A File System Based on Concept Analysis

Sébastien Ferré and Olivier Ridoux

IRISA, Campus Universitaire de Beaulieu, 35042 RENNES cedex,
{ferre,ridoux}@irisa.fr

Abstract. We present the design of a file system whose organization is based on Concept Analysis "à la Wille-Ganter". The aim is to combine querying and navigation facilities in one formalism. The file system is supposed to offer a standard interface but the interpretation of common notions like directories is new. The contents of a file system is interpreted as a Formal Context, directories as Formal Concepts, and the sub-directory relation as Formal Concepts inclusion. We present an organization that allows for an efficient implementation of such a Conceptual File System.

1 Introduction: Querying vs. Navigation

Information retrieval includes representation, storage, organization, and access to information. Two information retrieval methods are widely adopted and applied.

The first method is *hierarchical classification*, which is frequently found in computer tools: e.g., file systems, bookmarks, or menus. In this model, searches are done by *navigating* in a classification structure that is built and maintained manually. Navigating implies notions of *place*, being in a place, and going in another place. A notion of *neighborhood* helps specifying the "other place" relatively to the place one is currently in. Many applications require that a place is a place to read from as well as a place to write on. Sometimes, navigation is deemed to be rigid, but this is because it is based on a rigid neighborhood relation: e.g., a tree-like hierarchy. But even a tree can be made more flexible by adding links: e.g., a UNIX-like file system. However, using links opens the door to the problem of dangling links.

The second method is *boolean querying*, often found in information servers such as search engines on the Web (e.g., AltaVista). In this model, searches are done with *queries*, generally expressed in a kind of propositional logic. A recognized difficulty of this model is the necessity of having a good knowledge of the terminology used in the information system, and of having a precise idea of what is searched for.

Then, which search model should be prefered: navigation or querying? In fact, it depends on situations, and it is sometimes needed to use both of them in the same search. Hybrid systems combining hierarchical classification and boolean querying have been proposed in the domain of file systems (FS): e.g.,

— SFS (Semantic File System, [GJSO91]) extends the hierarchical model of usual FSs with virtual directories that correspond to queries. These queries concern file properties that are automatically extracted by *transducers*, and are expressed with valued attributes. So, two organization and storage methods coexist: the standard hierarchy that gives a name to files and virtual directories that enable associative searches on intrinsic file properties. Unfortunately, these two methods cannot be used together in general. In particular, virtual directories are not places to write into.
— HAC (Hierarchy And Content, [GM99]) also uses queries to build directories based on file contents, but these directories are integrated in the hierarchy. This enables to combine hierarchy and content in searches. Users are always allowed to move a file in a directory even if it does not satisfy the query associated to the directory, which results in consistency problems.

The drawback of these hybrid systems is their lack of consistency. Indeed, they have two search models that are not tightly connected, which makes it difficult to switch from one model to the other, and to combine both in the same search. We propose a scheme in which queries are really places to read from and to write into. The scheme is flexible in the sense that the neighborhood relation can be very dense. It incurs no inconsistency or dangling links problem, because the neighborhood relation is managed automatically. Finally, it supports both querying and navigation, and arbitrary combinations of both.

2 Logical Concept Analysis

Formal Concept Analysis (FCA [GW99]) has received attention for its application in many domains such as representing the modular structure of software [Sne98], navigating in software documentation [Lin95], software engineering [KS94], and several applications in Social Sciences. The interest of FCA as a navigation tool in general has also been recognized [GMA93].

Originally, FCA is elaborated using a *Formal Context* that is any relation between a set of objects and a set of attributes. The variety of application domains brings the need for more sophisticated formal contexts than the mere presence/absence of attributes. For instance, many application domains use numerical values (e.g., lengths, prices, ages), and the need to express negation and disjunction is often felt. Several enrichments to the attribute structure have been proposed: e.g., many valued attributes [GW99], and first-order terms [CM98]. However, not a single extended FCA framework covers all the concrete domains, and can pretend covering all the concrete domains to come. We use an extension of FCA that allows for fully abstracting from the object description language [FR99,FR00]. In the rest of this article, we will refer to both the original form of concept analysis, and to its extended form as Logical Concept Analysis (LCA).

2.1 Logical Context and Galois Connection

Definition 1 (context) *A* logical formal context *is a triple* $(\mathcal{O}, \mathcal{L}, i)$ *where:*

- *$\mathcal{O}$ is a finite set of objects,*
- *$\langle \mathcal{L}; \dot{\models} \rangle$ is a lattice of formulas, whose supremum is $\dot{\vee}$, and whose infimum is $\dot{\wedge}$; $\mathcal{L}$ denotes a logic whose deduction relation is $\dot{\models}$, and whose disjunctive and conjunctive operations are respectively $\dot{\vee}$ and $\dot{\wedge}$,*
- *i is a mapping from $\mathcal{O}$ to $\mathcal{L}$ that associates to each object a formula that describes it.*

If $f \dot{\models} g$ and $g \dot{\models} f$, f and g are called logically equivalent; we will consider them as different representations of the same equivalence class, and in fact we will consider that elements of $\mathcal{L}$ are the equivalence classes. Given a formal context, one can form a Galois connection between sets of objects (extents) and formulas (intents) with two applications σ and τ.

Definition 2 *Let $(\mathcal{O}, \mathcal{L}, i)$ be a logical context,*

$$\sigma : 2^{\mathcal{O}} \to \mathcal{L},\ \sigma(O) := \dot{\bigvee}_{o \in O} i(o) \quad \textit{and} \quad \tau : \mathcal{L} \to 2^{\mathcal{O}},\ \tau(f) := \{o \in \mathcal{O} | i(o) \dot{\models} f\}$$

Example. The following formal context, K_{ex}, will illustrate the rest of our development on LCA and Conceptual File Systems. It is deliberately small and simple as it is aimed at illustrating theoretic notions. It uses propositional logic. We define context K_{ex} by $(\mathcal{O}_{ex}, \mathcal{P}, i_{ex})$, where $\mathcal{O}_{ex} = \{x, y, z\}$, and where mapping i_{ex} is defined as $\{(x \mapsto a), (y \mapsto b), (z \mapsto c \wedge (a \vee b))\}$.

2.2 Logical Concepts

In this section, we present how formal concepts can be extracted from logical contexts.

Definition 3 (concept) *In a context $(\mathcal{O}, \mathcal{L}, i)$, a* concept *is a pair $c = (O, f)$ where $O \subseteq \mathcal{O}$, and $f \in \mathcal{L}$, such that $\sigma(O) \dot{=} f$ and $\tau(f) = O$.*

The set of objects O is the concept extent *(ext(c)), whereas formula f is its* intent *(int(c)).*

The set of all concepts that can be built in a context $(\mathcal{O}, \mathcal{L}, i)$ is denoted by $\mathcal{C}(\mathcal{O}, \mathcal{L}, i)$, and is partially ordered by $\leq^c$ defined as follows.

Definition 4 $(O_1, f_1) \leq^c (O_2, f_2) \Longleftrightarrow O_1 \subseteq O_2$

This order is compatible with order on intents.

Proposition 1 $(O_1, f_1) \leq^c (O_2, f_2) \Longleftrightarrow (f_1 \dot{\models} f_2)$

Definitions 3 and 4 lead to the following *fundamental theorem.*

Theorem 1 *Let $(\mathcal{O}, \mathcal{L}, i)$ be a context, and let J be a set of indices. The ordered set $\langle \mathcal{C}(\mathcal{O}, \mathcal{L}, i); \leq^c \rangle$ is a finite lattice, whose supremum (least upper bound) and infimum (greatest lower bound) operations are as follows:*

$$\bigvee_{j \in J}^{c} (O_j, f_j) =^c (\tau(\sigma(\bigcup_{j \in J} O_j)), \dot{\bigvee}_{j \in J} f_j) \textit{ and } \bigwedge_{j \in J}^{c} (O_j, f_j) =^c (\bigcap_{j \in J} O_j, \sigma(\tau(\dot{\bigwedge}_{j \in J} f_j)))$$

Example. Figure 1.(a) represents the Hasse diagram of the concept lattice of context K_{ex} (introduced in Example 2.1). Concepts are represented by a number and a box containing their extent on the left, and their intent on the right. The higher concepts are placed in the diagram the greater they are for order $\leq^c$. It can be observed that the concept lattice is not isomorphic to the power-set lattice of objects $\langle 2^{\mathcal{O}}; \subseteq \rangle$. E.g., the set of objects $\{x, y\}$ is not the extent of a concept, because $\tau(\sigma(\{x, y\})) = \tau(a \vee b) = \{x, y, z\}$.

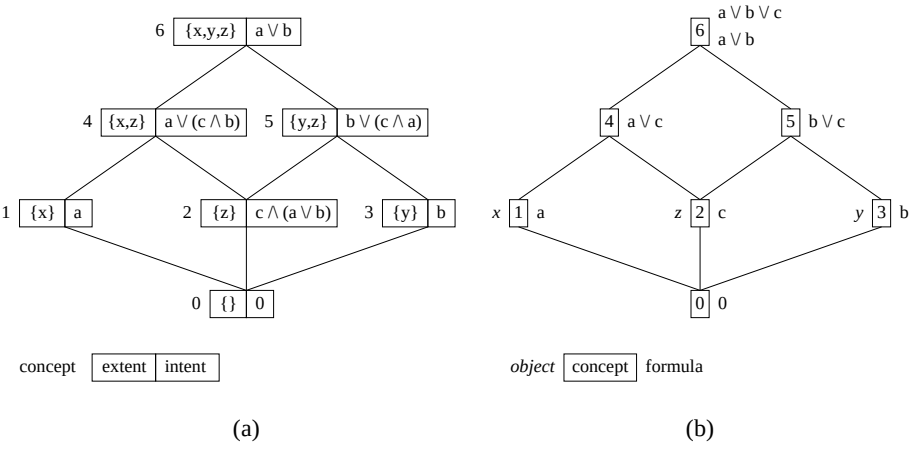

Fig. 1. The concept lattice of context K_{ex}, and its labelling.

2.3 Labelling of Concept Lattices

It is possible to label concept lattices with objects and formulas, but in the information system context we are mainly interested in labelling concepts with formulas; formulas are a means for retrieving objects.

Definition 5 *Let* $\mu : \mathcal{L} \to \mathcal{C}(\mathcal{O}, \mathcal{L}, i), \quad \mu(f) := (\tau(f), \sigma(\tau(f)))$

Images of mapping μ are indeed concepts from Definition 3 of concepts, and from properties of applications σ and τ. The next lemma gives interesting properties of the labelling of concept lattices.

Lemma 1 *Let* $(\mathcal{O}, \mathcal{L}, i)$ *be a context, and* $o \in \mathcal{O}$, $f \in \mathcal{L}$, $c \in \mathcal{C}(\mathcal{O}, \mathcal{L}, i)$.
(1) $c \leq^c \mu(f) \iff int(c) \dot{\models} f$ *(2)* μ *is surjective*
(3) $\mu(int(c)) =^c c$ *(4)* $int(\mu(f)) \dot{\models} f$.

Lemma 1.(1) shows that $\mu(f)$ is the greatest concept whose intent logically entails f; and Lemma 1.(2) establishes that every concept is labelled at least by one formula. Regarding relation between concept intents and concept labels, Lemma 1.(3) shows that every concept is labelled by its intent; and Lemma 1.(4) adds that every formula labelling a concept is logically entailed by the concept intent.

Example. Figure 1.(b) represents the same concept lattice as Figure 1.(a), but it does not associate the same information to concepts. The number of each concept is reused in its box so as to identify it; objects are placed on the left of their labelled concept, and formulas are placed on the right of the concept they label. For instance, concept 1 is labelled by formula a (i.e., $\mu(a) =^c 1$). In this example, we restrict labelling to disjunctive formulas, but every formula labels some context. Note that non-equivalent formulas may label identical concepts: e.g., both formulas $a \vee b$ and $a \vee b \vee c$ label concept 6. Similarly, both formulas c and $c \wedge (a \vee b)$ label concept 2 whose extent is z. This shows that an object can be designated by a formula that is much simpler than its description in the formal context.

Concept lattices support both search models. For navigation, concepts are considered as directories or classes (extent of the concept), and links are realized by generalization/specialization relations between concepts. For querying, concepts are designated/accessed by a query using the labelling function μ, and their extents form the answer. Because concepts serve at the same time as directories and as queries, it is possible to combine both search models in a flexible way. Another advantage is that concept lattices can be automatically built from a context. Therefore, there is no need for a manual maintenance of the information systems. Then, concept lattices appear to be an interesting alternative to usual methods for information retrieving.

3 Informal Specification of a Conceptual Shell

We present a Conceptual File System (CFS) through the use of a Conceptual Shell (CS). Data constitute a formal context $K = (\mathcal{O}, \mathcal{L}, i)$, where i is the mapping that associates to every object a logical description of its properties and features. CS commands are those of the UNIX shell, reinterpreted in the LCA framework as follows: *files* become *objects*, *paths* become *logical formulas*, *directories* become *concepts* or *contextualized formulas*, the *root* becomes concept $\top^c$ or formula $\dot{\top}$, and the *working directory* becomes a *working concept*. For the rest, commands have essentially the same effects as in a classical shell.

If the Conceptual Shell is instantiated with a logic whose formulas are expressed as conjunctions of names and if the conjunction operator is noted `/`, then formulas can be typed in exactly like classical paths. This Conceptual Shell could be used in the same way as a classical shell but the user would notice that the ordering of names in paths does not matter, that he/she could access a file without giving its whole path, and that answers of `ls` command are larger than expected (offering more navigation paths).

Before specifying CS commands, we introduce terms used in the sequel. Let f be a formula:
— the *concept of* f is the concept associated to f by the labelling mapping μ (see Definition 5), i.e., $\mu(f)$,
— the *extent of* f is in fact the extent of the concept of f; it is also the set of objects whose description satisfies f,

— the *object of* f is the object, if it exists, whose description is *contextually* equivalent to f (it is supposed to be unique to avoid access ambiguities); f serves as a *contextualized description* of this object; an object can thus have several contextualized descriptions (several access paths in a way), and its description is always the most precise of them. Concretely, it is possible to access an object using a formula that is simpler than the actual description of the object in the Formal Context. This can be viewed as implicit completion based on the context.

The following commands are used for navigating and querying in a formal context.

working directory: It is replaced by a stack of formulas corresponding to the *navigation history*. The top of this stack serves as the *working query*, which is noted wq.

`pwd`: Displays the working query wq.

`cd ..`: Pops the navigation history, unless it contains only one element. This command enables to go back in navigation, and replaces the move to the parent directory that is no more possible as the navigation structure is not a tree (this command is in fact similar to command "Back" in Web browsers).

`cd` *to*: Let us note $l(to)$ the combination (essentially a conjunction) of to with wq. This command pushes $l(to)$ in the navigation history, and "moves" into the concept of $l(to)$. The composition l leaves open the possibility of a notation that distinguishes *relative* formulas, which are combined with wq (e.g., $wq := wq \dot{\wedge} to$), and *absolute* formulas, which are not combined with wq (e.g., $wq := to$). This implements the "going into some place" part of navigation.

`ls` f: First, displays the object of $l(f)$, if one exists, and second, displays a list of formulas (preferably simple ones), called *increments*, that enable to refine query $l(f)$ while avoiding to lead to the empty query $\perp^c$, and ensuring that every element of the extent of $l(f)$ is reachable only using increments given by command `ls` (completeness condition). This implements the "looking into a place" part of navigation.

`ls -r` f: Displays each element of the extent of $l(f)$. This implements querying.

The following commands are used for updating a formal context. They can be used anywhere in conjunction with querying or navigating.

`mkfile` f c: Creates a new object with contents c and with description $l(f)$.

`rm` f: Remove the object of $l(f)$, if it exists.

`rm -r` f: Remove all elements of the extent of $l(f)$.

`cp` $from$ to: Copies the object of $l(from)$, if it exists, by copying its contents and by "transposing" its description from $l(from)$ to $l(to)$, i.e., by "substracting" $l(from)$ and then by "adding" $l(to)$ (see Section 4.1 for a discussion on the exact meaning of "adding" and "substracting").

`cp -r` $from$ to: Copies each element of the extent of $l(from)$ by copying its contents and by transposing its description from $l(from)$ to $l(to)$.

`mv` $from$ to: Move the object of $l(from)$, if it exists, by transposing its description from $l(from)$ to $l(to)$ (identity and contents are kept unchanged).

`mv -r` *from to*: Move each element of the extent of $l(from)$ by transposing its description from $l(from)$ to $l(to)$ (identity and contents of objects are kept unchanged).

Command `cd` is simple and is essentially useful for handling the working query. Command `ls` deserves more explications. With option `-r` the result is the one that querying systems give to a query: the list of all objects that satisfy the query. Without this option, command `ls` enables navigation, i.e., searching of objects by successive refinements of the working query: an increment x enables to refine the working query wq by $wq\dot{\wedge}x$.

But a badly chosen refinement $wq\dot{\wedge}x$ could return as many answers as the previous query (not enough refinement) or no answer at all (too much refinement). So as to avoid these extremes, we must impose the following condition on every increment x for a given working query wq (let us recall that $\tau(f)$ denotes the extent of query f): $\emptyset \subsetneq \tau(wq\dot{\wedge}x) \subsetneq \tau(wq)$. It can be proved that this condition is equivalent to $\bot^c <^c \mu(wq) \wedge^c \mu(x) <^c \mu(wq)$.

This informal specification shows the relevance of concept lattices to characterize navigation. Moreover, it shows the necessity of these concept lattices because the above condition is not expressible only within logic $\mathcal{L}$. The difficulty is then to find a finite set of increments Inc that satisfies this condition and that is *complete*; i.e., such that for every working query wq and for every object o of the working extent, it exists an increment x in Inc that strictly restricts the working extent while keeping o in the new one. This completeness condition ensures that every object is reachable, only using increments to refine queries. As the conceptual navigation is similar to the classical one (a finite set of increments is used in both cases, increments being formulas in the first case and names in the second case), many facilities offered by classical shells can be applied to our conceptual shell: e.g., name completion, graphical user interface.

In commands resulting in a modification of the context (`mkfile`, `rm`, `cp`, `mv`) the concept lattice is implicitly updated (i.e., formal concepts are used as places one can write into). However, the principle of CS is to provide access to objects by their properties and not through a fixed organization structure, whatever it is. The concept lattice makes no exception: it is interesting for designing and implementing CS, but users need neither know it, nor visualize it explicitly.

4 Formal Specification of a Conceptual Shell

In Section 3, a Conceptual Shell (CS) was presented through its general principles, but was not defined in a precise way. This section aims at giving commands of CS a formal definition based on logic and concept analysis. We begin by describing and defining a set of elementary operations with which CS commands are eventually defined.

4.1 Logical Operations

In CS, objects are described by formulas taken in a logic $\mathcal{L}$. The same formulas are used to express queries (in place of paths). The main logical operation is to

compare two formulas in order to know if an object description satisfies a query, i.e., if the object is an answer to the query. The operation that achieves this comparison is the deduction relation $\dot{\models}$. It establishes a specialization/generalization order on formulas.

In the presentation of CS (cf. Section 3), we talked about "adding" or "substracting" a formula to an object description (in commands `cp` and `mv`). "Adding" a formula f to an object description $i(o)$ must be understood as making this object satisfies this formula while keeping its previous properties. Formally, this means that, if we note the "adding" operation by $+$, the following condition has to be true: $i(o) + f \dot{\models} i(o)$ and $i(o) + f \dot{\models} f$. The most general formula that satisfies both $i(o)$ and f is $i(o) \dot{\wedge} f$. Then, the "adding" operation is well matched by the conjunction operation $\dot{\wedge}$ of logic $\mathcal{L}$.

"Substracting" a formula f to an object description $i(o)$ consists in generalizing it, which can be understood as the removal of some properties of the object. Formally, if we note the "substracting" operation by $-$, we have $i(o) \dot{\models} i(o) - f$. Furthermore, "substraction" is combined with "addition" to "transpose" an object from a concept to another: $i'(o) := (i(o) - from) + to$, where $from$ denotes the source of the transposition, and to denotes the target. When such an operation is done on an object, we know from the meaning of CS commands that $i(o) \dot{\models} from$ (i.e., o is an answer to query $from$). So, in the case where source and target of the transposition are both denoted by formula f, the object description must be kept unchanged, which is formally expressed by $i(o) \dot{\models} f \Longrightarrow (i(o) - f) + f \doteq i(o)$. Relative complementation is a logical operation that satisfies the two conditions above, and is then a good realization of the "substracting" operation. Relative complement is a weak form of implication, and is defined as follows.

Definition 1. *The* relative complement *is an internal binary operation on $\mathcal{L}$ that is noted $\dot{\Leftarrow}$, and is defined for all $f, g \in \mathcal{L}$ by $f \dot{\Leftarrow} g \doteq max\{x \in \mathcal{L} | x \dot{\wedge} g \dot{\models} f\}$.*

As max denotes the greatest element of its argument, not every logic is equiped with a relative complement. Yet, as soon as a logic is equiped with an implication (e.g., the propositional logic), it is equiped with a relative complement, as the later is a weak form of the former.

To summarize, a logic has to be equiped at least with three operations so as to be used in the frame of CS: deduction relation $\dot{\models}$, and conjunction $\dot{\wedge}$. Relative complement $\dot{\Leftarrow}$ is necessary for moving and copying objects by "transposition".

4.2 Context Operations

A formal context stores a set of objects $\mathcal{O}$ and their descriptions. We consider that every object o has a contents $c(o)$, and an *extrinsic description* $i_e(o)$ expressed in a logic $\mathcal{L}_e$. From this basic information, we now have to build the mapping i that describes each object. LCA is based on this mapping. First, intrinsic descriptions of objects $i_i(o)$, expressed in a logic $\mathcal{L}_i$, are automatically extracted from the contents of objects through an abstraction function α:

$i_i(o) := \alpha(c(o))$. Then, the whole description of an object is the mere product of its extrinsic and intrinsic decriptions: $i(o) := (i_e(o), i_i(o))$. The logic $\mathcal{L}$ used in LCA framework is therefore the product logic of $\mathcal{L}_e$ and $\mathcal{L}_i$. More precisely, $\mathcal{L}$ is defined as follows:

$$\mathcal{L} := \mathcal{L}_e \times \mathcal{L}_i \qquad (f_e, f_i) \dot{\models} (g_e, g_i) :\Leftrightarrow f_e \dot{\models}_e g_e \wedge f_i \dot{\models}_i g_i$$
$$(f_e, f_i)\, op\, (g_e, g_i) := (f_e\, op_e\, g_e, f_i\, op_i\, g_i), \text{ for } op \text{ in } \{\dot{\vee}, \dot{\wedge}, \dot{\Leftarrow}\}.$$

Moreover, we have the elementary operations *new_o* and *del_o* that respectively return a new object and remove a given object, and *extr* that returns the extrinsic part of a formula.

4.3 Querying and Navigation Operations

Section 3 introduced notions of *extent of a query* and *object of a query*. They correspond to the elementary operations performing querying. The extent of a query q is noted $\tau(q)$ and returns all objects whose description satisfies q. Operation τ is the same as in the Galois connection of LCA (cf. Definition 2).

The object of a query q is noted $t(q)$ and is the object whose description is contextually equivalent to q (i.e., has the same extent):

$$t(q) = o \in \mathcal{O}, \text{ such that } \tau(i(o)) = \tau(q).$$

If there is no such object the query is said *empty* ($\tau(q) = \emptyset$), and if there are several such objects the query is said *ambiguous*.

Navigation is performed by command `ls`. It returns a set of *increments*, which enable to refine the working query while keeping it non-empty. As already seen in previous section, an increment x of a query q has to satisfy

$$\emptyset \subsetneq \tau(q \dot{\wedge} x) \subsetneq \tau(q).$$

As $\mathcal{L}$ is a too wide search space, we consider a finite subset X of $\mathcal{L}$ in which increments are selected. The content of X is not strictly determined but it should contain simple formulas (according to the terminology), some often used formulas, and more generally, all formulas that users expect to see in `ls` responses. X can be finite because terminology and used formulas are. Furthermore, we keep only greatest increments as they correspond to smallest refinement steps. Then, we can now define the set of increments of a query q by

$$Inc(q) := \lceil \{x \in X | \emptyset \subsetneq \tau(q \dot{\wedge} x) \subsetneq \tau(q)\} \rceil,$$

where $\lceil E \rceil$ denotes the set of greatest elements of E according to the order $\dot{\models}$.

Because of these seemingly arbitrary selections among possible increments, we can wonder about the completeness of $Inc(q)$. If navigation is seen as a way for finding an object by refining the working query with a sequence of increments, the completeness can be formally expressed by

$$\forall q \in \mathcal{L} : \forall o \in \tau(q) : o \neq t(q) \Rightarrow \exists x \in Inc(q) : o \in \tau(q \dot{\wedge} x),$$

which means in English: for all working query q and for all object o of its extent, if o is not yet the object of q then it exists an increment that enables to strictly restrict the working extent while keeping o in it. As extents are finite and each increment strictly restricts the working extent, it follows that every navigation terminates and every object can be retrieved through navigation alone (i.e., querying is useful, but not necessary).

We proved that Inc is complete for all contexts if and only if every object description is equivalent to the conjunction of some elements of X.

This characterization leaves some flexibility in the choice of X, because it can be made larger than necessary. This flexibility enables to adjust X in order to make the navigation more progressive and natural.

4.4 Interface Operations

A few elementary operations are defined here to specify interface aspects of CS. For managing the history, we use a stack of queries initialized with one element, the root query $\dot{\top}$, and equiped with three operations: *push*, *pop*, and *wq* that returns the last pushed query. Beside, we have a function l that combines a formula from a command argument and the history, and a side-effect operation *out* that performs displays.

All elementary operations being defined, CS commands are formally specified in the following table (querying and navigation operations, τ, t, and Inc, are underlined for visibility). Wherever t is used, if it is not defined, the command is aborted and a message warns the user that its query is ambiguous or empty.

CS command	semantics
`pwd`	$out(wq())$
`cd ..`	$pop()$
`cd` *to*	$push(l(to))$
`ls` *f*	$out(\underline{t}(l(f)))$; $out(\underline{Inc}(l(f)))$
`ls -r` *f*	$out(\underline{\tau}(l(f)))$
`mkfile` *f c*	$o := new\_o()$; $c(o) := c$; $i_e(o) := extr(l(f))$
`rm` *f*	$del\_o(\underline{t}(l(f)))$
`rm -r` *f*	forall $o \in \underline{\tau}(l(f))$ do $del\_o(o)$
`mv` *from to*	$o := \underline{t}(l(from))$; $i_e(o) := (i_e(o) \Leftarrow extr(l(from))) \dot{\wedge} extr(l(to))$
`mv -r` *from to*	forall $o \in \underline{\tau}(l(from))$ do $i_e(o) := (i_e(o) \Leftarrow extr(l(from))) \dot{\wedge} extr(l(to))$
`cp` *from to*	$o := \underline{t}(l(from))$; $o' := new\_o()$; $c(o') := c(o)$; $i_e(o') := (i_e(o) \Leftarrow extr(l(from))) \dot{\wedge} extr(l(to))$
`cp -r` *from to*	forall $o \in \underline{\tau}(l(from))$ do { $o' := new\_o()$; $c(o') := c(o)$; $i_e(o') := (i_e(o) \Leftarrow extr(l(from))) \dot{\wedge} extr(l(to))$ }

5 Algorithms and Complexity

In this section, we present a possible design for data structures and algorithms that implement the elementary operations defined in Section 4. We also discuss about the complexity of these algorithms.

5.1 Data Structures

Contents and extrinsic descriptions, which are the basic information of a context, are stored in arrays indexed by object identifiers. Intrinsic descriptions are

extracted from contents by an abstraction function α. As this abstraction can be costly, it is important to cache its results to avoid to redo it at each access (there are well known problems of coherence between contents and intrinsic descriptions, but we do not want to focus on them here). The whole description of an object is then easily composed from its extrinsic and intrinsic ones.

A basic principle in CS is to access objects (and achieve some operations on them) through formulas. Elementary operations τ and t, the most often used ones, consist in finding one or several objects from a formula. There are two extreme solutions for implementing these operations. The first one consists in having no structure at all, and computing these operations, according to their definition, from scratch every time they are used. For $\tau(q)$ (the extent of q), this amounts to computing $i(o) \dot{\models} q$ for every object o of the system, which must be avoided. The second one consists in representing the whole concept lattice. However, even if concept lattices are finite, their size increases with the richness of the logic used in descriptions and the number of objects, and it quickly becomes unacceptable (exponential in the worst case). Happily, such a representation is not necessary.

> The solution we propose consists in representing only a subdiagram $(F, \prec)$ of the Hasse diagram (i.e., a subgraph that is anti-reflexive and anti-transitive) of the (possibly infinite) formula lattice $\langle \mathcal{L}; \dot{\models} \rangle$, where F contains $\dot{\top}$ (the root), $i(\mathcal{O})$ (all object descriptions), and a set X of increments (see elementary operation Inc in Section 4.3).

Why use a diagram of formulas rather than a diagram of concepts? Firstly, what is interesting in concepts is their extents (recall that we aim at identifying objects with formulas), and not their intents which can be too complicated to be exploitable. Nevertheless, any formula that labels a concept by μ is a consistent representation of it. Moreover, the extent of every concept is related to each formula labelling it by the application τ, which can be defined without using the notion of concept ($\forall f \in \mathcal{L} : ext(\mu(f)) = \tau(f)$): so, the diagram of formulas is sufficient for retrieving all information relative to extents, and therefore to concepts. Secondly, the diagram of formulas is easier to use than the concept lattice because command parameters are formulas, and not concepts, and easier to maintain because it is stable (it is defined by the logic $\mathcal{L}$, and more precisely by $\dot{\models}$), whereas the concept lattice evolves according to the context (i.e., every time an object is created, updated or removed). Why choose a subdiagram that is anti-reflexive and anti-transitive? This avoids redundancies and lighten the structure. Furthermore, relation $\dot{\models}$ can easily be retrieved from $\prec$ (by reflexive and transitive closure). Why F must contain $\dot{\top}$, $i(\mathcal{O})$, and X? $\dot{\top}$ is the root of the diagram and is used as an entry point for diagram traversals. Diagram nodes that are labelled by object descriptions are used to attach corresponding objects on them. X is used by Inc in command `ls`. Each node in F can be seen as a *view* that records the answers to a query, and $(F, \prec)$ is then a kind of *view hierarchy* that organizes and facilitates access to information.

Example. Assuming objects of context K_{ex} (cf. Example 2.1) have been created in the order x, z, y, Figure 2 draws the diagram of formulas before and after y is created. Formulas represented in these diagrams are object descriptions (labelled by the object) or increments which are parts of these descriptions, seen as conjunctions. Circles gather formulas that have the same extent, i.e, that label the same concept. In other words, each circle matches a concept in Figure 1.(b). Not all concepts are represented in diagrams, which is a good point considering that there can be an exponential number of concepts in some contexts.

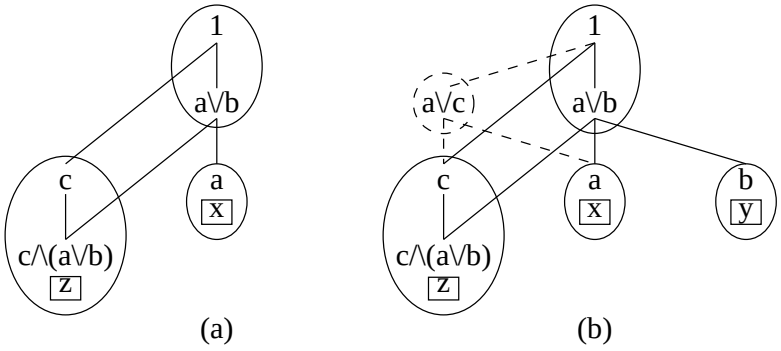

Fig. 2. Hasse diagrams of formulas for K_{ex} with (a) $\mathcal{O} = \{x, z\}$, and (b) $\mathcal{O} = \{x, z, y\}$.

5.2 Algorithms for Operations τ, t, and Inc

We express operations τ, t, and Inc using elementary accesses to the Hasse diagram of formulas: Inf, Inf^*, Sup, Sup^*, obj. The first two are defined for all $f \in F$ by

$$Inf(f) := \{g \in F | g \prec f\} \qquad Inf^*(f) := \{g \in F | g \dot{\models} f\}.$$

Sup and Sup^* are defined dually, and $obj(f)$ is the object that is attached to f (it exists only if f is an object description). From definitions given in Section 4, we get the following equalities for every $f \in F$:

- $\tau(f) = \{o \in \mathcal{O} | \exists g \in Inf^*(f) : obj(g) = o\}$,
- $t(f) = obj(max_{\prec}\{g \in Inf^*(f) | obj(g) \text{ is defined}\})$,
- $Inc(f) = \lceil \{x \in X | 0 < |\tau(f) \cap \tau(x)| < |\tau(f)|\} \rceil$.

These equalities show that algorithms for τ, t, and Inc consist in traversing the set Inf^* of some nodes of the diagram of formulas, while performing simple operations (e.g. collect an object, test and set some marks). These algorithms are independant from $\mathcal{L}$ because logical operations are not used here. More precisely, all useful consequences of the deduction relation, $\dot{\models}$, are cached in the diagram.

Example. In the diagram of Figure 2.(b), the following results can be computed with the above algorithms:

$\tau(a \vee b) = \{x, y, z\}$	$t(a \vee b)$ is undefined	$Inc(a \vee b) = \{a, b, c\}$
$\tau(c) = \{z\}$	$t(c) = z$	$Inc(c) = \{\}$.

We see that $a \vee b$ is an ambiguous query because $t(a \vee b)$ is undefined and $\tau(a \vee b)$ is not empty; whereas c identifies the object z, and has an empty set of increments because $\tau(c)$ is a singleton.

5.3 Algorithms for Locating and Inserting Formulas in the Diagram

Some formulas need to be located and inserted in the diagram of formulas F: descriptions of new objects, new increments, and arguments of τ, t and Inc. Thus, we need an algorithm that takes a formula as an argument, inserts it as a node in F, and returns this node. If a formula f is already in F (modulo $\doteq$), it is not inserted as a new node, but the existing node is returned. Inserting a formula f in F consists in computing $Inf(f)$ and $Sup(f)$. We designed an algorithm *insert* based on traversals of $(F, \prec)$ and comparisons between formulas with $\dot{\models}$ that achieves these computations (by lack of place, we do not detail it here).

Even if we tried to minimize the use of $\dot{\models}$, whose complexity depends on the chosen logic, algorithm *insert* remains costly and we avoid it as often as possible. Firstly, we observe that in practice the working query is built incrementally as a conjunction of increments. Indeed, the usual navigation paradigm is to alternate commands `ls` and `cd`. Even if a query is used instead of an increment given by `ls`, this query can be inserted as a new increment and then conjuncted to the working query. Secondly, algorithms for τ, t and Inc traverse only Inf^* and not Sup^*; and practice shows that they are the most often applied to the working query: in other cases (e.g, `ls -r f`), the command can be decomposed so that it becomes the case (e.g., `cd f; ls -r .; cd ..`).

These observations lead us to introduce a special node wq for representing the working query. This node is special in the sense that only $Inf(wq)$ is defined ($Sup(wq)$ and $obj(wq)$ are not), and the formula of wq is a conjunction of increments (elements of X). The advantage of this is that we have an algorithm $refine$ that refines the working query to $wq \dot{\wedge} x$ from wq and the node of an increment x by traversing $Inf^*(wq)$ and $Inf^*(x)$ without any call to $\dot{\models}$. Moreover, if x is already in F in the same syntactic form, it could be located in F in constant time, using a hashtable for example.

Example. From Figure 2, we can see the effect of adding the object y whose description is b, and the effect of refining the working query with formula $a \vee c$ that leads to insert $a \vee c$ in the diagram of formulas (in dotted lines on the figure).

5.4 Discussion about Complexity

In this section, we evaluate the complexity of the algorithms presented above according to the number of objects n and the complexity of $\dot{\models}$, which we note $O(\dot{\models})$.

We begin by stating some reasonable assumptions. First, we assume that each object description is the conjunction of a set of formulas whose elements belong to the set of increments X, and whose average size is a constant k. This assumption is rather natural and easily satisfied. Second, we assume that F is somewhat homogeneous, i.e., every increment x has a number of subformulas in F proportionnal to the size of its extent: i.e., the ratio $\frac{|Inf^*(x)|}{|\tau(x)|}$ does not depend on x.

From these assumptions, it comes that the average complexity of τ, t and $refine$ is $k(1 + \frac{n}{|X|})$, and that the average complexity of Inc is $k(n + |X|)$, where $|X|$ is the number of increments.

> A good compromise is to give X a size proportional to n. So doing, the average complexity of τ, t and $refine$ is constant, and the one of Inc is linear in n.

For algorithm $insert$, the worst case complexity is $nO(\dot{\models})$, which is unavoidable in some situations: for instance, the insertion of a new increment x, that is incomparable to existing ones, leads to compare x with all object descriptions.

> The best average complexity of $insert$, theoretically speaking, is $(\ln n)O(\dot{\models})$. This optimal complexity is realized under the conditions that, in the diagram of formulas, the height (the greatest path length) is in $(\ln n)$, and the size of Inf is bound.

To put a concrete form on these conditions, let us observe that if F is structured as a tree, it satisfies all conditions we put on the diagram of formulas.

To conclude, although the structure and the management of the diagram of formulas must be further specified to precise how to satisfy the above conditions, we already know that it is possible to implement the CS at a reasonable cost. Moreover, this is valid for any decidable logic and without any restriction to the model presented in section 3.

6 Conclusion and further works

We have presented the design of a file system shell in which the designation of objects is made by using formulas via Concept Analysis. In this design, conventional notions such as files and directories are matched by the notions of objects and concepts. The reference to an arbitrary logic may seem to challenge the practicalness of the design, but we have shown that the actual usage of a theorem prover for taking into account the logic is very limited. In fact, deductions can be "cached" in a partially ordered diagram of formulas that does no depend on the formal context. Thus, it needs not be updated when the Formal Context changes. Explicitely managing an evolving Concept Lattice is costly, and we have shown how to avoid it. Only an approximation of the Concept Lattice is actually represented.

This design has been implemented as a high-level generic Conceptual Shell prototype (CS) in which a theorem prover can be plugged in, and as a lower

level File System prototype (RFS for relational file system) in which a simple logic (a logic of sets of attributes) is wired in a file system. The prototypes have been tested with different applications: a Vietnamese cookbook (e.g., `cd fish_sauce/pineapple`) as an example of a small-size consumer oriented application, the management of home directories as an example of a medium-size professional application (we used CS with up to 5000 objects), and a personal organizer.

Further works on the short and medium term is to develop the algorithmic aspects of CFS, improve the navigation facilities, and develop a graphical user interface. Long term further works is to implement it at the level of a file system. This is necessary because not all accesses to objects are done via a shell; many more are done via system commands. So, one must offer the CFS service at this level.

References

[CM98] L. Chaudron and N. Maille. 1st order logic formal concept analysis: from logic programming to theory. *Computer and Information Science*, 13(3), 1998.

[FR99] S. Ferré and O. Ridoux. Une généralisation logique de l'analyse de concepts logique. Technical Report RR-3820, Inria, Institut National de Recherche en Informatique et en Automatique, December 1999. An english version is available at http://www.irisa.fr/lande/ferre.

[FR00] Sébastien Ferré and Olivier Ridoux. A logical generalization of formal concept analysis. In Guy Mineau and Bernhard Ganter, editors, *International Conference on Conceptual Structures*, August 2000. To appear.

[GJSO91] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. Jr O'Toole. Semantic file systems. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 16–25. ACM SIGOPS, October 1991.

[GM99] Burra Gopal and Udi Manber. Integrating content-based access mechanisms with hierarchical file systems. In *Proceedings of third symposium on Operating Systems Design and Implementation*, pages 265–278. USENIX Association, 1999.

[GMA93] R. Godin, R. Missaoui, and A. April. Experimental comparison of navigation in a Galois lattice with conventional information retrieval methods. *International Journal of Man-Machine Studies*, 38(5):747–767, 1993.

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

[KS94] M. Krone and G. Snelting. On the inference of configuration structures from source code. In *Proceedings of the 16th International Conference on Software Engineering*, pages 49–58. IEEE Computer Society Press, May 1994.

[Lin95] C. Lindig. Concept-based component retrieval. In *IJCAI95 Workshop on Formal Approaches to the Reuse of Plans, Proofs, and Programs*, 1995.

[Sne98] G. Snelting. Concept analysis — A new framework for program understanding. *ACM SIGPLAN Notices*, 33(7):1–10, July 1998.