Automatic Data Refinement

Peter Lammich

December 12, 2023

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 **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*
:: $(('c1 \times 'a1)\ set \Rightarrow \text{-}) \Rightarrow ('c1 \times 'a1)\ set \Rightarrow$ -

where *relAPP f x* ≡ *f x*

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

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'*)∈*Id* → *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*′)∈*R1* ∧ (*b*,*b*′)∈*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*]:
(⟨*R1*,*R2*⟩*prod-rel*) ≡ { ((*a*,*b*),(*a*′,*b*′)) . (*a*,*a*′)∈*R1* ∧ (*b*,*b*′)∈*R2* }
⟨*proof*⟩

lemma *prod-relI*: ⟦(*a*,*a*′)∈*R1*; (*b*,*b*′)∈*R2*⟧ ⟹ ((*a*,*b*),(*a*′,*b*′))∈⟨*R1*,*R2*⟩*prod-rel*
⟨*proof*⟩
lemma *prod-relE*:
assumes (*p*,*p*′)∈⟨*R1*,*R2*⟩*prod-rel*
obtains *a b a*′ *b*′ **where** *p*=(*a*,*b*) **and** *p*′=(*a*′,*b*′)
and (*a*,*a*′)∈*R1* **and** (*b*,*b*′)∈*R2*
⟨*proof*⟩

lemma *prod-rel-simp*[*simp*]:
((*a*,*b*),(*a*′,*b*′))∈⟨*R1*,*R2*⟩*prod-rel* ⟷ (*a*,*a*′)∈*R1* ∧ (*b*,*b*′)∈*R2*
⟨*proof*⟩

lemma *in-Domain-prod-rel-iff*[*iff*]: (*a*,*b*)∈*Domain* (*A*×$_r$*B*) ⟷ *a*∈*Domain A* ∧ *b*∈*Domain B*
⟨*proof*⟩

lemma *prod-rel-comp*: (*A* ×$_r$ *B*) *O* (*C* ×$_r$ *D*) = (*A O C*) ×$_r$ (*B O D*)
⟨*proof*⟩

Option

definition *option-rel* **where**
option-rel-def-internal:
option-rel R ≡ { (*Some a*,*Some a*′) | *a a*′. (*a*,*a*′)∈*R* } ∪ {(*None*,*None*)}

lemma *option-rel-def*[*refine-rel-defs*]:
⟨*R*⟩*option-rel* ≡ { (*Some a*,*Some a*′) | *a a*′. (*a*,*a*′)∈*R* } ∪ {(*None*,*None*)}

⟨*proof*⟩

lemma *option-relI*:
(*None*,*None*)∈⟨*R*⟩ *option-rel*
⟦ (*a*,*a′*)∈*R* ⟧ ⟹ (*Some a*, *Some a′*)∈⟨*R*⟩*option-rel*
⟨*proof*⟩

lemma *option-relE*:
assumes (*x*,*x′*)∈⟨*R*⟩*option-rel*
obtains *x*=*None* **and** *x′*=*None*
| *a a′* **where** *x*=*Some a* **and** *x′*=*Some a′* **and** (*a*,*a′*)∈*R*
⟨*proof*⟩

lemma *option-rel-simp*[*simp*]:
(*None*,*a*)∈⟨*R*⟩*option-rel* ⟷ *a*=*None*
(*c*,*None*)∈⟨*R*⟩*option-rel* ⟷ *c*=*None*
(*Some x*,*Some y*)∈⟨*R*⟩*option-rel* ⟷ (*x*,*y*)∈*R*
⟨*proof*⟩

Sum

definition *sum-rel* **where** *sum-rel-def-internal*:
sum-rel Rl Rr
≡ { (*Inl a*, *Inl a′*) | *a a′*. (*a*,*a′*)∈*Rl* } ∪
{ (*Inr a*, *Inr a′*) | *a a′*. (*a*,*a′*)∈*Rr* }

lemma *sum-rel-def*[*refine-rel-defs*]:
⟨*Rl*,*Rr*⟩*sum-rel* ≡
{ (*Inl a*, *Inl a′*) | *a a′*. (*a*,*a′*)∈*Rl* } ∪
{ (*Inr a*, *Inr a′*) | *a a′*. (*a*,*a′*)∈*Rr* }
⟨*proof*⟩

lemma *sum-rel-simp*[*simp*]:
⋀*a a′*. (*Inl a*, *Inl a′*) ∈ ⟨*Rl*,*Rr*⟩*sum-rel* ⟷ (*a*,*a′*)∈*Rl*
⋀*a a′*. (*Inr a*, *Inr a′*) ∈ ⟨*Rl*,*Rr*⟩*sum-rel* ⟷ (*a*,*a′*)∈*Rr*
⋀*a a′*. (*Inl a*, *Inr a′*) ∉ ⟨*Rl*,*Rr*⟩*sum-rel*
⋀*a a′*. (*Inr a*, *Inl a′*) ∉ ⟨*Rl*,*Rr*⟩*sum-rel*
⟨*proof*⟩

lemma *sum-relI*:
(*l*,*l′*)∈*Rl* ⟹ (*Inl l*, *Inl l′*) ∈ ⟨*Rl*,*Rr*⟩*sum-rel*
(*r*,*r′*)∈*Rr* ⟹ (*Inr r*, *Inr r′*) ∈ ⟨*Rl*,*Rr*⟩*sum-rel*
⟨*proof*⟩

lemma *sum-relE*:
assumes (*x*,*x′*)∈⟨*Rl*,*Rr*⟩*sum-rel*
obtains
l l′ **where** *x*=*Inl l* **and** *x′*=*Inl l′* **and** (*l*,*l′*)∈*Rl*
| *r r′* **where** *x*=*Inr r* **and** *x′*=*Inr r′* **and** (*r*,*r′*)∈*Rr*

⟨*proof*⟩

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* $(\lambda x\ x'.\ (x,x')\in R)\ l\ l'\}$
⟨*proof*⟩

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*; $P\ l\ l'\ ]\!]$
$\Longrightarrow P\ (x\#l)\ (x'\#l')$
shows $P\ l\ l'$
⟨*proof*⟩

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

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

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*
⟨*proof*⟩

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

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

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*
⟨*proof*⟩

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* $\Longrightarrow$ *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 rela-

tions. This is probably the reason why the old definition worked for most applications.

lemma *old-set-rel-sv-eq*: *single-valued R* $\Longrightarrow$ $\langle R\rangle$*old-set-rel* = $\langle R\rangle$*set-rel*
⟨*proof*⟩

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

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

lemma *set-relD1*: (*s*,*s′*)∈$\langle R\rangle$*set-rel* $\Longrightarrow$ *x*∈*s* $\Longrightarrow$ $\exists$ *x′*∈*s′*. (*x*,*x′*)∈*R*
⟨*proof*⟩

lemma *set-relD2*: (*s*,*s′*)∈$\langle R\rangle$*set-rel* $\Longrightarrow$ *x′*∈*s′* $\Longrightarrow$ $\exists$ *x*∈*s*. (*x*,*x′*)∈*R*
⟨*proof*⟩

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

lemma *set-relE2*[*consumes 2*]:
assumes (*s*,*s′*)∈$\langle R\rangle$*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*:
$\bigwedge$*R*. *single-valued R* $\Longrightarrow$ *single-valued R*
$\bigwedge$*R*. *R*=*Id* $\Longrightarrow$ *R*=*Id*
$\bigwedge$*R*. *R*=*Id* $\Longrightarrow$ *Id*=*R*
$\bigwedge$*R*. *Range R* = *UNIV* $\Longrightarrow$ *Range R* = *UNIV*
$\bigwedge$*R*. *Range R* = *UNIV* $\Longrightarrow$ *UNIV* = *Range R*
$\bigwedge$*R R′*. *R*⊆*R′* $\Longrightarrow$ *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*→*R2*) *O* (*R3*→*R4*) ⊆ ((*R1 O R3*) → (*R2 O R4*))
⟨*proof*⟩

lemma *fun-rel-mono*[*relator-props*]: ⟦ *R1*⊆*R2*; *R3*⊆*R4* ⟧ ⟹ *R2*→*R3* ⊆ *R1*→*R4*
⟨*proof*⟩

lemma *prod-rel-sv*[*relator-props*]:
⟦*single-valued R1*; *single-valued R2*⟧ ⟹ *single-valued* (⟨*R1*,*R2*⟩*prod-rel*)
⟨*proof*⟩

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

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

lemma *prod-rel-mono*[*relator-props*]:
⟦ *R2*⊆*R1*; *R3*⊆*R4* ⟧ ⟹ ⟨*R2*,*R3*⟩*prod-rel* ⊆ ⟨*R1*,*R4*⟩*prod-rel*
⟨*proof*⟩

lemma *prod-rel-range*[*relator-props*]: ⟦*Range Ra*=*UNIV*; *Range Rb*=*UNIV*⟧
⟹ *Range* (⟨*Ra*,*Rb*⟩*prod-rel*) = *UNIV*
⟨*proof*⟩

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

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

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

lemma *option-rel-mono*[*relator-props*]: *R*⊆*R'* ⟹ ⟨*R*⟩*option-rel* ⊆ ⟨*R'*⟩*option-rel*
⟨*proof*⟩

lemma *option-rel-range*: *Range R* = *UNIV* ⟹ *Range* (⟨*R*⟩*option-rel*) = *UNIV*
⟨*proof*⟩

lemma *option-rel-inter*[*simp*]: ⟨*R1* ∩ *R2*⟩*option-rel* = ⟨*R1*⟩*option-rel* ∩ ⟨*R2*⟩*option-rel*
⟨*proof*⟩

lemma *option-rel-constraint*[*simp*]:
(*x*,*x*)∈⟨*UNIV*×*C*⟩*option-rel* ⟷ (∀ *v*. *x*=*Some v* ⟶ *v*∈*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 ⟹ *single-valued* (⟨*R*⟩*set-rel*)
⟨*proof*⟩

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

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

lemma *set-rel-csv*[*relator-props*]:
$[\![\ single\text{-}valued\ (R^{-1})\ ]\!]$
$\Longrightarrow single\text{-}valued\ ((\langle R\rangle set\text{-}rel)^{-1})$
⟨*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**
$build\text{-}rel\ \alpha\ I \equiv \{(c,a)\ .\ a{=}\alpha\ c \wedge I\ c\}$
abbreviation *br*≡*build-rel*
lemmas *br-def*[*refine-rel-defs*] = *build-rel-def*

lemma *in-br-conv*: $(c,a){\in}br\ \alpha\ I \longleftrightarrow a{=}\alpha\ c \wedge I\ c$
⟨*proof*⟩

lemma *brI*[*intro?*]: $[\![\ a{=}\alpha\ c;\ I\ c\ ]\!] \Longrightarrow (c,a){\in}br\ \alpha\ I$
⟨*proof*⟩

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

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

lemma *br-sv*[*simp*, *intro*!,*relator-props*]: $single\text{-}valued\ (br\ \alpha\ I)$
⟨*proof*⟩

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

$\{(c,a)\ .\ a{=}\alpha\ c \wedge I\ c\}\ O\ R = \{\ (c,a)\ .\ I\ c \wedge (\alpha\ c,a){\in}R\ \}$
⟨*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 $\Longrightarrow$ *single-valued* $\{(c,\ a).\ (c,\ a) \in R \wedge I\ c\}$
⟨*proof*⟩

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

1.1.6 Miscellanneous

lemma *rel-cong*: $(f,g){\in}Id \Longrightarrow (x,y){\in}Id \Longrightarrow (f\ x,\ g\ y){\in}Id$ ⟨*proof*⟩
lemma *rel-fun-cong*: $(f,g){\in}Id \Longrightarrow (f\ x,\ g\ x){\in}Id$ ⟨*proof*⟩
lemma *rel-arg-cong*: $(x,y){\in}Id \Longrightarrow (f\ x,\ f\ y){\in}Id$ ⟨*proof*⟩

1.1.7 Conversion between Predicate and Set Based Relators

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

definition *rel2p R x y* $\equiv (x,y){\in}R$
definition *p2rel P* $\equiv \{(x,y).\ P\ x\ y\}$

lemma *rel2pD*: $[\![rel2p\ R\ a\ b]\!] \Longrightarrow (a,b){\in}R$ ⟨*proof*⟩
lemma *p2relD*: $[\![(a,b) \in p2rel\ R]\!] \Longrightarrow R\ a\ b$ ⟨*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 ($A{\to}B$) = *rel-fun* (*rel2p A*) (*rel2p B*)
rel2p ($A{\times_r}B$) = *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*: ⟨*A O B*, *C O D*⟩*prod-rel* = ⟨*A*,*C*⟩*prod-rel O* ⟨*B*,*D*⟩*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 $(l,\ map\ \alpha\ l) \in \langle br\ \alpha\ I\rangle$*list-rel* $\longleftrightarrow$ $(\forall x{\in}set\ l.\ I\ x)$
⟨*proof*⟩

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

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

lemma *finite-set-rel-transfer*: $[\![(s,s'){\in}\langle R\rangle$*set-rel*; *single-valued R*; *finite s*$]\!] \Longrightarrow$ *finite* s'
⟨*proof*⟩

lemma *finite-set-rel-transfer-back*: $[\![(s,s'){\in}\langle R\rangle$*set-rel*; *single-valued* (R^{-1}); *finite* $s'$$]\!]$ $\Longrightarrow$ *finite s*
⟨*proof*⟩

end

1.2 Basic Parametricity Reasoning

theory *Param-Tool*
imports *Relators*
begin

1.2.1 Auxiliary Lemmas

lemma *tag-both*: $[\![\ (Let\ x\ f,Let\ x'\ f'){\in}R\ ]\!] \Longrightarrow (f\ x,f'\ x'){\in}R$ ⟨*proof*⟩
lemma *tag-rhs*: $[\![\ (c,Let\ x\ f){\in}R\ ]\!] \Longrightarrow (c,f\ x){\in}R$ ⟨*proof*⟩
lemma *tag-lhs*: $[\![\ (Let\ x\ f,a){\in}R\ ]\!] \Longrightarrow (f\ x,a){\in}R$ ⟨*proof*⟩

lemma *tagged-fun-relD-both*:
$[\![\ (f,f'){\in}A{\to}B;\ (x,x'){\in}A\ ]\!] \Longrightarrow (Let\ x\ f,Let\ x'\ f'){\in}B$
and *tagged-fun-relD-rhs*: $[\![\ (f,f'){\in}A{\to}B;\ (x,x'){\in}A\ ]\!] \Longrightarrow (f\ x,Let\ x'\ f'){\in}B$
and *tagged-fun-relD-lhs*: $[\![\ (f,f'){\in}A{\to}B;\ (x,x'){\in}A\ ]\!] \Longrightarrow (Let\ x\ f,f'\ x'){\in}B$
and *tagged-fun-relD-none*: $[\![\ (f,f'){\in}A{\to}B;\ (x,x'){\in}A\ ]\!] \Longrightarrow (f\ x,f'\ x'){\in}B$
⟨*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*]:
({},{})∈⟨*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-valued R*; *single-valued* (R^{-1})⟧ ⟹ ((⊆), (⊆)) ∈ ⟨*R*⟩*set-rel* → ⟨*R*⟩*set-rel*
→ *bool-rel*
⟨*proof*⟩

lemma *param-subset*[*param*]:
⟦*single-valued* R; *single-valued* (R^{-1})⟧ ⟹ $((\subset), (\subset)) \in \langle R\rangle$*set-rel* → $\langle R\rangle$*set-rel* → *bool-rel*
⟨*proof*⟩

lemma *param-Ball*[*param*]: (*Ball*,*Ball*)∈⟨*Ra*⟩*set-rel*→(*Ra*→*Id*)→*Id*
⟨*proof*⟩

lemma *param-Bex*[*param*]: (*Bex*,*Bex*)∈⟨*Ra*⟩*set-rel*→(*Ra*→*Id*)→*Id*
⟨*proof*⟩

lemma *param-set*[*param*]:
single-valued Ra ⟹ (*set*,*set*)∈⟨*Ra*⟩*list-rel* → ⟨*Ra*⟩*set-rel*
⟨*proof*⟩

lemma *param-Collect*[*param*]:
⟦*Domain* A = *UNIV*; *Range* A = *UNIV*⟧ ⟹ (*Collect*,*Collect*)∈(A→*bool-rel*) → ⟨A⟩*set-rel*
⟨*proof*⟩

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

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

1.3.2 Standard HOL Constructs

lemma *param-if*[*param*]:
assumes (c,c')∈*Id*
assumes ⟦c;c'⟧ ⟹ (t,t')∈R
assumes ⟦¬c;¬c'⟧ ⟹ (e,e')∈R
shows (*If* c t e, *If* c' t' e')∈R
⟨*proof*⟩

lemma *param-Let*[*param*]:
(*Let*,*Let*)∈*Ra* → (*Ra*→*Rr*) → *Rr*
⟨*proof*⟩

1.3.3 Functions

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

lemma *param-fun-comp*[*param*]: ((*o*), (*o*)) ∈ (*Ra*→*Rb*) → (*Rc*→*Ra*) → *Rc*→*Rb*
⟨*proof*⟩

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

1.3.4 Boolean

lemma *rec-bool-is-case*: $old.rec\text{-}bool = case\text{-}bool$
⟨*proof*⟩

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

lemma *param-and-cong1*: $[\![(a,a') \in bool\text{-}rel; [\![a; a']\!] \Longrightarrow (b,b') \in bool\text{-}rel \,]\!] \Longrightarrow (a \wedge b, a' \wedge b') \in bool\text{-}rel$
⟨*proof*⟩
lemma *param-and-cong2*: $[\![(a,a') \in bool\text{-}rel; [\![a; a']\!] \Longrightarrow (b,b') \in bool\text{-}rel \,]\!] \Longrightarrow (b \wedge a, b' \wedge a') \in bool\text{-}rel$
⟨*proof*⟩

1.3.5 Nat

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

lemma *param-case-nat*[*param*]:
$(case\text{-}nat, case\text{-}nat) \in Ra \to (Id \to Ra) \to Id \to Ra$
⟨*proof*⟩

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

$(rec\text{-}nat,rec\text{-}nat) \in R \to (Id \to R \to R) \to Id \to R$
⟨*proof*⟩

1.3.6 Int

lemma *param-int*[*param*]:
$(0,\ 0::int) \in Id$
$(1,\ 1::int) \in Id$
$(numeral\ n::int, numeral\ n::int) \in Id$
$((<),\ (<)::int \Rightarrow \text{-}) \in Id \to Id \to Id$
$((\leq),\ (\leq)::int \Rightarrow \text{-}) \in Id \to Id \to Id$
$((=),\ (=)::int \Rightarrow \text{-}) \in Id \to Id \to Id$
$((+)::int \Rightarrow \text{-}, (+)) \in Id \to Id \to Id$
$((-)::int \Rightarrow \text{-}, (-)) \in Id \to Id \to Id$
$((*)::int \Rightarrow \text{-}, (*)) \in Id \to Id \to Id$
$((div)::int \Rightarrow \text{-}, (div)) \in Id \to Id \to Id$
$((mod)::int \Rightarrow \text{-}, (mod)) \in Id \to Id \to Id$
⟨*proof*⟩

1.3.7 Product

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

lemma *rec-prod-is-case*: $old.rec\text{-}prod = case\text{-}prod$
⟨*proof*⟩

lemma *param-prod*[*param*]:
$(Pair,Pair) \in Ra \to Rb \to \langle Ra,Rb\rangle prod\text{-}rel$
$(case\text{-}prod,case\text{-}prod) \in (Ra \to Rb \to Rr) \to \langle Ra,Rb\rangle prod\text{-}rel \to Rr$
$(old.rec\text{-}prod,old.rec\text{-}prod) \in (Ra \to Rb \to Rr) \to \langle Ra,Rb\rangle prod\text{-}rel \to Rr$
$(fst,fst) \in \langle Ra,Rb\rangle prod\text{-}rel \to Ra$
$(snd,snd) \in \langle Ra,Rb\rangle prod\text{-}rel \to Rb$
⟨*proof*⟩

lemma *param-case-prod′*:
$[\![\ (p,p') \in \langle Ra,Rb\rangle prod\text{-}rel;$
$\quad \bigwedge a\ b\ a'\ b'.\ [\![\ p=(a,b);\ p'=(a',b');\ (a,a') \in Ra;\ (b,b') \in Rb\ ]\!]$
$\qquad \Longrightarrow (f\ a\ b,\ f'\ a'\ b') \in R$
$\quad]\!] \Longrightarrow (case\text{-}prod\ f\ p,\ case\text{-}prod\ f'\ p') \in R$
⟨*proof*⟩

lemma *param-case-prod″*:
$[\![$
$\quad \bigwedge a\ b\ a'\ b'.\ [\![p=(a,b);\ p'=(a',b')]\!] \Longrightarrow (f\ a\ b, f'\ a'\ b') \in R$
$]\!] \Longrightarrow (case\text{-}prod\ f\ p,\ case\text{-}prod\ f'\ p') \in R$
⟨*proof*⟩

lemma *param-map-prod*[*param*]:
$(map\text{-}prod,\ map\text{-}prod)$

$\in (Ra \to Rb) \to (Rc \to Rd) \to \langle Ra,Rc\rangle prod\text{-}rel \to \langle Rb,Rd\rangle prod\text{-}rel$
⟨*proof*⟩

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

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

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

lemma *param-uncurry*[*param*]: $(uncurry,uncurry) \in (A \to B \to C) \to A \times_r B \to C$
⟨*proof*⟩

lemma *param-prod-swap*[*param*]: $(prod.swap,\ prod.swap) \in A \times_r B \to B \times_r A$ ⟨*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 $(\lambda a.\ \forall x\ xa.\ (x,\ xa) \in Rb \longrightarrow (a\ x,\ fixp\text{-}fun\ F'\ xa) \in Ra)$
assumes *bot*: $\bigwedge x\ xa.\ (x,\ xa) \in Rb \Longrightarrow (lub\ \{\},\ fixp\text{-}fun\ F'\ xa) \in Ra$
assumes *F*: $(F,F') \in (Rb \to Ra) \to Rb \to Ra$
assumes *A*: $(x,x') \in Rb$
shows $(fixp\text{-}fun\ F\ x,\ fixp\text{-}fun\ F'\ x') \in Ra$
⟨*proof*⟩
end

1.3.8 Option

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

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

lemma *param-case-option'*:
$[\![\ (x,x') \in \langle Rv\rangle option\text{-}rel;$
$[\![x=None;\ x'=None\ ]\!] \Longrightarrow (fn,fn') \in R;$

$\bigwedge v\ v'.\ [\![\ x{=}Some\ v;\ x'{=}Some\ v';\ (v,v')\in Rv\ ]\!] \Longrightarrow (fs\ v,\ fs'\ v')\in R$
$]\!] \Longrightarrow (case\text{-}option\ fn\ fs\ x,\ case\text{-}option\ fn'\ fs'\ x')\in R$
⟨*proof*⟩

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

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

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

1.3.9 Sum

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

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

lemma *param-case-sum'*:
$[\![\ (s,s')\in\langle Rl,Rr\rangle sum\text{-}rel;$
$\bigwedge l\ l'.\ [\![\ s{=}Inl\ l;\ s'{=}Inl\ l';\ (l,l')\in Rl\ ]\!] \Longrightarrow (fl\ l,\ fl'\ l')\in R;$
$\bigwedge r\ r'.\ [\![\ s{=}Inr\ r;\ s'{=}Inr\ r';\ (r,r')\in Rr\ ]\!] \Longrightarrow (fr\ r,\ fr'\ r')\in R$
$]\!] \Longrightarrow (case\text{-}sum\ fl\ fr\ s,\ case\text{-}sum\ fl'\ fr'\ s')\in R$
⟨*proof*⟩

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

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

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

lemma *sum-projr-param*[*param*]:
$[\![is\text{-}Inr\ s;\ (s',s)\in\langle Ra,Rb\rangle sum\text{-}rel]\!]$
$\Longrightarrow (Sum\text{-}Type.sum.projr\ s',Sum\text{-}Type.sum.projr\ s)\in Rb$

⟨*proof*⟩

1.3.10 List

lemma *list-rel-append1*: $(as \mathbin{@} bs,\ l) \in \langle R\rangle$*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*)
⟨*proof*⟩

lemma *list-rel-append2*: $(l,as \mathbin{@} bs) \in \langle R\rangle$*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*)
⟨*proof*⟩

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

lemma *param-list1*[*param*]:
$(Nil,Nil)\in\langle R\rangle$*list-rel*
$(Cons,Cons)\in R \to \langle R\rangle$*list-rel* $\to\ \langle R\rangle$*list-rel*
$(case\text{-}list,case\text{-}list)\in Rr\to(R\to\langle R\rangle list\text{-}rel\to Rr)\to\langle R\rangle list\text{-}rel\to Rr$
⟨*proof*⟩

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

lemma *param-case-list′*:
$[\![\ (l,l')\in\langle Rb\rangle$*list-rel*;
$\quad [\![l=[];\ l'=[]]\!] \Longrightarrow (n,n')\in Ra$;
$\quad \bigwedge x\ xs\ x'\ xs'.\ [\![\ l=x\#xs;\ l'=x'\#xs';\ (x,x')\in Rb;\ (xs,xs')\in\langle Rb\rangle$*list-rel* $]\!]$
$\quad \Longrightarrow (c\ x\ xs,\ c'\ x'\ xs')\in Ra$
$]\!] \Longrightarrow (case\text{-}list\ n\ c\ l,\ case\text{-}list\ n'\ c'\ l') \in Ra$
⟨*proof*⟩

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

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

lemma *param-list-all*[*param*]: $(list\text{-}all,list\text{-}all) \in (A\to bool\text{-}rel) \to \langle A\rangle list\text{-}rel \to bool\text{-}rel$
⟨*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*)
∈ ⟨*Re*⟩*list-rel* → (*Rs*→*Id*) → (*Re*→*Rs*→*Rs*) → *Rs* → *Rs*
⟨*proof*⟩

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 $(l!i,l'!i')\in R$
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *param-takeWhile*[*param*]:
$(takeWhile,\ takeWhile)\in(a\to bool\text{-}rel)\to\langle a\rangle list\text{-}rel\to\langle a\rangle list\text{-}rel$
⟨*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*]: $CAST ::_i I \rightarrow_i I$ ⟨*proof*⟩

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* $\equiv x = y$
lemma *STRUCT-EQ-tagI*: $x{=}y \Longrightarrow$ *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 $::_i I \to_i I$
⟨*proof*⟩

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

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 $(1,1) \in Id$
⟨*proof*⟩

2.2.4 Functional Combinators

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

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

lemma [*autoref-itype*]: *If* $::_i$ *i-bool* $\to_i I \to_i I \to_i I$ ⟨*proof*⟩
lemma *autoref-If*[*autoref-rules*]: $(If,If) \in Id \to R \to R \to R$ ⟨*proof*⟩
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 $\bigwedge y\ y'$. *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*

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

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

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

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

2.2.8 Product

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

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

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-Nil* :::$_i$ $\langle I\rangle_i$*i-list* $\rightarrow_i$ *i-bool*)$a
⟨*proof*⟩

lemma *autoref-is-Nil*[*param*,*autoref-rules*]:
(*is-Nil*,*is-Nil*)∈⟨*R*⟩*list-rel* → *bool-rel*
⟨*proof*⟩

lemma *conv-to-is-Nil*:
l=[] ⟷ *is-Nil l*
[]=*l* ⟷ *is-Nil l*
⟨*proof*⟩

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

definition [*simp*]: *op-list-singleton x* ≡ [*x*]
lemma *op-list-singleton-pat*[*autoref-op-pat*]:
[*x*] ≡ (*OP op-list-singleton* :::$_i$ *I* →$_i$ ⟨*I*⟩$_i$*i-list*)$*x* ⟨*proof*⟩
lemma *autoref-list-singleton*[*autoref-rules*]:
(λ*a*. [*a*],*op-list-singleton*) ∈ *R* → ⟨*R*⟩*list-rel*
⟨*proof*⟩

definition [*simp*]: *op-list-append-elem s x* ≡ *s*@[*x*]

lemma *pat-list-append-elem*[*autoref-op-pat*]:
s@[*x*] ≡ (*OP op-list-append-elem* :::$_i$ ⟨*I*⟩$_i$*i-list* →$_i$ *I* →$_i$ ⟨*I*⟩$_i$*i-list*)$*s*$*x*
⟨*proof*⟩

lemma *autoref-list-append-elem*[*autoref-rules*]:
(λ*s x*. *s*@[*x*], *op-list-append-elem*) ∈ ⟨*R*⟩*list-rel* → *R* → ⟨*R*⟩*list-rel*
⟨*proof*⟩

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

declare *param-all-interval-nat*[*autoref-rules*]
lemma [*autoref-op-pat*]:
(∀ *i*<*u*. *P i*) ≡ *OP List.all-interval-nat P 0 u*
(∀ *i*≤*u*. *P i*) ≡ *OP List.all-interval-nat P 0* (*Suc u*)
(∀ *i*<*u*. *l*≤*i* ⟶ *P i*) ≡ *OP List.all-interval-nat P l u*
(∀ *i*≤*u*. *l*≤*i* ⟶ *P i*) ≡ *OP List.all-interval-nat P l* (*Suc u*)
⟨*proof*⟩

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*)∈⟨*R*⟩*list-rel*
shows (*?f*::*?'c*, [] = *a*)∈*?R*
⟨*proof*⟩

schematic-goal
shows (*?f*::*?'c*, [*1*,*2*] = [*2*,*3*::*nat*])∈*?R*
⟨*proof*⟩

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* ⟹ *PREFER P x* ⟨*proof*⟩
lemma *PREFER-D*: *PREFER P x* ⟹ *P x* ⟨*proof*⟩

lemmas *PREFER-sv-D* = *PREFER-D*[*of single-valued*]
lemma *PREFER-id-D*: *PREFER-id R* ⟹ *R*=*Id* ⟨*proof*⟩

abbreviation *PREFER-RUNIV* ≡ *PREFER* (λ*R*. *Range R* = *UNIV*)
lemmas *PREFER-RUNIV-D* = *PREFER-D*[*of* (λ*R*. *Range R* = *UNIV*)]

lemma *SIDE-GEN-ALGO-D*: *SIDE-GEN-ALGO P* ⟹ *P* ⟨*proof*⟩

lemma *GEN-OP-D*: *GEN-OP c a R* ⟹ (*c*,*a*)∈*R*
⟨*proof*⟩

lemma *MINOR-PRIO-TAG-I*: *P* ⟹ (*MINOR-PRIO-TAG p* ⟹ *P*) ⟨*proof*⟩
lemma *MAJOR-PRIO-TAG-I*: *P* ⟹ (*MAJOR-PRIO-TAG p* ⟹ *P*) ⟨*proof*⟩
lemma *PRIO-TAG-I*: *P* ⟹ (*PRIO-TAG ma mi* ⟹ *P*) ⟨*proof*⟩

end