Melange User Guide

Release 1

Dylan Hackers

December 15, 2018

The Melange interface generator provides a mechanism for providing access to native C code. It is modeled upon Apple Computer's Creole, and shares Creole's goals of automatically providing full support for a foreign interface based upon existing interface descriptions. It also, like Creole, provides mechanisms for explicitly adapting these interfaces to provide a greater match between C and Dylan data models.

Melange, however, differs from Creole in a number of significant ways. This document, therefore, provides a gentle introduction to Melange without attempting to build upon any existing descriptions of Creole.

CHAPTER

ONE

INTRODUCTION

Melange is an automatic interface generator which provides transparent access to both functions and data defined or generated by existing C libraries. It allows users to import "interfaces" [1] from existing C header files, controlled by the contents of a `define interface` top-level form which may be included in the same file as arbitrary Dylan code. The user may use the functions and data specified by this interface as if they were native Dylan objects, and may export them to other modules.

Melange provides reasonable interpretations for the various sorts of C declarations which may appear in a header file, as well as mechanisms for explicitly modifying the default interpretations when necessary. For example, users may:

- specify rules for the translation of foreign names
- explicitly specify new names for specific objects or routines
- specify parameter passing conventions or mutability of foreign objects
- specify mappings or equivalences between "foreign" data and native equivalents
- choose to import only a subset of the declarations in the header file

All of these customizations, as well as the name of the C header file, are specified by a `define interface` clause. See *A Concrete Example* for an example.

[1] In fact, a C header file may contain arbitrary C code which Melange is unprepared to handle. By convention, however, `.h` files contain only "interface declarations" – type declarations, function prototypes, global variable declarations, and "preprocessor constants." Since Melange can meaningfully process all of these, it is capable of handling the vast majority of header files which will be encountered in practice.

CHAPTER

TWO

A CONCRETE EXAMPLE

In order to get a feel for using Melange, it is probably best to start with a concrete example. This section contains a complete program which will use native C libraries to list the contents of some directories. For now, you should simply skim this example to get a general overview of Melange's capabilities. These will be described in more detail in later sections.

We will first begin with an "interface file" which contains a mixture of basic Dylan code and `define interface` forms which will be processed by Melange. We will name this file `dirent.intr`.

```
module: Junk
synopsis: A poor imitation of "ls"

define library junk
  use dylan;
  use streams;
end library junk;

define module junk
  use dylan;
  use extensions;
  use extern;
  use streams;
  use standard-io;
end module junk;

define interface
  // This clause is more complex than it needs to be, but it does
  // demonstrate a lot of Melange's features.
  #include "/usr/include/sys/dirent.h",
    equate: {"char /* Any C declaration is legal */ *" => <c-string>},
    map: {"char *" => <byte-string>},
    // The two functions require callbacks, which we don't support.
    exclude: {"scandir", "alphasort", "struct _dirdesc"},
    seal-functions: open,
    read-only: #t,
    name-mapper: minimal-name-mapping;
  function "opendir", map-argument: {#x1 => <string>};
  function "telldir" => tell, map-result: <integer>;
  struct "struct dirent",
    prefix: "dt-",
    exclude: {"d_namlen", "d_reclen"};
end interface;

define method main (program, #rest args)
  for (arg in args)
    let dir = opendir(arg);
```

```
    for (entry = readdir(dir) then readdir(dir),
         until entry = $null-pointer)
      write-line(entry.dt-d-name, *standard-output*);
    end for;
    closedir(dir);
  end for;
end method main;
```

We will then process this file through Melange to produce a file of pure Dylan code. Melange can be invoked like this:

```
melange dirent.intr dirent.dylan
```

This command will process `dirent.intr` and write a file named `dirent.dylan`.

You can compile `dirent.dylan` normally within a Dylan library.

CHAPTER

THREE

BASIC USE

Although the `define interface` form provides a fairly rich sublanguage for specifying interfaces, it is often sufficient to use just the "minimal" form. For example, if `gc.h` contained the following code:

```
typedef char bool;
typedef struct obj obj_t;
typedef char *str;
extern obj_t alloc(obj_t class, int bytes);
extern void scavenge(obj_t *addr);
extern obj_t transport(obj_t obj, int bytes);
extern void shrink(obj_t obj, int bytes);
extern void collect_garbage(void);
extern bool TimeToGC;
#define ForwardingMarker ((obj_t)(0xDEADBEEF))
```

then you could import it by creating a file named `class.intr` which includes arbitrary Dylan code and the following:

```
define interface
   #include "gc.h";
end interface;
```

You would then run `melange class.intr class.dylan` which would produce a file of Dylan code which contains appropriate definitions for the classes `<bool>`, `<obj>`, `<obj_t>`, and `<str>`; the variable `TimeToGC`; and the functions `alloc`, `scavenge`, `transport`, `shrink`, and `collect_garbage`. (The constant `ForwardingMarker` will be excluded because it is not a simple literal.)

```
if (TimeToGC() ~= 0)
   collect_garbage();
end if;
```

This code fragment points out some of the hazards of "simple" imports. Melange has no way of knowing that `bool` should correspond to Dylan's `<boolean>` class, so you are stuck with a simple integer. Likewise, the system wouldn't be able to guess that `char *` should correspond to the Dylan class `<c-string>`. We will explain in later sections how "map:" or "equate:" options may be used to provide this information to Melange.

CHAPTER

FOUR

IMPORTING HEADER FILES

You import C definitions into Dylan by specifying one or more header files in an `#include` clause. This may take one of two different forms:

```
define interface
   #include "file1.h";
end interface;
```

or

```
define interface
   #include {"file1.h", "file2.h"};
end interface;
```

Melange will parse all of the named files in the specified order, and produce Dylan equivalents for (i.e. "import") some fraction of declarations in these files. By default, Melange will import all of the declarations from the named files, and any declarations in recursively included files (i.e. those specified via `#include` directives in the `.h` file) which are referenced by imported definitions. It will not, however, import every declaration in recursively included files. This insures that you will see a complete, usable, set of declarations without having to closely control the importation process. If you wish to exert more control over the set of objects to be imported, you may do so via the `import`, `exclude`, and `exclude-file` options.

If you only need a small set of definitions from a set of imported files, you can explicitly specify the complete list of declarations to be imported by using the `import:` option. You could, for example, say:

```
define interface
   #include "gc.h",
      import: {"scavenge", "transport" => move};
end interface;
```

This would result in Dylan definitions for `scavenge`, `move`, `<obj_t>`, and `<obj>`. The latter types would be dragged in because they are referenced by the two imported functions. Again, if you equated `obj_t` to `<object>` then neither of the types would be imported. The second import in the above example performs a renaming at the same time as it specifies the object to be imported. Other forms specify global behaviors. `Import: all` will cause Melange to import every "top level" definition which is not explicitly excluded. `Import: all-recursive` causes it to import definitions from recursively included files as well. `Import: none` restricts importation to those declarations which are explicitly imported or are used by an imported declaration. You may also use the `import:` option to specify importation behavior on a per-file basis. The options

```
import: "file.h" => {"import1", ...}
import: "file.h" => all
import: "file.h" => none
```

work like the options described above, except that they only apply to the symbols in a single imported file. The `exclude:` and `exclude-file:` options may be used to keep one or more unwanted definitions from being imported. For example, you could use:

```
define interface
   #include "gc.h",
      exclude: {"scavenge", "transport"},
      exclude-file: "gc1.h";
end interface;
```

This would prevent the two named functions and everything in the named file from being imported, while still including all of the other definitions from `gc.h`. Note that these options should be used with care, as they can easily result in "incomplete" interfaces in which some declarations refer to types which are not defined. This could result in errors in the generated Dylan code. (The `import: file => none` option described above is a safer way of achieving an effect similar to `exclude-file:`.)

You may also prevent some type declarations from being imported by using the `equate:` option (described in a later section). If, for example, you equated `obj_t` to `<object>`, then Melange would ignore the definition for `obj_t` and simply assume that the existing definition for `<object>` was sufficient.

You may have any number of `import:`, `exclude:`, and `exclude-file:` options, and may name the same declarations in multiple clauses. `Exclude:` options take priority over `import:`. If no `import:` options are specified, the system will import all non-excluded symbols, just as if you had said `import: all`.

CHAPTER

FIVE

SPECIFYING OBJECT NAMES

Because naming conventions differ between C and Dylan, Melange attempts to translate the names specified in C declarations into a form more appropriate to Dylan. This involves

- Adding angle brackets around type names.
- Adding dollar signs at the beginning of constant names.
- Translating (non-initial) underlines into hyphens.
- Adding `struct-name$` prefixes to slot accessors.

In many cases, this default behavior will be precisely what you want. However, Melange provides mechanisms for specifying different translations for some or all of the declarations.

Mapping functions

The translations described above are provided by calls to a built-in "name mapping function" named "minimal-name-mapping-with-structure-prefix". You may specify other mapping functions via a `name-mapper:` option. Our example interface might then look like this:

```
define interface
   #include "gc.h",
      name-mapper: c-to-dylan;
end interface;
```

Function	Result
minimal-name-mapping-with-structure-prefix	Provides the translations described above.
minimal-name-mapping	Same as above, but excludes `struct-name$`.
c-to-dylan	Like `minimal-name-mapping`, but: * Adds hyphens to reinforce "CaseBased" word * Adds `get-` prefixes to slot accessors.
identity-name-mapping	Does no translation.

New name-mapping functions can be implemented within melange by defining methods on the `map-name` generic function which accepts the following parameters:

`mapper` a `<symbol>` which is typically specialized by a singleton to select a specific name mapper method.

`category` a `<symbol>` which will always be one of: `#"type"`, `#"constant"`, `#"variable"`, or `#"function"`.

`prefix` a `<string>` which is typically prepended to the result string.

`name` a `<string>` which supplies the original C name.

sequence-of-classes a sequence of simple names for the classes which logically "contain" the given object. For example, if we were processing the declaration `struct str {int size; char *chars;}`, one of the calls to the mapping function would have with name bound to "size" and classes bound to `#["str"]`.

It must return a `<string>` which will be used as the Dylan name for the declaration.

Mapping functions may call `hyphenate-case-breaks` which performs the same "CaseBased separation" as is done by `c-to-dylan`. The trivial `identity-name-mapping` described above might be implemented by:

```
define method map-name
    (mapper == #"identity-name-mapping", category, prefix, name, classes)
=> (result :: <string>)
   name;
end method map-name;
```

You may specify different name mappers to be applied to the slots of "container types". This capability is described in a later section.

Prefixes

As noted above, the name mapping function is passed a `prefix` argument. By default, it is an empty string, but users may specify a different value by adding a `prefix:` option to the interface definition. For example, we might expand the previous example to:

```
define interface
   #include "gc.h",
      name-mapper: c-to-dylan,
      prefix: "gc-";
end interface;
```

This would cause Melange to tack `gc-` onto the beginning of every translated symbol. Because the system knows about the "standard" Dylan naming conventions, it can do this intelligently. You would, therefore, get names like `<gc-bool>`, `gc-time-to-gc`, and `gc-scavenge`.

Note that the interpretation of the `prefix` is entirely up to the name mapping routine. Identity-name-mapping, for example, completely ignores the prefix. All of the other standard mapping functions prepend it to the name before adding brackets or dollar signs, but after performing all other transformations.

Facilities for adding "localized" prefixes to slot accessors, enumeration literals, etc. will be described in later sections.

Explicit Renaming

Although the automatic name mapping described above is sufficient for most objects named within a header file, there are cases in which you might wish to explicitly control the name of one or more specific objects. You can do this through a `rename:` option. This options specifies a list of translations between raw C names and Dylan identifiers. For example, we might have:

```
define interface
   #include "gc.h",
      name-mapper: c-to-dylan,
      prefix: "gc-"
      rename: {"struct obj" => <C-Object>, "collect_garbage" => GC};
end interface;
```

Note that the "target" of the renaming is an ordinary Dylan variable and is therefore case-insensitive. However, the source is an "alien name", which is (like all C code) case sensitive. Alien names should refer to an object, function, or

type in exactly the same way you would refer to them in C. We therefore say `struct obj` instead of simply `obj`, and might also say `enum foo` or `union bar`. Alien names are actually parsed according to the standard lexical conventions of C, so you may use arbitrary spacing and even include comments if you really wish.

Note that `rename:` options supply names for new objects (and types) that are being imported into Dylan. You cannot, therefore, simply rename `bool` to `<Boolean>` to make it equivalent to the existing type – this would simply result in a name conflict. For these purposes, you would instead use the `equate` and `map` operations, which will be described later. (In fact, if the C declaration had defined a type name `boolean`, you might have to explicitly rename it to something else in order to avoid name conflicts with the existing type. Of course, in the above example, the `gc-` prefix would be sufficient to make the name unique.)

Anonymous Types

The alien names described above can also be used to refer to C's so-called "anonymous types". You can therefore refer to `char *`, `int [23]`, or even `int (*) (char *foo)` (i.e. a pointer to function which takes a string and returns an integer) [At present, function types are not fully supported. You should not depend upon them to work as expected.]. The ability to refer to anonymous types is useful because it allows you to use the `rename` option to provide explicit names for such types. Normally Melange would simply generate a an arbitrary "anonymous" identifier for the type. Without knowing the name of this type, you could not define new operations upon it. However, by saying, for example, `rename: {"char *" => <char-ptr>}`, you can provide a convenient handle to use in defining new operations.

CHAPTER

SIX

TYPE DEFINITIONS

When Melange encounters a "type definition" [The definition may be implicit, as in `char ** int` or `struct foo *bar`. Simply by being present these code fragments supply implicit definitions for `char *`, `char **` and `struct foo`.] within a header file, it will typically create a new Dylan class which corresponds to that C type. Usually, this will be a subclass of <statically-typed-pointer>, which encapsulates the raw C pointer value (i.e an object address). Each statically typed pointer class will have exactly the same structure (i.e. a single address), but the class itself can be used to determine what operations are supported on the data. This could include slot accessors for "struct"s and "union"s, dereference operations for "pointer" types, or general information about the objectUs size, etc.

There are times when you will find that some of the types defined in a header file are not really "new". It might be that they are completely identical to some type defined in another interface definition, or they might be "isomorphic" to some existing type which has more complete support. Melange provides support for both of these cases. The first case is handled by "equating" the two types, while the second is handled by "mapping" (i.e. transforming) one type into the other.

For example, many header files contain definitions use the types `char *` and `boolean`. The declarations of these types don't provide any semantic interpretations – `char *` is simply the address of a character, and boolean is nothing but a one-byte integer. However, by equating `char *` to the predefined `<c-string>` type, we can tell Melange that it is actually a `<string>` and should inherit all of the operations defined upon `<string>`. Likewise, we can map the integral `boolean` values into `#t` and `#f` to get a `<boolean>`. These integral values will be automatically translated into `<boolean>` when they are returned by a C function, and `<boolean>` will be translated back into integers when passed as arguments to C functions.

Implicit class definitions

Unless otherwise specified, new classes will be created for each type defined in a C header file. When the header file provides meaningful names for these types, then Melange will pass those names to the mapping functions to generate names for the Dylan classes. Otherwise, an anonymous name will be generated, limiting your ability to refer to the new type. For example, `struct foo` would typically generate the class `<foo>`, while `struct foo ***` might generate the class `<anonymous-107>`. In either case, you can explicitly specify the name for the new class by using the `rename:` option described above.

Different sorts of C declarations will yield different sorts of Dylan classes as well as different sets of operations defined upon them. Therefore, we will consider each variety separately:

Primitive types The types `int`, `char`, `long`, `short` and their unsigned counterparts are simply translated into `<integer>`, while `float` and `double` are translated into `<float>`. However, Melange knows the sizes of each of these types so that pointers and native C "vectors" of them (described below) will work properly. No new types are created for these types.

Pointer types Declarations like `int *` or `struct foo ***` generate new subclasses of `<statically-typed-pointer>`. Note that `struct foo *` is actually treated as a synonym for

`struct foo`, and does not get a distinct class, although any extra levels of indirection (i.e. `struct foo **`) will generate new pointer classes. Three operations are supported upon pointer classes:

```
pointer-value (pointer, #key index) => (value)
```

This function "dereferences" the pointer and returns the value. If index is supplied, then "pointer" is treated as a vector of values and the appropriate element is returned.

```
content-size (cls) => integer
```

Returns the size of the value referenced by instances of "cls". If the size is not known, this is 0.

Note that these types are not automatically treated as vectors. You may, however, make them so by using a `superclasses:` option to make them <c-vector>s.

Vector types Declarations like `char [256]` are treated almost identically to pointer types, but they are automatically defined as subclasses of <c-vector>, so that all vector operations will be defined on them. However, because many systems depend upon the lack of bounds checking in C, vector types have a default size of `#f`. You may explicitly define `size` functions to provide a more accurate size.

Structure types Declarations like `struct bar {int a; char *b;}` also generate new subclasses of `<statically-typed-pointer>`. Melange will define all of the operations defined for pointer values (described above), as well as accessors for each of the structure slots. Structure objects are always accessed through "pointers" to them. Therefore, unless a non-zero index is specified, `pointer-value` will simply return the object passed to it. (The operation is still defined because non-zero indices can be used for vector access.)

Union types Declarations like `union bar {int a, char *b;}` are treated the same as struct declarations, except that the slot accessors all refer to the same areas in memory. Enumeration types – Declarations like `enum foo {one, two, three};` are simply aliased to <integer>. However, constants are defined for each of the enumeration literals.

Typedefs Declarations like `typedef struct foo bar` simply define new names for existing types.

Specifying class inheritance

When Melange creates new `<statically-typed-pointer>` classes, it typically creates them as simple subclasses of `<statically-typed-pointer>`, with no other superclasses. However, you might sometimes need more control over the class hierarchy. For example, you might wish to specify that a C type should be considered a subtype of the abstract class `<sequence>`. You could accomplish this via the following declarations:

```
define interface
   #include "sequence.h";
   struct "struct cons_cell" => <c-list>,
      superclasses: {<sequence>};
   function "c_list_size" => size;
end interface;

define method forward-iteration-protocol (seq :: <c-list>)
  ...
```

Note that the type `<c-list>` will still be a subclass of `<statically-typed-pointer>` – we have simply added `<sequence>` to the list of superclasses. If `<statically-typed-pointer>` is not explicitly included in the `superclasses:` option, then it will be added at the end of the superclass list.

As demonstrated in the above example, you are still responsible for specifying whatever functions are required to satisfy the contract for the declared superclasses. `<C-list>` will be declared as a sequence, but you must specify a forward iteration protocol before any of the standard sequence operations will work properly.

The `superclasses:` option may currently be used within `struct`, `union`, and `pointer` clauses.

CHAPTER

SEVEN

TRANSLATING OBJECT REPRESENTATIONS

Whenever a native C object is returned from a function or a Dylan object is passed into a C function, it is necessary to translate between the object representations used by the two languages. From MelangeUs standpoint, native C objects consist of an arbitrary bit pattern which can be translated to or from a small number of "low level" Dylan types – namely <integer>, <float>, or any subclass of <statically-typed-pointer>. This translation is handled automatically, although the user may explicitly specify which of the possible Dylan types should be chosen for any given C object type. In some cases, a further translation may take place, converting the "low level" Dylan value to or from some arbitrary "high level" Dylan type. (For example, an <integer> might be translated into a <boolean> or a <character>, and a <c-string> might be translated into a <byte-string>.) These "high level" translations are automatically invoked at the appropriate times, but both the "target" types and the methods for performing the translation must be specified by the user.

Specifying low level transformations

The target Dylan type for "low level" translations is typically chosen automatically by Melange. Integer and enumeration types are translated into <integer>; floating point types are translated to <float>; and all other types are translated into newly created subclasses of <statically-typed-pointer>. However, you may explicitly declare the target Dylan type for any C type by means of an equate: option:

```
define interface
   #include "gc.h",
      equate: {"char *" => <c-string>};
end interface;
```

This declaration makes the very strong statement that any values declared in C as char * are identical in form to the predefined type <c-string> (which is described in Appendix I). The system will therefore not define a distinct type for char * and will ignore any structural information provided in the header file. You might also use an equate: option to equate a type mentioned in one interface definition with an identically named type which was defined in an earlier interface definition.

You should use caution when equating two types. Since Melange has no way of knowing when two types are equivalent, it must trust your declarations. No type checking can or will be done, so if you incorrectly equate two types, the results will be unpredictable. In some cases, you may wish to go with the less efficient but slightly safer technique of letting Melange create a new type and then "mapping" that new type into the desired type. (This is described in detail below.)

Note also that two types with identical purposes will not necessarily have identical representations. For example, C's boolean types are simple integers and are not equivalent to Dylan's <boolean>. Again, explicit "mapping" may be used to transform between these two representations.

In the current implementation, an equate: option only applies within a single interface definition. Other interface definitions will not automatically inherit the effects of the declaration. In future versions, we may add the ability to

"use" other interface definitions (just as you would "use" another module within a module definition) and thus pick up the effects of the `equate:` (and `map:`) options within those interfaces.

Specifying high level transformations

Sometimes you may wish to use instances of some C type as if they were instances of some existing Dylan class, even though they have different representations. In this case, you can specify a secondary translation phase which semi-automatically translates between a "low level" and a "high level" Dylan representation. In order to do this, you must provide a `map:` option:

```
define interface
   #include "gc.h",
      equate: {"char *" => <c-string>},
      map: {"bool" => <boolean>};
end interface;
```

This clause will cause any functions defined within the interface to call transformation functions wherever the original C functions accept or return values of type `bool`. Two different functions may be called:

```
import-value (high-level-class :: <class>, low-level-value :: <object>)
```

This function is called to transform result values returned by C functions into a "high level" Dylan class. It should always return an instance of "high-level-class".

```
export-value (lowlevel-class :: <class>, high-level-value :: <object>)
```

This function is called to transform "high level" argument values passed to C functions into the "low level" representations which will be meaningful to native C code. It should always return an instance of "low-level-class".

Default methods, which simply call `as`, are provided for each of these functions. This will be sufficient to transform C's integral `char` into `<character>`, `<c-string>` into other `<string>`, or one "pointer" type into another. There is also a predefined method which will transform `<integer>` into `<boolean>`. However, if you wish to perform arbitrary transformations upon the values, you may need to define additional methods for either or both of these functions. For example, the default methods for transforming to and from `<boolean>` are:

```
define method export-value (cls == <integer>, value :: <boolean>)
 => (result :: <integer>);
   if (value) 1 else 0 end if;
end method export-value;

define method import-value (cls == <boolean>, value :: <integer>)
 => (result :: <boolean>);
   value ~= 0;
end method import-value;
```

It is important to note that, unlike `equate:` options, `map:` options don't prevent Melange from creating new types. You may, in fact, both equate and map the same type. This will cause low level values to be created as instances of the "equated" type and then transformed into instances of the "target" type of the mapping. For example, you might take advantage of the defined transformations between string types by declaring:

```
define interface
   #include "/usr/include/sys/dirent.h",
      equate: {"char *" => <c-string>},
       map: {"char *" => <byte-string>};
end interface;
```

This causes the system to automatically translate `char *` pointers into `<c-string>` (i.e. a particular variety of statically typed pointer) and then to call `import-value` to translate the `<c-string>` into a `<byte-string>`. If we did not provide the `equate:` option, then we would have to explicitly provide a function to transform "pointers to characters" into `<byte-string>`. The `equate:` option lets us take advantage of all of the predefined functions for `<string>`, which includes transformation into other string types.

CHAPTER

EIGHT

OTHER FILE OPTIONS

There are a few other options that may be specified within an `#include` clause, but which do not fit into any of the above categories. These options are `define:`, `undefine:`, `seal-functions:` and `read-only:`.

The `define:` and `undefine:` options control the C preprocessor definitions which will be implicitly defined during parsing of the header files. If you specify neither of these options, Melange will use a default set of definitions which correspond to those used by a typical C compiler for the machine you are running on. The define options takes a string containing a single C token and an optional string or integer literal, which will be used as the expansion. (If no literal is specified, the token will be expanded to `1`.) The `undefine:` removes one or more of the default definitions. You might, for example, use:

```
define interface
   #include "gc.h",
      define: {"PMAX", "BSD_VERSION" => "4.3"},
      undefine: {"HPUX"};
end interface;
```

The `seal-functions:` option controls whether the various imported functions and slot accessors will be sealed or open. By default, functions are sealed, but you may explicitly specify this by using `seal-functions: sealed` or reverse it by using `seal-functions: open`. Melange does not support the Creole's `inline` sealing option as this is handled with the `inline-functions:` option instead.

The `inline-functions:` option specifies how functions should be inlined. It may have values of `inline`, `inline-only`, `may-inline` or `not-inline`.

The `read-only:` option specifies whether setter functions should be defined for slot and object accessors. They will be defined by default, but if you specify `read-only: #t`, no setters will be defined.

The effects of the `seal-functions:`, `inline-functions:` and `read-only:` options can be modified for particular container types. We will explain how to do this in a later section.

CHAPTER

NINE

FUNCTION CLAUSES

Imported functions can be easily invoked, in almost every case, without any additional declarations. However, by exerting explicit control over argument handling, the interfaces to some functions may be made cleaner. This control is exerted via function clauses. The primary purpose of these clauses is to specify additional type information for specific parameters or to specify alternative argument passing conventions. For example, if we had two alternate `read-integers` functions with the following declarations:

```
int ReadInts1(int **VectorPtr);  /* result is a count of integers */
int *ReadInts2(int *Count);      /* result is a vector of  integers */
```

we might use the following interface definition:

```
define interface
   #include "readints.h",
      rename: {"int *" => int-vector};
   function "ReadInts1",
      output-argument: 1;
   function "ReadInts2" => Read-Integers-Vector,
      output-argument: Count;
end interface;
```

This would produce two functions, both of which take 0 arguments but return two values. The first would return an `<integer>` following by an `<int-vector>`, while the second would return the `<int-vector>` first and the `<integer>` second.

```
let (count :: <integer>, values :: <int-vector>)
   = Read-Ints1();
let (values :: <int-vector>, count :: <integer>)
   = Read-Integers-Vector();
```

The function clause consists of a function name (which is a string), an optional renaming (as illustrated above), and an optional sequence of "options". The options include the following:

`inline:` specifies whether or not the resulting method should be inlined. Possible values are `inline`, `inline-only`, `may-inline` and `not-inline`. The default is to not specify an inlining adjective.

`seal:` specifies whether the resulting method should be sealed. Possible values are sealed or open, and the default is taken from the value specified in the initial file clause. (The "default default" is sealed.)

`equate-result:` overrides the default interpretation of the result type. The named type is assumed to be fully defined.

`map-result:` specifies that `import-value` should be called to map the result value to the named type.

`ignore-result:` specifies that the functions result value should be ignored, just as if the function had been declared `void`. Although you may specify any boolean literal, the only meaningful value is `#t`.

map-error-result: specifies that `import-value` should be called to map the result value to the named type, but the actual result value should be ignored. This is useful for wrapping C APIs which return error codes that should signal conditions when an error occurs.

equate-argument: overrides the default interpretation of some argument's type. The argument may be specified by name or by position.

map-argument: specifies that `export-value` should be called to map the given argument into the named type. Again, the argument may be specified by position or by name.

input-argument: indicates that the specified argument should be passed by value. This is the default.

output-argument: indicates that the specified argument should be be treated as a return value rather than a "parameter". The effect is to declare that the C parameter will be passed by reference and that the reference variable need not be initialized to any object. This option assumes that the C parameter will have been declared as a "pointer" type, and will strip one `*` off of the argument type. Thus, if the parameter declaration specifies `int **`, the actual value returned will have the Dylan type corresponding to `int *`.

input-output-argument: indicates that the specified argument should be considered both an input argument and that its (potentially modified) value should be returned as an additional result value. The effect is similar to that of `output-argument` except that the reference variable will be initialized with the argument value.

The following (nonsensical) example demonstrates all of the options, as they might be applied to the functions:

```
extern struct object *bar(int first, int *second, struct object **third);
extern baz(char first, struct object *second);
```

```
define interface
   #include "demo.h";
   function "bar",
      seal: open,
      equate-result: <object>,
      map-result: <bar-object>,
      input-argument: first,    // passed normally
      output-argument: 2,       // nothing passed in, second result value
            // will be <integer>
      input-output-argument: third;    // passed in as second argument,
            // returned as third result
   function "baz" => arbitrary-function-name,
      seal: sealed,       // default
      ignore-result: #t,
      equate-argument: {second => <object>},
      map-argument: {2 => <baz-object>};
end interface;
```

CHAPTER

TEN

STRUCT AND UNION CLAUSES

"Struct clauses" and "union clauses" (referred to collectively as "container clauses") are used to specify naming in inclusion of class slots in exactly the same way that the options in the file clause control the handling of global definitions. Like the function clauses described above, they consist of the reserved word `struct` or `union`, a string which gives the full C name of the container declaration, an optional renaming, and a list of options. If we have a structure defined by

```
typedef struct cons {
   int index;
   struct object *head;
   struct cons *tail;
} cons_cell;
```

we could use the following interface definition:

```
define interface
   #include "cons.h";
   struct "struct cons" => <c-list>,
      superclasses: {<sequence>},
      prefix: "c-list-",
      name-mapper: identity-name-mapping;
end interface;
```

Valid options for container clauses include: `import:`, `prefix:`, `rename:`, `seal-functions:`, `inline-functions:`, `read-only:`, `equate:`, and `map:`. These options act like the equivalent options which may be specified in a file clause, but they apply to the slots of a single "class" rather than to globally defined objects. Options specified within a container clause override any global defaults that might have been specified in the `#include` clause. Container clauses also permit the `superclasses:` option described in *Specifying class inheritance*.

When a pointer type needs to be created for a struct, this can be done with the `pointer-type-name:` option in the struct clause:

```
define interface
   #include "event.h";
   struct "struct event",
      pointer-type-name: <event*>;
end interface;
```

Note: This will not currently change the name of generated pointer types due to using a pointer type in the header file. It will only add a `pointer-type-name:` option to the generated `C-struct` or `C-union` definition.

Although the recommended method for specifying a container type is to use the full C name (i.e. `struct foo`), you may also use an alias defined by a typedef. Thus, in the above example, you could have specified either `struct`

`cons` or `cons_cell`, with identical results.

CHAPTER

ELEVEN

POINTER CLAUSES

"Pointer clauses" modify the definitions of pointer declarations such as `int *` or `struct foo ***`, or vector declarations such as `char [256]`. Like all such clauses, they may be used to specify renamings for the classes. This is particularly useful for pointer types since they are not automatically assigned user-meaningful names. It also allows specification of the `superclasses:` option described in *Specifying class inheritance*. A typical use might be:

```
define interface
   #include "vec.h";
   pointer "int *" => <int-vector>,
      superclasses: {<c-vector>};
   pointer "struct person **" => <people>,
      superclasses: {<c-vector>};
   pointer "char [256]" => <fixed-string>;
end interface;
```

This clause is particularly useful for declaring pointer types to be subclasses of `<c-vector>` so that they can be indexed via `element`. (Note that this is not necessary for vector declarations, since they are automatically declared to be <c-vectors>.)

CHAPTER

TWELVE

CONSTANT CLAUSES

Constant clauses are used to override the values of constants specified in header files (i.e. `#define MAXINT 27`). The `value:` option, which is the only one supported, specifies a Dylan literal which will be taken as the value of the named constant. A typical use might be:

```
define interface
   #include "const.h";
   constant "MAXINT" => $maximum-fixed-integer,
      value: 9999999;
end interface;
```

CHAPTER

THIRTEEN

VARIABLE CLAUSES

Global variables declared within C header files are translated into "getter" functions which retrieve the value of the C variables and optional "setter" functions to modify those values. In effect, they are treated as slots of a "null object" – the getter function takes no arguments and returns the value of the variable, while the setter function takes a single value which will be the new value of the variable. Type mapping takes place for the arguments and results of these functions, just as it would for slot accessors.

Variable clauses support the following options:

`getter:` specifies a Dylan variable name which will be used to hold the getter function.

`setter:` specifies either a Dylan variable name which will be used to hold the setter function, or #f to indicate that there should be no setter function.

`read-only:` specifies whether the variable should be settable. `read-only: #t` is equivalent to `setter: #f`.

`seal:` specifies whether the getter and setter functions should be sealed. Possible values are `sealed` or `open`, and the default is taken from the `seal-functions:` option in the `#include` clause (or `sealed` if not specified there).

`map:` specifies the high-level type to which the variable should be mapped.

`equate:` specifies the low-level type to which the raw C value should be implicitly converted.

CHAPTER

FOURTEEN

LOW LEVEL SUPPORT FACILITIES

The high level functions for calling C routines or for accessing global variables are all built upon a relatively small number of built-in primitives which perform specific low-level tasks. You should seldom have any need to deal with these primitives directly, but they are nonetheless available should you need to make use of them.

Melange generates code using the Open Dylan C-FFI. The lower level `direct-c-ffi` isn't documented yet.

CHAPTER

FIFTEEN

KNOWN LIMITATIONS

Although mostly complete, the current implementation of Melange is missing a few elements which might be required for some applications. The following capabilities probably should be present, but are not yet supported:

- Callbacks.
- Function types. (It is, however, possible to import a function as a simple `<statically-typed-pointer>` and then manipulate it like any other object.)

Note: This section may be entirely incorrect.

CHAPTER

SIXTEEN

PROPOSED MODIFICATIONS

Although Melange seems to be fairly useful in its present form, we are currently considering a number of ways in which it may be made more useful. This section contains a brief discussion of several potential changes which may be implemented in the future.

Enumeration clauses

At present, there is no way to modify the default handling of a C enumeration declaration. It is clear that you might wish a mechanism to specify several different explicit options: prefixes for the enumeration constants; re-specification of constant values; and, of course, explicit `import:` and `exclude:` options.

Inheritance of "map" and "equate" options

There are some cases in which a set of types imported within one interface definition might be used extensively within another. In the present implementation, the two interface definitions would be handled independently and equivalences between types would not be recognized in the absence of explicit `equate:` options.

One proposed solution would involve the ability to explicitly "use" one interface definition within another. This would result in all identically named types being implicitly equated and all top-level `map:` options being inherited. The "use" clause could support roughly the same syntax as the "use" clauses in library and module definitions.In order to make this work, it would be necessary to assign arbitrary names to interface definitions. This would have the added benefit of making them more consistent with other standard Dylan definition forms.

If this change were implemented, a typical interface definition might look something like the following:

```
define interface date
   #include "date.h";
   use time, import: {"struct time"};
end interface date;
```

A less ambitious version might remain compatible with the current syntax by replacing the interface name with an "interface-name" option, which would default to the root of the file name. Thus,

```
define interface
   #include "date.h",
      interface-name: "date";
end interface;
```

would yield the same effect as the previous example.

Re-merging of the “equate:” and “map:” options

It has been pointed out that the current method of specifying low-level and high-level mappings, while sufficiently expressive, is somewhat verbose and confusing. It would therefore be good to find an alternative notation.

It has been suggested that definitions like:

```
define interface
   #include "dirent.h",
      equate: {"char *" => <c-string>},
      map: {"char *" => <byte-string>};
end interface;
```

might be replaced by something like:

```
define interface
   #include "dirent.h",
      equate-and-map: {"char *" => <c-string> => <byte-string>};
end interface;
```

or

```
define interface
   #include "dirent.h";
   transform "char *",
      low-level: <c-string>,
      high-level: <byte-string>;
end interface;
```

CHAPTER

SEVENTEEN

HISTORICAL NOTES

Melange was originally created as part of the Gwydion project at Carnegie Mellon University. As such, it was built to target the mindy interpreter and the d2c compiler.

It has since been converted to target the OpenDylan environment, using OpenDylan's C-FFI support.

Differences from Creole

It would be difficult to produce an exhaustive list of the differences between Creole and Melange. We can, however, include a brief examination of the most important incompatibilities between the two systems.

- Creole's `type:` options have been replaced by Melange's `equate:` and `map:` options.
- Creole's "access path" options have been replaced by `object-file:` and `mindy-include-file:`.
- The interface to `import-value` and `export-value` differ between the two systems.
- Melange does not inherit type mappings from other `define interface` forms.
- Creole does not import definitions from "recursively included" header files, even if they are referenced by definitions which are imported.
- Creole does not support C vectors or "sub-structures" as first class objects.
- Melange does not presently support callbacks, `export-temporary-value`, `<pascal-string>`, `with-stack-structure`, `with-stack-block`, or `alien-method`.
- Creole will never consider instances of two distinct statically typed pointer classes to be =, even if they refer to the same address.

CHAPTER

EIGHTEEN

COPYRIGHT

This document was originally authored by Robert Stockton.