Fundamenta Informaticae XXI (2001) 1001–1022
IOS Press

Mechanical Analysis of Finite Idempotent Relations

Florian Kammüller *

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin

Abstract. We use the technique of interactive theorem proving to develop the theory and an enumeration technique for finite idempotent relations. Starting from a short mathematical characterization of finite idempotents defined and proved in Isabelle/HOL, we derive first an iterative procedure to generate all instances of idempotents over a finite set. From there, we develop a more precise theoretical characterization giving rise to an efficient predicate that can be executed in the programming language ML. Idempotent relations represent a very basic, general mathematical concept but the steps taken to develop their theory with the help of Isabelle/HOL are representative for developing algorithms from a mathematical specification.

Keywords: Formal Specification of Computer Systems, Idempotent Relations, Interactive Theorem Proving, Higher Order Logic

1. Introduction

In computer science, relations over finite domains are commonly accepted as the model for operations of hardware and software components. Idempotence of these operations is increasingly recognized as an important property. A concrete example is the specification of components in service-oriented architectures [7]. Since invoking them once has the same effect as invoking them several times, idempotent operations are suitable for modelling services, e.g. a cancellation service for bookings. Idempotence has a positive effect on the robustness of service-oriented applications and thus operations should – where possible – be idempotent [7, page 77]. This paper presents the derivation of an algorithm in Isabelle/HOL. The algorithm produces finite idempotent relations and their numbers. The presented work

Address for correspondence: TU Berlin., Sekr. TEL 12-3, Ernst-Reuter-Platz 7, 10587 Berlin, email: flokam@cs.tu-berlin.de
*Research for this paper has been partly performed in collaboration with J.W. Sanders, University of Oxford and has been supported by grant ARC D/03/20258 of the German Academic Exchange Service (DAAD).

is an updated and extended version of [11] and [10, Chapter 2]. In Section 2, we motivate the use of idempotent relations and illustrate their characterizing features by means of examples. The gathered insights are summarized in a formal characterization of idempotents. The mechanical proof of this theorem in Isabelle/HOL is described (Section 3). Section 4 refines the content of the idea behind the theorem and sketches an algorithm that can be extracted from the theoretical characterization. In contrast to [11] where this algorithm has been directly defined in ML, we examine the enumeration of idempotent relations to arrive at a more detailed characterization. Section 5 describes the development of an efficient predicate derived as an ML program from the refined characterization in Isabelle/HOL. The predicate defines the set of all finite idempotents for a given carrier set. Finally, in Section 6 we present the results of the derived ML code and discuss limits. Section 7 draws conclusions with respect to this application example of interactive theorem proving. Some sample output of idempotents for the carrier $\mathbb{N}_3$ created from an automatic ML transformation to LaTeX picture is contained in the Appendix. The full code of the algorithms, the Isabelle/HOL derivations, the LaTeX macros, and further sample output are available at the author's web page [8].

But, before delving into the theory of idempotents, we give in this first section an introduction to interactive theorem proving with Isabelle/HOL as used in this paper.

Isabelle/HOL

The interactive theorem prover Isabelle [14] has foremost been constructed as a generic tool to provide a framework for the creation of specialized theorem provers for various application logics. One of the first papers announcing Isabelle to the scientific community [13] is entitled *Isabelle: the next 700 theorem provers*.

In the overall architecture of Isabelle the core is its meta-logic, i.e. the logic providing a logical deduction framework enabling the inference with axioms and rules given by application logics. Isabelle's meta-logic is itself a fragment of Higher Order Logic (HOL): it contains just implication and universal quantification as junctors. There is no existential quantifier, no negation, no disjunction. Conjunction is mimicked using nested implication. The following meta-logical formula is an example illustrating the universal quantification with `!!`, higher order variables `P` and `Q`, and implication $\Longrightarrow$.

```
!! P Q x. [| P x; Q x |] ⟹ P x
```

The square brackets `[| |]` serve as pseudo-conjunction: they are just an abbreviation for nested implication, i.e the above is equivalent to

```
!! P Q x. P x ⟹ (Q x  ⟹ P x)
```

where the round brackets could even be omitted as the meta-level implication is right-associative. The meta-logic has its own equality == corresponding to logical equivalence.[1]

An embedding of an application, a so-called object-logic, into Isabelle, is realized by a theory. A theory is a file containing new types, constants and definitions of the types and constants. Theories in Isabelle are a very simple module concept, but there is also a concept of locales [12] that is fairly unique in interactive provers. Locales are basically a light-weight module concept that mainly focuses on enclosing local proof contexts, turning them into abstract, instantiatable units. It is particulary well-suited for mathematical proof.

[1]It serves also as equality when defining constants during an embedding.

Since the beginnings of Isabelle, various different object-logics have been embedded into its meta-logic. Three object-logics, namely HOL, constructive type theory (CTT), and Zermelo-Fraenkel set theory (ZF) are now part of the general distribution of Isabelle. Actually Isabelle/HOL, the embedding of HOL, and Isabelle/ZF, the embedding of ZF, are the logics that are mostly used. It may seem a bit strange that HOL, which is the basis of Isabelle's meta-logic, is again embedded into Isabelle as an object-logic. However, its wide application just shows that the full classical HOL is a useful means to reason about computer science related problems. It may seem even stranger that ZF, which is presumably more general than HOL, can be embedded at all into Isabelle. However, the formalization Isabelle/ZF uses basically just one type `i` representing the class of all sets. Hence, the axiomatization is not restricted by the type hierarchy. With respect to deciding which of these two main instantiations of Isabelle one should choose, we cite Larry Paulson: *If you prefer ML to Lisp, you'd prefer HOL to ZF*. The development that turned Isabelle into such a practical tool has been supported strongly by the entire Isabelle community mainly in Cambridge with Paulson and in Munich with Tobias Nipkow.

HOL Example

In computer science applications, we often reason about rather simple domains, like discrete structures, finite sets, partial recursive or primitive recursive functions. For such specialized domains, Isabelle/HOL offers also specialized support for formalization and proof. These specializations are internally resolved and mapped to the principles of conservative extension.

For example, the data type of lists can be defined in Isabelle/HOL using the `datatype` definition package. A `datatype` definition resembles almost exactly the corresponding ML version.

```
datatype α list =    Nil  ("[]")
 | Cons α "α list"  (infixr "#" 65)
```

Using the polymorphism of the type system of Isabelle/HOL, the above definition introduces the type of lists over an arbitrary type of elements. The datatype definition introduces a constructor `Nil` for the empty list and a constructor `Cons` that given an element of type α and a list of α elements constructs a new list. The code in brackets behind the constructors declares the pretty printing (or mixfix) syntax enabling the use of `[]` for the empty list and `x # l` for a constructed list, very similar to the notation in programming languages.

Among the internally generated rules for a datatype specification, there are induction rules for recursive types like the above and injectivity rules for the constructors.

Functions over a datatype may be defined as primitive recursive functions. As an illustrative example, consider the function that appends two lists to form a new list. First, we have to declare this function as a constant in a theory.

```
consts
  append :: "[α list, α list] ⇒ α list" (infixr "@" 65)
```

Next, the semantics of this function is given by two equations representing the base case of an empty list and the recursive call for a constructed list. The names before the colon : are optional rule names for later reference in proofs.

```
primrec
  append_Nil: "[] @ l = l"
  append_Cons: "(x # l1) @ l2 = x # (l1 @ l2)"
```

The primitive recursion scheme has constraints on the way recursion can be defined to ensure that functions defined in this way actually are primitive recursive. If it is possible to express a function in a specification that way, it is certainly advisable to do so because the automatic tactics of Isabelle/HOL are optimized for processing such definitions. Hence, in proofs where, for example, a term needs to be transformed using equational rewriting according to a primitive recursive function definition, this is basically performed fully automatically by Isabelle/HOL's tactical support.

To illustrate briefly the way an interactive proof is performed in Isabelle, we consider the proof of a simple equation that serves to determine the length of a list composed by the newly defined append operator. We assume further as given a function `length` that returns the length of a list. The entire proof script including the statement of the proof goal is as follows.

```
lemma length_composed: "length (l1 @ l2) = length l1 + length l2";
apply (rule_tac list = "l1" in list.induct);
by auto;
```

The first line states the goal to be proved. The second line advises Isabelle to apply the induction rule of the recently created datatype of lists in a backward resolution step to the first list `l1`. Backward resolution is invoked by `rule_tac` which is a special form of backward resolution enabling the explicit setting of rule parameters, here `l1`. The reply of the Isabelle system in an interactive session after this first line consists of two new subgoals.

```
1. length ([] @ l2) = length [] + length l2
2. !!a list.
      length (list @ l2) = length list + length l2
      ⟹ length ((a # list) @ l2) = length (a # list) + length l2
```

These subgoals correspond to the two premises of the list induction rule instantiated to the current proof goal. The remaining two subgoals can now be solved by rewriting with the defining equalities of the list constructors, the definition of `length` (which we have omitted), and the definition of append. These steps are all automatically performed with `auto`, the Isabelle tactic that subsumes other forms of automated tacticals like simplification and the classical reasoner.

After a theorem is proved like in the above case, Isabelle responds with `No subgoals!` The theorem is assigned to the name given by the user in the first line so that it can be used in future derivations.

2. Idempotent Relations

Idempotents are important in various areas of mathematics. An abstract structure is frequently able to be represented in a particular concrete form so that each abstract element is represented as a transformation on some underlying set and the abstract operation is represented as sequential (or functional) composition of transformations. In some cases the idempotents (i.e. the elements e satisfying $ee = e$, where we follow convention and write the algebraic operation as juxtaposition) form the basis for that representation.

Here are two examples. In analysis, a linear function on a vector space is idempotent under sequential composition iff it is a projection onto a subspace of the vector space. That is, the foundation of spectral resolution, in which a (say normal) linear operator is represented in terms of projections [6]. In algebra, much of the structure theory of (abstract) semigroups [3] and some of the theory of ideals in ring theory [4] depends on identifying the idempotents.

In computer science, a transaction consists of a sequence $t = t_1; \ldots ; t_m$ of actions. Given a second transaction $u = u_1; \ldots ; u_n$ the idea is to perform both transactions efficiently but as if each were executed atomically. Efficiency is achieved by interleaving the actions of t with those of u but correctness is preserved only if the desired interleaving is obtained from the sequential composition $t; u$ by interchanging those actions that commute: $t_i; u_j \sqsubseteq u_j; t_i$. If, in doing so, two identical actions become adjacent then one of them can be deleted if it is idempotent: $t_i; t_i = t_i$. Though not common, it is important to take advantage of such simplification whenever possible.

Linear operators, functions and actions are all special cases of relations. Whereas idempotent functions are rather standard, it appears there is not much published about idempotent relations. In the following analysis of idempotent relations, we start from a theoretical characterization and develop that into a verified executable algorithm for idempotent relations over a given finite carrier set. Besides enabling the contemplation of actual instances of finite idempotents, this algorithm can also be used as a basis for counting them.

Examples

A relation $r \subseteq A \times A$ on a given set A – with $\mathrm{domain}\ r \subseteq A$ – is idempotent if $r; r = r$ where ; is relational composition. Relational composition is generally defined as

$$r; s = \{(x, y).\ \exists\, z.\ (x, z) \in r \wedge (z, y) \in s\}\,.$$

In Isabelle, the notation $\circ$ is used to express relational composition because it resembles functional composition. However, since in the field of programming semantics the notation $r; r$ is more common to express relational composition, we adhere for this exposition to the latter. For simplicity, let $r(x)$ stand for the relational image $r.(\!|\{x\}|\!)$ (where $r.(\!|X|\!) = \{y.\ \exists\, x \in X.\ (x, y) \in r\}$). The characterization of idempotents is based on fixpoints of the relation fix r, i.e. elements x of the domain with $x \in r(x)$, or equivalently $(x, x) \in r$.

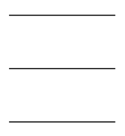
The Isabelle/HOL theory for idempotent relations contains one constant definition for idempotence. Non-recursive functions are introduced in Isabelle as constants with definitions by the `constdefs` device that expects the type and the defining meta-equality as arguments.

```
constdefs
  idempotent :: (α × α) set ⇒ bool  "idempotent r == (r ; r = r)"
```

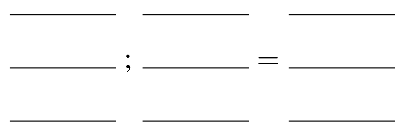
We give now various examples of idempotents to explain the concept and prepare the theoretical characterizations below. The following examples illustrate idempotents by depicting the points related by a finite idempotent relation from left to right. We consider labelled relations, i.e. the order of points matters. To mark the labels in examples, we use subsets of the natural numbers as label set but in principle the label set could be any other finite set. Therefore, in graphical illustrations we omit the labels. For example, the relation consisting of three fixpoints

$$\{(1, 1), (2, 2), (3, 3)\}$$

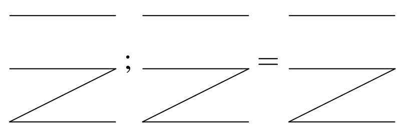
is graphically represented as follows.



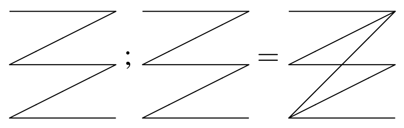
The simplest way to examine idempotence is by a diagrammatic equation as depicted below. Starting from left following the lines over the ; to the right, any point, that is reachable, is related in the resulting relation on the right side of $=$. As our simple example relation is invariant under relational composition with itself, it is idempotent.



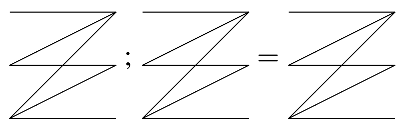
Now, consider a simple extension by relating one fixpoint with another: still idempotence is preserved.



However, if we apply a similar extension to the middle point, the resulting relation has one more edge due to transitivity. That is, the relation on the left of the figure below is *not* idempotent.



The resulting relation on the right of the above figure is again idempotent as verifed below.



In general, we can see that whenever in a scenario like the above, where all points on the left are related to themselves on the right, transitivity is not only necessary but already sufficient for idempotence.

We can already summarize this observation as a first theorem.

Theorem 2.1. Let r be reflexive and transitive. Then r is idempotent.

Proof:
It suffices to show that $r \subseteq r; r$. For any x and y with $(x, y) \in r$, also $(x, x) \in r$ as r is reflexive, hence $(x, y) \in r; r$. □

This paper style proof almost directly corresponds to its representation as a proof script in Isabelle/HOL:

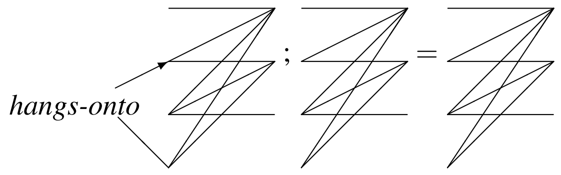
```
theorem 2_1: "[| trans r; refl (Domain r) r |] ⟹ idempotent r";
apply (erule trans_intrans_idemp);
apply auto;
apply (subgoal_tac "(a,a)∈ r");
apply auto;
apply (erule reflD);
by auto;
```

The first line states the goal. To unfold the definition we apply the lemma `trans_intrans_idemp` which reads

`[| trans r; r` $\subseteq$ `r ; r |]` $\Longrightarrow$ `idempotent r`

The main proof clue is to find that `(a,a)` $\in$ `r` which is given explicitly as subgoal to Isabelle. Now, the rest can be done automatically; we only need to input the name of lemma `reflD`, i.e. that reflexivity implies `(a,a)` $\in$ `r`.[2]

Now, we are not only concerned with reflexive relations but relations in general over some domain A. Hence, there might exist elements that are *not* fixpoints that come into play.



The new point on the bottom is not a fixpoint. If it is related to an element in the range of the relation, it can only copy entire ranges of fixpoints to create its own range. Therefore we call such points *hangers-on*. That is, a non-fixpoint cannot be mapped independently to elements; its range is precisely defined by the union of ranges of fixpoints.

$$\forall x.\, x \notin \text{range}\, r \Longrightarrow r(x) = \bigcup y \in \text{fix}\, r \cap r(x).\, r(y)$$

This property holds for all finite idempotent relations r and its conclusion also holds for x if x is a fixpoint (and thus $x \in \text{range}\, r$). The property is going to be proved in the following section and forms the basis for all further constructions.

A Characterization of Finite Idempotent Relations

We characterize idempotent relations under the assumption of finiteness, and show that we cannot do better. The characterization is given by the following theorem.

Theorem 2.2. Let r be a finite relation. Then,

$$\text{idempotent}\, r \quad \equiv \quad \begin{pmatrix} \forall x.\, r(x) = \bigcup y \in \text{fix}\, r \cap r(x).\, r(y) \\ \text{transitive}\, r \end{pmatrix}.$$

Clearly, idempotence $r;r = r$ implies transitivity $r;r \subseteq r$. We can omit $x \in \text{domain}\, r$ in the first conjunct and still assume it in proofs because for $x \notin \text{domain}\, r$ the equality holds trivially.

We are going to prove this theorem using the following lemmata.

Lemma 2.1. Let r be idempotent. Then

$$x \in r(x) \Rightarrow r(x) = \bigcup y \in \text{fix}\, r \cap r(x).\, r(y)\,.$$

[2]If we would have added the definition of reflexivity and `trans_intrans_idemp` to the simplification sets of `auto`, the entire proof would reduce to two lines: the central subgoal insertion and `auto`.

Proof:
Transitivity gives us $\forall y \in r(x).\ r(y) \subseteq r(x)$. Clearly,

$$\bigcup y \in \text{fix}\, r \cap r(x).\ r(y) \subseteq \bigcup y \in r(x).\ r(y)$$

and by transitivity of $\subseteq$ the left-hand-side is a subset of $r(x)$. Since $x \in r(x)$, i.e. $x \in \text{fix}\, r$, we have $r(x) \subseteq \bigcup y \in \text{fix}\, r \cap r(x).\ r(y)$, whereby we have equality. $\square$

Lemma 2.2. Let r be finite and idempotent. Then,

$$x \in \text{domain}\, r, x \notin r(x) \Rightarrow \forall z \in r(x).\ \exists\, y \in \text{fix}\, r \cap r(x).\ z \in r(y)\,.$$

Proof:
We prove that if the assumption and the negation of the conclusion hold, we get a contradiction to r being finite. Assume for contradiction

$$\exists\, z \in r(x).\ \neg\exists\, y \in \text{fix}\, r \cap r(x).\ z \in r(y)\,.$$

Since $(x, x) \notin r$, we need another $y_0 \neq x$ for (x, z) to be in r^2 in order to have $(x, y_0), (y_0, z) \in r$ and thereby $(x, z) \in r$. Now, $y_0 \neq z$ otherwise we had a $y = z$ with $z \in r(z)$ contradicting the assumption. Summarizing, $y_0 \neq x$ and $y_0 \neq z$. However, now $y_0 \in r(x)$ and $y_0 \notin r(y_0)$. By repetition of the argument, we need a y_1 with $(x, y_1), (y_1, y_0) \in r$ with $y_1 \notin \{x, z, y_0\}$, and so forth — ultimately leading to an infinite sequence of $y_i \in r(x)$, contradicting r being finite. $\square$

Now, we are prepared for the proof of the theorem.

Proof of Theorem 2.2:

Correctness ($\Rightarrow$): Let r be idempotent and $x \in \text{domain}\, r$ be arbitrary. If $x \in r(x)$, just apply Lemma 2.1. If $x \notin r(x)$, Lemma 2.2 gives us

$$\bigcup\{z \in r(x)\} \subseteq \bigcup y \in \text{fix}\, r \cap r(x).\ r(y)\,.$$

Since r is idempotent, it is also transitive. Hence, the right-hand-side $\subseteq r(x)$. Since the left-hand-side is equal to $r(x)$ we have that

$$r(x) = \bigcup y \in \text{fix}\, r \cap r(x).\ r(y)\,.$$

If $x \notin \text{domain}\, r$, this is trivial.
Completeness ($\Leftarrow$): Let $r(x) = \bigcup y \in \text{fix}\, r \cap r(x).\ r(y)$. For any $(x, y) \in r$, $y \in r(x)$. Due to assumption, there is y' with $y \in r(y')$ for some $y' \in r(y') \cap r(x)$, i.e. $(x, y') \in r$ and $(y', y') \in r$. Since $y \in r(y')$, also $(y', y) \in r$, hence $(x, y) \in r^2$.

As the second conjunct of the characterization is transitivity, we have on the other hand that if $(x, y) \in r^2$ then $(x, y) \in r$. $\square$

From the theorem, it follows immediately that if a finite idempotent relation is nonempty, then it has a fixpoint.

Corollary 2.1. If $r \neq \varnothing$ is finite and idempotent, then $\exists\, x.\ x \in \text{fix}\, r$.

By contraposition, this implies that if there is no fixpoint, the relation must be infinite.

Corollary 2.2. If r is idempotent, $r \neq \varnothing$ and fix $r = \varnothing$, then r is infinite.

An illustrative example is the relation $<$ on rational numbers.

Example 2.1. The relation $<: \mathbb{Q} \times \mathbb{Q}$ is idempotent and infinite.

The relation $<$ is obviously transitive, and for any x and y with $x < y$ there is an element between x and y.

3. Mechanical Proof

We extend the Isabelle/HOL theory of idempotents already introduced in Section 2 by two more definitions for fixpoints and non-fixpoints of a relation.[3]

```
fixp :: "(α × α) set ⇒ α set"  "fixp r == {x. (x, x) ∈ r}"

nfix :: "(α × α) set ⇒ α set"  "nfix r == {x. x ∈ Domain r ∧ (x, x) ∉ r}"
```

Theorem 2.2 is then proved in the scope of that theory. The relational image of a singleton set is denoted in Isabelle/HOL by `r"{x}`. Otherwise, it may be noted here that the Isabelle/HOL representation is almost exactly like the mathematical notation.

```
finite r ⟹ idempotent r ≡ (∀ x. r"{x} = ⋃ y ∈ fixp r ∩ r"{x}. r"{y} ∧
                                trans r)
```

Proof of Lemma 2.1

The proof of Lemma 2.1 is very simple in Isabelle. Using a lemma that infers transitivity from idempotence, it is just one application of the elimination rule for transitivity. The rest is done automatically using the tactic `auto`.

```
[| idempotent r; x ∈ r"{x} |] ⟹ r"{x} = ⋃ y ∈ fixp r ∩ r"{x}. r"{y}
```

Proof of Lemma 2.2

This part of the proof of Theorem 2.2 is the difficult bit. What is done on paper rather casually and informally by sketching a repetitive process in which yet another element y_i is needed and then concluding that the set $r(x)$ cannot be finite, is harder on the logical level. The repetitive process is first proved as a lemma (see Core Lemma below). Applying this lemma in an induction, the existence of an infinite sequence is proved. Some further theorems that generalize from the existence of this particular sequence then provide the possibility to infer infinity from there. These theorems can then be chained together to construct the contradiction to the assumption of finiteness.

[3]We have to use here `fixp` instead of `fix` as the latter is already used for the Tarski fixpoint-operator.

Core Lemma

The core lemma describes that under the assumptions of Lemma 2.2 it is possible to infer a new element y that is in relation r to all others so far but is not equal to any of the former ones.

```
[| idempotent r; x ∈ Domain r; x ∉ r"{x}; z ∈ r"{x};
 ¬ (∃ y. y ∈ r"{y} ∧ y ∈ r"{x} ∧ z ∈ r"{y}); z = s 0;
 ∀ j. j ≤ n ⟶ s j ∈ r"{x} ∧
    ∀ i. i < j ⟶ (s j,s i)∈ r ∧ s j ≠ s i
|] ⟹ ∃ y. y ∈ r"{x} ∧ ∀ j. j ≤ n ⟶ (y, s j)∈ r ∧ y ≠ s j
```

Similar to the paper style proof, it uses the properties of idempotence to infer that new "middle" element and furthermore transitivity to establish the invariant that it is related to all previous ones. We use here a variable s that formalizes a sequence over natural numbers used in the following to produce the infinite sequence.

A Chain of Lemmata

The first step of the proof, leading to the conclusion that there is an infinite sequence, is an induction that shows that, under the given assumptions of Lemma 2.2, there is such a sequence s.

```
[| idempotent r; x ∈ Domain r; x ∉ r"{x}; z ∈ r"{x};
   ¬ (∃ y. y ∈ r"{y} ∧ y ∈ r"{x} ∧ z ∈ r"{y})
|] ⟹ ∀ n. ∃ s:: nat ⇒ α . z = s 0 ∧
        (∀ j. j ≤ n ⟶ (s j) ∈ r"{x} ∧
        (∀ i. i < j ⟶ (s j, s i) ∈ r ∧ (s j) ≠ (s i)))
```

This proof is an induction over natural numbers. In the induction step, the core lemma is applied to produce the new element of the sequence s having the appropriate properties.
The conclusion of the previous step can be weakened.

```
∀ n. ∃ s:: nat ⇒ α . z = s 0 ∧ (∀ j. j ≤ n ⟶ (s j) ∈ r"{x} ∧
   (∀ i. i < j ⟶ (s j, s i) ∈ r ∧ s j ≠ s i))
⟹ ∀ n. ∃ s. ∀ j. j ≤ n ⟶ s j ∈ r"{x}
        ∧ (∀ i. i < j ⟶ s j ≠ s i)
```

The weaker set of properties of the sequence s is sufficient – abstracting over r"{x} as some set p – to infer that p contains subsets S with cardinality strictly larger than any n.

```
[| ∀ n. ∃ s:: nat ⇒ α .
 (∀ j. j ≤ n ⟶ (s j) ∈ p ∧ (∀ i. i < j ⟶ (s j) ≠ (s i))) |]
⟹ ∀ n. ∃ S. card S = Suc n ∧ S ⊆ p
```

Finally, the property derived in the previous step can be used to infer that the set p is infinite.

```
∀ n. ∃ S. card S = Suc n ∧ S ⊆ p ⟹  ¬ finite p
```

The variable p of type set can be instantiated to r"{x}. Thereby, chaining up all these lemmata, we can put together the proof of Lemma 2.2 by producing a contradiction to the assumption of finiteness of the relation.

```
[| finite r; idempotent r |] ⟹
  ∀ x ∈ Domain r. x ∉ r"{x} ⟶
    (∀ z ∈ r"{x}. ∃ y. y ∈ r"{y} ∧ y ∈ r"{x} ∧ z ∈ r"{y})
```

It may seem a bit odd that we have to derive first the existence of a cardinality for the sets S. However, as infinity is just the negation of finiteness, the only way to construct a contradiction is to arrive at a property typical for a finite set, i.e. a finite cardinality, and that clearly cannot be assumed for the sequence.

The proof of Lemma 2.2 is rather intricate similar to proofs in lattice theory, e.g. [2, 5]. It would be much easier if a sequence could be constructed on the outside of the universal quantification over n, i.e. $\exists s.\forall n \ldots$. However, this is not possible in our case. We have to show that such a sequence exists for each n. Fortunately, as the core lemma can be identified and applied inside the induction this sequence can be prolonged in each step and by identifying the commonality of the sequences — that they are all contained in some set p — we can construct the sequence of sets represented by the existentially quantified S.

Proof of the Theorem

The proof of correctness, i.e.

```
[| finite r; idempotent r |] ⟹ (∀ x. r"{x} = ⋃ y:  fixp r ∩ r"{x}. r"{y} ∧
                                  trans r)
```

just puts together Lemma 2.1 and Lemma 2.2; transitivity is simply contained in idempotence.

For completeness we can infer the property $r \subseteq r; r$ from the first conjunct of the characterization alone.

```
∀ x. r"{x} = ⋃ y ∈  fixp r ∩ r"{x}. r"{y} ⟹ r ⊆ r ; r
```

The other conjunct is transitivity so the inverse inclusion $r; r \subseteq r$ is proved trivially. Finally, we put the two parts together to finish the proof.

4. Constructing Idempotents

In [11] we present an algorithm for the enumeration of idempotents over a finite domain. This algorithm is an ML implementation that is based on the characterization in Isabelle/HOL as seen in the previous section and further refined to a representation of relations by the list datatype in Isabelle/HOL. However, one of the initial question, we set out with, is to have a method for counting idempotents. Our algorithm [11] is not suited to this end as it produces repetitions.

Ideally, we would like to have a simple formula that enables the explicit calculation of the number of idempotents for a finite domain of size n. Counting mathematical entities is not always possible with a simple formula: there are mathematical journals publishing new results on counting specific sequences, e.g. [1, 15] and a dedicated web page for integer sequences [17]. From the recent publication [15] we learn that in particular transitive relations are amongst the difficult entities. As transitivity is part of the definition of idempotence clearly we cannot hope that idempotence is simpler.

In fact, the way these integer sequences are calculated is by specialized software [16]. It seems that here is a field where mechanical computation is an accepted part of mathematics. To further this application of mechanization in science, we illustrate that theorem provers may be employed as development tools for counting algorithms that have been proved to be correct. In that sense, formal software engineering has found an application domain.

In the current section, we want to recapitulate the driving idea that transforms Theorems 2.1 and 2.2 into a construction algorithm. After that, we present a refined version of these theorems summarizing the idea. In the following section, we then show in detail how the Isabelle/HOL theorems may be transformed into executable form.

Construction Idea

Intuitively, we already know from the introductory examples how to construct all idempotent relations over a set A:

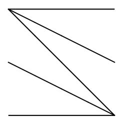
1. select a subset $F \subseteq A$ of fixpoints for r,
2. extend the ranges of the fixpoints with suitable[4] subsets of A,
3. choose ranges for all non-fixpoints n by selecting suitable subsets of fixpoints; let n hang onto the fixpoints' ranges, i.e. enclose their ranges in $r(n)$.

This is roughly the idea we used in [11] for the implementation of an enumeration algorithm. There is one difficulty in the selection of suitable subsets of the ranges: respecting the dependency of the ranges, as described by Theorem 2.2. This difficulty applies to the range extension of fixpoints and non-fixpoints equally. In Step 2. when extending fixpoint ranges, we need to respect possible (mutual) dependencies between ranges of selected fixpoints. For example, if we extend the range of a fixpoint f and select another fixpoint f' to be in $r(f)$ then we also need to incorporate $r(f')$ into $r(f)$. Similarly, if in Step 3. a set of fixpoints is selected, all ranges of other fixpoints contained in the ranges of the selected fixpoints are automatically included.

Whereas from a computational point of view, it is not difficult to resolve such dependencies by including all dependent ranges, it is a problem for the mathematical goal of the enumeration. Repetitions occur. For example, let $r = \{(1,1),(1,2),(2,2),(2,1)\}$. We want to extend for non-fixpoint 3. Whether we choose $\{1\}$, $\{2\}$, or $\{1,2\}$ as the set of fixpoints to hang-onto, does not make any difference. Either way, the result is the same relation $r_{ex} = \{(1,1),(1,2),(2,2),(2,1),(3,1),(3,2)\}$. That is, there are three different ways in the above construction idea to arrive at the same result: we are generating repetitions. The repetition is inherent in this relation because $(2,1)$ as well as $(1,2)$ are part of it building a cycle in the graph of the relation. As a countermeasure to such repetitions in the algorithm, one might think about defining equivalence classes of fixpoints over the relation $\{(x,y).\ (x,y) \in r \wedge (y,x) \in r\}$ and then define *free choices* over connected subgraphs in the resulting factorized noncyclic graph of the relation as differing subgraphs. However, this reduces basically to counting partially ordered sets, a problem already encountered to be difficult [1].

Also, when considering the abstract procedure sketched above, there is another problem that remains even after resolving those repetitions. It is best explained by example. Consider the idempotent relation depicted in the following figure (note, there is no cyclic subgraph).

[4]That is, the range extensions have to respect transitivity.



Here, the non-fixpoint in the middle, say 2, is already in the range of the top fixpoint 3. Imagine this relation was constructed in Step 3. by extending the range of 2 with the range of the bottom fixpoint 1, i.e. $r(1) = \{1\}$. Another possible choice — in another round of the construction — would be to extend the range of the non-fixpoint 2 with the range of fixpoint 3, i.e. $\{1, 2, 3\}$. However, then we would map 2 to itself and it becomes a fixpoint. Thereby, we would create repetitions: for the selection of fixpoints $F = \{1, 2, 3\}$ in Step 1. the resulting idempotent relation – containing 2 as a fixpoint – would occur again.

The repetition illustrated by this second example happens whenever a non-fixpoint is – prior to extension – already in the range of the relation. This scenario gives rise to a general observation about idempotents summarized in the following formula.

$$\forall\, f \in \text{fixp}\, r.\ \forall\, y \in \text{nfix}\, r \cap r(f).\ r(y) \subseteq r(f)$$

This formula characterizes in more detail the case illustrated by the last example: non-fixpoints in the range of fixpoints are constrained to choose their range from the former fixpoint. The above examples show that constructing idempotents inherently bears potential for repetitions. Instead of pursuing the idea of effectively constructing all the necessary combinations for the three cases of the naïve construction idea, we summarize the more refined observation about non-fixpoints in the following theorem.

Theorem 4.1. Let r be a finite relation. Then,

$$\text{idempotent}\ r \equiv \begin{pmatrix} \forall\, n \in N.\ r(n) = \bigcup x \in F \cap r(n).\ r(x) \\ r_F\ \text{transitive} \\ \forall\, f \in F.\ \forall\, y \in N \cap r(f).\ r(y) \subseteq r(f) \end{pmatrix}$$

where $r_F = \{(x, y).\ (x, y) \in r \wedge x \in F\}$, $F = \text{fixp}\, r$, and $N = \text{nfix}\, r$.

Proof:
The theorem can be proved straightforwardly using Theorems 2.1 and 2.2. The most difficult part is the proof of transitivity for the $\Leftarrow$-direction. We describe this part in a bit more detail, referring to the new representation of idempotence given by the right hand side of the equivalence stated in Theorem 4.1 as the *new representation*.

Now, given the three conjuncts described in this new representation, we have to show transitivity, or more formally, $\forall y \in r(x).\ r(y) \subseteq r(x)$ for all $x \in \text{domain}\ r$. The proof works by case analysis over x and y being fixpoints, or not. Most cases are again straightforward. The complicated case is $y \in N$ and $x \in N$, i.e. both are non-fixpoints. We show transitivity pointwisely, i.e. for any y_0 we show $y \in r(x) \wedge y_0 \in r(y) \Rightarrow y_0 \in r(x)$. In this case, however, because of the first conjunct of the new representation, we have a $z \in F$ such that $z \in r(x) \wedge y \in r(z)$ (see Figure 1). With the third conjunct of the new representation, we have, because of $z \in F$ and $y \in r(z) \cap N$, that $y_0 \in r(z)$. To arrive at $y_0 \in r(x)$, it remains to show that $r(z) \subseteq r(x)$. However, this is implied by the first conjunct of the new

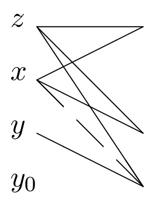


Figure 1. Scenario $x \in N$ and $y \in N$ in proof of Theorem 4.1

representation applied to x, i.e. $r(x) = \bigcup x_0 \in F \cap r(x).\ r(x_0)$ – just replace $r(x)$ in $r(z) \subseteq r(x)$ – which in turn is true for z as $z \in F$, $z \in r(x)$, and $z \in r(z)$. □

The Isabelle proof is about 200 lines of proof script (see [8]). The new characterization given by Theorem 4.1 is a refined version of Theorem 2.2. It partitions an idempotent relation into disjoint subsets determined by the range extended reflexive transitive relation r_F (the fixpoints and their ranges), the non-fixpoints and their ranges for the two cases of non-fixpoints inside and outside the range of the relation. By combining Theorem 2.1 for the reflexive part with Theorem 2.2 for the non-fixpoint part, the characterization of Theorem 4.1 minimizes the property of idempotence by reducing it to disjoint partitions of the relation. Hence, it is a predicate that is more efficient when we consider it as a procedure that checks idempotence of a relation.

Therefore, we use this more fine grained view to derive an efficient predicate implemented in ML.

5. Efficient Predicate

In this section, we present how Theorem 4.1 is transformed from an Isabelle/HOL set representation into an ML-like program inside Isabelle/HOL using the datatype of lists.

We first introduce our list representation for relations in Isabelle/HOL. Then, we show how the functions representing the predicate of Theorem 4.1 can be defined as primitive recursive definitions over that datatype representation. Finally, we give an implementation relation between relations and their implementation as lists and sketch the refinement proofs.

Data Representation

A relation is represented in Isabelle/HOL using the datatype of lists already part of the theory database (see Section 1). A finite relation is now represented as a list of pairs $(x, r(x))$ of elements x of the domain of r and their individual range. For example, the relation

$$\{(1,1),(1,2),(1,3),(2,2),(3,2),(3,3)\}$$

may be represented as follows.

```
[(1,[1,2,3]),(2,[2]),(3,[3,2])]
```

To introduce this representation of relations properly, we define the following type abbreviation using Isabelle's `types` definition.

```
types
  α relation = "(α × α list) list"
```

Implementation

To introduce the notions of all fixpoints `fixl` and non-fixpoints of a relation, we first define a function `fixr` restricting a relation to its fixpoint partition. From the theory database of lists, we use the `map` function, the membership predicate `mem`, and the very elegant filter expression `[x:l.P x]` filtering all elements `x` fulfilling a predicate `P` out of a list `l`.[5]

```
constdefs
  fixr :: "α relation ⇒ α relation"
  "fixr r ==  [x:r. (fst x) mem (snd x)]"

  fixl :: "α relation ⇒ α list"
  "fixl r == map fst (fixr r)"

  nfixl :: "α relation ⇒ α list"
  "nfixl r == map fst [x:r. ¬ ((fst x) mem (snd x))]"
```

The range of an individual element x of a relation, i.e. $r(x)$, is defined as a primitive recursive function `rat`. Therefore, first the constant is declared and second the definition is given by equations contained in a `primrec` block.[6]

```
consts
  rat :: "[α, α relation] ⇒ α list"
primrec
  "rat x [] = []"
  "rat x ((y,rn) # l) =
         if (x = y) then (rn @ (rat x l)) else (rat x l)"
```

Since we possibly find differently ordered lists representing the same set, we implement some auxiliary functions mimicking set containment `lsubset` and equality on the list representation `lset_eq`.

```
consts
  lsubset :: "[α list, α list] ⇒ bool"
primrec
  "lsubset [] l' = True"
  "lsubset (x # l') l = (x mem l) ∧ (lsubset l' l)"
constdefs
  lset_eq :: "[α list, α list] ⇒ bool"
  "lset_eq l l' == (lsubset l l') ∧ (lsubset l' l)"
```

Now, the first part of the efficient predicate represented by Theorem 4.1 states that the partition r_F of the relation r is transitive. To this end, we define the following transitivity check. We use two auxiliary functions that perform a transitivity check pointwisely.

[5]Compare this notation to the rather clumsy ML filter expression.

[6]We use here pattern matching on the left-hand-side of a primrec equation which is a slight modification of the original Isabelle code to enhance readability.

```
consts
  transone :: "[α relation, α list, α list] ⇒ bool"
  transpre :: "[α relation, α list list] ⇒ bool"
primrec
  "transone r ry [] = True"
  "transone r ry (a # l) = (lsubset (rat a r) ry) ∧ (transone r ry l)"
primrec
  "transpre r [] = True"
  "transpre r (ry # l) = (transone r ry ry) ∧ (transpre r l)"
```

Function `transpre` implements $\forall y \in r(x).\, r(y) \subseteq r(x)$ (cf. Theorem 4.1). The actual transitivity check is then realized by mapping the auxiliary function over the ranges of the relation's domain elements.

```
constdefs
  transp :: "α relation ⇒ bool"
  "transp r ==  transpre r (map snd r)"
```

For simple hangers-on, that is, such hangers-on that are not in the range of the relation, we check the following part of Theorem 4.1

$$\forall\, n \in N.\, r(n) = \bigcup x \in F \cap r(n).\, r(x)$$

which is implemented in the following function on the list representation.

```
constdefs
  nfone :: "[α, α relation] ⇒ bool"
  "nfone n r ==
   (let allrats = concat (map (λ x. (if (fst x) mem (rat n r)
                                    then (snd x) else [])) (fixr r))
    in lset_eq (rat n r) allrats)"
consts
  nf_ho :: "[α relation, α list] ⇒ bool"
primrec
  "nf_ho r [] = True"
  "nf_ho r (n # nl) = (nfone n r) ∧ (nf_ho r nl)"
```

Finally, for the case of non-fixpoint hangers-on that are also in the range of the relation we implement the last part of Theorem 4.1

$$\forall\, f \in F.\, \forall\, y \in N \cap r(f).\, r(y) \subseteq r(f)$$

using an auxiliary function `checkn` that checks $r(y) \subseteq r(f)$ given the relation r, the range $r(f)$ of a fixpoint, and a list of non-fixpoints y.

```
consts
  checkn :: "[α relation, α list, α list] ⇒ bool"
primrec
  "checkn r rf [] = True"
  "checkn r rf (n # nl) = (lsubset (rat n r) rf) ∧ (checkn r rf nl)"
```

Using the auxiliary function `checkn`, we can implement the predicate in two steps: first we implement it for just one $f \in \text{fixp } r$ as the function `nf_rng_one` and then we can apply the latter conjunctively to build the whole predicate as `nf_rng`.

```
constdefs
  nf_rng_one :: "[α list, α relation, α] ⇒ bool"
  "nf_rng_one N r f ==  let ratf = rat f r in
                        checkn r ratf [x:N. x mem ratf]"
consts
  nf_rng :: "[α list, α relation, α list] ⇒ bool"
primrec
  "nf_rng N r [] = True"
  "nf_rng N r (f # fl) = (nf_rng_one N r f) ∧ (nf_rng N r fl)"
```

Now, these predicates can be simply combined in a conjunction.

```
constdefs
  idemp :: "α relation ⇒ bool"
  "idemp r ==  let F = fixl r in
               let N = nfixl r in
               let rF = fixr r in
               (transp rF) ∧ (nf_ho r N) ∧ (nf_rng N r F)"
```

For any relation, we can apply this predicate to its list representation to check for idempotence.

Adequacy

To prove that the predicate actually encodes idempotence, we have to define an implementation relation [9] formally. This implementation relation relates each relation to its list representations independent of the order of the elements in the list or repetitions. However, each list representation has a unique relation associated with it. Hence, it can be seen as a function δ from the implementation to the specification of relations.

```
  δ :: "α relation ⇒ (α × α) set"
```

Consequently, the definition of this function in Isabelle/HOL uses list operations. An auxiliary function δ_x transforms the original datatype of relations into lists of pairs.

```
consts
  δx :: "α relation ⇒ (α × α) list"
primrec
  "δx [] = []"
  "δx (n, nl)# l = (map (λ x. (n, x)) nl) @ (δx  l)"
```

Finally, the function δ uses the function `set` – provided by the list theory, transforming a list into a set – thereby cancelling repetitions and discarding order of elements.

```
  defs  δ r == set(δx r)
```

In order to prove the adequacy of the implementation `idemp`, we can now use δ. As a global assumption, we use the following well-formedness condition on the type of relations.

```
a mem r ⟹ snd a = rat (fst a) r
```

From this assumption, we can infer the following coupling invariant [9] further specifying the implementation relation δ.

```
set(snd a) = δ r "{fst a}
```

To assert the well-formedness assumption, we define on one side a local proof context for the adequacy proofs by a locale [12] and on the other side for the counting application we make sure that the initial enumeration of relations respects this constraint. We show the following auxiliary equivalences associating the fixpoints and non-fixpoints of abstract and concrete representations.

```
fixp (δ r) = set(fixl r)
nfix (δ r) = set(nfixl r)
```

We show the following theorems corresponding to the conjuncts in the characterization of Theorem 4.1.

```
transp(fixr r) ≡ transitive{(x,y). (x,y)∈ δ r ∧ (x,x)∈ δ r}

[| N = nfixl r; F = fixl r |] ⟹
     nf_ho r N ≡ ∀ n ∈ set N.
                    δ r "{n} = ⋃ x ∈ F ∩ δ r "{n}. δ r "{x}

[| N = nfixl r; F = fixl r |] ⟹
   nf_rng N r F ≡ ∀ f ∈ set F. ∀ y ∈ set N ∩ δ r"{f}.
                    δ r "{y} ⊆ δ r "{f}
```

Finally, they can be grouped together and we can prove the adequacy of our Isabelle/HOL implementation of a predicate for idempotence.

```
finite(δ r) ⟹ idemp r ≡ idempotent (δ r)
```

All of the above function definitions are Isabelle/HOL definitions or primitive recursive definitions of functions over the chosen list datatype. They correspond very closely to actual ML function definitions and can be translated one-to-one into ML.[7] The use of the resulting functions for counting idempotents is discussed in the following section.

6. Coding Issues

The proved algorithm for checking a relation for idempotence can now be used to filter out and count idempotents. To this end, we use a simple function suc_rel (see [8]) that enumerates all relations over $\{1, \ldots, n\}$ for a given $n \in \mathbb{N}$. The fact, that we use integers as labelled points, does not impose any restriction as the resulting relations are isomorphic to ones with arbitrary labels. The results for relations up to six points are given in Figure 2. The whole set of idempotents with three points as produced by an automatic transformation written in ML into diagrammatic representation in LaTeX picture is contained in Appendix A for illustration purposes. For up to five elements, the corresponding lists may be found on the author's web page [8].

[7]Actually Isabelle/HOL offers a function codegen that automates the process of exporting fully defined executable formalizations to ML.

Points	Idempotents	All relations
idemp_count 2	11	16
idemp_count 3	123	512
idemp_count 4	2360	65536
idemp_count 5	73023	33554432
idemp_count 6	3494057	68719476736

Figure 2. numbers of idempotent relations

The run time of idemp_count 5 is around ten minutes on a Powerbook G4 with 768 MB RAM, 1,5 GHz. The run time for idemp_count 6 on a similar machine has been 63 days. For 7 the total number of relations is 562949953421312, already getting out of reach for a normal PC.

The obvious question is whether the original algorithm presented in [11] directly constructing idempotents cannot be adapted so that it does not produce repetitions. As it does not count through all billions of relations filtering out the right ones, one would expect it to be faster. In order to investigate this point we implemented such an improved version in ML. It avoids repetitions, but in a naïve way by overwriting repeated range selections. As explained in Section 4, an effective procedure to enumerate range selection without repetitions boils down to enumerating posets. The algorithm implemented in ML is available at [8]. It turns out that for 5 it is twice as fast. However, already for 6 it runs into a livelock. As we construct the idempotents in three passes the intermediate lists of relations have to be held in storage. At some point the run time system is constantly swapping from RAM onto the hard disk not being able to perform another computational step. Even though with greater storage one might get up a few steps, there is a limit to the storage we can use. The clear way out of this dilemma is to avoid the three passes. However, this results in finding an explicit non-repetitive successor function on idempotents which in turn is equivalent to finding a simple formula to explicitly calculate their numbers.

In any case, the problem of generating all idempotents is exponential by nature: a very weak lower bound on the number of idempotents over n labelled points is already 2^n the number of simple fixpoint relations. A little more accurately, when contemplating the numbers for up to six points in Figure 2, we see that for $n > 2$ – at least up to 6 – $2^{5(n-2)+1}$ is outgrown. Hence, the estimated value for 7 is above 2^{26} and for 8 we outgrow $2^{31} \sim 10^{10}$. So, even if our efficient construction would overcome storage complexity problems, for idempotents with more than 10 points we would hit the same time complexity bounds as with the efficient predicate for the same number of idempotents.

7. Conclusions

The main purpose of this paper is to show that Isabelle/HOL can be used to analyze a mathematical entity and infer an algorithm from a theoretical characterization. The development process takes place for the most part in the application domain of finite relations. The refinement of the mathematical characterization of idempotent relations leads to an abstract algorithm for constructing all idempotents over a given carrier set (see Section 4). Unfortunately, it is not possible to refine this abstract algorithm into a precise counting method as in building all possible combinations for each step, repetitions cannot be avoided.

As pointed out in Section 4, it is a known difficult problem to count certain relations, e.g. partial orders.

The refinement of the idempotent characterizations of Theorems 2.1 and 2.2 into the more fine-grained predicate given in Theorem 4.1 enables the construction of an efficient predicate that can be used for filtering idempotents. Thereby, all idempotents over a given finite set and their numbers can be generated in a formally verified way.

The transformation step from the refined Theorem 4.1 to the computable ML predicate on the list representation is an illustration of the classical way of deriving an implementation. Although it is a rather simple example, it shows how general software development theory is at last applied inside Isabelle/HOL to refine a mathematical characterization into its implementation in ML. An interesting follow up project would be to generalize the data refinement application inside Isabelle/HOL into a general framework.

This example shows that the effort to analyze a problem rigorously can lead to known hard problems. It also shows that some proofs that are simple on paper are intricate when done formally at the logical level, like the proof of Lemma 2.2. However, for the analysis of an unknown problem and the development of a new algorithm the effort is justified because there is a gain of general knowledge.

It is feasible to approach a problem with an interactive theorem prover in a straightforward way, only if the problem domain is of a reasonably limited size and well supported by the infrastructure of the prover as in this example of idempotents. In Isabelle/HOL the theories of sets, relations and lists are very well supported. Consequently, the additional theory for idempotents is small.

References

[1] G. Brinkmann and B. D. McKay Posets on up to 16 points. *Order* **19** (2002); no. 2, 147–179. MR 2003e:05002

[2] J. Burghardt, F. Kammüller and J. W. Sanders. On the Antisymmetry of Galois Embeddings. *Information Processing Letters*, **79**:2, Elsevier, June 2001.

[3] A. H. Clifford and G. B. Preston. The Algebraic Theory of Semigroups, volume 1. *Mathematical Surveys*, number 7. American Mathematical Society. 1961.

[4] C. W. Curtis and I. Reiner. *Representation Theory of Finite Groups and Associative Algebras.* Interscience Publishers, John Wiley, 1966.

[5] R. A. Davey and H. A. Priestley. *Introduction to Lattices and Order, Second Edition.* Cambridge University Press, 2002.

[6] N. Dunford and J. T. Schwartz. *Linear Operators. Part I: General Theory.* John Wiley, 1958.

[7] G. Engels *et. al. Quasar Enterprise – Anwendungslandschaften service-orientiert gestalten.* d-punkt.verlag, 2009.

[8] F. Kammüller. http://www.swt.cs.tu-berlin.de/∼flokam.

[9] He Jifeng, C. A. R. Hoare, J. W. Sanders Data Refinement Refined (resume). In *ESOP'86*, pages 187–196. Volume 213 of LNCS, Springer, 1986.

[10] F. Kammüller. *Interactive Theorem Proving in Software Engineering.* (Habilitationsschrift), VDM Müller, 2007.

[11] F. Kammüller and J. W. Sanders. Idempotent Relations in Isabelle/HOL. In *First International Colloquium on Theoretical Aspects of Computing*. Vol. 3407 LNCS, Springer, 2005.

[12] F. Kammüller, M. Wenzel, and L. C. Paulson. Locales - a Sectioning Concept for Isabelle. *Theorem Proving in Higher Order Logics*, 12th International Conference, TPHOLs'99. Vol. 1690 LNCS, Springer, 1999.

[13] L. C. Paulson. *Isabelle: the Next 700 Theorem Provers.* In: P. Odifreddi (editor), Logic and Computer Science, pages 361–386. Academic Press, 1990.

[14] L. C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer LNCS, **828**, 1994.

[15] G. Pfeiffer. Counting Transitive Relations. *Journal of Integer Sequences*, **7**:3, 2004.

[16] M. Schönert et al. *GAP: Groups, Algorithms and Programming.* RWTH Aachen, http://www.gapsystem.org

[17] N. J. A. Sloane *The on-line encyclopedia of integer sequences.* Published electronically at http://www.research.att.com/∼njas/sequences, 2003.

A. Graphical Output of Algorithm for $\mathbb{N}_3$

Here are the graphical representations of the output of the efficient predicate for three elements represented by $\mathbb{N}_3$. The construction algorithm (see [8]) produces the same relations but in different order. The graphical representations are generated by a macro that is written also in ML and generates LaTeX pictures [8](see there also for the graphical output for $\mathbb{N}_4$ and $\mathbb{N}_5$).