**RAC**SAM
*Rev. R. Acad. Cien. Serie A. Mat.*
VOL. **98 (**1), 2004, pp. 3–16
Ciencias de la Computación / Computational Sciences

# Verification of the Formal Concept Analysis

**José Antonio Alonso, Joaquín Borrego, María José Hidalgo, Francisco Jesús Martín–Mateos and José Luis Ruiz–Reina**

**Abstract.** This paper is concerned with a formal verification of the Formal Concept Analysis framework. We use the PVS system to represent and formally verify some algorithms of this theory. We also develop a method to transform specifications of algorithms based on finite sets into other executable ones, preserving its correctness. We illustrate this method by constructing an executable algorithm to compute an implicational base of the system of implications between attributes of a finite formal context.

**Verificación del Análisis formal de conceptos**

**Resumen.** En este trabajo se realiza una verificación formal de la teoría del Análisis formal de conceptos. Usamos el sistema PVS para especificar y verificar formalmente los fundamentos matemáticos de esta teoría. Además, desarrollamos un método para transformar especificaciones de algoritmos basadas en el uso de conjuntos finitos en otras evaluables, preservando la corrección. Ilustramos este método construyendo un algoritmo evaluable para calcular una base de implicaciones del sistema de implicaciones entre atributos de un contexto formal finito.

## 1. Introduction

Formal Concept Analysis (FCA) is a learning technique for discovering conceptual structures in a large amount of data, developed since 1982 by R. Wille and B. Ganter [6]. Using FCA techniques several systems have been created, making possible interactive exploration of large database [11].

This technique has been applied to a wide variety of issues related to knowledge representation and database exploration: inheritance and interfaces relationships [1], integration of free–text and ontologies [5], management conceptual of emails [3], knowledge and data discovery in medical texts [2], etc.

Our goal is to show an experience on formalization, validation and correct implementation of a mathematical concept (FCA) in an automated reasoning system. For that purpose, we will follow the first chapters of the book "Formal Concept Analysis" [6], in which the authors present the mathematical foundations of this theory as well as some algorithms. So far the FCA has been studied in the Mizar system, where a formalization and characterization of the lattice of concepts has been done [9]. In this paper, we show that the PVS theorem prover [8] provides an adequate support for the formalization of FCA.

We have two main reasons to tackle this work. First, show that the PVS specification language, in addition to its associated theorem prover, is adequate for achieving a suitable formal verification of the foundations of FCA. The definitions and reasoning in PVS are close to the theory FCA due, mainly, to the

Presentado por Luis M. Laita.
*Recibido*: February 27, 2004. *Aceptado*: October 13, 2004.
*Palabras clave / Keywords*: Formalized Mathematics, Formal Concept Analysis, Formal Methods, PVS, Higher Order Logic
*Mathematics Subject Classifications*: 68T15, 68T30, 03G10

use of the set theory included in PVS. In our work we have also done an extensive use of the available library of finite sets.

Second, we explore how we can transform specifications constructed over finite sets, hence not executable, in other executable ones. Although not every definitions in PVS are executable, there are a wide fragment of PVS executable, by generating Common Lisp code [10]. In that sense, we formalize the notions of data refinement and operation refinement used by Dold [4, 7] and we establish their main properties, building thus a framework in which we could relate differents specifications of the same notion. Then, we apply it to obtain executable specifications of the functions of the theory FCA.

## 2. The PVS system

The PVS system combines an expressive specification language with an interactive theorem prover [8]. In this section, we present a brief description of the PVS language and prover, introducing some of the notions used in this paper.

The PVS specification language is built on a classical typed higher–order logic with the basic types `bool`, `nat`, `int`, in addition to the function type constructor `[D -> R]` and the product type constructor `[A, B]`. The type system is also augmented with *dependent types* and *abstract data types*. In PVS, a set `A` with elements of a type `T` is identified with its characteristic predicates and thus, the expressions `pred[T]` and `set[T]` have the same meaning. A feature of the PVS specification language is *predicate subtypes*: the subtype $\{$`x:T | p(x)`$\}$ consists of elements of type `T` satisfying `p`. It uses `(A)` to indicate the subtype $\{$`x:T | A(x)`$\}$. Predicate subtypes are used for constraining domains and ranges of functions in a specification and, therefore, for defining partial functions. In general, type-checking with predicate subtypes is undecidable. So, the type-checker generates proof obligations, called *type correctness conditions*(TCCs). This TCCs are either discharged by specialized proof strategies or proved by the user. In particular, for defining a recursive function, it must be ensured that the function terminates. For this reason, in the definition of a recursive function, we have to provide a *measure function*. This generates a TCC which states that the measure function applied to the recursive arguments decreases with respect to a well–founded ordering.

A built-in *prelude* and loadable *libraries* provide standard specifications and proved facts on a large number of theories. PVS specifications are packaged as *theories* that can be parametrized in types and constants. The definitions and theorems of a theory can be used by another theory by *importing* it. Proofs in PVS are presented in a sequent calculus. The commands of the PVS prover include induction, quantifier instantiation, rewriting and simplification.

## 3. Formal Concept Analysis in PVS

In this section, we briefly describe the FCA foundations. At the same time, we show the formalization we have developed in PVS. The framework in which concepts are placed is known as a **formal context**: a tuple $(O, A, I)$, where $O$ is a set of **objects**, $A$ is a set of **attributes** and $I \subseteq O \times A$ is a **relation** such that $(d, a) \in I$ means "the object $d$ **has** the attribute $a$".

We illustrate this definition by means of an example from the theory of binary relations. The objects of our example are six binary relations. The attributes are five well known properties of binary relations: reflexive, symmetric, transitive, total and equivalence. A simple format for writing a finite formal context is a **cross table**: one row for each object and one column for each attribute, marking a cross in the intersection to show that an object has an attribute.

| | Reflexive (**R**) | Symetric (**S**) | Transitive (**Tr**) | Total (**To**) | Equivalence (**E**) |
|---|---|---|---|---|---|
| $S_1$ | **x** | **x** | **x** | **x** | **x** |
| $S_2$ | | **x** | | **x** | |
| $S_3$ | | **x** | **x** | **x** | |
| $S_4$ | | | **x** | **x** | |
| $S_5$ | **x** | | **x** | **x** | |
| $S_6$ | **x** | **x** | | **x** | |

In the sequel, we will use this example.

In order to specify this notion in PVS, we consider two uninterpreted types, `T1` and `T2`, denoting the type of the objects and the type of the attributes, respectively. We define the type `FFCT` for representing the general structures over which we will construct the type for representing a finite formal context:

```
FFCT: TYPE =  [# obj: finite_set[T1],
                 attrib: non_empty_finite_set[T2],
                 relation: finite_set[[T1, T2]] #]
```

where $[\#\ a_1 : t_1, \ldots, a_n : t_n\ \#]$ is the form for record types in the language of PVS. Then, the finite formal contexts can be represented by the following subtype of `FFCT` [1]:

```
FFC:TYPE = {C:FFCT | LET Ob = obj(C), A = attrib(C), I = relation(C) IN
                     ∀ pair∈I: LET (ob, a) = pair IN ob∈Ob ∧ a∈A}
```

In what follows, $C = (O, A, I)$ always denotes a finite formal context.

## 3.1. The lattice of concepts

FCA is based on the idea that a concept is an unit consisting of two parts: the extent, which covers all the objects belonging to the concept; and the intent, which compresses all the attributes valid for all those objects.

Thus, the **derivation operators** for a formal context $C$ can be defined as follows:

- Given a set $X$ of objects of $C$, the **intent** of $X$, denoted as $X'$, is the set of their common attributes

```
intent(C)(X: finite_set[T1]): finite_set[T2] =
  IF X = ∅ THEN attrib(C) ELSE
  {a: T2 | ∀ (d:(X)): (d, a) ∈ relation(C)} ENDIF
```

- Dually, given a set $Y$ of attributes of $C$, the **extent** of $Y$, denoted by $Y'$ is the set of the objects having all attributes from $Y$.

```
extent(C)(Y: finite_set[T2]): finite_set[T1] =
  IF Y = ∅ THEN obj(C) ELSE
  {d: T1 | ∀ (a: (Y)): (d, a) ∈ relation(C)} ENDIF
```

- A pair $(X, Y)$ is a **formal concept** of $C$ iff $X \subseteq O, Y \subseteq A, X' = Y$ and $Y' = X$.

```
concept?(C)(pair: [finite_set[T1], finite_set[T2]]): bool =
 LET (X, Y) = pair IN X ⊆ obj(C) ∧ Y ⊆ attrib(C) ∧
                      intent(C)(X) = Y ∧ extent(C)(Y) = X
```

[1] The syntax of PVS specifications has been modified to increase its readability with a more familiar mathematical notation.

The set of all formal concepts of $C = (O, A, I)$ is denoted as $\mathcal{B}(O, A, I)$. In PVS, we define the type representing the concepts of a formal context $C$ and the corresponding accessor functions [2] as follow:

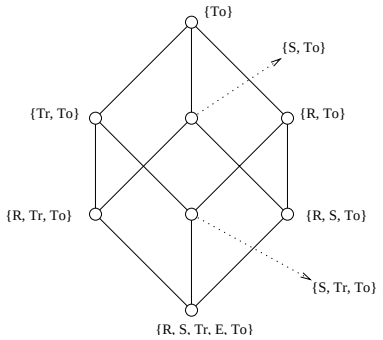
```
concept(C): TYPE = (concept?(C))

extent_concept(C)(pair: concept(C)): finite_set[T1] = proj_1(pair)
intent_concept(C)(pair: concept(C)): finite_set[T2] = proj_2(pair)
```

Formal concepts can be partially ordered in a natural way: a concept $(X_1, Y_1)$ is a **subconcept** of $(X_2, Y_2)$ iff $X_1 \subseteq X_2$ or, equivalently, $Y_2 \subseteq Y_1$.

```
subconcept?(C)(pair1, pair2: concept(C)): bool =
 extent_concept(C)(pair1) ⊆ extent_concept(C)(pair2)
```

We can view the relationship among the concepts of our example by the following diagram. In it, the nodes are the concepts, represented only by their intents, and the arcs represent the subconcept relation.




The main algebraic property of the set of concepts of a formal context $C$ is the following

**Theorem 1** *$(\mathcal{B}(O, A, I),$ `subconcept?`$)$ is a complete lattice, in which for every set of formal concepts, $\{(X_k, Y_k) : k \in K\}$, the supremum and the infimum are given by*

$$\bigvee(\{(X_k, Y_k) : k \in K\}) = \Big((\bigcup_{k\in K} X_k)'', \bigcap_{k\in K} Y_k\Big)$$

$$\bigwedge(\{(X_k, Y_k) : k \in K\}) = \Big(\bigcap_{k\in K} X_k, (\bigcup_{k\in K} Y_k)''\Big)$$

where we have denoted `intent(C)(extent(C)(Y))` by $Y''$ and `extent(C)(intent(C)(X))` by $X''$:

PROOF.

1. The relation is a partial order. We prove this property using the built in relation `partial_order?`, where `partial_order?[D](rel)` is verified if `rel` is a partial order in `D`:

   ```
   po_subconcept: LEMMA partial_order?[concept(C)](subconcept?(C))
   ```

2. We define in PVS the expressions to compute the supremum `lub_fca`. For this purpose, we use the `Union` and the `Intersection` of an arbitrary set of sets, and we define a function to obtain the set of extents (resp. intents) of a set of concepts, using the built in function `image(f)(S)`, that applies `f` to the elements of the set `S`.

[2] The built in selector `proj_i(x)` selects the i–th element of a tuple.

```
Union(I:set[set[T]]): set[T] = λ x: ∃ (A: (I)): A(x)
Intersection(I:set[set[T]]):set[T] = λ x: ∀ (A:(I)):A(x)

extents_set_concepts(C)(S: set[concept(C)]): set[set[T1]] =
   image(extent_concept(C))(S)

lub_fca(C)(S:set[concept(C)]): concept(C) =
 (extent(C)(intent(C)(Union(extents_set_concepts(C)(S)))),
  Intersection(intents_set_concepts(C)(S)))
```

In this specification, we have constrained both the domain and the range of the function `lub_fca`. So, it generates the following TCC ensuring that `lub_fca` satisfies the range constraint given by the predicate subtype `concept(C)`:

```
lub_fca_TCC1: OBLIGATION
  FORALL (C: FFC, S: set[concept(C)]):
  concept?(C)
    (extent(C)(intent(C)(union(extents_set_concepts(C)(S)))),
     intersection(intents_set_concepts(C)(S)))
```

This TCC is not automatically proved. In order to prove it, we establish that for an arbitrary set $\{X_k : k \in K\}$ of sets of objects, $\bigcap_{k\in K} X_k' = (\bigcup_{k\in K} X_k)'$. So, $\bigcap_{k\in K} Y_k = \bigcap_{k\in K} X_k' = (\bigcup_{k\in K} X_k)'$. Then, we have

$$\bigvee(\{(X_k, Y_k) : k \in K\}) = \Big((\bigcup_{k\in K} X_k)'', (\bigcup_{k\in K} X_k)'\Big)$$

and we prove that this is a formal concept of $C$.

3. Finally, we prove the following result using the built–in predicate `least_upper_bound?`, where `least_upper_bound?(rel)(x,S)` is satisfied if `x` is the least upper bound of `S`:

```
lub_fca_is_lub: LEMMA
 ∀ (S:set[concept(C)]):
    least_upper_bound?(subconcept?(C))(lub_fca(C)(S), S)
```

Hence, we have proved that each collection of formal concepts have a supremum, belonging to the lattice. In the same way, we prove the results corresponding to the infimum. ■

## 3.2. Implications between attributes

In those cases in which we have to classify a large number of objects with respect to a relative small number of attributes, we could deal with rules or the implications between attributes, i.e, with statements of the form: "every object with attributes $a, b, c \ldots$ also have attributes $a', b', \ldots$" . Then, we could characterize the intents of concepts of a formal context by a set of rules. In this situation, we study the possible attribute combinations, the *attribute logic*. Formally, an **implication between attributes** in $C = (O, A, I)$ is a pair of subsets of attributes $(Y_1, Y_2)$, denoted as $Y_1 \rightarrow Y_2$. The set $Y_1$ is the **premise or antecedent** of the implication and $Y_2$ its **conclusion**.

In PVS, we use a structure for representing a generalized type of implications

```
implication_gen: TYPE = [# premise: finite_set[T2],
                           conclusion: finite_set[T2] #]
```

and we specify the type of implications between attributes of a context $C$ as the subtype:

```
implication(C): TYPE =
 {imp:implication_gen | premise(imp)⊆attrib(C) & conclusion(imp)⊆attrib(C)}
```

Let us now see the semantic of implications between attributes, specifying it in PVS. A set of attributes $Z$ **respects** or is a **model** of the implication $Y_1 \to Y_2$ if $Y_1 \not\subseteq Z \vee Y_2 \subseteq Z$ (i.e., $Y_1 \subseteq Z \Rightarrow Y_2 \subseteq Z$).

```
respects(C)(Z: finite_set[T2], imp: implication(C)): bool =
 premise(imp) ⊈ Z ∨ conclusion(imp) ⊆ Z
```

A set of attributes $Z$ **respects a set** of implications $\mathcal{L}$ if $Z$ respects each implication in $\mathcal{L}$.

```
respects(C)(Z: finite_set[T2], L:set[implication(C)]): bool =
 ∀ (imp: (L)): respects(C)(Z, imp)
```

An implication $Y_1 \to Y_2$ **holds** in a set of sets of attributes if each of its elements respects $Y_1 \to Y_2$.

```
holds(C)(imp: implication(C), S:set[finite_set[T2]]): bool =
 ∀ (Z:(S)): respects(C)(Z, imp)
```

An implication $Y_1 \to Y_2$ **holds in a context** $C$, or $Y_1 \to Y_2$ **is an implication of** $C$, if the intent of every object of $C$ respects $Y_1 \to Y_2$. That is, whenever each object having all the attributes in $Y_1$ also has all the attributes in $Y_2$.

```
holds(C)(imp: implication(C)): bool =
 ∀ (ob: T1): ob ∈ obj(C) => respects(C)(intent(C)({ob}), imp)
```

Then, for each formal context $C$, we define the set of concepts of $C$, `concepts(C)`, the set of implications that holds in $C$, `holds(C)`, and the set of models of the implications hold in $C$, `models(C)`.

```
concepts(C) = {pair: [finite_set[T1], finite_set[T2]]| concept?(C)(pair)}
holds(C) = {imp: implication(C)| holds(C)(imp)}
models(C)(S) = {Z: finite_set[T2] | Z ⊆ attrib(C) ∧ respects(C)(Z,S)}
```

And we prove the relationship between the models of the implications that hold in $C$ and the concepts of $C$:

**Theorem 2** *The set of models of the implications that hold in C is the set of the intents of concepts of C.*

```
models_holds: THEOREM models(C)(holds(C))=intents_set_concepts(C)(concepts(C))
```

Let us note that to specify this result in PVS we have used the expression `intents_set_concepts(C)(S)`, that obtains the set of intents of the concepts of $S$.

Using this property, we can to obtain the concepts of a finite formal context $C$ from the models of the implications that hold in $C$, and vice versa. Nevertheless, the number of implications that hold in a context can be very large and contain many trivial implications. Therefore, it is natural to look for a small set of implications from which everything else could be derived: an *implicational base*. Specifically, an implication $Y_1 \to Y_2$ **follows (semantically)** from a set of implications $\mathcal{L}$ in $C$ if each subset of attributes of $C$ respecting $\mathcal{L}$ also respects $Y_1 \to Y_2$.

So, given formal context $C$, we look for a set of implications $\mathcal{L}$ having the following properties:

- **sound**: each implication in $\mathcal{L}$ holds in $C$

```
sound(C)(L): bool = ∀(imp:(L)): holds(C)(imp)
```

- **complete**: each implication that holds in $C$ follows from $\mathcal{L}$

```
complete(C)(L):bool=∀(imp:implication(C)):holds(C)(imp)⇒follows(C)(imp,L)
```

- **non redundant**: no implication in $\mathcal{L}$ follows from other implications in $\mathcal{L}$

```
non_redundant(C)(L):bool=∀(imp:(L)): NOT follows(C)(imp, remove(imp, L))
```

Guigess and Duquenne [6] have proved that for every context with a finite set of attributes $A$, there is a sound, complete and non redundant set of implications, called **stem base** or **Duquenne–Guigues–Basis**. For this purpose, it defines a **pseudo intent** as a set of attributes $S$ which is not an intent ($S'' \neq S$), but contains the closure ($P''$) of every proper subset that is also pseudo intent.
In our example:

- $\emptyset$ is a pseudo intent, since $\emptyset'' = \{To\} \neq \emptyset$
- $\{R\}$ is not a pseudo intent, since $\{R\}'' = \{S_1, S_5, S_6\}' = \{R, To\} \neq \{R\}$, but the only proper subset of $R$ that is a pseudo intent is $\emptyset$, and $\emptyset'' = \{To\} \not\subseteq \{R\}$

We specify this definition in PVS, providing the cardinal of $S$ as measure function to prove its termination:

```
pseudo_intent?(C)(S: finite_set[T2]): RECURSIVE  bool =
 S ≠ intent(C)(extent(C)(S)) ∧
 ∀ (P: finite_set[T2]): P ⊂ S ∧ pseudo_intent?(C)(P) =>
                        intent(C)(extent(C)(P) ⊆ S
 MEASURE card(S)
```

Then, we establish the following theorem:

**Theorem 3** *For every finite formal context $C$, the set of implications*

$$\mathcal{L} = \{S \to S'' : S \text{ is pseudo intent of } C\}$$

*is sound, non redundant and complete ($\mathcal{L}$ is called stem base).*

In our example, the stem base is

$$\{\emptyset \to \{To\}, \{E, To\} \to \{R, S, Tr, To, E\}, \{S, R, Tr, To\} \to \{S, R, Tr, To, E\}\}$$

and a slightly modified version of the stem base is

$$\{\emptyset \to \{To\}, \{E, To\} \to \{R, S, Tr\}, \{S, R, Tr, To\} \to \{E\}\}$$

obtained by $\{S \to S'' \setminus S : S \text{ is pseudo intent of } C\}$

Let us note that the specification in PVS of the notion of pseudo intent is more general than the definition itself. Therefore, in order to describe the stem base, we only consider the subsets of the attributes that verify the PVS–definition of pseudo intent:

```
pseudointents(C): set[finite_set[T2]] =
 {S: finite_set[T2] | S ⊆ attrib(C) ∧ pseudo_intent?(C)(S)}

stem_base(C): set[implication(C)] =
 LET fun = λ(S:finite_set[T2]):
           (# premise:= S,
              conclusion:= intent(C)(extent(C)(S)) #)
  IN image(fun)(pseudointents(C))
```

We have proved the stem base properties in PVS in a similar way as they are stated in [6]:

```
stem_base_sound: LEMMA  sound(C)(stem_base(C))
stem_base_non_redundant: LEMMA  non_redundant(C)(stem_base(C))
stem_base_complete: LEMMA  complete(C)(stem_base(C))
```

The stem base is not the only implicational base, but it plays a special role. Among other reasons, there is not other implicational base with less elements and it can be algorithmically obtained. Moreover, to built the stem base we only need an algorithm to compute the pseudo intents of $C$. We are going to specify such an algorithm, based on the recursive definition of pseudo intent. Here, we are interested in constructing an abstract and correct specification of this algorithm and later, we will transform it into another executable specification, preserving its correctness.

Hence, if `gen_pseudo_intents(C)` computes the set of pseudo intents of $C$, we can compute the stem base, as follows:

```
gen_stem_base(C): set[implication_gen] =
 LET fun = λ (Y: finite_set[T2]):
            (# premise:= Y,
               conclusion:= intent(C)(extent(C)(Y)) #)
 IN image(fun)(gen_pseudo_intents(C))
```

Let us describe the generation of the pseudo intents of $C = (O, A, I)$ stepwise, according to its cardinal:

- Step $0$: generate the pseudo intents of cardinal $0$: $\mathcal{S}_0 = \{\emptyset\}$ if $\emptyset$ is pseudo intent and $\mathcal{S}_0 = \emptyset$, otherwise.
- Step $k+1$: generate the pseudo intents of cardinal up to $k+1$, $\mathcal{S}_{k+1}$, from the pseudo intents of cardinal up to $k$, $\mathcal{S}_k$. For this, we introduce a restricted notion of pseudo intent. That is, $P$ is a *pseudo intent restricted to* a set of sets $\mathcal{S}$ if $P \neq P''$ and for every $X \in \mathcal{S}$, if $X \subset P$ then $X'' \subseteq P$. Now, the set $\mathcal{S}_{k+1}$ can be defined as follows:
$$\mathcal{S}_{k+1} = \mathcal{S}_k \cup \{P \subseteq A : |P| = k+1 \wedge P \text{ is a pseudo intent restricted to } \mathcal{S}_k\}.$$

Thus, if `subsets_card(R,k)` is the set $\{B : B \subseteq R \wedge |B| = k\}$ and `pseudo_restricts(C)(G,S)` is $\{P \in G : P$ is pseudo intent restricted to S$\}$, the specification of this algorithm in PVS is the following:

```
gen_pseudo_intents(C): set[finite_set[T2]] =
      gen_pseudo_intents_aux(C)(attrib(C), 0, ∅)

gen_pseudo_intents_aux(C)(A: finite_set[T2], k:upto(card(A)),
                          S: finite_set[finite_set[T2]]):
                          RECURSIVE set[finite_set[T2]] =
 LET NS = pseudo_restricts(C)(subsets_card(A,k), S) IN
  IF k = card(A) THEN S ∪ NS ELSE
  gen_pseudo_intents_aux(C)(A, k+1, S ∪ NS) ENDIF
  MEASURE card(A)-k
```

Finally, we prove the correctness of this specification in PVS, that is, `gen_pseudo_intents(C)` builds the set of pseudo intents of $C$.

```
correct_gen_pseudo_intents: THEOREM
  gen_pseudo_intents(C) = pseudointents(C)
```

PROOF. The proof of `gen_pseudo_intents(C)` $\subseteq$ `pseudointents(C)` is obtained by establishing that if $Y \in$ `gen_pseudo_intents_aux(A,k,`$\mathcal{S}$`)`, (i.e, $Y$ has been introduced in the step $k$), then $Y \subseteq A$ and $Y$ is pseudo intent. We prove it using `card(A)-k` as measure–induction. In order to prove `pseudointents(C)` $\subseteq$ `gen_pseudo_intents(C)`, we establish by induction to, that if $Y \subset A \wedge |Y| \leq k$ and $Y$ has not been introduced in the step $k$, then $Y$ is not a pseudo intent. ■

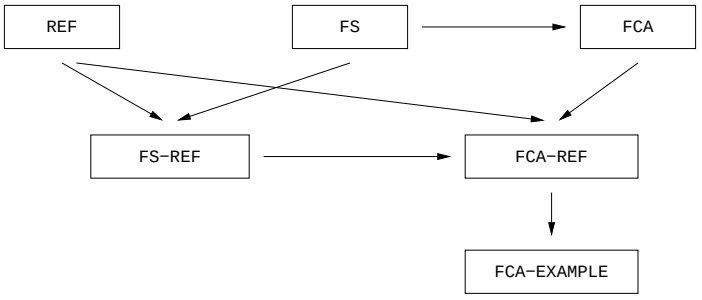
# 4. Executable specifications in FCA

The PVS specification language is designed to be expressive rather than executable, since it allows quantification over infinite domains and higher order equalities. However, a wide fragment of PVS is executable, by generating Common Lisp code from PVS, that can be evaluated in the PVS–ground–evaluator [10]. Specifically, the executable expressions are those that do not contain free variables, uninterpreted functions or constants, quantification over infinite domains and equality between higher–order terms. The specifications we have constructed in the previous section use operations over finite sets, which are not executable. Our main goal in this section is to show how to transform the specifications of FCA built so far into their respective executable specifications.

Based on the notion of refinement of data types used for Dold [4] and Jones [7] we have done the following:

- We have built a theory, `REF`, establishing the general notions of refinements and its main properties.
- We have established a refinement of the theory of finite sets, `FS`, by the theory of lists, `FS-REF`.
- We have used both theories to refine the theory `FCA` from the previous section by the theory `FCA-REF`, in which the specifications of the functions are executables.
- Finally, we have instantiated the parameters of the theory `FCA-REF` and we have evaluated some examples, included in the theory `FCA-EXAMPLE`.

We can see the relationship between these theories in the following diagram:




## 4.1. Refinement of data types.

Given the types $T$ and $R$, we say that a **data refinement** of type $T$ by the type $R$ is a surjective application $f : R \to T$.

```
f: VAR [R->T]
refinement?(f): bool = ∀ (t:T): ∃ (r:R): f(r) = t
```

Intuitively, this function provides the relationship between abstract values and their representations. It is clear that there is at least one representation, non necessarily unique, for any abstract value.

It is easy to see that a type can be refined stepwise by sequential composition of refinements. We have proved that, if $f : R \to T$ is a data refinement of $T$ by $R$ and $g : Q \to R$ is a data refinenemt of $R$ by $Q$, then $f \circ g : Q \to T$ is a data refinement of $T$ by $Q$.

On the other hand, let us consider a function $op : T_1 \to T_2$ and let us suppose that we have the data refinements given by the functions $f_1 : R_1 \to T_1$ and $f_2 : R_2 \to T_2$. Then, we say that the function $op_{ref} : R_1 \to R_2$ is a **refinement** of $op$ if the following diagram commutes:

$$\begin{array}{ccc} T_1 & \xrightarrow{op} & T_2 \\ {\scriptstyle f_1}\uparrow & & \uparrow{\scriptstyle f_2} \\ R_1 & \xrightarrow{op_{ref}} & R_2 \end{array}$$

```
op: VAR [T1 -> T2]  op_ref: VAR [R1 -> R2]
refinement_op?(op,op_ref): bool = ∀ (r1:R1): op(f1(r1)) = f2(op_ref(r1))
```

In essence, we require that $op_{ref}$ has the same behaviour as $op$. Being more precise, let us suppose that we have established a correctness theorem for $op$, in terms of pre and post conditions. That is, a theorem like

$$(\forall y \in T_1)[\phi(y) \Rightarrow \rho(y, op(y))]$$

where $\phi$ is the precondition and $\rho$ is the postcondition. Then, if $op_{ref}, \phi_{ref}$ and $\rho_{ref}$ are refinements of $op, \phi$ and $\rho$, respectively, we have proved that

$$(\forall x \in R_1)[\phi_{ref}(x) \Rightarrow \rho_{ref}(x, op_{ref}(x))]$$

which is just the correctness theorem corresponding to $op_{ref}$.

Moreover, a specification of an algorithm is normally built combining some other specifications of functions. Hence, we hope that for constructing a refinement of a specification of an algorithm it is enough to construct a refinement of each function used in it, and to replace it. In that sense, we have proved that if $op1_{ref}$ and $op2_{ref}$ are refinements of $op1$ y $op2$, respectively, then $op2_{ref} \circ op1_{ref}$ is a refinement of $op2 \circ op1$:

$$\begin{array}{ccccc} T_1 & \xrightarrow{op_1} & T_2 & \xrightarrow{op_2} & T_3 \\ {\scriptstyle f_1}\uparrow & & {\scriptstyle f_2}\uparrow & & {\scriptstyle f_3}\uparrow \\ R_1 & \xrightarrow{op_{1_{ref}}} & R_2 & \xrightarrow{op_{2_{ref}}} & R_3 \end{array}$$

## 4.2. Executable refinements of operations over finite sets

A natural way to represent finite sets is by means of lists. First, we can think in representing a finite set $A$ by a list with the same elements of $A$, although with possibly repeated elements. However, the elements of $A$ could also be finite sets, or another type that was necessary to refine to. Hence, let us consider

a data refinement given by $f : R \to T$. From this, we can establish a data refinement of type "finite sets with elements in $T$" by the type "lists with elements in $R$", by the function `c(f):list[R]` $\to$ `finite_set[T]`, defined as follows:

```
c(f)(l: list[R]): RECURSIVE finite_set[T] =
 CASES l OF
  null: ∅,
  cons(x, l1): add(f(x), c(f)(l1))
 ENDCASES
 MEASURE length(l)
```

We are interested in building refinements corresponding to the operations over finite sets specified in the theory FCA, developed in section 3. For this, we have used operations on lists such that "simulate" the behaviour of analogous operations on finite sets. We have proved that the built–in operations `null?`, `cons` and `append` are refinements of `empty?`, `add` and `union`, respectively. As for the membership relation, we have to notice that if $f$ is injective, the operation `member` from lists is a refinement of the operation `member` of finite sets. Otherwise, that is not true. Thus, we have defined a new membership relation, with respect to an equivalent relation, and we have proved that, if $\equiv_f$ is the equivalence relation induced in $R$ by $f$ ($r_1 \equiv_f r_2 \leftrightarrow f(r_1) = f(r_2)$), `member(`$\equiv_f$`)` is a refinement of `member`. The specification in PVS is the following:

```
member(rel:(equivalence?[R]))(x:R, l:list[R]): RECURSIVE bool =
 CASES l OF
  null: false,
  cons(a, l2): rel(a, x) ∨ member(rel)(x, l2)
 ENDCASES
 MEASURE length(l)
```

Note that if the relation $f(r_1) = f(r_2)$ is executable, then the operation `member(`$\equiv_f$`)` is also executable.

We have followed the same process for the refinements of `inter`, `subset?`, `remove`, `rest` and `card`. Without going into more details, we show a table with the names of operations over lists which are refinements of the corresponding operations over finite sets:

| finite sets | lists | finite sets | lists |
|---|---|---|---|
| empty? | null? | add | cons |
| member | member($\equiv_f$) | union | append |
| intersection | inter($\equiv_f$) | remove | remove_l($\equiv_f$) |
| choose | car | rest | rest_l($\equiv_f$) |
| subset? | subset?_l($\equiv_f$) | card | card_l($\equiv_f$) |
| image | map | Union | Append |
| powerset | powerset_ref | subsets_card | subsets_card_l |

In case that we have finite sets which elements are also finite sets, the data refinement is given by

`c(c(f)):list[list[R]]`$\to$`finite_set[finite_set[T]]`

and the relation `member(`$\equiv_{c(f)}$`)` is not executable because the expression `c(f)(l1) = c(f)(l2)`. In order to solve this problem we have defined an equivalence relation `equal_l?(f)` in `list[R]`, whose evaluability only depends on evaluability of $f$. This relation also verify `equal_l?(f)(l1, l2)` $\Leftrightarrow$ `c(f)(l1) = c(f)(l2)`.

Thus, we could consider that given a data refinement $f : R \to T$, for each operation over finite sets with elements in $T$, we have an operation over lists in $R$ which it is its refinement. Now, we are going to use them to construct in PVS a refinement of data and operations of the theory FCA specified in section 3.

## 4.3. Executable specifications in FCA

In order to construct an executable "refinement" of the theory FCA, we have to build a data refinement for each type or data type in FCA, and an executable refinement of each function specified in FCA. Next, we show a scheme of this process, applied to the function that computes the stem base of a finite formal context.

1. The types FFCT and FFC are refined by the types

```
FFCT_ref: TYPE = [# obj_ref: list[T1],
                    attrib_ref: (cons?[T2]),
                    relation_ref: list[[T1, T2]] #]
FFC_ref: TYPE =
 {R: FFCT_ref | LET Ob_ref = obj_ref(R), A_ref = attrib_ref(R),
                    Re_ref=relation_ref(R)
  IN every(λ (pair): proj_1(pair) ∈ obj_ref(R) &
                  proj_2(pair) ∈ attrib_ref(R))(Re_ref)}
```

   by the data refinement

```
trans(R: FFCT_ref): FFCT[T1,T2] =
 (# obj:= c(id[T1])(obj_ref(R)),
    attrib:= c(id[T2])(attrib_ref(R)),
    relation:= c(id[[T1,T2]])(relation_ref(R)) #)
```

   The types implication_gen and implication are refined by the types

```
R: VAR FFC_ref[T1, T2]
implication_gen_ref: TYPE = [# premise_ref: list[T2],
                                conclusion_ref: list[T2] #]
implication_ref(R): TYPE =
  {imp_r: implication_gen_ref |
          subset?_l(premise_ref(imp_r), attrib_ref(R)) ∧
          subset?_l(conclusion_ref(imp_r), attrib_ref(R))}
```

   by the data refinement

```
transf_imp_gen(imp_g_r:implication_gen_ref):implication_gen[T1, T2]=
  (# premise:= c(id[T2])(premise_ref(imp_g_r)),
     conclusion:= c(id[T2])(conclusion_ref(imp_g_r)) #)
```

2. A refinement of the specification to compute the set of pseudo intents is built replacing each function by its refinement:

```
gen_pseudo_intents_aux_ref(R)(ls: list[T2],
                              k:upto(cardinal_l[T2, T2](ls)),
                              S: list[list[T2]]):
                              RECURSIVE list[list[T2]] =
 LET NSL=pseudo_restricts_ref(R)(subsets_card_l(ls,k),S)
  IN IF k = cardinal_l(ls) THEN append(S, NSL) ELSE
        gen_pseudo_intents_aux_ref(R)(ls, k+1, append(S, NSL)) ENDIF
  MEASURE cardinal_l(ls)-k

gen_pseudo_intents_ref(R): list[list[T2]] =
 gen_pseudo_intents_aux_ref(R)(attrib_ref(R), 0, null)
```

   Let us remark the similarity of this specification with the specification showed in page 8.

3. The corresponding correctness theorem is proved using that a refinement preserves the correctness

```
correct_gen_pseudo_intents_ref: THEOREM
  pseudo_intents_set_ref?(R, gen_pseudo_intents_ref(R))

pseudo_intents_set_ref?(R, LL): bool =
 ∀ (l1:list[T2]): pseudo_intent_ref?(R)(l1) ⇔
                 ∃(l2:list[T2]): l2 ∈ LL ∧ equal_l?(l1, l2)
```

4. Finally, a refinement of the specification to compute the stem base is:

```
gen_stem_base_ref(R): list[implication_gen_ref[T1, T2]] =
 LET fun = λ( Y: list[T2]):
       (# premise_ref:= Y,
          conclusion_ref:= intent_ref(R)(extent_ref(R)(Y)) #)
 IN map(fun)(gen_pseudo_intents_ref(R))
```

In order to show a computation of the stem base, we represent in PVS the formal context of page 3:

```
IMPORTING  FCA_REF[string, string]
obj: list[string] = (: "S1", "S2", "S3", "S4", "S5", "S6" :)
atr: list[string] = (: "R", "S", "Tr", "To",''E'' :)
rel: list[[string, string]] =
   (: ("S1", "R"), ("S1", "S"), ("S1", "Tr"),..., ("S6", "To") :)
C: FFC_ref[string, string] =
   (# obj_ref:= obj, attrib_ref:= atr, relation_ref:= rel #)
```

and executing the function `gen_stem_base_ref` in the pvs–ground–evaluator, we obtain:

```
<GndEval> "gen_stem_base_ref(C)" ==>
(: (# premise_ref := (: :), conclusion_ref := (: "To" :) #),
   (# premise_ref := (: "E", "To" :),
      conclusion_ref := (: "R", "S", "Tr", "To", "E" :) #),
   (# premise_ref := (: "To", "Tr", "S", "R" :),
      conclusion_ref := (: "R", "S", "Tr", "To", "E" :) #) :)
```

Let us remark that the base obtained is the same of the stem base cited at page 7:

$$\{\emptyset \to \{To\}, \{E, To\} \to \{R, S, Tr, To, E\}, \{S, R, Tr, To\} \to \{S, R, Tr, To, E\}\}$$

# 5. Conclusions

We have presented in this paper an experience on formalization, validation and correct implementation of a mathematical concept (FCA) using PVS, introducing the basic concepts and results that allows us to verify related algorithms in the theory. As a first step, we have developed the theory using finite sets. The advantage of using finite sets is that the formalization is more natural in the sense that definitions, theorems and their proofs are very close to the ones of the "source theory".

Since one of our goals is to obtain formally verified executable algorithms, we have developed a methodology that allows us to obtain executable specifications (using lists) from specifications done using finite sets. For that purpose, we have formalized the notions of type refinement and operation refinement. This is a generic methodology which we have applied successfully to obtain executable specifications of algorithms of other theories.

## References

[1] Arévalo, G.; Mens, T. *Analysing Object-Oriented Application Frameworks Using Concept Analysis* Workshop on Managing Specialization/Generalization Hierarchies, LNCS 2426, pp. 53–63, September 2002

[2] Cole, R.; Eklund, P.; Wlaker, D. *Using Conceptual Scaling in Formal Concept Analysis for Knowledge and Data Discovery in Medical Texts* International Symposium on Knowledge Retrieval, Use, and Storage for Efficiency, pp.151-164, 1997

[3] Cole, R.; Stumme, G. *CEM: A Conceptual Email Manager* 7th International Conference on Conceptual Structures, ICCS'2000, Springer Verlag, 2000

[4] Dold, A. *Formal Software Development using Generic Development Steps* PhD thesis. Ulm University, 2000.

[5] Erdmann, M. *Formal Concept Analysis to learn from the Sisyphus-III Material* Gaines, Musen (eds.) Proceedings of the 11th knowledge Acquisition for Knowledge–Based Systems Workshop (KAN'98) Banff, Canada, 1998

[6] Ganter, B. and Wille, R. *Formal Concept Analysis*. Springer–Verlag, 1999.

[7] Jones, C. B. *Systematic Software Development using VDM* Prentice–Hall International, The University, Manchester, U.K., second edition, 1990.

[8] *PVS (Prototype Verification System), 2002*. See `http://pvs.csl.sri.com`

[9] Schwarzweller, C. *Mizar Formalization of Concept Lattices*. In Mechanize Mathematics and its Applications, vol. 1, 2000

[10] Shankar, N. *Efficiently executing PVS*. Technical report, SRI International, November 1999.

[11] Stumme, G.; Wille, R. and Wille, U. *Conceptual Knowledge Discovery in Databases Using Formal Concept Analysis Methods*. In: J. M. Zytkow, M. Quafofou (eds.): Principles of Data Mining and Knowledge Discovery. Proc. 2nd European Symposium on PKDD '98, LNAI 1510, Springer, Heidelberg 1998, 450-458

J.A. Alonso, J. Borrego, M.J. Hidalgo, F.J. Martín–Mateos and J.L. Ruiz–Reina
Dpto. de Ciencias de la Computación e Inteligencia Artificial
E.T.S. de Ingeniería Informática
Universidad de Sevilla
Avd. Reina Mercedes, s/n
41012 Sevilla
Spain
{`jalonso,jborrego,mjoseh,fmartin,jruiz`}`@us.es`