

# Automatic Data Refinement

Peter Lammich

April 16, 2025

### Abstract

We present the Autoref tool for Isabelle/HOL, which automatically refines algorithms specified over abstract concepts like maps and sets to algorithms over concrete implementations like red-black-trees, and produces a refinement theorem. It is based on ideas borrowed from relational parametricity due to Reynolds and Wadler. The tool allows for rapid prototyping of verified, executable algorithms. Moreover, it can be configured to fine-tune the result to the user's needs. Our tool is able to automatically instantiate generic algorithms, which greatly simplifies the implementation of executable data structures.

This AFP-entry provides the basic tool, which is then used by the Refinement and Collection Framework to provide automatic data refinement for the nondeterminism monad and various collection data-structures.

# Contents

<b>1</b>	<b>Parametricity Solver</b>	<b>5</b>
1.1	Relators . . . . .	5
1.1.1	Basic Definitions . . . . .	5
1.1.2	Basic HOL Relators . . . . .	6
1.1.3	Automation . . . . .	12
1.1.4	Setup . . . . .	17
1.1.5	Invariant and Abstraction . . . . .	21
1.1.6	Miscellaneous . . . . .	22
1.1.7	Conversion between Predicate and Set Based Relators . . . . .	22
1.1.8	More Properties . . . . .	24
1.2	Basic Parametricity Reasoning . . . . .	25
1.2.1	Auxiliary Lemmas . . . . .	25
1.2.2	ML-Setup . . . . .	25
1.2.3	Convenience Tools . . . . .	31
1.3	Parametricity Theorems for HOL . . . . .	32
1.3.1	Sets . . . . .	32
1.3.2	Standard HOL Constructs . . . . .	34
1.3.3	Functions . . . . .	34
1.3.4	Boolean . . . . .	34
1.3.5	Nat . . . . .	35
1.3.6	Int . . . . .	35
1.3.7	Product . . . . .	36
1.3.8	Option . . . . .	37
1.3.9	Sum . . . . .	38
1.3.10	List . . . . .	39
<b>2</b>	<b>Automatic Refinement</b>	<b>45</b>
2.1	Automatic Refinement Tool . . . . .	45
2.1.1	Standard setup . . . . .	45
2.1.2	Tools . . . . .	46
2.1.3	Advanced Debugging . . . . .	47
2.2	Standard HOL Bindings . . . . .	49
2.2.1	Structural Expansion . . . . .	49

2.2.2	Booleans	50
2.2.3	Standard Type Classes	51
2.2.4	Functional Combinators	52
2.2.5	Unit	53
2.2.6	Nat	53
2.2.7	Int	54
2.2.8	Product	54
2.2.9	Option	55
2.2.10	Sum-Types	56
2.2.11	List	57
2.2.12	Examples	61
2.3	Entry Point for the Automatic Refinement Tool	62
2.3.1	Convenience	62

# Chapter 1

## Parametricity Solver

### 1.1 Relators

```
theory Relators
imports ../Lib/Refine-Lib
begin
```

We define the concept of relators. The relation between a concrete type and an abstract type is expressed by a relation of type  $('c \times 'a)$  *set*. For each composed type, say  $'a$  *list*, we can define a *relator*, that takes as argument a relation for the element type, and returns a relation for the list type. For most datatypes, there exists a *natural relator*. For algebraic datatypes, this is the relator that preserves the structure of the datatype, and changes the components. For example,  $list-rel::('c \times 'a)$  *set*  $\Rightarrow ('c$  *list*  $\times 'a$  *list)* *set* is the natural relator for lists.

However, relators can also be used to change the representation, and thus relate an implementation with an abstract type. For example, the relator  $list-set-rel::('c \times 'a)$  *set*  $\Rightarrow ('c$  *list*  $\times 'a$  *set)* *set* relates lists with the set of their elements.

In this theory, we define some basic notions for relators, and then define natural relators for all HOL-types, including the function type. For each relator, we also show a single-valuedness property, and initialize a solver for single-valued properties.

#### 1.1.1 Basic Definitions

For smoother handling of relator unification, we require relator arguments to be applied by a special operator, such that we avoid higher-order unification problems. We try to set up some syntax to make this more transparent, and give relators a type-like prefix-syntax.

```
definition relAPP
  :: (('c1  $\times$  'a1) set  $\Rightarrow$  -)  $\Rightarrow$  ('c1  $\times$  'a1) set  $\Rightarrow$  -
```

where  $relAPP\ f\ x \equiv f\ x$

**syntax** *-rel-APP* ::  $args \Rightarrow 'a \Rightarrow 'b \langle \langle - \rangle \rightarrow [0,900] 900 \rangle$

**syntax-consts** *-rel-APP* == *relAPP*

**translations**

$\langle x, xs \rangle R == \langle xs \rangle (CONST\ relAPP\ R\ x)$   
 $\langle x \rangle R == CONST\ relAPP\ R\ x$

**ML** <

```
structure Refine-Relators-Thms = struct
  structure rel-comb-def-rules = Named-Thms (
    val name = @{binding refine-rel-defs}
    val description = Refinement Framework: ^
      Relator definitions
  );
end
>
```

**setup** *Refine-Relators-Thms.rel-comb-def-rules.setup*

### 1.1.2 Basic HOL Relators

#### Function

**definition** *fun-rel where*

*fun-rel-def-internal*:  $fun-rel\ A\ B \equiv \{ (f, f'). \forall (a, a') \in A. (f\ a, f'\ a') \in B \}$

**abbreviation** *fun-rel-syn* (**infixr**  $\langle \rightarrow \rangle$  60) **where**  $A \rightarrow B \equiv \langle A, B \rangle fun-rel$

**lemma** *fun-rel-def[refine-rel-defs]*:

$A \rightarrow B \equiv \{ (f, f'). \forall (a, a') \in A. (f\ a, f'\ a') \in B \}$

**by** (*simp add: relAPP-def fun-rel-def-internal*)

**lemma** *fun-relI[intro!]*:  $\llbracket \bigwedge a\ a'. (a, a') \in A \implies (f\ a, f'\ a') \in B \rrbracket \implies (f, f') \in A \rightarrow B$

**by** (*auto simp: fun-rel-def*)

**lemma** *fun-relD*:

**shows**  $((f, f') \in (A \rightarrow B)) \implies$

$(\bigwedge x\ x'. \llbracket (x, x') \in A \rrbracket \implies (f\ x, f'\ x') \in B)$

**apply rule**

**by** (*auto simp: fun-rel-def*)

**lemma** *fun-relD1*:

**assumes**  $(f, f') \in Ra \rightarrow Rr$

**assumes**  $f\ x = r$

**shows**  $\forall x'. (x, x') \in Ra \longrightarrow (r, f'\ x') \in Rr$

**using** *assms* **by** (*auto simp: fun-rel-def*)

**lemma** *fun-relD2*:  
**assumes**  $(f, f') \in Ra \rightarrow Rr$   
**assumes**  $f' x' = r'$   
**shows**  $\forall x. (x, x') \in Ra \longrightarrow (f x, r') \in Rr$   
**using** *assms* **by** (*auto simp: fun-rel-def*)

**lemma** *fun-relE1*:  
**assumes**  $(f, f') \in Id \rightarrow Rv$   
**assumes**  $t' = f' x$   
**shows**  $(f x, t') \in Rv$  **using** *assms*  
**by** (*auto elim: fun-relD*)

**lemma** *fun-relE2*:  
**assumes**  $(f, f') \in Id \rightarrow Rv$   
**assumes**  $t = f x$   
**shows**  $(t, f' x) \in Rv$  **using** *assms*  
**by** (*auto elim: fun-relD*)

## Terminal Types

**abbreviation** *unit-rel* ::  $(unit \times unit)$  set **where** *unit-rel* == *Id*

**abbreviation** *nat-rel*  $\equiv Id :: (nat \times -)$  set

**abbreviation** *int-rel*  $\equiv Id :: (int \times -)$  set

**abbreviation** *bool-rel*  $\equiv Id :: (bool \times -)$  set

## Product

**definition** *prod-rel* **where**  
*prod-rel-def-internal*: *prod-rel* *R1* *R2*  
 $\equiv \{ ((a, b), (a', b')) . (a, a') \in R1 \wedge (b, b') \in R2 \}$

**abbreviation** *prod-rel-syn* (**infixr**  $\langle \times_r \rangle$  70) **where**  $a \times_r b \equiv \langle a, b \rangle$  *prod-rel*

**lemma** *prod-rel-def*[*refine-rel-defs*]:  
 $((R1, R2) \text{prod-rel}) \equiv \{ ((a, b), (a', b')) . (a, a') \in R1 \wedge (b, b') \in R2 \}$   
**by** (*simp add: prod-rel-def-internal relAPP-def*)

**lemma** *prod-relI*:  $\llbracket (a, a') \in R1; (b, b') \in R2 \rrbracket \implies ((a, b), (a', b')) \in \langle R1, R2 \rangle \text{prod-rel}$   
**by** (*auto simp: prod-rel-def*)

**lemma** *prod-relE*:  
**assumes**  $(p, p') \in \langle R1, R2 \rangle \text{prod-rel}$   
**obtains**  $a \ b \ a' \ b'$  **where**  $p = (a, b)$  **and**  $p' = (a', b')$   
**and**  $(a, a') \in R1$  **and**  $(b, b') \in R2$   
**using** *assms*  
**by** (*auto simp: prod-rel-def*)

**lemma** *prod-rel-simp*[*simp*]:  
 $((a, b), (a', b')) \in \langle R1, R2 \rangle \text{prod-rel} \longleftrightarrow (a, a') \in R1 \wedge (b, b') \in R2$   
**by** (*auto intro: prod-relI elim: prod-relE*)

**lemma** *in-Domain-prod-rel-iff*[*iff*]:  $(a,b) \in \text{Domain } (A \times_r B) \iff a \in \text{Domain } A \wedge b \in \text{Domain } B$   
**by** (*auto simp: prod-rel-def*)

**lemma** *prod-rel-comp*:  $(A \times_r B) \circ (C \times_r D) = (A \circ C) \times_r (B \circ D)$   
**unfolding** *prod-rel-def*  
**by** *auto*

## Option

**definition** *option-rel* **where**

*option-rel-def-internal*:

*option-rel*  $R \equiv \{ (Some\ a, Some\ a') \mid a\ a'.\ (a,a') \in R \} \cup \{ (None, None) \}$

**lemma** *option-rel-def*[*refine-rel-defs*]:

$\langle R \rangle \text{option-rel} \equiv \{ (Some\ a, Some\ a') \mid a\ a'.\ (a,a') \in R \} \cup \{ (None, None) \}$

**by** (*simp add: option-rel-def-internal relAPP-def*)

**lemma** *option-relI*:

$(None, None) \in \langle R \rangle \text{option-rel}$

$\llbracket (a,a') \in R \rrbracket \implies (Some\ a, Some\ a') \in \langle R \rangle \text{option-rel}$

**by** (*auto simp: option-rel-def*)

**lemma** *option-relE*:

**assumes**  $(x,x') \in \langle R \rangle \text{option-rel}$

**obtains**  $x = None$  **and**  $x' = None$

**|**  $a\ a'$  **where**  $x = Some\ a$  **and**  $x' = Some\ a'$  **and**  $(a,a') \in R$

**using** *assms* **by** (*auto simp: option-rel-def*)

**lemma** *option-rel-simp*[*simp*]:

$(None, a) \in \langle R \rangle \text{option-rel} \iff a = None$

$(c, None) \in \langle R \rangle \text{option-rel} \iff c = None$

$(Some\ x, Some\ y) \in \langle R \rangle \text{option-rel} \iff (x,y) \in R$

**by** (*auto intro: option-relI elim: option-relE*)

## Sum

**definition** *sum-rel* **where** *sum-rel-def-internal*:

*sum-rel*  $Rl\ Rr$

$\equiv \{ (Inl\ a, Inl\ a') \mid a\ a'.\ (a,a') \in Rl \} \cup$   
 $\{ (Inr\ a, Inr\ a') \mid a\ a'.\ (a,a') \in Rr \}$

**lemma** *sum-rel-def*[*refine-rel-defs*]:

$\langle Rl, Rr \rangle \text{sum-rel} \equiv$

$\{ (Inl\ a, Inl\ a') \mid a\ a'.\ (a,a') \in Rl \} \cup$

$\{ (Inr\ a, Inr\ a') \mid a\ a'.\ (a,a') \in Rr \}$

**by** (*simp add: sum-rel-def-internal relAPP-def*)

**lemma** *sum-rel-simp*[*simp*]:



$\bigwedge a a'. (Inl\ a, Inl\ a') \in \langle Rl, Rr \rangle sum\text{-}rel \iff (a, a') \in Rl$   
 $\bigwedge a a'. (Inr\ a, Inr\ a') \in \langle Rl, Rr \rangle sum\text{-}rel \iff (a, a') \in Rr$   
 $\bigwedge a a'. (Inl\ a, Inr\ a') \notin \langle Rl, Rr \rangle sum\text{-}rel$   
 $\bigwedge a a'. (Inr\ a, Inl\ a') \notin \langle Rl, Rr \rangle sum\text{-}rel$   
**unfolding** *sum-rel-def* **by** *auto*

**lemma** *sum-relI*:

$(l, l') \in Rl \implies (Inl\ l, Inl\ l') \in \langle Rl, Rr \rangle sum\text{-}rel$   
 $(r, r') \in Rr \implies (Inr\ r, Inr\ r') \in \langle Rl, Rr \rangle sum\text{-}rel$   
**by** *simp-all*

**lemma** *sum-relE*:

**assumes**  $(x, x') \in \langle Rl, Rr \rangle sum\text{-}rel$   
**obtains**  
 $l\ l'$  **where**  $x = Inl\ l$  **and**  $x' = Inl\ l'$  **and**  $(l, l') \in Rl$   
 $|$   $r\ r'$  **where**  $x = Inr\ r$  **and**  $x' = Inr\ r'$  **and**  $(r, r') \in Rr$   
**using** *assms* **by** (*auto simp: sum-rel-def*)

## Lists

**definition** *list-rel* **where** *list-rel-def-internal*:

*list-rel*  $R \equiv \{(l, l').\ list\text{-}all2\ (\lambda x\ x'. (x, x') \in R)\ l\ l'\}$

**lemma** *list-rel-def[refine-rel-defs]*:

$\langle R \rangle list\text{-}rel \equiv \{(l, l').\ list\text{-}all2\ (\lambda x\ x'. (x, x') \in R)\ l\ l'\}$   
**by** (*simp add: list-rel-def-internal relAPP-def*)

**lemma** *list-rel-induct[induct set, consumes 1, case-names Nil Cons]*:

**assumes**  $(l, l') \in \langle R \rangle list\text{-}rel$   
**assumes**  $P \ \square \ \square$   
**assumes**  $\bigwedge x\ x'\ l\ l'. \ \llbracket (x, x') \in R; (l, l') \in \langle R \rangle list\text{-}rel; P\ l\ l' \rrbracket$   
 $\implies P\ (x\ \#l)\ (x'\ \#l')$   
**shows**  $P\ l\ l'$   
**using** *assms* **unfolding** *list-rel-def*  
**apply** *simp*  
**by** (*rule list-all2-induct*)

**lemma** *list-rel-eq-listrel*:  $list\text{-}rel = listrel$

**apply** (*rule ext*)

**apply** *safe*

**proof** *goal-cases*

**case** (1  $x\ a\ b$ ) **thus** *?case*

**unfolding** *list-rel-def-internal*

**apply** *simp*

**apply** (*induct a b rule: list-all2-induct*)

**apply** (*auto intro: listrel.intros*)

**done**

**next**

**case** 2 **thus** *?case*

```

  apply (induct)
  apply (auto simp: list-rel-def-internal)
  done
qed

```

**lemma** *list-relI*:

```

( $[], []$ )  $\in$   $\langle R \rangle$  list-rel
 $\llbracket (x, x') \in R; (l, l') \in \langle R \rangle$  list-rel  $\rrbracket \implies (x \# l, x' \# l') \in \langle R \rangle$  list-rel
by (auto simp: list-rel-def)

```

**lemma** *list-rel-simp[simp]*:

```

( $[], l'$ )  $\in$   $\langle R \rangle$  list-rel  $\longleftrightarrow l' = []$ 
( $l, []$ )  $\in$   $\langle R \rangle$  list-rel  $\longleftrightarrow l = []$ 
( $[], []$ )  $\in$   $\langle R \rangle$  list-rel
( $x \# l, x' \# l'$ )  $\in$   $\langle R \rangle$  list-rel  $\longleftrightarrow (x, x') \in R \wedge (l, l') \in \langle R \rangle$  list-rel
by (auto simp: list-rel-def)

```

**lemma** *list-relE1*:

```

assumes ( $l, []$ )  $\in$   $\langle R \rangle$  list-rel obtains  $l = []$  using assms by auto

```

**lemma** *list-relE2*:

```

assumes ( $[], l$ )  $\in$   $\langle R \rangle$  list-rel obtains  $l = []$  using assms by auto

```

**lemma** *list-relE3*:

```

assumes ( $x \# xs, l'$ )  $\in$   $\langle R \rangle$  list-rel obtains  $x'$   $xs'$  where
 $l' = x' \# xs'$  and  $(x, x') \in R$  and  $(xs, xs') \in \langle R \rangle$  list-rel
using assms
apply (cases  $l'$ )
apply auto
done

```

**lemma** *list-relE4*:

```

assumes ( $l, x' \# xs'$ )  $\in$   $\langle R \rangle$  list-rel obtains  $x$   $xs$  where
 $l = x \# xs$  and  $(x, x') \in R$  and  $(xs, xs') \in \langle R \rangle$  list-rel
using assms
apply (cases  $l$ )
apply auto
done

```

**lemmas** *list-relE* = *list-relE1 list-relE2 list-relE3 list-relE4*

**lemma** *list-rel-imp-same-length*:

```

( $l, l'$ )  $\in$   $\langle R \rangle$  list-rel  $\implies$  length  $l =$  length  $l'$ 
unfolding list-rel-eq-listrel relAPP-def
by (rule listrel-eq-len)

```

**lemma** *list-rel-split-right-iff*:

```

( $x \# xs, l$ )  $\in$   $\langle R \rangle$  list-rel  $\longleftrightarrow$  ( $\exists y$   $ys. l = y \# ys \wedge (x, y) \in R \wedge (xs, ys) \in \langle R \rangle$  list-rel)
by (cases  $l$ ) auto

```

**lemma** *list-rel-split-left-iff*:

$(l, y \# ys) \in \langle R \rangle \text{list-rel} \iff (\exists x \text{ xs. } l = x \# \text{xs} \wedge (x, y) \in R \wedge (\text{xs}, ys) \in \langle R \rangle \text{list-rel})$   
**by** (*cases l*) *auto*

## Sets

Pointwise refinement: The abstract set is the image of the concrete set, and the concrete set only contains elements that have an abstract counterpart

**definition** *set-rel where*

*set-rel-def-internal*:

$\text{set-rel } R \equiv \{(A, B). (\forall x \in A. \exists y \in B. (x, y) \in R) \wedge (\forall y \in B. \exists x \in A. (x, y) \in R)\}$

**term** *set-rel*

**lemma** *set-rel-def[refine-rel-defs]*:

$\langle R \rangle \text{set-rel} \equiv \{(A, B). (\forall x \in A. \exists y \in B. (x, y) \in R) \wedge (\forall y \in B. \exists x \in A. (x, y) \in R)\}$   
**by** (*simp add: set-rel-def-internal relAPP-def*)

**lemma** *set-rel-alt*:  $\langle R \rangle \text{set-rel} = \{(A, B). A \subseteq R^{-1} \text{``} B \wedge B \subseteq R \text{``} A\}$

**unfolding** *set-rel-def* **by** *auto*

**lemma** *set-relI[intro?]*:

**assumes**  $\bigwedge x. x \in A \implies \exists y \in B. (x, y) \in R$

**assumes**  $\bigwedge y. y \in B \implies \exists x \in A. (x, y) \in R$

**shows**  $(A, B) \in \langle R \rangle \text{set-rel}$

**using** *assms* **unfolding** *set-rel-def* **by** *blast*

Original definition of *set-rel* in refinement framework. Abandoned in favour of more symmetric definition above:

**definition** *old-set-rel where* *old-set-rel-def-internal*:

$\text{old-set-rel } R \equiv \{(S, S'). S' = R \text{``} S \wedge S \subseteq \text{Domain } R\}$

**lemma** *old-set-rel-def[refine-rel-defs]*:

$\langle R \rangle \text{old-set-rel} \equiv \{(S, S'). S' = R \text{``} S \wedge S \subseteq \text{Domain } R\}$

**by** (*simp add: old-set-rel-def-internal relAPP-def*)

Old definition coincides with new definition for single-valued element relations. This is probably the reason why the old definition worked for most applications.

**lemma** *old-set-rel-sv-eg*: *single-valued*  $R \implies \langle R \rangle \text{old-set-rel} = \langle R \rangle \text{set-rel}$

**unfolding** *set-rel-def* *old-set-rel-def* *single-valued-def*

**by** *blast*

**lemma** *set-rel-simp[simp]*:

$(\{\}, \{\}) \in \langle R \rangle \text{set-rel}$

by (auto simp: set-rel-def)

**lemma** *set-rel-empty-iff*[simp]:  
 $(\{\}, y) \in \langle A \rangle \text{set-rel} \longleftrightarrow y = \{\}$   
 $(x, \{\}) \in \langle A \rangle \text{set-rel} \longleftrightarrow x = \{\}$   
 by (auto simp: set-rel-def; fastforce)+

**lemma** *set-relD1*:  $(s, s') \in \langle R \rangle \text{set-rel} \implies x \in s \implies \exists x' \in s'. (x, x') \in R$   
 unfolding set-rel-def by blast

**lemma** *set-relD2*:  $(s, s') \in \langle R \rangle \text{set-rel} \implies x' \in s' \implies \exists x \in s. (x, x') \in R$   
 unfolding set-rel-def by blast

**lemma** *set-relE1*[consumes 2]:  
 assumes  $(s, s') \in \langle R \rangle \text{set-rel}$   $x \in s$   
 obtains  $x'$  where  $x' \in s'$   $(x, x') \in R$   
 using set-relD1[OF assms] ..

**lemma** *set-relE2*[consumes 2]:  
 assumes  $(s, s') \in \langle R \rangle \text{set-rel}$   $x' \in s'$   
 obtains  $x$  where  $x \in s$   $(x, x') \in R$   
 using set-relD2[OF assms] ..

### 1.1.3 Automation

#### A solver for relator properties

**lemma** *relprop-triggers*:  
 $\bigwedge R. \text{single-valued } R \implies \text{single-valued } R$   
 $\bigwedge R. R = \text{Id} \implies R = \text{Id}$   
 $\bigwedge R. R = \text{Id} \implies \text{Id} = R$   
 $\bigwedge R. \text{Range } R = \text{UNIV} \implies \text{Range } R = \text{UNIV}$   
 $\bigwedge R. \text{Range } R = \text{UNIV} \implies \text{UNIV} = \text{Range } R$   
 $\bigwedge R R'. R \subseteq R' \implies R \subseteq R'$   
 by auto

**ML** <  
 structure relator-props = Named-Thms (  
 val name = @{binding relator-props}  
 val description = Additional relator properties  
 )  
  
 structure solve-relator-props = Named-Thms (  
 val name = @{binding solve-relator-props}  
 val description = Relator properties that solve goal  
 )  
 >  
**setup** relator-props.setup

**setup** *solve-relator-props.setup*

**declaration** <

```

Tagged-Solver.declare-solver
  @{thms relprop-triggers}
  @{binding relator-props-solver}
  Additional relator properties solver
  (fn ctxt => (REPEAT-ALL-NEW (CHANGED o (
    match-tac ctxt (solve-relator-props.get ctxt) ORELSE'
    match-tac ctxt (relator-props.get ctxt)
  ))))

```

**declaration** <

```

Tagged-Solver.declare-solver
  []
  @{binding force-relator-props-solver}
  Additional relator properties solver (instantiate schematics)
  (fn ctxt => (REPEAT-ALL-NEW (CHANGED o (
    resolve-tac ctxt (solve-relator-props.get ctxt) ORELSE'
    match-tac ctxt (relator-props.get ctxt)
  ))))

```

**lemma**

```

relprop-id-orient[relator-props]:  $R=Id \implies Id=R$  and
relprop-eq-refl[solve-relator-props]:  $t = t$ 
by auto

```

**lemma**

```

relprop-UNIV-orient[relator-props]:  $R=UNIV \implies UNIV=R$ 
by auto

```

## ML-Level utilities

**ML** <

```

signature RELATORS = sig
  val mk-relT: typ * typ -> typ
  val dest-relT: typ -> typ * typ

  val mk-relAPP: term -> term -> term
  val list-relAPP: term list -> term -> term
  val strip-relAPP: term -> term list * term
  val mk-fun-rel: term -> term -> term

  val list-rel: term list -> term -> term

  val rel-absT: term -> typ
  val rel-concT: term -> typ

```

```

val mk-prodrel: term * term -> term
val is-prodrel: term -> bool
val dest-prodrel: term -> term * term

val strip-prodrel-left: term -> term list
val list-prodrel-left: term list -> term

val declare-natural-relator:
  (string*string) -> Context.generic -> Context.generic
val remove-natural-relator: string -> Context.generic -> Context.generic
val natural-relator-of: Proof.context -> string -> string option

val mk-natural-relator: Proof.context -> term list -> string -> term option

val setup: theory -> theory
end

structure Relators :RELATORS = struct
  val mk-relT = HOLogic.mk-prodT #> HOLogic.mk-setT

  fun dest-relT (Type (@{type-name set},[Type (@{type-name prod},[cT,aT])))
    = (cT,aT)
    | dest-relT ty = raise TYPE (dest-relT,[ty],[])

  fun mk-relAPP x f = let
    val xT = fastype-of x
    val fT = fastype-of f
    val rT = range-type fT
  in
    Const (@{const-name relAPP},fT-->xT-->rT)$f$x
  end

  val list-relAPP = fold mk-relAPP

  fun strip-relAPP R = let
    fun aux @{\mpat (?R)?S} l = aux S (R::l)
      | aux R l = (l,R)
  in aux R [] end

  val rel-absT = fastype-of #> HOLogic.dest-setT #> HOLogic.dest-prodT #>
snd
  val rel-concT = fastype-of #> HOLogic.dest-setT #> HOLogic.dest-prodT #>
fst

  fun mk-fun-rel r1 r2 = let
    val (r1T,r2T) = (fastype-of r1,fastype-of r2)
    val (c1T,a1T) = dest-relT r1T

```

```

    val (c2T,a2T) = dest-relT r2T
    val (cT,aT) = (c1T ---> c2T, a1T ---> a2T)
    val rT = mk-relT (cT,aT)
  in
    list-relAPP [r1,r2] (Const (@{const-name fun-rel},r1T--->r2T--->rT))
  end

  val list-rel = fold-rev mk-fun-rel

  fun mk-prodrel (A,B) = @{mk-term ?A ×r ?B}
  fun is-prodrel @{mpat - ×r -} = true | is-prodrel - = false
  fun dest-prodrel @{mpat ?A ×r ?B} = (A,B) | dest-prodrel t = raise TERM(dest-prodrel,[t])

  fun strip-prodrel-left @{mpat ?A ×r ?B} = strip-prodrel-left A @ [B]
    | strip-prodrel-left @{mpat (typs) unit-rel} = []
    | strip-prodrel-left R = [R]

  val list-prodrel-left = Refine-Util.list-binop-left @{term unit-rel} mk-prodrel

  structure natural-relators = Generic-Data (
    type T = string Symtab.table
    val empty = Symtab.empty
    val merge = Symtab.join (fn - => fn (-,cn) => cn)
  )

  fun declare-natural-relator tcp =
    natural-relators.map (Symbtab.update tcp)

  fun remove-natural-relator tname =
    natural-relators.map (Symbtab.delete-safe tname)

  fun natural-relator-of ctxt =
    Symtab.lookup (natural-relators.get (Context.Proof ctxt))

  (* [R1,..,Rn] T is mapped to ⟨R1,..,Rn⟩ Trel *)
  fun mk-natural-relator ctxt args Tname =
    case natural-relator-of ctxt Tname of
      NONE => NONE
    | SOME Cname => SOME let
        val argsT = map fastype-of args
        val (cTs, aTs) = map dest-relT argsT |> split-list
        val aT = Type (Tname,aTs)
        val cT = Type (Tname,cTs)
        val rT = mk-relT (cT,aT)
      in
        list-relAPP args (Const (Cname,argsT---->rT))
      end
    end

  fun

```

```

natural-relator-from-term (t as Const (name,T)) = let
  fun err msg = raise TERM (msg,[t])

  val (argTs,bodyT) = strip-type T
  val (conTs,absTs) = argTs |> map (HOLogic.dest-setT #> HOLogic.dest-prodT)
|> split-list
  val (bconT,babsT) = bodyT |> HOLogic.dest-setT |> HOLogic.dest-prodT
  val (Tcon,bconTs) = dest-Type bconT
  val (Tcon',babsTs) = dest-Type babsT

  val - = Tcon = Tcon' orelse err Type constructors do not match
  val - = conTs = bconTs orelse err Concrete types do not match
  val - = absTs = babsTs orelse err Abstract types do not match

in
  (Tcon,name)
end
| natural-relator-from-term t =
  raise TERM (Expected constant,[t]) (* TODO: Localize this! *)

local
  fun decl-natrel-aux t context = let
    fun warn msg = let
      val tP =
        Context.cases Syntax.pretty-term-global Syntax.pretty-term
          context t
      val m = Pretty.block [
        Pretty.str Ignoring invalid natural-relator declaration:,
        Pretty.brk 1,
        Pretty.str msg,
        Pretty.brk 1,
        tP
      ] |> Pretty.string-of
      val - = warning m
    in context end
  in
    try <declare-natural-relator (natural-relator-from-term t) context
      catch TERM (msg,-) => warn msg
      | exn => warn >
    end
  in
    val natural-relator-attr = Scan.repeat1 Args.term >> (fn ts =>
      Thm.declaration-attribute ( fn - => fold decl-natrel-aux ts)
    )
  end

val setup = I
#> Attrib.setup

```



```

    @{binding natural-relator} natural-relator-attr Declare natural relator

  end
>

```

```

setup Relators.setup

```

### 1.1.4 Setup

#### Natural Relators

```

declare [[natural-relator
  unit-rel int-rel nat-rel bool-rel
  fun-rel prod-rel option-rel sum-rel list-rel
]]

```

```

ML-val <
  Relators.mk-natural-relator
    @{context}
    [@{term Ra::('c×'a) set},@{term <Rb>option-rel}]
    @{type-name prod}
  |> the
  |> Thm.ctrm-of @{context}
;
  Relators.mk-fun-rel @{term <Id>option-rel} @{term <Id>list-rel}
  |> Thm.ctrm-of @{context}
>

```

#### Additional Properties

```

lemmas [relator-props] =
  single-valued-Id
  subset-refl
  refl

```

```

lemma eq-UNIV-iff: S=UNIV  $\longleftrightarrow$  ( $\forall x. x \in S$ ) by auto

```

```

lemma fun-rel-sv[relator-props]:
  assumes RAN: Range Ra = UNIV
  assumes SV: single-valued Rv
  shows single-valued (Ra  $\rightarrow$  Rv)
proof (intro single-valuedI ext impI allI)
  fix f g h x'
  assume R1: (f,g)  $\in$  Ra  $\rightarrow$  Rv
  and R2: (f,h)  $\in$  Ra  $\rightarrow$  Rv

```

```

from RAN obtain x where AR: (x,x')  $\in$  Ra by auto

```

**from**  $\text{fun-relD}[OF\ R1\ AR]$  **have**  $(f\ x, g\ x') \in Rv$  .  
**moreover from**  $\text{fun-relD}[OF\ R2\ AR]$  **have**  $(f\ x, h\ x') \in Rv$  .  
**ultimately show**  $g\ x' = h\ x'$  **using**  $SV$  **by**  $(\text{auto dest: single-valuedD})$   
**qed**

**lemmas**  $[\text{relator-props}] = \text{Range-Id}$

**lemma**  $\text{fun-rel-id}[\text{relator-props}]$ :  $\llbracket R1=Id; R2=Id \rrbracket \implies R1 \rightarrow R2 = Id$   
**by**  $(\text{auto simp: fun-rel-def})$

**lemma**  $\text{fun-rel-id-simp}[\text{simp}]$ :  $Id \rightarrow Id = Id$  **by**  $\text{tagged-solver}$

**lemma**  $\text{fun-rel-comp-dist}[\text{relator-props}]$ :  
 $(R1 \rightarrow R2) \ O \ (R3 \rightarrow R4) \subseteq ((R1 \ O \ R3) \rightarrow (R2 \ O \ R4))$   
**by**  $(\text{auto simp: fun-rel-def})$

**lemma**  $\text{fun-rel-mono}[\text{relator-props}]$ :  $\llbracket R1 \subseteq R2; R3 \subseteq R4 \rrbracket \implies R2 \rightarrow R3 \subseteq R1 \rightarrow R4$   
**by**  $(\text{force simp: fun-rel-def})$

**lemma**  $\text{prod-rel-sv}[\text{relator-props}]$ :  
 $\llbracket \text{single-valued } R1; \text{single-valued } R2 \rrbracket \implies \text{single-valued } (\langle R1, R2 \rangle \text{prod-rel})$   
**by**  $(\text{auto intro: single-valuedI dest: single-valuedD simp: prod-rel-def})$

**lemma**  $\text{prod-rel-id}[\text{relator-props}]$ :  $\llbracket R1=Id; R2=Id \rrbracket \implies \langle R1, R2 \rangle \text{prod-rel} = Id$   
**by**  $(\text{auto simp: prod-rel-def})$

**lemma**  $\text{prod-rel-id-simp}[\text{simp}]$ :  $\langle Id, Id \rangle \text{prod-rel} = Id$  **by**  $\text{tagged-solver}$

**lemma**  $\text{prod-rel-mono}[\text{relator-props}]$ :  
 $\llbracket R2 \subseteq R1; R3 \subseteq R4 \rrbracket \implies \langle R2, R3 \rangle \text{prod-rel} \subseteq \langle R1, R4 \rangle \text{prod-rel}$   
**by**  $(\text{auto simp: prod-rel-def})$

**lemma**  $\text{prod-rel-range}[\text{relator-props}]$ :  $\llbracket \text{Range } Ra=UNIV; \text{Range } Rb=UNIV \rrbracket$   
 $\implies \text{Range } (\langle Ra, Rb \rangle \text{prod-rel}) = UNIV$   
**apply**  $(\text{auto simp: prod-rel-def})$   
**by**  $(\text{metis Range-iff UNIV-I})+$

**lemma**  $\text{option-rel-sv}[\text{relator-props}]$ :  
 $\llbracket \text{single-valued } R \rrbracket \implies \text{single-valued } (\langle R \rangle \text{option-rel})$   
**by**  $(\text{auto intro: single-valuedI dest: single-valuedD simp: option-rel-def})$

**lemma**  $\text{option-rel-id}[\text{relator-props}]$ :  
 $R=Id \implies \langle R \rangle \text{option-rel} = Id$  **by**  $(\text{auto simp: option-rel-def})$

**lemma**  $\text{option-rel-id-simp}[\text{simp}]$ :  $\langle Id \rangle \text{option-rel} = Id$  **by**  $\text{tagged-solver}$

**lemma**  $\text{option-rel-mono}[\text{relator-props}]$ :  $R \subseteq R' \implies \langle R \rangle \text{option-rel} \subseteq \langle R' \rangle \text{option-rel}$   
**by**  $(\text{auto simp: option-rel-def})$

**lemma** *option-rel-range*:  $\text{Range } R = \text{UNIV} \implies \text{Range } (\langle R \rangle \text{option-rel}) = \text{UNIV}$   
**apply** (*auto simp: option-rel-def Range-iff*)  
**by** (*metis Range-iff UNIV-I option.exhaust*)

**lemma** *option-rel-inter*[*simp*]:  $\langle R1 \cap R2 \rangle \text{option-rel} = \langle R1 \rangle \text{option-rel} \cap \langle R2 \rangle \text{option-rel}$   
**by** (*auto simp: option-rel-def*)

**lemma** *option-rel-constraint*[*simp*]:  
 $(x,x) \in \langle \text{UNIV} \times C \rangle \text{option-rel} \longleftrightarrow (\forall v. x = \text{Some } v \longrightarrow v \in C)$   
**by** (*auto simp: option-rel-def*)

**lemma** *sum-rel-sv*[*relator-props*]:  
 $\llbracket \text{single-valued } Rl; \text{single-valued } Rr \rrbracket \implies \text{single-valued } (\langle Rl, Rr \rangle \text{sum-rel})$   
**by** (*auto intro: single-valuedI dest: single-valuedD simp: sum-rel-def*)

**lemma** *sum-rel-id*[*relator-props*]:  $\llbracket Rl = \text{Id}; Rr = \text{Id} \rrbracket \implies \langle Rl, Rr \rangle \text{sum-rel} = \text{Id}$   
**apply** (*auto elim: sum-relE*)  
**apply** (*case-tac b*)  
**apply** *simp-all*  
**done**

**lemma** *sum-rel-id-simp*[*simp*]:  $\langle \text{Id}, \text{Id} \rangle \text{sum-rel} = \text{Id}$  **by** *tagged-solver*

**lemma** *sum-rel-mono*[*relator-props*]:  
 $\llbracket Rl \subseteq Rl'; Rr \subseteq Rr' \rrbracket \implies \langle Rl, Rr \rangle \text{sum-rel} \subseteq \langle Rl', Rr' \rangle \text{sum-rel}$   
**by** (*auto simp: sum-rel-def*)

**lemma** *sum-rel-range*[*relator-props*]:  
 $\llbracket \text{Range } Rl = \text{UNIV}; \text{Range } Rr = \text{UNIV} \rrbracket \implies \text{Range } (\langle Rl, Rr \rangle \text{sum-rel}) = \text{UNIV}$   
**apply** (*auto simp: sum-rel-def Range-iff*)  
**by** (*metis Range-iff UNIV-I sumE*)

**lemma** *list-rel-sv-iff*:  
 $\text{single-valued } (\langle R \rangle \text{list-rel}) \longleftrightarrow \text{single-valued } R$   
**apply** (*intro iffI[rotated] single-valuedI allI impI*)  
**apply** (*clarsimp simp: list-rel-def*)

**proof** –

**fix**  $x\ y\ z$

**assume**  $SV$ : *single-valued*  $R$

**assume** *list-all2*  $(\lambda x\ x'. (x, x') \in R)\ x\ y$  **and**

*list-all2*  $(\lambda x\ x'. (x, x') \in R)\ x\ z$

**thus**  $y = z$

**apply** (*induct arbitrary: z rule: list-all2-induct*)

**apply** *simp*

**apply** (*case-tac z*)

**apply** *force*

**apply** (*force intro: single-valuedD[OF SV]*)

```

    done
  next
  fix x y z
  assume SV: single-valued ( $\langle R \rangle$ list-rel)
  assume (x,y) ∈ R (x,z) ∈ R
  hence ([x],[y]) ∈  $\langle R \rangle$ list-rel and ([x],[z]) ∈  $\langle R \rangle$ list-rel
    by (auto simp: list-rel-def)
  with single-valuedD[OF SV] show y=z by blast
qed

lemma list-rel-sv[relator-props]:
  single-valued R  $\implies$  single-valued ( $\langle R \rangle$ list-rel)
  by (simp add: list-rel-sv-iff)

lemma list-rel-id[relator-props]:  $\llbracket R=Id \rrbracket \implies \langle R \rangle$ list-rel = Id
  by (auto simp add: list-rel-def list-all2-eq[symmetric])

lemma list-rel-id-simp[simp]:  $\langle Id \rangle$ list-rel = Id by tagged-solver

lemma list-rel-mono[relator-props]:
  assumes A:  $R \subseteq R'$ 
  shows  $\langle R \rangle$ list-rel  $\subseteq$   $\langle R' \rangle$ list-rel
proof clarsimp
  fix l l'
  assume (l,l') ∈  $\langle R \rangle$ list-rel
  thus (l,l') ∈  $\langle R' \rangle$ list-rel
    apply induct
    using A
    by auto
qed

lemma list-rel-range[relator-props]:
  assumes A: Range R = UNIV
  shows Range ( $\langle R \rangle$ list-rel) = UNIV
proof (clarsimp simp: eq-UNIV-iff)
  fix l
  show l ∈ Range ( $\langle R \rangle$ list-rel)
    apply (induct l)
    using A[unfolded eq-UNIV-iff]
    by (auto simp: Range-iff intro: list-relI)
qed

lemma bijective-imp-sv:
  bijective R  $\implies$  single-valued R
  bijective R  $\implies$  single-valued ( $R^{-1}$ )
  by (simp-all add: bijective-alt)

declare bijective-Id[relator-props]

```

**declare** *bijjective-Empty*[*relator-props*]

Pointwise refinement for set types:

**lemma** *set-rel-sv*[*relator-props*]:  
*single-valued*  $R \implies \text{single-valued } (\langle R \rangle \text{set-rel})$   
**unfolding** *single-valued-def set-rel-def* **by** *blast*

**lemma** *set-rel-id*[*relator-props*]:  $R = \text{Id} \implies \langle R \rangle \text{set-rel} = \text{Id}$   
**by** (*auto simp add: set-rel-def*)

**lemma** *set-rel-id-simp*[*simp*]:  $\langle \text{Id} \rangle \text{set-rel} = \text{Id}$  **by** *tagged-solver*

**lemma** *set-rel-csv*[*relator-props*]:  
 $\llbracket \text{single-valued } (R^{-1}) \rrbracket$   
 $\implies \text{single-valued } ((\langle R \rangle \text{set-rel})^{-1})$   
**unfolding** *single-valued-def set-rel-def converse-iff*  
**by** *fast*

### 1.1.5 Invariant and Abstraction

Quite often, a relation can be described as combination of an abstraction function and an invariant, such that the invariant describes valid values on the concrete domain, and the abstraction function maps valid concrete values to its corresponding abstract value.

**definition** *build-rel* **where**  
*build-rel*  $\alpha I \equiv \{(c,a) . a = \alpha c \wedge I c\}$

**abbreviation** *br*  $\equiv$  *build-rel*

**lemmas** *br-def*[*refine-rel-defs*] = *build-rel-def*

**lemma** *in-br-conv*:  $(c,a) \in \text{br } \alpha I \iff a = \alpha c \wedge I c$   
**by** (*auto simp: br-def*)

**lemma** *brI*[*intro?*]:  $\llbracket a = \alpha c ; I c \rrbracket \implies (c,a) \in \text{br } \alpha I$   
**by** (*simp add: br-def*)

**lemma** *br-id*[*simp*]: *br id*  $(\lambda-. \text{True}) = \text{Id}$   
**unfolding** *build-rel-def* **by** *auto*

**lemma** *br-chain*:  
 $(\text{build-rel } \beta J) O (\text{build-rel } \alpha I) = \text{build-rel } (\alpha \circ \beta) (\lambda s. J s \wedge I (\beta s))$   
**unfolding** *build-rel-def* **by** *auto*

**lemma** *br-sv*[*simp, intro!, relator-props*]: *single-valued*  $(\text{br } \alpha I)$   
**unfolding** *build-rel-def*  
**apply** (*rule single-valuedI*)  
**apply** *auto*  
**done**

**lemma** *converse-br-sv-iff*[simp]:  
*single-valued* (*converse* (*br*  $\alpha$  *I*))  $\longleftrightarrow$  *inj-on*  $\alpha$  (*Collect* *I*)  
**by** (*auto intro!*: *inj-onI single-valuedI dest: single-valuedD inj-onD*  
*simp: br-def*) []

**lemmas** [*relator-props*] = *single-valued-relcomp*

**lemma** *br-comp-alt*: *br*  $\alpha$  *I O R* = { (*c,a*) . *I c*  $\wedge$  ( $\alpha$  *c,a*) $\in R$  }  
**by** (*auto simp add: br-def*)

**lemma** *br-comp-alt'*:  
{(*c,a*) .  $a=\alpha$  *c*  $\wedge$  *I c*} *O R* = { (*c,a*) . *I c*  $\wedge$  ( $\alpha$  *c,a*) $\in R$  }  
**by** *auto*

**lemma** *single-valued-as-brE*:  
**assumes** *single-valued R*  
**obtains**  $\alpha$  *invar* **where**  $R=br$   $\alpha$  *invar*  
**apply** (*rule that*[*of*  $\lambda x. THE y. (x,y)\in R$   $\lambda x. x\in Domain R$ ])  
**using** *assms unfolding br-def*  
**by** (*auto dest: single-valuedD*  
*intro: the-equality[symmetric] theI*)

**lemma** *sv-add-invar*:  
*single-valued R*  $\implies$  *single-valued* {(*c, a*) . (*c, a*)  $\in R$   $\wedge$  *I c*}  
**by** (*auto dest: single-valuedD intro: single-valuedI*)

**lemma** *br-Image-conv*[simp]: *br*  $\alpha$  *I* “ *S* = { $\alpha$  *x* | *x. x* $\in S$   $\wedge$  *I x*}  
**by** (*auto simp: br-def*)

### 1.1.6 Miscellaneous

**lemma** *rel-cong*: (*f,g*) $\in Id$   $\implies$  (*x,y*) $\in Id$   $\implies$  (*f x, g y*) $\in Id$  **by** *simp*

**lemma** *rel-fun-cong*: (*f,g*) $\in Id$   $\implies$  (*f x, g x*) $\in Id$  **by** *simp*

**lemma** *rel-arg-cong*: (*x,y*) $\in Id$   $\implies$  (*f x, f y*) $\in Id$  **by** *simp*

### 1.1.7 Conversion between Predicate and Set Based Relators

Autoref uses set-based relators of type (*'a*  $\times$  *'b*) *set*, while the transfer and lifting package of Isabelle/HOL uses predicate based relators of type *'a*  $\Rightarrow$  *'b*  $\Rightarrow$  *bool*. This section defines some utilities to convert between the two.

**definition** *rel2p* *R x y*  $\equiv$  (*x,y*) $\in R$

**definition** *p2rel* *P*  $\equiv$  {(*x,y*) . *P x y*}

**lemma** *rel2pD*: [*rel2p R a b*]  $\implies$  (*a,b*) $\in R$  **by** (*auto simp: rel2p-def*)

**lemma** *p2relD*: [(*a,b*)  $\in$  *p2rel R*]  $\implies$  *R a b* **by** (*auto simp: p2rel-def*)

**lemma** *rel2p-inv*[simp]:  
*rel2p* (*p2rel P*) = *P*  
*p2rel* (*rel2p R*) = *R*

by (auto simp: rel2p-def[abs-def] p2rel-def)

**named-theorems** rel2p

**named-theorems** p2rel

**lemma** rel2p-dflt[rel2p]:

```

rel2p Id = (=)
rel2p (A → B) = rel-fun (rel2p A) (rel2p B)
rel2p (A ×r B) = rel-prod (rel2p A) (rel2p B)
rel2p (⟨A, B⟩sum-rel) = rel-sum (rel2p A) (rel2p B)
rel2p (⟨A⟩option-rel) = rel-option (rel2p A)
rel2p (⟨A⟩list-rel) = list-all2 (rel2p A)
by (auto
  simp: rel2p-def[abs-def]
  intro!: ext
  simp: fun-rel-def rel-fun-def
  simp: sum-rel-def elim: rel-sum.cases
  simp: option-rel-def elim: option.rel-cases
  simp: list-rel-def
  simp: set-rel-def rel-set-def Image-def
)

```

**lemma** p2rel-dflt[p2rel]:

```

p2rel (=) = Id
p2rel (rel-fun A B) = p2rel A → p2rel B
p2rel (rel-prod A B) = p2rel A ×r p2rel B
p2rel (rel-sum A B) = ⟨p2rel A, p2rel B⟩sum-rel
p2rel (rel-option A) = ⟨p2rel A⟩option-rel
p2rel (list-all2 A) = ⟨p2rel A⟩list-rel
by (auto
  simp: p2rel-def[abs-def]
  simp: fun-rel-def rel-fun-def
  simp: sum-rel-def elim: rel-sum.cases
  simp: option-rel-def elim: option.rel-cases
  simp: list-rel-def
)

```

**lemma** [rel2p]: rel2p (⟨A⟩set-rel) = rel-set (rel2p A)

**unfolding** set-rel-def rel-set-def rel2p-def[abs-def]

by blast

**lemma** [p2rel]: left-unique A  $\implies$  p2rel (rel-set A) = (⟨p2rel A⟩set-rel)

**unfolding** set-rel-def rel-set-def p2rel-def[abs-def]

by blast

**lemma** rel2p-comp: rel2p A OO rel2p B = rel2p (A O B)

by (auto simp: rel2p-def[abs-def] intro!: ext)

**lemma** *rel2p-inj[simp]*:  $\text{rel2p } A = \text{rel2p } B \longleftrightarrow A=B$   
**by** (*auto simp: rel2p-def[abs-def]; meson*)

**lemma** *rel2p-left-unique*:  $\text{left-unique } (\text{rel2p } A) = \text{single-valued } (A^{-1})$   
**unfolding** *left-unique-def rel2p-def single-valued-def* **by** *blast*

**lemma** *rel2p-right-unique*:  $\text{right-unique } (\text{rel2p } A) = \text{single-valued } A$   
**unfolding** *right-unique-def rel2p-def single-valued-def* **by** *blast*

**lemma** *rel2p-bi-unique*:  $\text{bi-unique } (\text{rel2p } A) \longleftrightarrow \text{single-valued } A \wedge \text{single-valued } (A^{-1})$   
**unfolding** *bi-unique-alt-def* **by** (*auto simp: rel2p-left-unique rel2p-right-unique*)

**lemma** *p2rel-left-unique*:  $\text{single-valued } ((\text{p2rel } A)^{-1}) = \text{left-unique } A$   
**unfolding** *left-unique-def p2rel-def single-valued-def* **by** *blast*

**lemma** *p2rel-right-unique*:  $\text{single-valued } (\text{p2rel } A) = \text{right-unique } A$   
**unfolding** *right-unique-def p2rel-def single-valued-def* **by** *blast*

### 1.1.8 More Properties

**lemma** *list-rel-compp*:  $\langle A \ O \ B \rangle \text{list-rel} = \langle A \rangle \text{list-rel} \ O \ \langle B \rangle \text{list-rel}$   
**using** *list.rel-compp[of rel2p A rel2p B]*  
**by** (*auto simp: rel2p(2-)[symmetric] rel2p-comp*)

**lemma** *option-rel-compp*:  $\langle A \ O \ B \rangle \text{option-rel} = \langle A \rangle \text{option-rel} \ O \ \langle B \rangle \text{option-rel}$   
**using** *option.rel-compp[of rel2p A rel2p B]*  
**by** (*auto simp: rel2p(2-)[symmetric] rel2p-comp*)

**lemma** *prod-rel-compp*:  $\langle A \ O \ B, \ C \ O \ D \rangle \text{prod-rel} = \langle A, C \rangle \text{prod-rel} \ O \ \langle B, D \rangle \text{prod-rel}$   
**using** *prod.rel-compp[of rel2p A rel2p B rel2p C rel2p D]*  
**by** (*auto simp: rel2p(2-)[symmetric] rel2p-comp*)

**lemma** *sum-rel-compp*:  $\langle A \ O \ B, \ C \ O \ D \rangle \text{sum-rel} = \langle A, C \rangle \text{sum-rel} \ O \ \langle B, D \rangle \text{sum-rel}$   
**using** *sum.rel-compp[of rel2p A rel2p B rel2p C rel2p D]*  
**by** (*auto simp: rel2p(2-)[symmetric] rel2p-comp*)

**lemma** *set-rel-compp*:  $\langle A \ O \ B \rangle \text{set-rel} = \langle A \rangle \text{set-rel} \ O \ \langle B \rangle \text{set-rel}$   
**using** *rel-set-OO[of rel2p A rel2p B]*  
**by** (*auto simp: rel2p(2-)[symmetric] rel2p-comp*)

**lemma** *map-in-list-rel-conv*:  
**shows**  $(l, \text{map } \alpha \ l) \in \langle \text{br } \alpha \ I \rangle \text{list-rel} \longleftrightarrow (\forall x \in \text{set } l. I \ x)$   
**by** (*induction l*) (*auto simp: in-br-conv*)

**lemma** *br-set-rel-alt*:  $(s', s) \in \langle \text{br } \alpha \ I \rangle \text{set-rel} \longleftrightarrow (s = \alpha \ 's' \wedge (\forall x \in s'. I \ x))$   
**by** (*auto simp: set-rel-def br-def*)



**lemma** *finite-Image-sv*:  $\text{single-valued } R \implies \text{finite } s \implies \text{finite } (R \text{ `` } s)$   
**by** (*erule single-valued-as-brE*) *simp*

**lemma** *finite-set-rel-transfer*:  $\llbracket (s, s') \in \langle R \rangle \text{ set-rel}; \text{single-valued } R; \text{finite } s \rrbracket \implies \text{finite } s'$

**unfolding** *set-rel-alt*  
**by** (*blast intro: finite-subset[OF - finite-Image-sv]*)

**lemma** *finite-set-rel-transfer-back*:  $\llbracket (s, s') \in \langle R \rangle \text{ set-rel}; \text{single-valued } (R^{-1}); \text{finite } s' \rrbracket \implies \text{finite } s$

**unfolding** *set-rel-alt*  
**by** (*blast intro: finite-subset[OF - finite-Image-sv]*)

**end**

## 1.2 Basic Parametricity Reasoning

**theory** *Param-Tool*  
**imports** *Relators*  
**begin**

### 1.2.1 Auxiliary Lemmas

**lemma** *tag-both*:  $\llbracket (\text{Let } x \ f, \text{Let } x' \ f') \in R \rrbracket \implies (f \ x, f' \ x') \in R$  **by** *simp*

**lemma** *tag-rhs*:  $\llbracket (c, \text{Let } x \ f) \in R \rrbracket \implies (c, f \ x) \in R$  **by** *simp*

**lemma** *tag-lhs*:  $\llbracket (\text{Let } x \ f, a) \in R \rrbracket \implies (f \ x, a) \in R$  **by** *simp*

**lemma** *tagged-fun-relD-both*:

$\llbracket (f, f') \in A \rightarrow B; (x, x') \in A \rrbracket \implies (\text{Let } x \ f, \text{Let } x' \ f') \in B$

**and** *tagged-fun-relD-rhs*:  $\llbracket (f, f') \in A \rightarrow B; (x, x') \in A \rrbracket \implies (f \ x, \text{Let } x' \ f') \in B$

**and** *tagged-fun-relD-lhs*:  $\llbracket (f, f') \in A \rightarrow B; (x, x') \in A \rrbracket \implies (\text{Let } x \ f, f' \ x') \in B$

**and** *tagged-fun-relD-none*:  $\llbracket (f, f') \in A \rightarrow B; (x, x') \in A \rrbracket \implies (f \ x, f' \ x') \in B$

**by** (*simp-all add: fun-relD*)

### 1.2.2 ML-Setup

**ML**  $\langle$

*signature* *PARAMETRICITY* = *sig*

*type* *param-ruleT* = {

*lhs*: *term*,

*rhs*: *term*,

*R*: *term*,

*rhs-head*: *term*,

*arity*: *int*

}

*val* *dest-param-term*: *term*  $\rightarrow$  *param-ruleT*

```

val dest-param-rule: thm -> param-ruleT
val dest-param-goal: Proof.context -> int -> thm -> param-ruleT

val safe-fun-relD-tac: Proof.context -> tactic'

val adjust-arity: int -> thm -> thm
val adjust-arity-tac: int -> Proof.context -> tactic'
val unlambda-tac: Proof.context -> tactic'
val prepare-tac: Proof.context -> tactic'

val fo-rule: thm -> thm

(***) Basic tactics (***)
val param-rule-tac: Proof.context -> thm -> tactic'
val param-rules-tac: Proof.context -> thm list -> tactic'
val asm-param-tac: Proof.context -> tactic'

(***) Nets of parametricity rules (***)
type param-net
val net-empty: param-net
val net-add: thm -> param-net -> param-net
val net-del: thm -> param-net -> param-net
val net-add-int: Context.generic -> thm -> param-net -> param-net
val net-del-int: Context.generic -> thm -> param-net -> param-net
val net-tac: param-net -> Proof.context -> tactic'

(***) Default parametricity rules (***)
val add-dflt: thm -> Context.generic -> Context.generic
val add-dflt-attr: attribute
val del-dflt: thm -> Context.generic -> Context.generic
val del-dflt-attr: attribute
val get-dflt: Proof.context -> param-net

(** Configuration **)
val cfg-use-asm: bool Config.T
val cfg-single-step: bool Config.T

(** Setup **)
val setup: theory -> theory
end

structure Parametricity : PARAMETRICITY = struct
type param-ruleT = {
  lhs: term,
  rhs: term,
  R: term,
  rhs-head: term,
  arity: int

```

```

}

fun dest-param-term t =
  case
    strip-all-body t |> Logic.strip-imp-concl |> HOLogic.dest-Trueprop
  of
    @{mpat (?lhs,?rhs):?R} => let
      val (rhs-head,arity) =
        case strip-comb rhs of
          (c as Const -,l) => (c,length l)
        | (c as Free -,l) => (c,length l)
        | (c as Abs -,l) => (c,length l)
        | - => raise TERM (dest-param-term: Head,[t])
      in
        { lhs = lhs, rhs = rhs, R=R, rhs-head = rhs-head, arity = arity }
      end
    | t => raise TERM (dest-param-term: Expected (-,-):-,[t])

val dest-param-rule = dest-param-term o Thm.prop-of
fun dest-param-goal ctxt i st =
  if i > Thm.nprems-of st then
    raise THM (dest-param-goal,i,[st])
  else
    dest-param-term (Logic.concl-of-goal (Thm.prop-of st) i)

fun safe-fun-relD-tac ctxt = let
  fun t a b = fo-resolve-tac [a] ctxt THEN' resolve-tac ctxt [b]
  in
    DETERM o (
      t @{thm tag-both} @{thm tagged-fun-relD-both} ORELSE'
      t @{thm tag-rhs} @{thm tagged-fun-relD-rhs} ORELSE'
      t @{thm tag-lhs} @{thm tagged-fun-relD-lhs} ORELSE'
      resolve-tac ctxt @{thms tagged-fun-relD-none}
    )
  end

fun adjust-arity i thm =
  if i = 0 then thm
  else if i < 0 then funpow (~i) (fn thm => thm RS @{thm fun-rell}) thm
  else funpow i (fn thm => thm RS @{thm fun-relD}) thm

fun NTIMES k tac =
  if k <= 0 then K all-tac
  else tac THEN' NTIMES (k-1) tac

fun adjust-arity-tac n ctxt i st =
  (if n = 0 then K all-tac
   else if n > 0 then NTIMES n (DETERM o resolve-tac ctxt @{thms fun-rell})
  )

```

```

else NTIMES (~ n) (safe-fun-relD-tac ctxt) i st

fun unlambdac ctxt i st =
  case try (dest-param-goal ctxt i) st of
  NONE => Seq.empty
  | SOME g => let
    val n = Term.strip-abs (#rhs-head g) |> #1 |> length
  in NTIMES n (resolve-tac ctxt @ {thms fun-relI}) i st end

fun prepare-tac ctxt =
  Subgoal.FOCUS (K (PRIMITIVE (Drule.eta-contraction-rule))) ctxt
  THEN' unlambdac ctxt

fun could-param-rl ctxt rl i st =
  if i > Thm.nprems-of st then NONE
  else (
    case (try (dest-param-goal ctxt i) st, try dest-param-term rl) of
    (SOME g, SOME r) =>
      if Term.could-unify (#rhs-head g, #rhs-head r) then
        SOME (#arity r - #arity g)
      else NONE
    | - => NONE
  )

fun param-rule-tac-aux ctxt rl i st =
  case could-param-rl ctxt (Thm.prop-of rl) i st of
  SOME adj => (adjust-arity-tac adj ctxt THEN' resolve-tac ctxt [rl]) i st
  | - => Seq.empty

fun param-rule-tac ctxt rl =
  prepare-tac ctxt THEN' param-rule-tac-aux ctxt rl

fun param-rules-tac ctxt rls =
  prepare-tac ctxt THEN' FIRST' (map (param-rule-tac-aux ctxt) rls)

fun asm-param-tac-aux ctxt i st =
  if i > Thm.nprems-of st then Seq.empty
  else let
    val prems = Logic.premsof-goal (Thm.prop-of st) i |> tag-list 1

    fun tac (n,t) i st = case could-param-rl ctxt t i st of
      SOME adj => (adjust-arity-tac adj ctxt THEN' rprem-tac n ctxt) i st
      | NONE => Seq.empty
    in
      FIRST' (map tac prems) i st
    end

fun asm-param-tac ctxt = prepare-tac ctxt THEN' asm-param-tac-aux ctxt

```

```

type param-net = (param-ruleT * thm) Item-Net.T

local
  val param-get-key = single o #rhs-head o #1
in
  val net-empty = Item-Net.init (Thm.eq-thm o apply2 #2) param-get-key
end

fun wrap-pr-op f context thm = case try ('dest-param-rule) thm of
  NONE =>
    let
      val msg = Ignoring invalid parametricity theorem:
        ^ Thm.string-of-thm (Context.proof-of context) thm
      val - = warning msg
    in I end
  | SOME p => f p

val net-add-int = wrap-pr-op Item-Net.update
val net-del-int = wrap-pr-op Item-Net.remove

val net-add = Item-Net.update o 'dest-param-rule
val net-del = Item-Net.remove o 'dest-param-rule

fun net-tac-aux net ctxt i st =
  if i > Thm.nprems-of st then
    Seq.empty
  else
    let
      val g = dest-param-goal ctxt i st
      val rls = Item-Net.retrieve net (#rhs-head g)

      fun tac (r, thm) =
        adjust-arity-tac (#arity r - #arity g) ctxt
          THEN' DETERM o resolve-tac ctxt [thm]
    in
      FIRST' (map tac rls) i st
    end

fun net-tac net ctxt = prepare-tac ctxt THEN' net-tac-aux net ctxt

structure dflt-rules = Generic-Data (
  type T = param-net
  val empty = net-empty
  val merge = Item-Net.merge
)

fun add-dflt thm context = dflt-rules.map (net-add-int context thm) context
fun del-dflt thm context = dflt-rules.map (net-del-int context thm) context

```

```

val add-dflt-attr = Thm.declaration-attribute add-dflt
val del-dflt-attr = Thm.declaration-attribute del-dflt

val get-dflt = dflt-rules.get o Context.Proof

val cfg-use-asm =
  Attrib.setup-config-bool @{binding param-use-asm} (K true)
val cfg-single-step =
  Attrib.setup-config-bool @{binding param-single-step} (K false)

local
  open Refine-Util

  val param-modifiers =
    [Args.add -- Args.colon >> K (Method.modifier add-dflt-attr here),
     Args.del -- Args.colon >> K (Method.modifier del-dflt-attr here),
     Args.$$$ only -- Args.colon >>
      K {init = Context.proof-map (dflt-rules.map (K net-empty)),
        attribute = add-dflt-attr, pos = here}]

  val param-flags =
    parse-bool-config use-asm cfg-use-asm
  || parse-bool-config single-step cfg-single-step

in

val parametricity-method =
  parse-paren-lists param-flags |-- Method.sections param-modifiers >>
  (fn - => fn ctxt =>
    let
      val net2 = get-dflt ctxt
      val asm-tac =
        if Config.get ctxt cfg-use-asm then
          asm-param-tac ctxt
        else K no-tac

      val RPT =
        if Config.get ctxt cfg-single-step then I
        else REPEAT-ALL-NEW-FWD

    in
      SIMPLE-METHOD' (
        RPT (
          (assume-tac ctxt
            ORELSE' net-tac net2 ctxt
            ORELSE' asm-tac)
        )
      )
    end
  )

```

```

    )
  end

  fun fo-rule thm = case Thm.concl-of thm of
    @{\mpat Trueprop ((-,-)∈-->-)} => fo-rule (thm RS @{\thm fun-relD})
  | - => thm

  val param-fo-attr = Scan.succeed (Thm.rule-attribute [] (K fo-rule))

  val setup = I
    #> Attrib.setup @{\binding param}
      (Attrib.add-del add-dflt-attr del-dflt-attr)
      declaration of parametricity theorem
    #> Global-Theory.add-thms-dynamic (@{\binding param},
      map #2 o Item-Net.content o dflt-rules.get)
    #> Method.setup @{\binding parametricity} parametricity-method
      Parametricity solver
    #> Attrib.setup @{\binding param-fo} param-fo-attr
      Parametricity: Rule in first-order form

  end
>

setup Parametricity.setup

```

### 1.2.3 Convenience Tools

**ML** ‹

(\* Prefix p- or wrong type supresses generation of relAPP \*)

```

fun cnv-relAPP t = let
  fun consider (Var ((name,-),T)) =
    if String.isPrefix p- name then false
    else (
      case T of
        Type(@{\type-name set},[Type(@{\type-name prod},-)]) => true
      | - => false)
  | consider - = true

  fun strip-rcomb u : term * term list =
    let
      fun stripc (x as (f$t, ts)) =
        if consider t then stripc (f, t::ts) else x
      | stripc x = x
    in stripc(u,[]) end;

  val (f,a) = strip-rcomb t
in
  Relators.list-relAPP a f

```

```

end

fun to-relAPP-conv ctxt = Refine-Util.f-tac-conv ctxt
  cnv-relAPP
  (fn goal-ctxt => ALLGOALS (simp-tac
    (put-simpset HOL-basic-ss goal-ctxt addsimps @{thms relAPP-def})))

val to-relAPP-attr = Thm.rule-attribute [] (fn context => let
  val ctxt = Context.proof-of context
in
  Conv.fconv-rule (Conv.arg1-conv (to-relAPP-conv ctxt))
end)
>

attribute-setup to-relAPP = <Scan.succeed (to-relAPP-attr)>
  Convert relator definition to prefix-form

end

```

### 1.3 Parametricity Theorems for HOL

```

theory Param-HOL
imports Param-Tool
begin

```

#### 1.3.1 Sets

```

lemma param-empty[param]:
   $(\{\}, \{\}) \in \langle R \rangle \text{set-rel}$  by (auto simp: set-rel-def)

lemma param-member[param]:
   $\llbracket \text{single-valued } R; \text{ single-valued } (R^{-1}) \rrbracket \implies ((\in), (\in)) \in R \rightarrow \langle R \rangle \text{set-rel} \rightarrow \text{bool-rel}$ 

unfolding set-rel-def
by (blast dest: single-valuedD)

lemma param-insert[param]:
   $(\text{insert}, \text{insert}) \in R \rightarrow \langle R \rangle \text{set-rel} \rightarrow \langle R \rangle \text{set-rel}$ 
by (auto simp: set-rel-def)

lemma param-union[param]:
   $((\cup), (\cup)) \in \langle R \rangle \text{set-rel} \rightarrow \langle R \rangle \text{set-rel} \rightarrow \langle R \rangle \text{set-rel}$ 
by (auto simp: set-rel-def)

lemma param-inter[param]:
assumes single-valued R    single-valued (R-1)

```



**shows**  $((\cap), (\cap)) \in \langle R \rangle \text{set-rel} \rightarrow \langle R \rangle \text{set-rel} \rightarrow \langle R \rangle \text{set-rel}$   
**using** *assms*  
**unfolding** *set-rel-def*  
**by** (*blast dest: single-valuedD*)

**lemma** *param-diff*[*param*]:  
**assumes** *single-valued R single-valued (R<sup>-1</sup>)*  
**shows**  $((-), (-)) \in \langle R \rangle \text{set-rel} \rightarrow \langle R \rangle \text{set-rel} \rightarrow \langle R \rangle \text{set-rel}$   
**using** *assms*  
**unfolding** *set-rel-def*  
**by** (*blast dest: single-valuedD*)

**lemma** *param-subseteq*[*param*]:  
 $\llbracket \text{single-valued } R; \text{single-valued } (R^{-1}) \rrbracket \implies ((\subseteq), (\subseteq)) \in \langle R \rangle \text{set-rel} \rightarrow \langle R \rangle \text{set-rel}$   
 $\rightarrow \text{bool-rel}$   
**unfolding** *set-rel-def*  
**by** (*blast dest: single-valuedD*)

**lemma** *param-subset*[*param*]:  
 $\llbracket \text{single-valued } R; \text{single-valued } (R^{-1}) \rrbracket \implies ((\subset), (\subset)) \in \langle R \rangle \text{set-rel} \rightarrow \langle R \rangle \text{set-rel}$   
 $\rightarrow \text{bool-rel}$   
**unfolding** *set-rel-def*  
**by** (*blast dest: single-valuedD*)

**lemma** *param-Ball*[*param*]:  $(\text{Ball}, \text{Ball}) \in \langle Ra \rangle \text{set-rel} \rightarrow (Ra \rightarrow Id) \rightarrow Id$   
**by** (*force simp: set-rel-alt dest: fun-relD*)

**lemma** *param-Bex*[*param*]:  $(\text{Bex}, \text{Bex}) \in \langle Ra \rangle \text{set-rel} \rightarrow (Ra \rightarrow Id) \rightarrow Id$   
**by** (*fastforce simp: set-rel-def dest: fun-relD*)

**lemma** *param-set*[*param*]:  
 $\text{single-valued } Ra \implies (\text{set}, \text{set}) \in \langle Ra \rangle \text{list-rel} \rightarrow \langle Ra \rangle \text{set-rel}$

**proof**  
**fix** *l l'*  
**assume** *A: single-valued Ra*  
**assume**  $(l, l') \in \langle Ra \rangle \text{list-rel}$   
**thus**  $(\text{set } l, \text{set } l') \in \langle Ra \rangle \text{set-rel}$   
**apply** (*induct*)  
**apply** *simp*  
**apply** *simp*  
**using** *A apply (parametricity)*  
**done**

**qed**

**lemma** *param-Collect*[*param*]:  
 $\llbracket \text{Domain } A = \text{UNIV}; \text{Range } A = \text{UNIV} \rrbracket \implies (\text{Collect}, \text{Collect}) \in (A \rightarrow \text{bool-rel}) \rightarrow$   
 $\langle A \rangle \text{set-rel}$   
**unfolding** *set-rel-def*

```

apply (clarsimp; safe)
subgoal using fun-relD1 by fastforce
subgoal using fun-relD2 by fastforce
done

```

```

lemma param-finite[param]:  $\llbracket$ 
  single-valued R; single-valued ( $R^{-1}$ )
 $\rrbracket \implies (finite, finite) \in \langle R \rangle set\text{-rel} \rightarrow bool\text{-rel}$ 
using finite-set-rel-transfer finite-set-rel-transfer-back by blast

```

```

lemma param-card[param]:  $\llbracket$ single-valued R; single-valued ( $R^{-1}$ ) $\rrbracket$ 
 $\implies (card, card) \in \langle R \rangle set\text{-rel} \rightarrow nat\text{-rel}$ 
apply (rule rel2pD)
apply (simp only: rel2p)
apply (rule card-transfer)
by (simp add: rel2p-bi-unique)

```

### 1.3.2 Standard HOL Constructs

```

lemma param-if[param]:
  assumes  $(c, c') \in Id$ 
  assumes  $\llbracket c; c' \rrbracket \implies (t, t') \in R$ 
  assumes  $\llbracket \neg c; \neg c' \rrbracket \implies (e, e') \in R$ 
  shows (If c t e, If c' t' e')  $\in R$ 
  using assms by auto

```

```

lemma param-Let[param]:
   $(Let, Let) \in Ra \rightarrow (Ra \rightarrow Rr) \rightarrow Rr$ 
  by (auto dest: fun-relD)

```

### 1.3.3 Functions

```

lemma param-id[param]:  $(id, id) \in R \rightarrow R$  unfolding id-def by parametricity

```

```

lemma param-fun-comp[param]:  $((o), (o)) \in (Ra \rightarrow Rb) \rightarrow (Rc \rightarrow Ra) \rightarrow Rc \rightarrow Rb$ 
  unfolding comp-def[abs-def] by parametricity

```

```

lemma param-fun-upd[param]:
   $((=), (=)) \in Ra \rightarrow Ra \rightarrow Id$ 
 $\implies (fun\text{-upd}, fun\text{-upd}) \in (Ra \rightarrow Rb) \rightarrow Ra \rightarrow Rb \rightarrow Ra \rightarrow Rb$ 
  unfolding fun-upd-def[abs-def]
  by (parametricity)

```

### 1.3.4 Boolean

```

lemma rec-bool-is-case: old.rec-bool = case-bool
  by (rule ext) + (auto split: bool.split)

```

```

lemma param-bool[param]:
   $(True, True) \in Id$ 

```

$(False, False) \in Id$   
 $(conj, conj) \in Id \rightarrow Id \rightarrow Id$   
 $(disj, disj) \in Id \rightarrow Id \rightarrow Id$   
 $(Not, Not) \in Id \rightarrow Id$   
 $(case\text{-}bool, case\text{-}bool) \in R \rightarrow R \rightarrow Id \rightarrow R$   
 $(old.\text{rec}\text{-}bool, old.\text{rec}\text{-}bool) \in R \rightarrow R \rightarrow Id \rightarrow R$   
 $((\leftarrow), (\leftarrow)) \in Id \rightarrow Id \rightarrow Id$   
 $((\rightarrow), (\rightarrow)) \in Id \rightarrow Id \rightarrow Id$   
**by** *(auto split: bool.split simp: rec-bool-is-case)*

**lemma** *param-and-cong1*:  $\llbracket (a, a') \in bool\text{-}rel; \llbracket a; a \rrbracket \implies (b, b') \in bool\text{-}rel \rrbracket \implies (a \wedge b, a' \wedge b') \in bool\text{-}rel$   
**by** *blast*

**lemma** *param-and-cong2*:  $\llbracket (a, a') \in bool\text{-}rel; \llbracket a; a \rrbracket \implies (b, b') \in bool\text{-}rel \rrbracket \implies (b \wedge a, b' \wedge a') \in bool\text{-}rel$   
**by** *blast*

### 1.3.5 Nat

**lemma** *param-nat1*[*param*]:

$(0, 0::nat) \in Id$   
 $(Suc, Suc) \in Id \rightarrow Id$   
 $(1, 1::nat) \in Id$   
 $(numeral\ n::nat, numeral\ n::nat) \in Id$   
 $((<), (< :: nat \Rightarrow -)) \in Id \rightarrow Id \rightarrow Id$   
 $((\leq), (\leq :: nat \Rightarrow -)) \in Id \rightarrow Id \rightarrow Id$   
 $((=), (= :: nat \Rightarrow -)) \in Id \rightarrow Id \rightarrow Id$   
 $((+), (+ :: nat \Rightarrow -, (+))) \in Id \rightarrow Id \rightarrow Id$   
 $((-), (- :: nat \Rightarrow -, (-))) \in Id \rightarrow Id \rightarrow Id$   
 $((*), (* :: nat \Rightarrow -, (*))) \in Id \rightarrow Id \rightarrow Id$   
 $((div), (div :: nat \Rightarrow -, (div))) \in Id \rightarrow Id \rightarrow Id$   
 $((mod), (mod :: nat \Rightarrow -, (mod))) \in Id \rightarrow Id \rightarrow Id$   
**by** *auto*

**lemma** *param-case-nat*[*param*]:

$(case\text{-}nat, case\text{-}nat) \in Ra \rightarrow (Id \rightarrow Ra) \rightarrow Id \rightarrow Ra$   
**apply** *(intro fun-relI)*  
**apply** *(auto split: nat.split dest: fun-relD)*  
**done**

**lemma** *param-rec-nat*[*param*]:

$(rec\text{-}nat, rec\text{-}nat) \in R \rightarrow (Id \rightarrow R \rightarrow R) \rightarrow Id \rightarrow R$

**proof** *(intro fun-relI, goal-cases)*

**case**  $(1\ s\ s'\ f\ f'\ n\ n')$  **thus** *?case*

**apply** *(induct n' arbitrary: n s s')*

**apply** *(fastforce simp: fun-rel-def)+*

**done**

**qed**

### 1.3.6 Int

**lemma** *param-int*[*param*]:

$(0, 0::int) \in Id$   
 $(1, 1::int) \in Id$   
 $(numeral\ n::int, numeral\ n::int) \in Id$   
 $((<), (< ::int \Rightarrow -)) \in Id \rightarrow Id \rightarrow Id$   
 $((\leq), (\leq ::int \Rightarrow -)) \in Id \rightarrow Id \rightarrow Id$   
 $((=), (= ::int \Rightarrow -)) \in Id \rightarrow Id \rightarrow Id$   
 $((+) ::int \Rightarrow -, (+)) \in Id \rightarrow Id \rightarrow Id$   
 $((-) ::int \Rightarrow -, (-)) \in Id \rightarrow Id \rightarrow Id$   
 $((*) ::int \Rightarrow -, (*)) \in Id \rightarrow Id \rightarrow Id$   
 $((div) ::int \Rightarrow -, (div)) \in Id \rightarrow Id \rightarrow Id$   
 $((mod) ::int \Rightarrow -, (mod)) \in Id \rightarrow Id \rightarrow Id$   
**by** *auto*

### 1.3.7 Product

**lemma** *param-unit*[*param*]:  $(((), ())) \in unit\text{-}rel$  **by** *auto*

**lemma** *rec-prod-is-case*:  $old.\text{rec-prod} = \text{case-prod}$   
**by** (*rule ext*) + (*auto split: bool.split*)

**lemma** *param-prod*[*param*]:  
 $(Pair, Pair) \in Ra \rightarrow Rb \rightarrow \langle Ra, Rb \rangle prod\text{-}rel$   
 $(\text{case-prod}, \text{case-prod}) \in (Ra \rightarrow Rb \rightarrow Rr) \rightarrow \langle Ra, Rb \rangle prod\text{-}rel \rightarrow Rr$   
 $(old.\text{rec-prod}, old.\text{rec-prod}) \in (Ra \rightarrow Rb \rightarrow Rr) \rightarrow \langle Ra, Rb \rangle prod\text{-}rel \rightarrow Rr$   
 $(fst, fst) \in \langle Ra, Rb \rangle prod\text{-}rel \rightarrow Ra$   
 $(snd, snd) \in \langle Ra, Rb \rangle prod\text{-}rel \rightarrow Rb$   
**by** (*auto dest: fun-relD split: prod.split*  
*simp: prod-rel-def rec-prod-is-case*)

**lemma** *param-case-prod'*:  
 $\llbracket (p, p') \in \langle Ra, Rb \rangle prod\text{-}rel;$   
 $\quad \wedge a\ b\ a'\ b'. \llbracket p=(a, b); p'=(a', b'); (a, a') \in Ra; (b, b') \in Rb \rrbracket$   
 $\quad \implies (f\ a\ b, f'\ a'\ b') \in R$   
 $\rrbracket \implies (\text{case-prod}\ f\ p, \text{case-prod}\ f'\ p') \in R$   
**by** (*auto split: prod.split*)

**lemma** *param-case-prod''*:  
 $\llbracket$   
 $\quad \wedge a\ b\ a'\ b'. \llbracket p=(a, b); p'=(a', b') \rrbracket \implies (f\ a\ b, f'\ a'\ b') \in R$   
 $\rrbracket \implies (\text{case-prod}\ f\ p, \text{case-prod}\ f'\ p') \in R$   
**by** (*auto split: prod.split*)

**lemma** *param-map-prod*[*param*]:  
 $(\text{map-prod}, \text{map-prod})$   
 $\in (Ra \rightarrow Rb) \rightarrow (Rc \rightarrow Rd) \rightarrow \langle Ra, Rc \rangle prod\text{-}rel \rightarrow \langle Rb, Rd \rangle prod\text{-}rel$   
**unfolding** *map-prod-def*[*abs-def*]  
**by** *parametricity*

**lemma** *param-apfst*[*param*]:

$(apfst, apfst) \in (Ra \rightarrow Rb) \rightarrow \langle Ra, Rc \rangle prod-rel \rightarrow \langle Rb, Rc \rangle prod-rel$   
**unfolding** *apfst-def*[*abs-def*] **by** *parametricity*

**lemma** *param-apsnd*[*param*]:

$(apsnd, apsnd) \in (Rb \rightarrow Rc) \rightarrow \langle Ra, Rb \rangle prod-rel \rightarrow \langle Ra, Rc \rangle prod-rel$   
**unfolding** *apsnd-def*[*abs-def*] **by** *parametricity*

**lemma** *param-curry*[*param*]:

$(curry, curry) \in (\langle Ra, Rb \rangle prod-rel \rightarrow Rc) \rightarrow Ra \rightarrow Rb \rightarrow Rc$   
**unfolding** *curry-def* **by** *parametricity*

**lemma** *param-uncurry*[*param*]:  $(uncurry, uncurry) \in (A \rightarrow B \rightarrow C) \rightarrow A \times_r B \rightarrow C$

**unfolding** *uncurry-def*[*abs-def*] **by** *parametricity*

**lemma** *param-prod-swap*[*param*]:  $(prod.swap, prod.swap) \in A \times_r B \rightarrow B \times_r A$  **by** *auto*

**context** *partial-function-definitions* **begin**

**lemma**

**assumes** *M*: *monotone le-fun le-fun F*

**and** *M'*: *monotone le-fun le-fun F'*

**assumes** *ADM*:

*admissible*  $(\lambda a. \forall x xa. (x, xa) \in Rb \longrightarrow (a x, fixp-fun F' xa) \in Ra)$

**assumes** *bot*:  $\bigwedge x xa. (x, xa) \in Rb \implies (lub \{ \}, fixp-fun F' xa) \in Ra$

**assumes** *F*:  $(F, F') \in (Rb \rightarrow Ra) \rightarrow Rb \rightarrow Ra$

**assumes** *A*:  $(x, x') \in Rb$

**shows**  $(fixp-fun F x, fixp-fun F' x') \in Ra$

**using** *A*

**apply** (*induct arbitrary*:  $x x'$  *rule*: *ccpo.fixp-induct*[*OF ccpo - M*])

**apply** (*rule ADM*)

**apply** (*simp add*: *fun-lub-def bot*)

**apply** (*subst ccpo.fixp-unfold*[*OF ccpo M*^])

**apply** (*parametricity add*: *F*)

**done**

**end**

### 1.3.8 Option

**lemma** *param-option*[*param*]:

$(None, None) \in \langle R \rangle option-rel$

$(Some, Some) \in R \rightarrow \langle R \rangle option-rel$

$(case-option, case-option) \in Rr \rightarrow (R \rightarrow Rr) \rightarrow \langle R \rangle option-rel \rightarrow Rr$

$(rec-option, rec-option) \in Rr \rightarrow (R \rightarrow Rr) \rightarrow \langle R \rangle option-rel \rightarrow Rr$

**by** (*auto split*: *option.split*

*simp*: *option-rel-def case-option-def*[*symmetric*]

*dest*: *fun-relD*)

**lemma** *param-map-option*[*param*]:  $(map-option, map-option) \in (A \rightarrow B) \rightarrow \langle A \rangle option-rel \rightarrow \langle B \rangle option-rel$

```

apply (intro fun-relI)
apply (auto elim!: option-relE dest: fun-relD)
done

```

**lemma** *param-case-option'*:

```

[[ (x,x') ∈ ⟨Rv⟩option-rel;
  [x=None; x'=None] ⇒ (fn,fn') ∈ R;
  ∧ v v'. [x=Some v; x'=Some v'; (v,v') ∈ Rv ] ⇒ (fs v, fs' v') ∈ R
]] ⇒ (case-option fn fs x, case-option fn' fs' x') ∈ R
by (auto split: option.split)

```

**lemma** *the-paramL*:  $[[l \neq \text{None}; (l,r) \in \langle R \rangle \text{option-rel}] \Rightarrow (\text{the } l, \text{the } r) \in R$

```

apply (cases l)
by (auto elim: option-relE)

```

**lemma** *the-paramR*:  $[[r \neq \text{None}; (l,r) \in \langle R \rangle \text{option-rel}] \Rightarrow (\text{the } l, \text{the } r) \in R$

```

apply (cases l)
by (auto elim: option-relE)

```

**lemma** *the-default-param*[*param*]:

```

(the-default, the-default) ∈ R → ⟨R⟩option-rel → R
unfolding the-default-def
by parametricity

```

### 1.3.9 Sum

**lemma** *rec-sum-is-case*: *old.rec-sum = case-sum*

```

by (rule ext)+ (auto split: sum.split)

```

**lemma** *param-sum*[*param*]:

```

(Inl,Inl) ∈ Rl → ⟨Rl,Rr⟩sum-rel
(Inr,Inr) ∈ Rr → ⟨Rl,Rr⟩sum-rel
(case-sum,case-sum) ∈ (Rl → R) → (Rr → R) → ⟨Rl,Rr⟩sum-rel → R
(old.rec-sum,old.rec-sum) ∈ (Rl → R) → (Rr → R) → ⟨Rl,Rr⟩sum-rel → R
by (fastforce split: sum.split dest: fun-relD
  simp: rec-sum-is-case)+

```

**lemma** *param-case-sum'*:

```

[[ (s,s') ∈ ⟨Rl,Rr⟩sum-rel;
  ∧ l l'. [s=Inl l; s'=Inl l'; (l,l') ∈ Rl ] ⇒ (fl l, fl' l') ∈ R;
  ∧ r r'. [s=Inr r; s'=Inr r'; (r,r') ∈ Rr ] ⇒ (fr r, fr' r') ∈ R
]] ⇒ (case-sum fl fr s, case-sum fl' fr' s') ∈ R
by (auto split: sum.split)

```

**primrec** *is-Inl* **where** *is-Inl* (Inl -) = True | *is-Inl* (Inr -) = False

**primrec** *is-Inr* **where** *is-Inr* (Inr -) = True | *is-Inr* (Inl -) = False

**lemma** *is-Inl-param*[*param*]:  $(\text{is-Inl}, \text{is-Inl}) \in \langle Ra, Rb \rangle \text{sum-rel} \rightarrow \text{bool-rel}$

```

unfolding is-Inl-def by parametricity

```

**lemma** *is-Inr-param*[*param*]:  $(is-Inr, is-Inr) \in \langle Ra, Rb \rangle sum-rel \rightarrow bool-rel$   
**unfolding** *is-Inr-def* **by** *parametricity*

**lemma** *sum-projl-param*[*param*]:  
 $\llbracket is-Inl\ s; (s', s) \in \langle Ra, Rb \rangle sum-rel \rrbracket$   
 $\implies (Sum-Type.sum.projl\ s', Sum-Type.sum.projl\ s) \in Ra$   
**apply** (*cases s*)  
**apply** (*auto elim: sum-relE*)  
**done**

**lemma** *sum-projr-param*[*param*]:  
 $\llbracket is-Inr\ s; (s', s) \in \langle Ra, Rb \rangle sum-rel \rrbracket$   
 $\implies (Sum-Type.sum.projr\ s', Sum-Type.sum.projr\ s) \in Rb$   
**apply** (*cases s*)  
**apply** (*auto elim: sum-relE*)  
**done**

### 1.3.10 List

**lemma** *list-rel-append1*:  $(as\ @\ bs, l) \in \langle R \rangle list-rel$   
 $\longleftrightarrow (\exists\ cs\ ds.\ l = cs@ds \wedge (as, cs) \in \langle R \rangle list-rel \wedge (bs, ds) \in \langle R \rangle list-rel)$   
**apply** (*simp add: list-rel-def list-all2-append1*)  
**apply** *auto*  
**apply** (*metis list-all2-lengthD*)  
**done**

**lemma** *list-rel-append2*:  $(l, as\ @\ bs) \in \langle R \rangle list-rel$   
 $\longleftrightarrow (\exists\ cs\ ds.\ l = cs@ds \wedge (cs, as) \in \langle R \rangle list-rel \wedge (ds, bs) \in \langle R \rangle list-rel)$   
**apply** (*simp add: list-rel-def list-all2-append2*)  
**apply** *auto*  
**apply** (*metis list-all2-lengthD*)  
**done**

**lemma** *param-append*[*param*]:  
 $(append, append) \in \langle R \rangle list-rel \rightarrow \langle R \rangle list-rel \rightarrow \langle R \rangle list-rel$   
**by** (*auto simp: list-rel-def list-all2-appendI*)

**lemma** *param-list1*[*param*]:  
 $(Nil, Nil) \in \langle R \rangle list-rel$   
 $(Cons, Cons) \in R \rightarrow \langle R \rangle list-rel \rightarrow \langle R \rangle list-rel$   
 $(case-list, case-list) \in Rr \rightarrow (R \rightarrow \langle R \rangle list-rel \rightarrow Rr) \rightarrow \langle R \rangle list-rel \rightarrow Rr$   
**apply** (*force dest: fun-relD split: list.split*)  
**done**

**lemma** *param-rec-list*[*param*]:  
 $(rec-list, rec-list)$   
 $\in Ra \rightarrow (Rb \rightarrow \langle Rb \rangle list-rel \rightarrow Ra \rightarrow Ra) \rightarrow \langle Rb \rangle list-rel \rightarrow Ra$   
**proof** (*intro fun-relI, goal-cases*)

```

case prems: (1 a a' f f' l l')
from prems(3) show ?case
  using prems(1,2)
  apply (induct arbitrary: a a')
  apply simp
  apply (fastforce dest: fun-relD)
done

```

qed

**lemma** *param-case-list'*:

```

[[ (l,l') ∈ ⟨Rb⟩list-rel;
  [[ l=[]; l'=[] ] ⇒ (n,n') ∈ Ra;
  ∧ x xs x' xs'. [[ l=x#xs; l'=x'#xs'; (x,x') ∈ Rb; (xs,xs') ∈ ⟨Rb⟩list-rel ]
  ⇒ (c x xs, c' x' xs') ∈ Ra
]] ⇒ (case-list n c l, case-list n' c' l') ∈ Ra
by (auto split: list.split)

```

**lemma** *param-map[param]*:

```

(map,map) ∈ (R1 → R2) → ⟨R1⟩list-rel → ⟨R2⟩list-rel
unfolding map-rec[abs-def] by (parametricity)

```

**lemma** *param-fold[param]*:

```

(fold,fold) ∈ (Re → Rs → Rs) → ⟨Re⟩list-rel → Rs → Rs
(foldl,foldl) ∈ (Rs → Re → Rs) → Rs → ⟨Re⟩list-rel → Rs
(foldr,foldr) ∈ (Re → Rs → Rs) → ⟨Re⟩list-rel → Rs → Rs
unfolding List.fold-def List.foldr-def List.foldl-def
by (parametricity)+

```

**lemma** *param-list-all[param]*: (*list-all,list-all*) ∈ (A → bool-rel) → ⟨A⟩list-rel → bool-rel

**by** (*fold rel2p-def*) (*simp add: rel2p List.list-all-transfer*)

**context** begin

**private primrec** *list-all2-alt* :: ('a ⇒ 'b ⇒ bool) ⇒ 'a list ⇒ 'b list ⇒ bool

**where**

```

  list-all2-alt P [] ys ↔ (case ys of [] ⇒ True | - ⇒ False)
  | list-all2-alt P (x#xs) ys ↔ (case ys of [] ⇒ False | y#ys ⇒ P x y ∧ list-all2-alt
P xs ys)

```

**private lemma** *list-all2-alt*: *list-all2* P *xs ys* = *list-all2-alt* P *xs ys*

**by** (*induction xs arbitrary: ys*) (*auto split: list.splits*)

**lemma** *param-list-all2[param]*: (*list-all2, list-all2*) ∈ (A → B → bool-rel) → ⟨A⟩list-rel → ⟨B⟩list-rel → bool-rel

```

unfolding list-all2-alt[abs-def]
unfolding list-all2-alt-def[abs-def]
by parametricity

```

end



**lemma** *param-hd*[*param*]:  $l \neq [] \implies (l', l) \in \langle A \rangle \text{list-rel} \implies (\text{hd } l', \text{hd } l) \in A$   
**unfolding** *hd-def* **by** (*auto split: list.splits*)

**lemma** *param-last*[*param*]:  
**assumes**  $y \neq []$   
**assumes**  $(x, y) \in \langle A \rangle \text{list-rel}$   
**shows**  $(\text{last } x, \text{last } y) \in A$   
**using** *assms(2,1)*  
**by** (*induction rule: list-rel-induct*) *auto*

**lemma** *param-rotate1*[*param*]:  $(\text{rotate1}, \text{rotate1}) \in \langle A \rangle \text{list-rel} \rightarrow \langle A \rangle \text{list-rel}$   
**unfolding** *rotate1-def* **by** *parametricity*

**schematic-goal** *param-take*[*param*]:  $(\text{take}, \text{take}) \in (?R::(-\times-)) \text{ set}$   
**unfolding** *take-def*  
**by** (*parametricity*)

**schematic-goal** *param-drop*[*param*]:  $(\text{drop}, \text{drop}) \in (?R::(-\times-)) \text{ set}$   
**unfolding** *drop-def*  
**by** (*parametricity*)

**schematic-goal** *param-length*[*param*]:  
 $(\text{length}, \text{length}) \in (?R::(-\times-)) \text{ set}$   
**unfolding** *size-list-overloaded-def size-list-def*  
**by** (*parametricity*)

**fun** *list-eq* ::  $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$  **where**  
*list-eq* *eq* [] []  $\longleftrightarrow \text{True}$   
| *list-eq* *eq* (*a*#*l*) (*a'*#*l'*)  
 $\longleftrightarrow (\text{if } \text{eq } a \ a' \ \text{then } \text{list-eq } \text{eq } \text{ l } \ \text{l}' \ \text{else } \text{False})$   
| *list-eq* - - -  $\longleftrightarrow \text{False}$

**lemma** *param-list-eq*[*param*]:  
 $(\text{list-eq}, \text{list-eq}) \in$   
 $(R \rightarrow R \rightarrow \text{Id}) \rightarrow \langle R \rangle \text{list-rel} \rightarrow \langle R \rangle \text{list-rel} \rightarrow \text{Id}$

**proof** (*intro fun-rell, goal-cases*)  
**case** *prems*:  $(1 \ \text{eq} \ \text{eq}' \ \text{l1} \ \text{l1}' \ \text{l2} \ \text{l2}')$   
**thus** *?case*  
**apply** -  
**apply** (*induct eq' l1' l2' arbitrary: l1 l2 rule: list-eq.induct*)  
**apply** (*simp-all only: list-eq.simps* |  
*elim list-reLE* |  
*parametricity*)  
**done**  
**qed**

**lemma** *id-list-eq-aux*[*simp*]:  $(\text{list-eq } (=)) = (=)$   
**proof** (*intro ext*)

```

fix l1 l2 :: 'a list
show list-eq (=) l1 l2 = (l1 = l2)
  apply (induct (=) :: 'a ⇒ - l1 l2 rule: list-eq.induct)
  apply simp-all
  done
qed

```

```

lemma param-list-equals[param]:
   $\llbracket ((=), (=)) \in R \rightarrow R \rightarrow Id \rrbracket$ 
   $\implies ((=), (=)) \in \langle R \rangle list-rel \rightarrow \langle R \rangle list-rel \rightarrow Id$ 
  unfolding id-list-eq-aux[symmetric]
  by (parametricity)

```

```

lemma param-tl[param]:
   $(tl, tl) \in \langle R \rangle list-rel \rightarrow \langle R \rangle list-rel$ 
  unfolding tl-def[abs-def]
  by (parametricity)

```

```

primrec list-all-rec where
  list-all-rec P []  $\longleftrightarrow$  True
| list-all-rec P (a#l)  $\longleftrightarrow$  P a  $\wedge$  list-all-rec P l

```

```

primrec list-ex-rec where
  list-ex-rec P []  $\longleftrightarrow$  False
| list-ex-rec P (a#l)  $\longleftrightarrow$  P a  $\vee$  list-ex-rec P l

```

```

lemma list-all-rec-eq:  $(\forall x \in set\ l. P\ x) = list-all-rec\ P\ l$ 
  by (induct l) auto

```

```

lemma list-ex-rec-eq:  $(\exists x \in set\ l. P\ x) = list-ex-rec\ P\ l$ 
  by (induct l) auto

```

```

lemma param-list-ball[param]:
   $\llbracket (P, P') \in (Ra \rightarrow Id); (l, l') \in \langle Ra \rangle list-rel \rrbracket$ 
   $\implies (\forall x \in set\ l. P\ x, \forall x \in set\ l'. P'\ x) \in Id$ 
  unfolding list-all-rec-eq
  unfolding list-all-rec-def
  by (parametricity)

```

```

lemma param-list-bex[param]:
   $\llbracket (P, P') \in (Ra \rightarrow Id); (l, l') \in \langle Ra \rangle list-rel \rrbracket$ 
   $\implies (\exists x \in set\ l. P\ x, \exists x \in set\ l'. P'\ x) \in Id$ 
  unfolding list-ex-rec-eq[abs-def]
  unfolding list-ex-rec-def
  by (parametricity)

```

```

lemma param-rev[param]:  $(rev, rev) \in \langle R \rangle list-rel \rightarrow \langle R \rangle list-rel$ 
  unfolding rev-def

```

by (*parametricity*)

**lemma** *param-foldli*[*param*]: (*foldli*, *foldli*)  
 $\in \langle Re \rangle list\text{-rel} \rightarrow (Rs \rightarrow Id) \rightarrow (Re \rightarrow Rs \rightarrow Rs) \rightarrow Rs \rightarrow Rs$   
**unfolding** *foldli-def*  
**by** *parametricity*

**lemma** *param-foldri*[*param*]: (*foldri*, *foldri*)  
 $\in \langle Re \rangle list\text{-rel} \rightarrow (Rs \rightarrow Id) \rightarrow (Re \rightarrow Rs \rightarrow Rs) \rightarrow Rs \rightarrow Rs$   
**unfolding** *foldri-def*[*abs-def*]  
**by** *parametricity*

**lemma** *param-nth*[*param*]:  
**assumes** *I*:  $i' < \text{length } l'$   
**assumes** *IR*:  $(i, i') \in nat\text{-rel}$   
**assumes** *LR*:  $(l, l') \in \langle R \rangle list\text{-rel}$   
**shows**  $(l!i, l'!i') \in R$   
**using** *LR I IR*  
**by** (*induct arbitrary: i i' rule: list-rel-induct*)  
*(auto simp: nth.simps split: nat.split)*

**lemma** *param-replicate*[*param*]:  
 $(\text{replicate}, \text{replicate}) \in nat\text{-rel} \rightarrow R \rightarrow \langle R \rangle list\text{-rel}$   
**unfolding** *replicate-def* **by** *parametricity*

**term** *list-update*

**lemma** *param-list-update*[*param*]:  
 $(\text{list-update}, \text{list-update}) \in \langle Ra \rangle list\text{-rel} \rightarrow nat\text{-rel} \rightarrow Ra \rightarrow \langle Ra \rangle list\text{-rel}$   
**unfolding** *list-update-def*[*abs-def*] **by** *parametricity*

**lemma** *param-zip*[*param*]:  
 $(\text{zip}, \text{zip}) \in \langle Ra \rangle list\text{-rel} \rightarrow \langle Rb \rangle list\text{-rel} \rightarrow \langle \langle Ra, Rb \rangle prod\text{-rel} \rangle list\text{-rel}$   
**unfolding** *zip-def* **by** *parametricity*

**lemma** *param-upt*[*param*]:  
 $(\text{upt}, \text{upt}) \in nat\text{-rel} \rightarrow nat\text{-rel} \rightarrow \langle nat\text{-rel} \rangle list\text{-rel}$   
**unfolding** *upt-def*[*abs-def*] **by** *parametricity*

**lemma** *param-concat*[*param*]: (*concat*, *concat*)  $\in$   
 $\langle \langle R \rangle list\text{-rel} \rangle list\text{-rel} \rightarrow \langle R \rangle list\text{-rel}$   
**unfolding** *concat-def*[*abs-def*] **by** *parametricity*

**lemma** *param-all-interval-nat*[*param*]:  
 $(\text{List.all-interval-nat}, \text{List.all-interval-nat})$   
 $\in (nat\text{-rel} \rightarrow bool\text{-rel}) \rightarrow nat\text{-rel} \rightarrow nat\text{-rel} \rightarrow bool\text{-rel}$   
**unfolding** *List.all-interval-nat-def*[*abs-def*]  
**apply** *parametricity*  
**apply** *simp*  
**done**

**lemma** *param-dropWhile*[*param*]:  
(*dropWhile*, *dropWhile*)  $\in$  ( $a \rightarrow \text{bool-rel}$ )  $\rightarrow$   $\langle a \rangle \text{list-rel} \rightarrow \langle a \rangle \text{list-rel}$   
**unfolding** *dropWhile-def* **by** *parametricity*

**lemma** *param-takeWhile*[*param*]:  
(*takeWhile*, *takeWhile*)  $\in$  ( $a \rightarrow \text{bool-rel}$ )  $\rightarrow$   $\langle a \rangle \text{list-rel} \rightarrow \langle a \rangle \text{list-rel}$   
**unfolding** *takeWhile-def* **by** *parametricity*

**end**

## Chapter 2

# Automatic Refinement

### 2.1 Automatic Refinement Tool

```
theory Autoref-Tool
imports
  Autoref-Translate
  Autoref-Gen-Algo
  Autoref-Relator-Interface
begin
```

#### 2.1.1 Standard setup

Declaration of standard phases

```
declaration ⟨fn phi => let open Autoref-Phases in
  I
  #> register-phase id-op 10 Autoref-Id-Ops.id-phase phi
  #> register-phase rel-inf 20
     Autoref-Rel-Inf.roi-phase phi
  #> register-phase fix-rel 22
     Autoref-Fix-Rel.phase phi
  #> register-phase trans 30
     Autoref-Translate.trans-phase phi
end
  ⟩
```

Main method

```
method-setup autoref = ⟨let
  open Refine-Util
  val autoref-flags =
    parse-bool-config trace Autoref-Phases.cfg-trace
    || parse-bool-config debug Autoref-Phases.cfg-debug
    || parse-bool-config keep-goal Autoref-Phases.cfg-keep-goal

  val autoref-phases =
```

```

Args.*** phases |-- Args.colon |-- Scan.repeat1 Args.name

in
  parse-paren-lists autoref-flags
  |-- Scan.option (Scan.lift (autoref-phases)) >>
  ( fn phases => fn ctxt => SIMPLE-METHOD' (
    (
      case phases of
        NONE => Autoref-Phases.all-phases-tac
      | SOME names => Autoref-Phases.phases-tacN names
    ) (Autoref-Phases.init-data ctxt)
    (* TODO: If we want more fine-grained initialization here, solvers have
      to depend on phases, or on data that they initialize if necessary *)
  ))
end

› Automatic Refinement

```

## 2.1.2 Tools

```

setup ‹
  let
    fun higher-order-rl-of ctxt thm = case Thm.concl-of thm of
      @{\mpat Trueprop ((-,?t)∈-)} => let
        val (f,args) = strip-comb t
      in
        if length args = 0 then
          thm
        else let
          val cT = TVar(('c,0), @{\sort type})
          val c = Var ((c,0),cT)
          val R = Var ((R,0), HOLogic.mk-setT (HOLogic.mk-prodT (cT, fastype-of
f)))
          val goal =
            HOLogic.mk-mem (HOLogic.mk-prod (c,f), R)
            |> HOLogic.mk-Trueprop
          val goal-ctxt = Variable.declare-term goal ctxt
          val res-thm =
            Goal.prove-internal ctxt [] (Thm.cterm-of ctxt goal)
            (fn - =>
              REPEAT (resolve-tac goal-ctxt @{\thms fun-relI} 1)
              THEN (resolve-tac goal-ctxt [thm] 1)
              THEN (ALLGOALS (assume-tac goal-ctxt)))
            in
              res-thm
            end
          end
        | - => raise THM(Expected autoref rule,~1,[thm])

```

```

    val higher-order-rl-attr =
      Thm.rule-attribute [] (higher-order-rl-of o Context.proof-of)
  in
    Attrib.setup @{binding autoref-higher-order-rule}
      (Scan.succeed higher-order-rl-attr) Autoref: Convert rule to higher-order form
  end
  ›

```

### 2.1.3 Advanced Debugging

```

method-setup autoref-trans-step-keep = ‹
  Scan.succeed (fn ctxt => SIMPLE-METHOD' (
    Autoref-Translate.trans-dbg-step-tac (Autoref-Phases.init-data ctxt)
  ))
  › Single translation step, leaving unsolved side-conditions

```

```

method-setup autoref-trans-step = ‹
  Scan.succeed (fn ctxt => SIMPLE-METHOD' (
    Autoref-Translate.trans-step-tac (Autoref-Phases.init-data ctxt)
  ))
  › Single translation step

```

```

method-setup autoref-trans-step-only = ‹
  Scan.succeed (fn ctxt => SIMPLE-METHOD' (
    Autoref-Translate.trans-step-only-tac (Autoref-Phases.init-data ctxt)
  ))
  › Single translation step, not attempting to solve side-conditions

```

```

method-setup autoref-side = ‹
  Scan.succeed (fn ctxt => SIMPLE-METHOD' (
    Autoref-Translate.side-dbg-tac (Autoref-Phases.init-data ctxt)
  ))
  › Solve side condition, leave unsolved subgoals

```

```

method-setup autoref-try-solve = ‹
  Scan.succeed (fn ctxt => SIMPLE-METHOD' (
    Autoref-Fix-Rel.try-solve-tac (Autoref-Phases.init-data ctxt)
  ))
  › Try to solve constraint and trace debug information

```

```

method-setup autoref-solve-step = ‹
  Scan.succeed (fn ctxt => SIMPLE-METHOD' (
    Autoref-Fix-Rel.solve-step-tac (Autoref-Phases.init-data ctxt)
  ))
  › Single-step of constraint solver

```

```

method-setup autoref-id-op = ‹
  Scan.succeed (fn ctxt => SIMPLE-METHOD' (

```

```

    Autoref-Id-Ops.id-tac ctxt
  ))
>

method-setup autoref-solve-id-op = <
  Scan.succeed (fn ctxt => SIMPLE-METHOD' (
    Autoref-Id-Ops.id-tac (Config.put Autoref-Id-Ops.cfg-ss-id-op false ctxt)
  ))
>

```

```

ML <
  structure Autoref-Debug = struct
    fun print-thm-pairs ctxt = let
      val ctxt = Autoref-Phases.init-data ctxt
      val p = Autoref-Fix-Rel.thm-pairsD-get ctxt
        |> Autoref-Fix-Rel.pretty-thm-pairs ctxt
        |> Pretty.string-of
    in
      warning p
    end

    fun print-thm-pairs-matching ctxt cpat = let
      val pat = Thm.term-of cpat
      val ctxt = Autoref-Phases.init-data ctxt
      val thy = Proof-Context.theory-of ctxt

      fun matches NONE = false
        | matches (SOME (-(f,-))) = Pattern.matches thy (pat,f)

      val p = Autoref-Fix-Rel.thm-pairsD-get ctxt
        |> filter (matches o #1)
        |> Autoref-Fix-Rel.pretty-thm-pairs ctxt
        |> Pretty.string-of
    in
      warning p
    end
  end
>

```

General casting-tag, that allows type-casting on concrete level, while being identity on abstract level.

**definition** [*simp*]:  $CAST \equiv id$

**lemma** [*autoref-itype*]:  $CAST ::_i I \rightarrow_i I$  by *simp*

Hide internal stuff

**notation** (*input*) *rel-ANNOT* (**infix** <:::;<sub>r</sub>> 10)

**notation** (*input*) *ind-ANNOT* (**infix** <:::#<sub>r</sub>> 10)



```

locale autoref-syn begin
  notation (input) APP (infixl <$> 900)
  notation (input) rel-ANNOT (infix <::> 10)
  notation (input) ind-ANNOT (infix <::#> 10)
  notation OP (<OP>)
  notation (input) ABS (binder < $\lambda''$ > 10)
end

no-notation (input) APP (infixl <$> 900)
no-notation (input) ABS (binder < $\lambda''$ > 10)

no-notation (input) rel-ANNOT (infix <::> 10)
no-notation (input) ind-ANNOT (infix <::#> 10)

hide-const (open) PROTECT ANNOT OP APP ABS ID-FAIL rel-annot ind-annot

end

```

## 2.2 Standard HOL Bindings

```

theory Autoref-Bindings-HOL
imports Tool/Autoref-Tool
begin

```

### 2.2.1 Structural Expansion

In some situations, autoref imitates the operations on typeclasses and the typeclass hierarchy. This may result in structural mismatches, e.g., a hashcode side-condition may look like:

```
is-hashcode (prod-eq (=) (=)) hashcode
```

This cannot be discharged by the rule

```
is-hashcode (=) hashcode
```

In order to handle such cases, we introduce a set of simplification lemmas that expand the structure of an operator as far as possible. These lemmas are integrated into a tagged solver, that can prove equality between operators modulo structural expansion.

```

definition [simp]: STRUCT-EQ-tag  $x\ y \equiv x = y$ 
lemma STRUCT-EQ-tagI:  $x=y \implies \text{STRUCT-EQ-tag } x\ y$  by simp

```

```

ML <
  structure Autoref-Struct-Expand = struct
    structure autoref-struct-expand = Named-Thms (

```

```

    val name = @{binding autoref-struct-expand}
    val description = Autoref: Structural expansion lemmas
  )

  fun expand-tac ctxt = let
    val ss = put-simpset HOL-basic-ss ctxt addsimps autoref-struct-expand.get ctxt
  in
    SOLVED' (asm-simp-tac ss)
  end

  val setup = autoref-struct-expand.setup
  val decl-setup = fn phi =>
    Tagged-Solver.declare-solver @ { thms STRUCT-EQ-tagI } @ { binding STRUCT-EQ }

    Autoref: Equality modulo structural expansion
    (expand-tac) phi

  end
}

```

```

setup Autoref-Struct-Expand.setup
declaration Autoref-Struct-Expand.decl-setup

```

Sometimes, also relators must be expanded. Usually to check them to be the identity relator

```

definition [simp]: REL-IS-ID R  $\equiv$  R=Id
definition [simp]: REL-FORCE-ID R  $\equiv$  R=Id
lemma REL-IS-ID-trigger: R=Id  $\implies$  REL-IS-ID R by simp
lemma REL-FORCE-ID-trigger: R=Id  $\implies$  REL-FORCE-ID R by simp

```

```

declaration < Tagged-Solver.add-triggers
  Relators.relator-props-solver @ { thms REL-IS-ID-trigger } >

```

```

declaration < Tagged-Solver.add-triggers
  Relators.force-relator-props-solver @ { thms REL-FORCE-ID-trigger } >

```

```

abbreviation PREFER-id R  $\equiv$  PREFER REL-IS-ID R

```

```

lemmas [autoref-rel-intf] = REL-INTFI[of fun-rel i-fun]

```

## 2.2.2 Booleans

```

consts

```

*i-bool* :: interface

**lemmas** [autoref-rel-intf] = REL-INTFI[of bool-rel i-bool]

**lemma** [autoref-itype]:

True ::<sub>i</sub> i-bool  
 False ::<sub>i</sub> i-bool  
 conj ::<sub>i</sub> i-bool →<sub>i</sub> i-bool →<sub>i</sub> i-bool  
 (←→) ::<sub>i</sub> i-bool →<sub>i</sub> i-bool →<sub>i</sub> i-bool  
 (→) ::<sub>i</sub> i-bool →<sub>i</sub> i-bool →<sub>i</sub> i-bool  
 disj ::<sub>i</sub> i-bool →<sub>i</sub> i-bool →<sub>i</sub> i-bool  
 Not ::<sub>i</sub> i-bool →<sub>i</sub> i-bool  
 case-bool ::<sub>i</sub> I →<sub>i</sub> I →<sub>i</sub> i-bool →<sub>i</sub> I  
 old.rec-bool ::<sub>i</sub> I →<sub>i</sub> I →<sub>i</sub> i-bool →<sub>i</sub> I  
**by auto**

**lemma** autoref-bool[autoref-rules]:

(x,x) ∈ bool-rel  
 (conj,conj) ∈ bool-rel → bool-rel → bool-rel  
 (disj,disj) ∈ bool-rel → bool-rel → bool-rel  
 (Not,Not) ∈ bool-rel → bool-rel  
 (case-bool,case-bool) ∈ R → R → bool-rel → R  
 (old.rec-bool,old.rec-bool) ∈ R → R → bool-rel → R  
 ((←→), (←→)) ∈ bool-rel → bool-rel → bool-rel  
 ((→), (→)) ∈ bool-rel → bool-rel → bool-rel  
**by** (auto split: bool.split simp: rec-bool-is-case)

### 2.2.3 Standard Type Classes

**context begin interpretation** autoref-syn .

We allow these operators for all interfaces.

**lemma** [autoref-itype]:

(<) ::<sub>i</sub> I →<sub>i</sub> I →<sub>i</sub> i-bool  
 (≤) ::<sub>i</sub> I →<sub>i</sub> I →<sub>i</sub> i-bool  
 (=) ::<sub>i</sub> I →<sub>i</sub> I →<sub>i</sub> i-bool  
 (+) ::<sub>i</sub> I →<sub>i</sub> I →<sub>i</sub> I  
 (−) ::<sub>i</sub> I →<sub>i</sub> I →<sub>i</sub> I  
 (div) ::<sub>i</sub> I →<sub>i</sub> I →<sub>i</sub> I  
 (mod) ::<sub>i</sub> I →<sub>i</sub> I →<sub>i</sub> I  
 (\*) ::<sub>i</sub> I →<sub>i</sub> I →<sub>i</sub> I  
 0 ::<sub>i</sub> I  
 1 ::<sub>i</sub> I  
 numeral x ::<sub>i</sub> I  
 uminus ::<sub>i</sub> I →<sub>i</sub> I  
**by auto**

**lemma** pat-num-generic[autoref-op-pat]:

0 ≡ OP 0 ::<sub>i</sub> I

$1 \equiv OP\ 1 \ :::{}_i I$   
 $numeral\ x \equiv (OP\ (numeral\ x) \ :::{}_i I)$   
 by *simp-all*

**lemma** [*autoref-rules*]:  
**assumes** *PRIO-TAG-GEN-ALGO*  
**shows**  $((<), (<)) \in Id \rightarrow Id \rightarrow bool\text{-rel}$   
**and**  $((\leq), (\leq)) \in Id \rightarrow Id \rightarrow bool\text{-rel}$   
**and**  $((=), (=)) \in Id \rightarrow Id \rightarrow bool\text{-rel}$   
**and**  $(numeral\ x, OP\ (numeral\ x) \ ::: Id) \in Id$   
**and**  $(uminus, uminus) \in Id \rightarrow Id$   
**and**  $(0, 0) \in Id$   
**and**  $(1, 1) \in Id$   
 by *auto*

## 2.2.4 Functional Combinators

**lemma** [*autoref-itype*]:  $id \ :::{}_i I \rightarrow_i I$  **by** *simp*  
**lemma** *autoref-id*[*autoref-rules*]:  $(id, id) \in R \rightarrow R$  **by** *auto*

**term** (*o*)  
**lemma** [*autoref-itype*]:  $(\circ) \ :::{}_i (Ia \rightarrow_i Ib) \rightarrow_i (Ic \rightarrow_i Ia) \rightarrow_i Ic \rightarrow_i Ib$   
 by *simp*  
**lemma** *autoref-comp*[*autoref-rules*]:  
 $((o), (o)) \in (Ra \rightarrow Rb) \rightarrow (Rc \rightarrow Ra) \rightarrow Rc \rightarrow Rb$   
 by (*auto dest: fun-relD*)

**lemma** [*autoref-itype*]:  $If \ :::{}_i i\text{-bool} \rightarrow_i I \rightarrow_i I \rightarrow_i I$  **by** *simp*  
**lemma** *autoref-If*[*autoref-rules*]:  $(If, If) \in Id \rightarrow R \rightarrow R \rightarrow R$  **by** *auto*  
**lemma** *autoref-If-cong*[*autoref-rules*]:  
**assumes**  $(c', c) \in Id$   
**assumes** *REMOVE-INTERNAL*  $c \implies (t', t) \in R$   
**assumes**  $\neg$  *REMOVE-INTERNAL*  $c \implies (e', e) \in R$   
**shows**  $(If\ c'\ t'\ e', (OP\ If \ ::: Id \rightarrow R \rightarrow R \rightarrow R)) \$c \$t \$e \in R$   
**using** *assms* **by** (*auto*)

**lemma** [*autoref-itype*]:  $Let \ :::{}_i Ix \rightarrow_i (Ix \rightarrow_i Iy) \rightarrow_i Iy$  **by** *auto*  
**lemma** *autoref-Let*:  
 $(Let, Let) \in Ra \rightarrow (Ra \rightarrow Rr) \rightarrow Rr$   
**by** (*auto dest: fun-relD*)

**lemma** *autoref-Let-cong*[*autoref-rules*]:  
**assumes**  $(x', x) \in Ra$   
**assumes**  $\bigwedge y\ y'.\ REMOVE\text{-INTERNAL}\ (x=y) \implies (y', y) \in Ra \implies (f'\ y', f\ \$y) \in Rr$   
**shows**  $(Let\ x'\ f', (OP\ Let \ ::: Ra \rightarrow (Ra \rightarrow Rr) \rightarrow Rr)) \$x \$f \in Rr$   
**using** *assms* **by** (*auto*)

end

### 2.2.5 Unit

**consts** *i-unit* :: *interface*

**lemmas** [*autoref-rel-intf*] = *REL-INTFI*[*of unit-rel i-unit*]

**lemma** [*autoref-rules*]:  $((),()) \in \text{unit-rel}$  **by** *simp*

### 2.2.6 Nat

**consts** *i-nat* :: *interface*

**lemmas** [*autoref-rel-intf*] = *REL-INTFI*[*of nat-rel i-nat*]

**context begin interpretation** *autoref-syn* .

**lemma** *pat-num-nat*[*autoref-op-pat*]:

$0::\text{nat} \equiv \text{OP } 0 \text{ } ::_i \text{ } i\text{-nat}$

$1::\text{nat} \equiv \text{OP } 1 \text{ } ::_i \text{ } i\text{-nat}$

$(\text{numeral } x)::\text{nat} \equiv (\text{OP } (\text{numeral } x) \text{ } ::_i \text{ } i\text{-nat})$

**by** *simp-all*

**lemma** *autoref-nat*[*autoref-rules*]:

$(0, 0::\text{nat}) \in \text{nat-rel}$

$(\text{Suc}, \text{Suc}) \in \text{nat-rel} \rightarrow \text{nat-rel}$

$(1, 1::\text{nat}) \in \text{nat-rel}$

$(\text{numeral } n::\text{nat}, \text{numeral } n::\text{nat}) \in \text{nat-rel}$

$((<), (< :: \text{nat} \Rightarrow -)) \in \text{nat-rel} \rightarrow \text{nat-rel} \rightarrow \text{bool-rel}$

$((\leq), (\leq :: \text{nat} \Rightarrow -)) \in \text{nat-rel} \rightarrow \text{nat-rel} \rightarrow \text{bool-rel}$

$((=), (= :: \text{nat} \Rightarrow -)) \in \text{nat-rel} \rightarrow \text{nat-rel} \rightarrow \text{bool-rel}$

$((+ :: \text{nat} \Rightarrow -, (+)) \in \text{nat-rel} \rightarrow \text{nat-rel} \rightarrow \text{nat-rel}$

$((- :: \text{nat} \Rightarrow -, (-)) \in \text{nat-rel} \rightarrow \text{nat-rel} \rightarrow \text{nat-rel}$

$((\text{div} :: \text{nat} \Rightarrow -, (\text{div})) \in \text{nat-rel} \rightarrow \text{nat-rel} \rightarrow \text{nat-rel}$

$((*) , (*)) \in \text{nat-rel} \rightarrow \text{nat-rel} \rightarrow \text{nat-rel}$

$((\text{mod}), (\text{mod})) \in \text{nat-rel} \rightarrow \text{nat-rel} \rightarrow \text{nat-rel}$

**by** *auto*

**lemma** *autoref-case-nat*[*autoref-rules*]:

$(\text{case-nat}, \text{case-nat}) \in \text{Ra} \rightarrow (\text{Id} \rightarrow \text{Ra}) \rightarrow \text{Id} \rightarrow \text{Ra}$

**apply** (*intro fun-relI*)

**apply** (*auto split: nat.split dest: fun-relD*)

**done**

**lemma** *autoref-rec-nat*:  $(\text{rec-nat}, \text{rec-nat}) \in \text{R} \rightarrow (\text{Id} \rightarrow \text{R} \rightarrow \text{R}) \rightarrow \text{Id} \rightarrow \text{R}$

**apply** (*intro fun-relI*)

**proof** *goal-cases*

**case**  $(1 \text{ } s \text{ } s' \text{ } f \text{ } f' \text{ } n \text{ } n')$  **thus** *?case*

**apply** (*induct n' arbitrary: n s s'*)

**apply** (*fastforce simp: fun-rel-def*)**+**

**done**

**qed**

**end**

### 2.2.7 Int

**consts** *i-int* :: *interface*  
**lemmas** [*autoref-rel-intf*] = *REL-INTFI*[*of int-rel i-int*]

**context begin interpretation** *autoref-syn* .

**lemma** *pat-num-int*[*autoref-op-pat*]:  
 $0::int \equiv OP\ 0 \ ::_i\ i-int$   
 $1::int \equiv OP\ 1 \ ::_i\ i-int$   
 $(numeral\ x)::int \equiv (OP\ (numeral\ x) \ ::_i\ i-int)$   
**by** *simp-all*

**lemma** *autoref-int*[*autoref-rules (overloaded)*]:

$(0, 0::int) \in int-rel$   
 $(1, 1::int) \in int-rel$   
 $(numeral\ n::int, numeral\ n::int) \in int-rel$   
 $((<), (< \ ::int \Rightarrow -)) \in int-rel \rightarrow int-rel \rightarrow bool-rel$   
 $((\leq), (\leq \ ::int \Rightarrow -)) \in int-rel \rightarrow int-rel \rightarrow bool-rel$   
 $((=), (= \ ::int \Rightarrow -)) \in int-rel \rightarrow int-rel \rightarrow bool-rel$   
 $((+) \ ::int \Rightarrow -, (+)) \in int-rel \rightarrow int-rel \rightarrow int-rel$   
 $((-) \ ::int \Rightarrow -, (-)) \in int-rel \rightarrow int-rel \rightarrow int-rel$   
 $((div) \ ::int \Rightarrow -, (div)) \in int-rel \rightarrow int-rel \rightarrow int-rel$   
 $(uminus, uminus) \in int-rel \rightarrow int-rel$   
 $((*) \ ::int \Rightarrow -, (*)) \in int-rel \rightarrow int-rel \rightarrow int-rel$   
 $((mod), (mod)) \in int-rel \rightarrow int-rel \rightarrow int-rel$   
**by** *auto*

**end**

### 2.2.8 Product

**consts** *i-prod* :: *interface*  $\Rightarrow$  *interface*  $\Rightarrow$  *interface*  
**lemmas** [*autoref-rel-intf*] = *REL-INTFI*[*of prod-rel i-prod*]

**context begin interpretation** *autoref-syn* .

**lemma** *prod-refine*[*autoref-rules*]:

$(Pair, Pair) \in Ra \rightarrow Rb \rightarrow \langle Ra, Rb \rangle prod-rel$   
 $(case-prod, case-prod) \in (Ra \rightarrow Rb \rightarrow Rr) \rightarrow \langle Ra, Rb \rangle prod-rel \rightarrow Rr$   
 $(old.rec-prod, old.rec-prod) \in (Ra \rightarrow Rb \rightarrow Rr) \rightarrow \langle Ra, Rb \rangle prod-rel \rightarrow Rr$   
 $(fst, fst) \in \langle Ra, Rb \rangle prod-rel \rightarrow Ra$   
 $(snd, snd) \in \langle Ra, Rb \rangle prod-rel \rightarrow Rb$   
**by** (*auto dest: fun-relD split: prod.split*  
*simp: prod-rel-def rec-prod-is-case*)

**definition** *prod-eq* *eqa eqb x1 x2*  $\equiv$

*case x1 of (a1,b1)  $\Rightarrow$  case x2 of (a2,b2)  $\Rightarrow$  eqa a1 a2  $\wedge$  eqb b1 b2*

**lemma** *prod-eq-autoref*[*autoref-rules* (**overloaded**)]:  
 $\llbracket GEN-OP\ eqa\ (=)\ (Ra \rightarrow Ra \rightarrow Id); GEN-OP\ eqb\ (=)\ (Rb \rightarrow Rb \rightarrow Id) \rrbracket$   
 $\implies (prod-eq\ eqa\ eqb, (=)) \in \langle Ra, Rb \rangle prod-rel \rightarrow \langle Ra, Rb \rangle prod-rel \rightarrow Id$   
**unfolding** *prod-eq-def*[*abs-def*]  
**by** (*fastforce dest: fun-relD*)

**lemma** *prod-eq-expand*[*autoref-struct-expand*]:  $(=) = prod-eq\ (=)\ (=)$   
**unfolding** *prod-eq-def*[*abs-def*]  
**by** (*auto intro!: ext*)

**end**

### 2.2.9 Option

**consts** *i-option* :: *interface*  $\Rightarrow$  *interface*  
**lemmas** [*autoref-rel-intf*] = *REL-INTFI*[*of option-rel i-option*]

**context begin interpretation** *autoref-syn* .

**lemma** *autoref-opt*[*autoref-rules*]:  
 $(None, None) \in \langle R \rangle option-rel$   
 $(Some, Some) \in R \rightarrow \langle R \rangle option-rel$   
 $(case-option, case-option) \in Rr \rightarrow (R \rightarrow Rr) \rightarrow \langle R \rangle option-rel \rightarrow Rr$   
 $(rec-option, rec-option) \in Rr \rightarrow (R \rightarrow Rr) \rightarrow \langle R \rangle option-rel \rightarrow Rr$   
**by** (*auto split: option.split*)  
*simp: option-rel-def case-option-def*[*symmetric*]  
*dest: fun-relD*)

**lemma** *autoref-the*[*autoref-rules*]:  
**assumes** *SIDE-PRECOND* ( $x \neq None$ )  
**assumes**  $(x', x) \in \langle R \rangle option-rel$   
**shows** (*the*  $x'$ , (*OP the*  $::: \langle R \rangle option-rel \rightarrow R$ ) $\$x$ )  $\in R$   
**using** *assms*  
**by** (*auto simp: option-rel-def*)

**lemma** *autoref-the-default*[*autoref-rules*]:  
 $(the-default, the-default) \in R \rightarrow \langle R \rangle option-rel \rightarrow R$   
**by** *parametricity*

**definition** [*simp*]: *is-None*  $a \equiv case\ a\ of\ None \Rightarrow True \mid - \Rightarrow False$

**lemma** *pat-isNone*[*autoref-op-pat*]:  
 $a = None \equiv (OP\ is-None\ :::_i \langle I \rangle_i i-option \rightarrow_i i-bool)\$a$   
 $None = a \equiv (OP\ is-None\ :::_i \langle I \rangle_i i-option \rightarrow_i i-bool)\$a$   
**by** (*auto intro!: eq-reflection split: option.splits*)

**lemma** *autoref-is-None*[*param, autoref-rules*]:  
 $(is-None, is-None) \in \langle R \rangle option-rel \rightarrow Id$   
**by** (*auto split: option.splits*)

**lemma** *fold-is-None*:  $x = None \longleftrightarrow is-None\ x$  **by** (*cases x*) *auto*

**definition** *option-eq*  $eq\ v1\ v2 \equiv$   
*case*  $(v1, v2)$  *of*  
 $(None, None) \Rightarrow True$   
 $| (Some\ x1, Some\ x2) \Rightarrow eq\ x1\ x2$   
 $| - \Rightarrow False$

**lemma** *option-eq-autoref*[*autoref-rules* (**overloaded**)]:  
 $\llbracket GEN-OP\ eq\ (=)\ (R \rightarrow R \rightarrow Id) \rrbracket$   
 $\implies (option-eq\ eq, (=)) \in \langle R \rangle option-rel \rightarrow \langle R \rangle option-rel \rightarrow Id$   
**unfolding** *option-eq-def*[*abs-def*]  
**by** (*auto dest: fun-relD split: option.splits elim!: option-relE*)

**lemma** *option-eq-expand*[*autoref-struct-expand*]:  
 $(=) = option-eq\ (=)$   
**by** (*auto intro!: ext simp: option-eq-def split: option.splits*)

**end**

## 2.2.10 Sum-Types

**consts** *i-sum* :: *interface*  $\Rightarrow$  *interface*  $\Rightarrow$  *interface*  
**lemmas** [*autoref-rel-intf*] = *REL-INTFI*[*of sum-rel i-sum*]

**context begin interpretation** *autoref-syn* .

**lemma** *autoref-sum*[*autoref-rules*]:  
 $(Inl, Inl) \in Rl \rightarrow \langle Rl, Rr \rangle sum-rel$   
 $(Inr, Inr) \in Rr \rightarrow \langle Rl, Rr \rangle sum-rel$   
 $(case-sum, case-sum) \in (Rl \rightarrow R) \rightarrow (Rr \rightarrow R) \rightarrow \langle Rl, Rr \rangle sum-rel \rightarrow R$   
 $(old.rec-sum, old.rec-sum) \in (Rl \rightarrow R) \rightarrow (Rr \rightarrow R) \rightarrow \langle Rl, Rr \rangle sum-rel \rightarrow R$   
**by** (*fastforce split: sum.split dest: fun-relD*  
*simp: rec-sum-is-case*) $+$

**definition** *sum-eq*  $eql\ eqr\ s1\ s2 \equiv$   
*case*  $(s1, s2)$  *of*  
 $(Inl\ x1, Inl\ x2) \Rightarrow eql\ x1\ x2$   
 $| (Inr\ x1, Inr\ x2) \Rightarrow eqr\ x1\ x2$   
 $| - \Rightarrow False$

**lemma** *sum-eq-autoref*[*autoref-rules* (**overloaded**)]:  
 $\llbracket GEN-OP\ eql\ (=)\ (Rl \rightarrow Rl \rightarrow Id); GEN-OP\ eqr\ (=)\ (Rr \rightarrow Rr \rightarrow Id) \rrbracket$   
 $\implies (sum-eq\ eql\ eqr, (=)) \in \langle Rl, Rr \rangle sum-rel \rightarrow \langle Rl, Rr \rangle sum-rel \rightarrow Id$   
**unfolding** *sum-eq-def*[*abs-def*]  
**by** (*fastforce dest: fun-relD elim!: sum-relE*)

**lemma** *sum-eq-expand*[*autoref-struct-expand*]:  $(=) = sum-eq\ (=)\ (=)$   
**by** (*auto intro!: ext simp: sum-eq-def split: sum.splits*)



**lemmas** [autoref-rules] = is-Inl-param is-Inr-param

**lemma** autoref-sum-Projl[autoref-rules]:

[[SIDE-PRECOND (is-Inl s); (s',s)∈⟨Ra,Rb⟩sum-rel]]  
 ⇒ (Sum-Type.sum.proj1 s', (OP Sum-Type.sum.proj1 ∷ ⟨Ra,Rb⟩sum-rel →  
 Ra)§s)∈Ra  
 by simp parametricity

**lemma** autoref-sum-Projr[autoref-rules]:

[[SIDE-PRECOND (is-Inr s); (s',s)∈⟨Ra,Rb⟩sum-rel]]  
 ⇒ (Sum-Type.sum.proj2 s', (OP Sum-Type.sum.proj2 ∷ ⟨Ra,Rb⟩sum-rel →  
 Rb)§s)∈Rb  
 by simp parametricity

**end**

### 2.2.11 List

**consts** i-list :: interface ⇒ interface

**lemmas** [autoref-rel-intf] = REL-INTFI[of list-rel i-list]

**context** begin interpretation autoref-syn .

**lemma** autoref-append[autoref-rules]:

(append, append)∈⟨R⟩list-rel → ⟨R⟩list-rel → ⟨R⟩list-rel  
 by (auto simp: list-rel-def list-all2-appendI)

**lemma** refine-list[autoref-rules]:

(Nil,Nil)∈⟨R⟩list-rel  
 (Cons,Cons)∈R → ⟨R⟩list-rel → ⟨R⟩list-rel  
 (case-list,case-list)∈Rr→(R→⟨R⟩list-rel→Rr)→⟨R⟩list-rel→Rr  
**apply** (force dest: fun-relD split: list.split)+  
**done**

**lemma** autoref-rec-list[autoref-rules]: (rec-list,rec-list)

∈ Ra → (Rb → ⟨Rb⟩list-rel → Ra → Ra) → ⟨Rb⟩list-rel → Ra

**proof** (intro fun-relI, goal-cases)

**case** prems: (1 a a' f f' l l')

**from** prems(3) **show** ?case

**using** prems(1,2)

**apply** (induct arbitrary: a a')

**apply** simp

**apply** (fastforce dest: fun-relD)

**done**

**qed**

**lemma** refine-map[autoref-rules]:

(map,map)∈(R1→R2) → ⟨R1⟩list-rel → ⟨R2⟩list-rel

```

using [[autoref-sbias = -1]]
unfolding map-rec[abs-def]
by autoref

```

```

lemma refine-fold[autoref-rules]:
  (fold,fold) ∈ (Re → Rs → Rs) → ⟨Re⟩list-rel → Rs → Rs
  (foldl,foldl) ∈ (Rs → Re → Rs) → Rs → ⟨Re⟩list-rel → Rs
  (foldr,foldr) ∈ (Re → Rs → Rs) → ⟨Re⟩list-rel → Rs → Rs
unfolding List.fold-def List.foldr-def List.foldl-def
by (autoref)+

```

```

schematic-goal autoref-take[autoref-rules]: (take,take) ∈ (?R::(-×-) set)
unfolding take-def by autoref
schematic-goal autoref-drop[autoref-rules]: (drop,drop) ∈ (?R::(-×-) set)
unfolding drop-def by autoref
schematic-goal autoref-length[autoref-rules]:
  (length,length) ∈ (?R::(-×-) set)
unfolding size-list-overloaded-def size-list-def
by (autoref)

```

```

lemma autoref-nth[autoref-rules]:
assumes (l,l') ∈ ⟨R⟩list-rel
assumes (i,i') ∈ Id
assumes SIDE-PRECOND (i' < length l')
shows (nth l i, (OP nth ::: ⟨R⟩list-rel → Id → R)$l$i') ∈ R
unfolding ANNOT-def
using assms
apply (induct arbitrary: i i')
apply simp
apply (case-tac i')
apply auto
done

```

```

fun list-eq :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a list ⇒ bool where
  list-eq eq [] [] ←→ True
| list-eq eq (a#l) (a'#l')
  ←→ (if eq a a' then list-eq eq l l' else False)
| list-eq - - - ←→ False

```

```

lemma autoref-list-eq-aux:
  (list-eq,list-eq) ∈
    (R → R → Id) → ⟨R⟩list-rel → ⟨R⟩list-rel → Id
proof (intro fun-rell, goal-cases)
case (1 eq eq' l1 l1' l2 l2')
thus ?case
apply -
apply (induct eq' l1' l2' arbitrary: l1 l2 rule: list-eq.induct)
apply simp
apply (case-tac l1)

```

```

apply simp
apply (case-tac l2)
apply (simp)
apply (auto dest: fun-relD) []
apply (case-tac l1)
apply simp
apply simp
apply (case-tac l2)
apply simp
apply simp
done
qed

lemma list-eq-expand[autoref-struct-expand]: (=) = (list-eq (=))
proof (intro ext)
  fix l1 l2 :: 'a list
  show (l1 = l2)  $\longleftrightarrow$  list-eq (=) l1 l2
    apply (induct (=) :: 'a  $\Rightarrow$  - l1 l2 rule: list-eq.induct)
    apply simp-all
    done
qed

lemma autoref-list-eq[autoref-rules (overloaded)]:
  GEN-OP eq (=) (R  $\rightarrow$  R  $\rightarrow$  Id)  $\Longrightarrow$  (list-eq eq, (=))
   $\in \langle R \rangle$ list-rel  $\rightarrow \langle R \rangle$ list-rel  $\rightarrow$  Id
  unfolding autoref-tag-defs
  apply (subst list-eq-expand)
  apply (parametricity add: autoref-list-eq-aux)
  done

lemma autoref-hd[autoref-rules]:
   $\llbracket$  SIDE-PRECOND ( $l' \neq []$ ); ( $l, l'$ )  $\in \langle R \rangle$ list-rel  $\rrbracket \Longrightarrow$ 
  (hd l, (OP hd :::  $\langle R \rangle$ list-rel  $\rightarrow$  R)$l')  $\in$  R
  apply (simp add: ANNOT-def)
  apply (cases l')
  apply simp
  apply (cases l)
  apply auto
  done

lemma autoref-tl[autoref-rules]:
  ( $tl, tl$ )  $\in \langle R \rangle$ list-rel  $\rightarrow \langle R \rangle$ list-rel
  unfolding tl-def[abs-def]
  by autoref

definition [simp]: is-Nil a  $\equiv$  case a of []  $\Rightarrow$  True | -  $\Rightarrow$  False

lemma is-Nil-pat[autoref-op-pat]:
  a=[]  $\equiv$  (OP is-Nil :::  $\langle I \rangle_i$ i-list  $\rightarrow_i$  i-bool)$a

```

$\square = a \equiv (OP \text{ is-Nil} ::: \langle I \rangle_i i\text{-list} \rightarrow_i i\text{-bool}) \$ a$   
**by** (*auto intro!*: *eq-reflection split: list.splits*)

**lemma** *autoref-is-Nil*[*param, autoref-rules*]:  
 $(\text{is-Nil}, \text{is-Nil}) \in \langle R \rangle \text{list-rel} \rightarrow \text{bool-rel}$   
**by** (*auto split: list.splits*)

**lemma** *conv-to-is-Nil*:  
 $l = \square \longleftrightarrow \text{is-Nil } l$   
 $\square = l \longleftrightarrow \text{is-Nil } l$   
**unfolding** *is-Nil-def* **by** (*auto split: list.split*)

**lemma** *autoref-butlast*[*param, autoref-rules*]:  
 $(\text{butlast}, \text{butlast}) \in \langle R \rangle \text{list-rel} \rightarrow \langle R \rangle \text{list-rel}$   
**unfolding** *butlast-def conv-to-is-Nil*  
**by** *parametricity*

**definition** [*simp*]: *op-list-singleton*  $x \equiv [x]$   
**lemma** *op-list-singleton-pat*[*autoref-op-pat*]:  
 $[x] \equiv (OP \text{ op-list-singleton} ::: \langle I \rangle_i I \rightarrow_i \langle I \rangle_i i\text{-list}) \$ x$  **by** *simp*  
**lemma** *autoref-list-singleton*[*autoref-rules*]:  
 $(\lambda a. [a], \text{op-list-singleton}) \in R \rightarrow \langle R \rangle \text{list-rel}$   
**by** *auto*

**definition** [*simp*]: *op-list-append-elem*  $s \ x \equiv s @ [x]$

**lemma** *pat-list-append-elem*[*autoref-op-pat*]:  
 $s @ [x] \equiv (OP \text{ op-list-append-elem} ::: \langle I \rangle_i i\text{-list} \rightarrow_i I \rightarrow_i \langle I \rangle_i i\text{-list}) \$ s \$ x$   
**by** (*simp add: relAPP-def*)

**lemma** *autoref-list-append-elem*[*autoref-rules*]:  
 $(\lambda s \ x. s @ [x], \text{op-list-append-elem}) \in \langle R \rangle \text{list-rel} \rightarrow R \rightarrow \langle R \rangle \text{list-rel}$   
**unfolding** *op-list-append-elem-def [abs-def]* **by** *parametricity*

**declare** *param-rev*[*autoref-rules*]

**declare** *param-all-interval-nat*[*autoref-rules*]

**lemma** [*autoref-op-pat*]:  
 $(\forall i < u. P \ i) \equiv OP \ \text{List.all-interval-nat} \ P \ 0 \ u$   
 $(\forall i \leq u. P \ i) \equiv OP \ \text{List.all-interval-nat} \ P \ 0 \ (\text{Suc } u)$   
 $(\forall i < u. l \leq i \longrightarrow P \ i) \equiv OP \ \text{List.all-interval-nat} \ P \ l \ u$   
 $(\forall i \leq u. l \leq i \longrightarrow P \ i) \equiv OP \ \text{List.all-interval-nat} \ P \ l \ (\text{Suc } u)$   
**by** (*auto intro!*: *eq-reflection simp: List.all-interval-nat-def*)

**lemmas** [*autoref-rules*] = *param-dropWhile param-takeWhile*

**end**

### 2.2.12 Examples

Be careful to make the concrete type a schematic type variable. The default behaviour of *schematic-lemma* makes it a fixed variable, that will not unify with the inferred term!

```
schematic-goal
  (?f::?'c,[1,2,3]@[4::nat])∈?R
  by autoref
```

```
schematic-goal
  (?f::?'c,[1::nat,
    2,3,4,5,6,7,8,9,0,1,43,5,5,435,5,1,5,6,5,6,5,63,56
  ]
  )∈?R
  apply (autoref)
  done
```

```
schematic-goal
  (?f::?'c,[1,2,3] = [])∈?R
  by autoref
```

When specifying custom refinement rules on the fly, be careful with the type-inference between *notes* and *shows*. It's too easy to „decouple” the type *'a* in the *autoref*-rule and the actual goal, as shown below!

```
schematic-goal
  notes [autoref-rules] = IdI[where 'a='a]
  notes [autoref-itype] = itypeI[where 't='a::numeral and I=i-std]
  shows (?f::?'c, hd [a,b,c::'a::numeral])∈?R
```

The *autoref*-rule is bound with type *'a::typ*, while the goal statement has *'a::numeral!*

```
apply (autoref (keep-goal))
```

We get an unsolved goal, as it finds no rule to translate *a*

```
oops
```

Here comes the correct version. Note the duplicate sort annotation of type *'a*:

```
schematic-goal
  notes [autoref-rules-raw] = IdI[where 'a='a::numeral]
  notes [autoref-itype] = itypeI[where 't='a::numeral and I=i-std]
  shows (?f::?'c, hd [a,b,c::'a::numeral])∈?R
  by (autoref)
```

Special cases of equality: Note that we do not require equality on the element type!

```
schematic-goal
```

```

assumes [autoref-rules]: (ai,a)∈⟨R⟩option-rel
shows (?f::?'c, a = None)∈?R
apply (autoref (keep-goal))
done

```

**schematic-goal**

```

assumes [autoref-rules]: (ai,a)∈⟨R⟩list-rel
shows (?f::?'c, [] = a)∈?R
apply (autoref (keep-goal))
done

```

**schematic-goal**

```

shows (?f::?'c, [1,2] = [2,3::nat])∈?R
apply (autoref (keep-goal))
done

```

**end**

## 2.3 Entry Point for the Automatic Refinement Tool

**theory** *Automatic-Refinement*

**imports**

*Tool/Autoref-Tool*

*Autoref-Bindings-HOL*

**begin**

The automatic refinement tool should be used by importing this theory

### 2.3.1 Convenience

The following lemmas can be used to add tags to theorems

**lemma** *PREFER-I*:  $P\ x \implies \text{PREFER } P\ x$  **by** *simp*

**lemma** *PREFER-D*:  $\text{PREFER } P\ x \implies P\ x$  **by** *simp*

**lemmas** *PREFER-sv-D* = *PREFER-D*[of *single-valued*]

**lemma** *PREFER-id-D*:  $\text{PREFER-id } R \implies R = \text{Id}$  **by** *simp*

**abbreviation** *PREFER-RUNIV*  $\equiv \text{PREFER } (\lambda R. \text{Range } R = \text{UNIV})$

**lemmas** *PREFER-RUNIV-D* = *PREFER-D*[of  $(\lambda R. \text{Range } R = \text{UNIV})$ ]

**lemma** *SIDE-GEN-ALGO-D*:  $\text{SIDE-GEN-ALGO } P \implies P$  **by** *simp*

**lemma** *GEN-OP-D*:  $\text{GEN-OP } c\ a\ R \implies (c,a) \in R$   
**by** *simp*

### 2.3. ENTRY POINT FOR THE AUTOMATIC REFINEMENT TOOL 63

**lemma** *MINOR-PRIO-TAG-I*:  $P \implies (\text{MINOR-PRIO-TAG } p \implies P)$  **by** *auto*

**lemma** *MAJOR-PRIO-TAG-I*:  $P \implies (\text{MAJOR-PRIO-TAG } p \implies P)$  **by** *auto*

**lemma** *PRIO-TAG-I*:  $P \implies (\text{PRIO-TAG } ma\ mi \implies P)$  **by** *auto*

**end**