Typed Formal Concept Analysis

Ian Peake  Ian Thomas  Heinz Schmidt

\(^1\)Distributed Software Engineering and Architecture discipline group, School of Computer Science and Information Technology, and RMIT Platform Technology Research Institute, RMIT University, Melbourne, Australia

ICFCA’09
Overview

- **Background**: FCA for rapidly expanding knowledge webs
- **Approach**: try to extend FCA theory and infrastructure with domain knowledge in the form of abstract data types ("Typed FCA")
- **Contribution**:
  - Theory of *Typed Contexts* extends many-valued contexts with signatures + interpretations based on simple type theory
  - Evaluation of complementary, experimental implementations (Java, Scala) demonstrate typing for common patterns.
  - Simple statistical approximations
  - *Context-dependent types*
  - FCA directly at level of types
Speaker background: software engineering, software architecture, source code analysis, software behaviour modelling and analysis

e.g. architecture of (software for) on-the-fly personalised assembly of complex objects from distributed knowledge sources: transport, enterprise warehousing, manufacturing control or social networking..

Discover underlying models and taxonomies of subject domains; build conceptual frameworks

Personalised assemblies

Huge number of objects

Rapidly changing personal preferences, objectives, tasks, data
Example domain: feature matrix comprehension
(Image: Austin Modine, used by permission)

<table>
<thead>
<tr>
<th>Features</th>
<th>Commodore SX-64</th>
<th>Macbook Air</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal DVD/CD Player</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Built-in Ethernet Port</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD Card Slot</td>
<td></td>
<td></td>
</tr>
<tr>
<td>User-replaceable Battery</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Backlit Keyboard</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FireWire (IEEE) Interface</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Winner: Macbook Air!
FCA challenges

- Need for usable representations of prior knowledge
- How to improve FCA performance?
- How to assist FCA domain experts users in managing and reusing good abstractions for techniques e.g. scaling
Hypothesis: typing could Help

Cardelli, 1985:

- in any domain .. the classification of objects in terms of the purposes for which they are used eventually results in a more or less well-defined type system.
- protect an underlying untyped representation from arbitrary or unintended use.. [hiding it] and constrains the way objects may interact with other objects.
- languages in which the type of every expression can be determined by static analysis are said to be **statically typed**.. facilitating early detection of type errors and [allowing] greater execution-time efficiency .. [and making] programs more structured and easier to read..

In general, we should (strive for strong typing and) adopt static typing whenever possible.
Typed FCA rationale

- Type systems widely used and understood
- Possible benefits flowing from *domain typing*..
- Abstraction enabling succinct, readable, reusable capture of domain knowledge (interpretation)
- Early checking
- Prospect of optimisation
- Prospect of notions of context compatibility, evolution, extension
- Flow back into type theory from FCA?
- Pragmatic approach based on what is practically achievable using existing techniques
- Prospect to ensure safe context evolution, compatibility
- Must integrate with FCA (which has a richer but compatible notion of ”type”?)
Ideal (domain-)typed FCA process

Input (*typed context*):
- Prior, partial domain knowledge encoded as *domain object/attribute types*
- Data encoded as *typed context*

Outputs:
- Concept lattice systematically refines or falsifies context data conformance to prior types

Interactive/iterative:
- User may react, correct, repeat process
Typed Contexts

Definition

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.

Let \( U \) be a countable set (called the **universe of values**).

Definition

Given a set of type labels \( T \), a **(concrete) type interpretation** \( s \) for \( T \) is a total mapping \( s : T \rightarrow 2^U \).

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

- **Many-valued context extended with attribute type names**

<table>
<thead>
<tr>
<th></th>
<th>CPU:CardinalAnti</th>
<th>HDD:CardinalAnti</th>
<th>GBit:Bool</th>
<th>Btooth:Bool</th>
<th>Pro:Bool</th>
</tr>
</thead>
<tbody>
<tr>
<td>e7</td>
<td>80</td>
<td>8</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>
<td>false</td>
</tr>
<tr>
<td>r9</td>
<td>166</td>
<td>100</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>r7</td>
<td>180</td>
<td>120</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>s9</td>
<td>180</td>
<td>120</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>s15</td>
<td>200</td>
<td>160</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>s15p</td>
<td>200</td>
<td>160</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>m70</td>
<td>250</td>
<td>1000</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>
Example of a type domain

Note: correction needed in proceedings: $L_s = \{ s(t) \mid t \in T \}^\circ \cup \{ 1_s, 0_s \}$ where $A^\circ$ is the closure of a set of sets under union and intersection. Thus $(L, \subseteq, \cap, \cup)$ forms a lattice.
Implicit scales for each typed attribute

CPU, HDD | ≥166 | ≥180 | ≥200 | .. | .. | ..
--------- |------|------|------|----|----|----
166       | X    |      |      |    |    |    
--------- |------|------|------|----|----|----
180       | X    | X    |      |    |    |    
--------- |------|------|------|----|----|----
200       | X    | X    | X    |    |    |    
--------- |------|------|------|----|----|----

GBit, Btooth, Pro | true | X
true | false
Java implementation: composable unfolding semantics

- **AttributeType** class models attribute types: richer than concrete domain types, abstracts scale
- Instances of **AttributeType** names a concrete domain type, specifying functions to: check if value is in type, map values directly to bit-vector per attribute type, etc.; thus abstractions of scales
- **AttributeTypes** are composable (e.g. Product/Powerset)
- A global “product” method derives single-valued context based on attribute types
- Types are abstract, reusable, composable and enforce simple conformance (similarities to e.g. D-Sift, Coron)
- Heavyweight design with many variation points; large code base
- Partial support for specification, enforcement of type hierarchy on domain object types
- Variation point: use Java reflection mechanism to specify object type hierarchy and enforce conformance
Statistical types (Java implementation)

- **Statistical type families**
  - Quantile(\{b_1, b_2, .., b_n\}) - subdivide range by given bounds
  - Uniform(b_L, b_H, n) - subdivide b_L..b_H into n bins (n + 1 bounds)
  - Quantized(b_L, b_H, n, f) - subdivide equiprobably given distribution f

- **Variation:** Precision, Ordinal/AntiOrdinal/Interordinal, Underlying static type integer/Real

- **Examples (with common implicit scale)**
  - Quantile(\{0, 175, 250\}):
  - Uniform(0, 250, 2):
  - Quantized(0, 250, 2, f) where f(x) = c, c ≥ 0, for x in [0,..250]:

<table>
<thead>
<tr>
<th></th>
<th>≥0</th>
<th>≥175</th>
<th>≥250</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>166</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>180</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Java implementation: Context-Dependent Types

- InferredQuantized(n) - subdivide into n bins according to context (again c.f. Coron, others)

Example:

<table>
<thead>
<tr>
<th></th>
<th>CPU:CardinalAnti</th>
<th>HDD:InferredQuantized(2)</th>
<th>GBit:Bool</th>
<th>Btooth:Bool</th>
<th>Pro:Bool</th>
</tr>
</thead>
<tbody>
<tr>
<td>e7</td>
<td>80</td>
<td>8</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>
<td>false</td>
</tr>
<tr>
<td>r9</td>
<td>166</td>
<td>100</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>r7</td>
<td>180</td>
<td>120</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>s9</td>
<td>180</td>
<td>120</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>s15</td>
<td>200</td>
<td>160</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>s15p</td>
<td>200</td>
<td>160</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>m70</td>
<td>250</td>
<td>1000</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Implicit scale for HDD derived by InferredQuantized type based on observed distribution function:

<table>
<thead>
<tr>
<th></th>
<th>≥0</th>
<th>≥120</th>
<th>≥1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Direct Typed FCA

- C.f. Ganter’s Pattern Structures, Gugisch’s Many-Valued Contexts, etc.
- *Direct many-valued priming* defined directly in terms of subordinate priming operations on (implicit) attribute scales viewed as single-valued contexts, using typed intents
- *Direct priming types* abstractions of concept lattices of scales
- *Typed intents* abstract single-valued intents in terms of concepts in direct priming types. Eg.:
  - standard intent: \( \{ C \geq 166, C \geq 180, H \geq 100, H \geq 120, G \} \)
  - typed intent: \( \{(C, \geq 166), (H, \geq 160), (G, \{\text{true}\})\} \)

Theorems:
- Direct many-valued priming is isomorphic to priming in single-valued context
- Direct typed priming is isomorphic to direct many-valued priming,
- Concept lattice generated from typed priming is isomorphic to lattice of implicitly derivable single-valued context
Scala implementation

Types:
- Types extend direct-priming types, otherwise similar to Java
- Types encapsulate user and domain representation functions parsing, checking, conversion to value in domain
- Internal representation hidden by attribute type: optimisation possibilities (c.f. e.g. Ganter, Kaytoue, etc.)
- Type combinators e.g. Product for interordinal, many others are possible based on standard lattice transformations
- Abstraction: computable operations on large (incl. sometimes infinite) type domains
- Concept lattice builders based on NextClosure, AddIntent
Evaluation

- Potential benefits of (domain) typing FCA: abstraction, validation, standard language for expressing context interpretations, compatibility, optimisation, reusability (of interpretation knowledge)
- Need for a standard, complete, extensible language for context domain types (even in our own software)
- Entanglement between representation and interpretation may compromise compatibility
- Many related techniques already in use but perhaps not recognised as related to domain typing
- Many possible approaches based on partial/dependent types..
- (Cardelli85) “Static typing may also lead to a loss of flexibility and expressive power by prematurely constraining the behavior of objects to that associated with a particular type.”
- No notion of partial domain information / partial types
- Needs to be proven on some real applications
- Not yet formalised notions of domain object type hierarchy, context-dependent types
Conclusions

- Refined hypothesis: domain typing is intertwined with but distinct from the much richer notion of “typing” in FCA
- There are potential advantages in the systematic use of domain typing in FCA, e.g. standardised reuse of interpretation strategies, consistent checking, representation
- Future work: see evaluation, with highest priority on application in software architecture
Thank you. Questions?
Definition

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. Define direct many-valued priming operators $\uparrow^c : 2^G \to 2^N$, $\downarrow^c : 2^N \to 2^G$ for $A \subseteq G$ and $B \subseteq N$ as:

$$A^{\uparrow c} = \bigcup_{m \in M \land m \text{ is total in } A} \{m\} \times m(A)^{\uparrow s_m} \quad (1)$$

$$B^{\downarrow c} = \bigcap_{m \in M} m^{-1}(mB.\downarrow s_m) \quad (2)$$

(Note: $N$ and $B \subseteq N$ are relations.)

Lemma

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 $A \subseteq G$, $A^{\uparrow c} = A^{\uparrow D}$, and for all $B \subseteq N$, $A^{\downarrow c} = A^{\downarrow D}$. 
Definition

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

$$\downarrow t : 2^G \rightarrow O_t \quad (3)$$

$$\uparrow t : O_t \rightarrow 2^G \quad (4)$$

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

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 $O_t$. Define

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

the domain of constraints. Call $I = 2^{(M \times O)}$ the type of intents, if for all $i \in I$, $i : M \rightarrow O$, and for all $m \in \text{dom}(i)$, $i(m) \in O_{T(m)}$. 


Definition

Define **direct typed-priming operators** $\uparrow_C : 2^G \to I$ and $\downarrow_C : I \to 2^G$, given $A \subseteq G$ and $B \in I$, as:

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

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

$A \subseteq G^{\uparrow_C}$ is a **typed intent**; a set of **attribute constraints**, pairs of the form $(m_i, c_i)$ uniquely labelled by $m_i$, where $c_i \in O_t$.

Lemma

**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 $CL_C$. 