KR: Constraint-Based Knowledge Representation

Dario Giuse

April 1989

CMU-CS-89-142 2

The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213

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 the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under contract F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

List of Figures

1. Introduction

Frame systems have received much attention in the last two decades; [Brachman 79] contains a classic treatment of the field and discusses some of the more influential systems. Frame systems are characterized by great flexibility and representational power, and constitute one of the basic lines of research in knowledge representation. [Brachman and Levesque 85] give a recent overview of the field of knowledge representation.

KR [Giuse 87] is a very efficient knowledge representation system in the tradition of frame systems. Simplicity and efficiency are its main design goals and differentiate it sharply from the more conventional frame systems, as discussed in [Giuse 89a]. KR is positioned at the low end of the spectrum and offers superior performance that traditional high-end systems cannot achieve.

In addition to basic representation of knowledge as a network of frames, 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. KR guarantees that a value computed through a formula is constantly up to date, regardless of changes in the network. The constraint system is closely integrated with the basic knowledge representation mechanism and is actually a part of the same program interface.

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

KR is currently being used for a variety of applications which cover the whole range of programming styles. The first application we developed on top of it was the Chinese Tutor [Giuse 88a] [Giuse 88b], an intelligent tutoring system designed to teach Chinese to English speakers. KR is the sole form of knowledge representation in the Chinese Tutor, and is used, among other things, to store the online dictionary of Chinese characters, complete with English translations and various structural hierarchies.

The second major application of KR is the Garnet User Interface Development Environment [Myers 88], an advanced user interface development environment currently under development at the School of Computer Science of Carnegie Mellon University. We are using KR extensively to implement a graphical object system [Vander Zanden et al. 89], a constraint satisfaction system [Giuse 89b], and a graphical object editor [Myers 89] for the interactive creation of user interfaces. Other applications of KR are also being developed at Carnegie Mellon, especially in the area of speech understanding research [Young 89].

This document describes version 2.0 of KR, which is currently in use at the School of Computer Science at Carnegie Mellon University. Several aspects of this version differ from previous versions of the system, such as the one described in a previous report [Giuse 87]. The present document overrides all previous descriptions.

The document 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.

2. Structure of the System

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

The first component, frame-based knowledge representation, stores knowledge as a network of chunks of information. Unlike more traditional data-storage systems (such as relational data bases, for instance) networks in KR are built out of unstructured chunks. Each chunk, known as a frame or *schema*, can store any arbitrary piece of information, and is not restricted to a particular format or data structure. The general way to encode information is via attribute-value pairs.

A program or user is free to use a schema in any given way and to store as much information as needed in it. Moreover, schemata[1] can be modified as needed, even after they have been created. Relational data bases, by comparison, force each chunk to be in one of a small group of possible formats, and the format of a chunk cannot be modified after creation. Frame systems are also more flexible than most object-oriented programming systems, which often prohibit changes to the class structure once instances have been created.

The other important property that KR shares with most frame systems is that certain 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 mechanisms that enable application programs to specify the structure of a network and the relationship among components of the network.

The second component of KR is a simple object-oriented programming system. 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 follows the more flexible prototype-instance paradigm [Lieberman 86], which allows properties of instances to be determined dynamically by their prototypes. Object-oriented programming in KR is heavily based on the dynamic properties of the underlying frame system.

Finally, the third component of KR implements constraint maintenance. This component is logically independent of the first two, which may in fact be used stand-alone. 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 underlying knowledge representation, and for most users the distinction between the two is irrelevant. 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

[1]*Schemata* is the plural of *schema*.

access primitives may be used in both cases.

3. Knowledge Representation in KR

This section describes the first component of KR, i.e., frame-based knowledge representation. 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

A *schema* is the basic unit of representation in KR and consists of an optional *name*, a set of *slots*, and a set of *values* 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 and thus are slightly less convenient in some situations. They are, however, more compact than named schemata and account for the vast majority of schemata created by most applications. Unnamed schemata, in addition, are automatically garbage-collected once they are no longer needed, whereas named schemata have to be destroyed explicitly by the user. KR provides a mechanism to refer to unnamed, as well as named, schemata in interactive situations.

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-values pairs. The slot name indicates the attribute name; the slot values (if any) indicate its values. Slot names should be keywords. All slots in a schema must have distinct names, but different schemata may very well have slots with the same name. Slot names are not interpreted by KR in any way.

Each slot can contain zero or more *values*. Values are the actual data items stored in the schema, and may be of any Lisp type. KR provides functions to add, delete, and retrieve values 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. Consider a schema for John's pet, Fido:

```
{fido
  :is-a dog pet
  :owner John
  :color brown
  :age 5
  }
```

The schema is named FIDO and contains four slots named :IS-A, :OWNER, :COLOR, and :AGE. The slot :AGE contains one value, the integer 5. The slot :IS-A contains two values, DOG and PET.

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

```
(create-schema graphical-object
  (:color :blue)
  (:update-demon 'graphical-object-changed))

(create-schema box-object
  (:is-a graphical-object)
  (:thickness 1))

(create-schema rectangle-1
  (:is-a box-object)
  (:x 10)
  (:y 20))

(create-schema rectangle-2
  (:is-a box-object)
  (:x 34)
  (:y (formula '(+ (gvl :left-obj :y) 15)))
  (:left-obj rectangle-1))
```

The exact meaning of each expression above will become clear after we describe the functional interface of the system. Briefly, however, we can summarize the example as follows. The schema 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. As the example shows, BOX-OBJECT is placed below GRAPHICAL-OBJECT in the hierarchy, since its :IS-A slot points to the schema GRAPHICAL-OBJECT.

Finally, two actual 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 as an arrow going from a schema to the ones to which it is related.

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

```
{RECTANGLE-2
  :IS-A =  BOX-OBJECT
  :X =  34
  :Y =  f1306 (NIL . NIL)
  :LEFT-OBJ =  RECTANGLE-1
  }
```

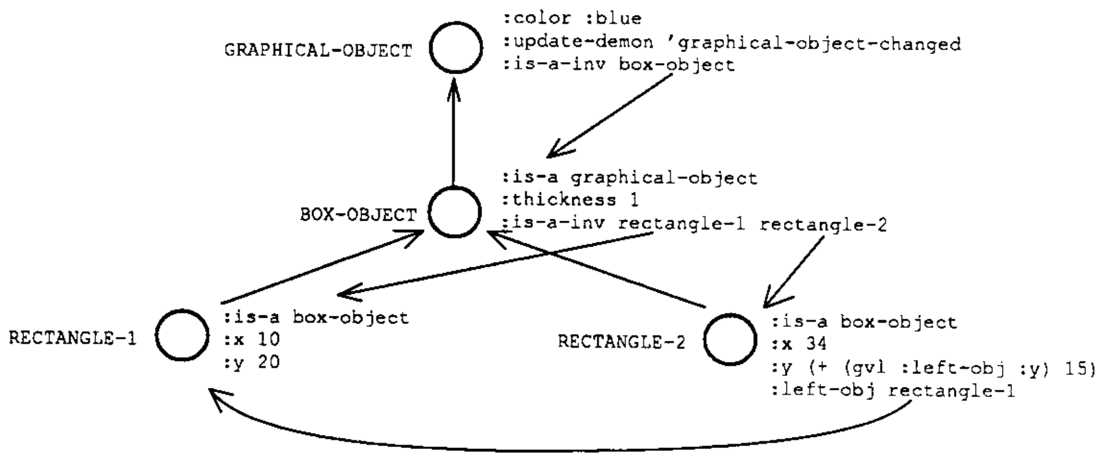

Figure 3-1: The resulting network of schemata

Note that slot :Y contains a formula, which is printed as "f1306 (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. Given the name of the slot, one might reasonably assume this to mean that the former is subordinated to the latter; graphically, this would mean that the position of RECTANGLE-2 is partially determined by that of RECTANGLE-1.

KR 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 a *relation*.

3.2.1. An Example of Inheritance

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,[2] then values that are not present in A may be inherited from B.

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

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 GRAPHICAL-OBJECT schema 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.

3.2.2. Multiple Inheritance

KR supports multiple inheritance, i.e., the situation where a schema may inherit values from more than one direct ancestor. This can be accomplished in two separate 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 multiple values. 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.

Note that an application program should not rely on the order in which KR searches different relations. The particular order chosen is implementation-dependent.

3.3. Relations

Slots such as :IS-A which enable knowledge to be inherited from other parts of a network are called *relations*. Inheritance along a relation is typically defined to proceed depth-first and may include any number of steps (in other words, the search terminates if a value is found or if no other schema can be reached via the relation).

KR allows the user to define new relations as desired. This is achieved through the function **create-relation** (see section 6.3), which performs all the necessary bookkeeping.

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 schema to another. Imagine, for instance, that we defined :PART-OF to be a relation having :HAS-PARTS as its inverse. Adding schema A to the slot :PART-OF of schema B would automatically add B to the slot :HAS-PARTS of schema A, thereby creating a reverse link.

3.3.1. Relation Maintenance

KR automatically maintains all relations and inverse relations described above, and the application programmer does not have to worry about them. This is probably one of the most convenient features of the system.

Imagine, for instance, that the two schemata A and B are linked by a certain relation and inverse relation. This means that schema A has schema B as the value in one of its slots. If the program decides to delete schema B, then, it is essential that the link from A to B also disappear. Failure to do so would cause the reference in A to be dangling: it would be an error to try to follow the reference, since the schema being pointed to (i.e., B) would no longer exist.

KR keeps track of similar situations whenever they occur and corrects them instantly. The KR function that deletes schema B automatically follows the reverse pointers and makes sure that any reference to B disappears as well. In a similar manner, whenever the name of a schema is assigned as a value to a slot which happens to be a relation, KR automatically creates an inverse link. This ensures that the state of the knowledge representation system is completely consistent at any point in time, independent of the particular sequence of operations.

4. Object-Oriented Programming

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

An object in KR is simply a schema. As in most object-oriented programming systems, objects consist of data (represented by values in slots) and methods (represented by procedural attachments, again stored as values in slots). Procedural attachments are invoked by "sending a message" to an object; this means that a method by the appropriate name is sought and executed. Different types of objects very often provide different methods by the same name; this ensures that the same message may be sent to different objects, which respond by performing different actions.

Both the data and the methods associated with an object can be either stored within the object or inherited. The usual inheritance rules are followed, including of course multiple inheritance. This allows the behavior of objects to be built up from that of other objects; it is possible, in particular, to create complex graphs of method inheritance. The object-oriented component of KR allows some combination of methods, since a method is allowed to invoke the corresponding method from a parent and to explicitly refer to the object which is handling the message. Method combination, however, is not as fully developed as in full-fledged object-oriented programming systems such as CLOS [Bobrow et al. 89].

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 in typical object-oriented systems, a prototype simply provides 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 determined by the prototype. The same is true for methods, which are simply represented as slots.

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 provides a uniform mechanism for handling object-dependent initialization tasks.

If the prototype provides a constraint for a certain slot, and the slot is not explicitly defined in the instance, the formula which implements the constraint is copied down and installed in the instance itself. This provides a convenient mechanism through which a prototype may determine some of 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. Unlike other frame-based systems, constraint maintenance is an integral part of KR and is tightly integrated with the basic knowledge representation.

5.1. Value and Action 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 is *value propagation* and ensures that the network is constantly kept in a consistent state. The second mechanism is *action propagation*, allowing an application program to cause certain actions to be triggered when parts of a network are modified.

The constraint system ensures that whenever a value in a slot is changed, all slots whose values depend on it are immediately invalidated, although not necessarily re-evaluated. This is what we refer to as value propagation. The fundamental notion if that of *dependency* of a value on another. This strategy 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 the correct values are recomputed when actually needed, thus giving the same results as eager re-evaluation. Value dependencies are embodied in formulas. This first mechanism of constraint maintenance is implemented by the KR system itself and guarantees that value dependencies are never violated.

In addition to this, a second mechanism is provided which allows an application program to perform special actions when a value is modified. We refer to this as action propagation. This second mechanism, which is totally controlled by the application program, is quite independent from the first. Action propagation is implemented through the concept of *demon*. A demon is an application-defined procedural attachment to a KR slot. Whenever a value of a slot in a schema is modified (either directly or as the result of value propagation), KR checks whether a demon is defined for that particular schema and slot. If so, the demon is invoked. This allows application programs to attach a certain behavior to their schemata and be notified every time a change occurs.

The following example illustrates the relationship between value propagation and demon invocation. The example shows the complete sequence of events when the basic value-setting function, **s-value**, is called to set slot A of schema B to the new value C:

1. If the value in slot A is identical to value C, nothing happens.
2. Otherwise, if a demon is defined for schema B, the demon is invoked. The demon should be a function of three arguments: a schema, a slot, and a value. The demon is called with schema B in its *old* state, which means that slot A still contains its old value. The third argument is the new value, i.e., C.
3. The change is recursively propagated. All slots whose value is a formula that depends on slot A 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 A is finally changed to C.

5.2. Demons

A demon is associated with a schema either directly or through inheritance. The demon is stored in a slot named :UPDATE-DEMON. This allows the application to gain fine control over the response to a value change.

To allow the application program even finer control over demon behavior, another slot is used by KR to determine which slots should actually trigger a demon when they are modified. This other slot is named :UPDATE-SLOTS and should contain a list of the slots for which demons are to be invoked. Whenever a slot whose name is contained in :UPDATE-SLOTS is modified, and a demon is defined for the schema, the demon is activated. If the slot is not among those in :UPDATE-SLOTS, or if no demon is defined for the schema, no demon is invoked.

A typical example of demon is a graphical demon, i.e., one which is attached to schemata that represent graphical entities. Since some of the schemata may be visible to the user (in a display window, for example), many application programs will want the display to change when the schemata are modified. Different types of demons can serve this need.

A simple-minded approach would be to define demons that simply erase the old image of a schema when one of the slots is modified, and immediately redraw the new image. This approach can be improved significantly. A second, and better, approach would be a demon which erases the old image and then marks the schema as "dirty", to indicate that its display image is not up to date. At some later point in time, the application may then redraw up-to-date images for all the "dirty" schemata. The advantage of this approach is that successive changes to a schema which is already marked dirty have no effect, and thus the total amount of erasing and redisplaying may be significantly reduced. Yet other approaches to this problem are possible, and they all rely on demons to notify the application program of changes in schemata.

5.3. Formulas

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

Formulas can be arbitrary Lisp expressions, and in general contain at least one reference to a particular KR value. The expression is used to recompute the value of the formula whenever a change in one of the values makes it necessary. A formula, therefore, contains two logically separate types of information:

1. A list of all the values on which it depends, i.e., a list of dependencies.
2. An expression which determines how to combine the values to compute the value of the formula itself.

As we mentioned earlier, 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 fairly low and most changes are local in nature. The availability of a local cache means that in most cases the formula is not recomputed at all, since the correct value is already available locally. Typical applications have a read-to-write ratio of around 100:1, which means that out of 100 accesses to a formula only 1 causes the formula to be recomputed.

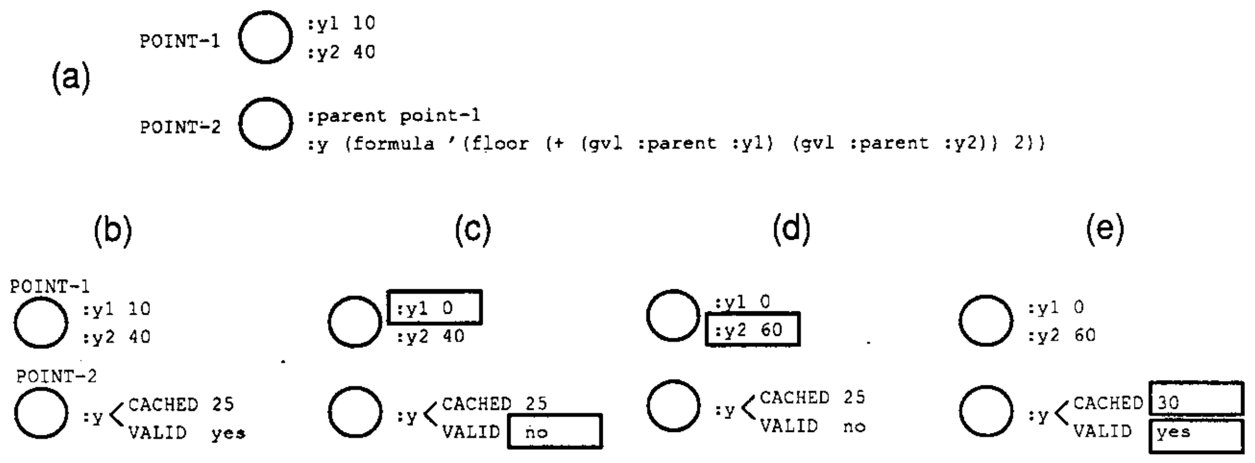

Figure 5-1: Successive changes in depended values

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, in particular, 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 has been evaluated and contains a valid cached value. Under these circumstances, any request for the value of slot :Y would simply return the cached value, without ever 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 take place, since the formula is already marked invalid.

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

5.3.1. Circular Dependencies

It is perfectly legal for constraints to involve circular chains of dependency. Slot A, for instance, might depend on slot B, which in turn depends on slot A; see section 8.1 for an example of a situation where this arises fairly naturally. Circular chains 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.

5.3.2. Dependency Paths

Typical formulas contain embedded references to other values and schemata. Many of these references span more than one link and 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 can happen, for instance, if one of the intermediate schemata is deleted. 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 paths, since the system handles the situation gracefully.

The typical application program does not have to be aware of the existence of dependency paths and the possibility that a path might be temporarily severed. A special function, **link-valid-p**, is available for those applications that need to be informed of this situation and take special actions.

5.4. Constraints and Multiple Values

Version 2.0 does not allow constraints on multiple values in a slot. The only value on which constraints are handled properly is the first one in a slot; **g-value** and the related functions work properly with such values.

The interaction between constraints and multiple values will be specified and implemented in future versions of KR. It will be possible, in particular, for a formula to depend on values in positions other than the first in a slot.

6. Functional Interface

This section contains a list of the more common functions and macros exported by the KR interface. It includes the functionality that most users are likely to need and covers knowledge representation, object-oriented programming, and constraint maintenance. Section 7 describes parts of the system that are much less commonly used. All functions and variables are defined and exported by the KR package.

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. Predicates

This group includes functions that test certain attributes of KR schemata and slots.

(SCHEMA-P *thing*) [Function]

This predicate returns T if *thing* is a valid KR schema, **nil** otherwise.

(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
```

(IS-A-P *schema other-schema*) [Function]

This predicate returns **T** if *schema* is related to *other-schema* via the :IS-A relation, either directly or through an inheritance chain. It returns **nil** otherwise.
Examples:

```
(is-a-p rectangle-1 box-object) ==> T
(is-a-p rectangle-1 graphical-object) ==> T
(is-a-p rectangle-1 rectangle-2) ==> NIL
```

(HAS-SLOT-P *schema slot*) [Function]

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

```
(has-slot-p rectangle-1 :is-a)  ==> non-nil value
(has-slot-p rectangle-1 :thickness) ==> NIL    ; not local
```

6.2. Schema Manipulation Functions

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

(CREATE-SCHEMA *schema-name* &rest *slot-definitions*) [Macro]

This macro creates and returns a new schema named *schema-name*. If *schema-name* is **nil**, an unnamed schema is created and returned. If *schema-name* is a symbol, a special variable by that name is created and bound to the new schema.

The *slot-definitions*, if present, are used to create initial slots and values for the schema. Each slot definition should be a list whose CAR is the name of a slot and whose CDR is a (possibly empty) list of values for that slot.

Note that if *schema-name* is the name of an existing schema, that schema is deleted first. This default behavior may be modified by using the keyword **:override** as part of the *slot-definitions*. This keyword requests that the existing schema 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.
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 graphical-object))
```

(DESTROY-SCHEMA *schema-name*) [Function]

Destroys the schema named by *schema-name*. Returns **T** if the schema was destroyed, **nil** if it did not exist. This function takes care of properly removing all constraint dependencies to and from the *schema-name*.

6.3. Slot Manipulation Functions

This group includes functions which create and delete slots in a schema. It also includes a convenient way to iterate a user-defined function over all the slots in a schema.

(CREATE-RELATION *name inherits-p* &rest *inverses*) [Macro]

Declares the slot *name* to be a relation. The new relation will have *inverses* (a 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)
```

(DESTROY-SLOT *schema slot*) [Function]

Destroys the *slot* from *schema*. Values previously stored in the slot, if any, are lost. All constraints to and from *schema* are modified accordingly.

(DOSLOTS *schema function*) [Function]

Iterates the *function* over all the slots of *schema*. The *function*, which should be a LISP function of two arguments, is applied in turn to each local slot of *schema*. The *function* is called with the *schema* itself as the first argument, and the name of the slot as the second argument. The *function* is called purely for side effects, and **doslots** simply returns **nil**.

```
(doslots rectangle-1 #'(lambda (schema slot)
   (format t "Slot ~S has value ~A~%"
     slot (g-value schema slot))))
;; prints out:
Slot :Y has value 20
Slot :X has value 10
Slot :IS-A has value BOX-OBJECT
```

6.4. Value Manipulation Functions

This group includes the most commonly used KR functions, i.e., the ones that retrieve or modify values in a slot. This section presents KR value manipulation functions that deal properly with constraints. A different set of primitive functions, which do not deal with constraints, is described in Section 7.

(G-VALUE *schema slot* &rest *other-slots*) [Macro]

This macro returns the first value in the *slot* from *schema*; if the slot is empty or not present, it returns **nil**. Inheritance may be used when looking for a value. This function handles constraints properly: If a formula is currently installed in the *slot*, the value is computed (if needed) and returned.
Examples:

```
(g-value rectangle-1 :is-a) ==> BOX-OBJECT
(g-value rectangle-1 :thickness) ==> 1      ; inherited
(g-value rectangle-1 :color) ==> :BLUE
(g-value rectangle-2 :y) ==> 35             ; computed formula
;; Change value in depended slot from 20 to 21
(incf (g-value rectangle-1 :y))
;; Now the constraint is propagated to RECTANGLE-2
(g-value rectangle-2 :y) ==> 36             ; recomputed
```

As shown by the expression `(incf (g-value rectangle-1 :y))` in the example above, a LISP **setf** form is defined for **g-value** and expands into **s-value**, the KR function which sets the value in a slot. This allows a variety of LISP constructs to be used in combination with **g-value**, such as the idiom

```
(incf (g-value schema slot))
```

which increments the value of a slot in the schema. Note that constraint propagation is fully enforced during this operation, just as it would be in the (equivalent) expression

```
(s-value schema slot (1+ (g-value schema slot)))
```

The macro **g-value** may be given any number of *other-slots*. This expands into repeated calls to **g-value**, where each slot is used to retrieve another schema. For instance, imagine we had defined a hierarchy of schemata to represent a family tree, and we were using the slot **:parent** to express the hierarchy. The following expressions, then, would be entirely equivalent and would both retrieve the value of the *:slot* from two levels up in the hierarchy:

```
(g-value grand-child :parent :parent :slot)
(g-value (g-value (g-value grand-child :parent) :parent)
   :slot)
```

(S-VALUE *schema slot value*) [Function]

This function is used to set a slot with a given value or formula. The *slot* in *schema* is set to contain the *value*. The most common case is the one where *value* is a regular LISP value and simply supersedes any previous value in the slot. If *value* is a formula, i.e. the result of a call to the function **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 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 default behavior is to keep the old formula in place, but to use the *value* 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, typically because some of the values on which it depends are modified. This default behavior may be explicitly overridden by setting the special variable ***allow-change-to-cached-value*** to **nil**. In this case, trying to set the *slot* with a *value* which is not a formula has no effect, and the old formula retains its original value.

s-value returns the new value of the *slot*.

(GET-VALUES *schema slot*) [Macro]

This macro returns a list of all the values in the *slot* of *schema*. If the *slot* is empty or not present, it returns **nil**. Inheritance may be used when looking for values. Note that this macro does not deal with constraints, i.e., it does not cause formulas to be evaluated.
Examples:

```
(get-values graphical-object :is-a-inv) ==> (BOX-OBJECT)
(get-values box-object :is-a-inv) ==>
         (RECTANGLE-2 RECTANGLE-1)
```

A **setf** form is defined for **get-values** and expands into a call to **set-values**.

(SET-VALUES *schema slot values*) [Function]

This function stores a list of values in the *slot* of *schema*. The entire list may subsequently be retrieved with **get-values**, or the first value may be retrieved with **g-value**. Note that **set-values** does not deal with constraints, i.e., the list of *values* cannot contain formulas.

(SET-VALUE-N *schema slot value position*) [Function]

This function is similar to S-VALUE, except that it sets a value other than the first one. The *position* is a 0-based number which indicates the number of the value that should be replaced.

Note that this function does not deal with constraints properly.

The *position* must be non-negative. If it is greater than the number of values currently present in the *slot*, the slot is padded in the middle with enough **nil** values to reach the appropriate position.

This function will be replaced in future versions of KR by **s-value-n**, which will deal with constraints properly.

(GV *schema slot* &rest *more-slots*) [Macro]

This macro is superficially similar to **g-value**, but it serves a different purpose and can only be used within formulas. The result of **gv** is the current value in the *slot* of *schema*, just like **g-value**. In addition, however, **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.

The first argument, *schema*, is the starting schema for the path; the remaining arguments are slots, which are accessed in turn until the end of the path is reached. The result of each access is a schema, which is then further accessed through the next slot name. For example, the following two expressions are equivalent:

```
(gv my-schema :parent :color)
(gv (gv my-schema :parent) :color)
```

Both expressions return the value of the *:color* of the schema which is contained in the *:parent* slot of *my-schema*. One can think of the slot *:parent* as providing the name of the place from which the next slot can be accessed.

Note that *schema* can be any schema, not necessarily the one on which the formula surrounding gv is installed. Specifying the reserved name **:self** for *schema* ensures that the path starts from the schema on which the formula is installed.
Examples:

```
(formula '(gv rectangle-1 :y))
(formula '(+ (gv :self :x) 15))
(formula '(equal (gv :self :parent :parent :color)
                 (gv :self :color)))
```

(GVL *slot* &rest *more-slots*) [Macro]

This is a useful shorthand notation for (**gv :self** *slot more-slots*). Like **gv**, it may only be used in formulas. For example, the expression (**gvl :color**) returns the current value of the :COLOR slot in the schema which contains the surrounding formula, and is equivalent to the expression (**gv :self :color**).

(DOVALUES (*variable schema slot* &optional *all-p*) &rest *body*) [Macro]

DOVALUES executes the *body* with the *variable* bound in turn to each value in the *slot* of *schema*. The *body* is executed purely for side effects, and DOVALUES returns **nil**. The *body* should not alter the contents in the *slot*, since this may cause unpredictable results. Example:

```
(set-values rectangle-1 :vertices '(a b d f))

(dovalues (v rectangle-1 :vertices)
   (format t "rectangle-1 has vertex ~S~%" v))
;; prints out:
rectangle-1 has vertex A
rectangle-1 has vertex B
rectangle-1 has vertex D
rectangle-1 has vertex F
```

If *all-p* is non-nil, the *body* is executed for all the values of *slot* that can be inherited by *schema*, including all levels of possible inheritance. The default behavior is to stop as soon as the *slot* is found locally or is inherited.

6.5. Constraint Manipulation Functions

These functions are concerned with the constraint maintenance part of KR. The fundamental notion here is that of a formula, which expresses how the value in a slot depends on other values. A formula expresses a dynamic link and is (logically) recomputed every time any of the depended values is modified.

ALLOW-CHANGE-TO-CACHED-VALUE [Special Variable]

This variable controls the system's behavior when functions like **s-value** are used to replace a formula with an ordinary value.

The default value of this variable, **T**, indicates that ordinary values are allowed to replace formulas. More specifically, an ordinary value is installed as the cached value of the formula and replaces whatever value the formula previously had. The formula itself, however, is still in place, and therefore may recompute a new value if some of the depended values change. See **destroy-constraint** for a function which physically eliminates a formula from a slot.

A value of **nil** indicates that trying to set a slot that contains a formula with an ordinary value has no effect. The operation is simply ignored, and the formula retains its previously cached value.

(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. The objects may be freely modified inside the *body* without interference from the demons.

(DESTROY-CONSTRAINT *schema slot*) [Function]

If the *slot* of the *schema* 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. If the *slot* contains an ordinary value, this function has no effect.

(FORMULA *form* &optional *initial-value*) [Function]

Given a *form*, this function returns a formula[3]. The *form* typically consists of Lisp expressions and **gv** or **gvl** references.
Examples:

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

The first example shows a formula which causes the slot on which it is installed to have the same value as slot :X of the schema contained in slot :ABOVE-GADGET of the current schema. 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 schema contained in slot :OTHER-GADGET of the current schema.

Note that *form* can also be an existing formula, rather than a Lisp expression. In this case, the new formula is linked via :IS-A to the existing formula, and inherits the expression from it. No local state is shared by the two formulas. An illustration of this case is given by the second call in the following example, which returns a new formula that inherits its expression from the first one:

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

If *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 will never be used under normal circumstances. The initial value is only used if the formula is part of a circular dependency or if one of the dependency paths is invalid. This means that only applications which contain circular chains of constraints need to specify an *initial-value*.

6.6. Object-Programming Functions

This section contains a few functions that support objected-oriented programming within KR. Note that KR does not support any predefined notion of class hierarchy, but of course arbitrary "class hierarchies" may be built on top of the system's powerful inheritance mechanism.

(DEFINE-METHOD *name schema arg-list* &body *body*) [Macro]

This macro defines a method named *name* for *schema*. While *schema* can be any schema, and in

[3]Formulas are internally represented by a schema.

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 *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 *name*, the string "-METHOD-", and the name of the *schema*. Example:

```
(define-method :print box-object (schema)
  (format t "A rectangle at (~D,~D).~%"
    (g-value schema :x) (g-value schema :y)))
```

After this, the :PRINT message 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.

(CREATE-INSTANCE *schema prototype* &rest *slot-definitions*) [Macro]

This macro is similar to **create-schema** but is more convenient for creating objects that are instances of a prototype. Besides acting like **create-schema**, **create-instance** also performs three operations that are particularly suited for instances.

First of all, it links the newly created *schema* to the *prototype* via the :IS-A link. Secondly, 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 slots that are not explicitly defined by the instances. Thirdly, if the *prototype* or the *schema* itself defines the :INITIALIZE method, **create-instance** sends the newly created *schema* 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:

```
(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:
RECTANGLE-4 initialized
```

(KR-SEND *schema slot* &rest *arguments*) [Macro]

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

If the function, i.e., the value of the expression `(g-value schema 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**.

(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 parent. It should never be used outside object methods. If a parent of the current object, i.e, the one which supplied the method currently being executed, also defines a method by the same name, the parent's method is invoked with *arguments* as the list of arguments. For example,

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

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

First of all, **kr-send** invokes the method defined locally by schema A. The method itself contains a call to **call-prototype-method**, and thus the hierarchy is searched for a parent of schema A which also defines the :NOTIFY method. If one exists, that method is invoked.

Note that a method is free to supply a parent 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 schema (+ level 1))**. It is customary, however, to invoke **call-prototype-method** with exactly the same parameters as the original call. Finally, note that the name of the original schema and the message name are not specified in **call-prototype-method**, and KR automatically provides the right values.

6.7. Printing and Debugging Functions

(PS *schema* &optional *control-schema*) [Function]

This function prints the contents of *schema*, and allows fine control over what to print and how. The default behavior is to print out all slots and all values in *schema*; this happens when the *control-schema* is **nil** or not specified. 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 values that are printed from each slot, thus preventing annoyingly long lists of values.

If a control schema is specified, it should contain (or inherit) four special slots. These slots determine what **ps** does, as follows:

- :SORTED-SLOTS contains a list of names of slots that should be printed at the beginning, in the correct order.

- :IGNORED-SLOTS contains a list of names of slots that should not be printed.
- :GLOBAL-LIMIT-VALUES contains an integer, the maximum number of values that will be printed for each slot. If a slot contains more than that many values, ellipsis are printed after the given number of values.
- :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.

The *control-schema* should be one of three things:

1. not specified or **nil**, which means that *schema* is printed in its entirety.
2. T, which means that the *schema* itself is used as the control schema. In most cases, the four slots are inherited from some of the parents of the *schema.*
3. a schema, which is used directly as the control schema.
4. the keyword :DEFAULT, which indicates that the default print control schema should be used. The name of the default print control schema is PRINT-SCHEMA-CONTROL; this default schema limits the number of values that are printed by **ps** to a maximum of five per slot.

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 no *control-schema* prints out the whole contents of the schema:

```
(ps colored-thing)
;; prints out:
{COLORED-THING
  :IS-A-INV =  s67 s66 s65 s64 s63 s62 s61 s60 s59 s58
     s57 s56 s55 s54 s53 s52 s51 s50 s49 s48
  :INTERNAL =  "Help information"
  :Y =  20
  :X =  10
  :COLOR =  :BLUE
  :WIDTH =  12.5
  :IS-A =  TOP-OBJECT
  }
```

Using the system-supplied default control schema reduces the clutter in the :IS-A-INV slot:

```
(ps colored-thing :default)
{COLORED-THING
  :WIDTH =  12.5
  :IS-A-INV =  S67 S66 S65 S64 S63 ...
  :INTERNAL =  "Help information"
  :Y =  20
  :X =  10
  :COLOR =  :BLUE
  :IS-A =  TOP-OBJECT
  }
```

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

```
(s-value top-object :global-limit-values 7)
(set-values top-object :sorted-slots
  '(:is-a :color :x :y))

(ps colored-thing t)
;; prints out:
{COLORED-THING
  :IS-A =  TOP-OBJECT
  :COLOR =  :BLUE
  :X =  10
  :Y =  20
  :IS-A-INV =  S67 S66 S65 S64 S63 S62 S61 ...
  }
```

Note that **ps** prints slots whose value is a formula in a special way. Besides the name of the formula schema, 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 (formula '(+ (gvl :left) 25))))
(g-value a :right)  ==> 35

(ps a)
;; prints out:
{A
  :RIGHT =  F88 (35 . T)
  :LEFT =  10
  }

(s-value a :left 50)
(ps a)
;; prints out:
{A
  :RIGHT =  F88 (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 F88 is marked invalid.

(S *number*) [Function]

This function is useful in some debugging situations. Whenever the function **ps** prints an unnamed schema, it shows it as an internal number. Unnamed schemata often appear as values in some other schema's slots. When this happens, and one needs to refer explicitly to an unnamed schema, the function **s** can be used to go from the internal number to the actual schema. For example, suppose that a call to **ps** causes the following to be printed:

```
{COLOR
  :IS-A-INV =  s101 s100 s99 s98 s97 ...
  }
```

One can then use the form **(s 99)** to refer to the third schema in the :IS-A-INV slot of schema COLOR, as shown below:

```
(ps (s 99))
{s99
  :IS-A =  COLOR
  }
```

Note that an alternative way to refer to an unnamed schema always exists, and thus the function s is not strictly indispensable. In the example above, for instance, we could have used the form:
`(ps (third (get-values color :is-a-inv)))`

(METHOD-TRACE *class message-name*) [Macro]

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

7. Additional Functions

This section describes a set of KR functions and macros that are used much more seldom that the ones we have seen so far. Some of these functions are obsolete, while others deal with aspects of the system that only advanced application programs need be concerned with.

7.1. Value Manipulation Functions

Functions in this group do not deal with constraints. They may be useful to applications that need to be aware of the distinction between ordinary values and formulas. The group also includes functions that deal with multiple values.

(GET-VALUE *schema slot*) [Macro]

Returns the first value in the *slot* from *schema*. If the slot is empty or not present, it returns **nil**. Inheritance may be used when looking for a value. Note that **get-value** does not deal with constraints at all; in particular, given a slot that contains a formula, **get-value** returns the formula itself, rather than its value. Therefore, use of **get-value** is limited to applications that manipulate formulas explicitly.

(GET-LOCAL-VALUES *schema slot*) [Macro]

Similar to GET-VALUES, but only local slots are examined and inheritance is never used. Examples:

```
(get-values rectangle-1 :thickness) ==> (1)
(get-local-values rectangle-1 :thickness) ==> NIL ; not local
```

Note that this macro does not deal with constraints, i.e., it does not cause formulas to be evaluated.

(APPEND-VALUE *schema slot value*) [Function]

This function adds a value to the end of the list of values in the *slot* of *schema*. Note that this function does not deal with constraints properly, i.e., *value* cannot be a formula.

(DELETE-VALUE-N *schema slot position*) [Function]

This function deletes the *position*-th value from the *slot* of the *schema*. *position* is a non-negative integer, with 0 indicating the first value in the *slot*. Note that this function does not deal with constraints properly.

7.2. Constraint Maintenance Functions

(LINK-VALID-P *schema slot* &rest *more-slots*) [Macro]

This macro can be used within formulas to check whether a reference through **gv** would succeed. If so, the macro returns **T**; otherwise, it returns **nil**. This macro allows a formula to ensure that all its dependency paths are intact before starting a recomputation.

This macro should only be used when it is important that the formula be able to detect whether a dependency path was interrupted and to handle this case in a special way. Under normal circumstances this test is not necessary, since the cached value of the formula is used by default when attempting to reference an invalid dependency path.

(FORMULA-P *thing*) [Macro]

A predicate that returns **T** if the *thing* (any Lisp object) is a valid formula, **nil** otherwise.

(CHANGE-FORMULA *schema slot expression*) [Function]

If the *slot* in *schema* contains a formula, the formula is modified to contain the *expression* as its new function. This function 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.

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.

(MARK-AS-CHANGED *schema slot*) [Function]

This function may be used to trigger the constraint propagation mechanism in KR for a *schema* whose *slot* has been modified by means other than **s-value**. Some applications may need to use destructive operations on values in a slot and then notify the system that certain values were changed.

Most users will probably never need to use **mark-as-changed** in their programs.

(G-CACHED-VALUE *schema slot*) [Function]

This function is similar to **g-value** 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 functionality, which in some cases may return values that are out of date. This function should be used with care.

8. 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. We first construct a schema with 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 some simple object-oriented programming techniques, and illustrates many of the dynamic capabilities on 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.

8.1. 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 (formula '(+ (* (gvl :celsius) 9/5) 32)
                        32))
  (:celsius (formula '(* (- (gvl :fahrenheit) 32) 5/9)
                        0)))
```

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 customary 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 at 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 change propagation when necessary.

Consider, for instance, setting the value of the :CELSIUS slot:

```
(s-value degrees :celsius 20)
(g-value degrees :celsius)         ==> 20
(g-value 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)
(g-value degrees :celsius)         ==> 100
(g-value degrees :fahrenheit)      ==> 212
```

8.2. 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
    (formula '(+ (* (gvl :celsius) 9/5) 32) 32)))


(define-method :print temperature-device (schema)
  (format t "Current temperature: ~,1F C (~,1F F)~%"
          (g-value schema :celsius)
          (g-value schema :fahrenheit)))
```

We now create two schemata 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 (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 a child schema, 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 (formula '(min (gvl :min)
                       (gvl :location :celsius))
                100))
  (:max (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~%"
          (g-value schema :min)
          (g-value 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:

```
(g-value 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 (g-value schema :celsius))
  (s-value schema :max (g-value 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
```

The examples above show a simple way to achieve the desired behavior. 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. Implementing the same example with one of those styles is a worthwhile exercise, and the reader is invited to spend some time trying different alternatives.

9. Summary

KR is a knowledge representation system which provides excellent performance and a combination of three powerful paradigms: frame-based knowledge representation, object-oriented programming, 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 knowledge representation component of KR offers multiple values, 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 variety of application programs.

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

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 in KR 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 detailed knowledge of its internal details. The same access functions, in particular, work on both regular values and values which are constrained by formulas.

In spite of its power, KR is a very small and simple system. 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] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon.
Common Lisp Object System Specification.
LISP and Symbolic Computation 1(3/4):245-394, January, 1989.

[Brachman 79] Brachman, R.J.
On the epistemological status of semantic networks.
Associative Networks: Representation and Use of Knowledge by Computers.
Academic Press, New York, 1979, pages 3-50.

[Brachman and Levesque 85]
Brachman, R.J. and Levesque, H.J.
Readings in knowledge representation.
Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1985.

[Giuse 87] Dario Giuse.
KR: an efficient knowledge representation system.
Technical Report CMU-RI-TR-87-23, The Robotics Institute, Carnegie-Mellon University, 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] Dario Giuse.
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 89a] Dario Giuse.
Efficient Knowledge Representation Systems.
1989.
Submitted for publication.

[Giuse 89b] Dario Giuse.
Frame Systems as Object-Oriented Systems.
1989.
Submitted for publication.

[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.

[Myers 88] Brad A. Myers.
The Garnet User Interface Development Environment: A Proposal.
Technical Report CMU-CS-88-153, Carnegie-Mellon University, September, 1988.

[Myers 89] Brad A. Myers, Brad Vander Zanden, and Roger B. Dannenberg.
Creating Graphical Objects by Demonstration.
1989.
Submitted for Publication.

[Steele 84] Steele, G.L.
Common LISP - The Language.
Digital Press, Burlington, MA, 1984.

[Vander Zanden et al. 89]
Brad Vander Zanden, Brad A. Myers, Dario Giuse, and John Kolojejchick.
An Incremental Automatic Redisplay Algorithm for Graphic Object Systems.
1989.
Submitted for Publication.

[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