Typed Formal Concept Analysis

Ian D. Peake, Ian E. Thomas, Heinz W. Schmidt

School of Computer Science and Information Technology, RMIT University, Australia

Abstract. Formal Concept Analysis (FCA) appears to be ideal for interpreting data in domains that are a-priori unstructured. In our research we focus on complex contexts with very large and rapidly changing numbers of real-world objects – for instance, data generated on-line and tracking real-time contexts in transport, enterprise warehousing, manufacturing control or social networking activities on the web. The ontology of such contexts, i.e., the types of attribute data and those of objects, and their interrelation (such as subtyping or which objects have which attributes or reference which other objects via surrogate attributes) is partly known to domain experts. Because of incomplete knowledge however, missing classes and relationships need to be inferred. In addition, domain experts need assistance in checking the partial ontology declarations against FCA interpretations added to the context.

To make FCA more accessible in such applications, we explore the use of types similar to those available in object-oriented design and programming, for which methods, tools, training and skills are widely available. We present typed FCA and report about supporting tools. We also introduce typed priming and show that it is consistent with conceptual scaling without requiring the generation of binary lattices.

1 Introduction

Formal Concept Analysis (FCA) [16] is a branch of applied lattice theory useful for classifying collections of objects (contexts) in a wide variety of applications. Such classification is based on shared single or multi-valued attributes.

In our research we are interested in semi-structured domains, where domain experts have a-priori knowledge of part but not all of the domain ontology, i.e., the types of attribute values and objects and their interrelations such as subtyping, or restrictions of attributes to certain classes of objects etc. Moreover we aim at dealing with contexts with very large and rapidly changing numbers of real-world objects (see also [6]), such as for instance:

– software-intensive distributed manufacturing and control requiring real-time monitoring and administration;
– large enterprise warehouses with a need to discover conceptual duplication and consolidate data just in time;
– social networking portals supporting hundreds of thousands or millions of users sharing niche interests.
Because of incomplete domain knowledge, classes of objects and of attributes may have to be inferred and may change frequently, while at the same time some such classes are fixed a-priori by the domain experts. In domain modelling, object-oriented type systems [3, 2] are now common with a rich theory for static type checking, type inference [12], and design methods accompanied by powerful interactive prototyping tools, training material and domain-specific skills. Efficient implementation and program optimization relies heavily on concrete type information.

In this contribution to the FCA theory and application, we propose weaving typing deep into the FCA semantics by a new ‘typed FCA’ model. In this model we revisit the fundamental priming operation underlying FCA classification, such that ‘typed priming’ arises without reference to the binary contexts and binary lattices underlying structured attribute types and the contexts they give rise to. We also report about two applications of this model in two separate implementations we have developed in our research projects.

As a running example we introduce the formal context shown in Table 1 whose concept lattice is shown in Figure 2. Capturing, modelling and classifying large, changing collections of complex, real-world objects in the presence of some – albeit incomplete – domain type information presents challenges to the use of FCA. For example, to handle richer object models, many-valued contexts are used, such as the one shown in Table 2, a more natural though untyped representation of the domain of laptops. A many-valued context must be interpreted via conceptual scaling as a single-valued context. Conceptual scaling requires specifying a scale per attribute. For example, based on the scales in Figure 1 for CPU, HD, and the boolean (true/false-valued) attributes, respectively, the many-valued notebooks context denotes a single-valued context virtually.

<table>
<thead>
<tr>
<th>CPU $\geq$ 166</th>
<th>CPU $\geq$ 180</th>
<th>CPU $\geq$ 200</th>
<th>HD $\geq$ 100</th>
<th>HD $\geq$ 120</th>
<th>HD $\geq$ 160</th>
<th>Gbit</th>
<th>Btooth</th>
<th>Pro</th>
</tr>
</thead>
<tbody>
<tr>
<td>r7</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>r8</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>r9</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>s9</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>s15</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>s15p</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>m70</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 1. A formal context: a set of notebook computer models (rows) from a single manufacturer c. 2007 and selected attributes (columns). An “X” indicates that a model has the attribute. Attribute abbreviations: “CPU” = CPU speed (x10MHz), “HD” = Hard disk capacity (GBytes), “Gbit” = Gigabit Ethernet built-in, “Btooth” = Bluetooth capability, “Pro” = “Professional” version of operating system pre-installed.
identical to the earlier single-valued notebooks context\(^1\) and likewise an almost identical concept lattice.

<table>
<thead>
<tr>
<th>CPU (x10MHz)</th>
<th>HD (GByte)</th>
<th>GBit</th>
<th>Btooth</th>
<th>Pro</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>8</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>166</td>
<td>100</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>166</td>
<td>100</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>180</td>
<td>120</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>180</td>
<td>120</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>200</td>
<td>160</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>200</td>
<td>160</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>2500</td>
<td>1000</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 2. A many-valued context of notebook computer models.

Conceptual scaling involves a design process trading precision against simplicity while determining and highlighting relevant distinctions. Our motivation for typed FCA is to enable domain experts to express attribute type discriminations in the expressive language of object-oriented types, and to reuse such definitions directly where available. FCA then can add further distinctions to, or discover inconsistencies in types hidden in the context data and/or elicited by the domain expert.

The contributions of this paper are as follows. First we propose a formal theory of \textit{typed contexts}, which shows how FCA can be recast in terms of object-oriented type theory\(^3\), enabling users to directly but abstractly capture prior knowledge or expectations about contexts in terms of the interpretation of attributes. We then show how this theory enables two extensions, and show their experimental implementations.

This paper is organised as follows. Section 2 gives mathematical background sufficient to make the paper self-contained. Section 3 describe the core theory

\(^1\) In the derived context attributes are pairs e.g. “(CPU, \(\geq 166\))”
of typed FCA as an extension of many-valued contexts. Section 4 describes an
extension for a direct, unfolding-free interpretation, and Section 5 describes the
implementation in Scala. Section 6 describes the extension and implementation
in Java with an unfolding semantics. These are followed by related work (Section
7) and conclusions (Section 8).

2 Preliminaries

This section briefly reviews key definitions of FCA the remainder of the paper
relies on. The reader is referred to Ganter and Wille [4, 5] for a more complete
foundation.

A formal context \( C = (G, M, I) \) is a triple of sets, where \( G \) is called a set
of objects, \( M \) is called a set of attributes and \( I \) is a binary relation \( \subseteq G \times M \). If
we say \( gIm \), then object \( g \) has attribute \( m \). Let \( C \) by a formal context \((G, M, I)\).

![Fig. 2. Concept lattice of single-valued notebooks context (reduced labelling)](image)

The priming operators (or derivation operators) of \( C \), \( \uparrow^C : 2^G \to 2^M \) and
\( \downarrow^C : 2^M \to 2^G \) over object and attribute sets, respectively, are defined as follows:

\[
A^{\uparrow^C} = \{ m \in M \mid A \subseteq .Im \}, \text{ for all } A \subseteq G
\]

\[
B^{\downarrow^C} = \{ g \in G \mid B \subseteq gI. \}, \text{ for all } B \subseteq M
\]

where \( .Im = \{ a \in G \mid aIm \} \) and \( gI. = \{ b \in M \mid gIb \} \). If \( C \), and the kinds of
sets \( A \) and \( B \) are unambiguous, alternatively write \( A' \) for \( A^{\uparrow^C} \) and \( B' \) for \( B^{\downarrow^C} \).
A many-valued context $C$ is a tuple $(G, W, M, I)$ where $G$ is a set (of objects), $W$ is a set (of attribute values), $M$ is a set of partial functions $m : G \rightarrow W$ and $I \subseteq G \times M \times W$ such that $m(g) = w$ iff $(g, m, w) \in I$.

A scale for the attribute $m$ of a many-valued context is a (one-valued) context $S_m = (G_m, M_m, I_m)$ with $m(G) \subseteq G_m$. The objects of a scale are called scale values, the attributes are called scale attributes.

For a many-valued context $C$ with scale $S_m = (G_m, M_m, I_m)$ for all $m \in M$, its derived context $D$ is the tuple $(G, N, J)$ where the derived attribute set is $N = \bigcup_{m \in M} \{m\} \times M_m$ and the derived incidence relation $J$ is: $gJ(m, n) :\iff m(g)I_m n$.

A formal concept of context $(G, M, I)$ is a pair $(A, B)$ with $A \subseteq G$, $B \subseteq M, A' = B$ and $B' = A$. Call $A$ the extent and $B$ the intent of the concept $(A, B)$. For concepts $(A_1, B_1)$ and $(A_2, B_2)$, $(A_1, B_1)$ is a subconcept of $(A_2, B_2)$ if $A_1 \subseteq A_2$ and a superconcept of $(A_2, B_2)$ if $A_2 \subseteq A_1$, written $(A_1, B_1) \leq (A_2, B_2)$. The set of all concepts of the context $(G, M, I)$, ordered by $\leq$ is called the concept lattice of $(G, M, I)$.

3 Typed FCA

In the following sections we introduce typed FCA, an FCA extension based on standard type-theoretic principles. We show how typed FCA provides improved usability over many-valued FCA with scaling. In particular, notions of signatures and interpretations encourage the expression of abstract invariants (or “contracts”) for changing contexts which can be exploited to enable precise and efficient classification.

Typed FCA enables a user to describe interpretations for attributes as types, as illustrated in Table 3. Types capture expectations about attribute values while delegating selected details of precise representation and meaning to the system, possibly in a context-dependent way. In tables representing typed contexts, each attribute name is suffixed by a colon (:) and a type name. Typed FCA introduces (syntactic) context signatures mapping many-valued attributes to attribute type names, and a type interpretation mapping attribute type names to concrete attribute types (sets) in some universal type domain (set of sets).

Definition 1. A syntactic context signature $\Sigma$ is a tuple $(M, T, \tau)$ where $M$ is a finite set (of attribute names), $T$ is a finite set (of type names), $\tau$ is a mapping $\tau : M \rightarrow T$, assigning type names to attributes.

Hereafter we will interpret type symbols by concrete types (sets of values). To this end let $U$ be a countable set (called the universe of values) for the rest of this paper.

Definition 2. Given a set of type labels $T$, a (concrete) type interpretation $s$ for $T$ is a total mapping $s : T \rightarrow 2^U$. 
Table 3. A typed context of notebook computer models and selected attributes.

<table>
<thead>
<tr>
<th>CPU</th>
<th>HDD</th>
<th>GBit</th>
<th>Btooth</th>
<th>Pro</th>
</tr>
</thead>
<tbody>
<tr>
<td>CardinalAnti</td>
<td>CardinalAnti</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>r8</td>
<td>166</td>
<td>100</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>r9</td>
<td>166</td>
<td>100</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>r7</td>
<td>180</td>
<td>120</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>r9</td>
<td>180</td>
<td>120</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>s15</td>
<td>200</td>
<td>160</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>s15p</td>
<td>200</td>
<td>160</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>m70</td>
<td>250</td>
<td>1000</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Let $s$ be a type interpretation $s : T \rightarrow 2^U$ for $T$, $1_s = \bigcup_{t \in T} s(t)$, and, $0_s = \bigcap_{t \in T} s(t)$. Now let $L_s = \{ s(t) \mid t \in T \} \cup \{ 1_s, 0_s \}$. Then $(L_s, \subseteq)$ forms a lattice with largest common subset and smallest common superset as meet and join operations, and $0_s$ as infimum and $1_s$ as supremum.

If we let $u \in T$ and $s(u) = 1_s$, then $L_s = T \cup \{ u \}$ forms a complete lattice with $t \leq t' \Leftrightarrow s(t) \subseteq s(t')$. It is convenient and natural to read $\leq$ as a subtype relationship on $T$.

Figure 3(a) shows a type interpretation of the typed notebooks context signature as a function from type names to concrete types (dashed edges to boxes), within the complete lattice of the subtype relation (black arrows).

Fig. 3. Complete (a) finite and (b) infinite lattices of concrete types and their domains

Subsequently we are interested in the so-called ideals, i.e., sublattices $L_a = \{ x \mid x \leq a \}$ (for any $a \in L_s$). More specifically we are interested in concrete type
ideals $L_{s(t)}$, i.e., the ideals selected by type symbols $t \in T$ and containing the subtypes of the given type interpretation for $t$.

A signature $\Sigma$ and interpretation $s$ together define a set of conformant ($\Sigma$, $s$)-typed many-valued contexts.

**Definition 3.** Let $\Sigma = (M, T, \tau)$ be a context signature and $s$ a type interpretation $s: T \rightarrow 2^U$. A many-valued context $C = (G, M, W, I)$ is a ($\Sigma$, $s$)-typed context if for all $m \in M$, $m: G \rightarrow s(\tau(m))$.

A typed context has a derived concept lattice via implicitly derived attribute type scales (since the typed context is also a many-valued context).

**Definition 4.** Let $\Sigma = (M, T, \tau)$ be a context signature where $s$ is a type interpretation $s: T \rightarrow 2^U$. Let $C = (G, M, W, I)$ be a ($\Sigma$, $s$)-typed context. Then for all $t \in T$, define the implicit scale of $t$ as $S_t = (G_t, M_t, I_t)$ where $G_t = s(t)$, $M_t = L_{G_t}$, and for all $g \in G_t$ and $n \in M_t$, $gI_t n \iff g \in n$. Define the implicit scales of $C$ as the set of scales $S_{\tau(m)}$ for all $m \in M$.

**Convention:** The concept lattice of a typed context $C$ is defined as the concept lattice of the derived single-valued context of $C$ using its implicit scales. We denote this concept lattice $CL_C$.

Figure 3(a) implies scales which are more detailed than those shown earlier in Figure 1 but co-incidentally, encode virtually the same interpretation; under this interpretation the concept lattice of the typed context from Table 1 is identical with the addition of a new (empty) infimum, since e.g. there is no notebook with $HD \geq 1000$. However, using the mechanisms of type theory it is now possible to refine the types for each attribute in intelligent ways, as shown in later sections. The presence of a type for each attribute of the context guarantees that the context can be changed without any further effort needed to define an interpretation to a concept lattice, provided new attribute values are still compatible with their respective type. E.g., any cardinal value for CPU speed would be considered acceptable.

Typed contexts are a generalisation of many-valued contexts in the following sense: for many-valued context $C$ and any set of scales, there exists a type signature and interpretation for the many-valued context making $C$ a typed context denoting the same concept lattice as for $C$ as a many-valued context (though not necessarily denoting the same single-valued context). The generalisation is non-trivial in two respects: First, normally scales need not distinguish between the case when $m$ is partial in $A$ and when $m(A)$ has no common scale attributes. Attribute types always distinguish these cases, since an attribute’s concrete type domain is always a subtype of itself. Second, in general scales may contain redundant attributes, whereas typed attributes never have redundant subtypes—subtypes are denoted by their concrete type (set).

**Lemma 1.** Let $C = (G, M, W, I)$ be a many-valued context with scale $S_m$ for all $m \in M$. Then there exists a many-valued context $C'$ with signature $\Sigma = (M, T, \tau)$ and type interpretation $s: T \rightarrow 2^U$ such that $C'$ is a ($\Sigma$, $s$)-typed context denoting the same concept lattice as $C$.
context, where for each implicit scale $S'_m$ of $C$ with respect to $(\Sigma,s)$, the concept lattice of $S_m$ denoted $CL_{S_m}$ is an order isomorphism (over the subconcept relation) of the concept lattice $CL_{S'_m}$ of $S'_m$. Call $C'$ a typed equivalent of $C$.

**Proof.** (Sketch) One suitable conversion creates a concrete type for each scale and scale attribute, suitably tagged to avoid clashes, where “undefined” values become members of the attribute’s concrete type (and none of its subtypes). Redundant attributes removed in deriving implicit scales do not change concept lattices. \hfill \square

**Corollary 1.** Let $C = (G, M, W, I)$ be a many-valued context with scale $S_m$ for all $m \in M$. Let $D$ be the derived context of $C$ with respect to scaling. Let $C'$ be a typed equivalent of $C$ with implicit scales $S'_m$ indexed by $m \in M$, and derived context $D'$. Then the concept lattices of $D$ and $D'$ are isomorphic.

Let $C = (G, M, W, I)$ be a many-valued context with scale $S_m$ for all $m \in M$. Let *undefined* be a symbol not in $W$. Define $\text{cod}(m) = m(G)$ according to $I$. Then let $C' = (G, M, W \cup \{\text{undefined}\}, I')$ such that $I' = I \cup \{(g, m, \text{undefined})\}$ where $m(g)$ is undefined according to $I$. Let signature $\Sigma = (M, T, \tau)$ and type interpretation $s : T \rightarrow 2^U$. For each attribute $m$, let $\tau(m)$ be some unique type in $T$. Construct the concrete type domain $L$ such that: (i) for each attribute $m \in M$, $s(\tau(m)) = \text{cod}(m) \times m \in L$, (ii) for each attribute $m \in M$ and scale attribute $n \in M_m$, $(I_m n) \times m \in L$. Then $C'$ is a $(\Sigma, s)$-typed context. Assert: $C'$ is a typed equivalent of $C$.

Ultimately there is a need for a user-accessible method for defining new types. In this paper we do not define a language for types but note that this is a standard problem with standard solutions. Options include referring to standard predefined types, composing or transforming existing types, or constructing types either explicitly or programatically. In principle types ought to be definable axiomatically or algebraically. We use two related approaches in the existing typed FCA implementations which are still in progress.

In typed FCA, concrete (sub)types correspond roughly to scale attributes. However, viewed as types, additional possibilities present themselves. First, abstraction can be carried further with the observation that higher types may be context-dependent (see later).

Another benefit is that it becomes possible to bring notions of compatibility to FCA. For example introducing a new concrete type (set) into a signature or otherwise extending it preserves backward compatibility with existing contexts.

So far it is possible for concrete type domains to be infinite, however, so will implicitly derived scales be infinite. Thus in basic implementations, there is no immediate way to compute a concept lattice since naive derivation of a single-valued context will not terminate. Given a direct semantics for typed contexts, it becomes possible and practical to refer to large or even infinite regular type domains and to build tools which reason about them (see next section).
4 Direct Typed FCA

Direct typed FCA involves a semantics of direct interpretation at the level of a typed context without unfolding. Direct typed FCA in turn rests on an abstraction of intents, typed intents, and direct many-valued priming.

4.1 Direct many-valued priming

For many-valued contexts, priming operations can be redefined directly in terms of scales, avoiding reference to its derived single-valued context. The redefinition takes into account that priming in the single-valued context is a disjoint union of the results of priming on respective scales (since scales are themselves contexts).

**Definition 5.** Let $C = (G, M, W, I)$ be a many-valued context with scale $S_m = (G_m, M_m, I_m)$ for $m \in M$ and $D = (G, N, J)$ be the derived single-valued context. The **direct many-valued priming operators** $↑_{C} : 2^G \rightarrow 2^N$ and $↓_{C} : 2^N \rightarrow 2^G$ are defined for $A \subseteq G$ and $B \subseteq N$ as:

$$A↑_{C} = \bigcup_{m \in M \wedge m \text{ is total in } A} \{m\} \times m(A)^{↓_{Sm}}$$ (1)

$$B↓_{C} = \bigcap_{m \in M} m^{-1}(mB)^{↓_{Sm}}$$ (2)

Noting in the above that $N$ and thus any $B \subseteq N$ is viewable as a relation (see the definition of derived context above).

**Lemma 2.** Let $C = (G, M, W, I)$ be a many-valued context with scale $S_m = (G_m, M_m, I_m)$ for $m \in M$ and $D = (G, N, J)$ be the derived single-valued context. For all $A \subseteq G$, $A↑_{C} = A↑_{D}$, and for all $B \subseteq N$, $A↓_{C} = A↓_{D}$.

**Proof.** (Sketch) Based on the equivalence given by the following supporting lemma (not proven here).

**Lemma 3.** Let $C$ be a many-valued context $C$ with scale $S_m = (G_m, M_m, I_m)$ for $m \in M$ and $D = (G, N, J)$ its derived context. Then

$$A \subseteq .J(m, n) \Leftrightarrow m \text{ is total in } A \wedge m(A) \subseteq .I_m n$$ (3)

That is, a set of objects $A$ have a common derived attribute $(m, n)$ iff all $m$-values of objects in $A$ have the $m$-scale attribute $n$.

4.2 Direct Typed FCA

A direct, generalised version of typed context priming is expressed in terms of typed intents. A typed intent is a set of attribute constraints, a pair consisting of an attribute and a value constraint, an opaque representation of a concept from the concept lattice of that attribute’s implicit scale. Viewed as a generalisation of direct many-valued priming, a value constraint is also a generalisation of the
intent which is a result of priming in the concept lattice of the relevant scale as used in direct many-valued priming.

Use of an opaque representation of concepts enables the possibility to delegate representation to types for optimisation, and the possibility to model and reason about infinite attribute type domains, even domains preserving a (countably) infinite number of distinctions.

Together with attribute constraints we introduce direct attribute priming operations, and use these to define direct typed priming operations. We will show that for finite attribute types direct typed priming is equivalent to direct many-valued priming on corresponding implicit scales.

**Definition 6.** Let $\Sigma = (M, T, \tau)$ be a context signature. Let $s$ be a type interpretation $s : T \rightarrow 2^U$. Let $C = (G, M, W, I)$ be a $(\Sigma, s)$-typed context. Let $O_t$ be a set (called the set of constraints), for all $t \in T$. A direct priming type (for any $t \in T$) over $O_t$ is a pair $(\downarrow t, \uparrow t)$ such that

$$\downarrow t : 2^G \rightarrow O_t$$

$$\uparrow t : O_t \rightarrow 2^G$$

and $(\downarrow t, \uparrow t)$ defines a concept lattice isomorphic to the concept lattice of the implicit scale of $t$, with equal extents.

**Definition 7.** Let $\Sigma$, $s$, $C$ be as for the definition above. And let $(\downarrow t, \uparrow t)$ be direct priming types for all $t \in \text{cod}(M)$, over suitable $O_t$. Call

$$O = \bigcup_{t \in \text{cod}(T)} O_t$$

the domain of constraints. We call $I = 2^{(M \times O)}$ the type of intents, if it satisfies for all $i \in I$, $i : M \rightarrow O$, and for all $m \in \text{dom}(i), i(m) \in O_{\tau(m)}$. Now we define direct typed-priming operators by $^\uparrow C : 2^G \rightarrow I$ and $^\downarrow C : I \rightarrow 2^G$, given $A \subseteq G$ and $B \in I$, as:

$$A ^\uparrow C = \bigcup_{m \in M \land m \text{ is total in } A} (m, m(A)^\downarrow t_{\tau(m)})$$

$$B ^\downarrow C = \bigcap_{m \in \text{dom}(B)} m^{-1}(B(m)^\uparrow t_{\tau(m)})$$

The result of $A ^\uparrow C$ for some $A \subseteq G$ is a typed intent: a set of attribute constraints, pairs of the form $(m_i, c_i)$ uniquely labelled by $m_i$, where $c_i \in O_i$.

**Lemma 4.** Given direct typed-priming operators $(^\uparrow C, ^\downarrow C)$, they form a Galois connection, with the two underlying lattices isomorphic to the intent/extent lattices of $CL_C$.

**Proof.** (Sketch) We note a correspondence between the operators $A ^\uparrow C$ and $A ^\downarrow D$. $A ^\uparrow C$ gives a typed intent and $A ^\downarrow D$ gives an intent in $CL_C$. Both operations are
structure-preserving with respect to each other since for all \( m \in M \), \( m(A) \uparrow \tau(m) \) and \( m(A) \uparrow \preceq \) are isomorphic (c.f. def. 1), and always \( m(A) \uparrow \preceq \neq \emptyset \) from the way implicit scales are generated. The order relation on extents remains the same in either case. The order relation on typed intents, \( \preceq \), is expressible as:

\[
B_1 \preceq B_2 \iff \forall b \in M : (b, c_1) \in B_2 \Rightarrow (b, c_2) \in B_1 \land (c_1 \preceq_T(b) c_2).
\]

(9)

It remains to explicitly satisfy the preconditions for Galois connection and concept lattice isomorphism.

Direct-typed FCA can be realised practically for large or even infinite attribute types (provided the context itself is finite), provided that there exists a suitable type in the host programming language and a bijection between values in the type and concepts in the concept lattice corresponding to the required concrete type ideal. (And of course the many-valued context under interpretation must still be finite and relatively small.)

Figure 3(b) shows an alternative, more precise, infinite interpretation of the typed notebooks context signature. Consider \( \text{CardinalAnti} \), the type of cardinal numbers with an anti-ordinal interpretation (a generalisation of scales for CPU and HD). That is, \( \text{CardinalAnti} \in T, s(\text{CardinalAnti}) = \{0, 1, \ldots\} \), and for all \( i \in s(\text{CardinalAnti}) \), \( \{i, i+1, \ldots\} \in L_{s(\text{CardinalAnti})} \) (type tags omitted), ordered by the subset relation.

Coincidentally, the concept lattice corresponding to a scale encoding the ideal \( L_{s(\text{CardinalAnti})} \) is isomorphic to the ideal. Thus we must have a representation for each concept corresponding to every expression for all \( x \) in \( s(\text{CardinalAnti}) \) of the form \( \{x \leq i \mid i \in \{0, 1, \ldots, +\infty\}\} \). We can model such predicates by extending a typical integer type with distinguished elements \(-\infty\) and \(+\infty\), representing each concept (opaquely) \( x \leq i \) simply by \( i \) itself. More importantly, the many families of similar or derived domains, e.g., where subtypes only distinguish between integers modulo 10, or products (for e.g. interordinals), there are corresponding representations.

To label a typed concept lattice, every attribute constraint must be expressible in a human-readable way. This in turn requires that every concept in the concept lattice corresponding to an attribute ideal have a unique label.

It is possible to generalise the well-known \( \text{addIntent} \) algorithm [15] for incremental concept lattice construction for typed intents. The \( \text{addIntent} \) algorithm is essentially a traversal on a Galois lattice and relies only on lattice-theoretic properties of intents. Since all such properties are preserved for typed intents, the generalisation is possible. For example \( \text{addIntent} \) uses the subset \( \subseteq \) operation and other similar operations to check ordering of intents. As we have shown above, the same operation can be achieved directly on typed intents, through replacement of operations on intents (propositional) as lattice-theoretic intent operations. The \( \cap \) and \( \cup \) operations are replaced by \( \land \) (meet) and \( \lor \) (join) respectively. This variant is implemented in our Scala prototype (see below).
The abstraction of intents enables optimisation. Since the type can choose an optimal representation for concepts (e.g. for ordinal-numerics, an upper bound). For a naive FCA implementation embodying the definition of derived context above, construction of the single-valued context would not terminate for scales with infinite attributes.

5 Scala Implementation

To demonstrate feasibility of a direct approach to typed FCA we have prototyped an implementation in the Scala language. The Scala implementation follows the mathematical formalisation of direct typed FCA above, with a few key variations.

Following Section 4, a foundational design choice is to realise direct typed FCA through delegation of attribute value parsing and construction to types, representation of attribute value scale priming results as (opaque) scale lattice concept references, and decomposition of direct many-valued object priming into separate elements incorporating opaque representations of scale priming results. The key point of variation is to bypass explicit construction of both the formal type domain and implicit scales, instead focussing on the concept lattice of each attribute type as the core semantic entity.

An attribute type is viewed as an interface abstracting the concept lattice of the underlying concrete type, providing operations corresponding to priming, lattice-theoretic operations such as meet and join, needed to implement key infrastructure such as the generalised \texttt{addIntent} discussed earlier, and operations for parsing values and labelling concepts.

Predefined types include \texttt{Bool}, \texttt{Nominal}, and \texttt{Ordinal}. Type composition operators include \texttt{Product} (records) and \texttt{PowerSet}.

Parsing operations are needed because attribute value construction is delegated to attribute types. This is motivated primarily by the observation that users prefer to model values \textit{uninterpreted}. Thus if the user simply enters the text 10 for an attribute with a suitable numeric type, the type is responsible for creating a suitable representation of the value 10 in its respective type\textsuperscript{2}.

For example, the \texttt{Bool} attribute type in our Scala implementation maps all of \texttt{True}, \texttt{true}, \texttt{t}, \texttt{1}, \texttt{x}, \texttt{X} (string values) and \texttt{true} (the value of Scala language type \texttt{boolean}) to the value \texttt{trueBool}. In order to satisfy the interface, type composition operators such as product must implement composite interpretation and construction operations in terms of their type parameters.

For labelling, each attribute type (or concept modelling selected subtype) provides a function mapping an argument modelling the attribute name to a string expressing a suitable constraint over elements of the type.

As discussed in Section 4, the Scala implementation copes in cases where type domains are infinite. Strictly the implementation does not yet provide a

\textsuperscript{2}The Scala implementation distinguishes interpretation and construction phases: When a context is created or modified, string representations of values are parsed and canonicalised. At concept analysis time canonical representations are converted to a representation in the actual attribute type domain.
type which takes fullest advantage of this feature. Ordinal types are currently based around fixed precision integer representations. Ordinals based on arbitrary precision numbers are yet to be implemented.

The Scala prototype has been proven so far for small examples, including as a tool for automatically generating most of the lattices in this paper.

As a consequence of the direct typed approach, some aspects of the user experience are more pleasant: for example, numeric attribute types, being modelled as infinite, do not require bounds. No systematic attempt has been made to compare performance with the Java prototype however comparable problems appear to be slightly faster with the Scala prototype. We speculate that constant factor improvements ought to be possible for genuinely many-valued contexts.

6 Java Implementation and Higher Types

Design goals for the Java implementation included: extensibility from propositional to many-valued and typed contexts; flexibility to define and constrain types for objects and contexts, and; flexibility to select freely between underlying data structures and algorithms such as context representations, lattice generation and search. We have also explored the use of statistical analysis to enable abstract specifications of abstract specification of interpretations of totally-ordered type domains such as the real numbers.

Types in the Java implementation are implemented by “unfolding” values into the equivalent one-valued context equivalents. Once characteristics of a type have been decided (for example, ordinal with a particular range), then the conceptual scale function is generated. Then for each value for that attribute in the objects, the equivalent of a corresponding row of the scale is used in the context. Absence of a value in the typed context is represented by a blank row in the corresponding one-valued context.

There is also provision for a user to provide a prior classification of objects, informing the FCA process. For example, we may designate all objects in the context as notebooks, while s15p and m70 are gamer-level laptops and the e7 is a netbook. Figure 4(a) shows a UML diagram for this classification hierarchy.

Type constructors based on standard patterns for statistical sampling are also provided. The Quantile constructor classifies according to “bins,” which are subranges of contiguous values defined by a user-defined sequence of boundaries. Values in the same bin are treated as equivalent under classification. Presence of elements in a bin is represented by × in the one-valued context. The Uniform constructor classifies values into a nominated number of uniformly-sized sized bins. The Quantized constructor bins values into a user-defined number of subranges, the boundaries of which are automatically chosen so as to be equiprobable according to a user-defined distribution function.

Finally the InferredQuantized constructor bins values as for Quantized but the distribution function is induced from observed values of the actual type in the context. The InferredQuantized constructor exemplifies a generalisation to context-dependent types, which are parameterised by the actual values of the
type used in the relevant attribute of a (conformant) context. A specific concrete attribute type is determined depending on the values used. The Java prototype realises this abstract type constructor by linking the type and the subcontext corresponding to its value instances, then using a preprocessing phase to create an actual **Quantized** type whenever the context is updated.

Figure 4(b) shows the concept lattice for the notebooks example, based on a prior classification and specification of HD and CPU attribute types as **InferredQuantized**. In output from the Java prototype, intents (or intent portions) are drawn as the first of a pair of items inside the concept ellipse rather than above it. Likewise extents are drawn as the second of the pair, rather than below the ellipse. The lattice is a combination of both the prior classification and the concept lattice that would otherwise arise without prior classification. This allows the user to reconcile prior belief with discovered classifications. Observe that the (“gamer”) machine m70 is a more powerful version of the (“notebook”) r7 and the (“netbook”) e7 is a low-powered notebook. Note in this example that the statistical types have resulted in different interpretations of the attributes HD and CPU than those used to derive previous concept lattices.

![Concept lattice diagram](image)

**Fig. 4.** (a) Notebooks user-classification; (b) Concept lattice based on user classification and context-dependent statistical types.

7 Related work

Type theories for object-oriented design and programming languages [3] date back to the fundamental work of Liskov et al [10, 9, 8], Cardelli [2, 1] and many
others. Our specific view on object-oriented types as algebraic types draws partly on the third author’s prior work on modular specification of software architectures [7, 13, 14]. Our combination of ordered algebraic types with FCA is novel.

The abstraction of scales in FCA is not novel. However such abstractions tend to be designed ad hoc in a domain-specific way. Higher-level concept lattices based on abstract intents are also not new. For example Messai et al. [11] recently introduced “many-valued concept lattices” in work on modelling modelling notions of fuzzy similarity at the level of many-valued contexts. All attributes are mapped into the interval [0, 1] and a Galois connection is noted on derivation operators between object sets paired with constraints over attribute values mapped into subranges of [0 – 1]. Ganter and Wille [5] documented combinatorial contexts in the form of e.g. products which were applied to scales and their concept lattices. However to the best of our knowledge there is no work which realises such an approach in a general type-theoretic setting. Using a type lattice or concept lattice of a scale as a basis for abstracting scales for FCA appears to be novel.

8 Conclusions and future work

We introduced typed FCA, its underlying theory and also discussed applications of this in two separate implementations in our current project. We demonstrated in theory and through an example that typed FCA enables the systematic expression of abstract but precise constraints on attribute value classes in terms of type declarations known from type systems in object-oriented design methods and programming languages. Our expectation is that this will ease domain experts into the use of FCA when domain ontologies are partly known and declared while many classes of objects in the context are still to be discovered. We have also shown that typed FCA permits implementations with infinite attribute subtype lattices, while this would require an infinite number of scale attributes which are possible in theory but not implemented in current FCA implementations to our knowledge.

Moreover, typed FCA is compatible with context-dependent type definitions, in which subtypes (of attribute values) are defined based on the occurrence of values in the context table. An example for such subtypes is the automatic partitioning of statistical or probabilistic attribute types (not object types) using a clustering tool. For rapidly changing context such attribute types would require recomputation of the scale tables for multi-valued contexts while in our formalisation and implementation the interpretation of the attribute types and their subtype structure is encapsulated in the type representation.

The full implications of context-dependent types is outside the scope of this paper and will require future work. In particular we are working on incrementally reclassifying objects when attribute subtypes change.

Further work also includes combining our results on typed FCA with parallel work on context graphs [6].
9 Acknowledgements

Some of the work described here was funded under the ARC grant LP0455105. We also thank Jens Koetters and David Squire for many fruitful discussions on typed priming. We are also grateful to Jane Jayaputera and Terence Law for their contributions to an earlier Java prototype.

References