Automatic Data Refinement

Peter Lammich

June 25, 2026

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 datastructures.

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 Miscellanneous . . . 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 **31**
- 2.1 Automatic Refinement Tool . . . 31
  - 2.1.1 Standard setup . . . 31
  - 2.1.2 Tools . . . 31
  - 2.1.3 Advanced Debugging . . . 31
- 2.2 Standard HOL Bindings . . . 32
  - 2.2.1 Structural Expansion . . . 32

2.2.2 Booleans . . . 33
2.2.3 Standard Type Classes . . . 33
2.2.4 Functional Combinators . . . 34
2.2.5 Unit . . . 35
2.2.6 Nat . . . 35
2.2.7 Int . . . 36
2.2.8 Product . . . 36
2.2.9 Option . . . 37
2.2.10 Sum-Types . . . 38
2.2.11 List . . . 39
2.2.12 Examples . . . 42
2.3 Entry Point for the Automatic Refinement Tool . . . 43
2.3.1 Convenience . . . 43

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* ≡ *f x*

syntax *-rel-APP* :: *args* ⇒ *'a* ⇒ *'b* (‹⟨-⟩-› [*0*,*900*] *900*)

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

translations
⟨*x*,*xs*⟩*R* == ⟨*xs*⟩(*CONST relAPP R x*)
⟨*x*⟩*R* == *CONST relAPP R x*

⟨*ML*⟩

1.1.2 Basic HOL Relators

Function

definition *fun-rel* **where**
fun-rel-def-internal: *fun-rel A B* ≡ { (*f*,*f'*). ∀(*a*,*a'*)∈*A*. (*f a*, *f' a'*)∈*B* }
abbreviation *fun-rel-syn* (**infixr** ‹→› *60*) **where** *A*→*B* ≡ ⟨*A*,*B*⟩*fun-rel*

lemma *fun-rel-def*[*refine-rel-defs*]:
A→*B* ≡ { (*f*,*f'*). ∀(*a*,*a'*)∈*A*. (*f a*, *f' a'*)∈*B* }
⟨*proof*⟩

lemma *fun-relI*[*intro*!]: ⟦⋀*a a'*. (*a*,*a'*)∈*A* ⟹ (*f a*,*f' a'*)∈*B*⟧ ⟹ (*f*,*f'*)∈*A*→*B*
⟨*proof*⟩

lemma *fun-relD*:
shows ((*f*,*f'*)∈(*A*→*B*)) ⟹
(⋀*x x'*. ⟦ (*x*,*x'*)∈*A* ⟧ ⟹ (*f x*, *f' x'*)∈*B*)
⟨*proof*⟩

lemma *fun-relD1*:
assumes (*f*,*f'*)∈*Ra*→*Rr*
assumes *f x* = *r*
shows ∀*x'*. (*x*,*x'*)∈*Ra* ⟶ (*r*,*f' x'*)∈*Rr*
⟨*proof*⟩

lemma *fun-relD2*:
assumes (*f*,*f'*)∈*Ra*→*Rr*
assumes *f' x'* = *r'*
shows ∀*x*. (*x*,*x'*)∈*Ra* ⟶ (*f x*,*r'*)∈*Rr*
⟨*proof*⟩

lemma *fun-relE1*:
assumes (*f*,*f'*)∈*Id* → *Rv*
assumes *t'* = *f' x*
shows (*f x*,*t'*)∈*Rv* ⟨*proof*⟩

lemma *fun-relE2*:
assumes $(f,f')\in Id \to Rv$
assumes $t = f\ x$
shows $(t,f'\ x)\in Rv$ ⟨*proof*⟩

Terminal Types

abbreviation *unit-rel* :: (*unit*×*unit*) *set* **where** *unit-rel* == *Id*

abbreviation *nat-rel* ≡ *Id*::(*nat*×-) *set*
abbreviation *int-rel* ≡ *Id*::(*int*×-) *set*
abbreviation *bool-rel* ≡ *Id*::(*bool*×-) *set*

Product

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

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

lemma *prod-rel-def*[*refine-rel-defs*]:
$(\langle R1,R2\rangle$*prod-rel*$) \equiv$ { $((a,b),(a',b'))$. $(a,a')\in R1 \wedge (b,b')\in R2$ }
⟨*proof*⟩

lemma *prod-relI*: $[\![(a,a')\in R1;\ (b,b')\in R2]\!] \Longrightarrow ((a,b),(a',b'))\in\langle R1,R2\rangle$*prod-rel*
⟨*proof*⟩
lemma *prod-relE*:
assumes $(p,p')\in\langle R1,R2\rangle$*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*]:
$((a,b),(a',b'))\in\langle R1,R2\rangle$*prod-rel* ⟷ $(a,a')\in R1 \wedge (b,b')\in R2$
⟨*proof*⟩

lemma *in-Domain-prod-rel-iff*[*iff*]: $(a,b)\in$*Domain* $(A\times_r B)$ ⟷ $a\in$*Domain A* ∧ $b\in$*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 ≡ { (*Some a*,*Some a'*) | *a a'*. $(a,a')\in R$ } ∪ {(*None*,*None*)}

lemma *option-rel-def*[*refine-rel-defs*]:
$\langle R\rangle$*option-rel* $\equiv$ { (*Some a*,*Some a'*) | *a a'*. (*a*,*a'*)$\in R$ } $\cup$ {(*None*,*None*)}
⟨*proof*⟩

lemma *option-relI*:
(*None*,*None*)$\in\langle R\rangle$ *option-rel*
⟦ (*a*,*a'*)$\in R$ ⟧ $\Longrightarrow$ (*Some a*, *Some a'*)$\in\langle R\rangle$*option-rel*
⟨*proof*⟩

lemma *option-relE*:
assumes (*x*,*x'*)$\in\langle R\rangle$*option-rel*
obtains *x*=*None* **and** *x'*=*None*
| *a a'* **where** *x*=*Some a* **and** *x'*=*Some a'* **and** (*a*,*a'*)$\in R$
⟨*proof*⟩

lemma *option-rel-simp*[*simp*]:
(*None*,*a*)$\in\langle R\rangle$*option-rel* $\longleftrightarrow$ *a*=*None*
(*c*,*None*)$\in\langle R\rangle$*option-rel* $\longleftrightarrow$ *c*=*None*
(*Some x*,*Some y*)$\in\langle R\rangle$*option-rel* $\longleftrightarrow$ (*x*,*y*)$\in R$
⟨*proof*⟩

Sum

definition *sum-rel* **where** *sum-rel-def-internal*:
sum-rel Rl Rr
≡ { (*Inl a*, *Inl a'*) | *a a'*. (*a*,*a'*)$\in$*Rl* } $\cup$
{ (*Inr a*, *Inr a'*) | *a a'*. (*a*,*a'*)$\in$*Rr* }

lemma *sum-rel-def*[*refine-rel-defs*]:
$\langle$*Rl*,*Rr*$\rangle$*sum-rel* $\equiv$
{ (*Inl a*, *Inl a'*) | *a a'*. (*a*,*a'*)$\in$*Rl* } $\cup$
{ (*Inr a*, *Inr a'*) | *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* $\Longrightarrow$ (*Inl l*, *Inl l'*) $\in\langle$*Rl*,*Rr*$\rangle$*sum-rel*
(*r*,*r'*)$\in$*Rr* $\Longrightarrow$ (*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** $(r,r')\in Rr$
⟨*proof*⟩

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*]:
$\langle R\rangle list\text{-}rel \equiv \{(l,l').\ list\text{-}all2\ (\lambda x\ x'.\ (x,x')\in R)\ l\ l'\}$
⟨*proof*⟩

lemma *list-rel-induct*[*induct set*,*consumes 1*, *case-names Nil Cons*]:
assumes $(l,l')\in\langle R\rangle\ list\text{-}rel$
assumes $P\ []\ []$
assumes $\bigwedge x\ x'\ l\ l'.\ [\![\ (x,x')\in R;\ (l,l')\in\langle R\rangle list\text{-}rel;\ P\ l\ l'\ ]\!]$
$\Longrightarrow P\ (x\#l)\ (x'\#l')$
shows $P\ l\ l'$
⟨*proof*⟩

lemma *list-rel-eq-listrel*: *list-rel* = *listrel*
⟨*proof*⟩

lemma *list-relI*:
$([],[])\in\langle R\rangle list\text{-}rel$
$[\![\ (x,x')\in R;\ (l,l')\in\langle R\rangle list\text{-}rel\ ]\!] \Longrightarrow (x\#l,x'\#l')\in\langle R\rangle list\text{-}rel$
⟨*proof*⟩

lemma *list-rel-simp*[*simp*]:
$([],l')\in\langle R\rangle list\text{-}rel \longleftrightarrow l'=[]$
$(l,[])\in\langle R\rangle list\text{-}rel \longleftrightarrow l=[]$
$([],[])\in\langle R\rangle list\text{-}rel$
$(x\#l,x'\#l')\in\langle R\rangle list\text{-}rel \longleftrightarrow (x,x')\in R \wedge (l,l')\in\langle R\rangle list\text{-}rel$
⟨*proof*⟩

lemma *list-relE1*:
assumes $(l,[])\in\langle R\rangle list\text{-}rel$ **obtains** $l=[]$ ⟨*proof*⟩

lemma *list-relE2*:
assumes $([],l)\in\langle R\rangle list\text{-}rel$ **obtains** $l=[]$ ⟨*proof*⟩

lemma *list-relE3*:
assumes $(x\#xs,l')\in\langle R\rangle list\text{-}rel$ **obtains** $x'\ xs'$ **where**
$l'=x'\#xs'$ **and** $(x,x')\in R$ **and** $(xs,xs')\in\langle R\rangle list\text{-}rel$
⟨*proof*⟩

lemma *list-relE4*:
assumes $(l,x'\#xs')\in\langle R\rangle list\text{-}rel$ **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* $\Longrightarrow$ *length* l = *length* l'
⟨*proof*⟩

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*$)$
⟨*proof*⟩
lemma *list-rel-split-left-iff*:
$(l,y\#ys)\in\langle R\rangle$*list-rel* $\longleftrightarrow$ $(\exists\, x\ xs.\ l{=}x\#xs \wedge (x,y)\in R \wedge (xs,ys)\in\langle R\rangle$*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 $\bigwedge x.\ x{\in}A \Longrightarrow \exists\, y{\in}B.\ (x,y)\in R$
assumes $\bigwedge y.\ y{\in}B \Longrightarrow \exists\, x{\in}A.\ (x,y)\in R$
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-eq*: *single-valued R* ⟹ ⟨*R*⟩*old-set-rel* = ⟨*R*⟩*set-rel*
⟨*proof*⟩

lemma *set-rel-simp*[*simp*]:
({},{})∈⟨*R*⟩*set-rel*
⟨*proof*⟩

lemma *set-rel-empty-iff*[*simp*]:
({},*y*)∈⟨*A*⟩*set-rel* ⟷ *y*={}
(*x*,{})∈⟨*A*⟩*set-rel* ⟷ *x*={}
⟨*proof*⟩

lemma *set-relD1*: (*s*,*s'*)∈⟨*R*⟩*set-rel* ⟹ *x*∈*s* ⟹ ∃ *x'*∈*s'*. (*x*,*x'*)∈*R*
⟨*proof*⟩

lemma *set-relD2*: (*s*,*s'*)∈⟨*R*⟩*set-rel* ⟹ *x'*∈*s'* ⟹ ∃ *x*∈*s*. (*x*,*x'*)∈*R*
⟨*proof*⟩

lemma *set-relE1*[*consumes 2*]:
assumes (*s*,*s'*)∈⟨*R*⟩*set-rel* *x*∈*s*
obtains *x'* **where** *x'*∈*s'* (*x*,*x'*)∈*R*
⟨*proof*⟩

lemma *set-relE2*[*consumes 2*]:
assumes (*s*,*s'*)∈⟨*R*⟩*set-rel* *x'*∈*s'*
obtains *x* **where** *x*∈*s* (*x*,*x'*)∈*R*
⟨*proof*⟩

1.1.3 Automation

A solver for relator properties

lemma *relprop-triggers*:
⋀*R*. *single-valued R* ⟹ *single-valued R*
⋀*R*. *R*=*Id* ⟹ *R*=*Id*
⋀*R*. *R*=*Id* ⟹ *Id*=*R*
⋀*R*. *Range R* = *UNIV* ⟹ *Range R* = *UNIV*
⋀*R*. *Range R* = *UNIV* ⟹ *UNIV* = *Range R*
⋀*R R'*. *R*⊆*R'* ⟹ *R*⊆*R'*
⟨*proof*⟩

⟨*ML*⟩

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

lemma
relprop-UNIV-orient[*relator-props*]: $R{=}UNIV \Longrightarrow UNIV{=}R$
⟨*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*: $S{=}UNIV \longleftrightarrow (\forall x.\ x{\in}S)$ ⟨*proof*⟩

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

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

lemma *fun-rel-id*[*relator-props*]: ⟦*R1*=*Id*; *R2*=*Id*⟧ ⟹ *R1* → *R2* = *Id*
⟨*proof*⟩

lemma *fun-rel-id-simp*[*simp*]: *Id*→*Id* = *Id* ⟨*proof*⟩

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

lemma *fun-rel-mono*[*relator-props*]: $[\![\ R1 \subseteq R2;\ R3 \subseteq R4\ ]\!] \Longrightarrow R2 \to R3 \subseteq R1 \to R4$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

⟦*single-valued Rl*; *single-valued Rr*⟧ ⟹ *single-valued* (⟨*Rl*,*Rr*⟩*sum-rel*)
⟨*proof*⟩

lemma *sum-rel-id*[*relator-props*]: ⟦*Rl*=*Id*; *Rr*=*Id*⟧ ⟹ ⟨*Rl*,*Rr*⟩*sum-rel* = *Id*
⟨*proof*⟩

lemma *sum-rel-id-simp*[*simp*]: ⟨*Id*,*Id*⟩*sum-rel* = *Id* ⟨*proof*⟩

lemma *sum-rel-mono*[*relator-props*]:
⟦ *Rl*⊆*Rl'*; *Rr*⊆*Rr'* ⟧ ⟹ ⟨*Rl*,*Rr*⟩*sum-rel* ⊆ ⟨*Rl'*,*Rr'*⟩*sum-rel*
⟨*proof*⟩

lemma *sum-rel-range*[*relator-props*]:
⟦ *Range Rl*=*UNIV*; *Range Rr*=*UNIV* ⟧ ⟹ *Range* (⟨*Rl*,*Rr*⟩*sum-rel*) = *UNIV*
⟨*proof*⟩

lemma *list-rel-sv-iff*:
single-valued (⟨*R*⟩*list-rel*) ⟷ *single-valued R*
⟨*proof*⟩

lemma *list-rel-sv*[*relator-props*]:
single-valued R ⟹ *single-valued* (⟨*R*⟩*list-rel*)
⟨*proof*⟩

lemma *list-rel-id*[*relator-props*]: ⟦*R*=*Id*⟧ ⟹ ⟨*R*⟩*list-rel* = *Id*
⟨*proof*⟩

lemma *list-rel-id-simp*[*simp*]: ⟨*Id*⟩*list-rel* = *Id* ⟨*proof*⟩

lemma *list-rel-mono*[*relator-props*]:
assumes *A*: *R*⊆*R'*
shows ⟨*R*⟩*list-rel* ⊆ ⟨*R'*⟩*list-rel*
⟨*proof*⟩

lemma *list-rel-range*[*relator-props*]:
assumes *A*: *Range R* = *UNIV*
shows *Range* (⟨*R*⟩*list-rel*) = *UNIV*
⟨*proof*⟩

lemma *bijective-imp-sv*:
bijective R ⟹ *single-valued R*
bijective R ⟹ *single-valued* (R^{-1})
⟨*proof*⟩

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

Pointwise refinement for set types:

lemma *set-rel-sv*[*relator-props*]:
single-valued R $\Longrightarrow$ *single-valued* (⟨*R*⟩*set-rel*)
⟨*proof*⟩

lemma *set-rel-id*[*relator-props*]: *R*=*Id* $\Longrightarrow$ ⟨*R*⟩*set-rel* = *Id*
⟨*proof*⟩

lemma *set-rel-id-simp*[*simp*]: ⟨*Id*⟩*set-rel* = *Id* ⟨*proof*⟩

lemma *set-rel-csv*[*relator-props*]:
⟦ *single-valued* (R^{-1}) ⟧
$\Longrightarrow$ *single-valued* ((⟨*R*⟩*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**
build-rel α *I* ≡ {(*c*,*a*) . *a*=α *c* ∧ *I c*}
abbreviation *br*≡*build-rel*
lemmas *br-def*[*refine-rel-defs*] = *build-rel-def*

lemma *in-br-conv*: (*c*,*a*)∈*br* α *I* $\longleftrightarrow$ *a*=α *c* ∧ *I c*
⟨*proof*⟩

lemma *brI*[*intro?*]: ⟦ *a*=α *c*; *I c* ⟧ $\Longrightarrow$ (*c*,*a*)∈*br* α *I*
⟨*proof*⟩

lemma *br-id*[*simp*]: *br id* (λ-. *True*) = *Id*
⟨*proof*⟩

lemma *br-chain*:
(*build-rel* β *J*) *O* (*build-rel* α *I*) = *build-rel* (α∘β) (λ*s*. *J s* ∧ *I* (β *s*))
⟨*proof*⟩

lemma *br-sv*[*simp*, *intro*!,*relator-props*]: *single-valued* (*br* α *I*)
⟨*proof*⟩

lemma *converse-br-sv-iff*[*simp*]:
single-valued (*converse* (*br* α *I*)) $\longleftrightarrow$ *inj-on* α (*Collect I*)
⟨*proof*⟩

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

lemma *br-comp-alt*: *br* α *I O R* = { (*c*,*a*) . *I c* ∧ (α *c*,*a*)∈*R* }

⟨*proof*⟩

lemma *br-comp-alt'*:
{(*c*,*a*) . *a*=*α c* ∧ *I c*} *O R* = { (*c*,*a*) . *I c* ∧ (*α c*,*a*)∈*R* }
⟨*proof*⟩

lemma *single-valued-as-brE*:
assumes *single-valued R*
obtains *α invar* **where** *R*=*br α invar*
⟨*proof*⟩

lemma *sv-add-invar*:
single-valued R ⟹ *single-valued* {(*c*, *a*). (*c*, *a*) ∈ *R* ∧ *I c*}
⟨*proof*⟩

lemma *br-Image-conv*[*simp*]: *br α I `` S* = {*α x* | *x*. *x*∈*S* ∧ *I x*}
⟨*proof*⟩

1.1.6 Miscellanneous

lemma *rel-cong*: (*f*,*g*)∈*Id* ⟹ (*x*,*y*)∈*Id* ⟹ (*f x*, *g y*)∈*Id* ⟨*proof*⟩
lemma *rel-fun-cong*: (*f*,*g*)∈*Id* ⟹ (*f x*, *g x*)∈*Id* ⟨*proof*⟩
lemma *rel-arg-cong*: (*x*,*y*)∈*Id* ⟹ (*f x*, *f y*)∈*Id* ⟨*proof*⟩

1.1.7 Conversion between Predicate and Set Based Relators

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

definition *rel2p R x y* ≡ (*x*,*y*)∈*R*
definition *p2rel P* ≡ {(*x*,*y*). *P x y*}

lemma *rel2pD*: ⟦*rel2p R a b*⟧ ⟹ (*a*,*b*)∈*R* ⟨*proof*⟩
lemma *p2relD*: ⟦(*a*,*b*) ∈ *p2rel R*⟧ ⟹ *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-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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma [*p2rel*]: *left-unique A* ⟹ *p2rel* (*rel-set A*) = (⟨*p2rel A*⟩*set-rel*)
⟨*proof*⟩

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

lemma *rel2p-inj*[*simp*]: *rel2p A* = *rel2p B* ⟷ *A*=*B*
⟨*proof*⟩

lemma *rel2p-left-unique*: *left-unique* (*rel2p A*) = *single-valued* (A^{-1})
⟨*proof*⟩

lemma *rel2p-right-unique*: *right-unique* (*rel2p A*) = *single-valued A*
⟨*proof*⟩

lemma *rel2p-bi-unique*: *bi-unique* (*rel2p A*) ⟷ *single-valued A* ∧ *single-valued* (A^{-1})
⟨*proof*⟩

lemma *p2rel-left-unique*: *single-valued* (($p2rel\ A)^{-1}$) = *left-unique A*
⟨*proof*⟩

lemma *p2rel-right-unique*: *single-valued* (*p2rel A*) = *right-unique A*
⟨*proof*⟩

1.1.8 More Properties

lemma *list-rel-compp*: ⟨*A O B*⟩*list-rel* = ⟨*A*⟩*list-rel O* ⟨*B*⟩*list-rel*
⟨*proof*⟩

lemma *option-rel-compp*: ⟨*A O B*⟩*option-rel* = ⟨*A*⟩*option-rel O* ⟨*B*⟩*option-rel*
⟨*proof*⟩

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

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

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

lemma *map-in-list-rel-conv*:
shows (*l*, *map* α *l*) ∈ $\langle br\ \alpha\ I\rangle$*list-rel* ⟷ (∀ *x*∈*set l*. *I x*)
⟨*proof*⟩

lemma *br-set-rel-alt*: (*s*′,*s*)∈$\langle br\ \alpha\ I\rangle$*set-rel* ⟷ (*s*=$\alpha$‘*s*′ ∧ (∀ *x*∈*s*′. *I x*))
⟨*proof*⟩

lemma *finite-Image-sv*: *single-valued R* ⟹ *finite s* ⟹ *finite* (*R*‘‘*s*)
⟨*proof*⟩

lemma *finite-set-rel-transfer*: ⟦(*s*,*s*′)∈$\langle R\rangle$*set-rel*; *single-valued R*; *finite s*⟧ ⟹ *finite s*′
⟨*proof*⟩

lemma *finite-set-rel-transfer-back*: ⟦(*s*,*s*′)∈$\langle R\rangle$*set-rel*; *single-valued* (R^{-1}); *finite s*′⟧ ⟹ *finite s*
⟨*proof*⟩

end

1.2 Basic Parametricity Reasoning

theory *Param-Tool*
imports *Relators*
begin

1.2.1 Auxiliary Lemmas

lemma *tag-both*: ⟦ (*Let x f*,*Let x*′ *f*′)∈*R* ⟧ ⟹ (*f x*,*f*′ *x*′)∈*R* ⟨*proof*⟩
lemma *tag-rhs*: ⟦ (*c*,*Let x f*)∈*R* ⟧ ⟹ (*c*,*f x*)∈*R* ⟨*proof*⟩
lemma *tag-lhs*: ⟦ (*Let x f*,*a*)∈*R* ⟧ ⟹ (*f x*,*a*)∈*R* ⟨*proof*⟩

lemma *tagged-fun-relD-both*:
⟦ (*f*,*f*′)∈*A*→*B*; (*x*,*x*′)∈*A* ⟧ ⟹ (*Let x f*,*Let x*′ *f*′)∈*B*
and *tagged-fun-relD-rhs*: ⟦ (*f*,*f*′)∈*A*→*B*; (*x*,*x*′)∈*A* ⟧ ⟹ (*f x*,*Let x*′ *f*′)∈*B*
and *tagged-fun-relD-lhs*: ⟦ (*f*,*f*′)∈*A*→*B*; (*x*,*x*′)∈*A* ⟧ ⟹ (*Let x f*,*f*′ *x*′)∈*B*
and *tagged-fun-relD-none*: ⟦ (*f*,*f*′)∈*A*→*B*; (*x*,*x*′)∈*A* ⟧ ⟹ (*f x*,*f*′ *x*′)∈*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*]:
({},{})∈⟨*R*⟩*set-rel* ⟨*proof*⟩

lemma *param-member*[*param*]:
⟦*single-valued R*; *single-valued* (R^{-1})⟧ ⟹ ((∈), (∈)) ∈ *R* → ⟨*R*⟩*set-rel* → *bool-rel*

⟨*proof*⟩

lemma *param-insert*[*param*]:
(*insert*,*insert*)∈*R*→⟨*R*⟩*set-rel*→⟨*R*⟩*set-rel*
⟨*proof*⟩

lemma *param-union*[*param*]:
((∪), (∪)) ∈ ⟨*R*⟩*set-rel* → ⟨*R*⟩*set-rel* → ⟨*R*⟩*set-rel*
⟨*proof*⟩

lemma *param-inter*[*param*]:
assumes *single-valued R* *single-valued* (R^{-1})
shows ((∩), (∩)) ∈ ⟨*R*⟩*set-rel* → ⟨*R*⟩*set-rel* → ⟨*R*⟩*set-rel*
⟨*proof*⟩

lemma *param-diff*[*param*]:
assumes *single-valued R* *single-valued* (R^{-1})
shows ((−), (−)) ∈ ⟨*R*⟩*set-rel* → ⟨*R*⟩*set-rel* → ⟨*R*⟩*set-rel*
⟨*proof*⟩

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

$[\![single\text{-}valued\ R;\ single\text{-}valued\ (R^{-1})]\!] \Longrightarrow ((\subseteq), (\subseteq)) \in \langle R\rangle set\text{-}rel \to \langle R\rangle set\text{-}rel \to bool\text{-}rel$
$\langle proof\rangle$

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

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

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

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

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

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

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

1.3.2 Standard HOL Constructs

lemma *param-if*[*param*]:
assumes $(c,c')\in Id$
assumes $[\![c;c']\!] \Longrightarrow (t,t')\in R$
assumes $[\![\neg c;\neg c']\!] \Longrightarrow (e,e')\in R$
shows $(If\ c\ t\ e,\ If\ c'\ t'\ e')\in R$
$\langle proof\rangle$

lemma *param-Let*[*param*]:
$(Let,Let)\in Ra \to (Ra\to Rr) \to Rr$
$\langle proof\rangle$

1.3.3 Functions

lemma *param-id*[*param*]: $(id,id)\in R\to R$ $\langle proof\rangle$

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

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

1.3.4 Boolean

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

lemma *param-bool*[*param*]:
(*True*,*True*)∈*Id*
(*False*,*False*)∈*Id*
(*conj*,*conj*)∈*Id*→*Id*→*Id*
(*disj*,*disj*)∈*Id*→*Id*→*Id*
(*Not*,*Not*)∈*Id*→*Id*
(*case-bool*,*case-bool*)∈*R*→*R*→*Id*→*R*
(*old.rec-bool*,*old.rec-bool*)∈*R*→*R*→*Id*→*R*
((⟷), (⟷))∈*Id*→*Id*→*Id*
((⟶), (⟶))∈*Id*→*Id*→*Id*
⟨*proof*⟩

lemma *param-and-cong1*: ⟦ (*a*,*a'*)∈*bool-rel*; ⟦*a*; *a'*⟧ ⟹ (*b*,*b'*)∈*bool-rel* ⟧ ⟹ (*a*∧*b*,*a'*∧*b'*)∈*bool-rel*
⟨*proof*⟩
lemma *param-and-cong2*: ⟦ (*a*,*a'*)∈*bool-rel*; ⟦*a*; *a'*⟧ ⟹ (*b*,*b'*)∈*bool-rel* ⟧ ⟹ (*b*∧*a*,*b'*∧*a'*)∈*bool-rel*
⟨*proof*⟩

1.3.5 Nat

lemma *param-nat1*[*param*]:
(*0*, *0*::*nat*) ∈ *Id*
(*Suc*, *Suc*) ∈ *Id* → *Id*
(*1*, *1*::*nat*) ∈ *Id*
(*numeral n*::*nat*,*numeral n*::*nat*) ∈ *Id*
((<), (<) ::*nat* ⇒ -) ∈ *Id* → *Id* → *Id*
((≤), (≤) ::*nat* ⇒ -) ∈ *Id* → *Id* → *Id*
((=), (=) ::*nat* ⇒ -) ∈ *Id* → *Id* → *Id*
((+) ::*nat*⇒-,(+))∈*Id*→*Id*→*Id*
((−) ::*nat*⇒-,(−))∈*Id*→*Id*→*Id*
((∗) ::*nat*⇒-,(∗))∈*Id*→*Id*→*Id*
((*div*) ::*nat*⇒-,(*div*))∈*Id*→*Id*→*Id*
((*mod*) ::*nat*⇒-,(*mod*))∈*Id*→*Id*→*Id*
⟨*proof*⟩

lemma *param-case-nat*[*param*]:
(*case-nat*,*case-nat*)∈*Ra* → (*Id* → *Ra*) → *Id* → *Ra*

⟨*proof*⟩

lemma *param-rec-nat*[*param*]:
(*rec-nat*,*rec-nat*) ∈ *R* → (*Id* → *R* → *R*) → *Id* → *R*
⟨*proof*⟩

1.3.6 Int

lemma *param-int*[*param*]:
(*0*, *0*::*int*) ∈ *Id*
(*1*, *1*::*int*) ∈ *Id*
(*numeral n*::*int*,*numeral n*::*int*) ∈ *Id*
((<), (<) ::*int* ⇒ -) ∈ *Id* → *Id* → *Id*
((≤), (≤) ::*int* ⇒ -) ∈ *Id* → *Id* → *Id*
((=), (=) ::*int* ⇒ -) ∈ *Id* → *Id* → *Id*
((+) ::*int*⇒-,(+))∈*Id*→*Id*→*Id*
((−) ::*int*⇒-,(−))∈*Id*→*Id*→*Id*
((∗) ::*int*⇒-,(∗))∈*Id*→*Id*→*Id*
((*div*) ::*int*⇒-,(*div*))∈*Id*→*Id*→*Id*
((*mod*) ::*int*⇒-,(*mod*))∈*Id*→*Id*→*Id*
⟨*proof*⟩

1.3.7 Product

lemma *param-unit*[*param*]: ((),())∈*unit-rel* ⟨*proof*⟩

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

lemma *param-prod*[*param*]:
(*Pair*,*Pair*)∈*Ra* → *Rb* → ⟨*Ra*,*Rb*⟩*prod-rel*
(*case-prod*,*case-prod*) ∈ (*Ra* → *Rb* → *Rr*) → ⟨*Ra*,*Rb*⟩*prod-rel* → *Rr*
(*old.rec-prod*,*old.rec-prod*) ∈ (*Ra* → *Rb* → *Rr*) → ⟨*Ra*,*Rb*⟩*prod-rel* → *Rr*
(*fst*,*fst*)∈⟨*Ra*,*Rb*⟩*prod-rel* → *Ra*
(*snd*,*snd*)∈⟨*Ra*,*Rb*⟩*prod-rel* → *Rb*
⟨*proof*⟩

lemma *param-case-prod′*:
⟦ (*p*,*p′*)∈⟨*Ra*,*Rb*⟩*prod-rel*;
⋀*a b a′ b′*. ⟦ *p*=(*a*,*b*); *p′*=(*a′*,*b′*); (*a*,*a′*)∈*Ra*; (*b*,*b′*)∈*Rb* ⟧
⟹ (*f a b*, *f′ a′ b′*)∈*R*
⟧ ⟹ (*case-prod f p*, *case-prod f′ p′*) ∈ *R*
⟨*proof*⟩

lemma *param-case-prod′′*:
⟦
⋀*a b a′ b′*. ⟦*p*=(*a*,*b*); *p′*=(*a′*,*b′*)⟧ ⟹ (*f a b*,*f′ a′ b′*)∈*R*
⟧ ⟹ (*case-prod f p*, *case-prod f′ p′*)∈*R*
⟨*proof*⟩

lemma *param-map-prod*[*param*]:
(*map-prod*, *map-prod*)
∈ (*Ra*→*Rb*) → (*Rc*→*Rd*) → ⟨*Ra*,*Rc*⟩*prod-rel* → ⟨*Rb*,*Rd*⟩*prod-rel*
⟨*proof*⟩

lemma *param-apfst*[*param*]:
(*apfst*,*apfst*)∈(*Ra*→*Rb*)→⟨*Ra*,*Rc*⟩*prod-rel*→⟨*Rb*,*Rc*⟩*prod-rel*
⟨*proof*⟩

lemma *param-apsnd*[*param*]:
(*apsnd*,*apsnd*)∈(*Rb*→*Rc*)→⟨*Ra*,*Rb*⟩*prod-rel*→⟨*Ra*,*Rc*⟩*prod-rel*
⟨*proof*⟩

lemma *param-curry*[*param*]:
(*curry*,*curry*) ∈ (⟨*Ra*,*Rb*⟩*prod-rel* → *Rc*) → *Ra* → *Rb* → *Rc*
⟨*proof*⟩

lemma *param-uncurry*[*param*]: (*uncurry*,*uncurry*) ∈ (*A*→*B*→*C*) → *A*×$_r$*B*→*C*
⟨*proof*⟩

lemma *param-prod-swap*[*param*]: (*prod.swap*, *prod.swap*)∈*A*×$_r$*B* → *B*×$_r$*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 (λ*a*. ∀ *x xa*. (*x*, *xa*) ∈ *Rb* ⟶ (*a x*, *fixp-fun F' xa*) ∈ *Ra*)
assumes *bot*: ⋀*x xa*. (*x*, *xa*) ∈ *Rb* ⟹ (*lub* {}, *fixp-fun F' xa*) ∈ *Ra*
assumes *F*: (*F*,*F'*)∈(*Rb*→*Ra*)→*Rb*→*Ra*
assumes *A*: (*x*,*x'*)∈*Rb*
shows (*fixp-fun F x*, *fixp-fun F' x'*)∈*Ra*
⟨*proof*⟩
end

1.3.8 Option

lemma *param-option*[*param*]:
(*None*,*None*)∈⟨*R*⟩*option-rel*
(*Some*,*Some*)∈*R* → ⟨*R*⟩*option-rel*
(*case-option*,*case-option*)∈*Rr*→(*R* → *Rr*)→⟨*R*⟩*option-rel* → *Rr*
(*rec-option*,*rec-option*)∈*Rr*→(*R* → *Rr*)→⟨*R*⟩*option-rel* → *Rr*
⟨*proof*⟩

lemma *param-map-option*[*param*]: (*map-option*, *map-option*) ∈ (*A* → *B*) → ⟨*A*⟩*option-rel*
→ ⟨*B*⟩*option-rel*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *the-paramL*: ⟦*l*≠*None*; (*l*,*r*)∈⟨*R*⟩*option-rel*⟧ ⟹ (*the l*, *the r*)∈*R*
⟨*proof*⟩

lemma *the-paramR*: ⟦*r*≠*None*; (*l*,*r*)∈⟨*R*⟩*option-rel*⟧ ⟹ (*the l*, *the r*)∈*R*
⟨*proof*⟩

lemma *the-default-param*[*param*]:
(*the-default*, *the-default*) ∈ *R* → ⟨*R*⟩*option-rel* → *R*
⟨*proof*⟩

1.3.9 Sum

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

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*]: (*is-Inl*,*is-Inl*) ∈ ⟨*Ra*,*Rb*⟩*sum-rel* → *bool-rel*
⟨*proof*⟩
lemma *is-Inr-param*[*param*]: (*is-Inr*,*is-Inr*) ∈ ⟨*Ra*,*Rb*⟩*sum-rel* → *bool-rel*
⟨*proof*⟩

lemma *sum-projl-param*[*param*]:
⟦*is-Inl s*; (*s′*,*s*)∈⟨*Ra*,*Rb*⟩*sum-rel*⟧
⟹ (*Sum-Type.sum.projl s′*,*Sum-Type.sum.projl s*) ∈ *Ra*
⟨*proof*⟩

lemma *sum-projr-param*[*param*]:
⟦*is-Inr s*; (*s'*,*s*)∈⟨*Ra*,*Rb*⟩*sum-rel*⟧
⟹ (*Sum-Type.sum.projr s'*,*Sum-Type.sum.projr s*) ∈ *Rb*
⟨*proof*⟩

1.3.10 List

lemma *list-rel-append1*: (*as* @ *bs*, *l*) ∈ ⟨*R*⟩*list-rel*
⟷ (∃ *cs ds*. *l* = *cs*@*ds* ∧ (*as*,*cs*)∈⟨*R*⟩*list-rel* ∧ (*bs*,*ds*)∈⟨*R*⟩*list-rel*)
⟨*proof*⟩

lemma *list-rel-append2*: (*l*,*as* @ *bs*) ∈ ⟨*R*⟩*list-rel*
⟷ (∃ *cs ds*. *l* = *cs*@*ds* ∧ (*cs*,*as*)∈⟨*R*⟩*list-rel* ∧ (*ds*,*bs*)∈⟨*R*⟩*list-rel*)
⟨*proof*⟩

lemma *param-append*[*param*]:
(*append*, *append*)∈⟨*R*⟩*list-rel* → ⟨*R*⟩*list-rel* → ⟨*R*⟩*list-rel*
⟨*proof*⟩

lemma *param-list1*[*param*]:
(*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*
⟨*proof*⟩

lemma *param-rec-list*[*param*]:
(*rec-list*,*rec-list*)
∈ *Ra* → (*Rb* → ⟨*Rb*⟩*list-rel* → *Ra* → *Ra*) → ⟨*Rb*⟩*list-rel* → *Ra*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *param-map*[*param*]:
(*map*,*map*)∈(*R1*→*R2*) → ⟨*R1*⟩*list-rel* → ⟨*R2*⟩*list-rel*
⟨*proof*⟩

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*
⟨*proof*⟩

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

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*
⟨*proof*⟩

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

end

lemma *param-hd*[*param*]: *l*≠[] ⟹ (*l*',*l*)∈⟨*A*⟩*list-rel* ⟹ (*hd l*', *hd l*)∈*A*
⟨*proof*⟩

lemma *param-last*[*param*]:
assumes *y* ≠ []
assumes (*x*, *y*) ∈ ⟨*A*⟩*list-rel*
shows (*last x*, *last y*) ∈ *A*
⟨*proof*⟩

lemma *param-rotate1*[*param*]: (*rotate1*, *rotate1*) ∈ ⟨*A*⟩*list-rel* → ⟨*A*⟩*list-rel*
⟨*proof*⟩

schematic-goal *param-take*[*param*]: (*take*,*take*)∈(*?R*::(-×-) *set*)
⟨*proof*⟩

schematic-goal *param-drop*[*param*]: (*drop*,*drop*)∈(*?R*::(-×-) *set*)
⟨*proof*⟩

schematic-goal *param-length*[*param*]:
(*length*,*length*)∈(*?R*::(-×-) *set*)
⟨*proof*⟩

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

(*list-eq*,*list-eq*) ∈
(*R* → *R* → *Id*) → ⟨*R*⟩*list-rel* → ⟨*R*⟩*list-rel* → *Id*
⟨*proof*⟩

lemma *id-list-eq-aux*[*simp*]: (*list-eq* (=)) = (=)
⟨*proof*⟩

lemma *param-list-equals*[*param*]:
⟦ ((=), (=)) ∈ *R*→*R*→*Id* ⟧
⟹ ((=), (=)) ∈ ⟨*R*⟩*list-rel* → ⟨*R*⟩*list-rel* → *Id*
⟨*proof*⟩

lemma *param-tl*[*param*]:
(*tl*,*tl*) ∈ ⟨*R*⟩*list-rel* → ⟨*R*⟩*list-rel*
⟨*proof*⟩

primrec *list-all-rec* **where**
list-all-rec P [] ⟷ *True*
| *list-all-rec P* (*a*#*l*) ⟷ *P a* ∧ *list-all-rec P l*

primrec *list-ex-rec* **where**
list-ex-rec P [] ⟷ *False*
| *list-ex-rec P* (*a*#*l*) ⟷ *P a* ∨ *list-ex-rec P l*

lemma *list-all-rec-eq*: (∀ *x*∈*set l*. *P x*) = *list-all-rec P l*
⟨*proof*⟩

lemma *list-ex-rec-eq*: (∃ *x*∈*set l*. *P x*) = *list-ex-rec P l*
⟨*proof*⟩

lemma *param-list-ball*[*param*]:
⟦(*P*,*P*′)∈(*Ra*→*Id*); (*l*,*l*′)∈⟨*Ra*⟩ *list-rel*⟧
⟹ (∀ *x*∈*set l*. *P x*, ∀ *x*∈*set l*′. *P*′ *x*) ∈ *Id*
⟨*proof*⟩

lemma *param-list-bex*[*param*]:
⟦(*P*,*P*′)∈(*Ra*→*Id*); (*l*,*l*′)∈⟨*Ra*⟩ *list-rel*⟧
⟹ (∃ *x*∈*set l*. *P x*, ∃ *x*∈*set l*′. *P*′ *x*) ∈ *Id*
⟨*proof*⟩

lemma *param-rev*[*param*]: (*rev*,*rev*) ∈ ⟨*R*⟩*list-rel* → ⟨*R*⟩*list-rel*
⟨*proof*⟩

lemma *param-foldli*[*param*]: (*foldli*, *foldli*)
∈ ⟨*Re*⟩*list-rel* → (*Rs*→*Id*) → (*Re*→*Rs*→*Rs*) → *Rs* → *Rs*
⟨*proof*⟩

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

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

lemma *param-nth*[*param*]:
assumes *I*: $i' < length\ l'$
assumes *IR*: $(i,i') \in nat\text{-}rel$
assumes *LR*: $(l,l') \in \langle R\rangle list\text{-}rel$
shows $(l!i, l'!i') \in R$
$\langle proof\rangle$

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

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

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

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

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

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

context
begin

qualified definition *all-interval-nat* :: $\langle (nat \Rightarrow bool) \Rightarrow nat \Rightarrow nat \Rightarrow bool\rangle$
where *all-interval-nat-eq* [*simp*]: $\langle all\text{-}interval\text{-}nat\ P\ m\ n \longleftrightarrow (\forall i \in \{m..<n\}.\ P\ i)\rangle$

end

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

lemma *param-dropWhile*[*param*]:
(*dropWhile*, *dropWhile*) ∈ (*a* → *bool-rel*) → ⟨*a*⟩*list-rel* → ⟨*a*⟩*list-rel*
⟨*proof*⟩

lemma *param-takeWhile*[*param*]:
(*takeWhile*, *takeWhile*) ∈ (*a* → *bool-rel*) → ⟨*a*⟩*list-rel* → ⟨*a*⟩*list-rel*
⟨*proof*⟩

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

⟨*ML*⟩

Main method

⟨*ML*⟩

2.1.2 Tools

⟨*ML*⟩

2.1.3 Advanced Debugging

⟨*ML*⟩

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** ‹$:::_r$› *10*)
notation (*input*) *ind-ANNOT* (**infix** ‹$::\#_r$› *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** ‹λ′› *10*)
end

no-notation (*input*) *APP* (**infixl** ‹$› *900*)
no-notation (*input*) *ABS* (**binder** ‹λ′› *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 hash-code 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* ≡ *x* = *y*
lemma *STRUCT-EQ-tagI*: *x*=*y* ⟹ *STRUCT-EQ-tag x y* ⟨*proof*⟩

⟨*ML*⟩

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

definition [*simp*]: *REL-IS-ID R* ≡ *R=Id*
definition [*simp*]: *REL-FORCE-ID R* ≡ *R=Id*
lemma *REL-IS-ID-trigger*: *R=Id* ⟹ *REL-IS-ID R* ⟨*proof*⟩
lemma *REL-FORCE-ID-trigger*: *R=Id* ⟹ *REL-FORCE-ID R* ⟨*proof*⟩

⟨*ML*⟩

abbreviation *PREFER-id R* ≡ *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 ::$_i$ *i-bool*
False ::$_i$ *i-bool*
conj ::$_i$ *i-bool* →$_i$ *i-bool* →$_i$ *i-bool*
(⟷) ::$_i$ *i-bool* →$_i$ *i-bool* →$_i$ *i-bool*
(⟶) ::$_i$ *i-bool* →$_i$ *i-bool* →$_i$ *i-bool*
disj ::$_i$ *i-bool* →$_i$ *i-bool* →$_i$ *i-bool*
Not ::$_i$ *i-bool* →$_i$ *i-bool*
case-bool ::$_i$ *I* →$_i$ *I* →$_i$ *i-bool* →$_i$ *I*
old.rec-bool ::$_i$ *I* →$_i$ *I* →$_i$ *i-bool* →$_i$ *I*
⟨*proof*⟩

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*
⟨*proof*⟩

2.2.3 Standard Type Classes

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

We allow these operators for all interfaces.

lemma [*autoref-itype*]:
$(<) ::_i I \to_i I \to_i$ *i-bool*
$(\leq) ::_i I \to_i I \to_i$ *i-bool*
$(=) ::_i I \to_i I \to_i$ *i-bool*
$(+) ::_i I \to_i I \to_i I$
$(-) ::_i I \to_i I \to_i I$
$(div) ::_i I \to_i I \to_i I$
$(mod) ::_i I \to_i I \to_i I$
$(*) ::_i I \to_i I \to_i I$
$0 ::_i I$
$1 ::_i I$
numeral $x ::_i I$
uminus $::_i I \to_i I$
$\langle proof \rangle$

lemma *pat-num-generic*[*autoref-op-pat*]:
$0 \equiv OP\ 0 :::_i I$
$1 \equiv OP\ 1 :::_i I$
numeral $x \equiv (OP\ (numeral\ x) :::_i I)$
$\langle proof \rangle$

lemma [*autoref-rules*]:
assumes *PRIO-TAG-GEN-ALGO*
shows $((<), (<)) \in Id \to Id \to$ *bool-rel*
and $((\leq), (\leq)) \in Id \to Id \to$ *bool-rel*
and $((=), (=)) \in Id \to Id \to$ *bool-rel*
and $(numeral\ x, OP\ (numeral\ x) ::: Id) \in Id$
and $(uminus, uminus) \in Id \to Id$
and $(0,0) \in Id$
and $(1,1) \in Id$
$\langle proof \rangle$

2.2.4 Functional Combinators

lemma [*autoref-itype*]: $id ::_i I \to_i I\ \langle proof \rangle$
lemma *autoref-id*[*autoref-rules*]: $(id,id) \in R \to R\ \langle proof \rangle$

term (o)
lemma [*autoref-itype*]: $(\circ) ::_i (Ia \to_i Ib) \to_i (Ic \to_i Ia) \to_i Ic \to_i Ib$
$\langle proof \rangle$
lemma *autoref-comp*[*autoref-rules*]:
$((o), (o)) \in (Ra \to Rb) \to (Rc \to Ra) \to Rc \to Rb$
$\langle proof \rangle$

lemma [*autoref-itype*]: $If ::_i$ *i-bool* $\to_i I \to_i I \to_i I\ \langle proof \rangle$
lemma *autoref-If*[*autoref-rules*]: $(If,If) \in Id \to R \to R \to R\ \langle proof \rangle$
lemma *autoref-If-cong*[*autoref-rules*]:
assumes $(c',c) \in Id$

assumes *REMOVE-INTERNAL c* $\Longrightarrow$ (*t'*,*t*)∈*R*
assumes ¬ *REMOVE-INTERNAL c* $\Longrightarrow$ (*e'*,*e*)∈*R*
shows (*If c' t' e'*,(*OP If* ::: *Id*→*R*→*R*→*R*)$*c*$*t*$*e*)∈*R*
⟨*proof*⟩

lemma [*autoref-itype*]: *Let* ::$_i$ *Ix* →$_i$ (*Ix*→$_i$*Iy*) →$_i$ *Iy* ⟨*proof*⟩
lemma *autoref-Let*:
(*Let*,*Let*)∈*Ra* → (*Ra*→*Rr*) → *Rr*
⟨*proof*⟩

lemma *autoref-Let-cong*[*autoref-rules*]:
assumes (*x'*,*x*)∈*Ra*
assumes ⋀*y y'*. *REMOVE-INTERNAL* (*x*=*y*) $\Longrightarrow$ (*y'*,*y*)∈*Ra* $\Longrightarrow$ (*f' y'*, *f*$*y*)∈*Rr*
shows (*Let x' f'*,(*OP Let* ::: *Ra* → (*Ra* → *Rr*) → *Rr*)$*x*$*f*)∈*Rr*
⟨*proof*⟩

end

2.2.5 Unit

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

lemma [*autoref-rules*]: ((),())∈*unit-rel* ⟨*proof*⟩

2.2.6 Nat

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

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma *pat-num-nat*[*autoref-op-pat*]:
0::*nat* ≡ *OP 0* :::$_i$ *i-nat*
1::*nat* ≡ *OP 1* :::$_i$ *i-nat*
(*numeral x*)::*nat* ≡ (*OP* (*numeral x*) :::$_i$ *i-nat*)
⟨*proof*⟩

lemma *autoref-nat*[*autoref-rules*]:
(*0*, *0*::*nat*) ∈ *nat-rel*
(*Suc*, *Suc*) ∈ *nat-rel* → *nat-rel*
(*1*, *1*::*nat*) ∈ *nat-rel*
(*numeral n*::*nat*,*numeral n*::*nat*) ∈ *nat-rel*
((<), (<) ::*nat* ⇒ -) ∈ *nat-rel* → *nat-rel* → *bool-rel*
((≤), (≤) ::*nat* ⇒ -) ∈ *nat-rel* → *nat-rel* → *bool-rel*
((=), (=) ::*nat* ⇒ -) ∈ *nat-rel* → *nat-rel* → *bool-rel*
((+) ::*nat*⇒-,(+))∈*nat-rel*→*nat-rel*→*nat-rel*
((−) ::*nat*⇒-,(−))∈*nat-rel*→*nat-rel*→*nat-rel*
((*div*) ::*nat*⇒-,(*div*))∈*nat-rel*→*nat-rel*→*nat-rel*

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

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

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

lemma *autoref-int*[*autoref-rules* (**overloaded**)]:
$(0,\ 0 :: int) \in int\text{-}rel$
$(1,\ 1 :: int) \in int\text{-}rel$
$(numeral\ n :: int, numeral\ n :: int) \in int\text{-}rel$
$((<), (<) :: int \Rightarrow \text{-}) \in int\text{-}rel \to int\text{-}rel \to bool\text{-}rel$
$((\leq), (\leq) :: int \Rightarrow \text{-}) \in int\text{-}rel \to int\text{-}rel \to bool\text{-}rel$
$((=), (=) :: int \Rightarrow \text{-}) \in int\text{-}rel \to int\text{-}rel \to bool\text{-}rel$
$((+) :: int \Rightarrow \text{-}, (+)) \in int\text{-}rel \to int\text{-}rel \to int\text{-}rel$
$((-) :: int \Rightarrow \text{-}, (-)) \in int\text{-}rel \to int\text{-}rel \to int\text{-}rel$
$((div) :: int \Rightarrow \text{-}, (div)) \in int\text{-}rel \to int\text{-}rel \to int\text{-}rel$
$(uminus, uminus) \in int\text{-}rel \to int\text{-}rel$
$((*), (*)) \in int\text{-}rel \to int\text{-}rel \to int\text{-}rel$
$((mod), (mod)) \in int\text{-}rel \to int\text{-}rel \to int\text{-}rel$
$\langle proof \rangle$
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* $\langle proof \rangle$

lemma *prod-refine*[*autoref-rules*]:
(*Pair*,*Pair*)∈*Ra* → *Rb* → ⟨*Ra*,*Rb*⟩*prod-rel*
(*case-prod*,*case-prod*) ∈ (*Ra* → *Rb* → *Rr*) → ⟨*Ra*,*Rb*⟩*prod-rel* → *Rr*
(*old.rec-prod*,*old.rec-prod*) ∈ (*Ra* → *Rb* → *Rr*) → ⟨*Ra*,*Rb*⟩*prod-rel* → *Rr*
(*fst*,*fst*)∈⟨*Ra*,*Rb*⟩*prod-rel* → *Ra*
(*snd*,*snd*)∈⟨*Ra*,*Rb*⟩*prod-rel* → *Rb*
⟨*proof*⟩

definition *prod-eq eqa eqb x1 x2* ≡
case x1 of (*a1*,*b1*) ⇒ *case x2 of* (*a2*,*b2*) ⇒ *eqa a1 a2* ∧ *eqb b1 b2*

lemma *prod-eq-autoref*[*autoref-rules* (**overloaded**)]:
⟦*GEN-OP eqa* (=) (*Ra*→*Ra*→*Id*); *GEN-OP eqb* (=) (*Rb*→*Rb*→*Id*)⟧
⟹ (*prod-eq eqa eqb*,(=)) ∈ ⟨*Ra*,*Rb*⟩*prod-rel* → ⟨*Ra*,*Rb*⟩*prod-rel* → *Id*
⟨*proof*⟩

lemma *prod-eq-expand*[*autoref-struct-expand*]: (=) = *prod-eq* (=) (=)
⟨*proof*⟩
end

2.2.9 Option

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

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

lemma *autoref-opt*[*autoref-rules*]:
(*None*,*None*)∈⟨*R*⟩*option-rel*
(*Some*,*Some*)∈*R* → ⟨*R*⟩*option-rel*
(*case-option*,*case-option*)∈*Rr*→(*R* → *Rr*)→⟨*R*⟩*option-rel* → *Rr*
(*rec-option*,*rec-option*)∈*Rr*→(*R* → *Rr*)→⟨*R*⟩*option-rel* → *Rr*
⟨*proof*⟩

lemma *autoref-the*[*autoref-rules*]:
assumes *SIDE-PRECOND* (*x*≠*None*)
assumes (*x'*,*x*)∈⟨*R*⟩*option-rel*
shows (*the x'*, (*OP the* ::: ⟨*R*⟩*option-rel* → *R*)$*x*) ∈ *R*
⟨*proof*⟩

lemma *autoref-the-default*[*autoref-rules*]:
(*the-default*, *the-default*) ∈ *R* → ⟨*R*⟩*option-rel* → *R*
⟨*proof*⟩

definition [*simp*]: *is-None a* ≡ *case a of None* ⇒ *True* | - ⇒ *False*
lemma *pat-isNone*[*autoref-op-pat*]:
a=*None* ≡ (*OP is-None* :::$_i$ ⟨*I*⟩$_i$*i-option* →$_i$ *i-bool*)$*a*
None=*a* ≡ (*OP is-None* :::$_i$ ⟨*I*⟩$_i$*i-option* →$_i$ *i-bool*)$*a*

```
    ⟨proof⟩
  lemma autoref-is-None[param,autoref-rules]:
    (is-None,is-None)∈⟨R⟩option-rel → Id
    ⟨proof⟩

  lemma fold-is-None: x=None ⟷ is-None x ⟨proof⟩

  definition option-eq eq v1 v2 ≡
    case (v1,v2) of
      (None,None) ⇒ True
    | (Some x1, Some x2) ⇒ eq x1 x2
    | - ⇒ False

  lemma option-eq-autoref[autoref-rules (overloaded)]:
    ⟦GEN-OP eq (=) (R→R→Id)⟧
    ⟹ (option-eq eq,(=)) ∈ ⟨R⟩option-rel → ⟨R⟩option-rel → Id
    ⟨proof⟩

  lemma option-eq-expand[autoref-struct-expand]:
    (=) = option-eq (=)
    ⟨proof⟩
end
```

2.2.10 Sum-Types

```
  consts i-sum :: interface ⇒ interface ⇒ interface
  lemmas [autoref-rel-intf] = REL-INTFI[of sum-rel i-sum]

context begin interpretation autoref-syn ⟨proof⟩


  lemma autoref-sum[autoref-rules]:
    (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
    ⟨proof⟩

  definition sum-eq eql eqr s1 s2 ≡
    case (s1,s2) of
      (Inl x1, Inl x2) ⇒ eql x1 x2
    | (Inr x1, Inr x2) ⇒ eqr x1 x2
    | - ⇒ False

  lemma sum-eq-autoref[autoref-rules (overloaded)]:
    ⟦GEN-OP eql (=) (Rl→Rl→Id); GEN-OP eqr (=) (Rr→Rr→Id)⟧
    ⟹ (sum-eq eql eqr,(=)) ∈ ⟨Rl,Rr⟩sum-rel → ⟨Rl,Rr⟩sum-rel → Id
    ⟨proof⟩
```

lemma *sum-eq-expand*[*autoref-struct-expand*]: (=) = *sum-eq* (=) (=)
⟨*proof*⟩

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.projl s'*, (*OP Sum-Type.sum.projl* ::: ⟨*Ra*,*Rb*⟩*sum-rel* → *Ra*)$*s*)∈*Ra*
⟨*proof*⟩

lemma *autoref-sum-Projr*[*autoref-rules*]:
⟦*SIDE-PRECOND* (*is-Inr s*); (*s'*,*s*)∈⟨*Ra*,*Rb*⟩*sum-rel*⟧
⟹ (*Sum-Type.sum.projr s'*, (*OP Sum-Type.sum.projr* ::: ⟨*Ra*,*Rb*⟩*sum-rel* → *Rb*)$*s*)∈*Rb*
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *autoref-append*[*autoref-rules*]:
(*append*, *append*)∈⟨*R*⟩*list-rel* → ⟨*R*⟩*list-rel* → ⟨*R*⟩*list-rel*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *autoref-rec-list*[*autoref-rules*]: (*rec-list*,*rec-list*)
∈ *Ra* → (*Rb* → ⟨*Rb*⟩*list-rel* → *Ra* → *Ra*) → ⟨*Rb*⟩*list-rel* → *Ra*
⟨*proof*⟩

lemma *refine-map*[*autoref-rules*]:
(*map*,*map*)∈(*R1*→*R2*) → ⟨*R1*⟩*list-rel* → ⟨*R2*⟩*list-rel*
⟨*proof*⟩

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*

⟨*proof*⟩

schematic-goal *autoref-take*[*autoref-rules*]: (*take*,*take*)∈(*?R*::(-×-) *set*)
⟨*proof*⟩
schematic-goal *autoref-drop*[*autoref-rules*]: (*drop*,*drop*)∈(*?R*::(-×-) *set*)
⟨*proof*⟩
schematic-goal *autoref-length*[*autoref-rules*]:
(*length*,*length*)∈(*?R*::(-×-) *set*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *list-eq-expand*[*autoref-struct-expand*]: (=) = (*list-eq* (=))
⟨*proof*⟩

lemma *autoref-list-eq*[*autoref-rules* (**overloaded**)]:
GEN-OP eq (=) (*R*→*R*→*Id*) ⟹ (*list-eq eq*, (=))
∈ ⟨*R*⟩*list-rel* → ⟨*R*⟩*list-rel* → *Id*
⟨*proof*⟩

lemma *autoref-hd*[*autoref-rules*]:
⟦ *SIDE-PRECOND* (*l'*≠[]); (*l*,*l'*) ∈ ⟨*R*⟩*list-rel* ⟧ ⟹
(*hd l*,(*OP hd* ::: ⟨*R*⟩*list-rel* → *R*)$*l'*) ∈ *R*
⟨*proof*⟩

lemma *autoref-tl*[*autoref-rules*]:
(*tl*,*tl*) ∈ ⟨*R*⟩*list-rel* → ⟨*R*⟩*list-rel*
⟨*proof*⟩

definition [*simp*]: *is-Nil a* ≡ *case a of* [] ⇒ *True* | - ⇒ *False*

lemma *is-Nil-pat*[*autoref-op-pat*]:
a=[] ≡ (*OP is-Nil* :::$_i$ ⟨*I*⟩$_i$*i-list* →$_i$ *i-bool*)$*a*

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

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

lemma *conv-to-is-Nil*:
$l{=}[] \longleftrightarrow is\text{-}Nil\ l$
$[]{=}l \longleftrightarrow is\text{-}Nil\ l$
$\langle proof\rangle$

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

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

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

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

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

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

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

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

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

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

lemmas *PREFER-sv-D* = *PREFER-D*[*of single-valued*]
lemma *PREFER-id-D*: *PREFER-id* $R \Longrightarrow R{=}Id\ \langle proof\rangle$

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* $P \Longrightarrow P\ \langle proof\rangle$

lemma *GEN-OP-D*: *GEN-OP* $c\ a\ R \Longrightarrow (c,a)\in R$
$\langle proof\rangle$

lemma *MINOR-PRIO-TAG-I*: $P \Longrightarrow$ (*MINOR-PRIO-TAG* $p \Longrightarrow P$) $\langle proof\rangle$
lemma *MAJOR-PRIO-TAG-I*: $P \Longrightarrow$ (*MAJOR-PRIO-TAG* $p \Longrightarrow P$) $\langle proof\rangle$
lemma *PRIO-TAG-I*: $P \Longrightarrow$ (*PRIO-TAG* $ma\ mi \Longrightarrow P$) $\langle proof\rangle$

end