Camelis 1.3

A Caml implementation of
Logical Information Systems

User's Manual

Sébastien Ferré

13th June 2007

1 Introduction

The original idea of Logical Information Systems (LIS) comes from Olivier Ridoux, who was unsatisfied by hierarchical file systems in particular, and hierarchical data organizations in general. He was not satisfied either by databases because they lack flexibility in the description of objects, integration with the operating system and other applications, and navigation capability to help non-expert users. His basic idea was to combine the expressivity of querying, by the use of logics, and the practicality of navigation as a way to suggest query increments.

When in 1999 I looked for a research subject for my master thesis, I was immediately convinced by these ideas, and eager to work on them. I was lucky to do my PhD on Logical Information Systems under supervision of Olivier Ridoux, and defended it in October 2002 [Fer02, FR04]. Its main result is to found LIS on logics and Formal Concept Analysis (FCA) [Wil82, GW99]. In parallel to theoretical works I have developped a LIS prototype, Camelis, that implements most of the ideas of my PhD thesis (and more), and is now mature enough, I think, to be distributed. However it is still a continuously evolving research prototype, and the main reason for distributing it, beside its possible usefulness, is to get some feedback about LIS.

A file system track has been started by Yoann Padioleau, who defended his PhD in February 2005. The main result is LISFS (a.k.a. LFS), a LIS implementation as a file system plus the ability to navigate into parts of files [PR03, Pad05]. This makes LIS benefits to existing applications without modification. Camelis and Lisfs are not in concurrency as they stand at different levels of use. On the contrary it is intended that the two tracks converge and finally fuse at some horizon.

The aim of this manual is firstly to explain the main concepts of LIS, and secondly to document the various displays and commands of Camelis. The central concept is the *context*, which comes from the theory of FCA. In short a context is the combination of a *logic*, a set of *objects*, and a mapping from objects to logical formulas. Section 2 and Section 3 respectively explain what is a logic, and what is an object. Further sections present operations that can be performed on a context, and how they are achieved through the Camelis interface: these operations are *browsing*, *importing* and *exporting* data, *updating*, and *acting* on objects. Section 8 shows how persistency is achieved from one session to another.

Camelis is a very generic system in that the logic and the type of imported/exported data can be changed at will. Section 9 presents a general purpose instance of Camelis (called a Camelis application) that can handle all sorts of files, and especially JPEG pictures, MP3 music files, BibTeX bibliography files, etc.

For more information about LIS, there is a web page at `http://www.irisa.fr/LIS` that contains links to people, papers, talks, and software.

2 Logic

In simple terms, a *logic* is a language of *formulas* equiped with an generalization ordering called *subsumption*. Almost everything in Camelis is represented by

formulas so that the various operations on a context can be expressed in a very uniform way. Formulas are used to describe the properties of an object, the features common to a set of objects, complex boolean queries, and updates.

Each CAMELIS application is given a custom logic L upon which are defined the various kinds of formulas. Each logic comes with the following elements:

- a sub-language L_D of *object descriptors*,
- a sub-language L_F of *features*,
- a generalization ordering, the *subsumption*, over formulas, where $f \sqsubseteq g$ means that formula g is more general than formula f,
- two *update operations add* (resp. *sub*) to make an object descriptor be subsumed (resp. not be subsumed) by some object descriptor (resp. by some formula).

L_D and L_F may be equal to L. The subsumption is assumed to be consistent, complete between object descriptors and features, but not necessarily fully complete. This is necessary and sufficient to ensure that no object is misclassified. The update operations may be only partially defined.

From this low-level logic, top-level formulas can be defined as follows:

- an *object description* is a set of object descriptors, and is interpreted under the Closed World Assumption, i.e., everything not said is considered as false,
- a *feature* is simply a feature,
- a *query* is a boolean combination of features, where connectors are `and`, `or`, `not`, `except`, and `all`,
- an *update* is a conjunction of object descriptors, negations of object descriptors, and negations of features.

Then the subsumption ordering is extended as follows:

- an object description D is subsumed by a feature f if there exists an object descriptor $d \in D$ such that $d \sqsubseteq f$,
- the subsumption of an object description D by a query q is defined according to the usual semantics of boolean connectors (e.g., $D \sqsubseteq q_1$ and q_2 iff $D \sqsubseteq q_1$ and $D \sqsubseteq q_2$, and $D \sqsubseteq$ not q iff $D \not\sqsubseteq q$).

Finally the update operation is extended as follows:

- the update of D by u_1 and u_2 is equal to the successive update of D by u_1 and u_2,
- the update of D by d is obtained by applying the update $add(d', d)$ to each descriptor d' in D when defined, and by adding the descriptor d to D when no addition was defined,
- the update of D by f is obtained by applying the update $sub(d', f)$ to each descriptor d' in D when defined, and by removing it from D otherwise.

Logics may also be equipped with the operation *axiom* that takes two features f, g, and makes sure that $f \sqsubseteq g$. This enables some customization of the logic after it has been defined and linked with CAMELIS. Depending on the semantics of the logic, the introduction of one axiom may produce several subsumption relations or none.

3 Object

An *object* is characterized as any entity that may be retrieved and/or given descriptors by users. Each object is uniquely identified by an *oid* (object identifier), and associated to a logical description (see Section 2).

Most often objects represent entities out of the context, mainly files, web pages, or parts of them (e.g., pictures, BibTeX entries, emails, URLs, songs). This makes it possible to automatically extract object descriptors from the content of these external entities (e.g., MP3 tags, sender and subject of emails, size of pictures). However not every descriptor can be automatically extracted, and users can add their own descriptors to objects (e.g., priority of an email, ranking of songs, event associated to a picture). The former descriptors are said *intrinsic*, while the latter are said *extrinsic*. Both kinds of descriptors are important as intrinsic descriptors save a lot of time to users, and extrinsic descriptors allows an arbitrary personalisation of descriptions.

From each object description a set of features is automatically extracted. Each extracted feature subsumes the description and represents a more or less general aspect of the object (e.g., a title word, the genre of a song, a size interval). Without these features the classification of objects would be completely flat as object descriptions are often unique and incomparable. They play a central role in browsing a context, which is presented in the next section.

4 Browsing

The main function of CAMELIS is to browse a context. The specificity of CAMELIS browsing is to allow for an arbitrary combination of querying and navigation. Every *query* defines a *navigation place*, and reciprocally every navigation place is defined by a query. A query also defines an *extent* that is the set of objects whose description is subsumed by the query. So a navigation place can always be seen at the logic-level through the query, and at the object-level through the extent.

As a consequence of the equivalence between queries and navigation places, *navigation links* are query increments, and *query increments* are navigation links. These are in fact features that can be combined in various ways with the current query resulting in a new query, navigation place, and extent. Last but not least, navigation links/query increments are automatically computed depending on the current query and the context, so that each link be relevant to the context, i.e. it never happens that a navigation link leads to a navigation place whose extent is empty. As in Web browsers it is possible to go back and forward in the history of visited queries.

The graphical interface is organized as follows:

- navigation window: the scrollable window at the left contains trees of links/increments as tabs,
- query area: the text area at the top contains the current query,
- extent window: the scrollable window at the right contains a list of extent objects, and shows at the top the range of displayed objects and the number of objects in the current extent.

4.1 The Navigation Window and Query Area

The navigation window displays a set of trees of features as tabs that initially contains a single tree, whose root is the most general feature and that is completely collapsed, so that only most general features are visible. More specific features can be made visible by expanding any element in any tree. For ease of navigation between different views, a feature can be expanded as the root of a new tab by activating the command `Expand as tab` in the contextual menu of the navigation window. This new tab can be seen as a zoom on the full feature tree.

Note that a same feature may appear more than once in the trees. This is because the set of features is in fact partially ordered by subsumption and is not a hierarchy. A feature can be inserted into a tree through the command `Logic > Add Feature...`. Where it is inserted is automatically determined by the subsumption relation. A set of selected features can be hidden through the command `Logic > Delete Features`. These 2 commands enable users to customize the navigation window according to their needs.

The number on the left of each feature is the number of objects in the current extent that is subsumed by the feature, i.e. the *support* of the feature. The sum of these numbers is usually greater than the size of the extent because objects are generally subsumed by many features. There lies a big difference with hierarchical systems where objects belong to only one path in the tree. This means that, for example, a user doesn't have to choose a song either by its artist or by its genre, but can successively select an artist and a genre in any order. Furthermore, after selecting an artist, only relevant genres are suggested, and reciprocally, after selecting a genre, only relevant artists are suggested.

The initial query is `all` so that the initial extent is the set of all objects of the context. Then this query can be modified by selecting query increments in some tree of features. The default behavior is to make the new query be the conjunction of the old query and the selected feature (connector `and`), and is obtained by double-clicking the feature in the navigation tree. This behavior is altered when the selected feature is orange- or red-colored, as in this case all selected objects already satisfy it, and then the extent would not be modified:

- if the feature is red, i.e. it is present in the query, it is removed,
- otherwise, if it is orange, it replaces any feature in the query that is more specific.

In both cases the resulting extent may be extended rather than restricted, providing upward navigation. For instance, suppose the current query is `France and Landscape` after selecting `France` then `Landscape`. Either feature can be removed from the query by selecting it in the navigation tree (and not only the

last one as usual in browsers). The query can also be generalized to `Europe and Landscape` simply by selecting `Europe` as a feature more general than `France`. This is particularly useful when queries get complex.

The browsing menu and the contextual menu of the navigation window allows for other combinations between the current query and selected features:

- new query = current query `and` feature (default),
- new query = current query `except` feature,
- new query = current query `or` feature,
- new query = feature,
- new query = `not` feature.

Moreover, the feature is replaced by a disjunction of features (connector `or`, which has higher precedence than `and`) if several features are selected. This enables to purport some choice, like when going directly from the query `Landscape` to the query `Landscape and France or Germany`. Alternately, an arbitrary new query can be defined simply by editing the query area. The button `Home` is a shorthand to reset the query to `all`.

Each time a new query is defined, both navigation and extent windows are refreshed, and it is added to the history of visited queries. Users can move backward and forward like in Web browsers through commands `Back` and `Forward` (available in the browsing menu, and as buttons). The additional command `Refresh` enables the refreshment of both navigation and extent windows when necessary.

When expanding a feature in order to see more specific features, default options are used. These options can be customized on each feature by the command `Expand...` in the contextual menu. We now describe these options and give their default value.

- *logical (vs. decreasing support) sorting of increments* (off): displays increments in logical order (lexicographic for strings, numerical for integers, chronological for dates and times, etc.) instead of the default decreasing support order.
- *conceptual (vs. logical) ordering* (off): the default behavior is to display query increments that are maximal for the subsumption (logical ordering). This option helps to reduce the number of increments by displaying query increments that have maximal extents (conceptual ordering). In particular it enables to make apparent a hierarchical structure after it has been flatten.
- *minimal support of increments* (auto): specifies the minimal support for a feature to be displayed. By default this support is computed automatically so that the set increments under a tree node has a reazonable size. When set to 0 it makes it possible to see every existing features and descriptors (this is useful when classifying new objects in existing classes).
- *regular expression as an increment filter* (off): restricts the search for increments to features whose representation matches the given Unix-like regular expression.

Expanding options defined on some feature are inherited by sub-features. Expanding options of the root (invisible) can be defined though the command `Expand...` in the browsing menu.

In addition to inserting and hidding features, users can customize the tree of features by expressing *axioms*. An axiom is specified by a pair of features (f, g), so that the first become subsumed by the second. The support for axioms depends on the chosen logic, and expressing an axiom may have no effect. An axiom (f, g) cannot be removed directly, but only by stating an axiom (f, h) where h subsumes g. The most common use of axioms is for building a taxonomy of terms. The first way to state an axiom is by a copy and paste mechanism available from both the contextual menu of the navigation window, and the logic menu: to state an axiom (f, g) first select and copy feature f, then select feature g, and paste. In order to state all axioms between 2 sets of features, simply selects several features where relevant. When no feature is selected when pasting, features from the current query are used instead. This is useful when the subsuming feature is the root feature `all` as it is not visible in the navigation window. The second way to state an axiom is simply to drag a feature on another feature.

Unlike hierarchies that are built top-down, taxonomies in LIS are rather built bottom-up, i.e. from more specific to more general features, because objects are described first, and classes emerges from these descriptions. So a common operation is to join a set of features under a more general feature. This is precisely what is provided by the command `Join...` in the contextual and logic menus: first select the specific terms, then apply the command `Join...`, which asks for a general joining feature.

4.2 The Extent Window

The extent window displays a partial list of objects in the current extent. The reason for displaying only a partial list is that displaying of very large extents is time-consuming and of no practical use. The size of the current extent is visible at the top right of the extent window, and the range of the displayed objects at the top left. The arrows on both sides of these figures provides page-scrolling in the extent: respectively from left to right, first page, next page, previous page, last page.

Each object is presented on a line by a preview, which can be a text, a picture thumbnail or both. A contextual menu is accessible from the extent window with operations applying to selected objects. The first commands are possible actions on selected objects, and are based on their action arguments (see Section 7). The last command provides options on the extent window such as:

- *local objects only*: only objects that have no more property than specified in the current query are displayed,
- *page start*: offset of the first object to be displayed,
- *page size*: number of objects to be displayed per page.

The one but last command of the contextual menu, `Intent`, is an additional navigation mechanism that allows to query a context from examples. Given a

set of selected objects this command sets the current query to the conjunction of features that are common to all selected objects. When applied on only one object, this can be performed by double-clicking on it, and makes visible all its descriptors in the navigation window.

Other commands are about updating objects, and are described in Section 6.

5 Importing and Exporting

Although it is possible to create objects from scratch as shown in Section 6, it is often useful that objects of the context reflect objects existing outside of Camelis contexts, i.e. files or parts of files residing in the file system (Section 5.1). Reciprocally it is also useful to make sets of objects be reflected by files, e.g. playlists for music or slide shows for pictures (Section 5.2). In addition to files, it is possible to import/export contexts (or parts of contexts) in order to share them with other users or applications (Section 5.3). Import and export commands are available from the menu `File`.

5.1 Importing Files and Directories

Directories and files can be imported into a context by activating the menu command `File > Import > Import file...`. Directories and files can be specified as local path or as web locations. This command opens a dialog where a file or a directory must be selected, and that offers a few options:

- *recursive traversal of directories* (on): if a directory is selected, recursively traverses the tree structure below it.
- *extract file parts when applicable* (off): extract relevant file parts as objects. For instance a bookmarks file can be seen as one object representing the file, or as a set of objects representing URLs in addition to the object representing the file.
- *filter files from their suffix* (off): when recursively traversing directories, import only files whose extension match one the given suffixes.
- *put objects in the current query* (off): use the current query as an initial update for objects produced by the selected files.

Once a file has been imported it becomes a registered *source* of the current context. The kind of files (and directories) that can be imported, and the way a set of objects is produced and logically described from these files, depend on the Camelis application, and more precisely on the chosen *source* modules. An example Camelis application is detailed in Section 9.

The effect of registering a new source can be seen as the top right little text entry contains the number of elementary updates that remain to be performed: the *update count*. When this count reaches 0, the navigation and extent windows are refreshed in order to reflect changes in the context. However it is possible to refresh these windows at any time by pushing the button `Refresh`.

The registered sources can be displayed as a dialog window by the menu command `File > Import > List sources...`. It contains the tree of source locations and a list of action buttons. A source is red-colored when its location

does not exists anymore, and green-colored otherwise. The buttons have the following effect:

- `Refresh`: refresh the tree of source locations,
- `Add`: add a new source like the command `File > Import > Import file...`,
- `Move`: relocate the selected source by specifying a new path in the file system,
- `Paste`: update the query associated to the selected sources (detailed in Section 6),
- `Delete`: delete the selected sources as well as all objects produced from them,
- `Clear`: deleter all sources whose location is lost (red-colored sources),
- `Update`: update the context w.r.t. the selected sources when they have changed,
- `Close`: close the source tree window.

Two shorthands for updating all sources are the menu command `File > Import > Update all`, and the button `Update`. The update count at the top right of the main window reflects the number of elementary updates to be performed.

5.2 Exporting Extents as Files

A set of objects can be exported as a file provided it is the extent of some query. Indeed it is more meaningful to specify an export by a query, i.e. a combination of features, than by an explicit set of objects. In this way each time an object appears or disappears from the extent of the query, the file is updated accordingly. For registering an export file, first set the current query to the desired one, then activate the menu command `File > Export > Export file...`, and select a path in the file system. This file is then registered as a *well*. The way a query extent is converted into a file depends on the CAMELIS application, and more precisely on the chosen *well* modules. An example CAMELIS application is detailed in Section 9.

The registered wells can be displayed as a dialog window by the menu command `File > Export > List wells...`. It contains the tree of wells (location and query) and a list of action buttons:

- `Refresh`: refresh the tree of wells,
- `Add`: add a new well like with command `Export file...`,
- `Move`: relocate the selected well by specifying a new path in the file system,
- `Paste`: replace the query associated to the selected wells by the current query,
- `Delete`: delete the selected wells (this may delete the associated files),
- `Close`: close the well tree window.

5.3 Importing and Exporting Contexts

As explained in Section 8 all elements of a context (objects, sources, wells, etc.) are saved in one large binary file. This makes it difficult to share data from one context to another. For instance if someone has spent time defining a taxonomy of terms with axioms (e.g. music genres), it may be useful to reuse it on different but similar data. This is solved by exporting and importing context data through textual files with extension `.ctx`.

These context files are simply lists of commands, one per line, and can convey any element of a context. These elements are separated in several categories:

- sources (command `src`),
- axioms (command `axiom`),
- extrinsic data on objects (commands `mk`, and `mv`),
- features visible in the navigation tree (commands `show`, and `hide`),
- wells (command `wll`),
- update rules (command `rule`, see Section 6),
- actions (command `action`, see Section 7).

When importing or exporting in a context, any of these categories can be unselected in order to control what kind of data is shared. Another way to control the scope of exporting a context is the current query as only objects in the current extent are considered for the export of extrinsic data.

In the case where the current query is `all`, and all categories are selected, the full context can be exported and regenerated by importing the context file in a new context. This is useful to transfer a context between 2 different Camelis applications or versions whose binary formats are incompatible.

6 Updating

In previous section objects where generated by importing files and directories. However it is also possible to create purely extrinsic objects that are under full control of Camelis. This is done by activating the command `Updating > Add object`. The initial description of the new object is made by applying the *selected update* to the empty description. The selected update is the conjunction of selected features in the navigation window, or alternately the current query (see Section 2 for the definition and use of updates).

Whereas intrinsic descriptors are under control of the source of objects, extrinsic descriptors can be added, changed, and removed from the description of objects, whether these objects have been generated from a source or not.

The common scope of an update is a set of selected objects, but we will see how this can be generalized to sources and queries. Before applying an update on objects, a set of objects must be selected through the command `Updating > Copy objects`, which is also available in the contextual menu of the extent window. It is important to note that, when no object is selected, the full current extent is taken as default.

Once a scope has been defined (copy), and an update has been selected, this update is applied on each selected object by activating one of the command `Updating > Paste` and `Updating > Paste not`, which are also available in the contextual menu of the navigation window, and also as a button. The latter applies negation on each feature of the selected update, which has the effect of removing these features from the descriptions of objects, instead of adding them with the command `Paste`. When the scope covers only one object, and the selected update is made of only one feature, the copy and paste operation can be performed by a simple drag-and-drop from the object onto the feature.

Adding and deleting objects can be seen as special cases of the copy-and-paste mechanism. Adding an object, as defined above, is like copying a new object, and pasting it in the selected update. Deleting objects is like copying some scope, and pasting them in *nowhere*, i.e. deleting them. When a source has generated several objects, and only a subset of them has been selected for deletion, there is an ambiguity as deleting these objects requires deleting the source. In such a case we take the conservative option not to delete these objects. Only when all objects from a source are selected this source is deleted (of course the corresponding file is not modified in any way, but only its handler in Camelis).

Updates can also be applied on sources through the source dialog (command `File > Import > List sources...`). Indeed sources are equipped with an initial description, which can be updated through the button `Paste` in the source dialog. Sources can also be directly deleted in this dialog through the buttons `Delete` and `Clear`.

Updates can also be applied on queries, where queries can be seen as dynamic sets of objects. In this case the selected update is applied to the extent of the query, and then a rule *query* $\rightarrow$ *update* is registered in order to be applied whenever an object is created or moved into the extent of the query. The query is selected by the command `Logic > Copy query`, also available contextually on the query area, and the update is defined by applying `Paste` or `Paste not` on selected features or another query. When applying the command `Logic > Delete query`, also available contextually, this creates a kind of *hole* as any object that comes into it is immediately deleted. A possible application of this is the filtering of spams in an email box. The list of all rules can be displayed in a dialog window by the command `Logic > List rules...`. On each line one can visualize the kind of the rule (`paste` or `delete`), the selected update, and the query. The available actions are:

- `Refresh`: refresh the list of rules,
- `Delete`: deletes the rule (no retroactive effect),
- `Close`: close the rule list window.

7 Acting

It is possible to define new actions for making external applications available from a context. These applications need to receive some parameters that are specific to each object. So, source (resp. well) modules now produce these parameters, called *action arguments*, for each object (resp. for each well). The

action arguments of an object are accessible from the contextual menu of the extent window. An action argument is made of three parts:

- *mime*: a kind of MIME type of the parameter (e.g., an audio file name, a URL, a pattern),
- *role*: the role played by the parameter w.r.t. the object (e.g., file, file list, PDF version),
- *value*: the value of the parameter (e.g., a file name, a URL).

An object can be given any number of parameters, and several parameters can have the same *mime* so as to specify different values for a type of parameters.

New actions can be defined through the command `Actions > Add action...`. An action is defined by 3 things:

- *Name of the action*: the text that will appear to the user,
- *Command to be executed*: the command that will be executed if the action is selected,
- *The query defining the scope*: enables to restrict the objects to which this action can be applied.

Both the name and the command can contain mimes as variables, which select relevant parameters for the action. For instance a audio player will use a mime variable `audio` for instance. A mime can be made more precise like in `audio/mp3`. Mimes must be parenthesized and prefixed by the character '$' for locating and replacing them by parameter roles in the action name, and parameter values in the action command.

Every mime variable occuring in the name should also occur in the command. The mime of an object parameter can be an extension of a mime variable: e.g., `$(audio)` matches `audio/mp3`. An action can be instantiated several times for a same object if several roles/values are available for the mime variable. Finally on Windows/Cygwin, if an action uses a Windows application, then any file path must be converted. This conversion is simply specified by adding `:win` after a mime variable, like in `$(image/jpeg:win)`.

The list of all actions can be displayed by the command `Actions > List actions...`, and offers the following commands as buttons:

- `Refresh`: refresh the list of actions,
- `Add`: open a dialog for defining a new action (using the selected action as default values),
- `Edit`: open a dialog for editing the selected action,
- `Delete`: delete the selected action,
- `Close`: close the action list window.

When selecting a set of objects (by default, the full extent), the contextual menu of the extent window contains all possible instanciations of actions according to the action arguments of the selected objects. Selecting an action executes the associated command. When a same action name applies to several

objects, the number of these objects appears between parentheses. This means the action will be exexuted many times, and requires confirmation from the user (as this may be dangerous to launch many times an application). If an export as a well is possible, actions are also given according to the action arguments of the would-be well. If such an action is selected, then the well is produced in a temporary file, and the action command is executed on this file.

8 Persistency

Persistency is already possible by exporting and importing a context as a `.ctx` file, as presented in Section 5.3. However importing a context can be time-consuming as all internal data structures have to be recomputed. For this reason another format is used by commands in the file menu (`New...`, `Open...`, `Save`, `Save as...`, `Close`): binary files with extent `.lis`. They allow opening a context as saved in a previous section in a short time. When saving a context both `.lis` and `.ctx` are produced. Indeed the `.ctx` version is more robust to changes w.r.t. the internal structures of Camelis and the chosen application. When opening a `.lis` file from a former version, Camelis will suggest to regenerate the context from the `.ctx` file.

When launching Camelis applications, some options and arguments can be given:

- The option `-readonly` controls whether the user has readonly access on opened contexts. Without this option, when opening a context, either it has not been opened by another user then a lock is put on it and the user has write access, or it has already been opened (there is a lock) and the user has readonly access on this context. In the latter case, the user can still save the context in another file, because there is one lock per context.
- A single `.lis` file can be given as the initial context. Otherwise the context is initially empty.
- Several `.ctx` files can be given as contexts to be imported in the initial context. They are imported in the specified order.

9 Glis: a general purpose LIS

Glis is an example of application that can be built with Camelis. It is a general purpose application handling various types of files I found useful in my own experience. It can be freely modified to fit your needs by changing the file `examples/glis.ml` and creating new logics, sources or wells. As it is not the purpose of this manual to explain how to define new applications, the following of this section is limited to present logics, sources, and wells available in Glis.

9.1 Logic

The logic of descriptors and features is made of valued attributes of various types. Every atomic formula is made of a non empty sequence of attributes followed by zero, one or several values. An attribute can be either an identifier (beginning with a lowercase letter), or a term (beginning with an uppercase

letter, underscore, or a quoted string). The difference is that axioms can only be stated between terms. When a value is given, it has one of the following types:

string : The value is a double-quoted string prefixed by `is`, `contains`, `beginswith`, `endswith`, `match` (each of these keywords can be abbreviated by its first letter). The string after `match` must be a Unix-like regular expression.

- `is "Rock"`
- `c "logic"`
- `beginswith "Once a time ago"`
- `match "[+-]?[0-9]+"`

integer : The value is either an integer prefixed by `=`, or a possibly open integer interval prefixed by the keyword `in` and where the separator is a double dot. Integers can approximated by using dashes, e.g., `2--` for two hundred something; and multiple dashes can summarized by the standard letters (k, M, G, etc.), e.g., `3-M` for thirties of millions.

- `= 182`
- `= 29-k`
- `in 2000 ..`
- `in 1990 .. 1999`
- `in ..`

date : The value is an interval of dates, where a date can be expressed at three levels of resolution: day, month, or year. Relative dates can be expressed by expressions such as `today`, `next day`, `last year`, `2 months ago`. The syntax w.r.t. intervals is the same as for integers, except there is the leading keyword `date`.

- `date = 5 may 2005`
- `date = tomorrow`
- `date in 2000 .. dec 2005`
- `date in 2 days ago ..` ("since 2 days ago" = "for 2 days")

time : The value is an interval of times, where a time can be expressed at three levels of resolution: hour, minute, or second. Relative times can be expressed by expressions such as `now`, `next hour`, `last minute`, `2 hours ago`. The syntax w.r.t. intervals is the same as for dates, except for the leading keyword that is `time`.

- `time = 13:07:59`
- `time = 1 hour ago`
- `time in 12 .. 13:05`
- `time in .. 13:59`
- `time in 1 hour ago .. now`

file permissions : The value is an interval of permissions, where a permission expresses a set of Unix file access permissions. For instance, `rwx r r` represents read/write/execute permissions for the file owner, and read access for group and other users. The syntax w.r.t. intervals is the same as for dates, except for the leading keyword that is `perm`.

- `perm = rwx rx r`
- `perm in r r r ..` (at least, everybody can read)
- `perm in .. rwx r r` (at most, group and other users can only read)
- `perm in r r r .. rwx r r`

9.2 Sources

Available source types are:

Directories : Directories allow recursive creation, update, and relocation of sources. However deletion is not recursive so that a directory can be deleted as a source without deleting existing sources under it. A directory produces no object by itself.

Files and URLs : every file and URL can be imported, and is then represented by an object described by properties derived from its name (directories, extension, host, etc.), as well as size, last modification date, etc. for local files. All the following sources include and extend this behaviour.

BibTeX files : When the option "extract file parts..." is selected, files with extent `.bib` produce an object per bibliograpic reference, and for each object a descriptor per field. The preview of these objects contains the reference, the authors, the title, the journal or booktitle, and the publication year.

MP3 files : Files with extent `.mp3` produce an object with ID3 tags as descriptors, and song artist and title as preview.

JPEG files : Files with extent `.jpg` or `.jpeg` produce an object with no descriptor, except the file location as for every sources. The preview is a thumbnail of the picture.

Mozilla bookmarks : When the option "extract file parts..." is selected, files with name `bookmarks.html` produces an object per URL, and uses folder names and descriptions as descriptors. The preview of URLs is the description associated to it in the bookmarks file.

Mozilla email folders : When the option "extract file parts..." is selected, files with extent `.msf` produce an object per email with sender, receivers, subject, date, and time as descriptors. The preview shows the subject, sender, and date.

CSV Files : Files with extent `.csv` produce an object for each line, except the first that is used for column names, and each column is made a descriptor. The preview is simply the line.

Java sources : When the option "extract file parts..." is selected, files with extent `.java` produce an object per method with name, class, modifiers, and signature as descriptors.

OCaml interface files : When the option "extract file parts..." is selected, files with extent `.mli` produce an object per value with name, module, and type as descriptors.

DBLP records : URLs starting by `http://dblp.uni-trier.de/rec/bibtex/` produce a BibTeX entry described as above.

DBLP search results : URLs starting by `http://www.informatik.uni-trier.de/` produce a set of DBLP records, which are handled by the previous source. When importing several overlapping search results, duplicates are avoided.

9.3 Wells

Available well types are:

BibTeX files : Builds a new BibTeX file with extent `.bib` from a set of objects produced from BibTeX source files.

Picture slide show : Builds a slide show from a set of picture objects as a text file containing a list of file pathes. This file is designed to be viewed by GQView on Linux (file extent `.gqv`), and Irfanview on Windows (file extent `.txt`).

Music playlist : Builds a playlist from a set of music objects as a text file containing a list of file pathes. The file must have `.m3u` as an extent.

9.4 Actions

A `.ctx` file containing definitions of actions for Linux and Windows/Cygwin is given. It comprises at least a music player for files and playlists, an image viewer, a slide show viewer, a text editor, a web browser, a PDF reader.

References

[Fer02] S. Ferré. *Systèmes d'information logiques : un paradigme logico-contextuel pour interroger, naviguer et apprendre.* Thèse d'université, Université de Rennes 1, October 2002. Accessible en ligne à l'adresse http://www.irisa.fr/bibli/publi/theses/theses02.html.

[FR04] S. Ferré and O. Ridoux. An introduction to logical information systems. *Information Processing & Management*, 40(3):383–419, 2004.

[GW99] B. Ganter and R. Wille. *Formal Concept Analysis — Mathematical Foundations.* Springer, 1999.

[Pad05] Y. Padioleau. *Logic File System, un système de fichier basé sur la logique.* Thèse d'université, Université de Rennes 1, February 2005.

[PR03] Y. Padioleau and O. Ridoux. A logic file system. In *Usenix Annual Technical Conference*, 2003.

[Wil82] R. Wille. *Ordered Sets*, chapter Restructuring lattice theory: an approach based on hierarchies of concepts, pages 445–470. Reidel, 1982.