DREF MANUAL

Contents

[in package DREF]

1 Links and Systems

Here is the official repository and the HTML documentation for the latest version.

- **[system]** **"dref"**
  - *Version:* 0.5.0
  - *Description:* Reify definitions, provide portable access to arglists, docstrings and source locations in an extensible framework.
  - *Long Description:* `defun` defines a first-class object: a `function`. `defvar` does not. The DRef library provides a way to refer to all definitions and smooths over the differences between implementations. This system has minimal dependencies. It autoloads the `dref/full` `asdf:system`.
  - *Licence:* MIT, see COPYING.
  - *Author:* Gábor Melis
  - *Mailto:* mega@retes.hu
  - *Homepage:* http://github.com/melisgl/dref/
  - *Bug tracker:* https://github.com/melisgl/dref/issues
  - *Source control:* GIT
  - *Depends on:* autoload, mgl-pax-bootstrap, named-readtables, pythonic-string-reader
  - *Auto depends on:* dref/full
  - *Defsystem depends on:* autoload
- **[system]** **"dref/full"**
  - *Description:* DRef with everything loaded. There should be no need to explicitly load this system (or depend on it) because it is autoloaded as necessary by all publicly accessible functionality in `dref`.

    However, to get the dependencies, install this system.

    For a discussion of conditional dependencies, see Backends.
  - *Depends on:* alexandria, closer-mop, dref, mgl-pax, sb-introspect(?), swank(?)

- *Defsystem depends on:* autoload

2 Introduction

What if definitions were first-class objects?

Some defining forms do not create first-class objects. For example, `defun` creates `function` objects, but `defvar` does not create variable objects, as no such thing exists in Common Lisp. The main purpose of this library is to fill this gap with the introduction of `dref` objects:

```
(defvar *my-var* nil
  "This is my var.")
(dref '*my-var* 'variable)
==> #<DREF *MY-VAR* VARIABLE>
```

`dref`s just package up a name (`*my-var*`) and a locative (`variable`) then check that the definition actually exists:

```
(dref 'junk 'variable)
.. debugger invoked on LOCATE-ERROR:
..   Could not locate JUNK VARIABLE.
```

The Basic Operations on definitions in DRef are `arglist`, `docstring` and `source-location`.

```
(docstring (dref '*my-var* 'variable))
=> "This is my var."
=> NIL
```

For definitions associated with objects, the definition can be `locate`d from the object:

```
(locate #'print)
==> #<DREF PRINT FUNCTION>
```

These objects designate their definitions, so `(docstring #'print)` works. Extending DRef and these operations is possible through Defining Locative Types. It is also possible to define new operations. For example, PAX makes `pax:document` extensible through `pax:document-object*`.

Finally, existing definitions can be queried with `definitions` and `dref-apropos`:

```
(definitions 'dref-ext:arglist*)
==> (#<DREF ARGLIST* GENERIC-FUNCTION>
-->  #<DREF ARGLIST* (METHOD (CLASS-DREF))>
-->  #<DREF ARGLIST* (METHOD (COMPILER-MACRO-DREF))>
-->  #<DREF ARGLIST* (METHOD (FUNCTION-DREF))>
-->  #<DREF ARGLIST* (METHOD (LAMBDA-DREF))>
-->  #<DREF ARGLIST* (METHOD (MACRO-DREF))>
-->  #<DREF ARGLIST* (METHOD (METHOD-DREF))>
-->  #<DREF ARGLIST* (METHOD (SETF-DREF))>
-->  #<DREF ARGLIST* (METHOD (TYPE-DREF))> #<DREF ARGLIST* (METHOD (DREF))>
-->  #<DREF ARGLIST* (METHOD (MGL-PAX::GO-DREF))>
-->  #<DREF ARGLIST* (METHOD (T))>
-->  #<DREF ARGLIST* (UNKNOWN
```

```
-->                    (DECLAIM ARGLIST*
-->                             FTYPE))>)
```

```
(dref-apropos 'locate-error :package :dref)
==> (#<DREF LOCATE-ERROR CONDITION> #<DREF LOCATE-ERROR FUNCTION>)

(dref-apropos "ate-err" :package :dref :external-only t)
==> (#<DREF LOCATE-ERROR CONDITION> #<DREF LOCATE-ERROR FUNCTION>)
```

3 References

After the Introduction, here we get into the details. Of special interest are:

- The `xref` function to construct an arbitrary reference without any checking of validity.
- `locate` and `dref` to look up the definition of an object (e.g `#'print`) or a reference (e.g. `(xref 'print 'function)`).
- `resolve` to find the first-class (non-`xref`) object the definition refers to, if any.

The Basic Operations (`arglist`, `docstring`, `source-location`) know how to deal with references (discussed in the Extending DRef).

- **[class]** **xref**

  An `xref` (cross-reference) is a reference. It may represent some kind of definition of its name in the context given by its locative. The definition may not exist and the locative may even be invalid. The subclass `dref` represents definitions that exist.

- **[function]** **xref** *name locative*

  A shorthand for `(make-instance 'xref :name name :locative locative)` to create `xref` objects. It does no error checking: the `locative-type` of `locative-type` need not be defined, and the `locative-args` need not be valid. Use `locate` or the `dref` function to create `dref` objects.

- **[function]** **xref=** *xref1 xref2*

  See if `xref1` and `xref2` have the same `xref-name` and `xref-locative` under `equal`. Comparing like this makes most sense for `dref`s. However, two `xref`s different under `xref=` may denote the same `dref`s.

- **[class]** **dref** *xref*

  `dref`s can be thought of as definitions that actually exist, although changes in the system can invalidate them (for example, a `dref` to a function definition can be invalidated by `fmakunbound`). `dref`s must be created with `locate` or the `dref` function.

  Two `dref`s created in the same dynamic environment denote the same thing if and only if they are `xref=`.

- **[function]** **locate** *object &optional (errorp t)*

Return a `dref` representing the definition of `object`.

`object` must be a supported first-class object, a `dref`, or an `xref`:

```
(locate #'print)
==> #<DREF PRINT FUNCTION>
```

```
(locate (locate #'print))
==> #<DREF PRINT FUNCTION>
```

```
(locate (xref 'print 'function))
==> #<DREF PRINT FUNCTION>
```

When `object` is a `dref`, it is simply returned.

Else, a `locate-error`(`0` `1`) is signalled if `object` is an `xref` with an invalid locative, or if no corresponding definition is found. If `errorp` is `nil`, then `nil` is returned instead.

```
(locate (xref 'no-such-function 'function))
.. debugger invoked on LOCATE-ERROR:
..   Could not locate NO-SUCH-FUNCTION FUNCTION.
..   NO-SUCH-FUNCTION does not name a function.
```

```
(locate (xref 'print '(function xxx)))
.. debugger invoked on LOCATE-ERROR:
..   Could not locate PRINT #'XXX.
..   Bad arguments (XXX) for locative FUNCTION with lambda list NIL.
```

```
(locate "xxx")
.. debugger invoked on LOCATE-ERROR:
..   Could not locate "xxx".
```

Use the `xref` function to construct an `xref` without error checking.

See Extending `locate`.

- **[function]** **dref** *name locative &optional (errorp t)*

  Shorthand for `(locate (xref name locative) errorp)`.

- **[function]** **resolve** *object &optional (errorp t)*

  If `object` is an `xref`, then return the first-class object associated with its definition if any. Return `object` if it's not an `xref`. Thus, the value returned is never an `xref`. The second return value is whether resolving succeeded.

```
(resolve (dref 'print 'function))
==> #<FUNCTION PRINT>
=> T
```

```
(resolve #'print)
==> #<FUNCTION PRINT>
=> T
```

If `object` is an `xref`, and the definition for it cannot be `locate`d, then `locate-error`(`0` `1`) is signalled.

```
(resolve (xref 'undefined 'variable))
.. debugger invoked on LOCATE-ERROR:
..   Could not locate UNDEFINED VARIABLE.
```

If there is a definition, but there is no first-class object corresponding to it, then `resolve-error`(`0` `1`) is signalled or `nil` is returned depending on `errorp`:

```
(resolve (dref '*print-length* 'variable))
.. debugger invoked on RESOLVE-ERROR:
..   Could not resolve *PRINT-LENGTH* VARIABLE.
```

```
(resolve (dref '*print-length* 'variable) nil)
=> NIL
=> NIL
```

`resolve` is a partial inverse of `locate`: if a `dref` is `resolve`able, then `locate`ing the object it resolves to recovers the `dref` equivalent to the original (`xref=` and of the same type but not `eq`).

Can be extended via `resolve*`.

- **[condition]** **locate-error** *error condition-context-mixin*

  Signalled by `locate` when the definition cannot be found, and `errorp` is true.

- **[condition]** **resolve-error** *error condition-context-mixin*

  Signalled by `resolve` when the object defined cannot be returned, and `errorp` is true.

3.1 Dissecting References

- **[reader]** **xref-name** *xref (:name)*

  The name of the reference.

- **[reader]** **xref-locative** *xref (:locative)*

  The locative of the reference.

  The locative is normalized by replacing single-element lists with their only element:

```
(xref 'print 'function)
==> #<XREF PRINT FUNCTION>
```

```
(xref 'print '(function))
==> #<XREF PRINT FUNCTION>
```

- **[reader]** **dref-name** *dref*

  The same as `xref-name`, but only works on `dref`s. Use it as a statement of intent.

- **[reader]** **dref-locative** *dref*

The same as xref-locative, but only works on drefs. Use it as a statement of intent.

- **[reader]** **dref-origin** *dref*

  The object from which locate constructed this dref. dref-origin may have presentation arguments, which are not included in locative-args as is the case with the initform argument of the variable locative:

```
(dref '*standard-output* '(variable "see-below"))
==> #<DREF *STANDARD-OUTPUT* VARIABLE>
```

```
(dref-origin (dref '*standard-output* '(variable "see-below")))
==> #<XREF *STANDARD-OUTPUT* (VARIABLE "see-below")>
```

  The initform argument overrides the global binding of *standard-output* when it's pax:documented:

```
(first-line
 (pax:document (dref '*standard-output* '(variable "see-below"))
               :stream nil))
=> "- [variable] *STANDARD-OUTPUT* \"see-below\""
```

- **[function]** **locative-type** *locative*

  Return locative type of the locative locative. This is the first element of locative if it's a list. If it's a symbol, it's that symbol itself.

- **[function]** **locative-args** *locative*

  Return the rest of locative locative if it's a list. If it's a symbol, then return nil. See locative.

The following convenience functions are compositions of {locative-type, locative-args} and {xref-locative, dref-locative}.

- **[function]** **xref-locative-type** *xref*
- **[function]** **xref-locative-args** *xref*
- **[function]** **dref-locative-type** *dref*
- **[function]** **dref-locative-args** *dref*

3.2 References Glossary

- **[glossary-term]** **reference**

  A reference is a name plus a locative, and it identifies a possible definition. References are of class xref. When a reference is a dref, it may also be called a definition.

- **[glossary-term]** **definition**

A definition is a reference that identifies a concrete definition. Definitions are of class `dref`. A definition `resolve`s to the first-class object associated with the definition if such a thing exists, and `locate` on this object returns the canonical `dref` object that's unique under `xref=`.

The kind of a definition is given by its locative type. There is at most one definition for any given name and locative type. Equivalently, there can be no two definitions of the same `dref-name` and `dref-locative-type` but different `dref-locative-args`.

- **[glossary-term] name**

  Names are symbols, strings and nested lists of the previous, which name functions, types, packages, etc. Together with locatives, they form references.

  See `xref-name` and `dref-name`.

- **[glossary-term] locative**

  Locatives specify a *type* of definition such as `function` or `variable`. Together with names, they form references.

  In their compound form, locatives may have arguments (see `locative-args`) as in `(method (number))`. In fact, their atomic form is shorthand for the common no-argument case: that is, `function` is equivalent to `(function)`.

  A locative is valid if it names an existing locative type and its `locative-args` match that type's lambda-list (see `define-locative-type`).

  ```
  (arglist (dref 'method 'locative))
  => (&REST QUALIFIERS-AND-SPECIALIZERS)
  => :DESTRUCTURING
  ```

  See `xref-locative` and `dref-locative`.

- **[glossary-term] locative type**

  The locative type is the part of a locative that identifies what kind of definition is being referred to. This is always a symbol.

  Locative types are defined with `define-locative-type` or `define-pseudo-locative-type`. See Basic Locative Types for the list locative types built into DRef, and PAX Locatives for those in PAX.

  Also, see `locative-type`, `xref-locative-type`, `dref-locative-type`, Defining Locative Types.

- **[glossary-term] presentation**

  references may have arguments (see Defining Locative Types) that do not affect the behaviour of `locate` and the Basic Operations, but which may be used for other, "presentation" purposes. For example, the `variable` locative's `initform` argument is used for presentation by `pax:document`. Presentation arguments are available via `dref:dref-origin` but do not feature in `dref-locative` to ensure the uniqueness of the definition under `xref=`.

4 dtypes

dtypes generalize locative types the same way Lisp types generalize classes. A dtype is either

- a locative type such as function, type and clhs, or
- a full locative such as (method (number)) and (clhs section), or
- nil (the empty dtype) and t (that encompasses all lisp-locative-types), or
- named with define-dtype (such as pseudo and top), or
- a combination of the above with and, or and not, or
- a member(0 1) form with locateable definitions, or
- a satisfies form with the name of a function that takes a single definition as its argument.

dtypes are used in definitions and dref-apropos to filter the set of definitions as in

```
(definitions 'print :dtype '(not unknown))
==> (#<DREF PRINT (CLHS FUNCTION)> #<DREF PRINT FUNCTION>)
```

```
(dref-apropos "type specifier" :dtype 'pseudo)
==> (#<DREF "1.4.4.6" #1=(CLHS SECTION)> #<DREF "1.4.4.6.1" #1#>
-->  #<DREF "1.4.4.6.2" #1#> #<DREF "1.4.4.6.3" #1#>
-->  #<DREF "1.4.4.6.4" #1#> #<DREF "4.2.3" #1#>
-->  #<DREF "atomic type specifier" #2=(CLHS GLOSSARY-TERM)>
-->  #<DREF "compound type specifier" #2#>
-->  #<DREF "derived type specifier" #2#> #<DREF "type specifier" #2#>)
```

- **[macro] define-dtype** *name lambda-list &body body*

  Like deftype, but it may expand into other dtypes.

  The following example defines method* as the locative method without its direct locative subtypes.

  ```
  (define-dtype method* () '(and method (not reader) (not writer)))
  ```

  See dtypep for the semantics and also the locative dtype.

- **[dtype] top**

  This is the top of the dtype hierarchy, much like t for Lisp types. It expands to (or t pseudo). While t matches every normal Lisp object and objectless definitions present in the running Lisp (see lisp-locative-types), top matches even pseudo definitions (see pseudo-locative-types).

- **[dtype] pseudo**

  This is the union of all pseudo-locative-types. It expands to (or ,@(pseudo-locative-types)).

- **[function] dtypep** *dref dtype*

See if `dref` is of `dtype`.

- *Atomic locatives:* If `dtype` is a locative type, then it matches definitions with that locative type and its locative subtypes.

  Because `constant` is defined with `variable` among its `locative-supertypes`:

```
(dtypep (dref 'pi 'constant) 'variable)
=> T
```

```
(dtypep (dref 'number 'class) 'type)
=> T
```

  It is an error if `dtype` is an `atom`(`0` `1`) but not a locative type. On the other hand, (the empty) argument list of atomic locatives is not checked even if having no arguments makes them invalid.

- *Compound locatives:* Locatives in their compound form are validated and must match exactly (under `equal`, as in `xref=`).

```
(defparameter *d* (dref 'dref* '(method (t t t))))
(defparameter *d2* (dref 'dref* '(method :around (t t t))))
(dtypep *d* 'method)
=> T
(dtypep *d* '(accessor))
.. debugger invoked on SIMPLE-ERROR:
..   Bad arguments NIL for locative ACCESSOR with lambda list
↪  (CLASS-NAME).
(dtypep *d* '(method (t t t)))
=> T
(dtypep *d2* '(method (t t t)))
=> NIL
```

- `dtype` may be constructed with `and`, `or` and `not` from Lisp types, locative types, full locatives and named `dtype`s:

```
(dtypep (dref 'locate-error 'condition) '(or condition class))
=> T
(dtypep (dref nil 'type) '(and type (not class)))
=> T
```

- For `(member &rest objs)`, each of `objs` is `locate`d and `dref` is matched against them with `xref=`:

```
(dtypep (locate #'print) `(member ,#'print))
=> T
```

- For `(satisfies pred)`, the predicate `pred` is funcalled with `dref`.

- `dtype` may be named by `define-dtype`:

```
(dtypep (locate #'car) 'top)
=> T
```

5 Listing Definitions

- **[function]** **definitions** *name &key (dtype t) (sort t)*

  List all definitions of `name` that are of `dtype` as `drefs`.

  Just as `(dref name locative)` returns the canonical definition, the `dref-name`s of returned by `definitions` may be different from `name`:

```
(definitions "PAX")
==> (#<DREF "MGL-PAX" PACKAGE>)
```

```
(definitions 'mgl-pax)
==> (#<DREF "MGL-PAX" PACKAGE> #<DREF "mgl-pax" ASDF/SYSTEM:SYSTEM>)
```

  Similarly, `dref-locative-type` may be more made more specific:

```
(definitions 'dref:locate-error :dtype 'type)
==> (#<DREF LOCATE-ERROR CONDITION>)
```

  If `sort`, the returned list is ordered according to `sort-references`.

  Can be extended via `map-definitions-of-name`.

- **[function]** **dref-apropos** *name &key package external-only case-sensitive (dtype t) (sort t)*

  Return a list of `dref`s corresponding to existing definitions that match the various arguments. First, `(dref-apropos nil)` lists all definitions in the running Lisp and maybe more (e.g. `mgl-pax:clhs`). Arguments specify how the list of definitions is filtered.

  `dref-apropos` itself is similar to `cl:apropos-list`, but

  - it finds definitions not `symbol`s,
  - it supports an extensible definition types, and
  - filtering based on them.

  PAX has a live browsing frontend.

  Roughly speaking, when `name` or `package` is a `symbol`, it must match the whole name of the definition:

```
(dref-apropos 'method :package :dref :external-only t)
==> (#<DREF METHOD CLASS> #<DREF METHOD LOCATIVE>)
```

  On the other hand, when `name` or `package` is a `string`(`0` `1`), they are matched as substrings to the definition's name `princ-to-string`ed:

```
(dref-apropos "method" :package :dref :external-only t)
==> (#<DREF SETF-METHOD LOCATIVE> #<DREF METHOD CLASS>
-->  #<DREF METHOD LOCATIVE> #<DREF METHOD-COMBINATION CLASS>
-->  #<DREF METHOD-COMBINATION LOCATIVE>)
```

  Definitions that are not of `dtype` (see `dtypep`) are filtered out:

```
(dref-apropos "method" :package :dref :external-only t :dtype 'class)
==> (#<DREF METHOD CLASS> #<DREF METHOD-COMBINATION CLASS>)
```

When `package` is `:none`, only non-symbol names are matched:

```
(dref-apropos "dref" :package :none)
==> (#<DREF "DREF" PACKAGE> #<DREF "DREF-EXT" PACKAGE>
-->  #<DREF "DREF-TEST" PACKAGE> #<DREF "dref" ASDF/SYSTEM:SYSTEM>
-->  #<DREF "dref/full" ASDF/SYSTEM:SYSTEM>
-->  #<DREF "dref/test" ASDF/SYSTEM:SYSTEM>
-->  #<DREF "dref/test-autoload" ASDF/SYSTEM:SYSTEM>)
```

The exact rules of filtering are as follows. Let `c` be the name of the candidate definition from the list of all definitions that we are matching against the arguments and denote its string representation `(princ-to-string c)` with `p`. Note that `princ-to-string` does not print the package of symbols. We say that two strings *match* if `case-sensitive` is `nil` and they are `equalp`, or `case-sensitive` is true and they are `equal`. `case-sensitive` affects *substring* comparisons too.

- If `name` is a `symbol`, then its `symbol-name` must *match* `p`.
- If `name` is a `string`, then it must be a *substring* of `p`.
- If `package` is `:any`, then `c` must be a `symbol`.
- If `package` is `:none`, then `c` must *not* be a `symbol`.
- If `package` is not `nil`, `:any` or `:none`, then `c` must be a symbol.
- If `package` is a `package`, it must be `eq` to the `symbol-package` of `c`.
- If `package` is a `symbol` other than `nil`, `:any` and `:none`, then its `symbol-name` must *match* the `package-name` or one of the `package-nicknames` of `symbol-package` of `c`.
- If `package` is a `string`, then it must be a *substring* of the `package-name` of `symbol-package` of `c`.
- If `external-only` and `c` is a symbol, then `c` must be external in a matching package.
- `dtype` matches candidate definition `d` if `(dtypep d dtype)`.

If `sort`, the returned list is ordered according to `sort-references`.

Can be extended via MAP-REFERENCES-OF-TYPE and `map-definitions-of-name`.

- **[macro] with-definitions-cached** *(&key (dtype 'top)) &body body*

  Allow caching of definition lookups and `dtype` computations. This can provide a speedup when multiple calls are made to `definitions` or `dref-apropos` within `body`.

  Using this macro makes the promise that, during the dynamic extent of `body`, the following are unchanged:

  - the set of definitions of `dtype`,

    - the set of all locative types,
    - the set of all `dtypes`.

  Note that definitions *not* of `dtype` are allowed to be made or deleted.

- **[function]** **sort-references** *references &key key*

  Order `references` in an implementation independent way.

    - Non-symbol named references go first and are sorted by the locative type first, then by name and locative args.
    - Symbol-based references go after and are sorted by name first, then by the locative-type, finally by the locative args.

- **[function]** **sort-locative-types** *locative-types*

  Sort `locative-types` in an implementation independent way. The order is alphabetical in `symbol-name`, with the `package-name` acting as a tie breaker. If `unknown` is present among `locative-types`, it goes last.

- **[glossary-term]** **reverse definition order**

  The lists of locative types returned by e.g. `locative-types`, `lisp-locative-types`, `pseudo-locative-types` and `locative-aliases` are in reverse order of the time of their definition. This order is not affected by redefinition, regardless of whether it's by `define-locative-type`, `define-pseudo-locative-type`, `define-symbol-locative-type` or `define-locative-alias`.

- **[function]** **locative-types**

  Return a list of non-alias locative types. This is the `union` of `lisp-locative-types` and `pseudo-locative-types`, which is the set of constituents of the `dtype` `top`.

  This list is in reverse definition order.

- **[function]** **lisp-locative-types**

  Return the locative types that correspond to Lisp definitions, which typically have `source-location`. These are defined with `define-locative-type` and `define-symbol-locative-type` and are the constituents of `dtype` `t`.

  This list is in reverse definition order.

- **[function]** **pseudo-locative-types**

  Return the locative types that correspond to non-Lisp definitions. These are the ones defined with `define-pseudo-locative-type` and are the constituents of `dtype` `pseudo`.

  This list is in reverse definition order.

- **[function]** **locative-aliases**

  Return the list of locatives aliases, defined with `define-locative-alias`.

  This list is in reverse definition order.

6 Basic Operations

The following functions take a single argument, which may be a `dref`, or an object denoting its own definition (see `locate`).

- **[function]** **arglist** *object*

  Return the arglist of the definition of `object` or `nil` if the arglist cannot be determined.

  The second return value indicates whether the arglist has been found. As the second return value, `:ordinary` indicates an ordinary lambda list, `:macro` a macro lambda list, `:deftype` a deftype lambda list, and `:destructuring` a destructuring lambda list. Other non-`nil` values are also allowed.

```
(arglist #'arglist)
=> (OBJECT)
=> :ORDINARY
```

```
(arglist (dref 'define-locative-type 'macro))
=> (LOCATIVE-TYPE-AND-LAMBDA-LIST LOCATIVE-SUPERTYPES &OPTIONAL
    DOCSTRING DREF-DEFCLASS-FORM)
=> :MACRO
```

```
(arglist (dref 'arglist* '(method (method-dref))))
=> ((DREF METHOD-DREF))
=> :SPECIALIZED
```

```
(arglist (dref 'method 'locative))
=> (&REST QUALIFIERS-AND-SPECIALIZERS)
=> :DESTRUCTURING
```

  This function supports `macros`, `compiler-macros`, `setf` functions, `functions`, `generic-functions`, `methods`, `types`, `locatives`. Note that `arglist` depends on the quality of `swank-backend:arglist`. With the exception of SBCL, which has perfect support, all Lisp implementations have minor omissions:

  - `deftype` lambda lists on ABCL, AllegroCL, CLISP, CCL, CMUCL, ECL;
  - default values in `macro` lambda lists on AllegroCL;
  - various edge cases involving traced functions.

  Can be extended via `arglist*`.

- **[function]** **arglist-parameters** *arglist &optional (arglist-type :ordinary)*

  A utility to extract the names of the parameters from `arglist` of `arglist-type`. See the function `arglist`, for the possible values of `arglist-type`.

  Note that if `arglist` is `nil`, then `arglist-type` may also be `nil`, in which case `nil` is returned.

```
(arglist #'source-location)
=> (OBJECT &KEY ERROR)
=> :ORDINARY
```

```
(multiple-value-call 'arglist-parameters (arglist #'source-location))
=> (OBJECT ERROR)
```

Can be extended via `arglist-parameters*`.

- **[function]** **docstring** *object*

  Return the docstring from the definition of `object`. As the second value, return the `*package*` that was in effect when the docstring was installed or `nil` if it cannot be determined (this is used by `pax:document` when determinining the Package and Readtable of docstrings). This function is similar in purpose to `cl:documentation(0 1)`.

  Note that some locative types such as `asdf:systems` and `declarations` have no docstrings, and some Lisp implementations do not record all docstrings. The following are known to be missing:

  - `compiler-macro` docstrings on ABCL, AllegroCL, CCL, ECL;
  - `method-combination` docstrings on ABCL, AllegroCL.

  Can be extended via `docstring*`.

- **[function]** **source-location** *object &key error*

  Return the Swank source location for the defining form of `object`.

  The returned Swank location object is to be accessed only through the Source Locations API or to be passed to e.g Slime's `slime-goto-source-location`.

  If no source location was found,

  - if `error` is `nil`, then return `nil`;
  - if `error` is `:error`, then return a list of the form `(:error <error-message>)` suitable for `slime-goto-source-location`;
  - if `error` is `t`, then signal an `error` condition with the same error message as in the previous case.

  Note that the availability of source location information varies greatly across Lisp implementations.

  With the `nil backend`, `dref:source-location` loses precision beyond toplevel forms.

  Can be extended via `source-location*`.

7 Basic Locative Types

The following are the locative types supported out of the box. Like all locative types, they are named by symbols. When there is a CL type corresponding to the reference's locative type, the

references can be resolved to a unique object as is the case in

```
(resolve (dref 'print 'function))
==> #<FUNCTION PRINT>
=> T
```

Even if there is no such CL type, the arglist, the docstring, and the source-location of the defining form is usually recorded unless otherwise noted.

The basic locative types and their inheritance structure is loosely based on the doc-type argument of cl:documentation.

7.1 Locatives for Variables

- **[locative]** **variable** *&optional initform*
    - Direct locative subtypes: concept, glossary-term, section, constant

    Refers to a global special variable. initform, or if not specified, the global value of the variable is to be used for presentation.

    ```
    (dref '*print-length* 'variable)
    ==> #<DREF *PRINT-LENGTH* VARIABLE>
    ```

    variable references do not resolve.

- **[locative]** **constant** *&optional initform*
    - Direct locative supertypes: variable

    Refers to a constant variable defined with defconstant. initform, or if not specified, the value of the constant is included in the documentation. The constant locative is like the variable locative, but it also checks that its object is constantp.

    constant references do not resolve.

7.2 Locatives for Macros

- **[locative]** **setf**
    - Direct locative subtypes: setf-method, setf-function

    Refers to a setf expander (see defsetf and define-setf-expander).

    Setf functions (e.g. (defun (setf name) ...) or the same with defgeneric) are handled by the setf-function, setf-generic-function, and setf-method locatives.

    setf expander references do not resolve.

- **[locative]** **macro**

    Refers to a global macro, typically defined with defmacro, or to a special operator.

    macro references resolve to the macro-function of their name or signal resolve-error(0 1) if that's nil.

- **[locative] symbol-macro**

  Refers to a global symbol macro, defined with `define-symbol-macro`. Note that since `define-symbol-macro` does not support docstrings, PAX defines methods on the `documentation`(`0` `1`) generic function specialized on `(doc-type (eql 'symbol-macro))`.

```
(define-symbol-macro my-mac 42)
(setf (documentation 'my-mac 'symbol-macro)
      "This is MY-MAC.")
(documentation 'my-mac 'symbol-macro)
=> "This is MY-MAC."
```

  `symbol-macro` references do not `resolve`.

- **[locative] compiler-macro**

  - Direct locative subtypes: setf-compiler-macro

  Refers to a `compiler-macro-function`, typically defined with `define-compiler-macro`.

- **[locative] setf-compiler-macro**

  - Direct locative supertypes: compiler-macro

  Refers to a compiler macro with a setf function name.

  `setf-compiler-macro` references do not `resolve`.

7.3 Locatives for Functions and Methods

- **[locative] function**

  - Direct locative subtypes: structure-accessor, setf-function, generic-function

  Refers to a global function, typically defined with `defun`. The name must be a function name. It is also allowed to reference `generic-function`s as `function`s:

```
(dref 'docstring 'function)
==> #<DREF DOCSTRING FUNCTION>
```

- **[locative] setf-function**

  - Direct locative supertypes: function, setf
  - Direct locative subtypes: structure-accessor, setf-generic-function

  Refers to a global `function`(`0` `1`) with a setf function name.

```
(defun (setf ooh) ())
(locate #'(setf ooh))
==> #<DREF OOH SETF-FUNCTION>
(dref 'ooh 'setf-function)
==> #<DREF OOH SETF-FUNCTION>
(dref '(setf ooh) 'function)
==> #<DREF OOH SETF-FUNCTION>
```

- **[locative]** **generic-function**
  - Direct locative supertypes: function
  - Direct locative subtypes: setf-generic-function

  Refers to a `generic-function`, typically defined with `defgeneric`. The name must be a function name.

- **[locative]** **setf-generic-function**
  - Direct locative supertypes: generic-function, setf-function

  Refers to a global `generic-function` with a setf function name.

```
(defgeneric (setf oog) ())
(locate #'(setf oog))
==> #<DREF OOG SETF-GENERIC-FUNCTION>
(dref 'oog 'setf-function)
==> #<DREF OOG SETF-GENERIC-FUNCTION>
(dref '(setf oog) 'function)
==> #<DREF OOG SETF-GENERIC-FUNCTION>
```

- **[locative]** **method** *&rest qualifiers-and-specializers*
  - Direct locative subtypes: writer, reader, setf-method

  Refers to a `method`. name must be a function name. `method-qualifiers-and-specializers` has the form

```
  (<QUALIFIER>* <SPECIALIZERS>)
```

  For example, the method

```
(defgeneric foo-gf (x y z)
  (:method :around (x (y (eql 'xxx)) (z string))
    (values x y z)))
```

  can be referred to as

```
(dref 'foo-gf '(method :around (t (eql xxx) string)))
==> #<DREF FOO-GF (METHOD :AROUND (T (EQL XXX) STRING))>
```

  `method` is not `exportable-locative-type-p`.

- **[locative]** **setf-method** *&rest method-qualifiers-and-specializers*
  - Direct locative supertypes: method, setf
  - Direct locative subtypes: accessor

  Refers to a `method` of a `setf-generic-function`.

```
(defgeneric (setf oog) (v)
  (:method ((v string))))
(locate (find-method #'(setf oog) () (list (find-class 'string))))
==> #<DREF OOG (SETF-METHOD (STRING))>
```

```
(dref 'oog '(setf-method (string)))
==> #<DREF OOG (SETF-METHOD (STRING))>
(dref '(setf oog) '(method (string)))
==> #<DREF OOG (SETF-METHOD (STRING))>
```

- **[locative]** **method-combination**

  Refers to a `method-combination`, defined with `define-method-combination`.

  `method-combination` references do not `resolve`.

- **[locative]** **reader** *class-name*

  - Direct locative supertypes: method
  - Direct locative subtypes: accessor

  Refers to a `:reader` method in a `defclass`:

```
(defclass foo ()
  ((xxx :reader foo-xxx)))

(dref 'foo-xxx '(reader foo))
==> #<DREF FOO-XXX (READER FOO)>
```

- **[locative]** **writer** *class-name*

  - Direct locative supertypes: method
  - Direct locative subtypes: accessor

  Like `accessor`, but refers to a `:writer` method in a `defclass`.

- **[locative]** **accessor** *class-name*

  - Direct locative supertypes: reader, writer, setf-method

  Refers to an `:accessor` in a `defclass`.

  An `:accessor` in `defclass` creates a reader and a writer method. Somewhat arbitrarily, `accessor` references `resolve` to the writer method but can be `locate`d with either.

- **[function]** **accessor-slot-definition** *dref*

  Return the SLOT-DEFINITION object corresponding to `dref`, which may denote a `reader`, a `writer` or an `accessor`.

- **[locative]** **structure-accessor** *&optional structure-class-name*

  - Direct locative supertypes: setf-function, function

  Refers to an accessor function generated by `defstruct` or `defstruct*`. A `locate-error` condition is signalled if the wrong `structure-class-name` is provided.

  Note that there is no portable way to detect structure accessors, and on some platforms, `(locate #'my-accessor)`, `definitions` and `dref-apropos` will return `function(0 1)` references instead. On such platforms, `structure-accessor` references do not `resolve`.

- **[macro]** **defstruct*** *name-and-options &rest slot-descriptions*

  Like `defstruct`, but support `:documentation` among slot options. The documentation is attached to the slot's `structure-accessor`. Example:

```
(defstruct* my-struct
  (my-slot nil :documentation "docstring"))
```

  In addition to the normal `defstruct` processing, the above also does the moral equivalent of

```
(setf (documentation 'my-struct-my-slot 'function) "docstring")
```

7.4 Locatives for Types and Declarations

- **[locative]** **type**
  - Direct locative subtypes: class

  This locative can refer to types and classes as well as conditions, simply put, to things defined by `deftype`, `defclass` and `define-condition`.

```
(deftype my-type () t)
(dref 'my-type 'type)
==> #<DREF MY-TYPE TYPE>
```

```
(dref 'xref 'type)
==> #<DREF XREF CLASS>
```

```
(dref 'locate-error 'type)
==> #<DREF LOCATE-ERROR CONDITION>
```

  `type` references do not `resolve`.

- **[locative]** **class**
  - Direct locative supertypes: type
  - Direct locative subtypes: condition, structure

  Naturally, `class` is the locative type for `class`es.

  `arglist` returns the compound type specifier syntax associated with system classes. E.g. for the `integer` class:

```
(arglist (find-class 'integer))
=> (&OPTIONAL LOWER-LIMIT UPPER-LIMIT)
=> :DEFTYPE
```

- **[locative]** **structure**
  - Direct locative supertypes: class

  Refers to a `structure-class`, typically defined with `defstruct`.

Also, see `defstruct*`.

- **[locative] declaration**

  Refers to a declaration, used in `declare`, `declaim` and `proclaim`.

  User code may also define new declarations with CLTL2 functionality, but their `arglist` is always `nil`. On implementations that support it, `docstring` returns `(documentation name 'declaration)`.

```
(cl-environments:define-declaration my-decl (&rest things)
  (values :declare (cons 'foo things)))
```

  `declaration` references do not `resolve`.

  Also, `source-location` on declarations currently only works on SBCL.

7.5 Locatives for the Condition System

- **[locative] condition**

  - Direct locative supertypes: class

  Although `condition` is not `subtypep` of `class`, actual condition objects are commonly instances of a condition class that is a CLOS class. HyperSpec ISSUE:CLOS-CONDITIONS and ISSUE:CLOS-CONDITIONS-AGAIN provide the relevant history.

  Whenever a `class` denotes a `condition`, its `dref-locative-type` will be `condition`:

```
(dref 'locate-error 'class)
==> #<DREF LOCATE-ERROR CONDITION>
```

- **[locative] restart**

  A locative to refer to the definition of a restart defined by `define-restart`.

- **[macro] define-restart** *symbol lambda-list &body docstring*

  Associate a definition with the name of a restart, which must be a symbol. `lambda-list` should be what calls like `(invoke-restart '<symbol> ...)` must conform to, but this is not enforced.

  PAX "defines" standard CL restarts such as `use-value`(`0` `1`) with `define-restart`:

```
(dref 'use-value 'restart)
==> #<DREF USE-VALUE RESTART>
(assert (docstring *))
(assert (source-location **))
```

  Note that while there is a `cl:restart` class, its *instances* have no docstring or source location.

7.6 Locatives for Packages and Readtables

- **[locative] asdf/system:system**

Refers to an already loaded `asdf:system` (those in `asdf:registered-systems`). The name may be anything `asdf:find-system` supports.

`asdf:system` is not `exportable-locative-type-p`.

- **[locative] package**

  Refers to a `package`, defined by `defpackage` or `make-package`. The name may be anything `find-package` supports.

  `package` is not `exportable-locative-type-p`.

- **[locative] readtable**

  Refers to a named `readtable` defined with `named-readtables:defreadtable`, which associates a global name and a docstring with the readtable object. The name may be anything `find-readtable` supports.

  `readtable` references `resolve` to `find-readtable` on their name.

7.7 Locatives for Unknown Definitions

- **[locative] unknown** *dspec*

  This locative type allows PAX to work in a limited way with definition types it doesn't know. `unknown` definitions come from `definitions`, which uses `swank/backend:find-definitions`. The following examples show PAX stuffing the Swank dspec `(:define-alien-type double-float)` into an `unknown` locative on SBCL.

```
(definitions 'double-float)
==> (#<DREF DOUBLE-FLOAT CLASS>
-->  #<DREF DOUBLE-FLOAT (UNKNOWN (:DEFINE-ALIEN-TYPE DOUBLE-FLOAT))>)
```

```
(dref 'double-float '(unknown (:define-alien-type double-float)))
==> #<DREF DOUBLE-FLOAT (UNKNOWN (:DEFINE-ALIEN-TYPE DOUBLE-FLOAT))>
```

  `arglist` and `docstring` return `nil` for `unknown`s, but `source-location` works.

7.8 Locatives for DRef Constructs

- **[locative] dtype**

  - Direct locative subtypes: locative

  Locative for `dtypes` defined with `define-dtype` and `locative` types. `dtype` is to `locative` as `type` is to `class`.

  The `top` of the `dtype` hierarchy:

```
(dref 'top 'dtype)
==> #<DREF TOP DTYPE>
```

  This very definition:

```
(dref 'dtype 'locative)
==> #<DREF DTYPE LOCATIVE>
```

- **[locative]** **locative**

    - Direct locative supertypes: dtype

    This is the locative for locative types defined with `define-locative-type`, `define-pseudo-locative-type` and `define-locative-alias`.

```
(first-line (source-location-snippet
             (source-location (dref 'macro 'locative))))
=> "(define-locative-type macro ()"
```

- **[locative]** **lambda** *&key arglist arglist-type docstring docstring-package file file-position snippet &allow-other-keys*

    A pseudo locative type that carries its `arglist`, `docstring` and `source-location` in the locative itself. See `make-source-location` for the description of `file`, `file-position`, and `snippet`. `lambda` references do not `resolve`. The name must be `nil`.

```
(arglist (dref nil '(lambda :arglist ((x y) z)
                            :arglist-type :macro)))
=> ((X Y) Z)
=> :MACRO
```

```
(docstring (dref nil '(lambda :docstring "xxx"
                              :docstring-package :dref)))
=> "xxx"
==> #<PACKAGE "DREF">
```

```
(source-location-file
 (source-location (dref nil '(lambda :file "xxx.el"))))
=> "xxx.el"
```

    Also, see the `pax:include` locative.

8 Backends

On SBCL, the `swank` `asdf:system` is not a dependency of `dref/full` because DRef is able to function without it. However, when Swank is available in the Lisp image, DRef can use it to provide more precise `source-location`s.

On other Lisps, DRef curently gets `arglist`s and `source-location`s via Swank and `swank` is an ASDF dependency of `dref/full`.

- **[function]** **backend-available-p** *backend*

    Check if `backend` is currently available in the running Lisp. `backend` may be `:swank` or `nil`. Currently, on SBCL, both are available; on other Lisps, only `:swank` is.

- **[function]** **backend**

Return the backend being used. This is either `:swank` or `nil`. The default backend is `:swank` if it is available when first loading `dref` into the image, else it is `nil`.

Note that on Lisps other than SBCL, this is currently always `:swank`, which is available due to the ASDF dependency of `dref/full` on the `swank` `asdf:system`.

- **[setf]** **backend** *backend*

  `setf`ing `backend` to a `backend` that is not `backend-available-p` is an error.

- **[macro]** **with-backend** *(backend) &body body*

  Like `setf backend`, but only change `backend` during the dynamic extent of `body`.

9 Extending DRef

9.1 Extension Tutorial

Let's see how to tell DRef about new kinds of definitions through the example of the implementation of the `class` locative. Note that this is a verbatim `pax:include` of the sources. Please ignore any internal machinery. The first step is to define the locative type:

```
(define-locative-type class (type)
  "Naturally, CLASS is the locative type for [CLASS][class]es.

  ARGLIST returns the [compound type specifier][clhs] syntax
  associated with [system class][clhs]es. E.g. for the [INTEGER][clhs]
  class:

  ```cl-transcript (:dynenv dref-std-env)
  (arglist (find-class 'integer))
  => (&OPTIONAL LOWER-LIMIT UPPER-LIMIT)
  => :DEFTYPE
  ```")
```

Then, we make it possible to look up `class` definitions:

```
(define-locator class ((class class))
  (make-instance 'class-dref :name (class-name class) :locative 'class))

(define-lookup class (symbol locative-args)
  (unless (and (symbolp symbol)
               (find-class symbol nil))
    (locate-error "~S does not name a class." symbol))
  (make-instance 'class-dref :name symbol :locative 'class))
```

`define-locator` makes `(locate (find-class 'dref))` work, while `define-lookup` is for `(dref 'dref 'class)`. Naturally, for locative types that do not define first-class objects, the first method cannot be defined.

Finally, we define a `resolve*` method to recover the `class` object from a `class-dref`. We also specialize `docstring*` and `source-location*`:

```
(defmethod resolve* ((dref class-dref))
  (find-class (dref-name dref)))

(defmethod docstring* ((class class))
  (documentation* class t))

(defmethod source-location* ((dref class-dref))
  #+sbcl
  (sb-one-source-location (dref-name dref) :class)
  #-sbcl
  (swank-source-location* (resolve dref) (dref-name dref) 'class))

(defmethod arglist* ((dref class-dref))
  (clhs-type-specifier-arglist (dref-name dref)))
```

We took advantage of having just made the class locative type being `resolve`able, by specializing `docstring*` on the `class` class. `source-location*` was specialized on `class-dref` to demonstrate how this can be done for non-`resolve`able locative types.

Classes have no arglist, so no `arglist*` method is needed. In the following, we describe the pieces in detail.

9.2 Locative Type Hierarchy

Locative types form their own hierarchy, that is only superficially similar to the Lisp `class` hierarchy. The hierarchies of `lisp-locative-types` and `pseudo-locative-types` are distinct. That is, the `dref-class` of a Lisp locative type must not be a subclass of a `pseudo` one, and vice versa. This is enforced by `define-locative-type` and `define-pseudo-locative-type`.

- **[function]** **dref-class** *locative-type*

  Return the name of the `class` used to represent definitions with `locative-type`. This is always a subclass of `dref`. Returns `nil` if `locative-type` is not a valid locative type.

  Note that the actual `type-of` a `dref` is mostly intended for Extending DRef. Hence, it is hidden when a `dref` is printed:

  ```
  (dref 'print 'function)
  ==> #<DREF PRINT FUNCTION>
  (type-of *)
  => FUNCTION-DREF
  ```

  Due to Canonicalization, the actual type may be a proper subtype of `dref-class`:

  ```
  (dref 'documentation 'function)
  ==> #<DREF DOCUMENTATION GENERIC-FUNCTION>
  (type-of *)
  => GENERIC-FUNCTION-DREF
  (subtypep 'generic-function-dref 'function-dref)
  => T
  => T
  ```

- **[function]** **locative-type-direct-supers** *locative-type*

  List the locative types whose `dref-class`es are direct superclasses of the `dref-class` of `locative-type`. These can be considered supertypes of `locative-type` in the sense of `dtypep`.

  This is ordered as in the corresponding definition.

- **[function]** **locative-type-direct-subs** *locative-type*

  List the locative types whose `dref-class`es are direct subclasses of the `dref-class` of `locative-type`. These can be considered subtypes of `locative-type` in the sense of `dtypep`.

  This list is in reverse definition order.

- **[function]** **locative-subtype-p** *locative-type-1 locative-type-2*

  Check if `locative-type-1` is in the transitive closure of `locative-type-2` via `locative-type-direct-subs`. It is an error if `locative-type-1` or `locative-type-2` is not a valid locative type.

9.3 Defining Locative Types

- **[macro]** **define-locative-type** *locative-type-and-lambda-list locative-supertypes &optional docstring dref-defclass-form*

  Declare `locative-type` as a `locative`, which is the first step in Extending DRef.

  - *Simple example*

    To define a locative type called `dummy` that takes no arguments and is not a locative subtype of any other locative type:

    ```
    (define-locative-type dummy ()
      "Dummy docstring.")
    ```

    With this definition, only the locatives `dummy` and its equivalent form `(dummy)` are valid. The above defines a `dref(0 1)` subclass called `dummy-dref` in the current package. All definitions with locative type `dummy` and its locatives subtypes must be instances of `dummy-dref`.

    `(locate 'dummy 'locative)` refers to this definition. That is, `arglist`, `docstring` and `source-location` all work on it.

  - *Complex example*

    `dummy` may have arguments `x` and `y` and inherit from locative types `l1` and `l2`:

    ```
    (define-locative-type (dummy x &key y) (l1 l2)
      "Dummy docstring."
      (defclass dummy-dref ()
        ((xxx :initform nil :accessor dummy-xxx))))
    ```

One may change name of dummy-dref, specify superclasses and add slots as with defclass. Behind the scenes, the dref classes of l1 and l2 are added automatically to the list of superclasses.

Arguments:

- The general form of locative-type-and-lambda-list is (locative-type &rest lambda-list), where locative-type is a symbol, and lambda-list is a destructuring lambda list. The locative-args of drefs with locative type locative-type (the argument given to this macro) always conform to this lambda list. See check-locative-args.

  If locative-type-and-lambda-list is a single symbol, then that's interpreted as locative-type, and lambda-list is nil.

- locative-supertypes is a list of locative types whose dref-classes are prepended to the list of superclasses this definition.

Locative types defined with define-locative-type can be listed with lisp-locative-types.

- **[macro]** **define-pseudo-locative-type** *locative-type-and-lambda-list locative-supertypes &optional docstring dref-defclass-form*

  Like define-locative-type, but declare that locative-type does not correspond to definitions in the running Lisp. Definitions with pseudo locatives are of dtype pseudo and are not listed by default by definitions.

  Locative types defined with define-pseudo-locative-type can be listed with pseudo-locative-types.

- **[macro]** **define-locative-alias** *alias locative-type &body docstring*

  Define alias that can be substituted for locative-type (both symbols) for the purposes of locateing. locative-type must exist (i.e. be among locative-types). For example, let's define object as an alias of the class locative:

```
(define-locative-alias object class)
```

  Then, locateing with object will find the class:

```
(dref 'xref 'object)
==> #<DREF XREF CLASS>
```

  The locative-args of object (none in the above) are passed on to class.

```
(arglist (dref 'object 'locative))
=> (&REST ARGS)
=> :DESTRUCTURING
```

  Note that locative-aliases are not locative-types and are not valid dtypes.

  Also, see Locative Aliases in PAX.

9.3.1 Symbol Locatives

Let's see how the opaque `define-symbol-locative-type` and the obscure `define-definer-for-symbol-locative-type` macros work together to simplify the common task of associating definition with a symbol in a certain context.

- **[macro]** **define-symbol-locative-type** *locative-type-and-lambda-list locative-supertypes &optional docstring dref-class-def*

  Similar to `define-locative-type`, but it assumes that all things `locate`able with `locative-type` are going to be symbols defined with a definer defined with `define-definer-for-symbol-locative-type`. Symbol locatives are for attaching a definition (along with arglist, documentation and source location) to a symbol in a particular context. An example will make everything clear:

```
(define-symbol-locative-type direction ()
  "A direction is a symbol.")

(define-definer-for-symbol-locative-type define-direction direction
  "With DEFINE-DIRECTION, one can document what a symbol means when
  interpreted as a DIRECTION.")

(define-direction up ()
  "UP is equivalent to a coordinate delta of (0, -1).")
```

  After all this, `(dref 'up 'direction)` refers to the `define-direction` form above.

  The `dref-class` of the defined locative type inherits from `symbol-locative-dref`, which may be used for specializing when implementing new operations.

- **[macro]** **define-definer-for-symbol-locative-type** *name locative-type &body docstring*

  Define a macro with `name` that can be used to attach a lambda list, documentation, and source location to a symbol in the context of `locative-type`. The defined macro's arglist is `(symbol lambda-list &optional docstring)`. `locative-type` is assumed to have been defined with `define-symbol-locative-type`.

9.4 Extending `locate`

Internally, `locate` finds an initial `dref` of its `object` argument with a lookup or with a locator. This initial `dref` is then canonicalized with a series of casts. In more detail, the process is as follows.

- If the `object` argument of `locate` is a `dref`, then it is returned without processing.

Else, `locate` first needs to find the initial definition.

9.4.1 Initial Definition

`locate` can find the initial definition in one of two ways:

- *With direct lookup*

If `object` is an `xref`(`0 1`), then the lookup for (`xref-locative-type object`) is invoked. For an `xref` with the locative `(method (number))`, this would be the lookup defined as

```
(define-lookup method (name locative-args) ...)
```

- *With locator search*

  Else, `object` is a normal Lisp object, such as a `method` object from `find-method`. The first of `lisp-locative-types` whose locator succeeds provides the initial definition, which may be defined like this:

  ```
  (define-locator method ((obj method)) ...)
  ```

  This is a locator that returns definitions with the `method` locative type and takes an argument named `obj` of class `method` (which is like a specializer in `defmethod`).

  - `lisp-locative-types` are tried one by one in the order specified there.
  - For a given locative type, if there are multiple locators, standard CLOS method selection applies.

9.4.2 Canonicalization

The initial definition thus found is then canonicalized so that there is a unique definition under `xref=`:

```
(locate #'arglist*)
==> #<DREF ARGLIST* GENERIC-FUNCTION>
(dref 'arglist* 'function)
==> #<DREF ARGLIST* GENERIC-FUNCTION>
(dref 'arglist* 'generic-function)
==> #<DREF ARGLIST* GENERIC-FUNCTION>
```

Canonicalization is performed by recursively attempting to downcast the current definition to one of its `locative-type-direct-subs` in a depth-first manner, backtracking if a cast fails.

Default Downcast By default, downcasting to direct locative subtypes is performed by looking up the definition where the locative type is replaced with its sub while the name and the locative args remain the same.

Cast Name Change Casts must be careful about changing `dref-name`.

Their `dref` argument and the `dref` returned must have the same `dref-name` (under `equal`, see `xref=`) or it must be possible to upcast the returned value to the `dref` argument's `dref-locative-type`.

- *Implementation note*

  The purpose of this rule is to allow `dtypep` answer this correctly:

  ```
  (defclass foo ()
    ((a :accessor foo-a)))
  ```

```
(dref '(setf foo-a) '(method (t foo)))
==> #<DREF FOO-A (ACCESSOR FOO)>
(dtypep * '(method (t foo)))
=> T
;; Internally, DTYPEP upcast #<DREF FOO-A (ACCESSOR FOO)>
;; and checks that the locative args of the resulting
;; definition match those in (METHOD (T FOO)).
(locate* ** 'method)
==> #<DREF (SETF FOO-A) (METHOD (T FOO))>
```

For even more background, also note that if the name remains the same but locative args change, then `dtypep` can simply check with `dref` if there is a definition of the name with the given locative:

```
(defclass foo ()
  ((r :reader foo-r)))
(dref 'foo-r '(reader foo))
==> #<DREF FOO-R (READER FOO)>
(dtypep * '(method (foo)))
=> T
;; Behind the scenes, DTYPEP does this:
(xref= ** (dref 'foo-r '(method (foo))))
=> T
```

9.4.3 Defining Lookups, Locators and Casts

As we have seen, the Initial Definition is provided either by a lookup or a locator, then Canonicalization works with casts. Here, we look at how to define these.

Implementation note: All three are currently implemented as methods of generic functions with `eql` specializers for the locative type, which may easily prove to be problematic down the road. To make future changes easier, the generic function and the methods are hidden behind e.g. the `define-lookup` and `call-lookup` macros.

- **[variable]** ***check-locate*** *nil*

  Enable runtime verification of invariants during `locate` calls. This carries a performance penalty and is intended for testing and debugging.

  In particular, enforce the rule of Cast Name Change and that lookups, locators and casts obey the following:

  - The value returned must be either `nil` or a `dref`(`0` `1`). Alternatively, `locate-error`(`0` `1`) may be signalled.
  - If a `dref` is returned, then its `dref-locative-type` must be `locative-type`, and its class must be the `dref-class` of `locative-type`.
  - `locative-args` must be congruent with the destructuring lambda list in the definition of `locative-type`.

- **[macro]** **define-lookup** *locative-type (name locative-args) &body body*

Define a method of looking up definitions of `locative-type` with the given `locative-args`. Lookups are invoked by `locate` when its `object` argument is an `xref`(`0` `1`) with `locative-type` but it is not a `dref`(`0` `1`), as in the case of `(dref 'print 'function)`. When called, the variables `name` and `locative-args` are bound to `xref-name` and `xref-locative-args` of the `xref`. `locative-args` is validated with `check-locative-args` before `body` is evaluated.

```
(define-lookup variable (name locative-args)
  (unless (special-variable-name-p name)
    (locate-error))
  (make-instance 'variable-dref :name name :locative 'variable))
```

  - `locative-type` is a valid locative type.
  - `name` and `locative-args` are both `symbol`s.

  The above are enforced at macro-expansion time.

  - `body` must follow the rules in `*check-locate*`.

- **[macro]** **call-lookup** *name locative-type locative-args*

  Call the lookup for `locative-type` with `name` and `locative-args`.

- **[macro]** **define-locator** *locative-type ((object class)) &body body*

  Define a method of finding the definition with `locative-type` of instances of `class`. When a locator's `body` is evaluated, `object` is bound to such an instance.

```
(define-locator class ((class class))
  (make-instance 'class-dref :name (class-name class) :locative 'class))
```

  - `locative-type` is one of `lisp-locative-types`. This is because `pseudo-locative-types` never `resolve` to first-class objects.
  - `object` is a `symbol`.
  - `class` names a `class` that is not a subtype of `xref`. For how to convert definitions from one locative type to another, see `define-cast`.

  The above are enforced at macro-expansion time.

  - `body` must follow the rules in `*check-locate*`.

  In contrast to when the Initial Definition is created from an `xref` (see `define-lookup`), here `locative-args` are determined from `object`.

- **[macro]** **call-locator** *object locative-type*

  Call the locator for `locative-type` with `object`.

- **[macro]** **define-cast** *locative-type ((dref dref-class)) &body body*

  Define a method of converting a definition to another with `locative-type`. When a cast's `body` is evaluated, `dref` is bound to an instance `dref-class`, which denotes a valid but potentially non-canonical definition.

Note that the Default Downcast often suffices, and defining a cast is only necessary if the name or the locative args change:

```
(define-cast accessor ((dref reader-dref))
  (let ((name (dref-name dref))
        (class (second (dref-locative dref))))
    (when (ignore-errors (find-accessor-slot-definition name class))
      (make-instance 'accessor-dref :name name
                     :locative `(accessor ,class)))))
```

  - `locative-type` is a valid locative type.
  - If `locative-type` is one of `pseudo-locative-types`, then `dref-class` must be of another pseudo locative type.
  - `dref-class` is either a direct *downcast* or a potentially non-direct *upcast*.
    * *Downcast:* In this case, `locative-type` is one of `locative-type-direct-subs` of (`dref-class-to-locative-type dref-class`).

      Downcasting to non-direct subtypes is done in multiple steps. Consequently, the `body` of a downcast can rely on (`class-of dref`) being `class`, not any subclass thereof.
    * *Upcast:* `locative-type` is different but reachable from (`dref-class-to-locative-type dref-class`) by repeatedly choosing one of `locative-type-direct-supers`. Upcasting to non-direct supertypes is done in one step.

  The above are enforced at macro-expansion time.

  - `body` must follow the rules in `*check-locate*`, including those in Cast Name Change.

- **[macro]** **call-cast** *locative-type dref*

  Call the cast to `locative-type` with `dref`.

- **[function]** **locate-error** *&optional format-control &rest format-args*

  Call this function to signal a `locate-error` condition from the dynamic extent of a `locate` call, that is, from the `body`s of `define-lookup`, `define-locator` and `define-cast`. It is an error to call `locate-error` elsewhere.

  `format-control`, if non-`nil`, is a format control for which `format-args` are suitable.

- **[macro]** **check-locative-args** *locative-type locative-args*

  Signal a `locate-error` condition if `locative-args` do not match the `lambda-list` argument of `locative-type` (not evaluated).

9.5 Extending Everything Else

- **[generic-function]** **resolve*** *dref*

Return the object defined by the definition `dref` refers to. Signal a `resolve-error` condition by calling the `resolve-error` function if the lookup fails.

To keep `resolve` a partial inverse of `locate`, `define-locator` may be necessary for `resolve`able definitions. This function is for extending `resolve`. Do not call it directly.

It is an error for methods of this generic function to return an `xref`.

- **[function]** **resolve-error** *&rest format-and-args*

  Call this function to signal a `resolve-error` condition from the dynamic extent of a `resolve*` method. It is an error to call `resolve-error` elsewhere.

  `format-and-args`, if non-`nil`, is a format string and arguments suitable for `format`.

- **[generic-function]** **map-definitions-of-name** *fn name locative-type*

  Call `fn` with `dref`s which can be `locate`d with an `xref`(`0` `1`) with `name`, `locative-type` and some `locative-args`. The strange wording here is because there may be multiple ways (and thus `xref`s) that refer to the same definition.

  For most locative types, there is at most one such definition, but for `method`, for example, there may be many. The default method simply does `(dref name locative-type nil)` and calls `fn` with result if `dref` succeeds.

  `fn` is not called with the same (under `xref=`) definition multiple times.

- **[generic-function]** **map-definitions-of-type** *fn locative-type*

  Call `fn` with `dref`s that can be `locate`d with an `xref`(`0` `1`) with `locative-type` with some `name` and `locative-args`.

  - Return `nil` if all possibilities have been exhausted.
  - Return true if there may be definitions left unmapped. In this case, the caller is responsible for trying all interned symbols with `map-definitions-of-name`. The default method simply returns true.

  `fn` may be called with `dref`s that are `xref=` and differ only in their `dref-origin`.

- **[generic-function]** **arglist*** *object*

  To extend `arglist`, specialize `object` on a normal Lisp type or on a subclass of `dref`.

  `arglist` first calls `arglist*` with its `object` argument. If that doesn't work (i.e. the second value returned is `nil`), then it calls `arglist*` with `object` either `resolve`d (if it's a `dref`) or `locate`d (if it's not a `dref`).

  - The default method returns `nil`, `nil`.
  - There is also a method specialized on `dref`s, that looks up the `definition-property` called `arglist` and returns its value with `values-list`. Thus, an arglist and its kind can be specified with something like

    ```
    (setf (definition-property xref 'arglist)
          (list arglist :destructuring))
    ```

This function is for extension only. Do not call it directly.

- **[generic-function]** **arglist-parameters*** *arglist arglist-type*

  To extend `arglist-parameters`, *`eql(0 1)`-specialize `arglist-type` on, say, :BOA (see BOA lambda lists. This must be done if `arglist` is extended to return an unsupported value.*

  This function is for extension only. Do not call it directly.

- **[generic-function]** **docstring*** *object*

  To extend `docstring`, specialize `object` on a normal Lisp type or on a subclass of `dref`.

  `docstring` first calls `docstring*` with its `object` argument. If that doesn't work (i.e. `nil` is returned), then it calls `docstring*` with `object` either `resolve`d (if it's a `dref`) or `locate`d (if it's not a `dref`).

  - The default method returns `nil`.
  - A three-stage logic is implemented by an unspecialized `:around` method for docstrings and packages.
    * First, the primary method is called. If it returns a non-`nil` docstring and package, then these are returned by `docstring*`.
    * *Definition default*: The `definition-property` with indicator `docstring` of the `locate`d definition of `object` provides defaults for the docstring and/or the package, whichever are `nil`. These can be set for example as

      ```
      (setf (definition-property xref 'docstring)
            (list docstring *package*))
      ```

      Note that the docstring and the package or both may be `nil`.

    * *Package-wide default*: If the `dref-name` of the `locate`d definition is a symbol, then an irregular `definition-property` provides further defaults. To default the docstring package of all names of package `x` to package `y`, you could write

      ```
      (setf (definition-property `(:package ,(find-package :x)) 'docstring)
            (list "FIXME: Missing documentation." (find-package :y)))
      ```

  See Package and Readtable.

  This function is for extension only. Do not call it directly.

- **[generic-function]** **source-location*** *object*

  To extend `source-location`, specialize `object` on a normal Lisp type or on a subclass of `dref`.

  `source-location` first calls `source-location*` with its `object` argument. If that doesn't work (i.e. `nil` or `(:error <message>)` is returned), then it calls `source-location*` with `object` either `resolve`d (if it's a `dref`) or `locate`d (if it's not a `dref`).

  `source-location` returns the last of the `(:error <message>)`s encountered or a generic error message if only `nil`s were returned.

- The default method returns `nil`.
- There is also a method specialized on `drefs`, that looks up the `definition-property` called `source-location`. If present, it must be a function of no arguments that returns a source location or `nil`. Typically, this is set up in the defining macro like this:

```
(setf (definition-property xref 'source-location)
      (this-source-location))
```

This function is for extension only. Do not call it directly.

- **[macro]** **nth-value-or-with-obj-or-def** *(obj nth-value) &body body*

  Evaluate `body`. If its `nth-value` is `nil`, evaluate it again with `obj` bound to a `resolve`d object (if `obj` was a definition) or a definition (if `obj` was not a definition). This is intended for implementing new operations with the fallback mechanism of `arglist`, *`docstring`* and `source-location*`.

9.5.1 Definition Properties

Arbitrary data may be associated with definitions. This mechanism is used by `arglist*`, `docstring*` and `source-location*` for easy extension.

The following functions take an `xref` argument and not a `dref(0 1)` to allow working with non-canonical or non-existent definitions.

- **[function]** **definition-property** *obj indicator*

  Return the value of the property associated with `obj` whose name is `eql(0 1)` to `indicator`. The second return value indicates whether the property was found. `setf`able.

  - *Regular definition*: `obj` is commonly an `xref(0 1)`. `xref`s that are `xref=` are equivalent for the purposes of `definition-property`.
  - *Irregular case*: If `obj` is not an `xref` (and, by extension, not a `dref(0 1)`), it is equivalent to other objects to which it is `equal`.

  For example, see the `docstring*` generic-function, which uses both the regular and the irregular cases.

- **[function]** **delete-definition-property** *obj indicator*

  Delete the property `indicator` of `obj` established by setting `definition-property`. Return true if the property was found.

- **[function]** **definition-properties** *xref*

  Return the properties of `xref` as an association list.

- **[function]** **delete-definition-properties** *xref*

  Delete all properties associated with `xref`.

- **[function]** **move-definition-properties** *from-xref to-xref*

Associate all properties of `from-xref` with `to-xref`, as if readding them one-by-one with `(setf definition-property)`, and deleting them from `from-xref` with `delete-definition-property`.

9.6 `dref-class`es

These are the `dref-class`es corresponding to Basic Locative Types. They are exported to make it possible to go beyond the Basic Operations (e.g. `pax:document-object*`). For Defining Locative Types, they are not necessary, as `define-locative-type` handles inheritance automatically based on its `locative-supertypes` argument.

for Variables

- **[class]** **variable-dref** *dref*

  `dref-ext:dref-class` of `variable`.

- **[class]** **constant-dref** *variable-dref*

  `dref-ext:dref-class` of `mgl-pax:constant`.

for Macros

- **[class]** **macro-dref** *dref*

  `dref-ext:dref-class` of `mgl-pax:macro`.

- **[class]** **symbol-macro-dref** *dref*

  `dref-ext:dref-class` of `mgl-pax:symbol-macro`.

- **[class]** **compiler-macro-dref** *dref*

  `dref-ext:dref-class` of `compiler-macro`.

- **[class]** **setf-dref** *dref*

  `dref-ext:dref-class` of `setf`.

- **[class]** **setf-compiler-macro-dref** *compiler-macro-dref*

  `dref-ext:dref-class` of `dref:setf-compiler-macro`.

for Functions

- **[class]** **function-dref** *dref*

  `dref-ext:dref-class` of `function`.

- **[class]** **setf-function-dref** *function-dref setf-dref*

  `dref-ext:dref-class` of `dref:setf-function`.

- **[class]** **generic-function-dref** *function-dref*

  `dref-ext:dref-class` of `generic-function`.

- **[class]** **setf-generic-function-dref** *generic-function-dref setf-function-dref*

  `dref-ext:dref-class` of `dref:setf-generic-function`.
- **[class]** **method-dref** *dref*

  `dref-ext:dref-class` of `method`.
- **[class]** **setf-method-dref** *method-dref setf-dref*

  `dref-ext:dref-class` of `dref:setf-method`.
- **[class]** **method-combination-dref** *dref*

  `dref-ext:dref-class` of `method-combination`.
- **[class]** **reader-dref** *method-dref*

  `dref-ext:dref-class` of `mgl-pax:reader`.
- **[class]** **writer-dref** *method-dref*

  `dref-ext:dref-class` of `mgl-pax:writer`.
- **[class]** **accessor-dref** *reader-dref writer-dref setf-method-dref*

  `dref-ext:dref-class` of `mgl-pax:accessor`.
- **[class]** **structure-accessor-dref** *setf-function-dref function-dref*

  `dref-ext:dref-class` of `mgl-pax:structure-accessor`.

for Types and Declarations

- **[class]** **type-dref** *dref*

  `dref-ext:dref-class` of `type`.
- **[class]** **class-dref** *type-dref*

  `dref-ext:dref-class` of `class`.
- **[class]** **declaration-dref** *dref*

  `dref-ext:dref-class` of `declaration`.

for the Condition System

- **[class]** **condition-dref** *class-dref*

  `dref-ext:dref-class` of `condition`.
- **[class]** **restart-dref** *symbol-locative-dref*

  `dref-ext:dref-class` of `restart`.

for Packages and Readtables

- **[class]** **asdf-system-dref** *dref*

  `dref-ext:dref-class` of `asdf/system:system`.

- **[class]** **package-dref** *dref*

  `dref-ext:dref-class` of `package`.

- **[class]** **readtable-dref** *dref*

  `dref-ext:dref-class` of `readtable`.

for Unknown Definitions

- **[class]** **unknown-dref** *dref*

  `dref-ext:dref-class` of `mgl-pax:unknown`.

for DRef Constructs

- **[class]** **dtype-dref** *dref*

  `dref-ext:dref-class` of `dref:dtype`.

- **[class]** **locative-dref** *dtype-dref*

  `dref-ext:dref-class` of `mgl-pax:locative`.

- **[class]** **symbol-locative-dref** *dref*

  All locative types defined with `define-symbol-locative-type` inherit from this class.

- **[class]** **lambda-dref** *dref*

  `dref-ext:dref-class` of `lambda`.

9.7 Source Locations

These represent the file or buffer position of a defining form and are returned by the `source-location` function. For the details, see the Elisp function `slime-goto-source-location`.

- **[function]** **make-source-location** *&key file file-position buffer buffer-position snippet*

  Make a Swank source location. The ultimate reference is `slime-goto-source-location` in `slime.el`. When `snippet` is provided, the match nearest to `file-position` is determined (see the Elisp `slime-isearch` and `source-location-adjusted-file-position`).

- **[function]** **source-location-p** *object*

  See if `object` is a source location object.

- **[function]** **source-location-file** *location*

  Return the name of the file of the defining form. This may be `nil`, for example, if `location` is of a defining form that was entered at the REPL, or compiled in the `*slime-scratch*` buffer.

- **[function]** **source-location-file-position** *location*

  Return the file position of the defining form or `nil` if it's not available. The first position is 0.

- **[function]** **source-location-buffer** *location*

  Return the name of the Emacs buffer of the defining form or `nil` if there is no such Emacs buffer.

- **[function]** **source-location-buffer-position** *location*

  Return the position of the defining form in `source-location-buffer` or `nil` if it's not available. The first position is 1.

- **[function]** **source-location-snippet** *location*

  Return the defining form or a prefix of it as a string or `nil` if it's not available.

- **[function]** **source-location-adjusted-file-position** *location*

  Return the actual file position `location` points to allowing for some deviation from the raw `source-location-file-position`, which is adjusted by searching for the nearest occurrence of `source-location-snippet` in the file. The file is read using the same external format that ASDF would use to compile it. Needless to say, this can be a very expensive operation.

  If `source-location-file` is `nil`, `nil` is returned. If there is no snippet, or it doesn't match, then `source-location-file-position` (or 0 if that's `nil`) is returned.

  This is a non-interactive companion to the Elisp function `slime-location-offset`, supporting only file positions and non-partial matching of snippets.

- **[macro]** **this-source-location**

  The value of this macro form is a function of no arguments that returns its own `source-location`.

10 Indices

Referrer definition type abbreviations:

- *f*: for definitions in the function namespace (macros, compiler macros and also methods)
- *t*: `DEFTYPE`s, classes, conditions, structs
- *d*: documentation sections and glossary terms
- *l*: definitions of definition types
- *s*: ASDF systems
- *p*: packages
- *n*: named readtables
- *v*: special variables and constants
- *r*: restarts
- *?*: other

10.1 Function and Macro Index

10.2 Variable and Constant Index

10.3 Type Index

10.4 Misc Index

10.5 Concept Index