Object Description Format (ODF)

Designed by Rob MacLachlan, documented by Nick Kramer

1. Introduction

The basic requirement of the "Object Description Format" (ODF) is to allow an open-ended variety of objects to be dumped into a file, where:

- instances may refer to each other in an arbitrary graph structure, and
- some of the references may be to objects that are defined in distinct object description units (files, etc.)

In order to make the design interesting, the ODF attempts to satisfy various incompatible requirements:

- To support efficient parsing and unparsing on a variety of architectures, but also to potentially serve as a cross-platform interchange format.
- To work well on sequential media (like a socket), but also to potentially support random-access demand-loading of objects when possible (e.g. on files.)

2. Dumping Objects

begin-dumping [Function]
(name :: <data-unit-name>, type :: <fixed-integer>, #key where :: false-or(<string>)) => dump-state :: <dump-state>

Returns a dump state. *name* is a symbol. *Type* is "unit type code"; currently the only defined type is **$library-summary-unit-type**. *where* is the file name to dump to; this defaults to `name.du`.

end-dumping [Function]
(state :: <dump-state>) => ();

Actually output the file.

dump-od [Generic Function]
(obj :: <object>, buf :: <dump-buffer>)

Dump *obj* to the dump buffer (dump-states are subclasses of **<dump-buffer>**). Users can either define methods on **dump-od**, or call **add-make-dumper** to magically add methods to **dump-od**.

<dump-state> [Class]
Contains all information about the current object-dump in progress. It is illegal to use a dump-state after it has been passed to **end-dumping**. All instances of **<dump-state>** are created by calls to **begin-dumping**.

3. Loading Dumped Objects

<dispatcer> [Class]
A dispatcher is an object that tells the loader how to convert binary data into Dylan objects. It supports the *initialize-from:* keyword, which defaults to ***default-dispatcher***. If *initialize-from:* is not **#f**, the dispatcher returned will be a copy of the *initialize-from:* dispatcher.

default-dispatcher [Variable]

A dispatcher that can handle most of the objects defined in the Dylan language.

data-unit-search--path [Variable]

A sequence of strings, each string a Unix directory name. Used by **find-data-unit** to locate the dump files for data units.

find-data-unit

*(name :: <data-unit-name>, type :: <fixed-integer>, #key location-hint :: false-or(<location-hint>), check-hash: expected-hash :: false-or(<integer>), dispatcher :: <dispatcher> = *default-dispatcher*) => res :: <data-unit>;*

Load the data unit corresponding to *name*, and having the *type* ODF. *Location-hint:* is a string that's a file name for the dump file, which will be used if the data unit can not be found in the current ***data-unit-search-path***. *check-hash* can be used to detect version mismatches between data units. *dispatcher:* provides the dispatcher to be used for the loading process.

4. Defining Simple Dumpers and Loaders

To write a dumper for an object class, one needs to think of a name for the class (usually just take the **<>**'s off the class name), register the name with register-object-id, and define a dumper. If your data structure contains any object sharing or circular usages, also see "Circular Usage" on page 5 and "Object Sharing" on page 6. For automatic generation of object dumpers and loaders, see "Autodump" on page 3.

register-object-id [Function]

(name :: <symbol>, id :: <fixed-integer>) => ()

Equate the name with the id. This function is *not* exported; rather, all register-object-id calls must occur in the od-format module.

dump-od [Generic Function]

(obj :: <object>, buf :: <dump-buffer>)

One way to define a dumper for a class is to add a method to **dump-od**. See "Dumping Objects" on page 1 for more information.

add-od-loader [Function]

(dispatcher :: <dispatcher>, name :: <symbol>, meth :: <method>) => ()

Add a load method for name to the dispatcher. The loader for name will be meth, which has the signature method (state :: <load-state>) => res :: <object-type-read-in>.

If you use add-od-loader, you will need to define a corresponding method on dump-od.

add-make-dumper [Function]

(name :: <symbol>, dispatcher :: <dispatcher>, obj-class :: <class>, slots :: <list>, #key load-external :: <boolean>, dumper-only :: <boolean>, load-side-effect :: false-or(<method>)) => ();

A convenient way to define both a loader and a dumper for objects of class obj-class. Slots is a list which is in this format:

```
#(slot-accessor-function-1, init-keyword-1, setter-function-1,
slot-accessor-function-2, init-keyword-2, setter-function-2,
...)
```

Note that the stuff is implicitly grouped in clumps of 3. Either of the init-keyword or the setter may be #f, but not both. The "make dumper" requests backpatching when an unresolved forward reference is encountered (due to circularities.) Any slot which might have a forward reference must have a setter function (it can also have an init keyword, which will be used if possible.)

A "make dumper" may be used when an object can be reconstructed from some accessor values by reinstantiating the object with Make and backpatching some slots. This is pretty general, since the accessor and setter function don't really have to be slot accessors and setters.

5. Autodump

[Description of Autodump coming soon]

6. ODF Representation of Objects

In the dump files, objects contain the following parts: A header, raw data, sub-objects, and an end-of-object marker. The header is a single machine word which contains the class of the object. (Actually, it contains the object-id, which is looked up in the registry to find the class name) The header also says if there is any raw data, and if there are any sub-objects. The raw data section is zero or more machine-words which only have meaning to the object's loader. If raw data is present, it is preceded by a byte-count (which is a single machine word). If there are sub-objects, they come next. The last object in a sub-object list is a magical **$end-object**, which denotes both the end of the sub-objects list and the last word of the current object.

For more detail on the binary representation and on issues such as endian-ness and machine word size, see "Binary Representation" on page 6.

When an object is loaded, its header is read in by the loading code. The object class id is extracted from the header, and is looked up in the dispatcher to obtain a load function. This load function is expected to read the remainder of the data associated with the object (the header has already been read), and return the actual object.

7. Functions for Writing Dumpers

dump-definition-header [Function]

(name :: <symbol>, buf :: <dump-buffer>, #key subobjects = #f, raw-data = $odf-no-raw-data-format) => ()

Dump the object's header. Subobjects: #t means there are subobjects, otherwise there won't be any sub-objects.

dump-end-entry [Function]

(start-position :: <fixed-integer>, buf :: <dump-buffer>) => ();

Dump an "end of object" marker. This is only necessary if there are subobjects.

dump-simple-object [Function]

(name :: <symbol>, buf :: <dump-buffer>, #rest slots) => ();

Dump an object that has only a constant number of subobjects (no raw data). The subobjects are in the #rest args.

dump-word [Function]

(obj :: <integer>, buf :: <dump-buffer>) => ();

Dump an integer as a single machine word.

dump-raw-data [Function]
(obj :: <collection>, bsize :: <fixed-integer>, buf :: <dump-buffer>) => ();

BLT the contents of obj into the buffer. Currently we do whatever copy-into-buffer! does.

8. Functions for Writing Loaders

<load-state> [Class]
A load-state is an object that describes the current state of the loading process. Among other things, it contains an input <stream> and the buffer to the stream.

od-stream [Function]
(load-state :: <load-state>) => stream :: <stream>

The stream for load-state.

od-buffer [Function]
(load-state :: <load-state>) => buffer :: <buffer>

The buffer corresponding to **load-state.od-stream**.

od-next [Function]
(load-state :: <load-state>) => buffer-index :: <buffer-index>

The buffer index corresponding to load-state.od-stream and load-state.od-buffer.

od-next-setter [Function]
(buffer-index :: <buffer-index>, load-state :: <load-state>) => buffer-index :: <buffer-index>

Update the buffer index.

od-end [Function]
(load-state :: <load-state>) => end-index :: <buffer-index>

The end-index for the buffer and stream.

$end-object [Constant]
A magical object which is always the last sub-object in an object.

load-object-dispatch [Function]
(state :: <load-state>) => res :: <object>

Read an object header and invoke the appropriate loader. Return the object that the loader returns.

load-raw-data [Function]
(res :: <byte-string>, elsize :: <fixed-integer>, state :: <load-state>) => nnext :: <buffer-index>;

Loads data into res, destructively modifying it. elsize is the size of the elements. Calls copy-from-buffer to actually write to the result string.

load-subobjects-vector [Function]

(state :: <load-state>) => res :: <stretchy-vector>;

Loads all remaining subobjects in the current object. The object header has already been read.

load-sole-subobject [Function]

(state :: <load-state>) => res :: <object>;

Load a single subobject. An error is signaled if there is more than one subobject left.

assert-end-object [Function]

(state :: <load-state>) => ();

Reads from the buffer, and signals an error if the next thing read isn't an end-of-object marker.

9. Circular Usage

To deal with circular object usage graphs, several mechanisms are employed. Objects can be dumped as forward references and defined later. The loader for the object containing the forward reference has the option of delaying object creation until the forward referenced object is loaded, or requesting that the object be *backpatched* when the forward referenced object is finally loaded.

new-local-id [Function]

(buf :: <dump-state>) => res :: <fixed-integer>

Return a new local object ID. "Local" means local to the current data unit.

label-next-object [Function]

(id :: <fixed-integer>, buf :: <dump-state>) => id :: <fixed-integer>

Associate the next object dumped with ID so that references to the ID refer to this object. The argument ID is also returned (so that new-local-id(buf) can be the arg.)

dump-local-reference [Function]

(id :: <fixed-integer>, buf :: <dump-buffer>) => ()

Dump a reference to an object identified by id, rather than dumping the actual object. This can be used to implement object sharing when the sharing can only occur in well defined ways.

<forward-ref> [Class]

<forward-ref> is an IOU that that can be returned by any call to the loader in a place where there might be a reference to an object that hasn't been loaded yet. A loader can resolve forward references either by backpatching or by requiring users of the result to be prepared to stumble across <forward-ref> objects (probably by using actual-obj everywhere.)

When created, the forward-ref is stored in the local index of the <load-state> that is supposed to define the object. We discover that there was a forward ref when we go to store the actual value.

obj-resolved? [Function]

(obj :: <object>) => answer :: <boolean>

#t if obj has been resolved, #f otherwise.

actual-obj [Function]

(obj :: <object>) => object:: <object>

If obj is a <forward-ref>, this returns the object that the forward reference refers to. If the object has not yet been resolved, an error is signaled. If obj is not a <forward-ref>, the object is returned. (This is so loaders that might or might not need forward references are easier to write)

r**equest-backpatch** [Function]

(ref :: <forward-ref>, fun :: <function>) => ();

Register a backpatching function (normally a closure) for a particular forward reference. This function is called with the actual value when the reference is resolved, and is responsible for storing this value in the appropriate place. The signature for the function is method (object :: <object>) => ().

resolve-forward-ref [Function]

(ref :: <forward-ref>, value :: <object>) => ();

Resolve a forward reference, doing any necessary backpatching. This may be called by loaders to resolve a reference early (before the loader for the object has returned.)

10. Object Sharing

By default, it is assumed that the object being dumped has no object sharing, i.e., for all sub-objects the ref-count is one. For any object who's ref-count is greater than one, that object must be a sub-class of **<identity-preserving-mixin>**. (If it is ok to dump the object multiple times and load it back as separate objects, then this superclass may be omitted)

<identity-preserving-mixin> [Class]

This mixin is inherited by objects that want to preserve object identity, either within or across data unit boundaries (local or external references. The maybe-dump-reference function can be used by dumpers for objects that have this mixin in order to ensure that only one copy is dumped, and other mentions of the object become references.

maybe-dump-reference [Function]

(obj :: <identity-preserving-mixin>, buf :: <dump-state>) => res :: <boolean>;

If Obj has already been dumped, dump a reference to it, and return #F. Otherwise, mark the next object dumped as being the definition of this object, and return #T. In other words, the work of dumping is complete when we #F. When we return #T, the caller needs to actually dump the guts of the object.

Callers that want to use the ref mechanism say this in their dump-od method:

```
when (maybe-dump-reference(obj, buf))
            ... actually dump othe object....
end unless;
```

11. Binary Representation

See the first few pages of comments in `od-format.dylan`.