APPENDIX B

The Outer Limits

As you've seen, GNU make can do some pretty incredible things, but I haven't seen very much that really pushes the limits of make 3.80 with its eval construct. In this exercise, we'll see if we can stretch it further than usual.

Data Structures

One of the limitations of make that occasionally chaffs when writing complex *makefile*s is make's lack of data structures. In a very limited way, you can simulate a data structure by defining variables with embedded periods (or even -> if you can stand it):

```
file.path = /foo/bar
file.type = unix
file.host = oscar
```

If pressed, you can even "pass" this file structure to a function by using computed variables:

```
define remote-file
  $(if $(filter unix,$($1.type)), \
    /net/$($1.host)/$($1.path),   \
    //$($1.host)/$($1.path))
endef
```

Nevertheless, this seems an unsatisfying solution for several reasons:

- You cannot easily allocate a instance of this "structure." Creating a new instance involves selecting a new variable name and assigning each element. This also means these pseudo-instances are not guaranteed to have the same fields (called *slots*).
- The structure exists only in the user's mind, and as a set of different make variables, rather than as a unified entity with its own name. And because the structure has no name, it is difficult to create a reference (or pointer) to a structure, so passing them as arguments or storing one in a variable is clumsy.

- There is no safe way to access a slot of the structure. Any typographical error in either part of the variable name yields the wrong value (or no value) with no warning from make.

But the remote-file function hints at a more comprehensive solution. Suppose we implement structure instances using computed variables. Early Lisp object systems (and even some today) used similar techniques. A structure, say file-info, can have instances represented by a symbolic name, such as file_info_1.

Another instance might be called file_info_2. The slots of this structure can be represented by computed variables:

```
file_info_1_path
file_info_1_type
file_info_1_host
```

Since the instance has a symbolic name, it can be saved in one or more variables (as usual, using recursive or simple variables is the choice of the programmer):

```
before_foo = file_info_1
another_foo = $(before_foo)
```

Elements of a file-info can be accessed using Lisp-like getters and setters:

```
path := $(call get-value,before_foo,path)
$(call set-value,before_foo,path,/usr/tmp/bar)
```

We can go further than this by creating a template for the file-info structure to allow the convenient allocation of new instances:

```
orig_foo := $(call new,file-info)
$(call set-value,orig_foo,path,/foo/bar)

tmp_foo := $(call new,file-info)
$(call set-value,tmp_foo,path,/tmp/bar)
```

Now, two distinct instances of <literal>file-info</literal> exist. As a final convenience, we can add the concept of default values for slots. To declare the file-info structure, we might use:

```
$(call defstruct,file-info,  \
  $(call defslot,path,),     \
  $(call defslot,type,unix), \
  $(call defslot,host,oscar))
```

Here, the first argument to the defstruct function is the structure name, followed by a list of defslot calls. Each defslot contains a single (*name*, *default value*) pair. Example B-1 shows the implementation of defstruct and its supporting code.

Example B-1. Structure definition in make

```
# $(next-id) - return a unique number
next_id_counter :=
define next-id
$(words $(next_id_counter))$(eval next_id_counter += 1)
endef
```

Example B-1. Structure definition in make (continued)

```
# all_structs - a list of the defined structure names
all_structs :=

value_sep := XxSepxX

# $(call defstruct, struct_name, $(call defslot, slot_name, value), ...)
define defstruct
  $(eval all_structs += $1)                                          \
  $(eval $1_def_slotnames :=)                                        \
  $(foreach v, $2 $3 $4 $5 $6 $7 $8 $9 $(10) $(11),                  \
    $(if $($v_name),                                                 \
      $(eval $1_def_slotnames          += $($v_name))                \
      $(eval $1_def_$($v_name)_default := $($v_value))))
endef

# $(call defslot,slot_name,slot_value)
define defslot
  $(eval tmp_id := $(next_id))
  $(eval $1_$(tmp_id)_name := $1)
  $(eval $1_$(tmp_id)_value := $2)
  $1_$(tmp_id)
endef

# all_instances - a list of all the instances of any structure
all_instances :=

# $(call new, struct_name)
define new
$(strip                                                          \
  $(if $(filter $1,$(all_structs)),,                             \
    $(error new on unknown struct '$(strip $1)'))                \
  $(eval instance := $1@$(next-id))                              \
  $(eval all_instances += $(instance))                           \
  $(foreach v, $($(strip $1)_def_slotnames),                     \
    $(eval $(instance)_$v := $($(strip $1)_def_$v_default)))     \
  $(instance))
endef

# $(call delete, variable)
define delete
$(strip                                                                \
  $(if $(filter $($(strip $1)),$(all_instances)),,                     \
    $(error Invalid instance '$($(strip $1))'))                        \
  $(eval all_instances := $(filter-out $($(strip $1)),$(all_instances))) \
  $(foreach v, $($(strip $1)_def_slotnames),                           \
    $(eval $(instance)_$v := )))
endef

# $(call struct-name, instance_id)
define struct-name
$(firstword $(subst @, ,$($(strip $1))))
endef
```

Example B-1. Structure definition in make (continued)

```
# $(call check-params, instance_id, slot_name)
define check-params
  $(if $(filter $($(strip $1)),$(all_instances)),,              \
    $(error Invalid instance '$(strip $1)'))                    \
  $(if $(filter $2,$($(call struct-name,$1)_def_slotnames)),,   \
    $(error Instance '$($(strip $1))' does not have slot '$(strip $2)'))
endef

# $(call get-value, instance_id, slot_name)
define get-value
$(strip                         \
  $(call check-params,$1,$2)    \
  $($($(strip $1))_$(strip $2)))
endef

# $(call set-value, instance_id, slot_name, value)
define set-value
  $(call check-params,$1,$2) \
  $(eval $($(strip $1))_$(strip $2) := $3)
endef

# $(call dump-struct, struct_name)
define dump-struct
{ $(strip $1)_def_slotnames "$($(strip $1)_def_slotnames)"      \
  $(foreach s,                                                  \
    $($(strip $1)_def_slotnames),$(strip                        \
    $(strip $1)_def_$s_default "$($(strip $1)_def_$s_default)")) }
endef

# $(call print-struct, struct_name)
define print-struct
{ $(foreach s,                                  \
    $($(strip $1)_def_slotnames),$(strip        \
    { "$s" "$($(strip $1)_def_$s_default)" })) }
endef

# $(call dump-instance, instance_id)
define dump-instance
{ $(eval tmp_name := $(call struct-name,$1))    \
  $(foreach s,                                  \
    $($(tmp_name)_def_slotnames),$(strip        \
    { $($(strip $1))_$s "$($($(strip $1))_$s)" })) }
endef

# $(call print-instance, instance_id)
define print-instance
{ $(foreach s,                                          \
    $($(call struct-name,$1)_def_slotnames),"$(strip    \
    $(call get-value,$1,$s))") }
endef
```

Examining this code one clause at a time, you can see that it starts by defining the function next-id. This is a simple counter:

```
# $(next-id) - return a unique number
next_id_counter :=
define next-id
$(words $(next_id_counter))$(eval next_id_counter += 1)
endef
```

It is often said that you cannot perform arithmetic in make, because the language is too limited. In general, this is true, but for limited cases like this you can often compute what you need. This function uses eval to redefine the value of a simple variable. The function contains two expressions: the first expression returns the number of words in next_id_counter; the second expression appends another word to the variable. It isn't very efficient, but for numbers in the small thousands it is fine.

The next section defines the defstruct function itself and creates the supporting data structures.

```
# all_structs - a list of the defined structure names
all_structs :=

value_sep := XxSepxX

# $(call defstruct, struct_name, $(call defslot, slot_name, value), ...)
define defstruct
  $(eval all_structs += $1)                              \
  $(eval $1_def_slotnames :=)                            \
  $(foreach v, $2 $3 $4 $5 $6 $7 $8 $9 $(10) $(11),      \
    $(if $($v_name),                                     \
      $(eval $1_def_slotnames          += $($v_name))    \
      $(eval $1_def_$($v_name)_default := $($v_value))))
endef

# $(call defslot,slot_name,slot_value)
define defslot
  $(eval tmp_id := $(next_id))
  $(eval $1_$(tmp_id)_name := $1)
  $(eval $1_$(tmp_id)_value := $2)
  $1_$(tmp_id)
endef
```

The variable all_structs is a list of all known structures defined with defstruct. This list allows the new function to perform type checking on the structures it allocates. For each structure, *S*, the defstruct function defines a set of variables:

```
S_def_slotnames
S_def_slotn_default
```

The first variable defines the set of slots for a structure. The other variables define the default value for each slot. The first two lines of the defstruct function append to all_structs and initialize the slot names list, respectively. The remainder of the function iterates through the slots, building the slot list and saving the default value.

Each slot definition is handled by `defslot`. The function allocates an id, saves the slot name and value in two variables, and returns the prefix. Returning the prefix allows the argument list of `defstruct` to be a simple list of symbols, each of which provides access to a slot definition. If more attributes are added to slots later, incorporating them into `defslot` is straightforward. This technique also allows default values to have a wider range of values (including spaces) than simpler, alternative implementations.

The `foreach` loop in `defstruct` determines the maximum number of allowable slots. This version allows for 10 slots. The body of the `foreach` processes each argument by appending the slot name to `S_def_slotnames` and assigning the default value to a variable. For example, our `file-info` structure would define:

```
file-info_def_slotnames := path type host
file-info_def_path_default :=
file-info_def_type_default := unix
file-info_def_host_default := oscar
```

This completes the definition of a structure.

Now that we can define structures, we need to be able to instantiate one. The `new` function performs this operation:

```
# $(call new, struct_name)
define new
$(strip                                                    \
  $(if $(filter $1,$(all_structs)),,                       \
    $(error new on unknown struct '$(strip $1)'))          \
  $(eval instance := $1@$(next-id))                        \
  $(eval all_instances += $(instance))                     \
  $(foreach v, $($(strip $1)_def_slotnames),               \
    $(eval $(instance)_$v := $($(strip $1)_def_$v_default)))  \
  $(instance))
endef
```

The first `if` in the function checks that the name refers to a known structure. If the structure isn't found in `all_structs`, an error is signaled. Next, we construct a unique id for the new instance by concatenating the structure name with a unique integer suffix. We use an at sign to separate the structure name from the suffix so we can easily separate the two later. The `new` function then records the new instance name for type checking by accessors later. Then the slots of the structure are initialized with their default values. The initialization code is interesting:

```
$(foreach v, $($(strip $1)_def_slotnames),               \
  $(eval $(instance)_$v := $($(strip $1)_def_$v_default)))
```

The `foreach` loop iterates over the slot names of the structure. Using `strip` around on the structure name allows the user to add spaces after commas in the `call` to `new`. Recall that each slot is represented by concatenating the instance name and the slot name (for instance, `file_info@1_path`). The righthand side is the default value computed from the structure name and slot name. Finally, the instance name is returned by the function.

Note that I call these constructs functions, but they are actually macros. That is, the symbol `new` is recursively expanded to yield a new piece of text that is inserted into the *makefile* for reparsing. The reason the `defstruct` macro does what we want is because all the work is eventually embedded within `eval` calls, which collapse to nothing. Similarly, the `new` macro performs its significant work within `eval` calls. It can reasonably be termed a function, because the expansion of the macro conceptually yields a single value, the symbol representing the new instance.

Next, we need to be able to get and set values within our structures. To do this, we define two new functions:

```
# $(call get-value, instance_id, slot_name)
define get-value
$(strip                          \
  $(call check-params,$1,$2)     \
  $($($(strip $1))_$(strip $2)))
endef

# $(call set-value, instance_id, slot_name, value)
define set-value
  $(call check-params,$1,$2) \
  $(eval $($(strip $1))_$(strip $2) := $3)
endef
```

To get the value of a slot, we simply need to compute the slot variable name from the instance id and the slot name. We can improve safety by first checking that the instance and slot name are valid strings with the `check-params` function. To allow more aesthetic formating and to ensure that extraneous spaces do not corrupt the slot value, we wrap most of these parameters in `strip` calls.

The set function also checks parameters before setting the value. Again, we strip the two function arguments to allow users the freedom to add spaces in the argument list. Note that we do not strip the slot value, because the user might actually need the spaces.

```
# $(call check-params, instance_id, slot_name)
define check-params
  $(if $(filter $($(strip $1)),$(all_instances)),,              \
    $(error Invalid instance '$(strip $1)'))                    \
  $(if $(filter $2,$($(call struct-name,$1)_def_slotnames)),,   \
    $(error Instance '$($(strip $1))' does not have slot '$(strip $2)'))
endef

# $(call struct-name, instance_id)
define struct-name
$(firstword $(subst @, ,$($(strip $1))))
endef
```

The `check-params` function simply checks that the instance id passed to the setter and getter functions is contained within the known instances list. Likewise, it checks that the slot name is contained within the list of slots belonging to this structure. The

structure name is computed from the instance name by splitting the symbol on the `@` and taking the first word. This means that structure names cannot contain an at sign.

To round out the implementation, we can add a couple of print and debugging functions. The following print function displays a simple user-readable representation of a structure definition and a structure instance, while the dump function displays the implementation details of the two constructs. See Example B-1 for the implementations.

Here's an example defining and using our `file-info` structure:

```
include defstruct.mk

$(call defstruct, file-info,    \
  $(call defslot, path,),       \
  $(call defslot, type,unix),   \
  $(call defslot, host,oscar))

before := $(call new, file-info)
$(call set-value, before, path,/etc/password)
$(call set-value, before, host,wasatch)

after := $(call new,file-info)
$(call set-value, after, path,/etc/shadow)
$(call set-value, after, host,wasatch)

demo:
        # before       = $(before)
        # before.path  = $(call get-value, before, path)
        # before.type  = $(call get-value, before, type)
        # before.host  = $(call get-value, before, host)
        # print before = $(call print-instance, before)
        # dump before  = $(call dump-instance, before)
        #
        # all_instances   = $(all_instances)
        # all_structs     = $(all_structs)
        # print file-info = $(call print-struct, file-info)
        # dump file-info  = $(call dump-struct, file-info)
```

and the output:

```
$ make
# before       = file-info@0
# before.path  = /etc/password
# before.type  = unix
# before.host  = wasatch
# print before = { "/etc/password" "unix" "wasatch" }
# dump before  = {  { file-info@0_path "/etc/password" } { file-info@0_type "unix" }
{ file-info@0_host "wasatch" } }
#
# all_instances   =  file-info@0 file-info@1
# all_structs     =  file-info
# print file-info = { { "path" "" } { "type" "unix" } { "host" "oscar" } }
# dump file-info  = { file-info_def_slotnames " path type host" file-info_def_path_
default "" file-info_def_type_default "unix" file-info_def_host_default "oscar" }
```

Also note how illegal structure uses are trapped:

```
$ cat badstruct.mk
include defstruct.mk
$(call new, no-such-structure)
$ make -f badstruct.mk
badstruct.mk:2: *** new on unknown struct 'no-such-structure'.  Stop.

$ cat badslot.mk
include defstruct.mk
$(call defstruct, foo, defslot(size, 0))
bar := $(call new, foo)
$(call set-value, bar, siz, 10)
$ make -f badslot.mk
badslot.mk:4: *** Instance 'foo@0' does not have slot 'siz'.  Stop.
```

Of course, there are lots of improvements that can be made to the code, but the basic ideas are sound. Here is a list of possible enhancements:

- Add a validation check to the slot assignment. This could be done with a hook function that must yield empty after the assignment has been performed. The hook could be used like this:

```
# $(call set-value, instance_id, slot_name, value)
define set-value
  $(call check-params,$1,$2)                                    \
  $(if $(call $(strip $1)_$(strip $2)_hook, value),             \
    $(error set-value hook, $(strip $1)_$(strip $2)_hook, failed)) \
  $(eval $($(strip $1))_$(strip $2) := $3)
endef
```

- Support for inheritance. A `defstruct` could accept another `defstruct` name as a superclass, duplicating all the superclass's members in the subclass.
- Better support for structure references. With the current implementation, a slot can hold the ID of another structure, but accessing is awkward. A new version of the `get-value` function could be written to check for references (by looking for *`defstruct@number`*), and perform automatic dereferencing.

Arithmetic

In the previous section, I noted that it is not possible to perform arithmetic in `make` using only its native features. I then showed how you could implement a simple counter by appending words to a list and returning the length of the list. Soon after I discovered the increment trick, Michael Mounteney posted a cool trick for performing a limited form of addition on integers in `make`.

His trick manipulates the number line to compute the sum of two integers of size one or greater. To see how this works, imagine the number line:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Now, notice that (if we could get the subscripts right), we could add, say 4 plus 5, by first taking a subset of the line from the fourth element to the end then selecting the fifth element of the subset. We can do this with native make functions:

```
number_line = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
plus = $(word $2, $(wordlist $1, 15, $(number_line)))
four+five = $(call plus, 4, 5)
```

Very clever, Michael! Notice that the number line starts with 2 rather than 0 or 1. You can see that this is necessary if you run the plus function with 1 and 1. Both subscripts will yield the first element and the answer must be 2, therefore, the first element of the list must be 2. The reason for this is that, for the word and wordlist functions, the first element of the list has subscript 1 rather than 0 (but I haven't bothered to prove it).

Now, given a number line, we can perform addition, but how do we create a number line in make without typing it in by hand or using a shell program? We can create all numbers between 00 and 99 by combining all possible values in the tens place with all possible values in the ones place. For example:

```
make -f - <<< '$(warning $(foreach i, 0 1 2, $(addprefix $i, 0 1 2)))'
/c/TEMP/Gm002568:1:  00 01 02  10 11 12  20 21 22
```

By including all digits 0 through 9, we would produce all numbers from 00 to 99. By combining the foreach again with a hundreds column, we would get the numbers from 000 to 999, etc. All that is left is to strip the leading zeros where necessary.

Here is a modified form of Mr. Mounteney's code to generate a number line and define the plus and gt operations:

```
# combine - concatentate one sequence of numbers with another
combine = $(foreach i, $1, $(addprefix $i, $2))

# stripzero - Remove one leading zero from each word
stripzero = $(patsubst 0%,%,$1)

# generate - Produce all permutations of three elements from the word list
generate = $(call stripzero,                              \
             $(call stripzero,                            \
               $(call combine, $1,                        \
                 $(call combine, $1, $1))))

# number_line - Create a number line from 0 to 999
number_line := $(call generate,0 1 2 3 4 5 6 7 8 9)
length      := $(word $(words $(number_line)), $(number_line))

# plus - Use the number line to add two integers
plus = $(word $2,                                         \
         $(wordlist $1, $(length),                        \
           $(wordlist 3, $(length), $(number_line))))

# gt - Use the number line to determine if $1 is greater than $2
gt = $(filter $1,                                         \
```

```
        $(wordlist 3, $(length),                          \
          $(wordlist $2, $(length), $(number_line))))


all:
        @echo $(call plus,4,7)
        @echo $(if $(call gt,4,7),is,is not)
        @echo $(if $(call gt,7,4),is,is not)
        @echo $(if $(call gt,7,7),is,is not)
```

When run, the *makefile* yields:

```
$ make
11
is not
is
is not
```

We can extend this code to include subtraction by noting that subscripting a reversed list is just like counting backwards. For example, to compute 7 minus 4, first create the number line subset 0 to 6, reverse it, then select the fourth element:

```
number_line := 0 1 2 3 4 5 6 7 8 9...
1through6   := 0 1 2 3 4 5 6
reverse_it  := 6 5 4 3 2 1 0
fourth_item := 3
```

Here is the algorithm in make syntax:

```
# backwards - a reverse number line
backwards := $(call generate, 9 8 7 6 5 4 3 2 1 0)

# reverse - reverse a list of words
reverse    = $(strip                                    \
               $(foreach f,                             \
                 $(wordlist 1, $(length), $(backwards)), \
                 $(word $f, $1)))

# minus - compute $1 minus $2
minus      = $(word $2,                               \
               $(call reverse,                        \
                 $(wordlist 1, $1, $(number_line))))

minus:
        # $(call minus, 7, 4)
```

Multiplication and division are left as an exercise for the reader.