Debugging Tools for Garnet
Reference Manual

Roger B. Dannenberg
Andrew Mickish
Dario Giuse

October 1993

Abstract

Debugging a constraint-based graphical system can be difficult because critical interdependencies can be hard to visualize or even discover. The debugging tools for Garnet provide many convenient ways to inspect objects and constraints in Garnet-based systems.

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597. Additional support for Garnet has been provided by NEC, Apple, Adobe, and General Electric.

1. Introduction

This manual is intended for users of the Garnet system and assumes that the reader is familiar with Garnet. Other reference manuals cover the object and constraint system KR [Giuse 89], the graphics system Opal [Myers 89a], Interactors [Myers 89b] for handling keyboard and mouse input, Aggregadgets [Marchal 89] for making instances of aggregates of Opal objects.

1.1. Notation in this Manual

In the examples that follow, user type-in follows the asterisk (`*`), which is the prompt character in CMU Common Lisp on the RT. Function results are printed following the characters ‘‘`-->`’’. This is not what CMU Common Lisp prints, but is added to avoid confusion, since most debugging functions print text in addition to returning values:

```
* (some-function an-arg or-two)
some-function prints out this information,
    which may take several lines
--> function-result-printed-here
```

1.2. Loading and Using Debugging Tools

Normally, debugging tools will be loaded automatically when you load the file `garnet-loader.lisp`. Presently, the debugging tools are located in the files `debug-fns.lisp` and `objsize.lisp`. A few additional functions are defined in the packages they support.

Most of the debugging tools are in the `GARNET-DEBUG` package, and you should ordinarily type

```
(use-package "GARNET-DEBUG")
```

to avoid typing the package name when using these tools. Functions and symbols mentioned in this document that are *not* in the `GARNET-DEBUG` package will be shown with their full package name.

2. Inspecting Objects

2.1. Inspector

The `Inspector` is a powerful tool that can be of significan help in debugging. It pops up a window showing an object, and also shows the aggregate and is-a hierarchy for objects, and the dependencies for formulas. Various operations can be performed on objects and slots. In general, the `Inspector` is quite useful for debugging programs, and provides interfaces to many of the other debugging functions in Garnet. A view of an object being inspected in shown in Figure 2-1.

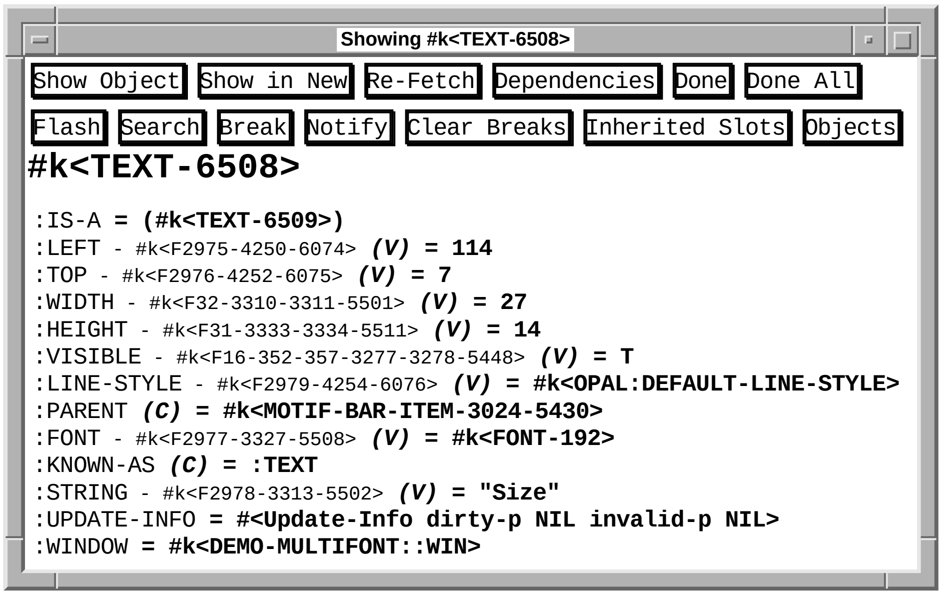

Figure 2-1: The Inspector showing a text object.

The `Inspector` is loaded automatically when you load the debugging tools which is enabled by default in garnet-loader, but it can also be loaded explicitly using `(garnet-load "debug:inspector")` The `Inspector` is in the `garnet-debug` package.

An example of using the Inspector is included in the Tutorial at the beginning of this Reference Manual.

2.1.1. Invoking the Inspector

There are a number of ways to inspect objects. The easiest is to put the mouse over an object and hit the `HELP` keyboard key. This will print a message in the Lisp listener window and pop up a window like Figure 2-1. If you want to use the `HELP` keyboard key for something else, you can set the variable `garnet-debug:*inspector-key*` to a different key (or NIL for none) *before* loading the `Inspector`.

Alternatively, you can explicitly invoke the `Inspector` on an object using either

`garnet-debug:Inspector` *obj* [*Function*]
`gd:Inspector` *obj*

(`gd` is an abbreviation for `garnet-debug`).
To inspect the *next* interactor that runs, you can hit `CONTROL-HELP` on the keyboard (the mouse position

is irrelevant), or call the function

`gd:Inspect-Next-Inter` [*Function*]

Hitting `CONTROL-HELP` a second time before an interactor runs will cancel the invocation of the `Inspector`. To change the binding of this function, set the variable `gd:*inspector-next-inter-key*` *before* loading the `Inspector`.

By default, `SHIFT-HELP` is bound to a little function that simply prints the object under the mouse to the Lisp Listener, and does not invoke the `Inspector`. Example output from it is:

```
--> (24,96) = #k<MULTIFONT-LINE-1447> in window #k<INTERACTOR-WINDOW-1371>

--> No object at (79,71) in window #k<INTERACTOR-WINDOW-1371>
```

To change the binding of this function, use the variable `gd:*show-object-key*`.

2.1.2. Schema View

The schema view shown in Figure 2-1 tells all the local slots of an object. To see the inherited slots also, click on the `Inherited Slots` button. For each slot, the display is:

- The slot name.
- A ***(C)*** if the slot is constant.
- An ***(I)*** if the slot is inherited.
- The formula for the slot, if any.
- If there is a formula, then a ***(V)*** if the slot value is valid, otherwise a ***(IV)*** for invalid.
- The current value of the slot, which may wrap to multiple lines if the value is long.
- The entire line is red if the slot is a *parameter* to the object (if it is in the `:parameters` list), otherwise the line is black.

If the object's values change while it is being inspected, the view is *not* updated automatically. To see the current value of slots, hit the "`Re-Fetch`" button.

To change the value of a slot of an object, click in the value part of the slot (after the =), and edit the value to the desired value and hit return. The object will immediately be updated and the `Inspector` display will be re-fetched. If you change your mind about editing the value before hitting return, simply hit `control-g`. If you try to set a slot which is marked constant, the `Inspector` will go ahead and set the slot, but it gives you a warning because often dependencies based on the slot will no longer be there, so the effect of setting the slot may not work.

If a slot's value is an object and you want to inspect that object, or if you want to inspect a formula, you can double-click the left button over the object name, and hit the "`Show Object`" button. Also, you can use the "`Show in New`" button if you want the object to be inspected in a new window.

2.1.3. Object View

Hitting the "`Objects`" button brings up the view in Figure 2-2. This view shows the name of the object being inspected at the top, then the `is-a` hierarchy. In Figure 2-2, `TEXT-6509` is the immediate prototype of the inspected object (`TEXT-6508`), and `TEXT-7180` is the prototype of `TEXT-6509`, and so on. The next set of objects shows the aggregate hierarchy. Here, `TEXT-6508` is in the aggregate `MOTIF-BAR-ITEM-3024-5430`, etc. The last item in this list is always the window that the object is in (even though that is technically not the `:parent` of the top-level aggregate). The final list is simply the list of objects that have been viewed in this window, which forms a simple history of views.

To return to the schema view of the current object, use the "`Re-Fetch`" button. You can double-click on any object and use "`Show Object`" or "`Show in New`" to see its fields, or you can hit "`Objects`" to go to the object view of the selected object.

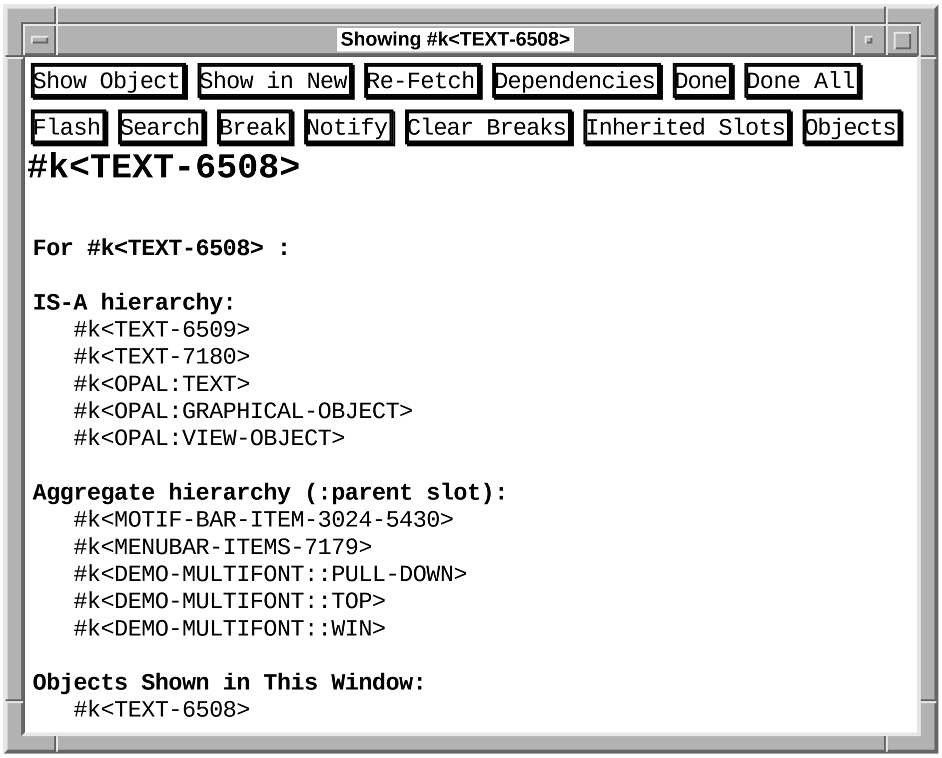

Figure 2-2: The Inspector showing the objects related to the text object of Figure 2-1.

2.1.4. Formula Dependencies View

If you select a formula or a slot name (by double-clicking on it) and then hit the "`Dependencies`" button, you get the view of Figure 2-3. This slows the slots used in calculating the value in the formula.

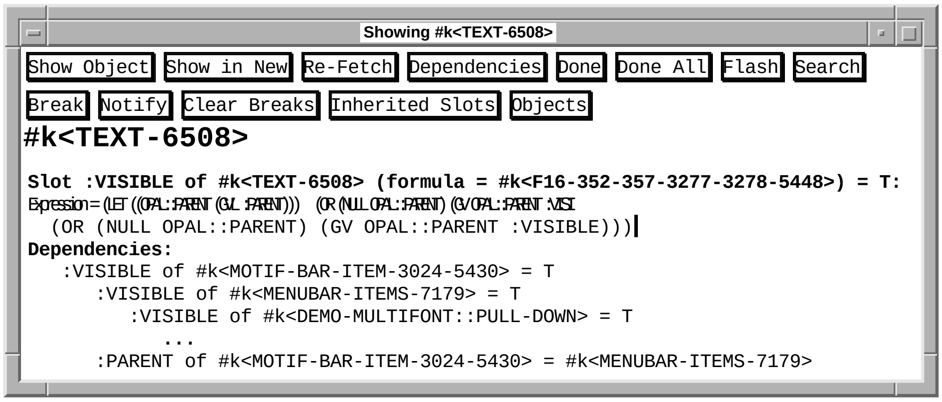

Figure 2-3: The Inspector showing the dependencies of the `:visible` slot of the object shown in Figure 2-1.

The first lines show the object, the slot, the formula name, and the expression of the formula. Then the dependencies are shown. The outer-most level of indenting are those slots that are immediately used by the formula. In this case, the :visible slot of #k<MOTIF-BAR-ITEM-3024-5430>. Note that only non-constant slots are shown, which is why the :parent slot of TEXT-6508 is not listed (it is constant). Indented underneath each slot are the slots it depends on in turn, so :visible of #k<MOTIF-BAR-ITEM-3024-5430> depends on its :parent and its parent's :visible. The "..." means that there are more levels of dependencies. To see these, you can double click on the "..." or on any slot name and hit the "Dependencies" button.

2.1.5. Summary of Commands

double-clicking the left button on an object or slot will select it, and it will then be the parameter for further commands.

single-clicking the left button after the = will let you edit the value. Hit return to set the value or control-g to abort.

Show Object - displays the selected object in the same window.

Show in New - displays the selected object in a new window.

Re-Fetch - redisplay the current object, and re-fetch the values of all slots, in case any have changed. This command is also used to get back to the schema view from the object or dependency views.

Dependencies - when a formula or a slot containing a formula is selected, then shows the slots that are used to calculate it (see section 2.1.4).

Done - get rid of this Inspector window.

Done All - get rid of all Inspector windows.

Flash - if an object is selected, flash it, otherwise flash the current object being inspected. The object is flashed by bringing its window to the top and putting an XOR rectangle over it (using the function gd:flash).

Search - Find a slot of the object and display it at the top of the list. This helps you find slots in a long list, and it will find inherited slots, so you don't have to hit Inherited Slots and get the whole list when you are only interested in one slot. After hitting the Search button, you will be prompted for the slot name, and you can type in a few letters, hit RETURN, and the Inspector will try to fill out the name based on all current slots of the object.

Notify - If a slot is selected, then will print a message in the Inspector *and* in the Lisp Listener window whenever the selected slot of the object is set. If no slot is selected, then will print a message whenever *any* slot of the object is set. You can be waiting for a Notify or Break on multiple slots of multiple objects at the same time. Note that execution is much slower when there are *any* Breaks or Notifies in effect.

Break - If a slot is selected, then will break into the debugger whenever the selected slot of the object is set. If no slot is selected, then will break into the debugger whenever *any* slot of the object is set. You should go to your Lisp Listener window to handle the break, and then *continue* from the break (rather than aborting or popping from the break). The Inspector will not operate while you are in the debugger unless you type (inter:main-event-loop).

Clear Breaks - Clear all the breaks and notifies. All Breaks and Notifies are also cleared when you hit the Done or Done All buttons. There is no interface for clearing a single break or notify.

Inherited Slots - Toggle the display of inherited slots in the schema view.

Objects - Switch to the object view that shows the is-a and aggregate hierarchy (see section 2.1.3).

2.2. PS -- Print Schema

The same information that is shown in the Inspector for an object's slots and values can be printed with the simpler `kr:ps`. This function does not create a new window to show the information, but instead prints right into the lisp listener.

`kr:PS` *object* `&key` *types-p all-p* `(`*control* `T) (`*inherit* `NIL)`
`(`*indent* `0) (`*stream* `*standard-output*)` [*Function*]

(All the nuances of this function are described in the KR manual.)

2.3. Look, What, and Kids

For quick inspection of objects, the `look`, `what`, and `kids` functions may be used:

`gd:Look` *object* `&optional (`*detail* `2)` [*Function*]
`gd:What` *object* [*Function*]
`gd:Kids` *object* [*Function*]

The `look` function prints out varying amounts of information about an object, depending upon the optional argument *detail*:

- `(look obj 0)` prints a one-line description of `obj`. This is equivalent to calling `(what obj)`.
- `(look obj 1)` prints a one-line description of `obj` and also shows the immediate components of `obj` if it is an aggregate. This form is equivalent to calling `(kids obj)`.
- `(look obj 2)` recursively prints all components of `obj`. This is the default, equivalent to typing `(look obj)`. Use it to look at the structure of an aggregate.
- `(look obj 3)` prints slots of `obj`, using `ps`, and then prints the tree of components.
- `(look obj 4)` prints slots of `obj` and its immediate components. Any (trees of) sub-components are also printed.
- `(look obj 5)` prints what is essentially complete information about a tree of objects, including all slots of all components.

For example,

```
* (what mywindow)
#k<MYWINDOW> is-a #k<INTERACTOR-WINDOW> (WINDOW)
--> NIL
* (look mywindow)
#k<MYWINDOW> is-a #k<INTERACTOR-WINDOW> (WINDOW)
   #k<MYAGG> is-a #k<AGGREGATE> (VIEW-OBJECT)
      #k<MYRECT> is-a #k<MOVING-RECTANGLE> (RECTANGLE)
      #k<MYTEXT> is-a #k<CURSOR-MULTI-TEXT> (MULTI-TEXT)
--> NIL
```

2.4. Is-A-Tree

Look prints the parent of the object and then the ‘‘standard parent’’ of the object's parent in parentheses. The ‘‘standard parent’’ is the first named object encountered traveling up the `:is-a` tree. If `look` does not print enough information about an object, the `is-a-tree` function might be useful:

`gd:Is-A-Tree` [*Function*]

This function traces up `:is-a` links and prints the resulting tree:

```
* (is-a-tree mytext)
#k<MYTEXT> is-a
   #k<CURSOR-MULTI-TEXT> is-a
      #k<MULTI-TEXT> is-a
         #k<TEXT> is-a
            #k<GRAPHICAL-OBJECT> is-a
               #k<VIEW-OBJECT>
--> NIL
```

2.5. Finding Graphical Objects

It is often necessary to locate a graphical object or figure out why a graphical object is not visible. The function

`gd:Where` *object* [*Function*]

prints out the *object*'s `:left`, `:top`, `:width`, `:height`, and `:window` in a one-line format.

```
* (where mywindow)
#k<MYWINDOW> :TOP 43 :LEFT 160 :WIDTH 355 :HEIGHT 277
--> NIL
* (where myagg)
#k<MYAGG> :TOP 20 :LEFT 80 :WIDTH 219 :HEIGHT 150 :WINDOW #k<MYWINDOW>
--> NIL
```

If you are not sure which screen image corresponds with a particular Opal object, use the following function:

`gd:Flash` *object* [*Function*]

The `flash` function will invert the bounding box of *object* making the object flash on and off. `flash` has two interesting features:

1. You can `flash` aggregates, which are otherwise invisible.
2. If the object is not visible, `flash` will try hard to tell you why not. Possible reasons include:
    - The object does not have a window,
    - The window does not have an aggregate,
    - The object is missing a critical slot (e.g. `:left`),
    - The object is outside of its window,
    - The object's `:visible` slot is `nil`,
    - The aggregate containing the object is not visible, or
    - The object is outside of its aggregate (a problem with the aggregate).

`Flash` does not test to see if the object is obscured by another window. If `flash` does not complain and you do not see any blinking, use `where` to find the object's window. Then use `where` (or `flash`) applied to the window to locate the window on your screen. Bring the window to the front and try again.

The `invert` function is similar to `flash`, but it leaves the object inverted. The `uninvert` function will undo the effect of `invert`:

`gd:Invert` *object* [*Function*]
`gd:Uninvert` *object* [*Function*]

`invert` uses a single Opal rectangle to invert an area of the screen. If the rectangle is in use, it is first removed, so at most one region will be inverted at any given time. Unlike, `flash`, `invert` depends upon Opal, so if Opal encounters problems with redisplay, `invert` will not work (see `fix-up-window` in Section 6).

The previous functions are only useful if you know the name of a graphical object. To obtain the name of an object that is visible on the screen, use:

`gd:Ident` [*Function*]

`Ident` waits for the next input event and reports the object under the mouse at the time of the event. In addition to printing the leaf object under the mouse, `ident` runs up the `:parent` links and prints the chain of aggregates up to the window. Some interesting features to note are:

- `ident` will report a window if you do not select an object.

- `ident` returns a list[1] in the form (*object window x y code*) so you can then use the selection in another expression, e.g. `(kr:ps (car (ident)))`. *Object* will be `nil` if none was selected.
- `ident` also prints the input event and mouse location. For instance, use `ident` if you want to know the Lisp name for the character transmitted when you type the key labeled ‘‘Home’’ on your keyboard or to tell you the window coordinates of the mouse.

Another way to locate a window is to use the function:

`gd:Windows` [*Function*]

which prints a list of Opal windows and their locations. The list of windows is returned. Only mapped windows are listed, so `windows` will only report a window that has been `opal:update`’d. For example:

```
* (windows)
#k<MYWINDOW> :TOP 43 :LEFT 160 :WIDTH 355 :HEIGHT 277
#k<DEMO-GROW::VP> :TOP 23 :LEFT 528 :WIDTH 500 :HEIGHT 300
--> (#k<DEMO-GROW::VP> #k<MYWINDOW>)
```

[1] A list is returned rather than a multiple value because multiple values print out on multiple lines in CMU Common Lisp, taking too much screen space when `ident` is used interactively.

3. Inspecting Constraints

Formulas often have unexpected values, and program listings do not always help when formulas and objects are inherited and/or created at run time. To make dependencies explicit, the `explain-slot` function can be used:

`gd:Explain-Slot` *object slot* [*Function*]

`explain-slot` will track down all dependencies of *object*'s *slot* and prints them. Indirect dependencies that occur when a formula depends upon the value of another formula are also printed. The complete set of dependencies is a directed graph, but the printout is tree-structured, representing a depth-first traversal of the graph. The search is cut off whenever a previously visited node is encountered. This can represent either a cycle or two formulas with a common dependency.

In the following example, the `:top` of `mytext` depends upon the `:top` of `myrect` which in turn depends upon its own `:box` slot:

```
* (explain-slot mytext :top)
#k<MYTEXT>'s :TOP is #k<F2449> (20 . T),
which depends upon:
   #k<MYRECT>'s :TOP is #k<F2439> (20 . T),
   which depends upon:
      #k<MYRECT>'s :BOX is (80 20 100 150)
--> NIL
```

When `explain-slot` is too verbose, a non-recursive version can be used:

`gd:Explain-Short` *object slot* [*Function*]

For example:

```
* (explain-short mytext :top)
#k<MYTEXT>'s :TOP is #k<F2449> (20 . T),
which depends upon:
   #k<MYRECT>'s :TOP is #k<|1803-2439|> (20 . T),
      ...
--> NIL
```

Warning: `explain-slot` and `explain-short` may produce incorrect results in the following ways:

- Both `explain-slot` and `explain-short` rely on dependency pointers maintained for internal use by KR. In the present version, KR sometimes leaves dependencies around that are no longer current. This is not a bug because, at worst, extra dependencies only cause formulas to be reevaluated unnecessarily. However, this may cause `explain-slot` or `explain-short` to print extra dependencies.

- Formulas may access slots but not use the values. This will create the appearance of a dependency when none actually exists.

- Formulas that that try to follow a null link, e.g. `(gv :self :feedback-obj :top)` where `:feedback-obj` is `nil`, may be marked as invalid and have their dependency lists cleared. `explain-slot` and `explain-short` will detect this case and warn you if it happens.

4. Choosing Constant Slots

Since the use of constants can significantly reduce the storage requirements and execution time of an application, we have provided several new functions that help you to choose which slots should be declared constant. The following functions are used in conjunction to identify slots that are candidates for constant declarations.

4.1. Suggest-Constants

`gd:Record-From-Now` [*Function*]

`gd:Suggest-Constants` *object* `&key` *max* (*recompute-p* `T`) (*level* `1`) [*Function*]

To use these functions, bring up the application you want to analyze, and execute `record-from-now`. Exercise all the parts and gadgets of the interface that are expected to be operated during normal use, and then call `suggest-constants`. Information will be printed out that identifies slots which, if declared constant, would cause dependent formulas to be replaced by their actual values.

Keep in mind that it is usually not necessary to declare every reported slot constant. Many formulas will <u>become</u> constant if they depend on constant slots. For example, declaring many of the parameters of a `gg:text-button-panel` constant in the top-level gadget is sufficient to eliminate the internal formulas that depend on them.

Also, it is important to exercise <u>all</u> parts of the application in order to get an accurate list of constant slot candidates. If you forget to operate a certain button while recording, slots may be suggested that would cause the button to become inoperable, since `suggest-constants` would assume it was a static object.

`Suggest-constants` will tell you if a potential slot is in the object's `:maybe-constant` list. When the slot is in this list, then it can be declared constant by supplying the value of T in the `:constant` list. As you add constants, though, you may want to carefully name each slot individually in the `:constant` list to avoid erroneous constant declarations.

The parameters to `suggest-constants` are used as follows:

object - This can be any Garnet object, but it is usually a window or its top-level aggregate. The function examines formulas in *object* and all its children.

max - This parameter controls how many constant slot candidates are printed out. The default is to print all potential constant slots that are found in *object* and all its children.

recompute-p - Set this parameter to NIL if you do not need to reexamine all the objects and you trust what was computed earlier (the same information that was printed out before will be printed out again, without checking that it is still valid).

level - The default value of *level*, which is 1, causes the function to print only slots which would, by themselves, eliminate some formula. If *level* is made higher, slots will be printed that may not eliminate formulas by themselves, but will at least eliminate some dependencies from the formulas that remain.

For example, consider a formula that depends on slots A and B. Declaring constant either A or B alone would not eliminate the formula, so with *level* set to 1, slots A and B would not be suggested by `suggest-constants`. Setting *level* to 2, however, will printe both A and B, since the combination of the two slots would indeed eliminate a formula. Higher values of *level* make `suggest-constants` print out formulas that are less and less likely to eliminate formulas.

4.2. Explain-Formulas and Find-Formulas

`gd:Explain-Formulas` *aggregate* `&optional (`*limit* `50)` *eliminate-useless-p* [*Function*]

`Explain-formulas` is used to analyze all the formulas that were not evaluated since the last call to `record-from-now`. These formulas might have been evaluated when the application was first created, to position the objects appropriately, but are not a dynamic part of the interface, and are thus candidates for constant declarations. If the *eliminate-useless-p* option is non-NIL, then formulas that are in fact unnecessary (i.e., would go away if they were recomputed) are actually eliminated immediately.

`gd:Find-Formulas` *aggregate* `&optional (`*only-totals-p* `T) (`*limit* `50)` *from* [*Function*]

If the function `find-formulas` is called with a non-NIL *only-totals-p* option, it will print out the total number of formulas that have not been reevaluated since the last call to `record-from-now`. If *only-totals-p* is NIL and *limit* is specified, it will print out at most *limit* formula names. If *limit* is NIL, all formula names will be printed out.

You will seldom need to specify the *from* parameter. This allows you to print out formulas that have been unevaluated since *from*. The default value is the number returned by the last call to `record-from-now`; specifying a smaller number reduces the number of formulas that are printed out, since formulas that were evaluated earlier are discarded.

4.3. Count-Formulas and Why-Not-Constant

`gd:Count-Formulas` *object* [*Function*]

`Count-formulas` will print a list of all existing formulas in *object* and all its children. It is important to note that formulas are not copied down into an object until they are specifically requested by a `g-value` or `gv` call. Thus, you may not get an accurate count of the real number of formulas in an object until you exercise the object in its intended way. For example, if a prototype `A` has a formula in its `:left` slot and you count the formulas in `B`, an instance of `A`, before asking for `B`'s `:left` slot, then `B`'s `:left` formula will not be counted, because it has not been copied down yet.

`gd:Why-Not-Constant` *object slot* [*Function*]

This function is extremely useful when you are trying to get rid of formulas by declaring constant slots. If `count-formulas` tells you that formulas still exist in your application that you think should go away due to propagation of constants, then you can call `why-not-constant` on a particular slot to find out what its formula depends on. The function will print out a list of dependencies for the formula in the *slot*, which will give you a hint about what other slot could be declared constant to make this formula go away.

5. Noticing when Slots are Set

It is often useful to be notified when a slot of an object is set, so now we provide a set of debugging functions that do this. There is also an interface to these functions through the `Inspector` (section 2.1) which makes them more convenient to use.

Note that the implementation of this is *very* inefficient and is intended only for debugging. Don't use this as general-purpose demon technique since a search is performed for *every* formula evaluation and every slot setting when any notifies or breaks are set.

`gd:Notify-On-Slot-Set &key` *object slot value* [*Function*]

This will print out a message in the Lisp Listener window whenever the appropriate slot is set. If a value is supplied, then only notifies when the slot is set to that particular value. If an object is provided, then only notifies when a slot of that object is set. If no object is supplied, then notifies whenever *any* object is set. If a slot is provided, then only notifies when that slot is set. If no slot is supplied, then notifies whenever *any* slot is set. If object is NIL, then clears all breaks and notifies. If all parameters are missing, then shows current status. For example,

```
(gd:Notify-On-Slot-Set :object obj :slot :left) ;notify when :left of obj set
(gd:Notify-On-Slot-Set :object obj :slot :left :value 0) ;notify when :left of obj set to 0
(gd:Notify-On-Slot-Set :value NIL) ;notify when any slot of any obj set to NIL
(gd:Notify-On-Slot-Set :object obj) ;notify when any slot of obj set
```

Each call to `Notify-On-Slot-Set` adds to the previous list of breaks and notifies, unless the object is NIL. You can use `clear-slot-set` to remove a break or notify (see below).

`gd:Break-On-Slot-Set &key` *object slot value* [*Function*]

Same as `Notify-On-Slot-Set`, but breaks into the debugger when the appropriate slot is set.

`gd:Call-Func-On-Slot-Set` *object slot value fnc extra-val* [*Function*]

This gives you more control, since you get to supply the function that is called when the appropriate slot is set. The parameters here are not optional, so if you don't want to specify the object, slot or value, use the special keyword `:*any*`. The function `fnc` is called as:

```
(lambda (obj slot val reason extra-val))
```

where the *slot* of *obj* is being set with *val*. The *reason* explains why the slot is being set and will be one of :S-VALUE, :FORMULA-EVALUATION, :INHERITANCE-PROPAGATION or :DESTROY-SLOT. *Extra-val* can be anything and is the same value passed into `Call-Func-On-Slot-Set`.

`gd:Clear-Slot-Set &key` *object slot value* [*Function*]

Clear the break or notify for the object, slot and value. If nothing is specified or object is NIL, then clears all breaks and notifies.

6. Opal Update Failures

Opal assumes that graphical objects have valid display parameters such as `:top` or `:width`. If a parameter is computed by formula and there is a bug, the problem will often cause an error within Opal's `update` function.

A "quarantine slot" named `:in-progress` exists in all Garnet windows. If there was a crash during the last update of the window, then the window will stop being updated automatically along with the other Garnet windows, until you can fix the problem and update the window successfully. The quarantine slot is discussed in detail in the Opal Manual.

There are several ways to proceed after an update failure. The first and easiest action is to run `opal:update` with the optional parameter `t`:

```
(opal:update window t)
```

This forces `opal:update` to do a complete update of `window` as opposed to an incremental update. This may fix your problem by bringing all slots up-to-date and expunging previous display parameters.

Another possibility is, after entering the debugger, call

`gd:Explain-NIL` [*Function*]

This functionwill check to see if a formula tried to follow a null link (a typical cause of Opal object slots becoming `nil`). If so, the object and slot associated with the formula will be printed followed by objects and slots on which the formula depends[2]. One of the slots depended upon will be the null link that caused the formula to fail.

Warning: `explain-nil` will always attempt to describe the last formula that failed due to a null link *since the last time* `explain-nil` *was evaluated*. This may or may not be relevant to the bug you are searching for. The last error is cleared every time `explain-nil` is evaluated to reduce confusion over old errors. If there has been no failure, `explain-nil` will print

```
No errors in formula evaluation detected
```

A third possibility is to run

`gd:Fix-Up-Window` *window* [*Function*]

on the window in question. (You may want to use `windows` to find the window object.) `fix-up-window` will do type checking without attempting a redisplay. If an error is detected, `fix-up-window` will allow you to interactively remove objects with problems from the window.

After fixing the problem that caused `update` to crash, you should be able to do a successful total update on the window (discussed above). A successful total update will clear the quarantine slot, and will allow interactions to take place in the window normally.

[2]`Explain-nil` does not use the same technique for finding dependencies as `explain-slot`, which uses forward pointers from the formula's `:depends-on` slot. Since `:depends-on` is currently cleared when a null link is encountered, `explain-nil` uses back pointers from the objects back to the formula. These are in the `:depended-upon` slot of objects. To locate the back pointers, `explain-nil` searches for all components of all Opal windows. Only objects in windows are searched, so dependencies on non-graphical objects will be missed.

7. Inspecting Interactors

7.1. Tracing

A common problem is to create some graphical objects and an interactor but to discover that nothing happens when you try to interact with the program. If you know what interactor is not functioning, then you can trace its behavior using the function

`inter:Trace-Inter` *interactor* [*Function*]

This function enables some debugging printouts in the interactors package that should help you determine what is wrong. A set of things to trace is maintained internally, so you can call `inter:trace-inter` several times to trace several things. In addition to interactors, the parameter can be one of:

- `t` — trace everything.
- `NIL` — untrace everything, same as calling `inter:untrace-inter`.
- `:window` — trace things about interactor windows such as `create` and `destroy` operations.
- `:priority-level` — trace changes to priority levels.
- `:mouse` — trace `set-interested-in-moved` and `ungrab-mouse`.
- `:event` — show all events that come in.
- `:next` — start tracing when the next interactor runs, and trace that interactor.
- `:short` — report only the name of the interactor that runs, so that the output is much less verbose. This is very useful if you suspect that more than one interactor is accidentally running at a time.

Tracing any interactor will turn on `:event` tracing by default. Call `(inter:untrace-inter :event)` (see below) to stop `:event` tracing.

Just typing

```
(inter:trace-inter)
```

will print out the interactors currently being traced.

`inter:Untrace-Inter` *interactor* [*Function*]

can be used to selectively stop tracing a single interactor or other category. You can also pass `t` or `nil`, or no argument to `untrace` to stop all tracing:

```
(inter:untrace-inter)
```

7.2. Describing Interactors

If you are not debugging a particular interactor, there are a few ways to proceed other than wading through a complete interactor trace. First, you can find out what interactors are active by calling:

`gd:Look-Inter &optional` *interactor-or-object detail* [*Function*]

The parameter *interactor-or-object* can be:

- NIL to list all active interactors (see below),
- an interactor to describe,
- a window, to list all active interactors on that window,
- an interactor priority-level, to list all active interactors on that level,
- a graphical object, to try to find all interactors that affect that object,

- `:next` to wait and describe the next interactor that runs

With no arguments (or NIL as an argument), `look-inter` will print all active interactors (those with their `:active` and `:window` slots set to something) sorted by priority level :

```
* (look-inter)
Interactors that are :ACTIVE and have a :WINDOW are:
Level #k<RUNNING-PRIORITY-LEVEL>:
Level #k<HIGH-PRIORITY-LEVEL>: #k<DEMO-GROW::INTER2>
Level #k<NORMAL-PRIORITY-LEVEL>: #k<MYTYPER> #k<MYMOVER> #k<DEMO-GROW::INTER3>
 #k<DEMO-GROW::INTER4> #k<DEMO-GROW::INTER1>
--> NIL
```

If *detail* is 1, `look-inter` will show the `:start-event` and `:start-where` of each active interactor:

```
* (look-inter 1)
Interactors that are :ACTIVE and have a :WINDOW are:
Level #k<RUNNING-PRIORITY-LEVEL>:
Level #k<HIGH-PRIORITY-LEVEL>: #k<DEMO-GROW::INTER2>
Level #k<NORMAL-PRIORITY-LEVEL>: #k<MYTYPER> #k<MYMOVER> #k<DEMO-GROW::INTER3>
 #k<DEMO-GROW::INTER4> #k<DEMO-GROW::INTER1>
#k<DEMO-GROW::INTER2> (MOVE-GROW-INTERACTOR)
   starts when :LEFTDOWN (:ELEMENT-OF #k<AGGREGATE-164>)
#k<MYTYPER> (TEXT-INTERACTOR)
   starts when :RIGHTDOWN (:IN #k<MYTEXT>)
#k<MYMOVER> (MOVE-GROW-INTERACTOR)
   starts when :LEFTDOWN (:IN #k<MYRECT>)
#k<DEMO-GROW::INTER3> (MOVE-GROW-INTERACTOR)
   starts when :MIDDLEDOWN (:ELEMENT-OF #k<AGGREGATE-136>)
#k<DEMO-GROW::INTER4> (MOVE-GROW-INTERACTOR)
   starts when :RIGHTDOWN (:ELEMENT-OF #k<AGGREGATE-136>)
#k<DEMO-GROW::INTER1> (BUTTON-INTERACTOR)
   starts when :LEFTDOWN (:ELEMENT-OF-OR-NONE #k<AGGREGATE-136>)
--> NIL
```

To get information about a single interactor, pass the interactor as a parameter:

```
* (look-inter mymover)
#k<MYMOVER>'s :ACTIVE is T, :WINDOW is #k<MYWINDOW>
#k<MYMOVER> is on the #k<NORMAL-PRIORITY-LEVEL> level
#k<MYMOVER> (MOVE-GROW-INTERACTOR)
   starts when :LEFTDOWN (:IN #k<MYRECT>)
--> NIL
```

In some cases you need to know what interactor will affect a given object (perhaps located using the `ident` function). This is not possible in general since the object(s) an interactor changes may be referenced by arbitrary application code. However, if you use interactors in fairly generic ways, you can call `look-inter` with a graphical object as argument to search for relevant interactors:

```
* (look-inter myrect)
#k<MYMOVER>'s :start-where is (:IN #k<MYRECT>)
--> NIL
* (look-inter mytext)
#k<MYTYPER>'s :start-where is (:IN #k<MYTEXT>)
--> NIL
```

The search algorithm used by look-inter is fairly simple: the current value of `:start-where` is interpreted to see if it could refer to the argument. Then the `:feedback-obj` and `:obj-to-change` slots are examined for an exact match with the argument. If formulas are encountered, only the current value is considered, so there are a number of ways in which `look-inter` can fail to find an interactor.

8. Sizes of Objects

Several functions are provided to help make size measurements of Opal objects and aggregates.

`gd:ObjBytes` *object* [*Function*]

will measure the size of a single Opal object or interactor in bytes.

`gd:AggBytes` *aggregate* `&optional` *verbose* [*Function*]

will measure the size of an Opal aggregate and all of its components in bytes. The first argument may also be a list of aggregates, a window, or a list of windows. For example, to compute the total size of all graphical objects, you can type this:

```
(aggbytes (windows))
```

The output will include various statistics on size according to object type. Sizes are printed in bytes, and the returned value will be the total size in bytes. The size information *does not* include any interactors because interactors can exist independent of the aggregate hierarchy. The optional *verbose* flag defaults to `t`; setting it to nil will reduce the detail of the printed information.

`gd:InterBytes &interactor` *window verbose* [*Function*]

will report size information on the interactors whose `:window` slot *currently* contains the specified window. If the *window* parameter is omitted, `t` or `nil`, then the size of all interactors is computed. (Use `objsize` for a single interactor.) Note that an interactor may operate in more than one window and that interactors can follow objects from window to window. As with `aggbytes`, the *verbose* flag defaults to `t`; setting it to nil will reduce the detail of the printed information.

`gd:*Avoid-Shared-Values*` [*Variable*]

Normally, `aggbytes` does not consider the fact that list structures may be shared, so shared storage is counted multiple times. To avoid this (at the expense of using a large hash table), set `avoid-shared-values` to `t`.

`gd:*Avoid-Equal-Values*` [*Variable*]

To measure the potential for sharing, set this variable to `t`. This will do hashing using `#'equal` so that equal values will be counted as shared instead of `#'eq`, which measures actual sharing.

`gd:*Count-Symbols*` [*Variable*]

Ordinarily, storage for object names is not counted as part of the storage for objects. By setting this variable for true, the sizes reported by `objbytes` and `aggbytes` will include this additional symbol storage overhead.

Note: Size information for an object includes the size of any attached formulas. At present, only objects and cons cells are counted. Storage for structures (other than KR schema), strings, and arrays is *not* counted.

References

[Giuse 89] Dario Giuse.
KR: Constraint-Based Knowledge Representation.
Technical Report CMU-CS-89-142, Carnegie Mellon University Computer Science Department, April, 1989.

[Marchal 89] Philippe Marchal and Andrew Mickish.
Aggregadgets and AggreLists Reference Manual.
The Garnet Toolkit Reference Manuals: Support for Highly-Interactive, Graphical User Interfaces in Lisp.
Carnegie Mellon University Computer Science Department Technical Report CMU-CS-89-196, 1989, pages 179-200.

[Myers 89a] Brad A. Myers, John A. Kolojejchick, and Edward Pervin.
Opal Reference Manual; The Garnet Graphical Object System.
The Garnet Toolkit Reference Manuals: Support for Highly-Interactive, Graphical User Interfaces in Lisp.
Carnegie Mellon University Computer Science Department Technical Report CMU-CS-89-196, 1989, pages 85-126.

[Myers 89b] Brad A. Myers.
Interactors Reference Manual: Encapsulating Mouse and Keyboard Behaviors.
The Garnet Toolkit Reference Manuals: Support for Highly-Interactive, Graphical User Interfaces in Lisp.
Carnegie Mellon University Computer Science Department Technical Report CMU-CS-89-196, 1989, pages 126-178.

Index

Table of Contents