Automatic Data Refinement

Peter Lammich

October 2, 2013

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 users 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

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

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

1.1.2 Basic HOL Relators

Function

definition *fun-rel* **where**
fun-rel-def-internal: *fun-rel A B* ≡ { (*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* }
by (*simp add*: *relAPP-def fun-rel-def-internal*)

lemma *fun-relI*[*intro*!]: ⟦⋀*a a'*. (*a*,*a'*)∈*A* ⟹ (*f a*,*f' a'*)∈*B*⟧ ⟹ (*f*,*f'*)∈*A*→*B*
by (*auto simp*: *fun-rel-def*)

lemma *fun-relD*:
shows ((*f*,*f'*)∈(*A*→*B*)) ⟹
(⋀*x x'*. ⟦ (*x*,*x'*)∈*A* ⟧ ⟹ (*f x*, *f' x'*)∈*B*)
apply *rule*
by (*auto simp*: *fun-rel-def*)

lemma *fun-relD1*:
assumes (*f*,*f'*)∈*Ra*→*Rr*
assumes *f x* = *r*
shows ∀*x'*. (*x*,*x'*)∈*Ra* ⟶ (*r*,*f' x'*)∈*Rr*
using *assms* **by** (*auto simp*: *fun-rel-def*)

lemma *fun-relD2*:
assumes (*f*,*f'*)∈*Ra*→*Rr*

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

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

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

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** $\times_r$ *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*$) \equiv \{ ((a,b),(a',b'))\ .\ (a,a')\in R1 \wedge (b,b')\in R2 \}$
by (*simp add*: *prod-rel-def-internal relAPP-def*)

lemma *prod-relI*: $[\![(a,a')\in R1;\ (b,b')\in R2]\!] \Longrightarrow ((a,b),(a',b'))\in\langle R1,R2\rangle$*prod-rel*
by (*auto simp*: *prod-rel-def*)
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** $(b,b')\in R2$
using *assms*
by (*auto simp*: *prod-rel-def*)

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

Option

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

lemma *option-rel-def*[*refine-rel-defs*]:
$\langle R \rangle option\text{-}rel \equiv \{\ (Some\ a, Some\ a') \mid a\ a'.\ (a,a') \in R\ \} \cup \{(None, None)\}$
by (*simp add*: *option-rel-def-internal relAPP-def*)

lemma *option-relI*:
$(None, None) \in \langle R \rangle\ option\text{-}rel$
$[\![\ (a,a') \in R\ ]\!] \Longrightarrow (Some\ a,\ Some\ a') \in \langle R \rangle option\text{-}rel$
by (*auto simp*: *option-rel-def*)

lemma *option-relE*:
assumes $(x,x') \in \langle R \rangle option\text{-}rel$
obtains $x = None$ **and** $x' = None$
| $a\ a'$ **where** $x = Some\ a$ **and** $x' = Some\ a'$ **and** $(a,a') \in R$
using *assms* **by** (*auto simp*: *option-rel-def*)

lemma *option-rel-simp*[*simp*]:
$(None, a) \in \langle R \rangle option\text{-}rel \longleftrightarrow a = None$
$(c, None) \in \langle R \rangle option\text{-}rel \longleftrightarrow c = None$
$(Some\ x, Some\ y) \in \langle R \rangle option\text{-}rel \longleftrightarrow (x,y) \in R$
by (*auto intro*: *option-relI elim*: *option-relE*)

Sum

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

lemma *sum-rel-def*[*refine-rel-defs*]:
$\langle Rl, Rr \rangle sum\text{-}rel \equiv$
$\{\ (Inl\ a,\ Inl\ a') \mid a\ a'.\ (a,a') \in Rl\ \} \cup$
$\{\ (Inr\ a,\ Inr\ a') \mid a\ a'.\ (a,a') \in Rr\ \}$
by (*simp add*: *sum-rel-def-internal relAPP-def*)

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

lemma *sum-relI*:
$(a,a') \in Rl \Longrightarrow (Inl\ a,\ Inl\ a') \in \langle Rl, Rr \rangle sum\text{-}rel$
$(a,a') \in Rr \Longrightarrow (Inr\ a,\ Inr\ a') \in \langle Rl, Rr \rangle sum\text{-}rel$

by *simp-all*

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** $(r,r')\in Rr$
using *assms* **by** (*auto simp*: *sum-rel-def*)

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

lemma *list-rel-induct*[*induct set*,*consumes 1*, *case-names Nil Cons*]:
assumes $(l,l')\in\langle R\rangle$ *list-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 $P\ l\ l'$
using *assms* **unfolding** *list-rel-def*
apply *simp*
by (*rule list-all2-induct*)

lemma *list-rel-eq-listrel*: *list-rel* = *listrel*
apply (*rule ext*)
proof *safe*
case *goal1* **thus** *?case*
unfolding *list-rel-def-internal*
apply *simp*
apply (*induct a b rule*: *list-all2-induct*)
apply (*auto intro*: *listrel.intros*)
done
next
case *goal2* **thus** *?case*
apply (*induct*)
apply (*auto simp*: *list-rel-def-internal*)
done
qed

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

lemma *list-rel-simp*[*simp*]:
([],*l'*)∈⟨*R*⟩*list-rel* ⟷ *l'*=[]
(*l*,[])∈⟨*R*⟩*list-rel* ⟷ *l*=[]
([],[])∈⟨*R*⟩*list-rel*
(*x*#*l*,*x'*#*l'*)∈⟨*R*⟩*list-rel* ⟷ (*x*,*x'*)∈*R* ∧ (*l*,*l'*)∈⟨*R*⟩*list-rel*
by (*auto simp*: *list-rel-def*)

lemma *list-relE1*:
assumes (*l*,[])∈⟨*R*⟩*list-rel* **obtains** *l*=[] **using** *assms* **by** *auto*

lemma *list-relE2*:
assumes ([],*l*)∈⟨*R*⟩*list-rel* **obtains** *l*=[] **using** *assms* **by** *auto*

lemma *list-relE3*:
assumes (*x*#*xs*,*l'*)∈⟨*R*⟩*list-rel* **obtains** *x' xs'* **where**
l'=*x'*#*xs'* **and** (*x*,*x'*)∈*R* **and** (*xs*,*xs'*)∈⟨*R*⟩*list-rel*
using *assms*
apply (*cases l'*)
apply *auto*
done

lemma *list-relE4*:
assumes (*l*,*x'*#*xs'*)∈⟨*R*⟩*list-rel* **obtains** *x xs* **where**
l=*x*#*xs* **and** (*x*,*x'*)∈*R* **and** (*xs*,*xs'*)∈⟨*R*⟩*list-rel*
using *assms*
apply (*cases l*)
apply *auto*
done

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

lemma *list-rel-imp-same-length*:
(*l*, *l'*) ∈ ⟨*R*⟩*list-rel* ⟹ *length l* = *length l'*
unfolding *list-rel-eq-listrel relAPP-def*
by (*rule listrel-eq-len*)

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 ≡ {(*S*,*S'*). *S'*=*R*``*S* ∧ *S*⊆*Domain R*}

lemma *set-rel-def*[*refine-rel-defs*]:
⟨*R*⟩*set-rel* ≡ {(*S*,*S'*). *S'*=*R*``*S* ∧ *S*⊆*Domain R*}
by (*simp add*: *set-rel-def-internal relAPP-def*)

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

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

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'*
by *auto*

ML ⟪
structure relator-props = *Named-Thms* (
val name = @{*binding relator-props*}
val description = *Additional relator properties*
)
⟫
setup *relator-props.setup*

declaration ⟪
Tagged-Solver.declare-solver
@{*thms relprop-triggers*}
@{*binding relator-props-solver*}
Additional relator peoperties solver
(*fn ctxt* => (*REPEAT-ALL-NEW* (*match-tac* (*relator-props.get ctxt*))))
⟫

declaration ⟪
Tagged-Solver.declare-solver
[]
@{*binding force-relator-props-solver*}
Additional relator properties solver (*instantiate schematics*)
(*fn ctxt* => (*REPEAT-ALL-NEW* (*resolve-tac* (*relator-props.get ctxt*))))
⟫

lemma *relprop-id-orient*[*relator-props*]:
R=*Id* ⟹ *Id*=*R*
Id = *Id*
by *auto*

lemma *relprop-UNIV-orient*[*relator-props*]:
R=*UNIV* ⟹ *UNIV*=*R*
UNIV = *UNIV*
by *auto*

ML-Level utilities

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

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

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

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

    val setup: theory -> theory
  end

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

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

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

    val list-relAPP = fold mk-relAPP

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

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

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

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

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

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

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

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

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

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

local
  fun decl-natrel-aux t context = let
    fun warn msg = let
      val tP =
```

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

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

val setup = I
  #> Attrib.setup
    @{binding natural-relator} natural-relator-attr Declare natural relator

end
⟫

setup Relators.setup
```

1.1.4 Setup

Natural Relators

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

ML-val ⟪
  Relators.mk-natural-relator
    @{*context*}
    [@{*term Ra*::('*c*×'*a*) *set*},@{*term* ⟨*Rb*⟩*option-rel*}]
    @{*type-name prod*}
  |> *the*
  |> *cterm-of* @{*theory*}
;
  Relators.mk-fun-rel @{*term* ⟨*Id*⟩*option-rel*} @{*term* ⟨*Id*⟩*list-rel*}
  |> *cterm-of* @{*theory*}
⟫

Additional Properties

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

lemma *eq-UNIV-iff*: *S*=*UNIV* ⟷ (∀ *x*. *x*∈*S*) **by** *auto*

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

  from *RAN* **obtain** *x* **where** *AR*: (*x*,*x'*)∈*Ra* **by** *auto*
  from *fun-relD*[*OF R1 AR*] **have** (*f x*,*g x'*) ∈ *Rv* **.**
  moreover from *fun-relD*[*OF R2 AR*] **have** (*f x*,*h x'*) ∈ *Rv* **.**
  ultimately show *g x'* = *h x'* **using** *SV* **by** (*auto dest*: *single-valuedD*)
qed

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

lemma *fun-rel-id*[*relator-props*]: ⟦*R1*=*Id*; *R2*=*Id*⟧ ⟹ *R1* → *R2* = *Id*
  by (*auto simp*: *fun-rel-def*)

lemma *fun-rel-id-simp*[*simp*]: *Id*→*Id* = *Id* **by** *tagged-solver*

lemma *fun-rel-comp-dist*[*relator-props*]:
  (*R1*→*R2*) *O* (*R3*→*R4*) ⊆ ((*R1 O R3*) → (*R2 O R4*))

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

lemma *fun-rel-mono*[*relator-props*]: ⟦ *R1*⊆*R2*; *R3*⊆*R4* ⟧ ⟹ *R2*→*R3* ⊆ *R1*→*R4*
by (*force simp*: *fun-rel-def*)

lemma *prod-rel-sv*[*relator-props*]:
⟦*single-valued R1*; *single-valued R2*⟧ ⟹ *single-valued* (⟨*R1*,*R2*⟩*prod-rel*)
by (*auto intro*: *single-valuedI dest*: *single-valuedD simp*: *prod-rel-def*)

lemma *prod-rel-id*[*relator-props*]: ⟦*R1*=*Id*; *R2*=*Id*⟧ ⟹ ⟨*R1*,*R2*⟩*prod-rel* = *Id*
by (*auto simp*: *prod-rel-def*)

lemma *prod-rel-id-simp*[*simp*]: ⟨*Id*,*Id*⟩*prod-rel* = *Id* **by** *tagged-solver*

lemma *prod-rel-mono*[*relator-props*]:
⟦ *R2*⊆*R1*; *R3*⊆*R4* ⟧ ⟹ ⟨*R2*,*R3*⟩*prod-rel* ⊆ ⟨*R1*,*R4*⟩*prod-rel*
by (*auto simp*: *prod-rel-def*)

lemma *prod-rel-range*[*relator-props*]: ⟦*Range Ra*=*UNIV*; *Range Rb*=*UNIV*⟧
⟹ *Range* (⟨*Ra*,*Rb*⟩*prod-rel*) = *UNIV*
apply (*auto simp*: *prod-rel-def*)
by (*metis Range-iff UNIV-I*)+

lemma *option-rel-sv*[*relator-props*]:
⟦*single-valued R*⟧ ⟹ *single-valued* (⟨*R*⟩*option-rel*)
by (*auto intro*: *single-valuedI dest*: *single-valuedD simp*: *option-rel-def*)

lemma *option-rel-id*[*relator-props*]:
R=*Id* ⟹ ⟨*R*⟩*option-rel* = *Id* **by** (*auto simp*: *option-rel-def*)

lemma *option-rel-id-simp*[*simp*]: ⟨*Id*⟩*option-rel* = *Id* **by** *tagged-solver*

lemma *option-rel-mono*[*relator-props*]: *R*⊆*R′* ⟹ ⟨*R*⟩*option-rel* ⊆ ⟨*R′*⟩*option-rel*
by (*auto simp*: *option-rel-def*)

lemma *option-rel-range*: *Range R* = *UNIV* ⟹ *Range* (⟨*R*⟩*option-rel*) = *UNIV*
apply (*auto simp*: *option-rel-def Range-iff*)
by (*metis Range-iff UNIV-I option.exhaust*)

lemma *sum-rel-sv*[*relator-props*]:
⟦*single-valued Rl*; *single-valued Rr*⟧ ⟹ *single-valued* (⟨*Rl*,*Rr*⟩*sum-rel*)
by (*auto intro*: *single-valuedI dest*: *single-valuedD simp*: *sum-rel-def*)

lemma *sum-rel-id*[*relator-props*]: ⟦*Rl*=*Id*; *Rr*=*Id*⟧ ⟹ ⟨*Rl*,*Rr*⟩*sum-rel* = *Id*
apply (*auto elim*: *sum-relE*)
apply (*case-tac b*)
apply *simp-all*
done

lemma *sum-rel-id-simp*[*simp*]: ⟨*Id*,*Id*⟩*sum-rel* = *Id* **by** *tagged-solver*

lemma *sum-rel-mono*[*relator-props*]:
⟦ *Rl*⊆*Rl'*; *Rr*⊆*Rr'* ⟧ ⟹ ⟨*Rl*,*Rr*⟩*sum-rel* ⊆ ⟨*Rl'*,*Rr'*⟩*sum-rel*
by (*auto simp*: *sum-rel-def*)

lemma *sum-rel-range*[*relator-props*]:
⟦ *Range Rl*=*UNIV*; *Range Rr*=*UNIV* ⟧ ⟹ *Range* (⟨*Rl*,*Rr*⟩*sum-rel*) = *UNIV*
apply (*auto simp*: *sum-rel-def Range-iff*)
by (*metis Range-iff UNIV-I sumE*)

lemma *list-rel-sv-iff*:
single-valued (⟨*R*⟩*list-rel*) ⟷ *single-valued R*
apply (*intro iffI*[*rotated*] *single-valuedI allI impI*)
apply (*clarsimp simp*: *list-rel-def*)
proof −
fix *x y z*
assume *SV*: *single-valued R*
assume *list-all2* (λ*x x'*. (*x*, *x'*) ∈ *R*) *x y* **and**
list-all2 (λ*x x'*. (*x*, *x'*) ∈ *R*) *x z*
thus *y*=*z*
apply (*induct arbitrary*: *z rule*: *list-all2-induct*)
apply *simp*
apply (*case-tac z*)
apply *force*
apply (*force intro*: *single-valuedD*[*OF SV*])
done
next
fix *x y z*
assume *SV*: *single-valued* (⟨*R*⟩*list-rel*)
assume (*x*,*y*)∈*R* (*x*,*z*)∈*R*
hence ([*x*],[*y*])∈⟨*R*⟩*list-rel* **and** ([*x*],[*z*])∈⟨*R*⟩*list-rel*
by (*auto simp*: *list-rel-def*)
with *single-valuedD*[*OF SV*] **show** *y*=*z* **by** *blast*
qed

lemma *list-rel-sv*[*relator-props*]:
single-valued R ⟹ *single-valued* (⟨*R*⟩*list-rel*)
by (*simp add*: *list-rel-sv-iff*)

lemma *list-rel-id*[*relator-props*]: ⟦*R*=*Id*⟧ ⟹ ⟨*R*⟩*list-rel* = *Id*
by (*auto simp add*: *list-rel-def list-all2-eq*[*symmetric*])

lemma *list-rel-id-simp*[*simp*]: ⟨*Id*⟩*list-rel* = *Id* **by** *tagged-solver*

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

```
proof clarsimp
  fix l l'
  assume (l,l')∈⟨R⟩list-rel
  thus (l,l')∈⟨R'⟩list-rel
    apply induct
    using A
    by auto
qed

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

Pointwise refinement for set types:

```
lemma set-rel-sv[relator-props]:
  single-valued (⟨R⟩set-rel)
  by (auto intro: single-valuedI dest: single-valuedD simp: set-rel-def) []

lemma set-rel-id[relator-props]: R=Id ⟹ ⟨R⟩set-rel = Id
  by (auto simp add: set-rel-def)

lemma set-rel-id-simp[simp]: ⟨Id⟩set-rel = Id by tagged-solver

lemma set-rel-csv[relator-props]:
  ⟦ single-valued (R⁻¹) ⟧
  ⟹ single-valued ((⟨R⟩set-rel)⁻¹)
  apply (rule single-valuedI)
  apply (simp only: converse-iff)

  apply (simp add: single-valued-def Image-def set-rel-def)
  apply (intro allI impI equalityI)
    apply (clarsimp, blast) []
    apply (clarsimp, drule (1) set-mp, blast) []
  done
```

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 $I \equiv \{(c,a)$. $a{=}\alpha$ $c \wedge I$ $c\}$
abbreviation *br*≡*build-rel*
lemmas *br-def*[*refine-rel-defs*] = *build-rel-def*

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

lemma *br-id*[*simp*]: *br id* (λ-. *True*) = *Id*
unfolding *build-rel-def* **by** *auto*

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

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

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

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

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

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

Convenience rule:

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

end

1.2 Basic Parametricity Reasoning

theory *Param-Tool*
imports *Relators*
begin

1.2.1 Auxiliary Lemmas

lemma *tag-both*: $[\![(Let\ x\ f, Let\ x'\ f') \in R]\!] \Longrightarrow (f\ x, f'\ x') \in R$ **by** *simp*
lemma *tag-rhs*: $[\![(c, Let\ x\ f) \in R]\!] \Longrightarrow (c, f\ x) \in R$ **by** *simp*
lemma *tag-lhs*: $[\![(Let\ x\ f, a) \in R]\!] \Longrightarrow (f\ x, a) \in R$ **by** *simp*

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*: $[\![(f,f') \in A \to B;\ (x,x') \in A]\!] \Longrightarrow (f\ x, f'\ x') \in B$
by (*simp-all add*: *fun-relD*)

1.2.2 ML-Setup

ML ⟪
signature PARAMETRICITY = sig
type param-ruleT = {
lhs: *term*,
rhs: *term*,
R: *term*,
rhs-head: *term*,
arity: *int*
}

val dest-param-term: *term -> param-ruleT*
val dest-param-rule: *thm -> param-ruleT*
val dest-param-goal: *int -> thm -> param-ruleT*

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

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

val fo-rule: *thm -> thm*

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

(*** *Nets of parametricity rules* ***)
type param-net
val net-empty: *param-net*
val net-add: *thm -> param-net -> param-net*
val net-del: *thm -> param-net -> param-net*
val net-add-int: *thm -> param-net -> param-net*

```
  val net-del-int: thm -> param-net -> param-net
  val net-tac: param-net -> Proof.context -> tactic'

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

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

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

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

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

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

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

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

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

fun adjust-arity-tac n ctxt i st =
  (if n = 0 then K all-tac
  else if n>0 then NTIMES n (DETERM o rtac @{thm fun-relI})
  else NTIMES (~n) (safe-fun-relD-tac ctxt)) i st

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

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


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

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

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

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

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

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

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

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

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

fun wrap-pr-op f thm = case try (`dest-param-rule) thm of
  NONE =>
    let
      val msg = Ignoring invalid parametricity theorem:
        ^ Display.string-of-thm-without-context thm
      val - = warning msg
    in I end
| SOME p => f p

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

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

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

      fun tac (r,thm) =
        adjust-arity-tac (#arity r - #arity g) ctxt
        THEN' DETERM o rtac thm

    in
      FIRST' (map tac rls) i st
    end

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

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

fun add-dflt thm = dflt-rules.map (net-add-int thm)
fun del-dflt thm = dflt-rules.map (net-del-int thm)
val add-dflt-attr = Thm.declaration-attribute add-dflt
val del-dflt-attr = Thm.declaration-attribute del-dflt

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

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

local
  open Refine-Util

  val param-modifiers =
    [Args.add -- Args.colon >> K (I, add-dflt-attr),
     Args.del -- Args.colon >> K (I, del-dflt-attr),
     Args.$$$ only -- Args.colon
       >> K (Context.proof-map (dflt-rules.map (K net-empty)),
             add-dflt-attr)]

  val param-flags =
    parse-bool-config use-asm cfg-use-asm
```

```
    || parse-bool-config single-step cfg-single-step

  in

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

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

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

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

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

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

end
⟫
```

setup *Parametricity.setup*

end

1.3 Parametricity Theorems for HOL

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

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 $(If\ c\ t\ e,\ If\ c'\ t'\ e')\in R$
using *assms* **by** *auto*

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

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

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

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

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

lemma *param-bool*[*param*]:
$(True,True)\in Id$
$(False,False)\in Id$
$(conj,conj)\in Id\to Id\to Id$
$(disj,disj)\in Id\to Id\to Id$
$(Not,Not)\in Id\to Id$
$(bool\text{-}case,bool\text{-}case)\in R\to R\to Id\to R$
$(bool\text{-}rec,bool\text{-}rec)\in R\to R\to Id\to R$
$(op \longleftrightarrow,\ op \longleftrightarrow)\in Id\to Id\to Id$
$(op \longrightarrow,\ op \longrightarrow)\in Id\to Id\to Id$
by (*auto split*: *bool.split simp*: *bool-case-def*[*symmetric*])

lemma *param-nat1*[*param*]:
$(0,\ 0::nat) \in Id$

(*Suc*, *Suc*) ∈ *Id* → *Id*
(*1*, *1*::*nat*) ∈ *Id*
(*numeral n*::*nat*,*numeral n*::*nat*) ∈ *Id*
(*op* <, *op* <::*nat* ⇒ -) ∈ *Id* → *Id* → *Id*
(*op* ≤, *op* ≤::*nat* ⇒ -) ∈ *Id* → *Id* → *Id*
(*op* =, *op* =::*nat* ⇒ -) ∈ *Id* → *Id* → *Id*
(*op* +::*nat*⇒-,*op* +)∈*Id*→*Id*→*Id*
(*op* −::*nat*⇒-,*op* −)∈*Id*→*Id*→*Id*
(*op* ∗::*nat*⇒-,*op* ∗)∈*Id*→*Id*→*Id*
(*op div*::*nat*⇒-,*op div*)∈*Id*→*Id*→*Id*
(*op mod*::*nat*⇒-,*op mod*)∈*Id*→*Id*→*Id*
by *auto*

lemma *param-nat-case*[*param*]:
(*nat-case*,*nat-case*)∈*Ra* → (*Id* → *Ra*) → *Id* → *Ra*
apply (*intro fun-relI*)
apply (*auto split*: *nat.split dest*: *fun-relD*)
done

lemma *param-nat-rec*[*param*]:
(*nat-rec*,*nat-rec*) ∈ *R* → (*Id* → *R* → *R*) → *Id* → *R*
apply (*intro fun-relI*)
proof −
case (*goal1 s s′ f f′ n n′*) **thus** *?case*
apply (*induct n′ arbitrary*: *n s s′*)
apply (*fastforce simp*: *fun-rel-def*)+
done
qed

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

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

```
  by (auto dest: fun-relD split: prod.split
    simp: prod-rel-def prod-case-def[symmetric])

lemma param-prod-case':
  ⟦ (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
    ⟧ ⟹ (prod-case f p, prod-case f' p') ∈ R
  by (auto split: prod.split)

lemma param-map-pair[param]:
  (map-pair, map-pair)
  ∈ (Ra→Rb) → (Rc→Rd) → ⟨Ra,Rc⟩prod-rel → ⟨Rb,Rd⟩prod-rel
  unfolding map-pair-def[abs-def]
  by parametricity

lemma param-apfst[param]:
  (apfst,apfst)∈(Ra→Rb)→⟨Ra,Rc⟩prod-rel→⟨Rb,Rc⟩prod-rel
  unfolding apfst-def[abs-def] by parametricity

lemma param-apsnd[param]:
  (apsnd,apsnd)∈(Rb→Rc)→⟨Ra,Rb⟩prod-rel→⟨Ra,Rc⟩prod-rel
  unfolding apsnd-def[abs-def] by parametricity

lemma param-curry[param]:
  (curry,curry) ∈ (⟨Ra,Rb⟩prod-rel → Rc) → Ra → Rb → Rc
  unfolding curry-def by parametricity

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 (λa. ∀x xa. (x, xa) ∈ Rb ⟶ (a x, 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
    using A
    apply (induct arbitrary: x x' rule: ccpo.fixp-induct[OF ccpo - M])
    apply (rule ADM)
    apply (subst ccpo.fixp-unfold[OF ccpo M'])
    apply (parametricity add: F)
    done
end


lemma param-option[param]:
  (None,None)∈⟨R⟩option-rel
  (Some,Some)∈R → ⟨R⟩option-rel
```

(*option-case*,*option-case*)∈*Rr*→(*R* → *Rr*)→⟨*R*⟩*option-rel* → *Rr*
(*option-rec*,*option-rec*)∈*Rr*→(*R* → *Rr*)→⟨*R*⟩*option-rel* → *Rr*
by (*auto split*: *option.split*
simp: *option-rel-def option-case-def*[*symmetric*]
dest: *fun-relD*)

lemma *param-option-case′*:
⟦ (*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*
⟧ ⟹ (*option-case fn fs x*, *option-case fn′ fs′ x′*) ∈ *R*
by (*auto split*: *option.split*)

lemma *the-paramL*: ⟦*l*≠*None*; (*l*,*r*)∈⟨*R*⟩*option-rel*⟧ ⟹ (*the l*, *the r*)∈*R*
apply (*cases l*)
by (*auto elim*: *option-relE*)

lemma *the-paramR*: ⟦*r*≠*None*; (*l*,*r*)∈⟨*R*⟩*option-rel*⟧ ⟹ (*the l*, *the r*)∈*R*
apply (*cases l*)
by (*auto elim*: *option-relE*)

lemma *the-default-param*[*param*]:
(*the-default*, *the-default*) ∈ *R* → ⟨*R*⟩*option-rel* → *R*
unfolding *the-default-def*
by *parametricity*

lemma *param-sum*[*param*]:
(*Inl*,*Inl*) ∈ *Rl* → ⟨*Rl*,*Rr*⟩*sum-rel*
(*Inr*,*Inr*) ∈ *Rr* → ⟨*Rl*,*Rr*⟩*sum-rel*
(*sum-case*,*sum-case*) ∈ (*Rl* → *R*) → (*Rr* → *R*) → ⟨*Rl*,*Rr*⟩*sum-rel* → *R*
(*sum-rec*,*sum-rec*) ∈ (*Rl* → *R*) → (*Rr* → *R*) → ⟨*Rl*,*Rr*⟩*sum-rel* → *R*
by (*fastforce split*: *sum.split dest*: *fun-relD*
simp: *sum-case-def*[*symmetric*])+

lemma *param-sum-case′*:
⟦ (*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*
⟧ ⟹ (*sum-case fl fr s*, *sum-case fl′ fr′ s′*)∈*R*
by (*auto split*: *sum.split*)

lemma *param-append*[*param*]:
(*append*, *append*)∈⟨*R*⟩*list-rel* → ⟨*R*⟩*list-rel* → ⟨*R*⟩*list-rel*
by (*auto simp*: *list-rel-def list-all2-appendI*)

lemma *param-list1*[*param*]:
(*Nil*,*Nil*)∈⟨*R*⟩*list-rel*
(*Cons*,*Cons*)∈*R* → ⟨*R*⟩*list-rel* → ⟨*R*⟩*list-rel*
(*list-case*,*list-case*)∈*Rr*→(*R*→⟨*R*⟩*list-rel*→*Rr*)→⟨*R*⟩*list-rel*→*Rr*

```
  apply (force dest: fun-relD split: list.split)+
  done

lemma param-list-rec[param]:
  (list-rec,list-rec)
  ∈ Ra → (Rb → ⟨Rb⟩list-rel → Ra → Ra) → ⟨Rb⟩list-rel → Ra
proof (intro fun-relI)
  case (goal1 a a' f f' l l')
  from goal1(3) show ?case
    using goal1(1,2)
    apply (induct arbitrary: a a')
    apply simp
    apply (fastforce dest: fun-relD)
    done
qed

lemma param-list-case':
  ⟦ (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
   ⟧ ⟹ (list-case n c l, list-case n' c' l') ∈ Ra
  by (auto split: list.split)

lemma param-map[param]:
  (map,map)∈(R1→R2) → ⟨R1⟩list-rel → ⟨R2⟩list-rel
  unfolding List.map-def by (parametricity)

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

schematic-lemma param-take[param]: (take,take)∈(?R::(-×-) set)
  unfolding take-def
  by (parametricity)

schematic-lemma param-drop[param]: (drop,drop)∈(?R::(-×-) set)
  unfolding drop-def
  by (parametricity)

schematic-lemma param-length[param]:
  (length,length)∈(?R::(-×-) set)
  unfolding List.list.list-size-overloaded-def
  by (parametricity)

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 (intro fun-relI)
  case (goal1 eq eq' l1 l1' l2 l2')
  thus ?case
    apply −
    apply (induct eq' l1' l2' arbitrary: l1 l2 rule: list-eq.induct)
    apply (simp-all only: list-eq.simps |
      elim list-relE |
      parametricity
    )+
    done
qed

lemma id-list-eq-aux[simp]: (list-eq op =) = (op =)
proof (intro ext)
  fix l1 l2 :: 'a list
  show list-eq op = l1 l2 = (l1 = l2)
    apply (induct op = :: 'a ⇒ - l1 l2 rule: list-eq.induct)
    apply simp-all
    done
qed

lemma param-list-equals[param]:
  ⟦ (op =, op =) ∈ R→R→Id ⟧
  ⟹ (op =, op =) ∈ ⟨R⟩list-rel → ⟨R⟩list-rel → Id
  unfolding id-list-eq-aux[symmetric]
  by (parametricity)

lemma param-tl[param]:
  (tl,tl) ∈ ⟨R⟩list-rel → ⟨R⟩list-rel
  unfolding tl-def
  by (parametricity)


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*: $(\forall\, x{\in}set\ l.\ P\ x) = list\text{-}all\text{-}rec\ P\ l$
by (*induct l*) *auto*

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

lemma *param-list-ball*[*param*]:
$[\![(P,P'){\in}(Ra{\to}Id);\ (l,l'){\in}\langle Ra\rangle\ list\text{-}rel]\!]$
$\Longrightarrow (\forall\, x{\in}set\ l.\ P\ x,\ \forall\, x{\in}set\ l'.\ P'\ x) \in Id$
unfolding *list-all-rec-eq*
unfolding *list-all-rec-def*
by (*parametricity*)

lemma *param-list-bex*[*param*]:
$[\![(P,P'){\in}(Ra{\to}Id);\ (l,l'){\in}\langle Ra\rangle\ list\text{-}rel]\!]$
$\Longrightarrow (\exists\, x{\in}set\ l.\ P\ x,\ \exists\, x{\in}set\ l'.\ P'\ x) \in Id$
unfolding *list-ex-rec-eq*[*abs-def*]
unfolding *list-ex-rec-def*
by (*parametricity*)

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

lemma *param-Ball*[*param*]: $(Ball,Ball){\in}\langle Ra\rangle set\text{-}rel{\to}(Ra{\to}Id){\to}Id$
by (*auto simp*: *set-rel-def dest*: *fun-relD*)
lemma *param-Bex*[*param*]: $(Bex,Bex){\in}\langle Ra\rangle set\text{-}rel{\to}(Ra{\to}Id){\to}Id$
apply (*auto simp*: *set-rel-def dest*: *fun-relD*)
apply (*drule* (*1*) *set-mp*)
apply (*erule DomainE*)
apply (*auto dest*: *fun-relD*)
done

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

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

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 $(l!i,l'!i') \in R$
using *LR I IR*

by (*induct arbitrary*: *i i'* *rule*: *list-rel-induct*)
(*auto simp*: *nth.simps split*: *nat.split*)

lemma *param-replicate*[*param*]:
(*replicate*,*replicate*) ∈ *nat-rel* → *R* → ⟨*R*⟩*list-rel*
unfolding *replicate-def* **by** *parametricity*

term *list-update*
lemma *param-list-update*[*param*]:
(*list-update*,*list-update*) ∈ ⟨*Ra*⟩*list-rel* → *nat-rel* → *Ra* → ⟨*Ra*⟩*list-rel*
unfolding *list-update-def*[*abs-def*] **by** *parametricity*

lemma *param-zip*[*param*]:
(*zip*, *zip*) ∈ ⟨*Ra*⟩*list-rel* → ⟨*Rb*⟩*list-rel* → ⟨⟨*Ra*,*Rb*⟩*prod-rel*⟩*list-rel*
unfolding *zip-def* **by** *parametricity*

lemma *param-upt*[*param*]:
(*upt*, *upt*) ∈ *nat-rel* → *nat-rel* → ⟨*nat-rel*⟩*list-rel*
unfolding *upt-def*[*abs-def*] **by** *parametricity*

lemma *param-concat*[*param*]: (*concat*, *concat*) ∈
⟨⟨*R*⟩*list-rel*⟩*list-rel* → ⟨*R*⟩*list-rel*
unfolding *concat-def*[*abs-def*] **by** *parametricity*

1.3.1 Sets

lemma *param-empty*[*param*]:
({},{})∈⟨*R*⟩*set-rel* **by** (*auto simp*: *set-rel-def*)

lemma *param-insert*[*param*]:
single-valued R ⟹ (*insert*,*insert*)∈*R*→⟨*R*⟩*set-rel*→⟨*R*⟩*set-rel*
by (*auto simp*: *set-rel-def dest*: *single-valuedD*)

lemma *param-union*[*param*]:
(*op* ∪, *op* ∪) ∈ ⟨*R*⟩*set-rel* → ⟨*R*⟩*set-rel* → ⟨*R*⟩*set-rel*
by (*auto simp*: *set-rel-def*)

lemma *param-inter*[*param*]:
assumes *single-valued* (*R*$^{-1}$)
shows (*op* ∩, *op* ∩) ∈ ⟨*R*⟩*set-rel* → ⟨*R*⟩*set-rel* → ⟨*R*⟩*set-rel*
using *assms* **by** (*auto dest*: *single-valuedD simp*: *set-rel-def*)

lemma *param-diff*[*param*]:
assumes *single-valued* (*R*$^{-1}$)
shows (*op* −, *op* −) ∈ ⟨*R*⟩*set-rel* → ⟨*R*⟩*set-rel* → ⟨*R*⟩*set-rel*
using *assms*
by (*auto dest*: *single-valuedD simp*: *set-rel-def*)

lemma *param-set*[*param*]:

```
  single-valued Ra ⟹ (set,set)∈⟨Ra⟩list-rel → ⟨Ra⟩set-rel
proof
  fix l l'
  assume A: single-valued Ra
  assume (l,l')∈⟨Ra⟩list-rel
  thus (set l, set l')∈⟨Ra⟩set-rel
    apply (induct)
    apply simp
    apply simp
    using A apply (parametricity)
    done
qed

end
```

Chapter 2

Automatic Refinement

2.1 Automatic Refinement Tool

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

2.1.1 Standard setup

Declaration of standard phases

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

Main method

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

    val autoref-phases =
```

```
    Args.$$$ phases |-- Args.colon |-- Scan.repeat1 Args.name

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

end

⟩⟩ Automatic Refinement
```

2.1.2 Tools

```
setup ⟪
  let
    fun higher-order-rl-of ctxt thm = case concl-of thm of
      @{mpat Trueprop ((-,?t)∈-)} => let
        open HOLogic
        val (f,args) = strip-comb t
      in
        if length args = 0 then
          thm
        else let
          val cT = TVar(('c,0),typeS)
          val c = Var ((c,0),cT)
          val R = Var ((R,0),mk-setT (mk-prodT (cT, fastype-of f)))
          val goal =
            HOLogic.mk-mem (HOLogic.mk-prod (c,f), R)
            |> HOLogic.mk-Trueprop
            |> cterm-of (Proof-Context.theory-of ctxt)

          val res-thm = Goal.prove-internal [] goal (fn - =>
            REPEAT (rtac @{thm fun-relI} 1)
            THEN (rtac thm 1)
            THEN (ALLGOALS atac)
          )

        in
          res-thm
        end
      end
```

```
    | - => raise THM(Expected autoref rule,~1,[thm])

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

2.1.3 Advanced Debugging

```
method-setup autoref-trans-step = ⟪
  Scan.succeed (fn ctxt => SIMPLE-METHOD' (
    Autoref-Translate.trans-dbg-step-tac (Autoref-Phases.init-data ctxt)
  ))
  ⟫ Single translation step, leaving unsolved side-coditions

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

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

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

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

method-setup autoref-id-op = ⟪
  Scan.succeed (fn ctxt => SIMPLE-METHOD' (
    Autoref-Id-Ops.id-tac ctxt
  ))
⟫
```

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

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

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

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

end
```

2.2 Standard HOL Bindings

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

2.2.1 Structural Expansion

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

is-hashcode (*prod-eq op= op=*) *hashcode*

This cannot be discharged by the rule

is-hashcode op= 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* **by** *simp*

ML ⟪
structure Autoref-Struct-Expand = *struct*
structure autoref-struct-expand = *Named-Thms* (
val name = @{*binding autoref-struct-expand*}
val description = *Autoref*: *Structural expansion lemmas*
)

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

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

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

end
⟫

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

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

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

declaration ⟪ *Tagged-Solver.add-triggers*
Relators.relator-props-solver @{*thms REL-IS-ID-trigger*} ⟫

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

abbreviation *PREFER-id R* ≡ *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*]:
$(x::bool) ::_i$ *i-bool*
conj $::_i$ *i-bool* $\rightarrow_i$ *i-bool* $\rightarrow_i$ *i-bool*
op $\longleftrightarrow ::_i$ *i-bool* $\rightarrow_i$ *i-bool* $\rightarrow_i$ *i-bool*
op $\longrightarrow ::_i$ *i-bool* $\rightarrow_i$ *i-bool* $\rightarrow_i$ *i-bool*
disj $::_i$ *i-bool* $\rightarrow_i$ *i-bool* $\rightarrow_i$ *i-bool*
Not $::_i$ *i-bool* $\rightarrow_i$ *i-bool*
bool-case $::_i I \rightarrow_i I \rightarrow_i$ *i-bool* $\rightarrow_i I$
bool-rec $::_i I \rightarrow_i I \rightarrow_i$ *i-bool* $\rightarrow_i I$
by *auto*

lemma *autoref-bool*[*autoref-rules*]:
$(x,x)\in$*bool-rel*
$(conj,conj)\in$*bool-rel*$\rightarrow$*bool-rel*$\rightarrow$*bool-rel*
$(disj,disj)\in$*bool-rel*$\rightarrow$*bool-rel*$\rightarrow$*bool-rel*
$(Not,Not)\in$*bool-rel*$\rightarrow$*bool-rel*
$(bool\text{-}case,bool\text{-}case)\in R\rightarrow R\rightarrow$*bool-rel*$\rightarrow R$
$(bool\text{-}rec,bool\text{-}rec)\in R\rightarrow R\rightarrow$*bool-rel*$\rightarrow R$
$(op \longleftrightarrow, op \longleftrightarrow)\in$*bool-rel*$\rightarrow$*bool-rel*$\rightarrow$*bool-rel*
$(op \longrightarrow, op \longrightarrow)\in$*bool-rel*$\rightarrow$*bool-rel*$\rightarrow$*bool-rel*
by (*auto split*: *bool.split simp*: *bool-case-def*[*symmetric*])

2.2.3 Standard Type Classes

We allow these operators for all interfaces.

lemma [*autoref-itype*]:
op $< ::_i I \rightarrow_i I \rightarrow_i$ *i-bool*
op $\leq ::_i I \rightarrow_i I \rightarrow_i$ *i-bool*
op $= ::_i I \rightarrow_i I \rightarrow_i$ *i-bool*
op $+ ::_i I \rightarrow_i I \rightarrow_i I$
op $- ::_i I \rightarrow_i I \rightarrow_i I$
op div $::_i I \rightarrow_i I \rightarrow_i I$
op mod $::_i I \rightarrow_i I \rightarrow_i I$
op $* ::_i I \rightarrow_i I \rightarrow_i I$

0 ::$_i$ *I*
1 ::$_i$ *I*
numeral x ::$_i$ *I*
neg-numeral x ::$_i$ *I*
by *auto*

lemma *pat-num-generic*[*autoref-op-pat*]:
0 ≡ *OP 0* :::$_i$ *I*
1 ≡ *OP 1* :::$_i$ *I*
numeral x ≡ (*OP* (*numeral x*) :::$_i$ *I*)
neg-numeral x ≡ (*OP* (*neg-numeral x*) :::$_i$ *I*)
by *simp-all*

lemma [*autoref-rules*]:
assumes *PRIO-TAG-GEN-ALGO*
shows (*op <*, *op <*) ∈ *Id*→*Id*→*bool-rel*
and (*op* ≤, *op* ≤) ∈ *Id*→*Id*→*bool-rel*
and (*op =*, *op =*) ∈ *Id*→*Id*→*bool-rel*
and (*numeral x*,*OP* (*numeral x*) ::: *Id*) ∈ *Id*
and (*neg-numeral x*,*OP* (*neg-numeral x*) ::: *Id*) ∈ *Id*
and (*0*,*0*) ∈ *Id*
and (*1*,*1*) ∈ *Id*
by *auto*

2.2.4 Functional Combinators

lemma [*autoref-itype*]: *id* ::$_i$ *I* →$_i$ *I* **by** *simp*
lemma *autoref-id*[*autoref-rules*]: (*id*,*id*)∈*R*→*R* **by** *auto*

term *op o*
lemma [*autoref-itype*]: *op* ∘ ::$_i$ (*Ia*→$_i$*Ib*) →$_i$ (*Ic* →$_i$ *Ia*) →$_i$ *Ic* →$_i$ *Ib*
by *simp*
lemma *autoref-comp*[*autoref-rules*]:
(*op o*, *op o*) ∈ (*Ra* → *Rb*) → (*Rc* → *Ra*) → *Rc* → *Rb*
by (*auto dest*: *fun-relD*)

lemma [*autoref-itype*]: *If* ::$_i$ *i-bool* →$_i$ *I* →$_i$ *I* →$_i$ *I* **by** *simp*
lemma *autoref-If*[*autoref-rules*]: (*If*,*If*)∈*Id*→*R*→*R*→*R* **by** *auto*
lemma *autoref-If-cong*[*autoref-rules*]:
assumes (*c′*,*c*)∈*Id*
assumes *REMOVE-INTERNAL c* ⟹ (*t′*,*t*)∈*R*
assumes ¬ *REMOVE-INTERNAL c* ⟹ (*e′*,*e*)∈*R*
shows (*If c′ t′ e′*,(*OP If* ::: *Id*→*R*→*R*→*R*)$*c*$*t*$*e*)∈*R*
using *assms* **by** (*auto*)

lemma [*autoref-itype*]: *Let* ::$_i$ *Ix* →$_i$ (*Ix*→$_i$*Iy*) →$_i$ *Iy* **by** *auto*
lemma *autoref-Let*[*autoref-rules*]:
(*Let*,*Let*)∈*Ra* → (*Ra*→*Rr*) → *Rr*
by (*auto dest*: *fun-relD*)

2.2.5 Unit

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

lemma [*autoref-rules*]: ((),())∈*unit-rel* **by** *simp*

2.2.6 Nat

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

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*)
by *simp-all*

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*
(*op* <, *op* <::*nat* ⇒ -) ∈ *nat-rel* → *nat-rel* → *bool-rel*
(*op* ≤, *op* ≤::*nat* ⇒ -) ∈ *nat-rel* → *nat-rel* → *bool-rel*
(*op* =, *op* =::*nat* ⇒ -) ∈ *nat-rel* → *nat-rel* → *bool-rel*
(*op* +::*nat*⇒-,*op* +)∈*nat-rel*→*nat-rel*→*nat-rel*
(*op* −::*nat*⇒-,*op* −)∈*nat-rel*→*nat-rel*→*nat-rel*
(*op div*::*nat*⇒-,*op div*)∈*nat-rel*→*nat-rel*→*nat-rel*
(*op* ∗, *op* ∗)∈*nat-rel*→*nat-rel*→*nat-rel*
(*op mod*, *op mod*)∈*nat-rel*→*nat-rel*→*nat-rel*
by *auto*

lemma *autoref-nat-case*[*autoref-rules*]:
(*nat-case*,*nat-case*)∈*Ra* → (*Id* → *Ra*) → *Id* → *Ra*
apply (*intro fun-relI*)
apply (*auto split*: *nat.split dest*: *fun-relD*)
done

lemma *autoref-nat-rec*: (*nat-rec*,*nat-rec*) ∈ *R* → (*Id* → *R* → *R*) → *Id* → *R*
apply (*intro fun-relI*)
proof −
case (*goal1 s s′ f f′ n n′*) **thus** *?case*
apply (*induct n′ arbitrary*: *n s s′*)
apply (*fastforce simp*: *fun-rel-def*)+
done
qed

2.2.7 Int

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

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

lemma *autoref-int*[*autoref-rules* (**overloaded**)]:
(*0*, *0*::*int*) ∈ *int-rel*
(*1*, *1*::*int*) ∈ *int-rel*
(*numeral n*::*int*,*numeral n*::*int*) ∈ *int-rel*
(*op <*, *op <*::*int* ⇒ -) ∈ *int-rel* → *int-rel* → *bool-rel*
(*op* ≤, *op* ≤::*int* ⇒ -) ∈ *int-rel* → *int-rel* → *bool-rel*
(*op =*, *op =*::*int* ⇒ -) ∈ *int-rel* → *int-rel* → *bool-rel*
(*op +*::*int*⇒-,*op +*)∈*int-rel*→*int-rel*→*int-rel*
(*op −*::*int*⇒-,*op −*)∈*int-rel*→*int-rel*→*int-rel*
(*op div*::*int*⇒-,*op div*)∈*int-rel*→*int-rel*→*int-rel*
(*uminus*,*uminus*)∈*int-rel*→*int-rel*
(*op ∗*, *op ∗*)∈*int-rel*→*int-rel*→*int-rel*
(*op mod*, *op mod*)∈*int-rel*→*int-rel*→*int-rel*
by *auto*

2.2.8 Product

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

lemma *prod-refine*[*autoref-rules*]:
(*Pair*,*Pair*)∈*Ra* → *Rb* → ⟨*Ra*,*Rb*⟩*prod-rel*
(*prod-case*,*prod-case*) ∈ (*Ra* → *Rb* → *Rr*) → ⟨*Ra*,*Rb*⟩*prod-rel* → *Rr*
(*prod-rec*,*prod-rec*) ∈ (*Ra* → *Rb* → *Rr*) → ⟨*Ra*,*Rb*⟩*prod-rel* → *Rr*
(*fst*,*fst*)∈⟨*Ra*,*Rb*⟩*prod-rel* → *Ra*
(*snd*,*snd*)∈⟨*Ra*,*Rb*⟩*prod-rel* → *Rb*
by (*auto dest*: *fun-relD split*: *prod.split*
simp: *prod-rel-def prod-case-def*[*symmetric*])

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 op* = (*Ra*→*Ra*→*Id*); *GEN-OP eqb op* = (*Rb*→*Rb*→*Id*)⟧

$\Longrightarrow$ (*prod-eq eqa eqb*,*op* =) ∈ ⟨*Ra*,*Rb*⟩*prod-rel* → ⟨*Ra*,*Rb*⟩*prod-rel* → *Id*
unfolding *prod-eq-def*[*abs-def*]
by (*fastforce dest*: *fun-relD*)

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

2.2.9 Option

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

lemma *autoref-opt*[*autoref-rules*]:
(*None*,*None*)∈⟨*R*⟩*option-rel*
(*Some*,*Some*)∈*R* → ⟨*R*⟩*option-rel*
(*option-case*,*option-case*)∈*Rr*→(*R* → *Rr*)→⟨*R*⟩*option-rel* → *Rr*
(*option-rec*,*option-rec*)∈*Rr*→(*R* → *Rr*)→⟨*R*⟩*option-rel* → *Rr*
by (*auto split*: *option.split*
simp: *option-rel-def option-case-def*[*symmetric*]
dest: *fun-relD*)

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

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

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*
by (*auto intro*!: *eq-reflection split*: *option.splits*)
lemma *autoref-is-None*[*autoref-rules*]:
(*is-None*,*is-None*)∈⟨*R*⟩*option-rel* → *Id*
by (*auto split*: *option.splits*)

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 op* = (*R*→*R*→*Id*)⟧
⟹ (*option-eq eq*,*op* =) ∈ ⟨*R*⟩*option-rel* → ⟨*R*⟩*option-rel* → *Id*
unfolding *option-eq-def*[*abs-def*]
by (*auto dest*: *fun-relD split*: *option.splits elim*!: *option-relE*)

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

2.2.10 Sum-Types

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

lemma *autoref-sum*[*autoref-rules*]:
(*Inl*,*Inl*) ∈ *Rl* → ⟨*Rl*,*Rr*⟩*sum-rel*
(*Inr*,*Inr*) ∈ *Rr* → ⟨*Rl*,*Rr*⟩*sum-rel*
(*sum-case*,*sum-case*) ∈ (*Rl* → *R*) → (*Rr* → *R*) → ⟨*Rl*,*Rr*⟩*sum-rel* → *R*
(*sum-rec*,*sum-rec*) ∈ (*Rl* → *R*) → (*Rr* → *R*) → ⟨*Rl*,*Rr*⟩*sum-rel* → *R*
by (*fastforce split*: *sum.split dest*: *fun-relD*
simp: *sum-case-def*[*symmetric*])+

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 op* = (*Rl*→*Rl*→*Id*); *GEN-OP eqr op* = (*Rr*→*Rr*→*Id*)⟧
⟹ (*sum-eq eql eqr*,*op* =) ∈ ⟨*Rl*,*Rr*⟩*sum-rel* → ⟨*Rl*,*Rr*⟩*sum-rel* → *Id*
unfolding *sum-eq-def*[*abs-def*]
by (*fastforce dest*: *fun-relD elim*!: *sum-relE*)

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

2.2.11 List

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

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

```
  by (auto simp: list-rel-def list-all2-appendI)

lemma refine-list[autoref-rules]:
  (Nil,Nil)∈⟨R⟩list-rel
  (Cons,Cons)∈R → ⟨R⟩list-rel → ⟨R⟩list-rel
  (list-case,list-case)∈Rr→(R→⟨R⟩list-rel→Rr)→⟨R⟩list-rel→Rr
  apply (force dest: fun-relD split: list.split)+
  done

lemma autoref-list-rec[autoref-rules]: (list-rec,list-rec)
  ∈ Ra → (Rb → ⟨Rb⟩list-rel → Ra → Ra) → ⟨Rb⟩list-rel → Ra
proof (intro fun-relI)
  case (goal1 a a' f f' l l')
  from goal1(3) show ?case
    using goal1(1,2)
    apply (induct arbitrary: a a')
    apply simp
    apply (fastforce dest: fun-relD)
    done
qed

lemma refine-map[autoref-rules]:
  (map,map)∈(R1→R2) → ⟨R1⟩list-rel → ⟨R2⟩list-rel
  using [[autoref-sbias = -1]]
  unfolding List.map-def
  by autoref

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

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

lemma autoref-nth[autoref-rules]:
  assumes (l,l')∈⟨R⟩list-rel
  assumes (i,i')∈Id
  assumes SIDE-PRECOND (i' < length l')
  shows (nth l i,(OP nth ::: ⟨R⟩list-rel → Id → R)$l'$i')∈R
  unfolding ANNOT-def
```

```
  using assms
  apply (induct arbitrary: i i')
  apply simp
  apply (case-tac i')
  apply auto
  done

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

lemma autoref-list-eq-aux:
  (list-eq,list-eq) ∈
    (R → R → Id) → ⟨R⟩list-rel → ⟨R⟩list-rel → Id
proof (intro fun-relI)
  case (goal1 eq eq' l1 l1' l2 l2')
  thus ?case
    apply −
    apply (induct eq' l1' l2' arbitrary: l1 l2 rule: list-eq.induct)
    apply simp
    apply (case-tac l1)
    apply simp
    apply (case-tac l2)
    apply (simp)
    apply (auto dest: fun-relD) []
    apply (case-tac l1)
    apply simp
    apply simp
    apply (case-tac l2)
    apply simp
    apply simp
    done
qed

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

lemma autoref-list-eq[autoref-rules (overloaded)]:
  GEN-OP eq op = (R→R→Id) ⟹ (list-eq eq, op =)
  ∈ ⟨R⟩list-rel → ⟨R⟩list-rel → Id
  unfolding autoref-tag-defs
```

apply (*subst list-eq-expand*)
apply (*parametricity add*: *autoref-list-eq-aux*)
done

lemma *autoref-hd*[*autoref-rules*]:
⟦ *SIDE-PRECOND* (*l'*≠[]); (*l*,*l'*) ∈ ⟨*R*⟩*list-rel* ⟧ ⟹
(*hd l*,(*OP hd* ::: ⟨*R*⟩*list-rel* → *R*)$*l'*) ∈ *R*
apply (*simp add*: *ANNOT-def*)
apply (*cases l'*)
apply *simp*
apply (*cases l*)
apply *auto*
done

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

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*)$*a*
[]=*a* ≡ (*OP is-Nil* :::$_i$ ⟨*I*⟩$_i$*i-list* →$_i$ *i-bool*)$*a*
by (*auto intro*!: *eq-reflection split*: *list.splits*)

lemma *autoref-is-Nil*[*param*,*autoref-rules*]:
(*is-Nil*,*is-Nil*)∈⟨*R*⟩*list-rel* → *bool-rel*
by (*auto split*: *list.splits*)

lemma *conv-to-is-Nil*:
l=[] ⟷ *is-Nil l*
[]=*l* ⟷ *is-Nil l*
unfolding *is-Nil-def* **by** (*auto split*: *list.split*)

lemma *autoref-butlast*[*param*, *autoref-rules*]:
(*butlast*,*butlast*) ∈ ⟨*R*⟩*list-rel* → ⟨*R*⟩*list-rel*
unfolding *butlast-def conv-to-is-Nil*
by *parametricity*

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* **by** *simp*
lemma *autoref-list-singleton*[*autoref-rules*]:
(λ*a*. [*a*],*op-list-singleton*) ∈ *R* → ⟨*R*⟩*list-rel*
by *auto*

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

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

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

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

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

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

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

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

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

apply (*autoref* (*keep-goal*))

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

oops

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

schematic-lemma
notes [*autoref-rules-raw*] = *IdI*[**where** *'a='a*::*numeral*]

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

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

schematic-lemma

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

schematic-lemma

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

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

end

2.3 Entry Point for the Automatic Refinement Tool

theory *Automatic-Refinement*
imports
Tool/Autoref-Tool
Autoref-Bindings-HOL
begin

The automatic refinement tool should be used by importing this theory

end