

Idempotent Relation in Isabelle/HOL

Bericht-Nr. 2004-04

Florian Kammüller and Jeffrey W. Sanders

ISSN 1436-9915

Idempotent Relations in Isabelle/HOL

Florian Kammüller
Jeffrey W. Sanders

Technische Universität Berlin
Institute for Software Engineering and Theoretical Computer Science
and
University of Oxford, Computing Laboratory

Abstract. A characterization of idempotent relations is presented first as paper style proof, then by its formalization in Isabelle/HOL. The novel characterization gives rise to the construction of idempotent relations by an algorithm. This algorithm is rigorously developed inside Isabelle/HOL using primitive recursive function definitions.

1 Introduction

This paper is inspired by a question that arose in a project on formal programming semantics. There the notion of relations is more commonly used than functions because when looking at specification and refinement nondeterminism of operations can be more naturally expressed using relations.

A relation r being idempotent now means that $r \circ r = r$ where \circ is relational composition, i.e.

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

Intuitively, the idempotence of a relation when interpreted in a functional sense corresponds to the fact that a state is reached where no change occurs any more. When considering composition of operations in specifications, see e.g. [HHS86, Nip86], relations are used to represent operations that are not yet fully defined. The question of idempotence of such operations is an indication that this relation may be an abstract version of the identity function, or that the operation represents some fixpoint of a sequence of operations. In general, as relations are used for the representation of functions it is desirable build a functional calculus for relations, and idempotence is just one special question when considering composition of functional behaviour.

This paper now does several things. First, it introduces a theorem and a proof characterizing idempotent relations which we believe to be original — one can find references for idempotence of relations, but the works usually are concerned with building on the definition rather than exploring it, for example [KR94]. It seems likely, that it has not been discovered yet as the interest in discrete relational composition from a computational point of view is mainly interesting to a rather small group of researchers. One sometimes finds the notion in the context of work related to distributive lattices, e.g. [DP02, Ran53].

Next, this theorem is formalized and proved in Isabelle/HOL [Pau94] which is an interesting application in particular as the proof involves some reasoning about finite sets. The proof is quite tricky and could be interesting to others.

Finally, we devise an algorithm that computes idempotent relations. This algorithm is defined inside Isabelle/HOL using primitive recursive functions. This final aspect is from an engineering point of view probably the most interesting aspect of this work: starting from a theoretical characterization an effective procedure is derived from there and can be proved inside the same framework to be correct. Thereby, we show that Isabelle/HOL, can be used as a program development framework in which correct programs can be derived — and even be tested on the fly, as the primitive recursive functions can be translated one to one into ML.

The paper is organized as follows: the next section introduces the theoretical characterization with proof. After that, Section 3 presents the mechanization of the theorem in Isabelle. Section 4 then starts explaining the actual content of the theoretical characterization by introducing some representative examples, leading on to an informal description of a construction. Section 4.2 is used for

the presentation of the algorithm based on primitive recursive functions. Section 4.3 then shows the properties that have to be shown to prove the correctness of the algorithm. Finally, in Section 5 we draw some conclusions. An appendix contains the full ML version of the algorithm, some output examples from the ML translation of the algorithm, and the full proof of the theorem in Isabelle/HOL.

2 A Characterization of Finite Idempotent Relations

We give a characterization of idempotent finite relations with proof. Finiteness of the relation is a necessary precondition for the theorem. In the remainder of this section we give some corollaries and an example that show that we cannot do better.

For simplicity, let $r(x)$ stand for the relational image $r(\{x\})$ and r^2 for $r \circ r$. In the field of programming semantics the notation $r; r$ is more common to express relational composition. However, we prefer the former symbol here, as it corresponds closer to mathematical notation and is also used in Isabelle as concrete syntax (`0` in ASCII syntax). The characterization is based on elements x of the domain with $x \in r(x)$, that is $(x, x) \in r$. We refer to them as fixpoints of the relation.

The characterization is given by the following theorem.

Theorem 1. *Let r be a finite relation, then*

$$\text{idempotent } r \equiv \forall x \in \mathbf{dom} \ r. \left(\begin{array}{l} r(x) = \bigcup y \in r(y) \cap r(x). \ r(y) \\ \forall y_a \in \mathbf{dom} \ r \cap r(x). \ r(y_a) \subseteq r(x) \end{array} \right)$$

Clearly, idempotence $r \circ r = r$ implies transitivity $r \circ r \subseteq r$. The second conjunct is equivalent to r is transitive, but we prefer to write it this way as it emphasizes the relationship between the ranges of single elements. The first conjunct describes $r \subseteq r \circ r$ and is the major clue to the construction that is to follow.

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

Lemma 1. *Let r be idempotent, then*

$$x \in r(x) \Rightarrow r(x) = \bigcup y \in r(y) \cap r(x). \ r(y)$$

proof: Transitivity gives us $\forall y \in \mathbf{dom} \ r \cap r(x). \ r(y) \subseteq r(x)$. Clearly,

$$\bigcup y \in r(y) \cap r(x). \ r(y) \subseteq \bigcup y \in \mathbf{dom} \ r \cap r(x). \ r(y)$$

and by transitivity of \subseteq the lefthand side is a subset of $r(x)$. Since $x \in r(x)$, $r(x) \subseteq \bigcup y \in r(y) \cap r(x). \ r(y)$, whereby we have equality. \square

Lemma 2. *Let r be finite and idempotent, then*

$$x \in \mathbf{dom} \ r, x \notin r(x) \Rightarrow \forall z \in r(x). \ \exists y \in r(y) \cap r(x). \ z \in r(y)$$

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

$$\exists z \in r(x). \neg \exists y \in r(y) \cap r(x). z \in r(y)$$

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 is finite. \square

Proof of Theorem 1 Now, we are prepared for the proof of the theorem.

Correctness (\Rightarrow): If $x \in r(x)$, just apply Lemma 2. If $x \notin r(x)$, Lemma 1 gives us

$$\bigcup z \in r(x) \subseteq \bigcup y \in r(y) \cap r(x). r(y).$$

Since r is idempotent, it is also transitive. Hence, the righthand side $\subseteq r(x)$. Since the lefthand side is equal to $r(x)$ we have also for $x \notin r(x)$ that

$$r(x) = \bigcup y \in r(y) \cap r(x). r(y).$$

Completeness (\Leftarrow): Let $r(x) = \bigcup y \in r(y) \cap r(x). r(y)$. 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 corresponds to transitivity, we have on the other hand that if $(x, y) \in r^2$ then $(x, y) \in r$. \square

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

Corollary 1. *If $r \neq \emptyset$ is finite and idempotent, then $\exists x. x \in r(x)$.*

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

Corollary 2. *If $r \neq \emptyset$ and $\neg \exists x. x \in r(x)$, then r is infinite.*

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

Example 1. The relation $<: \mathbb{Q} \times \mathbb{Q}$ is idempotent and infinite.

The relation $<$ is obviously transitive, and if $x < y$ there is always an element z between x and y , i.e. $x < z < y$.

3 Mechanical Proof

The theory for idempotent relations contains just one definition for idempotence.

```
idempotent :: (α × α) set => bool
"idempotent r == (r o r = r)"
```

The representation of the theorem in Isabelle is almost like the paper style theorem. However, we had to resolve the self reference in the binder of the union as otherwise the binding would not have worked because it is too self referential. The relational image of a singleton set is denoted by $r''\{x\}$.

```
finite r ==> idempotent r =
  ∀ x ∈ Domain r. r''{x} = ⋃ y: {z. z ∈ r''{z} ∧ z ∈ r''{x}}. r''{y} ∧
  ∀ ya ∈ r''{x} ∩ Domain r. r''{ya} ⊆ r''{x}
```

3.1 Proof of Lemma 1

The proof of Lemma 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 `auto`.

```
[| idempotent r; x: r''{x} |] ==>
  r''{x} = ⋃ y ∈ {z. z ∈ r''{z} ∧ z ∈ r''{x}}. r''{y}
```

3.2 Proof of Lemma 2

This part of the proof of Theorem 1 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. 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.

Core Lemma The core lemma describes that under the assumptions of Lemma 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, there is such a sequence s .

$$\begin{aligned} & [| \text{idempotent } r; x \in \text{Domain } r; x \notin r''\{x\}; z \in r''\{x\}; \\ & \quad \neg (\exists y. y \in r''\{y\} \wedge y \in r''\{x\} \wedge z \in r''\{y\}) \\ & |] \implies \forall n. \exists s :: \text{nat} \Rightarrow \alpha . z = s 0 \wedge \\ & \quad (\forall j. j \leq n \longrightarrow (s j) \in r''\{x\} \wedge \\ & \quad (\forall i. i < j \longrightarrow (s j, s i) \in r \wedge (s j) \neq (s i))) \end{aligned}$$

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.

$$\begin{aligned} & \forall n. \exists s :: \text{nat} \Rightarrow \alpha . z = s 0 \wedge (\forall j. j \leq n \longrightarrow (s j) \in r''\{x\} \wedge \\ & \quad (\forall i. i < j \longrightarrow (s j, s i) \in r \wedge s j \neq s i)) \\ \implies & \forall n. \exists s. \forall j. j \leq n \longrightarrow s j \in r''\{x\} \wedge (\forall i. i < j \longrightarrow s j \neq s i) \end{aligned}$$

The weaker set of properties of the sequence is sufficient to infer the existence of a set whose cardinality is always growing and whose elements are all subsets of a set p – which will ultimately be $r''\{x\}$ in our case.

$$\begin{aligned} & [| \forall n. \exists s :: \text{nat} \Rightarrow \alpha . \\ & \quad (\forall j. j \leq n \longrightarrow (s j) \in p \wedge (\forall i. i < j \longrightarrow (s j) \neq (s i))) |] \\ \implies & \forall n. \exists S. \text{card } S = \text{Suc } n \wedge S \subseteq p \end{aligned}$$

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

$$\forall n. \exists S. \text{card } S = \text{Suc } n \wedge S \subseteq p \implies \neg \text{finite } p$$

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 by producing a contradiction with the assumption of finiteness of the relation.

$$\begin{aligned} & [| \text{finite } r; \text{idempotent } r |] \implies \\ & \quad \forall x \in \text{Domain } r. x \notin r''\{x\} \longrightarrow \\ & \quad (\forall z \in r''\{x\}. \exists y. y \in r''\{y\} \wedge y \in r''\{x\} \wedge z \in r''\{y\}) \end{aligned}$$

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

The proof of Lemma 2 is rather intricate similar to proofs in lattice theory, e.g. [BKS01,DP02]. It would be much easier, if a sequence could be constructed on the outside of the universal quantification over n , i.e. $\exists s. \forall n. \dots$. 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 in some set p — we can construct the sequence of sets represented by the existentially quantified S .

Proof of the Theorem The proof of the correctness, i.e.

$$\begin{aligned} [\text{finite } r; \text{idempotent } r \] \implies \\ (\forall x \in \text{Domain } r. r''\{x\} = (\bigcup y: \{z. z: r''\{z\} \wedge w \in r''\{x\}\}. r''\{y\}) \wedge \\ (\forall ya \in r''\{x\} \cap \text{Domain } r. r''\{ya\} \subseteq r''\{x\})) \end{aligned}$$

simply puts together Lemma 1 and Lemma 2 with transitivity.

For the completeness it is interesting to note that we can infer the property $r \subseteq r \circ r$ from the first conjunct of the characterization alone.

$$\begin{aligned} (\forall x \in \text{Domain } r. r''\{x\} = (\bigcup y: \{z. z \in r''\{z\} \wedge w \in r''\{x\}\}. r''\{y\})) \\ \implies r \subseteq r \circ r \end{aligned}$$

The other conjunct is equivalent to transitivity so we can prove the other inclusion almost automatically. Finally, we put the two parts together to finish the proof.

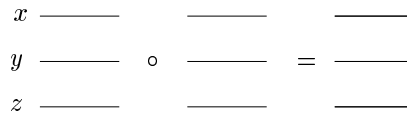
4 Algorithm

The proof of a theorem on paper is the first step. The mechanical proof is nice in that it reassures us that we have not made any mistakes. However, for a result like the one presented in this paper one may actually be content with the paper version — although the reasoning with finite sets is tricky and error prone. However, it really becomes interesting when a theoretical characterization bears the potential to produce constructive solutions to common problems. In this case it is not so obvious what the constructive content of the characterization of idempotent relations may be. Hence, before introducing the construction algorithm, we illustrate how the characterization may be seen as a recipe for constructing idempotent relations. After that, in Section 4.2 we introduce the primitive recursive definitions of the algorithm. Finally in Section 4.3 we outline the major steps in proving the correctness and discuss the solution in Section 4.4.

4.1 Constructing Idempotent Relations

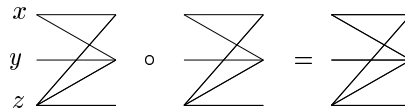
To understand the characteristics of idempotent relations we are considering some representative examples illustrating on them the meaning of the characterization given by Theorem 1. We consider small examples depicting the relations graphically as lines connecting points picking out typical cases that illustrate the scope of idempotence and prepare the ground to find an effective procedure to construct them.

Fixpoints The first and most simple example is that of a relation that only consists of fixpoints. For the three elements x, y, z the relation $r = \{(x, x), (y, y), (z, z)\}$ is depicted next together with a graphical illustration why it is idempotent.



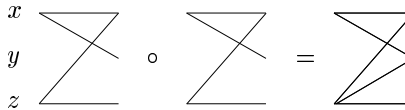
The points that are connected by the relation are all fixpoints. Where in symbols it is hard to decide whether a relation is idempotent, graphically it reduces to following up all paths from the left of the left graph to the right of the right graph.

Range Extensions Starting from a pure fixpoint relation a relation that is constructed by extending the ranges of the fixpoints is in most cases also idempotent. Consider the following example, where starting from the same fixpoints the ranges of x and z are extended by y and $\{x, y\}$ respectively.

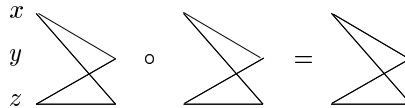


The composition gives the same element, hence this relation is idempotent. This extension conforms to the first conjunct of Theorem 1 for the case of fixpoints, i.e. Lemma 1: the ranges of x and z are defined by just their own ranges.

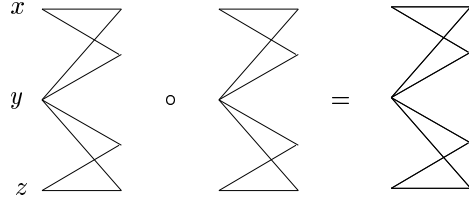
However, even though in most cases the range extension of fixpoints is arbitrary, one has to respect the second conjunct of Theorem 1, transitivity. In the following example we have just the fixpoints x and z , but the range of z contains y , while $r(y) \not\subseteq r(z)$! Hence, the following relation on the left is **not** idempotent (the resulting one on the right is).



Hangers-on As a final step relations that are constructed from fixpoints and admissible range extensions (such extensions that respect transitivity) can be further extended by adding non-fixpoint elements on the domain side. These additional domain extensions of the relations can be considered as *hangers-on*: they hang on to the fixpoint elements, thereby using their “idempotence”. This is an instance of the first conjunct of Theorem 1 (more precisely Lemma 2): for elements of the domain of the relations with $x \notin r(x)$, i.e. non-fixpoints, there must exist fixpoints whose ranges constitute the range of these non-fixpoints. An example is given by the following relation where the non fixpoint x is hanging-on to the fixpoint z having exactly the same range as z .

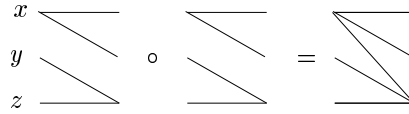


Hangers-on need to copy the entire range of the fixpoints they hang-on to. However, they can have more than one fixpoint they hang on to. The following example with five points, shows how the hanger-on y can actually accumulate the ranges of both fixpoints x and z .



The previous example illustrates why in the characterization of Theorem 1 we need the union over all fixpoints for the case of hangers-on.

For domain extensions we also have to respect transitivity, as is illustrated by the following example.



Although each of the components of the graphs in this relation respects the rules found so far, the combination is not idempotent. The way to avoid this when building domain extensions is to consider only such sets of hangers-on that are not contained in any range of a fixpoint.

Informal Algorithm Now, we want to use the intuition given above to develop an algorithm. The algorithm can be described informally as follows:

- For a given list of fixpoints l build range extensions for each of the fixpoints.
 - The fixpoint is always contained in the range extension. The range extensions are initially all possible subsets of the intended range R of the relation resulting in a list relations that are lists of pairs (l_i, s_i) of fixpoint and range extension, where $l_i \in s_i$.
 - Check the resulting lists of pairs (l_i, s_i) of fixpoint and range extension for transitivity and chuck out the relations not being transitive.
- Build now from the list of range extended relations all combinations d_i of possible domain extensions. A domain extension describes which elements hang-on to a fixpoint, i.e. share its range.
 - A domain extension is any subset of the domain of the prospective relation that does not contain fixpoints from l and that is not in the range s_i of any fixpoint.
 - The elements in the domain extension all have the same range as the fixpoints they are hanging-on to. So it suffices to record the element together with the fixpoints.
- Finally, the derived list of elements of the form

$$[(l_1, s_1, d_1), \dots, (l_n, s_n, d_n)]$$

represents a relation that can be recovered from this list representation as

$$r = \{(x, y). \exists i. (x = l_i \vee x \in d_i) \wedge y \in s_i\}$$

4.2 Primitive Recursive Construction Functions

Instead of defining a nested loop that builds the entire set of all idempotent relations for given domain, range, and fixpoints, we implement the algorithm in a functional manner. Thereby, we can use primitive recursive functions directly as they are provided in Isabelle/HOL.

Initially, we need at various points in the algorithm a function that produces all possible subsets of a set. As we decided to implement the algorithm using lists, the function `sublists` builds sublists of a list rather than sets.

```
sublists :: "[ $\alpha$  list =>  $\alpha$  list list"
```

Assuming that the recursion returns all possible sublists of a list `l` the step of the function build the possible sublists of list `c # l` by concatenating all sublists of `l` with all sublists of `l` where the element `c` has been put in. The recursion finishes by returning the list with the empty sublist.

```
primrec
  sublists_empty: "sublists_fn [] = [[]]"

  sublists_step: "sublists (c # l) =
    (let ll = sublists l in ll @ (map( $\lambda$  x. c # x) ll))"
```

This is an example of a primitive recursive definition as indicated by the keyword `primrec`. We are going to omit the keyword in the following when it is clear from context.

Next we consider the function that given the prospective range of the relation and a list of fixpoints builds the range extensions `range_exs R l`.

```
range_exs :: "[ $\alpha$  list,  $\alpha$  list] => ( $\alpha \times \alpha$  list)list list"
```

This function is rather complex. For each fixpoint `li` in `l` it first builds all sublists of the range `R` omitting `li` to avoid repetitions as `li` has to be in the range extension – and is inserted into each range extension afterwards.

```
range_exs_empty: "range_exs R [] = []"

range_exs_step: "range_exs R (li # ll) =
  (let sl = (sublists [x:R. x  $\neq$  li])
   in (if (ll = []) then map ( $\lambda$  x. [(li,li # x)]) sl
       else combine_re (map ( $\lambda$  x. (li, li#x)) sl) (range_exs R ll))))"
```

The expression `[x:R. x \neq li]` is a filter expression and denotes the list of elements in list `R` that are \neq `li`. In the function body a function `combine_re` is used that will be explained next.

```
combine_re :: "[ $\alpha$  list,  $\alpha$  list list] =>  $\alpha$  list list"
```

The way the algorithm works here is that it builds all combinations of range extensions by working through the list of fixpoints `l` from back to front. Given that the recursion `range_exs R l` returns all possible combinations of range

extensions for the postfix `l1` of the fixpoints `l`, all possible range extensions for the postfix `li #l1` are built by combining each possible range extension for `li`, say `(li, si)` with each of the lists in `range_exts R l` by adding it as first element. This is basically what `combine_re` does apart from making that adding dependent on the check `fp_check`.

```
combine_re_empty: "combine_re [] l = []"

combine_re_step: "combine_re (a # l1) l =
  (map (λ x. if (fp_check a x) then (a # x) else []) l)
  @ (combine_re l1 l)"
```

The function `fp_check` that is used in `combine_re` checks whether the range extension that is created by adding the actual pair to an already range extended postfix, introduces violations of transitivity in which case `combine_re` deletes this element from the combinations.

```
fp_check:: "[(\alpha × \alpha list), (\alpha × \alpha list)list] => bool"
```

This check concerns the property of transitivity, formulated in Theorem 1 as:

$$\forall x \in \mathbf{dom} \ r. \forall y_a \in \mathbf{dom} \ r \cap r(x). \ r(y_a) \subseteq r(x)$$

If an element y_a of the domain is also in the range of another x than y_a 's range has to be contained in the range of x . For fixpoints, which we are considering just now, this property can be slightly simplified to considering such fixpoints that are themselves in the domain of other fixpoints: in the case that a fixpoint l_i is in the range of a fixpoint l_j , the range of l_j has to be a subset of the range of l_i . In the construction of the range extension this criteria applies two ways: the newly added fixpoint l_i could be in the range of an "old" l_j or vice versa. If both are contained in the ranges of each other, clearly their ranges have to be the same¹. The constructor `set` transforms a list into a set.

```
fp_check_empty: "fp_check a [] = True"

fp_check_step: "fp_check (li, si) ((lj, sj) # lr) =
  (if (li mem sj) then (if (lj mem si) then (set si) = (set sj)
    else (set si ⊆ set sj))
  else (if (lj mem si) then (set sj ⊆ set si)
    else fp_check a lr))"
```

It is safe to cancel such elements in the construction of the range extensions that do not pass the function `fp_check`, because the algorithm goes through all possible combinations. Hence, just a bit further down the line the current range extension `(li, si)` is going to be combined with a slight variation that fits.

¹ To make the definition more readable, we use pattern matching of pairs in the argument position of the following primitive recursive definition, but this is actually not supported in Isabelle/HOL.

The next step in the algorithm is to construct the domain extensions. Here, the procedure is structurally very similar to the range extension. That is, we are going to use again a combination function to build all possible combinations of domain extensions this time using a simple version of the function `combine_re` called simply `combine`.

```
combine_empty: "combine [] l = []"
```

```
combine_step: "combine (a # l1) l = (map (Cons a) l) @ (combine l1 l)"
```

Now, a domain extension of a fixpoint on a given domain for the relation and list of fixpoints can be applied to a list of range extensions, that is to a list of elements of type $\alpha \times \alpha \text{list}$. It returns a list of lists of triples, that are the range extensions extended by an additional list as third element, representing the elements of the domain that share the range of the fixpoints.

```
domain_exs :: "[ $\alpha$  list,  $\alpha$  list, ( $\alpha \times \alpha$  list)list]
=> ( $\alpha \times \alpha$  list  $\times$   $\alpha$  list)list list"
```

For a list D representing the relations domain and a list l of fixpoints, a domain extension of a fixpoint li is a subset of the domain that has neither elements of l nor elements in the range extensions si of any li . This constraint is realized by using again the filter construct in $[x:D. (\neg(x \text{ mem } l)) \wedge (\neg(x \text{ mem } fpre))]$. The argument $fpre$ is the list containing all range extensions of that relation. In one step, that is for one range extension pair (li, si) the function `domain_exs_fn` builds all possible sublists of the result of that filtering and combines all those with (li, si) to build the domain extended point (li, si, di) .

```
domain_exs_empty_fn: "domain_exs_fn D l fpre [] = []"
```

```
domain_exs_step: "domain_exs_fn D l fpre ((l,si) # lr) =
let sl = (sublists [x:D. (\neg(x mem l)) \wedge (\neg(x mem fpre))])
in if (lr = []) then map (\x. [(li,si,x)]) sl
else combine (map(\x. (li,si,x)) sl)(domain_exs_fn D l fpre lr)"
```

The function `domain_exs_fn` can now be used to define the actual function for domain extension as a constant by building the parameter $fpre$ of all range extensions of fixpoints and then applying the former function.

```
domain_exs D l rl == domain_exs_fn D l (concat (map snd rl)) rl
```

The meaning of a range extension point (li, si) is that li is a fixpoint and has range si containing li . The domain extension element di that is now added as the third element to those points represents all hangers-on of li . That is, the elements of di are all non fixpoints that have in their range all the elements of the range of li , i.e. the elements of si ². For efficiency of representation and simplicity it is advisable to chose this representation rather than copying the ranges for each hanger-on.

² Note, that in general a hanger-on can have more elements in its range, as it can simultaneously hang-on to other fixpoints.

Keeping this explanation of the domain extensions in mind the following function should be easily comprehensible. This function, called `build_rel` is necessary to close the loop of developing the algorithm from the ideas contained in Theorem 1 back to where it started by giving a procedure to translate the list representation calculated from the previous set of functions into a relation again.

```
build_rel :: "( $\alpha$   $\times$  ( $\alpha$ )list  $\times$  ( $\alpha$ )list)list => ( $\alpha$   $\times$   $\alpha$ )set"
```

For a relation represented as a list of the described triple type, the function `build_rel` now builds a set of pairs: a pair (x,y) is in the relation if x is the fixpoint we are considering and y is in the current range extension `si` or x is a hanger-on from the domain extension. In the latter case, as hangers-ons have the same range as their fixpoints, y has to be in `si` too.

```
build_rel_empty: "build_rel [] = "
```

```
build_rel_step: "build_rel ((li,si,di) # ll) =
  {(x,y). (x = li  $\vee$  x mem di)  $\wedge$  y mem si}  $\cup$  (build_rel ll)"
```

As we will see in the next section, the functions presented in this section can be simply applied to produce idempotent relations. As they are primitive recursive function definitions, and the syntax chosen in Isabelle for these functions is very similar to ML, it is possible to translate them one-to-one into ML. As an example consider the output in the Appendix.

The initial question of this research was to produce all idempotent relations for a given domain and range. We can build up the corresponding list in our representation defining a constant.

```
list_of_all_idempotents :: "[ $\alpha$  list,  $\alpha$  list]
=> ( $\alpha$   $\times$   $\alpha$  list  $\times$   $\alpha$  list)list list"
```

The definition of that constant just applies the functions for range extension and domain extension to the list of all possible sets of fixpoints in the domain D and building all possible range subsets of R as starting points for the process. As we produce list of lists of relations in each step it is necessary to flatten those lists using `concat` when putting it together.

```
list_of_all_idempotents D R ==
  (let all_ranges = sublists R
   in concat(map ( $\lambda$  l. concat (map (domain_exs D l)
    (concat (map (range_exs R) all_ranges))))(sublists D)))
```

4.3 Properties

Now, the function `build_rel` closes the loop of development. With its help we can express the correctness of the algorithm quite concisely as

```
[| finite r; unique l; set l = {x. (x,x)  $\in$  r}; set D = Domain r; set R = Range r |]
 $\implies$  idempotent r = ( $\exists$  re_fn. re_fn mem (range_exs R l)  $\wedge$ 
  ( $\exists$  de_fn. de_fn mem (domain_exs D l re_fn)  $\wedge$  (r = build_rel de_fn)))
```

That is, for a given set l of fixpoints, a given domain and range for the relation, the property of idempotence of relation is equivalent to the existence of a range extension and a domain extension that are built by the corresponding functions applied consecutively and which represent the relation. The premise `unique l` is a predicate ensuring that the list `l` does not contain repetitions. This restriction is necessary to exclude lists with repetitions as input to the algorithm.

Otherwise, the proof is a rather straightforward unfolding of definitions of function definitions using inductions over lists to show that the properties of the list representation actually translate into the properties of the characterization.

From this property we can derive the more general one concerning the set of all idempotent relations which corresponds to completeness.

```
{r. finite r & Domain r = set R & Range r = set R & idempotent r} =
  set(map build_rel(list_of_all_idempotents D R))
```

4.4 Discussion

In this section we have introduced via characteristic examples an algorithm for the construction of finite idempotent relations. From there we have developed a set of functions that compute them.

The way the function `range_exts` deletes half finished elements when they do not respect transitivity is the main problem with the algorithm. It would be nicer to exclude unsuitable lists at the stage when all sublists s_i for the range extensions are generated — that is before `range_exts` is called. However, the problem arises only later — that is, when the single sets s_i are generated we cannot anticipate how they are going to be combined. More precisely, they are going to be combined in all possible ways. As the transitivity property is a property that concerns each element of the relation it can only be judged when putting the relation together. Surely, we could have started building admissible sets of sets s_i before adding the fixpoints l_i to the range extensions but that is basically what we are doing just at the same time when the corresponding fixpoints are added.

Another point worth mentioning is that our function is not efficient for another reason. When the domain extensions are built there are cases where repetitions may arise: when two fixpoints x and z have exactly the same range, there are two ways a hanger-on can achieve his domain extension, by hanging on to x and by hanging on to z . Hence, our algorithm produces two identical relations. An example can be seen in the Appendix, where in the second output sample the domain extensions `[(1, [1, 2], []), (2, [2, 1], [3])]` and `[(1, [1, 2], [3]), (2, [2, 1], [])]` will be mapped to the same relation by `build_rel`. Another frequent repetition occurring in the produced lists is the empty list. This is a remainder of the deletion of non admissible lists during the range extension and vanishes when concatenating.

5 Conclusion

We have presented a result characterizing finite idempotent relations. This result has been proved, first on paper and then its formalization and proof in Isabelle/HOL have been characterized. A construction algorithm has been derived from there and the correctness results for this algorithm in the same framework have been sketched.

Independent of the contribution that the result and the algorithm may represent, this work is a case study that illustrates that a logic like HOL in the implementation given by Isabelle/HOL is a framework that may well be used to develop programs rigorously. It is although small, a convincing demonstration that Isabelle/HOL is suited as a formal method in itself, as has been proposed before [NW98]. Rather than embedding a formal method into a generic prover like Isabelle and then using the resulting instance as a framework for developing software in a rigorous way — which is a rather complex endeavor — the infrastructure provided by Isabelle/HOL seems to be sufficient to develop algorithms on simple datastructures.

Clearly, it would be wrong to deduce from the current case study that HOL even with a mature support like Isabelle is a substitute for advanced formal methods providing support for abstract high level concepts like refinement, object orientation, and modularity, but it may be an encouragement to extend the existing features, like data types and recursive function definitions, in order to make the rigorous development in Isabelle/HOL more powerful. The methodical shortcut that is given by a manner of application like the present, is crucial for the complete specification and proof of critical software.

In particular the close relation to the programming language ML has proved as very helpful in this project. All the functions presented here could be tested easily by copying them almost one-to-one to the ML system.

Acknowledgements

We are grateful to Dr. Ingrid Rewitzky, University of Cape Town, Department of Mathematics and Applied Mathematics because the idea for this research has been initiated when both authors were visiting her group. Furthermore, this work has been supported by the program ARC funded by the British Council and DAAD.

References

- [BKS01] J. Burghardt, F. Kammüller and J. W. Sanders. *On the Antisymmetry of Galois Embeddings*. Information Processing Letters, **79**:2, Elsevier, June 2001.
- [DP02] R. A. Davey and H. A. Priestley. *Introduction to Lattices and Order, Second Edition*. Cambridge University Press, 2002.
- [HHS86] J. F. He and C. A. R. Hoare and J. W. Sanders. *Data refinement refined*. In European Symposium on Programming, (ESOP 86), Springer LNCS, **213**, 1986.

- [KR94] E.Kani and M. Rosen. *Idempotent relations among arithmetic invariants attached to number fields and algebraic varieties*. J. Number Theory, **46**, 1994.
- [NW98] W. Naraschewski and M. Wenzel. *Object-Oriented Verification Based on Record Subtyping in Higher-Order Logic*, In Proceedings of TPHOLs 98, Springer LNCS, **1479**, 1998.
- [Nip86] Tobias Nipkow. *Non-Deterministic Data Types: Models and Implementations*. Acta Informatica, **22**:629-661, 1986.
- [Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, Springer LNCS, **828**, 1994.
- [Ran53] G. Raney. *A subdirect-union representation for completely distributive complete lattices*, Proc. Amer. Math. Soc, **4**, 1953.

Appendix A

This first appendix contains the actual ML implementation of the algorithm for the enumeration of idempotent relations. It is interesting only as a reference and to show how close the Isabelle representation is related to the actual executable program. Even more so, the syntax in Isabelle is in some parts nicer because mathematical notation may be used and in ML we had to implement some logical connectives.

```

fun sublists [] l = l
| sublists (c::l) ll = (sublists l ll) @ (map (fn x => c :: x)
      (sublists l ll));

fun Sublists l = sublists l [[]];

fun Sublists [] = [[]]
| Sublists (c::l) = (Sublists l) @ (map (fn x => c :: x)
      (Sublists l));

fun combine [] l = []
| combine (a :: l1) l = (map (cons a) l) @ (combine l1 l);

fun mand x y = if x then y else false;

fun mor x y = if x then true else y;

fun domain_exs_fn D l fpre [] = []
| domain_exs_fn D l fpre ((li,si)::lr) =
  (let val sl = Sublists (filter
      (fn x => mand (not(x mem l)) (not(x mem fpre))) D)
    in (if (lr = []) then map (fn x => [(li,si,x)]) sl
      else combine (map (fn x => (li, si, x)) sl) (domain_exs_fn D l fpre lr))
    end);

fun domain_exs D l rl = domain_exs_fn D l (flat (map snd rl)) rl;

fun msubset l1 l2 = (filter (fn x => not(x mem l2)) l1) = [];

```

```

fun fp_check (li,si) [] = true
  | fp_check (li,si) ((lj,sj)::lr) =
    if (li mem sj) then
      (if (lj mem si) then mand (si subset sj) (sj subset si)
        else msubset si sj)
    else (if (lj mem si) then msubset sj si
          else fp_check (li,si) lr);

fun combine_re [] l = []
  | combine_re (a :: l1) l =
    (map (fn x => if (fp_check a x) then (cons a x) else []) l)
    @ (combine_re l1 l);

fun range_exs R [] = []
  | range_exs R (li :: l) =
    (let val sl = Sublists (filter (fn x => not(x = li)) R)
     in (if (l = []) then map (fn x => [(li, li :: x)]) sl
        else combine_re (map (fn x => (li, li::x)) sl) (range_exs R l)) end);

```

Appendix B

We present a short output sample produced by the ML translations of the algorithm for idempotent relations. The final translation into relations is not implemented, as the corresponding datatypes in ML were not available.

```

range_exs [1,2,3] [1,2];

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

domain_exs [1,2,3,4] [1,2] [(1, [1]), (2, [2])];

[[ (1, [1], []), (2, [2], []) ], [ (1, [1], []), (2, [2], [4]) ],
  [ (1, [1], []), (2, [2], [3]) ], [ (1, [1], []), (2, [2], [3, 4]) ],
  [ (1, [1], [4]), (2, [2], []) ], [ (1, [1], [4]), (2, [2], [4]) ],
  [ (1, [1], [4]), (2, [2], [3]) ], [ (1, [1], [4]), (2, [2], [3, 4]) ],
  [ (1, [1], [3]), (2, [2], []) ], [ (1, [1], [3]), (2, [2], [4]) ],
  [ (1, [1], [3]), (2, [2], [3]) ], [ (1, [1], [3]), (2, [2], [3, 4]) ],
  [ (1, [1], [3, 4]), (2, [2], []) ], [ (1, [1], [3, 4]), (2, [2], [4]) ],
  [ (1, [1], [3, 4]), (2, [2], [3]) ],
  [ (1, [1], [3, 4]), (2, [2], [3, 4]) ] ]

domain_exs [1,2,3] [1,2] [(1, [1, 2]), (2, [2, 1])];

```

```

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

domain_exs [1,2,3] [1,3] [ (1, [1]), (3, [3, 2]) ];

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

```

Appendix C

This appendix contains the theory and proof files for Theorem 1 in Isabelle version 2002.

```

(* Title: idempotent/Idempotent.thy
   Author: Florian KammueLLer flokam@cs.tu-berlin.de *)

theory Idempotent = Main:

constdefs
  idempotent :: "('a * 'a) set => bool"
  "idempotent r == (r 0 r = r)"

end

(* Idempotent.ML *)
(* Lemmata *)
Goal "[| x <= Suc n; x ~= Suc n |] ==> x <= n";
be contrapos_np 1;
auto();
val Suc_leq_lemma = result();

Goal "[| (j :: nat) <= n; j ~= n |] ==> j < n";
by (arith_tac 1);
val le_neq_imp_less = result();

Goal "P (SOME x. P x) = (? x. P x)";
auto();
br someI 1;
ba 1;
val hilbert_eq = result();

Goal "x <= n ==> x < Suc n";
auto();
val Suc_le = result();

Goal "finite p ==> ? n. card p = n";
auto();
val finite_imp_ex_card = result();

```

```

Goalw [Range_def,Domain_def,converse_def] "finite r ==> finite (Range r)";
by (forw_inst_tac [("h","% (x,y). y")] finite_imageI 1);
auto();
by (subgoal_tac "(% (x, y). y) ' r) = x. EX y. (y, x) : r" 1);
auto();
val finite_rel_imp_range = result();

Goal "[| finite r; x: Domain r |] ==> finite (r 'x)";
br finite_subset 1;
be finite_rel_imp_range 2;
auto();
val finite_rel_imp_single = result();

Goal "(~(! x: A. P x)) = (? x: A. ~(P x))";
auto();
val notBall_eq_Bexnot = result();

(* idempotent lemmata *)
Goalw [thm "idempotent_def",trans_def] "idempotent r ==> trans r";
auto();
val idempotent_imp_trans = result();

Goalw [thm "idempotent_def"]
  "[| idempotent r; (x,z): r |] ==> (x,z): r 0 r";
be ssubst 1;
ba 1;
val idempotent_double = result();

Goal "[| idempotent r; (x,z): r |] ==> ? y. (x,y): r & (y,z): r";
bd idempotent_double 1;
ba 1;
by (rewrite_goals_tac [rel_comp_def]);
bd CollectD 1;
by (Asm_full_simp_tac 1);
val idempotent_ex_middle = result();

Goalw [Image_def] "(x,z): r = (z: r 'x)";
auto();
val image_single = result();

Goalw [Image_def] "(x,x): r = (x: r 'x)";
auto();
val fixpoint_eq = result();

(* core lemma for Lemma2 *)
Goal "[| 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:: nat) --> 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)";
  by (forw_inst_tac [("x","x"),("z","s n")] idempotent_ex_middle 1);
  by (SELECT_GOAL (fold_goals_tac [image_single RS eq_reflection]) 1);
  by (SELECT_GOAL Auto_tac 1);
  be exE 1;
  by (res_inst_tac [("x","y")] exI 1);
  br conjI 1;
  be conjE 1;
  be (image_single RS subst) 1;
  br allI 1;
  br impI 1;
  br conjI 1;
  (* 1. (y, s j) : r *)
  by (case_tac "j = n" 1);
  by (rotate_tac ~1 1);
  be ssubst 1;
  be conjunct2 1;
  (* ok first case *)
  by (subgoal_tac "j < n" 1);
  be le_neq_imp_less 2;
  ba 2;
  by (res_inst_tac [("b","s n")] transD 1);
  be idempotent_imp_trans 1;
  be conjunct2 1;
  by (SELECT_GOAL Auto_tac 1);
  (* remaining: y ~ = s j *)
  br notI 1;
  by (subgoal_tac "(y, s j): r" 1);
  (* second same as before -> Lemma? *)
  by (case_tac "j = n" 2);
  by (rotate_tac ~1 2);
  be ssubst 2;
  be conjunct2 2;
  by (subgoal_tac "j < n" 2);
  be le_neq_imp_less 3;
  ba 3;
  by (res_inst_tac [("b","s n")] transD 2);
  be idempotent_imp_trans 2;
  be conjunct2 2;
  by (SELECT_GOAL Auto_tac 2);
  (* end potential Lemma *)
  by (subgoal_tac "s j : r 'x & (s j, s j): r & (s j, z): r" 1);
  by (SELECT_GOAL Auto_tac 2);
  by (rotate_tac 4 1);
  be notE 1;
  by (res_inst_tac [("x","s j")] exI 1);
  by (fold_goals_tac [image_single RS eq_reflection]);
  by (REPEAT (etac conjE 1));
  by (REPEAT (etac conjI 1));

```

```

ba 1;
val core_lemma = result();

Goal "[| idempotent r; x : Domain r; x ~: r 'x; z : r 'x;
      ~ (EX y. y : r 'y & y : r 'x & z : r 'y) |]
  ==> !n. ? s:: nat => 'a . z = s 0 &
  (! j. j <= n --> (s j) : r 'x &
    (! i. i < j --> (s j, s i) : r & (s j) ~= (s i)))";
br allI 1;
br nat.induct 1;
by (res_inst_tac [("x", "% m. z")] exI 1);
(* I. start *)
br conjI 1;
br refl 1;
br allI 1;
br impI 1;
br conjI 1;
br conjI 1;
ba 1;
br allI 1;
br impI 1;
br FalseE 1;
by (SELECT_GOAL Auto_tac 1);
(* rest *)
be exE 1;
bd core_lemma 1;
ba 1;
ba 1;
ba 1;
ba 1;
be conjunct1 1;
be conjunct2 1;
be exE 1;
by (res_inst_tac [("x", "% m. if m <= na then s m else y")] exI 1);
auto();
val Lemma2_step1 = result();

(* weakening to simplify *)
Goal "!n. ? s:: nat => 'a . z = s 0 &
  (! j. j <= n --> (s j) : r 'x &
    (! i. i < j --> (s j, s i) : r & (s j) ~= (s i)))
  ==> !n. ? s:: nat => 'a .
  (! j. j <= n --> (s j) : r 'x &
    (! i. i < j --> (s j) ~= (s i)))";
br allI 1;
be allE 1;
be exE 1;
by (res_inst_tac [("x", "s")] exI 1);
auto();
val simp_assumptions = result();

```

```

Goal "[!n. ? s:: nat => 'a .
  (! j. j <= n --> (s j) : p & (! i. i < j --> (s j) ~= (s i))) ]
  ==> !n. ? S. card S = Suc n & S <= p";
br allI 1;
be allE 1;
be exE 1;
by (res_inst_tac [("x", "x. ? j. j <= n & (x = s j)"]) exI 1);
br conjI 1;
auto();
by (subgoal_tac "ALL j. j <= n --> (ALL i. i < j --> s j ~= s i)" 1);
by (Asm_full_simp_tac 2);
by (thin_tac "ALL j. j <= n --> s j : p & (ALL i. i < j --> s j ~= s i)" 1);
br mp 1;
ba 2;
by (res_inst_tac [("n", "n")] nat.induct 1);
auto();
(* solves I.A. *)
by (subgoal_tac "finite x. EX j. j <= na & x = s j" 1);
(* problems with set equality *)
by (subgoal_tac "x. EX j. j <= Suc na & x = s j = insert (s (Suc na))
  x. EX j. j <= na & x = s j" 1);
(* direction 1 *)
br equalityI 2;
br subsetI 2;
be CollectE 2;
be exE 2;
br insertCI 2;
be conjE 2;
be contrapos_np 2;
br CollectI 2;
(* need Lemma *)
by (subgoal_tac "j ~= Suc na" 2);
by (res_inst_tac [("x", "j")] exI 2);
br conjI 2;
ba 3;
be Suc_leq_lemma 2;
ba 2;
by (SELECT_GOAL Auto_tac 2);
(* other direction *)
br subsetI 2;
be insertE 2;
br CollectI 2;
by (res_inst_tac [("x", "Suc na")] exI 2);
by (Asm_full_simp_tac 2);
by (Asm_full_simp_tac 2);
be exE 2;
by (res_inst_tac [("x", "j")] exI 2);
by (SELECT_GOAL Auto_tac 2);
(* done *)
by (rotate_tac ~1 1);

```

```

be ssubst 1;
by (res_inst_tac [("t","Suc na")] subst 1);
ba 1;
br card_insert_disjoint 1;
by (res_inst_tac [("t","card x. EX j. j <= na & x = s j")] ssubst 2);
ba 2;
ba 1;
(* alright: 2: finite ..., 1: s (Suc na) ~: ... *)
by (subgoal_tac "! x: x. EX j. j <= na & x = s j. s (Suc na) ~= x" 1);
by (SELECT_GOAL Auto_tac 1);
br ballI 1;
by (SELECT_GOAL Auto_tac 1);
(* ok *)
(* lemma necessary
  [| ALL j. j <= n --> (ALL i. i < j --> s i ~= s j);
    ALL j. j <= Suc na --> (ALL i. i < j --> s i ~= s j);
    card x. EX j. j <= na & x = s j = Suc na |]
  ==> finite x. EX j. j <= na & x = s j
*)
by (res_inst_tac [("f","% y. @ m. m <= na & y = s m")] finite_imageD 1);
br inj_onI 2;
bd CollectD 2;
bd CollectD 2;
by (SELECT_GOAL (fold_goals_tac [hilbert_eq RS eq_reflection]) 2);
by (Asm_full_simp_tac 2);
br bounded_nat_set_is_finite 1;
auto();
(* remains: (SOME m. m <= na & s j = s m) < ?n71 n s y na *)
by (subgoal_tac "(SOME m. m <= na & s j = s m) <= na & s j = s (SOME m. m <=
  na & s j = s m)" 1);
by (res_inst_tac [("n", "na"), ("x", "(SOME m. m <= na & s j = s m)")]
  Suc_le 1);
be conjunct1 1;
by (subgoal_tac "? m. m <= na & s j = s m" 1);
by (SELECT_GOAL (fold_goals_tac [hilbert_eq RS eq_reflection]) 1);
ba 1;
by (res_inst_tac [("x","j")] exI 1);
be conjI 1;
br refl 1;
val Lemma2_step2 = result();

Goal "!n. ? S. card S = Suc n & S <= p ==> ~finite p";
br notI 1;
by (forward_tac [finite_imp_ex_card] 1);
be exE 1;
by (dres_inst_tac [("x","n")] spec 1);
be exE 1;
by (etac conjE 1);
by (forward_tac [finite_subset] 1);
ba 1;

```

```

by (forward_tac [card_mono] 1);
ba 1;
auto();
val Lemma2_step3 = result();

Goal "[| !n. ? s:: nat => 'a .
  (! j. j <= n --> (s j) : p & (! i. i < j --> (s j) ~= (s i))) |]
  ==> ~finite p";
br Lemma2_step3 1;
br Lemma2_step2 1;
ba 1;
val Lemma2_step23 = result();

(* Lemma2 *)
Goal "[| finite r; idempotent r |] ==>
  ! x: Domain r. x ~: r `` x -->
    (! z: r `` x. ? y. y: r `` y & y: r `` x & z: r `` y)";
br ballI 1;
br impI 1;
br notnotD 1;
br notI 1;
by (rewrite_goals_tac [notBall_eq_Bexnot RS eq_reflection]);
be bexE 1;
(* ok starting point :
  1. !!x z.
    [| finite r; idempotent r; x : Domain r; x ~: r `` x; z : r `` x;
      ~ (EX y. y : r `` y & y : r `` x & z : r `` y) |]
    ==> False
*)
bd finite_rel_imp_single 1;
ba 1;
by (subgoal_tac "~ finite (r `` x)" 1);
by (Fast_tac 1);
br Lemma2_step23 1;
br simp_assumptions 1;
be Lemma2_step1 1;
ba 1;
ba 1;
ba 1;
ba 1;
val Lemma2 = result();

Goal "[| idempotent r; x: r `` x |] ==>
  r `` x = (UN y: w. w: r `` w & w: r `` x. r `` y)";
auto();
br transD 1;
ba 3;
ba 2;
be idempotent_imp_trans 1;
val Lemma3 = result();

```

```

Goal "[| finite r; idempotent r |] ==>
  (! x: Domain r. r `` x =
    (UN y: w. w: r `` w & w: r `` x. r `` y));
br ballI 1;
by (case_tac "x : r `` x" 1);
be Lemma3 1;
ba 1;
by (forward_tac [Lemma2] 1);
ba 1;
(* ### *)
auto();
bd idempotent_imp_trans 1;
be transD 1;
ba 2;
ba 1;
val Theorem1_corr_a = result();

Goalw [thm "idempotent_def"] "[| idempotent r; x: Domain r |] ==>
  (! ya: r `` x Int Domain r. r `` ya <= r `` x);
auto();
val Theorem1_corr_b = result();

Goal "[| finite r; idempotent r |] ==>
  (! x: Domain r. r `` x =
    (UN y: w. w: r `` w & w: r `` x. r `` y) &
  (! ya: r `` x Int Domain r. r `` ya <= r `` x));
by (forward_tac [Theorem1_corr_a] 1);
ba 1;
br ballI 1;
br conjI 1;
be bspec 1;
ba 1;
by (forward_tac [Theorem1_corr_b] 1);
ba 1;
ba 1;
val Theorem1_correctness = result();

Goalw [thm "idempotent_def"]
  "[| r 0 r <= r; r <= r 0 r |] ==> idempotent r";
be equalityI 1;
ba 1;
val idempotentI = result();

Goal "[| (! x: Domain r. r `` x =
  (UN y: w. w: r `` w & w: r `` x. r `` y)) |]
  ==> r <= r 0 r";
by (rewrite_goals_tac [rel_comp_def]);
by (SELECT_GOAL Auto_tac 1);
val Theorem1backw_a = result();

```

```

Goal "[| (! x: Domain r. r `` x =
        (UN y: w. w: r `` w & w: r `` x. r `` y) &
        (! ya: r ``x Int Domain r. r ``ya <= r ``x))|]
  ==> idempotent r";
br idempotentI 1;
br Theorem1backw_a 2;
br ballI 2;
br conjunct1 2;
be bspec 2;
ba 2;
by (SELECT_GOAL Auto_tac 1);
val Theorem1_completeness = result();

Goal "finite r ==>
  idempotent r = (! x: Domain r. r `` x =
    (UN y: w. w: r `` w & w: r `` x. r `` y) &
    (! ya: r ``x Int Domain r. r ``ya <= r ``x))";
br iffI 1;
be Theorem1_correctness 1;
ba 1;
be Theorem1_completeness 1;
val Theorem1 = result();

```