

Automatic Data Refinement

Peter Lammich

May 23, 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

1	Parametricity Solver	5
1.1	Relators	5
1.1.1	Basic Definitions	5
1.1.2	Basic HOL Relators	6
1.1.3	Automation	11
1.1.4	Setup	12
1.1.5	Invariant and Abstraction	15
1.1.6	Miscellaneous	16
1.1.7	Conversion between Predicate and Set Based Relators	16
1.1.8	More Properties	17
1.2	Basic Parametricity Reasoning	18
1.2.1	Auxiliary Lemmas	18
1.2.2	ML-Setup	19
1.2.3	Convenience Tools	19
1.3	Parametricity Theorems for HOL	19
1.3.1	Sets	19
1.3.2	Standard HOL Constructs	20
1.3.3	Functions	20
1.3.4	Boolean	21
1.3.5	Nat	21
1.3.6	Int	22
1.3.7	Product	22
1.3.8	Option	23
1.3.9	Sum	24
1.3.10	List	25
2	Automatic Refinement	29
2.1	Automatic Refinement Tool	29
2.1.1	Standard setup	29
2.1.2	Tools	29
2.1.3	Advanced Debugging	29
2.2	Standard HOL Bindings	30
2.2.1	Structural Expansion	30

2.2.2	Booleans	31
2.2.3	Standard Type Classes	31
2.2.4	Functional Combinators	32
2.2.5	Unit	33
2.2.6	Nat	33
2.2.7	Int	34
2.2.8	Product	34
2.2.9	Option	35
2.2.10	Sum-Types	36
2.2.11	List	37
2.2.12	Examples	40
2.3	Entry Point for the Automatic Refinement Tool	41
2.3.1	Convenience	41

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 $-rel-APP :: \text{args} \Rightarrow 'a \Rightarrow 'b \langle \langle - \rangle \rightarrow [0,900] 900 \rangle$

syntax-consts $-rel-APP == relAPP$

translations

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

$\langle ML \rangle$

1.1.2 Basic HOL Relators

Function

definition $fun-rel\ \text{where}$

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

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

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

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

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

$\langle proof \rangle$

lemma $fun-relD:$

shows $((f, f') \in (A \rightarrow B)) \implies$
 $(\bigwedge x\ x'. [\bigwedge (x, x') \in A] \implies (f\ x, f'\ x') \in B)$
 $\langle proof \rangle$

lemma $fun-relD1:$

assumes $(f, f') \in Ra \rightarrow Rr$
assumes $f\ x = r$
shows $\forall x'. (x, x') \in Ra \longrightarrow (r, f'\ x') \in Rr$
 $\langle proof \rangle$

lemma $fun-relD2:$

assumes $(f, f') \in Ra \rightarrow Rr$
assumes $f'\ x' = r'$
shows $\forall x. (x, x') \in Ra \longrightarrow (f\ x, r') \in Rr$
 $\langle proof \rangle$

lemma $fun-relE1:$

assumes $(f, f') \in Id \rightarrow Rv$
assumes $t' = f'\ x$
shows $(f\ x, t') \in Rv \langle proof \rangle$

lemma *fun-relE2*:
assumes $(f,f') \in Id \rightarrow Rv$
assumes $t = f x$
shows $(t,f' x) \in Rv$ *<proof>*

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]*:
 $((R1,R2) \text{prod-rel}) \equiv \{ ((a,b),(a',b')) . (a,a') \in R1 \wedge (b,b') \in R2 \}$
<proof>

lemma *prod-relI*: $\llbracket (a,a') \in R1 ; (b,b') \in R2 \rrbracket \implies ((a,b),(a',b')) \in \langle R1,R2 \rangle \text{prod-rel}$
<proof>

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** $(b,b') \in R2$
<proof>

lemma *prod-rel-simp[simp]*:
 $((a,b),(a',b')) \in \langle R1,R2 \rangle \text{prod-rel} \longleftrightarrow (a,a') \in R1 \wedge (b,b') \in R2$
<proof>

lemma *in-Domain-prod-rel-iff[iff]*: $(a,b) \in \text{Domain } (A \times_r B) \longleftrightarrow a \in \text{Domain } A \wedge b \in \text{Domain } B$
<proof>

lemma *prod-rel-comp*: $(A \times_r B) O (C \times_r D) = (A O C) \times_r (B O D)$
<proof>

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*]:

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

lemma *option-relI*:

$(None, None) \in \langle R \rangle \text{option-rel}$
 $\llbracket (a, a') \in R \rrbracket \implies (Some\ a, Some\ a') \in \langle R \rangle \text{option-rel}$
<proof>

lemma *option-relE*:

assumes $(x, x') \in \langle R \rangle \text{option-rel}$
obtains $x = None$ **and** $x' = None$
 $\mid a\ a'$ **where** $x = Some\ a$ **and** $x' = Some\ a'$ **and** $(a, a') \in R$
<proof>

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

$(None, a) \in \langle R \rangle \text{option-rel} \longleftrightarrow a = None$
 $(c, None) \in \langle R \rangle \text{option-rel} \longleftrightarrow c = None$
 $(Some\ x, Some\ y) \in \langle R \rangle \text{option-rel} \longleftrightarrow (x, y) \in R$
<proof>

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*]:

$\langle Rl, Rr \rangle sum-rel \equiv$
 $\{ (Inl\ a, Inl\ a') \mid a\ a'.\ (a, a') \in Rl \} \cup$
 $\{ (Inr\ a, Inr\ a') \mid a\ a'.\ (a, a') \in Rr \}$
<proof>

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

$\bigwedge a\ a'.\ (Inl\ a, Inl\ a') \in \langle Rl, Rr \rangle sum-rel \longleftrightarrow (a, a') \in Rl$
 $\bigwedge a\ a'.\ (Inr\ a, Inr\ a') \in \langle Rl, Rr \rangle sum-rel \longleftrightarrow (a, a') \in Rr$
 $\bigwedge a\ a'.\ (Inl\ a, Inr\ a') \notin \langle Rl, Rr \rangle sum-rel$
 $\bigwedge a\ a'.\ (Inr\ a, Inl\ a') \notin \langle Rl, Rr \rangle sum-rel$
<proof>

lemma *sum-relI*:

$(l, l') \in Rl \implies (Inl\ l, Inl\ l') \in \langle Rl, Rr \rangle sum-rel$
 $(r, r') \in Rr \implies (Inr\ r, Inr\ r') \in \langle Rl, Rr \rangle sum-rel$
<proof>

lemma *sum-relE*:

assumes $(x, x') \in \langle Rl, Rr \rangle sum-rel$
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 $(r,r')\in Rr$
 \langle proof \rangle

Lists

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

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

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

$\langle R \rangle list-rel \equiv \{(l,l').\ list-all2\ (\lambda x\ x'.\ (x,x')\in R)\ l\ l'\}$
 \langle proof \rangle

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

assumes $(l,l')\in \langle R \rangle list-rel$

assumes $P\ []\ []$

assumes $\bigwedge x\ x'\ l\ l'.\ []\ (x,x')\in R;\ (l,l')\in \langle R \rangle list-rel;\ P\ l\ l'\ []$

$\implies P\ (x\#l)\ (x'\#l')$

shows $P\ l\ l'$

\langle proof \rangle

lemma *list-rel-eq-listrel*: $list-rel = listrel$

\langle proof \rangle

lemma *list-relI*:

$([],[]) \in \langle R \rangle list-rel$

$[[\ (x,x')\in R;\ (l,l')\in \langle R \rangle list-rel\]\] \implies (x\#l, x'\#l') \in \langle R \rangle list-rel$

\langle proof \rangle

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

$([],l') \in \langle R \rangle list-rel \longleftrightarrow l'=[]$

$(l,[]) \in \langle R \rangle list-rel \longleftrightarrow l=[]$

$([],[]) \in \langle R \rangle list-rel$

$(x\#l, x'\#l') \in \langle R \rangle list-rel \longleftrightarrow (x,x')\in R \wedge (l,l')\in \langle R \rangle list-rel$

\langle proof \rangle

lemma *list-relE1*:

assumes $(l,[]) \in \langle R \rangle list-rel$ **obtains** $l=[]$ \langle proof \rangle

lemma *list-relE2*:

assumes $([],l) \in \langle R \rangle list-rel$ **obtains** $l=[]$ \langle proof \rangle

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** $(xs,xs')\in \langle R \rangle list-rel$

\langle proof \rangle

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 $(xs,xs')\in\langle R\rangle list-rel$
 ⟨proof⟩

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

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

$(l, l') \in \langle R \rangle list-rel \implies length\ l = length\ l'$
 ⟨proof⟩

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

$(x\#xs, l) \in \langle R \rangle list-rel \iff (\exists y\ ys. l=y\#ys \wedge (x, y) \in R \wedge (xs, ys) \in \langle R \rangle list-rel)$
 ⟨proof⟩

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

$(l, y\#ys) \in \langle R \rangle list-rel \iff (\exists x\ xs. l=x\#xs \wedge (x, y) \in R \wedge (xs, ys) \in \langle R \rangle list-rel)$
 ⟨proof⟩

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

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

term $set-rel$

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

$\langle R \rangle set-rel \equiv \{(A, B). (\forall x \in A. \exists y \in B. (x, y) \in R) \wedge (\forall y \in B. \exists x \in A. (x, y) \in R)\}$
 ⟨proof⟩

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

⟨proof⟩

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 $(A, B) \in \langle R \rangle set-rel$

⟨proof⟩

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

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

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

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

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

<proof>

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 $R \implies \langle R \rangle_{old-set-rel} = \langle R \rangle_{set-rel}$*
<proof>

lemma *set-rel-simp[simp]:*
 $(\{\}, \{\}) \in \langle R \rangle_{set-rel}$
<proof>

lemma *set-rel-empty-iff[simp]:*
 $(\{\}, y) \in \langle A \rangle_{set-rel} \iff y = \{\}$
 $(x, \{\}) \in \langle A \rangle_{set-rel} \iff x = \{\}$
<proof>

lemma *set-relD1: $(s, s') \in \langle R \rangle_{set-rel} \implies x \in s \implies \exists x' \in s'. (x, x') \in R$*
<proof>

lemma *set-relD2: $(s, s') \in \langle R \rangle_{set-rel} \implies x' \in s' \implies \exists x \in s. (x, x') \in R$*
<proof>

lemma *set-relE1[consumes 2]:*
assumes $(s, s') \in \langle R \rangle_{set-rel} \quad x \in s$
obtains x' **where** $x' \in s' \quad (x, x') \in R$
<proof>

lemma *set-relE2[consumes 2]:*
assumes $(s, s') \in \langle R \rangle_{set-rel} \quad x' \in s'$
obtains x **where** $x \in s \quad (x, x') \in R$
<proof>

1.1.3 Automation

A solver for relator properties

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

<ML>

lemma

relprop-id-orient[*relator-props*]: $R=Id \implies Id=R$ **and**
relprop-eq-refl[*solve-relator-props*]: $t = t$
 ⟨*proof*⟩

lemma

relprop-UNIV-orient[*relator-props*]: $R=UNIV \implies UNIV=R$
 ⟨*proof*⟩

ML-Level utilities

⟨*ML*⟩

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*⟩

Additional Properties

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

lemma *eq-UNIV-iff*: $S=UNIV \iff (\forall x. x \in S)$ ⟨*proof*⟩

lemma *fun-rel-sv*[*relator-props*]:
assumes *RAN*: $Range\ Ra = UNIV$
assumes *SV*: *single-valued Rv*
shows *single-valued* ($Ra \rightarrow Rv$)
 ⟨*proof*⟩

lemmas [*relator-props*] = *Range-Id*

lemma *fun-rel-id*[*relator-props*]: $\llbracket R1=Id; R2=Id \rrbracket \implies R1 \rightarrow R2 = Id$
 ⟨*proof*⟩

lemma *fun-rel-id-simp*[*simp*]: $Id \rightarrow Id = Id$ ⟨*proof*⟩

lemma *fun-rel-comp-dist*[relator-props]:
 $(R1 \rightarrow R2) \circ (R3 \rightarrow R4) \subseteq ((R1 \circ R3) \rightarrow (R2 \circ R4))$
 ⟨proof⟩

lemma *fun-rel-mono*[relator-props]: $\llbracket R1 \subseteq R2; R3 \subseteq R4 \rrbracket \implies R2 \rightarrow R3 \subseteq R1 \rightarrow R4$
 ⟨proof⟩

lemma *prod-rel-sv*[relator-props]:
 $\llbracket \text{single-valued } R1; \text{single-valued } R2 \rrbracket \implies \text{single-valued } (\langle R1, R2 \rangle \text{prod-rel})$
 ⟨proof⟩

lemma *prod-rel-id*[relator-props]: $\llbracket R1 = \text{Id}; R2 = \text{Id} \rrbracket \implies \langle R1, R2 \rangle \text{prod-rel} = \text{Id}$
 ⟨proof⟩

lemma *prod-rel-id-simp*[simp]: $\langle \text{Id}, \text{Id} \rangle \text{prod-rel} = \text{Id}$ ⟨proof⟩

lemma *prod-rel-mono*[relator-props]:
 $\llbracket R2 \subseteq R1; R3 \subseteq R4 \rrbracket \implies \langle R2, R3 \rangle \text{prod-rel} \subseteq \langle R1, R4 \rangle \text{prod-rel}$
 ⟨proof⟩

lemma *prod-rel-range*[relator-props]: $\llbracket \text{Range } Ra = \text{UNIV}; \text{Range } Rb = \text{UNIV} \rrbracket$
 $\implies \text{Range } (\langle Ra, Rb \rangle \text{prod-rel}) = \text{UNIV}$
 ⟨proof⟩

lemma *option-rel-sv*[relator-props]:
 $\llbracket \text{single-valued } R \rrbracket \implies \text{single-valued } (\langle R \rangle \text{option-rel})$
 ⟨proof⟩

lemma *option-rel-id*[relator-props]:
 $R = \text{Id} \implies \langle R \rangle \text{option-rel} = \text{Id}$ ⟨proof⟩

lemma *option-rel-id-simp*[simp]: $\langle \text{Id} \rangle \text{option-rel} = \text{Id}$ ⟨proof⟩

lemma *option-rel-mono*[relator-props]: $R \subseteq R' \implies \langle R \rangle \text{option-rel} \subseteq \langle R' \rangle \text{option-rel}$
 ⟨proof⟩

lemma *option-rel-range*: $\text{Range } R = \text{UNIV} \implies \text{Range } (\langle R \rangle \text{option-rel}) = \text{UNIV}$
 ⟨proof⟩

lemma *option-rel-inter*[simp]: $\langle R1 \cap R2 \rangle \text{option-rel} = \langle R1 \rangle \text{option-rel} \cap \langle R2 \rangle \text{option-rel}$
 ⟨proof⟩

lemma *option-rel-constraint*[simp]:
 $(x, x) \in (\text{UNIV} \times C) \text{option-rel} \iff (\forall v. x = \text{Some } v \implies v \in C)$
 ⟨proof⟩

lemma *sum-rel-sv*[relator-props]:

$\llbracket \text{single-valued } Rl; \text{ single-valued } Rr \rrbracket \Longrightarrow \text{single-valued } (\langle Rl, Rr \rangle \text{sum-rel})$
 $\langle \text{proof} \rangle$

lemma *sum-rel-id*[relator-props]: $\llbracket Rl=Id; Rr=Id \rrbracket \Longrightarrow \langle Rl, Rr \rangle \text{sum-rel} = Id$
 $\langle \text{proof} \rangle$

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

lemma *sum-rel-mono*[relator-props]:
 $\llbracket Rl \subseteq Rl'; Rr \subseteq Rr' \rrbracket \Longrightarrow \langle Rl, Rr \rangle \text{sum-rel} \subseteq \langle Rl', Rr' \rangle \text{sum-rel}$
 $\langle \text{proof} \rangle$

lemma *sum-rel-range*[relator-props]:
 $\llbracket \text{Range } Rl=UNIV; \text{Range } Rr=UNIV \rrbracket \Longrightarrow \text{Range } (\langle Rl, Rr \rangle \text{sum-rel}) = UNIV$
 $\langle \text{proof} \rangle$

lemma *list-rel-sv-iff*:
 $\text{single-valued } (\langle R \rangle \text{list-rel}) \longleftrightarrow \text{single-valued } R$
 $\langle \text{proof} \rangle$

lemma *list-rel-sv*[relator-props]:
 $\text{single-valued } R \Longrightarrow \text{single-valued } (\langle R \rangle \text{list-rel})$
 $\langle \text{proof} \rangle$

lemma *list-rel-id*[relator-props]: $\llbracket R=Id \rrbracket \Longrightarrow \langle R \rangle \text{list-rel} = Id$
 $\langle \text{proof} \rangle$

lemma *list-rel-id-simp*[simp]: $\langle Id \rangle \text{list-rel} = Id$ $\langle \text{proof} \rangle$

lemma *list-rel-mono*[relator-props]:
assumes $A: R \subseteq R'$
shows $\langle R \rangle \text{list-rel} \subseteq \langle R' \rangle \text{list-rel}$
 $\langle \text{proof} \rangle$

lemma *list-rel-range*[relator-props]:
assumes $A: \text{Range } R = UNIV$
shows $\text{Range } (\langle R \rangle \text{list-rel}) = UNIV$
 $\langle \text{proof} \rangle$

lemma *bijection-imp-sv*:
 $\text{bijection } R \Longrightarrow \text{single-valued } R$
 $\text{bijection } R \Longrightarrow \text{single-valued } (R^{-1})$
 $\langle \text{proof} \rangle$

declare *bijection-Id*[relator-props]
declare *bijection-Empty*[relator-props]

Pointwise refinement for set types:

lemma *set-rel-sv*[*relator-props*]:
single-valued $R \implies \text{single-valued } (\langle R \rangle \text{set-rel})$
<proof>

lemma *set-rel-id*[*relator-props*]: $R = \text{Id} \implies \langle R \rangle \text{set-rel} = \text{Id}$
<proof>

lemma *set-rel-id-simp*[*simp*]: $\langle \text{Id} \rangle \text{set-rel} = \text{Id}$ *<proof>*

lemma *set-rel-csv*[*relator-props*]:
 $\llbracket \text{single-valued } (R^{-1}) \rrbracket$
 $\implies \text{single-valued } ((\langle R \rangle \text{set-rel})^{-1})$
<proof>

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

abbreviation $br \equiv \text{build-rel}$

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

lemma *in-br-conv*: $(c,a) \in br \ \alpha I \iff a = \alpha c \wedge I c$
<proof>

lemma *brI*[*intro?*]: $\llbracket a = \alpha c ; I c \rrbracket \implies (c,a) \in br \ \alpha I$
<proof>

lemma *br-id*[*simp*]: $br \ id \ (\lambda-. \ True) = \text{Id}$
<proof>

lemma *br-chain*:
 $(\text{build-rel } \beta J) \ O \ (\text{build-rel } \alpha I) = \text{build-rel } (\alpha \circ \beta) \ (\lambda s. J s \wedge I (\beta s))$
<proof>

lemma *br-sv*[*simp, intro!, relator-props*]: *single-valued* $(br \ \alpha I)$
<proof>

lemma *converse-br-sv-iff*[*simp*]:
single-valued $(\text{converse } (br \ \alpha I)) \iff \text{inj-on } \alpha \ (\text{Collect } I)$
<proof>

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

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

<proof>

lemma *br-comp-alt'*:

$\{(c,a) . a = \alpha c \wedge I c\} \circ R = \{(c,a) . I c \wedge (\alpha c,a) \in R\}$
<proof>

lemma *single-valued-as-brE*:

assumes *single-valued R*
obtains α *invar* **where** $R = br \alpha$ *invar*
<proof>

lemma *sv-add-invar*:

single-valued R \implies *single-valued* $\{(c, a) . (c, a) \in R \wedge I c\}$
<proof>

lemma *br-Image-conv[simp]*: $br \alpha I \text{ `` } S = \{\alpha x \mid x. x \in S \wedge I x\}$

<proof>

1.1.6 Miscellaneous

lemma *rel-cong*: $(f,g) \in Id \implies (x,y) \in Id \implies (f x, g y) \in Id$ *<proof>*

lemma *rel-fun-cong*: $(f,g) \in Id \implies (f x, g x) \in Id$ *<proof>*

lemma *rel-arg-cong*: $(x,y) \in Id \implies (f x, f y) \in Id$ *<proof>*

1.1.7 Conversion between Predicate and Set Based Relators

Autoref uses set-based relators of type $(\text{'}a \times \text{'})b$ *set*, while the transfer and lifting package of Isabelle/HOL uses predicate based relators of type $\text{'}a \implies \text{'})b \implies \text{bool}$. 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*: $\llbracket rel2p \ R \ a \ b \rrbracket \implies (a,b) \in R$ *<proof>*

lemma *p2relD*: $\llbracket (a,b) \in p2rel \ R \rrbracket \implies R \ a \ b$ *<proof>*

lemma *rel2p-inv[simp]*:

$rel2p \ (p2rel \ P) = P$

$p2rel \ (rel2p \ R) = R$

<proof>

named-theorems *rel2p*

named-theorems *p2rel*

lemma *rel2p-dftt[rel2p]*:

$rel2p \ Id = (=)$

$rel2p \ (A \rightarrow B) = rel-fun \ (rel2p \ A) \ (rel2p \ B)$

$rel2p \ (A \times_r B) = rel-prod \ (rel2p \ A) \ (rel2p \ B)$

$rel2p \ (\langle A, B \rangle sum-rel) = rel-sum \ (rel2p \ A) \ (rel2p \ B)$

$rel2p \ (\langle A \rangle option-rel) = rel-option \ (rel2p \ A)$

$rel2p \langle A \rangle list\text{-rel} = list\text{-all2} (rel2p A)$
 $\langle proof \rangle$

lemma $p2rel\text{-dflt}[p2rel]$:

$p2rel (=) = Id$
 $p2rel (rel\text{-fun} A B) = p2rel A \rightarrow p2rel B$
 $p2rel (rel\text{-prod} A B) = p2rel A \times_r p2rel B$
 $p2rel (rel\text{-sum} A B) = \langle p2rel A, p2rel B \rangle sum\text{-rel}$
 $p2rel (rel\text{-option} A) = \langle p2rel A \rangle option\text{-rel}$
 $p2rel (list\text{-all2} A) = \langle p2rel A \rangle list\text{-rel}$
 $\langle proof \rangle$

lemma $[rel2p]$: $rel2p \langle A \rangle set\text{-rel} = rel\text{-set} (rel2p A)$
 $\langle proof \rangle$

lemma $[p2rel]$: $left\text{-unique} A \implies p2rel (rel\text{-set} A) = \langle p2rel A \rangle set\text{-rel}$
 $\langle proof \rangle$

lemma $rel2p\text{-comp}$: $rel2p A \circ\circ rel2p B = rel2p (A \circ B)$
 $\langle proof \rangle$

lemma $rel2p\text{-inj}[simp]$: $rel2p A = rel2p B \iff A=B$
 $\langle proof \rangle$

lemma $rel2p\text{-left-unique}$: $left\text{-unique} (rel2p A) = single\text{-valued} (A^{-1})$
 $\langle proof \rangle$

lemma $rel2p\text{-right-unique}$: $right\text{-unique} (rel2p A) = single\text{-valued} A$
 $\langle proof \rangle$

lemma $rel2p\text{-bi-unique}$: $bi\text{-unique} (rel2p A) \iff single\text{-valued} A \wedge single\text{-valued} (A^{-1})$
 $\langle proof \rangle$

lemma $p2rel\text{-left-unique}$: $single\text{-valued} ((p2rel A)^{-1}) = left\text{-unique} A$
 $\langle proof \rangle$

lemma $p2rel\text{-right-unique}$: $single\text{-valued} (p2rel A) = right\text{-unique} A$
 $\langle proof \rangle$

1.1.8 More Properties

lemma $list\text{-rel-comp}$: $\langle A \circ B \rangle list\text{-rel} = \langle A \rangle list\text{-rel} \circ \langle B \rangle list\text{-rel}$
 $\langle proof \rangle$

lemma $option\text{-rel-comp}$: $\langle A \circ B \rangle option\text{-rel} = \langle A \rangle option\text{-rel} \circ \langle B \rangle option\text{-rel}$
 $\langle proof \rangle$

lemma *prod-rel-compp*: $\langle A \ O \ B, \ C \ O \ D \rangle \text{prod-rel} = \langle A, C \rangle \text{prod-rel} \ O \ \langle B, D \rangle \text{prod-rel}$
 ⟨proof⟩

lemma *sum-rel-compp*: $\langle A \ O \ B, \ C \ O \ D \rangle \text{sum-rel} = \langle A, C \rangle \text{sum-rel} \ O \ \langle B, D \rangle \text{sum-rel}$
 ⟨proof⟩

lemma *set-rel-compp*: $\langle A \ O \ B \rangle \text{set-rel} = \langle A \rangle \text{set-rel} \ O \ \langle B \rangle \text{set-rel}$
 ⟨proof⟩

lemma *map-in-list-rel-conv*:

shows $(l, \text{map } \alpha \ l) \in \langle \text{br } \alpha \ I \rangle \text{list-rel} \longleftrightarrow (\forall x \in \text{set } l. I \ x)$
 ⟨proof⟩

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

lemma *finite-Image-sv*: *single-valued* $R \implies$ *finite* $s \implies$ *finite* $(R \ 's)$
 ⟨proof⟩

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

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

end

1.2 Basic Parametricity Reasoning

theory *Param-Tool*
imports *Relators*
begin

1.2.1 Auxiliary Lemmas

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

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

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

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*: $\llbracket (f, f') \in A \rightarrow B; (x, x') \in A \rrbracket \implies (f \ x, f' \ x') \in B$

<proof>

1.2.2 ML-Setup

<ML>

1.2.3 Convenience Tools

<ML>

end

1.3 Parametricity Theorems for HOL

theory *Param-HOL*
imports *Param-Tool*
begin

1.3.1 Sets

lemma *param-empty*[*param*]:
 $(\{\}, \{\}) \in \langle R \rangle \text{set-rel}$ *<proof>*

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

lemma *param-insert*[*param*]:
 $(\text{insert}, \text{insert}) \in R \rightarrow \langle R \rangle \text{set-rel} \rightarrow \langle R \rangle \text{set-rel}$
<proof>

lemma *param-union*[*param*]:
 $((\cup), (\cup)) \in \langle R \rangle \text{set-rel} \rightarrow \langle R \rangle \text{set-rel} \rightarrow \langle R \rangle \text{set-rel}$
<proof>

lemma *param-inter*[*param*]:
assumes *single-valued* *R* *single-valued* (R^{-1})
shows $((\cap), (\cap)) \in \langle R \rangle \text{set-rel} \rightarrow \langle R \rangle \text{set-rel} \rightarrow \langle R \rangle \text{set-rel}$
<proof>

lemma *param-diff*[*param*]:
assumes *single-valued* *R* *single-valued* (R^{-1})
shows $((-), (-)) \in \langle R \rangle \text{set-rel} \rightarrow \langle R \rangle \text{set-rel} \rightarrow \langle R \rangle \text{set-rel}$
<proof>

lemma *param-subseteq*[*param*]:

$\llbracket \text{single-valued } R; \text{ single-valued } (R^{-1}) \rrbracket \implies ((\subseteq), (\subseteq)) \in \langle R \rangle \text{set-rel} \rightarrow \langle R \rangle \text{set-rel}$
 $\rightarrow \text{bool-rel}$
 $\langle \text{proof} \rangle$

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

lemma *param-Ball*[*param*]: $(\text{Ball}, \text{Ball}) \in \langle Ra \rangle \text{set-rel} \rightarrow (Ra \rightarrow Id) \rightarrow Id$
 $\langle \text{proof} \rangle$

lemma *param-Bex*[*param*]: $(\text{Bex}, \text{Bex}) \in \langle Ra \rangle \text{set-rel} \rightarrow (Ra \rightarrow Id) \rightarrow Id$
 $\langle \text{proof} \rangle$

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

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

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

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

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 $(\text{If } c \ t \ e, \text{ If } c' \ t' \ e') \in R$
 $\langle \text{proof} \rangle$

lemma *param-Let*[*param*]:
 $(\text{Let}, \text{Let}) \in Ra \rightarrow (Ra \rightarrow Rr) \rightarrow Rr$
 $\langle \text{proof} \rangle$

1.3.3 Functions

lemma *param-id*[*param*]: $(\text{id}, \text{id}) \in R \rightarrow R \langle \text{proof} \rangle$

lemma *param-fun-comp*[*param*]: $((o), (o)) \in (Ra \rightarrow Rb) \rightarrow (Rc \rightarrow Ra) \rightarrow Rc \rightarrow Rb$
 ⟨*proof*⟩

lemma *param-fun-upd*[*param*]:
 $((=), (=)) \in Ra \rightarrow Ra \rightarrow Id$
 $\implies (fun-upd, fun-upd) \in (Ra \rightarrow Rb) \rightarrow Ra \rightarrow Rb \rightarrow Ra \rightarrow Rb$
 ⟨*proof*⟩

1.3.4 Boolean

lemma *rec-bool-is-case*: $old.rec-bool = case-bool$
 ⟨*proof*⟩

lemma *param-bool*[*param*]:
 $(True, True) \in Id$
 $(False, False) \in Id$
 $(conj, conj) \in Id \rightarrow Id \rightarrow Id$
 $(disj, disj) \in Id \rightarrow Id \rightarrow Id$
 $(Not, Not) \in Id \rightarrow Id$
 $(case-bool, case-bool) \in R \rightarrow R \rightarrow Id \rightarrow R$
 $(old.rec-bool, old.rec-bool) \in R \rightarrow R \rightarrow Id \rightarrow R$
 $((\longleftrightarrow), (\longleftrightarrow)) \in Id \rightarrow Id \rightarrow Id$
 $((\longrightarrow), (\longrightarrow)) \in Id \rightarrow Id \rightarrow Id$
 ⟨*proof*⟩

lemma *param-and-cong1*: $\llbracket (a, a') \in bool-rel; \llbracket a; a' \rrbracket \implies (b, b') \in bool-rel \rrbracket \implies (a \wedge b, a' \wedge b') \in bool-rel$
 ⟨*proof*⟩

lemma *param-and-cong2*: $\llbracket (a, a') \in bool-rel; \llbracket a; a' \rrbracket \implies (b, b') \in bool-rel \rrbracket \implies (b \wedge a, b' \wedge a') \in bool-rel$
 ⟨*proof*⟩

1.3.5 Nat

lemma *param-nat1*[*param*]:
 $(0, 0 :: nat) \in Id$
 $(Suc, Suc) \in Id \rightarrow Id$
 $(1, 1 :: nat) \in Id$
 $(numeral n :: nat, numeral n :: nat) \in Id$
 $((<), (< :: nat \Rightarrow -)) \in Id \rightarrow Id \rightarrow Id$
 $((\leq), (\leq :: nat \Rightarrow -)) \in Id \rightarrow Id \rightarrow Id$
 $((=), (= :: nat \Rightarrow -)) \in Id \rightarrow Id \rightarrow Id$
 $((+), (+ :: nat \Rightarrow -, (+))) \in Id \rightarrow Id \rightarrow Id$
 $((-), (- :: nat \Rightarrow -, (-))) \in Id \rightarrow Id \rightarrow Id$
 $((*), (* :: nat \Rightarrow -, (*))) \in Id \rightarrow Id \rightarrow Id$
 $((div), (div :: nat \Rightarrow -, (div))) \in Id \rightarrow Id \rightarrow Id$
 $((mod), (mod :: nat \Rightarrow -, (mod))) \in Id \rightarrow Id \rightarrow Id$
 ⟨*proof*⟩

lemma *param-case-nat*[*param*]:
 $(case-nat, case-nat) \in Ra \rightarrow (Id \rightarrow Ra) \rightarrow Id \rightarrow Ra$

$\langle proof \rangle$

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

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

$\langle proof \rangle$

1.3.6 Int

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

$(0, 0::int) \in Id$

$(1, 1::int) \in Id$

$(numeral\ n::int, numeral\ n::int) \in Id$

$((<), (<) :: int \Rightarrow -) \in Id \rightarrow Id \rightarrow Id$

$((\leq), (\leq) :: int \Rightarrow -) \in Id \rightarrow Id \rightarrow Id$

$((=), (=) :: int \Rightarrow -) \in Id \rightarrow Id \rightarrow Id$

$((+) :: int \Rightarrow -, (+)) \in Id \rightarrow Id \rightarrow Id$

$((-) :: int \Rightarrow -, (-)) \in Id \rightarrow Id \rightarrow Id$

$((*) :: int \Rightarrow -, (*)) \in Id \rightarrow Id \rightarrow Id$

$((div) :: int \Rightarrow -, (div)) \in Id \rightarrow Id \rightarrow Id$

$((mod) :: int \Rightarrow -, (mod)) \in Id \rightarrow Id \rightarrow Id$

$\langle proof \rangle$

1.3.7 Product

lemma *param-unit*[*param*]: $(((), ())) \in unit\text{-}rel$ $\langle proof \rangle$

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

$\langle proof \rangle$

lemma *param-prod*[*param*]:

$(Pair, Pair) \in Ra \rightarrow Rb \rightarrow \langle Ra, Rb \rangle prod\text{-}rel$

$(case\text{-}prod, case\text{-}prod) \in (Ra \rightarrow Rb \rightarrow Rr) \rightarrow \langle Ra, Rb \rangle prod\text{-}rel \rightarrow Rr$

$(old.\text{rec-prod}, old.\text{rec-prod}) \in (Ra \rightarrow Rb \rightarrow Rr) \rightarrow \langle Ra, Rb \rangle prod\text{-}rel \rightarrow Rr$

$(fst, fst) \in \langle Ra, Rb \rangle prod\text{-}rel \rightarrow Ra$

$(snd, snd) \in \langle Ra, Rb \rangle prod\text{-}rel \rightarrow Rb$

$\langle proof \rangle$

lemma *param-case-prod'*:

$\llbracket (p, p') \in \langle Ra, Rb \rangle prod\text{-}rel;$

$\bigwedge a\ b\ a'\ b'. \llbracket p=(a, b); p'=(a', b'); (a, a') \in Ra; (b, b') \in Rb \rrbracket$

$\implies (f\ a\ b, f'\ a'\ b') \in R$

$\rrbracket \implies (case\text{-}prod\ f\ p, case\text{-}prod\ f'\ p') \in R$

$\langle proof \rangle$

lemma *param-case-prod''*:

\llbracket

$\bigwedge a\ b\ a'\ b'. \llbracket p=(a, b); p'=(a', b') \rrbracket \implies (f\ a\ b, f'\ a'\ b') \in R$

$\rrbracket \implies (case\text{-}prod\ f\ p, case\text{-}prod\ f'\ p') \in R$

$\langle proof \rangle$

lemma *param-map-prod*[*param*]:
 (*map-prod*, *map-prod*)
 $\in (Ra \rightarrow Rb) \rightarrow (Rc \rightarrow Rd) \rightarrow \langle Ra, Rc \rangle \text{prod-rel} \rightarrow \langle Rb, Rd \rangle \text{prod-rel}$
<proof>

lemma *param-apfst*[*param*]:
 (*apfst*, *apfst*) $\in (Ra \rightarrow Rb) \rightarrow \langle Ra, Rc \rangle \text{prod-rel} \rightarrow \langle Rb, Rc \rangle \text{prod-rel}$
<proof>

lemma *param-apsnd*[*param*]:
 (*apsnd*, *apsnd*) $\in (Rb \rightarrow Rc) \rightarrow \langle Ra, Rb \rangle \text{prod-rel} \rightarrow \langle Ra, Rc \rangle \text{prod-rel}$
<proof>

lemma *param-curry*[*param*]:
 (*curry*, *curry*) $\in (\langle Ra, Rb \rangle \text{prod-rel} \rightarrow Rc) \rightarrow Ra \rightarrow Rb \rightarrow Rc$
<proof>

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

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

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

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

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

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

shows $(\text{fixp-fun } F\ x, \text{fixp-fun } F'\ x) \in Ra$

<proof>

end

1.3.8 Option

lemma *param-option*[*param*]:
 (*None*, *None*) $\in (R) \text{option-rel}$
 (*Some*, *Some*) $\in R \rightarrow \langle R \rangle \text{option-rel}$
 (*case-option*, *case-option*) $\in Rr \rightarrow (R \rightarrow Rr) \rightarrow \langle R \rangle \text{option-rel} \rightarrow Rr$
 (*rec-option*, *rec-option*) $\in Rr \rightarrow (R \rightarrow Rr) \rightarrow \langle R \rangle \text{option-rel} \rightarrow Rr$
<proof>

lemma *param-map-option*[*param*]: (*map-option*, *map-option*) $\in (A \rightarrow B) \rightarrow \langle A \rangle \text{option-rel} \rightarrow \langle B \rangle \text{option-rel}$
<proof>

lemma *param-case-option'*:

$$\begin{aligned} & \llbracket (x,x') \in \langle Rv \rangle \text{option-rel}; \\ & \quad \llbracket x = \text{None}; x' = \text{None} \rrbracket \implies (fn, fn') \in R; \\ & \quad \bigwedge v v'. \llbracket x = \text{Some } v; x' = \text{Some } v'; (v, v') \in Rv \rrbracket \implies (fs\ v, fs'\ v') \in R \\ & \rrbracket \implies (\text{case-option } fn\ fs\ x, \text{case-option } fn'\ fs'\ x') \in R \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *the-paramL*: $\llbracket l \neq \text{None}; (l, r) \in \langle R \rangle \text{option-rel} \rrbracket \implies (\text{the } l, \text{the } r) \in R$
 $\langle \text{proof} \rangle$

lemma *the-paramR*: $\llbracket r \neq \text{None}; (l, r) \in \langle R \rangle \text{option-rel} \rrbracket \implies (\text{the } l, \text{the } r) \in R$
 $\langle \text{proof} \rangle$

lemma *the-default-param*[*param*]:
 $(\text{the-default}, \text{the-default}) \in R \rightarrow \langle R \rangle \text{option-rel} \rightarrow R$
 $\langle \text{proof} \rangle$

1.3.9 Sum

lemma *rec-sum-is-case*: $\text{old.rec-sum} = \text{case-sum}$
 $\langle \text{proof} \rangle$

lemma *param-sum*[*param*]:
 $(\text{Inl}, \text{Inl}) \in Rl \rightarrow \langle Rl, Rr \rangle \text{sum-rel}$
 $(\text{Inr}, \text{Inr}) \in Rr \rightarrow \langle Rl, Rr \rangle \text{sum-rel}$
 $(\text{case-sum}, \text{case-sum}) \in (Rl \rightarrow R) \rightarrow (Rr \rightarrow R) \rightarrow \langle Rl, Rr \rangle \text{sum-rel} \rightarrow R$
 $(\text{old.rec-sum}, \text{old.rec-sum}) \in (Rl \rightarrow R) \rightarrow (Rr \rightarrow R) \rightarrow \langle Rl, Rr \rangle \text{sum-rel} \rightarrow R$
 $\langle \text{proof} \rangle$

lemma *param-case-sum'*:
 $\llbracket (s, s') \in \langle Rl, Rr \rangle \text{sum-rel};$
 $\quad \bigwedge l l'. \llbracket s = \text{Inl } l; s' = \text{Inl } l'; (l, l') \in Rl \rrbracket \implies (fl\ l, fl'\ l') \in R;$
 $\quad \bigwedge r r'. \llbracket s = \text{Inr } r; s' = \text{Inr } r'; (r, r') \in Rr \rrbracket \implies (fr\ r, fr'\ r') \in R$
 $\rrbracket \implies (\text{case-sum } fl\ fr\ s, \text{case-sum } fl'\ fr'\ s') \in R$
 $\langle \text{proof} \rangle$

primrec *is-Inl* **where** $\text{is-Inl } (\text{Inl } -) = \text{True} \mid \text{is-Inl } (\text{Inr } -) = \text{False}$
primrec *is-Inr* **where** $\text{is-Inr } (\text{Inr } -) = \text{True} \mid \text{is-Inr } (\text{Inl } -) = \text{False}$

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

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

lemma *sum-projl-param*[*param*]:
 $\llbracket \text{is-Inl } s; (s', s) \in \langle Ra, Rb \rangle \text{sum-rel} \rrbracket$
 $\implies (\text{Sum-Type.sum.proj1 } s', \text{Sum-Type.sum.proj1 } s) \in Ra$
 $\langle \text{proof} \rangle$

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

1.3.10 List

lemma *list-rel-append1*: $(as\ @\ bs,\ l) \in \langle R \rangle list-rel$
 $\longleftrightarrow (\exists\ cs\ ds.\ l = cs@ds \wedge (as,cs) \in \langle R \rangle list-rel \wedge (bs,ds) \in \langle R \rangle list-rel)$
 $\langle proof \rangle$

lemma *list-rel-append2*: $(l, as\ @\ bs) \in \langle R \rangle list-rel$
 $\longleftrightarrow (\exists\ cs\ ds.\ l = cs@ds \wedge (cs,as) \in \langle R \rangle list-rel \wedge (ds,bs) \in \langle R \rangle list-rel)$
 $\langle proof \rangle$

lemma *param-append*[*param*]:
 $(append,\ append) \in \langle R \rangle list-rel \rightarrow \langle R \rangle list-rel \rightarrow \langle R \rangle list-rel$
 $\langle proof \rangle$

lemma *param-list1*[*param*]:
 $(Nil, Nil) \in \langle R \rangle list-rel$
 $(Cons, Cons) \in R \rightarrow \langle R \rangle list-rel \rightarrow \langle R \rangle list-rel$
 $(case-list, case-list) \in Rr \rightarrow (R \rightarrow \langle R \rangle list-rel \rightarrow Rr) \rightarrow \langle R \rangle list-rel \rightarrow Rr$
 $\langle proof \rangle$

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

lemma *param-case-list'*:
 $\llbracket (l, l') \in \langle Rb \rangle list-rel;$
 $\llbracket l = []; l' = [] \rrbracket \implies (n, n') \in Ra;$
 $\bigwedge x\ xs\ x'\ xs'. \llbracket l = x\#xs; l' = x'\#xs'; (x, x') \in Rb; (xs, xs') \in \langle Rb \rangle list-rel \rrbracket$
 $\implies (c\ x\ xs,\ c'\ x'\ xs') \in Ra$
 $\rrbracket \implies (case-list\ n\ c\ l,\ case-list\ n'\ c'\ l') \in Ra$
 $\langle proof \rangle$

lemma *param-map*[*param*]:
 $(map, map) \in (R1 \rightarrow R2) \rightarrow \langle R1 \rangle list-rel \rightarrow \langle R2 \rangle list-rel$
 $\langle proof \rangle$

lemma *param-fold*[*param*]:
 $(fold, fold) \in (Re \rightarrow Rs \rightarrow Rs) \rightarrow \langle Re \rangle list-rel \rightarrow Rs \rightarrow Rs$
 $(foldl, foldl) \in (Rs \rightarrow Re \rightarrow Rs) \rightarrow Rs \rightarrow \langle Re \rangle list-rel \rightarrow Rs$
 $(foldr, foldr) \in (Re \rightarrow Rs \rightarrow Rs) \rightarrow \langle Re \rangle list-rel \rightarrow Rs \rightarrow Rs$
 $\langle proof \rangle$

lemma *param-list-all*[*param*]: $(list\text{-}all, list\text{-}all) \in (A \rightarrow bool\text{-}rel) \rightarrow \langle A \rangle list\text{-}rel \rightarrow bool\text{-}rel$
 $\langle proof \rangle$

context begin

private primrec *list-all2-alt* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow bool$

where

$list\text{-}all2\text{-}alt\ P\ []\ ys \longleftrightarrow (case\ ys\ of\ [] \Rightarrow True \mid - \Rightarrow False)$
 $| list\text{-}all2\text{-}alt\ P\ (x\#\!xs)\ ys \longleftrightarrow (case\ ys\ of\ [] \Rightarrow False \mid y\#\!ys \Rightarrow P\ x\ y \wedge list\text{-}all2\text{-}alt\ P\ xs\ ys)$

private lemma *list-all2-alt*: $list\text{-}all2\ P\ xs\ ys = list\text{-}all2\text{-}alt\ P\ xs\ ys$
 $\langle proof \rangle$

lemma *param-list-all2*[*param*]: $(list\text{-}all2, list\text{-}all2) \in (A \rightarrow B \rightarrow bool\text{-}rel) \rightarrow \langle A \rangle list\text{-}rel \rightarrow \langle B \rangle list\text{-}rel \rightarrow bool\text{-}rel$
 $\langle proof \rangle$

end

lemma *param-hd*[*param*]: $l \neq [] \Longrightarrow (l', l) \in \langle A \rangle list\text{-}rel \Longrightarrow (hd\ l', hd\ l) \in A$
 $\langle proof \rangle$

lemma *param-last*[*param*]:
assumes $y \neq []$
assumes $(x, y) \in \langle A \rangle list\text{-}rel$
shows $(last\ x, last\ y) \in A$
 $\langle proof \rangle$

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

schematic-goal *param-take*[*param*]: $(take, take) \in (?R::(-\times-) set)$
 $\langle proof \rangle$

schematic-goal *param-drop*[*param*]: $(drop, drop) \in (?R::(-\times-) set)$
 $\langle proof \rangle$

schematic-goal *param-length*[*param*]:
 $(length, length) \in (?R::(-\times-) set)$
 $\langle proof \rangle$

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

lemma *param-list-eq*[*param*]:

$(list\text{-}eq, list\text{-}eq) \in$
 $(R \rightarrow R \rightarrow Id) \rightarrow \langle R \rangle list\text{-}rel \rightarrow \langle R \rangle list\text{-}rel \rightarrow Id$
 $\langle proof \rangle$

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

lemma *param-list-equals[param]*:
 $\llbracket ((=), (=)) \in R \rightarrow R \rightarrow Id \rrbracket$
 $\implies ((=), (=)) \in \langle R \rangle list\text{-}rel \rightarrow \langle R \rangle list\text{-}rel \rightarrow Id$
 $\langle proof \rangle$

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

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

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

lemma *list-all-rec-eq*: $(\forall x \in set\ l.\ P\ x) = list\text{-}all\text{-}rec\ P\ l$
 $\langle proof \rangle$

lemma *list-ex-rec-eq*: $(\exists x \in set\ l.\ P\ x) = list\text{-}ex\text{-}rec\ P\ l$
 $\langle proof \rangle$

lemma *param-list-ball[param]*:
 $\llbracket (P, P') \in (Ra \rightarrow Id); (l, l') \in \langle Ra \rangle list\text{-}rel \rrbracket$
 $\implies (\forall x \in set\ l.\ P\ x, \forall x \in set\ l'.\ P'\ x) \in Id$
 $\langle proof \rangle$

lemma *param-list-bex[param]*:
 $\llbracket (P, P') \in (Ra \rightarrow Id); (l, l') \in \langle Ra \rangle list\text{-}rel \rrbracket$
 $\implies (\exists x \in set\ l.\ P\ x, \exists x \in set\ l'.\ P'\ x) \in Id$
 $\langle proof \rangle$

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

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

lemma *param-foldri[param]*: $(foldri, foldri)$

$\in \langle Re \rangle list\text{-rel} \rightarrow (Rs \rightarrow Id) \rightarrow (Re \rightarrow Rs \rightarrow Rs) \rightarrow Rs \rightarrow Rs$
 $\langle proof \rangle$

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 $(!i, !i') \in R$
 $\langle proof \rangle$

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

term *list-update*

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

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

lemma *param-upt*[*param*]:
 $(upt, upt) \in nat\text{-rel} \rightarrow nat\text{-rel} \rightarrow \langle nat\text{-rel} \rangle list\text{-rel}$
 $\langle proof \rangle$

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

lemma *param-all-interval-nat*[*param*]:
 $(List.all\text{-interval-nat}, List.all\text{-interval-nat})$
 $\in (nat\text{-rel} \rightarrow bool\text{-rel}) \rightarrow nat\text{-rel} \rightarrow nat\text{-rel} \rightarrow bool\text{-rel}$
 $\langle proof \rangle$

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

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

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

$\langle ML \rangle$

Main method

$\langle ML \rangle$

2.1.2 Tools

$\langle ML \rangle$

2.1.3 Advanced Debugging

$\langle ML \rangle$

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

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

lemma [*autoref-itype*]: $CAST ::_i I \rightarrow_i I$ $\langle proof \rangle$

Hide internal stuff

notation (*input*) *rel-ANNOT* (**infix** $\langle ::_r \rangle$ 10)

notation (*input*) *ind-ANNOT* (**infix** $\langle ::\#_r \rangle$ 10)

```

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

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

no-notation (input) rel-ANNOT (infix  $\langle ::: \rangle$  10)
no-notation (input) ind-ANNOT (infix  $\langle ::\# \rangle$  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:  $x=y \implies \text{STRUCT-EQ-tag } x\ y$  (proof)

```

```
 $\langle ML \rangle$ 
```

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 $REL-IS-ID-trigger$: $R=Id \implies REL-IS-ID R$ $\langle proof \rangle$

lemma $REL-FORCE-ID-trigger$: $R=Id \implies REL-FORCE-ID R$ $\langle proof \rangle$

$\langle ML \rangle$

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

$i\text{-bool} :: \text{interface}$

lemmas $[autoref-rel-intf] = REL-INTFI[of bool-rel i\text{-bool}]$

lemma $[autoref-itype]$:

$True ::_i i\text{-bool}$

$False ::_i i\text{-bool}$

$conj ::_i i\text{-bool} \rightarrow_i i\text{-bool} \rightarrow_i i\text{-bool}$

$(\leftarrow) ::_i i\text{-bool} \rightarrow_i i\text{-bool} \rightarrow_i i\text{-bool}$

$(\rightarrow) ::_i i\text{-bool} \rightarrow_i i\text{-bool} \rightarrow_i i\text{-bool}$

$disj ::_i i\text{-bool} \rightarrow_i i\text{-bool} \rightarrow_i i\text{-bool}$

$Not ::_i i\text{-bool} \rightarrow_i i\text{-bool}$

$case\text{-bool} ::_i I \rightarrow_i I \rightarrow_i i\text{-bool} \rightarrow_i I$

$old.\text{rec}\text{-bool} ::_i I \rightarrow_i I \rightarrow_i i\text{-bool} \rightarrow_i I$

$\langle proof \rangle$

lemma $autoref\text{-bool}[autoref\text{-rules}]$:

$(x,x) \in \text{bool}\text{-rel}$

$(conj, conj) \in \text{bool}\text{-rel} \rightarrow \text{bool}\text{-rel} \rightarrow \text{bool}\text{-rel}$

$(disj, disj) \in \text{bool}\text{-rel} \rightarrow \text{bool}\text{-rel} \rightarrow \text{bool}\text{-rel}$

$(Not, Not) \in \text{bool}\text{-rel} \rightarrow \text{bool}\text{-rel}$

$(case\text{-bool}, case\text{-bool}) \in R \rightarrow R \rightarrow \text{bool}\text{-rel} \rightarrow R$

$(old.\text{rec}\text{-bool}, old.\text{rec}\text{-bool}) \in R \rightarrow R \rightarrow \text{bool}\text{-rel} \rightarrow R$

$((\leftarrow), (\leftarrow)) \in \text{bool}\text{-rel} \rightarrow \text{bool}\text{-rel} \rightarrow \text{bool}\text{-rel}$

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

$\langle proof \rangle$

2.2.3 Standard Type Classes

context begin interpretation $autoref\text{-syn}$ $\langle proof \rangle$

We allow these operators for all interfaces.

lemma *[autoref-itype]*:

$(<) ::_i I \rightarrow_i I \rightarrow_i i\text{-bool}$
 $(\leq) ::_i I \rightarrow_i I \rightarrow_i i\text{-bool}$
 $(=) ::_i I \rightarrow_i I \rightarrow_i i\text{-bool}$
 $(+)$ $::_i I \rightarrow_i I \rightarrow_i I$
 $(-)$ $::_i I \rightarrow_i I \rightarrow_i I$
 (div) $::_i I \rightarrow_i I \rightarrow_i I$
 (mod) $::_i I \rightarrow_i I \rightarrow_i I$
 $(*)$ $::_i I \rightarrow_i I \rightarrow_i I$
 0 $::_i I$
 1 $::_i I$
numeral x $::_i I$
uminus $::_i I \rightarrow_i I$
 $\langle proof \rangle$

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

$0 \equiv OP\ 0 ::_i I$
 $1 \equiv OP\ 1 ::_i I$
numeral $x \equiv (OP\ (\textit{numeral}\ x) ::_i I)$
 $\langle proof \rangle$

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 $(\textit{numeral}\ x, OP\ (\textit{numeral}\ x) ::_i Id) \in Id$
and $(\textit{uminus}, \textit{uminus}) \in Id \rightarrow Id$
and $(0, 0) \in Id$
and $(1, 1) \in Id$
 $\langle proof \rangle$

2.2.4 Functional Combinators

lemma *[autoref-itype]*: $id ::_i I \rightarrow_i I$ $\langle proof \rangle$

lemma *autoref-id[autoref-rules]*: $(id, id) \in R \rightarrow R$ $\langle proof \rangle$

term (o)

lemma *[autoref-itype]*: $(o) ::_i (Ia \rightarrow_i Ib) \rightarrow_i (Ic \rightarrow_i Ia) \rightarrow_i Ic \rightarrow_i Ib$
 $\langle proof \rangle$

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

$((o), (o)) \in (Ra \rightarrow Rb) \rightarrow (Rc \rightarrow Ra) \rightarrow Rc \rightarrow Rb$
 $\langle proof \rangle$

lemma *[autoref-itype]*: $If ::_i i\text{-bool} \rightarrow_i I \rightarrow_i I \rightarrow_i I$ $\langle proof \rangle$

lemma *autoref-If[autoref-rules]*: $(If, If) \in Id \rightarrow R \rightarrow R \rightarrow R$ $\langle proof \rangle$

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 $c' t' e', (OP\ If \ ::\ Id \rightarrow R \rightarrow R \rightarrow R) \$c \$t \$e) \in R$
 $\langle proof \rangle$

lemma [autoref-itype]: $Let \ ::_i\ Ix \rightarrow_i\ (Ix \rightarrow_i\ Iy) \rightarrow_i\ Iy \ \langle proof \rangle$

lemma autoref-Let:

$(Let, Let) \in Ra \rightarrow (Ra \rightarrow Rr) \rightarrow Rr$
 $\langle proof \rangle$

lemma autoref-Let-cong[autoref-rules]:

assumes $(x',x) \in Ra$
assumes $\bigwedge y\ y'.\ REMOVE-INTERNAL\ (x=y) \implies (y',y) \in Ra \implies (f'\ y',$
 $f\ \$y) \in Rr$
shows $(Let\ x'\ f', (OP\ Let \ ::\ Ra \rightarrow (Ra \rightarrow Rr) \rightarrow Rr) \$x \$f) \in Rr$
 $\langle proof \rangle$

end

2.2.5 Unit

consts $i\text{-unit} \ ::\ interface$

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

lemma [autoref-rules]: $(((), ())) \in unit\text{-rel} \ \langle proof \rangle$

2.2.6 Nat

consts $i\text{-nat} \ ::\ interface$

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

context begin interpretation autoref-syn $\langle proof \rangle$

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

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

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

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

$\langle proof \rangle$

lemma autoref-nat[autoref-rules]:

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

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

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

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

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

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

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

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

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

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

$((*), (*)) \in \text{nat-rel} \rightarrow \text{nat-rel} \rightarrow \text{nat-rel}$
 $((\text{mod}), (\text{mod})) \in \text{nat-rel} \rightarrow \text{nat-rel} \rightarrow \text{nat-rel}$
 $\langle \text{proof} \rangle$

lemma *autoref-case-nat*[*autoref-rules*]:
 $(\text{case-nat}, \text{case-nat}) \in \text{Ra} \rightarrow (\text{Id} \rightarrow \text{Ra}) \rightarrow \text{Id} \rightarrow \text{Ra}$
 $\langle \text{proof} \rangle$

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

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* $\langle \text{proof} \rangle$

lemma *pat-num-int*[*autoref-op-pat*]:
 $0::\text{int} \equiv \text{OP } 0 \text{ } ::_i \text{ } i\text{-int}$
 $1::\text{int} \equiv \text{OP } 1 \text{ } ::_i \text{ } i\text{-int}$
 $(\text{numeral } x)::\text{int} \equiv (\text{OP } (\text{numeral } x) \text{ } ::_i \text{ } i\text{-int})$
 $\langle \text{proof} \rangle$

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

$(0, 0::\text{int}) \in \text{int-rel}$
 $(1, 1::\text{int}) \in \text{int-rel}$
 $(\text{numeral } n::\text{int}, \text{numeral } n::\text{int}) \in \text{int-rel}$
 $((<), (< :: \text{int} \Rightarrow -)) \in \text{int-rel} \rightarrow \text{int-rel} \rightarrow \text{bool-rel}$
 $((\leq), (\leq :: \text{int} \Rightarrow -)) \in \text{int-rel} \rightarrow \text{int-rel} \rightarrow \text{bool-rel}$
 $((=), (= :: \text{int} \Rightarrow -)) \in \text{int-rel} \rightarrow \text{int-rel} \rightarrow \text{bool-rel}$
 $((+) :: \text{int} \Rightarrow -, (+)) \in \text{int-rel} \rightarrow \text{int-rel} \rightarrow \text{int-rel}$
 $((-) :: \text{int} \Rightarrow -, (-)) \in \text{int-rel} \rightarrow \text{int-rel} \rightarrow \text{int-rel}$
 $((\text{div}) :: \text{int} \Rightarrow -, (\text{div})) \in \text{int-rel} \rightarrow \text{int-rel} \rightarrow \text{int-rel}$
 $(\text{uminus}, \text{uminus}) \in \text{int-rel} \rightarrow \text{int-rel}$
 $((*), (*)) \in \text{int-rel} \rightarrow \text{int-rel} \rightarrow \text{int-rel}$
 $((\text{mod}), (\text{mod})) \in \text{int-rel} \rightarrow \text{int-rel} \rightarrow \text{int-rel}$
 $\langle \text{proof} \rangle$

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* $\langle \text{proof} \rangle$

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*) $\in \langle Ra,Rb \rangle prod-rel \rightarrow Rb$
 $\langle proof \rangle$

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*,(=)) $\in \langle Ra,Rb \rangle prod-rel \rightarrow \langle Ra,Rb \rangle prod-rel \rightarrow Id$
 $\langle proof \rangle$

lemma *prod-eq-expand*[*autoref-struct-expand*]: (=) = *prod-eq* (=) (=)

$\langle proof \rangle$

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* $\langle proof \rangle$

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*) $\in Rr \rightarrow (R \rightarrow Rr) \rightarrow \langle R \rangle option-rel \rightarrow Rr$
 $\langle proof \rangle$

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*) $\in R$
 $\langle proof \rangle$

lemma *autoref-the-default*[*autoref-rules*]:

(*the-default*, *the-default*) $\in R \rightarrow \langle R \rangle option-rel \rightarrow R$
 $\langle proof \rangle$

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

$\langle \text{proof} \rangle$
lemma *autoref-is-None*[*param, autoref-rules*]:
 $(\text{is-None}, \text{is-None}) \in \langle R \rangle \text{option-rel} \rightarrow \text{Id}$
 $\langle \text{proof} \rangle$

lemma *fold-is-None*: $x = \text{None} \longleftrightarrow \text{is-None } x$ $\langle \text{proof} \rangle$

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

lemma *option-eq-autoref*[*autoref-rules (overloaded)*]:
 $\llbracket \text{GEN-OP eq } (=) (R \rightarrow R \rightarrow \text{Id}) \rrbracket$
 $\implies (\text{option-eq eq}, (=)) \in \langle R \rangle \text{option-rel} \rightarrow \langle R \rangle \text{option-rel} \rightarrow \text{Id}$
 $\langle \text{proof} \rangle$

lemma *option-eq-expand*[*autoref-struct-expand*]:
 $(=) = \text{option-eq } (=)$
 $\langle \text{proof} \rangle$

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* $\langle \text{proof} \rangle$

lemma *autoref-sum*[*autoref-rules*]:
 $(\text{Inl}, \text{Inl}) \in Rl \rightarrow \langle Rl, Rr \rangle \text{sum-rel}$
 $(\text{Inr}, \text{Inr}) \in Rr \rightarrow \langle Rl, Rr \rangle \text{sum-rel}$
 $(\text{case-sum}, \text{case-sum}) \in (Rl \rightarrow R) \rightarrow (Rr \rightarrow R) \rightarrow \langle Rl, Rr \rangle \text{sum-rel} \rightarrow R$
 $(\text{old.rec-sum}, \text{old.rec-sum}) \in (Rl \rightarrow R) \rightarrow (Rr \rightarrow R) \rightarrow \langle Rl, Rr \rangle \text{sum-rel} \rightarrow R$
 $\langle \text{proof} \rangle$

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

lemma *sum-eq-autoref*[*autoref-rules (overloaded)*]:
 $\llbracket \text{GEN-OP eql } (=) (Rl \rightarrow Rl \rightarrow \text{Id}); \text{GEN-OP eqr } (=) (Rr \rightarrow Rr \rightarrow \text{Id}) \rrbracket$
 $\implies (\text{sum-eq eql eqr}, (=)) \in \langle Rl, Rr \rangle \text{sum-rel} \rightarrow \langle Rl, Rr \rangle \text{sum-rel} \rightarrow \text{Id}$
 $\langle \text{proof} \rangle$

lemma *sum-eq-expand*[*autoref-struct-expand*]: $(=) = \text{sum-eq } (=) (=)$
 ⟨*proof*⟩

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

lemma *autoref-sum-Projl*[*autoref-rules*]:
 $\llbracket \text{SIDE-PRECOND } (is\text{-Inl } s); (s',s) \in \langle Ra, Rb \rangle \text{sum-rel} \rrbracket$
 $\implies (Sum\text{-Type.sum.proj1 } s', (OP\ Sum\text{-Type.sum.proj1} \ ::: \langle Ra, Rb \rangle \text{sum-rel} \rightarrow$
 $Ra)\$s) \in Ra$
 ⟨*proof*⟩

lemma *autoref-sum-Projr*[*autoref-rules*]:
 $\llbracket \text{SIDE-PRECOND } (is\text{-Inr } s); (s',s) \in \langle Ra, Rb \rangle \text{sum-rel} \rrbracket$
 $\implies (Sum\text{-Type.sum.proj2 } s', (OP\ Sum\text{-Type.sum.proj2} \ ::: \langle Ra, Rb \rangle \text{sum-rel} \rightarrow$
 $Rb)\$s) \in Rb$
 ⟨*proof*⟩

end

2.2.11 List

consts *i-list* :: *interface* \Rightarrow *interface*

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

context begin interpretation *autoref-syn* ⟨*proof*⟩

lemma *autoref-append*[*autoref-rules*]:
 $(\text{append}, \text{append}) \in \langle R \rangle \text{list-rel} \rightarrow \langle R \rangle \text{list-rel} \rightarrow \langle R \rangle \text{list-rel}$
 ⟨*proof*⟩

lemma *refine-list*[*autoref-rules*]:
 $(Nil, Nil) \in \langle R \rangle \text{list-rel}$
 $(Cons, Cons) \in R \rightarrow \langle R \rangle \text{list-rel} \rightarrow \langle R \rangle \text{list-rel}$
 $(\text{case-list}, \text{case-list}) \in Rr \rightarrow (R \rightarrow \langle R \rangle \text{list-rel} \rightarrow Rr) \rightarrow \langle R \rangle \text{list-rel} \rightarrow Rr$
 ⟨*proof*⟩

lemma *autoref-rec-list*[*autoref-rules*]: $(\text{rec-list}, \text{rec-list})$
 $\in Ra \rightarrow (Rb \rightarrow \langle Rb \rangle \text{list-rel} \rightarrow Ra \rightarrow Ra) \rightarrow \langle Rb \rangle \text{list-rel} \rightarrow Ra$
 ⟨*proof*⟩

lemma *refine-map*[*autoref-rules*]:
 $(\text{map}, \text{map}) \in (R1 \rightarrow R2) \rightarrow \langle R1 \rangle \text{list-rel} \rightarrow \langle R2 \rangle \text{list-rel}$
 ⟨*proof*⟩

lemma *refine-fold*[*autoref-rules*]:
 $(\text{fold}, \text{fold}) \in (Re \rightarrow Rs \rightarrow Rs) \rightarrow \langle Re \rangle \text{list-rel} \rightarrow Rs \rightarrow Rs$
 $(\text{foldl}, \text{foldl}) \in (Rs \rightarrow Re \rightarrow Rs) \rightarrow Rs \rightarrow \langle Re \rangle \text{list-rel} \rightarrow Rs$
 $(\text{foldr}, \text{foldr}) \in (Re \rightarrow Rs \rightarrow Rs) \rightarrow \langle Re \rangle \text{list-rel} \rightarrow Rs \rightarrow Rs$

$\langle proof \rangle$

schematic-goal *autoref-take*[*autoref-rules*]: $(take, take) \in (?R :: (- \times -) \text{ set})$

$\langle proof \rangle$

schematic-goal *autoref-drop*[*autoref-rules*]: $(drop, drop) \in (?R :: (- \times -) \text{ set})$

$\langle proof \rangle$

schematic-goal *autoref-length*[*autoref-rules*]:

$(length, length) \in (?R :: (- \times -) \text{ set})$

$\langle proof \rangle$

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

assumes $(l, l') \in \langle R \rangle list\text{-rel}$

assumes $(i, i') \in Id$

assumes *SIDE-PRECOND* $(i' < length\ l')$

shows $(nth\ l\ i, (OP\ nth\ ::\ \langle R \rangle list\text{-rel}\ \rightarrow\ Id\ \rightarrow\ R)\ \$l\ \$i') \in R$

$\langle proof \rangle$

fun *list-eq* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list \Rightarrow bool$ **where**

list-eq *eq* [] [] $\longleftrightarrow True$

| *list-eq* *eq* (*a*#*l*) (*a'*#*l'*)

$\longleftrightarrow (if\ eq\ a\ a'\ then\ list\text{-eq}\ eq\ l\ l'\ else\ False)$

| *list-eq* - - - $\longleftrightarrow False$

lemma *autoref-list-eq-aux*:

$(list\text{-eq}, list\text{-eq}) \in$

$(R \rightarrow R \rightarrow Id) \rightarrow \langle R \rangle list\text{-rel} \rightarrow \langle R \rangle list\text{-rel} \rightarrow Id$

$\langle proof \rangle$

lemma *list-eq-expand*[*autoref-struct-expand*]: $(=) = (list\text{-eq}\ (=))$

$\langle proof \rangle$

lemma *autoref-list-eq*[*autoref-rules* (**overloaded**)]:

GEN-OP *eq* $(=)$ $(R \rightarrow R \rightarrow Id) \Longrightarrow (list\text{-eq}\ eq, (=))$

$\in \langle R \rangle list\text{-rel} \rightarrow \langle R \rangle list\text{-rel} \rightarrow Id$

$\langle proof \rangle$

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

$\llbracket\ \textit{SIDE-PRECOND}\ (l' \neq []) ; (l, l') \in \langle R \rangle list\text{-rel}\ \rrbracket \Longrightarrow$

$(hd\ l, (OP\ hd\ ::\ \langle R \rangle list\text{-rel}\ \rightarrow\ R)\ \$l') \in R$

$\langle proof \rangle$

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

$(tl, tl) \in \langle R \rangle list\text{-rel} \rightarrow \langle R \rangle list\text{-rel}$

$\langle proof \rangle$

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

lemma *is-Nil-pat*[*autoref-op-pat*]:

$a = [] \equiv (OP\ is\text{-Nil}\ ::\ ;\ \langle I \rangle_i\ i\text{-list}\ \rightarrow_i\ i\text{-bool})\ \a

$\square = a \equiv (OP\ is\ Nil \ ::: _i \langle I \rangle_i\ i\text{-list} \rightarrow_i\ i\text{-bool}) \$ a$
 $\langle proof \rangle$

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

lemma *conv-to-is-Nil*:
 $l = \square \longleftrightarrow is\ Nil\ l$
 $\square = l \longleftrightarrow is\ Nil\ l$
 $\langle proof \rangle$

lemma *autoref-butlast*[*param, autoref-rules*]:
 $(butlast, butlast) \in \langle R \rangle list\text{-rel} \rightarrow \langle R \rangle list\text{-rel}$
 $\langle proof \rangle$

definition [*simp*]: *op-list-singleton* $x \equiv [x]$

lemma *op-list-singleton-pat*[*autoref-op-pat*]:
 $[x] \equiv (OP\ op\text{-list}\text{-singleton} \ ::: _i\ I \rightarrow_i \langle I \rangle_i\ i\text{-list}) \$ x$ $\langle proof \rangle$

lemma *autoref-list-singleton*[*autoref-rules*]:
 $(\lambda a. [a], op\text{-list}\text{-singleton}) \in R \rightarrow \langle R \rangle list\text{-rel}$
 $\langle proof \rangle$

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

lemma *pat-list-append-elem*[*autoref-op-pat*]:
 $s @ [x] \equiv (OP\ op\text{-list}\text{-append}\text{-elem} \ ::: _i \langle I \rangle_i\ i\text{-list} \rightarrow_i\ I \rightarrow_i \langle I \rangle_i\ i\text{-list}) \$ s \$ x$
 $\langle proof \rangle$

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

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

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

lemma [*autoref-op-pat*]:
 $(\forall i < u. P\ i) \equiv OP\ List.all\text{-interval}\text{-nat}\ P\ 0\ u$
 $(\forall i \leq u. P\ i) \equiv OP\ List.all\text{-interval}\text{-nat}\ P\ 0\ (Suc\ u)$
 $(\forall i < u. l \leq i \longrightarrow P\ i) \equiv OP\ List.all\text{-interval}\text{-nat}\ P\ l\ u$
 $(\forall i \leq u. l \leq i \longrightarrow P\ i) \equiv OP\ List.all\text{-interval}\text{-nat}\ P\ l\ (Suc\ u)$
 $\langle proof \rangle$

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
⟨proof⟩
```

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
⟨proof⟩
```

schematic-goal

```
(?f::?'c,[1,2,3] = [])∈?R
⟨proof⟩
```

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!*

```
⟨proof⟩
```

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
⟨proof⟩
```

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
⟨proof⟩
```


schematic-goal

assumes [*autoref-rules*]: $(ai, a) \in \langle R \rangle$ *list-rel*
shows $(?f :: ?'c, [] = a) \in ?R$
 $\langle proof \rangle$

schematic-goal

shows $(?f :: ?'c, [1, 2] = [2, 3 :: nat]) \in ?R$
 $\langle proof \rangle$

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*: $P\ x \implies PREFER\ P\ x$ $\langle proof \rangle$

lemma *PREFER-D*: $PREFER\ P\ x \implies P\ x$ $\langle proof \rangle$

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

lemma *PREFER-id-D*: $PREFER-id\ R \implies R=Id$ $\langle proof \rangle$

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

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

lemma *SIDE-GEN-ALGO-D*: $SIDE-GEN-ALGO\ P \implies P$ $\langle proof \rangle$

lemma *GEN-OP-D*: $GEN-OP\ c\ a\ R \implies (c, a) \in R$

$\langle proof \rangle$

lemma *MINOR-PRIO-TAG-I*: $P \implies (MINOR-PRIO-TAG\ p \implies P)$ $\langle proof \rangle$

lemma *MAJOR-PRIO-TAG-I*: $P \implies (MAJOR-PRIO-TAG\ p \implies P)$ $\langle proof \rangle$

lemma *PRIO-TAG-I*: $P \implies (PRIO-TAG\ ma\ mi \implies P)$ $\langle proof \rangle$

end