

Abstraction of Pathnames and Streams in CLforJava

Robert K. Shields

(College of Charleston, USA
robert.k.shields@gmail.com)

Hector Raphael Mojica

(College of Charleston, USA
hectorraphaelmojica@gmail.com)

Seth Rylan Gainey

(College of Charleston, USA
seth.rylan@gmail.com)

Jerry Boetje

(College of Charleston, USA
boetjeg@cofc.edu)

Abstract: The ubiquity of networked resources necessitates that programming environments evolve to facilitate access to these resources. Common Lisp, based on 1989 specifications, was not designed to access Internet resources. In recent years, some Common Lisp implementations have attempted to fill this niche in various ways. Our work on an ANSI Common Lisp implementation for the Java Virtual Machine has led us to extend the representation of pathnames by mapping pathname components to those of Uniform Resource Identifiers. Additionally, the stream facility was similarly updated to allow streams to include support for accessing modern remote resources.

Keywords: Common Lisp, Java, HTTP, URI, File Systems, Pathnames

Categories: D.2.1, E.5, D.3.3, F.3.3, K.3.2

1 Introduction

“Common Lisp is not so much a language, as a constraint on a language [Pittman, 88].” Many implementation details are left unspecified leading to various unique interpretations and implementations. Due to this nature, Common Lisp (CL) possesses the characteristic of core adaptability. With that adaptability, implementations are free to reinterpret certain types to match the ever-changing field of Computer Science. We show that it is possible to adapt a type (PATHNAME) mainly intended for local file system representations to incorporate modern Internet resources.

1.1 CLforJava

CLforJava is an ongoing research project at the College of Charleston. As the core of the capstone software engineering course, undergraduate students work on an original, open-source implementation of Common Lisp designed to execute on the Java Virtual Machine. CLforJava is unique in that it allows for intertwining of the two languages

without reliance on a Foreign Function Interface or syntactic tricks. A set of Java API and libraries provide access to Lisp components while Java methods are available through CLOS generic functions [Boetje, 05].

1.2 History of Pathnames

Common Lisp provides two methods of representing files: *namestrings* and *pathnames*. The former are simple strings representing the location of a file in a specific file system and are, by definition, implementation-dependent. Pathnames, on the other hand, are intended to abstract filenames in an implementation-independent way [Steele, 90]. Pathnames are defined in the 1989 Common Lisp specification as “structured object[s] which represent a filename.” This abstract data object represents filenames using six components, each corresponding to a generalized attribute of a given filename. The specification of pathnames is intentionally vague. Many of the particulars relating to pathnames and their associated functions are left up to the discretion of the implementer, leading to many differences among pathnames in current Common Lisp implementations [Pitman, 88].

Pathnames were loosely specified to allow for successful abstraction of filenames across many existing and even future file systems [Steele, 90]. At a time when the nascent Internet was still mainly an academic tool, pathnames were primarily used to designate local resources. Certainly by today’s standards, support for networked resources in CL is limited. The host component of pathnames allows a specific file system to be associated with a filename [Steele, 90]. Such support allows for multiple file systems on a network but is inadequate for accessing modern Internet resources. The flexibility of the pathname specification allows the pathname implementation to also support modern networked computing.

1.3 Rationale for the Extension of Pathnames in CLforJava

Transparent resource access, regardless of physical location, is something programmers have come to take for granted in the Internet Age. An enabling Internet technology, the Uniform Resource Identifier (URI) [Berners-Lee, 05], provides an abstraction of resource location and access methods. By further abstracting the pathname construct, we have found it is possible to map its specifications onto the remote resource descriptions provided by URIs. This interpretation maintains compatibility with ANSI Common Lisp while adding support for URI specifications.

Other Common Lisp implementations have provided similar support by implementing URI functionality as a facility separate from pathnames [Allegro, 07]. This is a reasonable design alternative, but one that relegates the now-ubiquitous URI to the position of a language add-on. Incorporating URI support directly into the pathname abstraction allows both local and networked resources to be transparently represented by the same facility. By doing so, CLforJava endows pathnames with more power, flexibility, and portability. Additionally, URIs can now be handled within an already defined specification that is familiar to all Common Lisp programmers.

2 Implementation of Pathnames in CLforJava

Pathnames in CLforJava have been designed to handle not only traditional local filenames but also any resource described by a URI. Core support for URIs was added without extensions to the language or exceeding the bounds of the CL specification.

2.1 Pathname Components

Existing implementations of Common Lisp provide sufficient, albeit differing, interpretations of mapping local filenames onto the six components of a pathname. For example, Table 1 illustrates the different parsing strategies employed by Allegro CL and LispWorks when a pathname is created from “C:\foo\bar\file.txt.” The directory, name, and type components are parsed the same in both implementations. However, there are different interpretations on how the device, host, and version components should be filled.

Components	Allegro CL 8.1 (FEE)	LispWorks PE 5.0.1
Device	"C"	NIL
Host	NIL	"C"
Directory	(:absolute "foo" "bar")	(:absolute "foo" "bar")
Name	"file"	"file"
Type	"txt"	"txt"
Version	:UNSPECIFIC	NIL

Table 1: (pathname "C:\foo\bar\file.txt")

URIs possess a standard syntax with their own set of components [Berners-Lee, 05] that are somewhat analogous to pathname components. There are, however, nine possible URI components [Figure 1] versus the six of pathnames. Being constrained to the syntax and semantics of CL pathnames, our implementation was prohibited from adding components to the pathname type. The version concept of pathnames is not particularly meaningful for URIs or most popular modern file systems. As such, this component is essentially unused. However, it was decided to leave the version component alone to allow for future file systems which might once again utilize this concept. As a result, the design challenge was actually to map nine URI components to the remaining *five* of pathnames.

During design, the key insight lay in recognizing the concept of a “meta-device” as defined by URI schemes. URI schemes such as “file”, “http” or “mailto” specify how a given resource is structured and accessed. More importantly, they provide a namespace, which enforces the uniqueness of a given identifier. Similarly, the device component of pathnames specifies a physical or logical storage area in which each resource has a unique path. Therefore, our pathname implementation abstracts the

device concept to include all possible URI schemes, represented as a keyword in the device component.

For the base case of local filenames, there is the URI scheme “file.” As shown in Table 2, any pathname representing a local file will have the keyword `:file` as the value of its device component. The name and type is parsed in the obvious manner and the version component is always set to `:UNSPECIFIC`, following the example of Allegro CL. The directory component is quite similar to other implementations. However, some file systems have chosen to separately name a physical device or volume (e.g. drive letters in Microsoft file systems). It was decided that such information, if present, would be included as part of the directory list. This seems reasonable since such a device is certainly part of the hierarchical path uniquely identifying a file. Also, this strategy offers more consistency between filenames on Windows systems and those on UNIX-like systems. This is convenient for a multi-platform implementation of Common Lisp such as CLforJava.

Namestring	"C:\foo\file.txt"	"foo/bar/"	"/foo/.hidden"
Components			
Device	:file	:file	:file
Host	NIL	NIL	NIL
Directory	(:ABSOLUTE "C" "foo")	(:RELATIVE "foo" "bar")	(:ABSOLUTE "foo")
Name	"file"	NIL	".hidden"
Type	"txt"	NIL	NIL
Version	:UNSPECIFIC	:UNSPECIFIC	:UNSPECIFIC

Table 2: File based pathnames in CLforJava

URIs with other schemes are also easily handled. Example 1 shows the mapping of an http-based URI to pathname components. The *authority* [Section 2.2] part maps to the host component. Since this URI’s *path* includes a filename (which is not always the case), the name and type information was retrieved and stored accordingly. Also, the *query* part was stored in the directory list. If there had been a *fragment* part, it would have been similarly stored.

Example 1 – (pathname “http://www.cofc.edu:8080/foo/index.htm?query=x”)

```

Device      :http
Host        "www.cofc.edu:8080"
Directory   (:ABSOLUTE "foo" "?query=x")
Name        "index"
Type        "htm"
Version     :UNSPECIFIC

```

URIs that specify a scheme are said to be *absolute*. There are also valid URIs that do not specify a scheme. Such URIs are termed *relative*, and are analogous to relative filenames. While a filename may be relative within a file system, URIs can be relative within a set of supported schemes. An ambiguity arises with such relative

URIs: there is no syntactical way to distinguish a relative URI from a local filename. In practice this presents little difficulty since a relative URI is only useful when it is later merged with an absolute URI. CLforJava handles this ambiguity by returning a pathname with device `:file` (which may, as a filename, have an absolute path), as shown in Example 2.

Example 2 – (pathname “/foo/bar/index.htm”)

Device	:file
Host	NIL
Directory	(:ABSOLUTE “foo” “bar”)
Name	“index”
Type	“htm”
Version	:UNSPECIFIC

If desired, MERGE-PATHNAMES can later merge this with another pathname representing an absolute URI (e.g. “http://www.cofc.edu/”). The resulting pathname will represent a usable URI (“http://www.cofc.edu/foo/bar/index.htm”).

The file- and http-based URIs shown so far are examples of hierarchical URIs. There are also non-hierarchical URIs that are said to be opaque. Two common examples of opaque URIs are

```
mailto:username@cofc.edu
news:comp.lang.lisp.
```

Example 3 shows how CLforJava handles these simple URIs.

Example 3 – (pathname “mailto:username@cofc.edu”)

Device	:mailto
Host	username@cofc.edu
Directory	(:ABSOLUTE)
Name	NIL
Type	NIL
Version	:UNSPECIFIC

As an opaque URI, there is no further parsing that can be done per URI syntax. Additional decomposition, if any, would be scheme-dependent and is therefore not a part of the URI specification and also not an appropriate responsibility for pathnames.

It is important to note that in all the above examples, the interface for pathname creation has remained unchanged. Under the hood, an abstract factory is responsible for ensuring that a pathname is properly created according to the scheme. This design feature allows the pathname facility in CLforJava to transparently support URIs and traditional local filenames.

2.2 Mapping Strategy Explained

As alluded to in Section 2.1, URI syntax has a hierarchical nature. At the highest level, a URI is opaque and has the form

$$[scheme :]scheme\text{-specific-part}[\#fragment].$$

Syntactically a URI is identified as opaque if it is absolute (specifies a scheme) and its *scheme-specific-part* does not begin with a forward slash ('/'). Opaque URIs are not subject to parsing beyond what is stated above.

Non-opaque URIs are called hierarchical. With these URIs the *scheme-specific-part* may be further parsed according to the syntax

$$[scheme :][//authority][path][?query][\#fragment]$$

and the *authority* component may be further parsed as

$$[user\text{-info}@]host[:port].$$

CLforJava takes advantage of this hierarchy by collapsing a URI's nine possible components into only three: scheme, scheme-specific-part or authority, and path. The resulting parsing strategy implemented by CLforJava is summarized in Figure 1 and discussed in detail below.

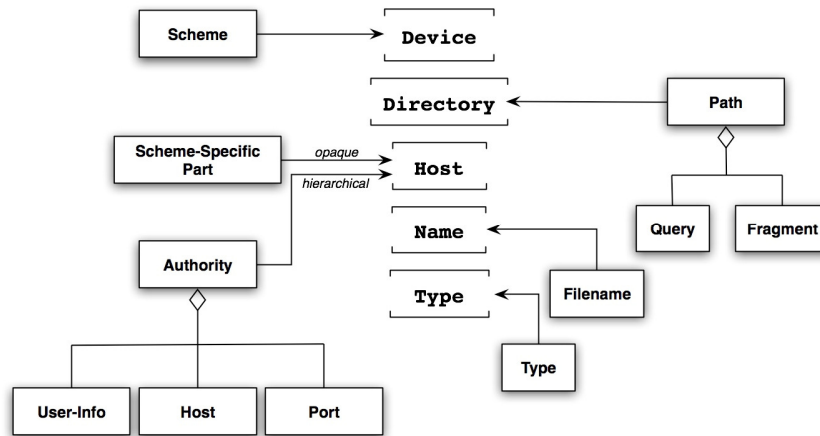


Figure 1 – URI component mapping

Either the scheme-specific part of an opaque URI or the authority part of a hierarchical URI is stored in the pathname's host component. Doing so preserves and encapsulates all the access information in an intuitive location. It also seemed logical

to store the path information in the directory component of the pathname. Each element of the path therefore exists as an element of the directory list. Strictly speaking, the fragment and query parts of a URI are not subparts of the path. Since one of our derived requirements for pathnames was that we preserve syntax, it was unacceptable to add any new components to the pathname type. Therefore, we chose to treat the fragment and query as part of the path so that they can be easily stored in the directory list. Due to the syntax of URIs, these items are easily identified within the list (queries begin with ‘?’ while fragments begin with ‘#’). Some URIs may specify a specific file such as the common “index.htm” in an http-based URI. URI syntax does not explicitly provide separate slots for the name and type of such a file; it is simply included as part of the path. However, since pathnames have components for identifying the name and type of a file, it seemed reasonable to utilize them. CLforJava will parse the path of a hierarchical URI, retrieve any available name and type information, and store it appropriately.

2.3 Scheme-specific Convenience Functions

Pathnames in CLforJava, as described up to this point, are fully capable of handling URIs in addition to traditional local filenames. All information present in a URI is preserved and mapped to pathname components. However, there may be a scheme-dependent need for additional decomposition (e.g. a web application needs to extract the port number from the authority). This task is typically a simple matter of string parsing, facilitated by the well-defined syntax of URIs. It was deemed desirable, however, to add a few convenience functions to allow for more agile handling of URIs. For example, `PATHNAME-AUTHORITY` decomposes the authority part of the URI (stored in the pathname’s host component as shown in Figure 1). It returns a property list indicating the values of each subpart of the authority (user-info, host, and port). Such additions are not critical to URI support in CLforJava but will likely be appreciated by programmers manipulating URIs. Table 3 summarizes these additions.

Function	
<code>pathname-authority</code>	Decompose the URI authority; returns a property list of values for <code>:USER-INFO</code> , <code>:HOST</code> , and <code>:PORT</code> ; returns NIL in the case of an opaque URI
<code>pathname-query</code>	Returns the URI query, or NIL
<code>pathname-fragment</code>	Returns the URI fragment, or NIL
<code>pathname-opaque-p</code>	Returns T if URI is opaque, else NIL

Table 3 – URI Language Extensions

3 Abstracting URI Streams

Abstracting pathnames is merely the first step in harnessing the sophistication of modern, networked data storage. After abstracting pathnames to support URIs, the next logical step was to similarly abstract the creation of streams. Common Lisp

defines character or binary transfer with the stream being limited to simple, unstructured sources and sinks of data. But with the advent of the Internet and URIs, the stream must become a more adaptable agent in the transaction. As with the pathname, the stream must transform into an abstract entity that can morph into a scheme-specific stream as required by the pathname.

3.1 OPEN as Mediator

When the simple pathname was abstracted, the basic stream had grown into a matching abstraction layer. Implementations of URI-based streams must provide stream factories that map to the appropriate type of pathname as determined by the URI scheme. The OPEN function now delegates common abstractions between the pathname/stream pair. It is the responsibility of OPEN to locate the concrete stream based on the type of pathname and return an appropriate active stream, or to signal an error.

Each URI-based stream must encapsulate behavior that is appropriate for the given URI scheme. We have created an *http-stream* type in CLforJava which illustrates this principle. The *http-stream* object defines behavior that is appropriate for the HTTP protocol. As with the *file-stream* object, certain options or combinations of options to the OPEN function are invalid. For example, an *http-stream* supports directions of :INPUT or :IO but not :OUTPUT. The behavior of :INPUT and :IO *http-streams* recreates the nature of the HTTP request. The :INPUT stream uses only the given “query,” whereas the :IO stream must have POST-DATA written to the query before being read.

3.2 Stream Functions

Any added stream types should preserve the semantics common to *file-stream* types. To use the character-based *http-stream* type as an example, the read functions return characters or strings (acting as the request GET method), and FILE-WRITE-DATE and FILE-LENGTH return the same metrics as they would for a *file-stream* type by gathering this information from HTTP headers. Write functions act as a request sender, writing POST-DATA to the given request. All the functions that currently interact with *file-streams* may not be meaningful for all types of URI streams. This is part of the behaviour which must be defined for any added URI-stream types.

3.3 Current Status

Currently, while only supporting *http* and *file* types, CLforJava has laid the groundwork for adding other stream types. The abstraction of the stream system makes building such additions practical. All future additions must merely include the proposed stream implementation, the URI scheme it is related to, and the basic functions of the stream such as reading and writing. Like any responsible architecture, CLforJava requires no retroactive changes to other stream functions such as READ-LINE.

4 Future Work

There are two near-term areas of research into URIs, pathnames, and streams: specifically opaque URIs and programming techniques to extend URI-based pathnames. Although we have demonstrated mappings of hierarchical URIs to pathnames and streams, opaque URIs remain less clear. More development is needed to determine which opaque schemes will be compatible with abstract pathnames and streams.

Presently new URI pathnames and streams can only be implemented by the Java programmer. Development of standard patterns for Java code, properties, interfaces, and jar structure will provide a roadmap for others to extend pathnames and streams. When CLforJava implements the full CLOS system, providing a connection to the Java constructs, Lisp programmers will be able to fully create their own URI support.

5 Conclusion

The CLforJava project attempts, among other things, to take advantage of developments in technology that have arisen since the Common Lisp specification was finalized in 1989. The result has been a Common Lisp implementation that runs on the multi-platform Java Virtual Machine and intertwines Lisp and Java such that strengths of both languages can be leveraged [Boetje, 05]. Our work in updating Common Lisp pathnames and streams for use in an Internet-dominated world is yet another example.

URIs are most commonly used to represent resources on the Internet. Accessing these resources was the primary motivation behind extending the pathname and stream facilities in CLforJava. But URIs were designed with much more than the Internet in mind. They provide a hierarchical namespace capable of representing nearly any imaginable resource. As a result, there exists the potential for using pathnames in CLforJava for far more than accessing Internet resources. Abstract pathnames and streams are potentially more powerful, flexible, and portable than ever before.

CLforJava provides these updated facilities while maintaining adherence to the Common Lisp specifications and without resorting to language extensions to provide core support. Moreover, syntax and semantics has been preserved as much as possible. Well-established interfaces to file systems and streams remain untouched ensuring that CLforJava adheres to the “principle of least astonishment.” Above all, the groundwork has been laid for a more sophisticated and thorough treatment of Internet resources, or perhaps any resource imaginable.

References

[Allegro, 07] URI Support in Allegro CL, 2007,
<http://www.franz.com/support/documentation/8.0/doc/uri.htm>

[Berners-Lee, 05] Burners-Lee, T. RFC 3986, Uniform Resource Identifier (URI): Generic Syntax. Network Working Group. <http://gbiv.com/protocols/uri/rfc/rfc3986.html>

[Boetje, 05] Boetje, Jerry, "Common Lisp for Java"; ILC 2005.

[Pitman, 88] Pitman, Kent M.: "Interactions in Lisp, In Lisp Evolution and Standardization"; Proc. of the First International Workshop, February 1988, <http://www.nhplace.com/kent/Papers/Interactions.html>

[Steele, 90] G.L. Steele, Jr., Common LISP: The Language, 2ed, 1990.