsed, the production `active` prints it to the `.act` file with either a '@' or a 'n' next to it. The '@' symbol is for active values whose values *must change* before their associated rules are fired.

The production `other_type` is for active values which are arrays, structures, or unions. The information after the label with the '.' or the [ ] is printed in the `.act` file next to the correspong label. `uorstr_type` parses unions and structures as active values and `literal_exp` parses the information bewteen [ ].

When **feature** *feature_name* is parsed, the *feature_name* is printed to the `.act` file and `numactive` is incremented. Then when **class** *class_name* **::=** is parsed, (at the beginning of class declaration), the variable `class_size` is initialized to 0, `numactive` is incremented and the string *"class class_name"* is printed to the `.act` file. For every method or rule parsed, `class_size` is incremented. Finally, at the end of a class, both the string *"$class_size"* and the value of class_size are printed to the `.act` file and `numactive` is incremented.

At the productions `idcltr_list` and `dcltr_1`, tow global variables `dim` and `max_dim` are used to find the largest dimension of arrays which are used as instance variables. For example,

```
/*
 * sample.m
 */
feature f
interface:
implementation:
    c c_obj = c.Create();
class c ::=
    int a, b[20],d;
    struct s {
       int a;
       int b; } s1;
methods:
    (a,d@)-->  { printf("rule0\n"); }
    (b[3]@)--> { printf("rule1\n"); }
    (s1.a)-->  { printf("rule2\n"); }
    (s1,b)-->  { printf("rule3\n"); }
void init() {
    struct s s2;

    a = 1;
    b[0] = 1;
    b[3] = 1;
    d = 1;
    s1.a = 1;
    s2.a = 2;
    s2.b = 2;
    s1 = s2;
}
end class c
end feature
```

```
/*
 * sample.act
 */
f
class c
a n
d @
$ruledone$ finit
b[3] @
$ruledone$ finit
s1.a n
$ruledone$ finit
s1 n
b n
$ruledone$ finit
$class_size 5
```

8.4.2 Preparing for the Second Coming

The file descriptor `active_fp` (global variable) is assigned to read from the `.act` file and the routine `active_list()` is called; the source code for `active_list()` along with all of the other active value routines can be found in `active.c`. `active_list` reads from the `.act` file(s) and setup the data structure `active`. The variable `active` is of type pointer to `active_value_type` which is defined in `parser.h` to be `typedef char active_value_type[NSIZE];` Thus `active_value_type *active;`. Essentially, `active` is just a double dimension array, however by typedefing, I was able to do two additional things.

1. The memory pointed to by active can be dynamically allocated.

2. By incrementing `active` (`active++`), I am able to skip an entire array of `NSIZE` chars, not just a single char.

`active_list()` also stores the name, class, and feature for every '@' active value in a structure `really_change` and stores all of the class_sizes written in the `.act` file in `class_sizes`. With the above example, the two storage structures look like :

```
active->""((struct $f$c *)_obj->a" $f$c$rule_0() n"
        ""((struct $f$c *)_obj->d" $f$c$rule_0() @"
        ""((struct $f$c *)_obj->b[3]" $f$c$rule_1() @"
        ""((struct $f$c *)_obj->s1.a" $f$c$rule_2() n"
        ""((struct $f$c *)_obj->s1" $f$c$rule_3() n"
        ""((struct $f$c *)_obj->b" $f$c$rule_3() n
```

Each entry in the structure contains an active value name, the rule call that is associated with it, and a character to tell if it is '@' or not.

```
really_change->"f c d"
               "f c b"
```

To facilitate the ease of matching variables with active values, `active_list()` then calls `fix_active_list_part1()` to change the structure active so that the active variables that were declared with single dimension arrays or a field of a structures/unions now look like :

```
""((struct $f$c *)_obj->b[" $f$c$rule_1() @ 3"
""((struct $f$c *)_obj->s1." $f$c$rule_2() n a"
```

8.4.3 The Second Coming

`yy_2_parse()` is called after `active_list()`. The file `p2.y` has the same additional non_terminals `active, active_value_rule, other_type, literal_exp, literal_exp1, and uorstr_type` added in the grammar.

The global variables are initialized at the beginning of a class declaration :

- `rule_counter` set to 0
- `array_buf` to `NULL`
- `active_prefix` to

$$"((struct\$featurename\$classname*)\_obj \rightarrow "$$

- `feat_class` to "$featurename$classname"
- `class_counter` to 0

The length of `active_prefix` is stored in `prefix_len` at this time as well. The global variable `rules_exist` is set to the return value of the function `do_rules_exist()`. If rules are present in the class currently being parsed, a 1 is returned, otherwise 0. If there are rules declared for active values for this class, the circular dependency tree is created calling `create_depend_list()`. It first determines if there are any active values in the class currently being parsed, if there are, it allocates the appropriate memory to the pointer `depend_list` and stores the number of rule/method entries for the *current* class in the variable `class_size`.

Parsing Instance Variables

It is first checked if an instance variable is an array or structure. If so, `is_active()` is called to see if it is an active value. If the array or structure is an active value, `fix_active_list_part2` must be called to change the active value list to account for entire arrays, structures or unions. So that in the structure `active`, entire arrays or structure/unions now look like :

```
""((struct $f$c *)_obj->b[" $f$c$rule_1() @ $all"
""((struct $f$c *)_obj->s1." $f$c$rule_2() n $all"  instead of
""((struct $f$c *)_obj->b" $f$c$rule_1() @"
""((struct $f$c *)_obj->s1" $f$c$rule_2() n"
```

Remember, *specific* elements of arrays,structures,and unions were already handled by `fix_active_list_part1()`.

Next `is_really_active()` checks if the variable is an active value that was declared with a '@', the function returns 1 and a declaration for an _$prev$ copy of that instance variable is added. For example, when we have `(a@)` where `a` is an `int`, we declare a variable called `int a_$prev$;`. This shadow variable is declared with an initial expression of the actual instance varaible. So in fact the declaration looks like `int a_$prev$ = a;` If the '@' active value is an array, the shadow variable is declared but the initialization is handled later in the function `struct_init_code()` of `decl_tab.c`.

The variable `dependency` is initialized to 0 where it is declared. After all the instance variables have been parsed, `dependency` is set to 1. This adds all rule and method calls to the dependency tree for checking circular dependency among rules/methods in the class.

Parsing an Active Value Declaration

When parsing an active value declaration in the production `active_value_rule`, a function prototype is printed to the `..h` and `..c` files. By using the *featurename*, *classname*, and the global variable `rule_counter`, a unique rulename is generated and stored in the variable `rulename`. The variable `assgn_counter` is initialized to 0. This variable tells how many active value assignments have been made in the rule/method currently being parsed.

The routine `add_node_to_depend_list(rulename)` is then called. This routine copies the name of the rule being parsed into the appropriate entry of the `depend_list`. The entry in the `depend_list` is determined by the global variable `rule_counter`. `depend_list` is a pointer to type `rule_node` (found in `parser.h`). If there are three rules in this class and the first rule is being parsed, `depend_list` will look like this.

```
depend_list-> "rule_0" | -> NULL
              "" | -> NULL
              "" | -> NULL
```

A declaration frame is pushed and all of the `$_temp_$` variables determined by `max_dim` are added to the declaration table to become local variables in the rule. These temp variables are needed for using arrays that are active values.

In addition, the `tmp$` variable is added to the local variable declarations of the rule. This variable is the type of a unique structure generated in the `..h` file. It is used for the return types of every active value assignment in the rule.

The variable `mp_type` is set to `CTYPE` as discussed earlier while parsing the body of a rule with `r_compound_stmt`, like any method body would be.

Parsing a Method

Before parsing the method body, all of the `$_temp_$` variables for active value arrays are added to the method's local variables just like what was done for a rule. If `rules_exist`, the `tmp$` structure is also declared. In addition, if `rules_exist`, the `methodname` is added to the dependency list with `add_node_to_depend_list (methodname)`. The variable `assgn_counter` is initialized to 0.

At the end of a method or a rule, the structure

```
struct $featurename$classnametempn$ {
                type $1;
                type $2;
                type $3; }
```

is printed to the `..h` file. Each structure is unique within a class by the number `n`, the variable `class_counter`. Furthermore each structure is unique within a program by the *$featurename$classname* prefix. If `assgn_counter` is greater than 0, there have been active value assignements in the method/rule just parsed. Even if `assgn_counter` is 0, the structure must stiill be created since the compiler already added the `tmp$` variable to the declaration table. So a `char dummy;` field is created in the structure; since a structure must have atleast one field to compile using the C compiler. If a method is being parsed, the structure needs to be created only if there are rules. For a method, `class_counter` is then incremented and for a rule, both `rule_counter` and `class_counter` are incremented.

The End of a Class Declaration

At the end of a class declaration, the variable `dependency` is re-initialized to 0. The function `advance_to_next_class()` is then called. This function alters the variables really_change and active by advancing them to point past all entries of the class just parsed.

Next the code to initialize the `_$prev$` copies of '@' active values which are arrays is generated. The code is generated as a function for each array that needs to be initialized. `struct_init_code()` only generated the function calls to initialize the arrays, now the compiler actually generates the code for the functions that do the array initialization.

The End of a Feature Defintion

At the end of a feature, the function `free_active_info()` is called to free the variables used for active value implementation that were malloced during compilation. This is a precaution so that when many files are compiled together, the compiler will not run out of memory.

8.4.4 Becoming Active

There are four places where code needs to be generated for active values in a method body or rule body.

- at an instance variable declaration with an initial expression
- at the production of `cond_exp ASSGN exp`
- at the production of `COPERATOR prefix_exp`
- at the production of `post-fix_exp COPERATOR`

Instance Variable Declarations with Initial Expressions

At the end of a class declaration, the function `struct_init_code()` is called. It is here that all declarararations with initial expressions are handled. For every variable with an initial expression, the routine `is_active(exp)` is called. If `exp` is an active value, it will be located in the structure `active[is_active(exp)-1]`, and the index to its location is returned and the global variable `init_flag` is set to 1, otherwise it returns 0.

Next the function `rule_calls()` is called to generate the inserted active value rule function calls. For example, `is_active(exp)` returns `4`.

`rule_calls(exp,3)` will start looking at `active[3]` and return all the rule calls associated with that active value. It might return the string "rule_0(),rule_5(),rule_4()" for instance. If exp is an '@' active value, slightly different code is generated. For example, with `a@` where we have `int a=4;` `rule_calls()` might generate: `"a_$prev$=4,rule_0(),rule_5,rule_4"` When `rule_calls()` returns, `init_flag` is set back to 0. `struct_init_code()` then generates for loops to initialize any shadow copies of active value arrays.

cond_exp ASSGN exp

The productions `COPERATOR prefix_exp`, and `post-fix_exp COPERATOR` use very similar code but parses assignements like `++a` and `a++` respectively. The following discussion is on `cond_exp ASSGN exp`, but the algorithm is similar for all three productions.

This production parses the majority of assignments. Such as `=, +=, *=`, etc. So for `a=4;`, if `a` is an active value, active value rule calls must be inserted into the statement. The `is_active()` function call not only tells the compiler if the variable is active, but it also sets the global variables `uorstr` and `array`. These variables check to see if the expression in the assignment is an array, structure, or union.

```
if (array) x=abs(array-exp);    /* As in a[3] */
if (uorstr) y=abs(uorstr-exp);  /* As in a.c */
if ((x!=0) && (y!=0)) {
   if (x<y) uorstr=NULL;        /*As in a[3].c */
   else array=NULL; }           /*As in a.c[3] */
```

So either just `array` will have a value, just `uorstr` will have a value, or they both will be set to `NULL`.

Source code for this section can be found in the file `p2.y` under the production `cond_exp ASSGN exp`. Notice how, if the variable is an active value, `assgn_counter` is incremented. This variable counts the number of active value assignments in a particular method or rule. The type of the expression containing the active value is stored in the array `temp_type[assgn_counter]`. This type information is needed to construct the structure type of `tmp$` for every method and rule parsed.

So for `a=4;`, where `a` is active, and is the fifth active value assignment in this method/rule, we generate:

```
(((struct $ff$c$f2 *)_frame)->tmp$.$5=((struct $ff$c *)_obj)->a = 4,
 $ff$c$$rule_4(),
 ((struct $ff$c$f2 *)_frame)->tmp$.$5);
```

By having `tmp$` at the end of the comma expression, we make sure that the expression returns the same type it would have if there weren't any active values.

For example, with the statement `a=4;` where we have:

```
""((struct $ff$c *)_obj->a" $ff$c$rule_1() n"
""((struct $ff$c *)_obj->a" $ff$c$rule_3() @"
""((struct $ff$c *)_obj->a" $ff$c$rule_10() n"
```

and this is the fifth active value assignment, code generated would look like :

```
(((struct $ff$c$f2 *)_frame)->tmp$.$5=((struct $ff$c *)_obj)->a = 4,
  $ff$c$rule_1(),
  (a_$prev$!=a) ? (a_$prev$=a,$ff$c$rule_3(),$ff$c$rule_10)
                : ((void) 1,
   $ff$c$rule_10),
 ((struct $ff$c$f2 *)\_frame)->tmp$.$5);
```

8.4.5 Of the Same Kind

For any single dimensional array such as `int q[15];`, any active value declarations are legal; the whole array or specific elements within the array can be declared as active values. However, for multi-dimensional arrays like `char q[12][20][20];`, *only* these declarations are supported:

```
(q)-->   {rule}
(q@)-->  {rule}
```

Only the whole array can be declared active. Declarations of specific elements or dimensions of the array as active will cause a parse error in the first pass.

For `int q[15]` and declarations

```
(q)-->   {rule_0}
(q[2])-->{rule_1}
```

`q[i]=4;` will generate:

```
(((struct $ff$c$f2 *)_frame)->tmp$.$1=
```

```
  ((struct $ff$c *)_obj)->q[$_temp_$1=i] = 4,
  $ff$c$rule_0(),
  (2==$_temp_$) ? $ff$c$rule_1()
: (void) 1,
  ((struct $ff$c$f2 *)_frame)->tmp$.$1);
```

With the additional rules and in the following order:

```
(q)-->      {rule_0}
(q@)-->     {rule_1}
(q[2])-->   {rule_2}
(q[12]@)--> {rule_3}
```

`q[i]=4;` will generate:

```
(((struct $ff$c$f2 *)_frame)->tmp$.$1=
 ((struct $ff$c *)_obj)->q[$_temp_$1=i] = 4,
 $ff$c$rule_0(),
 (memcmp(&q_$prev$[$_temp_$1],&q[$_temp_$1], sizeof(q[$_temp_$1]))) ?
   (q_$prev$[$_temp_$1]=q[$_temp_$1],
   $ff$c$rule_1(),
     (2==$_temp_$) ? $ff$c$rule_2()
   : (void) 1,
     (12==$_temp_$) ? $ff$c$rule_3())
                    : ((void) 1,
   (2==$_temp_$) ? $ff$c$rule_2()
  : (void) 1));
```

The importance of `$_temp_$` becomes evident with a statement like: `q[x++]=4` or `q[foo(1)][foo(1)] = 4;` Since the indices to the array has to be referenced more than once in the generated code for various purposes, they are stored in the `$_temp_$` to avoid the side effects of referencing them more than it's expected by the programmer.

For any '@' active values, `rule_calls()` it calls `question_marks()` which generates code for a decision tree to determine at runtime whether the rules for '@' active variables should be invoked or not.

8.4.6 Of Various Kind

An instance variable of type structure or union can be declared active. However, to stress again, any expression that contains a "-¿" or a "*" can *never* trigger a rule. They are instances of dereferenced pointers and it would be too difficult to trace all links to a memory space and consider all such links as being active. Thus, for an active value `y`, `*y=3` will not trigger any rules; similarly for an active value `s`, s.b-¿e=4 will not trigger any rules.

```
(a.b.c)-->    rule_0
(a@)-->       rule_1
(a.b.c.e@)--> rule_2
(a)-->        rule_3
```

These are all legal active value declarations. Structure assignments can generate simple code, like in:

```
(a.b.c)--> {rule_0}
(a)--> {rule_1}
(a.e)--> {rule_2}
a..b = x;
```

`rule_calls()` will generate the string `"rule_0(),rule_1()"`.

However, when we find '@' active variables which are of structure type, things get very ugly. For structures and unions, `rule_calls()` uses the variables `two_way` and `single_comp` to determine how many comparisons need to be generated. `single_comp` remains 1 unless it finds an '@' structure that matches and is *larger* (a whole structure is *larger* than one of its fields) than the `exp`. For example, where `a.b.c` is declared active and `a.b` is assigned to, `single_comp` is set to 0. Whenever `single_comp` or `two_way` are true, we only need to make one comparison.

With the above active value declarations and `a.b.c.e=4;`, only one comparsion is needed since `two_way` is 0 and `single_comp` is 1. And `rule_calls()` generates `"rule_0(),"` and then calls `question_marks()` which generates :

```
"(memcmp(&a_$prev,&a,sizeof(a)) ? (a_$prev$=a,rule_1(),rule_2(),rule_3()) :
 ((void) 1,rule_3))"
```

`rule_calls()` then returns the concatenation of the two strings.

For a statement like `a=y;` both `two_way` and `single_comp` are 0, so the compiler generates a string which is a 2**2 way decision tree. `question_marks()` calls `question_for_struct()` to generate this string. The power which 2 is raised to is determined by the number of "@" active values; in this particular case, `rule_1` and `rule_2`. `rule_0` and `rule_3` must be called, but for the other two rules, `rule_1` and `rule_2`, there are 2**2 possibilities; namely `r1,r2, r1, r2, (void) 1`.

With the code below, we need an 8-way test:

```
(s3@)-->{rule_1}                /*
(s3.b.a)-->{rule_2}              * active value declarations
(s3.b.a)-->{rule_3}              */
(s3.b.a@)-->{rule_4}
(s3.b.a@)-->{rule_5}
```

```
   s3.b=s2;                        /* assignment */

(((struct $f$c$init *)_frame)->tmp$.$2=((struct $f$c *)_obj)->s3.b =
   ((struct $f$c *)_obj)->s2,
 (memcmp(&((struct $f$c *)_obj)->s3_$prev$.b.a,
         &((struct $f$c *)_obj)->s3.b.a,
         sizeof(((struct $f$c *)_obj)->s3.b.a))) ?
   (memcmp(&((struct $f$c *)_obj)->s3_$prev$.b.a,
           &((struct $f$c *)_obj)->s3.b.a,
           sizeof(((struct $f$c *)_obj)->s3.b.a))) ?
      (memcmp(&((struct $f$c *)_obj)->s3_$prev$,
              &((struct $f$c *)_obj)->s3,
              sizeof(((struct $f$c *)_obj)->s3))) ?
          (((struct $f$c *)_obj)->s3_$prev$=((struct $f$c *)_obj)->s3,
           $f$c$$rule_1(),$f$c$$rule_2(),$f$c$$rule_3(),$f$c$$rule_4(),
           $f$c$$rule_5())
        : (((struct $f$c *)_obj)->s3_$prev$=((struct $f$c *)_obj)->s3,
            $f$c$$rule_2(),$f$c$$rule_3(),$f$c$$rule_4(),$f$c$$rule_5())
      : (memcmp(&((struct $f$c *)_obj)->s3_$prev$,
             &((struct $f$c *)_obj)->s3,
             sizeof(((struct $f$c *)_obj)->s3))) ?
          (((struct $f$c *)_obj)->s3_$prev$=((struct $f$c *)_obj)->s3,
           $f$c$$rule_1(),$f$c$$rule_2(),$f$c$$rule_3(),$f$c$$rule_5())
          : (((struct $f$c *)_obj)->s3_$prev$=((struct $f$c *)_obj)->s3,
              $f$c$$rule_2(),$f$c$$rule_3(),$f$c$$rule_5())
  : (memcmp(&((struct $f$c *)_obj)->s3_$prev$.b.a,
             &((struct $f$c *)_obj)->s3.b.a,
```

```
             sizeof(((struct $f$c *)_obj)->s3.b.a))) ?
      (memcmp(&((struct $f$c *)_obj)->s3_$prev$,
              &((struct $f$c *)_obj)->s3,
              sizeof(((struct $f$c *)_obj)->s3))) ?
          (((struct $f$c *)_obj)->s3_$prev$=((struct $f$c *)_obj)->s3,
            $f$c$$rule_1(),$f$c$$rule_2(),$f$c$$rule_3(),$f$c$$rule_4())
          : (((struct $f$c *)_obj)->s3_$prev$=((struct $f$c *)_obj)->s3,
             $f$c$$rule_2(),$f$c$$rule_3(),$f$c$$rule_4())
      : (memcmp(&((struct $f$c *)_obj)->s3_$prev$,
                   &((struct $f$c *)_obj)->s3,
                   sizeof(((struct $f$c *)_obj)->s3))) ?
          (((struct $f$c *)_obj)->s3_$prev$=((struct $f$c *)_obj)->s3,
            $f$c$$rule_1(),$f$c$$rule_2(),$f$c$$rule_3())
          :  ((void) 1,
              $f$c$$rule_2(),$f$c$$rule_3()),
((struct $f$c$init *)_frame)->tmp$.$2);
```

`question_for_struct()` calls two recursive functions to help it generate the correct n-way decision strings. Those two functions are `gen_code_seq()` and `gen_rule_seq()`. For an 8-way decision string,

```
From gen_code_seq(3,p):
p->
2
?                   An 'r' means a set of rules calls generated by
r                   gen_rule_seq(3,p).
:                   The 5th 'r' means the 5th string generated by
r                   gen_rule_seq.
```

```
:
0                   Each number represents the active value that needs
?                   to be memcmp'ed.  The number is an index into the
r                   variable stack.  Stack was defined in rule_calls and
:                   modified in question_for_struct to remove the non_'@'
r                   active values by the function call remove_non_at_avs.
:
1                   The stack for the previous example looks similar to this:
?
0                   index 2  s3. rule_5 @ b.a
?                   index 1  s3. rule_4 @ b.a
r                   index 0  s3  rule_1 @ $all
:
r
:
0
?
r
":    "
"r    "

From gen_rule_seq(3,p):

p->
"0 1 2 -1"          The numbers are indexes into stack to tell
"1 2 -1"            question_for_struct which rules need to be called.  A -1
"0 2 -1"            marks the end of a rule set.
"2 -1"
```

```
"0 1 -1"

"1 -1"

"0 -1"

"-1"
```

8.4.7 Round like a Circle

While parsing a method or rule body, the compiler may find function calls or rule calls. Thus at the production `primary_exp '.' message` and the production `send_statement`, the methods are added to the dependency tree if there are active values declared in the class. Likewise, when a rule call is found, it is also added to the dependency tree. The function call `add_to_depend_list (method/rulename)` is made to add the method/rule to the dependency tree. So after several methods/rules have been parsed, the `depend_list` may look like this:

```
depend_list-> "init"   | -> NULL
              "rule_0" | -> "rule_4" -> NULL
              "rule_4" | -> "init" -> "rule_8" -> NULL
              "rule_8" | -> "rule_0" -> NULL
```

The circular dependency check for the rules of the class just parsed is conducted at the end of a class declaration. The circular depependency check is done by the function call `check_for_dependencies()`. This function firsts calls `link_depend_list()` to link all the children of a child, to that particular child. For example:

```
depend_list-> "init"   | -> NULL
```

```
              "rule_0" | -> "rule_4" -> NULL
              "rule_4" | -> "init" -> "rule_8" -> NULL
              "rule_8" | -> "rule_0" -> NULL
becomes:
                               ---------------
                               |             |
                               v             |
              "rule_8" | -> "rule_0" -> NULL  |
                               |             |
                               v             |
              "rule_0" | -> "rule_4" -> NULL /
                               |           /
                               v          /
              "rule_4" | -> "init" -> "rule_8" -> NULL
                               |
                               v
              "init"   | -> NULL
```

After the linking is done, the compiler then checks for circular dependencies that begin at each rule/method. It calls `circ_depend()` to do the recursive search. The circular dependency warning is printed only if the circular dependency ends with a rule. The messages are just warnings because the compiler cannot be sure that the rules and methods are going to cause a circular dependency. That can only be determined for certain during runtime.

8.5 Pàs De Probléme

The depth of active value implementation in MeldC is fairly thorough with the main limitations being:

1. The scope of active values are limited to instance variables.
2. Dereferenced pointers of anykind as active values are not supported.
3. Specifc elements of multi-dimensional array as active values are not supported
4. Active values are not active when they are passed as parameters to functions.

Other problems are discussed in the following sections.

8.5.1 Initialization of Structures

I found that MeldC has a few problems with it that have nothing to do with active values. Namely, the compiler will not allow an array or structure generation with an initial expression. So `int q[10] = {1,2,3,4,5,6,7,8,9,10};` will not compile.

8.5.2 typdef

In addition, typedefs may cause some problems for active values. Namely, for a typedef like:

```
typedef int int_type;
int_type a;
(a)-->{rule}
```

There is no problem. But if the typedef was: `typedef int q[10] array_type`
or

```
typedef struct {
        int a;
        int b; } struct_type;
```

With

```
struct_type a;
array_type b;
(a)-->{rule}
(b)-->{rule}
```

These two active values will never trigger their rules because the compiler does not treat them as arrays or as structures. So when the instance variables are being parsed, inside the active_value_list the variables "a" and "b" will never be changed to "a." and "b[".

8.5.3 Active Values with Inheritance

It should be noted that my work with active values in no way touched upon the implementation of inheritance. So we can assume that active values are not inherited but that when a child class calls a method inherited from a class which has active values, the active values of that parent class will trigger their rules.

8.5.4 Method Calls in Rules

Another major problem is that the current implementation does *not* support method calls of *meld* type methods inside rules. This is due to the fact that rules are implemented as ctype in an attempt to guarantee atomicity. However, being so, whenever there is a context switch, stack variables will be displaced, and therefore corrupted. This is not particular to implementation of rules of active values, but is a fundamental problem of *ctype* code.

Moreover, having the body of a rule as a *ctype* function does not gurantee true atmocity either. But currently, there is no constructs in MELDC which could achieve this purpose in its multi-threads environment.

Chapter 9

Code Generation

The MeldC compiler does *not* keep a parse tree and generates code off from the parse tree. Instead, it generates code at a statement level. Expressions are stored and concatenated in a list and the corresponding C code is generated at each statement.

9.1 Expressions

An expression in MeldC is stored in a *dual string* which is a linked list of two items: the first item the corresponding expression in C, the second the type of the expression in postfix notation. Support functions are in the file `list.c` and the following macros and functions are used:

- append
- char_to_myunion

- freelist
- last
- listlen
- list_find_str
- list_to_myunion
- newlist
- printlist
- _UCONV_
- TALLOCATE
- T_DUAL_STR
- DUAL_STR
- DUAL_STR_DUP
- FIRST
- SECOND
- FREE_DUAL_STR
- compress_dual_str_list

9.1.1 Variable Access

A major aspect in generating C code is the name mangling of an identifier expression. MeldC has the following basic expressions:

- $self - translates to _obj
- $sender - translates to _frame->$.sender
- $thread - translates to _frame->$.threadid
- $option - translates to $option

Variables, with the exception of global(feature scope) variables and class variables(static instance variables) follow a standard convention when translated into C code. The list below itemizes each category of variable access. Note _obj and _frame are the global pointers to the current object and frame data structure. In order to distinguish different types of structs a C cast must appear prior to a variable access.

- Global objects translates to
  $<featurename>$<objectname>
- Class variables translates to
  $<featurename>$<classname>$<variable>
- An instance variable x translates to
  ((struct $<featurename>$<classname> *) _obj)->x
- A formal parameter x translates to
  ((struct $<featurename>$<classname>$<methodname> *) _frame)->x
- A top-level local variable x translates to
  ((struct $<featurename>$<classname>$<methodname> *) _frame)->x
- A second-level local variable x translates to
  ((struct $<featurename>$<classname>$<methodname> *) _frame)->$$.$0.x

- Assigning the return expression, `return exp;` translates to

```
((struct $<featurename>$<classname>$<methodname> *) _frame)->$RET=exp;
return;
```

9.2 Message Passing

Message passing in MeldC invokes the execution of a method on an object. At compile time, static binding is performed and strict parameter type checking is enforced. If a method is successfully bound, a C expression which will create a frame for the method and fill in the parameters is generated.

Messages can be passed *synchronously* or *asynchronously*. Code generation also generates a flag which the runtime system recognizes to indicate this.

9.2.1 Symbolic Message

A symbolic message is in the form `object.method(arg1, ...)` if passed synchronously, or `send method() to object` asynchronously. As such an expression is parsed, the function `proc_send_stmt()` is called to generate the corresponding C code. If `object`'s type is known, i.e. it's *not* a *generic object*, the function `static_send()` will be called for static binding; otherwise, `dynamic_send()` is called.

If the message correspond to a method defined in the class which the object belongs to, C code 1) to create a frame of the method and 2) to fill in the formal parameters in the frame is generated. A temporary variable is also defined in the current frame structure to hold the return value of the method. Note that for messages which invoke a method defined in a `ctype` class, the C expression generated looks like a normal C function call, without setting up frames, and parameters are passed normally. An example of code generated by static binding of a symbolic message: `$self.spacing(total_space)`

```
  (NEW\_FRAME((((struct $Producer\_and\_Consumer$ProducerConsumer$init *)\_frame)->$P),
 sizeof(struct $Producer\_and\_Consumer$ProducerConsumer$spacing),
 \_obj,
 ((struct $Producer\_and\_Consumer$ProducerConsumer *)\_obj)->$spacing,
 \_frame,
 &($Producer\_and\_Consumer$ProducerConsumer$sym\_method\_tab[3]),
 SYNC),
    (((struct $Producer\_and\_Consumer$ProducerConsumer$spacing *) (((struct $Producer\_and\_Consumer$ProducerConsumer$init *)\_frame)->$P))->x) = ((struct $Producer\_and\_Consumer$ProducerConsumer *)\_obj)->total\_space,
    ((struct $Producer\_and\_Consumer$ProducerConsumer$spacing *) (((struct $Producer\_and\_Consumer$ProducerConsumer$init *)\_frame)->$P))->$RET = &(((struct $Producer\_and\_Consumer$ProducerConsumer$init *)\_frame)->$TMP0),
    (((struct $Producer\_and\_Consumer$ProducerConsumer$init *)\_frame)->$P)->$.sender = \_obj,
    ENQUEUE\_FRAME((((struct $Producer\_and\_Consumer$ProducerConsumer$init *)\_frame)->$P)),
    YIELD
  ,
    ((struct $Producer\_and\_Consumer$ProducerConsumer$init *)\_frame)->$TMP0)
```

9.2.2 String Message

A string message is in the form of `object."<str_msg>"` if passed synchronously, or `send "<str_msg>" to object` asynchronously, where `str_msg` is a *regular expression*. A string message is always bind dynamically at runtime. At compile time, C code is generated to invoke dynamic binding with information about the message and the parameters passed. The C function corresponding to a string method always has *one* formal parameter

`$selector` which is a pointer to the space which holds the input string message. At the moment, string message does *not* support user defined parameters.

9.3 Message Passing Structure

To discuss dynamic binding, it is essential to discuss an internal message structure which is used to hold information for dynamic binding as well as optional parameters. Each frame contains a message structure which stores the name of the message, its return type, whether it is passed synchronously or asynchronously, the number of arguments it is passed, the size of the whole structure and a block of memory which stores the value and the type of the arguments. A more detailed description is given in the next chapter, in the discussion of optional parameters.

If at compile time, the message cannot be bind, as in the case of generic objects, a C expression is generated to pack the information about the message into the frame's message structure, and to invoke dynamic binding. At runtime, the function which performs dynamic binding knows to look into this structure and tries to bind the message. An example of C code for dynamic binding (`an_obj.draw()`) follows:

```
(
((struct $inheritance$test$init *) _frame)->$.tmp_msg = NEW_MSG(sizeof(struct _msg)),
((struct $inheritance$test$init *) _frame)->$.tmp_msg->_type = 0,
((struct $inheritance$test$init *) _frame)->$.tmp_msg->_size = sizeof(struct _msg),
((struct $inheritance$test$init *) _frame)->$.tmp_msg->_msg_type = SYM_MSG,
((struct $inheritance$test$init *) _frame)->$.tmp_msg->_msg_name = "draw",
```

```
((struct $inheritance$test$init *) _frame)->$.tmp_msg->_return_type = &($inheritance$_types_tab[0]),
((struct $inheritance$test$init *) _frame)->$.tmp_msg->_return_addr = 0,
((struct $inheritance$test$init *) _frame)->$.tmp_msg->_num_arg = 0,

BINDING((((struct $inheritance$test$init *)_frame)->$P),
((struct $inheritance$test$init *)_frame)->an_obj,
&(((struct $inheritance$test$init *)_frame)->$TMP0),
&($inheritance$_types_tab[12]),
SYM_MSG,
SYNC_MSG),
(((struct $inheritance$test$init *)_frame)->$P)->$.sender = _obj,
ENQUEUE_FRAME((((struct $inheritance$test$init *)_frame)->$P)),
YIELD
,
((struct $inheritance$test$init *)_frame)->$TMP0 ? (char *)*(int *)((struct $inheritance$test$init *)_frame)->$TMP0: 0)
```

Another example of C code for dynamic binding `send "str_draw" to an_obj;`:

```
(
((struct $inheritance$test$init *) _frame)->$.tmp_msg = NEW_MSG(sizeof(struct _msg)),
((struct $inheritance$test$init *) _frame)->$.tmp_msg->_type = 0,
((struct $inheritance$test$init *) _frame)->$.tmp_msg->_size = sizeof(struct _msg),
((struct $inheritance$test$init *) _frame)->$.tmp_msg->_msg_type = STR_MSG,
((struct $inheritance$test$init *) _frame)->$.tmp_msg->_msg_name = "str_draw",
((struct $inheritance$test$init *) _frame)->$.tmp_msg->_return_type = &($inheritance$_types_tab[0]),
((struct $inheritance$test$init *) _frame)->$.tmp_msg->_return_addr = 0,
((struct $inheritance$test$init *) _frame)->$.tmp_msg->_num_arg = 0,
```

```
  BINDING((((struct $inheritance$test$init *)_frame)->$P),
  ((struct $inheritance$test$init *)_frame)->an_obj,
  &(((struct $inheritance$test$init *)_frame)->$TMP1),
  &($inheritance$_types_tab[12]),
  STR_MSG,
  ASYNC_MSG),
  (((struct $inheritance$test$init *)_frame)->$P)->$.sender = _obj,
  ENQUEUE_FRAME((((struct $inheritance$test$init *)_frame)->$P))
)
```

9.4 Initializer & Terminator

For each class two functions `$$init()` and `$$temr()` is implicitly defined by the compiler after parsing the class. The initialization function initializes any instance variables that needs to be initialized. The terminator function is currently an empty function. It is intended when an object of the class is destroyed, this function would be called to perform some clean up tasks.

9.5 List of Functions

These functions in the file `codegen.c` supports code generation for message passing:

- compute_size
- count_args

- ctype_gen
- dynamic_send
- fill_args
- generate_mesg_structure
- gen_initializer
- gen_merge_sync_code
- gen_static_parameter
- gen_terminator
- meldtype_gen
- merge_sync_code
- pack_para_types_ptrs
- proc_send_stmt
- static_send

Chapter 10

Optional Parameters

10.1 Introduction

The MeldC language supports the use of optional parameters of methods. Programmers can declare a MeldC method with optional parameters and reference the optional parameters using macros. This is a very useful feature in a programming language which allows programmer more flexibility by making modules more compatible and maintainable; little changes is required if suddenly a method needs an extra parameter.

10.2 Syntax

10.2.1 Declaring a method with optional parameter

```
return_type method_name(formal_parameter_declarations, optional)
{
  ...
}
```

The keyword `optional` appearing after (always) the required formal_parameter_declarations says that this method have optional parameters.

```
        int foo(int x, optional)        /* x is required
        {                                 * & other optional parameters
          ...                                 */
        }
        int bar(optional)                /* no required parameters
        {                                  * just optional parameters
          ...                                 */
        }
```

10.2.2 Referencing a particular optional parameter

When the programmer references a a particular optional parameter passed into the method, it is assumed that the programmer knows the type of the optional parameter he is expecting, and also the position of the optional parameter, whether it's the 1st or 2nd or the nth optional parameter.

A pointer variable of that type has to be declared first, and the macro `ARG_ADDR(i)`, where i is the position (range from 0-the number of optional parameters passed in), can be used to reference the address of the desired optional parameter in the following fashion :

```
parameter_type *ptr;
ptr = ARG_ADDR(i);
```

And then by dereferencing the pointer variable, the programmer can access the value of the optional parameter. For example, assume that the function foo() has 1 optional parameter and it's of type int.

```
int foo(optional)
{
   int *opt_ptr;
   int x;

   opt_ptr = ARG_ADDR(0);
   x = *opt_ptr;
}
```

10.2.3 Referencing All Optional Parameters Passed in as a Whole

Sometimes it may be desirable to reference all the optional parameters passed in as a whole and pass into another method as an argument. This kind of reference can be done by $option anywhere in the method.

This illustrates the flexibility that optional parameters allow. The method foo() does not have any knowledge about what parameters it gets, but just directly pass them down to the method bar() which will eventually decode them. It the parameter declaration of method bar() need to change, the parameter declaration of method foo() will not be affected.

```
e.g.         obj.method($option);
        send method($option) to obj;
        obj.method(1, "hello", $option);

        int foo(optional)
        {
           $self.bar(1, "hello", $option);
        }
```

10.3 Design and Implementation

A structure (`struct _msg`) in the frame structure is used to pass optional parameters. (Note that this structure is also used to pass parameters for dynamic binding.) C code is generated to allocate the structure at runtime, and copy the values of the optional parameters into the structure.

10.3.1 The Message Structure

Two ptrs to the structure is declared as a field in the frame structure to implement optional parameters passing: `_frame->$.msg & _frame->$.tmp_msg` Both pointers are of type `struct _msg`.

```
struct _msg {
   int _type;           /* 0 for internal, 1 for external */
   int _size;           /* size of the whole structure allocated */
   int _msg_type;       /* SYM\_MSG or STR\_MSG */
   char *_msg_name;     /* name of method calling */
   struct _elem_tab *\_return\_type; /* return type of method calling */
   char *_return_addr;  /* addr of return value */
   int _num_arg;        /* number of args contained in structure */
   struct _arg_desc \_para[1]; /* ptr to an array of description of args */
};
```

The last field in `struct _msg`, `_para[1]` is actually a pointer to an array of `struct _arg_desc` which is defined as

```
struct _arg_desc {
   struct _elem_tab *type;
   int offset;
};
```

The first field of `struct _arg_desc`, `type` is a pointer to a structure which contains information of the type of the parameter. The second field of `offset` is the address of where the value of the parameter is located.

What follows the array (`_para[n]`), is a piece of memory where the values of the parameters are stored. i.e. every offset in `_para[i]` should point to somewhere in this block of memory.

10.3.2 Representation of $option

$option is represented by `_frame->$.msg` of the current frame(method). Several macros are provided to facilitate easy reference to information of $option.

```
#define MSG             (struct \_msg *) _frame->$.msg
#define MSG_TYPE        (MSG ? MSG->_type : 0)
#define MSG_SIZE        (MSG ? MSG->_size : 0)
#define MSG_MSG_TYPE    (MSG ? MSG->_msg_type : SYM_MSG)
#define MSG_MSG_TYPE    (MSG ? MSG->_msg_type : SYM_MSG)
#define MSG_NAME        (MSG ? MSG->_msg_name : "")
#define MSG_RETURN_TYPE (MSG ? MSG->_return_type : (struct _elem_tab *) NULL)
#define MSG_RETURN_ADDR (MSG ? MSG->_return_addr : (char *) NULL)
#define MSG_NUM_ARG     (MSG ? MSG->_num_arg : 0)
#define MSG_ARG_DESC    (MSG ? MSG->_para : (struct _arg_desc *) NULL)
```

10.3.3 Allocating the Structure

C code is generated to allocate the structure at a call to a function which is passing optional parameters along. `_frame->$.msg` points to the piece of memory allocated.

If there are n optional parameters, and the total size of all n optional parameters is known to be p_size, the size allocated is:
sizeof(struct _msg) + (n-1)*sizeof(struct _arg_desc) + p_size

If $option is passed in as an optional argument with n other optional

parameters, the number of arguments would be
n + MSG_NUM_ARGS
where MSG_NUM_ARGS is a macro to evaluate the number of arguments in `_frame->$.msg` of the current frame(method).

The size of the new structure allocated would be
sizeof(struct _msg) + (n + MSG_NUM_ARGS -1)*sizeof(struct _arg_desc) + p_size + MSG_SIZE
where MSG_SIZE is a macro to evaluate the size of `_frame->$.msg` of the current frame(method).

If some parameters in `_frame->$.msg` has been used to fill in the required parameters already, the number of arguments and the size to be allocated need to be calculated differently. This will be discussed in the following sections.

10.3.4 Filling in _para[i]

For every argument, its type is represented by an entry in a types_table: `$<featurename>$_types_tab[i]`.

`_para[i].type` will thus contain the address to this entry. `_para[i].offset` will contain the address of the value of the parameter.

C assignment statements are generated to fill in these two fields usually. However, if $option is passed as an optional argument, a `memcpy()` function call would be generated with the following arguments:

```
destination address         _para[i],
source address              MSG->_para[0]
```

```
size                        MSG_NUM_ARGS * sizeof(struct _arg_desc)
```

The calculation differs slightly if some parameters in MSG has been used to fill in the required parameters already. This will be discussed later. This is supported by `pack_para_types_ptrs()`.

10.3.5 Filling in the Values of the Parameters

For every argument, its value would be copied into the structure. The address to copy to is calculated by

```
_frame->$.msg + sizeof(struct _msg) + (n-1) * sizeof(struct _arg_desc)
+ sizeof(all optional arguments before)
```

C assignment statement is generated to put the value of the argument at the calculated address. *(addr) = argument;

If $option is used as an optional argument, a `memcpy()` function call would be generated with the following arguments:

```
destination address          addr
source address               MSG->para[0].offset
size                         MSG_SIZE -
                               (MSG_NUM_ARG -1)*sizeof(struct _arg_desc)
```

`memcpy()` will copy all arguments in the MSG into `addr`.

The calculation differs slightly if some parameters in MSG has been used to fill in the required parameters already. This will be discussed later. The function `fill_args()` supports this.

10.3.6 Mapping Optional Parameters

```
------------------------------------------------------------------------------
actual arguments        formal parameters        mapping
------------------------------------------------------------------------------

(i, "hello")                (int x, char *s)        i -> x, "hello" -> s
(i, "hello")                (int x, optional)        i -> x, MSG has 1 arg: "hello"
(i, "hello")                (optional)                MSG has 2 args: i, "hello"

------------------------------------------------------------------------------
(i, $option)                (int x, char *s)        i -> x,
                                                1st arg in $option -> s
                                                rest of arg in $option discarded
(i, $option)                (int x, optional)        i -> x, all of $option -> MSG
(i, $option)                (optional)                i, all of $option -> MSG

------------------------------------------------------------------------------
($option)                (int x, char *s)        1st arg of $option -> x
                                                2nd arg of $option -> s
                                                rest of $option discarded
($option)                (int x, optional)        1st of $option -> x
                                                rest of $option -> MSG
($option)                (optional)                all of $option -> MSG
------------------------------------------------------------------------------
```

Note that in some cases, some arguments in $option will be mapped into the required formal parameters (the 8th example). When `memcpy()`

the information in $option into the new MSG that we are constructing, we would need to skip those arguments that have already been mapped to required formal parameters.

For filling in _para[i] (the destination address in the new MSG), the source address would be computed as

`&( MSG->_para[k])`

where k is the index of the first parameter that's not mapped yet and the size copied would be computed as

`(MSG_NUM_ARG - l) * sizeof(struct _arg_desc)`

where l is the number of parameters which has been mapped already

For filling in the values of the arguments, the source address would be computed as

`(int) (MSG->_para[m-1].offset) + ARG_SIZE(m-1)`

where m is the first parameter that's not been mapped yet, (address of the last mapped parameter + its size)

And the size copied would be computed as

`(int) MSG + MSG_SIZE - (int) (MSG->_para[m-1].offset) + ARG_SIZE(m-1)`

(address of the msg + msg size = address of end of msg)

The relevant functions are `pack_para_types_ptrs()`, and `fill_args()`.

10.4 Optional Parameters with Dynamic Binding

Implementing optional parameters involves modifying the runtime source code as well, specifically metaClass.m. A field of type `struct _msg`, `_frame->$.tmp_msg`, is used to pass the parameters. when a method is to be bound dynamically.

Note that where as `_frame->$.msg` is used to pass option parameters for static binding, it however, always contain to the incoming parameters with respect to a method. This asymmetric design is due to the mechanism for static binding & dynamic binding differs. When a method is bind statically, a new frame for the method is allocated before enqueuing the frame. Thus C code can be generated to setup all optional parameters in the newly allocated frame's $.msg. However, for dynamic binding, the binding code (runtime code) allocates the new frame dynamically, but parameters have to be setup before this allocation. Thus we need a $.tmp_msg in the current frame to pass parameters, where as $.msg in current frame still references the incoming parameters.

The internal_binding() code in class DynamicBinding which performs method binding at runtime needs to allocate the `_frame->$.msg` and initialize it accordingly before returning. The algorithm is the same used for static binding. This is implemented in the runtime in the file `metaClass.m`: class DynamicBinding method internal_binding().

10.5 Other Issues

- not sure how optional parameters works with inheritance. now looks like it works ok, at least with static binding
- `char *`, seems weird, need to use `char **p` to dereference
- freeing the `_frame->$.msg` & `_frame->$.tmp_msg` is done by the runtime

Chapter 11

Types

11.1 Static Type Checking

The MeldC compiler supports strong typing at compile time with static type checking. The type of an expression is stored as a postfix notation string during compilation, and the type checking functions most often simply do a smart string comparison. The postfix notation strings are are constructed in the second pass of the compilation as it parses. When a declaration is parsed, the global variables `std_name`, `std_type` contains the identifier of the declaration and its type in postfix notation respectively. For example, with the declaration '`int (*foo)()`', `std_name` will contain the string "foo" and `std_type` will contain the string "*() int".

The postfix notation string is constructed by the function `compose_types()` which dispatches the work to `compose_prefix_type(), compose_postfix_type()` and `compose_binary_types()` which deal with prefix, postfix, and binary operators on an expression respectively. `compose_binary_types()` tries to find the type of the resulting expression on a case by case analysis of the operator. It now handles: relational operators and logical operators, ptr reference, structure reference and common binary arithmetic operators.

The type composing functions often access a conversion table `conv[]` (`decl_tab.c`) for a quick check of type compatibility of basic types with some basic unary operator. The table's entry contains the resulting type if they are compatible. Some arithmetic type promotion is also returned by the table.

The function `typecmp()` takes two postfix notation strings and determine if the two types are compatible types by doing a smart character by character comparison. Special cases such as checking against an object against a *generic* object, a pointer with an array, and stripping off meaningless parathesis are also handled (not that well though). As the pointers advanced on both strings, `typecmp()` calls itself recursively on the sub-strings. If the two postfix notation strings are type compatible, 0 is returned, following the convention of the C function `strcmp()`.

The following functions are relevant for type checking (`decl_tab.c`):

- compose_types()
- compose_prefix_type()
- compose_postfix_type()
- compose_binary_types()
- type_exists()
- type_number()

- typecmp()
- typedef_transform()

11.2 CTD : In-Line C type Description

The purpose of CTD is to provide a standard format to describe a C type or a sequence of C types. A CTD string can be used in passing parameters to an unknown function (i.e., binding between the caller and the callee is determined at runtime.It is also called dynamic binding), passing self-described data structures to remote hosts, storing self-describe data structures in files, or querying the types of remote data structures. CTD serves as a build block to many sofware requiring dynamic behavior.

The language is devided into three parts: build-in type description, sequence description and composed type description.

11.2.1 Build-in Type Description

The key is to use so called postfix notation of C type description. This is easy to describe in examples. For type declarations and their postfix notation strings.

```
int foo                 --> "int"
char *foo[5]            --> "[5]* char"
unsigned int *(*foo)()  --> "*()* unsigned int"
```

In CTD, every build-in C type will be assigned a single character for the efficiency reason. The mapping is as follows

```
int      --> i
float    --> f
double   --> d
char     --> c
long     --> l
unsign   --> n         *** u is used for union
short    --> h         *** s is used for struct
void     --> v
constant --> t         *** c is used for char
enum     --> e
```

Thus the CTD string for the above examples are

```
int foo                 --> "i"
char *foo[5]            --> "[5]* c"
unsigned int *(*foo)()  --> "*()* ni"
```

11.2.2 Sequence Description

To describe a sequeuce of types, comma "," is used to seperate consecutive CDT strings. For eaxmple, to describe a sequence for the above three examples, the CTD string is `i, [5]* c, *()* ui`

11.2.3 Composed Type Description

Three elements are considered as the building blocks for composed C types: `struct`, `union` and `enum`. A character is assigned for each one: `s` for `struct`, `u` for `union`, and `e` for `enum`. The syntax for describing a structure/union/enum is as follows:

```
s {<cdt1>, <cdt2>, ..., <cdtn>}
u {<cdt1>, <cdt2>, ..., <cdtn>}
e {<cdt1>, <cdt2>, ..., <cdtn>}
```

This also handles the case of nested strcutures. To handle the recursive definition, it is necessary to used labels. The labels used in CDT are integers. For example, two C structures are

```
struct foo {
  int x;
  struct bar *p;
};

struct bar {
  float x;
  struct foo *p;
};
```

The CDT string for `struct foo` is `s1 {i,s{i, *s1}}`

The most important function implementing CTD is `get_type_descriptor()` which recursively calls itself to construct the CTD for nested structures/unions.

The relevant functions (in `decl_tab.c` involved are:

- map_types()
- get_type_descriptor_label()

- get_type_descriptor()
- add_StructNode()
- in_struct_list()

11.3 Type Description Tables

The type description tables supports the runtime system in message passing for remote objects. These tables provides all the type information which is encapsulated by instance variables in classes or elements in a structure or union.

A linearizing table is generated for each feature at the end of the cfile generated. The name of the linear table will be $<featurename>$linear_tab. The type of this table is struct _linear_tab which is defined in meldc_user.h. The generated code will contain information for typedef, struct, and union defined within the feature. There will be one entry for each typedef/struct/union in the table. Information in the structs will be seperated by commas. If it is a union, then the linear_str will be (char *) 0 (NULL!) since a union cannot be send over the net anyways. The last entry of the table is always a NULL entry i.e. { UNDEFINED, (char *) 0, (char *) 0 }.

```
e.g.
CLASS blah ::=
  struct foo {
        int x;
        char *y;
        struct foo *next;
```

```
  };
  typedef int bar;
  union foo1 {
        int a;
        float b;
  };
END CLASS

-->
   struct _linear_tab $feature$class$linear_tab[] = {
          { STRUCT, "foo", "int,char,foo" },
          { TYPEDEF, "bar", "int" },
          { UNION, "foo1", (char *) 0 }
   };
```

The definition of the tables used to represent types follows:

```
struct _elem_tab
    {
     char *c_type_desc;             /*
                                     * the C language type description string
                                     * we talked about
                                     */
     char *name;                     /*
                                      * name of the type (not really neccessary)
                                      */
     int  size;                      /*
                                      * sizeof this type
```

```
                                    */
     int  offset;                  /*
                                    * offset of this member inside a struct,
                                    * make sense only in the second level
                                    * of indirection
                                    */
     struct _elem_tab *elem_tab;   /*
                                    * 2nd level if this member is a struct
                                    */
    };
```

An *elem_tab* is generated for each user defined struct/union or class, and its names is `$<featurename>$<name of strcut/union>$_elem_tab` or `$<featurename>$<classname>$_elem_tab` in the latter case. The number of entries in this table equals the number of members the struct/union has. The last entry of this table is always {`(char *) 0, (char *) 0, 0, 0, (struct _elem_tab *) 0` } The `offset` field in the entry is offset for the member in the struct/union.

A linear table is generated for each feature and its name is `$<featurename>$_linear_tab`.

Its entries are entries for all user defined struct/union/typedefs, as well as basic types. the fields `offset` in these entries are meaningless. If the entry is a struct/union, then its `elem_tab` field is a pointer to the element table for that struct/union, otherwise, it is NULL.

```
Example :

definitions in .m file

  struct bar {
     int y;
  };
  struct foo {
     int x;
     char *s;
     struct bar bar_s;
  };
  typedef int haha;


tables generated in ..c file

struct _elem_tab $fff$bar$_elem_tab[] = {
  {"i", " int", (int) sizeof(int ), (int) &((struct bar *)0)->y, (struct _elem_tab *) 0},
  {(char *) 0, (char *) 0, 0, 0, (struct _elem_tab *) 0 }
};

struct _elem_tab $fff$foo$_elem_tab[] = {
  {"i", " int", (int) sizeof(int ), (int) &((struct foo *)0)->x, (struct _elem_tab *) 0},
  {"*c", "* char", (int) sizeof(char *), (int) &((struct foo *)0)->s, (struct _elem_tab *) 0},
  {"s{ i }", " struct bar", (int) sizeof(struct bar ), (int) &((struct foo *)0)->bar_s, $fff$bar$_elem_tab},
  {(char *) 0, (char *) 0, 0, 0, (struct _elem_tab *) 0 }
};
```

```
struct _elem_tab $fff$_types_tab[] = {
  { "s", "struct bar", (int) sizeof (struct bar), 0, $fff$bar$_elem_tab },
  { "s", "struct foo", (int) sizeof (struct foo), 0, $fff$foo$_elem_tab },
  { "typedef", "haha", (int) sizeof(haha), 0, (struct _elem_tab *) 0 },
  /* the standard types */
  { "i", "int", (int) sizeof(int), 0, (struct _elem_tab *) 0 },
  { "f", "float", sizeof(float), 0, (struct _elem_tab *) 0 },
  { "c", "char", (int) sizeof(char), 0, (struct _elem_tab *) 0 },
  { "h", "short", (int) sizeof(short), 0, (struct _elem_tab *) 0 },
  { "l", "long", (int) sizeof(long), 0, (struct _elem_tab *) 0 },
  { (char *) 0, (char *) 0, 0, 0, (struct _elem_tab *) 0 }
};
```

The relevant functions (`decl_tab.c`) are:

- init_linear_defn()
- add_linear_defn()
- print_obj_elem_tab()
- print_struct_elem_tab()
- print_linearize_tab()

Chapter 12

Buffer Management

To improve performance. 20 buffers of 1K bytes are pre-allocated to be used by the compiler. The `catn()` functions uses these buffers. However, these buffers are reused throughout the compilation, and the data stored in these buffers may be overwritten. These buffers should only be used for storing data for a very short time, i.e. a few instructions.

The functions `strcatn()` returns buffers allocated on the heap dynamically. Hence, to ensure there's no memory leakage, these buffers should be freed after used. Currently, the compiler's memory management is not uniform, and memory usage is not very careful.

The data structure storing expressions has its own managing functions and it is discussed with code generation on expressions.

- cat1
- cat2

- cat3
- cat4
- cat5
- cat6
- cat7
- strcat2
- strcat3
- strcat4
- strcat5
- strcat6
- strcat7
- strsave

Chapter 13

Problems

- No inheritance support for `ctype` classes.
- Active values are not inherited. This can be easily fixed: while parsing an identifier, while checking if it is an inherited instance variable, it can also be checked to see if it is active. If so, code can be generated to expand into a C comma expression which triggers the procedure associated with the active value. (See chapter X for details about active values)
- Currently, in the procedure associated with the active value, no message passing is allowed. We would like the effect of triggering an active value be *instantaneous*, hence, the procedure is implemented as `ctype` methods. Since the local variables and parameters are not preserved on the stack during context switch, no `meldtype` function calls can be made. The solution may lie in having a global lock which suspends execution of all threads until the synchronous message passing

triggered by the active value is finished.

- Class variables in (MeldC are implemented as global variables in C. They should really be associated with the class object.

- The `$term()` function in each class is an empty function. It is not clear what its functionality is. Its existence came about because we want a parallel to `$init()`.

- The compiler supports two kinds of parameter declarations, both ANSI C as well as K&R C, and this causes some lex/yacc hack in the parser. Should we make this uniform and just accept one kind of declaration?

- Some lex/yacc problems also arises from allowing some MeldC keywords to be used as identifiers in expressions. A classic example is the keyword `send` where it is also a useful C system call `send()`.

- Type checking could be improved.

- Memory problems with `catn()` and `strcatn()`. While we have to worry about overwriting the memory when using `catn()` (since it recycles the 20 buffers), we have to worry about freeing the memory allocated by `strcatn()`. Would probably like a more uniform way to manage memory.

- However, the biggest challenge ... lies in error handling in the first pass of the compilation. The responsibility of the first pass is just to collect information about the classes. Hence it does not care about the implementation of the methods and ignores them with a few obscure grammar rules. Unfortunately, these rules are not perfect and will behave in an unexpected manner if the input file is erroneous. Thus the problem: how to catch as much syntax error as possible while not ignoring too much of the code. Another solution is to bring back all the grammar rules which really parses the expressions; this, however, will do more than required in the first compilation. The grammar rules below tries to ignore the expressions.

```
method\_body
        : regular_begin anything regular_end
        | regular_begin regular_end
        ;

/*
 * THESE RULES DOES NOT TAKE CARE OF THE CASE
 * OF MISSING A ';' AT THE LAST STMT INSIDE A BLOCK
 * AND WOULD SILENTLY FAIL
 */
anything:
    error ';' /* 1 stmt */
        | anything error ';' /* recursive stmts */
        | error begin anything end /* compound stmt */
        | anything begin end /* recursive empty compound stmt */
        | anything begin anything end /* recursive compound stmt */
        | error '(' error ')' /* func(arguments ... ) */
        | anything '(' error ')'
        | anything error regular_begin anything regular_end
        ;
```

Appendix A

User's Guide

MeldC programs are compiled with `mcc`, a UNIX script file that runs the MeldC compiler and linker on the input files. While it is possible to use the compiler and linker alone, the `mcc` script can handle almost all of a programmer's needs.

`mcc`'s syntax is

`mcc` *options list-of-filenames*

where *list-of-filenames* consists of the files of the MeldC features making up the program. Each filename should end with "`.m`" if it is a MeldC program file. `mcc` also accepts "`.c` and "`.o`" files (C source or object files) to aid with separate compilation. .

`mcc` reports errors and warnings from the MeldC compiler, but does not report warnings generated by the C compiler, only errors.

`mcc` accepts any options accepted by conventional C compilers such as `cc` and the Gnu Project's `gcc`, plus some special ones:

- `-err<number>` Do not abort when the compiler signals an error. If the optional `<number>` parameter is used, `mcc` will abort only after `<number>` errors are found.
- `-g` Keep auxiliary files. If this switch is set then the `mcc` script will not delete certain files created by the compiler and linker.
- `-m` Do not automatically import the MetaClass. This should only be used if the programmer wishes to manually import the MetaClass.
- `-p` Only go through pass one of the compiler.
- `-v` Verbose. Print out various MELDC compiler information.

A `Makefile` is used to compile the MELDC compiler. Since we have two parsers in our compiler, the files generated by Yacc have to be renamed by a `sed` command. All identifiers starting with "yy" generated from the file `p1.y` is renamed to start with "yy_1_", while those in `p2.y` are renamed to start with "yy_2_". The C file generated from `p1.y` is renamed as `parse1.c`; from `p2.y` `parse2.c`. The two header files generated by Yacc are renamed to `y.tab1.h` and `y.tab2.h` respectively.

Appendix B

Debugging Tips

Debugging the MeldC compiler requires great patience, so the first rule is - go out and take a deep breath before you start. Cheers! The tools available helps, but it really requires a thorough understanding of how the compiler works, how each different components relate to each other. When dealing with shift/reduce errors of the parser, `ydb` could be helpful. Turning on `yydebug` and have it print our all the reductions when the compiler runs also helps. Note that any identifier starts with `yy` is renamed by the `sed` script which compiles two parser in our compiler. Another trick to stop in a rule is to define a dummy function, insert a call to this function in the rule, then ask `dbx` to stop in the function. I have discovered that this works better than `ydb`. And of course, lots of printfs helps too.

Segmentation faults are probably due to some memory being overwritten and the stack is corrupted. This often happens during code generation for a method call, since the number of parameters is arbitrary, buffer space may run out. Stored expressions can get corrupted because the `TMPSTR` returned by the `catn()` is recycled already.

The compiler uses an environment variable `MCC_PATH` to look for external auxiliary files such as the `MeldC.info` and others. However, the `dbx` environment is ignorant of this, thus to run the compiler in `dbx`:

```
run -E<$MCC_PATH/extern_info/$ARCH> filename..m
```

Note that successful execution of the compiler does not mean successful code generation. The cfiles and hfiles together with the file `meldC.out.c` should compile and the executable `a.out` should run. If the files compile but core dumps when running, it may be due to incompatibility with the runtime libraries. Re-compiling the meta-class and other MeldC object libraries may help. It may be safe to say that if the compiler generates correct code for the meta-class, it should be in a healthy state.

Appendix C

List of Functions

The following files implements the MeldC compiler:

- active.c
- build_classes.h build_classes.c
- codegen.c
- decl_tab.h decl_tab.c
- extern.c
- generic.c
- header.c
- list.h list.c
- machdep.c

- main.c
- p1.y p2.y
- s1.l s2.l
- support.h support.c

The following functions implements the compiler:

- active_list src/active.c 2
- address_of_node src/active.c 10
- add_child src/build_classes.c 17
- add_class src/build_classes.c 5
- add_class_decl src/decl_tab.c 25
- add_class_node src/build_classes.c 18
- add_decl src/decl_tab.c 6
- add_defn src/decl_tab.c 21
- ADD_EXTERN_NAME include/decl_tab.h 1
- add_feature_decl src/decl_tab.c 25
- add_formals src/decl_tab.c 26
- add_imports src/build_classes.c 11
- add_instance src/build_classes.c 6
- add_instance_node src/build_classes.c 18
- add_linear_defn src/decl_tab.c 51

- cat3 src/support.c 1
- cat4 src/support.c 2
- cat5 src/support.c 2
- cat6 src/support.c 2
- cat7 src/support.c 2
- char_to_myunion src/list.c 1
- check_circular_inheritance src/build_classes.c 20
- CHECK_CLASS src/decl_tab.c 20
- check_conflict_inst_var src/build_classes.c 32
- check_duplicate_instance_var src/build_classes.c 21
- check_duplicate_instance_var_parent src/build_classes.c 21
- CHECK_FORMAL src/decl_tab.c 20
- check_for_dependencies src/active.c 11
- check_inode src/build_classes.c 32
- check_non_existent src/build_classes.c 22
- check_non_existent_parent src/build_classes.c 22
- CHECK_TMPNAME src/decl_tab.c 2
- CHECK_TMPSTR include/support.h 1
- circ_depend src/active.c 11
- CLASS_NAME src/decl_tab.c 27
- cleanup src/main.c 5

- decl_class_as_obj src/decl_tab.c 12
- decl_number src/decl_tab.c 15
- decl_str src/decl_tab.c 4
- defn_typ2str src/decl_tab.c 50
- delete_class_node src/build_classes.c 18
- do_rules_exist src/active.c 2
- DUAL_STR_DUP src/build_classes.c 39
- duplicate src/build_classes.c 27
- dynamic_send src/codegen.c 3
- END_STR include/parser.h 2
- enq_new_defn src/decl_tab.c 34
- ERROR include/parser.h 2
- fill_args src/codegen.c 22
- find_all_member_defns src/decl_tab.c 23
- find_ancester src/build_classes.c 41
- find_ClassNode src/build_classes.c 19
- find_class_defn src/decl_tab.c 23
- find_decl src/decl_tab.c 8
- find_defn src/decl_tab.c 22
- find_enum_type src/decl_tab.c 38
- find_export_file src/extern.c 11

- free_defn_list src/decl_tab.c 37
- free_depend_list src/active.c 12
- FULL_CLASS_LABEL src/codegen.c 1
- full_obj_name src/decl_tab.c 9
- generate_mesg_structure src/codegen.c 18
- gen_code_seq src/active.c 13
- gen_initializer src/codegen.c 12
- gen_merge_sync_code src/codegen.c 13
- gen_rule_seq src/active.c 14
- gen_save_mesg_structure src/codegen.c 26
- gen_static_parameter src/codegen.c 16
- gen_terminator src/codegen.c 12
- get_full_classname src/decl_tab.c 38
- get_import_fname src/build_classes.c 13
- get_para_no src/decl_tab.c 34
- get_para_values src/build_classes.c 38
- get_parent_obj src/build_classes.c 42
- get_size_str src/decl_tab.c 55
- get_type_descriptor src/decl_tab.c 58
- get_type_descriptor_label src/decl_tab.c 58
- handle_exports src/build_classes.c 13

- is_identifier src/extern.c 7
- is_imported_class src/build_classes.c 12
- is_really_change src/active.c 4
- itoa src/decl_tab.c 3
- last src/list.c 2
- last_defn src/decl_tab.c 21
- link_depend_list src/active.c 10
- listlen src/list.c 2
- list_find_str src/list.c 2
- list_to_myunion src/list.c 1
- main src/main.c 2
- make_CharListNodeHeader src/build_classes.c 15
- make_ClassNode src/build_classes.c 15
- make_ClassNodeHeader src/build_classes.c 15
- make_create_para src/decl_tab.c 31
- make_InstanceNode src/build_classes.c 16
- make_ListNodeHeader src/build_classes.c 15
- make_MergeParamList src/build_classes.c 36
- make_merge_arg src/build_classes.c 39
- make_MethodNode src/build_classes.c 16
- make_ParamList src/build_classes.c 35

- print_class_info_ifile src/decl_tab.c 30
- print_class_init src/decl_tab.c 33
- print_decl_structure src/decl_tab.c 14
- print_decl_tabs src/decl_tab.c 49
- print_defn_symbol src/decl_tab.c 49
- print_defn_tabs src/decl_tab.c 49
- print_header src/header.c 1
- print_instance_tab src/decl_tab.c 13
- print_last_n_data_decls src/decl_tab.c 16
- print_last_n_init_stmts src/decl_tab.c 17
- print_last_n_objects src/decl_tab.c 18
- print_last_n_pseudo_decls src/decl_tab.c 17
- print_linearize_tab src/decl_tab.c 55
- print_obj_elem_tab src/decl_tab.c 52
- print_pseudo_decl src/decl_tab.c 16
- print_static_init_stmts src/decl_tab.c 18
- print_struct_elem_tab src/decl_tab.c 54
- print_struct_union_enum_typedef src/decl_tab.c 39
- print_usage src/main.c 7
- process_parents src/build_classes.c 9
- proc_send_stmt src/codegen.c 2

- SKIP_NON_LABEL include/parser.h 2
- SKIP_NON_SPACE include/parser.h 1
- SKIP_PARENTHESIS include/parser.h 2
- skip_parenthesis src/decl_tab.c 3
- skip_pointer src/generic.c 1
- SKIP_SPACE include/parser.h 2
- skip_space_or_comma src/extern.c 7
- static_send src/codegen.c 5
- strcat2 src/support.c 3
- strcat3 src/support.c 3
- strcat4 src/support.c 4
- strcat5 src/support.c 4
- strcat6 src/support.c 4
- strcat7 src/support.c 5
- strsave src/support.c 3
- strstr src/machdep.c 2
- struct_init_code src/decl_tab.c 18
- sync_inherited_members src/build_classes.c 34
- TALLOCATE src/list.c 1
- TITLE_HEAD src/decl_tab.c 3
- TITLE_TAIL src/decl_tab.c 3