Descot: Distributed Code Repository Framework

Aaron W. Hsu
Indiana University
awhsu@indiana.edu

Abstract

Programming language communities often have repositories of code to which the community submits libraries and from which libraries are downloaded and installed. In communities where many implementations of the language exist, or where the community uses a number of language varieties, many such repositories can exist, each with their own toolset to access them. These diverse communities often have trouble collaborating accross implementation boundaries, because existing tools have not addressed inter-repository communication. Descot enables this collaboration, making it possible to collaborate without forcing large social change within the community. Descot is a metalanguage for describing libraries and a set of protocols for repositories to communicate and share information. This paper discusses the benefits of a public interface for library repositories and details the library metalanguage, the server protocol, and a server API for convenient implementation of Descot-compatible servers.

1. Introduction

All programming language communities must share code to be effective. Issues of portability and ease of distribution arise often within most language communities. In order to share code effectively, programmers must be able to run code portably over different implementations of a language, and they must have some means of distributing their code to users who need an easy way to install and manage their collection of libraries. In communities with a single dominant implementation, the first requirement is usually moot, and a central repository of portable libraries usually satisfies the second requirement (e.g. — CPAN [15]). However, in diverse communities, where many language standards and implementations may actively coexist in close proximity to one another, portability and easy distribution and installation can elude the community as a whole.

For example, the Scheme community actively uses at least four standards [19, 8, 21, 29] and even more actively developed and maintained Scheme implementations. Other communities share these same features. The Scheme community has made progress through efforts like the R6RS [29] `library` form at improving the overall portability of Scheme code. The Scheme community also has a number of repositories and tools for managing libraries. Many of these are implementation specific, but some, like Snow [12] are portable across implementations and try to store portable code packages.

In communities like Scheme, standardizing a single set of tools and a repository for managing, locating, and installing libraries of code is a difficult proposition at best. Rather than trying to create a standard toolchain and a central repository within a community that promotes diverse solutions and approaches, it would be better if many tools could be developed and many repositories created in such a way that they could all interoperate and communicate with one another. This would allow the tools to grow as needed around the segments of the community to which they were best suited, and would prevent any other segment of the community from losing out on advances made by the rest of the community. If a public interface existed for library repositories and tools to communicate among themselves in an effective, extensible manner, the benefits of a central repository could be retained, as well as the ability to develop different approaches to the issue.

Descot [18] realizes just such an interface by utilizing an RDF-based Schema [5, 24] to define a language for expressing library metadata and defining a set of protocols for interfacing with servers. It defines protocols for querying the server about code, retrieving specific library metadata, submitting code to the repository, and for mirroring one repository from another. Note that Descot does not try to replace existing management tools, which already take care of installing code, and it does not attempt to establish any specific repository. Instead, Descot enables the communication between repositories and among repositories and tools. Descot does not attempt to deal with the portability of the code itself, and leaves such efforts to other standards such as R6RS [29]; it deals only with the metadata of a library and the means to access this metadata.

With diverse communities like Scheme, collaboration between libraries will often break down into as much of a social as a technical problem. Descot cannot hope to solve social opposition, but it does enable the community to collaborate while maintaining the normal benefits of decentralized, separate development. Descot was specifically designed to minimize the impact on the social structure of the community that adopts it. This paper details the first steps towards enabling collaboration, by providing the techincal foundation. The author intends to undertake further efforts, such as the development of easy tools and libraries for deploying and integrating Descot, which will further reduce the barriers that usually exist when trying to improve the collaboration of largely separate efforts.

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

Among the largest of obstacles, implementors and designers of repositories often have their own ideas about the design and use of the repository. Descot enables an unbounded number of clients, each with their own unique features, to operate on a wide variety of repositories. The basic design of Descot also enables easy extension to the metalanguage, which means that additional features for specific repositories may be added easily, without making it impossible for existing tools to work with the basic metadata. With Descot almost every feature and detail is left to the designer of the system, except for the parts necessary for useful communication, and even these are made flexible enough to allow a tremendous range of freedom.

While Descot is not tied to one specific language community, the remainder of this paper discusses Descot within the context of the Scheme community. Section 2 discusses the existing tools surrounding library distribution that are necessary for Descot to be useful. Section 3 details the Descot system itself. Section 4 lists some of the work others have done which relates to Descot. Section 5 contains concluding remarks.

2. Background

In order to effectively share code in a community such as Scheme, there must be a way of running another author's code, and there must be a way of searching, installing, and submitting code to the public. Scheme implementations often have a central repository for that implementation to which authors usually submit their code [31, 26, 30]. In order to manage large programs effectively, most Scheme implementations provide a module system that helps to control the visibility of procedures and macros defined in a block of code. Additionally, these repositories have convenient tools that allow libraries to be automatically downloaded and installed if desired. Often, merely specifying a requirement for one or more libraries is enough to guarantee that an user of a program can automatically install the libraries, assuming that they are visible in the central repository.

Other repositories attempt to host portable libraries that work across implementations. Snow [12] is a good example of this family of repositories (see Section 4 for more examples of these systems), and has a number of useful tools, including command line management tools and a packaging system. Because Snow tries to be portable across implementations, the tools themselves are able to run on a variety of Scheme implementations, and the libraries available in the Snow repository often run on more than one implementation.

Traditionally, authors of Scheme libraries would simply host their files in tarballs or flat files, and would maintain a set of dependencies that their code used. (See, for example, see Oleg Kiselyov's collection of Scheme code [23].) User's wanting to use their code would then either use the semi-portable libraries provided, with a little work, or would attempt to find them in their implementation's repository. This effort has been made somewhat easier by the recent standardization of the R6RS library form [29], which defines a standard library syntax, enabling code to be more easily shared among implementations and users.

Still, there are a wide variety of tools for library management, and many different module systems in active use. Clearly, the cooperation of these various tools, repositories, and implementations would benefit the community as a whole.

Library	Binding	SCM	CVS	Retrieval-method
Archive	Single-file	License	Person	Implementation

Table 1. Descot Classes

3. Descot

The Descot system itself divides roughly into the schema [17], which is the actual language for libraries, the server protocol, which specifies how servers ought to behave, a query protocol, for handling server queries, and an API that assists in the development of Descot servers. Descot itself consists of the first three elements, and the API exists as a convenience for developers.

3.1 Schema

Descot defines an RDF Schema [17, 24] for describing libraries of code. It augments the existing default RDF Schema [5] and is itself written using RDF. RDF is a specification for describing metainformation as directed graphs and has a number of syntactic representations. Current Descot tools support arbitrary representation formats, but by default, use SRDF (see the Appendix). The author chose RDF as the basic metalanaguage because it already has existing tools written around it and is relatively mature. RDF was designed specifically with this sort of problem in mind, and allows extensions as a matter of course. This makes it ideal as the basic language from the perspective of market share and technical features. The XML representation of RDF, however, is tedious and unpleasant to write by hand. SRDF is an S-expression based RDF format designed to mirror Turtle [3]. SRDF makes it easy to write RDF graphs by hand, while remaining easy to manipulate and parse using basic Scheme functions. The author actually began by writing his own S-expression based metalanguage, but soon realized that it was essentially a reimplementation of RDF. By using RDF, many features and semantics may be left to the RDF designers, greatly simplifying the specification of Descot's metalanguage. Descot also supports Turtle out of the box provided that the necessary libraries exist. Since the Schema itself is based on RDF, it is also format neutral; any other RDF format could be used, including, for example, SXML [23]. The Schema itself is a set of URIs to which we ascribe semantic meaning, and is used in the description of RDF Triples. All the URIs start with the prefix:

`http://descot.sacrideo.us/10-rdf-schema#`

All terms mentioned in this section are the tails of URIs prefixed by the above string.

The terms are divided roughly into Classes (see Table 1) and Properties. Most of the properties apply directly to Libraries (Table 3), but there are some general, person, and CVS properties as well (Table 2).

Every class is a type for a specialized node in a Descot Graph. Every node in a descot graph is expected to have a type property associated with it to identify its class.

`Library` nodes represent libraries, and most of the properties stem from Library nodes. Library nodes are also the main root node for most retrievals.

`Binding` nodes represent information about a procedure or macro that is exported or imported from a library. These nodes can be used to store information such as alternate names for procedures. They may also point to documentation about a specific procedure, but the only required property is the name.

name	alts	desc	homepage	e-mail
cvs-root	cvs-module			

Table 2. General/Miscellaneous Descot Properties

deps	names	license
creation	modified	contact
authors	categories	copyright-year
exports	location	implementation
copyright-owner	version	

Table 3. Descot Library Properties

`Archive` nodes contain file archive download information. Generally, they may point directly to the location of an Archive, such as a tarball. As such, these will usually be end nodes in a Descot graph, because they will not contain further information.

`Single-file` nodes are similar to `Archive` nodes, but they point to single Scheme files instead of archives. Generally, single files do not need to be processed by Descot clients further before being fed into a compatible implementation.

`License` nodes contain information about a License type, such as ISC, BSD, GPL, or a proprietary license of some sort. They may point somewhere else as the main reference, and have only a short description of the actual license in the graph, or they may contain the entire text of the license as the description. A short name should be provided that servers can use when they want to display licensing information without presenting the entire description, usually given on one line.

`Person` and `Implementation` nodes follow a similar pattern, describing people and implementations, respectively. People have names and e-mail addresses associated with them, and may have additional information. Implementations generally have a web site and a name associated with them.

`SCM` is a general class for "Source Control" based libraries. That is, `SCM` is a sub-class of `Retrieval-method` like `Archive` and `Single-file` are, but it describes a retrieval via some source control module, like CVS. `CVS` is the sub-class of the `SCM` class that describes CVS server modules particularly. Generally, one would use the `CVS` module or some other equivalent (such as for SVN or Darcs) rather than using `SCM`, but `SCM` properties may be defined to give generic information about a source module to a server that may not recognize the particular type of source control used.

Every node may be associated with a particular `name` which can be anything, and is not specific to the type. Library names are generally strings, but they could be extended to include other information or other types if a server desired. Generally, however, it is recommended to stick with the same types for existing classes, and change the range of the `name` property only for new classes introduced specifically for some specific server or purpose, so that other Descot-compatible systems do not have to work much harder on classes that are already defined.

For any given node, it may also happen that there are alternate nodes that would work in place of the given node. `alts` is expected to point to an rdf `Alt` node that will list the alternates. For example, a library may be implemented by a number of authors, and each library could be listed as an alternate to the others.

`desc` is a property pointing to a string node that contains a description of the node. This could be the license text in the case of a `License` node, or may be a human-readable description of a library for `Library` nodes.

`homepage` can be used where applicable to associate a given homepage to a node. The homepage referenced should be a Resource, and not, for example, a blank node.

The `CVS` node class also has two properties associated with it: `cvs-root` and `cvs-module`. These point to strings which contain the root of the CVS server and the module name for the library, respectively. This is enough information, generally, to obtain the library via CVS, but servers may wish to list additional information, such as the supported protocols for the CVS server.

`email` associates a string representation of an e-mail address with a given `Person` class node. The author did not use e-mail as a unique ID for people because e-mail addresses do not map directly in a one-to-one fashion to people. However, implementations may want to resolve conflicts of people who have the same name by differentiating them by their e-mail addresses.

The following properties all expect to have `Library` nodes as their domains/subjects.

`deps` points to a `List` of Libraries upon which the subject library is dependent.

`names` is a `List` of strings of short library names. These are expected to be alternative short names frequently used to identify the library, as opposed to the long `name` property string, which identifies the normal title of the library.

`exports` is a `List` of `Binding` nodes which represent the procedures and macros that the given library exports.

`license` points to a `License` node that is the license of the given `Library` node.

`authors` is a `List` of `Person` nodes that represents the authors of the library, but not necessarily the maintainer of the Descot metainformation.

`creation` points to a date time string that is the date of creation for the library metainformation, *not* necessarily the creation date of the library itself.

`modified` points to a date time string that represents the date and time of the last modification made to the library metadata, and not necessarily the date and time of the last update to the library itself.

`contact` points to a single person who has claimed responsibility for maintaining the metadata of a given library. This field must exist, and the `authors` property is not a substitute.

`implementation` points to an `Implementation` node, which identifies the implementation or language for which the code was designed to run. This could be a literal implementation, or may be an R6RS `Implementation` node to represent all R6RS compliant Scheme implementations, for example.

`version` is a string that identifies the version of the library. This could be a version number such as "3.5" or it could be something like "-Current". The later is useful for storing the metadata of the latest snapshot of development for a library, such as what one might find from a CVS server.

`location` points to a `Retrieval-method` node or a node of a type that is a sub-class of `Retrieval-method`. This node should tell a Descot client how to obtain the library itself. Notice that this is a very extensible property, and sophisticated servers may provide new `Retrieval-method` sub-classes to describe the details of library retrieval. PLT's

PLaneT, for example, may have a class for libraries that are distributed through the PLaneT packaging system.

`categories` points to a `List` of strings that are categories or tags for the given library. These tags are assumed to be case-insensitive for all intents and purposes.

`copyright-year` and `copyright-owner` are two parts of the Copyright information. `copyright-year` points to a year string, while `copyright-owner` may point to a `Person` or a `List` of `Person` nodes.

3.2 Server Protocol

Descot-compatible servers follow a simple set of rules that allow them to interact with one another. Servers handle three types of requests: mirroring, library/node requests, and queries. Queries are handled in Section 3.3. This section details only mirroring and node requests.

Every server must have a mirroring URI. When a request for this URI comes into the server, the server must respond with the RDF graph consisting of every library node in the server's store with one and only one branch. That branch must be the `modified` property pointing to the last modification time of the referenced library node. In this way, a server which is mirroring the content of another server may identify which libraries need to be updated, and pull only the given information into its own store.

The format of transmission should be arranged in an appropriate manner by the servers or server and client. No specific format is required, and no format need be recognized.

Servers and clients may also make node requests to a server. These are requests for the relevant information about a given node. For example, a client may wish to obtain the metadata for a library for some URI. It does so by accessing the URI and parsing the response from the server. The method of access depends on the protocol specified by the URI. HTTP will likely be a common protocol, but others, such as FTP, Gopher, or HTTPS could also be used. The response should be an RDF graph in either the format requested by the client [server] or the attempt by the server if it does not support the requested format. (Again, the way to request a particular format is protocol dependent, and not specified here.)

The graph returned by a server handling a node request contains a subset of the entire store on the server. Its root or starting node has the URI of the request. The server should then walk the paths going out from the requested URI in the store and return the graph that it walks. The server should stop pursuing a particular path when it encounters a node which has its own unique, accessible URI that can be requested individually. That is, the returned graph contains the descendants or the paths starting from the node with the URI requested, stopping at nodes which themselves have valid URIs. Blank nodes, then, are the only means by which the depth of the graph may grow beyond one. When encountering a blank node while walking the graph, a server will descend into it and continue its walk, but otherwise, the server will not descend into a node, which will have a valid URI if it is not a blank node.

These two request methods provide enough structure for servers and clients to communicate clearly and efficiently. No other behavior is required of a Descot server, though handling query requests is permitted and defined for any Descot server. Most servers will not handle queries, and instead, specific Descot servers will develop to mirror smaller servers and index them to provide a place to search many repositories at once.

3.3 Query Server

Since Descot uses RDF to describe its metadata, it may also utilize the tools available to RDF graphs. SPARQL is a query language and protocol for querying RDF graphs. If a Descot server wishes to provide Querying, then it should follow the protocols and language laid down in the SPARQL specification [28, 7, 2]. Implementing query request handling for a Descot server is not required.

Query-enabled servers enable lightweight clients to interact in useful and interesting ways with servers. Many systems which allow multiple repositories to be used often require that clients cache data about the repositories that it searches. This is fine when there are only a few repositories, but in systems where every developer may potentially have a repository, it may not make sense to cache all the data on every client. While nothing stops a client from caching server data from a Descot server, lightweight clients may use query-enabled Descot servers that mirror other repositories to search and find libraries and code which may have been obscured if the user of the client had to find and install repository information manually.

Query-enabled servers may thus become hubs among the web of Descot servers, providing users the benefit of a central repository, without many of the disadvantages.

3.4 Server API

A Server API has been developed to assist designers in writing Descot servers easily and quickly. They can also be utilized by scripts to assist in dealing with Descot stores. While the current Descot source code contains a number of additional modules, the utilities, printing, and server modules will generally help the most.

This code is currently available via revision control, and a packaged release will be made once some of the features have been completed. This API is the one used by the Descot server that runs (currently only as a proof of concept) at the Descot homepage [18]. The API is provided to assist developers of servers and clients, and implementors may opt to implement the Descot protocol and specification in other ways.

The `rdf-printing` module provides three procedures for printing RDF graphs in Turtle format.

`write-rdf-triple->turtle` takes an RDF triple and an optional port argument, and writes out that triple in Turtle form. `write-rdf-triples->turtle` and `write-rdf-graph->turtle` work the same way but take a list of triples and an RDF graph as their first argument respectively.

The `descot-rdf-utilities` module defines and exports common RDF and Descot URIs for use in other applications. It also defines the following procedures and macros.

`store-categories` : ⟨*graph*⟩ → ⟨*category list*⟩

Produces from an Descot RDF graph a list of all the categories found in the store.

`libraries-in-category` : ⟨*cat*⟩ → ⟨*library list*⟩

Produces a list of libraries that have a category ⟨*cat*⟩.

`in-rdf-list` : ⟨*store*⟩ ⟨*node*⟩ → #⟨void⟩

`in-rdf-list` is a foof loop [6] iterator over RDF `List` nodes. It allows one to iterate over RDF lists in the same way one might iterate over a normal Scheme list.

The iterator is used in `for` clauses of foof loops, as in, `(for elem rest (in-rdf-list store list-head-node))`.

`parse-turtle-file` : ⟨*file*⟩ [⟨*graph*⟩] → ⟨*graph*⟩

Parses a given ⟨*file*⟩ into a given ⟨*graph*⟩ or an empty graph if none is given.

`library-ids` : ⟨*store*⟩ → ⟨*id list*⟩
`library-title` : ⟨*rdf-map*⟩ → ⟨*library name*⟩
`library-names` : ⟨*store*⟩ ⟨*rdf-map*⟩ → ⟨*name list*⟩
`library-description` : ⟨*rdf-map*⟩ → ⟨*desc string*⟩
`library-copyright` : ⟨*store*⟩ ⟨*rdf-map*⟩ → ⟨*copy pair*⟩
`library-homepage` : ⟨*rdf-map*⟩ → ⟨*uri*⟩
`library-license-name` : ⟨*store*⟩ ⟨*rdf-map*⟩ → ⟨*name*⟩
`library-authors` : ⟨*store*⟩ ⟨*rdf-map*⟩ → ⟨*author list*⟩
`library-contact` : ⟨*store*⟩ ⟨*rdf-map*⟩ → ⟨*person pair*⟩
`library-created` : ⟨*rdf-map*⟩ → ⟨*date string*⟩
`library-modified` : ⟨*rdf-map*⟩ → ⟨*date string*⟩
`library-version` : ⟨*rdf-map*⟩ → ⟨*version string*⟩
`library-implementation` : ⟨*store*⟩ ⟨*rdf-map*⟩ → ⟨*impl pair*⟩
`library-location` : ⟨*store*⟩ ⟨*rdf-map*⟩ → ⟨*location*⟩

The above procedures are standard accesser procedures to different elements of a Descot library node. They can be used to quickly get pieces of the graph instead of walking the graph explicitly. ⟨*store*⟩ refers to the Descot store, and ⟨*rdf-map*⟩ refers to a specific RDF map containing the child nodes of a given library node.

The actual `descot-server` module available in the Descot source provides a generalized, format-neutral API for handling server requests. Currently, it handles node requests, mirroring requests, and provides conveniences for handling submissions of new libraries into the existing store.

The Descot Server API uses a file system hierarchy to store the RDF graph in a manner that makes it convenient to retrieve server request information. The entire graph is stored under a single ⟨*root*⟩ directory, and for any subject node with a valid URI, there exists a single file which holds the information necessary to serve a node request for that URI. The path to this file is formed by the following scheme:

```
<root>/<scheme>/<domain>/<path>[#<fragment>]
```

where ⟨*domain*⟩ is the domain of the URI with the terms reversed and separated by forward slashes rather than dots. The API provides a procedure for generating this path from a given URI:

`descot-uri->store-path` : ⟨*uri string*⟩ → ⟨*path string*⟩

and also defines a parameter `descot-store` to hold the root location.

The API also defines reader and writer parameters for the store. The reader parameter `descot-api-reader` contains a procedure

`reader` : ⟨*fname*⟩ [⟨*graph*⟩] → ⟨*rdf graph*⟩

that will read the files in the store. This allows the format of the store to be any format for which an user can provide a proper reader. This parameter defaults to `parse-srdf-file` from the `srdf` module (see the Appendix).

The `descot-api-triples-writer` parameter holds a procedure that will be used whenever a graph must be written to a file. It defaults to `write-rdf-triples->srdf` and any procedure that replaces the default should have the same signature (see the Appendix). This writer is also used when no preferred format is detected for an incoming node request. Since detection of format preference is not yet built into the API, this parameter effectively controls all RDF output from the API, and not just the format from the store.

The above parameters are used to separate the api from the format of the repository. They are not expected to change after initializing a server using this API.

`write-descot-request` : ⟨*subject uri*⟩ ⟨*port*⟩ → #⟨void⟩

`write-descot-request` handles node requests for the server and writes out the proper response to the given ⟨*port*⟩.

`write-descot-updates` : ⟨*port*⟩ → #⟨void⟩

`write-descot-updates` writes out the mirroring graph to the given port.

`write-descot-store` : ⟨*graph*⟩ → #⟨void⟩

When new libraries are submitted to a server, normally they will go through a vetting process, after which, they must be stored in the main repository database. `write-descot-store` allows a store to be written safely to the store and is the main procedure to use when adding new data to the store.

Since the API does not yet provide enough detailed access to make direct graph walking along the graph easy, a convenience procedure is exported from the server API to allow applications to read in the entire store for work.

`read-descot-store` : ⟨*root*⟩ → ⟨*RDF graph*⟩

It works with any subdirectory of the root location and the root location itself as the ⟨*root*⟩ value, so one can selectively graph pieces of a graph if necessary.

3.5 Example

The following is a complete example of a relatively self contained graph with all the information necessary to serve all the node requests. It is written in the SRDF format defined in the Appendix.

```
(= authors
   "http://descot.sacrideo.us/rdf/authors/")
(= impls
   "http://descot.sacrideo.us/rdf/impls/")
(= licenses
   "http://descot.sacrideo.us/rdf/licenses/")
(= bindings
   "http://descot.sacrideo.us/rdf/bindings/")
(= dscts
   "http://descot.sacrideo.us/10-rdf-schema#")
(= rdf
 "http://www.w3.org/1999/02/22-rdf-syntax-ns#")
(= xsd
   "http://www.w3.org/2001/XMLSchema#")
(= dsct
   "http://descot.sacrideo.us/rdf/libs/system/")

((: dsct "malloc#chez")
 ((: rdf "type") (: dscts "Library"))
 ((: dscts "name")
  (& "Garbage Collected Malloc" en))
 ((: dscts "names")
  (($ "malloc") ($ "gc-malloc")))
 ((: dscts "desc")
  ($ "Create malloced regions of memory that
      are handled by the garbage collector."))
 ((: dscts "exports") ((: bindings "gc-malloc")))
 ((: dscts "license")
  (: licenses "public-domain"))
```

```
 ((: dscts "authors") ((: authors "dybvig")))
 ((: dscts "creation")
  (^ "2009/03/08 23:33:10" (: xsd "dateTime")))
 ((: dscts "modified")
  (^ "2009/05/12 00:41:44" (: xsd "dateTime")))
 ((: dscts "copyright-year")
  (^ "2008" (: xsd "gYear")))
 ((: dscts "copyright-owner")
  (: authors "dybvig"))
 ((: dscts "contact") (: authors "arcfide"))
 ((: dscts "version") ($ "1.0"))
 ((: dscts "location")
  (* ((: rdf "type") (: dscts "CVS"))
     ((: dscts "cvs-root")
      ($ "anoncvs@anoncvs.sacrideo.us:/cvs"))
     ((: dscts "cvs-module") ($ "lib/malloc.ss"))))
 ((: dscts "implementation") (: impls "chez"))
 ((: dscts "categories") (($ "system"))))

((: licenses "public-domain")
 ((: rdf "type") (: dscts "Licenses"))
 ((: dscts "name") ($ "Public Domain")))

((: bindings "gc-malloc")
 ((: rdf "type") (: dscts "Binding"))
 ((: dscts "name") ($ "malloc"))
 ((: dscts "desc")
  ($ "Garbage Collected Malloc")))

((: authors "dybvig")
 ((: rdf "type") (: dscts "Person"))
 ((: dscts "name") ($ "R. Kent Dybvig"))
 ((: dscts "email") ($ "dyb@scheme.com"))
 ((: dscts "homepage") "http://www.scheme.com"))

((: authors "arcfide")
 ((: rdf "type") (: dscts "Person"))
 ((: dscts "name") ($ "Aaron W. Hsu"))
 ((: dscts "email") ($ "arcfide@sacrideo.us"))
 ((: dscts "homepage")
  "http://www.sacrideo.us"))

((: impls "chez")
 ((: rdf "type") (: dscts "Implementation"))
 ((: dscts "name") ($ "Chez Scheme"))
 ((: dscts "homepage") "http://www.scheme.com"))
```

If a node request came it, it would come for one of the top-level s-expressions defined above. The data transmitted back to the requesting client would be equivalent to the data contained in that top-level s-expression. That is, if a request for

```
(: dsct "malloc#chez")
```

came in to a server, it would return only the data found in the s-expression above that has

```
(: dsct "malloc#chez")
```

as the first element. The server would ignore the other top-level s-expressions.

4. Related Work

Since Descot only describes a library and does not attempt to make it portable across implementations or languages, efforts to make portable code, such as those from Snow [12] and especially module systems like R6RS libraries [29] contribute invaluable features to a complete repository system.

Snow is only one of many repositories that exist in Scheme, each with its own unique features and focus. These include library suites such as SLIB [20] and implementation-specific repositories such as those found for PLT [26], Chicken [31], and Bigloo [30].

Other attempts at portable library repositories include CSAN [9] and CxAN [27]. The latter is unique because it is not a Scheme specific project.

Implementations often support libraries internally without making them into separate libraries, or they may package libraries with their distributions, which makes it interesting to deal with that information. Descot is general enough to represent these internal libraries, which almost all Scheme implementations have, even though they are not generally considered repositories [22, 11, 25].

Other projects have created distributed networks of repositories quite successfully, though not specifically focused on library code distribution. The Debian packaging system [1], often known as apt, caches server information on the clients to enable multiple repositories to be used by one client. The client can then download the desired packages and install them as appropriate. System such as the RPM-based [10] yum [14] also behave in a similiar manner. An user specifices a series of repositories to use, and the client caches information about the software packages available from the repositories listed. Sites such as RPMfind [4] also make packages available via web browser. These clients will often scan many repositories over all different distributions to obtain their indexes.

While the above are similiar to Descot by their distributed nature, the packages they references are actual software packages and contain all the binaries or source code inside them. The BSD family of operating systems (and Gentoo, which follows a similar pattern [13]) uses a series of files that contain metadata about how to build and install a given software package. Descot's metadata representation more closely resembles these so called ports systems than the packaging used by systems like apt or yum. When, say, an OpenBSD user wishes to build a package, rather than install it via binary package, the user would navigate to a prefilled filesystem containing port metadata. The user would then run a command that would fetch, build, and install the package [16]. Similar tools could be made for Descot repositories.

5. Conclusion

The Descot system described above provides the means by which fragmented or diverse communities can cooperate and leverage development efforts that previously existed in isolation of one another. Since most communities do not lack for tools or repositories of code, but rather, a means of common access, Descot focuses entirely on fostering the communication among existing systems, rather than trying to rewrite existing tools and change previous workflows. Since Descot is extensible and dynamic, it can fit into a wide range of domains, and can adapt to handle the needs of a community, rather than trying to fit different communities or sub-cultures into a single methodology. Descot is built on common, well documented technologies and so should easily travel where less standards-based systems may not. Descot provides an open, clearly specified infrastructure so that communities can collaborate together and avoid redundant work. It provides the convenience of central code distribution

without forcing large, top-down changes on a community that may not respond well to such pressure.

6. Acknowledgments

The author would like to thank Kent Dybvig for his comments, which led to improvements in the presentation this this paper.

References

[1] Osamu Aoki. *Debian Reference*, June 2009. *http://www.debian.org/doc/manuals/debian-reference/*.

[2] Dave Beckett and Jeen Broekstra. Sparql query results xml format. W3c recommendation, W3C, January 2008. *http://www.w3.org/TR/rdf-sparql-XMLres/*.

[3] David Beckett and Tim Berners-Lee. Turtle - terse rdf triple language. W3c team submission, W3C, January 2008. *http://www.w3.org/TeamSubmission/turtle/*.

[4] Fabrice Bellet. Rpmfind, June 2009. *http://www.rpmfind.net*.

[5] Dan Brickley and R. V. Guha. Rdf vocabulary description language 1.0: Rdf schema. W3c recommendation, W3C, February 2004. *http://www.w3.org/RDF/*.

[6] Taylor Campbell. foof loop, June 2009. *http://mumble.net/~campbell/darcs/foof-loop/loop.scm*.

[7] Kendall Grant Clark, Lee Feigenbaum, and Elias Torres. Sparql protocol for rdf. W3c recommendation, W3C, January 2008. *http://www.w3.org/TR/rdf-sparql-protocol/*.

[8] William Clinger and Jonathan Rees. *Revised*[4] *Report on the Algorithmic Language Scheme*, September 1991. ftp://ftp.cs.indiana.edu/pub/scheme-repository/doc/standards/r4rs.ps.gz.

[9] CSAN. Comprehensive scheme archive network, June 2009. *http://www.cliki.net/Community*.

[10] Alexandre de Abreu. *All you have to know about RPM*, March 2004. *http://fedoranews.org/alex/tutorial/rpm/*.

[11] R. Kent Dybvig. *Chez Scheme Version 7 User's Guide*. Cadence Research Systems, July 2007. *http://www.scheme.com/csug7/*.

[12] Marc Feeley. *Scheme Now! Documentation*, June 2009. *http://snow.iro.umontreal.ca/?tab=Documentation*.

[13] Gentoo Foundation. Gentoo linux, June 2009. *http://www.gentoo.org*.

[14] Michael Hideo. *Red Hat Enterprise Linux 5 Deployment Guide*. Red Hat Inc., Raleigh, NC, 5 edition, November 2008. *http://www.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/5/html/Deployment_Guide/index.html*.

[15] Jarkko Hietaniemi. Comprehensive perl archive network. http://www.cpan.org.

[16] Nick Holland. *The OpenBSD packages and ports system*. OpenBSD, May 2009. *http://www.openbsd.org*.

[17] Aaron W. Hsu. Descot rdf schema. Rdf schema, May 2009. *http://descot.sacrideo.us/10-rdf-schema*.

[18] Aaron W. Hsu. Descot technical documentation. Programmer's documentation, May 2009. *http://descot.sacrideo.us*.

[19] IEEE. *1178-1990 IEEE Standard for the Scheme Programming Lanuage*, 1990.

[20] Aubrey Jaffer. *SLIB: The Portable Scheme Library*, February 2008. *http://people.csail.mit.edu/jaffer/slib_toc.html*.

[21] Richard Kelsey, William Clinger, and Jonathan Rees. *Revised*[5] *Report on the Algorithmic Language Scheme*, February 1998. http://www.schemers.org/Documents/Standards/R5RS/r5rs.ps.

[22] Richard Kelsey, Jonathan Rees, and Mike Sperber. *The Incomplete Scheme 48 Reference Manual for release 1.8*, January 2008. *http://www.s48.org/1.8/manual/manual.html*.

[23] Oleg Kiselyov. Scheme hash, June 2009. *http://okmij.org/ftp/Scheme*.

[24] Frank Manola and Eric Miller. Rdf primer. W3c recommendation, W3C, February 2004. *http://www.w3.org/TR/rdf-primer/*.

[25] Massachusetts Institute of Technology. *MIT/GNU Scheme 7.7.90+ Reference Manual*, 2008. *http://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/index.html*.

[26] Jacob Matthews. PLaneT: Automatic package distribution. Reference Manual PLT-TR2009-planet-v4.2, PLT Scheme Inc., June 2009. *http://plt-scheme.org/techreports/*.

[27] Hans Oesterholt. Cxan, July 2004. *http://cxan.sourceforge.net/*.

[28] Eric Prud'hommeaux and Andy Seaborne. Sparql query language for rdf. W3c recommendation, W3C, January 2008. *http://www.w3.org/TR/rdf-sparql-query/*.

[29] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten. *Revised*[6] *Report on the Algorithmic Language Scheme*, September 2007. http://www.r6rs.org/final/r6rs.pdf.

[30] Vladimir Tsichevski. *Bigloo libraries*, December 2003. *http://bigloo-lib.sourceforge.net/*.

[31] Felix Winkelmann. *The CHICKEN User's Manual*, April 2009. *http://chicken.wiki.br/man/4/The User's Manual*.

Appendix: SRDF Format

SRDF is an s-expression based format for describing RDF graphs. It is meant to be mostly equivalent in its form to Turtle. Since the language is S-expression based, it is easier for Scheme and Lisp parsers to parse it. Parsers for other languages can also be written very easily. This makes it particularly nice for use in automated systems or in areas where S-expressions are the natural representation format. SRDF is designed to work for most Scheme's `read` procedures.

SRDF documents are composed of a series of RDF triples and, possibly, prefix definitions. Prefixes take the form `(= name "uri")`, and associate a given Scheme symbol with a URI string. Otherwise, the form is an RDF triple or a set of triples.

Normal triples are just a list of three elements, each a URI. Multiple triples with the same subject can be declared in one expression by replacing the list that would hold the single predicate and object with a list of such predicates and objects. Likewise, one can specify more objects to be associated with a given subject and predicate by doing the same thing with the object list, and replacing the list tail that would normally hold the object with a list of such objects.

If the second element of a predicate pair contains a list of objects, this represents a collection of objects, and is created in the same way that a turtle collection syntax is created: by associating a series of blank nodes with the right predicates with each of the objects listed.

An object that differs from a list of objects that are each associated with the subject and predicate. The following is an instance of the former:

```
("subject-uri" "predicate-uri"
  ("object1" "object2" ...))
```

Whereas the following is an instance of the latter:

```
("subject-uri" "predicate-uri"
  "object1"
  "object2"
  "object3")
```

Normal RDF triples take the form:

```
("subject-uri" "predicate-uri" "object-uri")
```

A blank node may be inlined into the graph by using a '*' as the beginning symbol in an object context like so:

```
("subject" "pred" (* "pred" "object"))
```

Of course, blank nodes may have anything that is a valid predicate `cdr` as its `cdr` so the following is also valid:

```
("subject" "pred"
 (* ("pred1" "object1") ("pred2" "object2")))
```

URIs may be described by their full path names as strings, as prefix combined paths, or as blank node paths. The following are all valid URIs:

```
"http://some.domain/path/to#blah"
"blah"
(: prefix "blah")
(_ "uniqueid")
```

We use ':' for prefixes and '_' for blank nodes. In addition to URIs, we permit literals as valid `car`s for objects. Literals can be strings, numbers, booleans, or may be strings with either languages or types associated with them. The following are examples of languages and types, respectively:

```
($ "Language unspecified.")
(& "English Sentence lies here." en)
(^ "2008/01/03 14:00" (: xsd "date"))
```

The following is a fairly formal BNF grammar with the exception of tokens such as strings, numbers, and booleans being undefined and presumed to be defined lexical values. Additionally, we define S-expression in terms of atoms and pairs, so the BNF grammar is also defined in the "longhand" notation for pairs and lists. This means that while the BNF Grammar states something like `("subj" . ("pred" . ("obj" . ())))` as the valid simplistic RDF triple, it is also legal in practice to use the shorthand version of this: `("subj" "pred" "obj")`

⟨*rdf sexp*⟩→ ⟨*rdf triple*⟩ | ⟨*rdf triple*⟩ ⟨*rdf sexp*⟩

⟨*rdf triple*⟩→ '(' ⟨*uri*⟩ '.' ⟨*rdf subject tail*⟩ ')'
| '(' '=' name string ')'

⟨*rdf subject tail*→ ⟨*rdf predicate*⟩
| ⟨*rdf predicate list*⟩

⟨*rdf pred list*⟩→ '()'
| '(' ⟨*rdf predicate*⟩ '.' ⟨*rdf pred list*⟩ ')'

⟨*rdf predicate*⟩→ '(' ⟨*uri*⟩ '.' ⟨*rdf object list*⟩ ')'

⟨*rdf object list*⟩→ '()'
| '(' ⟨*rdf object*⟩ '.' ⟨*rdf object list*⟩ ')'

⟨*rdf object*⟩→ ⟨*uri*⟩ | ⟨*literal*⟩
| ⟨*rdf object list*⟩ | ⟨*blank node list*⟩

⟨*blank node list*⟩→ '(' '*' '.' ⟨*rdf subject tail*⟩ ')'

⟨*uri*⟩→ *uri*
| '(' ':' '.' ⟨*uri list*⟩ ')'
| '(' '_' *name* ')'

⟨*uri list*⟩→ '()'
| '(' ⟨*uri name*⟩ '.' ⟨*uri list*⟩ ')'

⟨*uri name*⟩→ *uri* | *name*

⟨*literal*⟩→ *number* | *boolean*
| '(' '$' *string* ')'
| '(' '&' *string name* ')'
| '(' '^' *string* ⟨*uri*⟩ ')'