Internet Fish

by
Brian A. LaMacchia

Artificial Intelligence Laboratory
and
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

Abstract

I have invented "Internet Fish," a novel class of resource-discovery tools designed to help users extract useful information from the Internet. Internet Fish (IFISH) are semi-autonomous, persistent information brokers; users deploy individual IFISH to gather and refine information related to a particular topic. An IFISH will initiate research, continue to discover new sources of information, and keep tabs on new developments in that topic. As part of the information-gathering process the user interacts with his IFISH to find out what it has learned, answer questions it has posed, and make suggestions for guidance.

Internet Fish differ from other Internet resource discovery systems in that they are persistent, personal and dynamic. As part of the information-gathering process IFISH conduct extended, long-term conversations with users as they explore. They incorporate deep structural knowledge of the organization and services of the net, and are also capable of on-the-fly reconfiguration, modification and expansion. Human users may dynamically change the IFISH in response to changes in the environment, or IFISH may initiate such changes itself. IFISH maintain internal state, including models of its own structure, behavior, information environment and its user; these models permit an IFISH to perform meta-level reasoning about its own structure.

To facilitate rapid assembly of particular IFISH I have created the Internet Fish Construction Kit. This system provides enabling technology for the entire class of Internet Fish tools; it facilitates both creation of new IFISH as well as additions of new capabilities to existing ones. The Construction Kit includes a collection of encapsulated heuristic knowledge modules that may be combined in mix-and-match fashion to create a particular IFISH; interfaces to new services written with the Construction Kit may be immediately added to "live" IFISH.

Using the Construction Kit I have created a demonstration IFISH specialized for finding World-Wide Web documents related to a given group of documents. This "Finder" IFISH includes heuristics that describe how to interact with the Web in general, explain how to take advantage of various public indexes and classification schemes, and provide a method for discovering similarity relationships among documents.

Thesis Supervisor: Gerald J. Sussman
Matsushita Professor of Electrical Engineering

This report is a revised version of a thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology in May, 1996.

Acknowledgments

This thesis could not have been completed without the support of a great many people. I wish to take this opportunity to express my appreciation for their help in bringing this thesis to a successful conclusion.

First, my sincerest thanks to those whose work became part of the Internet Fish. **Steven Adams** wrote the Scheme code to support s-expression HTML. **Michael ("Ziggy") Blair** provided the demonstration problem for the Finder IFish. The Architext index and search engine (now called "Excite for Web servers") was provided courtesy of **Excite, Inc**. The HTTP proxy server code was written by **CERN**.

I am deeply grateful to all my friends at **Silverglate and Good** for their overwhelming generosity, which permitted me to work from home this past year and write this thesis. Thanks especially to **Harvey Silverglate**, **Andy Good**, **Sandie Fennell**, **Dana Gurwitch**, and **Gia Barresi**.

For the past year I have been fortunate in having a retreat up Massachusetts Ave. to which I could escape when I needed to do something other than thesis. I thank **Professors Arthur Miller** and **Charles Nesson** of **Harvard Law School** for graciously allowing me to audit **Copyright** during the Fall term of 1995 and **Law, Internet and Society** during the Spring term of 1996. My thanks also to the students in both classes who helped make those classes the most fun I've had in a classroom since I entered graduate school at MIT.

My legal education began at **Silverglate and Good** and **Zalkind, Rodriguez, Lunt and Duncan**; in addition to those already mentioned above I'd like to thank **Sharon Beckman**, **Phil Cormier**, **Jason Gull** and **Daffodil Tyminski** of S&G and **David Duncan** of ZRL&D for their help leading me through the finer points of the law.

When I wasn't working on IFish or reading case law, I was helping others overcome the wonders of modern hardware software. Thanks much to all the clients of LaMacchia Computer Consulting: **Tricia Prevett** at **KHJ Integrated Marketing** (formerly **KelleyHabibJohn Marketing and Advertising**), **Hearst New Media**, **Terry Ehling** at **The MIT Press**, **Errol Morris** at **Fourth Floor Productions**, **Jay Lupica** at **Buyers Advantage**, and **John Habib** at **Alexander Mortgage**.

For the past four years I have been fortunate to be part of the local fundraising efforts of **St. Jude Children's Research Hospital**. Thanks to all the people involving in making **TomorrowNite '93-'96** happen, especially **Paul and Jane Ayoub**, **Joe and Christa Ayoub**, and **Steven and Karen Salhaney**.

For almost ten years I have worked as a member of **Project MAC** (the Project on Mathematics and Computation) at the MIT Artificial Intelligence Laboratory. What made that group such a great place to work in the years gone by were the people who inhabited it: **Michael Blair**, **Liz Bradley**, **Mike Eisenberg**, **Arthur Gleckler**, **Philip Greenspun**, **Kleanthes Koniaris**, **Bill Rozas**, **Thanos Siapas**, **Jason Wilson** and **Henry Wu**. Together they formed a very special collection of people, one with which I was proud to have been associated.

My thanks to **Ellen Spertus**, **David LaMacchia**, **Jim Miller** and **Gerald Sussman** for providing comments on early drafts of this thesis.

For most of my graduate career I was fortunate to have been supported by an **AT&T Foundation PhD Fellowship**. My thanks to the **AT&T Foundation** and the former **AT&T Bell Laboratories** for that financial support. The majority of the work in this thesis was funded by this fellowship.

Portions of this thesis were also supported by the **Open Software Foundation Research Institute**. Portions of this technical report were supported by a **Packard Fellowship in Science and Engineering** from the **David and Lucile Packard Foundation**.

Gerald J. Sussman, Matsushita Professor of Electrical Engineering, supervised this thesis. **Harold Abelson**, Class of 1922 Professor of Computer Science and Engineering, and **James Miller**, World Wide Web Consortium and MIT Laboratory for Computer Science, served as readers on my thesis committee.

There are others whose actions contributed to the completion of this thesis at this time, in this manner. While their actions must be acknowledged, it does not seem appropriate to do so here in this place. I commend to readers interested in those stories my forthcoming book, *Defending Dave, and other tales of the 'Net*, which shines the bright light of truth and public scrutiny in a number of dark corners.

This thesis describes research conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-92-J-4097.

This one's dedicated to a lot of people.

In memory of my father's parents

Hon. Otto H. and Dinah LaMacchia

(I know the Judge approves of what I have done)

and

In honor of my mother's parents

Gerald and Leona Tigar

(who have been my home away from home here in Boston)

For my parents,

Robert and Sherry LaMacchia

who have always given of themselves so that I
might have the best possible education, whatever the price

For all my friends who stood by me in the darkness,
keeping alive the flickering flame of hope:

Ziggy, who taught me how to argue,
Arthur, who opened my eyes to life outside the lab,
Russ, who taught me how to keep score in a ballgame,
Liz, who taught me how to climb mountains, real and metaphorical,
Retta, who always had an ear, or advice, or just a shoulder to cry on,
Philip, who showed me Lexis, Westlaw, and the path to Harvard Law School,
Bill, who taught me to stand by my principles and beliefs, no matter the cost,

and, most especially,

Henry,
my mentor, office mate, friend, confidant and drinking buddy,
who taught me how to appreciate wine, route network cable,
design circuits, give of myself to charity, and hack,
be it code, restaurants or hotels,

but, most of all,

For Dave,

who had to endure what no man should ever have to endure,
and in doing so showed a depth and strength of
character, conviction and shear will
that I can only hope to equal.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Internet: Evolution in Action

Take any current newspaper and lay it side-by-side with an issue of the same paper two or three years old. Now compare the two papers: of course the names and faces and places "in the news" will be different, but look beyond that. Look at the "mail addresses" that appear in today's paper, like "`losers@wpost.com`[1]." See the strange lines of characters in the automobile ads (`http://www.honda.com/`) or in the movie ads (`http://www.mca.com/othello`). Look through the classified ads at the number of people offering "Web services" or "unlimited Internet Access for only $15/month." Pick up a copy of Newsweek (which declared 1995 "The Year of the Internet") and turn to the weekly feature called "Cyberscope." Thirty-six months ago any mention of the Internet in the popular press was rare; now not a day goes by without an Internet news story.

There is no denying the hype surrounding the Internet; one need only look the initial public offerings (IPOs) and subsequent stock price fluctuations of Internet-related companies like Netscape Communications or Yahoo for a tangible indication of the future expectations placed on the medium. Earlier this year we celebrated the 25th anniversary of the "founding" of the ARPAnet, and yet over all those years the continued connections of widely-separated computation sources and data storage repositories barely received notice. Whereas before this recent explosion the types of information available on-line via the Internet were severely limited, now the first place to look for information of any type is "the net." Over the last 36 months we have seen tremendous growth in both the quantity and variety of information being published on the Internet. More people are acquiring access to the Internet (either directly or via an Internet Service Provider (ISP)), more people are learning how to access remote sources of information, and more people are putting their own data "on the net" for public consumption. If you want to know today's baseball schedule (assuming the players are not on strike), share prices for mutual funds, the latest weather reports and satellite images, or even the Latin name for the sticky wattle plant, you can obtain that information from sources on the Internet.

The problem facing Internet users today is not whether relevant information is available on-line (it most likely is), or whether it is possible to gain access to the relevant information repositories. Today's problem is in *finding the repositories* that have the desired information. Recent distributed

[1] The e-mail address for the Washington Post's weekly Style Invitational contest.

information systems have made publishing information easy, but finding a single piece of desired data in the Internet's sea of information is like trying to find a needle in 10,000 haystacks. *Resource discovery* is the process by which customers find sources for the data they desire. As the amount of Internet-accessible data continues to grow, there is an increasing need for tools to discover, organize and categorize new sources of information.

The resource discovery problem is not new; research on resource discovery dates back thirty years or more [52]. What *have* changed are some of the fundamental assumptions concerning the underlying pool of information that is accessible to the user. Nothing in the sea of available information may be considered static anymore. The information itself is dynamic, the repositories of information are dynamic, the primitive indexing tools available are dynamic, and even the set of available tools is dynamic as new services aimed at helping people find the information they desire announce themselves to the world constantly. The books published that purport to contain "catalogs" of the Internet or "yellow pages-like" listings are dated as soon as they are committed to paper. Even the well-maintained on-line categorical listings of data repositories are having trouble maintaining themselves in the face of exponential growth[2].

Resource discovery is generally interpreted to refer to the problem of finding a network-accessible resource that contains some piece of desired information. However, there is another angle to the same problem: maintaining watch over a growing stream of information. In this case, finding related information resources is not the problem (the resources are already known); the problem is that the rate at which new information is added to that resource is growing and thus more effort is required to maintain the same level of awareness with respect to the information resource. (Anyone who has tried to remain current in a Usenet newsgroup as that newsgroup undergoes an explosion of traffic has experienced this problem.) Maintaining a fixed level of awareness with respect to an information source as that source grows is a variant of the resource discovery problem.

The majority of Internet resource discovery tools to date have been based on the idea of providing large, monolithic[3] servers that hold indexes of available information. The Archie [18], DejaNews [14], Veronica [23], Excite [20], and WebCrawler [47] services are all representative examples of monolithic servers[4]; there are of course many other similar systems [15, 28, 29, 35, 36, 39, 59]. The source and type of information indexed varies by service: Archie and Veronica, for example, index only filenames or header lines. Other services provide summary-based indexing [36, 47] or full-text indexing [15, 31].

While these broad-based indexes often provide useful starting points for further resource discovery, they themselves are not sufficient tools for dealing with the growing collection of Internet-accessible data. They often suffer from the "too little/too much" problem: for a given query, the index in question returns either too few resources (thus frustrating the user) or too many resources (which the user cannot deal with in a reasonable manner)[5] There is also no notion of "ongoing

[2]The Yahoo [59] service is now reporting over 5000 new addition requests every week. A project that was started by a couple of graduate students providing a useful service in their spare time has blossomed into a full-blown commercial enterprise.

[3]"Monolithic" here refers to the logical representation of the server to the user. The fact that a server may be fully-replicated across multiple machines for performance reasons doesn't change the basic model.

[4]Recent work on the Harvest [9] system has extended the "monolithic server" approach to a distributed, multi-level indexing scheme.

[5]For example, a simple query to Digital's Alta Vista service [15] on the term "cryptography" returns a list of "documents 1-10 of about 20000 matching some of the query terms, best matches first." Having access to 20000 possible articles is great, but it's not practical for the user to page through the results ten at a time to find the

queries" with these systems; an incoming query is processed, answered and dismissed. The index itself is a representation of some fixed set of data at some fixed point in time. In order to detect new information resources as they appear, the index itself must be updated on a continuing basis and users must query the index periodically and look for results not found by prior queries.

This thesis describes the design, implementation and deployment of a system for constructing a different type of resource discovery tool, one that tailors itself to finding resources that contain information relating to a particular topic and perhaps maintaining that collection of resources over an indefinite period of time. The tool runs autonomously, although both it and the user that creates it are able to initiate discourse when desired. We call this tool an "Internet Fish" (IFish) because it "swims" in the "sea of information" that is the Internet looking for "tasty bits of information" that relate to its topic of interest. Internet Fish both find new sources of information and also monitor known information streams for new, interesting data.

The remainder of this chapter lays the background for our discussion of the Internet Fish. Section 1.2 below briefly outlines past and current work in the field of resource discovery on the Internet, including the myriad of currently-popular index services available on the World Wide Web (Section 1.2.3). In Section 1.3 we discuss how these current systems fail to make use of encapsulated heuristic knowledge (Section 1.3.1) and long-term conversations (Section 1.3.2). Section 1.4 outlines the structure of the remainder of the thesis.

1.2 Resource Discovery on the Internet

Research on resource discovery techniques has grown concurrently with widespread publication of information on the Internet. This is a pure case of necessity being the mother of invention; as the amount of information available has increased it has outstripped the ability of individuals to manually "keep up" and organize the information. Thus, as is natural, automated methods of indexing available information were developed. This section briefly outlines the more well-known methods of publishing information on the Internet and their associated indexes.

1.2.1 Jurassic Net—FTP and Usenet

The earliest form of data publication on the Internet is probably the anonymous FTP server [49]. Use of FTP, the File Transfer Protocol, to move files across the network has always been one of the primary uses of the ARPAnet (and later the Internet). Historically FTP traffic accounted for over 40% of the total byte traffic on the NSFNET backbone. Even as late as April, 1995, when WWW byte traffic exceeded FTP traffic, FTP traffic still accounted for over 20% of the use of the network[6]. Any machine that can act as an FTP daemon has the capability to serve files anonymously; that is, to allow unlimited access to portions of its filesystem so that anyone may access and download

particular resources that help him or her best. If the user can provide a more refined query (e.g. "cryptography AND knapsack") the search results may become tractable, but this is not always possible. In particular, if the user performing the query doesn't already know what he or she wants to find, it may be impossible to narrow the search to produce a reasonable number of documents.

[6] These statistics were collected by MERIT, Inc., as part of their operation of the NSFNET backbone network. The NSFNET ceased to exist in May, 1995, when government subsidies for the backbone ended; traffic that was carried on the NSFNET backbone has since migrated to multiple parallel networks operated by private companies.

the data stored there. This capability has long been used to maintain data archives on the Internet that are freely accessible.

Given the name of a machine that allows anonymous FTP access and the name of a particular file on that server, it is easy to gain access to the server and download the specified file. However, when one does not have both of these two pieces of information the task of finding the desired data is significantly more complicated. The FTP protocol itself does not provide methods for searching the contents of remote servers, and there is no global registry of machines running anonymous FTP. The Archie [18] system connects to each of a fixed list of anonymous FTP servers and recursively extracts the server's directory structure. Archie then indexes the absolute pathnames of all the files on the server in question. The index is replicated across multiple, distributed servers and is available via a number of interfaces. Archie does not index the *contents* of files available via anonymous FTP; only the names of files are indexed.

The other major data publishing system that has long been available on the Internet is the collection of Usenet newsgroups, often collectively called "netnews." Today, there are literally thousands of individual Usenet newsgroups[7]; the number of newsgroups available to any particular user is a function of what newsgroups are received and stored by the user's netnews server machine. Whereas FTP servers are used almost exclusively for the distribution of static files (data, programs, etc.), netnews is used for the most part to distribute communication (messages among groups of users).

Usenet itself provides for only limited archival storage of posted messages; the amount of time any particular message "lives" on a news server depends in part on how long that news server holds netnews messages[8]. Typically messages are archived for 7-30 days, enough time for interested users to read messages and maintain some sense of continuity in threads of discussion, but short enough to limit required disk space. Permanent archives and indexing mechanisms are provided by others on an as-needed (or as-desired) basis. For example, the `sci.math` newsgroup is archived at URL[9]:

`gopher://math.lfc.edu/11/MathRelItems/scimathArchive`

The archive retains every message and automatically builds a WAIS [31] index of message contents. The `usenet-address` archive [46] maintains a list of recently-seen e-mail addresses by extracting addresses from the stream of netnews messages. Many "Frequently Asked Questions" lists (FAQs) are "archived" by automatically reposting the contents to netnews periodically; the reposting tasks are handled by automated servers. Some netnews FAQs are also archived via anonymous FTP at specific sites around the Internet; this is a special case of the more recent phenomenon of fully-indexed netnews servers[10]. Usenet newsgroups may also be gatewayed with particular e-mail mailing lists; the gateway provides bidirectional exchange of postings and articles so that newsgroup and mailing list see the same collection of writings. Many mailing lists are archived automatically, so Usenet postings may eventually end up on an FTP server somewhere well after the article has

[7] Currently my own `.newsrc` file, which records what newsgroups I subscribe to and what articles I have read in each newsgroup) has over 10750 newsgroups in it to which I could subscribe.

[8] Netnews server software allows server administrators to decide when articles in a particular newsgroup should expire and be removed from the system. Usually the higher the volume of traffic in a particular newsgroup the shorted the time period articles are left on the server.

[9] Throughout this document we reference documents published on the Internet with Uniform Resource Locators [6] (URLs).

[10] Currently both Alta Vista [15] and DejaNews [14] provide fully-indexed access to their collection of newsgroups. These services do expire articles over time, but provide approximately 60-90 days worth of access in some newsgroups.

disappeared from news servers. Or, since some mailing lists and individual newsgroups may also be gatewayed through hypertext-generating servers, content originating in netnews may also be available via the World Wide Web (see Section 1.2.3 below).

1.2.2 Gopher and other Campus-Wide Information Systems (CWIS)

The Gopher [40] system, introduced in 1992, quickly became popular as a publication system because it simplified access to a web of distributed information. Gopher presents the user with a hierarchy of menus; leaf nodes in the graph may contain text, graphics or sound. Once a Gopher server is set up on a machine, publishing new data in "Gopherspace" is relatively easy, as is providing local links to remote data sources. Gopher also provides a limited search/index mechanism for *user-published indexes*; that is, an indexed database can be added to Gopherspace and simple keyword searches may be performed on that database via the Gopher interface. Gopher itself does not provide any sort of indexing of server contents, although individual server operators may index the contents of their servers manually (using something like WAIS) and publish those indexes via the Gopher search/index interface.

The Veronica service [23] provides title-based indexing of the contents of Gopherspace. Veronica walks through all the Gopher servers accessible from the Univ. of Minnesota "Mother Gopher" server compiling a list of all accessible documents and directories. Titles are extracted from each visited document during the tree walk and collected. This collection is then sorted, indexed and distributed to the various Veronica query servers on the Internet. Users are then able to access the Veronica index via limited boolean keyword searches.

Gopher was one of a number of early campus-wide information systems (CWIS). The TechInfo system [38], developed at MIT, was another early attempt at providing distributed public information. Like Gopher, TechInfo was a hierarchical collection of files where leaf nodes could contain text, graphic, or sound files. Unlike Gopher, TechInfo did not have a centralized server answering queries. Instead, TechInfo distributed a description of the hierarchy of information nodes via and underlying distributed filesystem (AFS [42, 57], the Andrew File System) to provide distributed access to information. TechInfo clients only needed access to this distributed database in order to build a menu of possible choices for the user. When a leaf node was requested the information in the distributed database told the TechInfo client where it could find the desired file within the larger AFS system. TechInfo was first subsumed into Gopherspace via a Gopher-to-TechInfo gateway that ran as part of MIT's early Gopher service. Later, a gateway from HTTP to TechInfo was deployed, thus giving every WWW browser access to TechInfo information.

1.2.3 The World Wide Web

The introduction of the World Wide Web (WWW or "the Web" [3]) in 1993, and in particular the Mosaic [1] family of WWW browsers, has been primarily responsible for the phenomenal growth in Internet publishing seen today. Mosaic/WWW is often quoted in the popular press as the Internet's first "killer app[lication]," akin to the role Visicalc played in the popularity of the Apple II microcomputer during the early 1980s. Mosaic's "point-and-click" interface to the constantly-growing hypertext that is the WWW, together with its ability to seamlessly intermix text, graphics and sound, has drawn many new people onto the Internet. The WWW architecture and the popular

HTTP servers available for free from W3C[11] and NCSA make it particularly easy for users to author information and publish it on the Web.

It is difficult at best to measure the size of the WWW. New WWW servers are constantly being added to the network and new documents are constantly being added to existing servers; similarly old servers and documents may no longer be available. Some Web documents are interfaces to underlying databases; the complete content may be accessible from the Web but not in a fashion that makes indexing possible. Finally, many Web browsers support multiple protocols: Netscape 2.0, for example, supports HTTP, Gopher, FTP, Usenet (through a user-customizable particular netnews server), and e-mail (via SMTP [48] and POP [43]). Thus, when one publishes data via Gopher, it is available not only to people using Gopher-specific clients but also to anyone running a WWW client that understands the Gopher protocol. Because these browsers provide effectively seamless access to a wide range of information, any meaningful measure of the size of the WWW must be qualified and state which particular definitions of "size" and "WWW" they depend upon.

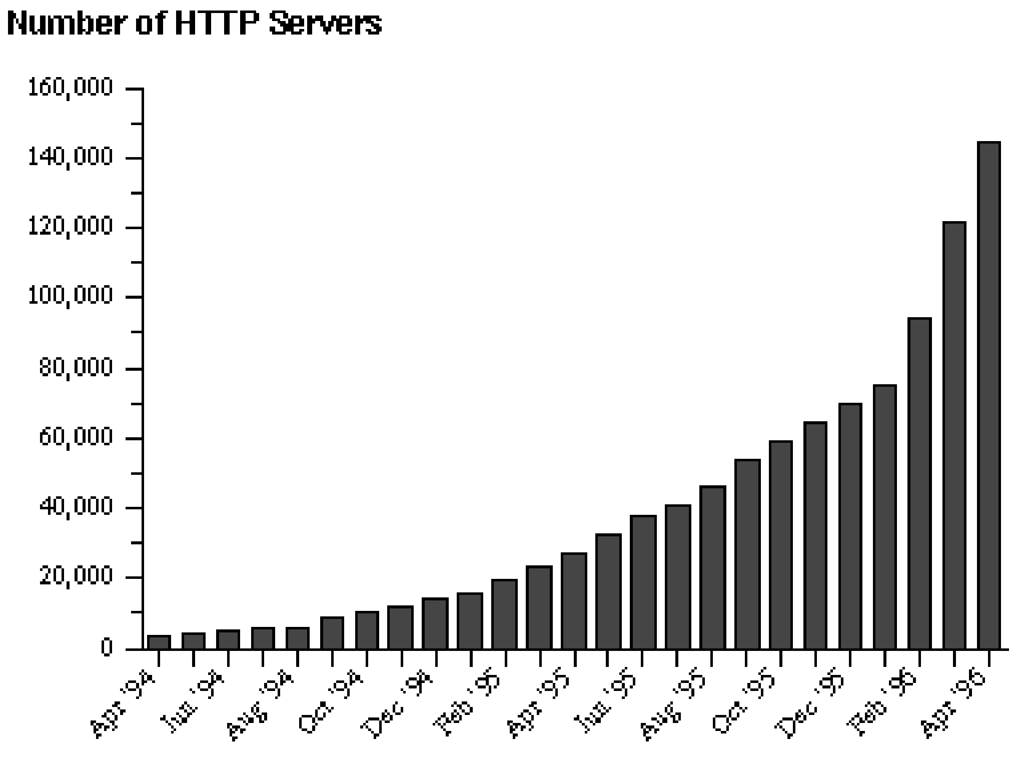

Figure 1-1: Growth of the World Wide Web, as measured by WebCrawler [47].

There have been and continue to be numerous attempts to "walk the WWW" and count available WWW servers, documents and even words; our best statistics to date come from the various Web indexing services that attempt to exhaustively traverse the Web gathering documents. Figure 1-1 shows the growth of the WWW as measured by the number of accessible HTTP servers discovered over time by WebCrawler[12]. Claims of being the "most comprehensive" index have lead

[11] The World Wide Web Consortium.

[12] The WebCrawler-derived data necessarily undercounts the true size of the Web as it only represents servers that

to a rivalry and informal competition among the various indexing projects that continues to this day. For a while Lycos [36] was the leading index service, registering hundreds of thousands of accesses each day; Lycos first identified over one million separate Web documents in November, 1994. The WebCrawler service [47] (acquired by America Online in June, 1995) was the leading competitor until December, 1995, when Digital's Alta Vista [15] service went on-line. Alta Vista began service with a index covering over ten million individual Web documents and over eight billion individual words. Between December, 1995, and February, 1996, Alta Vista indexed another eleven million web pages, bringing the total number of indexed documents to twenty-one million. Lycos has also crossed the "ten million" barrier and continues to search out new Web servers and new documents.

This tremendous growth in the amount of data published via WWW has led to a number of attempts to index the Web's contents. Initially these attempts proceeded along the lines of the Veronica project in Gopherspace: they attempted to collect title-like information about every reachable page of data on the WWW and build boolean keyword searches into that database. The JumpStation [22], probably the first well-known attempt to index any WWW information, collected only "`<TITLE>`"-tagged information from pages it encountered. (The WWW Wanderer [26] was a similar early project, although its goal was only to discover available WWW servers. It only collected information about machines that were running WWW servers.) Later projects like the World Wide Web Worm (WWWW) [39], the WebCrawler and Lycos, collected more summary information to index from each WWW page. For example, Lycos tries to summarize the actual content of documents in addition to collecting heading text:

> For each document fetched, Lycos keeps the title, headings, subheadings, and links, plus the 100 highest weighted words (using Tf*IDf[13] weighting) plus the first 20 lines. [36]

The Lycos approach thus takes advantage of human-tagged information (headings), often-appearing keywords and "introductory" text that generally appears at the beginning of a file.

As mentioned above in Section 1.1, the shear amount of accessible and indexed information often leads to the "too much/too little" problem. The usefulness of being able to search many millions of documents for particular keywords depends in part on the number of documents found. Keyword searches that return hundreds of thousands of "hits" (document matches) are not particularly useful, especially if the documents are "sorted in no particular order[14]." These drawbacks led to the creation of "moderated" Web site listing services and classification schemes, the most successful to date being the Yahoo [59] indexing service. The graduate students who founded Yahoo[15] created a broad, hierarchical classification scheme for Web sites based on subject matter; the classification scheme is similar to those used by the Library of Congress (for classifying books and periodicals) and the American Mathematical Society's Math Reviews publications (which classify

---

have been discovered by the indexing robot. The data also counts only HTTP servers and does not include FTP or Gopher servers.

[13](Term frequency) * (Inverse Document frequency) weighting is described in [52].

[14]Simple keyword searches via Alta Vista may not be sorted; complicated searches may be able to use a scoring algorithm to sort the search results. Lycos and WebCrawler both return documents according to internal scoring rules.

[15]Yahoo was started by David Filo and Jerry Yang, graduate students in Electrical Engineering at Stanford, as a free classification service, similar to a Yellow Pages listing. Yahoo became so successful that it was spun off as a start-up company. To date Yahoo's revenue stream is generated solely by advertising; small advertisements are placed in each page of search results returned. Many other free Internet services are supported in similar fashion.

scholarly mathematical publications by topic). The various Yahoo headings, the actual text of referenced URLs, and even a small description of the contents of the Web site may all be searched by keyword. Yahoo's success spawned multiple similar services: Point Communications' Point [35] service (now allied with Lycos), and the McKinley Group's Magellan index [56] are two of the largest competitors. Point and Magellan both provide actual reviews and numerical ratings of Web sites. The proliferation of these various indexing and search services, each with its own domain of coverage, has led to the creation of "meta-search" services like MetaCrawler [54], SavvySearch [17] and WebCompass [50]. These services use the monolithic indexing services as subroutines: they query each of the indexing services in parallel, aggregate the results and return the composite to the user who made the query.

Maintaining the quality of a classification scheme like Yahoo's is time consuming; entries must be checked, updated, and revised periodically. Yahoo allows people to submit pointers to new Web sites to be included in the hierarchy, but each submission must still be validated by hand for relevance and appropriateness. Reviews (like those provided by Point) and ratings systems (like that used by Magellan) must also be revised periodically or they will cease to be useful. Some of this work may be pushed back onto the operators of the Web servers themselves; after all, a site wishing to be listed in one of these classification services will gladly provide the (relatively minimal) effort of maintaining their own site's information, and the Web makes this sort of distributed collaboration easy[16]. IBM's Aquí database [29] of Web pages allows individual users to link related pages together; these links are bidirectional and may be found at either the referencing or referenced page. Thus, even if the original Web document does not contain a particular pointer such a link may be added to Aquí and will be displayed when the page is retrieved via the Aquí system.

Finally, we should point out that as the capabilities of these various indexing services increased, other rapidly-changing information streams were able to be indexed, too. A large subset of the available Usenet newsgroups are now fully indexed by both Alta Vista and DejaNews [14]. DejaNews in particular provides access to approximately the past three months worth of netnews postings. It is possible to search the collected database by subject keyword, by title keyword, or by author (either name or e-mail address). Author searches yield not only the individual articles written by someone but also a summary of articles by newsgroup. Articles that contain Usenet referral headers are hyperlinked to the referenced postings, thus making it easy to climb backwards along a thread of discussion.

1.2.4 Indexing Local Filesystems

There has also been much work recently in providing advanced indexing services for local filesystems. The Essence [27] system provides semantic file indexing for a wide variety of file types. Essence uses heuristics (such as filename suffixes or "magic numbers" in Unix binary files) to determine the type of information contained in a particular file. Content summaries are then produced for each file; content summary generation depends on the type of the file. Summaries are then indexed using a modified version of WAIS to provide fast searches over the collection of summaries.

The Semantic File System (SFS) [24] is another indexing system that uses semantic information

[16]One of the best examples of harnessing the collective power of the Web to produce collaborative work is the Internet Movie Database, available at `http://www.msstate.edu/Movies`. All the information in the database was submitted by volunteers; the resulting collection is perhaps the most comprehensive set of movie-related facts and trivia ever published.

to classify files in a local filesystem. SFS uses *transducers* (filters) to convert files into attribute-value pairs, which are then used as the basis of classifying files in virtual file systems. Virtual directory names in SFS are interpreted by the system as index queries; the content of a virtual directory is the set of files that matches the query. Gifford's Content Router [55], a query system for distributed information servers, is built on top of the SFS. Queries in the Content Router system describe desired object attributes. The contents of individual information servers are described by *content labels*; these labels are then registered with *content routers* that receive user queries. Content routers compare an incoming user query to the set of known content labels and forward the query to appropriate information servers. Virtual directories generated by individual information servers are then merged into one view presented to the user.

The GLIMPSE [37] system provides another approach to indexing local filesystems. GLIMPSE facilitates fast searching through the use of a very small, approximate inverted index. GLIMPSE divides a collection of documents into 256 blocks[17] and builds an inverted index listing, for each search term, the blocks in which that term occurs. Linear search is then performed over the indicated blocks to find the exact location of the search term in the filesystem. GLIMPSE is built on top of *agrep* [58], an *approximate regular expression* search tool for individual files; no semantic analysis is performed when building the GLIMPSE index.

1.2.5 Client-side Approaches

As mentioned above, most of the resource discovery tools available to Internet data publishers are large, monolithic systems that "robotically" gather data from the network. One notable exception to this rule is the WWW browser-based "fish search" work by De Bra and Post [13]. De Bra and Post modify a Mosaic client to perform limited WWW searches starting from a user-designated root document. "Fish search" is thus basically a depth- and breadth-limited tree walk from the given root, except that the depth and breadth limits dynamically change over time based on the relevance of documents "in the hierarchy" already retrieved.

The Letizia system [32, 33] is another client-based tool that assists resource discovery. Letizia is tightly coupled with a particular Web browser and constantly monitors how the user makes use of the browser. Letizia attempts to learn about user preferences and interests by watching and recording which links the user chooses on each page. Using this information Letizia tries to predict which links will be chosen next by the user and prefetches that information while the user is reading the contents of the current page.

Web browsers themselves are growing in ways that make them more useful for long-term resource discovery and maintenance. The "hotlinks" feature of Mosaic ("bookmarks" in Netscape) is a user-created list of URLs; these links are easily accessible within the browser, typically from a pull-down menu. Netscape 2.0 clients keep track of when the user last visited the documents references by each bookmark and is able to automatically visit these sites to check for changed content. Thus, a user interested in watching over a hundred individual Web pages may quickly discover which pages have recently changed and require review[18].

[17] In standard filesystem parlance a *block* is often a subdivision of an individual file. A block in GLIMPSE most likely contains many files, as a block is a large subdivision of the entire filesystem.

[18] Knowing only that a Web page has changed, of course, is not necessarily sufficient, as it may be difficult to figure out *what* on the page has changed. The AT&T Internet Difference Engine [16] provides one solution to this problem.

The Letizia and "fish search" tools both interface with particular browsers via a client-control interface that provides limited access to the browser. Browsers are becoming more extendable, and this development should produce more client-side resource discovery tools. For example, the Netscape 2.0 browser now supports Java [2] applets (mini-applications) on certain platforms. Java applets are downloaded from Web servers as part of particular Web documents (like in-lined images) but these applets actually perform computations within the client browser and display information locally. Netscape also supports "JavaScript" which is a simplified version of Java but does allow access to the network from within Netscape. (Java itself was designed to be a "safe" language and as such the Java virtual machine is not supposed to allow Java applets access to external resources like disk drives, serial ports or network connections.) These extension languages have already been used to display running stock ticket quotations (user-configurable, of course); we will certainly see more complex applications in the near future.

Finally, we should point out that not all client-side tools require modification of or even close contact with a Web browser. The OreO[19] [10] development kit, written by this author and others at the Open Software Foundations Research Institute (OSF RI), provides a simple mechanism for creating client-side applications, interposed between a browser and a network connection, that watch and possibly mutate data flow between the browser and the outside world. Using OreOs it is possible to provide on-the-fly annotation of retrieved Web pages: for example, we can build an OreO that watches the datastream for likely ZIP Code numbers and annotates that information with hyperlinks to Postal Service and census information concerning that area of the country. We could also use OreOs to improve system response by performing look-ahead caching of Web pages, or provide better history mechanisms for a browser, or even assist groupwide communication.

1.3 The Need for Something More – the Internet Fish

Section 1.2 above outlines a number of resource discovery systems currently operating on the Internet. Unfortunately these systems, and the straight-line improvements of them that have and will continue to follow, are not sufficient to meet the resource discovery needs of all users. In particular, there are three areas in which these large, monolithic systems are lacking:

1. Current resource discovery systems do not avail themselves of certain types of heuristic knowledge about the structure of the Internet and the data sources available on it. This information can be quite powerful and is readily available to experienced human Internet navigators.
2. Current resource discovery systems contain no notion of "long-term conversations" with users or any method of "remembering" queries over time. This information can also be quite powerful.
3. Current systems lack the ability to provide *serendipitous* resource discovery[20].

Our goal in designing and building the "Internet Fish Construction Kit" was to build the enabling technology for constructing simple resource discovery tools that could take advantage of encapsulated heuristic knowledge and long-term conversations, as well as provide perhaps serendipitous

[19] The name "OreO" was chosen because OreO-based applications look like HTTP proxy servers on all interfaces (Web browser and server) and contain a "filling" in between that does something interesting.

[20] Some services, like Alta Vista, allow users to travel to a randomly (or semi-randomly) chosen URL. Such behavior may be considered a very limited form of serendipitous resource discovery.

research discovery beyond "send me to a randomly chosen URL." This section provides an overview of each of these areas; subsequent chapters detail how the Internet Fish Construction Kit supports these goals.

1.3.1 Heuristic Knowledge

Current Internet resource discovery systems utilize little knowledge (if any at all) concerning the "structure" of the Internet and the information published therein. For example, within the community of U.S. university mathematics departments it is common practice to name servers accessible to non-local users via the convention `math.<school>.edu`; this is one of many naming conventions on the Internet that often provide clues to human searchers. Similarly, were I trying to locate a web server for the University of California at Berkeley, the first machine I would try to contact would be `www.berkeley.edu`. From experience, I know that `berkeley.edu` is UCB's reserved Internet domain and that it is conventional to name the Web server for a particular domain `www.<domain>`. Schwartz's Netfind [53] program for discovering electronic mail addresses made significant use of this type of information to figure out what SMTP (e-mail) servers to talk to. Knowledge about naming conventions, the structure of the Domain Naming System (DNS) [41] and specific Internet protocols (finger [60] and SMTP) was encapsulated within the system as heuristic rules.

In addition to "structural" information it is also desirable to be able to encode heuristic information about certain existing services on the Internet. Internet Fish, for example, should be able to take advantage of other known indexing services and be able to interpret search results. On-line resources like dictionaries, thesauri, geographic nameservers, and ZIP Code services are available for ready reference whenever need arises; IFish need to be aware that such services exist and that they may draw upon them as they desire. We also want the ability to capture representation knowledge for retrieved documents; IFish need to understand what might be "interesting" in an HTML document[21] or what a URL looks like in an e-mail message. A dynamic system for storing heuristic knowledge about the network is necessary if we want IFish to be able to learn about their environment.

To this end, one of two main design goals for the Internet Fish Construction Kit is to facilitate the addition of heuristic information and integration of that information into the overall search engine. As the set of accessible Internet services useful to an Internet Fish changes over time, our heuristic knowledge system must be capable of on-the-fly modification and expansion. Furthermore, we wish to keep the representation model of heuristic information as simple as possible, since eventually heuristic content is going to be provided by a number of sources.

1.3.2 Long-Term Conversations

Another area of weakness in current Internet resource discovery systems is the lack of long-term memory and any notion of conversation between indexing services and their users. Current servers view each query made of them as an independent event; they assume that there is no linkage or relationship between any two queries. This assumption allows servers with large client bases to efficiently process the many unrelated queries they receive, but it throws away information that

[21] Recall from Section 1.2.3 above that Lycos explicitly indexes all title, heading and subheading terms. Lycos assumes that words and phrases appearing in these contexts are have special meaning because the author of the document chose them to represent a block of text.

might be useful to an individual user trying to find items through multiple queries of the same service.

The usefulness of interaction between the resource discovery tool and the user is not in doubt; much of the research in traditional information retrieval (IR) methods has focussed on relevance feedback from the user to constrain the search. For example, the Content Router [55] contains an extremely simple level of interaction: query refinement. After submitting initial queries to the server, users have the option of trying to "refine" those queries by making them more specific. The server analyzes the index terms in the user's query, looks at what other terms co-occur with those terms in the index, and suggests additional refinement terms to the user based on conditional probability of co-occurrence. This very simple form of data-dependent interaction is one example of a "short-term" conversation between a user and an index server.

Internet Fish will use conversations in a much more significant manner than simple query refinement. Long-term conversations between an IFISH and its creator will allow higher-level "interest refinement" but also re-allocation of resources and human assistance in the process of acquiring knowledge. Since Internet Fish will exist over long periods of time they have the opportunity to collect more information and perform more analysis on their particular topic of interest than more general indexing services. Further, because IFISH will gather information gradually, over time we can expect IFISH to make deductions about the field of interest and request additional information or confirmation from users as appropriate. Finally, as the number of information sources discovered increases, scarce IFISH resources will be in greater demand; interaction will allow users to direct how IFISH resources should be utilized.

What might one of these conversations look like? Below are two examples of human-Internet Fish interactions which demonstrate how Internet Fish could take advantage of both encapsulated heuristic knowledge and long-term conversations. Note that we assume the existence of a natural language interface between the user and the IFISH in these examples; the current Construction Kit permits only limited, structured interaction.

User: *System, create a new Internet Fish for me.*

System: Internet Fish "Fish17" created, initialized and ready for use.

User: *Fish17, I'm interested in finding information about a particular person. The person's name is "L. Craig Evans." He's a mathematician.*

Fish17: OK, this may take a little while...

[Time passes...]

Fish17: You told me that "L. Craig Evans" is a mathematician. Based on that information I decided to go look at the membership of the "American Mathematical Society." I found a "Lawrence Craig Evans" listed. Is this the person you're interested in?

User: *Maybe. How many other "Evans" did you find there?*

Fish17: 17 other "Evans" but none approximated "L. Craig."

User: *What have you learned so far about "Lawrence Craig Evans?"*

Fish17: There's an e-mail address given: evans@math.berkeley.edu

User: *OK, assume with high probability that he's the right guy. Make searching for other possibilities a low-priority task.*

Fish17: OK.

[Time passes...]

User: *So, what do you know now about L. Craig Evans?*

Fish17: Well, he's definitely at Berkeley. I found addresses for him at both the American Mathematical Society and in an on-line phone book at Berkeley. Mailing addresses are slightly different, although the e-mail addresses and phone numbers are identical. Here's what I got from the AMS:

Lawrence Craig Evans
Department of Mathematics
University of California Berkeley
Berkeley, CA 94720-0001
510-643-7417
evans@math.berkeley.edu

and here's what I got from Berkeley:

EVANS, Lawrence C.
Professor, Math
907 Evans Hall
Berkeley, CA 94720-3840
(510) 643-7417
EVANS@MATH.Berkeley.EDU

Judging by the MR classifications of his research papers, Evans seems to do a lot of research in the area, "Partial differential equations." In the past 5 years he has had 18 papers reviewed in Math. Reviews, and all but five were listed in category 35 (Partial differential equations).

Here are the MR numbers and titles for his five most recent books/papers:

1. (no title information available) MR number `94k:35117`.
2. Hardy spaces and the two-dimensional Euler equations with nonnegative vorticity. (a paper written with S. Müller) MR number `94h:35205`.
3. Partial regularity for stationary harmonic maps into spheres. (a paper) MR number `93m:58026`.
4. Phase transitions and generalized motion by mean curvature. (a paper written with H. M. Sonner and P. E. Souganidis). MR number `93g:35064`.
5. Measure theory and fine properties of functions. (a book written with R. F. Gariepy). MR number `93f:28001`.

I've not yet found copies of the papers on-line, although I've found an anonymous FTP server at Berkeley that has an "L.C.Evans" directory. It appears to contain errata for the Evans-Gariepy book and a set of notes on PDEs.

I'm now searching Usenet for information about L. Craig Evans.

User: *Where in Usenet are you searching?*

Fish17: I've started in `sci.math` and `sci.math.research`.

User: *OK, get me copies of his five most recent papers in partial differential equations. This is a high-priority task.*

Fish17: OK.

[Time passes...]

Fish17: Three of the papers you requested are available on-line via a commercial service. It will cost $8 per paper to get electronic versions. Can I spend $24 to get all three?

User: *Go ahead.*

Fish17: OK, the other two papers do not seem to be available on-line. I can request printed copies from his office if you like.

User: *Do that, and when you've finished go into stand-by mode.*

Most of the information that Fish17 cited above is readily available on-line, if one knows where to look for it. The phrase "He's a mathematician" gives the IFish a starting point: the AMS on-line directory and index of reviewed papers[22]. From that list, one can imagine searching the Science Citation and Math Reviews databases for paper titles and cross-references. Netnews searches are performed daily by numerous "clipping services." Finding anonymous FTP servers at Berkeley is not difficult given some knowledge about server naming conventions, or via Archie[23].

The above example demonstrates an Internet Fish's ability to go out and find information on a new topic of interest. Internet Fish can also act as information brokers for particular topics of interest, watching data streams for new information in those topics. Here is an example of an Internet Fish that knows about and continuously monitors "cryptography:"

User: *Fish23, tell me what's happened in cryptography recently.*

Fish23: Well, OK. I've seen five interesting items since you last asked:

1. Crypto '96 was held in Santa Barbara, CA, from Aug 18-22. 75 papers were scheduled to be presented.

[22] `http://www.ams.org/committee/publications/author-lookup.html`

[23] Some Archie servers allow regular-expression searches of the server's list of anonymous FTP servers. Asking the Archie server at `archie.rutgers.edu` for all anonymous FTP servers matching "*berkeley*" yields nine servers, one of which is `math.berkeley.edu`.

2. There's been a lot of discussion on `sci.crypt` about a paper titled "Failsafe Key Escrow" by Leighton and Killian.
3. There's also been a lot of discussion about something called "MD5" and a weakness relating to "collisions" both in `sci.crypt` and on the `Cypherpunks` mailing list.
4. A new newsgroup has appeared, `sci.crypt.research`. I've added it to the list of newsgroups I monitor.
5. Finally, there seems to be a new FAQ announced on `Cypherpunks`, available from the URL `ftp://ftp.netcom.com/pub/tcmay/CP-FAQ`.

User: *How many newsgroups are you monitoring?*

Fish23: I'm currently watching five different newsgroups: `sci.crypt`, `sci.crypt.research`, `alt.security.pgp`, `talk.policy.crypto`, and `comp.org.eff.talk`. The last two don't contribute much as cryptography information sources, although they have lots of traffic.

User: *Follow the MD5 discussion closely, and send me copies of anything sent by people in the domain "`rsa.com`." See if you can find a copy of the Leighton & Killian paper on-line somewhere.*

[Time passes...]

Fish23: I managed to find a copy of "Failsafe Key Escrow" in an anonymous FTP archive at `lcs.mit.edu`. The URL of the paper is:

ftp://ftp-pubs.lcs.mit.edu/pub/lcs-pubs/tr.outbox/MIT-LCS-TR-636.ps.gz

I've put a copy of the paper in your `papers` subdirectory.

User: *Fish23, thank you.*

1.3.3 Serendipitous Resource Discovery

The third "limitation" of current resource discovery systems, the lack of *serendipitous* resource discovery, is really more a desired property for Internet Fish and not a limitation of current systems. Perhaps the most enticing feature of the WWW is that users are never quite sure what they are going to find when they follow a hyperlink. There is always the possibility of finding something unexpected while browsing the Web, something the user finds interesting yet did not know about previously. We call the process of finding interesting information in an unexpected place or manner *serendipitous resource discovery*, for it was "lucky" that we found what we did.

Opportunity for serendipitous resource discovery seems to be one of the most attractive features of the WWW for new users. When the Web was still relative small, almost invariably new users returned frequently to the "What's New With NCSA Mosaic and the WWW" page looking for new Web sites to explore[24] Services exist that cater to Web browsers' desire for "unexpected

[24]The "What's New With NCSA Mosaic" page [44] listed announcements of new Web servers and new content that was maintained by the Mosaic developer team at NCSA. At that time (early 1994) the WWW consisted of at most a few thousand servers and new server announcements numbered at most ten daily; it was possible at the time to visit every new server and at least glance through its on-line content. As of January, 1996, new Web servers were being brought on-line at a rate in excess of 300 per day, and that rate itself continues to increase.

but interesting" information[25]. Ideally, Internet Fish will become "generators" of serendipitous resource discovery; in the course of their searches they may uncover information that is related to their search topic in an unexpected manner, or they may discover an unexpected relationship between two "independent" bodies of knowledge.

1.3.4 Other Goals and Limitations

Our Internet Fish Construction Kit provides the framework for building Internet Fish and tailoring them for particular types of information retrieval tasks. Our design is sufficiently general and abstract to allow IFISH to operate in a wide variety of information environments. However, to prove the viability of our approach we have chosen to make certain assumptions concerning the operating environment.

First, Internet Fish created by this thesis will "swim" in the sea of information freely available on the Internet; IFISH will not interact at this time with commercial database systems. Public, free, Internet-accessible resources have the benefits of network connectivity (and connectivity problems), free access, a more uniform network protocol for access, and the abundance of poorly-organized information. While commercial systems certainly contain an abundance of information, there are fewer opportunities to "add value" via an Internet Fish to a commercial database that is already fully indexed and accessible.

Second, the currently favored method of data publication on the Internet is the World Wide Web (which implicitly includes all of Gopherspace and all anonymous FTP servers as well as the set of information available from HTTP servers). Given that the overwhelming majority of Internet-published information sources are available via WWW protocols, the Internet Fish must live in the WWW. Additional protocols will be added as necessary and desired, but the primary goal of this thesis will be to produce an Internet Fish capable of swimming in the WWW.

Third, for the purposes of this project we assume that individual Internet Fish may communicate with only one user at a time and that individual IFISH exist independently and do not communicate with other IFISH. Were we to carry on with the fish metaphor there would be "schools" of Internet Fish swimming in the Internet, and each IFISH would be able to communicate and swap information with other IFISH. IFISH could then become more specialized (in terms of area of interest) and "bloated" IFISH could subdivide into multiple smaller units.

Finally, the sample conversations shown above demonstrate typical Internet Fish search behavior, but these conversations provide a view of only the lower level of an IFISH's operation, namely the process of acquiring actual information in response to particular criteria of interest. Internet Fish also operate at a meta-level of resource discovery; the goal at this level is not to find particular information but to discover effective techniques for finding information. Internet Fish have the pos-

---

[25] For example, the "Cool Site of the Day" service at

`http://www.infi.net/cool.html`

points to a different WWW page each day. The "URL Roulette Wheel" at

`http://kuhttp.cc.ukans.edu/cwis/organizations/kucia/uroulette/uroulette.html`

returns a different, randomly chosen URL every time it is accessed. Alta Vista [15] and WebCrawler [47] also provide services which will direct a users to a randomly chosen URL at his request.

sibility of discovering information about resource discovery. Although not addressed in this thesis, Internet Fish may permit new methods for discovering effective procedures for resource discovery. Consider the task a research librarian faces when given a new, unfamiliar topic to investigate. The librarian often begins work with no particular knowledge of the field of inquiry, and the criteria for determining whether something is of interest to the client may not be specific. The librarian, therefore, needs to be able to determine whether retrieved information has a reasonable chance of interesting the client (is the information "in the ballpark" of the topic of interest). At a higher level, the research librarian must be able to find information resources that are suitable for searching the particular topic under study. To accomplish these tasks, the librarian draws upon two types of knowledge:

1. Particular knowledge about the specific field encompassing the topic of interest.
2. General knowledge concerning meta-resources or structure that may be used to find topical, specific information sources to answer the query.

Ideally, the research librarian would have a significant amount of knowledge particular to the topic of interest; this would allow the librarian to quickly focus in on resources likely to answer the client's query. If such specific knowledge is not available, the librarian uses more general resources and indexes to find candidate resources that might contain the desired information.

The parallels between research librarians and Internet Fish are obvious. When Internet Fish begin life, they do not contain lots of inherent knowledge about their particular area of study; this knowledge must be acquired over the lifetime of the IFish. To acquire this particular knowledge IFish, like research librarians, depend on general knowledge of research techniques and other available resource discovery tools. Research librarians acquire this knowledge through training; Internet Fish will acquire some of this knowledge through built-in heuristics and some via deduction over time.

To build the Internet Fish spawning ground we will need one or more languages to describe the construction of an IFish. IFish need to be able to represent their internal state, acquired knowledge, and the procedures by which information is transformed or acquired. Such languages must exist if we desire eventual inter-IFish communication or wish to interface IFish into general applications. As IFish are consumers and transformers of information we need to understand what the "primitive operations" are on units of information, and what primitives are appropriate for IFish construction. Methods of combining these primitive operations also need to be addressed, for we want IFish to be able to combine and synthesize new types of IFish as necessary. Together, our experiences with Internet Fish will yield a starting point for the construction of future "network librarians."

1.4 The Road Ahead

Armed now with a grasp of the history and current practices of Internet resource discovery systems we may dive into the details of the Internet Fish Construction Kit. We begin with the goal of facilitating encapsulation of heuristic knowledge. Chapter 2 discusses programming and system features provided by the Construction Kit that enable rapid encoding of such knowledge and also allow particular IFish to take advantage of that knowledge during the course of operation. Chapter 3 details Construction Kit features that support long-term conversations between the IFish and its user. This chapter also describes how IFish determine "interestingness;" that is, in the absence of user interaction how the IFish attempts to order information based on how interesting it might be to the user. Chapter 4 provides a detailed look at the construction of a particular Internet Fish that is designed to search out Web pages similar to pages provided by the user. This "Finder" IFish makes use of many common Web index engines and shows how simple heuristic knowledge and user interaction together provide a powerful research tool. Finally, thoughts on serendipitous resource discovery, evaluating IFish performance, and the future of IFish-like tools are presented in Chapter 5. We conclude that chapter, and this thesis, with some predictions on the future information "seas" that will be homes to vast schools of Internet Fish.

Chapter 2

Encapsulating Heuristic Knowledge

At their core, Internet Fish are consumers of information. Their entire existence is predicated on the gathering, digesting, and processing of diverse bits of information that may be scattered across the globe in various databases and repositories. We cannot predict what information an IFISH will find as the consequence of a particular starting state, nor can we even predict what types of information will necessarily be encountered. What we *are* able to state with some certainty, though, are a number of basic, fundamental properties of the information space in which IFISH exist. We can describe for the IFISH how certain pieces of information may be linked to other pieces of information, or how various references to information may be decomposed and reassembled. IFISH must be taught how to interact with their environment, and it is the basic properties of that interaction, the heuristic knowledge we wish to implant in Internet Fish, that is the subject of this chapter.

We concentrate below on the portion of the Internet Fish system designed to facilitate the encapsulation of heuristic knowledge. By "encapsulation of heuristic knowledge" we mean specifically the encoding of certain rules, assumptions, axioms, processes, procedures, etc., that allow an Internet Fish to interact with its environment. For example, since we require Internet Fish to interact with the World Wide Web (see 1.3.4 above), we must teach IFISH how to recognize possible URLs, how to validate whether a particular URL is accessible, and how to retrieve the information document that is specified by a given URL. Some heuristics may relate to the structure of the network, perhaps teaching Fish about the SMTP protocol and procedures likely to validate e-mail addresses, or perhaps rules that help fish to recognize that the string "`www.ai.mit.edu`" (a) looks like a fully-qualified domain name (FQDN) for an Internet host (because of the string structure), and (b) is likely to be running a WWW server on port 80[26] (because the FQDN begins with "www"). The heuristics might be more specific, explaining perhaps how to parse the HTML returned by a query to the Lycos [36] database and turn the page of text intended for human consumption into further information to be considered and investigated by the Fish. If we wish to create a Fish particularly good at finding mathematical information, we can add heuristics for dealing with the American Mathematical Society's on-line directory listings or electronic versions of *Math Reviews* on CD-ROM.

We begin by describing in Section 2.1 the claims and assumptions under which we have chosen to operate, as well as some of the issues that weigh in favor or opposition for each. These assumptions,

[26] Port 80 is the canonical TCP port for HTTP servers.

although fairly basic, drive many of our design choices. Sections 2.2 and 2.3 then give the technical details of our implementations of *infochunks* (primary units of information) and *rules* for operating on those infochunks. In Section 2.4 we discuss the interactions between infochunks and rules in the Fish system. The topics of user interaction and assessing the "interestingness" of infochunks are left for Chapter 3 below.

2.1 Claims and Assumptions

The first step in the design process for the Internet Fish system was to compile a list of known facts and reasonable assumptions under which Internet Fish operate. These claims have a direct impact on many particular IFISH design choices, as outlined below.

Claim 1 *The Internet Fish operates in a dynamic environment:*

a. The information space of interest, the World Wide Web, is dynamic over time.

b. Fixed transformations on data may yield different results over time.

c. The set of operations an Internet Fish can perform changes over time.

It is without question that the total collection of information available via the World Wide Web is constantly in flux. As mentioned in Section 1.2.3 above, the Web is growing at a tremendous rate: Yahoo alone reports that they are receiving thousands of requests each week to add new server listings to their service. The statistics gathered by Lycos, Alta Vista and other web indexing engines clearly indicate the number of WWW "pages" (documents) that are accessible to an unprivileged WWW client is growing at a greater-than-linear rate. Furthermore, we know these statistics must be undercounting the true nature of the Web because they cannot account for dynamic databases accessed via a static WWW interface[27]. Thus, because the information space can change over time, the Internet Fish cannot treat any information it extracts from the WWW as guaranteed static. In fact, because the connectivity of the Web is also not guaranteed (see Claim 2 below) a (slightly paranoid) Internet Fish needs to cache locally copies of retrieved data, along with the date and time of the retrieval.

Yet the fact that the Web is dynamic permeates the IFISH design beyond simple data caching. Because the Web is dynamic certain Web services that the IFISH uses as primitive tools (e.g. DEC's Alta Vista index [15]) necessarily return different answers over time. Thus, any IFISH that depends on such services always has the option to re-query the services in the hope that the query will turn up new information sources previously undiscovered. Even local data transformations, analyses that the IFISH performs on its own local data set, will change over time as the IFISH consumes more information[28]. Therefore we need to be able to identify certain IFISH operations as possibly time- and/or network-dependent, since each such operation may need to be repreated at a later time.

[27] For example, the FedEx database of package tracking information, accessible at `http://www.fedex.com/`, changes state every time a FedEx worker picks up a package or transfers a package from one point to another. The interface page, however, is static.

[28] For example, the current IFISH implementation performs clustering of retrieved HTML documents periodically to determine relationships among groups of documents.

IFish are dynamic in a third way, one which is independent of the possible changes in the information space. As mentioned in Section 1.3.4 above, we would like IFish to be able to discover and generate new transformation rules over time and recognize patterns of operation that are effective resource discovery procedures. IFish, therefore, should be able to modify their own rules sets over time, and thus rules may be discovered at a later date that can be applied to information previously thought to be completely processed. IFish also must maintain records of what rules were applied to particular pieces of data and when those applications took place[29]. Our implementation must therefore permit rules to be changed easily over time without disturbing or invalidating the collection of information that has been gathered up to that point.

Separate from the issues relating to dynamic information environments are issues related to the structure, sequence and organization of the information space.

Claim 2 *An Internet Fish cannot assume* a priori*:*

a. The set of all possible types of information that it will encounter.

b. The existence of a "best" or "guaranteed" measure of the relevance of a particular document, or that it is possible to perfectly quantize the "interestingness" of a particular piece of information in either general or relative terms.

These claims, too, seem fairly obvious, but as will be shown below they constrain our design to some extent. Type information, or the lack thereof, is the easiest to support. By "type information" we refer to semantic labels on particular content, not primitive object types that may be determined by the operating system or environment. For example, we may want an IFish that understands that strings satisfying the regular expression: `[^@\%.:-]+@([^.]+.)*[^.]+` are possible e-mail addresses [12]. Rules in the IFish can use this probable type information as a precondition satisfier for a routine that tries to validate e-mail addresses via SMTP port 25[30]. Thus, our IFish must support loose semantic type information, including qualitative modifiers such as "possibly," "definitely," and "definitely not." Of course our type system should not be bounded or limited if possible, since we cannot predict when new types will be added to the system or what types of information will be encountered.

"Interestingness" (that is, how inherently interesting is a particular piece of information to an IFish at a given moment in time) is discussed in detail in Chapter 3 below, but we should mention here one related underlying assumption. Most of the literature in resource discovery relates to document scoring: algorithms for trying to determine relevance of one document to another document, to a group of documents, or to the user making the query. Our current implementation of IFish uses one such algorithm as a way to gather evidence of possible relationships among documents. A problem arises, however, if we try to use some such quantitative measurement to decide what subproblem an IFish should work on next. Since our information space is unbounded, it is extremely unlikely that an IFish would run out of tasks (rule applications) it could perform. Thus, in order to conserve IFish effort (a finite resource) some method of targeting effort toward those information pieces most likely to yield relevant data is desirable. There is a danger, however,

[29] IFish must also record error-related information, as discussed below.

[30] Mail handlers that speak SMTP, the Simple Mail Transfer Protocol, operate on TCP port 25. Using the `VRFY` and/or `EXPN` commands it is often (but not always) possible to confirm that a given e-mail address is received at a particular destination.

in putting "too much faith" in numerical scoring of documents, namely that some truly interesting and relevant documents will never be explored because they fail to meet some numerical cutoff[31]. We must be careful, therefore, to not overburden whatever measure of interestingness we use with too many dependent system applications (like processor and memory resource allocation).

There are other constraints, too, that are imposed on IFISH . Our information sources are all located at remote sites; the slowest part of our system[32] is the network itself and retrieval of information over that network. (Even if the IFISH is running on a host with fast Internet connections, the server at the far end may be heavily loaded, or separated from the IFISH by a slow network link, or both.) We would like IFISH to have a structure that permits easy multitasking so that when the network is the slow link the processor is not idle busy-waiting for the network. Furthermore, since the majority of Web resources we access are designed to be human-readable as opposed to machine-readable, we must assume that some significant effort will be spent parsing retrieved information into machine-friendly structures and converting machine queries into the (perhaps quite cumbersome) human-friendly formats expected by the services we access. Thus, anything we can do to minimize linear dependencies in the system and foster multitasking or multithreading of an IFISH's tasks is a good idea.

Finally, we must remember that one of our goals for IFISH is to begin to discover effective procedures for resource discovery: techniques and chains of procedures that are particularly effective at discovering information. We are interested therefore not only in the raw information extracted by IFISH but also in the way in which that information was obtained or generated. Capturing the "generation record" is also important for communicating with IFISH users; we want to be able to answer questions like, "How did you discover fact x?" or "What evidence supports y?" Thus it is important as we are building local hypertexts of related information to also keep track of rule invocations used to generate each piece of information.

2.2 Infochunks

In the IFISH universe there are basically two types of interesting objects: *infochunks* and *rules*[33]. An *infochunk* is a "chunk of information," a piece of information which may be acted upon; *rules* are procedures that operate on infochunks and may create new infochunks, modify existing infochunks, or perform some other task[34], possibly with side effects. This section describes the structure of infochunks and the hypertext structure that IFISH build to contain retrieved information as they research.

IFISH infochunks are implemented as MIT Scheme structures; every infochunk contains six value slots, as illustrated in Figure 2-1. Most of the infochunk value slots are self-explanatory. The

[31] These problems are not new, of course. MIT's Undergraduate Admissions Office may overlook a well-qualified applicant because the numerical scores assigned to the application are not completely representative, to name one such example.

[32] Currently, the portion of the IFISH system that consumes the most realtime is that which deals with the network. We can easily imagine having routines in the IFISH that are so analysis-intensive that their resource cost (in terms of memory and processor time consumed) exceeds that of the network.

[33] This split of the universe of IFISH-interesting objects we call the "information/operation dichotomy," playfully invoking the idea/expression dichotomy that is a foundation of modern U.S. copyright law. See *Baker v. Selden*, 101 U.S. 99 (1879), and its progeny.

[34] Rules may also create, modify or remove rules.

Infochunk slot name	Possible content
data	any piece of information
typeinfo	list of meta-information statements, possibly empty
interestingness	interesting structure for data
backward-links	list of links to generating infochunks, possibly empty
forward-links	list of links to generated infochunks, possibly empty
invocation-list	list of rules applied to this infochunk and when the rule application occurred

Figure 2-1: Value slots in the infochunk data structure and typical content of each slot.

data slot is a pointer to a raw piece of information[35]. The interestingness structure for this piece of information is contained in the *interestingness* slot. The *invocation-list* is simply a list of rules that have been applied to this infochunk along with a timestamp of when that rule application occurred.

Forward-links and *backward-links* are lists of *link* structures and are used to weave together individual infochunks into the hypertext structure being built by the IFISH. Figure 2-2 shows a typical links structure. Each link contains a pointer to the infochunk at the "other end" of the

Link slot name	Possible content
infochunk	infochunk to which the link points
rule-name	symbolic name of the rule that created this link
timestamps	non-empty list of times at which this link was created/re-created

Figure 2-2: Value slots in the infochunk data structure and typical content of each slot.

link, along with the name of the rule[36] that created the link and the time(s) of link creation. We must include the rule name here because it is possible to have multiple links, representing different rules applications, between two particular infochunks and we do not wish to elide that information. Similarly, since rule application is not guaranteed static in time, and indeed since rules themselves may change over time, we must allow for the possibility that multiple applications of the named rule will occur and thus the existence and validity of the link may be asserted multiple times. All links are directed relationships, and links are always added in pairs so that every forward-link has a corresponding backward-link. A forward-link from infochunk I_1 to infochunk I_2 via rule R means that the rule application $R(I_1)$ generated infochunk I_2; a backward link from I_1 to I_2 implies that $R(I_2)$ generated I_1.

The *typeinfo* slot contains a list, possibly empty, of declarations of meta-information concerning the contents of the *data* slot. Recall that under our assumptions, IFISH cannot assume much concerning the types of information they will encounter out on the Internet. Nevertheless,

[35] For implementation-specific reasons the contents of the data slot may actually be spilled to disk and not resident in core memory. Spilling data is automatically handler by the IFISH interaction loop, as discussed in Section 2.4.4 below. Retrieving spilled data from disk is transparent to higher-order IFISH procedures.

[36] See Section 2.3 for a discussion of rules versus rule names and why we use the latter here.

there is meta-information that IFISH can use to discuss possible types of retrieved information and restrict rule applications to only relevant or well-formed inputs. *Typeinfo* declarations conform to the grammar shown in Figure 2-3. As an example of the meta-information captured

```
<typeinfo-declaration>  ==  (UNKNOWN) | (KNOWN <symbol>) | (NOT <symbol>)
                            | (POSSIBLE <symbol>)
                            | (POSSIBLE <symbol> <confidence>)
<symbol>                ==  any legal Scheme symbol
<confidence>            ==  floating-point value x, 0 < x < 1
```

Figure 2-3: The syntax of *typeinfo* declarations

by this simple *typeinfo* syntax, consider what happens when an IFISH comes across the string "`http://www-swiss.ai.mit.edu/`" in a retrieved HTML document. The HTML parsing routines identify this string as a `(POSSIBLE URL-STRING)`–"possible" because the string needs to be parsed by a URL parser and verified as conforming to the URL specification. The `(POSSIBLE URL-STRING)` *typeinfo* then triggers a rule application that tries to parse the URL in accordance with RFC 1738 [6]. If the parse succeeds the rule may update the infochunk's *typeinfo* from `(POSSIBLE URL-STRING)` to `(KNOWN URL-STRING)`. Similarly, the newly-generated URL structure would be labeled a `(POSSIBLE URL)`; it can be upgraded to `(KNOWN URL)` only after the IFISH has verified that the URL actually names some retrievable content on the web. Figure 2-4 shows a representative infochunk pulled from the hypertext structure built by an actual IFISH. The data object contained in this infochunk is a URL structure, representing a URL split into its constituent parts (access protocol, host name and path). The *typeinfo* declaration `(KNOWN URL)` tells the system that the data object is a valid URL structure and is in fact the name of an accessible WWW document. The IFISH determined that the URL was indeed valid by successfully performing an HTTP HEAD request on the URL[37]. The lone link in the forward-link list connects this infochunk to the infochunk containing the results of the HEAD request, which was generated from this infochunk via the `URL->HTTP-REQUEST-HEAD` rule, and the invocation-list contains a timestamp for when that rule application took place.

Notice that the infochunk in Figure 2-4 contains five backward links; each link represents a rule application that generated the URL contained in the data slot. The infochunk each backward-link points to contains a list of HTML anchors (hyperlinks) and URLs. These lists of URLs are generated from other retrieved documents, so together these links represent five HTML documents that all contains links to the URL `http://martigny.ai.mit.edu:80/~bal/pks-toplev.html`. We can find the URLs of the referencing documents by following chains of backward-links until we reach the infochunks containing the desired URLs.

[37] An HTTP HEAD request is identical in operation to the GET request except that the content server returns only the HTTP request headers and not the actually content of the document. IFISH always perform HEAD requests first where possible in order to both confirm that the URL points to actual content and also to check the MIME type and size of the content document. The current prototype IFISH will only retrieve documents that are both of type HTML and are relatively small compared to the amount of heap memory available (since the document is temporarily stored in its entirety in the heap.)

```
#[infochunk #[interest 0:1:0]:#[url http://martigny.ai.mit.edu:80/~bal/pks-toplev.html]]

(data #[url http://martigny.ai.mit.edu:80/~bal/pks-toplev.html])
(typeinfo ((known url))
(interestingness #[interest])
(backward-links
 (#[link #[infochunk #[interest]
          ((#[url http://www.viacrypt.com:80/] . ViaCrypt)
           (#[url http://www.SLED.com:80/] . Sled Corp.))
          anchors+urls->url]
  #[link #[infochunk #[interest]
          ((#[url http://www.pegasus.esprit.ec.org:80/people/arne/pgp.html] ...))]
          anchors+urls->url]
  #[link #[infochunk #[interest]
          ((#[url http://bs.mit.edu:8001/pgp-form.html] . PGP from the Web))
          anchors+urls->url]
  #[link #[infochunk #[interest]
          ((#[url http://web.mit.edu:80/network/pgpfone] . PGPfone)
           (#[url http://bs.mit.edu:8001/pgp-form.html] . here))] anchors+urls->url]
  #[link #[infochunk #[interest]
          ((#[url http://www.yahoo.com:80/bin/menu1/-Computers_and_Internet/]...))]
          anchors+urls->url]
(forward-links
 (#[link #[infochunk #[interest]
          #[http-request http://martigny.ai.mit.edu:80/~bal/pks-toplev.html]]
          url->http-request-head]))
(invocation-list
 ((url->http-request-head . 816672957) (url/remember-hostname! . 816672956)))
```

Figure 2-4: A sample infochunk and its internal components

2.3 Operations: Tranducers and Rules

The second class of objects in the IFISH universe contains *rules*. Rules describe procedures that operate on infochunks and perform some task, usually the generation of new infochunks. It is via predefined rules that we build heuristic knowledge into IFish. Methods for obtaining new information over the network, procedures to identify and parse particular representations of data objects, and even meta-rules that generate new rules that extend IFISH in a particular fashion are examples of the types of heuristic knowledge that can be encapsulated in rule objects.

Every rule in the current implementation of IFISH operates on exactly one infochunk. This restriction no more limits the set of theoretically-expressible heuristics than does "currying" arguments in λ-calculus. Each rule in the current IFISH implementation consists of four parts: a *name*, a *precondition*, a *transducer*, and an *error-handler*. We describe each of these components in detail below.

Every IFISH rule has a unique *name* by which it may be referenced. In our IFISH implementation the name of a rule is simply a Scheme symbol. The need for unique, symbolic names for rules is created by our desire to be able to save and restore the complete state of an IFISH[38]. Furthermore, as one of our goals is to have IFISH analyze their own information-gathering procedures we need to be able to compare rules that may have identical function but not consist of identical Scheme procedure objects. The rule name adds a layer of indirection that allows us to maintain a handle on rules without having to retain raw procedure objects. IFISH rules are conventionally named by the *typeinfo* components of the input and output infochunks (e.g. `URL->HTTP-REQUEST-HEAD` is the rule the takes a URL and attempts to retrieve via the HTTP HEAD command the content headers of the given URL).

A rule's *precondition* and *transducer* define respectively what infochunks are acceptable inputs for the rule and what action the rule has with respect to that infochunk. An IFISH working with a particular infochunk determines what rules apply to that infochunk by invoking in sequence each known rule's precondition to the infochunk. Rule preconditions are always boolean procedures of exactly one argument (an infochunk); infochunks that return a value of true when given as inputs to a rule's precondition are in the domain of the rule's *transducer*. The *transducer* is the portion of the rule that actually performs work. Usually a transducer applies some operation to the input infochunk, and if new information is generated the transducer constructs a new infochunk to contain that new information, links the new infochunk into the existing hypertext, and "announces" the existence of the new infochunk to the IFISH system. Should an error occur while the rule is running, the rule's *error-handler* is invoked to handle the signalled condition.

Figure 2-5 shows the Scheme source code for an example rule, in this case the rule that describes the first phase of retrieving the WWW document described by a particular URL[39]. The `DEFINE-TRANSDUCER` macro accepts as argument a symbolic name for a transducer object and a procedure of the form `(lambda (infochunk) ...)` and creates a transducer object in the Scheme environment referenced by the symbolic name that can be used in subsequent rule definitions[40]. The `DEFINE-RULE` macro takes as arguments a rule name (a Scheme symbol), a precondition procedure, a transducer, and optionally an error-handler, which are used to both create a rule object and install that rule in the IFISH system. Macros are provided for common precondition cases, such as `SIMPLE-TYPE-PRECONDITION` which checks that the *typeinfo* of an infochunk satisfies the boolean equation specified as the first argument (in this case a disjunction of the types `(POSSIBLE URL)` and `(KNOWN URL)`), and if so passes the infochunk on to the second argument (if present) and returns that value. Thus, the precondition for `URL->HTTP-REQUEST-HEAD` guarantees that the input (a) has *typeinfo* `(POSSIBLE URL)` or `(KNOWN URL)`, and (b) the data slot of the infochunk contains a Scheme `URL` structure.

When the rule `URL->HTTP-REQUEST-HEAD` is applied to a particular infochunk, the URL structure is extracted from the infochunk and passed on to low-level HTTP network routines that try to

[38] MIT Scheme cannot in general dump procedure objects to disk without dumping the entire contents of the Scheme heap.

[39] IFISH use a two-step process to retrieve the "contents" stored at a particular URL. In the first phase, IFISH try to retrieve the specified URL using the HTTP `HEAD` command. `HEAD` is similar to `GET` except that the remote server only returns the HTTP result code and content headers and not the actual content [5]. The returned content headers may include information relating to the size of the specified object, the date and time the object was last modified, a MIME [8] type, or even a different URL where the requested content can be found (a forwarding pointer). IFISH use this information to avoid downloading large documents that cannot be analyzed by the current system, such as images or sound files.

[40] The `DEFINE-TRANSDUCER` macro hides some complexity from the user created by the current IFISH implementation.

```
(DEFINE-TRANSDUCER tdcr/url->http-request-head
  (lambda (infochunk)
    (let* ((url (infochunk/data infochunk))
           ;; don't let the HEAD request take more than 10 seconds
           (raw-result
            (run-for-n-seconds (lambda () (url/head-url url)) 10)))
      (if (condition? raw-result)
          raw-result
          (let* ((result (http-raw-result->http-request url raw-result))
                 (new-infochunk
                  (make-infochunk
                   result
                   (cons '(known http-request-head)
                         (typeinfo/filter-out-basetype infochunk 'url)))))
            (ANNOUNCE-AND-LINK new-infochunk)
            (VALIDATE-TYPEINFO 'url)
            )))))

(DEFINE-RULE 'URL->HTTP-REQUEST-HEAD
  (SIMPLE-TYPE-PRECONDITION
   '(or (possible url) (known url))
   (lambda (infochunk) (url? (infochunk/data infochunk))))
  tdcr/url->http-request-head
  ;; try out the network default
  default-network-error-handler)
```

Figure 2-5: An example IFish rule: `URL->HTTP-REQUEST-HEAD`.

perform a HEAD request on the given URL. Assuming that the request succeeds[41], the raw result (a string) is parsed into an *http-request* structure and encapsulated in a new infochunk. The rule must now notify the IFish system that it has created a new infochunk that should be linked into the hyperstructure of known infochunks; this is accomplished via the macro `ANNOUNCE-AND-LINK`. Finally, since the HTTP request succeeded, the rule also knows that the input URL does indeed specify an actual, accessible document, and the *typeinfo* on the URL's infochunk must be updated to reflect this knowledge. The `(VALIDATE-TYPEINFO 'url)` macro expression simply guarantees that any *typeinfo* entries of the form `(POSSIBLE URL)` are changed to `(KNOWN URL)`.

Notice that by making each step of the process of transforming information a separate rule application it is very easy to leverage already-defined IFish rules when creating new rules. A particular transducer may be invoked by multiple distinct rules, each with separate preconditions and error handlers. Further, rules may be redefined "on-the-fly" in response to changing conditions. IFish may be easily modified to include new sources of information as they become available.

[41]If the network request fails the rule transducer will return a Scheme error condition as its value. The IFish mechanisms for dealing with errors is discussed in detail in Section 2.4.3 below.

2.4 Infochunk-rule Interactions and Supporting System Software

Having defined both infochunks and rules, we turn now to the "low-level" systems of the IFISH that both trigger infochunk-rule interactions and also provide various auxiliary systems, such as error recovery and event handling. Together, these pieces of the IFISH provide the substrate upon which we explore user-IFISH conversations. We begin in Section 2.4.1 with the IFISH interaction loop, which is responsible for selecting and evaluating infochunk-rule pairs. Section 2.4.2 discusses in brief the IFISH event handling system, which allows tasks to be scheduled to occur at particular times. Error handling and recovery is detailed in Section 2.4.3. IFISH system support for user interactions may be found in Chapter 3 along with a broader discussion of user interactions and interestingness.

2.4.1 The Interaction Loop

The interaction loop is the core of the IFISH system; it is this loop that controls which rules are applied to which infochunks and the order in which those applications are made. At its most basic level, the interaction loop operates as outlined in Figure 2-6. Each cycle through the interaction loop begins with a `gc-check!`, which checks the amount of available Scheme heap and forces a garbage collection if that amount is below a particular threshold. Part of the IFISH system attaches itself to the Scheme garbage collector and will spill infochunk data slots to disk as necessary to free up heap memory.

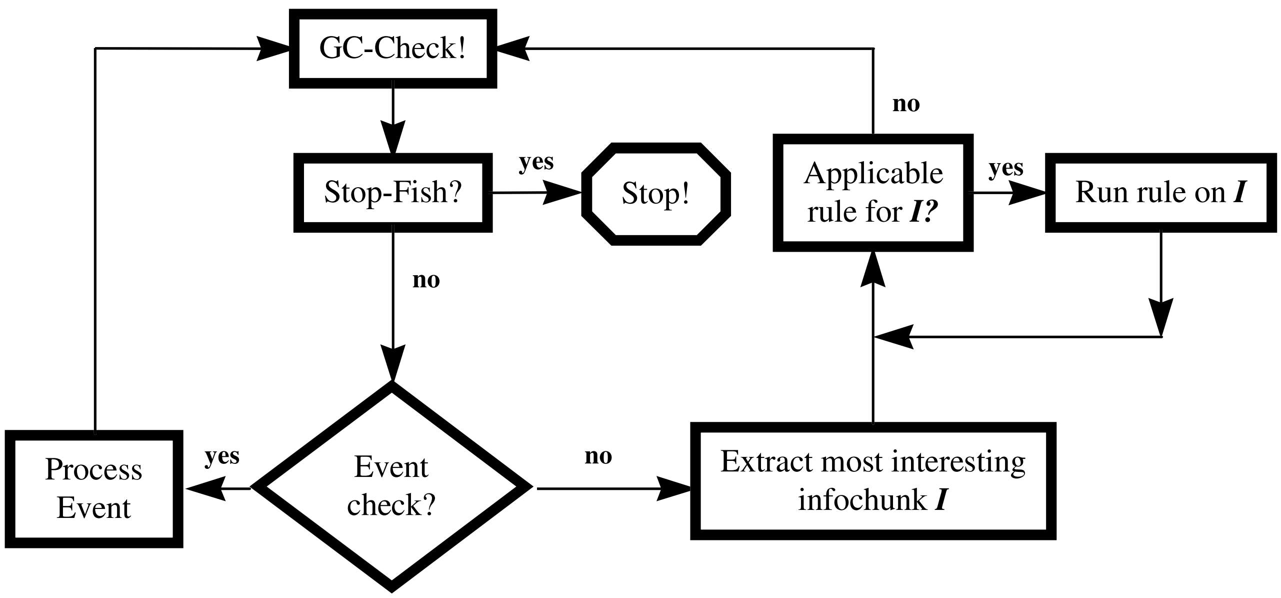

Figure 2-6: A simplified view of the IFISH interaction loop.

After the `gc-check!` has been performed the interaction loop checks a particular global variable for a signal to end execution; assuming that signal is not raised the interaction loop proceeds to check for time-dependent *events* that need to be processed. Section 2.4.2 describes event handling in detail; each event has associated with it a trigger time, and all events with trigger times before the current time of the event check are processed immediately.

Assuming that there are no events needing to be processed, the next step in the loop is to identify an infochunk upon which to work during the cycle. Infochunks are sorted[42] by "interestingness" (see Section 3.2 below) and the "most interesting" infochunk is identified. This infochunk then becomes the focus of attention for the IFISH until the cycle is completed.

Once the current "most interesting infochunk" has been identified every rule precondition is invoked on that infochunk. Any rule whose precondition evaluates to TRUE is an *applicable rule* (i.e. the current most interesting infochunk is a valid input for that rule). Each applicable rule is run in turn on the infochunk under study until every applicable rule has been processed; when finished, the interaction loop then starts over. Every IFISH rule application occurs within a protected environment so that errors may be caught and handled when they occur. When an errors does occur, it is trapped by the system and the Scheme error condition is passed to the rule's *error-handler* for further processing.

Notice that the interaction loop does not itself impose any additional sequential execution constraints on the IFISH. Thus, in a multiprocessor environment the IFISH may spawn multiple tasks simultaneously so long as shared areas of memory are guarded against overlapping accesses. The current IFISH implementation in fact uses MIT Scheme's threads package to allow simultaneous execution of the interaction loop itself as well as the miniature WWW server that is used to provide user interaction (see Section 3.1.2 below).

2.4.2 Events

Event handling is another important piece of the IFISH substrate, for it is through events that time-dependent actions enter into the IFISH equation. Remember the assumptions of Claim 1 above: everything an IFISH retrieves is subject to change and every rule involving the network may incur transient failures. We need to give IFISH rudimentary routines scheduling process executions, refreshing suspect or short-lived data, and other time-dependent tasks that may arise. The current IFISH event model is suspiciously trivial, but it provides the minimum set of necessary functionality.

Event slot name	Possible content
thunk	Scheme procedure of zero arguments to invoke
time	Time after which invocation is allowed

Figure 2-7: Events

An event object contains only two items: a *thunk* and a *time*. The *thunk* may be any Scheme procedure of zero arguments; the *time* is a standard Unix timestamp of the current time[43]. Events

[42]Notice that we are assuming here the existence of a total order, based on "interestingness", for the set of infochunks under consideration. A total order is not strictly necessary; we could use the same algorithm with a partial order. If no order exists among infochunks then we must depend solely on user interaction if we wish to do better than random selection.

[43]The low-level system call on most Unix systems returns an integer representing the number of seconds since

are "declared" by inserting them into a global *event-heap* which automatically keeps them sorted in order of earliest permitted execution time. Every invocation of the interaction loop then simply compares the earliest timestamp to the current system time. Thus, given a `thunk` and a `time`, creating a "single-shot" event is as easy as `(event/install! (make-event thunk time))`. For recurring events, the IFISH system provides the following convenient construct:

```
(define (event/make-recurring-event! thunk loop-time #!optional start-time)
  (let ((start-time (if (default-object? start-time)
                        (get-universal-time)
                        start-time)))
    (let ((recurring-thunk
           (lambda ()
             (thunk)
             (event/make-recurring-event!
              thunk loop-time (+ loop-time start-time)))))
      (event/install! (make-event recurring-thunk start-time)))))
```

The "Finder" IFISH described in Chapter 4 uses recurring events for a variety of housekeeping functions, such as making periodic snapshots and summary reports of the state of the IFISH. Recurring events also allow IFISH rules to defer portions of their executions until later points in time.

2.4.3 Error Handling and Recovery

The third component of the IFISH support system involves error handling and recovery. IFISH live in a dynamic environment filled with networks that may be inaccessible, data sources which may fail, and procedures that may not properly handle the "unexpected." This demands that IFISH have very robust error handling. IFISH must not only be able to withstand errors but also handle them properly, perhaps by retrying the computation that caused the error or asking for user intervention. Our current IFISH implementation has relatively simple mechanisms for dealing with errors but these mechanisms are quite sufficient for our purposes.

Recall that every rule application carried out by the IFISH interaction loop occurs within a protected environment so that errors may be trapped and handled within the IFISH itself. Every IFISH rule includes as one of its components an *error-handler* element; a default error handler is used if no explicit routine is provided to the `DEFINE-RULE` macro. An IFISH *error-handler* is a procedure of three arguments: infochunk, rule and error condition. In appropriate cases such as temporary network outages error handlers thus can retry or reschedule or the errant rule application.

By default, any rule definition that does not include its own error handler is assigned the default error handler shown in Figure 2-8. The default handler first saves away a copy of its arguments in case the user should wish to examine the error condition later[44]. Then the error handler calls the question-maker `qm/default-error-handler` to construct a yes/no question asking the user whether the rule application that raised the error should be retried; Section 3.1.2 describes user questions and question-makers in detail. The resulting yes/no question is then installed into the system and presented to the user through the normal conversation interface. No further action

January 1, 1970.

[44]This is most useful for debugging purposes.

```
(define (default-error-handler infochunk rule condition)
  (let* ((condition-type/name
           (access %condition-type/name (->environment '(runtime error-handler))))
          (condition/field-values
           (access %condition/field-values (->environment '(runtime error-handler))))
          (infochunk-data (infochunk/data infochunk))
          (rule-name (rule/name rule))
          (condition-object
           (list (condition-type/name (condition/type condition))
                 (with-output-to-string
                   (lambda () (display (condition/field-values condition))))))
          (error-record
           (vector infochunk-data rule-name condition-object (get-universal-time))))
    (with-values
        (lambda () (make-question 'qm/default-error-handler error-record))
      (lambda (the-question the-question-ichunk)
        (let ((new-infochunk
                (make-infochunk the-question-ichunk '((known question)))))
          ;; This is normally done by ANNOUNCE-AND-LINK,
          ;; but we're not within a DEFINE-TRANSDUCER here
          (link-infochunk infochunk new-infochunk rule-name)
          (infochunk/recompute-interestingness! infochunk)
          (infochunk/recompute-interestingness! new-infochunk)
          (announce-new-infochunk new-infochunk))
        ;; install the question
        (question/install! the-question)
        ;; that's it!
        ))))
```

Figure 2-8: The IFish default error handler.

is taken until the user's answer indicates whether he wishes the IFish to retry the failed rule application.

The default error handler does not attempt to restart or redo a rule application on its own because it handles internal Scheme errors (which do arise) along with network-related problems. However, the majority of errors encountered by an IFish arise from transient conditions on the Internet; a remote host may be unaccessible or some portion of the DNS[45] hierarchy may be distributing incorrect information, or any of a number of other ills. All of these errors are transient in nature and often simply retrying the rule application that raised the error in the first place is sufficient. The IFish system provides an alternative error handler for such errors that is appropriate for network-related rules. The `default-network-error-handler` attempts to retry the failed rule application at a future specified time (by default five minutes after the first attempt) via the IFish

[45] Domain Name Service [41], the mechanism by which symbolic host names (e.g. `freeside.ai.mit.edu`) are converted into numeric IP addresses (e.g. `18.43.0.178`).

event system; if this second attempt fails then the default error handler (or a rule-specific handler) is invoked.

2.4.4 Resource Management

To the user, an IFish appears singularly devoid of any resource limitations or constraints. Unfortunately the IFish system does not have the luxury of unlimited memory, storage or network bandwidth for individual IFish and thus the system must maintain a watchful eye over each of these precious resources. Network access and bandwidth are by far the most limiting resources, since an IFish and the far end of a slow 28.8 Kbps connection must necessarily be conscious of the time cost of retrieving any particular document. Network resource management, because it is intertwined with particular rules and particular documents, is best handled via extra preconditions on network-related operations. For example, the rule `HTTP-REQUEST-HEAD->HTTP-REQUEST`, which is responsible for retrieving the actual contents of a particular Web page, checks both the content type and length before initiating a transfer. A page must be sufficiently small and in a useful form before it will be downloaded and decoded[46].

Other types of resource management are better handled on a global scale. In a multi-threaded environment we might have to restrict the total number of network connections created, or perhaps we have access to a large data stream that can only be accessed linearly (like a tape with possibly interesting data on it). IFish have to deal with limited in-core storage; dynamic memory is not infinite and since the IFish has knowledge about the data it has already collected it is in a better position to spill data to disk than a general virtual memory system. The IFish system includes its own spill code to transparently move infochunk data to and from magnetic storage (hard disks).

When the available Scheme heap memory falls below a threshold, the spill system detects the low-memory condition and begins moving infochunk data slots to disk. Replacing spilled data objects are small records that contain both the filename of the spilled code as well as a hash value derived from the spilled data. The computed hash is the same as that used to store and retrieve infochunks within the system[47], thus overall system performance is not degraded too much by having many data objects present only on disk.

The interaction loop works together with the spill subsystem to keep a certain minimum amount of memory available. The `(gc-check!)` call at the beginning of the interaction look tests the Scheme object heap to see if the memory threshold has been crossed; if it has, then the standard Scheme garbage collector (GC) is invoked. The Scheme GC provides a hook (`hook/gc-finish`) which is called after the completion of garbage collection; IFish replaces the default hook with its own that causes infochunk data to be spilled to disk if the GC pass did not free up enough memory.

[46] Currently IFish depend on the value of the MIME Content-type header that is sent with every HTTP 1.0 response. Similarly, IFish depend on the Content-length header for the size of the document about to be retrieved.

[47] For efficiency reasons infochunks are kept in a hash table, since for every new piece of infochunk data we want to check whether we have seen that piece of data before. Hash tables provide amortized constant cost for such lookups. IFish use specialized hash tables to take advantage of the cached hash values left behind when data is spilled.

Chapter 3

User Interactions and Interestingness

Throughout the previous chapter we were concerned only with IFISH as consumers of information. The entire substrate up to this point has focussed on facilitating easy incorporation of new data sources and other heuristic knowledge into the IFISH system. Now it is time to consider what IFISH do with the information they consume. This chapter concentrates on these aspects of IFISH: how IFISH communicate with the user and how IFISH internally model "interestingness." The two topics are intertwined with each other, as IFISH need user communication to more accurately model interestingness, and also as IFISH use interestingness to control which questions are put to the user.

The first section of this chapter, *User Interaction*, describes how user-IFISH communication is integrated into the overall IFISH system. Section 3.2 describes the requirements IFISH impose upon possible systems for measuring or otherwise comparing infochunks based on their interestingness. These two components, together with infochunks, rules and transducers, complete the IFISH substrate and pave the way for the prototype IFISH described in Chapter 4 below.

3.1 User Interaction

We have seen how heuristic knowledge allows an IFISH to interact with indexing services available over the WWW; these services accept questions (queries) from an IFISH and (usually) answer them promptly. If the service does not answer promptly we may detect this condition and handle the error appropriately. Assume now, however, that we wish to interact with a particularly ill-behaved service, one which has the following properties:

- The service may or may not answer an IFISH query,
- Should the service answer your question, it may take an arbitrarily long time to answer it, and
- The order in which questions are presented to the service is important, as the service may be more likely to answer some questions over others depending on how the questions are presented.

Any service on the network sharing these properties might be rationally classified as "misbehaved." Consider for a moment, though, how user queries must appear to an IFISH, and it becomes apparent that user interaction from the IFISH perspective is dealing with a remote service that misbehaves. Dealing with the user, and taking advantage of the user's knowledge and guidance, thus requires some modifications and extensions to the IFISH substrate. We need to make it easy both for IFISH to pose questions and for the user to answer them, and we also need to integrate the mechanism for user interaction into the overall IFISH system.

Note that we assume implicitly here that any long-term conversation between an IFISH and the user may be represented as a series of question-answer pairs. For our demonstration IFISH (the "Finder" IFISH in Chapter 4) this is certainly the case: every IFISH question put to the user asks him to confirm or deny particular statements about specific infochunks. A similar situation arises when the user asks the IFISH a question, although often the question is implied by some user action (such as pushing a button or clicking on a particular URL).

3.1.1 Questions and Answers

To add user-interaction capabilities to the IFISH substrate we must first identify how user-IFISH information exchange compares with the information exchange between an IFISH and a remote WWW server. Like a Web server, a user accepts requests for information from an IFISH and may (if the user so chooses) respond to the information request in an appropriate manner. The user's answer to a question may then trigger some particular action within the IFISH system. For example, if the user tells the IFISH that a particular document is not relevant, an IFISH can incorporate that data into its own structure and weed out or otherwise downgrade new documents that correlate well with the known non-relevant document.

This cycle of operation for questions, "generate a question, pose it to the user, and act upon the user's response," is not very different from how IFISH process infochunk-rule interactions. Unprocessed infochunks have "applicable rules" repeatedly applied to them; some of the rule applications may generate new information. Similarly, when a question is posed to the user, the user's response may or may not add new information into the IFISH system. Thus, at an abstract level human users appear to IFISH as another network service, although one with different low-level behavior properties.

The fact that interactions with the user may be viewed as just another network communication by an IFISH suggests that we not attempt to create a *sui generis* system for user-interaction but rather that we try to incorporate it into the IFISH substrate that already exists. We could create an entirely new method for handling user queries (and in fact an early prototype of the system did treat user communication as a completely separate and orthogonal component of the IFISH substrate), but the similarities between user questions and other information requests are compelling.

What information must an object representing a user question contain? Every question must contain information that:

1. Allows the IFISH system to generate a representation of the question that the user can understand, as well as a method for responding to the question, and

2. Tells the IFISH system, depending on the user's response to the question, what action(s) the IFISH should take.

These two items, a method of declaring itself to the user and a method for dealing with the user's response, are the bare minimum requirements for a user question. Without a representation generator the IFISH does not know how to properly pose the question to the user. Without a user-dependent action there's no reason for an IFISH to pose the question in the first place.

Each question in the IFISH system is a structure that contains six pieces of information; Figure 3-1 illustrates a typical question. The first two slots in the question structure, the *sexp-html-proc*

Question slot name	Possible content
sexp-html-proc	Procedure for generating an s-expression HTML description of the question
answer-proc	Procedure for acting upon the user's response
serial-number	A unique identifier for each question
question-maker-name	The procedure that generated this question
question-maker-arguments	The arguments to the procedure that generated this question
question-ichunk	Data object of the infochunk associated with this question.

Figure 3-1: Value slots in the *question* data structure and typical content of each slot.

and *answer-proc*, hold procedures that respectively know how to generate s-expression HTML[48] representations of the question and how to act upon the user's response. The question's *serial-number* is a unique identifier across all questions in a particular IFISH; the *serial-number* is used in conjunction with the *sexp-html-proc* and the IFISH user-interaction code to identify user answers specific to this particular question. The *question-maker-name* and *question-maker-arguments* slots exist so we may regenerate the question at a later time if necessary[49]. *Question-ichunk* is a pointer to the infochunk associated with this particular question.

Questions are generated by IFISH rules just as other infochunks are. When a question-generating rule is applied to an infochunk, the rule generates both the question structure and an associated infochunk. The new infochunk is linked into the infochunk hypertext as before; the question is installed separately into the IFISH user-interaction mechanism, which makes the system aware of the new user question. Questions are presented to the user in order from most to least interesting, where the interestingness of a question is the interestingness of the associated infochunk. This allows the general infrastructure for estimating the interestingness of a particular infochunk (detailed below in Section 3.2) to be applied uniformly to both "regular" and question-related infochunks.

Section 3.1.2 below details the operation of the user-IFISH interaction mechanism. In brief, when the user asks the IFISH for a list of pending (asked but unanswered) questions, the IFISH generates textual representations of each question using the *sexp-html-proc* component of each question structure. The question itself controls the format of the user's response. Thus, when a question is answered, the user's answer is simply passed to the question's *answer-proc*, which evaluates the answer and takes action appropriately. Usually, the action involves modifying the

[48]S-expression HTML is a variant of HTML that is used throughout the IFISH system for communicating with the user. Section 3.1.2 describes the advantages of s-expression HTML in detail.

[49]In particular, these two values are used to regenerate questions after restoring an IFISH from disk. Because Scheme has difficulty dumping procedure objects to disk, we need to be able to store `fasdump`-able representation of pending user questions that allow us to regenerate the question procedures themselves.

particular infochunk that gave rise to the question in the first place, as well as recording the user's answer for future reference.

User questions are treated by the IFISH system exactly like any other piece of information except where absolutely necessary. By incorporating questions into the infochunk structure we leverage all of the existing IFISH system for operating on infochunks and estimating their interestingness. User replies to questions are also incorporated into the permanent infochunk structure. Where questions differ from other infochunks is in how users are asked to answer questions and how user responses are captured by IFISH.

3.1.2 System Support

The IFISH substrate contains a number of special-purpose modules for supporting user-interaction. Roughly speaking we can divide these special functions into three groups:

1. Support for a user-interaction language and/or format,
2. Primitives for creating questions of a particular type or format, and
3. Functions that pose questions to the user and process the user's response.

In the IFISH system all of these functions are closely related to the overall use of the Web as a means of retrieving information. Since IFISH already must "swim" in the Web, and since the Web (as of HTML 2.0) supports user-questions via interactive forms, it seemed logical to use this already-existing infrastructure to ask IFISH questions and receive responses. Also, as there already exist nice graphical user interface clients[50] for the Web, by using the Web as our question-response medium we leverage all the GUI code built into Web clients and need write very little code to control the appearance of the user interface.

Notice that whereas normally IFISH are Web clients, seeking out information that exists on remote Web servers, in the case of user interaction the IFISH itself plays the role of the Web server and it is the user at the other end of a network connection using a Web client. Thus, it was necessary to build into IFISH not only the ability to mimic a Web client (to retrieve information from remote web servers) but also the ability to run a small Web server (to talk to the user's web client). To this end, we first describe s-expression HTML, an HTML variant, and then move on to the subjects of question-generating primitives and the IFISH WWW server.

S-Expression HTML

The HyperText Markup Language (HTML) standard [4] for content on the WWW uses only strings as language elements; every HTML document is a concatenation of strings that contain either content ("Welcome to my homepage") or semantic labels (`<a href=index.html>`). Since HTML depends only on a common character representation it is extremely portable, but that portability requires a lot of string manipulation and parsing. For IFISH, which need to be able to construct "on the fly" many HTML documents (such as user question and response forms), a more friendly internal representation of HTML is desired.

[50] e.g. Netscape

```
<html>
I think the document
<a HREF="http://www.stat-usa.gov:80/">
STAT-USA&#47;Internet Site Economic&#44; Trade&#44; Business Information
</a>
<a HREF="file:///home/bal/thesis/phd/src/tmp/html_960318_5/57874_0.html">
&#91;
local copy</a>
&#93; is relevant. Is it&#63; <br>
<form METHOD="get">
<input TYPE="hidden" NAME="serial_number" VALUE="1">
<select SIZE="1" NAME="answer">
<option VALUE="1" SELECTED> Known user-relevant
<option VALUE="2"> Possibly user-relevant
<option VALUE="3"> Not relevant
</select>
<input TYPE="submit" VALUE="Submit!">
</form>
</html>
```

Figure 3-2: An example HTML document

S-Expression HTML[51] (s-exp HTML) is a variant of HTML that uses Scheme s-expressions instead of strings for its representation language. Documents written in s-exp HTML are maintained in pre-parsed form, which makes it very easy to combine, splice and subdivide content as required. HTML semantic tags are maintained as headed lists in s-exp HTML, and the scope of such tags is precisely the contents of the headed list[52]. Figures 3-2 and 3-3 show a sample HTML document and the corresponding s-exp HTML. Notice that each s-exp HTML tag is itself a list containing both the tag label and attribute-value pairs.

S-exp HTML is currently used by IFISH for all HTML-based interactions with the user. The IFISH WWW server accepts s-exp HTML expressions as input and converts them to HTML strings just before sending requested content to the user's browser; the conversion process also handles certain HTML character translations[53]. Used in conjunction with Scheme's `quasiquote` construct s-exp HTML allows us to create large, structured HTML documents with very compact procedures.

[51] Alan Bawden originally proposed creating an s-expression variant of HTML. The implementation of s-exp HTML detailed here was implemented by Stephen Adams for another Scheme-related project and subsequently incorporated, with slight modifications, into the IFISH system.

[52] In HTML, the scope of some tags is delimited by opening and closing tags, such as and , which delimit content that should be displayed in boldface.

[53] Certain characters appearing in HTML content must be encoded because they have special meaning within HTML. For example, the character < is the open delimiter for HTML tags; used within the body of content it must be encoded as <.

```
(html
 "I think the document "
 ((a (href "http://www.stat-usa.gov:80/"))
  "STAT-USA/Internet Site Economic, Trade, Business Information")
 " ["
 ((a (href
      "file:///home/bal/thesis/phd/src/tmp/html_960318_5/57874_0.html"))
  "local copy")
 "] "
 " is relevant.  Is it?"
 (br)
 ((form (method get))
  ((input (type hidden) (name "serial_number") (value "1")))
  ((select (size 1) (name answer))
   ((option (value 1) (selected)) "Known user-relevant")
   ((option (value 2)) "Possibly user-relevant")
   ((option (value 3)) "Not relevant"))
  ((input (type submit) (value "Submit!")))))
```

Figure 3-3: The HTML document in Figure 3-2 represented in s-exp HTML

Creating Questions

Creating IFISH questions is fairly straightforward, although the IFISH system must do some work to keep the various question-related structures synchronized with each other. There are three types of objects within the question subsystem: *questions*, *question-ichunks*, and *question-makers*. We start with *question-makers* and then proceed to the other two structures.

Question-makers were created to provide an artificial boundary layer at which question information was not lost while still permitting the IFISH data structures to be written to and restored from disk via the Scheme primitives `fasdump` and `fasload` respectively. A *question-maker* is a named procedure within the system, like a rule, that is generally declared by IFISH modules at system load time. *Question-makers* accept as arguments Scheme dump-able objects, such as lists and vectors but not procedures or environments. *Question-makers* may accept any number of arguments.

A *question-maker's* procedure is invoked by name upon a list of arguments via `make-question`. `Make-question` creates new *question* and *question-ichunk* structures (which will be related in the IFISH system) and passes the new *question* structure to the named *question-maker* along with any other arguments. When the *question-maker* finishes it will have filled in all the slots in the *question* structure. `Make-question` then copies data from the *question* into the *question-ichunk*, assigns a unique serial number to this pair of objects, and returns both the *question* and *question-ichunk* structures via Scheme's `values/with-values` multiple-value-return system.

The IFISH system provides a number of primitive question constructors that *question-makers* may call to quickly assemble *sexp-html-proc* and *answer-proc* procedures. These primitives generally accept as arguments an s-exp HTML description and action thunks that correspond to each of the possible answers to the question. For example, the `question/make-yes-no-pieces` procedure accepts an s-exp HTML description of a question, a "yes" thunk and a "no" thunk; it constructs a question containing an HTML form with "yes" and "no" buttons. Each button, when pressed, triggers invocation of the appropriate thunk. Similar procedures exist to construct "choose one from this list" and "choose some from this list" questions.

Once a rule transducer has received the *question* and *question-ichunk* from `make-question`, it must install these two structures in the IFISH system. The *question-ichunk* becomes the *data* slot of a new infochunk[54] which is installed, linked to and announced in the usual manner. (The new infochunk will be linked to the infochunk that triggered the question.) The *question* structure is passed to `question-install!` which adds it to the list of questions for the user maintained by the IFISH's Web server.

The IFISH WWW Server

The IFISH system contains within it a limited-functionality Web server which is used for communicating IFISH information to the user interacting with the IFISH. We chose to implement the server as part of the IFISH process to facilitate easy dynamic access to the contents of an IFISH. Also, the forms capabilities of HTML 2.0 provide an easy way to gain user interaction while leveraging the various GUI Web browsers that already permeate the marketplace.

IFISH run Web server processes within a separate Scheme execution thread; the locking mechanisms built into the Scheme threads library provide synchronization and guarantees of exclusive access to particular data structures where needed. The server itself is basically a tree of *path-handlers*, which are procedures that operate on URL pathnames. *Path-handlers* are linked together in parent-child relationships by path components. When a parent handler is passed a pathname, it extracts the first component[55] and looks to see if that component is the label on any link to any of its children. If it is, then the child handler is recursively invoked on the remainder of the pathname. If no link is named by the extracted component then the parent path handler itself is responsible for handling the request. All path-handlers return s-exp HTML expressions in response to being invoked on a pathname. When a response is ready to be sent to the user, a single call to `shtml->ascii-string` converts the s-exp HTML into string-based HTML which is then written to the Scheme output port connected to the user's browser.

3.1.3 Putting It All Together

We are now ready to create an IFISH rule that takes advantage of user interaction. The best way to understand the interplay between rules, transducers, *question-makers*, *questions*, *question-ichunks*, and the IFISH Web server is to walk through an extended example from the Finder IFISH. Figure 3-4 shows the rule declaration for the rule `KEYWORD/KEYWORD->RELEVANCE-QUESTION`, which is one of the Finder IFISH's heuristics. This rule looks for possible keywords generated[56] by the IFISH

[54]Notice that we cannot use the *question* structure itself as the *data* slot of the new infochunk since the *question* structure contains procedure objects.

[55]Pathnames, as defined in the URL specification [6], are concatenations of path components separated by forward-slashes (`/`).

[56]When the Finder IFISH encounters a known user-relevant infochunk containing a retrieved document it invokes the rule `ARCHITEXT/KNOWN-RELEVANT-DOC->POSSIBLE-KEYWORD-LIST` to generate a small list of possible keywords. (The default size of the keyword list is five.) These keywords are subsequently separated into individual infochunks and each word may potentially trigger a number of keyword-related rules, including calls to various Web search engines. Every keyword generated in this manner is considered initially to be "possibly user-relevant;" the IFISH

that may be related to the user's current interests. The rule generates a question to the user of the form, "I think this keyword is relevant. Do you find it relevant?" This rule is very straight-

```
(DEFINE-RULE 'KEYWORD/KEYWORD->RELEVANCE-QUESTION
  ;; precondition: infochunk must contain a url
  (SIMPLE-TYPE-PRECONDITION
   '(and (known keyword) (possible user-relevant)))
  tdcr/keyword/keyword->relevance-question)
```

Figure 3-4: Rule declaration for `KEYWORD/KEYWORD->RELEVANCE-QUESTION`

forward: any infochunk with both `(KNOWN KEYWORD)` and `(POSSIBLE USER-RELEVANT)` *typeinfo* declarations will trigger rule application. (Keywords derived from a user-relevant document by the Finder IFish are tagged with *typeinfo* `((KNOWN KEYWORD) (POSSIBLE USER-RELEVANT))`[57]. The transducer called by the rule is also fairly trivial; it is shown in Figure 3-5. When in-

```
(DEFINE-TRANSDUCER tdcr/keyword/keyword->relevance-question
  (lambda (infochunk)
    (let ((the-keyword (infochunk/data infochunk)))
      (with-values
          (lambda ()
            (make-question 'qm/keyword/keyword->relevance-question the-keyword))
        (lambda (the-question the-question-ichunk)
          (let ((new-infochunk
                 (make-infochunk the-question-ichunk '((known question)))))
            (ANNOUNCE-AND-LINK new-infochunk))
          (question/install! the-question))))))
```

Figure 3-5: The transducer `tdcr/keyword/keyword->relevance-question`.

voked, the transducer extracts the contents of the infochunk's *data* slot and passes that data to the *question-maker* `qm/keyword/keyword->relevance-question` via the `with-values` construct. When `qm/keyword/keyword->relevance-question` completes, it returns the new *question* and *question-ichunk*. These are then installed, respectively, in the IFish question space and infochunk hypertext.

needs the user's assistance to turn that qualification into either "definitely relevant" or "definitely *not* relevant." Thus the need for a user question.

[57]Recall that *typeinfo* lists may contain an arbitrary number of declarations. In this particular case the IFish knows that the output of the Architext keyword-generating rule is always a keyword, thus `(KNOWN KEYWORD)`, and also knows that the keyword was derived from a `(KNOWN USER-RELEVANT)` document, thus `(POSSIBLE USER-RELEVANT)`.

The final component of this rule is the *question-maker* `qm/keyword/keyword->relevance-question`, shown in Figure 3-6. This procedure extracts some information from IFish infochunks

```
(DEFINE-QUESTION-MAKER 'qm/keyword/keyword->relevance-question
  (lambda (the-question the-keyword)
    (let ((the-infochunk (infochunk/data->infochunk the-keyword)))
      (question/make-one-of-many-dropdown-pieces
       the-question
       `(seq "I think the keyword "
             (b ,the-keyword)
             " is relevant." (br)
             (b "Interestingness: ")
             ,(with-output-to-string
                (lambda () (display (infochunk/interestingness the-infochunk))))
             " Is it?")
       `(((seq "Known user-relevant")
          ,(lambda ()
             ;; in this case, add '(known user-relevant) to the typeinfo
             (add-or-merge-typeinfo! the-infochunk '(known user-relevant))
             ;; since we've changed the structure of the infochunk, we have
             ;; to re-apply already applied rules & look for changes
             (infochunk/changed! the-infochunk)))
         ((seq "Possibly user-relevant")
          ,(lambda ()
             ;; in this case, add '(known user-relevant) to the typeinfo
             (add-or-merge-typeinfo! the-infochunk '(possibly user-relevant))
             (infochunk/changed! the-infochunk)))
         ((seq "Not relevant")
          ,(lambda ()
             ;; in this case, add '(known user-relevant) to the typeinfo
             (add-or-merge-typeinfo! the-infochunk '(not user-relevant))
             (infochunk/changed! the-infochunk))))))))
```

Figure 3-6: The question-maker `qm/keyword/keyword->relevance-question`

and constructs a "choose one of many"-type question. The routine `question/make-one-of-many-dropdown-pieces` turns an s-exp HTML description of the question and a list of 2-lists of possible choices and converts it into an HTML SELECT element. The user is presented with the keyword the IFish derived from a `(KNOWN USER-RELEVANT)` document and is asked to classify the keyword as either "definitely relevant," "possibly relevant" or "definitely *not* relevant." Selecting one of these options installs an event that appropriately updates the *typeinfo* of the keyword-containing infochunk.

3.1.4 Ordering Questions

Deciding which questions to ask the user, and even how and when to ask those questions, is still only part of the story. Not all questions are equal in importance, and an IFish's communication channel with its user is extremely limited. Since the user, our hypothetical "ill-behaved" server from above, may only answer a few of the questions posed to it by the IFish, we want to present the most important questions first. How we determine which questions are "most important" is yet another aspect of determining the "interestingness" of a piece of information. Since questions themselves are IFish infochunks, if we can approximate the intrinsic interestingness of an IFish infochunk we can use that information to order user questions. This leads naturally to the second half of this chapter.

3.2 Interestingness

The final component of our IFish Construction Kit is the subsystem that measures the "interestingness" of an IFish infochunk (or question). What is interestingness? Interestingness is a function that provides the IFish with a *relative* measure of how interesting information *may be* to the user. Heed well the words in emphasis: IFish interestingness makes very weak claims about the information it measures. Interestingness is used to rank infochunks relative to each other, but we must be careful not to put too much weight on the numbers it produces. Similarly, interestingness measures are most useful as predictors of possible user interest. If IFish can make good guesses as to how interested the user will be in particular infochunks, it can make the most of the the IFish-user communication channel. We begin below with a more detailed description of the design goals for interestingness and the minimum required primitive operations that any acceptable IFish interestingness measure must support. Our prototype interestingness implementation is presented in Section 3.2.2.

3.2.1 Design Goals

As in Chapter 2 above, the interestingness portion of the IFish substrate is dictated by a number of design goals and requirements:

- Interestingness need not be a perfect predictor of user interest; it need only be reasonably effective in order to constrain the search for information.

- The interestingness subsystem needs to be extensible, flexible and modifiable by other portions of the IFish system.

- Interestingness must provide a comparison function that is a total order over infochunks.

- The interestingness subsystem must permit use and application of appropriate heuristic knowledge.

- Interestingness functions must have full access to infochunks, including both the data and the hypertext structures within infochunks.

- Like infochunks themselves, interestingness structures[58] need to be dumpable Scheme objects.
- There is a minimal set of operations that the interestingness subsystem must provide to the other portions of the IFish system.

Of these goals the first one is most important. Designing a perfect model of a user's interest in a particular piece of information is itself a foundational question in artificial intelligence. We cannot hope to solve that problem here as part of a single thesis. It is important to realize, however, that we really do not *need* to solve that problem. It is a fundamental assumption of the IFish system that the data space is dynamic and thus total search over that space is not feasible. IFish use interestingness to focus attention on a subset of available tasks and thus constrain the boundaries of its search. If interestingness information successfully directs IFish effort it is useful.

The second design goal for interestingness is that it be as extensible, flexible and modifiable as the rest of the IFish system. If IFish rules are able to rewrite themselves or other rules on the fly, certainly interestingness should be able to as well. This leads naturally to an interestingness implementation similar to IFish rules, which already permit dynamic creation of "interest rules" that score interestingness of an infochunk.

In order to constrain search, IFish must be able to compare the interestingness of two infochunks and decide which of the two is "more interesting" given the current state of the IFish. Thus, the interestingness subsystem must provide a comparison function for interestingness and the "less than/greater that/equal to" trichotomy property. For the prototype implementation of interestingness described in Section 3.2.2 below a *total order* function was implemented for interestingness; the comparison function `infochunk-interest>?` imposes the total order[59] on all pairs of interestingness structures.

Incorporating heuristic knowledge in the IFish interestingness system is as important as in the infochunk portion of the substrate. For example, we may want IFish to consider Web server "home pages" more interesting than other pages[60]. There are also heuristics that can be applied to the structure of the infochunk hypertext itself; infochunks may be more or less interesting because they are linked to other particular infochunks. Thus any interestingness system should permit a similar form of heuristic encapsulation as that used in generating infochunks in the first place. Of course, interestingness functions obviously need access to infochunk structural information in order to use heuristics that depend on that information.

Finally, we must also consider what primitive, interestingness-related operations the implementation must support. Some primitive operations will be dictated by the actual implementation, including procedures to create and modify interestingness structures. Other primitive operations, such as `infochunk-interest>?`, are necessitated by the rest of the IFish system. IFish-required primitives may be divided into four categories:

1. Constructors, accessors, parsers and print procedures for the *interestingness* structure itself.

[58]Recall from Chapter 2 that every infochunk contains within it an interestingness structure. To the infochunk that structure is opaque; it is only useful to interestingness-related routines. That infochunks must be dumpable Scheme objects necessitates the same requirement for interestingness structures.

[59]Note that it is possible to weaken the total order requirement and use a partial order instead, since we only use the order to choose tasks to work on next. However, the fewer pairs of interestingness structures that are related under the partial order, the closer the IFish's task management procedure approaches random selection.

[60]Deciding whether a particular HTML page is a home page is also open to heuristic analysis; home pages generally have either empty path elements in their URLs or paths that end in `index.html`.

2. A comparison operation for the total order imposed upon infochunks.
3. Procedures to compute and recompute an infochunk's interestingness.
4. Support for interestingness-related heuristic rules.

The first category, providing constructors and other basic routines to create and take apart interestingness structures, are dictated for the most part by the data structures used in the implementation. Obviously we need to have a comparison function that implements the total order relation over the set of infochunks; it is sufficient to define only `infochunk-interestingness>?`, but for the sake of convenience an equivalence operation (`infochunk-interestingness=?`) may also be warranted. IFISH routines also need methods to calculate the interestingness of an infochunk and force a recalculation when some element of an infochunk has changed, if the implementation itself does not already transparently provide these services. Finally, just as IFISH provides substrate for transducers and rules, interestingness-related rules also require support functions. The next section provides a detailed description of a prototype implementation of interestingness that is sufficient for the Finder IFISH described in Chapter 4 below.

3.2.2 Prototype Implementation of Interestingness

The prototype IFISH implementation of interestingness closely parallels that of infochunks and rules; here our "infochunks" are *interestingness* data structures and our "rules" are *interest rules.* An interest rule is similar to a rule for heuristic information; every interest rule has a *name*, a *precondition* and an *action.* To compute the interestingness of an infochunk, every interest rule with a precondition satisfied by the infochunk is invoked in sequence. Invoking an interest rule applies the rule's action to the infochunk[61].

The set of possible actions an interest rule may take when invoked is limited to modifications to the interest structure. Figure 3-7 illustrates a typical *interestingness* structure; the structure contains four data slots. Each data slot in the structure contains a list (possibly empty) of (*interest-rule-name*, *value*) pairs, where the *interest-rule-name* is a symbol associated with a particular interest rule and the *value* is an integer quantity.

Interestingness slot name	Possible content
user-slot	An interest-rule-value list, possibly empty.
self-slot	An interest-rule-value list, possibly empty.
forward-links-slot	An interest-rule-value list, possibly empty.
backward-links-slot	An interest-rule-value list, possibly empty.

Figure 3-7: Value slots in the interestingness data structure.

[61] Actually, in the current implementation interest rules take two arguments: an infochunk and an interestingness structure. Interest rules update the interestingness structure explicitly passed to them, not the structure contained within the infochunk. This allows the system to completely recalculate the interestingness of an infochunk without having to worry that an error or other system exception will leave a half-calculated interestingness structure within an infochunk.

Two Scheme procedures are available to interest rules for modifying the contents of a data slot. These rules are named `interest/increment-<slotname>` and `interest/decrement-<slotname>` respectively, where `<slotname>` is any slot in the interestingness structure (e.g. *user-slot*). When an interest rule wants to express an increase in the interestingness of an infochunk, it does so by calling an `interest/increment-<slotname>` procedure of the appropriate slot. For example, Figure 3-8 shows the interest rule `BACKWARD-LINK-TO-RELEVANT`, which increases the interestingness of any infochunk that was derived from an infochunk known to be relevant to the user[62]. The

```
(DEFINE-INTEREST-RULE 'BACKWARD-LINK-TO-RELEVANT
  ;; precondition
  (lambda (infochunk)
    (> (length (infochunk/backward-links infochunk)) 0))
  ;; action
  (lambda (infochunk interest)
    (for-each
     (lambda (the-backward-link)
       (let ((the-quality (infochunk/contains-basetype?
                            (link/infochunk the-backward-link) 'user-relevant)))
         (cond
          ((not the-quality) #f)
          ((eq? (car the-quality) 'known)
           (interest/increment-forward-links-slot! interest *rule-name*))
          ((eq? (car the-quality) 'not)
           (interest/decrement-forward-links-slot! interest *rule-name*)))))
     (infochunk/backward-links infochunk)))
  )
```

Figure 3-8: The interest rule `BACKWARD-LINK-TO-RELEVANT`

precondition for this rule is very simple; any infochunk with a backward link satisfies it. The rule's action increments the *user-slot* once for every backward link to a known user-relevant document and decrements the slot once for every backward link to a document known to be *not relevant* to the user. By default, these incrementing (and decrementing) procedures add (subtract) one to a value contained in the slot; larger values may be passed as optional arguments to override this default.

The interest structure itself is simply a collection of lists containing interest rule names and values (an "interest-rule-value" list in Figure 3-7). When an interest rule action calls a slot incrementer the incrementer looks for an entry in the interest-rule-value list containing the interest rule's name. If there is such an entry, then the associated value is simply incremented as appropriate. If the interest rule's name does not appear in the list, then a cons cell containing the rule and the increment value are added to the list. Figure 3-9 shows an interestingness structure from a sample infochunk. The infochunk's interestingness is currently determined by the interest rules `foo`, `bar` and `baz`.

[62]Note that this rule also decreases the interestingness of an infochunk derived from an infochunk known *not* to be relevant to the user.

user-slot	`((foo . 3))`
self-slot	`()`
forward-links-slot	`()`
backward-links-slot	`((bar . 1) (baz . 1))`

Figure 3-9: An example interestingness structure, including its contents.

There is one component remaining to be defined in order to satisfy the minimal requirements for an IFISH interestingness system: the function imposing a total order upon the set of all possible interestingness structures. For this implementation, we assume that there exists a function $f : I \rightarrow Z$ mapping I, the set of all possible interestingness structures, to Z, the set of integers. Interestingness structures are then compared by comparing the results of applying f to each of the interestingness structures. Since the "greater than" function $>$ is a total order over Z, we have constructed a total order over I[63].

```
(define (interest/interest->number the-interest)
  (let ((user-slot (interest/user-slot the-interest))
        (self-slot (interest/self-slot the-interest))
        (forward-links-slot (interest/forward-links-slot the-interest))
        (backward-links-slot (interest/backward-links-slot the-interest))
        (user-val 0)
        (self-val 0)
        (forward-links-val 0)
        (backward-links-val 0))
    (if (not (null? user-slot))
        (set! user-val (reduce + 0 (map cdr user-slot))))
    (if (not (null? self-slot))
        (set! self-val (reduce + 0 (map cdr self-slot))))
    (if (not (null? forward-links-slot))
        (set! forward-links-val (reduce + 0 (map cdr forward-links-slot))))
    (if (not (null? backward-links-slot))
        (set! backward-links-val (reduce + 0 (map cdr backward-links-slot))))
    (+ (* user-val 100) (* self-val 10) forward-links-val backward-links-val)))
```

Figure 3-10: A simple interestingness evaluation function.

The Scheme code implementing function f is shown in Figure 3-10. Basically, f sums the numerical portions of each interest-rule-value for each slot and compares these "slot summary" values. The constants in f were chosen so that interestingness would be heavily weighted in favor of the *user-slot*, since that slot is where effects related to user-relevance appear in the interestingness structure. The *forward-* and *backward-links* slots are weighted least since these slot values are

[63]Notice that at some level the constructed order function over $I \times I$ violates Claim 2(b), since we are explicitly quantizing the intrinsic interestingness of infochunks. Although we cannot assume the existence of such a function in general, it works well for our Finder IFISH and is easy to implement.

used to record secondary and indirect interestingness effects on infochunks. This definition of `interest/interest->number` allows us to compare two infochunks by numerically comparing the two respective interestingness structures:

```
(define (interest>? interest1 interest2)
  (> (interest/interest->number interest1)
     (interest/interest->number interest2)))

(define (infochunk-interest>? ichunk1 ichunk2)
  ;; grab the interest structs
  (let ((interest1 (infochunk/interestingness ichunk1))
        (interest2 (infochunk/interestingness ichunk2)))
    (interest>? interest1 interest2)))
```

IFISH use `infochunk-interest>?` as the sorting procedure for heaps holding infochunks as well as rank-ordering questions awaiting user attention (see Section 3.1.4 above).

The order function used in the interestingness subsystem is necessarily related to the goals and particular heuristic knowledge of a specific IFISH. An IFISH's set of interest rules defines the possible values that can appear in an interestingness structure. The simple order function just described does not behave differently based on the particular rules that modified the structure, but it would certainly be reasonable for it to do so.

Chapter 4

The "Finder" IFish

4.1 Building an IFish that Finds Web Pages "Like These"

For two chapters we have defined, described and detailed the IFish substrate; now it is time for the payoff. In this chapter we use the various tools built into the substrate to construct a functioning IFish. The demonstration IFish described below was designed to find Web documents that are similar to a set of Web documents provided by the user. That is, the goal of this "Finder" IFish is to solve the "find me more Web pages like these" problem. Such an IFish might be used to keep watch over a set of pointers to various related resources on the Web, or to gather together pages that may be related but spread out all over the network. We assume that the user provides the Finder IFish with an initial set of infochunks, and from only this information, heuristic knowledge and user interaction the Finder attempts to gather together related information.

Our description of the various components of the Finder IFish follows that of the general substrate in Chapters 2 and 3 above; because IFish components are truly "mix-and-match," pieces of the IFish were written as new Web services or new analysis techniques became available. Section 4.2 discusses the two types of heuristic information inside the Finder: heuristics that provide methods of finding new Web pages, and heuristics that are used to analyze the pages already retrieved over the network. When the Finder thinks it has discovered a relevant Web page, it asks the user to view the page and confirm or refute the relevancy assumption. Section 4.3 details this question mechanism. Interestingness rules written specifically for the Finder IFish are outlined in Section 4.4. Finally, Section 4.5 presents an actual user-IFish conversation to find Web pages of a particular type.

4.2 Heuristic Knowledge in the Finder IFish

The initial problems faced in building an IFish are organizational in nature. At least a broad, general picture of the types of modules required and how those modules will interact with one other is needed before one can begin writing those modules. For the Finder there are two classes of heuristic knowledge that we know will be needed from the start. First, the Finder IFish needs heuristics that describe how to find and retrieve information objects over the network, including objects located on remote information servers. This in turn implies that the Finder must be able to use the various monolithic search engines in order to seek out new information sources. Second, after the Finder retrieves documents it needs some mechanism for analyzing those documents and comparing the information contained within to the user-supplied information. Together, these two categories of heuristics will provide the Finder IFish with the ability to:

- Use known relevant information to find new sources of information,
- Retrieve new information over the network from these new information sources,
- Analyze the new information for any that appears related to the known relevant information, and
- Use the new, relevant information to re-seed this process and begin again.

The IFish substrate already provides much of this framework, so long as the Finder IFish is designed to take proper advantage of these facilities. As is the case with many computer programs, a little thought at the beginning over where and how to draw abstraction boundaries will save much effort later on.

4.2.1 Heuristics to Find New Sources of Information

Learning the *lingua franca* of the Web

In order to find new information sources, this IFish must first be able to communicate effectively with remote servers and both send and receives information in mutually-recognized protocols. Recall from Section 1.3.4 the assumption that IFish operate within the scope of the World Wide Web and must be able to swim freely within this medium. Thus, the logical starting point for the Finder IFish is to create the heuristics that will permit it to interface with the rest of the Web[64]. The IFish substrate already includes Scheme routines that provide communication facilities with other Web servers; we thus need only to wrap these routines within IFish transducers and rules.

To begin, consider the most primitive non-trivial element of the Web, the Uniform Resource Locator (URL) [6]. A URL is a pointer to an object somewhere within the Web. Figure 4-1 shows the IFish rule that describes how character strings representing URLs may be converted into URL objects themselves. The transducer `tdcr/string->url` is very simple. First, the transducer extracts the string which is the contents of the *data* slot of the input infochunk. The string is converted to a URL via `url/string->url`, an internal Scheme procedure that implements a parser/scanner conforming to the URL RFC. The resulting URL is then encapsulated within a new infochunk, appropriately tagged as a `(POSSIBLE URL)`[65], and announced and linked into the infochunk hyperstructure. The rule declaration of `URL-STRING->URL` creates a new rule that transforms "url-strings" into URLs simply by invoking the transducer.

Getting a correctly-parsed URL is only the first step in the process of turning a pointer to a Web document into the referenced document itself. Once the IFish system has generated what

[64] It is arguable that the low-level heuristics for interacting with the network should be considered part of the IFish substrate and not of any particular IFish. For the purposes of this chapter's exposition it is illustrative to see how the Web heuristics interact with the other modules.

[65] The new URL is tagged as "possible" instead of "known" because until an attempt is made to actually retrieve the object pointed to by the URL it is not known whether the referenced document actually exists.

```
(DEFINE-TRANSDUCER tdcr/string->url
  (lambda (infochunk)
    (let* ((string (infochunk/data infochunk))
           (result (url/string->url string))
           (new-infochunk (make-infochunk result '((possible url)))))
      (ANNOUNCE-AND-LINK new-infochunk)
      (VALIDATE-TYPEINFO 'url-string)
      )))

(DEFINE-RULE 'URL-STRING->URL
  (SIMPLE-TYPE-PRECONDITION
   '(or (possible url-string) (known url-string))
   (lambda (infochunk) (string? (infochunk/data infochunk))))
  tdcr/string->url)
```

Figure 4-1: The rule `URL-STRING->URL`.

appears to be a valid URL, the obvious next step is to attempt to verify that the URL actually points to something. Verification is accomplished by trying to perform an HTTP HEAD request[66] on the URL; the rule `URL->HTTP-REQUEST-HEAD`[67] accomplishes this task (see Figure 2-5 on page 27). This rule demonstrates some of the automated error-recovery procedures of the IFISH substrate. The actual network request, triggered by the evaluation of `(url/head-url url)`, is performed within a protected Scheme thread that is only permitted to run for a few seconds. If the execution does not complete within that time `raw-result` will be a Scheme error condition object instead of the raw string returned by the remote Web server. A rule transducer application that return an error condition automatically triggers that rule's error handler (in this case the `default-network-error-handler`).

Once the HTTP headers for a particular URL have been successfully retrieved, the IFISH

[66] A HEAD request is like a GET request except that the remote server returns only the HTTP 1.0 headers for the requested object instead of both the headers and the object content. Typical HTTP headers include both MIME information such as "Content-type" and "Content-length" as well as other headers specific to the server. For example, in response to a HEAD request for the URL `http://www-swiss.ai.mit.edu/`, the Web server running on `www-swiss.ai.mit.edu` returns the following header lines:

- Server: Netscape-Communications/1.1
- Date: Wednesday, 03-Apr-96 02:28:06 GMT
- Last-modified: Thursday, 15-Feb-96 22:53:37 GMT
- Content-length: 3911
- Content-type: text/html

[67] The author apologizes for the nomenclature of the `http-request` data structure. One might think that an `http-request` is a data structure containing outbound information that is part of an HTTP GET or HEAD request. Within the IFISH system, however, `http-request` structures contain the *results* of GET and HEAD requests, which follows the naming scheme used by the W3C WWW library reference implementation [45] used as the basis for this code.

considers whether it can retrieve the entire document. Figure 4-2 shows the rule declaration for the `HTTP-REQUEST-HEAD->HTTP-REQUEST` rule; the transducer is essentially similar to the transducer in Figure 2-5 above. Notice how the rule preconditions work together with infochunk content to

```
(DEFINE-RULE 'HTTP-REQUEST-HEAD->HTTP-REQUEST
  (SIMPLE-TYPE-PRECONDITION
   '(known http-request-head)
   (lambda (infochunk)
     (let ((http-request (infochunk/data infochunk)))
       (and (http-request? http-request)
            (= (http-request/status-code http-request) 200)
            (let ((content-type
                   (http-request/lookup-header http-request "Content-type")))
              (and content-type
                   (or
                    (string=? "text/html" (second content-type))
                    (string=? "text/plain" (second content-type)))))
            (let ((content-length
                   (http-request/lookup-header http-request "Content-length")))
              (if content-length
                  ;; if there is a Content-length header, make sure the length
                  ;; is smaller that 1/4th the available heap
                  (< (string->number (second content-length)) (/ (gc-flip) 4))
                  ;; if no Content-length header, go for it anyway (unsafe!)
                  #t))))))
  tdcr/http-request-head->http-request
  default-network-error-handler)
```

Figure 4-2: The rule `HTTP-REQUEST-HEAD->HTTP-REQUEST`

implement heuristic restrictions. The precondition first checks that the HEAD request returned an HTTP status code of 200, meaning that the HEAD request was completed successfully. After verifying the status code the Content-type and Content-length headers are checked for acceptable values[68]; if these checks succeed then the IFISH may invoke the rule's action, which will retrieve the entire document over the network.

In implementing these HTTP-related rules we have consciously represented the process as multiple small heuristic rules as opposed to one big heuristic that might go directly from URL (or even a string representation of a URL) to document contents. The reason for this approach is twofold. First, coding the heuristics this way exposes what might otherwise be internal data structures containing useful information to the rest of the IFISH. Second, by splitting the heuristic

[68]In this particular IFISH we only permit content with MIME content types “text/html” or “text/plain” to be retrieved. This restrictions prevents the IFISH from retrieving non-textual Web pages (e.g. images, sound files, or movies), which is useful at the moment because the IFISH does not have any routines for analyzing non-textual data. Content-length is limited to a fraction of available Scheme heap so as not to exhaust memory during the network transfer; this limitation also restricts the IFISH from grabbing very large documents which tend not to contain text.

into multiple pieces it is easy to add tailored rules for particular special cases at a later date. For example, Figure 4-3 shows the rule `HTTP-REDIRECT-HEAD`. `HTTP-REDIRECT-HEAD` complements the

```
(DEFINE-TRANSDUCER tdcr/http-redirect-head
  (lambda (infochunk)
    (let* ((http-request-head (infochunk/data infochunk))
           (location-entry
            (assoc "Location" (http-request/header-alist http-request-head))))
      (if location-entry
          (let ((redirected-url (second location-entry)))
            (let ((new-infochunk
                   (make-infochunk
                    redirected-url
                    (cons '(possible url-string)
                          (typeinfo/filter-out-basetype infochunk 'http-request-head)))))
              (ANNOUNCE-AND-LINK new-infochunk)
              ))))))

(DEFINE-RULE 'HTTP-REDIRECT-HEAD
  (SIMPLE-TYPE-PRECONDITION
   '(known http-request-head)
   (lambda (infochunk)
     (let ((http-request (infochunk/data infochunk)))
       (and (http-request? http-request)
            (= (http-request/status-code http-request) 302)))))
  tdcr/http-redirect-head
  default-network-error-handler)
```

Figure 4-3: The rule `HTTP-REDIRECT-HEAD`

rule `HTTP-REQUEST-HEAD->HTTP-REQUEST` in Figure 4-2; `HTTP-REQUEST-HEAD` understands how to process HEAD requests that do not complete successfully (status code 200) but instead reply that the requested content has been moved (redirected, status code 302) to another location. Servers that issue redirection pointers include the new location of the content in the HTTP headers; this rule extracts that new location from the headers and generates a new infochunk, labelled with *typeinfo* `url-string`, containing the new content pointer. Since the IFISH system already knows how to work with url-strings, the addition of only this small rule has given IFISH the ability to properly handle HTTP redirect requests.

These HTTP-related rules, together with other heuristics, provide for the IFISH basic network connectivity services. All IFISH network requests are processed in HTTP proxy mode; an HTTP proxy server operating at a known IP address and port is used by the IFISH for all network requests. The proxy server handles multiple URL protocols, including HTTP, FTP and Gopher. Thus, the IFISH need only be able to speak the HTTP proxy protocol to the proxy server in order to gain access to remote HTTP, FTP and Gopher content servers.

Working with Monolithic Search Engines

Now that the IFish has the ability to converse with a vast array of remote information servers, the next challenge is to figure out what to do with that connectivity. Beyond simple document retrieval and extraction of hyperlinks pointing to other documents, the Finder IFish needs heuristics for dealing with the variety of search and indexing engines that are available on the Web. The majority of these servers provide at least keyword searching; some, like Alta Vista, permit more advanced queries such as searches over only the destination URLs of hyperlinks.

We begin with keyword-based queries to search engines such as Lycos, WebCrawler and Infoseek. Each of these services has a unique user interface and thus requires customized heuristics, but the basic structure of the engine-specific heuristics is constant. The Lycos heuristics are a representative example of the customizations required to interface with particular servers. Assume that there exists a sufficiently-interesting infochunk containing a keyword (a string) within its *data* slot. Many search engines, including Lycos, encode the keyword(s) upon which the search is to be based within a URL; retrieving the contents of this URL via the HTTP GET command causes a search on that keyword to be executed dynamically and the results to be returned as the content of the URL. In the case of Lycos, valid search URLs are of the form:

```
http://www.lycos.com/cgi-bin/pursuit?ab=lycos&query=<keyword>
```

where “<keyword>” is replaced by the actual keyword at the base of the search. Figure 4-4 shows the IFish rule that generates a Lycos-specific URL of this form from a generic keyword.

```
(define (lycos/make-search-url keyword)
  (url/string->url (string-append
                    "http://www.lycos.com/cgi-bin/pursuit?ab=lycos&query="
                    keyword)))

(DEFINE-TRANSDUCER tdcr/keyword->lycos-url
  (lambda (infochunk)
    (let* ((keywords (infochunk/data infochunk))
           (lycos-url (lycos/make-search-url keywords))
           (new-infochunk (make-infochunk lycos-url '((known lycos-url)))))
      (ANNOUNCE-AND-LINK new-infochunk)
      )))

(define-rule 'KEYWORD->LYCOS-URL
  (SIMPLE-TYPE-PRECONDITION
   '(known keyword)
   (lambda (infochunk) (string? (infochunk/data infochunk))))
  tdcr/keyword->lycos-url)
```

Figure 4-4: The rule `KEYWORD->LYCOS-URL`

Notice that although the output of this rule is an infochunk with a URL in its *data* slot, the *typeinfo* associated with the URL is not `(KNOWN URL)` but rather `(KNOWN LYCOS-URL)`. This change prevents the IFish from using the generic URL-handling machinery and instead permits use of rules tailored specifically for Lycos results, as shown in Figure 4-5. Because we know that the Lycos server exists and that the constructed search URL is valid we can immediately retrieve the contents of the URL and in doing so perform the search. Furthermore, when we extract HTML anchors from the search results that contain pointers to other Web documents, we specifically exclude any pointer that either (a) appears in the list `lycos/urls-to-exclude` or (b) points to a document residing on the Lycos server itself (`www.lycos.com`). There is certain fixed material that is returned as part of every search, such as pointers to Lycos's hiring opportunities[69], measurements of the size of the current Lycos database[70], and pointers to Lycos searches on keywords alphabetically close to the keyword we submitted. Lycos-specific heuristics prevent the IFish from blindly following these pointers.

```
(define lycos/urls-to-exclude
 (map url/string->url
      '("http://www.pointcom.com/" "mailto:webmaster@lycos.com")))

(DEFINE-TRANSDUCER tdcr/lycos-url->anchors+urls
  (lambda (infochunk)
    (let* ((url (infochunk/data infochunk))
           (raw-result (url/get-url url))
           (http-request (http-raw-result->http-request url raw-result))
           (anchor-url-list (http-request->anchors+urls http-request))
           (result (list-transform-negative anchor-url-list
                     (lambda (x)
                       (or (member (car x) lycos/urls-to-exclude)
                           (string-ci=? (url/host-name (car x)) "www.lycos.com")))))
           (new-infochunk (make-infochunk result '((known list-of (pair url text))))))
      (ANNOUNCE-AND-LINK new-infochunk)
      ;; http-request is about to be GC'd, but the file on disk is not,
      ;; so manually clear the file by invoking set-http-request/body!
      (set-http-request/body! http-request "")
      )))

(define-rule 'LYCOS-URL->ANCHORS+URLS
  (SIMPLE-TYPE-PRECONDITION
   '(known lycos-url)
   (lambda (infochunk)
     (url? (infochunk/data infochunk))))
  tdcr/lycos-url->anchors+urls)
```

Figure 4-5: The rule `LYCOS-URL->ANCHORS+URLS`

Although they do not take advantage of many features of Lycos, the two rules `KEYWORD->LYCOS-URL`

[69] `http://www.lycos.com/lycosinc/jobs.html`

[70] `http://www.lycos.com/sow/TrueCounting.html`

and LYCOS-URL->ANCHORS+URLS are sufficient to make IFISH aware of the Lycos server and be able to use it productively[71]. Similar modules provide keyword-based interfaces between the IFISH and the Yahoo, Infoseek, WebCrawler and Excite[72] search engines. Once the IFISH has discovered a keyword, all of these keyword-based rules will trigger, providing new information to be digested.

Digital's Alta Vista search engine provides simple keyword-based searches of its database, but it also permits more complicated (and, perhaps, more interesting) operations. For the Finder IFISH we include not only keyword-based heuristics (like those for Lycos above) but also a sample set of heuristics that demonstrates IFISH use of the advanced functionality. Alta Vista permits searches over not only the text content of Web pages but also over the textual representations of document URLs and anchor URLs that appear. That is, an Alta Vista search of the form `+link:http://www-swiss.ai.mit.edu/bal/pks-toplev.html` will return all indexed documents that contain pointers matching all or part of the query[73]. This capability is extremely useful for IFISH because it may be used to find Web documents that contain links to a particular document. In particular, the Alta Vista query

```
+link:<the-url> -url:<the-url-hostname>
```

will retrieve every document that contains a pointer to `<the-url>` that is not being served by the same server as `<the-url>`. Figure 4-6 shows the rule that adds this heuristic ability to the IFISH system. Any document known to be relevant to the user will trigger this rule, which in turn queries Alta Vista for documents that point to the known relevant document and are not located on the same server as the known relevant document. Note that there is some complexity that is hidden within the call to `altavista/simple-query`, but that complexity is due solely to the need to parse the results of an Alta Vista search, intended to be read by humans, into an IFISH-friendly format.

The ALTAVISTA/KNOWN-USER-RELEVANT-URL->FIND-REFERENCING-URLS rule is but one of many heuristics that could be written for IFISH to make better use of the full capabilities of the Alta Vista engine; the Finder IFISH includes only a few such heuristics to demonstrate the power and ease of use of the underlying substrate. Similarly, an IFISH programmer could also add numerous routines to improve the interface to other search engines. Of course, should a new service become available while an IFISH is running new rules for the service may be written and installed in the IFISH on the fly, thus making the new service immediately available to the running IFISH[74]. The important issue to realize is that the IFISH substrate provides structure and tools that in turn allow particular heuristics to be quickly coded and installed as part of the IFISH system. How easy it is to write such heuristics varies with every situation. Heuristics to interface to many remote search engines are relatively easy to write, whereas, as we shall see immediately below, analysis heuristics may require more effort.

[71] At the present time, IFISH use only the first page of search results returned by Lycos in response to a search, even if there are many pages of "hits." The Lycos heuristics may be easily extended to sift through all the returned database entries.

[72] Except for Excite, the heuristics for these other services are essentially identical to the Lycos rules presented here. The only significant changes are that the URLs excluded (e.g. URLs that point to `www.lycos.com` in the Lycos case) vary with each site. Excite uses HTTP POST requests for searches instead of GET requests, and thus the Excite-related heuristics were also modified to work with a slightly different protocol.

[73] Alta Vista splits strings on punctuation boundaries, so this search will match documents pointing to `http://www-swiss.ai.mit.edu/bal/` as well as the `pks-toplev.html` file.

[74] In fact, a new search engine wanting to attract users to its service might even make available IFISH-specific heuristics as part of their service. If IFISH knew to look in a standard location for such extensions it could download the new heuristics and modify itself on the fly to talk to the new server.

```
(DEFINE-TRANSDUCER tdcr/altavista/known-user-relevant-url->find-referencing-urls
  (lambda (infochunk)
    (let* ((the-url (infochunk/data infochunk))
           (altavista-simple-query-string
            (string-append
             "+link:" (url/url->string the-url #f) " -url:"
             (url/host-name the-url))))
      (let ((the-result
             (altavista/simple-query altavista-simple-query-string)))
        (if the-result
            (let ((new-infochunks
                   (map (lambda (pair-url-text)
                          (make-infochunk (car pair-url-text) '((possible url))))
                        (altavista-result/result-list the-result))))
             (ANNOUNCE-AND-LINK-MANY new-infochunks)))))))

(DEFINE-RULE 'ALTAVISTA/KNOWN-USER-RELEVANT-URL->FIND-REFERENCING-URLS
  (SIMPLE-TYPE-PRECONDITION '(and (known url) (known user-relevant)))
  tdcr/altavista/known-user-relevant-url->find-referencing-urls)
```

Figure 4-6: The rule `ALTAVISTA/KNOWN-USER-RELEVANT-URL->FIND-REFERENCING-URLS`.

4.2.2 Heuristics to Look For Relationships Among Retrieved Objects

IFISH heuristics for retrieving information across the Web solve only half the problem: once information has been gathered it must also be analyzed. We turn our attention now to the rules built into the Finder IFISH that try to generate new information through study of what has already been retrieved.

The heuristics described in the previous section for finding new sources of information on the Web all depend on either keywords or document URLs in order to generate new information. Keywords known to be relevant to the user are used in conjunction with various search engines to generate new pointers to possibly-relevant documents. URLs pointing to "known relevant" documents generate promising new pointers by locating documents that contain pointers to the relevant document. Therefore, in order for these services to be useful the Finder IFISH needs to be able to deduce either keywords or new documents that the user would consider relevant. The ability to make deductions of this form allows gathered information to automatically generate new information without having to ask constantly for user assistance.

To begin, consider what information may be deduced when an IFISH validates a particular URL `http://host:port/abs_path`, say by performing an HTTP HEAD request on the URL and receiving a "successful completion" response. First, of course, the success of a HEAD request indicates that the URL does indeed reference a particular document. But the success of the HEAD request also indicates that there is an HTTP server running on port `port` on the host `host`. Since web servers generally have a "home page" that they serve when presented with a null path argument (i.e. when `abs_path` is the empty string), when an IFish discovers a new Web site containing possibly-relevant documents it probably wants to also see if the site has a home page in one of the standard home page locations[75]. Furthermore, the `abs_path` component may consist of one or more character sequences separated by forward slashes (e.g. `foo/bar/baz/quux.html`). Typically there is a correlation between the structure of the `abs_path` and the structure of the Web server's underlying filesystem. If so, then an IFish can walk "up" the `abs_path` by repeatedly removing slash-delimited path components and testing the resulting subpaths for existence and contents.

These simple heuristics may help locate more Web documents, but what the IFish really needs are tools for measuring how relevant or similar a retrieved document is, if at all, to documents known to interest the user. Of course, this particular task is a classic information retrieval problem, and IFish may make use of traditional IR methods in evaluating retrieved documents. For the Finder IFish we chose to interface it to Architext [19], an indexing and query engine distributed by the authors of the Excite Web search service. In addition to being freely available, Architext provides both "concept" and "gather" search capabilities for local collections of text and HTML files. Although little information is available concerning Architext's various algorithms (since the engine itself is proprietary), Architext "concept" searching appears to perform some form of vector clustering among documents; "gather" searching appears to implement a scatter/gather algorithm in which a subset of documents are clustered together into a finite number of groups based on pairwise measures of relevance among the documents. As part of the gather process the Architext engine also generates representative keywords for the document cluster. Thus, IFish can use Architext indexes of retrieved documents and subsequent searches to generate possible keywords and also measure document relevance; the information so generated is exactly what the heuristics described in Section 4.2.1 need in order to find new information sources.

Writing IFish heuristics to take advantage of Architext functionality is a little more complicated than previous examples, but the complexity arises mostly from interface issues. The Architext indexer and search engine are "black boxes" as far as the IFish is concerned: the programs are distributed over the Internet as precompiled binaries and thus the interface to both programs is fixed. Furthermore, the output generated by the search engine is intended to be human-friendly, not machine-friendly. Thus, much of the IFish heuristics necessarily deals with constructing Architext queries in their particular format, invoking the indexer and search engine with appropriate inputs to simulate a human user, and parsing the output of the search engine into an IFish-useful format. For example, consider the IFish heuristic `ARCHITEXT/KNOWN-RELEVANT-DOC->POSSIBLE-KEYWORD-LIST`, which attempts to generate relevant keywords for a particular HTML document via Architext's gather function. The rule declaration and transducer are straightforward; Figure 4-7 show the internal IFish function `architext/infochunk->keywords`, which performs the actual computation. After checking that the Architext index is current[76], the infochunk (which in this particular rule contains a relevant HTML document) is converted into an internal document number that Architext uses to name the document. This document number is required by the Architext interface for the "gather" function. Finally, the results of a gather performed on the infochunk contents are parsed to extract the computed keywords.

Although probable keyword extraction is useful, for IFish what is most important about Architext is its ability to rank order documents by how "related" they are to a particular document.

[75]In addition to the "null path" location, home pages are also often served from the path "`index.html`."

[76]Since generating a new index takes time, the IFish only updates the Architext index on demand. The current distribution of Architext does not permit incremental updates of the index.

```
(define (architext/infochunk->keywords infochunk)
  ;; this function only applies to infochunks that contain HTML documents
  (if (not (architext/infochunk-is-html? infochunk))
      '()
      (begin
        ;; first, check that the index is up-to-date
        (guarantee-architext-index!)
        ;; convert the infochunk to the Architext index internal document number
        (let* ((ichunk-filename (http-request/body-filename (infochunk/data infochunk)))
               (a-r (architext/filename->architext-result ichunk-filename))
               (docnum (architext-result/document-number a-r)))
          ;; do the gather operation
          (let* ((raw-gather (architext/raw-gather
                              *architext-current-index* (list '. docnum)))
                 (parsed-gather (car (architext/parse-raw-gather raw-gather))))
            (architext-gather-result/summary-words parsed-gather))))))
```

Figure 4-7: The function `architext/infochunk->keywords`, which computes likely keywords for a particular HTML document.

The prototype IFISH makes significant use of this feature in order to deduce whether particular documents are likely to be found relevant by the user. The rule declaration for

`ARCHITEXT/KNOWN-RELEVANT-DOC->FIND-RELATED-DOCS-EVENT`

is as follows:

```
(DEFINE-RULE 'ARCHITEXT/KNOWN-RELEVANT-DOC->FIND-RELATED-DOCS-EVENT
  (SIMPLE-TYPE-PRECONDITION
   '(known user-relevant)
   architext/infochunk-is-html?)
  tdcr/architext/known-relevant-doc->find-related-docs-event)
```

Every `(KNOWN USER-RELEVANT)` document triggers this rule, which in turn invokes the transducer[77]

```
(DEFINE-TRANSDUCER tdcr/architext/known-relevant-doc->find-related-docs-event
  (tdcr-helper/relevant-docs->related-docs
   'ARCHITEXT/KNOWN-RELEVANT-DOC->FIND-RELATED-DOCS-1
   'ARCHITEXT/KNOWN-RELEVANT-DOC->FIND-RELATED-DOCS-2))
```

[77] The transducer is parameterized because an almost-identical transducer is used in a mirror-image rule that looks for documents related to a document declared to be `(NOT USER-RELEVANT)` by the user. Such documents as identified by the mirror-rules are negatively scored as appropriate.

The heart of the analysis in captured by `tdcr-helper/relevant-docs->related-docs`. When called, `tdcr-helper/relevant-docs->related-docs` first creates a *thunk* (a Scheme procedure of zero arguments) which encapsulates a computation to be performed at a future time. Invoking the thunk performs the encapsulated computation; in this case the invoked thunk issues an Architext concept query requesting that documents be ranked relative to the particular document that was contained within the infochunk passed as an argument to the rule. The created thunk is used to build a recurring system event, and this event is then installed into the running IFish system as the rule's last action. Thus, although this rule is only run once for each known relevant document, when run the rule modifies the running IFish system to periodically re-query the Architext index and recompute for each document how "related" it is to the known relevant document. The IFish needs to periodically update earlier relationships deduced by this rule since the Architext ranking of documents changes as the underlying index changes.

Using the information generated by an Architext concept search is relatively straightforward. Let D be an HTML document retrieved by the IFish and let I_D be the IFish infochunk that contains D within its data slot. Assume that the user has declared that D is `(KNOWN USER-RELEVANT)`, either by directly modifying IFish data structures or in response to an IFish question[78] about D. Performing an Architext concept query on D returns a list of all documents within the Architext index sorted by relevance to D. Table 4.1 shows the output of a typical Architext concept query. The "rating," scaled to the range 0–100, measures how related a particular document is to document D. In addition to the relevance scores Architext also groups documents into rougher categories, represented by the "group" number. The IFish uses these group numbers to establish relationships among HTML document infochunks. Any document D' that is listed in group 1 causes infochunk I'_D to be linked to I_D with the label `ARCHITEXT/KNOWN-RELEVANT-DOC->FIND-RELATED-DOCS-1`; group 2 documents are similarly linked with link label `ARCHITEXT/KNOWN-RELEVANT-DOC->FIND-RELATED-DOCS-2`. As discussed in Section 4.4 below, these links trigger IFish interest rules that make highly-related documents more interesting.

These Architext-based heuristics, together with the other heuristics outlined above, provide the Finder IFish with both the means of acquiring new pieces of information and also some basic routines for making educated guesses as to the relevance of those pieces of information to the overall search goal. In the next two sections we shift focus to the Finder IFish's ability to ask specific questions of the user and also the specific interest rules added to the IFish substrate to make use of the additional information added by these heuristics.

4.3 Querying the User to Refine the Search

In the previous section relatively few heuristics provided the Finder IFish with abilities both to gather new information sources over the web and also to perform some primitive analysis of the usefulness of collected information. Yet these functions demonstrate only one-half of the capabilities of the IFish substrate. Our focus shifts now to the user-interaction portions of the IFish Construction Kit and how the Finder IFish makes use of these facilities. At a minimum, IFish must be able to communicate to the user what relevance assumptions it has made and ask the user to confirm or refute its guesses. The user need not answer these questions for the IFish to

[78] Section 4.3 below discusses IFish questions concerning document relevance that may be generated by the Finder IFish.

Doc. #	Rating	Group	Filename	Document Title (`<title>`...`</title>`)
39	99	1	21467_0.html	Avant Garde: a virtual marketplace
175	96	1	7300_0.html	Information Analytics: 4W.COM
173	94	1	7036_0.html	makepage.pl
28	92	1	17518_0.html	Telebyte NW - Hot and Cool
169	92	1	64642_0.html	TRAVEL - Internet Travel Network...
51	91	1	25369_0.html	INTERNET ENTERTAINMENT NETWORK
132	91	2	48920_0.html	Star City Mall: Lincoln, Nebraska USA
160	91	2	60579_0.html	Travel & Entertainment Network (TEN-IO) home page
164	91	2	61784_0.html	Nebraska Investment Finance Authority
60	90	2	29188_0.html	University of Illinois at Urbana-Champaign UIUC
77	90	2	3246_0.html	Hoagie's Homepage
172	89	2	6537_0.html	The Prairie Astronomy Club
41	88	2	22451_0.html	The Internet Movie Database FAQ
13	88	3	13643_0.html	Virtual Adventures Domain
71	87	3	31349_0.html	Welcome to Lightside
0	87	3	10001_0.html	Entertainment
45	87	3	23986_0.html	Internet Address Finder
73	87	3	31410_0.html	Griffith University Welcome Page
163	86	3	61649_0.html	Nebraska Investment Finance Authority
14	86	3	13879_0.html	Sun Microsystems
54	86	3	25956_0.html	Puzzle Depot: Marketing Services
62	86	3	29778_0.html	Copyright
166	86	3	62627_0.html	The Film Festivals Server

Table 4.1: Results of an Architext concept search. The listed filename is the IFISH-generated filename of a locally-cached copy of the document. The document title is the HTML-tagged title in the document, if one exists.

continue working, but as the user answers more questions accurately the IFish can dynamically modify itself to more closely match the user's stated interests.

The Finder IFish is able to ask the user three types of questions: questions related to IFish errors, questions related to automatically-generated keywords, and questions related to HTML documents that the Finder thinks are relevant. Questions of the first type, pertaining to IFish errors, are generated either by the default IFish error handler, as discussed in Section 2.4.3, or by rule-specific error handlers. Figure 2-8 shows the Scheme code that generates questions for the default error handler. When an IFish encounters an error during rule application the rule, input infochunk and error condition are bundled up into a “yes/no” question that is installed into the IFish system.

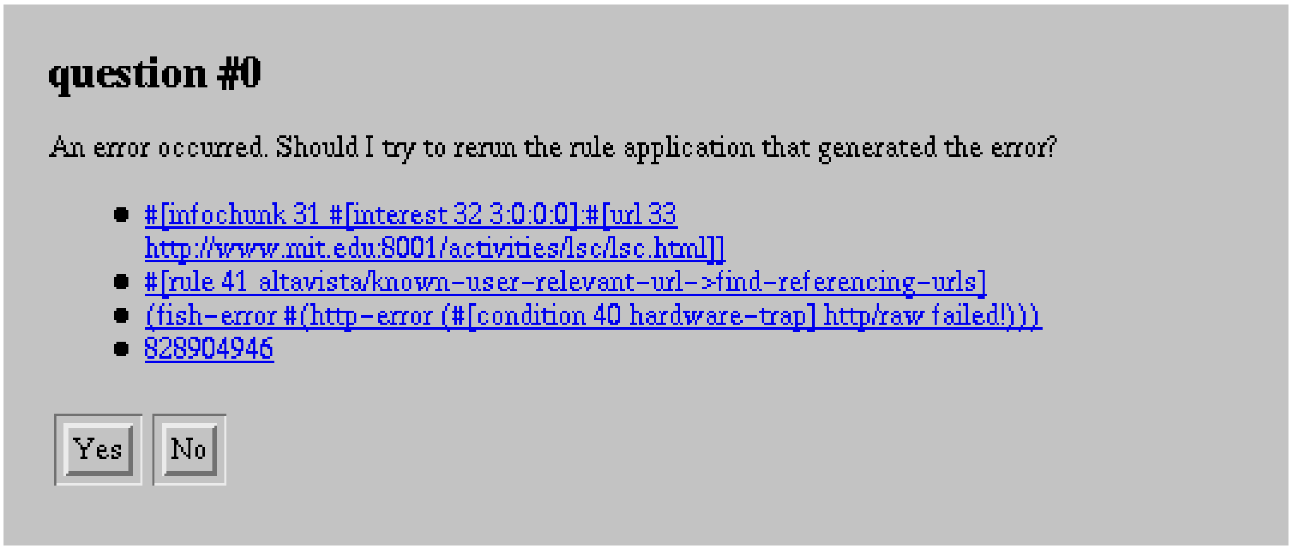

Figure 4-8: A sample error-handler question.

Figure 4-8 shows how these “yes/no” error handler questions appear to the user. In addition to providing a means of responding to the question (via buttons labelled “yes” and “no”), the question itself is hypertext linked to the the rule, infochunk, and error condition that were involved in the error. This allows the user to gather further information and take the details into account before answering the question. Upon receiving an affirmative response to this question, the IFish will retry the rule application; a negative response removes the question from the queue and the IFish will not pursue the rule application further.

Keyword questions posed by the IFish are a little more complicated than error questions; the code the Finder IFish uses to generate keyword-related questions was presented as the comprehensive example in Section 3.1.3. Any keyword tagged with *typeinfo* `(POSSIBLE USER-RELEVANT)` generates a user question asking for confirmation or refutation of the “user relevant” *typeinfo* claim. Since all keywords extracted via Architext processing from known relevant HTML documents are so tagged, every such keyword generates a user question. How relevant a keyword is to the user impacts on that keyword's interestingness, which in turn affects when the IFish will focus attention on that keyword and perform a number of keyword-based search engine queries. A typical keyword question, at it appears to the user, is shown in Figure 4-9.

Figure 4-9: A sample keyword-related question.

The most interesting questions generated by the Finder IFISH are those challenging the relevance of a particular document. Recall from Section 4.2.2 above the heuristic `ARCHITEXT/KNOWN-RELEVANT-DOC->FIND-RELATED-DOCS-EVENT`, which dynamically modifies the IFISH by installing a recurring event into the system. This event, invoked periodically, searches for already-retrieved documents that seem to be related to a particular "seed" document. When this process believes it has found a related document, it links that related document's infochunk to the infochunk containing the seed document. The label on the link between the two infochunks depends upon how related the IFISH thinks the two documents are[79], and the presence of such a link in turn triggers a question. Figure 4-10 shows the rule declaration for the rule `ARCHITEXT/RELATED-DOC->RELEVANCE-QUESTION`. This rule looks for HTML documents with links generated by the Architext relevance-matching heuristic and generates a question to the user of the form, "I think this document is relevant. Do you find it relevant?" Notice that this rule has a non-trivial precondition; infochunks must satisfy all of these properties for the rule to apply to them:

- The infochunk must contain an HTML document.
- The infochunk must not contain types `(KNOWN USER-RELEVANT)` or `(NOT USER-RELEVANT)` (we don't want to ask the user a redundant question).
- The infochunk must contains a backward link labelled either
  - `ARCHITEXT/KNOWN-RELEVANT-DOC->FIND-RELATED-DOCS-1`, or
  - `ARCHITEXT/KNOWN-RELEVANT-DOC->FIND-RELATED-DOCS-2`.

[79] The two possible link labels, `ARCHITEXT/KNOWN-RELEVANT-DOC->FIND-RELATED-DOCS-1` and `ARCHITEXT/KNOWN-RELEVANT-DOC->FIND-RELATED-DOCS-2`, represent the Architext-generated groups of relevant documents.

```
(DEFINE-RULE 'ARCHITEXT/RELATED-DOC->RELEVANCE-QUESTION
  (lambda (infochunk)
    (and (architext/infochunk-is-html? infochunk)
         ;; this piece makes sure we don't consider anything
         ;; that's either (known user-relevant) or (not user-relevant)
         (let* ((typeinfo (infochunk/contains-basetype? infochunk 'user-relevant))
                (the-quality (and typeinfo (car typeinfo))))
           (if the-quality
               (and (not (eq? the-quality 'known))
                    (not (eq? the-quality 'not)))
               #t))
         (not
          (null?
           (list-transform-positive (infochunk/backward-links infochunk)
             (lambda (link)
              (or (eq? (link/rule-name link)
                       'ARCHITEXT/KNOWN-RELEVANT-DOC->FIND-RELATED-DOCS-1)
                  (eq? (link/rule-name link)
                       'ARCHITEXT/KNOWN-RELEVANT-DOC->FIND-RELATED-DOCS-2))))))))
  tdcr/architext/related-doc->relevance-question)
```

Figure 4-10: Rule declaration for `ARCHITEXT/RELATED-DOC->RELEVANCE-QUESTION`

When triggered, this rule invokes the transducer `tdcr/architext/related-doc->relevance-question`, which is essentially the same as `tdcr/keyword/keyword->relevance-question` in Figure 3-5 above. The user question itself is generated and installed in the IFish system by the *question-maker* `qm/architext/related-doc->relevance-question`. Each "related" infochunk generates a question asking the the user to declare the document "known [to be] user-relevant," "possibly user-relevant," or "not relevant." Figure 4-11 shows how a typical question appears to the

Figure 4-11: A sample question generated by `qm/architext/related-doc->relevance-question`.

user. If the user chooses to answer the question, the infochunk's *typeinfo* declarations are updated as appropriate. In this case, were the user to declare the document relevant (i.e. adding `(KNOWN USER-RELEVANT)` to the infochunk's *typeinfo*), that change would satisfy rule preconditions that were previously false and the infochunk would be re-scheduled for further processing. Notice that the user has the option of confirming the relevance of a document or declaring the document not relevant. Both statements are important pieces of information, for documents that closely correlate with "not relevant" documents are themselves not as likely to be relevant to the user. As mentioned in footnote 77 on page 59 above, the Finder IFish contains mirror-image heuristics for dealing with HTML documents known to be not relevant to the user. These rules, together with their interest rule counterparts that appear in Section 4.4 below, act to account for both positive and negative information.

Together these error questions, keyword questions and document questions create a rudimentary conversation between the Finder IFish and the user. The IFish poses questions as a means of evaluating its own judgements and interestingness measures. As the user answers more questions the self-modifications that the IFish makes in response to each question further refine the IFish's vision of what the user wants. Note that in the case of the Finder IFish actions taken in response to answered questions do not directly modify IFish rules or interest rules. Rather, the actions modify the *typeinfo* associated with particular infochunks and, as we shall see in the next section, that change in type information in turn modifies the behavior of interest rules. When interest rules change the IFish model of interestingness itself is changed; thus the user's answers to questions indirectly cause the IFish to change its perception of all the infochunks in the system.

4.4 Approximating Interestingness of Web Pages

The last component of the Finder IFISH we describe is the set of application-specific interest rules used to estimate the "interestingness" of particular infochunks. These interest rules supplement the general rules of Section 3.2.2, which represent interestingness that derives from the data structures the IFISH constructs internally (as opposed to interestingness that derives from the current operating environment of the IFISH). Application-specific interest rules in the Finder IFISH fall into two categories: rules that relate to the structure of the Web, and rules that deal with information generated by Architext.

To begin, recall from Section 4.2.1 that the IFISH contains numerous heuristics for working efficiently over the Web. Although not explicitly mentioned above, the Finder IFISH contains two heuristics aimed at finding relevant "home pages." Every time it encounters a validated HTTP URL (i.e. a URL known to point to a document on an accessible HTTP server), the Finder IFISH extracts from that URL the protocol, host name and port of the remote server and encapsulates *that* information in its own infochunk together with the *typeinfo* declaration `(KNOWN WEBSERVER)`. That is, if the IFISH has verified via HTTP that the URL

`http://www-swiss.ai.mit.edu/bal/pks-toplev.html`

points to an accessible Web document, then the IFISH knows that the host `www-swiss.ai.mit.edu` is running an HTTP server on port 80[80]. The infochunk containing the server-related information later triggers a rule that constructs likely "home page" URLs for that particular server, and these URLs are ultimately announced to the IFISH system; if the IFISH finds these URLs sufficiently interesting it will attempt to verify them by testing for the existence of the document pointed to by the URL.

```
(DEFINE-INTEREST-RULE 'NULL-PATH-URL
  ;; precondition
  (lambda (infochunk)
    (let ((the-typeinfo (infochunk/contains-basetype? infochunk 'url)))
      (and the-typeinfo
           (eq? (car the-typeinfo) 'known)
           (or (string=? (url/path (infochunk/data infochunk)) "/")
               (string=? (url/path (infochunk/data infochunk)) "/index.html")
           ))))
  ;; action
  (lambda (infochunk interest)
    (interest/increment-self-slot! interest *rule-name*)))
```

Figure 4-12: The interest rule `NULL-PATH-URL`.

Given two documents identical except for their location on a particular Web server, we want

[80]Port 80 is the default port for HTTP servers and is assumed in URLs that do not specify a port number.

the Finder IFish to consider "home page" documents slightly more interesting. The act of placing a particular document at the root of a server indicates some level of wide applicability or relevance of that particular document relative to the other documents around it on the server. Causing the IFish to evaluate "home page" URLs in a slightly more favorable light is easy, as demonstrated by the interest rule shown in Figure 4-12. This rule applies to all verified URLs (i.e. with *typeinfo* `(KNOWN URL)`) with URL paths that are either empty ("/") or contain exactly "/index.html". The interest rule's action simply adds a small-value tag `(NULL-PATH-URL 1)` to the infochunk's interestingness *self-slot*. All other things being equal, an infochunk satisfying this rule's precondition will be favored slightly over other infochunks that do not.

```
(DEFINE-INTEREST-RULE 'WEBSERVER
  ;; precondition
  (SIMPLE-TYPE-PRECONDITION '(known webserver))
  ;; action
  (lambda (infochunk interest)
    (let ((back-links (infochunk/backward-links infochunk)))
      (interest/increment-self-slot!
       interest
       *rule-name*
       (let* ((the-interest-slots
               (map (lambda (the-link)
                      (interest/self-slot
                       (infochunk/interestingness
                        (link/infochunk the-link))))
                    back-links))
              (collapsed-slots
               (map
                (lambda (a-self-slot)
                  (cond
                   ((null? a-self-slot) 0)
                   (else
                    (reduce + 0 (map cdr a-self-slot)))))
                the-interest-slots)))
         (reduce max 0 collapsed-slots)
         )))))
```

Figure 4-13: The interest rule `WEBSERVER`.

Moving up the complexity scale a little bit, consider next possible interest rules for the `(KNOWN WEBSERVER)` data structures. How should the Finder IFish evaluate the interestingness of a particular Web server? For now, the interestingness of a Web server is a function of the most interesting URL pointing to a document that resides on that server. While this particular measure fails to take into account the distribution of multiple interesting URLs on a single server, it demonstrates how interestingness may be derived from the collective interestingness of an infochunk's neighbors. Figure 4-13 shows the Scheme code for the interest rule `WEBSERVER`. For each infochunk that is the source of a pointer to the "Web server" infochunk[81], the rule computes the sum of the infochunk's *self-slot* interestingness declarations. The "Web server" infochunk's interestingness *self-slot* is then incremented by the maximum of all these calculated values.

These two interest rules provide some interestingness information to the Finder IFISH, but the interest rules that really allow the Finder to separate the information wheat from the chaff are those that work with the various Architext-based heuristics. Recall from Section 4.2.2 above that the recurring events installed in the IFISH system by the rule `ARCHITEXT/KNOWN-RELEVANT-DOC->FIND-RELATED-DOCS-EVENT` periodically create links between HTML-document infochunks based on relationships suggested by the Architext search engine. These links record relationships between known relevant documents and related documents that may themselves be relevant to the user. Thus, whether an infochunk is connected to a relevant document by such a link, as well as the type of the link itself, has implications for the interestingness of the infochunk.

```
(DEFINE-INTEREST-RULE 'ARCHITEXT/KNOWN-RELEVANT-DOC->FIND-RELATED-DOCS-1
  ;; precondition -- HTML docs only
  (lambda (infochunk)
    (and (architext/infochunk-is-html? infochunk)
         (not (null? (list-transform-positive (infochunk/backward-links infochunk)
                        (lambda (link)
                          (eq? (link/rule-name link)
                               'ARCHITEXT/KNOWN-RELEVANT-DOC->FIND-RELATED-DOCS-1)))))))
  ;; action
  (lambda (infochunk interest)
    (interest/increment-user-slot! interest *rule-name* 3)))

(DEFINE-INTEREST-RULE 'ARCHITEXT/KNOWN-RELEVANT-DOC->FIND-RELATED-DOCS-2
  ;; precondition -- HTML docs only
  (lambda (infochunk)
    (and (architext/infochunk-is-html? infochunk)
         (not (null? (list-transform-positive (infochunk/backward-links infochunk)
                        (lambda (link)
                          (eq? (link/rule-name link)
                               'ARCHITEXT/KNOWN-RELEVANT-DOC->FIND-RELATED-DOCS-2)))))))
  ;; action
  (lambda (infochunk interest)
    (interest/increment-user-slot! interest *rule-name*)))
```

Figure 4-14: Interest rules for `ARCHITEXT/KNOWN-RELEVANT-DOC->FIND-RELATED-DOCS-EVENT`.

The two interest rules shown in Figure 4-14 together create a dependency between the links generated by the recurring Architext queries and the interestingness of the infochunks so linked. Each link between infochunk I_R known to be relevant and infochunk I containing a possibly-relevant HTML document increments the *user-slot* of I's interestingness structure. Links indicating that

[81] Equivalently, these infochunks are the set of destination infochunks defined by the "Web server" infochunk's backward links.

I is contained within I_R's "group 1[82]" (most related group) of infochunks increments I's *user-slot* by three[83]; "group 2" links increment I's *user-slot* by one. Since the contents of an interestingness structure's user slot is most heavily weighted by the function `interest/interest->number` (Section 3.2.2), these rules serve to quickly bring infochunks related (in the Architext sense) to known relevant documents to the attention of the IFish and ultimately the user[84].

One final interest rule completes the set of application-specific rules for the Finder IFish. Recall that user questions themselves have representative infochunks in the IFish system, and the order in which questions are presented to the user is based on the interestingness of those associated infochunks. We want Architext-related questions generated by the prototype IFish (described in Section 4.3 above) to be as interesting as the particular documents they reference; that way the more interesting the IFish thinks a particular HTML document is the greater the importance of any Architext-related question. Thus we need an interest rule that will cause Architext-related question infochunks to inherit their interestingness from the "parent" infochunk (which contains the actual document). The interest rule in Figure 4-15 performs this task automatically.

```
(DEFINE-INTEREST-RULE 'ARCHITEXT/RELATED-DOC->RELEVANCE-QUESTION
  ;; precondition -- only questions
  (lambda (infochunk)
    (and (question-ichunk? (infochunk/data infochunk))
         (eq? (question-ichunk/question-maker-name (infochunk/data infochunk))
              'qm/architext/related-doc->relevance-question)))
  ;; action -- inherit interestingness from parent
  (lambda (infochunk interest-struct)
    (let ((the-back-link
           (car
            (list-transform-positive (infochunk/backward-links infochunk)
              (lambda (link) (eq? (link/rule-name link)
                                  'ARCHITEXT/RELATED-DOC->RELEVANCE-QUESTION))))))
      (let ((parent (link/infochunk the-back-link)))
        (infochunk/inherit-interestingness parent interest-struct)))))
```

Figure 4-15: The interest rule `ARCHITEXT/RELATED-DOC->RELEVANCE-QUESTION`.

With the specification of the `ARCHITEXT/RELATED-DOC->RELEVANCE-QUESTION` interest rule we have completed our description of the various components of the Finder IFish. In the next (and final) section of this chapter we "test-drive" the Finder and provide an example sessions of a user working with the it.

[82]Architext grouping constructs are described in Section 4.2.2 on page 58.

[83]The values three and one were chosen arbitrarily by the author so that the IFish would favor but not work solely on "group 1" links. Such constants would hopefully be chosen and modified in future IFish by the IFish themselves as part of its overall self-modification abilities.

[84]Similarly, the mirror-rules for "not relevant" documents have mirror-interest-rules that cause documents related to "not relevant" documents to be buried deep within the IFish system and make it unlikely that the IFish will ever expend significant effort on them.

4.5 A Session with the IFISH

Now that we have defined the initial rule set for the Finder IFISH it is time to see how well the tool works. To begin, we must create an instance of the Finder's input problem; that is, we need a set of related Web pages for which we want to find similar pages. Once we have the initial set we can start the IFISH running and periodically check its status.

Our sample problem for this IFISH session was kindly provided by Michael ("Ziggy") Blair. Blair's research currently involves partial evaluation and run-time code generation [7] and he has collected some URLs for "partial evaluation" pages that are relevant (in his opinion) to his own research. The Finder IFISH was seeded with five such URLs, as shown in Figure 4-16. In this

`http://www-swiss.ai.mit.edu/~ziggy/descartes.html`
Document title: Ziggy's Descartes Page at MIT – HTML Version

`http://www.cs.cmu.edu:80/afs/cs.cmu.edu/user/wls/www/sbpm/people.html`
Document title: Semantics Based People

`http://www.irisa.fr/EXTERNE/projet/lande/consel/overview.html`
Document title: PE Group - Overview

`http://www.irisa.fr/EXTERNE/projet/lande/Lande_anglais.html`
Document title: Les projets

`http://www.cs.washington.edu/homes/pardo/rtcg.d/index.html`
Document title: Runtime Code Generation (RTCG)

Figure 4-16: Seed URLs for the Finder IFISH.

experiment, the seed URLs were introduced into the Finder IFISH by a set of Scheme infochunk declarations (see Figure 4-17); a Web-based form communicating with the Finder's Web server could also have been used.

Once the IFISH was initialized, it was released into the Internet and allowed to proceed without interruption or any user interaction for about an hour. During that time the IFISH generated many questions, primarily document relevancy queries created by the rule `ARCHITEXT/RELATED-DOC->RELEVANCE-QUESTION`. After this first hour of interaction the IFISH was "retrieved from the sea" for a period of user interaction. The Finder had already identified approximately one hundred documents; Figure 4-18 shows the state snapshot presented to the user. Every discovered HTML document is listed, ordered by the interestingness of the IFISH infochunk containing the document. The label `[KNOWN]` is added to any document with a *typeinfo* declaration of `(KNOWN USER-RELEVANT)` contained within the *typeinfo* slot of the document's enclosing infochunk. Thus, since no user-interaction occurred between the initial seeding of the IFISH and the time when this snapshot was taken, the only documents known to be relevant are the initial seeds.

In addition to the ordered list of HTML documents, the IFISH also presents an ordered list of questions. The questions are also ranked by interestingness; in the Finder IFISH the interest rule in Figure 4-15 above sets the interestingness for document-relevancy questions to be that of the

```
(define *seed-urls*
  '("http://www.cs.washington.edu/homes/pardo/rtcg.d/index.html"
    "http://www.irisa.fr/EXTERNE/projet/lande/Lande_anglais.html"
    "http://www.irisa.fr/EXTERNE/projet/lande/consel/overview.html"
    "http://www-swiss.ai.mit.edu/~ziggy/descartes.html"
    "http://www.cs.cmu.edu:80/afs/cs.cmu.edu/user/wls/www/sbpm/people.html"
  ))

(for-each
 (lambda (url)
   (let ((ichunk (make-infochunk
                  (url/string->url url)
                  '((known url) (known user-relevant)))))
     (infochunk/recompute-interestingness! ichunk)
     (announce-new-infochunk ichunk)))
 *seed-urls*)
```

Figure 4-17: Infochunk declarations for the seed URLs.

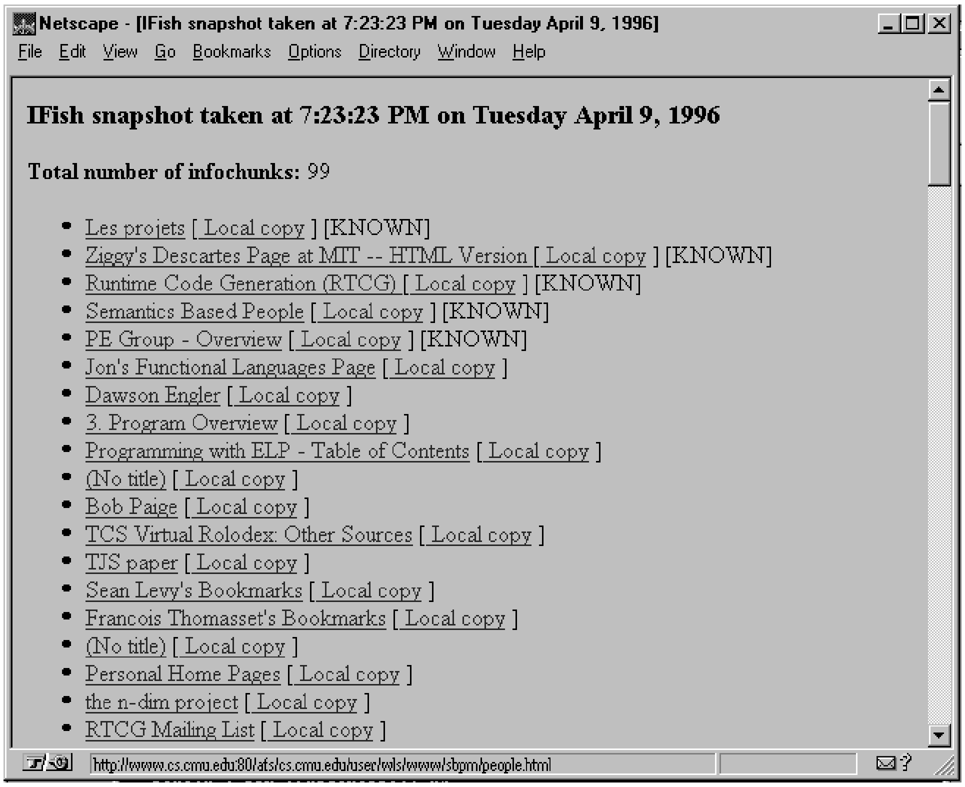

Figure 4-18: Discovered documents ranked by interestingness.

HTML document to which the question pertains. The top two questions are shown in Figure 4-19. Notice that in both the document relevancy list and the question list hypertext links are included to both the source where the IFISH found the document and also a locally-cached copy. Thus the user can immediately "click-through" the links on a question to evaluate and rate the questioned document.

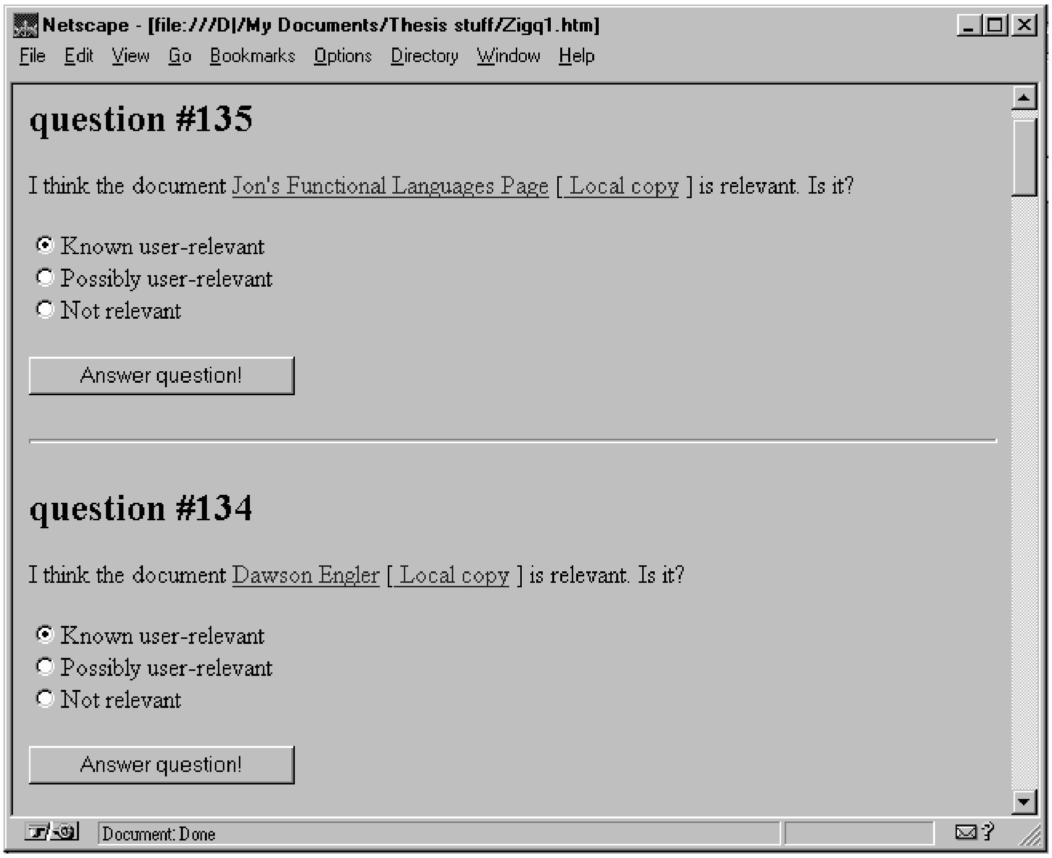

Figure 4-19: User questions generated by the Finder IFISH.

To continue the demonstration, Blair evaluated and answered the top sixteen questions, all of which asked him to rate the relevance of a particular document. The top discovered documents, along with Blair's rankings, are shown in Table 4.2 below. Each question presented three possible options: "relevant," "not relevant," and "possibly relevant." An up arrow (↑) indicates that the document was ranked "relevant;" "not relevant" is designated by a down arrow (↓) and "possibly relevant" by a sideways-pointing arrow (↔).

Two of the ratings made by Blair require explanation and highlight some of the limitations of the current Finder IFISH interface. The document titled "3. Program Overview" and marked with a † symbol is a portion of the U.S. Government's implementation plan for the High Performance Computing and Communications Program (HPCC) for fiscal year 1995[85]. This document did not contain technical content that was related to "partial evaluation" but did contain information

[85]The URL for the discovered document is `http://www.hpcc.gov/imp95/section.3.html`.

Document Title	Rating
Jon's Functional Languages Page	↑
Dawson Engler	↑
3. Program Overview†	↑
Programming with ELP - Table of Contents	↔
(No title)	↔
Bob Paige	↑
TCS Virtual Rolodex: Other Sources	↑
TJS paper	↑
Sean Levy's Bookmarks	↔
Francois Thomasset's Bookmarks	↓
(No title)	↓
Personal Home Pages‡	↓
the n-dim project	↓
RTCG Mailing List	↑
Glen Weaver's Compiler Related Links Page	↑
P. Cousot	↑

Table 4.2: User evaluation of top documents found by the Finder IFish.

related to possible funding opportunities in that field. This is one example of serendipitous resource discovery, as the Finder IFish found something interesting yet unexpected. The Finder interface does not currently allow the user to create multiple categories of relevancy, thus if the user rates the document "known relevant" (as was done in this case) the Finder will assume it to have equal weight and applicability as other "known relevant" documents. The document[86] marked with a ‡ symbol, titled "Personal Home Pages," was another example of "interesting yet unexpected" information. In his own words, Blair said that this document led to "another page that contained information and hyperlinks related to *Babylon 5* and *The X-Files*, which while not related to partial evaluation were still very interesting to the user."

Another limitation of the current Finder interface is the inability to rate the relevancy of portions of retrieved documents. For example, the document[87] "Jon's Functional Languages Page" is an organized collection of hyperlinks to other documents related to functional languages. Within that collection were certain pointers and groupings that were closely related to Blair's thesis research, but there were also many other pointers not so closely related. There is no way to communicate such information back to the Finder IFish within the current user-interaction model, or to otherwise restrict user statements to portions of documents. This limitation could be remedied within the IFish framework by expanding the set of allowable responses to document-relevancy questions, or by allowing ratings of particular URLs within a given document.

After Blair evaluated the documents in Table 4.2 and answered the associated questions, the Finder IFish was again "turned loose" and told to continue gathering information. Armed with new knowledge from its user (the ratings of the top previously-found documents and the new rules

[86] The URL for this document is `http://www.dcs.napier.ac.uk:80/personal.html`.
[87] "Jon's Functional Languages Page" may be found at the URL `http://carol.fwi.uva.nl:80/~jon/func.html`.

those ratings created), the Finder IFISH continued to gather documents. Figure 4-20 shows a snapshot of the Finder's state after running for about another hour. Notice that the ten documents rated as relevant by Blair have "floated" to the top of the list as their interestingness has increased. These documents now generate new sources of information for the Finder IFISH to download and investigate, which in turn generate new user questions and information sources. Although not shown in the figure, documents negatively rated have "sunk" to the bottom of the list as they are now less interesting than virgin, unaccessed information.

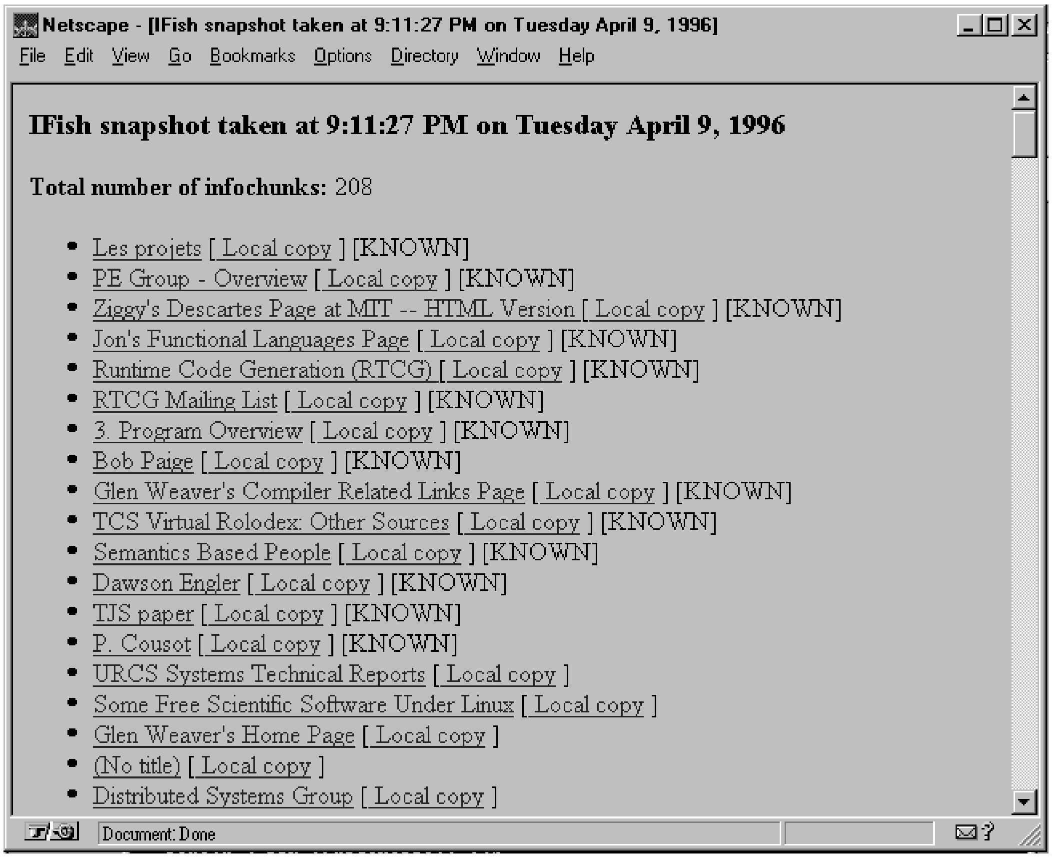

Figure 4-20: Discovered documents ranked by interestingness.

The feedback process the Finder IFISH follows, gathering new information, generating user questions and using user responses to retarget itself, can in theory continue indefinitely. In practice, of course, there may be resource limits that constrain the process. Also, over time user desires may change, thus rendering as uninteresting documents previously viewed as relevant.

Chapter 5

The Future of IFISH

We have described in this thesis an architecture for a new class of resource discovery tool and demonstrated a system for constructing individual members of that class. The Finder IFISH described in Chapter 4 is a prototype, and while its success provides proof-of-concept it is just the first (and perhaps one of the most simple) of an entire family of IFISH. To achieve a better understanding of the full capabilities and limitations of the entire class of IFISH a number of areas need to be investigated. First, we need to develop meaningful measures of IFISH performance. Second, we must continue to "evolve" IFISH; Section 5.2 discusses four possible research directions for IFISH improvement. We conclude this chapter, and the thesis itself, with some remarks and predictions on the role IFISH and IFISH-like services may play in future information markets.

5.1 Evaluating IFISH Performance

How should we attempt to evaluate and characterize the performance of individual IFISH, or even entire "schools" of related IFISH? Are meaningful evaluations even possible? At the end of the previous chapter we saw anecdotal evidence concerning the performance of the "Finder" IFISH, but anecdotes alone are not sufficient to fully evaluate IFISH performance. Ideally, we would like robust qualitative and quantitative measures so that just as IFISH themselves are able to order individual infochunks, so may we order and rank groups of IFISH.

Traditional information retrieval (IR) performance is often characterized by two quantities: precision and recall. Let C be a collection of documents, Q be a particular query to an information retrieval process over C, and $Q(C)$ denote the subset of C returned in response to Q by the IR process. Let $R(X)$ denote the subset of document collection X that contains exactly all "relevant" documents in X (where relevancy is measured by some oracle, such as the user himself). The *precision* p is then defined to be:

$$p = \frac{|R(Q(C))|}{|Q(C)|} \tag{5.1}$$

where $|X|$ denotes the cardinality of set X. The precision p is the fraction of documents returned by query Q found to be relevant by the oracle. The *recall* r is defined as the fraction of all relevant documents in the entire collection (as determined by the oracle) which are returned by the query;

that is,

$$r = \frac{|R(Q(C))|}{|R(C)|} \tag{5.2}$$

Notice that a query $Q'(C)$ that simply returns the entire collection C must have a recall $r' = 1$ and a precision p' that is the percentage of relevant documents in the collection:

$$p' = \frac{|R(Q(C))|}{|Q(C)|} = \frac{|R(C)|}{|C|} \tag{5.3}$$

A query Q'' that always returned the empty set would have a recall $r'' = 0$ and an undefined precision (since $|R(Q(C))| = |Q(C)| = 0$).

How would such measures translate into the IFish environment? To start, notice that there are four underlying assumptions in the definitions of precision and recall. First, it is assumed that the collection C of all documents in the universe under study is fixed; if C is not fixed than the subset of relevant documents in C denoted by $R(C)$ is also not fixed and thus recall cannot be determined precisely. Second, by definition recall measures the percentage of relevant documents actually returned by the query process, thus the query must have the possibility of actually seeing and returning every relevant document in C. Third, the relevancy of a document, as defined by the oracle, is also assumed to be fixed: if the set of all relevant documents changes then both precision and recall may change. For example, if the user's interests change as part of the user-interaction process, then the relevance criteria are not static. Finally, notice that there is an assumption that the query process is independent of the relevancy oracle; if not, then every query engine would be able to score perfect precision by invoking the oracle as part of the query mechanism.

IFish break all of these assumptions. IFish operate in a dynamic environment, where the document collection, as well as the subset of the document collection that is accessible to the IFish, may change over time. Further, although the collection may change over time, it may not be possible for an IFish to detect such changes or discover all the documents in the collection at a given point in time. The relevancy oracle in the IFish system is ultimately the user; as the user's interests may change over time his relevancy responses may change. Also, IFish have access throughout the query process to the user, and thus by definition the relevancy oracle embodied in the user. IFish use such information as part of the querying process and modify themselves in response to statements made by the oracle. Thus, it appears that we cannot categorize IFish using precision and recall.

Another possible method for measuring IFish performance is to compare the amount of time required by the IFish working with the user and by the user working alone to generate a particular result. Obviously such a measurement would be highly dependent on the skill of the particular user as well as the type on information being sought. In addition, it is not at all obvious that the comparison should require the user-assisted IFish to find the exact same sources of information discovered by the user working alone; instead, we may consider whether the results are equivalent in terms of their appeal to the user.

There is a deep question concerning the implementation of the suggested comparison: what exactly is meant by "time required" when we are discussing the capabilities of an IFish or a lone user? There are obviously many different possible "times:" the actual elapsed "real" time that passes during the entire process, the processor time consumed, even the amount of time occupied by active user participation. It is this last possibility that may be the most representative way to gauge IFish performance; if the goal of a particular IFish is to free the user from doing the drudgery and thus making his precious time more productive, then we should at a minimum weigh heavily how much time users must spend answering IFISH questions against the results IFISH obtained with that limited interaction. Taking this concept a step further, we can imagine scoring IFISH performance as the ratio g/t, where g is the number of "golden nuggets of information" discovered by the process and t is the total amount of user time spent in order to generate those golden nuggets of information. Of course, such a measure requires an oracle that decides whether a particular piece of information qualifies as a golden nugget or not.

Finally, we can envision trying to perform direct comparisons between an IFISH search and similar searches using the already-existing monolithic indexing services. For example, does the Finder IFISH find significantly more "high-quality" information sources than a similar search performed using Alta Vista? Of course, we would have to define what a "high-quality" information source is, a determination that can probably only be made by the user himself. Furthermore, the inputs to the Finder and to Alta Vista are not comparable, so we would need to invent some scheme for determining comparable sets of inputs.

5.2 Future Work

In this thesis we have spent much time and effort creating the Internet Fish Construction Kit, a substrate that makes it easy to build various species of IFISH. We have demonstrated that the Construction Kit may be used to create useful IFISH such as the Finder IFISH described in Chapter 4, but the Construction Kit is only the first step in exploring the capabilities and properties of this new class of resource discovery tools. Of the many ways in which IFISH research may proceed from here we detail five:

1. Straight-line improvements to the Construction Kit and the assorted sets of heuristics.
2. Providing self-analysis capabilities for IFISH.
3. Enabling and utilizing inter-IFISH communication.
4. Deploying IFISH in other information oceans.
5. Moving toward IFISH that strive for serendipitous resource discovery.

Each of these possible extensions is orthogonal to the others and may be explored independently.

5.2.1 Straight-line Improvements

The Finder IFISH is a minimalist creation; it incorporates relatively few heuristics for accessing information and performing analysis of retrieved data. One obvious direction for future work, therefore, is to expand the number and quality of the heuristics used in the Finder IFISH. We can spend time adding heuristics to deal with new monolithic indexing services, such as the A2Z [34], Point [35] and Magellan [56] review services. In addition, current IFISH heuristics may be expanded or refined to extract more data from these services. For example, the Yahoo-related heuristics can be improved and expanded to take greater advantage of the hierarchical structure of the Yahoo database. Similarly, it is a relatively straightforward task to extend the Alta Vista heuristics so that IFish are not limited to the first 200 hits detected by an Alta Vista search, although not entirely trivial[88].

The data analysis heuristics in the Finder IFish could also be expanded significantly. For example, currently the Finder IFish performs almost no analysis on the structure of discovered URLs; that is, except for the interest rule that slightly biases URL interestingness in favor of likely home pages, the structure inherent in the path components of a URL are never recognized or used. The path components of a URL typically represent some hierarchical data structure within a Web server; the hierarchy is not necessary for the server to operate properly, but exists mainly for the convenience of the maintainers of the data on the server. The fact that I as a server maintainer chose to place a particular information object at `/foo/bar/baz.html` suggests to the IFish that other documents similar to `baz.html` may be found within `/foo/bar/*`. Similar weight should be given to the strings within an HTML document that are identified as headers, since the author of the document presumably chose those strings because they accurately summarized sections of the text.

5.2.2 Self-analysis

Recall from Section 1.3.4 that a long-term goal for Internet Fish is a hope that IFish may eventually permit new methods for discovering effective procedures for resource discovery. That is, at the same time as IFish are discovering information concerning particular topics, they are also discovering meta-information about the *process* of discovering information. If IFish are to evolve into somewhat-feeble research librarians, then they need not only self-reflection but the tools to analyze their own "thought processes" to recognize when they have found a particularly successful method for finding new, relevant information.

IFish already maintain much of the information that is needed for such self-analysis. Every transitional link between two infochunks is labelled with both the name of the rule that created the link as well as the time at which the link was created. Together with a complete history of rules in the IFish system, this information can be used to reconstruct the derivation chain for any infochunk[89]. The generated derivation history may then be used as input to self-inspection procedures that look for patterns in the derivations. That is, just as IFish may contain analysis rules that look for patterns in the data extracted from the Internet, so can the same IFish look for patterns within its own execution history. The capability to perform self-inspection and to draw inferences concerning the effectiveness of their own information-retrieval procedures is the first step in the process of turning IFish into primitive Internet research librarians.

[88] Although an Alta Vista search may result in thousands of "hits," the service permits browsing of only the top 200 hits as ranked by Alta Vista's scoring mechanism. Thus, query refinement is needed in order to access hits that are ranked below 200.

[89] Depending on the IFish implementation some additional record-keeping may be required in addition to that described in Chapter 2 in order to guarantee that full derivation is possible at an arbitrary point in the future. For example, when a new infochunk is installed into the system with a data slot that matches the data slot of an already-existing infochunk, the new infochunk and the existing infochunk are merged together. This merger process, in the current IFish implementation, also merges the two *typeinfo* declaration lists into one new list. In order to be able to reverse the merging process a record of the pre-merge *typeinfo* lists is necessary.

5.2.3 Inter-IFish Communication

The IFish substrate provides mechanism for both network communication between an IFish and a remote Web server as well as communication between an IFish and the user operating or running it. Both classes of communication utilize the Web as the underlying network protocol; IFish act as Web clients when talking to remote server and act as Web servers when interacting with the user. There is, however, a third form of IFish communication not investigated within this thesis: IFish-IFish (or inter-IFish) communication. That is, we would also like the IFish substrate to facilitate interaction among multiple IFish, as every IFish itself is a potential information source for other IFish.

Although perhaps not the most convenient protocol, the WWW functionality built into the current IFish substrate may be used for inter-IFish communication. Imagine that I_1 and I_2 are two independent IFish existing in the same information space, and assume that I_1 and I_2 are aware of each other. I_1 may then communicate in a limited fashion with I_2 by answering questions posed by I_2 via $I_2's$ WWW server; I_2 may similarly answer questions posed by I_1. Defining a suitable representation language for inter-IFish communications is a separate (and generally orthogonal) task[90], just as HTTP and HTML together provide only a mechanism for information exchange.

It is fairly easy to justify the need for inter-IFish communication. Obviously each running IFish is an information source and thus a potential information resource for another IFish. Similarly, inter-IFish communication is a necessary piece of infrastructure if we wish to explore the behavior of groups of IFish that act in concert with each other. There are other compelling reasons, too. For example, as an individual IFish grows in size it may be necessary or desirable to subdivide it into a "school" of multiple IFish, where each member of the school has a more specific domain of interest than the original IFish. In order for such a school to maintain the same properties and external behavior as the original IFish each school element must have access to part of the internal structure of every member of the school[91]. It may be quite cumbersome to cast these types of communications within the question/answer framework for user interaction of Chapter 3; a more general knowledge-transfer mechanism (such as KQML [21]) is likely required.

5.2.4 IFish in Other Information Oceans

Although we have only considered IFish operating within the environment of the World Wide Web, there is no reason to confine the IFish model to that particular information space. Indeed, as pointed out in Section 1.3.4 IFish could just as easily operate on data contained within a commercial database or local filesystem. Obviously we would need to provide heuristics for acquiring information from these spaces in order to construct such IFish, but the IFish architecture and overall method of operation are still applicable.

As an example alternative information space for IFish, consider the database of transaction

[90] We say "generally orthogonal" because the protocol peculiarities of HTTP may dictate some of the representation language's syntax.

[91] Recall that one of the properties of IFish as described in Chapters 2 and 3 above is that individual IFish never forget the *data* slot contents of any infochunk and thus always merge infochunks that have equivalent *data* slot contents. For a school of IFish to maintain this property, each IFish in the school must either maintain a representation of the aggregated information content of the entire school or otherwise have access to information stored in any member of the school.

information associated with a major credit card issuer[92] or long-distance telephone company[93]. Such databases may themselves be spread over many local hosts in order to satisfy certain performance guarantees, thus creating a locally-networked information space. Drawing inferences from and finding relationships among the contents of these databases, a task often referred to as "data mining," is a goal of every creator of large databases (or at least their marketing departments).

We can see immediately how the IFish framework might be used to create a specialized data mining tool. First, we would need to construct new information-gathering heuristics to replace the Web-specific methods that are part of the IFish Construction Kit. If the data is stored within a structured database then these heuristics may simply take as inputs structured queries, send those queries to the database, and return the received responses. Once new information-gathering heuristics were in place we could then write data mining analysis algorithms as IFish rules and transducers and add them to the IFish.

5.2.5 Toward Serendipitous Resource Discovery

Finally, then, we consider the notion of *serendipitous* resource discovery, introduced in Section 1.3.3 above as the process of discovering interesting information in an unexpected place or manner. We have already seen above in Section 4.5 above how the Finder IFish is capable of uncovering similarly-related documents, but serendipitous resource discovery is more than just finding information that we know (or suspect) exists. A serendipitous Finder IFish, for example, would also discover information spaces that may not be directly related to the set of "known relevant" documents supplied by the user. In addition to finding clearly-related documents, this IFish would suggest areas of interest that the user may not already consider relevant. Thus, the user is not only surprised by the uncovered data but also by its relationship to the known, desired information. This process is similar in many respects to data mining and other attempts to find causal relationships among data.

Extending IFish to enhance their chances for serendipitous resource discovery is closely related to the process of adding self-analysis routines (Section 5.2.2). In both cases we want to extend IFish's data-analysis routines to include new techniques; for serendipitous discovery though that analysis will concentrate on the gathered data itself as opposed to the meta-information concerned with *how* the IFish found that particular data. Serendipitous discovery will also require better communication among related IFish (Section 5.2.3), since the data relationship we hope to discover may likely cross individual IFish boundaries.

5.3 IFish and the Future of Information Markets

As we reach the end of this dissertation it seems both fitting and appropriate to conclude with some speculation concerning the future growth of information markets and the variety of roles that IFish and similar tools are likely to play. Obviously it is difficult, at best, to make predictions

[92] The AT&T Universal Card database creates approximately two million transaction records per day; each transaction records buyer, merchant and purchase information.

[93] The AT&T long-distance telephone network generates approximately 200 million transaction records per day. Each call creates a record containing the calling and called parties and the start and end times of the call; this information is later used off-line for billing and fraud-detection purposes.

concerning a medium that is undergoing rapid, radical change, but I believe that that there are certain unavoidable trends that will shape how we think about information and time as commodities with respect to the Internet. These trends are, namely:

1. The marginal cost of information content is being driven to zero,
2. The marginal price of an individual's time is not necessarily zero, and
3. As the price of content is driven to zero (following marginal cost), value will increasingly be generated by systems that automatically find information or otherwise save time for humans.

I address each of these claims in turn below.

5.3.1 The Marginal Cost of Content

"The marginal cost of information content is being driven to zero." A simple statement, one which at first seems perhaps even a tautology. If we assume information is equivalent to a collection of bits, then the cost of producing copies of that information is almost entirely an up-front expense. Once bits are generated and made available electronically, reproducing those bits can be done essentially for free. Generating a particular, meaningful collection of bits in the first place is not cost-free, as some effort (creative or otherwise) must be expended initially to generate the desired information. However, once the information has been generated in electronic form, further copies require only minimal new media, storage and effort. If the content is available from multiple sources, market forces will drive the marginal price of that content to be equivalent to its marginal cost, and price will continue to approach zero. Even if the content is available from only one source, if equivalent (but not identical) content can be found elsewhere then competition among providers of equivalent content will drive the price toward zero.

If the physical cost of reproducing bits is truly essentially zero, why is it the case that hundreds (if not thousands) of content providers are staking their claims to territory in cyberspace and announcing their intent to make money by charging users for access to content? That is, why are so many content-providing companies willing to make an economic decision which implies that the marginal price of content is *not* being driven to zero? The answer lies with our assumptions: either equivalent content is *not* always available, or the cost of duplicating bits is not approaching zero. Whether "equivalent content" is available (and thus whether there is competition in the market for particular information) is highly dependent on the particular content being offered. Some content, such as current sports scores or mutual fund quotations, may be available from a variety of sources[94]. Other information, like full-text versions of legal opinions, may only be available from a single information provider[95]. Furthermore, what one may consider "equivalent content" may not be so equal to another; information equivalency may also be buyer-dependent.

[94]The price history of twenty-minute-delayed stock quotations on the Internet illustrates content price being driven to zero. It used to be the case that delayed stock quotations could only be accessed over the Internet by purchasing a subscription. Then multiple companies began to offer stock quotations for free in return for certain demographic information; the user data was then used for marketing purposes or for business solicitations. Now it is possible to receive stock quotations absolutely free; the businesses that make quotes available view that service as advertising or an enticement to lure users to their Web site.

[95]If one is the first provider of particular content on-line, then it is possible to charge for access to that content even if that content is eventually available on-line from multiple sources. In this case the first content provider is extracting value from its lead-time over the competition, a practice often referred to as "making money on the bleeding edge."

For equivalent content that is available from multiple sources, if price is not driven to zero then the cost of reproducing bits must be some positive quantity. True, there are some marginal costs associated with the act of duplicating bits over the network (power to drive the network connection, for example), but these costs are so small compared with the up-front hardware requirements that they cannot be responsible for a long-term nonzero content price. The most likely culprit, at least within Berne Union[96] countries, is copyright. Assuming that the information content in question is copyrightable within a particular country[97], then copyright may be used to collect a license fee from every use of the information, thus creating a nonzero cost to duplicating the information. Furthermore, if the owner of the copyright so chooses, certain uses of the information may be prohibited completely. Thus, copyright law provides both a mechanism by which the cost of duplicating bits may have cost as well as a mechanism by which the universe of "equivalent" content may be regulated.

Within a strong copyright regime, it is certainly possible that no equivalent, alternative content exists for a particular piece of information. For example, there may not exist an equivalent for an Ansel Adams photograph, or a Matisse cut-out, or a Washington, DC, subway map. In these cases the copyright holder may be able to impose a nonzero reproduction or distribution cost on the information and thus extract value by simply selling access to their protected content. For content that is not exclusive or otherwise protected, or for which there is a competitive market, we cannot expect to extract value by simply "selling the content." We need to sell something else, something that has nonzero value to customers. Luckily there still exists such an item, even on the Internet: time.

5.3.2 The Marginal Price of Time

If perfect competition in content markets drives the cost of content to zero, we can still extract value (that is, make money) on the Internet from content-related services[98]. The price of content may be zero, but there is still a transaction cost [11] involved when a user searches for particular content: the time required to find the desired content. Assuming that I value my time at some nonzero price, then even if the content itself is free I must expend time finding the content I want. Given this model, the reasoning behind the "golden nuggets per unit time" performance measurement suggested in Section 5.1 is now clear: the less user-interaction time required by an IFISH to perform a particular task, the greater the value of that IFISH to the user. Thus, when we sell autonomous services on the Internet we are in reality selling expertise and time: expertise in using and manipulating the network, and time that the human user need no longer spend to perform some particular task.

[96] The "Berne Union" or "Berne Convention" (defined in 17 USC §101 as the signatories to the "Convention for the Protection of Literary and Artistic Works, signed at Berne, Switzerland, on September 9, 1886, and all acts, protocols, and revisions thereto") is an international agreement among countries concerning the granting and enforcement of copyrights. Most countries are signatories to the Berne Convention; the United States is one of the most recent members of the Berne Union, having ratified the Convention in 1988.

[97] What information is subject to copyright is a property of the Berne Union, the country where the information may be copied, and the copyright laws of that particular country. Within the United States copyright law is fairly broad, requiring only a modicum of originality, but does not extend, for example, to ideas or facts (17 USC §102(b)).

[98] Obviously, those companies in the business of providing Internet *access* still have viable business models. Even without charges for bit carriage, the owners and operators of the physical substrate of the Internet have a marketable product. Whether indirect charges for content, such as content subsidized by advertising, will still be a viable business model is an open question.

5.3.3 Selling Time: the Next Layer of the "Internet Wars"

Finally, then, we come to a startling revelation. For all the hype and all the fanfare surrounding the commercialization of the Internet, we really have not yet even begun to fight the truly interesting battles. This week's trade publication may tout a new round in the "browser wars" between Netscape and Microsoft, or a new offensive in the "server wars," but these battles are just opening skirmishes. The *real*, interesting fights are yet to come; they will involve the next layers of the infrastructure, the layers above the servers and browsers and monolithic search engines that IFISH and IFISH-like tools inhabit.

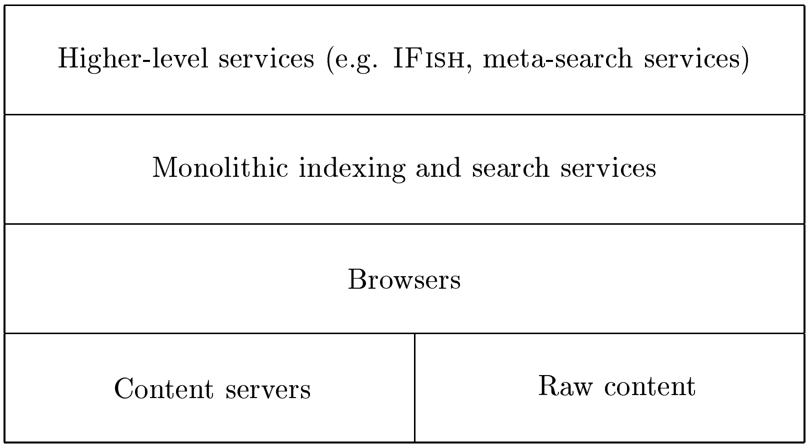

Figure 5-1: Web infrastructure layers

Consider for a moment the current infrastructure of the World Wide Web, pictured in Figure 5-1, and the various "wars" supposedly being waged thereon. At the bottom layer we have the various content servers and the content itself that is available on the Web. The "server wars" are collectively the heated battles among the various companies, individuals and nonprofit groups that maintain and distribute server software to gain market share. Hardly a week goes by without a declaration from one camp that their latest, nifty-keen, server out-performs everyone else's servers on some particular benchmark. Right now the various commercial server companies are so anxious to gain market share that they foist their products for free upon just about any breathing party that should happen to wander past their portion of the Internet[99]. Yet all of this effort to gain market share seems short-sighted, since content servers are part of the initial, sunk cost for making content available on the Web; content servers are part of the equipment required to create a Web server, but once a machine is turned into a Web server neither hardware nor software require further maintenance[100]. What *will* change over time, if anything, is the content that is being stored on the server.

[99] To date, `www-swiss.ai.mit.edu`, a Web server maintained in part by the author, has run at one time or another server software from CERN, NCSA, Netscape, Open Market, Navisoft, and the Apache group. The CERN, NCSA and Apache servers are freely available on the Internet; the other servers were provided for free upon claiming (truthfully) that the Web server hardware belonged to an educational, non-profit organization.

[100] Of course, both the hardware and software may need to be upgraded over time as demand for the server's content changes. However, the point of this discussion is that the hardware and software that together create a Web server can be viewed in most cases as a one-time purchase.

While the major players in the Web server market are fighting for "mindshare" among the population of Web server system administrators, a similarly heated battle is being fought to win the allegiance of individual Web users (i.e. Web "surfers"). These "browser wars" are at least as pitched battles as the "server wars," but they are targeted at the next layer up in the Web infrastructure and a much larger user population. Again we see companies making highly irregular business decisions[101], presumably in the hope of gaining "market share" in a market where the goods being "sold" are more often than not being given away for free. Presumably the browser manufacturers fighting these wars believe that the market for human interfaces to content on the Web (which is all a browser is, after all) is going to be quite lucrative, since the browser is the visible gateway to remote content. But as semi-autonomous information-gathering tools, such as IFish, become more widespread, increasingly it will be the automated tools that interact with the Web, not the humans sitting in front of browsers. *Direct* human browsing time may stagnate, or even decline, as *indirect* human browsing of the Web increases in popularity.

Proceeding to the next level of the Web infrastructure we find even more competition among its occupants. This layer of the infrastructure consists of the monolithic indexing and search services, such as Yahoo, Lycos, WebCrawler and Alta Vista. All of the "lower-level" services that our Finder IFish used in Chapter 4 as primitive methods of discovering new sources of information inhabit this layer. Here the battles are not over which browser is better or which server is faster; instead, since the products being offered are services, the claims and counter-claims concern which service is "better" than the other services. "Better" is obviously a highly-subjective measure for these services, and trying to determine whether there is a quantifiable measure of search services is as difficult a problem here as it is for IFish in general (see Section 5.1 above). The reason that the fights among these services have so devolved is that, with few exceptions[102], they generate revenue solely from advertising. When a user browses one of these services, the results of the user's queries are returned on Web pages including advertisements. Every visit to a services site, every query that is performed, is a revenue-generating event[103]. Just as with all other forms of advertising, revenue is directly correlated with pervasiveness of the advertising media and the size of the population viewing the advertisement. Thus, in order to survive these services need to convince advertisers that advertising on their service is an effective way to reach potential customers and also convince users to use their service instead of their competitors' services.

Whether or not the advertising model is a successful means of generating revenue on the Internet is currently a hotly-contested claim. Supporters of the advertising model argue that the Internet is too much like broadcast media for anything but advertising to work. Detractors argue in response

[101]It is also interesting to note that United States antitrust law appears to work differently in cyberspace than in industries grounded in physical goods. For example, Microsoft is currently giving away, for free, copies of its Internet Explorer 2.0 (IE2.0) Web browser; Microsoft acquired the technology contained inside IE2.0 from Spyglass and pays royalties on every copy of IE2.0 distributed. Since the free distribution of IE2.0 does not appear to be a limited-time offer, it seems unlikely that Microsoft can defend the give-away of IE2.0 as a temporary, promotional "sales" tactic; in a physical-goods market Microsoft's pricing scheme would almost certainly be seen as predatory in nature and thus a violation of §2 of the Sherman Act, which prescribes unilateral acts of monopolization and attempted monopolization.

[102]The notable exception here is Alta Vista, which is not currently an attempt to generate revenue by selling advertising space to others. Digital Equipment Corp., the developers of Alta Vista, currently view the service as a showcase product for Digital's hardware technology. Whether Digital will continue to view Alta Vista itself as purely an advertisement is an open question.

[103]We assume here that the service's advertising rates are calculated based on the number of times the advertisement is shown. If the advertising rates instead are based on the number of people who "click-through" the advertisement to reach the advertiser's own Web site, then each advertisement presentation is only an opportunity for a revenue-generating event [30].

that bit-carriage, pay-per-view or subscription payment models are viable alternatives, once there is infrastructure in place that permits efficient movement of very small amounts of digital currency across the network. Such "micropayment" schemes [25, 51], so named because they aim to allow efficient transfer of amounts on the order of a tenth of a cent, are a current topic of research but as yet none have been moved out of the laboratory and into public trials. However one feels about advertising, there is a fundamental assumption being made by the services selling advertising that is unlikely to hold true, namely that the "users" of their services are direct human browsers. Advertising depends upon the possibility of purchasers being influenced by the ads; take away the connection to possible customers and the model falls apart. If indirect human browsing grows in popularity, as is argued in this thesis, then increasingly the "people" reading the advertisements on Magellan or Yahoo or WebCrawler will be automated processes, such as IFISH.

In fact, if we move up to the top layer of infrastructure depicted in Figure 5-1 and look at the current inhabitants of that space, we will already find automated services "feeding" off the monolithic indexes that ignore inserted advertisements altogether. For example, Selberg and Etzioni's MetaCrawler [54] combines together search results from six other search engines; the SavvySearch service [17] can access and merge results from up to twenty different resources. The Finder IFISH itself basically implements a similar service as a by-product of its ongoing research (see Section 4.2.1 above). Such "meta-search" engines[104] do not pass on the advertisements of the underlying search services that they use as information sources; when displaying results obtained from the Excite service, for example, SavvySearch is kind enough to remove the ads inserted by Excite[105]. The advertising model, therefore, may not hold if the "browsers" of advertising-supported services are themselves other automated services that are not influenced by the ads.

We see now the truth behind the claim made at the beginning of this section, namely that the truly interesting fights on the Internet, as well as the opportunities to extract value, will occur in the layers of infrastructure built on top of today's current collection of content servers, browsers, and monolithic indexing engines. Remember, time is a valuable commodity, and we can "sell time" on the Internet by selling access to services that save humans time and perform certain tasks automatically. I cannot "sell" the contents of a Yahoo-like service via advertising if the users of the service are themselves automated, IFISH-like tools. Furthermore, as Section 5.3.1 pointed out above, I cannot sell content itself if equivalent content is available elsewhere, since competition will drive the marginal price of content to zero. I *can* sell time, though, and (for most people) time has a nonzero marginal price. Thus, we can save people time by constructing layers of autonomous, persistent programs to perform tasks users want accomplished, and the value of these services will be determined by the time they save as well as the quality of the product they generate.

In closing, let us return to the image with which we began this thesis, that of the Internet as a vast "sea of information" in which various information-gathering tools like the IFISH "swim." We can imaging the Internet sea filled with diverse populations of Internet Fish, some big, some small, some flat like flounder, some round like salmon. These IFISH swim in the sea of information in order to perform some task or satisfy some goal, and in the process of pursuing those goals IFISH interact with each other. Bigger IFISH may "consume" or exchange data with smaller IFISH in the process of satisfying their particular goals. In our "layers of infrastructure" model these various types of IFISH inhabit different portions of the upper layer of the infrastructure. Lower-level services,

[104] Those less charitable label such services "para-sites" as they truly do feed off the work of other sites.

[105] It should be noted that such rewriting may be an infringement of Excite's copyrights, depending on whether the URLs returned by Excite are themselves copyrightable.

"smaller" IFISH, are subservices to be utilized by bigger services as needed. An IFISH's "user" may in fact be another IFISH occupying a higher level of infrastructure, swimming at a different depth in the sea. As a true, human user, the Internet services with which I interact may be pieces of the topmost levels of infrastructure that are very good at understanding requests for information phrased in English and recognizing what lower-level services would be particularly well-suited to answering my request. The top-level IFISH with which I interact may know nothing about "lattice basis reduction algorithms" in particular but may be very good at classifying that topic as a math-related inquiry and pass the problem off to a known "math IFISH," perhaps operated by the American Mathematical Society. The AMS IFISH in turn may recognize my query as belonging to a particular field, subfield and branch of some hierarchical organization of mathematical knowledge, and pass on the request (or some other request derived from it) to more-specialized IFISH, etc. Eventually the desired information percolates back to the top and I have my answer.

Finally, notice that every time an infrastructure layer boundary is crossed, every time one IFISH uses another IFISH as a "subfish" (because the subfish is either more efficient or more specialized for the task at hand), we have the potential for an information market transaction. If IFISH A recognizes that subfish B can answer a query for less than A would have to expend to answer the query itself, A can purchase access to B to answer the particular query. Digital currency (as well as other tokens, perhaps) can be freely exchanged among IFISH in order to satisfy my request. It is somewhat ironic that the motivating factor for putting content on the Web was a human population available to browse that information, and that as the amount of information available vastly outstrips human capacity to handle that load, we will draw back from the Web and increasingly leave the information-finding to autonomous, persistent processes.

5.4 Conclusion

In this thesis I have detailed the invention of the "Internet Fish," a new class of resource-discovery tools for finding information on the Internet. Designed to be personal, persistent, and semi-autonomous, Internet Fish are deployed, gather information, and return to the user to present the current results of their search. In Chapters 2 and 3 we have described a new language for constructing Internet Fish, built on top of Scheme, and an Internet Fish Construction Kit that makes its easy to build and deploy particular IFISH. The Construction Kit facilitates both the encapsulation and inclusion of heuristic knowledge (Chapter 2) as well as long-term conversations between users and IFISH (Chapter 3). Combined, these two techniques give IFISH a powerful advantage over static, monolithic search engines and allow IFISH to modify their own behavior over time in response to outside stimula.

Using the Construction Kit I have built a sample IFISH, the Finder IFISH, which demonstrates the usefulness of both heuristic knowledge and user interaction. I named this particular IFISH the "Finder" because it is designed to find Web pages that are "like" a given set of Web pages. "Likeness" for the Finder IFISH is determined by a combination of text-based document relevancy measures as well as preference statements from the user. Chapter 4 presented the Finder IFISH and detailed the heuristics it uses to manipulate the Web, take advantage of various search engines and indexing services, and query the user for confirmation of its own guesses.

In this chapter I have argued that as the universe of content and services available on the Internet grows, services like Internet Fish will not only become prevalent but will be the primary method of extracting revenue. Because bits can be duplicated and distributed for essentially free, the price for accessing non-unique content on the Internet will be driven to zero. Services like Internet Fish, however, that not only aggregate content but in doing so also decrease required user time, will continue to have marketable value, since the marginal price of time is nonzero. We are already seeing the beginning of this trend on the Internet with the increased popularity of meta-search services ("para-sites") that leverage the work of various monolithic indexing services.

Bibliography

[1] M. Andreessen. NCSA Mosaic technical summary. Technical report, NCSA, 1993. Available from URL: ftp://ftp.ncsa.uiuc.edu/Web/Mosaic/Papers/mosaic.ps.Z.

[2] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996. ISBN 0-201-63455-4.

[3] T. Berners-Lee, R. Cailliau, J.-F. Groff, and B. Pollermann. World-Wide Web: The information universe. *Electronic Networking: Research, Applications and Policy*, 1(2), 1992.

[4] T. Berners-Lee and D. Connolly. Hypertext markup language – 2.0. Technical Report RFC 1866, IETF Network Working Group, November 1995. Available from URL: ftp://ds.internic.net/rfc/rfc1866.txt.

[5] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – http/1.0 (draft 05). Technical report, IETF HTTP Working Group, February 1996. Available from URL: http://www.w3.org/pub/WWW/Protocols/HTTP/1.0/spec.html.

[6] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators (URL). Technical Report RFC 1738, IETF Network Working Group, December 1994. Available from URL: http://www.w3.org/pub/WWW/Addressing/rfc1738.txt.

[7] M. Blair. *Descartes: A Dynamically Adaptive Scheme Compiler And Run-Time Execution Specializer*. PhD thesis, Massachusetts Institute of Technology, forthcoming. Available from URL: http://www-swiss.ai.mit.edu/~ziggy/descartes.html.

[8] N. Borenstein and N. Freed. Mime (multipurpose internet mail extensions) part one: Mechanisms for specifying and describing the format of internet message bodies. Technical Report RFC 1521, Internet Network Working Group, 1993. Available from URL: ftp://ds.internic.net/rfc/rfc1521.txt.

[9] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. Harvest: A scalable, customizable discovery and access system. Technical report, University of Colorado at Boulder, 1994.

[10] C. Brooks, M. S. Mazer, S. Meeks, and J. Miller. Application-specific proxy servers as http stream transducers. In *Proceedings of the Fourth International World Wide Web Conference*, December 1995. Available from URL: http://www.w3.org/pub/Conferences/WWW4/Papers/56/.

[11] R. H. Coase. *The Firm, the Market, and the Law.* The University of Chicago Press, 1988.

[12] D. Crocker. Standard for the format of arpa internet text messages. Technical Report RFC 822, IETF, August 1982. Available from URL: ftp://ds.internic.net/rfc/rfc822.txt.

[13] P. M. E. de Bra and R. D. J. Post. Information retrieval in the World-Wide Web: Making client-based searching feasible. In O. Nierstrasz, editor, *Proceedings of the First Annual World Wide Web Conference*, Geneva, May 1994. CERN. Available from URL: http://www.win.tue.nl/win/cs/is/reinpost/www94/www94.html.

[14] Deja News Research Service, Inc. Deja news research service™. Available from URL: http://www.dejanews.com/.

[15] Digital Equipment Corporation. Alta Vista: Main Page. Available from URL: http://www.altavista.digital.com/.

[16] F. Douglis and T. Ball. Tracking and viewing changes on the web. In *Proc. of the 1996 USENIX Technical Conference*, 1996. Available from URL: http://www.research.att.com/orgs/ssr/people/douglis/papers/aide.ps.gz.

[17] D. Dreilinger. SavvySearch home page. Available from URL: http://www.cs.colostate.edu/ dreiling/smartform.html.

[18] A. Emtage and P. Deutsch. Archie–an electronic directory service for the Internet. In *Proc. of the USENIX Winter Conference*, pages 93–110, Jan 1992.

[19] Excite, Inc. Excite for web servers (EWS). Available from URL: http://www.excite.com/navigate/home.html. Previously distributed as "Architext Excite for Web Servers" by Architext Software. "Architext Software" is now Excite, Inc.

[20] Excite, Inc. excite Netsearch. Available from URL: http://www.excite.com/.

[21] T. Finin et al. Draft specification of the kqml agent-communication language. Technical report, The DARPA Knowledge Sharing Initiative External Interfaces Working Group, June 1993. Available from URL: http://retriever.cs.umbc.edu/kqml/.

[22] J. Fletcher. JumpStation Front Page. Available from URL: http://www.stir.ac.uk/jsbin/js.

[23] S. Foster and F. Barrie. Common questions and answers about Veronica, a title search and retrieval system for use with the Internet Gopher. Available from URL: gopher://pogonip.scs.unr.edu/00/veronica/veronica-faq.

[24] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole. Semantic file systems. In *Proc. 13th ACM Symposium on Operating Systems Principles*, pages 16–25. Assoc. Comp. Mach., Oct 1991.

[25] S. Glassman, M. Manasse, M. Abadi, P. Gauthier, and P. Sobalvarro. The millicent protocol for inexpensive electronic commerce. In *Proceedings of the Fourth International World Wide Web Conference*. World Wide Web Consortium, 1995. Available from URL: http://www.research.digital.com/SRC/millicent/papers/millicent-w3c4/millicent.html.

[26] M. Gray. The World-Wide Web Wanderer. Available from URL: http://www.mit.edu:8001/people/mkgray/web-growth.html.

[27] D. R. Hardy and M. F. Schwartz. Essence: A resource discovery system based on semantic file indexing. In *Proceedings of the 1993 Winter USENIX*, pages 361–374, Jan 1993.

[28] Infoseek Corportation. Infoseek guide. Available from URL: http://www.infoseek.com/.

[29] International Business Machines. aquí. Available from URL: http://www.aqui.ibm.com/.

[30] Isabel Maxwell, Executive Vice President, The McKinley Group. Personal communication. (The McKinley Group operates the Magellan search, index, review and rating service.).

[31] B. Kahle and A. Medlar. An information system for corporate users: Wide area information servers. *ConneXions — The Interoperability Report*, 5(11):2–9, Nov 1991. Available from URL: ftp://think.com/wais/wais-corporate-paper.text.

[32] H. Lieberman. An automated channel-surfing interface agent for the web. Available from URL: http://lieber.www.media.mit.edu/people/lieber/Lieberary/Letizia/WebFive/Overview.html.

[33] H. Lieberman. Letizia: An agent that assists web browsing. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Montreal, 1995. Available from URL: http://lieber.www.media.mit.edu/people/lieber/Lieberary/Letizia/Letizia-Intro.html.

[34] Lycos, Inc. A2Z. Available from URL: http://a2z.lycos.com/, 1996.

[35] Lycos, Inc. Point. Available from URL: http://www.pointcom.com/, 1996.

[36] Lycos, Inc. The Lycos^{TM} Catalog of the Internet. Available from URL: http://www.lycos.com/, 1996.

[37] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. Technical Report TR 93-34, University of Arizona, 1993. Available from URL: ftp://ftp.cs.arizona.edu/glimpse/glimpse.ps.Z.

[38] Massachusetts Institute of Technology. TechInfo overview. Available from URL: http://web.mit.edu:1962/tiserve.mit.edu/9000/26323.html, July 1992.

[39] O. A. McBryan. GENVL and WWWW: Tools for taming the web. In O. Nierstrasz, editor, *Proceedings of the First Annual World Wide Web Conference*, Geneva, May 1994. CERN. Available from URL: http://www1.cern.ch/PapersWWW94/mcbryan.ps.

[40] M. McCahill. The internet gopher: A distributed server information system. *ConneXions–The Interoperability Report*, 6(7):10–14, Jul 1992.

[41] P. Mockapetris. Domain Names - Concepts and Facilities. Technical Report RFC 1034, IETF Network Working Group, November 1987. Available from URL: ftp://ds.internic.net/rfc/rfc1034.txt; see also RFC 1035.

[42] J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal, and F. Smith. Andrew: A distributed personal computing environmen. *Communications of the ACM*, 29:184–201, 1986.

[43] J. Myers and M. Rose. Post Office Protocol – Version 3. Technical Report RFC 1725, IETF Network Working Group, November 1994. Available from URL: ftp://ds.internic.net/rfc/rfc1725.txt.

[44] NCSA Mosaic Project. What's new with ncsa mosaic and the www. Available from URL: http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/whats-new.html.

[45] H. F. Nielsen, T. Berners-Lee, H. W. Lie, and J.-F. Groff. The w3c reference library. Available from URL: http://www.w3.org/pub/WWW/Library/.

[46] owner-mail server@rtfm.mit.edu. Help file for the usenet-addresses database. Send e-mail to mail-server@rtfm.mit.edu with 'send usenet-addresses/help' in the body of the message.

[47] B. Pinkerton. Finding what people want: Experiences with the WebCrawler. In *Proceedings of the Second Annual WWW/Mosaic Conference*, to appear. Available from URL: ftp://www.biotech.washington.edu/pub/WebCrawler.ps.gz.

[48] J. Postel. Simple Mail Transfer Protocol. Technical Report RFC 821, IETF Network Working Group, August 1982. Available from URL: ftp://ds.internic.net/rfc/rfc821.txt.

[49] J. Postel and J. Reynolds. File Transfer Protocol (FTP). Technical Report RFC 959, Internet Network Working Group, Oct 1985. Available from URL: http://www.w3.org/pub/WWW/Addressing/rfc1738.txt.

[50] Quarterdeck Corp., Inc. Webcompass fact sheet. Available from URL: http://arachnid.qdeck.com/qdeck/products/webcompass/. Demo version (WebCompass Personal Edition) available from URL: http://arachnid.qdeck.com/qdeck/demosoft/webcompass_lite/.

[51] R. L. Rivest and A. Shamir. PayWord and MicroMint: Two simple micropayment schemes. 1996. Available from URL: http://theory.lcs.mit.edu/˜rivest/RivestShamir-mpay.ps.

[52] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

[53] M. F. Schwartz and P. G. Tsirigotis. Experience with a semantically cognizant internet white pages directory tool. *Journal of Internetworking–Research and Experience*, to appear.

[54] E. Selberg and O. Etzioni. Multi-service search and comparison using the metacrawler. In *Proceedings of the Fourth International World Wide Web Conference*. World Wide Web Consortium, 1995. Available from URL: http://www.cs.washington.edu/homes/etzioni/.

[55] M. A. Sheldon, A. Duda, R. Weiss, J. W. O'Toole, and D. K. Gifford. A content routing system for distributed information servers. In *Proc. Fourth International Conference on Extending Database Technology*, number 779 in Lecture Notes in Computer Science, Cambridge, England, Mar 1994. Springer-Verlag.

[56] The McKinley Group, Inc. Magellan internet guide. Available from URL: http://www.mckinley.com/, 1996.

[57] Transarc Corp. Transarc product information: Afs. Available from URL: http://www.transarc.com:80/afs/transarc.com/public/www/Public/ProdServ/Product/AFS/index.html, 1996.

[58] S. Wu and U. Manber. Fast text searching with errors. Technical Report TR 91-11, University of Arizona, 1991. Available from URL: ftp://ftp.cs.arizona.edu/agrep/agrep.ps.1.

[59] Yahoo, Inc. Yahoo! Available from URL: http://www.yahoo.com/.

[60] D. Zimmerman. The finger user information protocol. Technical Report RFC 1196, IETF Network Working Group, December 1990. Available from URL: ftp://ds.internic.net/rfc/rfc822.txt.