

# **The Oz Inspector**

**Thorsten Brunklaus**

**Version 1.2.3**  
**December 1, 2001**



## **Abstract**

The Inspector is a graphical and interactive tool for displaying and examining Oz values. It combines fast display services with powerful interactive in-place manipulation of the datastructures.

## **Credits**

Mozart logo by Christian Lindig

## **License Agreement**

This software and its documentation are copyrighted by the German Research Center for Artificial Intelligence (DFKI), the Swedish Institute of Computer Science (SICS), and other parties. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

The authors hereby grant permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE AND ITS DOCUMENTATION ARE PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

---

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Inspector Basics</b>	<b>3</b>
2.1	Simple Invocation . . . . .	3
2.2	Inspecting within Multiple Widgets . . . . .	3
2.2.1	Adding and Deleting Widgets . . . . .	4
2.3	Value Representation . . . . .	4
<b>3</b>	<b>Interactive Examination</b>	<b>7</b>
3.1	Node Access Points . . . . .	7
3.2	Node Operations . . . . .	7
3.2.1	Exploration . . . . .	8
3.2.2	Filtering and Mapping . . . . .	9
3.2.3	Triggering Actions . . . . .	10
3.3	Using Selections . . . . .	11
3.3.1	Substructure Lifting . . . . .	11
<b>4</b>	<b>GUI Configuration</b>	<b>13</b>
4.1	Structure Settings . . . . .	13
4.2	Appearance Settings . . . . .	15
4.3	Configuration Range . . . . .	15
<b>5</b>	<b>API Reference</b>	<b>19</b>
5.1	Inspector class methods . . . . .	20
5.2	Inspector Options . . . . .	20
5.3	Value Representation . . . . .	21
5.4	Visual Settings . . . . .	21
5.4.1	Color Assignment . . . . .	22
5.5	Customizing Context Menus . . . . .	22

5.5.1	Registering context menus . . . . .	23
5.5.2	Mapping Functions . . . . .	23
5.5.3	Action Procedures . . . . .	24
5.6	Equivalence Relations . . . . .	24
5.7	Inspecting user-defined types . . . . .	24

---

# Introduction

**Overview** The Inspector is designed to obliterate the barrier between datastructures and their graphical representation, in particular, to make the user feel operating directly with the values without abstractions involved. This is achieved by providing the following features:

- Fast drawing services
- Human readable and predictable data layout
- Interactive and automatic type-directed tree transformations
- Multiple views of the same data
- Widely configurable



---

# Inspector Basics

This chapter provides some basic understanding about how the Inspector can be used.

## 2.1 Simple Invocation

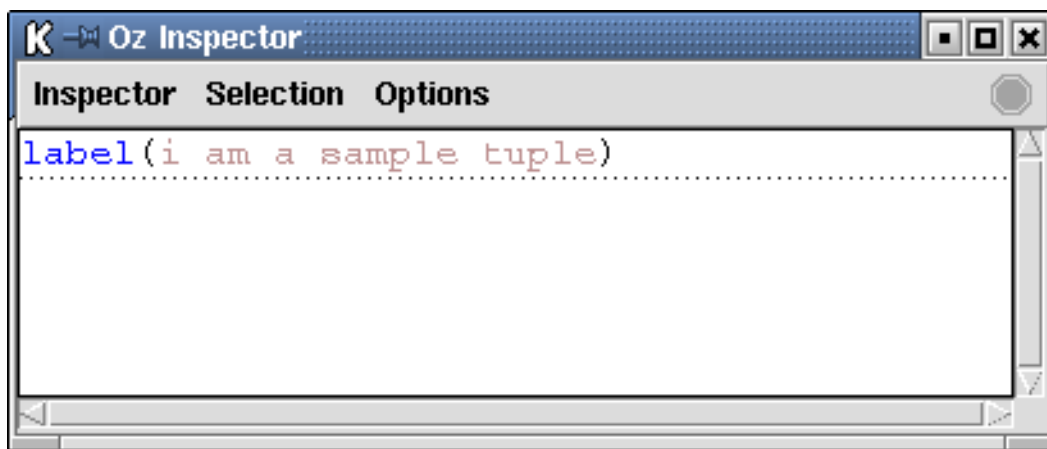
The simplest usage of the Inspector is to execute the statement `{Inspect X}`, where `X` can be any Oz value. Unless the Inspector window is already open, this will create a new window displaying the given value. For instance, the window in Figure 2.1 was created using the statement

```
{Inspect label(i am a sample tuple)}
```

It is possible to inspect more than one value at the same time. By default, new values are appended to the current window.

---

Figure 2.1: The Oz Inspector



---

## 2.2 Inspecting within Multiple Widgets

The Inspector can handle an arbitrarily number of viewing windows (called *widgets*) which can be accessed and configured independently.

### 2.2.1 Adding and Deleting Widgets

To open a new display widget, click ‘Add new Widget’ from the Inspector menu as shown in Figure 2.2.

When more than one widget is used, the notion of the *active widget* becomes important. The active widget receives all work caused by applications of `Inspect`. By default, the initial widget is the active widget. This can be changed using the TAB key. The active widget is indicated by a box with extra-thick line width around its viewing area.

To show the value `X` in the `I`-th window, execute the statement `{InspectN I X}`. The widgets are numbered starting from zero.

The currently active widget can be removed by clicking ‘Delete active Widget’ from the Inspector menu (see Figure 2.2). The widget below the active widget becomes the new active widget. If only a single widget is left, this action is ignored.

## 2.3 Value Representation

The Inspector can be configured to display values using either a tree-like or graph-like representation (see Figure 2.3).

**Tree** The value will be shown as a tree with respect to given depth and width exploration limits. Values larger than the given limits are truncated, the cut positions being indicated by down and right arrows, respectively.

**Graph** The value will be shown not only with respect to given depth and width exploration limits but also with respect to an equivalence relation: Repeatedly occurring sub-values will only be displayed once, all later occurrences being replaced by references to the first occurrence. This yields a more compact representation revealing more information about the value’s structure. Equivalence is computed in a depth-first left-to-right traversal over the entire (truncated) value.

Figure 2.2: Adding a new display widget

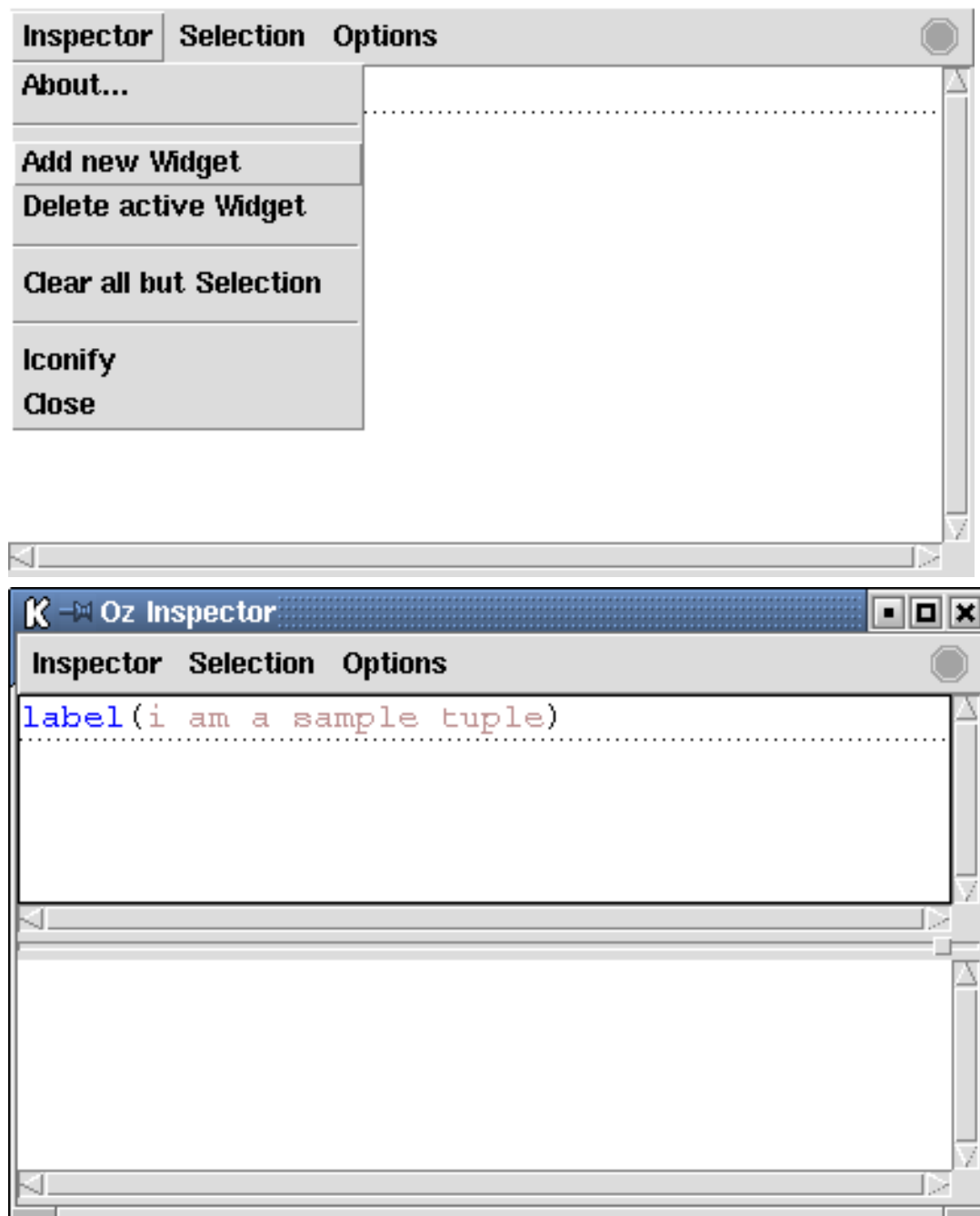
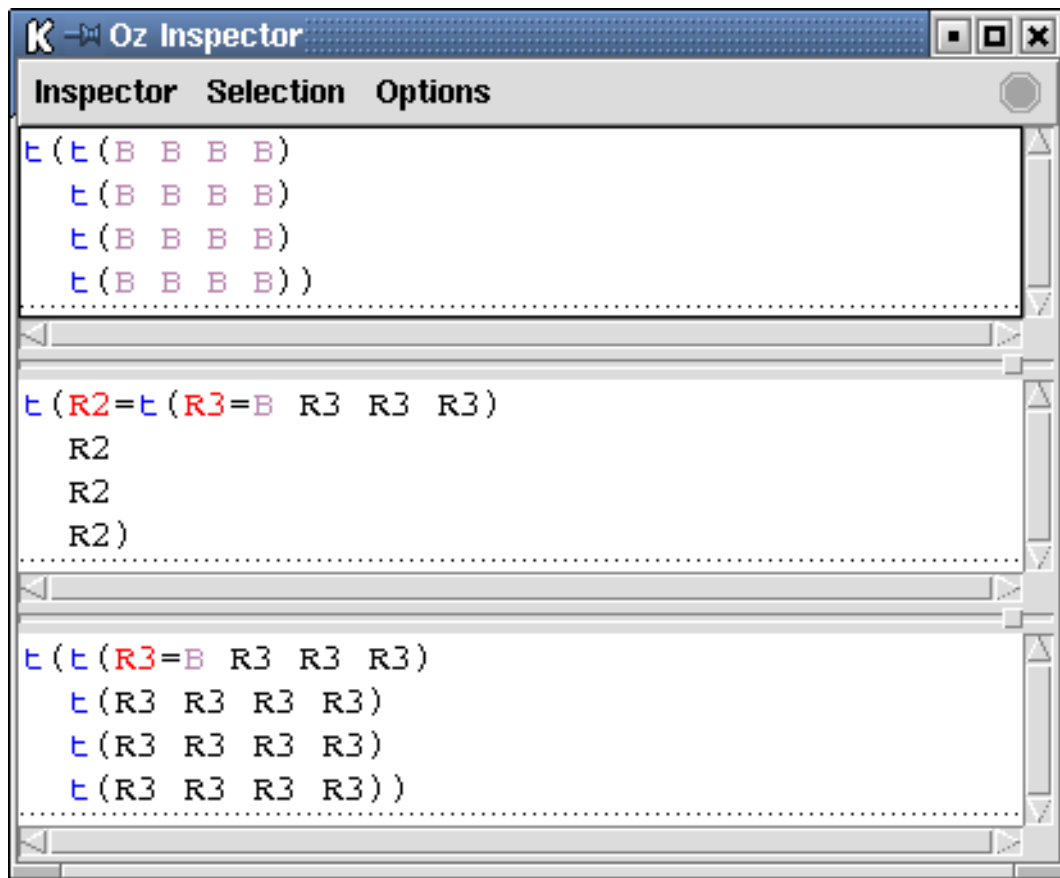


Figure 2.3: Three views on the same datastructure



---

# Interactive Examination

Once a value is displayed, it can be examined more closely by using the context menus attached to its nodes. A node's context menu offers a configurable set of type-specific operations on that node and can be opened by right-clicking the node's so-called *access point*.

## 3.1 Node Access Points

Node access points are defined as follows:

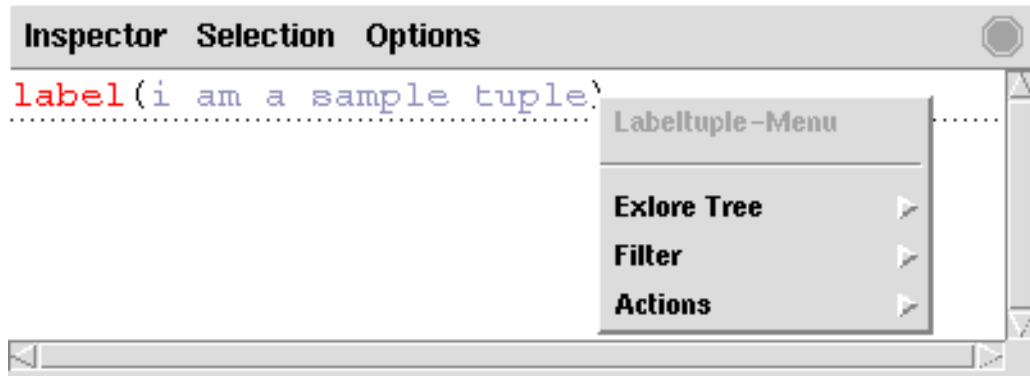
- Any atomic node is a node access point for itself. (An *atomic node* is a node that has no subnodes.)
- For any tuple excluding lists, the label and the parentheses are access points for the entire tuple node.
- For any record, the label and the parentheses are access points for the entire record node. In particular, the record's features are defined *not* to be node access points.
- The node access points of lists depend on the display mode. In tree mode, the brackets and pipe symbols ( `' | '` ) are node access points for the entire list. In graph mode, each pipe symbol is a node access point for the sublist starting at that point.

## 3.2 Node Operations

The Inspector offers three types of operations on nodes: exploration, filtering and mapping, and actions (see Figure 3.1).

- *Exploration* allows to locally modify the depth and width exploration limits, thereby expanding or shrinking the value to the region of interest.
- *Mappings* allow to transform values for display with respect to a given mapping function. This makes it possible to either arbitrarily prune regions of the value (also called *filtering*) or to extract information from abstract datatypes.

Figure 3.1: A context menu opened over the closing parenthesis



- *Actions* allow to apply side-effecting operations on displayed values. For instance, this allows to make values available to external tools such as the experimental Constraint Investigator, for gathering information beyond the scope of the Inspector.

These types of operations are explained in the following sections.

### 3.2.1 Exploration

When inspecting a large data structure, one often needs to zoom in on a subvalue or hide currently irrelevant data. This is what the exploration operations are intended for. Two different kinds of operations exist, for modifying the depth and the width limits respectively (see Figure 3.2).

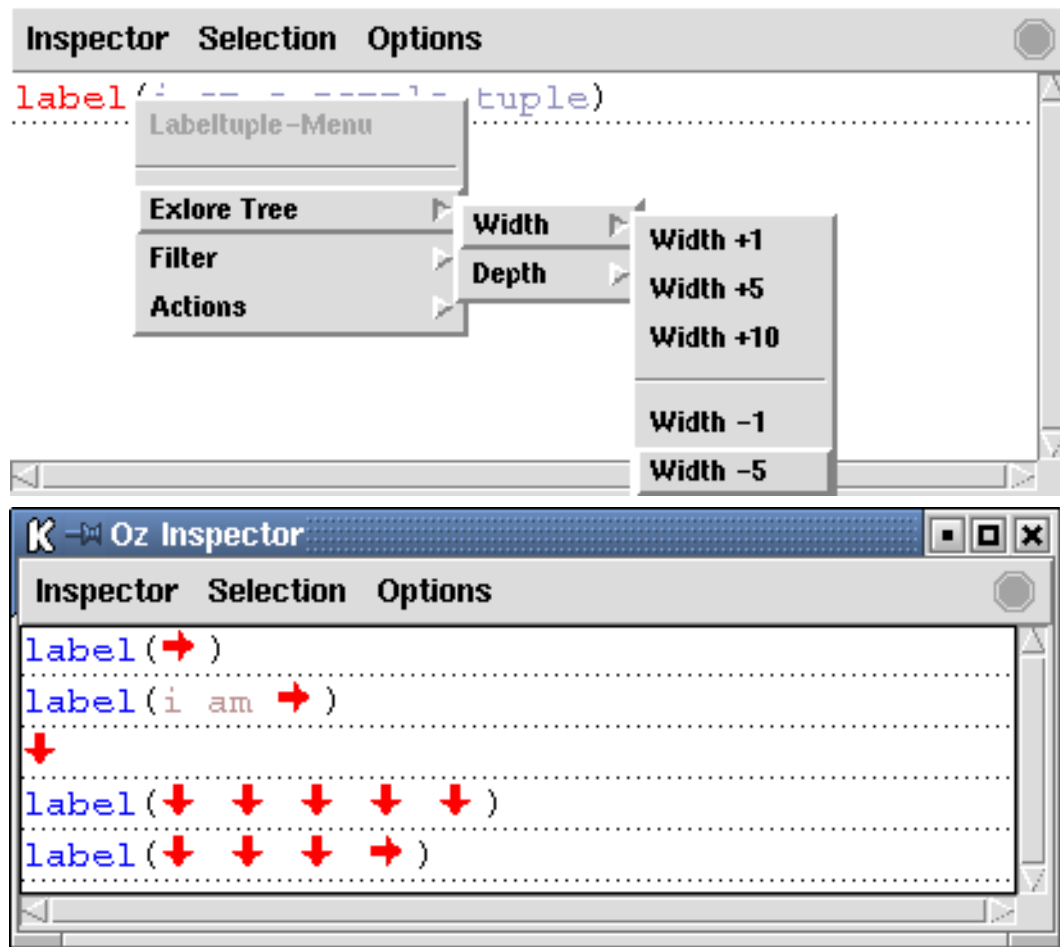
**Depth** Applying the operation ‘Depth  $+n$ ’ to a node causes the subtree starting at that node to be redrawn, with a depth of  $n$  levels. Applying the operation ‘Depth  $-n$ ’ causes the node to be pruned, as well as its  $n - 1$  parent nodes. The effect of the global depth limit is that when new values are inspected, immediately the operation ‘Depth  $+d$ ’ is applied to them, where  $d$  is the global depth limit.

**Width** Every node also has a display width associated with it. If a node has a display width  $w$ , only its first  $w$  subtrees will be displayed (in the case that  $w$  is zero, that means none). Initially, *all* nodes are created with the default display width. Applying the operation ‘Width  $+n$ ’ or ‘Width  $-n$ ’ to a node modifies that node’s display width by the specified amount and causes it to be redrawn accordingly.

Figure 3.2 shows some examples of the effects of the operations described above. They have all been obtained from inspecting the sample tuple given above, and applying operations to it as follows:

1. The sample tuple has been shrunk by applying ‘Width  $-5$ ’.
2. The sample tuple has been subject to applying ‘Width  $-5$ ’ once and ‘Width  $+1$ ’ twice.

Figure 3.2: Sample inspection and various results

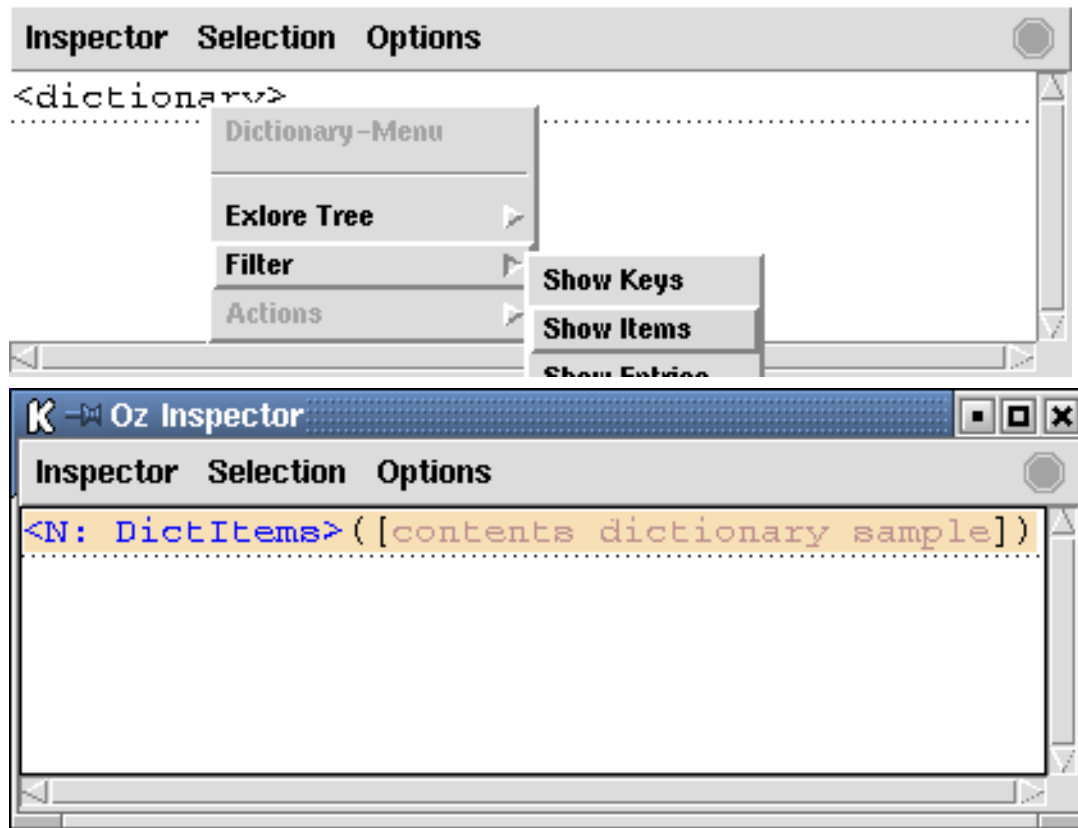


3. The sample tuple has been collapsed by applying 'Depth -1'.
4. The sample tuple is only partially displayed when applying 'Depth +1'. Note that the operation does not augment the display depth, but replaces it.
5. The sample tuple has been partially displayed using 'Depth +1' and then shrunk in width by applying 'Width -1' twice. This shows that the exploration functions are compositional.

### 3.2.2 Filtering and Mapping

The Inspector defines default mappings for many data types, to perform commonly-used operations such as inspecting the state of mutable data structures. Figure 3.3 demonstrates an example of this: By default, dictionaries are displayed as the un-specific tag <dictionary>. The default mapping 'Show Items' for dictionaries replaces the dictionary node by a snapshot of its contents (specifically, as the result of the `Dictionary.items` procedure).

Figure 3.3: Example mapping invocation and its result



Note how the different choice of background color indicates that this is, in fact, a mapped representation. Mappings can be undone using the context menu option ‘un-map’.

Mappings have the following properties:

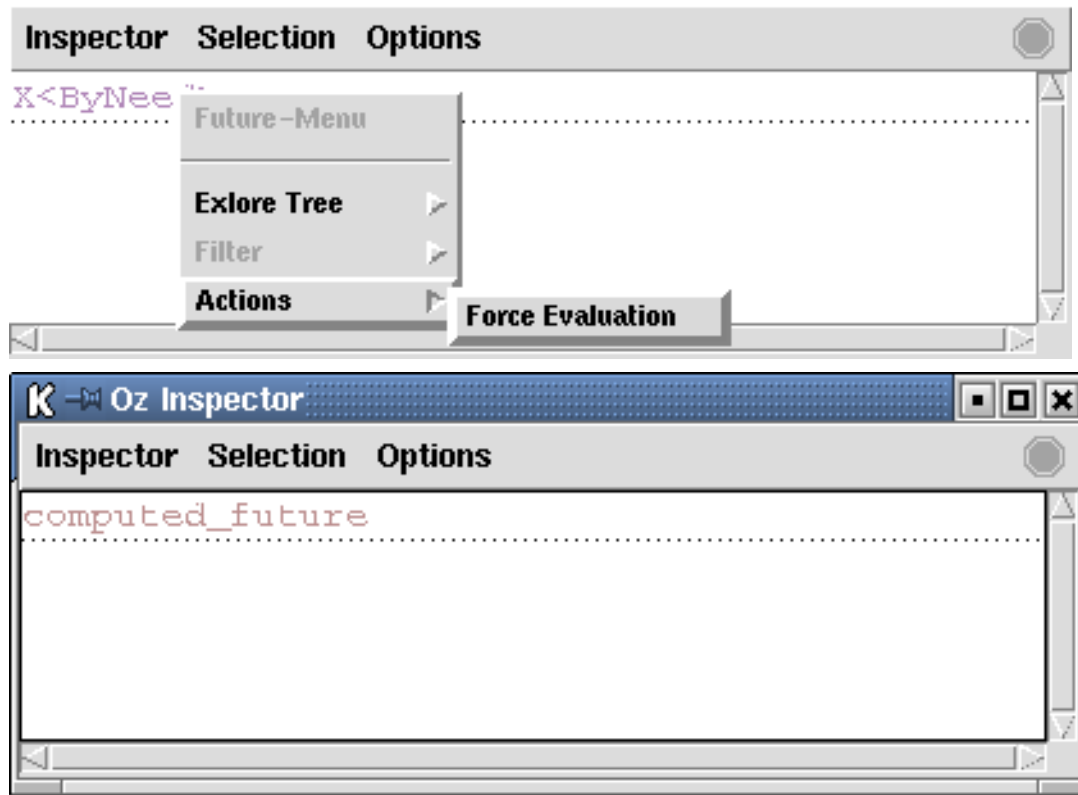
- Mappings are fully compositional, meaning that any other mapping can be applied to an already-mapped node.
- Mapping works on a per-node basis. In particular, the Inspector will apply new mappings to any child nodes of a mapped node.
- Mapping functions are type-specific. That means that every type offers its own collection of mapping functions.

Mapping functions can be applied automatically before inspection of the value. See Section 4.1 for details on how to do this. Keep in mind that the results can be confusing due to the per-node mapping strategy.

### 3.2.3 Triggering Actions

Figure 3.4 demonstrates the invocation of an action. The by-need future is forced, whereupon the Inspector will update its display accordingly, showing its value.

Figure 3.4: Example action invocation and its result



Like mappings, actions are type-specific. See Section 5.5.3 for details on how to register new actions.

### 3.3 Using Selections

Some of the operations presented above can also be triggered on the selected node by the 'Selection' menu. To 'select' a node means to left-click a node access point: The corresponding node will be drawn with a selection frame around it. Figure 3.5 shows the operations that can be applied to the selected node.

#### 3.3.1 Substructure Lifting

Figure 3.6 demonstrates another application of selections: Any selected substructure can be lifted to toplevel by applying the 'Clear all but Selection' command from the Inspector menu.

Substructure lifting is intended to rapidly prune large trees and currently works only with one structure selection. It cannot be undone.

Figure 3.5: The selection interface

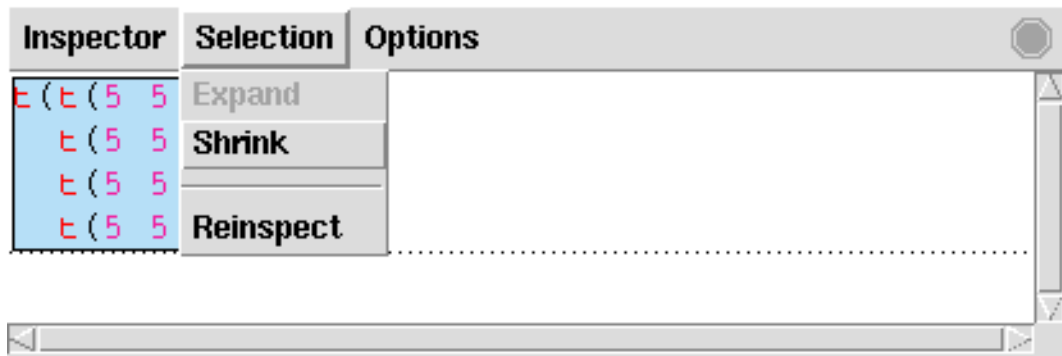
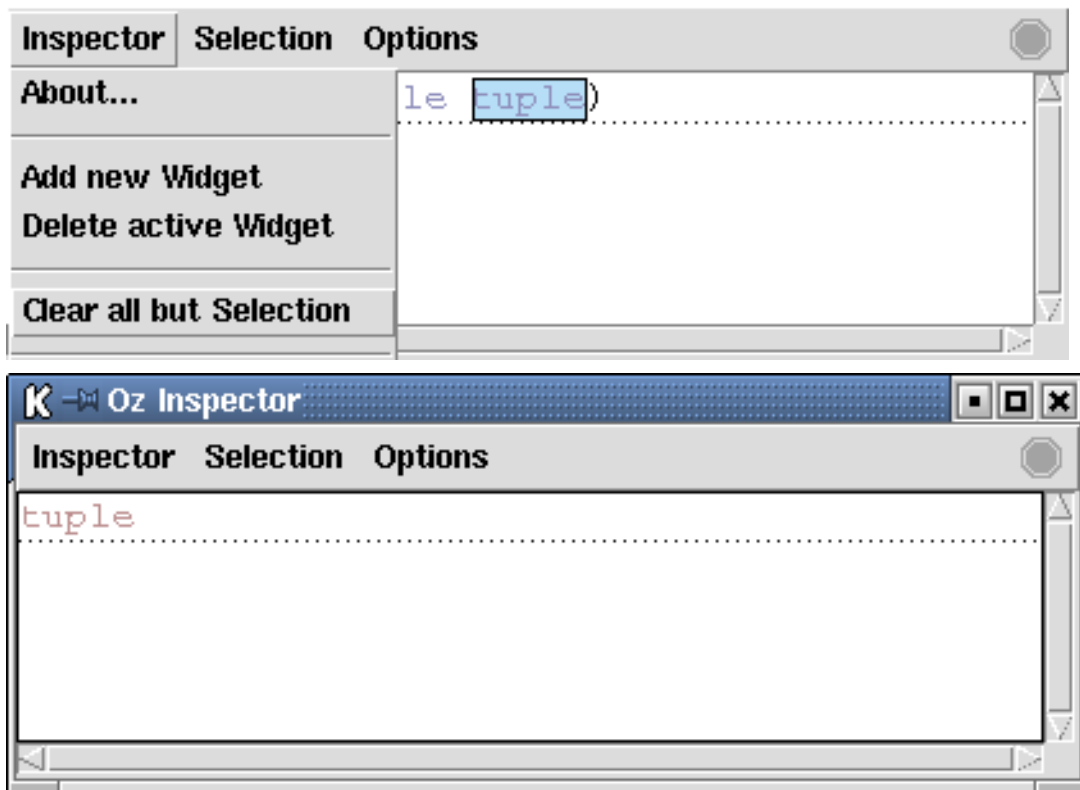


Figure 3.6: Substructure lifting



---

# GUI Configuration

The Inspector is widely user configurable. The options are basically organized in two groups as follows:

- Structure-related options control what node representation the Inspector chooses for a given value. These include the traversal limits, mode (tree or graph) and the mapping details.
- Appearance-related options control the display style to use, such as fonts, colors, and subtree alignment, i.e., how subtrees are arranged.

Since the Inspector handles more than one widget, the user can specify whether the settings should affect all widgets or the active widget only.

This chapter explains how to navigate through the graphical configuration dialog.

## 4.1 Structure Settings

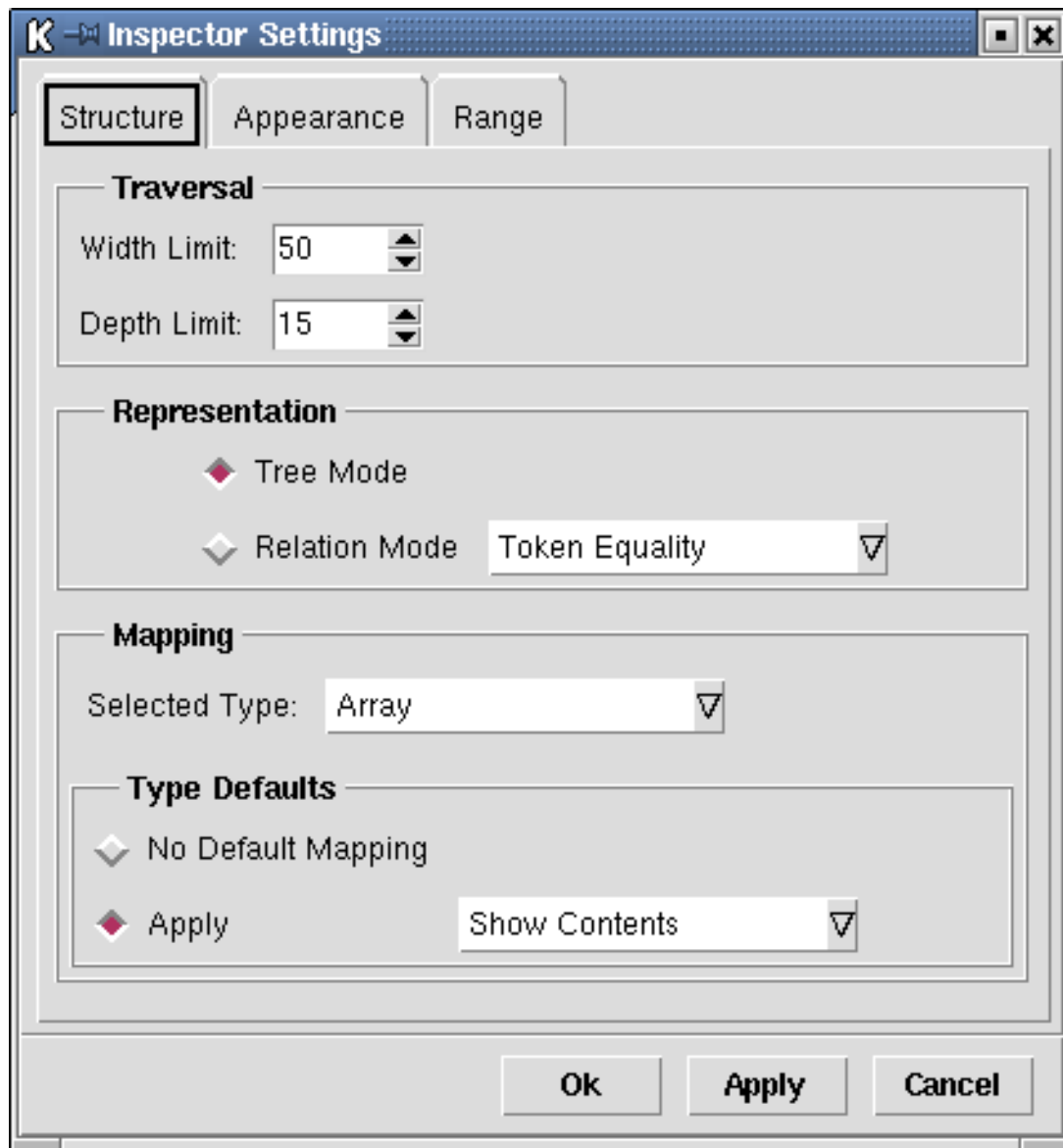
Figure 4.1 shows the structure-related options.

**Traversal** The ‘Traversal’ box allows to change the default exploration limits for newly inspected values. See Section 3.2.1 for details on how these parameters are used.

**Representation** The ‘Representation’ box allows to configure the traversal mode. In *tree mode*, all structures are displayed as their (possibly infinite) tree unrolling, up to the traversal limits. In particular, this mode does not detect cycles. In contrast, *relation mode* both detects cycles and shared substructure. Relation mode also requires to select the equivalence relation to use. (Note that the corresponding combo box is active only when relation mode is selected.)

By default, the following equivalence relations are supported: The default of ‘token equality’ uses `System.eq`, while ‘structural equality’ is unification-based. See Section 5.6 for details on how to add new relations.

Figure 4.1: The selection dialog: 'Structure' tab



**Mapping** The ‘Mapping’ box allows to change the assignment of auto-mappings for the selected type. A type is selected via the corresponding combo box to the right (see Figure 4.1).

Auto-mappings can be deactivated or activated for specific types by first selecting the type, then either clicking ‘No default mapping’ or ‘Apply’, respectively. The latter then requires to select one of the registered mapping functions for that type. (Note again that the corresponding combo box box is active only if ‘Apply’ is selected.)

See Section 5.5.2 for details on how to add new mappings.

## 4.2 Appearance Settings

Figure 4.2 shows the appearance-related options. It should be self-explanatory.

## 4.3 Configuration Range

Figure 4.3 shows the options controlling the applicability of the settings made on the previous tabs. Again, this should be self-explanatory.

Figure 4.2: The selection dialog: ‘Appearance’ tab

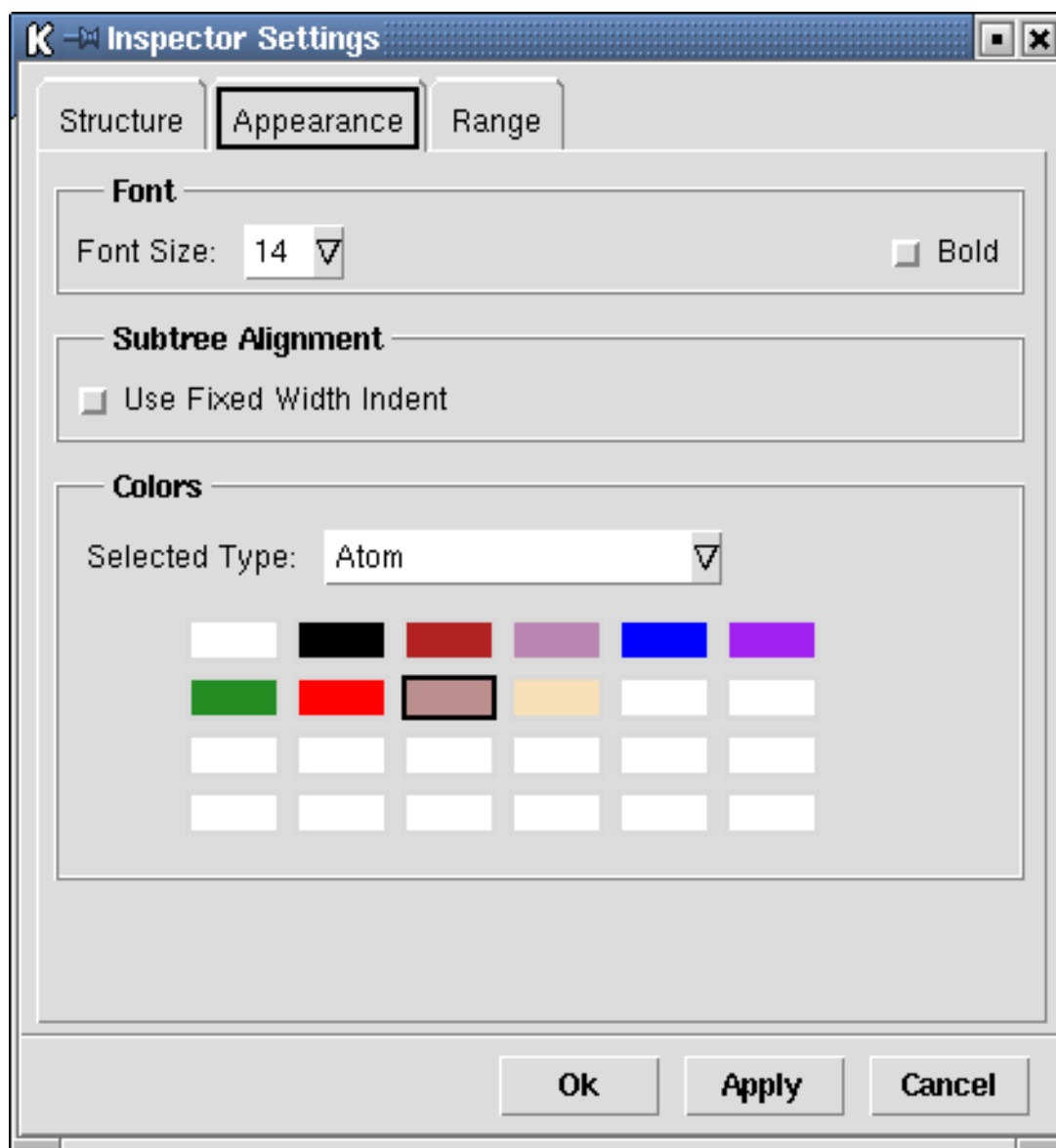


Figure 4.3: The selection dialog: Range tab





# API Reference

The Inspector functor is available for import at the URI `'x-oz://system/Inspector'`. It exports the following application programmer's interface:

## `Inspect`

```
{Inspector.inspect X}
```

opens a new Inspector window if none exists, then displays X in the active widget.

## `inspectN`

```
{Inspector.inspectN +I X}
```

opens a new Inspector window if none exists, then displays X in the widget with number I, counting from zero. Note that the chosen widget must have been created already. This is always the case for number zero.

## `configure`

```
{Inspector.configure +Key +Value}
```

sets the configuration option with key Key to value Value. The following sections starting with Section 5.2 explain in detail which keys and values are valid parameters.

## `configureN`

```
{Inspector.configureN +I +Key X}
```

behaves like `configure` but directly addresses the widget number I which must be existing already.

## `close`

```
{Inspector.close}
```

closes the Inspector window, if any.

## `'class'`

```
Inspector.'class'
```

is the class from which Inspector instances can be created.

## `object`

```
Inspector.object
```

is the default instance of class `Inspector.'class'` which is implicitly used by `Inspect`, `Inspector.inspectN`, `Inspector.configure`, `Inspector.configureN` and `Inspector.close`. Note that this object is not thread safe. Use the wrappers provided or a server abstraction.

## 5.1 Inspector class methods

The inspector class provides the methods shown below.

`create`

initializes a new inspector object with reasonable defaults.

`inspect(X)`

inspects the value X.

`inspectN(+N X)`

inspects the value denoted by X using widget number N. Keep in mind then the corresponding widget must have been created already.

`configureEntry(+Key +Value)`

tells the inspector to update the option denoted by Key with value Value.

`close`

closes and unmaps the inspector object.

*Caution:* The created inspector object itself is not thread safe. Use a server wrapper instead if concurrency is required. This limitation applies only to the inspector object itself but not its widgets.

## 5.2 Inspector Options

This section covers global Inspector options.

`inspectorWidth`

`{Inspector.configure inspectorWidth +I}`

adjusts the Inspector's horizontal window dimension to I pixels. Defaults to 600.

`inspectorDepth`

`{Inspector.configure inspectorDepth +I}`

adjusts the Inspector's vertical window dimension to I pixels. Defaults to 400.

`inspectorOptionsRange`

`{Inspector.configure inspectorOptionsRange +A}`

determines which widgets will be affected by reconfiguration. A can be either 'active' or 'all', with the obvious meaning.

## 5.3 Value Representation

This section explains how to configure the value representation using the API.

**widgetTreeWidth**

```
{Inspector.configure widgetTreeWidth +I}
```

adjusts the global width traversal limit to *I*. Defaults to 50.

**widgetTreeDepth**

```
{Inspector.configure widgetTreeDepth +I}
```

adjusts the global depth traversal limit to *I*. Defaults to 15.

**widgetTreeDisplayMode**

```
{Inspector.configure widgetTreeDisplayMode +B}
```

switches between tree and graph representation, depending on whether *B* denotes **true** or **false**, respectively.

**widgetUseNodeSet**

```
{Inspector.configure widgetUseNodeSet +I}
```

determines the subtree alignment. If *I* is 1, the standard alignment is used. A value of 2 selects fixed width indentation.

**widgetShowStrings**

```
{Inspector.configure widgetShowStrings +B}
```

switches between normal and readable representation of strings. It defaults to **false**.

## 5.4 Visual Settings

This section explains how to configure the visual aspects of node drawing using the API.

**widgetTreeFont**

```
{Inspector.configure widgetTreeFont font(family:+F size:+S weight:+W)}
```

selects the widget font as follows:

- *+F* denotes either **'courier'** or **'helvetica'**.
- *+S* denotes any integer out of {10, 12, 14, 18, 24}.
- *+W* denotes either **'normal'** or **'bold'**.

The default is `font(family:'courier' size:12 weight:'normal')`.

**widgetContextMenuFont**

```
{Inspector.configure widgetContextMenuFont +V)}
```

selects the font used to draw the context menus. *V* denotes any valid X font description. It defaults to **'-adobe-helvetica-bold-r-\*-\*\*-100-\*'**.

### 5.4.1 Color Assignment

```
{Inspector.configure +A +Color)}
```

assigns color `Color` to item type `A`. `Color` denotes an atom describing any valid hexadecimal RGB color code. `A` is composed of a type prefix and a 'Color' suffix. For example,

```
{Inspector.configure intColor '#AAFF11'}
```

chooses to draw integer nodes with color `'#AAFF11'`. The complete list of known types is shown in table Figure 5.1.

Figure 5.1: Known atomic types

<code>int</code>	<code>float</code>	<code>atom</code>
<code>bool</code>	<code>unit</code>	<code>name</code>
<code>bytestring</code>	<code>procedure</code>	<code>future</code>
<code>free</code>	Oz variables	
<code>fdint</code>	Oz finite domain variables	
<code>label</code>	record/tuple labels	
<code>feature</code>	record features	
<code>background</code>	widget background	
<code>variableref</code>	using occurrence of graph reference	
<code>ref</code>	defining occurrence of graph reference	
<code>generic</code>	the generic value	
<code>braces</code>	parentheses around mixfix tuples	
<code>colon</code>	separator between feature and subtree	
<code>widthbitmap</code>	left arrow	
<code>depthbitmap</code>	down arrow	
<code>separator</code>	separators such as <code>#</code> and <code> </code>	
<code>proxy</code>	background color used for mappings	
<code>selection</code>	background color used for selections	

## 5.5 Customizing Context Menus

This section explains how to create context menus. The current implementation only allows to attach entire context menus to a node. A context menu simply is a tuple `menu(WidthList DepthList MappingList ActionList)` as follows:

- `WidthList` and `DepthList` denote integer lists describing the possible limit changes. The integer zero serves as separator indicator. Both lists default to `[1 5 10 0 ~1 ~5 ~10]`.
- `MappingList` denotes a list of tuples following the pattern `'Shown Label' (MapFun)`. See Section 5.5.2 for details about writing mappings.

If the function should be used for auto-mapping, use `auto('Shown Label' (MapFun))` instead. Each list must not contain more than one auto mapping entry.

- `ActionList` denotes a list of tuples following the pattern `'Shown Label' (ActionProc)`. See Section 5.5.3 for details about writing actions.

### 5.5.1 Registering context menus

Context menus are registered using the statement

```
{Inspector.configure +Type +Menu}
```

where `Type` denotes an atom composed of a valid type prefix as shown in Figure 5.2 and a `Menu` suffix. `Menu` denotes the menu as described in Section 5.5.

Figure 5.2: Menu types

<code>hashtuple</code>	Tuples with label <code>#</code>
<code>pipetuple</code>	Possible open lists such as streams
<code>list</code>	Closed lists
<code>labetuple</code>	Tuple with any other label
<code>record</code>	Oz Records (without kinded records)
<code>kindedrecord</code>	Feature constraints
<code>future</code>	Futures
<code>free</code>	Variables
<code>fdint</code>	Finite domain variables
<code>fset</code>	Finite set menus
<code>array</code>	Oz arrays
<code>dictionary</code>	Oz dictionaries
<code>class</code>	Oz classes
<code>object</code>	Oz objects
<code>chunk</code>	Oz chunks
<code>cell</code>	Oz cell

For example, executing

```
{Inspector.configure recordMenu Menu}
```

assigns a new record context menu denoted by `Menu`.

### 5.5.2 Mapping Functions

Writing mapping functions is easy: Every function follows the pattern below.

```
fun {MyMapFunction Value MaxWidth MaxDepth}
  if {WantToMap Value} then
    ... /* computations */ ...
  else Value
  end
end
```

`MaxWidth` and `MaxDepth` are integers denoting the node's width and depth limits. This allows to handle cycles independently of whether they would have been recognized or not. Keep in mind that mapping functions should not have side-effects.

### 5.5.3 Action Procedures

Action procedures follow the pattern below.

```
proc {MyAction Value}
  ... /* computations */ ...
end
```

## 5.6 Equivalence Relations

Equivalence relations are registered using the statement

```
{Inspector.configure widgetRelationList ['RelationName'(RelFun) /* ... more relations */]}
```

Every `RelFun` is expected to compute the characteristic function of its relation and therefore follows the pattern

```
fun {RelFun X Y}
  /* computations resulting true or false */
end
```

These functions should not have side effects. Again, the implementation currently only supports the replacement of the all relations.

## 5.7 Inspecting user-defined types

Oz allows to add new types to the system. The Inspector is able to display such values without being rebuilt provided that they can be distinguished using `Value.status` and transformed to a built-in type. In this case, it is sufficient to add a new mapping function attached to that type. In the default case, the inspector will use `Value.toVirtualString` to obtain a useful representation of the unknown value. Depending on its type, the value will be monitored and updated, too.

For example, to introduce weak dictionaries to the inspector use the code shown below.

```
local
  fun {ItemFun V W D}
    {WeakDictionary.items V}
  end
  Key      = case {Value.status {WeakDictionary.new _}} of det(T) then T [] _
  TypeKey = {VirtualString.toAtom Key#'Menu'}
in
  {Inspector.configure TypeKey menu(nil nil [auto('Show Contents'(ItemFun))])}
end
```

See Section 5.5.2 for details about how `ItemFun` is built.