Using the

Parka Parallel Knowledge Representation System

(Version 3.2)[1]

Technical Report CS-TR-3485 (UMIACS TR-95-68, ISR 95-56)

Brian Kettler, William Andersen, James Hendler, and Sean Luke

Department of Computer Science,
Institute for Systems Research, and
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742
Email: parka-users-request@cs.umd.edu, hendler@cs.umd.edu

May 1995

[1]This research was supported in part by grants from NSF (IRI-9306580), ONR (N00014-J-91-1451), AFOSR (F49620-93-1-0065), the ARPA/Rome Laboratory Planning Initiative (F30602-93-C-0039), and the ARPA I3 Initiative (N00014-94-10907) and by ARI (MDA-903-92-R-0035, subcontract through Microelectronics and Design, Inc.) Dr. Hendler is also affiliated with the UM Institute for Systems Research (NSF Grant NSF EEC 94-02384) and the UM Institute for Advanced Computer Studies.

Abstract

Parka is a symbolic, semantic network knowledge representation system that takes advantage of the massive parallelism of supercomputers such as the Connection Machine. The Parka language has many of the features of traditional semantic net/frame-based knowledge representation languages but also supports several kinds of rapid parallel inference mechanisms that scale to large knowledge-bases of hundreds of thousands of frames or more. Parka is intended for general-purpose use and has been used thus far to support A.I. systems for case-based reasoning and data mining.

This document is a user manual for the current version of Parka, version 3.2. It describes the Parka language and presents some examples of knowledge representation using Parka. Details about the parallel algorithms, implementation, and empirical results are presented elsewhere.

Contents

List of Figures

Chapter 1

Overview

1.1 Introduction

This document is the user manual for the Parka Knowledge Representation System, version 3.2 ("Parka 3"). The purpose of this document is to describe the features of the Parka system and to provide some examples of how knowledge can be represented using the Parka language. Details on the research issues addressed by Parka and the Parallel Server operation can be found in other documents described below.

Parka has much in common with typical frame-based knowledge representation systems. What distinguishes Parka is its ability to perform certain types of very useful inferences faster than any existing system we know of. This speed is accomplished through the use of massively parallel hardware, such as the Connection Machine (CM-2 and CM-5) supercomputer made by Thinking Machines Corporation. If a parallel machine is not available, Parka can still be run in serial mode. If a CM is available as a parallel server, certain inferences will be done more quickly in parallel.

Parka has been developed by the Parallel Understanding Systems A.I. Research Group, under the direction of Prof. James Hendler, in the Department of Computer Science at the University of Maryland at College Park. The original version of Parka was designed and implemented by Matt Evett and Lee Spector. Bill Andersen designed and implemented Parka 3. Some aspects of the Parka 3 language design were suggested by Brian Kettler. Sean Luke implemented a graphical browser and graphical query window for Parka 3. Merwyn Taylor assisted in the testing of Parka 3.

To obtain and install Parka, see the instructions available on our Worldwide Web page, described in Section 1.1.2, or send mail to `parka-users-request@cs.umd.edu`.

1.1.1 Guide to This Document

This chapter presents a brief overview of Parka. Section 1.2.3 describes the major changes from the previous version. In Chapter 2 the concepts and features of the Parka language are described. Chapter 3 describes the facilities of the Parka system for end-user interaction and knowledge-base

(KB) maintenance.[1] The syntax of Parka commands is described in Appendix A.

1.1.2 Other Sources of Information on Parka

Research Papers

Details and results for the fast inferencing algorithms of Parka can be found in [2, 1].

The *Parka Internals Manual* describes the internals of the Parka implementation in detail including the serial client/simulator and the parallel server.[2]

The original Parka language and its use for knowledge representation is described in [5, 4]. These documents are largely superseded by this document.

Several Parka applications are being built including a case-based planning system, CaPER [3] and a hybrid knowledge-base/database system.

Mailing List

A Parka mailing list exists (`parka-users@cs.umd.edu`). To be added to this list, send email to `parka-users-request@cs.umd.edu`.

Worldwide Web

A Parka Worldwide Web (WWW) page has been set up at URL
`http://www.cs.umd.edu/projects/plus/Parka/parka.html` .
We expect this to be the primary source for the latest information available on Parka. Information on obtaining and installing Parka can be found here. Updated versions of this document will also be posted there.

1.1.3 Acknowledgements

Thanks to Matt Evett and Lee Spector for their previous work on Parka and input on the current version. Thanks to the Scott Andrews, Dave Rager, Kate Sanders, Merwyn Taylor, and Clare Voss for reviewing various drafts of this manual and to Kutluhan Erol for his LaTeX advice.

This research was supported in part by grants from NSF(IRI-9306580), ONR (N00014-J-91-1451), AFOSR (F49620-93-1-0065), the ARPA/Rome Laboratory Planning Initiative (F30602-93-C-0039), and the ARPA I3 Initiative (N00014-94-10907) and by ARI (MDA-903-92-R-0035, subcontract through Microelectronics and Design, Inc.) Dr. Hendler is also affiliated with the UM Institute for Systems Research (NSF Grant NSF EEC 94-02384).

[1] Parka KBs are also called "nets".

[2] This document is in progress.

1.2 Overview of Parka

Parka is a frame-based knowledge representation system that is intended to provide extremely fast inferences – especially inheritance inferences. It is also designed to accommodate very large knowledge bases (KBs) on the order of millions of frames. Even on very large KBs, Parka can perform "top-down" inheritance inferences (e.g., "What is the color of every truck?", or "Which trucks are yellow?") in a few hundredths of a second. Parka's speed is possible through the use of massively parallel hardware. The system has been implemented to date on the CM-2 and CM-5 parallel supercomputers.

Parka can perform multiple top-down inferences almost as fast as a single inference.[3] This is possible through the use of a parallel processing technique called "pipelining", in which multiple operations in various stages of completion can be active simultaneously in disjoint subsets of the CM's processors. [4]

Certain tradeoffs have been made in the design of Parka in order to facilitate fast query processing. One such tradeoff is that more work may be done at KB update time so that less work needs to be done at query time.

1.2.1 Parka Architecture

The main components of the Parka system are the front-end system and the back-end system. The front-end system runs on a serial machine. This system interacts with an end-user, or, more commonly, a user application. This user application may be running on a machine remote from that of the front-end machine.[5]

The Parka back-end system runs on a parallel machine. This system typically communicates with the serial front-end system via TCP/IP. Thus the front-end serial machine may reside at a location remote from the parallel system.

1.2.2 Implementation Platforms

The back-end (parallel) code of Parka is currently implemented on the CM-2 (Parka version 2) and the CM-5 (Parka version 3) supercomputers made by Thinking Machines Corporation (TMC). The implementation language is *Lisp (version 4.1 by TMC).

The front-end (serial) code of Parka is currently implemented in Common Lisp and has been tested to date on Macintosh Common Lisp (version 2 by Apple Computer and Digitool, Inc.).

[3] A naive implementation (simply performing the multiple inferences in series) would require $O(dm)$ time for m inferences where d is the depth of the semantic net KB. Through pipelining, Parka can perform these inferences in $O(d+m)$ time. A serial system, in contrast, could perform these inferences in $O(nm)$ time, where n is the size of the KB.

[4] Due to its greater variety of inference mechanisms, the pipelining capability in Parka 3 may differ from that in previous versions.

[5] The mechanisms for interaction between the front-end and a remote user application are still being developed. It is anticipated that the front-end will play the role of a remote knowledge server (perhaps processing queries in the KQML/KRSL formats) with transactions passing over the Internet. Details about the client-server interface will be described in a subsequent version of this document.

1.2.3 Major Changes from Previous Versions

Parka 3 is intended to be a “shrink wrapped” product: users can write AI systems to exploit Parka's speed without having to know anything about programming parallel machines. Parka 3 offers a much richer set of representation facilities than previous versions. The purpose of the first two systems as research projects was to create and analyze novel parallel algorithms for performing fast top-down inheritance inferences. Parka 3 includes these algorithms as well as other representational features found in well known knowledge representation systems such as KL-ONE and KRYPTON. In the spirit of the original versions of Parka, expressiveness is sacrificed for the sake of speed. Thus features that would slow down any of the core inference algorithms have not been included in the language.

The other major change that Parka 3 represents is the introduction of a loosely coupled client/server architecture, which allows client AI systems running on remote machines to have access to a Parka server running on a Connection Machine. This decoupling also allows Parka clients to be implemented in a number of different languages. The initial offering will include only a Lisp-based client, however.

For a detailed list of changes see Appendix C.

Chapter 2

The Parka Language

2.1 Conceptualizing Parka

<table>
<tr><td>Knowledge Level</td><td colspan="2">Entities</td><td colspan="2">Relations</td></tr>
<tr><td>Ontological Level</td><td>Categories</td><td>Instances</td><td>Predicates</td><td>Assertions</td></tr>
<tr><td>Frame Level
(semantic network)</td><td>Category Frame
(node)</td><td>Instance Frame
(node)</td><td>Predicate Frame
(node)</td><td>Frame Slots
(arcs)</td></tr>
<tr><td>Implementation Level</td><td colspan="4">low-level data structures</td></tr>
</table>

Figure 2.1: Conceptual Levels of Parka

Knowledge representation in Parka can be conceptualized on several levels, as shown in Figure 2.1. Though the boundaries between these levels may not be sharply delineated, this framework provides a way to understand the concepts of Parka.

Figure 2.2 shows part of a KB, drawn as a semantic network, that will be used to illustrate these concepts. The series of Parka commands used to create this KB can be found in appendix D.1.

At the **knowledge level**, things in the world and their properties can be viewed in terms of

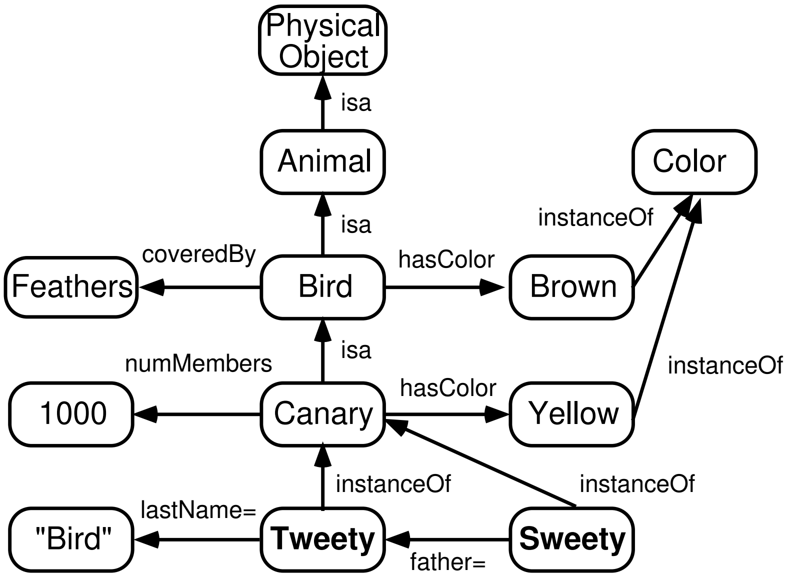

Figure 2.2: A Sample KB

entities and **relations**, respectively. The knowledge level is independent of a particular knowledge representation system such as Parka.

At the **ontological level** in Parka, entities and relations are represented using **categories**, **individuals**, and **predicates**. Categories represent sets of entities: e.g., the category Bird represents the set of all birds. Categories may have members (instances). Individuals represent a particular object: e.g., the individual Tweety represents a particular bird, Tweety. Every individual is an instance (member) of at least one category. Individuals are also termed **instances** if they are not categories. Categories and instances are described in detail in Section 2.2.

Predicates represent relations among entities. A binary predicate represents a binary relation between two entities. All predicates in Parka are binary predicates, i.e. taking two arguments. The arguments for a binary predicate can be two categories or a category and an instance. For example, the predicate instanceOf represents the set-membership relation and relates an instance to a category it belongs to. A predicate with its arguments represents a particular relation and is termed an assertion. For example, the assertion *(instanceOf Tweety Bird)* represents the fact that Tweety is a member of the set of all Birds, where Tweety is an instance representing Tweety and Bird is a category representing the set of all Birds. Predicates and assertions are described in detail in Section 2.3.

At the **frame level** in Parka, the main concepts are **frames**, a high-level data structure used by Parka, and **slots**, components of frames. Frames are used to specify categories, instances, and predicates to Parka. More specifically, category frames, instance frames, and predicate frame specify categories, instances, and predicates, respectively. For example, the frames *#!Bird*, *#!Tweety*, and

#!hasColor specify the category Bird, instance Tweety, and predicate hasColor.

Frame names have the prefix "#!". By convention this prefix is followed by an uppercase letter for the names of category frames and instance frames and a lowercase letter for the names of predicate frames. Parka does distinguish case for frame names: e.g, *#!Foo* names a different frame than *#!foo*.

A frame can have one or more slots used to specify assertions about a frame. A slot s of frame f consists of a predicate frame p and a value v, where v is a frame or an immediate value. For example, the frame *#!Canary* might have a value *#!Yellow* for its *#!hasColor* slot. Slots correspond to assertions. For example, the assertion that a canary has color yellow by default – *(hasColor Canary Yellow)* – can be specified by the frame *#!Canary* having the value *#!Yellow* for its *#!hasColor* slot. Making this assertion is equivalent to specifying a value for this slot (termed "filling the slot").

Frames and slots can also be viewed as nodes and arcs (links) in a semantic net, respectively. See Figure 2.2 for an example of this. Frames and slots are described in detail in Section 2.4.

At the **implementation level** in Parka, frames and slots are implemented using lower-level data structures. The details of this implementation can be found in the *Parka Internals Manual.*

2.2 Categories and Instances

Categories and instances are used to represent entities in the real world. Categories represent sets and instances represent individuals.[1] Because categories (instances) are specified using category frames (instance frames), the term "category" and "category frame" will be often used interchangeably in this discussion, as will "instance" and "instance frame".

For example, the set of all canaries is represented by category Canary, which is specified by the category frame *#!Canary*. A particular canary, Tweety, represented by the instance Tweety, can be specified by the instance frame *#!Tweety*.

Categories can have members, which are termed "instances" of the category. For example, the category Canary may have Tweety as one of its instances. In other words, Tweety is an element of the set of all canaries. Category membership is indicated by the "instance of" relation (see Section 2.3.3). Certain properties (see Section 2.3.2) may be specified as common to all members of a category. These properties are inherited by all members of a category and all member's of the category's subcategories (see Section 2.6).

Category frames can be instances of categories too. For example, every category is by definition an instance of the category frame *#!Category*, the set of all categories. Thus, category frame *#!Bird* will be an instance of category frame *#!Category*. Because it represents a set, frame *#!Bird* is still termed a "category frame", rather than an "instance frame". A frame can be an instance of multiple categories. Instance frames are also referred to as **Individuals**, because they represent individuals in the real world and every instance frame is an (by transitivity) instance of the category Individual, represented by the category frame *#!Individual*.[2] A category can have subcategories and supercategories, corresponding to subsets and supersets, respectively, of the set the category represents. For example, category *#!Bird* might have supercategories *#!Mammal* and *#!FlyingThing* and subcategories *#!Canary*, *#!Robin*, etc. Category inclusion is indicated by the "is a" relation (see Section 2.3.3).

An instance of a category is also an instance (member) of all of that category's supercategories (supersets). Thus if category *#!Canary* is an instance of category *#!Bird*, *#!Tweety*, an instance of *#!Canary*, is also an instance of *#!Bird* (though this need not be represented explicitly – see Section 2.6).

Frames that are categories must be designated as such when they are defined. A frame designated as a category may have instances. Instance frames cannot have instances of their own.

Categories and instances are defined using the Parka **new-category** and **new-instance** commands, respectively. Certain categories are "special frames" which have particular significance to Parka. These are listed in Appendix B.

2.2.1 Prototype Instances

To facilitate querying about a category, a prototype mechanism has been added to Parka. Some instance frames are prototypes. In fact, Parka automatically creates an special instance of each

[1] A category in Parka represents the extensional definition of a concept, i.e. the set of all instances of the concept.

[2] More specifically, if x is a individual, the following assertion (*#!everyInstanceOf s #!Individual*) holds automatically.

category called a "prototype" and fills the category's *#!prototype* slot with this instance frame. Prototype instances are normally not accessed directly by the user. The user cannot make assertions about prototypes. They exist to provide a mechanism for querying about a category's instance properties: those properties common to all instances of the category (see Section 2.3.2).

Prototype instance frames created by Parka have the prefix "#!%P-". For example, when the category *#!Canary* is defined by a user, the prototype instance frame *#!%P-Canary* is created by Parka and is made the value of *#!Canary*'s *#!prototype* slot.

2.3 Predicates and Assertions

Predicates and assertions are used to represent relationships among entities in the real world. Predicates define a relation and assertions specify occurrences of the relation. For example, the predicate hasColor might define the "has color" relation that relates things to colors. The assertion *(hasColor Canary Yellow)* would be used to specify that canaries are (by default) yellow. Parka currently supports only binary predicates: a predicate that takes two arguments. An assertion for a binary predicate consists of the predicate plus two arguments, each of which is a category, instance, or predicate.

Predicates are specified to Parka by predicate frames. For example, the predicate hasColor is specified by the frame *#!hasColor*. Thus the terms "predicate" and "predicate frame" will often be used interchangeably in this discussion. Predicate frames are instance frames: i.e., a predicate frame is an instance of some category frame such as *#!InheritablePredicate*. Predicate frames may have slots (like any other frame) and can inherit slot values from their category frames.

Predicates are defined using the Parka **new-pred** command. Certain predefined predicates are "special frames" which have particular significance to Parka. These are described in appendix B.

Assertions are specified to Parka using the Parka **assert!** command. An assertion *(p f v)* consists of a category/instance frame f, a predicate frame p, and a value v (see Section 2.4.1). Unlike categories, slots, and predicates, assertions are not represented internally in Parka as frames. [3]

2.3.1 Domain and Range of Predicates

All (binary) predicates have a domain and range for the relation they represent. The **domain** is a category to which the first argument of the predicate must be an instance of. The **range** is a category to which the second argument of the predicate must be a subcategory or an instance of (see Section 2.4.1).

For example, the predicate *#!hasColor* might have a domain of *#!PhysicalObject* and a range of *#!Color*. This means that the "has color" relation maps *instances* of *#!PhysicalObject* to *instances* of *#!Color*. Thus the assertion *(#!hasColor #!Canary #!Yellow)* is legal and means that every instance of *#!Canary* has color *#!Yellow* (by default). The assertion *(#!hasColor #!Canary #!Chair)* is illegal because *#!Chair* is not an instance of *#!Color*.

Predicate domain and ranges are specified using the Parka commands **new-pred** (i.e., when a slot is defined) or **assert!** (i.e., after a slot is defined).

2.3.2 Category and Instance Predicates

Category predicates are predicates whose domain includes set of all categories. Thus the first argument of a category predicate is a category. Category predicates are used to represent properties of the set represented by the category. Since categories are sets (i.e., non-physical, mathematical entities), they cannot have physical properties (e.g., color). For example, predicate *#!numMembers* could be defined with domain *#!Category* and range *#!IntegerValue*. This means that the predicate

[3] In other words, assertions are not reified in Parka.

is "defined" over all instances of the domain *#!Category*: i.e., the set of all categories. Thus the assertion *(#!numMembers #!Canary 1000)* could be used to represent the fact that the set of canaries has 1000 members.

Instance predicates are used to represent properties common to all members of a set, rather than the set itself. Instance predicates do not have *#!Category* or a supercategory of *#!Category* (e.g., *#!Thing*) for their domains. For example, the predicate *#!hasColor* has a domain of *#!PhysicalObject*. This means that *#!hasColor* is defined for all instances of PhysicalObject. Thus the assertion *(#!hasColor #!Canary #!Yellow)* could be used to represent the fact that every instance of Canary is yellow (by default). In this case the predicate *#!hasColor* applies to members of the set, rather than the set itself.

Category and instance predicates are handled differently by the IDO inheritance mechanism (see Section 2.6).

2.3.3 Instance-Of and Is-A

Two predicates, instanceOf and isa, deserve special attention because they are treated specially during IDO inheritance (see Section 2.6.1). Assertions containing these predicates along with the categories and instances in the KB form the "Is-A Hierarchy" of a KB.

The predicate *#!instanceOf* represents the set-membership relation. Assertions for *#!instanceOf* have the form *(#!instanceOf x y)*, where x is an instance frame and y is a category frame, and means that the individual represented by x is an element of the set represented by y. x is called an "instance", "member", or "element" of y. y is called a "category" of x. Note that x may be an instance of more than one y.

The predicate *#!isa* represents the subset relation. Assertions for *#!isa* have the form *(#!isa x y)*, where x and y are category frames, and means that the set represented by x is a subset of the set represented by y. x is called a "subcategory", "subset", or "is-a child" of y. y is called a "supercategory", "superset", or "is-a parent" of x. Note that x may be a subcategory of more than one y.

2.3.4 Predicate (Slot) Inverses

The inverse mechanism in Parka saves the user from having to make some assertions. For example, the user could define a predicate *#!father=* with domain *#!Animal* and range *#!Male*. Then, one could state that Sweety's father is Tweety with the assertion *(#!father= #!Sweety #!Tweety)*. Suppose one has also defined *#!fatherOf*, the inverse predicate of *#!father=*. Rather than the user also having to assert *(#!fatherOf #!Tweety #!Sweety)*, the system will do so automatically if *#!fatherOf* is specified as inverse of *#!father=*. Note that the domain of *#!fatherOf* should be defined as *#!Male* and the range as *#!Animal* (the opposite of *#!father=*).

To specify that a predicate frame p_1 is the inverse of predicate frame p_2, one must assert *(#!inverse p_2 p_1)* after both p_1 and p_2 have been defined. A predicate may be its own inverse: i.e., p_1 and p_2 can be the same frame. A predicate may not have more than one inverse. Note that p_1 is a many-to-one relation, then its inverse p_2 should be defined as a one-to-many relation by specifying a cardinality of *#!MultiValue* (see section 2.4.1).

Note that the system explicitly asserts an inverse of an assertion when that assertion is made by the user (i.e, at update time). It does not compute the value of the inverse slot at query time. An additional example of slot inverses is given in the example in Figure 2.4.

2.3.5 Superpredicates (Superslots)

The superpredicate mechanism in Parka saves the user from having to make some assertions. For example, the user could define a predicate *#!father=* with domain *#!Animal* and range *#!Male*. Then, one could state that Sweety's father is Tweety with the assertion *(#!father= #!Sweety #!Tweety)*. Suppose one has also defined *#!parent=*. Rather than the user also having to assert *(#!parent= #!Sweety #!Tweety)*, the system will do so automatically if *#!parent=* is specified as a superpredicate of *#!father=*. Note that the domain of *#!father=* should be defined as *#!Animal* and the range as *#!Animal* (where *#!Male* is a subcategory of *#!Animal*).

To specify that a predicate frame p_1 has predicate frame p_2 as its superpredicate, one must assert *(#!superSlot* p_1 p_2*)* after both p_1 and p_2 have been defined. A predicate may have more than one superpredicate.

Note that the system explicitly asserts the "superslot" assertion when the "subslot" assertion is made by the user (i.e., at update time). It does not compute the value of the superslot at query time. An additional example of slot inverses is given in the example in Figure 2.4.

2.3.6 Inheritance Methods and Predicates

The slots for certain predicates may be inheritable. Parka's built-in inheritance methods include inferential distance ordering (IDO), transitivity, transfers-through, composition, and no inheritance. They are described in section 2.6. Each inheritable predicate has an inheritance method associated with it that is used to compute inherited slots for that predicate.

The value of a predicate's *#!toCompute* slot, v, specifies the inheritance method for that predicate. The inheritance method associated with predicate p is typically specified implicity when p is defined using the **new-pred** command. **new-pred** allows the user to specify value c for its optional **:instance-of** parameter. Thus p will be an instance of category c, where the default value for c is *#!InheritablePredicate*, and p will inherit a particular value for v from c as shown in the following table:

Inheritance Method	*Category (c)*	*toCompute Value (v)*
IDO	InheritablePredicate	*#!IDOinheritance*
Transitive	TransitivePredicate	*#!ComputeTransitive*
TransfersThrough	TransfersThroughPredicate	*#!ComputeTransfersThrough*
Composition	CompositePredicate	*#!ComputeByComposing*
None	DoesNotInheritPredicate	*#!NoInheritance*

2.4 Frames and Slots

Frames are the basic high-level data structures in Parka. Categories, instances, and predicates are specified via category frames, instance frames, and predicate frames, respectively. A frame may have slots, which correspond to assertions made about the frame.

Category, instance, and predicate frames are created by the Parka commands **new-category**, **new-instance**, and **new-pred**, respectively.

2.4.1 Slots

Slots consist of a predicate and a value and correspond to assertions made about a frame. For each assertion *(p f v)*, a *p* slot is created for frame *f* with value *v*. Assertions are made using the Parka command **assert!**. Making an assertion fills a slot with a value.

Explicit Slot Values

Explicit values for slots are those specified by the user via assertions. The simplest type of slot value is a single frame or immediate value. Immediate values may be numbers, boolean values, strings, and symbols. Numbers may be of type integer or float. Boolean values are T (true) or NIL (false).

The type of value allowed for a slot is specified by the range value for the slot's predicate. If the range is a non-special frame the value of the slot must be a frame that is transitively a subcategory or instance of the range. For example, if predicate *#!hasColor* has range *#!Color*, the *#!hasColor* slot of any frame must be filled with a frame that is (transitively or directory) a subcategory or an instance of *#!Color* such as *#!Yellow*.

The following special frames are used as range values for predicates whose slots can have immediate values:

Boolean	*#!BooleanValue*
Float	*#!FloatValue*
Integer	*#!IntegerValue*
String	*#!StringValue*
Symbol	*#!SymbolValue*

For example, to specify that a person's last name has an immediate value of type string, one would define the range of predicate *#!lastName=* as *#!StringValue*.

Multiply-Filled Slots

Some predicates can be defined so that there can be multiple assertions for the predicate for a given frame. In other words, the slot for that predicate for a given frame can be filled with multiple values. For example, to specify that the U.S. Flag has the colors red, white, and blue, one could define the predicate *#!hasColor* with a range of *#!Color* and multiple cardinality (the value of *#!cardinality* for

#!hasColor is *#!MultiValue* instead of *#!SingleValue*, the default). Once *#!hasColor* is defined, one may then make the assertions: *(#!hasColor #!US-Flag #!Red)*, *(#!hasColor #!US-Flag #!White)*, and *(#!hasColor #!US-Flag #!Blue)*. This will fill the *#!hasColor* slot for *#!US-Flag* with the 3 frame values: *#!Red*, *#!White*, and *#!Blue*.

A slot can also be filled with multiple immediate values, if the predicate's range is defined appropriately (see the previous section).

Set-Valued Slots

It is possible to have a slot filled with a set (the Lisp data type `list`). Elements of the set must be in the range of that slot's predicate. One can have a set of frames or a set of immediate values. All elements of the set must be in the same range set. Set-valued slots differ from multiply-filled slots in that the set is treated as an atomic object by Parka. Parka cannot "look" inside a set when processing a query.

For example, one could define *#!color-set=* to have range *#!Color* and slot format of *#!SetFormat*. This means that one could then assert *(#!color-set= #!US-Flag (#!Red #!White #!Blue))* The difference between this and the example of the previous section is that here we have 1 value (a set) for *#!US-Flag*'s *#!color-set=* slot versus 3 values (each individual frame) for *#!US-Flag*'s *#!hasColor* slot. In the latter case, we can query about the particular values in the set.

To specify a slot can be filled with a set, one asserts *(#!slotFormat p #!SetFormat)*, where *p* is the predicate frame of the slot.

Vector-Valued Slots

It is possible to have a slot filled with a vector (the Lisp data type `vector`). Elements of the vector must be in the range of that slot's predicate. One can have a vector of frames or a vector of immediate values. All elements of the vector must be in the same range set. Vector-valued slots differ from multiply-filled slots in that the vector is treated as an atomic object by Parka. Parka cannot "look" inside a vector when processing a query. They differ from set-valued slots in that the elements of the vector are assumed to be ordered.

For example, one could define *#!actions=* with to have range *#!Action* and slot format of *#!VectorFormat*. This means that one could then assert *(#!actions= #!Jane #(#!Hop #!Skip #!Jump))*.

To specify a slot can be filled with a vector, one asserts *(#!slotFormat p #!VectorFormat)*, where *p* is the predicate frame of the slot.

2.5 Other Language Features

2.5.1 Update Methods

Unlike previous versions of Parka, Parka 3 is capabable of limited forms of procedural attachment. Users can specify a (Lisp) function to be run under the following conditions:

- before an assertion is performed (a before-asserting method)
- after an assertion is performed (an after-asserting method)
- before an assertion is retracted (a before-retracting method)
- after an assertion is retracted (an after-retracting method)

Functions must take three arguments: the predicate and the two arguments of an assertion. For on-asserting methods, the function is run only if the assertion has not already been explicitly made in the current KB. For on-retraction methods, the function is run only if an assertion has been explicitly made in the current KB. A particular predicate may have multiple methods for each of the above types associated with it.

The following is an example of one use of an update method. This method (used internally by Parka) causes an inverse assertion to be added when an assertion is made on a predicate for which an inverse predicate was specified. For example, if the following method is specified for predicate *#!foo* with inverse *#!bar*, the inverse assertion *(#!bar b a)* will be made by the method when *(#!foo a b)* is initially asserted.

```
(defun maintain-inverse/assert (pred arg1 arg2)
   ;; Assert (pred arg2 arg1 ctxt) for inverse
   (assert2 (pred-inverse pred) arg2 arg1))
```

To specify a function for a method for a particular predicate, one must make an assertion of the form: `(assert! '(`p_m p f`))`, where p_m is a predicate specifying the type of method – *#!beforeAssertingMethods*, *#!afterAssertingMethods*, *#!afterDeletingMethods*, or *#!beforeDeletingMethods*; p is the predicate frame that the method is being associated with); or f is the name (symbol) of the function (e.g., maintain-inverse/assert).

2.6 Inheritance

Inheritance allows a slot for predicate p to be inferred for frame f_1 from one other frame f_2. The particular inheritance method determines which frame f_2, if any, a slot for p will be inherited from by frame f_2. A slot for predicate p is inherited only by frames which do not already have an explicit slot for p (an explicit slot is one created by the **assert!** command). The value inherited for f_1's p-slot can be a single value (frame or immediate value) or multiple values depending on whether f_2's p-slot is singly-valued or multiply-valued, respectively. Parka supports multiple inheritance: a particular frame f_1 can inherit slots from multiple frame, but f_1 inherit a slot for any particular predicate from one frame only.

Parka's built-in inheritance methods include inferential distance ordering (IDO), transitivity, transfers-through, composition, and no inheritance. They are described in the following sections. Each predicate has an inheritance method associated with it. Section 2.3.6 describes how to specify an inheritance method for a particular predicate.

2.6.1 Basic IDO Inheritance

This form of inheritance allows properties (slots) describing a category's instances to be inherited by instances of that category (direct instances) or by instances of that category's subcategories (indirect instances). Thus properties are passed "down" over one or more *#!isa* links and a single *#!instanceOf* link. Only slots for instance predicates (see Section 2.3.2) are inherited. The predicate must be specified as inheritable (see Section 2.3.6). In the sample net of Figure 2.2, the slot (*#!hasColor #!Yellow*) will be inherited from the category frame *#!Canary* by its instance *#!Tweety*.

Sometimes a conflict may occur when an instance belongs to multiple categories, each of which has a slot (explicit or inherited) for the same predicate. This is a conflict because a frame can inherit a slot for a particular predicate from one other frame only. For example, consider the net in Figure 2.3. From which of its categories does the Richard Nixon inherit the slot for the predicate pacifist?.

To resolve these kinds of conflicts, Parka uses an approximation of the Touretzky's IDO (inferential distance ordering) algorithm. The details of Parka's algorithm can be found in [2]. When a frame f can inherit from multiple "parent" frames (categories in Parka), Parka uses the topological depth numbers to resolve the conflict. These topological depth numbers are calculated by downward traversal in parallel of the *#!isa*/*#!instanceOf* hierarchy. Parka's IDO approximation cannot resolve inheritance conflicts of the sort in figure 2.3 if a slot can be inherited for the predicate from multiple frames with the same topological sort numbers (e.g., if *#!Republican* and *#!Quaker* had the same topological sort numbers). If an inheritance conflict cannot be resolved, Parka randomly chooses one of the frames to inherit the slot from.

Inheritance of Instance and Category Slot Values

A **category-predicate slot** is a slot whose predicate is a category predicate; an **instance-predicate slot** is a slot whose predicate is an instance predicate (see Section 2.3.2). Category-

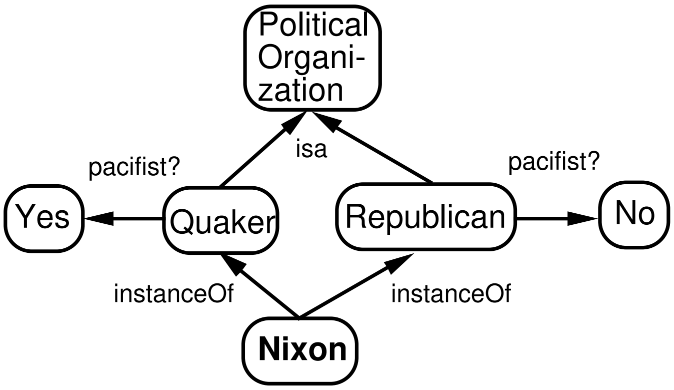

Figure 2.3: A Problematic Net for Inheritance ("The Nixon Diamond")

predicate slots that are explicitly valued for a (category) frame are not inherited by the frame's subcategories nor instances. Instance-predicate slots that are explicitly valued for a (category) frame are inherited by the frame's instances (unless the slot is specified as noninheritable).

To continue the example from Section 2.3.2, the *#!numMembers* slot (and its value of 1000) will not be inherited by subcategories and instances of *#!Canary*, because *#!numMembers* is a category predicate. Thus *#!Tweety* will have no value for *#!numMembers*. On the other hand, the *#!hasColor* slot (and its value of *#!Yellow*) can be inherited by instances of *#!Canary*, because *#!hasColor* is an instance predicate. Thus unless *#!Tweety* is explicitly valued for *#!hasColor*, it will have the value *#!Yellow* for *#!hasColor*.

2.6.2 Transitive Predicate Inheritance

The transitive inheritance mechanism allows some assertions to be inferred at query time. If a predicate is declared to be transitive (see section 2.3.6), then frames with slots for that predicate can obtain values both explicitly and through transitivity. For example, if predicate p is asserted to be transitive and the assertions *(p f v)* and *(p f w)* made (where f is a frame, v and w are values), then f's p slot would have both the explicitly asserted value v and the inferred value w.

The following example shows how transitive predicates, slot inverses, and superslots can be used to represent group membership (a kind of partonomy). In the net in Figure 2.4 (created by the commands in Appendix D.2), we represent the following facts: Captain Picard, Commander Riker, and Mr. Data are members of the Bridge Crew; Dr. Crusher is a member of the Enterprise Crew (but not of Bridge Crew); and the Bridge crew is a subset of the Enterprise Crew.

Direct membership is represented using the predicate *#!member=* with domain *#!Group* and range (one or more instances of) *#!Person*: e.g., the value of *#!Bridge-Crew*'s *#!member=* slot is the frames *#!Capt-Picard*, *#!Cmdr-Riker*, and *#!Mr-Data*. The inverse of *#!member=* is the predicate *#!member-Of* with domain *#!Person* and range *#!Group*: e.g., *#!Capt-Picard*'s *#!member=* slot has the value *#!Bridge-Crew*.

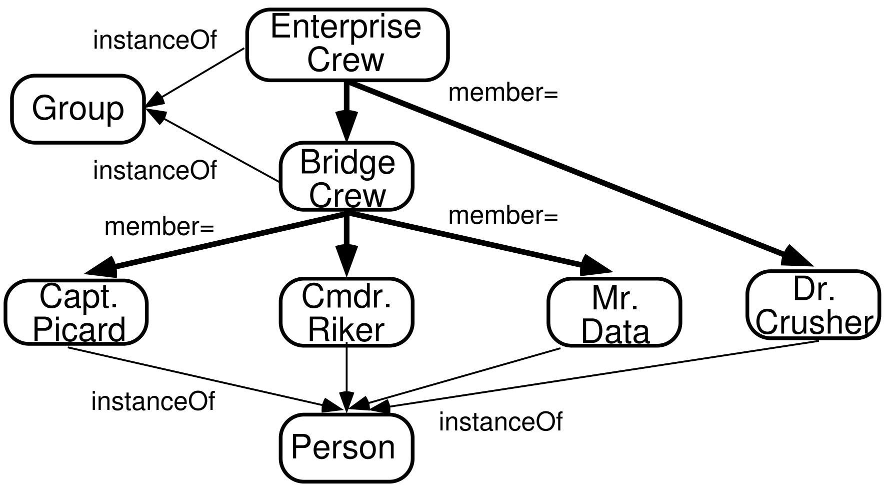

Figure 2.4: A Sample Net for Transitive Inheritance (created using commands in Appendix D.2).

If x is a member of y and y is a member of z, then x is a member of z by transitivity. Transitive membership is represented using the predicate *#!tmember=*, with domain *#!Group* and range (one or more instances of) *#!Person*, and by the inverse predicate *#!tmember-Of*, with domain *#!Person* and range *#!Group*. Both *#!tmember=* and *#!tmember-Of* are specified as transitive predicates. The predicate *#!tmember=* is specified as a superslot of predicate *#!member=* because the transitive member relation represented by the former subsumes the direct member relation represented by the latter. This is also the case with *#!tmember-Of* and *#!member-Of*.

The following are some queries on the net in Figure 2.4 showing transitive inheritance via the predicates *#!tmember* and *#!tmember-Of*. The query syntax is described in Section 2.7 (Note that lines in the following dialog prefaced with "?" are the user's input to Lisp).

```
; What are the (direct) members of the Bridge Crew?
? (query! '(#!member= #!Bridge-Crew ?x))
(((?X . #!Capt-Picard)) ((?X . #!Cmdr-Riker)) ((?X . #!Mr-Data)))

; What are the (direct) members of the Enterprise Crew?
? (query! '(#!member=  #!USS-Enterprise-Crew ?x))
(((?X . #!Bridge-Crew)) ((?X . #!Dr-Crusher)))

; What are the (direct and) transitive members of the Enterprise Crew?
? (query! '(#!tmember= #!USS-Enterprise-Crew ?x))
(((?X . #!Capt-Picard)) ((?X . #!Cmdr-Riker)) ((?X . #!Mr-Data))
 ((?X . #!Bridge-Crew)) ((?X . #!Dr-Crusher))
 ((?X . #!USS-Enterprise-Crew)))

; What crew(s) is Riker a (direct) member of?
```

```
? (query! '(#!member-Of  #!Cmdr-Riker ?x))
(((?X . #!Bridge-Crew)))

; What crew(s) is Rider a (direct and) transitive member of?
? (query! '(#!tmember-Of #!Cmdr-Riker ?x))
(((?X . #!USS-Enterprise-Crew)))
```

2.6.3 Transfers-Through Inheritance

The transfers-through inheritance method allows some assertions to be inferred at query time. One predicate p_1 can be declared to transfer through another predicate p_2 (see Section 2.3.6). If frame x lacks an explicit slot for p_1, x has a slot for p_2 filled by frame y, and y has value v for its p_1 slot, then x will have the inferred value v as the value of its p_1 slot.

For example, if predicate *#!lastName=* is asserted to transfer through *#!father=*. This specifies that last names are obtained from the frame filling the *#!father=* slot. *#!Sweety* has no slot for *#!lastName=* but has *#!Tweety* for its *#!father=* slot. *#!Tweety* has the string "Bird" as the value of its *#!lastName=* slot. Thus a value of "Bird" will be inferred for the *#!lastName=* slot of *#!Sweety*.

2.6.4 Predicate Composition Inheritance

The predicate composition inheritance method allows some assertions to be inferred at query time though the composition of other assertions. The user can specify that a particular predicate p is composed of one or more other predicates $p_1, p_2, \ldots p_n$ (p_i itself may be a composite predicate).

For example, the predicate *#!uncle* can be defined as a *#!CompositePredicate* that is the composition of the predicates *#!parent* and *#!brother*. Thus if the assertions *(parent Tom Dick)* (i.e., Tom's parent is Dick) and *(brother Dick Harry)* (i.e., Dick's brother is Harry) hold, the assertion *(uncle Tom Harry)* will be inferred (i.e., Tom's uncle is Harry). A sample example of the commands to define predicate composition is in Appendix D.3.

Other inheritance methods that run at query time (i.e., transfers-through, transitive, IDO) can use composite predicates. For example, the slot value for some other predicate could be asserted as transferring through the preciate *#!uncle*.

2.7 Querying

Parka's query facility provides access to the assertions contained in the KB using Parka's parallel inference mechanisms.

2.7.1 Simple Queries

Simple queries have the syntax `(query! '(`$p\ x\ y$`))`, where p is predicate frame; x is a frame or variable; and y is a frame, immediate value, or variable. Parka query **variables** are Lisp structures that are specified by a "?" prefix: e.g., *?foo*, *?bar*, etc. Immediate values are those described in section 2.4.1. The command **query!** returns NIL or a list of sets of bindings if argument x or y is a variable. Otherwise, **query!** returns NIL or the list `(((T . T)))`. A set of bindings contains one binding for each variable in the query. Bindings are of the form: $(v.z)$, where v is a variable specified for argument x or y; and z is a frame or immediate value that is bound to v by the matcher.

Thus the following types of queries can be issued depending on the type of arguments specified for `(query! '(`$p\ x\ y$`))`:

	Arg. x	*Arg. y*	*Description*
1	frame	frame or immed. value	Does frame x have y filling its p slot?
2	frame	variable	What fills frame x's p slot?
3	variable	frame or immed. value	Which frames have y filling their p slot?
4	variable	variable	Which frames have anything filling their p slot?

Queries of type in (1) and (2) are "bottom-up queries". Queries of type in (3) and (4) are "top-down" queries, which can be answered extremely quickly via Parka's parallel inference methods.

The following are same queries corresponding to the above query types for the sample KB shown in Figure 2.2. (Note that lines in the following dialog prefaced with "?" are the user's input to Lisp).

```
;;; 1. Is Tweety yellow?
? (query! '(#!hasColor #!Tweety #!Yellow))
(((T . T))) ; answer is true

;;; 2. What color is Tweety?
? (query! '(#!hasColor #!Tweety ?y))
(((?Y . #!Yellow))) ; answer is yellow

;;; 3. What is yellow?
? (query! '(#!hasColor ?x #!Yellow))
(((?X . #!%P-Canary)) ((?X . #!Tweety)) ((?X . #!Sweety)))
; answer is the prototypical canary, Tweety, and Sweety

;;; 4. What is colored?
```

```
? (query! '(#!hasColor ?x ?y))
; ?X is the colored thing, ?Y is the color
(((?X . #!%P-Canary) (?Y . #!Yellow))
 ((?X . #!Tweety) (?Y . #!Yellow))
 ((?X . #!Sweety) (?Y . #!Yellow))
 ((?X . #!%P-Bird) (?Y . #!Brown)))
```

In addition to simple queries, Parka supports conjunctive queries consisting of multiple simple queries. These conjunctive queries are of two types: recognition queries and structure queries.

2.7.2 Recognition Queries

Recognition queries are of the form "find all frames x such that $p_1(x, y_1)$ and $p_2(x, y_2)$ and ... and $p_n(x, y_n)$", where p_i is a predicate frame, x is a variable, and y_i is a frame or immediate value. Recognition queries have syntax: `(query! '(:and (`p_1 x y_1`) (`p_2 x y_2`) ... (`p_n x y_n`))`. Note that recognition queries have only a single variable (i.e., x). The elements (clauses) following ":and" can be in any order.

The following query on the sample net in Figure 2.2 finds all things that are yellow and covered by feathers:

```
? (query! '(:and (#!hasColor ?x #!Yellow)
                 (#!coveredBy ?x #!Feathers)))
(((?X . #!Tweety)) ((?X .  #!%P-Canary)) ((?X . #!Sweety)))
```

This query correctly returns all canaries in the KB (including the prototypical canary).

2.7.3 Structure Queries

Structure queries are similar to recognition queries. Unlike recognition queries, however, structure queries can contain multiple variables. Structure queries are of the form "find all subgraphs in the semantic net (graph) of the KB that match a probe graph".

The syntax for structure queries is `(query! '(:and` c_1 c_2 `...` c_n`))`, where c_i is a "constraint". Each c_i has the form (p_i x_i y_i), where p_i is a predicate frame; x_i is a variable; and y_i is a variable, frame or immediate value. If y_i is a frame or immediate value, c_i is a "unary constraint". If y_i is a variable, c_i is a "binary constraint". Variables correspond to nodes in the probe graph (variables are not allowed for edges). Constraints correspond to edges in the probe graph. A unary constraint is an edge connecting a node corresponding to a constant to a node corresponding to a variable. A binary constraint is an edge connecting two nodes that correspond to variables.

There are two special cases in which structure query constraint c_i is not processed as a normal unary or binary constraint. If c_i is `(`*#!equal* x_i y_i`)` and x_i and y_i are both (different) variables, then c_i is interpreted to mean return all the sets of bindings where variables x_i and y_i are bound to *the same* values. If c_i is `(`*#!neq* x_i y_i`)` and x_i and y_i are both (different) variables, then c_i is interpreted to mean return all the sets of bindings where variables x_i and y_i are bound to *different* values.

Probe graphs are matched to the KB by the Parka Structure Matcher mechanism [1]. Probe graphs must match the KB exactly. If an exact match is found, list of bindings for the variables in the query are returned. Otherwise, NIL is returned.

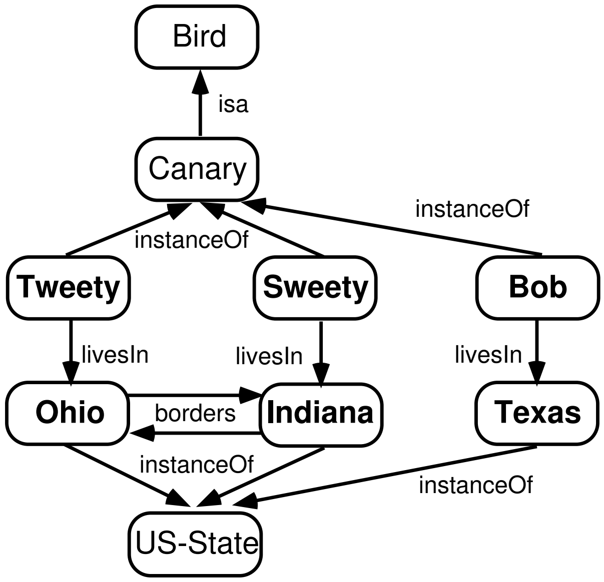

Figure 2.5: A Sample Net for Structure Querying

For example, consider the semantic net (graph) in Figure 2.5, an extension of that in Figure 2.2, and the query "find two canaries that live in neighboring states". The probe graph corresponding to this query is shown in figure 2.6. The darker edges in the probe graph are binary constraints. The lighter edges are unary constraints. The following is the specification of the query and its result (variables are prefixed with "?"):

```
? (query! '(:and (#!instanceOf ?can1 #!Canary)   ; unary constr.
               (#!instanceOf ?can2 #!Canary)     ; unary constr.
               (#!instanceOf ?state1 #!US-State) ; unary constr.
               (#!instanceOf ?state2 #!US-State) ; unary constr.
               (#!livesIn ?can1 ?state1)         ; binary constr.
               (#!livesIn ?can2 ?state2)         ; binary constr.
               (#!borders ?state1 ?state2)       ; binary constr.
         ))

(((?CAN1 . #!Sweety) (?CAN2 . #!Tweety) (?STATE1 . #!Indiana)
(?STATE2 . #!Ohio))
 ((?CAN1 . #!Tweety) (?CAN2 . #!Sweety) (?STATE1 . #!Ohio)
 (?STATE2 . #!Indiana)))
```

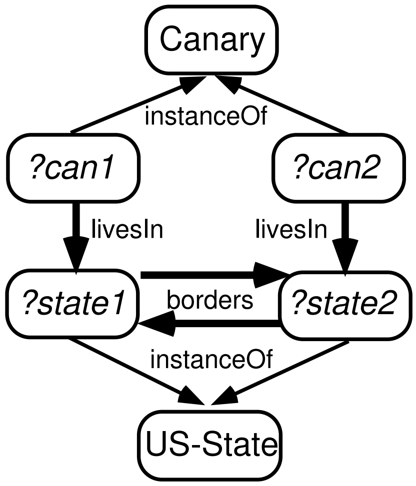

Figure 2.6: A Sample Probe Graph for Structure Querying.

Note that the two sets of bindings are returned, one for each combination of bindings to *?can1* and *?can* (or *?state1* and *?state2*).

The constraints in a structure query can be specified in any order. Parka processes the unary constraints first. Care should be taken to ensure that each variable has at least one corresponding unary constraint that defines its domain (e.g., (*#!instanceOf x #!Foo*)). Otherwise, the domain of a variable defaults to *#!Thing*, which may result in longer query processing time and/or unintended results being returned.

2.7.4 Built-in Queries

Certain frequently used queries can be issued using other query functions built into Parka. These functions are provided for convenience and are listed in Appendix A. For example, the query function **instances** returns all instances of a frame. The following commands are equivalent (though they differ in the format of their results):

```
? (instances #!Canary)
(#!%P-Canary #!Tweety #!Sweety)
? (query! '(#!instanceOf ?x #!Canary))
(((?X . #!%P-Canary)) ((?X . #!Tweety)) ((?X . #!Sweety)))
```

Chapter 3

The Parka System

3.1 Knowledge-Base Maintenance

A Parka KB (or Parka "net") is built by first initializing the KB via the **init-parka** command. An initialized KB will contain only the predefined frames and assertions of the Parka base ontology (see Appendix B. The initialize KB is rooted at frame *#!Thing*.

A user can define frames and make assertions using the commands described in Appendix A. A user can choose to save a KB via the Parka **dump-kb** command. This creates a binary file which can then be loaded later using the **load-kb** command. Loading a binary file is much faster than loading a file of Parka KB definition commands.

In the Macintosh Common Lisp version of Parka, a user can initialize a KB and dump/load a binary file from the Parka menu on the Lisp menu bar.

Frames in a Parka KB can be viewed using the **describe-frame** command, which display information about a frame textually, or the graphical frame browser described in the following section. The **frame-apropos** command can be used to search for a frame name using part of a name, or one may invoke the graphical frame-apropos dialog from the menu bar.[1]

For additional KB commands, see Appendix A.

[1]Currently this menu option is only available in the Macintosh Common Lisp implementation of Parka.

3.2 The Graphical Browser

The Parka Browser is a graphical, hierarchical browser for viewing the Parka frame space. You can create many simultaneous browsers, each showing different parts of the KB. The browser system offers several different ways of finding frames to browse, and several different ways of viewing browsable data.

The graphical browser is currently supported only on the Macintosh Common Lisp version of the Parka front-end system.

3.2.1 A Walk-Through

Choose "New" from the "Browse" menu (shown in figure). A default Parka Browser appears: a three-column browser rooted at *#!Thing* (the built-in root frame of any Parka KB). A snapshot of the browser is shown in figure . Each column in the browser represents a frame, the root (first) frame being the leftmost colum. As you browse from frame to frame, new frames are added to the right, scrolling the columns to make room if necesssary (you can scroll back and forth through your past collection using the ← and → buttons at the bottom). If you like, you may fit more columns in the browser at one time by stretching the browser window.

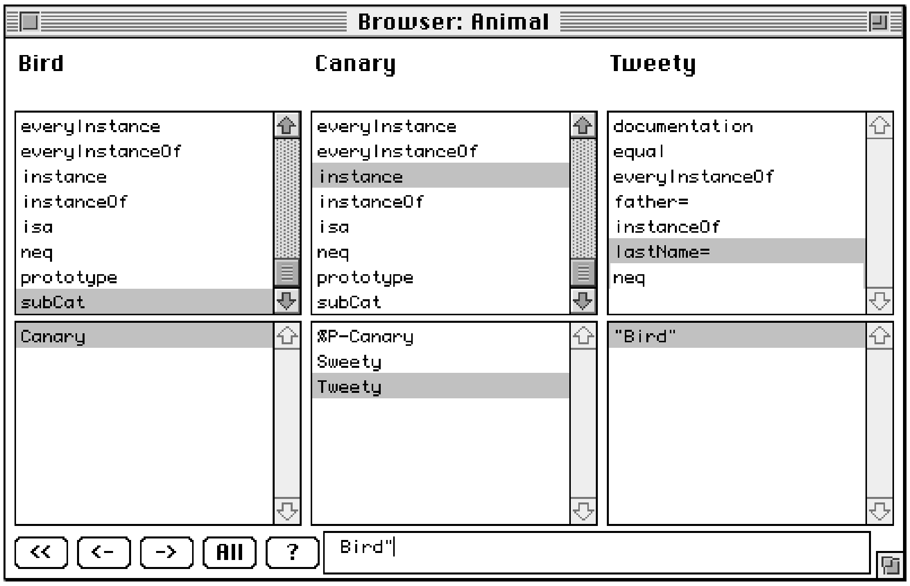

Figure 3.1: A Snapshot of a Browser Window

A browser column lets you examine a particular frame and all the slots and data associated with it. A column consists of three items (from top to bottom): the frame's name, a list of slots in the frame, and (if you've selected one of these slots) a list of values for the selected slot. Values can be pointers to other frames, or non-frame data like strings, numbers, vectors, sets, or CLOS methods, and the list of values changes depending on the slot you select.

When you select a value and that value is a pointer to a frame, this new frame appears in the column to the right: you've "moved" to the frame. If the browser doesn't have enough space to display the new frame, it scrolls to the left to make room. This way the browser builds up a history of previously-visited frames. Each frame-column is related to its "parent" column (its left neighbor) by the parent's selected slot and value. This hierarchy goes all the way back to the root frame (often *#!Thing*) at the far left of the list. Figure 3.1 shows part of our **Tweety** ontology: *#!Canary* is an instance of *#!Bird*, *#!Tweety* is an instance of *#!Canary*, and "Bird" is the last name of *#!Tweety*. The browser is rooted at *#!Animal*, which has been scrolled off the browser.

Whenever you select a slot or value, the slot or value also appears in the text field at the bottom of the browser. For example, in the browser in figure 3.1), the value "Bird" appears, because "Bird" was last item selected). Since the columns in the browser aren't very wide, this text field helps you read slots or values wider than the width of a column.

3.2.2 Getting More Information

The **?** button calls **frame apropos** on your current (rightmost) frame.

The **All** button brings forward a Fred window displaying all the slot and value information in the entire frame.

3.2.3 Resetting Browsers

The ≪ button resets the browser, removes all the columns, and displays the root frame.

To re-root an existing browser to its far-right column (its "current" frame), choose "Root Top Frame" from the "Browse" menu.

To completely remove all browsers and associated browser windows, choose "Reset Browsers" from the "Browse" menu. This should be done whenever your Parka database accepted an action which could create a data integrity problem with your existing browsers, such as adding any new slot or value, deleting a slot or value, etc.

3.2.4 Choosing the Root for a Browser

There are several ways to create browsers rooted at something other than *#!Thing*:

- Choose "By Global List" from the "New..." submenu of the "Browse" menu.

  This brings up a global list of all frames in the Parka database. Select a frame and press the "New" button, and a browser rooted at that frame appears.

- Choose "From Top Frame" from the "New..." submenu of the "Browse" menu.

  This brings up a new browser rooted at the frame in the far-right column of your current browser. This does not remove your original browser.

- Browse an item from Frame Apropos

  There's a "Browse" button in the Frame Apropos window. Pressing it will create a browser rooted at your current selection in the Frame Apropos window.

- Browse an item from a Query window

  You can browse constants and ground instances of variables in the Query window by choosing "Browse Binding" from the "Query" Menu.

3.2.5 Changing the Browser's Display

By default, the browser sorts its lists and displays just slots local to the frame. To display nonlocal slots, deselect "Local" from the "Browse" menu. To display all lists unsorted (this is often much faster for large databases), deselect "Sorted" from the "Browse" menu. Note that selecting or deselecting either of these options automatically chooses "Reset Browsers" (see above), since both options could create data integrity problems with the browsers.

Next to each slot in a slot list, the browser can display the number of values associated with the slot, or (if there's just one value for the slot) the value itself directly to the right of the slot. This feature is convenient but often slow for large databases, so it's turned off by default. To turn this feature on, select "Display Info" from the "Browse" menu. This is a lazy feature, so frames will start reflecting this choice only when redisplayed (by creating them fresh or resizing the window).

3.2.6 Tidbits

Double-clicking on a slot creates a new browser rooted on the slot itself.

Double-clicking on a vector or set value pops up a list of elements in the vector or set. From this list you may choose a value–if the value is a frame, the browser moves to that frame. This way you can browse not only frames but vectors and lists stored within them.

Double-clicking on a non-vector, non-set value does the same thing as the **All** button described above.

You can manipulate the browser using only keystrokes. Type text and the slot list or value list you're currently working in will try to find the closest match to your typing. You can also scroll these lists using the ↑ and ↓ keys, or Page-Up, Page-Down, Home, and End. Choose elements in them by typing Enter, Tab, Return, or →. Go back to previous lists by typing ←.

3.2.7 Browser Commands

As an alternative to selecting items from the Browser Menu, a user or application program may invoke the browser through the following commands.

new-browser (&optional (root #!*Thing*))
Creates a new browser rooted at frame *root*.

reset-browsers ()
Resets and closes all browsers and associated windows.

3.3 The Query Window

A Query Window provides an interface to creating and reading queries in Parka. It does this by presenting query data as a network where nodes in the network are data (frames, strings, variables, etc.), and edges between the nodes are relationships (slot-values) between nodes. For example, if we wanted to ask `(query! '(`*#!isa ?X #!Thing*`))`, we'd create a node called *?X*, a node called *#!Thing*, and an edge called *#!isa* connecting *?X* to *#!Thing*.

The Query Window is currently supported only on the Macintosh Common Lisp version of the Parka front-end system.

3.3.1 A Walk-Through

Rather than lay out all the features of the system immediately, let's do a walk-through tutorial on using a Query Window.

Our Example Database

For our walk-through, we'll use a small subset of the **Tweety** net shown in Figure 2.2. Enter the following into Parka before starting the walk-through:

```
(new-category "Canary")
(new-instance "Tweety")
(new-instance "Sweety")
(new-pred "father=" #!Canary #!Canary)
(assert! '(#!instanceOf #!Tweety #!Canary))
(assert! '(#!instanceOf #!Sweety #!Canary))
(assert! '(#!father= #!Sweety #!Tweety))
```

Before continuing, choose **Reset Connection Cache** from the **Query** menu. We'll explain why this is necessary later.

Building a Simple Query

Choose **New Query** from the **Query** menu. The Query Window that appears has two parts:

- A **query palette** where you lay out your query graphically,
- A **query list** where the results for a query are shown.

Creating a Query

Click in the query palette area to create a node in the graph. A **data input window** appears. Here you enter the name for the node. A node represents Parka data (a frame, variable, or other data) that you will relate to other data.

In the data input window, type *#!Canary* and press **Okay**.

The data input window accepts any lisp-readable form. If you want your node to be *#!Thing*, enter *#!Thing* in the window. If you want a variable, say, *?X*, enter **?x** in the window. All valid Parka data can be entered here: variable symbols, strings, lists (sets), vectors, frames, integers or other numbers.

A node called *#!Canary* appears in the palette. The text is colored black, which indicates that the node represents a constant (a non-variable). A variable's text is colored red, and its name will appear without the initial question-mark.

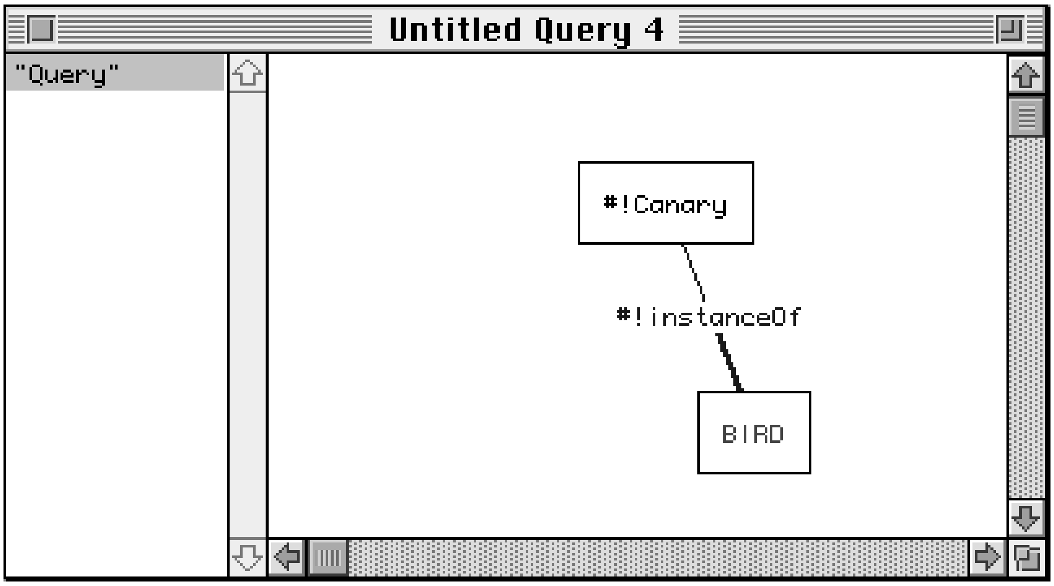

Figure 3.2: Query Window with a Finished Simple Query

Create another node in the graph, and name it *?Bird* (a variable). Now option-drag from *?Bird* to *#!Canary*. The *?connection input window* appears, listing all the predicates the Parka query system knows about. From this list, choose *#!instanceOf*.

Figure 3.2 shows the query you've created. The connection established between *?Bird* and *#!Canary* is the same as saying `(query! '(`*#!instanceOf ?Bird #!Canary*`))`. The thicker end of the connection line indicates the frame, and the thinner end of the line indicates the value in the frame's slot.

Submitting the Query

Choose **Process Query** from the **Query** menu. This submits the query we've created to Parka and fills the query list with the results. In this case, we should get three results back (*#!Tweety*, *#!Sweety*, and *#!Canary*'s prototype instance, *#!%P-Canary*).The list will show them along with the original query (still the first item in the list). The original query's name, **"Query"**, changes to **"Query = 3"** to indicate that there were three results returned.

Choose any of the three new results in the query list. The query palette's variable nodes automatically get filled with the bindings for the result you picked. This way you can use the Query Window not only to lay out and submit queries graphically, but to use the same system to visualize them.

Building a Conjunctive Query

Add another node to the graph: *?Father* (a variable). Don't worry if there are bindings displayed in your query palette; ignore them. If they bother you, choose the **"Query = 3"** to display just your original query.

Option-drag from *?Bird* to *?Father*, and label the connection *#!father=*. Also, option-drag from *?Father* to *#!Canary*, and label the connection *#!instanceOf*. Now choose **Process Query** from the **Query** menu. Figure 3.3 shows the query we've created.

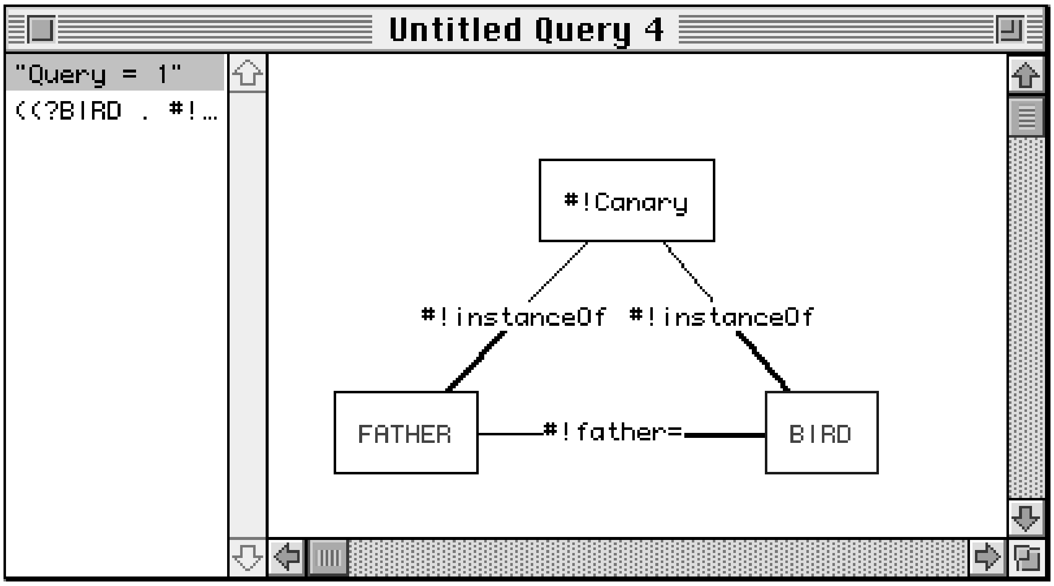

Figure 3.3: Query Window with a More Complex Query

This submits the query: `(query! '(:and (`*#!instanceOf ?Bird #!Canary*`) (`*#!instanceOf ?Father #!Canary*`) (`*#!father= ?Bird ?Father*`))`. Clearly, there's only one set of bindings that matches this, so only one result is returned. Choose it, and see how the bindings fill in the nodes.

3.3.2 Other Features

Editing a Query

You can change the name of a node or connection by double-clicking on its name. To delete a connection or node, click on it (selecting it and turning it blue), and press the **Delete** key. To delete all nodes and bindings in the window, choose **Clear**. To move a node, drag it with the mouse. Lastly, you can reverse the order of a connection by clicking on the connection (selecting it and turning it blue) and choosing **Reverse Connection**. For example, this would change the connection (*#!isa ?X ?Y*) to (*#!isa ?Y ?X*).

Describing Parallel Predicates

It might appear at first glance that you can't create a query like `(query! '(:and (`*#!foo ?X ?Y*`)` `(`*#!bar ?X ?Y*`)))`, since you can only draw one line between two nodes. However, it's perfectly fine to create *two* nodes, each labeled *?X*. Connect one to *?Y* for one predicate, and the other to *?Y* for the second predicate.

File Operations

You can save queries to disk with **Save Query...** and load queries from disk with **Load Query...**

Previewing a Query

Choosing **Preview Query** will display (in a Fred Window) the actual Parka Query you've created. Choosing **Preview Query and Results** will display not only the query, but the results of executing the query (if it can).

Examining Bindings

You can browse a connection or the binding of a node by clicking on it (selecting it and turning it blue) and selecting **Browse Binding**. Likewise, you can examine the full contents of the node in a Fred window with **Examine Binding**. You can also examine the contents by option-double-clicking on a node or connection.

Adjusting the Window

In addition to resizing and scrolling the Query Window, you can also change the amount of the window taken up by the Query list by carefully dragging on the border between the query list and the query palette. To make the query list appear from the top of the window instead of from the left, choose **Flip Query Window**.

Predicates Not Appearing

To generate the connection input window, Parka must gather together all valid binary predicates. If there are a large number of these, gathering them may take quite a while, so the Query Window caches the results. If you add or delete predicates, this cache is no longer valid, so choose **Reset Connection Cache** to tell Parka to re-gather the predicates.

Creating a Query Window Programmatically

new-parka-query-palette (&optional (title size pos))
Creates a new Query Window with the title *title*, of size *size*, and located at position *pos*.

build-graphical-query (query-form)
Creates an initial graphical query for *query-form*. *query-form* must be of the same style as required by **query!**

Appendix A

Parka Version 3 Commands

A.1 Command Names (Commands Grouped Functionally)

- **KB Modification**
  - new-category
  - new-instance
  - new-pred
  - delete-frame
  - assert!
  - retract!
  - set-range-constraint
- **KB Querying (General)**
  - query!
  - local-slots
  - inherited-slots
  - legal-preds
  - category-prototype
  - category-p
  - predicate-p
  - xval-p
  - frame-p
  - frame-to-name
  - name-to-frame
  - num-frames

- num-assertions

- **KB Querying (Isa/InstanceOf Relations)**
  - child-p, children
  - parent-p, parents
  - descendant-p, descendants
  - ancestor-p, ancestors
  - instance-p, instances
  - instance-of-p, instance-of
  - every-instance
  - every-instance-of

- **KB Querying (Predicates)**
  - pred-domain
  - pred-range
  - pred-cardinality
  - pred-format
  - pred-to-compute
  - pred-inverse
  - pred-transfers-through
  - on-asserting-methods
  - on-deleting-methods

- **Other**
  - variable
  - variable-name
  - variable-p

- **KB Display**
  - describe-frame, describe-all-frames
  - frame-apropos, frame-apropos-list

- **KB Maintenance**
  - init-parka
  - dump-kb
  - load-kb

A.2 Command Syntax (Commands Listed Alphabetically)

In the command descriptions below, the following argument type designators are used:

catf	Category Frame
instf	Instance Frame (i.e., not a frame)
predf	Predicate Frame
f	Any Frame (includes all of the above)

ancestor-p ($catf_1$ $catf_2$)
Returns T if $catf_1$ is an ancestor (immediate or transitive supercategory) of $catf_2$ (i.e., `(query! '(`*#!ancestor* $catf_2$ $catf_1$`))` is true) or if $catf_1$ is the same as $catf_2$.

ancestors (*catf*)
Returns ancestors (immediate and transitive supercategories) of category *catf* (i.e., all frames x where `(query! '(`*#!ancestor catf* x`))` is true). Ancestors include *catf*.

assert! (assertion)
Fills x's slot for *pred* with y. Returns NIL.

Argument	*Type*	*Description*
assertion	(*pred* x y)	where *pred* is a predicate frame; x is a frame; y is a frame or immediate value.

category-p (*f*)
Returns T if *f* is a category frame (i.e., `(query! '(`*#!instanceOf f #!Category*`))` is true).

category-prototype (*catf*)
Returns category prototype for *catf* (i.e., frame x where `(query! '(`*#!prototypeOf* x *catf*`))` is true).

child-p ($catf_1$ $catf_2$)
Returns T if $catf_1$ is a child (immediate subcategory) of $catf_2$ (i.e., `(query! '(`*#!isa* $catf_1$ $catf_2$`))` is true).

children (*catf*)
Returns children (immediate subcategories) of category *catf* (i.e., all frames x where `(query! '(`*#!isa* x *catf*`))` is true).

delete-frame (*f*)
Deletes a frame *f* from the KB and all references to *f* (in assertions, etc.).

descendant-p $(catf_1\ catf_2)$

Returns T if $catf_1$ is descendant (immediate or transitive subcategory) of $catf_2$ (i.e., `(query! '(`*#!descendant* $catf_2$ $catf_1$`))` is true) or if $catf_1$ is the same as $catf_2$.

descendants $(catf)$

Returns descendants (immediate and transitive subcategories) of category $catf$ (i.e., all frames x where `(query! '(`*#!descendant* $catf$ x`))` is true). Descendants include $catf$.

describe-all-frames ()

Print information about all frames in KB.

describe-frame (f)

Print information about f.

dump-kb (file)

Save KB to file $file$. Saves a binary-encoded data file.

every-instance $(catf)$

Returns immediate and transitive instances of category $catf$ (i.e., all frames x where `(query! '(`*#!everyInstanceOf* x $catf$`))` is true).

every-instance-of $(instf)$

Returns immediate and transitive categories of f (i.e., all frames x where `(query! '(`*#!everyInstanceOf* $instf$ x`))` is true).

frame-apropos (s)

Print list of frames whose name contains s, a string or symbol.

frame-apropos-list (s)

Return list of frames whose name contains s, a string or symbol.

frame-p (x)

Returns T if lisp object x is a frame.

frame-to-name (f)

Returns the name (string) of f.

inherited-slots (f)

Returns list of $(pred\ value(s))$, for each slot that is inherited by f.

init-parka ()
Initialize KB (removes all user-defined frames and assertions).

instance-of (*instf*)
Returns immediate categories of *instf* (i.e., all frames x where `(query! '(`*#!instanceOf instf* x`))` is true).

instance-p (*catf instf*)
Returns T if category *catf* has an immediate or transitive instance *instf*. (i.e., `(query! '(`*#!instanceOf instf catf*`))` is true)

instances (*catf*)
Returns immediate instances of category *catf* (i.e., all frames x where `(query! '(`*#!instanceOf* x *catf*`))` is true).

instance-of-p (*instf catf*)
Returns T if *instf* is a direct instance of category *catf* (i.e., `(query! '(`*#!instance instf catf*`))` is true).

legal-preds (f)
Returns list of predicates (frames) having domain frame d for which `'(query! '(`*#!everyInstanceOf #!frame* d`))` is true.

local-slots (f)
Returns list of (*pred value*(s)), for each slot that is local to (i.e., not inherited by) f.

name-to-frame (*string*)
Returns the frame with name *string* if it exists. Else, NIL is returned.

new-category (name &rest super-categories)
Defines a new category frame.

Argument	*Type*	*Description*
name	string	name for new frame
super-categories	list of frames	values for new frame's *#!isa* slot

new-instance (name &rest instance-of)
Defines a new instance frame.

Argument	*Type*	*Description*
name	string	name for new frame
instance-of	list of frames	values for new frame's *#!instanceOf* slot

new-pred (name domain range &key cardinality instanceOf format assertions)

Defines a new predicate frame.

Argument	*Type*	*Description*
name	string	name for new frame
domain	*catf*	domain for predicate (default: *#!Thing*)
range	*catf*	range for predicate (default: *#!Thing*)
cardinality	frame	cardinality for predicate: *#!SingleValue* or *#!MultiValue*
instanceOf	*catf*	category for predicate (default: *#!InheritablePredicate*)
format	frame	slot value format for predicate: *#!RegularFormat* (default), *#!SlotFormat*, or *#!VectorFormat*)

num-assertions ()

Return number of assertions in KB.

num-frames ()

Return number of frames in KB.

on-asserting-methods (*predf*)

Returns list of on-asserting-methods for predicate *predf*.

on-deleting-methods (*predf*)

Returns list of on-deleting-methods for predicate *predf*.

parent-p ($catf_1$ $catf_2$)

Returns T if $catf_1$ is a parent (immediate supercategory) of $catf_2$. (i.e., `(query! '(`*#!isa* $catf_2$ $catf_1$`))` is true)

parents (*catf*)

Returns immediate supercategories of category *catf* (i.e., all frames *x* where `(query! '(`*#!isa* *catf* *x*`))` is true).

pred-cardinality (*predf*)

Returns the allowable cardinality (frame) for predicate *predf*.

pred-domain (*predf*)

Returns domain (frame) for predicate *predf*.

pred-format (*predf*)

Returns slot value format (frame) for predicate *predf*.

pred-inverse (*predf*)
Returns inverse predicate (frame) for predicate *predf*.

pred-range (*predf*)
Returns range (frame) for predicate *predf*.

pred-to-compute (*predf*)
Returns inheritance method (frame) for predicate *predf*.

pred-transfers-through (*predf*)
Returns predicate (frame) that inheritance for predicate *predf* transfers through.

query! (form)
Simple query (see section 2.7.1). Returns list of bindings or NIL.

Argument	*Type*	*Description*
form	$(p\ x\ y)$	where p is a predicate frame; x is a frame or variable; y is a frame or immediate value or variable.

query! ('(:and $form_1$ $form_2$ ... $form_n$))
Recognition query (see section 2.7.2). Returns list of bindings or NIL.

Argument	*Type*	*Description*
$form_i$	$(p_i\ x\ y_i)$	where p is a predicate frame; x is a frame or variable; y_i is a frame or immediate value.

query! ('(:and $form_1$ $form_2$ ... $form_n$))
Structure query (see section 2.7.3). Returns list of bindings or NIL.

Argument	*Type*	*Description*
$form_i$	$(p_i\ x_i\ y_i)$	where p is a predicate frame; x_i is a frame, variable, y_i is a frame or immediate value or variable.
	(*#!equal* var_1 var_2)	var_j is a variable.
	(*#!neq* var_1 var_2)	var_j is a variable.

retract! (assertion)
Removes value y from x's slot for *pred*. Returns NIL.

Argument	*Type*	*Description*
assertion	(*pred* x y)	where *pred* is a predicate frame; x is a frame; y is a frame or immediate value.

set-range-constraint (pred lambda-expr &optional name)
pred is a predicate frame and $lambda - expr$ is a lambda expression.

variable (s)
Returns a Parka variable with name s, where s is a symbol. E.g., `(variable 'foo)` returns `?FOO`.

variable-name (v)
Returns the name (symbol) of a Parka variable structure v. E.g., `(variable-name '?FOO)` returns `FOO`.

variable-p (x)
Returns T if x, a Lisp objects, is a Parka variable structure. E.g., `(variable-p '?FOO)` returns T.

xval-p (*predf f*)
Returns T if f is explicitly valued for predicate *predf*. Else returns NIL.

Appendix B

Parka 3 Base Ontology

The following frames are built into Parka and are collectively, along with their corresponding slots, known as the Parka "Base Ontology". Figure B.1 shows the top-level frames. Figure B.2 shows the metadata frames of the base ontology: i.e., those frames used to represent properties of other frames that are used by Parka.

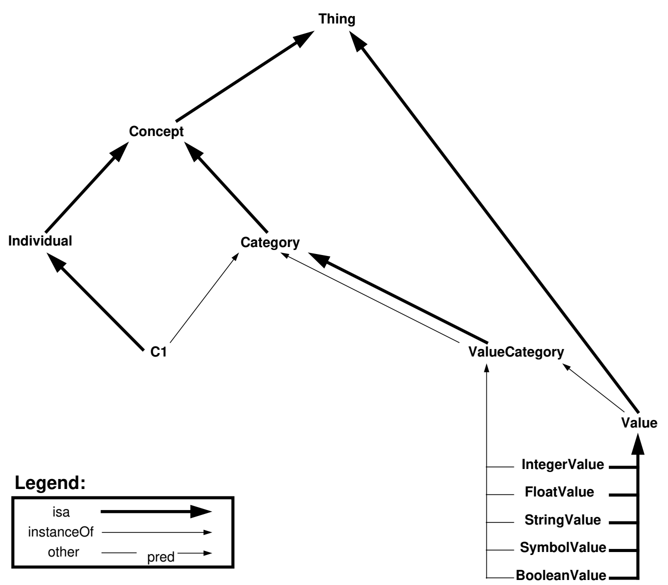

Figure B.1: Top-level frames of Parka Base Ontology

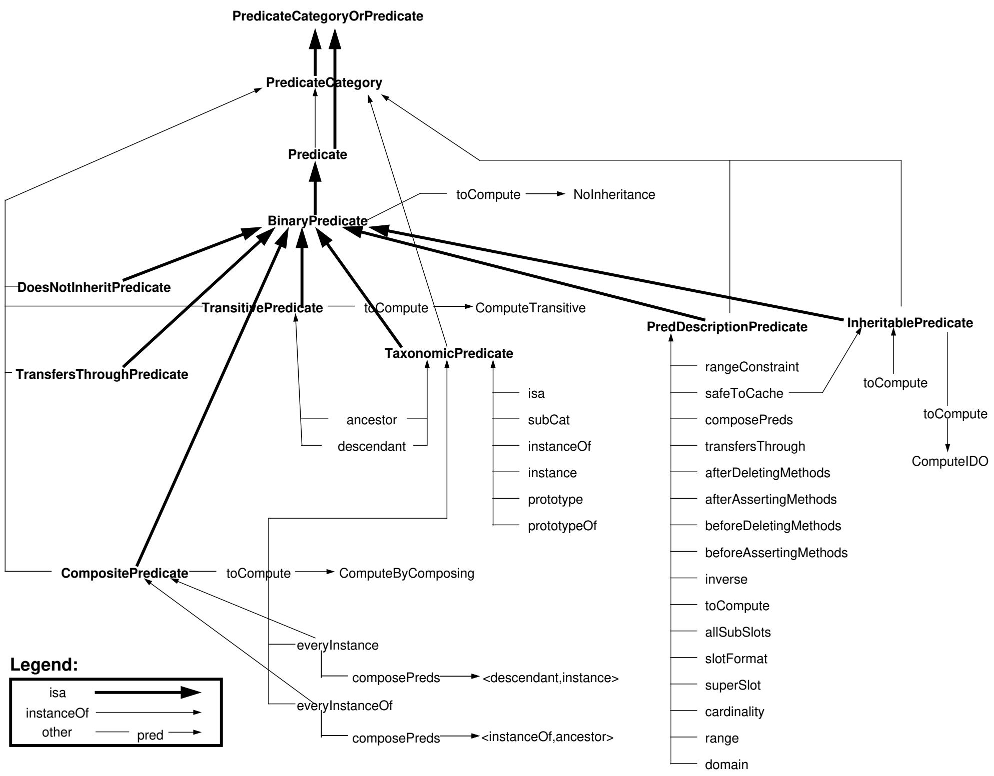

Figure B.2: Metadata frames of Parka Base Ontology

Appendix C

Differences from Parka Version 2

C.1 New Parka Features in Version 3

- Predicates are now first-class objects and are represented by frames. They have an associated domain and range (Section 2.3) and optional update methods (Section 2.5.1).
- Slots for a predicate can have inverses and superslots (sections 2.3.4 and 2.3.5, respectively).
- Slots can have values that are inferred via transitivity or that transfer-through from another frame (sections 2.6.2 and 2.6.3, respectively).
- Slots can have multiple values (fillers) (Section 2.4.1).
- Slots can have set or vector values (sections 2.4.1 and 2.4.1, respectively).
- *#!instanceOf* links related category members (instances) to categories.
- Category frames have automatically-created prototypes (Section 2.2.1).

C.2 Version 2 Features Changed/Removed

- *#!isa* links now relate a category to its supercategories. *#!instanceOf* links are used for representing category membership (previously *#!isa* links were also used for representing category membership).
- :restriction and :definitional tags for individual slots (assertions) are no longer allowed (instead see the domain/range specification for predicates in section 2.3).
- set-union and set-intersection are no longer supported.
- Aggregations are no longer supported (see Figure 2.4 for an example of how to represent partonomies in Parka 3).
- Slot (frame) names are now strings.

- Parka is now sensitive to the case of frame names (strings).
- The batch update mechanism is no longer necessary and has thus been removed.

C.3 Commands Changed

Ver. 2 Command	*Ver. 3 Command*
all-slots	local-slots, inherited-slots
ancestor	ancestor-p (instance-of-p)
ancestors	ancestors (every-instance-of)
child-p	child-p (instance-of-p)
children	children (instances)
delete-isa	retract!
descendant	descendant-p (instance-p)
descendants	descendants (every-instance)
fill-slot	assert!
frame-name	frame-to-name
frame-slot-value	query!
frames-with	query!
frames-with-all	query!
frames-with-some	query!
get-frame-slots	local-slots
isa	assert!
make-category	new-category
make-instance	new-instance
names-a-frame	name-to-frame
name– >frame	name-to-frame
parent-p	parent-p (instance-p)
parents	parents (instance-of)
structure-retrieve	query!
unfill-slot	retract!

The following Version 2 commands do not exist in Parka 3:

- individual-p
- xrest
- xrest-p
- with-batch-update

Appendix D

Sample Parka Commands

D.1 The Tweety Example

The following series of commands was used to create the KB shown in Figure 2.2.

```
(init-parka)

(new-category "PhysicalObject" #!Thing)
(new-category "Animal" #!PhysicalObject)
(new-category "Male"   #!Animal)
(new-category "Female" #!Animal)

(new-category "Bird"     #!Animal)
(new-category "Canary"   #!Bird)
(new-category "Feathers" #!PhysicalObject)

(new-category "Color"  #!Thing)
(new-instance "Yellow" #!Color)
(new-instance "Brown"  #!Color)

(new-pred "hasColor"   #!PhysicalObject #!Color #!MultiValue)
(new-pred "coveredBy"  #!PhysicalObject #!PhysicalObject)
(new-pred "numMembers" #!Category #!IntegerValue)

(new-pred "father="  #!Animal   #!Male)
(new-pred "fatherOf" #!Male     #!Animal #!MultiValue)
(assert! '(#!inverse #!fatherOf #!father=))

(new-pred "parent=" #!Animal #!Animal)
(assert!  '(#!superSlot #!father= #!parent=))

(new-pred "lastName=" #!Animal #!StringValue)
(assert!  '(#!transfersThrough #!lastName= #!father=))
```

```
(assert! '(#!coveredBy #!Bird #!Feathers))
(assert! '(#!hasColor  #!Bird #!Brown))
(assert! '(#!hasColor  #!Canary #!Yellow))
(assert! '(#!numMembers #!Canary 1000))

(new-instance "Tweety" #!Canary #!Male)
(assert! '(#!lastName= #!Tweety "Bird"))

(new-instance "Sweety" #!Canary #!Male)
(assert! '(#!father=  #!Sweety #!Tweety))
```

D.2 The Star Trek Example

The following series of commands was used to create the KB shown in Figure 2.4.

```
(init-parka)

(new-category "Group" #!Thing)
(new-category "Person" #!Thing)

;; slots member and member-Of are inverses
(new-pred "member="    #!Group #!Thing #!MultiValue)  ; member=(group,mbr*)
(new-pred "member-Of"  #!Thing #!Group #!SingleValue) ; member-Of(mbr,group)
;; transitive versions of above slots (slots tmember and tmember-Of are inverses)
(new-pred "tmember="   #!Group #!Thing #!MultiValue
          :instance-of #!TransitivePredicate)        ; tmember=(group,mbr*)
(new-pred "tmember-Of" #!Thing #!Group #!SingleValue
          :instance-of #!TransitivePredicate)        ; tmember-Of(mbr,group)

(assert! '(#!inverse    #!tmember-Of #!tmember=))
(assert! '(#!inverse    #!member-Of  #!member=))
; tmember= is a superslot of member=
(assert! '(#!superSlot  #!member=    #!tmember=))
; tmember-Of is a superslot of member-Of
(assert! '(#!superSlot  #!member-Of  #!tmember-Of))

(new-instance "Capt-Picard" #!Person)
(new-instance "Cmdr-Riker"  #!Person)
(new-instance "Mr-Data"     #!Person)
(new-instance "Dr-Crusher"  #!Person)
(new-instance "Bridge-Crew"         #!Group)
(new-instance "USS-Enterprise-Crew" #!Group)

(assert! '(#!member= #!Bridge-Crew #!Capt-Picard))
(assert! '(#!member= #!Bridge-Crew #!Cmdr-Riker))
(assert! '(#!member= #!Bridge-Crew #!Mr-Data))
```

```
(assert! '(#!member= #!USS-Enterprise-Crew #!Bridge-Crew))
(assert! '(#!member= #!USS-Enterprise-Crew #!Dr-Crusher))
```

D.3 An Example of Predicate Composition

The following series of commands was used to define a composite predicate *#!uncle*, discussed in section 2.6.4.

```
;; Make some categories to define family relations on:
(new-category "Person")

;; Define parent predicate.  Note that father and mother
;; are subslots of parent:

(new-pred "parent" #!Person #!Person
          :cardinality #!MultiValue
          :instance-of #!BinaryPredicate)

;; Define brother predicate

(new-pred "brother" #!Person #!Person
          :cardinality #!MultiValue
          :instance-of #!BinaryPredicate)

;; Define uncle predicate.  We have to make it an instance
;; of #!CompositePredicate in order to be able to compute
;; it by composition:

(new-pred "uncle" #!Person #!Person
          :cardinality #!MultiValue
          :instance-of #!CompositePredicate)

;; This is how the composition is defined.  We put the
;; (vector) value #(#!parent #!brother) on the #!composePreds
;; slot of the predicate #!uncle.

(assert! '(#!composePreds #!uncle #(#!parent #!brother)))

;; Make some people.  These happen to be in my family:

(new-instance "Tom" #!Person)
(new-instance "Dick" #!Person)
(new-instance "Harry" #!Person)

;; Assert family relations.
```

```
(assert! '(#!parent #!Tom #!Dick))
(assert! '(#!brother #!Dick #!Harry))

;; Some sample query results:

(query! '(#!uncle #!Tom ?uncle))
;; => (((?UNCLE . #!Harry))))

(query! '(#!uncle ?who #!Harry))
;; => (((?WHO . #!Tom)))
```

Bibliography

[1] William A. Andersen, James A. Hendler, Matthew P. Evett, and Brian P. Kettler. Massively parallel matching of knowledge structures. In Hiroaki Kitano and James Hendler, editors, *Massively Parallel Artificial Intelligence*, pages 52–73. AAAI Press/The MIT Press, Menlo Park, California, 1994.

[2] Matthew P. Evett, James A. Hendler, and Lee Spector. Parallel knowledge representation on the Connection Machine. *Journal of Parallel and Distributed Computing*, 22:168–184, 1994.

[3] Brian P. Kettler, James A. Hendler, William A. Andersen, and Matthew P. Evett. Massively parallel support for case-based planning. *IEEE Expert*, pages 8–14, February '1994.

[4] Lee Spector, Bill Andersen, James Hendler, Brian Kettler, Eugene Schwartzman, Cynthia Woods, and Matthew Evett. Knowledge representation in PARKA – part 2: Experiments, analysis, and enhancements. Technical report, University of Maryland at College Park, Computer Science Department, 1992.

[5] Lee Spector, James A. Hendler, and Matthew P. Evett. Knowledge representation in PARKA. Technical Report CS-TR-2410, University of Maryland at College Park, Department of Computer Science, February 1990.