DOSS: a Dylan Object Storage System

Eliot Miranda
The Harlequin Group

1. Introduction

The **Dylan Object Storage System** (or **DOSS** for the lazy) is a simple mechanism for storing arbitrary objects. The DOSS *dumper* traverses a graph of objects and encodes the objects as a sequence of bytes. These bytes are later interpreted by the DOSS *loader* which reconstructs an isomorphic object-graph in which cyclic structures and shared references are preserved. The sequence of bytes can be stored in a file, providing a simple persistent object-storage model. Since the loader constructs a ***new*** object-graph the system does not preserve object-identity, and hence is inferior to an object-oriented database. However, for many applications it is sufficient means for saving graphs of objects produced by one Dylan program which are to be used by a subsequent run of a (potentially different) Dylan program. DOSS provides ways to control what slots in objects should be stored, and to decide what objects should be stored by reference (e.g. classes are typically not stored as a structure but as a name).

2. Concepts

DOSS is a simple mechanism for storing arbitrary Dylan objects that need to persist across program executions. It is similar to a number of Object-Storage Systems (OSSs) found in other Object-Oriented environments, such as Binary Object Storage System (BOSS) in Smalltalk-80, “pickles” in Modula-3, and FASt Load (FASL) in Common Lisp. The problem these systems (including DOSS) solve is to traverse a possibly cyclic graph of objects and produce a stream of bytes which encodes this graph in such a way that a structurally identical graph of objects can be built by interpreting the byte stream.

2.1 The Basic Algorithm

The basic storage algorithm is as follows. Starting from a root object, a graph of objects is traversed. Each time a newly-encountered object is found it is assigned an identifier, and a decision is made as to whether the object should be traversed (and later reconstructed) or referenced (and later looked up). If the object is to be traversed then bytecodes that encode sufficient information to instantiate an analogous object are output. The bytecodes typically encode the object’s identifier, class, slots, and the number of repeated slots (e.g. the number of elements in a vector). The traverse then continues with the object’s slots and repeated slots, appending encodings of the objects contained in these slots. If the object is to be referenced then bytecodes encoding its identifier and name are output. When the traverse encounters an object already seen bytecodes encoding the identifier assigned that object are output and the traverse of the object is not repeated.

Certain objects don't have a slot-like representation, e.g. immediate values such as integers and characters. These objects values are appropriately encoded.

The basic reconstruction algorithm is to interpret the byte stream, instantiating objects when indicated by the byte codes, and assigning subsequent objects encoded in the byte stream to the slots of instantiated objects. Referenced objects are looked up in appropriate directories. By maintaining a table from identifiers to objects, references to previously instantiated or referenced objects can be resolved by indexing the table. At the end of the reconstruction algorithm a new graph of objects has been built, isomorphic to the object graph originally traversed, in which cyclic structure and shared references are preserved.

2.2 Necessary and/or Useful Extensions

There are a number of elaborations that can be made to the dumping system to make it more useful:

- being able to specify that only a subset of an object's slots are saved. e.g. an object may cache information in a slot. Not saving that slot may save space in the graph dump, and time during loading and dumping.
- being able to substitute a specific value for a slot during dumping. e.g. a file accessor holding a file descriptor in a slot. When the file accessor is reconstructed, possibly in another program, the file is logically closed. The value of its file descriptor slot should not be an obsolete file descriptor, but rather a sensible invalid value.
- being able to specify that certain ordinary objects (not just classes) are to be stored by reference. e.g. a structured graphic object may contain (references to) fonts. A naive traversal of the object would include the fonts in the dump. More appropriate would be to store the names of the fonts in the dump, and on loading, look up those names in e.g. a font directory.

An object-oriented design that allows the programmer such fine-grained control is to split the dumping process between a *dumper* object whose job it is to traverse the object graph, maintain object identifiers, and produce the byte stream, and a *policy* object whose job it is to decide which mechanism to use for objects, which to store by reference, what slots to be dumped etc. The DOSS system uses this design.

There are a number of facilities that are necessary during object graph reconstruction, without which, OSSs have limited use.

- to allow post-processing on reconstructed objects after the construction process. For example, hash tables may need to be rehashed after reconstruction. It is occasionally impossible to hash elements into a table during reconstruction because the hash function depends on the structure of the object being added. In the presence of circularities this hash value could only be computable after all objects have been reconstructed.

- to allow an object to be reconstructed even though its class definition is different in the loading system to that in the dumping system. The loader should be able to ignore values for non-existent slots, and supply defaults for un-stored slots.
- to allow an object to be reconstructed as the result of evaluating an arbitrary expression. In some systems first-class objects such as closures may not be constructed via normal instantiation and slot assignment, but may be built from appropriate primitives such as curry & compose.

2.3 Dylan-Specific Object Storage

There are three areas of the algorithm specific to Dylan, object structure, object references, and object identity.

Firstly, a Dylan object's structure can be thought of as the set of its instance-allocated slots. *Named* slots are accessed through getter and setter generic functions. *Repeated* slots are accessed via integer indices using the element and element-setter generic functions. Currently only named slots can be unbound (repeated slots being initialised with some default fill value).

DOSS lets the client control object dumping via its slots by allowing the client to specify which slots are to be dumped (although a default is provided), see section 3.2.1. DOSS allows the client to control the contents of slots to be dumped by overriding the default slot access through getters, and by handling unbound slots, see section 3.2.2.

Secondly, Dylan some objects are named (and hence made potentially globally accessible) through module variables. A module variable has a unique identifier string within a module, and a module has a unique name within a uniquely named library. So DOSS identifies a module variable by the triple (***module variable name***, ***module name***, ***library name***). DOSS provides facilities to look up objects in the set of module variables and allows the client to override the default encoding of ***module variable name***, see section 3.2.4.

Lastly, although an OSS does not preserve object identity (an OSS is essentially a graph *copier*), it does preserve object identity ***within*** an object graph. That is, any pair of references within the graph to objects which are **id?** will reference objects that are **id?** on reconstruction. The DOSS dumper is however free to dump objects the Dylan language considers computationally equivalent[1] in any way that permits reconstruction of a computationally equivalent object. So instances of **<character>**, **<integer>** and **<float>** are stored by value. On reconstruction the bytes are interpreted to re-compute an equivalent object.

1. see == in the Dylan Interim Reference Manual, p 39.

3. The Client Interface

3.1 Basic Dumping and Loading of Objects

To dump an object one instantiates a **<doss-dumper>** with an appropriate ***binary*** stream and optional policy and then uses **store-object**, e.g.:

```
let my-file = write-stream-over(as(<pathname>,"an-object.doss"), element-type: #"byte");
let dumper = make(<doss-dumper>, stream: my-file);

store-object(an-object,dumper);
close(my-file)
```

To load an object one instantiates a **<doss-loader>** with an appropriate stream and then uses **fetch-object**:

```
let my-file = read-stream-over(as(<pathname>,"an-object.doss"), element-type: #"byte");
let loader = make(<doss-loader>, stream: my-file);
let object = loader.fetch-object;

close(my-file);
object
```

There is no need for a client to write any code for their objects to make this work, e.g. no equivalent of CLOS's make-load-form. The default mechanism is to dump information identifying an object's class and slot setters, and then to dump each of the objects in its slots. However there are facilities for overriding the default mechanism.

3.2 Altering the Default Dumping Behaviour

The following sections explain how to alter DOSS's default dumping behaviour for specific objects through the use of policy objects. Clients using such facilities in the LispWorks Dylan emulator should be aware of the caveats in section 5.1, DOSS and the Emulator. The caveats are minor and boil down to specialising **<emulator-doss-policy>** instead of **<basic-doss-policy>**.

3.2.1 Controlling Slots to be Stored

DOSS uses the method **doss-dumpable-slots** to choose the slots to be traversed (and hence, the objects referenced by those slots to be dumped) during dumping. The method returns a sequence

and slots are dumped in the order defined by that sequence. Note that the method specialises on a policy object. The default method returns all an object's instance-allocated slots:

```
define method doss-dumpable-slots (a-class :: <class>, dp :: <doss-policy>) ⇒ s :: <sequence>;
    choose( method (slot-descriptor) slot-descriptor.allocation == #"instance" end,
            a-class.slot-descriptors[1])
end
```

A client can easily filter out unwanted slots from the default set, e.g.[2]

```
define method doss-dumpable-slots (a-class :: <my-class>, dp :: <doss-policy>) ⇒ s :: <sequence>
    choose( method (slot-descriptor) slot-descriptor.getter ~== cached-info end,
            next-method())
end
```

Note that this information is cached by the dumper, so **doss-dumpable-slots** is called on a class the first time an instance of that class is encountered during dumping, ***not*** every time an instance of the class is dumped.

Some classes have integer-indexable slots (e.g. **<vector>** & **<string>**). These slots we term *repeated* slots. At the moment Dylan has no reflective facilities for dealing with repeated slots. So DOSS uses a method **has-repeated-slots?** to identify the classes that have such slots. Currently there's no fine-grain control over repeated slots (such as being able to specify a sub-range of the valid indices). Simply note that if you hack-up a class with repeated slots it must return true from **has-repeated-slots?** if DOSS is to store the contents of its instances' repeated slots, e.g.:

```
define method has-repeated-slots? (a-class :: <class>)           ⇒ well? :: <boolean>      #f   end method;
define method has-repeated-slots? (class-string == <string>)    ⇒ well? :: <boolean>;     #t   end method;
define method has-repeated-slots? (class-vector == <vector>)    ⇒ well? :: <boolean>;     #t   end method;
// etc....
define method number-of-repeated-slots (seq == <sequence>) ⇒ n :: <integer>;      seq.size end method;
```

1. The DIRM does not include slot descriptors. Some analogous mechanism is required to support the default implementation of doss-dumpable-slots.

2. A client can supply an explicit slot sequence, but this practice is error prone. If slots are added to the class definition it is easy to forget to change the doss-dumpable-slots method to supply the new slots.

3.2.2 Controlling the Objects Stored for Slots

DOSS uses the method **doss-slot-value** to get the object to be stored for a particular slot. The default behaviour is to return the value in that slot:

```
define method doss-slot-value (getter :: <function>, obj, dd :: <doss-dumper>) ⇒ o :: <object>;
    if (slot-initialized?(getter,obj))
        getter(obj)
    else
        dd.unbound-proxy
    end
end method
```

The method can be specialised to provide specific values for specific objects. E.g. an **<external-stream>** is a stream with some connection to the outside world. When it is dumped it needs to be dumped in a closed state (since the connection to the outside world won't necessarily have been made when the **<external-stream>** is loaded). It has a slot **open?** containing a flag, **#t** if the stream is open, and **#f** if closed. Here is its doss-slot-value method which arranges that the slot is always dumped with a value of **#f**:

```
define method doss-slot-value (getter == open?, obj :: <external-stream>, dd :: <doss-dumper>)
                ⇒ closed :: <boolean>;
    #f
end method
```

Unbound slots are handled by DOSS substituting a special marker for unbound slots, **a-doss-dumper.unbound-proxy**, which the DOSS loader interprets appropriately. A client is free to use this value for any named slot, e.g.

```
define method doss-slot-value (getter == file-descriptor, obj :: <external-file-accessor>, dd :: <doss-dumper>)
                ⇒ unbound :: <object>;
    dd.unbound-proxy
end method
```

DOSS allows one to control the values stored for repeated slots through the **doss-repeated-slot-element method**, which defaults to a call of **element**:

```
define method doss-repeated-slot-element (obj, i :: <integer>, dd :: <doss-dumper>) ⇒ o :: <object>;
    obj[i]
end method;
```

Note that if you want to control slot values during dumping on the basis of the policy you're using, you can do so since the dumper holds onto the current policy in its policy slot.

3.2.3 Using Policy Objects

A policy object decides which objects are to be dumped specially (i.e. not dumped via the default mechanism of dumping information identifying an object's class and slot setters, and then dump-

ing each of the objects in its slots) and how. **<basic-doss-policy>** is the default policy used by **<doss-dumper>**. Each time a DOSS dumper encounters a new object to be stored it allows its policy to store the object in some special way using **put-specially**. If the policy 'decides' to store the object in some way it does so using one of the mechanisms in the **<doss-dumper>** policy protocol, see section 6.1.2, and **put-specially** returns **#t**. If the policy decides the object is not special **put-specially** returns **#f** and the dumper uses the default traversing storage mechanism.

<doss-policy> and **<basic-doss-policy>** provide the following implementations of **put-specially** to arrange that classes are stored as a module variable name and a sequence of setters, and functions are stored as module variable names:

```
define method put-specially (obj, policy :: <doss-policy>, dd :: <doss-dumper>) ⇒ object-dumped? :: <boolean>;
    #f
end method;

define method put-specially (obj :: <function>, policy :: <basic-doss-policy>, dd :: <doss-dumper>)
                ⇒ object-dumped? :: <boolean>;
    put-reference(obj, dd);
    #t
end method;

define method put-specially (obj :: <class>, policy :: <basic-doss-policy>, dd :: <doss-dumper>)
                ⇒ object-dumped? :: <boolean>;
    put-class-description(obj, dd);
    #t
end method;
```

By adding **put-specially** methods to **<basic-doss-policy>** or one's own subclass of **<basic-doss-policy>** one can tailor DOSS's dumping behaviour appropriately.

By splitting the making of policy decisions off from the dumping process its possible to dump a given object graph in two completely different ways in the same program. This is done by using two different policy objects to mediate each dump respectively. Why is this useful? An example is the dumping of code for the virtual machine. When a piece of code is compiled for the virtual machine the actual code vector is dumped in such a way that the code can be loaded into a dylan program and executed. Such a process will involve naming variables to conform with the loading system. At the same time the code may need to be dumped in a different form to the derived database.

3.2.4 Controlling the Names Used for Objects Stored by Reference

When **put-reference** or **put-class-description** store an object as a module variable they do so using **locate-variable-via-policy** and **encode-variable** from the variable search protocol, see section 6.4. If **locate-variable-via-policy** finds a module variable referencing an object it calls **encode-variable** which decides how to encode the module variable name. By default **encode-variable** returns three values, the module variable name as a symbol, the module name as a symbol, and the library name as **#f**

(since libraries are not yet implemented). If appropriate one can add methods to **encode-variable** to provide more compact encodings, see see section 7.2 for an example.

To allow policy objects to rename objects **locate-variable-via-policy** specialises both on the object being looked-up and the current policy. The default method simply defers to **locate-variable**, which examines information on the state-of-play of module variables. By adding methods to **locate-variable-via-policy** that specialise on your own policies you can control the naming of objects as required.

3.3 Dumping and Modularisation

There is a tension between the module system, which exists to hide information, and DOSS, which attempts to ferret it out. Methods such as **doss-dumpable-slots** and **doss-slot-value** may need access to potentially un-exported getters and setters. To cope this one usually needs an extra module that imports doss and contains the module-specific doss code (example?).

3.4 Altering the Default Load Behaviour

The client interface for loading of objects is described above, see section 3.1. Internally DOSS loads, or more accurately, reconstructs objects by one of three mechanisms, looking up a module variable (see section 6.1.2 **put-reference**, **put-variable** & **put-class-description**), evaluating a function (see section 6.1.2 **put-apply**), or by instantiation and slot assignment (see section 6.1.2 **put-object**). If this last mechanism is used an object is created rather crudely, instantiated via **allocate**, which returns an uninitialized instance, and initialized by applying the setters that were dumped with an object's class.

For some objects this approach will not produce a well-formed object. For example, an **<object-table>** hashes objects within it according to their identity. Since objects are reconstructed on loading a loaded table's contents will have different identities and consequently the table's contents will not be correctly hashed. To allow objects to correct such reconstruction problems, after all objects have been loaded the generic function **post-load-cleanup** is applied to each reconstructed object. Objects can take whatever action is appropriate within their post-load-cleanup methods. post-load-cleanup is applied in reverse order of dumping. For example, the post-load-cleanup method on **<hashed-collection>** (a superclass of **<object-table>** and other hashed collections) is as follows:

```
define method post-load-cleanup (hash-table :: <hashed-collection>) ⇒ ();
    rehash!(hash-table);
end method
```

3.4.1 Using put-apply

The **put-apply** mechanism allows one to substitute for a dumped object the value of an arbitrary function call, evaluated by the loader. For example, consider wanting to dump a font which is available from some external font server. One would want to avoid dumping the actual font object because a) it will take considerable space in the doss file, and b) it may already be available in the loading system. One can use the **put-apply** mechanism to e.g. arrange to fetch the font from the font server on loading. Given the following plausible definitions:

```
define class <loadable-font> (<font>) ...
    slot font-name :: <string>, ...

define class <font-dumping-policy> (<basic-doss-policy>) end class;

define method load-font(name :: <string>) ⟹ (f :: <font>); ...
```

we could arrange that a font gets dumped as the result of calling load-font as follows:

```
define method put-specially(font :: <loadable-font>, policy :: <font-policy>, dd :: <doss-dumper>) ⟹ <boolean>;
    put-apply(font, dd, load-font, font.font-name);
    #t
end method;
```

The font will not be dumped to the file. In its stead, the name of the load-font function (found via variable search) is dumped, along with the font's name. On loading, load-font will get called with the font's name as its argument, and the value load-font returns will be substituted in pkace of the font in the reconstructed object-graph.

3.4.2 Using put-variable

The put-variable mechanism allows one to provide names for objects in the loaded system, for example for objects that may be unavailable in the dumping system, or for objects that need to be substituted-for (effectively by renaming). Extending the previous example, imagine that the load-font function is unavailable in the dumping system. One can use the put-variable mechanism, with a proxy object for the function to dump a reference to load-font, even though it is not present:

```
define constant $load-font-proxy = vector(); // unique object as proxy(could use a class of objects);

define method put-specially(font :: <loadable-font>, policy :: <font-policy>, dd :: <doss-dumper>) ⟹ <boolean>;
    put-apply(font, dd, $load-font-proxy, font.font-name); // use proxy in place of function
    #t
end method

define method put-specially(proxy == $load-font-proxy, p :: <font-policy>, dd :: <doss-dumper>) ⟹ <boolean>;
    put-variable(proxy, dd, #"load-font", #"font-module", #"font-library"); // dump proxy as reference to function
    #t
end method;
```

4. DOSS Internals

4.1 Dumping

to be written - its all straight-forwardly grubby stuff - you don't want to know about it anyway - alright, already, go look at the code sweet masochistic fool.

4.2 The Dumper/Policy Maker Split

To make DOSS more flexible, mechanism has been separated from policy. Essentially the split allows a given network of objects to be dumped in two different ways from the same running program. If policy were implemented by adding methods to dumpers then different policies could not co-exist in the same program. **<doss-dumper>** provides the mechanism (the various ways of dumping an object) and subclasses of **<doss-policy>** provide policies (the choice of mechanism). There are currently 4 dumping mechanisms, see <doss-dumper> Policy Protocol, section 6.1.2:

- dump an object as a structure, via **put-object**.
- dump an object as a module variable reference, via **put-reference** and **put-variable**. A module variable reference is a triple of (***module variable name***, ***module name***, ***library name***), represented by symbols. If **put-reference** is used, this triple is looked up using the Variable Search Protocol, section 6.4. If **put-variable** is used, the client supplies their own values for (***module variable name***, ***module name***, ***library name***). A client can alter the encoding of ***module variable name*** via **encode-variable**. Each object in the triple is dumped, most probably using the **put-object** mechanism.
- dump an object as a function call via **put-apply**. A function call is a function and a sequence of arguments
-

4.3 The Bytecodes

DOSS uses a bytecoded language to represent object graphs. Each bytecode is followed by an arbitrary, possibly empty, sequence of bytes that encode the information necessary to reconstruct a given object. DOSS bytecodes can be considered in groups, immediate object encodings, unique object encodings, value object encodings, generic object encodings, and miscellaneous encodings. To provide a compact representation of integers and characters the bytecode space is divided into quarters. Bytecodes 0 to 63 are used for all encodings except integers and characters. Bytecodes 64 to 127 are used to encode integers, bytecodes 128 to 191 are used for the characters, and bytecodes 192 to 255 are reserved for future use. Each bytecode or range of bytecodes is kept in class slots of <doss-io-manager>:

4.3.1 unique object encodings

Dylan integers, characters, floats, and symbols are effectively “unique” objects. At the language level its impossible to have two of any of these objects **=** to each other but not **= =** to each other. To reconstruct these objects DOSS can simply compute them.

Since integers are very common, DOSS tries to provide a compact representation for them For simplicity, characters are represented using the same scheme. As mentioned above bytecodes 64 to 127 are used to encode integers, and bytecodes 128 to 191 are used for the characters. The least significant 6 bits of these bytecodes encodes the number of subsequent bytes, in which the integer’s value, or the character’s code, is stored as a little-endian two’s complement value.

The routine **dump-int(integer :: <integer>, code :: <integer>, dd :: <doss-dumper>)** does the work. code is or’ed into the byte count **dump-int** computes for integer, so integers are dumped using the integer start code (64) dump-int(dd.integer-start). If integer is zero then the bytecode is zero.

```
   0-63 conventional codes for references to slot objects, classes, variables etc
64-127 integer codes, integer follows #bytes encoded in least 6 bits
128-191 character codes (same encoding as integers)
192-255 reserved (for other (tagged) immediate objects

// codes for () #t & #f
constant slot empty-list-code :: <integer>, init-value: 0;
constant slot true-code       :: <integer>, init-value: 1;
constant slot false-code      :: <integer>, init-value: 2;

// next encoded integer is 2's complement integer
constant slot integer-start   :: <integer>, init-value: 64;

// next encoded integer is ascii code of a character
constant slot character-start :: <integer>, init-value: 128;

// next encoded integer is id of loaded object
constant slot object-id-code  :: <integer>, init-value: 3;

// slot to be left unbound
constant slot unbound-code    :: <integer>, init-value: 4;

// object id followed by object's definition
constant slot object-code     :: <integer>, init-value: 5;
```

```
  // encodings of 'variable' references
constant slot class-code      :: <integer>, init-value: 6;
constant slot keyword-code    :: <integer>, init-value: 7;
constant slot symbol-code     :: <integer>, init-value: 8;
constant slot variable-code   :: <integer>, init-value: 9;
constant slot string-code     :: <integer>, init-value: 10;
constant slot apply-code      :: <integer>, init-value: 11;

  // to fix the circularity of the void element in tables being used as a key in object-ids
constant slot void-code       :: <integer>, init-value: 12;

  // for "value" objects (symbols strings) that need ids.
constant slot val-obj-id-code :: <integer>, init-value: 13;

  // run-length encoding
constant slot repeat-code     :: <integer>, init-value: 14;

  // specials; pair
constant slot pair-code       :: <integer>, init-value: 15;

  // the various float formats
constant slot float-code      :: <integer>, init-value: 16;
constant slot double-code     :: <integer>, init-value: 17;
constant slot extended-code   :: <integer>, init-value: 18;
```

4.4 Loading

not done (need load policies?)

5. Bugs, Limitations and Future Work

5.1 DOSS and the Emulator

For pragmatic reasons the emulator Dylan system over LispWorks does not implement a seamless Dylan object model. In particular

- keywords and symbols differ
- certain Dylan objects are represented by underlying CLOS objects, and hence **allocate** and specializers of type **<class>** don't work for these classes.
- **<object-table>**s are implemented in terms of a CLOS equivalent

To cope with these differences DOSS in the emulator uses some extra classes that override relevant behaviour and attempt to make it appear as if files produced by the emulator DOSS are the same as that produced by a native Dylan system. The extra classes are **<emulator-doss-dumper>**,

<emulator-doss-loader>, and **<emulator-doss-policy>**. Of these, only **<emulator-doss-policy>** is relevant to users since the other two classes are internal to DOSS

<emulator-doss-policy> inherits from **<basic-doss-policy>**, and fixes problems to do with **<class>** specializers not working for the built-in classes **<simple-object-vector>** and **<byte-string>**. If you're working in the emulator then you should arrange that your policies inherit from **<emulator-doss-policy>**, and any methods you want to add to **<basic-doss-policy>**, should be added to **<emulator-doss-policy>** instead. The curious are directed to section 7.1, Emulator Tables for the grimy details.

5.2 Miscellaneous problems

Currently there is no support for floats, this will be remedied RSN.

The system relies on slot-descriptors, which do not appear in the DIRM. Hence the handling of slot getters & setters may change at some later date.

Currently there is no generic support for closures. Even though closures are first class objects DOSS doesn't know how to traverse them or how to store references to their code.

Although theoretically classes should be storable (i.e. by traversal) it hasn't been done in practice.

Loading an object which, in the loading system, has lost setters will not work. There needs to be an interface whereby the loader can know which setters it should apply.

post-load-cleanup is applied in reverse order of dumping. If this significantly different from leaf-first then it should be changed.

6. Appendix A - DOSS Reference

6.1 <doss-dumper> Protocols

6.1.1 <doss-dumper> Client Protocol

make *class* :: subclass(<doss-dumper>)[1] #key *policy* :: <doss-policy> *stream* :: <writable-positionable-stream>
⇒ *dd* :: <doss-dumper> [Method]

Instantiates a **<doss-dumper>** with optional *policy* and *stream* objects. If *policy* is not supplied it defaults to an instance of **<basic-doss-policy>**. There is no default for *stream*. If supplied, *stream* must be a general instance of **<writable-positionable-stream>** and should have an **element-type:** of **#"byte"**.

store-object *object dd* :: <doss-dumper> ⇒ *object* [Method]

Causes *dd* to store a description of *object* in DOSS format on *dd*'s stream, preceded by a standard DOSS header. A convenience function equivalent to **put-header(dd); put-object(dd);**

policy *dd* :: <doss-dumper> ⇒ *policy* :: <doss-policy> [Method]

returns *dd*'s policy object. By default this is an instance of **<basic-doss-policy>**.

policy-setter *policy* :: <doss-policy> *dd* :: <doss-dumper> ⇒ *policy* :: <doss-policy> [Method]

assigns *dd*'s policy object. Can also be done via the **policy:** keyword argument to **make** above.

stream *dd* :: <doss-dumper> ⇒ *stream* :: <writable-positionable-stream> [Method]

returns *dd*'s stream object.

stream-setter *stream* :: <writable-positionable-stream> *dd* :: <doss-dumper>
⇒ *stream* :: <writable-positionable-stream> [Method]

assigns *dd*'s stream object. Can also be done via the **stream:** keyword to **make** above.

---

1. N.B. since each-subclass specializers are not yet implemented the make methods use singleton specializers.

6.1.2 <doss-dumper> Policy Protocol

This protocol defines the basic object storage mechanisms available in DOSS. Policies choose which mechanisms are used for particular objects using this protocol.

put-reference *object dd* :: <doss-dumper> ⇒ *object* [Method]

Causes *dd* to store *object* as a module variable reference on *dd*'s stream. The variable name used will be that found via the variable search protocol, see section 6.4.

On loading *object* will be reconstructed as the current value of the module variable in the loading system.

put-variable *object dd* :: <doss-dumper> *variable-name module-name library-name* ⇒ *object* [Method]

Causes *dd* to store *object* as a module variable reference on *dd*'s stream, using *variable-name module-name* and *library-name* to identify the variable. This mechanism allows policy objects to rename objects, since a different name that that found via the variabl search protocol can be substituted.

On loading *object* will be reconstructed as the current value of the module variable in the loading system.

put-class-description *class* :: <object> *dd* :: <doss-dumper> ⇒ *object* [Method]

Causes *dd* to store *class* on *dd*'s stream as a module variable reference, a boolean indicating if *class* has repeated slots, a count of the number of slot setters, and a sequence of slot setters. The *class* variable is explicitly typed <object> to allow the use of proxy objects, i.e. a proxy for a class can be dumped in some manner instead of the class itself.

On loading *class* will be reconstructed as the current value of the module variable in the loading system, and the sequence of setters will be used to assign values to the slots of loaded instances of *class*.

put-apply *object dd* :: <doss-dumper> *function* :: <object> #rest *args* ⇒ *object* [Method]

Causes *dd* to store *object* on *dd*'s stream as the result of evaluating *function* with *args*. The *function* variable is explicitly typed <object> to allow the use of proxy objects, i.e. a proxy for a class can be dumped in some manner instead of the class itself.

On loading *object* will be reconstructed by applying the reconstruction of *function* to the reconstruction of the *args* sequence.

put-object *object dd* :: <doss-dumper> ⇒ *object* [Method]

Causes *dd* to store *object* on *dd*'s stream as *object*'s class followed by each of the *object*'s slot contents as accessed via **doss-dumpable-slots**. If *object* has repeated slots then this is followed by a count of *object*'s repeated slots and the contents of *object*'s repeated slots as accessed via **doss-repeated-slot-element**.

On loading *object* will be reconstructed by calling **allocate**[1] on *object*'s class, and then assigning *object*'s named and repeated slots with the corresponding reconstructions of *object*'s slot contents.

put-object invokes the default mechanism for dumping objects.

put-header *dd* :: <doss-dumper> [Method]

Causes *dd* to store a **DOSS** header on *dd*'s stream which serves to identify the information on the stream as being in DOSS format. A header is automatically prepended when store-object is used to dump an object-graph. However, if an object-graph is to be dumped via one of the special mechanisms above (e.g. put-apply) then put-header must be used first to ensure a valid header appears on the stream.

6.1.3 <doss-dumper> Slot Access Protocol

doss-slot-value *getter* :: <function> *object dd* :: <doss-dumper> ⇒ *slot-contents* [Method]

used by *dd* to access the contents of a slot in an object being traversed during dumping. When *object* is reconstructed the slot will be assigned the reconstruction of *slot-contents* using *getter*'s corresponding setter. If the slot is to be left unbound on reconstruction **doss-slot-value** should return **dd.unbound-proxy**.

doss-repeated-slot-element *object index* :: <integer> *dumper* :: <doss-dumper> ⇒ *slot-contents* [Method]

used by *dd* to access the contents of a repeated slot in an object being traversed during store. When *object* is reconstructed its corresponding repeated slot will be assigned the reconstruction of *slot-contents* using **element-setter**. Unbound repeated slots are not supported.[2]

1. **allocate** is at a lower level than **make**, it returns an uninitialized instance.
2. because there is no way of instantiating an object with an uninitialized repeated slot.

6.2 <doss-policy> Protocols

6.2.1 <doss-policy> Slot Access Protocol

doss-dumpable-slots *class* :: <class> *policy*:: <doss-policy> ⇒ *slot-descriptors* :: <sequence> [Method]

used by a **<doss-policy>** to select which slots will have their contents dumped. Clients are free to specialize this method to customize slot dumping.

6.2.2 <doss-policy> Dumper Protocol

put-specially *object policy* :: <doss-policy> *dd* :: <doss-dumper> ⇒ *object-dumped?* :: <boolean> [Method]

called by a **<doss-dumper>** whenever it encounters a new object in the dumping process. If the method returns **#t** *dd* assumes *object* has been dumped in some special way by *policy*. If the method returns **#f** the *dd* dumps *object* using the default traversal mechanism (see section 6.1.2, **put-object**).

Policies should specialize this method to implement specific storage policy decisions for specific objects. Specializations of **put-specially** are expected to use one of the mechanisms in section 6.1.2, <doss-dumper> Policy Protocol.

locate-variable-via-policy *object policy* :: <doss-policy> ⇒ *variable-name module-name library-name* [Method]

used by a **<doss-dumper>** to look-up a module variable name for an object being stored via **put-reference** or **put-class-description**. The default method simply calls **locate-variable**, see section 6.4, below.

6.3 <doss-loader> Protocol

load-doss-stream *stream-or-path-string* ⇒ *object* [Method]

Simple utility that takes a stream over, or a string that is the pathname of a file containing, an object encoded in **DOSS** format , builds a **<doss-loader>** and returns the object reconstructed by the loader. See **fetch-object** below. **load-doss-stream** takes care to correctly close the stream even in the presence of errors.

make *class* == <doss-loader> #key *stream* :: <readable-positionable-stream> ⇒ *dd* :: <doss-loader> [Method]

Instantiates a **<doss-loader>** with an optional *stream* object. There is no default for *stream*. If supplied, *stream* must be a general instance of **<readable-positionable-stream>** and should have an **element-type:** of **#"byte"**.

fetch-object dl :: <doss-loader> ⟹ *object* [Method]

Causes *dl* to interpret the byte stream in its *stream* object and reconstruct the object encoded by that byte stream. The byte stream should begin with an appropriate DOSS header (which is supplied by a **<doss-dumper>**). *dl* checks version information in the header and complains if it smells a rat.

stream *dl* :: <doss-loader> ⟹ *stream* :: <readable-positionable-stream> [Method]

returns *dl*'s stream.

stream-setter *stream* :: <readable-positionable-stream> *dl* :: <doss-loader> [Method]

assigns *dl*'s stream. *stream* should have an **element-type:** of **#"byte"**.

6.4 Variable Search Protocol

locate-variable *object* ⟹ *variable-name* :: <symbol> *module-name* :: <symbol> *library-name* :: <symbol> [Method]

used by a **<doss-policy>** to look-up a module variable name for an object being stored via **put-reference** or **put-class-description**. Clients should not specialize this method, rather they should specialize **encode-variable**.

module-name *module* :: <module> ⟹ *module-name* :: <symbol> [Method]

returns *module*'s name. N.B. in the emulator **<module>** is currently called **<translator-module>**.

encode-variable *object module* :: <module> *library-name* :: union(<symbol>,singleton(#f)) *variable-name* :: <symbol>
⟹ variable-*encoding module-name* :: <symbol> *library-name* :: <symbol> [Method]

used by locate-variable to return an encoding of a module variable referencing object. The default method returns three values, *variable-name*, *module-name* and *library-name*. Clients can specialize on *module* to provide more compact encodings for module variables, in which case an alternative value for *variable-name* should be returned along with *module-name* and *library-name*, e.g. an integer.

variable-value *variable-encoding module* :: union(<symbol>,<module>) *library-name* :: <symbol> [Method]

used by a **<doss-loader>** to resolve a module variable. The default method requires that *variable-encoding* is a symbol, and returns the value of the corresponding module variable. Clients can specialize on *module* to decode more compact encodings provided by **encode-variable**. If *variable-encoding* is not recognised by such specialised method then they should call **next-method** to allow the default method to resolve the reference. N.B. in the emulator **<module>** is currently called **<translator-module>**.

7. Appendix B - Examples

7.1 Emulator Tables

For efficiency reasons **<object-table>**s in the emulator are hacked to use LispWorks' underlying CLOS equivalent. However, it is still possible to dump emulator **<object-table>**s as if they were represented as DylanWorks **<object-table>**s by using the slot access protocols, see section 6.1.3 & Section 6.2.1 on page 15. A standard DylanWorks **<object-table>** holds the number of elements in its **tally** slot, and its key-value pairs in its **elements** slot. **elements** is a vector containing key-value pairs in adjacent slots. Unused slots contain **<table>**'s void element.

In the emulator **<object-table>** has two extra slots, **table-table** containing a lisp eql table, and **table-values** containing a lisp list of values (for efficient iteration). Further, no effort is made to keep the **tally** and **elements** slots up-to-date.

The problem is to dump the hacked **<object-table>** as if it were represented as a standard Dylan-Works **<object-table>**. The solution is to

- not dump the **table-table** and **table-value** slots
- dump faked-up values for the **tally** and **elements** slots

Here's the code:

```
Module:     emulator-doss
Synopsis:   Dylan Object Storage System; Emulator Table DOSS Support

/* Frig for emulator id tables
<lisp-object-table> has a table-table slot that holds onto a LispWorks eq-table. We intercept the access of this slot
to copy across values from the LispWorks table into our vector of elements, etc.

First ensure that table-table and table-values are eliminated from the set of dumped slots. (we don't want them to
be dumped, they are the CLOS wrinkles we want to smooth out. */

define method doss-dumpable-slots (class == <object-table>, policy :: <doss-policy>) ⇒ slots :: <sequence>;
    choose(method (desc)
                (desc.slot-getter ~== table-table) & (desc.slot-getter ~== table-values)
            end,
            next-method())
end method;

/* Next frig access to the tally and elements slots.  For tally we simply return the size. */

define method doss-slot-value(getter == tally, obj :: <object-table>, dd :: <doss-dumper>) ⇒ o :: <object>;
    obj.size
end method;
```

```
/* For elements we make a fake elements vector and copy our key-value pairs into it and then return this fake
vector. */
define method doss-slot-value(getter == elements, obj :: <object-table>, dd :: <doss-dumper>) ⟹ o :: <object>;
    // fake up a suitable elements vector
    let i= 0;
    let s = 8;
    while (s <= (obj.size * 2)) s := s + s; end; // ensure s a large enough power-of-two
    let fake-elements = make(<vector>, size: s, fill: obj.void-element);

    do(  method(key) fake-elements[i] := key; fake-elements[i + 1] := obj[key]; i:= i+ 2; end,
        obj.key-sequence); // copy the table's key-value pairs into the fake elements vector
    fake-elements
end method;

/* On loading all we need to do is to initialize the table (which initializes table-table & table-values) and copy the
key-value pairs back into table-table. */

define method post-load-cleanup(obj :: <object-table>)
    let index = 0;
    let limit = obj.tally * 2;
    let vec   = obj.elements;

    obj.initialize;              // this creates the underlying CLOS objects in table-table & table-values
    while (index < limit)
        obj[vec[index]] := vec[index + 1];
        index := index + 2;
    end;

    obj
end method;
```

7.2 IDVM Variable Encodings

```
Module:       IDVM-doss
Language:     infix-dylan
Synopsis:     DOSS interface for IDVM (to provide compact encoding of IDVM code under DOSS)
Author:       Eliot Miranda
Copyright: (c) 1994, The Harlequin Group Limited, All rights reserved.

define constant bytecodes-to-opcodes = vector(
        idvm-return, idvm-return-false, idvm-return-lit, idvm-return-loc,

        idvm-call-with-res, idvm-call-with-res-returning, idvm-call-0, idvm-call-0-returning,

    // etc, etc, for all 128 opcode methods.....

        idvm-mv-bind,
        idvm-process-keys
    );

define method encode-module-variable (object :: <function>,
                                      the-idvm-module == find-translator-module(#"idvm"),
                                      library,
                                      hint)
    let opcode = find-key(bytecodes-to-opcodes,curry(id?,object));

    if (opcode)
        values(opcode, the-idvm-module.module-name, library)
    else
        next-method()
    end if
end

define method variable-value (opcode :: <integer>, the-idvm-module == find-translator-module(#"idvm"), library)
    element(bytecodes-to-opcodes,opcode,default: #f) | next-method()
end method;
```