KR: Constraint-Based Knowledge Representation

Dario Giuse

December 1994

Abstract

KR is a very efficient knowledge representation language implemented in Common Lisp. It provides powerful frame-based knowledge representation with user-defined inheritance and relations, and an integrated object-oriented programming system. In addition, the system supports a constraint maintenance mechanism which allows any value to be computed from a combination of other values. KR is simple and compact and does not include some of the more complex functionality often found in other knowledge representation systems. Because of its simplicity, however, it is highly optimized and offers good performance. These qualities make it suitable for many applications that require a mixture of good performance and flexible knowledge representation.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

1. Introduction

This document is the reference manual for the KR system. KR implements objects, also known as *schemata*, which can contain any amount of information and which can be connected in arbitrary ways. All Garnet objects are implemented as KR schemata. KR [Giuse 87] can also be used as a very efficient frame-based representation system. Simplicity and efficiency are its main design goals and differentiate it sharply from more conventional frame systems, as discussed in [Giuse 90].

In addition to the basic representation of knowledge as a network of schemata, KR provides object-oriented programming and an integrated constraint maintenance system. Constraint maintenance is implemented through *formulas*, which constrain certain values to combinations of other values. The constraint system is closely integrated with the basic object system and is part of the same program interface.

Close integration between objects and constraint maintenance yields several advantages. First of all, constraint maintenance is seen as a natural extension of object representation; the same access functions work on regular values and on values constrained by a formula. Second, the full power of the representation language is available in the specification of constraints. Third, since the two mechanisms are integrated at a fairly low level, the constraint maintenance system offers very good performance. These advantages make the KR constraint maintenance system a practical tool for the development of applications that require flexibility, expressive power, and performance comparable to that obtained with conventional data structures.

In addition to being one of the building blocks of the Garnet project, KR can be used as a self-contained knowledge representation system. Besides Garnet, KR is used in the Chinese Tutor [Giuse 88a] [Giuse 88b], an intelligent tutoring system designed to teach Chinese to English speakers, and in speech understanding research [Young 89] currently underway at Carnegie Mellon.

This document describes version 2.3.1 of KR, which is part of release 2.2 of the Garnet system. Several aspects of this version differ from previous versions of the system, such as the ones described in previous reports [Giuse 89] [Giuse 87]. The present document overrides all previous descriptions.

The orientation of this manual is for users of KR as an object system. Users who are more interested in using KR as a knowledge representation system should consult a previous paper [Giuse 90]. This manual begins with a description of the features of the system that beginners are most likely to need. Some of the less common features are only presented near the end of the document, in order to avoid obscuring the description with irrelevant details. Sections 6 and 7 contain the detailed description of the program interface of KR. This is a complete description of the system and its features. Most application programs will only need a small number of features, described in section 6.

2. Structure of the System

KR is an object system implemented in Common Lisp [Steele 84]. It includes three closely integrated components: *object-oriented programming*, *constraint maintenance*, and *knowledge representation*.

The first component of KR is an object oriented programming system based on the prototype-instance paradigm. Schemata can be used as objects, and inheritance can be used to determine their properties and behavior. Objects can be sent *messages*, which are implemented as procedural attachments to certain slots; messages are inherited through the same mechanism as values.

Instead of the class-instance paradigm, common in object-oriented programming languages, KR implements the more flexible prototype-instance paradigm [Lieberman 86], which allows properties of instances to be determined dynamically by their prototypes. This means that the class structure of a system can be modified dynamically as needed, without any need for recompilation.

The second component of KR implements constraint maintenance. Constraint maintenance is implemented through *formulas*, which may be attached to slots and determine their values based on the values of other slots in the system.

Constraint maintenance is closely integrated with the other components. The user, for example, does not need to know which slots in a schema contain ordinary values and which ones are constrained by a formula, since the same access primitives may be used in both cases.

The third component, frame-based knowledge representation, stores knowledge as a network of chunks of information. Networks in KR are built out of unstructured chunks, i.e., *schemata*. Each schema can store arbitrary pieces of information, and is not restricted to a particular format or data structure. Information is encoded via attribute-value pairs.

Values in a schema can be interpreted as links to other schemata. This enables the system to support complex network structures, which can be freely extended and modified by application programs. KR provides simple ways to specify the structure of a network and the relationship among its components.

3. Basic Concepts

This section describes the basic elements of KR, i.e., objects. More details about the design philosophy of the system and some of the internal implementation may be found in [Giuse 87], which describes a previous version of the system that did not support constraint maintenance.

3.1. Main Concepts: Schema, Slot, Value

An object in KR is known as a *schema*. A schema is the basic unit of representation and consists of an optional *name*, a set of *slots*, and a *value* for each slot. The user can assemble networks of schemata by placing a schema as the value in a slot of another schema; this causes the two schemata to become linked.

A schema may be named or unnamed. Named schemata are readily accessible and are most useful for interactive situations or as the top levels of a hierarchy, since their names act as global handles. Unnamed schemata do not have meaningful external names. They are, however, more compact than named schemata and account for the vast majority of schemata created by most applications. Unnamed schemata are automatically garbage-collected when no longer needed, whereas named schemata have to be destroyed explicitly by the user.

The name of a named schema is a symbol. When a named schema is created, KR automatically creates a special variable by the same name and assigns the schema itself as the value of the special variable. This makes named schemata convenient to use.

A schema may have any number of *slots*, which are simply attribute-value pairs. The slot name indicates the attribute name; the slot value (if there is one) indicates its value. Slot names are keywords, and thus always begin with a colon. All slots in a schema must have distinct names, but different schemata may very well have slots with the same name.

Each slot may contain only one value. A value is the actual data item stored in the schema, and may be of any Lisp type. KR provides functions to add, delete, and retrieve the value from a given slot in a schema.

The printed representation of a schema shows the schema name followed by slot/value pairs, each on a separate line. The whole schema is surrounded by curly braces. For example,

```
{#k<fido>
  :owner =  #k<john>
  :color =  #k<brown>
  :age =  5
}
```

The schema is named FIDO and contains three slots named `:owner`, `:color`, and `:age`. The slot `:age` contains one value, the integer 5.

The default printed name of a schema is of the form #K<NAME>, where *name* is the actual name of the schema. This representation makes it very easy to distinguish KR schemata from other objects. Note, however, that this convention is only used when printing, and is not used when typing the name of a schema.

In order to illustrate the main features of the system, we will repeatedly use a few schemata. We present the definition of those schemata at this point and will later refer to them as needed. These schemata might be part of some graphical package, and are used here purely for explanation purposes. In practice, there is no need to define such schemata in a Garnet application, since the Opal component of Garnet (see the Opal manual) already provides a complete graphical object system.

The following KR code is the complete definition of the example schemata:

```
(create-instance 'MY-GRAPHICAL-OBJECT NIL
  (:color :blue))

(create-instance 'BOX-OBJECT MY-GRAPHICAL-OBJECT
  (:thickness 1))

(create-instance 'RECTANGLE-1 BOX-OBJECT
  (:x 10)
  (:y 20))

(create-instance 'RECTANGLE-2 BOX-OBJECT
  (:x 34)
  (:y (o-formula (+ (gvl :left-obj :y) 15)))
  (:left-obj RECTANGLE-1))
```

The exact meaning of the expressions above will become clear after we describe the functional interface of the system. Briefly, however, the example can be summarized as follows. The schema MY-GRAPHICAL-OBJECT is at the top of a hierarchy of graphical objects. The schema BOX-OBJECT represents an intermediate level in the hierarchy, and describes the general features of all graphical objects which are rectangular boxes. BOX-OBJECT is placed below MY-GRAPHICAL-OBJECT in the hierarchy, and its `:is-a` slot points to the schema MY-GRAPHICAL-OBJECT. This is done automatically by the macro create-instance.

Finally, two rectangles (RECTANGLE-1 and RECTANGLE-2) are created and placed below BOX-OBJECT in the hierarchy. RECTANGLE-1 defines the values of the two slots `:x` and `:y` directly, whereas RECTANGLE-2 uses a formula for its `:y` slot. The formula states that the value of `:y` is constrained to be the `:y` value of another schema plus 15. The other schema can be located by following the `:left-obj` slot of RECTANGLE-2, as specified in the formula, and initially corresponds to RECTANGLE-1.

Figure 3-1 shows the four schemata after the definitions above have been executed. Relations are indicated by an arrow going from a schema to the ones to which it is related.

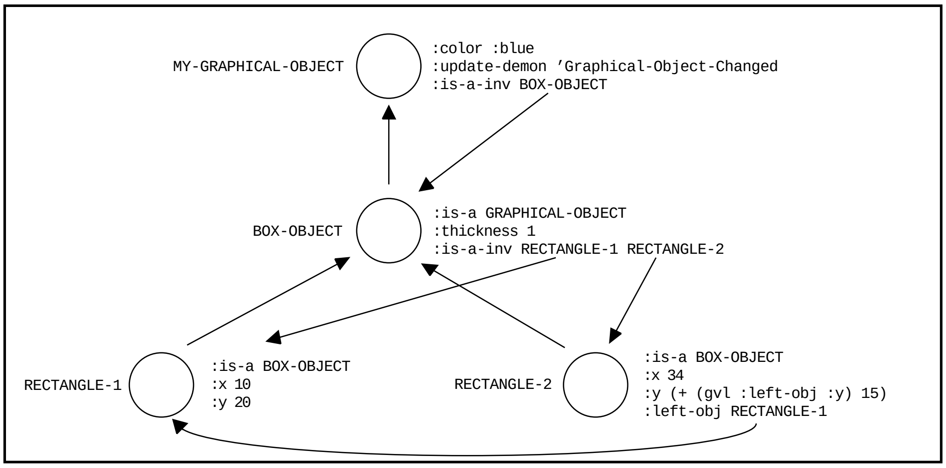

Figure 3-1: The resulting network of schemata

Asking the system to print out the current status of schema RECTANGLE-2 would produce the following output:

```
{#k<RECTANGLE-2>
  :IS-A =  #k<BOX-OBJECT>
  :LEFT-OBJ =  #k<RECTANGLE-1>
  :Y =  #k<F2289>(NIL . NIL)
  :X =  34
}
```

Note that slot `:y` contains a formula, which is printed as "#k<F2289>(NIL . NIL)". This is simply an internal representation for the formula and will yield the correct value of `:y` when needed.

3.2. Inheritance

The primary function of values is to provide information about the object represented by a schema. In the previous example, for instance, asking the system for the `:x` value of RECTANGLE-1 would simply return the value `10`.

Values can also perform another function, however: They can establish *connections between schemata*. Consider the `:left-obj` slot in the example above: if we interpret RECTANGLE-1 as a schema name, then the slot tells us that the schema RECTANGLE-2 is somehow related to the schema RECTANGLE-1. Graphically, this will mean that the position of RECTANGLE-2 is partially determined by that of RECTANGLE-1.

KR also makes it possible to use values to perform *inheritance*, i.e., to control the way information is inherited by a particular schema from other schemata to which it is connected. Inheritance allows information to be arranged in a hierarchical fashion, with lower-level schemata inheriting most of their general features from higher-level nodes and possibly providing local refinements or modifications. A connection that enables inheritance of values is called an *inheritance relation*. Inheritance relations always contain a list of values; in many cases, this is a list of only one value.

The most common example of inheritance is provided by the `:is-a` relation. If schema A is connected to schema B by the `:is-a` relation,[1] then values that are not present in A may be inherited from B.

Consider the schema RECTANGLE-1 in our example. If we were to ask "What is the color of RECTANGLE-1?", we would not be able to find the answer by just looking at the schema itself. But since we stated that RECTANGLE-1 is a box object, which is itself a graphical object, the value can be inherited from the schema MY-GRAPHICAL-OBJECT through two levels of `:is-a`. The answer would thus be "RECTANGLE-1 is blue." Inheritance is possible in this case because the slot `:is-a` is pre-defined by the system as a relation.

[1]In other words, if schema B appears as a value in the `:is-a` slot of schema A.

4. Object-Oriented Programming

This section describes the object-oriented programming component of KR. This component entails two concepts: the concept of message sending, and the concept of prototype/instance.

4.1. Objects

The fundamental data structure in KR is the *schema*, which is equivalently referred to as an *object*. Objects consist of data (represented by values in slots) and methods (represented by procedural attachments, again stored as values in slots). Methods are similar to functions, except that a method can do something different depending on the object that it is called on. A procedural attachment is invoked by "sending a message" to an object; this means that a method by the appropriate name is sought and executed. Different objects often provide different methods by the same name, and thus respond to the same message by performing different actions.

The data and methods associated with an object can be either stored within the object or inherited. This allows the behavior of objects to be built up from that of other objects. The object-oriented component of KR allows some combination of methods, since a method is allowed to invoke the corresponding method from an ancestor schema and to explicitly refer to the object which is handling the message. Method combination, however, is not as developed as in full-fledged object-oriented programming systems such as CLOS [Bobrow et al. 89].

4.2. Prototypes vs. Classes

The notion of *prototype* in KR is superficially similar to that of *class* in conventional object-oriented programming languages, since a prototype object can be used to partially determine the behavior of other objects (its *instances*). A prototype, however, plays a less restricting role than a class. Unlike classes, prototypes simply provide a place from which the values of certain slots may be inherited. The number and types of slots which actually appear in an instance is not in any way restricted by the prototype. The same is true for methods, which are simply represented as values in a slot.

Prototypes in KR serve two specific functions: they provide an initialization method, and they provide default constraints. When a KR schema is created via the function `create-instance`, and its prototype has an `:initialize` method, the method is invoked on the instance itself. This results in a uniform mechanism for handling object-dependent initialization tasks.

4.3. Inheritance of Formulas

If a prototype provides a constraint for a certain slot, and the slot is not explicitly redefined in an instance, the formula which implements the constraint is copied down and installed in the instance itself. The formula, however, is not actually copied down until a value is requested for that slot (e.g., when `gv` is used). This is a convenient mechanism through which a prototype may partially determine the behavior of its instances. Note that this behavior can be overridden both at instance-creation time (by explicitly specifying values for the instance) and at any later point in time.

5. Constraint Maintenance

This section describes the constraint maintenance component of KR. The purpose of constraint maintenance is to ensure that changes to a schema are automatically propagated to other schemata which depend on it.

5.1. Value Propagation

The KR constraint system offers two distinct mechanisms to cause changes in a part of network to propagate to other parts of the network. The first mechanism, *value propagation*, ensures that the network is kept in a consistent state after a change. The second mechanism, *demon invocation*, allows certain actions to be triggered when parts of a network are modified. Demons are described in section 8.9.

Value propagation is based on the notion of *dependency* of a value on another. Value dependencies are embodied in formulas. Whenever a value in a slot is changed, all slots whose values depend on it are immediately invalidated, although not necessarily re-evaluated. This strategy, known as lazy evaluation, does not immediately recompute the values in the dependent slots, and thus it typically does less work than an eager re-evaluation strategy. The system simply guarantees that correct values are recomputed when actually needed.

5.2. Formulas

Formulas represent one-directional connections between a *dependent value* and any number of *depended values*. Formulas specify an expression which computes the dependent value based upon the depended values, as well as a permanent dependency which causes the dependent value to be recomputed whenever any of the other values change.

Formulas can contain arbitrary Lisp expressions, which generally reference at least one particular depended value. The Lisp expression is used to recompute the value of the formula whenever a change in one of the depended values makes it necessary.

Formulas are not recomputed immediately when one of the depended values changes. This reduces the amount of unnecessary computation. Moreover, formulas are not recomputed every time their value is accessed. Each formula, instead, keeps a cache of the last value it computed. Unless the formula is marked invalid, and thus needs to be recomputed, the cached value is simply reused. This factor causes a dramatic improvement in the performance of the constraint maintenance system, since under ordinary circumstances the rate of change is low and most changes are local in nature.

Figure 5-1, part **(a)**, shows an example of a formula installed on slot `:y` of schema POINT-2. The formula depends on two values, i.e., the value of slots `:y1` and `:y2` in schema POINT-1. The formula specifies that slot `:y` is constrained to be the sum of the two values divided by 2, i.e., the average of the two values. Figure 5-1, part **(b)**, shows the internal state of the formula in a steady-state situation where the formula contains a valid cached value. Under these circumstances, any request for the value of slot `:y` would simply return the cached value, without recomputing the formula.

Parts **(c)** and **(d)** show the effects of changes to the depended values. Changes are illustrated by small rectangles surrounding the modified information. The first change is to slot `:y1` and causes the value in the formula to be marked invalid. Note that the formula is not actually recomputed at this point, and the cached value is left untouched. The second change is to slot `:y2` and does not cause any action to occur, since the formula is already marked invalid.

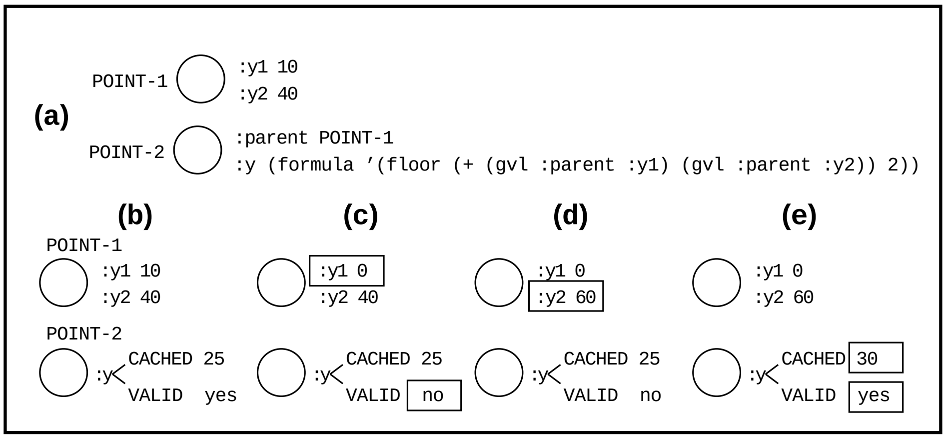

Figure 5-1: Successive changes in depended values

Finally, part **(e)** shows what happens when the value in slot `:y` is eventually needed. The value of the formula is recomputed and again cached locally; the cache is marked as valid. The system is then back to steady state. Note that the formula was recomputed only once, when needed, rather than eagerly after each value changed.

5.2.1. Circular Dependencies

Constraints may involve circular chains of dependency. Slot A, for instance, might depend on slot B, which in turn depends on slot A; see section 9.1 for an example of a situation where this arises fairly naturally. Circular dependencies may also be used to provide a limited emulation of two-way constraint maintenance.

KR is able to deal with circular dependencies without any trouble. This is handled during formula evaluation; if a formula is evaluated and requests a value which depends of the formula itself, the cycle is broken and the cached value of the formula is used instead. This algorithm guarantees that the network is left in a consistent state, even though the final result may of course depend on where evaluation started from.

5.2.2. Dependency Paths

Typical formulas contain embedded references to other values and schemata. The formula in Figure 5-1, for example, contains an indirect reference to schema POINT-1 through the contents of the `:other` slot. Such references are known as dependency paths. Whenever a formula is evaluated, its dependency paths are used to recompute the updated value.

It is possible for a dependency path to become temporarily unavailable. This would happen, for instance, if schema POINT-1 in Figure 5-1 was deleted, or if slot `:other` in schema POINT-2 was temporarily set to NIL. KR handles such situations automatically. If a formula needs to be evaluated but one of its dependency paths is broken, the current cached value of the formula is simply reused. This makes it completely safe to modify schemata that happen to be involved in a dependency path, since the system handles the situation gracefully.

6. Functional Interface: Common Functions

This section contains a list of the more common functions and macros exported by the KR interface. It includes the functionality that most Garnet users are likely to need and covers schema representation, object-oriented programming, and constraint maintenance. Section 8 describes parts of the system that are less commonly used.

All functions and variables are defined and exported by the KR package. The easiest way to make them accessible to an application program is to execute the following line:

```
(use-package "KR")
```

Throughout this and the following section, we will use the schemata defined in section 3.1 as examples. All examples assume the initial state described there.

6.1. Schema Manipulation

This group includes functions that create, modify, and delete whole schemata.

`kr:Create-Instance` *object-name prototype* `&rest` *slot-definitions* [*Macro*]

This macro creates an instance of the *prototype* with certain slots filled in; if *prototype* is NIL, the instance will have no prototype. The instance is named *object-name*. If *object-name* is NIL, an unnamed object is created and returned. If *object-name* is a symbol, a special variable by that name is created and bound to the new object.

The *slot-definitions*, if present, are used to create initial slots and values for the object. Each slot definition should be a list whose first element is the name of a slot, and whose second element is a value for that slot.

In addition to this basic slot-filling behavior, this macro also performs three operations that are connected to inheritance and constraint maintenance. First of all, create-instance links the newly created object to the *prototype* via the `:is-a` link, thus making it an instance.

Second, if the *prototype* contains any slot with a formula, and the *slot-definitions* do not redefine that slot, create-instance copies the formula down into the instance. This means that the *prototype* can conveniently provide default formulas for any slot that is not explicitly defined by its instances.

Third, if either the *prototype* or the object itself defines the `:initialize` method, create-instance sends the newly created object the `:initialize` message. This is done after all other operations have been completed, and provides an automatic way to perform object-dependent initializations.

Example:

```
(create-instance 'RECTANGLE-4 BOX-OBJECT (:x 14) (:y 15))
```

The following example demonstrates the use of the `:initialize` method at the prototype level[2]:

[2]Defining methods on Garnet objects is seldom necessary in practice, since real Opal prototypes already have built-in `:initialize` methods.

```
(define-method :initialize BOX-OBJECT (schema)
   (s-value schema :color :RED)
   (format t "~S initialized~%" schema))

(create-instance 'RECTANGLE-4 BOX-OBJECT (:x 14) (:y 15))
;; prints out:
#k<RECTANGLE-4> initialized
```

Create-instance understands the **:override** keyword and the **:name-prefix** keyword; see 8.13 for more details. The uniform declaration syntax with the **:declare** keyword is used to define "local only slots", constant slots, and many others (see section 8.2).

`kr:PS` *schema* [*Function*]

This function prints the contents of the *schema*. In its simplest form, described here, the function is called with the *schema* as its sole argument, and prints out the contents of the schema in a standard format. Optional arguments also allow you to control precisely what is printed out; the more complicated form is described in section 8.14.1.

The following example shows the simple form of `ps`:

```
(ps RECTANGLE-1)
;; prints out:
{#k<RECTANGLE-1>
  :Y =  20
  :X =  10
  :IS-A =  #k<BOX-OBJECT>
  }
```

`kr:Schema-P` *thing* [*Function*]

This predicate returns T if *thing* is a valid KR schema, NIL otherwise. It returns NIL if *thing* is a destroyed schema. It also returns NIL if *thing* is a formula.

```
(schema-p RECTANGLE-1) ==> T
(schema-p 'random) ==> NIL
```

`kr:Is-A-P` *schema* *thing* [*Function*]

This predicate returns T if *schema* is related to *thing* (another schema) via the `:is-a` relation, either directly or through an inheritance chain. It returns NIL otherwise.

Note that *thing* may have the special value T, which is used as a "super-class" indicator; in this case, `is-a-p` returns T if *schema* is any schema. If the *schema* is identical to the *thing*, `is-a-p` also returns T. Examples:

```
(is-a-p RECTANGLE-1 BOX-OBJECT) ==> T
(is-a-p RECTANGLE-1 MY-GRAPHICAL-OBJECT) ==> T
(is-a-p RECTANGLE-1 RECTANGLE-2) ==> NIL
(is-a-p RECTANGLE-1 T) ==> T
```

6.2. Slot and Value Manipulation Functions

This group includes the most commonly used KR functions, i.e., the ones that retrieve or modify the value in a slot. This section presents KR value manipulation functions that deal with constraints. A different set of primitive functions, which do not deal with constraints, is described in Section 8.

6.2.1. Getting Values with G-Value and GV

When called outside of a formula, `g-value` and `gv` behave identically. When used inside a formula, the function `gv` not only returns the value of a slot, but also establishes a dependency for the formula on the slot. This special property of `gv` is discussed in section 6.2.4.

Novice Garnet users only need to learn about `gv`, but `g-value` is supplied for the rare case in which you want to retrieve a slot value from inside a formula without establishing a dependency.

`kr:Gv` *object* `&rest` *slot-names* [*Macro*]

`kr:G-Value` *object* `&rest` *slot-names* [*Macro*]

These macros return the value in a slot of an object. If the slot is empty or not present, they return NIL. Inheritance may be used when looking for a value. `G-value` and `gv` handle constraints properly: If a formula is currently installed in the slot, the value is computed (if needed) and returned. `G-value` will work in place of `gv` in any of the following examples:

```
(gv RECTANGLE-1 :is-a) ==> #k<BOX-OBJECT>
(gv RECTANGLE-1 :thickness) ==> 1    ; inherited
(gv RECTANGLE-1 :color) ==> :BLUE
(gv RECTANGLE-2 :y) ==> 35           ; computed formula
```

Although it is common to call `g-value` and `gv` with only one slot name, these macros may actually be given any number of *slot-names*. The macros expand into repeated calls to `g-value` and `gv`, where each slot is used to retrieve another object. The given slot in the final object (which is, in general, different from the *object*) is then accessed. For example:

```
(gv RECTANGLE-2 :left-obj :x)
```

is equivalent to

```
(gv (gv RECTANGLE-2 :left-obj) :x)
```

Both expressions return the value of the `:x` slot of the object which is contained in the `:left-obj` slot of RECTANGLE-2. One can think of the slot `:left-obj` as providing the name of the place from which the next slot can be accessed. Such a slot is often called a *link*, since it provides a link to another object which can be used to compute values.

6.2.2. Setting Values with S-Value

`kr:S-Value` *object slot* [*more-slots*] *value* [*Function*]

This function is used to set a slot with a given value or formula. The *slot* in the *object* is set to contain the *value*. Like with `g-value` and `gv`, `s-value` can be given multiple slots in argument list, when the slot to be set is several levels away from *object*. In the normal case, *value* is an ordinary LISP value and simply supersedes any previous value in the slot. If *value* is a formula (i.e. the result of a call to `o-formula` or `formula`), the formula is installed in the *slot* and internal bookkeeping information is set up appropriately.

If the *slot* already contains a formula, the following two cases arise. If *value* is also a formula, the old formula is replaced and any dependencies are removed. If *value* is not a formula, the old formula is kept in place, but the new *value* is used as its new, temporary cached value. This means that the *slot* will keep the *value* until such time as the old formula needs to be re-evaluated (because some of the values on which it depends are modified).

`S-value` returns the new value of the *slot*.

Note that a **setf** form is defined for `gv` and `g-value`, and expands into `s-value`. This allows a variety of LISP constructs to be used in combination with `gv` and `g-value`, such as the idiom

```
(incf (gv object slot))
```

which increments the value of a slot in the object. For example,

```
;; Change value in depended slot from 20 to 21
(incf (gv RECTANGLE-1 :y))
;; The constraint is propagated to RECTANGLE-2:
(gv RECTANGLE-2 :y) ==> 36          ; recomputed
```

Constraint propagation is fully enforced during this operation, just as it would be in the equivalent expression

```
(s-value RECTANGLE-1 :y (1+ (gv RECTANGLE-1 :y)))
```

6.2.3. Formula and O-Formula

`kr:O-Formula` *form* `&optional` *initial-value* [*Macro*]

Given a *form*, this macro returns a formula (formulas are internally represented by special structures). The *form* typically consists of Lisp expressions and `gv` or `gvl` references (see below).
Examples:

```
(o-formula (gvl :ABOVE-GADGET :x))
(o-formula (min (gvl :ABOVE-GADGET :x)
                (+ (gvl :OTHER-GADGET :width) 10)))
```

The first example creates a formula which causes the slot on which it is installed to have the same value as slot `:x` of the object contained in slot :ABOVE-GADGET of the current object. The second formula is more complex and constrains the slot on which it is installed to have as its value the minimum of two values. One value is computed as before, and the other is computed by adding 10 to the `:width` slot of the object contained in slot :OTHER-GADGET of the current object.

The *form* can also be an existing formula, rather than a Lisp expression. In this case, the new formula is linked to the existing formula, and inherits the expression from it. No local state is shared by the two formulas. This form of the call should be used as often as possible, since inherited formulas are smaller and more efficient than top-level formulas. An illustration of this case is given by the second call in the following example, which creates a new formula that inherits its expression from the first one:

```
(setf f (o-formula (+ (gvl :ABOVE :y)
                      (floor (gvl :ABOVE :height) 2))))
(setf g (o-formula f))
```

If an *initial-value* is specified, it is used as the initial cached value for the formula. This cached value is recorded in the formula but marked invalid, and thus it is never used under normal circumstances. The initial value is only used if the formula is part of a circular dependency, or if one of its dependency paths is invalid. Most applications need not be concerned about this feature.

A reader macro has been defined to abbreviate the definition of o-formulas. Instead of typing `(o-formula (...))`, you could type `#f(...)`, which expands into a call to `o-formula`. For example, one may write:

```
(s-value a :left #f(gvl :top))
```

instead of the equivalent expression

```
(s-value a :left (o-formula (gvl :top)))
```

`kr:Formula` *form* `&optional` *initial-value* [*Function*]

Given a *form*, this function returns a formula. It is similar to `o-formula`, except that the code in *form* is not compiled until run-time, when the `formula` call is actually executed.

Code that can be compiled early should use `o-formula`, which yields more efficient formula evaluation and reduces the amount of storage. `Formula` might be required when local variables are used in *form*, and are not set until run-time. See the "Hints and Caveats" section of the Tutorial for more discussion of when a formula created with `formula` might be needed.

`kr:Formula-P` *thing* [*Macro*]

A predicate that returns T if the *thing* (any Lisp object) is a formula created with `o-formula` or `formula`, NIL otherwise.

6.2.4. GV and GVL in Formulas

`kr:Gv` *object* `&rest` *slot-names* [*Macro*]

This macro, which we saw in section 6.2.1, serves a special purpose when used within formulas. In addition to returning a value like `g-value`, `gv` records the dependency path and ensures that the formula in which it is embedded is recomputed whenever the dependency path or the value changes.

Note that the *object* can be any object, not just the one on which the formula containing `gv` is installed. Specifying the reserved name `:self` for *object* ensures that the path starts from the object on which the formula is installed. This can be achieved more simply via `gvl`, as explained below.

The following examples show how to use `gv` within formulas:

```
(o-formula (gv RECTANGLE-1 :y))
(o-formula (+ (gv :self :x) 15))
(o-formula (equal (gv :self :other :other :color)
                  (gv :self :color)))
```

As a special case, the expression `(gv :self)` (without any slot name) may be used within a formula to refer to the object to which the formula is attached. This is sometimes useful for formulas which need a way to explicitly reference the object on which they are currently installed.

`kr:Gvl` *slot* `&rest` *more-slots* [*Macro*]

This is a useful shorthand notation for `(gv :self` *slot more-slots*`)`. It may only be used within formulas, since it looks for *slot* in the object on which the surrounding formula is installed. For example, the expression `(gvl :color)` returns the current value of the `:color` slot in the object which contains the formula, and is equivalent to the expression `(gv :self :color)`.

6.3. Object-Oriented Programming

This group includes functions which support objected-oriented programming within KR.

`kr:Define-Method` *slot-name object arg-list* `&rest` *body* [*Macro*]

This macro defines a method named *slot-name* for the *object*. While *object* can be any object, and in particular any instance, it is customary to define methods at the level of prototypes; this allows prototypes to provide methods for all their instances.

The method is defined as a function whose argument list is *arg-list* and whose body is given by the *body*.

The method is installed on slot *slot-name*, which is created if needed. In order to facilitate debugging, the function which implements the method is given a meaningful name formed by concatenating the *slot-name*, the string "-METHOD-", and the name of the *object*. Example:

```
(define-method :print BOX-OBJECT (object)
  (format t "A rectangle at (~D,~D).~%"
    (gv object :x) (gv object :y)))
```

After this, the `:print` method can be inherited by any instance of BOX-OBJECT. Sending the message to RECTANGLE-2, for example, would cause the following to happen:

```
(kr-send RECTANGLE-2 :print RECTANGLE-2)
;; prints out:
A rectangle at (34,35).
```

The generated name of the `:print` method, in this example, would be PRINT-METHOD-BOX-OBJECT.

`kr:KR-Send` *object slot* `&rest` *arguments* [*Macro*]

This macro implements the primitive level of message sending. The *slot* in *object* should yield a Lisp function object; the function is then called with the arguments specified in *arguments*. Note that the function may be local to the *object*, or it may be inherited.

If the function, i.e., the value of the expression `(g-value object slot)`, is NIL, nothing happens and `kr-send` simply returns NIL. Otherwise, the function is invoked and the value(s) it returns are returned by `kr-send`.

`kr:Call-Prototype-Method &rest` *arguments* [*Macro*]

This macro can be used inside an object's method to invoke the method attached to the object's prototype. It can only be used inside object methods. If a prototype of the current object (i.e, the one which supplied the method currently being executed) also defines a method by the same name, the prototype's method is invoked with *arguments* as the list of arguments. For example,

```
(define-method :notify A (object level)
  ;; Execute object-specific code:
  ;; ...
  ;; Now invoke :notify method from prototype, if any:
  (call-prototype-method object level))))

(kr-send A :notify A 10)
```

First of all, `kr-send` invokes the method defined locally by object A. Since the method itself contains a call to `call-prototype-method`, the hierarchy is searched for a prototype of object A which also defines the `:notify` method. If one exists, that method is invoked.

A method is free to supply a prototype method with any parameters it wants; this can be accomplished simply by using different values in the call to `call-prototype-method`. In the example above, for instance, we could have written `(call-prototype-method object (+ level 1))`. It is customary, however, to invoke `call-prototype-method` with exactly the same parameters as the original call.

Note that the name of the original object and the message name are not specified in `call-prototype-method`. KR automatically provides the right values.

`kr:Apply-Prototype-Method &rest` *args* [*Macro*]

The macro `apply-prototype-method` is similar to `call-prototype-method`, but the method defined by the prototype is invoked using `apply` rather than `funcall`. This macro may be useful for methods that take `&rest` arguments.

`kr:Method-Trace` *object message-name* [*Macro*]

This macro can be used to trace method execution. Trace information is printed every time an instance of the *object* is sent the message named *message-name*. Since this expands into a call to the primitive macro **trace**, the Lisp expression **(untrace)** may be used later to eliminate trace information.
Example:

```
(method-trace BOX-OBJECT :print)
```

6.4. Reader Macros

A reader macro is defined by default for the `#k<...>` notation, which is produced by the functions `ps` and `gv` when the variable `kr::*print-as-structure*` is non-NIL. This macro allows objects written with the `#k` notation to be read back in as a KR object. If necessary, this feature may be disabled by recompiling KR after pushing the keyword `:no-kr-reader` onto the `*features*` list.

A second reader macro is defined for convenience, as discussed previously. This reader macro allows o-formulas to be entered using the `#f()` notation, which expands into a call to `o-formula`. For example, one may write:

```
(s-value a :left #f(gvl :top))
```

instead of the equivalent expression

```
(s-value a :left (o-formula (gvl :top)))
```

7. The Type-Checking System

KR supports complete type-checking for slots. Any slot in any object can be declared of a certain type. Slots can be declared with one of the pre-defined types Garnet provides, which cover most of the commonly occurring situations, or new types may be created as needed using the macro `def-kr-type` (see section 7.1). Type expressions use the same syntax as in the Common Lisp type system. Type declarations are inherited, so it is generally not necessary to specify types for the slots of an instance (unless, of course, the instance is to behave differently from the prototype).

Every time the value of a typed slot changes, KR checks that the new value is compatible with the declared type of the slot. If not, a continuable error is generated. More specifically, the type of a value is checked against the type specification for a slot under the following circumstances:

- when the slot is first created using `create-instance`: if a value is specified and the value is of the wrong type, an error is generated;
- when a slot is set to a certain value using `s-value`;
- when the value in a slot is computed by a formula, and the formula is evaluated;
- when the type of a slot that already contains a value is changed using `s-type` (see below).

This mechanism ensures that potential problems are detected immediately; without type-checking, it would be possible for a bad value in a slot to cause hard-to-track errors later on. For example, if a slot in an object is supposed to contain an integer value, a formula in another object would typically assume that the value is correct, and compute an expression such as `(+ 10 (gv obj :left))`. If the value in the slot `:left` is incorrectly set to NIL, however, this would not cause an error until much later, when the formula is actually recomputed and the operator + is given a null value. When type-checking is enabled, on the other hand, the user would see an error immediately when the value is set to NIL, because NIL does not meet the "integer" declaration.

The KR type-checking mechanism is independent of the lisp type system. Even if a type is defined with lisp's `deftype`, another corresponding type must be defined with KR's `def-kr-type`. The two types may have the same name. The important thing is that the new type must be registered with KR's type system.

Type-checking may be turned off completely, for maximum performance in finished systems, by setting the variable `kr::*types-enabled*` to NIL. However, the performance overhead associated with type-checking is small, and we recommend that you always keep type-checking enabled. This ensures early detection of problems that might otherwise be difficult to track down.

7.1. Creating Types

New types may be declared as needed with the macro `def-kr-type`, which is exported from the KR package. The syntax of the macro is as follows:

`kr:Def-KR-Type` *name-or-type* `&optional` *args body doc-string* [*Macro*]

This macro defines a new type for KR's type-checking mechanism. Every type used in slot declarations must have been defined with `def-kr-type`. However, Garnet already predefines the most common types, so you do not have to worry about those.

The macro may be called in two different styles, one named, one unnamed. The first style is used to define types that have a name; you may then use either the name or the corresponding expression in actual type declarations. The second style simply defines a type expression, which is not named and hence must be used verbatim in type declarations. Here are examples of the two styles:

```
(def-kr-type my-type () '(or keyword null))

(def-kr-type '(or keyword null))
```

The first style uses the same syntax as Lisp's `deftype`; the *body* should be a type expression acceptable to `deftype`, and is used for typechecking when the name is used. In the current implementation of the type system, the *args* parameter should always be NIL.[3] With either example above you could then specify some object's type to be `'(or keyword null)`. With the first style, however, you could also specify the type to be `'my-type`, which may be more convenient and easier to maintain in the long run. The named style also allows a *doc-string* to be specified. This is a human-readable documentation string that is associated with the type, and is useful for debugging purposes. For example, the first call above could be written as:

```
(def-kr-type my-type () '(or keyword null)
   "Either NIL or a keyword")
```

7.2. Declaring the Type of a Slot

Types are associated with slots either statically or dynamically. The former mechanism is by far the most common, and is done at object creation time using the :DECLARE option in `create-instance`. For example, consider the following code:

```
(create-instance 'R1 opal:rectangle
  :declare (:type (integer :left :top)
                  ((integer 0) :width :height)
                  ((or keyword null) :link-name))
  (:link-name :PARENT)
  (:left 10) (:height (+ 15 (o-formula (gvl :width)))))
```

The example declares that the values contained in slots `:left` and `:top` must be integers, the values in slots `:width` and `:height` must be positive integers, and the value in slot `:link-name` must be either a keyword or NIL. Note that this declaration is legal, as the type `(or keyword null)` was declared above using `def-kr-type`. Note also that the declarations for slots `:left`, `:top`, `:width`, and `:height` are, in fact, not necessary, as they would normally be inherited from the prototype.

Types can also be associated with slots dynamically, i.e., after object creation time. This is done with the function

`kr:S-Type` *object slot type* `&optional (`*check-p* `T)` [*Function*]

This function changes the type declaration for the *slot* in the *object* to the given *type*. If *check-p* is non-NIL (the default), the function signals a continuable error if the value currently in the *slot* does not satisfy the new type. Setting *check-p* to NIL disables the error; note that this should only be used with caution, as it may leave the system in an inconsistent state (i.e., the *slot* may in fact contain an illegal value). The function returns the *type* it was given.

The type associated with a slot can be retrieved by the function

`kr:G-Type` *object slot* [*Function*]

If a type is associated with the slot, it is returned (more precisely, if the type is named, the name is returned; otherwise, the type expression is returned). If there is no type, the function returns NIL.

---

[3] The presence of the *args* parameter is to maintain consistency of syntax with the standard lisp function `deftype`. If you need to pass a parameter to your predicate, then define the predicate using `satisfies`.

7.3. Type Documentation Strings

Given a type (for example, something returned by `g-type`), its associated documentation string can be retrieved using:

`kr:Get-Type-Documentation` *type* [*Function*]

This function returns the human-readable type documentation string, or NIL if there is none.

Given a type, it is also possible to modify its string documentation, using the function:

`kr:Set-Type-Documentation` *type doc-string* [*Function*]

This function associates the *doc-string* with the *type*. When an error message which concerns the type is printed, the documentation string is printed in addition to the raw type.

7.4. Retrieving the Predicate Expression

When types are named, `g-type` returns just the name of the type, rather than its associated expression. Sometimes it is useful to retrieve the predicate of the type associated with the type name. The following function serves this purpose:

`kr:Get-Type-Definition` *type-name* [*Function*]

Given a symbol which names a KR type (i.e., a named type defined with `def-kr-type`), this function returns the type expression that was used to define the type. If no such expression is found, the function returns NIL.

7.5. Explicit Type-Checking

In addition to KR's built-in type checking, which happens when the value in a slot is changed, it is also possible to check whether a value is of the right type for a slot. This can be done with the function:

`kr:Check-Slot-Type` *object slot value* `&optional (`*error-p* `T)` [*Function*]

The function checks whether the given *value* is of the right type for the *slot* in the *object*. If not, it raises a continuable error, unless *error-p* is set to NIL; in this case, it returns a string which describes the error. This function is called automatically by KR any time a slot is modified, so you normally do not have to call it explicitly.

7.6. Temporarily Disabling Types

It is possible to execute a piece of code with type-checking temporarily disabled, using the macro

`kr:With-Types-Disabled &body` *body* [*Macro*]

This macro is similar to others, such as `with-constants-disabled`. During the execution of the *body*, type-checking is disabled, and no errors are given if a value does not meet the type specification of its slot. Just as with `with-constants-disabled`, this macro should only be used with caution, as it may leave the system in an inconsistent state.

7.7. System-Defined Types

The following type predicate can be used to declare types:

`kr:Is-A-P` *prototype* [*Type Predicate*]

This is a type predicate, NOT a function or macro; it can only be used within type specifiers. This predicate declares that the value in a slot should be an instance of the *prototype*, either directly or indirectly. The predicate is true of all objects for which a call to the function `kr:is-a` would return true.

For example, the following definition can be used as the type of all rectangles:

```
(def-kr-type rect-type () '(is-a-p opal:rectangle))
```

Garnet defines a number of types, which cover the types of the most commonly used slots. This is the list of pre-defined basic types:

T - Any value satisfies this type.

KR-BOOLEAN - Same as `T`, but specifically intended for slots which take a NIL or non-NIL value, often used as boolean variables.

NULL - Only the value NIL satisfies this type.

STRING - Strings satisfy this type.

KEYWORD - All Lisp keywords satisfy this type.

INTEGER - All integers (fixnums and bignums) satisfy this type.

NUMBER - This type includes all numbers: integers, floating point, complex numbers, and fractions.

LIST - Any list satisfies this type.

CONS - Any cons cell (lists and dotted pairs) satisfies this type.

SCHEMA - Any non-destroyed KR object satisfies this type.

Garnet also defines many non-basic types, which are typically used by many objects throughout the system. The following types do not have a name. They are often used for slots in Opal fonts, line styles, etc. Because they are predefined, you don't need to call `def-kr-type` for them.

```
'(real 0 1)
'(integer 0 1)
'(integer 0)
'(integer 1)
'(integer 2)
'(member 0 1 2 3)
'(or null integer)
'(or null (integer 0))
'(or keyword (integer 0))
'(or number null)
'(member :even-odd :winding)
'(or (member :below :left :right) list)
'(or keyword character list)
'(or list string)
'(or list (member t))
'(or list (satisfies schema-p))
'(or string atom)
'(or string (satisfies schema-p))
'(or function symbol)
'(or list integer function symbol)
'(or null function symbol)
'(or null keyword character)
'(or null string)
'(or null (satisfies schema-p))
'(or null string keyword (satisfies schema-p))
'(or string keyword (satisfies schema-p))
```

The following non-basic types are named, and have associated documentation strings. Users can reference these types anywhere in Garnet programs. To access each type's own documentation string, use `get-type-documentation`.

KNOWN-AS-TYPE - a keyword (this type is used in the `:known-as` slot)

AGGREGATE - an instance of `opal:aggregate`

AGGREGATE-OR-NIL - either an instance of `opal:aggregate` or NIL

BITMAP - an instance of `opal:bitmap`

BITMAP-OR-NIL - either an instance of `opal:bitmap` or NIL

COLOR - an instance of `opal:color`

COLOR-OR-NIL - either an instance of `opal:color` or NIL

FONT - either an instance of `opal:font` or `opal:font-from-file`

FONT-FAMILY - one of `:fixed`, `:serif`, or `:sans-serif`

FONT-FACE - one of `:roman`, `:bold`, `:italic`, or `:bold-italic`

FONT-SIZE - one of `:small`, `:medium`, `:large`, or `:very-large`

FILLING-STYLE - an instance of `opal:filling-style`

FILLING-STYLE-OR-NIL - either an instance of opal:filling-style or NIL

LINE-STYLE - an instance of `opal:line-style`

LINE-STYLE-OR-NIL - either an instance of `opal:line-style` or NIL

INTER-WINDOW-TYPE - a single `inter:interactor-window`, or a list of windows, or T, or NIL.

WINDOW - an instance of `inter:interactor-window`

WINDOW-OR-NIL - either an instance of `inter:interactor-window` or NIL

FILL-STYLE - one of `:solid`, `:stippled`, or `:opaque-stippled`

DRAW-FUNCTION - one of `:copy, :xor, :no-op, :or, :clear, :set, :copy-inverted, :invert, :and, :equiv, :nand, :nor, :and-inverted, :and-reverse, :or-inverted, :or-reverse`

H-ALIGN - one of `:left`, `:center`, or `:right`

V-ALIGN - one of `:top`, `:center`, or `:bottom`

DIRECTION - either `:vertical` or `:horizontal`

DIRECTION-OR-NIL - either `:vertical`, `:horizontal`, or NIL

ITEMS-TYPE - list of items: ("Label2"...)

ACCELERATORS-TYPE - list of lists: ((#\r "Alt-r" #\meta-r)...)

FILENAME-TYPE - a string that represents a pathname

PRIORITY-LEVEL - an instance of `inter:priority-level`

8. Functional Interface: Additional Topics

This section describes features of KR that are seldom needed by casual Garnet users. These features are useful for large application programs, especially ones which manipulate constraints directly, or for application programs which use the more advanced knowledge representation features of KR.

8.1. Schema Manipulation

`kr:Create-Schema` *object-name* `&rest` *slot-definitions* [*Macro*]

This macro creates and returns a new object named *object-name*. It is much more primitive than create-instance, since it does not copy down formulas from a prototype and does not call the `:initialize` method.

If *object-name* is NIL, an unnamed object is created and returned. If *object-name* is a symbol, a special variable by that name is created and bound to the new object. The *slot-definitions*, if present, are used to create initial slots and values for the object. Each slot definition should be a list whose first element is the name of a slot, and whose second element is the value for that slot.

`Create-schema` understands the **:override** keyword and the **:name-prefix** keyword; see 8.13 for more details.

Examples:

```
(create-schema 'RECTANGLE-3 (:is-a BOX-OBJECT) (:x 70))
(create-schema 'RECTANGLE-3 :override (:y 12))   ; add a slot
(create-schema NIL (:is-a MY-GRAPHICAL-OBJECT))
```

`kr:Create-Prototype` *object* `&rest` *slot-definitions* [*Macro*]

This macro is slightly more primitive than create-instance. Unlike create-instance, it does not allow a prototype to be specified directly. Moreover, it does not automatically send the `:initialize` message to the newly created *object*. Like create-instance, it copies formulas from any prototype into the newly created *object*.

The following two examples are roughly equivalent:

```
(create-instance NIL BOX-OBJECT (:x 12))

;;; The hard way to do the same thing
(let ((a (create-prototype NIL
           (:is-a BOX-OBJECT) (:x 12))))
  (kr-send a :initialize a))
```

Most applications will find create-instance much more convenient. The only case when `create-prototype` should be used is when it is important that the `:initialize` message *not* be sent to an object at creation time.

`Create-prototype` also understands the **:override** keyword and the **:name-prefix** keyword; see 8.13 for more details.

`kr:Destroy-Schema` *object* [*Function*]

Destroys the *object*. Returns T if the object was destroyed, NIL if it did not exist. This function takes care of properly removing all constraint dependencies to and from the *object*. Any formula installed on any slot of the *object* is also destroyed.

Usually, Garnet users do not call this function directly. Instead, they use `(opal:destroy object)`, which performs all necessary clean-up operations and eventually calls `destroy-schema`.

`kr:Destroy-Slot` *object slot* [*Function*]

Destroys the *slot* from the *object*. The value previously stored in the slot, if there was one, is lost. All constraints to and from *object* are modified accordingly. The invalidate demon is run on the slot before it is destroyed, ensuring that any changes caused by this action become visible to formulas that depend on the slot. Using `destroy-slot` on slots that are declared constant gives a continuable error. Continuing from the error causes the slot to be destroyed anyway. This behavior can be overridden by using the macro `with-constants-disabled`.

`kr:Name-For-Schema` *object* [*Function*]

Given a *object*, this function returns its name as a string. The special notation *#k<>* is never used, i.e., the name is the actual name of the object. The return value should never be modified by the calling program.

8.2. Uniform Declaration Syntax

One syntax can be used for all kinds of declarations associated with slots in an object. Declarations are generally specified at object creation time. In some cases (notably, in the case of types), it is also meaningful to modify declarations after an object has been created; in such cases, a separate function (such as `s-type`) is provided. (For details on the type-checking mechanism, see Chapter 7.)

The general syntax for declarations in `create-instance` is as follows:

```
(create-instance instance prototype
  [:DECLARE ((declaration-1 [slot1 slot2 ...])
             (declaration-2 [slot1 ...])
             ...)]
  [:DECLARE ((declaration-3 [slot1 ...])
            ...)]
  slot-specifiers ...)
```

The keyword :DECLARE introduces a list of declarations. The keyword may appear more than once, which allows separate groups of declarations. Each group of declarations may contain one or more declarations; if there is only one, a level of parentheses may be omitted. Each declaration in a list consists of a keyword, which specifies what property is being declared, followed by any number of slot names (including zero). All slots are declared of the given property.

Consider the following, rather complex example:

```
(create-instance 'REC A
  :declare ((:type (vector :BOX)
                   (integer :LEFT :TOP)
                   ((or (satisfies schema-p) null) :PARENT))
            (:type ((member :yes :no) :VALUE))
            (:update-slots :LEFT :TOP :WIDTH :HEIGHT :VALUE))
  :declare (:type (list :IS-A-INV))

  (:left (o-formula (+ (gvl :parent :left) (floor (gvl :width) 2))))
  (:top 10))
```

The first declaration group defines types (in two separate lists) and the list of update-slots for the object. Slot `:box` is declared as a Lisp vector; left and top are declared as integers; slot `:parent` must be either null or a valid KR object; and slot `:value` must contain either the value :yes or the value :no. The second declaration group shows the simplified form, in which only one declaration is used and therefore the outside parentheses are dropped.

The following keywords can be used to declare different slot properties:

:CONSTANT - The slots that follow are declared constant. Note that (in this case only) the special value T indicates that the slots in the prototype's `:maybe-constant` slot should be used. (See section 8.5.)

:IGNORED-SLOTS - The slots that follow will not be printed by the function `ps`. (See section 8.14.1.)

:LOCAL-ONLY-SLOTS - The values that follow should be lists of the form *(slot-name copy-down-p)*. The *slot-name* specifies the name of a slot which should be treated as local-only, i.e., should not be inherited by the object's instances. If *copy-down-p* is NIL, the slot will have value NIL in the instances. Otherwise, the value from the object will be copied down when instances are created and marked as local; this prevents further inheritance, even if the value in the prototype is changed. (See section 8.12.)

:MAYBE-CONSTANT - Specifies the list of slots that can be made constant in this object's instances simply by specifying the special value T. (See section 8.5.)

:OUTPUT - Specifies the list of output slots for the object, i.e., the slots that are computed by formulas and may provide useful output values for communication with other objects.

:PARAMETERS - Specifies the list of parameters for the object, i.e., the slots designed to allow users to customize the appearance or behavior of the object. This slot is used extensively in the Garnet Gadgets to indicate user-settable slots.

:SORTED-SLOTS - Specifies the list of slots (in the appropriate order) that `ps` should always print first. (See section 8.14.1.)

:TYPE - Introduces type declarations for one or more slots. (See chapter 7.)

:UPDATE-SLOTS - The list of update slots for the object, i.e., the slots that should trigger the `:invalidate-demon` when modified. (See section 8.9.2.)

8.2.1. Declarations in Instances

Most inherited declarations follow the standard KR scheme, where a :MAYBE-CONSTANT or :UPDATE-SLOTS declaration in an instance will completely override the declaration in the prototype. One important exception is the :TYPE declaration, which is *additive* from prototype to instance. That is, all of the types declared in a prototype will be valid in its instances, along with any new type declarations in the instance. So you do not need to repeat type declarations in the instances of an object.

For other kinds of declarations besides :TYPE, a convenient syntax has been provided for specifying declarations in instances. If you want all the declarations in a prototype to be inherited by the instance along with several new ones, you could either retype all the declarations in the instance, or you could use the `T` and `:except` syntax. For example, it is possible to write

```
(create-instance 'REC A
  :declare ((:output T :new-slot)
            (:parameters T :except :left)))
```

to indicate that object REC's list of output slots includes all the ones declared in object A, plus the `:new-slot`. Also, the list of parameter slots is equal to the one in A, minus the slot `:left`.

Declarations made in a prototype can be eliminated with an empty declaration in an instance. This may be particularly convenient for declarations such as :MAYBE-CONSTANT. For example, the expression

```
:declare ((:TYPE) (:MAYBE-CONSTANT))
```

in a call to `create-instance` would clear the :MAYBE-CONSTANT declarations from the prototype, and eliminate all type declarations.

However, note that redefining the :CONSTANT declaration may not yield the expected results. When a slot becomes constant in a prototype, that slot will be constant for all instances. This makes sense because any formulas in the prototype that relied on the constant slot have been eliminated, and cannot be restored in the instance. See section 8.5 for an elaborate discussion of constant slots.

8.2.2. Examining Slot Declarations

The following functions may be used to determine what slot declarations are associated with a particular slot in an object, or to retrieve all slot declarations for an object. Note that there is no function to alter the declaration on an object after the object has been created, as most properties can only be set meaningfully at object creation time.

`kr:Get-Declarations` *object selector* [*Function*]

Returns a list of all the slots in the *object* that have associated declarations of the type given by *selector*, which should be one of the keywords listed above. If *selector* is `:type`, the return value is a list of lists, such as

```
((:left (or integer null)) (:top (or integer null)))
```

If *selector* is one of other keywords, the function returns a list of all the slots that have the corresponding declaration.

`kr:Get-Slot-Declarations` *object slot* [*Function*]

This function returns a list of all the declarations associated with the *slot* in the *object*. The list consists of keywords, such as :CONSTANT and :UPDATE-SLOT, or (in the case of type declarations) a list of the form `(:type` *type-specification*`)`.

8.3. Relations and Slots

KR supports special slots called *relations*. Relations serve two purposes: allowing inheritance, and automatically creating inverse connections. In addition to a handful of predefined relations, application programs can create new relations as needed via the function `create-relation` (see below).

Slots such as `:is-a`, which enable knowledge to be inherited from other parts of a network, are called *inheritance relations*. Inheritance along such relations proceeds depth-first and may include any number of steps. The search terminates if a value is found, or if no other object can be reached.

Any relation, including user-defined ones, may also be declared to have an inverse relation. If this is the case, KR automatically generates an inverse link any time the relation is used to connect one object to another. Imagine, for instance, that we defined `:part-of` to be a relation having `:has-parts` as its inverse. Adding object A to the slot `:part-of` of object B would automatically add B to the slot `:has-parts` of object A, thereby creating a reverse link.

KR automatically maintains all relations and inverse relations, and the application programmer does not have to worry about them. In the example above, if slot `:part-of` in object B is deleted, the value B is also removed from the slot `:has-parts` of object A. The same would happen is object B is deleted. This ensures that the state of the system is consistent at any point in time, independent of the particular sequence of operations.

The following functions handle user-defined relations and slots:

`kr:Create-Relation` *name inherits-p* `&rest` *inverses* [*Macro*]

Declares the slot *name* to be a relation. The new relation will have *inverses* (a possibly empty list of slot names) as its inverse relations. If *inherits-p* is non-NIL, *name* becomes a relation with inheritance, and values may be inherited through it.

The following form defines the non-inheritance relation `:has-parts` and its two inverses, `:part-of` and `:subsystem-of`:

```
(create-relation :has-parts NIL :part-of :subsystem-of)
```

`kr:Relation-P` *thing* [*Macro*]

This predicate returns NIL if *thing* is not a relation, or a non-NIL value if it is the name of a relation slot. Examples:

```
(relation-p :is-a) ==> non-NIL value
(relation-p :color) ==> NIL
```

`kr:Has-Slot-P` *object slot* [*Function*]

A predicate that returns T if the *object* contains a slot named *slot*, NIL otherwise. Note that *slot* must be local to the *object*; inherited slots are not considered.
Examples:

```
(has-slot-p RECTANGLE-1 :is-a)  ==> T
(has-slot-p RECTANGLE-1 :thickness) ==> NIL    ; not local
```

`kr:DoSlots (`*slot-var object* `&optional` *inherited*`) &rest` *body* [*Macro*]

Iterates the *body* over all the slots of the *object*. The *slot-var* is bound to each slot in turn. The *body* is executed purely for side effects, and `doslots` returns NIL. Example:

```
(doslots (slot RECTANGLE-1)
  (format t "Slot ~S has value ~A~%"
          slot (gv RECTANGLE-1 slot)))
;; prints out:
Slot :Y has value 20
Slot :X has value 10
Slot :IS-A has value #k<BOX-OBJECT>
```

By default, `doslots` only iterates over the local slots of *object*. But if the *inherited* parameter is T, then all slots that have been inherited from the *object*'s prototype will be iterated over as well. Note: Only those slots that have actually been inherited will be included in the list of inherited slots. If they are merely defined in the prototype and have not been `gv`'d in the instance, then they will not be included in the iteration list. See the description of the function `ps` in section 8.14.1 for a way to display all the slots that could possibly be inherited by the object.

8.4. Constraint Maintenance

These functions are concerned with the constraint maintenance part of KR.

`kr:Change-Formula` *object slot form* [*Function*]

If the *slot* in *object* contains a formula, the formula is modified to contain the *form* as its new function. `change-formula` works properly on any formula, regardless of whether the old function was local or inherited from another formula. If formula inheritance is involved, this function makes sure that all the links are modified as appropriate. If the *slot* does not contain a formula, nothing happens.

Note that this function cannot be used to install a fixed value on a slot where a formula used to be; `change-formula` only modifies the expression within a formula.

`kr:Recompute-Formula` *object slot* [*Function*]

This function can be called to force a formula to be recalculated. It may be used in situations where a formula depends on values which are outside of KR (such as application data, for example). The formula stored in the *slot* of the *object* is recalculated. Formulas which depend on the *slot*, if any, are then marked invalid.

`kr:Mark-As-Changed` *object slot* [*Function*]

This function may be used to trigger constraint propagation for a *object* whose *slot* has been modified by means other than `s-value`. Some applications may need to use destructive operations on the value in a slot, and then notify the system that certain values were changed. `Mark-as-changed` is used for this purpose.

`kr:Copy-Formula` *formula* [*Function*]

This function returns a copy of the given *formula*, which should be a formula object. The copy shares the same prototype with the *formula*, and its initial value is the current cached value of the *formula*.

`kr:Move-Formula` *from-object from-slot to-object to-slot* [*Function*]

This function takes a formula from a slot in an object and moves it to another slot in another object. This function is needed because one cannot move a formula from one slot to another simply by storing the formula in some temporary variable (this creates potentially serious problems with formula dependencies).

`kr::Make-Into-O-Formula` *formula* `&optional` *compile-p* [*Function*]

This function modifies formulas created using the function `formula` to behave as if they were created using `o-formula`. This is useful for tools like Lapidary that need to construct formulas on the fly. The converted formulas will be handled properly by functions such as `opal:write-gadget`. It is also possible to specify that the formula's expression be compiled during the transformation. If *compile-p* is non-NIL, the *formula*'s expression is compiled in the process.

`kr:G-Cached-Value` *object slot* [*Function*]

This function is similar to `gv` if the *slot* contains an ordinary value. If the *slot* contains a formula, however, the cached value of the formula is returned even if the formula is invalid; the formula itself is never re-evaluated. Only advanced applications may need this function, which in some cases returns out-of-date values and therefore should be used with care.

`kr:Destroy-Constraint` *object slot* [*Function*]

If the *slot* of the *object* contains a formula, the constraint is removed and replaced by the current value of the formula. The formula is discarded and all dependencies are updated. Dependent formulas are notified that the formula has been replaced by the formula's value, even if the actual value does not change. If the *slot* contains an ordinary value, this function has no effect.

Note that the expression `(s-value object slot (gv object slot))` cannot be used to simulate `destroy-constraint`. This is because `s-value` does not remove a formula when it sets a slot to an ordinary value, and thus the expression above would simply set the cached value of the formula without removing the formula itself.

`kr::With-Dependencies-Disabled &body` *body* [*Macro*]

This macro can be used to prevent the evaluation of `gv` and `gvl` inside formulas from setting up dependencies. Inside the body of the macro, `gv` and `gvl` effectively behave (temporarily) exactly like `g-value`. This macro should be used with great care, as it may cause formulas not to be re-evaluated if dependencies are not set up correctly.

8.5. Constant Formulas

It is possible to declare that certain slots are constant, and cause all formulas that only depend on constant slots to be eliminated automatically. The main advantage of this approach is that it reduces storage and execution time.

A slot in an object can be declared constant at object creation time. This guarantees that the application program will never change the value of the slot after the object is created. When a formula is evaluated for the first time, KR checks whether it depends exclusively on constant slots. If this is the case, the formula is eliminated and its storage is reused. The slot on which the formula was originally installed takes the value that was computed by the formula.

A slot can become constant in one of three ways. First, the slot may be declared constant explicitly. This is done by listing the name of the slot in the `:constant` slot of an object (see below for more details), or calling `declare-constant` on the slot after its object has already been created. For example, adding the following code to `create-instance` for object *A* will cause slots `:left` and `:top` to be declared constant in object *A*: `(:constant '(:left :top))`. Note that it is possible for the value of the `:constant` slot to be computed by a formula, which is evaluated once at object creation time.

Second, a slot may become constant because it is declared constant in the object's prototype. In the example above, if object *B* is created with *A* as its prototype, slots `:left` and `:top` will be declared constant in *B*, even if they are not explicitly mentioned in object *B*'s `:constant` slot.

Third, a slot may become constant because it contains a formula which depends exclusively on constant slots. After the formula is removed, the slot on which it was installed is declared constant. Thus, constants propagate recursively through formulas.[4] If you cannot figure out why a formula is not being eliminated, the function `garnet-debug:why-not-constant` and related functions in the Debugging Tools Reference Manual may be useful.

To facilitate the creation of the list of constant slots for an object, the syntax of the `:constant` slot is extended as follows. First, a prototype may specify a list of all the slots that its instances may choose to declare constant. This is done by specifying a list of slot names in the prototype, using the slot `:maybe-constant`. When this is done in the prototype, an instance may choose to declare all of those slots constants by simply adding the value `T` to its `:constant` slot. Note that `T` does *not* mean that *all* slots are constant; it only means that all slots in the `:maybe-constant` list become constant.

It is also possible for the instance to add more constant slots as necessary. Consider the following example:

```
(create-instance 'PROTO NIL (:maybe-constant '(:left :x1 :x2 :width)))
(create-instance 'INST PROTO (:constant '(:top :height T)))
```

No slot is declared constant in the prototype, i.e., object PROTO, because the `:maybe-constant` slot does not act on the object itself. However, because object INST includes the value `T` in its `:constant` slot, the list of constant slots in the instance is the union of the slots that are declared constant locally and the slots named in the `:maybe-constant` slot of the prototype. Therefore, the following slots are constant in INST: `:left, :top, :width, :height, :x1,` and `:x2`.

The slot `:maybe-constant` is typically used in prototypes to specify the list of all the parameters of the instances, i.e., the slots that an instance may customize to obtain gadgets with the desired appearance. Consider, for example, the prototype of a gadget. If the application is such that a gadget instance will never be changed after it is created, the application programmer may simply specify `(:constant '(T))`. This informs the system that all parameters declared by the creator of the prototype are, in fact, constant, and formulas that depend on them can be eliminated once the gadget is created. All of the standard objects and gadgets supply a `:maybe-constant` slot.

The syntax of the `:constant` slot also allows certain slots that appeared in the `:maybe-constant` list to be explicitly excluded from the constant slots in an object. This can be done by using the marker `:except` in the `:constant` slot. The slots following this marker are removed from the list that was specified by the prototype. If a slot was not mentioned in the prototype's `:maybe-constant` slot, the `:except` marker has no effect on the slot. The following is a comprehensive example of the syntax of the `:constant` slot:

```
(create-instance 'INST-2 PROTO
   (:CONSTANT '(:top :height T :EXCEPT :width :x2)))
```

As a result, these slots are declared constant in object INST-2: `:left, :top, :height,` and `:x1`.

It is an error to set slots that have been declared constant. This can happen in three cases: a slot may be set using `s-value` after having been declared constant, a call to `create-instance` may redefine in the instance a slot that was declared constant in the prototype, or `destroy-slot` may be used. In all cases, a continuable error is signaled. Note that this behavior can be overridden by wrapping the code in the macro `with-constants-disabled` (see below).

`kr:Declare-Constant` *object slot* [*Function*]

The function `declare-constant` may be used to declare slots constant in an object after creation time.

[4]In the most elegant programming style, a minimum number of constants will be declared in an object, and formulas will be allowed to become constant because of their dependencies on the constant slots (rather than bluntly declaring the formulas constant). This is certainly not a requirement of programming with constants, however.

The function takes an object and a slot, which is declared constant. The behavior is the same as if the slot had been declared in the `:constant` slot at instance creation time, although of course the change does not affect formulas which have already been evaluated. The `:constant` slot of the object is modified accordingly: the new slot is added, and it is removed from the `:except` portion if it was originally declared there. As a special case, if the second argument is `T` all the slots that appear in the slot `:maybe-constant` (typically inherited from a prototype) are declared constant. This is similar to specifying T in the `:constant` slot at instance creation time.

If `declare-constant` is executed on a slot while constants are disabled (i.e., inside of a `with-constants-disabled` body), the call will have no effect and the slot will not become constant.

`kr:With-Constants-Disabled &body` *body* [*Macro*]

The macro `with-constants-disabled` may be used to cause all constant declarations to be temporarily ignored. During the execution of the body, no error is given when slots are set that are declared constant. Additionally, constant declarations have no effect when `create-instance` is executed inside this macro. This macro, therefore, is intended for experienced users only.

Several functions in the `garnet-debug` package (loaded with Garnet by default) can be helpful in determining which slots in your application should be declared constant for maximum benefit, and can help you determine why some slots are not becoming constant. These functions are documented in the Debugging Tools Reference Manual, which starts on page 461:

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

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

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

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

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

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

8.6. Efficient Path Definitions

The function `kr-path` can be used to improve the efficiency of formula access to slots that are obtained via indirect links. Inside formula expressions, macros such as `gv` are used to access a slot indirectly, traversing a number of objects until the last slot is obtained. This is sometimes called a *link* or a *path*. For example, the expression `(gvl :parent :parent :left)` will access the `:left` slot in the parent's parent. If the application program can guarantee that the intermediate path will not change, the function `kr-path` provides better performance. The expression above could be written as:

```
(gv (kr-path 0 :parent :parent) :left)
```

The call to `kr-path` computes the object's parent's parent only once, and stores the result as part of the formula. Subsequent evaluations of the formula only need to access the `:left` slot of the target object. The syntax is:

`kr:KR-Path` *path-number* `&rest` *slots* [*Macro*]

The path-number is a 0-based integer which indicates the number of this path within the formula expression. In most cases, a formula contains only one call to `kr-path`, and path-number is 0. If more than one path appears in a formula expression, different numbers should be used. For example,

```
(or (gv (kr-path 0 :parent :parent) :left)
    (gv (kr-path 1 :alternate :parent) :left))
```

Note that `kr-path` can only be used inside a formula expression.

8.7. Tracking Formula Dependencies

The function `kr::i-depend-on` can be used to find out all the objects and slots upon which a certain formula depends directly. The syntax is:

`kr::I-Depend-On` *object slot* [*Function*]

If the *slot* in the *object* does not contain a formula, this function returns NIL. Otherwise, the function returns a list of dotted pairs of the form `(obj . slot)`, which contains all the slots upon which the formula depends. Note that this is the list of only those slots that are used by the formula directly; if some of those slots contain other formulas, `kr::i-depend-on` does not pursue those additional formulas' dependencies.

Example:

```
(create-instance 'A NIL (:left 7))
(create-instance 'B A (:left 14) (:top #f(+ (gvl :left) (gv a :left))))
(gv b :top)     ; set up the dependencies

(kr::i-depend-on B :top)
    ==> ((B . :LEFT) (A . :LEFT))
```

8.8. Formula Meta-Information

It is possible to associate arbitrary information (sometimes known as meta-information) with formulas, for example for documentation or debugging purposes. Meta-information is internally represented by a KR object which is associated with the formula; this allows essentially any slot to be added to formulas. Meta-information can be inherited from parent formulas, and is copied appropriately by functions such as `copy-formula`.

In addition, it is possible to access built-in formula information (such as the lambda expression that was used to create the formula) using exactly the same mechanism that is used to access meta-information. This provides a single, well-documented way to access all information associated with a formula.

8.8.1. Creating Meta-Information

Meta-information can be specified statically at formula creation time, and also dynamically for already existing formulas. Static meta-information is specified by additional parameters to the functions `formula` and `o-formula`. The additional parameters are slot specifications, in the style of `create-instance` (except that, of course, special `create-instance` keywords such as :DECLARE or :OVERRIDE are not supported). For example, the expression:

```
(o-formula (gv a :top) 15
  (:creator 'GILT) (:date "today"))
```

creates a new formula with initial value 15, and two meta-slots named `:creator` and `:date`.

Note that in order to specify meta-information statically, one has to specify the default initial value for the formula, which is also an optional parameter.

Meta-information may also be created dynamically, using the function

`kr:S-Formula-Value` *formula slot value* [*Function*]

This function sets the value of the meta-slot *slot* in the *formula* to be the specified *value*. If the *formula* does not already have an associated meta-object, one is created.

It is not possible to use this function to alter one of the built-in formula slots, such as the formula's lambda expression or its list of dependencies.

8.8.2. Accessing Meta-Information

Meta-information can be retrieved using the function `g-formula-value`. In addition to slots that were specified explicitly, this function also makes it possible to retrieve the values of all the special formula slots, such as the formula's parent or its compiled expression.

`kr:G-Formula-Value` *formula slot* [*Function*]

The function returns the value of meta-slot *slot* for the *formula*. If the latter is not a formula, or the meta-slot is not present, the function returns NIL. If the *formula* inherits from some other formula, inheritance is used to find the meta-slot.

As a convenience, *slot* can also be the name of an internal formula slot, i.e., one of the structure slots used by KR when handling formulas. Such slots should be treated strictly as read-only, and should never be modified by application programs. The built-in slot names are:

:DEPENDS-ON - returns the object, or list of objects, on which the formula depends.

:SCHEMA - returns the object on which the formula is currently installed.

:SLOT - returns the slot on which the formula is currently installed.

:CACHED-VALUE - returns the current cached value of the formula, whether or not the formula is currently valid.

:VALID - returns T if the formula is currently valid, NIL otherwise.

:PATH - returns the path accessor associated with the formula, if any.

:IS-A - returns the parent formula, or NIL if none exists.

:FUNCTION - returns the compiled formula expression.

:LAMBDA - returns the original formula expression, as a lambda list.

:IS-A-INV - returns the list of formulas that inherit from the *formula*, or NIL. If there is only one such formula, a single value (not a list) is returned.

:NUMBER - returns the internal field which encodes the valid/invalid bit, and the cycle counter.

:META - returns the entire meta-object associated with the formula, or NIL if none exists.

When the function `ps` is given a formula, it can print associated meta-information. The latter is printed as an object, immediately after the formula itself. For example:

```
lisp> (create-instance 'A NIL
        (:left (o-formula (gvl :parent :left) 100
                          ;; Supply meta-information here
                          (:name "Funny formula")
                          (:creator "Application-1"))))
#k<A>

lisp> (ps (get-value A :left))          ; prints the following:
{F8
  lambda:        (gvl :parent :left)
  cached value:  (100 . NIL)
  on schema A, slot :LEFT
  }
  ---- meta information (S7):
{S7
  :NAME =  "Funny formula"
  :CREATOR =  "Application-1"
  }
```

8.9. Demons

The demon mechanism allows an application program to perform a certain action when a value is modified. This mechanism, which is totally controlled by the application program, is independent from value propagation. Regular Garnet users do not need to know the contents of this section, since Garnet already defines all appropriate demons. Garnet applications should never modify the default demons, which are defined by Opal and automatically update the graphical representation of the application's objects.

8.9.1. Overview of the Demon Mechanism

A demon is an application-defined procedural attachment to a KR schema. Demons are user-defined fragments of code which are invoked when certain actions are performed on a schema. Whenever the value of a slot in a schema is modified (either directly or as the result of value propagation), KR checks whether a demon should be invoked. This allows application programs to be notified every time a change occurs.

Two separate demons invoked at different times allow an application program to have fine control over the handling of value changes. These demons are only invoked on slots that are listed in the :UPDATE-SLOTS list of a schema (see section 8.9.2).

The first demon is the *invalidate demon*. This demon is invoked every time a formula is invalidated. At the time the demon is invoked, the formula has not yet been re-evaluated, and thus it contains the old cached value. This demon is contained in the `:invalidate-demon` slot of an object. This makes it possible for different objects to provide customized demons to handle slot invalidation.

The second demon is the pre-set demon. It is invoked immediately before the value in a formula is actually modified, and it is passed the new value. This allows the pre-set demon to record the difference between the old and the new value, if needed. This demon is stored in the variable `kr::*PRE-SET-DEMON*`. Garnet does not use the pre-set demon.

The relationship between value propagation and demon invocation is best illustrated by showing the complete sequence of events for the invalidate demon. This is what happens when `s-value` is called to set slot **s** of schema S to value **v**:

1. If slot **s** already contains value **v**, nothing happens.
2. Otherwise, if slot **s** should trigger demons, the demon is invoked. The demon is called with schema S in its *old* state, which means that slot **s** still contains its old value.
3. The change is recursively propagated. All slots whose value is a formula that depends on slot **s** are invalidated. The process is similar to the one described in step 2, but there is no check corresponding to step 1 at this point. Demons are invoked normally on any slot that is modified during this phase.
4. The value of slot **s** is finally changed to **v**.

Both the invalidate demon and the pre-set demon should be functions of three arguments. The first argument is the schema which is being modified. The second argument is the name of the slot which is being modified. The third argument is always NIL for the invalidate demon. For the pre-set demon, the third argument is the new value which is about to be installed in the slot. This allows the pre-set demon to examine both the old value (which is still in the slot) and the new value.

8.9.2. The :Update-Slots List

The KR demons are only invoked on slots that are listed in the :UPDATE-SLOTS list of the schema containing them. For example, Garnet defines a particular demon that is responsible for redrawing the objects in a window as the values of their "interesting" slots change. These "interesting" slots are declared in each object's :UPDATE-SLOTS declaration during `create-instance` (the declaration is usually inherited from the prototype, so that typical Garnet users will never see this declaration). The :UPDATE-SLOTS list contains all the slots in an object that should cause Opal's special demon to be invoked when they are modified. When an update-slot is modified, Opal's demon will "invalidate" the object, causing it to be redrawn during the next pass of the update algorithm.

The :UPDATE-SLOTS list can only be set directly at `create-instance` time. That is, after an object is created it is no longer sufficient to modify the value of the `:update-slots` slot to change whether a slot is an update-slot or not. This is because update-slots are internally represented by a bit associated with the slot, which is set during the `create-instance` call. Instead of setting the `:update-slots` slot, you must call the function:

`kr::Add-Update-Slot` *object slot* `&optional (`*turn-off* `NIL)` [*Function*]

If *turn-off* is NIL (the default), the *slot* in the *object* is declared as an update-slot; if *turn-off* is non-NIL, the slot is no longer an update slot. In addition to setting or resetting the internal bit, the function also modifies the :UPDATE-SLOTS slot accordingly, by adding or removing the *slot* from the list.

8.9.3. Examples of Demons

The following example shows how to define the invalidate demon for an object, and how the demon is invoked.

```
;;; Define an invalidate demon
;;;
(defun inv-demon (schema slot v)
  (declare (ignore v))     ; v is not used
  (format t
          "schema ~s, slot ~s is being invalidated.~%"
          schema slot))

(create-schema 'A (:left 10)
  (:top (o-formula (1+ (gvl :left))))
  (:update-slots '(:top))
  (:invalidate-demon 'inv-demon))

(gv A :top)  ==> 11
(s-value A :left 1)
;; prints out:
schema #k<A>, slot :TOP is being invalidated.
(gv A :top)  ==> 2
```

8.9.4. Enabling and Disabling Demons

`kr:With-Demons-Disabled &body` *body* [*Macro*]

The *body* of this macro is executed with demons disabled. Constraints are propagated as usual, but demons are not invoked.

This macro is often useful when making temporary changes to schemata which have un update demon. This happens, for instance, when a program is changing graphical objects but does not want to display the changes to the user, or when some of the intermediate states would be illegal and would cause an error if demons were to run. Objects may be freely modified inside the *body* of this macro without interference from the demons.

`kr:With-Demon-Disabled` *demon* `&body` *body* [*Macro*]

This is similar to `with-demons-disabled`, except that it allows a specific demon to be disabled. Normally, when `with-demons-disabled` is used, all demons are disabled. This macro allows all demons except a specific one to execute normally; only the specific demon is disabled.

The forms in the *body* are executed, but the given *demon* is not invoked. For example, the following will selectively disable the invalidate demon provided by object FOO:

```
(with-demon-disabled (gv FOO :invalidate-demon)
  (s-value FOO :left 100))
```

While FOO's own demon is not executed, formulas in other objects which depend on FOO's `:left` slot will be invalidated, and the corresponding invalidate demons will be invoked normally.

`kr:With-Demon-Enabled` *demon* `&body` *body* [*Macro*]

This macro enables a particular demon if it had been disabled, either explicitly or with `with-demons-disabled`.

8.10. Multiple Inheritance

KR supports multiple inheritance: a schema may inherit values from more than one direct ancestor. This can be accomplished in two ways. The first way is simply to connect the schema to more than one ancestor schema through a relation. The relation slot, in other words, may contain a list of slots. When performing inheritance, KR searches each ancestor slot in turn until a value is found.

The second way to achieve multiple inheritance is by using more than one relation with inheritance. Any schema may have several slots defined as relations with inheritance; in this case, all relations are searched in turn until a value is found. The two mechanisms may be combined, of course.

Application programs should not rely on the order in which KR searches different relations. The particular order is implementation-dependent.

8.11. Local Values

This group contains functions which deal with local values in a slot. Some of these functions do not treat formulas as special objects, and thus can be used to access formulas stored in a slot (remember that functions like `gv`, for example, return the *value* of a formula, rather than the formula object itself).

`kr:Get-Value` *object slot* [*Macro*]

Returns the value in the *slot* from *object*. If the slot is empty or not present, it returns NIL. Inheritance may be used when looking for a value. Given a slot that contains a formula, `get-value` returns the formula itself, rather than its value. Therefore, its use is limited to applications that manipulate formulas explicitly.

`kr:Get-Local-Value` *object slot* [*Macro*]

Returns the value in the *slot* from *object*. If the slot is empty or not present, it returns NIL. Inheritance is not used, and only local values are considered. Given a slot that contains a formula, `get-local-value` returns the formula itself, rather than the formula's value. Therefore, use of this macro is limited to applications that manipulate formulas explicitly.

`kr:G-Local-Value` *object slot* `&rest` *other-slots* [*Macro*]

This macro is very similar to `g-value`, except that it only considers local values. Inheritance is never used when looking for a value.

`kr:Gv-Local` *object slot* `&rest` *more-slots* [*Macro*]

This macro is similar to `gv`, except that it only considers local values, and it never returns an inherited value. `Gv-local` should be used in situations where it is important to only retrieve values that are local to the *object*.

8.12. Local-only Slots

There are cases when certain slots in an object should not be inherited by any instance of the object. An example of this situation might be a slot which is used as a unique identifier; clearly, the slot should never be inherited, or else errors will occur. Such slots are called *local-only slots*.

This effect can be achieved in KR by listing the names of all such slots in the prototype object. The names are listed in the :LOCAL-ONLY-SLOTS declaration (the general declaration syntax is discussed in section 8.2). This declaration should contain a list of two-element sub-lists. The first element in each sub-list specifies the name of a local-only slot. The second element can be T or NIL.

The value NIL specifies that the local-only slot is always initialized to NIL in any instance which does not define it explicitly. The value T, on the other hand, specifies that the current value of the local-only slot in the prototype will be used to initialize the slot in the instance. The value, however, is physically copied down into the instance, and thus inheritance is no longer used for that instance. Modifying the value in the prototype, in particular, will have no effect on the instance. This second option is used more rarely than NIL.

Note that none of the above applies to slots whose value (in the prototype) is a formula. Slots which contain formulas are always inherited, independent of whether the slot is listed in `:local-only-slots`.

8.13. Schema Creation Options

Two special keywords can be used in the macros that create schemata. These options are recognized by `create-instance`, `create-schema`, and `create-prototype`. They are:

`:OVERRIDE` [*Keyword*]

If the *object-name* in one of the object-creating macros names an existing object, that object is normally deleted, together with its instances, and replaced by a brand new object. The default behavior may be modified by using the keyword **:override** as part of the *slot-definitions*. This keyword causes the existing object to be modified in place and contain the union of its previous slots and those specified by `create-schema`. Previous slots that are not mentioned in the call retain whatever values they had before the operation. For example,

```
(create-schema 'RECTANGLE-1 :override (:color :magenta))
```

adds a slot to the object RECTANGLE-1 if it already exists. Without the `:override` keyword, this would have destroyed the object and created a new one with a single slot.

`:NAME-PREFIX` *string* [*Keyword*]

The keyword **:name-prefix** may be used to specify a name prefix for unnamed objects. Unnamed objects are normally named after the object they are an instance of; this option allows a specific string to be used as the name prefix. The option, if specified, should be immediately followed by a string, which is used as the prefix. Example:

```
(create-schema NIL :name-prefix "ORANGE"
  (:left 34)) ==> #k<ORANGE-2261>
```

8.14. Print Control

This section describes the slots that control what portions of an object are printed, and how they are printed. The need for fine control over printing arises, for example, when certain slots contain very large data structures that take a long time to print.

The print control slots are taken from the object which is specified as the *print-control* in the complicated form of `ps`, described below. In many cases, the slots are actually inherited by the object being printed.

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

This form of `ps` prints the contents of the *object*, and allows fine control over what to print and how. A possible behavior is to print out all slots and all values in *object*; this happens when the *control* object is NIL. It is possible, however, to cause `ps` to ignore certain slots and to specify that others should be printed in a given order. It is also possible to limit the number of elements printed for list values, thus preventing annoyingly long lists of values.

The function `ps` can print out type information, if desired. This can be specified with a non-null value for the new keyword parameter *types-p* (the default value is NIL). Type declarations are printed in square brackets.

Supplying a non-NIL value for the *all-p* parameter will cause `ps` to print out all slots of the *object*, including slots that do not currently have any value. The default for *all-p* is NIL.

The value of *control* should be one of four things:

1. NIL, which means that the *object* is printed in its entirety.
2. T, the default, which means that the *object* itself is used as the control object. In most cases, the control slots are inherited from an ancestor of the *object*. All Opal prototypes, for example, define appropriate slots which reduce the amount of information that is shown by `ps`.
3. an object, which is used directly as the control object.
4. the keyword `:default`, which indicates that the KR-supplied default print control object should be used. The name of the default print control object is PRINT-SCHEMA-CONTROL , an object in the KR package. This default object limits the length of lists that are printed by `ps` to a maximum of ten for ordinary slots, and five for the `:is-a-inv` slot.

If the *inherit* option is NIL (the default), only local slots are printed. Otherwise, all inheritable values from all prototypes of *object* are inherited and printed; inherited values are clearly indicated in the printout. As discussed in Chapter 4, formulas are not copied down from prototypes until they are requested by `gv` or `g-value`. Formulas that have not yet been copied down will not be shown by `ps`, unless the *inherit* parameter is non-NIL.

The `:indent` option is only used by debugging code which needs to specify an indentation level. This option is not needed by regular application programs.

`Ps` prints slots whose value is a formula in a special way. Besides the name of the formula, the current cached value of the formula is printed in parentheses, followed by `T` if the cache is valid or NIL otherwise. Example:

```
(create-schema 'A
  (:left 10) (:right (o-formula (+ (gvl :left) 25))))
(gv A :right)  ==> 35

(ps A)
;; prints out:
{#k<A>
  :DEPENDED-SLOTS =  (:LEFT #k<F2285>)
  :RIGHT =  #k<F2285>(35 . T)
  :LEFT =  10
}
(s-value A :left 50)
(ps A)
;; prints out:
{#k<A>
  :DEPENDED-SLOTS =  (:LEFT #k<F2285>)
  :RIGHT =  #k<F2285>(35 . NIL)
  :LEFT =  50
}
```

The cached value is not correct, of course, but it will be recomputed as soon as its value is requested because formula `F2285` is marked invalid.

The function `ps` prints the expression of a formula, when given the formula as argument. A formula is printed with three pieces of information: the expression, the cached value (which is printed as before), and the object and slot on which the formula is installed.

If *stream* is specified, it is used for printing to a stream other than standard output.

8.14.1. Print Control Slots

If a control object is specified in a call to `ps`, it should contain (or inherit) six special slots. These slots determine what `ps` does. The meaning of the print control slots is as follows:

- `:sorted-slots` contains a list of names of slots that should be printed before all other slots, in the desired order.
- `:ignored-slots` contains a list of names of slots that should not be printed. A summary printed at the end of the object indicates which slots were ignored.
- `:global-limit-values` contains an integer, the maximum number of elements that should be printed for each list that is a value for a slot. If a list contains more than that many elements, ellipsis are printed after the given number to indicate that not all elements of the list were actually displayed.
- `:limit-values` allows the same control on a slot-by-slot basis. It should contain lists of the form `(slot number)`. If a slot name appears in one of these lists, the number specified there is used instead of the one specified in `:global-limit-values`.
- `:print-as-structure` can be `T`, in which case the #k<> notation is used when printing object names, or NIL, in which case only pure object names are printed.
- `:print-slots` is a list of the slots that are printed as part of the *#k<>* notation. It is

possible to cause `ps` to print a few slots from each object, inside the *#k<>* printed representation; this may make it easier to identify different schemata. `:print-slots` should contain a list of the names of the slots which should be printed this way. Note that this option has no effect if schema names are not being printed with the *#k<>* notation.

The following is a rather comprehensive example of fine control over what `ps` prints.

```
; Use top level of the hierarchy to control printing.
(create-schema 'TOP-OBJECT
  (:ignored-slots :internal :width))

(create-schema 'COLORED-THING (:color :blue) (:x 10)
  (:is-a TOP-OBJECT) (:width 12.5) (:y 20)
  (:internal "Some information"))

(dotimes (i 20) (create-instance NIL COLORED-THING))
```

Using `ps` with a null *control* prints out the whole contents of the schema:

```
(ps COLORED-THING :control NIL)
;; prints out:
{#k<COLORED-THING>
  :IS-A-INV =  #k<COLORED-THING-2265>
    #k<COLORED-THING-2266> #k<COLORED-THING-2267>
    #k<COLORED-THING-2268> #k<COLORED-THING-2269>
    #k<COLORED-THING-2270> #k<COLORED-THING-2271>
    #k<COLORED-THING-2272> #k<COLORED-THING-2273>
    #k<COLORED-THING-2274> #k<COLORED-THING-2275>
    #k<COLORED-THING-2276> #k<COLORED-THING-2277>
    #k<COLORED-THING-2278> #k<COLORED-THING-2279>
    #k<COLORED-THING-2280> #k<COLORED-THING-2281>
    #k<COLORED-THING-2282> #k<COLORED-THING-2283>
    #k<COLORED-THING-2284>
  :INTERNAL =  "Some information"
  :Y =  20
  :X =  10
  :COLOR =  :BLUE
  :WIDTH =  12.5
  :IS-A =  #k<TOP-OBJECT>
}
```

Using the system-supplied default control object reduces the clutter in the `:is-a-inv` slot, and also eliminates printing of schemata with the special #k<> convention:

```
(ps COLORED-THING :control :default)
{COLORED-THING
  :WIDTH =  12.5
  :IS-A-INV =  COLORED-THING-2265 COLORED-THING-2266
     COLORED-THING-2267 COLORED-THING-2268
     COLORED-THING-2269 ...
  :INTERNAL =  "Some information"
  :Y =  20
  :X =  10
  :COLOR =  :BLUE
  :IS-A =  TOP-OBJECT
}
```

We can make things even better by using the object itself to inherit the control slots. We add sorting information and a global limit to the number of elements to be printed for each list. We do this at the highest level in the hierarchy, so that every object can inherit the information:

```
(s-value TOP-OBJECT :global-limit-values 3)
(s-value TOP-OBJECT :sorted-slots
  '(:is-a :color :x :y))

(ps COLORED-THING)
;; prints out:
{COLORED-THING
  :IS-A =  TOP-OBJECT
  :COLOR =  :BLUE
  :X =  10
  :Y =  20
  :IS-A-INV =  COLORED-THING-2265 COLORED-THING-2266
     COLORED-THING-2267 ...
  List of ignored slots:   WIDTH INTERNAL
}
```

8.14.2. Slot Printing Functions

It is possible to use the basic mechanism used by the function `ps` to print or format objects in a customized way. This facility is used by applications such as the `garnet-debug:Inspector`, which need full control over how objects are displayed. This mechanism is supported by the following function.

`kr::Call-On-PS-Slots` *object function* `&key (`*control* `T)` *inherit* `(`*indent* `NIL)` *types-p all-p* [*Function*]

The *function* is called in turn on each slot that would be printed by `ps`. The keyword arguments have exactly the same meaning as in `ps`. The *function* should take nine arguments, as follows:

```
(lambda (object slot formula is-inherited valid real-value types-p bits indent limits))
```

When the function is called, the first argument is the object being displayed; the second argument is bound to each slot in the object, in turn. The *formula* is set to NIL (for slots that contain non-formula values), or to the actual formula in the *slot*. The parameter *is-inherited* is T if the value in the *slot* was inherited, NIL if the value was defined locally. The parameter *valid* is NIL if the *slot* contains a formula whose cached value is invalid; it contains T if the slot contains a valid formula, or any non-formula value. The parameter *real-value* is whatever g-value would actually return. The parameter *types-p* is set to T if the *function* should process type information for the *slot*; its value simply reflects the value passed to `kr::call-on-ps-slot`. The parameter *bits* contains the internal bitwise representation of the slot's features and type, as an integer. The parameter *indent* is the level of indentation. The parameter *limits* is a number (the maximum number of values from the *slot* that are to be processed by the *function*), or NIL if all values in the slot should be processed.

A similar function is used when only one slot in an object is to be processed:

`kr::Call-On-One-Slot` *object slot function* [*Function*]

This function returns T if the slot exists and the *function* was called, and NIL otherwise.

8.15. Control Variables

The following variable can be set globally to achieve the same effect as the slot `:print-as-structure` described above:

kr::*PRINT-AS-STRUCTURE* [*Variable*]

This variable may be used to determine whether schema names are printed with the notation #K<NAME> (the default) or simply as NAME. The former notation is more perspicuous, since it makes it immediately clear which objects are KR schemata. The second notation is more compact, and is obtained by setting kr::*PRINT-AS-STRUCTURE* to NIL.

In addition to kr::*PRINT-AS-STRUCTURE*, other special variables can be used to control the behavior of the system. The following variables are used to control what debugging information is printed. The default settings are such that very little debugging information is printed.

kr::*PRINT-NEW-INSTANCES* [*Variable*]

This variable controls whether a notification is printed when `create-schema` or create-instance are compiled from a file. The message is printed when kr::*PRINT-NEW-INSTANCES* is T (the default), and may be useful to determine how far into a file compilation has progressed. Setting this variable to NIL turns off the notification.

kr::*WARNING-ON-NULL-LINK* [*Variable*]

This variable controls whether a notification is printed when a null link is encountered during the evaluation of a formula. When the variable is NIL (the default), the stale value of the formula is simply reused without any warning. Setting the variable to T causes a notification describing the situation to be printed; the formula then returns the stale value, as usual.

kr::*WARNING-ON-CIRCULARITY* [*Variable*]

This variable controls whether a notification is printed when a circularity is detected during formula evaluation. When the variable is NIL (the default), no warnings are generated. Setting the variable to T causes a notification describing the situation to be printed.

kr::*WARNING-ON-CREATE-SCHEMA* [*Variable*]

This variable controls whether a notification is printed when `create-instance` creates an object that has the same name as an old object, and the old object is destroyed. If T (the default), then a warning will be printed when an object is redefined.

kr::*WARNING-ON-EVALUATION* [*Variable*]

This variable controls whether a warning is printed whenever a formula is evaluated. If its value is non-NIL, then a warning will describe the object, slot, and name of any formula that is evaluated. This can be useful for debugging.

kr::*STORE-LAMBDAS* [*Variable*]

The variable kr::*STORE-LAMBDAS* (default T) may be set to NIL to prevent the expression of a formula from being stored in the formula itself. This produces smaller run-time programs, but because the expression is lost it may be impossible to dump a set of objects to a file using `opal:write-gadget`.

9. An Example

This section develops a more comprehensive example than the ones so far, and highlights the operations with which most users of the system should be familiar. Note that this example does not use graphical operations at all; refer to the Opal manual for examples of graphical applications.

We will first construct a simple example of constraints and show how constraints work. The example uses constraints to compute the equivalence between a temperature expressed in degrees Celsius and in degrees Fahrenheit. This first part also illustrates how KR deals with circular chains of constraints.

The second part of the example shows simple object-oriented programming techniques, and illustrates many of the dynamic capabilities of KR. Note that this example is purely indicative of a certain way to program in KR, and different programming styles would be possible even for such a simple task.

9.1. The Degrees Schema

First of all, we will create the DEGREES schema as a demonstration of constraints in KR. This is a schema with two slots, namely, `:celsius` and `:fahrenheit`. The schema can be created with the following call to `create-schema`:

```
(create-schema 'DEGREES
  (:fahrenheit (o-formula (+ (* (gvl :celsius) 9/5) 32)
                          32))
  (:celsius (o-formula (* (- (gvl :fahrenheit) 32) 5/9)
                       0)))
;; and now:
(gv DEGREES :celsius)    ==> 0
(gv DEGREES :fahrenheit) ==> 32
```

Each of the two slots contains a formula. The formula in the `:celsius` slot, for instance, indicates that the value is computed from the value in the `:fahrenheit` slot, using the appropriate expression. The initial value, moreover, is 32. The formula in the `:fahrenheit` slot, similarly, is constrained to be a function of the value in the `:celsius` slot and is initialized with the value 0.

It is clear that this example involves a circular chain of constraints. The value of `:celsius` depends on the value of `:fahrenheit`, which itself depends on the value of `:celsius`. This circularity, however, is not a problem for KR. The system is able to detect such circularities and reacts appropriately by stopping value propagation when necessary.

Consider, for instance, setting the value of the `:celsius` slot:

```
(s-value DEGREES :celsius 20)
(gv DEGREES :celsius)          ==> 20
(gv DEGREES :fahrenheit)       ==> 68
```

As the example shows, KR propagates the change to the `:fahrenheit` slot, which is given the correct value. Similarly, if we modify the value in the `:fahrenheit` slot, we have correct propagation in the opposite direction:

```
(s-value DEGREES :fahrenheit 212)
(gv DEGREES :celsius)          ==> 100
(gv DEGREES :fahrenheit)       ==> 212
```

9.2. The Thermometer Example

Let us now build an example of a thermometer from which one can read the temperature in both degrees Celsius and Fahrenheit, and show a more extensive application of constraints. This example also shows the role of inheritance in object-oriented programming, and a simple method combination.

We begin with TEMPERATURE-DEVICE, a simple prototype which contains a formula to translate degrees Celsius into Fahrenheit (the formula is the same we used in the previous example) and a `:print` method which prints out both values:

```
(create-schema 'TEMPERATURE-DEVICE
  (:fahrenheit
    (o-formula (+ (* (gvl :celsius) 9/5) 32) 32)))


(define-method :print TEMPERATURE-DEVICE (schema)
  (format t "Current temperature: ~,1F C (~,1F F)~%"
          (gv schema :celsius)
          (gv schema :fahrenheit)))
```

We now create two objects to hold the current temperature outdoors and indoors, and we create the schema THERMOMETER, which will be the basic building block for other thermometers:

```
(create-schema 'OUTSIDE
  (:celsius 10))

(create-schema 'INSIDE
  (:celsius 21))

(create-instance 'THERMOMETER TEMPERATURE-DEVICE
  (:celsius (o-formula (gvl :location :celsius))))
```

Note that THERMOMETER can act as a prototype, since it provides a formula which constrains the value of the `:celsius` slot to follow the value of the `:celsius` slot of a particular location. Thermometer schemata created as instances of THERMOMETER will then simply track the value of temperature at the location with which they are associated. Note that instances of THERMOMETER inherit the `:print` method from TEMPERATURE-DEVICE.

```
(create-instance 'TH1 THERMOMETER
                 (:location outside))

(create-instance 'TH2 THERMOMETER
                 (:location inside))

(kr-send TH2 :print TH2)
;; prints out:
Current temperature: 21.0 C (69.8 F)

(kr-send TH1 :print TH1)
;; prints out:
Current temperature: 10.0 C (50.0 F)
```

Since the temperature in the OUTSIDE schema is 10, and thermometer TH1 is associated with OUTSIDE, it prints out the current temperature outside. Changing the slot `:location` of TH1 to INSIDE would automatically change the temperature reading, because of the dependency built into the formula in that slot.

We now want to specialize the THERMOMETER in order to provide a new kind of thermometer that keeps track of minimum and maximum temperature, as well as the current temperature. We do this by creating an instance, MIN-MAX-THERMOMETER, which inherits all the features of THERMOMETER and defines two new formulas for computing minimum and maximum temperatures. Note the initial values in the formulas. Also, we create an instance of MIN-MAX-THERMOMETER named MIN-MAX, and send it the `:print` message.

```
(create-instance 'MIN-MAX-THERMOMETER THERMOMETER
  (:min (o-formula (min (gvl :min)
                        (gvl :location :celsius))
                   100))
  (:max (o-formula (max (gvl :max)
                        (gvl :location :celsius))
                   -100)))

(create-instance 'MIN-MAX MIN-MAX-THERMOMETER
   (:location outside))

(kr-send MIN-MAX :print MIN-MAX)
;; prints out:
Current temperature: 10.0 C (50.0 F)
```

The `:print` method inherited from TEMPERATURE-DEVICE is not sufficient for our present purpose, since it does not show minimum and maximum temperatures. We thus specialize the `:print` method, but we still use the default `:print` method to print out the current values. Let us specialize the method, print out the current status, change the temperature outside a few times, and then print out the status again:

```
(define-method :print MIN-MAX-THERMOMETER (schema)
  ;; print out temperature, as before
  (call-prototype-method schema)
  ;; print out minimum and maximum readings.
  (format t "Minimum and maximum: ~,1F  ~,1F~%"
          (gv schema :min)
          (gv schema :max)))

(kr-send MIN-MAX :print MIN-MAX)
;; prints out:
Current temperature: 10.0 C (50.0 F)
Minimum and maximum: 10.0  10.0

(s-value OUTSIDE :celsius 14)
(kr-send MIN-MAX :print MIN-MAX)
;; prints out:
Current temperature: 14.0 C (57.2 F)
Minimum and maximum: 10.0  14.0

(s-value OUTSIDE :celsius 12)
(kr-send MIN-MAX :print MIN-MAX)
;; prints out:
Current temperature: 12.0 C (53.6 F)
Minimum and maximum: 10.0  14.0
```

Note that the `:fahrenheit` slot in any of these schemata can be accessed normally, and the constraints keep it up to date at all times:

```
(gv MIN-MAX :fahrenheit)  ==> 268/5 (53.6)
```

Finally, we can add a method to reset the minimum and maximum temperature, in order to start a new reading. This is shown in the next fragment of code:

```
(define-method :reset MIN-MAX-THERMOMETER (schema)
  (s-value schema :min (gv schema :celsius))
  (s-value schema :max (gv schema :celsius)))

(kr-send MIN-MAX :reset MIN-MAX)   ; reset min, max

(kr-send MIN-MAX :print MIN-MAX)
;; prints out:
Current temperature: 12.0 C (53.6 F)
Minimum and maximum: 12.0  12.0

(s-value OUTSIDE :celsius 14)

(kr-send MIN-MAX :print MIN-MAX)
;; prints out:
Current temperature: 14.0 C (57.2 F)
Minimum and maximum: 12.0  14.0
```

Other choices of programming style would have been possible, ranging from entirely object-oriented (i.e., without using constraints at all) to entirely demon-based.

10. Summary

KR provides excellent performance and three powerful paradigms: object-oriented programming, knowledge representation, and constraint maintenance. The system is designed for high performance and has a very simple program interface, which makes it easy to learn and easy to use.

The object-oriented programming component of KR is based on the prototype-instance paradigm, which is more flexible than the class-instance paradigm. Prototypes are simply objects from which other objects (called instances) may inherit values or methods. This relationship is completely dynamic, and an object can be made an instance of a different prototype as needed. Object methods are implemented as procedural attachments which are stored in an object's slots. Methods are inherited through the usual mechanism.

The knowledge representation component of KR offers multiple inheritance and user-defined relations. This component provides completely dynamic specification of a network's characteristics: inheritance, for example, is determined through user-specified relations, which the user may modify at run-time as needed. The performance of this component is very good and compares favorably with that of basic Lisp data structures. Inheritance, in particular, is efficient enough to provide the basic building block across a wide variety of application programs.

The constraint maintenance component of KR provides integrated, efficient constraint maintenance and is implemented through formulas, i.e, expressions which compute the value of a slot based on the values in other slots. Constraint maintenance uses lazy evaluation and value caching to yield excellent performance in a completely transparent way. Constraint maintenance is totally integrated with the rest of the system and can be used even without any knowledge of its internal details. The same access functions, in particular, work on both regular values and on values which are constrained by formulas.

In spite of its power, KR is small and simple. This makes it easy to maintain and extend as needed, and also makes it ideally suited for experimentation on efficient knowledge representation. The system is entirely written in portable Common Lisp and can run efficiently on any machine which supports the language. These features make KR an attractive foundation for a number of applications which use a combination of frame-based knowledge representation, object-oriented programming, and constraint maintenance.

References

[Bobrow et al. 89] Bobrow, D.G.; DeMichiel, L.G.; Gabriel, R.P.; Keene, S.E., Kiczales, G.; and Moon, D.A.
Common Lisp Object System Specification.
LISP and Symbolic Computation 1(3/4):245-394, January, 1989.

[Giuse 87] Dario Giuse.
KR: an Efficient Knowledge Representation System.
Technical Report CMU-RI-TR-87-23, Carnegie Mellon University Robotics Institute, October, 1987.

[Giuse 88a] Dario Giuse.
LISP as a rapid prototyping environment: the Chinese Tutor.
LISP and Symbolic Computation 1(2):165-184, September, 1988.

[Giuse 88b] Giuse, D.A.
Intelligent Tutoring Systems for Foreign Language Acquisition.
In *Proceedings of the Asia-Pacific Conference on Computer Education (APCCE 88)*, pages 33-58. Chinese Computer Federation, Shanghai, China, 1988.

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

[Giuse 90] Giuse, D.A.
Efficient Knowledge Representation Systems.
Knowledge Engineering Review 5(1):35-50, 1990.

[Lieberman 86] Henry Lieberman.
Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems.
Sigplan Notices 21(11):214-223, November, 1986.
ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'86.

[Steele 84] Guy L. Steele, Jr. (editor).
Common Lisp; The Language.
Digital Press, 1984.

[Young 89] Sheryl R. Young, Alexander G. Hauptmann, Wayne H. Ward, Edward T. Smith, and Philip Werner.
High-level Knowledge Sources in Usable Speech Recognition Systems.
Communications of the ACM 32(2):183-194, February, 1989.

Index

Table of Contents