Automatic Data Refinement

Peter Lammich

April 23, 2013

Abstract

We present the Autoref tool for Isabelle/HOL, which auto- matically 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. Thanks to its integration with the Isabelle Refinement Framework and the Isabelle Collection Framework, Autoref can be used as a backend to a stepwise refinement based development approach, having access to a rich library of verified data structures. We have evaluated the tool by synthesizing efficiently executable refinements for some complex algorithms, as well as by implementing a library of generic algorithms for maps and sets.

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* $\Rightarrow$ -) $\Rightarrow$ (*'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

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

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

lemma *fun-relE2*:
assumes (*f*,*f'*)∈*Id* → *Rv*
assumes *t* = *f x*

shows $(t,f'\ x){\in}Rv$ **using** *assms*
by (*auto elim*: *fun-relD*)

Terminal Types

definition *unit-rel* **where** *unit-rel* == {((),())}

lemma *unit-rel-simps*[*simp*]: $(a,b){\in}$*unit-rel* **unfolding** *unit-rel-def* **by** *simp*

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*
≡ { $((a,b),(a',b'))$. $(a,a'){\in}R1 \wedge (b,b'){\in}R2$ }

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

lemma *prod-relI*: $[\![(a,a'){\in}R1;\ (b,b'){\in}R2]\!] \Longrightarrow ((a,b),(a',b'))\in$⟨*R1*,*R2*⟩*prod-rel*
by (*auto simp*: *prod-rel-def*)
lemma *prod-relE*:
assumes $(p,p')\in$⟨*R1*,*R2*⟩*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*]:
$((a,b),(a',b'))\in$⟨*R1*,*R2*⟩*prod-rel* ⟷ $(a,a'){\in}R1 \wedge (b,b'){\in}R2$
by (*auto intro*: *prod-relI elim*: *prod-relE*)

Option

definition *option-rel-def-internal*:
option-rel R ≡ { (*Some a*,*Some a'*) | *a a'*. $(a,a'){\in}R$ } ∪ {(*None*,*None*)}

lemma *option-rel-def*:
⟨*R*⟩*option-rel* ≡ { (*Some a*,*Some a'*) | *a a'*. $(a,a'){\in}R$ } ∪ {(*None*,*None*)}
by (*simp add*: *option-rel-def-internal relAPP-def*)

lemma *option-relI*:
(*None*,*None*)∈⟨*R*⟩ *option-rel*
$[\![\ (a,a'){\in}R\ ]\!] \Longrightarrow$ (*Some a*, *Some a'*)∈⟨*R*⟩*option-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*:
$sum\text{-}rel\ Rl\ Rr$
$\equiv \{ (Inl\ a,\ Inl\ a') \mid a\ a'.\ (a,a')\in Rl \} \cup$
$\{ (Inr\ a,\ Inr\ a') \mid a\ a'.\ (a,a')\in Rr \}$

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

Lists

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

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

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

lemma *list-rel-induct*[*induct set*,*consumes 1*, *case-names Nil Cons*]:
  assumes (*l*,*l'*)∈⟨*R*⟩ *list-rel*
  assumes *P* [] []
  assumes ⋀*x x' l l'*. ⟦ (*x*,*x'*)∈*R*; (*l*,*l'*)∈⟨*R*⟩*list-rel*; *P l l'* ⟧
    ⟹ *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*:
  ([],[])∈⟨*R*⟩*list-rel*
  ⟦ (*x*,*x'*)∈*R*; (*l*,*l'*)∈⟨*R*⟩*list-rel* ⟧ ⟹ (*x*#*l*,*x'*#*l'*)∈⟨*R*⟩*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 $(l,x'\#xs')\in\langle R\rangle$*list-rel* **obtains** *x xs* **where**
$l=x\#xs$ **and** $(x,x')\in R$ **and** $(xs,xs')\in\langle R\rangle$*list-rel*
using *assms*
apply (*cases* *l*)
apply *auto*
done

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

lemma *list-rel-imp-same-length*:
$(l, l') \in \langle R\rangle$*list-rel* $\Longrightarrow$ *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 $R \equiv \{(S,S').\ S'=R``S \wedge S\subseteq Domain\ R\}$

lemma *set-rel-def*:
$\langle R\rangle$*set-rel* $\equiv \{(S,S').\ S'=R``S \wedge S\subseteq Domain\ R\}$
by (*simp add*: *set-rel-def-internal relAPP-def*)

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

1.1.3 Automation

A solver for relator properties

lemma *relprop-triggers*:
$\bigwedge R.$ *single-valued* $R \Longrightarrow$ *single-valued* R
$\bigwedge R.\ R=Id \Longrightarrow R=Id$
$\bigwedge R.\ R=Id \Longrightarrow Id=R$
$\bigwedge R.\ Range\ R = UNIV \Longrightarrow Range\ R = UNIV$
$\bigwedge R.\ Range\ R = UNIV \Longrightarrow UNIV = Range\ R$
$\bigwedge R\ R'.\ R\subseteq R' \Longrightarrow R\subseteq R'$
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))))
⟫

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*: $S{=}UNIV \longleftrightarrow (\forall x.\ x{\in}S)$ **by** *auto*

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

  from *RAN* **obtain** x **where** *AR*: $(x,x'){\in}Ra$ **by** *auto*
  from *fun-relD*[*OF R1 AR*] **have** $(f\ x,g\ x') \in Rv$ **.**
  moreover from *fun-relD*[*OF R2 AR*] **have** $(f\ x,h\ x') \in Rv$ **.**
  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*]: $[\![R1{=}Id;\ R2{=}Id]\!] \Longrightarrow R1 \rightarrow R2 = Id$
  by (*auto simp*: *fun-rel-def*)

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

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

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

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

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

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

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

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

lemma *option-rel-sv*[*relator-props*]:
  $[\![$*single-valued R*$]\!] \Longrightarrow$ *single-valued* ($\langle R\rangle$*option-rel*)
  by (*auto intro*: *single-valuedI dest*: *single-valuedD simp*: *option-rel-def*)

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

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

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

lemma *option-rel-range*: *Range R* = *UNIV* $\Longrightarrow$ *Range* ($\langle R\rangle$*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*$]\!] \Longrightarrow$ *single-valued* ($\langle Rl,Rr\rangle$*sum-rel*)
  by (*auto intro*: *single-valuedI dest*: *single-valuedD simp*: *sum-rel-def*)

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

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

lemma *sum-rel-mono*[*relator-props*]:
  $[\![$ $Rl \subseteq Rl'$; $Rr \subseteq Rr'$ $]\!] \Longrightarrow$ $\langle Rl,Rr\rangle$*sum-rel* $\subseteq$ $\langle Rl',Rr'\rangle$*sum-rel*
  by (*auto simp*: *sum-rel-def*)

lemma *sum-rel-range*[*relator-props*]:
  $[\![$ *Range Rl*=*UNIV*; *Range Rr*=*UNIV* $]\!] \Longrightarrow$ *Range* ($\langle Rl,Rr\rangle$*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 ($\langle R\rangle$*list-rel*) $\longleftrightarrow$ *single-valued R*
  apply (*intro iffI*[*rotated*] *single-valuedI allI impI*)
  apply (*clarsimp simp*: *list-rel-def*)
proof –
  fix *x y z*
  assume *SV*: *single-valued R*
  assume *list-all2* ($\lambda x\ x'.\ (x, x') \in R$) *x y* **and**
    list-all2 ($\lambda x\ x'.\ (x, x') \in R$) *x z*
  thus *y*=*z*
    apply (*induct arbitrary*: *z rule*: *list-all2-induct*)

```
    apply simp
    apply (case-tac z)
    apply force
    apply (force intro: single-valuedD[OF SV])
    done
next
  fix x y z
  assume SV: single-valued (⟨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*]: $R{=}Id \Longrightarrow \langle R\rangle$*set-rel* = *Id*
by (*auto simp add*: *set-rel-def*)

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

lemma *set-rel-csv*[*relator-props*]:
$[\![$ *single-valued* (R^{-1}) $]\!]$
$\Longrightarrow$ *single-valued* $((\langle R\rangle set\text{-}rel)^{-1})$
apply (*rule single-valuedI*)
apply (*simp only*: *converse-iff*)
apply (*auto simp*: *single-valued-def Image-def set-rel-def*)
apply *blast*
apply (*drule* (*1*) *set-mp*)
by (*smt Domain-iff mem-Collect-eq*)

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 $I \equiv \{(c,a) \;.\; a{=}\alpha\ c \land I\ c\}$
abbreviation *br*≡*build-rel*
lemmas *br-def* = *build-rel-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* $(\alpha \circ \beta)$ $(\lambda s.\ J\ s \land I\ (\beta\ s))$
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* ∧ (α *c*,*a*)∈*R* }
  by (*auto simp add*: *br-def*)

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

Convenience rule:

end

1.2 Basic Parametricity Reasoning

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

1.2.1 Auxiliary Lemmas

lemma *tag-both*: ⟦ (*Let x f*,*Let x' f'*)∈*R* ⟧ ⟹ (*f x*,*f' x'*)∈*R* **by** *simp*
lemma *tag-rhs*: ⟦ (*c*,*Let x f*)∈*R* ⟧ ⟹ (*c*,*f x*)∈*R* **by** *simp*
lemma *tag-lhs*: ⟦ (*Let x f*,*a*)∈*R* ⟧ ⟹ (*f x*,*a*)∈*R* **by** *simp*

lemma *tagged-fun-relD-both*:
  ⟦ (*f*,*f'*)∈*A*→*B*; (*x*,*x'*)∈*A* ⟧ ⟹ (*Let x f*,*Let x' f'*)∈*B*
  and *tagged-fun-relD-rhs*: ⟦ (*f*,*f'*)∈*A*→*B*; (*x*,*x'*)∈*A* ⟧ ⟹ (*f x*,*Let x' f'*)∈*B*
  and *tagged-fun-relD-lhs*: ⟦ (*f*,*f'*)∈*A*→*B*; (*x*,*x'*)∈*A* ⟧ ⟹ (*Let x f*,*f' x'*)∈*B*
  and *tagged-fun-relD-none*: ⟦ (*f*,*f'*)∈*A*→*B*; (*x*,*x'*)∈*A* ⟧ ⟹ (*f x*,*f' x'*)∈*B*
  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'

  (*** 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

val param-fo-attr =
  let
```

```
      fun f thm = case concl-of thm of
        @{mpat Trueprop ((-,-)∈-→-)} => f (thm RS @{thm fun-relD})
      | - => thm
    in
      Scan.succeed (Thm.rule-attribute (K f))
    end

  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* =) ∈ *Ra*→*Ra*→*Id*
  ⟹ (*fun-upd*,*fun-upd*) ∈ (*Ra*→*Rb*) → *Ra* → *Rb* → *Ra* → *Rb*
  unfolding *fun-upd-def* [*abs-def*]
  by (*parametricity*)

lemma *param-bool*[*param*]:
  (*True*, *True*)∈*Id*
  (*False*,*False*)∈*Id*
  (*conj*,*conj*)∈*Id*→*Id*→*Id*
  (*disj*,*disj*)∈*Id*→*Id*→*Id*
  (*Not*,*Not*)∈*Id*→*Id*
  (*bool-case*,*bool-case*)∈*R*→*R*→*Id*→*R*
  (*bool-rec*,*bool-rec*)∈*R*→*R*→*Id*→*R*
  (*op* ⟷, *op* ⟷)∈*Id*→*Id*→*Id*
  by (*auto split*: *bool.split simp*: *bool-case-def* [*symmetric*])

lemma *param-nat1* [*param*]:
  (*0*, *0*::*nat*) ∈ *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*
  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*

$(op <,\ op <::int \Rightarrow \text{-}) \in Id \rightarrow Id \rightarrow Id$
$(op \leq,\ op \leq::int \Rightarrow \text{-}) \in Id \rightarrow Id \rightarrow Id$
$(op =,\ op =::int \Rightarrow \text{-}) \in Id \rightarrow Id \rightarrow Id$
$(op +::int\Rightarrow\text{-},op +)\in Id\rightarrow Id\rightarrow Id$
$(op -::int\Rightarrow\text{-},op -)\in Id\rightarrow Id\rightarrow Id$
by *auto*

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

lemma *param-prod-case'*:
$[\![\ (p,p')\in\langle Ra,Rb\rangle prod\text{-}rel;$
$\bigwedge a\ b\ a'\ b'.\ [\![\ p=(a,b);\ p'=(a',b');\ (a,a')\in Ra;\ (b,b')\in Rb\ ]\!]$
$\Longrightarrow (f\ a\ b,\ f'\ a'\ b')\in R$
$]\!] \Longrightarrow (prod\text{-}case\ f\ p,\ prod\text{-}case\ f'\ p') \in R$
by (*auto split*: *prod.split*)

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

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

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

lemma *param-curry*[*param*]:
$(curry,curry) \in (\langle Ra,Rb\rangle prod\text{-}rel \rightarrow Rc) \rightarrow Ra \rightarrow Rb \rightarrow Rc$
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*:
$admissible\ (\lambda a.\ \forall x\ xa.\ (x,\ xa) \in Rb \longrightarrow (a\ x,\ fixp\text{-}fun\ F'\ xa) \in Ra)$
assumes *F*: $(F,F')\in(Rb\rightarrow Ra)\rightarrow Rb\rightarrow Ra$
assumes *A*: $(x,x')\in Rb$

```
  shows (fixp-fun F x, fixp-fun F' x')∈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 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: (∀ x∈set l. P x) = list-all-rec P l
  by (induct l) auto

lemma list-ex-rec-eq: (∃ x∈set l. P x) = list-ex-rec P l
  by (induct l) auto

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
  unfolding list-all-rec-eq
  unfolding list-all-rec-def
  by (parametricity)

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
  unfolding list-ex-rec-eq[abs-def]
  unfolding list-ex-rec-def
  by (parametricity)

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

lemma param-Ball[param]: (Ball,Ball)∈⟨Ra⟩set-rel→(Ra→Id)→Id
  by (auto simp: set-rel-def dest: fun-relD)
lemma param-Bex[param]: (Bex,Bex)∈⟨Ra⟩set-rel→(Ra→Id)→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)
  ∈ ⟨Re⟩list-rel → (Rs→Id) → (Re→Rs→Rs) → Rs → Rs
  unfolding foldli-def
  by parametricity

lemma param-foldri[param]: (foldri, foldri)
  ∈ ⟨Re⟩list-rel → (Rs→Id) → (Re→Rs→Rs) → Rs → Rs
  unfolding foldri-def[abs-def]
```

by *parametricity*

lemma *param-nth*[*param*]:
assumes *I*: i'<*length* l'
assumes *IR*: (i,i')∈*nat-rel*
assumes *LR*: (l,l')∈⟨*R*⟩*list-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-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 $(op\ -,\ op\ -) \in \langle R\rangle set\text{-}rel \rightarrow \langle R\rangle set\text{-}rel \rightarrow \langle R\rangle set\text{-}rel$
using *assms*
by (*auto dest*: *single-valuedD simp*: *set-rel-def*)

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

end

Chapter 2

Automatic Refinement

2.1 Automatic Refinement

theory *Autoref*
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-Alt.id-phase phi
  #> register-phase rel-inf 20
       Autoref-Rel-Inf-Alt.roi-phase phi
  #> register-phase fix-rel 21
       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-Alt.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 *Autoref ../Parametricity/Parametricity*
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 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 = HOL-basic-ss 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*

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* $\to_i$ *i-bool* $\to_i$ *i-bool*
op $\longleftrightarrow$ $::_i$ *i-bool* $\to_i$ *i-bool* $\to_i$ *i-bool*
disj $::_i$ *i-bool* $\to_i$ *i-bool* $\to_i$ *i-bool*
Not $::_i$ *i-bool* $\to_i$ *i-bool*
bool-case $::_i$ $I \to_i I \to_i$ *i-bool* $\to_i I$
bool-rec $::_i$ $I \to_i I \to_i$ *i-bool* $\to_i I$
by *auto*

lemma *autoref-bool*[*autoref-rules*]:
(*x*,*x*)∈*bool-rel*
(*conj*,*conj*)∈*bool-rel*→*bool-rel*→*bool-rel*
(*disj*,*disj*)∈*bool-rel*→*bool-rel*→*bool-rel*
(*Not*,*Not*)∈*bool-rel*→*bool-rel*
(*bool-case*,*bool-case*)∈*R*→*R*→*bool-rel*→*R*
(*bool-rec*,*bool-rec*)∈*R*→*R*→*bool-rel*→*R*
(*op* ⟷, *op* ⟷)∈*bool-rel*→*bool-rel*→*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 $<$ $::_i$ $I \to_i I \to_i$ *i-bool*
op $\leq$ $::_i$ $I \to_i I \to_i$ *i-bool*
op $=$ $::_i$ $I \to_i I \to_i$ *i-bool*
op $+$ $::_i$ $I \to_i I \to_i I$
op $-$ $::_i$ $I \to_i I \to_i I$
op div $::_i$ $I \to_i I \to_i I$
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 $(0,0) \in Id$
and $(1,1) \in Id$
by *auto*

2.2.4 Functional Combinators

lemma [*autoref-itype*]: $id ::_i I \to_i I$ **by** *simp*
lemma *autoref-id*[*autoref-rules*]: $(id,id) \in R \to R$ **by** *auto*

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

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

lemma [*autoref-itype*]: $Let ::_i Ix \to_i (Ix \to_i Iy) \to_i Iy$ **by** *auto*
lemma *autoref-Let*[*autoref-rules*]:
$(Let,Let) \in Ra \to (Ra \to Rr) \to Rr$
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*]: $(x,x) \in unit\text{-}rel$ **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*]:
$0::nat \equiv OP\ 0\ :::_i\ i\text{-}nat$
$1::nat \equiv OP\ 1\ :::_i\ i\text{-}nat$
$(numeral\ x)::nat \equiv (OP\ (numeral\ x)\ :::_i\ i\text{-}nat)$
by *simp-all*

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

$(0,\ 0::nat) \in nat\text{-}rel$
$(Suc,\ Suc) \in nat\text{-}rel \rightarrow nat\text{-}rel$
$(1,\ 1::nat) \in nat\text{-}rel$
$(numeral\ n::nat, numeral\ n::nat) \in nat\text{-}rel$
$(op\ <,\ op\ <::nat \Rightarrow \text{-}) \in nat\text{-}rel \rightarrow nat\text{-}rel \rightarrow bool\text{-}rel$
$(op\ \leq,\ op\ \leq::nat \Rightarrow \text{-}) \in nat\text{-}rel \rightarrow nat\text{-}rel \rightarrow bool\text{-}rel$
$(op\ =,\ op\ =::nat \Rightarrow \text{-}) \in nat\text{-}rel \rightarrow nat\text{-}rel \rightarrow bool\text{-}rel$
$(op\ +::nat\Rightarrow\text{-}, op\ +)\in nat\text{-}rel\rightarrow nat\text{-}rel\rightarrow nat\text{-}rel$
$(op\ -::nat\Rightarrow\text{-}, op\ -)\in nat\text{-}rel\rightarrow nat\text{-}rel\rightarrow nat\text{-}rel$
$(op\ div::nat\Rightarrow\text{-}, op\ div)\in nat\text{-}rel\rightarrow nat\text{-}rel\rightarrow nat\text{-}rel$
by *auto*

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

lemma *autoref-nat-rec*: $(nat\text{-}rec, nat\text{-}rec) \in R \rightarrow (Id \rightarrow R \rightarrow R) \rightarrow Id \rightarrow R$
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*]:
$0::int \equiv OP\ 0\ :::_i\ i\text{-}int$
$1::int \equiv OP\ 1\ :::_i\ i\text{-}int$
$(numeral\ x)::int \equiv (OP\ (numeral\ x)\ :::_i\ i\text{-}int)$
$(neg\text{-}numeral\ x)::int \equiv (OP\ (neg\text{-}numeral\ x)\ :::_i\ i\text{-}int)$
by *simp-all*

lemma *autoref-int*[*autoref-rules* (**overloaded**)]:
$(0,\ 0::int) \in int\text{-}rel$
$(1,\ 1::int) \in int\text{-}rel$
$(numeral\ n::int, numeral\ n::int) \in int\text{-}rel$
$(op\ <,\ op\ <::int \Rightarrow \text{-}) \in int\text{-}rel \rightarrow int\text{-}rel \rightarrow bool\text{-}rel$
$(op\ \leq,\ op\ \leq::int \Rightarrow \text{-}) \in int\text{-}rel \rightarrow int\text{-}rel \rightarrow bool\text{-}rel$
$(op\ =,\ op\ =::int \Rightarrow \text{-}) \in int\text{-}rel \rightarrow int\text{-}rel \rightarrow bool\text{-}rel$
$(op\ +::int\Rightarrow\text{-}, op\ +)\in int\text{-}rel\rightarrow int\text{-}rel\rightarrow int\text{-}rel$

(*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*
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*)⟧
⟹ (*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* ($x \neq None$)
  assumes $(x',x) \in \langle R \rangle$*option-rel*
  shows (*the* x', (*OP the* ::: $\langle R \rangle$*option-rel* $\to$ R)\$$x$) $\in R$
  using *assms*
  by (*auto simp*: *option-rel-def*)

definition [*simp*]: *is-None a* $\equiv$ *case a of None* $\Rightarrow$ *True* | - $\Rightarrow$ *False*
lemma *pat-isNone*[*autoref-op-pat*]:
  a=None $\equiv$ (*OP is-None* $:::_i$ $\langle I \rangle_i$*i-option* $\to_i$ *i-bool*)\$*a*
  None=a $\equiv$ (*OP is-None* $:::_i$ $\langle I \rangle_i$*i-option* $\to_i$ *i-bool*)\$*a*
  by (*auto intro*!: *eq-reflection split*: *option.splits*)
lemma *autoref-is-None*[*autoref-rules*]:
  (*is-None*,*is-None*)$\in \langle R \rangle$*option-rel* $\to$ *Id*
  by (*auto split*: *option.splits*)

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

lemma *option-eq-autoref*[*autoref-rules* (**overloaded**)]:
  $[\![$*GEN-OP eq op* = ($R \to R \to Id$)$]\!]$
  $\Longrightarrow$ (*option-eq eq*,*op* =) $\in \langle R \rangle$*option-rel* $\to$ $\langle R \rangle$*option-rel* $\to$ *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* $\Rightarrow$ *interface* $\Rightarrow$ *interface*
lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of sum-rel i-sum*]

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

definition *sum-eq eql eqr s1 s2* $\equiv$
  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* $:::_i$ $\langle I\rangle_i$*i-list* $\to_i$ *i-bool*)$*a*
[]=*a* ≡ (*OP is-Nil* $:::_i$ $\langle I\rangle_i$*i-list* $\to_i$ *i-bool*)$*a*
by (*auto intro*!: *eq-reflection split*: *list.splits*)

lemma *autoref-is-Nil*[*autoref-rules*]:

(*is-Nil*,*is-Nil*)∈⟨*R*⟩*list-rel* → *Id*
by (*auto split*: *list.splits*)

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

Chapter 3

Generic Collections Framework

3.1 Orderings By Comparison Operator

theory *Intf-Comp*
imports
  ~~/src/HOL/Library/Zorn
  ../../Parametricity/Param-HOL
  ../../Autoref/Autoref-Bindings-HOL
begin

3.1.1 Basic Definitions

datatype *comp-res* = *LESS* | *EQUAL* | *GREATER*

consts *i-comp-res* :: *interface*
abbreviation *comp-res-rel* ≡ *Id* :: (*comp-res* × -) *set*
lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of comp-res-rel i-comp-res*]

definition *comp2le cmp a b* ≡
  case cmp a b of LESS ⇒ *True* | *EQUAL* ⇒ *True* | *GREATER* ⇒ *False*

definition *comp2lt cmp a b* ≡
  case cmp a b of LESS ⇒ *True* | *EQUAL* ⇒ *False* | *GREATER* ⇒ *False*

definition *comp2eq cmp a b* ≡
  case cmp a b of LESS ⇒ *False* | *EQUAL* ⇒ *True* | *GREATER* ⇒ *False*

locale *linorder-on* =
  fixes *D* :: *'a set*
  fixes *cmp* :: *'a* ⇒ *'a* ⇒ *comp-res*
  assumes *lt-eq*: ⟦*x*∈*D*; *y*∈*D*⟧ ⟹ *cmp x y* = *LESS* ⟷ (*cmp y x* = *GREATER*)
  assumes *refl*[*simp*, *intro*!]: *x*∈*D* ⟹ *cmp x x* = *EQUAL*
  assumes *trans*[*trans*]:

$\llbracket x{\in}D;\ y{\in}D;\ z{\in}D;\ cmp\ x\ y = LESS;\ cmp\ y\ z = LESS\rrbracket \Longrightarrow cmp\ x\ z = LESS$
$\llbracket x{\in}D;\ y{\in}D;\ z{\in}D;\ cmp\ x\ y = LESS;\ cmp\ y\ z = EQUAL\rrbracket \Longrightarrow cmp\ x\ z = LESS$
$\llbracket x{\in}D;\ y{\in}D;\ z{\in}D;\ cmp\ x\ y = EQUAL;\ cmp\ y\ z = LESS\rrbracket \Longrightarrow cmp\ x\ z = LESS$
$\llbracket x{\in}D;\ y{\in}D;\ z{\in}D;\ cmp\ x\ y = EQUAL;\ cmp\ y\ z = EQUAL\rrbracket \Longrightarrow cmp\ x\ z = EQUAL$
begin
  abbreviation *le* ≡ *comp2le cmp*
  abbreviation *lt* ≡ *comp2lt cmp*

  lemma *eq-sym*: $\llbracket x{\in}D;\ y{\in}D\rrbracket \Longrightarrow cmp\ x\ y = EQUAL \Longrightarrow cmp\ y\ x = EQUAL$
    apply (*cases cmp y x*)
    using *lt-eq lt-eq*[*symmetric*]
    by *auto*
end

abbreviation *linorder* ≡ *linorder-on UNIV*

lemma *linorder-to-class*:
  assumes *linorder cmp*
  assumes [*simp*]: $\bigwedge x\ y.\ cmp\ x\ y = EQUAL \Longrightarrow x{=}y$
  shows *class.linorder* (*comp2le cmp*) (*comp2lt cmp*)
proof −
  interpret *linorder-on UNIV cmp* **by** *fact*
  show *?thesis*
    apply (*unfold-locales*)
    unfolding *comp2le-def comp2lt-def*
    apply (*auto split*: *comp-res.split comp-res.split-asm*)
    using *lt-eq* **apply** *simp*
    using *lt-eq* **apply** *simp*
    using *lt-eq*[*symmetric*] **apply** *simp*
    apply (*drule* (*1*) *trans*[*rotated 3*], *simp-all*) []
    apply (*drule* (*1*) *trans*[*rotated 3*], *simp-all*) []
    apply (*drule* (*1*) *trans*[*rotated 3*], *simp-all*) []
    apply (*drule* (*1*) *trans*[*rotated 3*], *simp-all*) []
    using *lt-eq* **apply** *simp*
    using *lt-eq* **apply** *simp*
    using *lt-eq*[*symmetric*] **apply** *simp*
    done
qed

definition *dflt-cmp le lt a b* ≡
  if lt a b then LESS
  else if le a b then EQUAL
  else GREATER

lemma (**in** *linorder*) *class-to-linorder*:
  linorder (*dflt-cmp op* ≤ *op* <)

apply (*unfold-locales*)
unfolding *dflt-cmp-def*
by (*auto split*: *split-if-asm*)

lemma *restrict-linorder*: $[\![$*linorder-on D cmp* ; *D′⊆D*$]\!] \Longrightarrow$ *linorder-on D′ cmp*
apply (*rule linorder-on.intro*)
apply (*drule* (*1*) *set-rev-mp*)+
apply (*erule* (*2*) *linorder-on.lt-eq*)
apply (*drule* (*1*) *set-rev-mp*)+
apply (*erule* (*1*) *linorder-on.refl*)
apply (*drule* (*1*) *set-rev-mp*)+
apply (*erule* (*5*) *linorder-on.trans*)
apply (*drule* (*1*) *set-rev-mp*)+
apply (*erule* (*5*) *linorder-on.trans*)
apply (*drule* (*1*) *set-rev-mp*)+
apply (*erule* (*5*) *linorder-on.trans*)
apply (*drule* (*1*) *set-rev-mp*)+
apply (*erule* (*5*) *linorder-on.trans*)
done

3.1.2 Operations on Linear Orderings

Map with injective function

definition *cmp-img* **where** *cmp-img f cmp a b* ≡ *cmp* (*f a*) (*f b*)

lemma *img-linorder*[*intro?*]:
assumes *LO*: *linorder-on* (*f‘D*) *cmp*
shows *linorder-on D* (*cmp-img f cmp*)
apply *unfold-locales*
unfolding *cmp-img-def*
apply (*rule linorder-on.lt-eq*[*OF LO*], *auto*) []
apply (*rule linorder-on.refl*[*OF LO*], *auto*) []
apply (*erule* (*1*) *linorder-on.trans*[*OF LO*, *rotated* −*2*], *auto*) []
apply (*erule* (*1*) *linorder-on.trans*[*OF LO*, *rotated* −*2*], *auto*) []
apply (*erule* (*1*) *linorder-on.trans*[*OF LO*, *rotated* −*2*], *auto*) []
apply (*erule* (*1*) *linorder-on.trans*[*OF LO*, *rotated* −*2*], *auto*) []
done

Combine

definition *cmp-combine D1 cmp1 D2 cmp2 a b* ≡
if a∈D1 ∧ *b∈D1 then cmp1 a b*
else if a∈D1 ∧ *b∈D2 then LESS*
else if a∈D2 ∧ *b∈D1 then GREATER*
else cmp2 a b

lemma *UnE′*:
assumes *x∈A∪B*

```
  obtains x∈A | x∉A x∈B
  using assms by blast

lemma combine-linorder[intro?]:
  assumes linorder-on D1 cmp1
  assumes linorder-on D2 cmp2
  assumes D = D1∪D2
  shows linorder-on D (cmp-combine D1 cmp1 D2 cmp2)
  apply unfold-locales
  unfolding cmp-combine-def
  using assms apply −
  apply (simp only:)
  apply (elim UnE)
  apply (auto dest: linorder-on.lt-eq) [4]

  apply (simp only:)
  apply (elim UnE)
  apply (auto dest: linorder-on.refl) [2]

  apply (simp only:)
  apply (elim UnE')
  apply simp-all [8]
  apply (erule (5) linorder-on.trans)
  apply (erule (5) linorder-on.trans)

  apply (simp only:)
  apply (elim UnE')
  apply simp-all [8]
  apply (erule (5) linorder-on.trans)
  apply (erule (5) linorder-on.trans)

  apply (simp only:)
  apply (elim UnE')
  apply simp-all [8]
  apply (erule (5) linorder-on.trans)
  apply (erule (5) linorder-on.trans)

  apply (simp only:)
  apply (elim UnE')
  apply simp-all [8]
  apply (erule (5) linorder-on.trans)
  apply (erule (5) linorder-on.trans)
  done
```

3.1.3 Universal Linear Ordering

With Zorn's Lemma, we get a universal linear (even wf) ordering

```
definition univ-order-rel ≡ (SOME r. well-order-on UNIV r)
definition univ-cmp x y ≡
```

```
  if x=y then EQUAL
  else if (x,y)∈univ-order-rel then LESS
  else GREATER

lemma univ-wo: well-order-on UNIV univ-order-rel
  unfolding univ-order-rel-def
  using well-order-on[of UNIV]
  ..

lemma univ-linorder[intro?]: linorder univ-cmp
  apply unfold-locales
  unfolding univ-cmp-def
  apply (auto split: split-if-asm)
  using univ-wo
  apply –
  unfolding well-order-on-def linear-order-on-def partial-order-on-def
    preorder-on-def
  apply (auto simp add: antisym-def) []
  apply (unfold total-on-def, fast) []
  apply (auto simp add: antisym-def) []
  apply (unfold trans-def, fast)
  done
```

Extend any linear order to a universal order

```
definition cmp-extend D cmp ≡
  cmp-combine D cmp UNIV univ-cmp

lemma extend-linorder[intro?]:
  linorder-on D cmp ⟹ linorder (cmp-extend D cmp)
  unfolding cmp-extend-def
  apply rule
  apply assumption
  apply rule
  by simp
```

Lexicographic Order on Lists

```
fun cmp-lex where
  cmp-lex cmp [] [] = EQUAL
| cmp-lex cmp [] - = LESS
| cmp-lex cmp - [] = GREATER
| cmp-lex cmp (a#l) (b#m) = (
    case cmp a b of
      LESS ⇒ LESS
    | EQUAL ⇒ cmp-lex cmp l m
    | GREATER ⇒ GREATER)

primrec cmp-lex' where
  cmp-lex' cmp [] m = (case m of [] ⇒ EQUAL | - ⇒ LESS)
```

```
| cmp-lex' cmp (a#l) m = (case m of [] ⇒ GREATER | (b#m) ⇒
    (case cmp a b of
      LESS ⇒ LESS
    | EQUAL ⇒ cmp-lex' cmp l m
    | GREATER ⇒ GREATER
  ))

lemma cmp-lex-alt: cmp-lex cmp l m = cmp-lex' cmp l m
  apply (induct l arbitrary: m)
  apply (auto split: comp-res.split list.split)
  done

lemma (in linorder-on) lex-linorder[intro?]:
  linorder-on (lists D) (cmp-lex cmp)
proof
  fix l m
  assume l∈lists D    m∈lists D
  thus (cmp-lex cmp l m = LESS) = (cmp-lex cmp m l = GREATER)
    apply (induct cmp≡cmp l m rule: cmp-lex.induct)
    apply (auto split: comp-res.split simp: lt-eq)
    apply (auto simp: lt-eq[symmetric])
    done
next
  fix x
  assume x∈lists D
  thus cmp-lex cmp x x = EQUAL
    by (induct x) auto
next
  fix x y z
  assume M: x∈lists D    y∈lists D    z∈lists D

  {
    assume cmp-lex cmp x y = LESS    cmp-lex cmp y z = LESS
    thus cmp-lex cmp x z = LESS
      using M
      apply (induct cmp≡cmp x y arbitrary: z rule: cmp-lex.induct)
      apply (auto split: comp-res.split-asm comp-res.split)
      apply (case-tac z, auto) []
      apply (case-tac z,
        auto split: comp-res.split-asm comp-res.split,
        (drule (4) trans, simp)+
      ) []
      apply (case-tac z,
        auto split: comp-res.split-asm comp-res.split,
        (drule (4) trans, simp)+
      ) []
      done
  }
```

```
  {
    assume cmp-lex cmp x y = LESS    cmp-lex cmp y z = EQUAL
    thus cmp-lex cmp x z = LESS
      using M
      apply (induct cmp≡cmp x y arbitrary: z rule: cmp-lex.induct)
      apply (auto split: comp-res.split-asm comp-res.split)
      apply (case-tac z, auto) []
      apply (case-tac z,
        auto split: comp-res.split-asm comp-res.split,
        (drule (4) trans, simp)+
      ) []
      apply (case-tac z,
        auto split: comp-res.split-asm comp-res.split,
        (drule (4) trans, simp)+
      ) []
      done
  }

  {
    assume cmp-lex cmp x y = EQUAL    cmp-lex cmp y z = LESS
    thus cmp-lex cmp x z = LESS
      using M
      apply (induct cmp≡cmp x y arbitrary: z rule: cmp-lex.induct)
      apply (auto split: comp-res.split-asm comp-res.split)
      apply (case-tac z,
        auto split: comp-res.split-asm comp-res.split,
        (drule (4) trans, simp)+
      ) []
      done
  }

  {
    assume cmp-lex cmp x y = EQUAL    cmp-lex cmp y z = EQUAL
    thus cmp-lex cmp x z = EQUAL
      using M
      apply (induct cmp≡cmp x y arbitrary: z rule: cmp-lex.induct)
      apply (auto split: comp-res.split-asm comp-res.split)
      apply (case-tac z)
      apply (auto split: comp-res.split-asm comp-res.split)
      apply (drule (4) trans, simp)+
      done
  }
qed
```

Lexicographic Order on Pairs

```
fun cmp-prod where
  cmp-prod cmp1 cmp2 (a1,a2) (b1,b2)
  = (
```

```
    case cmp1 a1 b1 of
      LESS ⇒ LESS
    | EQUAL ⇒ cmp2 a2 b2
    | GREATER ⇒ GREATER)

lemma cmp-prod-alt: cmp-prod = (λcmp1 cmp2 (a1,a2) (b1,b2). (
    case cmp1 a1 b1 of
      LESS ⇒ LESS
    | EQUAL ⇒ cmp2 a2 b2
    | GREATER ⇒ GREATER))
  by (auto intro!: ext)

lemma prod-linorder[intro?]:
  assumes A: linorder-on A cmp1
  assumes B: linorder-on B cmp2
  shows linorder-on (A×B) (cmp-prod cmp1 cmp2)
proof −
  interpret A: linorder-on A cmp1
    + B: linorder-on B cmp2 by fact+

  show ?thesis
    apply unfold-locales
    apply (auto split: comp-res.split comp-res.split-asm,
      simp-all add: A.lt-eq B.lt-eq,
      simp-all add: A.lt-eq[symmetric]
      ) []

    apply (auto split: comp-res.split comp-res.split-asm) []

    apply (auto split: comp-res.split comp-res.split-asm) []
    apply (drule (4) A.trans B.trans, simp)+

    apply (auto split: comp-res.split comp-res.split-asm) []
    apply (drule (4) A.trans B.trans, simp)+

    apply (auto split: comp-res.split comp-res.split-asm) []
    apply (drule (4) A.trans B.trans, simp)+

    apply (auto split: comp-res.split comp-res.split-asm) []
    apply (drule (4) A.trans B.trans, simp)+
    done
qed
```

3.1.4 Universal Ordering for Sets that is Effective for Finite Sets

Sorted Lists of Sets

Some more results about sorted lists of finite sets

lemma *set-to-map-set-is-map-of*:
  distinct (*map fst l*) $\Longrightarrow$ *set-to-map* (*set l*) = *map-of l*
  apply (*induct l*)
  apply (*auto simp*: *set-to-map-insert*)
  done

context *linorder* **begin**

  lemma *sorted-list-of-set-eq-nil*[*simp*]:
    assumes *finite A*
    shows *sorted-list-of-set A* = [] $\longleftrightarrow$ *A*={}
    using *assms*
    apply (*induct rule*: *finite-induct*)
    apply *simp*
    apply *simp*
    done

  lemma *sorted-list-of-set-eq-nil2*[*simp*]:
    assumes *finite A*
    shows [] = *sorted-list-of-set A* $\longleftrightarrow$ *A*={}
    using *assms*
    by (*auto dest*: *sym*)

  lemma *set-insort*[*simp*]: *set* (*insort x l*) = *insert x* (*set l*)
    by (*induct l*) *auto*

  lemma *sorted-list-of-set-inj-aux*:
    fixes *A B* :: *'a set*
    assumes *finite A*
    assumes *finite B*
    assumes *sorted-list-of-set A* = *sorted-list-of-set B*
    shows *A*=*B*
    using *assms*
  proof −
    from ‹*finite B*› **have** *B* = *set* (*sorted-list-of-set B*) **by** *simp*
    also from *assms* **have** ... = *set* (*sorted-list-of-set* (*A*))
      by *simp*
    also from ‹*finite A*›
    have *set* (*sorted-list-of-set* (*A*)) = *A*
      by *simp*
    finally show *?thesis* **by** *simp*
  qed

  lemma *sorted-list-of-set-inj*: *inj-on sorted-list-of-set* (*Collect finite*)
    apply (*rule inj-onI*)
    using *sorted-list-of-set-inj-aux*
    by *blast*

  lemma *the-sorted-list-of-set*:

```
  assumes distinct l
  assumes sorted l
  shows sorted-list-of-set (set l) = l
  using assms
  by (simp
    add: sorted-list-of-set-sort-remdups distinct-remdups-id sorted-sort-id)


definition sorted-list-of-map m ≡
  map (λk. (k, the (m k))) (sorted-list-of-set (dom m))

lemma the-sorted-list-of-map:
  assumes distinct (map fst l)
  assumes sorted (map fst l)
  shows sorted-list-of-map (map-of l) = l
proof –
  have dom (map-of l) = set (map fst l) by (induct l) force+
  hence sorted-list-of-set (dom (map-of l)) = map fst l
    using the-sorted-list-of-set[OF assms] by simp
  hence sorted-list-of-map (map-of l)
    = map (λk. (k, the (map-of l k))) (map fst l)
    unfolding sorted-list-of-map-def by simp
  also have ... = l using ‹distinct (map fst l)›
    apply (induct l)
    apply auto
    by (smt List.map.compositionality image-set map-ext)
  finally show ?thesis .
qed

lemma map-of-sorted-list-of-map[simp]:
  assumes FIN: finite (dom m)
  shows map-of (sorted-list-of-map m) = m
  unfolding sorted-list-of-map-def
proof –
  have set (sorted-list-of-set (dom m)) = dom m
    and DIST: distinct (sorted-list-of-set (dom m))
    by (simp-all add: FIN)

  have [simp]: (fst ∘ (λk. (k, the (m k)))) = id by auto

  have [simp]: (λk. (k, the (m k))) ‘ dom m = map-to-set m
    by (auto simp: map-to-set-def)

  show map-of (map (λk. (k, the (m k))) (sorted-list-of-set (dom m))) = m
    apply (subst set-to-map-set-is-map-of[symmetric])
    apply (simp add: DIST)
    apply (subst set-map)
    apply (simp add: FIN map-to-set-inverse)
    done
```

```
  qed

  lemma sorted-list-of-map-inj-aux:
    fixes A B :: 'a⇀'b
    assumes [simp]: finite (dom A)
    assumes [simp]: finite (dom B)
    assumes E: sorted-list-of-map A = sorted-list-of-map B
    shows A=B
    using assms
  proof –
    have A = map-of (sorted-list-of-map A) by simp
    also note E
    also have map-of (sorted-list-of-map B) = B by simp
    finally show ?thesis .
  qed

  lemma sorted-list-of-map-inj:
    inj-on sorted-list-of-map (Collect (finite o dom))
    apply (rule inj-onI)
    using sorted-list-of-map-inj-aux
    by auto
end

definition cmp-set cmp ≡
  cmp-extend (Collect finite) (
    cmp-img
      (linorder.sorted-list-of-set (comp2le cmp))
      (cmp-lex cmp)
  )

thm img-linorder

lemma set-ord-linear[intro?]:
  linorder cmp ⟹ linorder (cmp-set cmp)
  unfolding cmp-set-def
  apply rule
  apply rule
  apply (rule restrict-linorder)
  apply (erule linorder-on.lex-linorder)
  apply simp
  done

definition cmp-map cmpk cmpv ≡
  cmp-extend (Collect (finite o dom)) (
    cmp-img
      (linorder.sorted-list-of-map (comp2le cmpk))
      (cmp-lex (cmp-prod cmpk cmpv))
  )
```

lemma *map-to-set-inj*[*intro*!]: *inj map-to-set*
  apply (*rule inj-onI*)
  unfolding *map-to-set-def*
  apply (*rule ext*)
  apply (*case-tac x xa*)
  apply (*case-tac* [!] *y xa*)
  apply *force+*
  done

corollary *map-to-set-inj'*[*intro*!]: *inj-on map-to-set S*
  by (*metis map-to-set-inj subset-UNIV subset-inj-on*)

lemma *map-ord-linear*[*intro?*]:
  assumes *A*: *linorder cmpk*
  assumes *B*: *linorder cmpv*
  shows *linorder* (*cmp-map cmpk cmpv*)
proof −
  interpret *lk*!: *linorder-on UNIV cmpk* **by** *fact*
  interpret *lv*!: *linorder-on UNIV cmpv* **by** *fact*

  show *?thesis*
    unfolding *cmp-map-def*
    apply *rule*
    apply *rule*
    apply (*rule restrict-linorder*)
    apply (*rule linorder-on.lex-linorder*)
    apply (*rule*)
    apply *fact*
    apply *fact*
    apply *simp*
    done
qed

locale *eq-linorder-on* = *linorder-on* +
  assumes *cmp-imp-equal*: $[\![x \in D;\ y \in D]\!] \Longrightarrow cmp\ x\ y = EQUAL \Longrightarrow x = y$
begin
  lemma *cmp-eq*[*simp*]: $[\![x \in D;\ y \in D]\!] \Longrightarrow cmp\ x\ y = EQUAL \longleftrightarrow x = y$
    by (*auto simp*: *cmp-imp-equal*)
end

abbreviation *eq-linorder* ≡ *eq-linorder-on UNIV*

lemma *dflt-cmp-2inv*[*simp*]:
  dflt-cmp (*comp2le cmp*) (*comp2lt cmp*) = *cmp*
  unfolding *dflt-cmp-def*[*abs-def*] *comp2le-def*[*abs-def*] *comp2lt-def*[*abs-def*]
  apply (*auto split*: *comp-res.splits intro*!: *ext*)
  done

lemma (**in** *linorder*) *dflt-cmp-inv2*[*simp*]:
  shows
  (*comp2le* (*dflt-cmp op* ≤ *op* <))= *op* ≤
  (*comp2lt* (*dflt-cmp op* ≤ *op* <))= *op* <
proof –
  show (*comp2lt* (*dflt-cmp op* ≤ *op* <))= *op* <
    unfolding *dflt-cmp-def*[*abs-def*] *comp2le-def*[*abs-def*] *comp2lt-def*[*abs-def*]
    apply (*auto split*: *comp-res.splits intro*!: *ext*)
    done

  show (*comp2le* (*dflt-cmp op* ≤ *op* <)) = *op* ≤
    unfolding *dflt-cmp-def*[*abs-def*] *comp2le-def*[*abs-def*] *comp2lt-def*[*abs-def*]
    apply (*auto split*: *comp-res.splits intro*!: *ext*)
    done

qed

lemma *eq-linorder-class-conv*:
  eq-linorder cmp ⟷ *class.linorder* (*comp2le cmp*) (*comp2lt cmp*)
proof
  assume *eq-linorder cmp*
  then interpret *eq-linorder-on UNIV cmp* **.**
  have *linorder cmp* **by** *unfold-locales*
  show *class.linorder* (*comp2le cmp*) (*comp2lt cmp*)
    apply (*rule linorder-to-class*)
    apply *fact*
    by *simp*
next
  assume *class.linorder* (*comp2le cmp*) (*comp2lt cmp*)
  then interpret *linorder comp2le cmp comp2lt cmp* **.**

  from *class-to-linorder* **interpret** *linorder-on UNIV cmp*
    by *simp*
  show *eq-linorder cmp*
  proof
    fix *x y*
    assume *cmp x y* = *EQUAL*
    hence *comp2le cmp x y* ¬*comp2lt cmp x y*
      by (*auto simp*: *comp2le-def comp2lt-def*)
    thus *x*=*y* **by** *simp*
  qed
qed

lemma (**in** *linorder*) *class-to-eq-linorder*:
  eq-linorder (*dflt-cmp op* ≤ *op* <)
proof –
  interpret *linorder-on UNIV dflt-cmp op* ≤ *op* <
    by (*rule class-to-linorder*)

```
  show ?thesis
    apply unfold-locales
    apply (auto simp: dflt-cmp-def split: split-if-asm)
    done
qed

lemma eq-linorder-comp2eq-eq:
  assumes eq-linorder cmp
  shows comp2eq cmp = op =
proof −
  interpret eq-linorder-on UNIV cmp by fact
  show ?thesis
    apply (intro ext)
    unfolding comp2eq-def
    apply (auto split: comp-res.split dest: refl)
    done
qed

lemma restrict-eq-linorder:
  assumes eq-linorder-on D cmp
  assumes S: D'⊆D
  shows eq-linorder-on D' cmp
proof −
  interpret eq-linorder-on D cmp by fact

  show ?thesis
    apply (rule eq-linorder-on.intro)
    apply (rule restrict-linorder[where D=D])
    apply unfold-locales []
    apply fact
    apply unfold-locales
    using S
    apply −
    apply (drule (1) set-rev-mp)+
    apply auto
    done
qed

lemma combine-eq-linorder[intro?]:
  assumes A: eq-linorder-on D1 cmp1
  assumes B: eq-linorder-on D2 cmp2
  assumes EQ: D=D1∪D2
  shows eq-linorder-on D (cmp-combine D1 cmp1 D2 cmp2)
proof −
  interpret A: eq-linorder-on D1 cmp1 by fact
  interpret B: eq-linorder-on D2 cmp2 by fact
  interpret linorder-on (D1 ∪ D2)   (cmp-combine D1 cmp1 D2 cmp2)
    apply rule
```

```
    apply unfold-locales
    by simp

  show ?thesis
    apply (simp only: EQ)
    apply unfold-locales
    unfolding cmp-combine-def
    by (auto split: split-if-asm)
qed

lemma img-eq-linorder[intro?]:
  assumes A: eq-linorder-on (f`D) cmp
  assumes INJ: inj-on f D
  shows eq-linorder-on D (cmp-img f cmp)
proof –
  interpret eq-linorder-on f`D cmp by fact
  interpret L: linorder-on (D)    (cmp-img f cmp)
    apply rule
    apply unfold-locales
    done

  show ?thesis
    apply unfold-locales
    unfolding cmp-img-def
    using INJ
    apply (auto dest: inj-onD)
    done
qed

lemma univ-eq-linorder[intro?]:
  shows eq-linorder univ-cmp
  apply (rule eq-linorder-on.intro)
  apply rule
  apply unfold-locales
  unfolding univ-cmp-def
  apply (auto split: split-if-asm)
  done

lemma extend-eq-linorder[intro?]:
  assumes eq-linorder-on D cmp
  shows eq-linorder (cmp-extend D cmp)
proof –
  interpret eq-linorder-on D cmp by fact
  show ?thesis
    unfolding cmp-extend-def
    apply (rule)
    apply fact
    apply rule
    by simp
```

```
qed

lemma lex-eq-linorder[intro?]:
  assumes eq-linorder-on D cmp
  shows eq-linorder-on (lists D) (cmp-lex cmp)
proof –
  interpret eq-linorder-on D cmp by fact
  show ?thesis
    apply (rule eq-linorder-on.intro)
    apply rule
    apply unfold-locales
  proof –
    case (goal1 l m)
    thus ?case
      apply (induct cmp≡cmp l m rule: cmp-lex.induct)
      apply (auto split: comp-res.splits)
      done
  qed
qed

lemma prod-eq-linorder[intro?]:
  assumes eq-linorder-on D1 cmp1
  assumes eq-linorder-on D2 cmp2
  shows eq-linorder-on (D1×D2) (cmp-prod cmp1 cmp2)
proof –
  interpret A: eq-linorder-on D1 cmp1 by fact
  interpret B: eq-linorder-on D2 cmp2 by fact
  show ?thesis
    apply (rule eq-linorder-on.intro)
    apply rule
    apply unfold-locales
    apply (auto split: comp-res.splits)
    done
qed

lemma set-ord-eq-linorder[intro?]:
  eq-linorder cmp ⟹ eq-linorder (cmp-set cmp)
  unfolding cmp-set-def
  apply rule
  apply rule
  apply (rule restrict-eq-linorder)
  apply rule
  apply assumption
  apply simp

  apply (rule linorder.sorted-list-of-set-inj)
  apply (subst (asm) eq-linorder-class-conv)
  .
```

lemma *map-ord-eq-linorder*[*intro?*]:
  ⟦*eq-linorder cmpk*; *eq-linorder cmpv*⟧ ⟹ *eq-linorder* (*cmp-map cmpk cmpv*)
  unfolding *cmp-map-def*
  apply *rule*
  apply *rule*
  apply (*rule restrict-eq-linorder*)
  apply *rule*
  apply *rule*
  apply *assumption*
  apply *assumption*
  apply *simp*

  apply (*rule linorder.sorted-list-of-map-inj*)
  apply (*subst* (*asm*) *eq-linorder-class-conv*)
  .

definition *cmp-unit* :: *unit* ⇒ *unit* ⇒ *comp-res*
  where [*simp*]: *cmp-unit u v* ≡ *EQUAL*

lemma *cmp-unit-eq-linorder*:
  eq-linorder cmp-unit
  by *unfold-locales simp-all*

3.1.5 Parametricity

lemma *param-cmp-extend*[*param*]:
  assumes (*cmp*,*cmp'*)∈*R* → *R* → *Id*
  assumes *Range R* ⊆ *D*
  shows (*cmp*,*cmp-extend D cmp'*) ∈ *R* → *R* → *Id*
  unfolding *cmp-extend-def cmp-combine-def*[*abs-def*]
  using *assms*
  by (*force dest*: *fun-relD*)

lemma *param-cmp-img*[*param*]:
  (*cmp-img*,*cmp-img*) ∈ (*Ra*→*Rb*) → (*Rb*→*Rb*→*Rc*) → *Ra* → *Ra* → *Rc*
  unfolding *cmp-img-def*[*abs-def*]
  by *parametricity*

lemma *param-comp-res*[*param*]:
  (*LESS*,*LESS*)∈*Id*
  (*EQUAL*,*EQUAL*)∈*Id*
  (*GREATER*,*GREATER*)∈*Id*
  (*comp-res-case*,*comp-res-case*)∈*Ra*→*Ra*→*Ra*→*Id*→*Ra*
  by (*auto split*: *comp-res.split*)

term *cmp-lex*
lemma *param-cmp-lex*[*param*]:
  (*cmp-lex*,*cmp-lex*)∈(*Ra*→*Rb*→*Id*)→⟨*Ra*⟩*list-rel*→⟨*Rb*⟩*list-rel*→*Id*
  unfolding *cmp-lex-alt*[*abs-def*] *cmp-lex'-def*

```
  by (parametricity)

term cmp-prod
lemma param-cmp-prod[param]:
  (cmp-prod,cmp-prod)∈
  (Ra→Rb→Id)→(Rc→Rd→Id)→⟨Ra,Rc⟩prod-rel→⟨Rb,Rd⟩prod-rel→Id
  unfolding cmp-prod-alt
  by (parametricity)

lemma param-cmp-unit[param]:
  (cmp-unit,cmp-unit)∈Id→Id→Id
  by auto

lemma param-comp2eq[param]: (comp2eq,comp2eq)∈(R→R→Id)→R→R→Id
  unfolding comp2eq-def[abs-def]
  by (parametricity)



lemma cmp-combine-paramD:
  assumes (cmp,cmp-combine D1 cmp1 D2 cmp2)∈R→R→Id
  assumes Range R ⊆ D1
  shows (cmp,cmp1)∈R→R→Id
  using assms
  unfolding cmp-combine-def[abs-def]
  apply (intro fun-relI)
  apply (drule-tac x=a in fun-relD, assumption)
  apply (drule-tac x=aa in fun-relD, assumption)
  apply (drule RangeI, drule (1) set-rev-mp)
  apply (drule RangeI, drule (1) set-rev-mp)
  apply simp
  done

lemma cmp-extend-paramD:
  assumes (cmp,cmp-extend D cmp')∈R→R→Id
  assumes Range R ⊆ D
  shows (cmp,cmp')∈R→R→Id
  using assms
  unfolding cmp-extend-def
  apply (rule cmp-combine-paramD)
  done

end
```

3.2 Map Interface

theory *Intf-Map*

imports *../../Autoref/Autoref-Bindings-HOL*
begin

consts *i-map* :: *interface* ⇒ *interface* ⇒ *interface*

definition [*simp*]: *op-map-empty* ≡ *Map.empty*
definition *op-map-lookup* :: *'k* ⇒ (*'k*⇀*'v*) ⇀ *'v*
  where [*simp*]: *op-map-lookup k m* ≡ *m k*
definition [*simp*]: *op-map-update k v m* ≡ *m*(*k*↦*v*)
definition [*simp*]: *op-map-delete k m* ≡ *m* |‘ (−{*k*})
definition [*simp*]: *op-map-restrict P m* ≡ *m* |‘ {*k*∈*dom m*. *P* (*k*, *the* (*m k*))}
definition [*simp*]: *op-map-isEmpty x* ≡ *x*=*Map.empty*
definition [*simp*]: *op-map-isSng x* ≡ ∃ *k v*. *x*=[*k*↦*v*]
definition [*simp*]: *op-map-ball m P* ≡ *Ball* (*map-to-set m*) *P*
definition [*simp*]: *op-map-bex m P* ≡ *Bex* (*map-to-set m*) *P*
definition [*simp*]: *op-map-size m* ≡ *card* (*dom m*)
definition [*simp*]: *op-map-size-abort n m* ≡ *min n* (*card* (*dom m*))

lemma [*autoref-op-pat*]:
  Map.empty ≡ *op-map-empty*
  (*m*::*'k*⇀*'v*) *k* ≡ *op-map-lookup*$*k*$*m*
  m(*k*↦*v*) ≡ *op-map-update*$*k*$*v*$*m*
  m |‘ (−{*k*}) ≡ *op-map-delete*$*k*$*m*
  m |‘ {*k*∈*dom m*. *P* (*k*, *the* (*m k*))} ≡ *op-map-restrict*$*P*$*m*

  m=*Map.empty* ≡ *op-map-isEmpty*$*m*
  Map.empty=*m* ≡ *op-map-isEmpty*$*m*
  dom m = {} ≡ *op-map-isEmpty*$*m*
  {} = *dom m* ≡ *op-map-isEmpty*$*m*

  ∃ *k v*. *m*=[*k*↦*v*] ≡ *op-map-isSng*$*m*
  ∃ *k v*. [*k*↦*v*]=*m* ≡ *op-map-isSng*$*m*
  ∃ *k*. *dom m*={*k*} ≡ *op-map-isSng*$*m*
  ∃ *k*. {*k*} = *dom m* ≡ *op-map-isSng*$*m*
  1 = *card* (*dom m*) ≡ *op-map-isSng*$*m*

  Ball (*map-to-set m*) *P* ≡ *op-map-ball*$*m*$*P*
  Bex (*map-to-set m*) *P* ≡ *op-map-bex*$*m*$*P*

  card (*dom m*) ≡ *op-map-size*$*m*

  min n (*card* (*dom m*)) ≡ *op-map-size-abort*$*n*$*m*
  min (*card* (*dom m*)) *n* ≡ *op-map-size-abort*$*n*$*m*
  by (*auto*
    intro!: *eq-reflection ext*
    simp: *restrict-map-def dom-eq-singleton-conv card-Suc-eq*
    dest!: *sym*[*of Suc 0* *card* (*dom m*)] *sym*[*of -* *dom m*]
)

lemma [*autoref-itype*]:
op-map-empty $::_i$ ⟨*Ik*,*Iv*⟩$_i$*i-map*
op-map-lookup $::_i$ *Ik* $\rightarrow_i$ ⟨*Ik*,*Iv*⟩$_i$*i-map* $\rightarrow_i$ ⟨*Iv*⟩$_i$*i-option*
op-map-update $::_i$ *Ik* $\rightarrow_i$ *Iv* $\rightarrow_i$ ⟨*Ik*,*Iv*⟩$_i$*i-map* $\rightarrow_i$ ⟨*Ik*,*Iv*⟩$_i$*i-map*
op-map-delete $::_i$ *Ik* $\rightarrow_i$ ⟨*Ik*,*Iv*⟩$_i$*i-map* $\rightarrow_i$ ⟨*Ik*,*Iv*⟩$_i$*i-map*
op-map-restrict
$::_i$ (⟨*Ik*,*Iv*⟩$_i$*i-prod* $\rightarrow_i$ *i-bool*) $\rightarrow_i$ ⟨*Ik*,*Iv*⟩$_i$*i-map* $\rightarrow_i$ ⟨*Ik*,*Iv*⟩$_i$*i-map*
op-map-isEmpty $::_i$ ⟨*Ik*,*Iv*⟩$_i$*i-map* $\rightarrow_i$ *i-bool*
op-map-isSng $::_i$ ⟨*Ik*,*Iv*⟩$_i$*i-map* $\rightarrow_i$ *i-bool*
op-map-ball $::_i$ ⟨*Ik*,*Iv*⟩$_i$*i-map* $\rightarrow_i$ (⟨*Ik*,*Iv*⟩$_i$*i-prod* $\rightarrow_i$ *i-bool*) $\rightarrow_i$ *i-bool*
op-map-bex $::_i$ ⟨*Ik*,*Iv*⟩$_i$*i-map* $\rightarrow_i$ (⟨*Ik*,*Iv*⟩$_i$*i-prod* $\rightarrow_i$ *i-bool*) $\rightarrow_i$ *i-bool*
op-map-size $::_i$ ⟨*Ik*,*Iv*⟩$_i$*i-map* $\rightarrow_i$ *i-nat*
op-map-size-abort $::_i$ *i-nat* $\rightarrow_i$ ⟨*Ik*,*Iv*⟩$_i$*i-map* $\rightarrow_i$ *i-nat*
op ++ $::_i$ ⟨*Ik*,*Iv*⟩$_i$*i-map* $\rightarrow_i$ ⟨*Ik*,*Iv*⟩$_i$*i-map* $\rightarrow_i$ ⟨*Ik*,*Iv*⟩$_i$*i-map*
map-of $::_i$ ⟨⟨*Ik*,*Iv*⟩$_i$*i-prod*⟩$_i$*i-list* $\rightarrow_i$ ⟨*Ik*,*Iv*⟩$_i$*i-map*
by *simp-all*

lemma *hom-map1* [*autoref-hom*]:
CONSTRAINT Map.empty (⟨*Rk*,*Rv*⟩*Rm*)
CONSTRAINT map-of (⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel* → ⟨*Rk*,*Rv*⟩*Rm*)
CONSTRAINT op ++ (⟨*Rk*,*Rv*⟩*Rm* → ⟨*Rk*,*Rv*⟩*Rm* → ⟨*Rk*,*Rv*⟩*Rm*)
by *simp-all*

term *op-map-restrict*
lemma *hom-map2* [*autoref-hom*]:
CONSTRAINT op-map-lookup (*Rk*→⟨*Rk*,*Rv*⟩*Rm*→⟨*Rv*⟩*option-rel*)
CONSTRAINT op-map-update (*Rk*→*Rv*→⟨*Rk*,*Rv*⟩*Rm*→⟨*Rk*,*Rv*⟩*Rm*)
CONSTRAINT op-map-delete (*Rk*→⟨*Rk*,*Rv*⟩*Rm*→⟨*Rk*,*Rv*⟩*Rm*)
CONSTRAINT op-map-restrict ((⟨*Rk*,*Rv*⟩*prod-rel* → *Id*) → ⟨*Rk*,*Rv*⟩*Rm* → ⟨*Rk*,*Rv*⟩*Rm*)
CONSTRAINT op-map-isEmpty (⟨*Rk*,*Rv*⟩*Rm*→*Id*)
CONSTRAINT op-map-isSng (⟨*Rk*,*Rv*⟩*Rm*→*Id*)
CONSTRAINT op-map-ball (⟨*Rk*,*Rv*⟩*Rm*→(⟨*Rk*,*Rv*⟩*prod-rel*→*Id*)→*Id*)
CONSTRAINT op-map-bex (⟨*Rk*,*Rv*⟩*Rm*→(⟨*Rk*,*Rv*⟩*prod-rel*→*Id*)→*Id*)
CONSTRAINT op-map-size (⟨*Rk*,*Rv*⟩*Rm*→*Id*)
CONSTRAINT op-map-size-abort (*Id*→⟨*Rk*,*Rv*⟩*Rm*→*Id*)
by *simp-all*

definition *finite-map-rel R* ≡ *Range R* ⊆ *Collect* (*finite* ∘ *dom*)
lemma *finite-map-rel-trigger*: *finite-map-rel R* ⟹ *finite-map-rel R* **.**

declaration ⟪ *Tagged-Solver.add-triggers*
Relators.relator-props-solver @{*thms finite-map-rel-trigger*} ⟫

end

3.3 Set Interface

theory *Intf-Set*
imports *../../Autoref/Autoref-Bindings-HOL* *../../Monadic/Refine*
begin
consts *i-set* :: *interface* ⇒ *interface*

definition [*simp*]: *op-set-delete x s* ≡ *s* − {*x*}
definition [*simp*]: *op-set-isEmpty s* ≡ *s* = {}
definition [*simp*]: *op-set-isSng s* ≡ *card s* = *1*
definition [*simp*]: *op-set-size-abort m s* ≡ *min m* (*card s*)
definition [*simp*]: *op-set-disjoint a b* ≡ *a*∩*b*={}
definition [*simp*]: *op-set-filter P s* ≡ {*x*∈*s*. *P x*}
definition [*simp*]: *op-set-sel P s* ≡ *SPEC* (λ*x*. *x*∈*s* ∧ *P x*)
definition [*simp*]: *op-set-pick s* ≡ *SPEC* (λ*x*. *x*∈*s*)

lemma [*autoref-op-pat*]:
s − {*x*} ≡ *op-set-delete*$*x*$*s*

s = {} ≡ *op-set-isEmpty*$*s*
{}=*s* ≡ *op-set-isEmpty*$*s*

card s = *1* ≡ *op-set-isSng*$*s*
∃ *x*. *s*={*x*} ≡ *op-set-isSng*$*s*
∃ *x*. {*x*}=*s* ≡ *op-set-isSng*$*s*

min m (*card s*) ≡ *op-set-size-abort*$*m*$*s*
min (*card s*) *m* ≡ *op-set-size-abort*$*m*$*s*

a∩*b*={} ≡ *op-set-disjoint*$*a*$*b*

{*x*∈*s*. *P x*} ≡ *op-set-filter*$*P*$*s*

SPEC (λ*x*. *x*∈*s* ∧ *P x*) ≡ *op-set-sel*$*P*$*s*
SPEC (λ*x*. *P x* ∧ *x*∈*s*) ≡ *op-set-sel*$*P*$*s*

SPEC (λ*x*. *x*∈*s*) ≡ *op-set-pick*$*s*
by (*auto intro*!: *eq-reflection simp*: *card-Suc-eq*)

lemma [*autoref-op-pat*]:
SPEC (λ(*u*,*v*). (*u*,*v*)∈*s*) ≡ *op-set-pick*$*s*
SPEC (λ(*u*,*v*). *P u v* ∧ (*u*,*v*)∈*s*) ≡ *op-set-sel*$(*prod-case P*)$*s*
SPEC (λ(*u*,*v*). (*u*,*v*)∈*s* ∧ *P u v*) ≡ *op-set-sel*$(*prod-case P*)$*s*
by (*auto intro*!: *eq-reflection*)

lemma [*autoref-itype*]:
{} $::_i$ ⟨*I*⟩$_i$*i-set*
insert $::_i$ *I* $\rightarrow_i$ ⟨*I*⟩$_i$*i-set* $\rightarrow_i$ ⟨*I*⟩$_i$*i-set*

$\textit{op-set-delete} ::_i I \rightarrow_i \langle I\rangle_i \textit{i-set} \rightarrow_i \langle I\rangle_i \textit{i-set}$
$\textit{op} \in ::_i I \rightarrow_i \langle I\rangle_i \textit{i-set} \rightarrow_i \textit{i-bool}$
$\textit{op-set-isEmpty} ::_i \langle I\rangle_i \textit{i-set} \rightarrow_i \textit{i-bool}$
$\textit{op-set-isSng} ::_i \langle I\rangle_i \textit{i-set} \rightarrow_i \textit{i-bool}$
$\textit{op} \cup ::_i \langle I\rangle_i \textit{i-set} \rightarrow_i \langle I\rangle_i \textit{i-set} \rightarrow_i \langle I\rangle_i \textit{i-set}$
$\textit{op} \cap ::_i \langle I\rangle_i \textit{i-set} \rightarrow_i \langle I\rangle_i \textit{i-set} \rightarrow_i \langle I\rangle_i \textit{i-set}$
$\textit{op} - ::_i \langle I\rangle_i \textit{i-set} \rightarrow_i \langle I\rangle_i \textit{i-set} \rightarrow_i \langle I\rangle_i \textit{i-set}$
$\textit{op} = ::_i \langle I\rangle_i \textit{i-set} \rightarrow_i \langle I\rangle_i \textit{i-set} \rightarrow_i \textit{i-bool}$
$\textit{op} \subseteq ::_i \langle I\rangle_i \textit{i-set} \rightarrow_i \langle I\rangle_i \textit{i-set} \rightarrow_i \textit{i-bool}$
$\textit{op-set-disjoint} ::_i \langle I\rangle_i \textit{i-set} \rightarrow_i \langle I\rangle_i \textit{i-set} \rightarrow_i \textit{i-bool}$
$\textit{Ball} ::_i \langle I\rangle_i \textit{i-set} \rightarrow_i (I \rightarrow_i \textit{i-bool}) \rightarrow_i \textit{i-bool}$
$\textit{Bex} ::_i \langle I\rangle_i \textit{i-set} \rightarrow_i (I \rightarrow_i \textit{i-bool}) \rightarrow_i \textit{i-bool}$
$\textit{op-set-filter} ::_i (I \rightarrow_i \textit{i-bool}) \rightarrow_i \langle I\rangle_i \textit{i-set} \rightarrow_i \langle I\rangle_i \textit{i-set}$
$\textit{card} ::_i \langle I\rangle_i \textit{i-set} \rightarrow_i \textit{i-nat}$
$\textit{op-set-size-abort} ::_i \textit{i-nat} \rightarrow_i \langle I\rangle_i \textit{i-set} \rightarrow_i \textit{i-nat}$
$\textit{set} ::_i \langle I\rangle_i \textit{i-list} \rightarrow_i \langle I\rangle_i \textit{i-set}$
$\textit{op-set-sel} ::_i (I \rightarrow_i \textit{i-bool}) \rightarrow_i \langle I\rangle_i \textit{i-set} \rightarrow_i \langle I\rangle_i \textit{i-nres}$
$\textit{op-set-pick} ::_i \langle I\rangle_i \textit{i-set} \rightarrow_i \langle I\rangle_i \textit{i-nres}$
$\textit{Sigma} ::_i \langle Ia\rangle_i \textit{i-set} \rightarrow_i (Ia \rightarrow_i \langle Ib\rangle_i \textit{i-set}) \rightarrow_i \langle\langle Ia,Ib\rangle_i \textit{i-prod}\rangle_i \textit{i-set}$
$\textit{op} \text{ ` } ::_i (Ia \rightarrow_i Ib) \rightarrow_i \langle Ia\rangle_i \textit{i-set} \rightarrow_i \langle Ib\rangle_i \textit{i-set}$
by *simp-all*

lemma *hom-set1*[*autoref-hom*]:
$\textit{CONSTRAINT}\ \{\}\ (\langle R\rangle Rs)$
$\textit{CONSTRAINT insert}\ (R \rightarrow \langle R\rangle Rs \rightarrow \langle R\rangle Rs)$
$\textit{CONSTRAINT op} \in (R \rightarrow \langle R\rangle Rs \rightarrow Id)$
$\textit{CONSTRAINT op} \cup (\langle R\rangle Rs \rightarrow \langle R\rangle Rs \rightarrow \langle R\rangle Rs)$
$\textit{CONSTRAINT op} \cap (\langle R\rangle Rs \rightarrow \langle R\rangle Rs \rightarrow \langle R\rangle Rs)$
$\textit{CONSTRAINT op} - (\langle R\rangle Rs \rightarrow \langle R\rangle Rs \rightarrow \langle R\rangle Rs)$
$\textit{CONSTRAINT op} = (\langle R\rangle Rs \rightarrow \langle R\rangle Rs \rightarrow Id)$
$\textit{CONSTRAINT op} \subseteq (\langle R\rangle Rs \rightarrow \langle R\rangle Rs \rightarrow Id)$
$\textit{CONSTRAINT Ball}\ (\langle R\rangle Rs \rightarrow (R \rightarrow Id) \rightarrow Id)$
$\textit{CONSTRAINT Bex}\ (\langle R\rangle Rs \rightarrow (R \rightarrow Id) \rightarrow Id)$
$\textit{CONSTRAINT card}\ (\langle R\rangle Rs \rightarrow Id)$
$\textit{CONSTRAINT set}\ (\langle R\rangle Rl \rightarrow \langle R\rangle Rs)$
$\textit{CONSTRAINT op} \text{ ` } ((Ra \rightarrow Rb) \rightarrow \langle Ra\rangle Rs \rightarrow \langle Rb\rangle Rs)$
by *simp-all*

lemma *hom-set2*[*autoref-hom*]:
$\textit{CONSTRAINT op-set-delete}\ (R \rightarrow \langle R\rangle Rs \rightarrow \langle R\rangle Rs)$
$\textit{CONSTRAINT op-set-isEmpty}\ (\langle R\rangle Rs \rightarrow Id)$
$\textit{CONSTRAINT op-set-isSng}\ (\langle R\rangle Rs \rightarrow Id)$
$\textit{CONSTRAINT op-set-size-abort}\ (Id \rightarrow \langle R\rangle Rs \rightarrow Id)$
$\textit{CONSTRAINT op-set-disjoint}\ (\langle R\rangle Rs \rightarrow \langle R\rangle Rs \rightarrow Id)$
$\textit{CONSTRAINT op-set-filter}\ ((R \rightarrow Id) \rightarrow \langle R\rangle Rs \rightarrow \langle R\rangle Rs)$
$\textit{CONSTRAINT op-set-sel}\ ((R \rightarrow Id) \rightarrow \langle R\rangle Rs \rightarrow \langle R\rangle Rn)$
$\textit{CONSTRAINT op-set-pick}\ (\langle R\rangle Rs \rightarrow \langle R\rangle Rn)$
by *simp-all*

lemma *hom-set-Sigma*[*autoref-hom*]:
CONSTRAINT Sigma (⟨*Ra*⟩*Rs* → (*Ra* → ⟨*Rb*⟩*Rs*) → ⟨⟨*Ra*,*Rb*⟩*prod-rel*⟩*Rs2*)
by *simp-all*

definition *finite-set-rel R* ≡ *Range R* ⊆ *Collect* (*finite*)

lemma *finite-set-rel-trigger*: *finite-set-rel R* ⟹ *finite-set-rel R* **.**

declaration ⟪ *Tagged-Solver.add-triggers*
Relators.relator-props-solver @{*thms finite-set-rel-trigger*} ⟫

end

3.4 Generic Compare Algorithms

theory *Gen-Comp*
imports *../Intf/Intf-Comp* *../../Autoref/Autoref*
begin

3.4.1 Order for Product

lemma *autoref-prod-cmp-dflt-id*[*autoref-rules-raw*]:
(*dflt-cmp op* ≤ *op* <, *dflt-cmp op* ≤ *op* <) ∈
⟨*Id*,*Id*⟩*prod-rel* → ⟨*Id*,*Id*⟩*prod-rel* → *Id*
by *auto*

lemma *gen-prod-cmp-dflt*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP cmp1* (*dflt-cmp op* ≤ *op* <) (*R1* → *R1* → *Id*)
assumes *GEN-OP cmp2* (*dflt-cmp op* ≤ *op* <) (*R2* → *R2* → *Id*)
shows (*cmp-prod cmp1 cmp2*, *dflt-cmp op* ≤ *op* <) ∈
⟨*R1*,*R2*⟩*prod-rel* → ⟨*R1*,*R2*⟩*prod-rel* → *Id*
proof −
have *E*: *dflt-cmp op* ≤ *op* <
= *cmp-prod* (*dflt-cmp op* ≤ *op* <) (*dflt-cmp op* ≤ *op* <)
by (*auto simp*: *dflt-cmp-def prod-less-def prod-le-def intro*!: *ext*)

show *?thesis*
using *assms*
unfolding *autoref-tag-defs E*
by *parametricity*
qed

end

3.5 Iterators

theory *Gen-Iterator*
imports *../../Monadic/Refine* *../Lib/Proper-Iterator*
begin

Iterators are realized by to-list functions followed by folding. A post-optimization step then replaces these constructions by real iterators.

term *it-to-list*
lemma *param-it-to-list*[*param*]: (*it-to-list*,*it-to-list*) ∈
(*Rs* → (*Ra* → *bool-rel*) →
(*Rb* → ⟨*Rb*⟩*list-rel* → ⟨*Rb*⟩*list-rel*) → ⟨*Rc*⟩*list-rel* → *Rd*) → *Rs* → *Rd*
unfolding *it-to-list-def* [*abs-def*]
by *parametricity*

3.5.1 Set iterators

definition *is-set-to-sorted-list-deprecated ordR Rk Rs tsl* ≡ ∀ *s s′*.
(*s*,*s′*)∈⟨*Rk*⟩*Rs* ⟶
(*RETURN* (*tsl s*),*it-to-sorted-list ordR s′*)∈⟨⟨*Rk*⟩*list-rel*⟩*nres-rel*

definition *is-set-to-sorted-list ordR Rk Rs tsl* ≡ ∀ *s s′*.
(*s*,*s′*)∈⟨*Rk*⟩*Rs*
⟶ (∃ *l′*. (*tsl s*,*l′*)∈⟨*Rk*⟩*list-rel*
∧ *RETURN l′* ≤ *it-to-sorted-list ordR s′*)

definition *is-set-to-list* ≡ *is-set-to-sorted-list* (λ- -. *True*)

lemma *is-set-to-sorted-listE*:
assumes *is-set-to-sorted-list ordR Rk Rs tsl*
assumes (*s*,*s′*)∈⟨*Rk*⟩*Rs*
obtains *l′* **where** (*tsl s*,*l′*)∈⟨*Rk*⟩*list-rel*
and *RETURN l′* ≤ *it-to-sorted-list ordR s′*
using *assms* **unfolding** *is-set-to-sorted-list-def* **by** *blast*

lemma *it-to-sorted-list-weaken*:
R≤*R′* ⟹ *it-to-sorted-list R s* ≤ *it-to-sorted-list R′ s*
unfolding *it-to-sorted-list-def*
by (*auto intro*!: *sorted-by-rel-weaken*[**where** *R*=*R*])

lemma *set-to-list-by-set-to-sorted-list*[*autoref-ga-rules*]:
assumes *GEN-ALGO-tag* (*is-set-to-sorted-list ordR Rk Rs tsl*)
shows *is-set-to-list Rk Rs tsl*
using *assms*
unfolding *is-set-to-list-def is-set-to-sorted-list-def autoref-tag-defs*
apply (*safe*)
apply (*drule spec*, *drule spec*, *drule* (*1*) *mp*)

apply (*elim exE conjE*)
apply (*rule exI*, *rule conjI*, *assumption*)
apply (*rule order-trans*, *assumption*)
apply (*rule it-to-sorted-list-weaken*)
by *blast*

definition *det-fold-set R c f σ result* ≡
$\forall$ *l*. *distinct l* ∧ *sorted-by-rel R l* ⟶ *foldli l c f σ* = *result* (*set l*)

lemma *det-fold-setI*[*intro?*]:
assumes ⋀*l*. ⟦*distinct l*; *sorted-by-rel R l*⟧
⟹ *foldli l c f σ* = *result* (*set l*)
shows *det-fold-set R c f σ result*
using *assms* **unfolding** *det-fold-set-def* **by** *auto*

Template lemma for generic algorithm using set iterator

lemma *det-fold-sorted-set*:
assumes *1*: *det-fold-set ordR c' f' σ' result*
assumes *2*: *is-set-to-sorted-list ordR Rk Rs tsl*
assumes *SREF*[*param*]: (*s*,*s'*)∈⟨*Rk*⟩*Rs*
assumes [*param*]: (*c*,*c'*)∈*Rσ*→*Id*
assumes [*param*]: (*f*,*f'*)∈*Rk* → *Rσ* → *Rσ*
assumes [*param*]: (*σ*,*σ'*)∈*Rσ*
shows (*foldli* (*tsl s*) *c f σ*, *result s'*) ∈ *Rσ*
proof −
obtain *tsl'* **where**
[*param*]: (*tsl s*,*tsl'*) ∈ ⟨*Rk*⟩*list-rel*
and *IT*: *RETURN tsl'* ≤ *it-to-sorted-list ordR s'*
using *2 SREF*
by (*rule is-set-to-sorted-listE*)

have (*foldli* (*tsl s*) *c f σ*, *foldli tsl' c' f' σ'*) ∈ *Rσ*
by *parametricity*
also have *foldli tsl' c' f' σ'* = *result s'*
using *1 IT*
unfolding *det-fold-set-def it-to-sorted-list-def*
by *simp*
finally show *?thesis* **.**
qed

lemma *det-fold-set*:
assumes *det-fold-set* (λ- -. *True*) *c' f' σ' result*
assumes *is-set-to-list Rk Rs tsl*
assumes (*s*,*s'*)∈⟨*Rk*⟩*Rs*
assumes (*c*,*c'*)∈*Rσ*→*Id*
assumes (*f*,*f'*)∈*Rk* → *Rσ* → *Rσ*
assumes (*σ*,*σ'*)∈*Rσ*
shows (*foldli* (*tsl s*) *c f σ*, *result s'*) ∈ *Rσ*

using *assms*
unfolding *is-set-to-list-def*
by (*rule det-fold-sorted-set*)

3.5.2 Map iterators

Build relation on keys

definition *key-rel* :: (*'k* ⇒ *'k* ⇒ *bool*) ⇒ (*'k*×*'v*) ⇒ (*'k*×*'v*) ⇒ *bool*
where *key-rel R a b* ≡ *R* (*fst a*) (*fst b*)

definition *is-map-to-sorted-list-deprecated ordR Rk Rv Rm tsl* ≡ ∀ *m m'*.
(*m*,*m'*)∈⟨*Rk*,*Rv*⟩*Rm* ⟶
(*RETURN* (*tsl m*),*it-to-sorted-list* (*key-rel ordR*) (*map-to-set m'*))
∈⟨⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*⟩*nres-rel*

definition *is-map-to-sorted-list ordR Rk Rv Rm tsl* ≡ ∀ *m m'*.
(*m*,*m'*)∈⟨*Rk*,*Rv*⟩*Rm* ⟶ (
∃ *l'*. (*tsl m*,*l'*)∈⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*
∧ *RETURN l'* ≤ *it-to-sorted-list* (*key-rel ordR*) (*map-to-set m'*))

definition *is-map-to-list Rk Rv Rm tsl*
≡ *is-map-to-sorted-list* (λ- -. *True*) *Rk Rv Rm tsl*

lemma *is-map-to-sorted-listE*:
assumes *is-map-to-sorted-list ordR Rk Rv Rm tsl*
assumes (*m*,*m'*)∈⟨*Rk*,*Rv*⟩*Rm*
obtains *l'* **where** (*tsl m*,*l'*)∈⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*
and *RETURN l'* ≤ *it-to-sorted-list* (*key-rel ordR*) (*map-to-set m'*)
using *assms* **unfolding** *is-map-to-sorted-list-def* **by** *blast*

lemma *map-to-list-by-map-to-sorted-list*[*autoref-ga-rules*]:
assumes *GEN-ALGO-tag* (*is-map-to-sorted-list ordR Rk Rv Rm tsl*)
shows *is-map-to-list Rk Rv Rm tsl*
using *assms*
unfolding *is-map-to-list-def is-map-to-sorted-list-def autoref-tag-defs*
apply (*safe*)
apply (*drule spec*, *drule spec*, *drule* (*1*) *mp*)
apply (*elim exE conjE*)
apply (*rule exI*, *rule conjI*, *assumption*)
apply (*rule order-trans*, *assumption*)
apply (*rule it-to-sorted-list-weaken*)
unfolding *key-rel-def*[*abs-def*]
by *blast*

definition *det-fold-map R c f σ result* ≡
∀ *l*. *distinct* (*map fst l*) ∧ *sorted-by-rel* (*key-rel R*) *l*
⟶ *foldli l c f σ* = *result* (*map-of l*)

lemma *det-fold-mapI*[*intro?*]:

assumes $\bigwedge l.$ ⟦*distinct* (*map fst l*); *sorted-by-rel* (*key-rel R*) *l*⟧
  $\Longrightarrow$ *foldli l c f σ* = *result* (*map-of l*)
shows *det-fold-map R c f σ result*
using *assms* **unfolding** *det-fold-map-def* **by** *auto*

lemma *det-fold-map-aux*:
assumes *1*: ⟦*distinct* (*map fst l*); *sorted-by-rel* (*key-rel R*) *l* ⟧
  $\Longrightarrow$ *foldli l c f σ* = *result* (*map-of l*)
assumes *2*: *RETURN l* ≤ *it-to-sorted-list* (*key-rel R*) (*map-to-set m*)
shows *foldli l c f σ* = *result m*
proof −
from *2* **have** *distinct l* **and** *set l* = *map-to-set m*
  and *SORTED*: *sorted-by-rel* (*key-rel R*) *l*
  unfolding *it-to-sorted-list-def* **by** *simp-all*
hence ∀ (*k*,*v*)∈*set l*. ∀ (*k'*,*v'*)∈*set l*. *k*=*k'* ⟶ *v*=*v'*
  apply *simp*
  unfolding *map-to-set-def*
  apply *auto*
  done
with ‹*distinct l*› **have** *DF*: *distinct* (*map fst l*)
  apply (*induct l*)
  apply *simp*
  apply *force*
  done
with ‹*set l* = *map-to-set m*› **have** [*simp*]: *m* = *map-of l*
  by (*metis map-of-map-to-set*)

from *1*[*OF DF SORTED*] **show** *?thesis* **by** *simp*
qed

Template lemma for generic algorithm using map iterator

lemma *det-fold-sorted-map*:
assumes *1*: *det-fold-map ordR c' f' σ' result*
assumes *2*: *is-map-to-sorted-list ordR Rk Rv Rm tsl*
assumes *MREF*[*param*]: (*m*,*m'*)∈⟨*Rk*,*Rv*⟩*Rm*
assumes [*param*]: (*c*,*c'*)∈*Rσ*→*Id*
assumes [*param*]: (*f*,*f'*)∈⟨*Rk*,*Rv*⟩*prod-rel* → *Rσ* → *Rσ*
assumes [*param*]: (*σ*,*σ'*)∈*Rσ*
shows (*foldli* (*tsl m*) *c f σ*, *result m'*) ∈ *Rσ*
proof −
obtain *tsl'* **where**
  [*param*]: (*tsl m*,*tsl'*) ∈ ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*
  and *IT*: *RETURN tsl'* ≤ *it-to-sorted-list* (*key-rel ordR*) (*map-to-set m'*)
  using *2 MREF* **by** (*rule is-map-to-sorted-listE*)

have (*foldli* (*tsl m*) *c f σ*, *foldli tsl' c' f' σ'*) ∈ *Rσ*
  by *parametricity*
also have *foldli tsl' c' f' σ'* = *result m'*
  using *det-fold-map-aux*[*of tsl' ordR c' f' σ' result*] *1 IT*

```
      unfolding det-fold-map-def
      by clarsimp
    finally show ?thesis .
  qed

  lemma det-fold-map:
    assumes det-fold-map (λ- -. True) c' f' σ' result
    assumes is-map-to-list Rk Rv Rm tsl
    assumes (m,m')∈⟨Rk,Rv⟩Rm
    assumes (c,c')∈Rσ→Id
    assumes (f,f')∈⟨Rk,Rv⟩prod-rel → Rσ → Rσ
    assumes (σ,σ')∈Rσ
    shows (foldli (tsl m) c f σ, result m') ∈ Rσ
    using assms
    unfolding is-map-to-list-def
    by (rule det-fold-sorted-map)

lemma it-to-sorted-list-by-tsl[autoref-rules]:
  assumes MINOR-PRIO-TAG −11
  assumes SV: PREFER single-valued Rk
  assumes TSL: SIDE-GEN-ALGO (is-set-to-sorted-list R Rk Rs tsl)
  shows (λs. RETURN (tsl s), it-to-sorted-list R)
    ∈ ⟨Rk⟩Rs → ⟨⟨Rk⟩list-rel⟩nres-rel
proof (intro fun-relI nres-relI)
  fix s s'
  assume (s,s')∈⟨Rk⟩Rs
  with TSL obtain l' where
    R1: (tsl s, l') ∈ ⟨Rk⟩list-rel and R2: RETURN l' ≤ it-to-sorted-list R s'
    unfolding is-set-to-sorted-list-def autoref-tag-defs
    by blast

  have RETURN (tsl s) ≤ ⇓(⟨Rk⟩list-rel) (RETURN l')
    apply (rule RETURN-refine-sv)
    using SV unfolding autoref-tag-defs apply tagged-solver
    by fact
  also note R2
  finally show RETURN (tsl s) ≤ ⇓ (⟨Rk⟩list-rel) (it-to-sorted-list R s') .
qed

lemma it-to-list-by-tsl[autoref-rules]:
  assumes MINOR-PRIO-TAG −10
  assumes SV: PREFER single-valued Rk
  assumes TSL: SIDE-GEN-ALGO (is-set-to-list Rk Rs tsl)
  shows (λs. RETURN (tsl s), it-to-sorted-list (λ- -. True))
    ∈ ⟨Rk⟩Rs → ⟨⟨Rk⟩list-rel⟩nres-rel
  using assms(2−) unfolding is-set-to-list-def
  by (rule it-to-sorted-list-by-tsl[OF PRIO-TAGI])

lemma dres-it-FOREACH-it-simp[iterator-simps]:
```

dres-it-FOREACH (*λs. dRETURN* (*i s*)) *s c f σ*
  = *foldli* (*i s*) (*dres-case False False c*) (*λx s. s* ≫= *f x*) (*dRETURN σ*)
 unfolding *dres-it-FOREACH-def*
 by *simp*

Locale to be interpreted for proper iterators. TODO/FIXME: * Integrate mono-prover properly into solver-infrastructure, i.e. tag a mono-goal. * Tag iterators, such that, for the mono-prover, we can just convert a proper iterator back to its foldli-equivalent!

lemma *proper-it-mono-dres-pair*:
 assumes *PR*: *proper-it′ it it′*
 assumes *A*: ⋀*k v x. f k v x* ≤ *f′ k v x*
 shows
  it′ s (*dres-case False False c*) (*λ*(*k*,*v*) *s. s* ≫= *f k v*) *σ*
  ≤ *it′ s* (*dres-case False False c*) (*λ*(*k*,*v*) *s. s* ≫= *f′ k v*) *σ* (**is** *?a* ≤ *?b*)
proof −
 from *proper-itE*[*OF PR*[*THEN proper-it′D*]] **obtain** *l* **where**
  A-FMT:
   ?a = *foldli l* (*dres-case False False c*) (*λ*(*k*,*v*) *s. s* ≫= *f k v*) *σ*
    (**is** - = *?a′*)
  and *B-FMT*:
   ?b = *foldli l* (*dres-case False False c*) (*λ*(*k*,*v*) *s. s* ≫= *f′ k v*) *σ*
    (**is** - = *?b′*)
  by *metis*

 from *A* **have** *A′*: ⋀*kv x. prod-case f kv x* ≤ *prod-case f′ kv x*
  by *auto*

 note *A-FMT*
 also have
  ?a′ = *foldli l* (*dres-case False False c*) (*λkv s. s* ≫= *prod-case f kv*) *σ*
  apply (*fo-rule fun-cong*)
  apply (*fo-rule arg-cong*)
  by *auto*
 also note *foldli-mono-dres*[*OF A′*]
 also have
  foldli l (*dres-case False False c*) (*λkv s. s* ≫= *prod-case f′ kv*) *σ* = *?b′*
  apply (*fo-rule fun-cong*)
  apply (*fo-rule arg-cong*)
  by *auto*
 also note *B-FMT*[*symmetric*]
 finally show *?thesis* **.**
qed

lemma *proper-it-mono-dres*:
 assumes *PR*: *proper-it′ it it′*
 assumes *A*: ⋀*kv x. f kv x* ≤ *f′ kv x*
 shows
  it′ s (*dres-case False False c*) (*λkv s. s* ≫= *f kv*) *σ*

$\leq$ *it′ s* (*dres-case False False c*) (λ*kv s. s* $\gg=$ *f′ kv*) σ
apply (*rule proper-itE*[*OF PR*[*THEN proper-it′D*[**where** *s=s*]]])
apply (*erule-tac t=it′ s* **in** *ssubst*)
apply (*rule foldli-mono-dres*[*OF A*])
done

lemma *pi′-dom*[*icf-proper-iteratorI*]: *proper-it′ it it′*
$\Longrightarrow$ *proper-it′* (*map-iterator-dom o it*) (*map-iterator-dom o it′*)
apply (*rule proper-it′I*)
apply (*simp add*: *comp-def*)
apply (*rule icf-proper-iteratorI*)
apply (*erule proper-it′D*)
done

lemma *proper-it-mono-dres-dom*:
assumes *PR*: *proper-it′ it it′*
assumes *A*: $\bigwedge$*kv x. f kv x* $\leq$ *f′ kv x*
shows
(*map-iterator-dom o it′*) *s* (*dres-case False False c*) (λ*kv s. s* $\gg=$ *f kv*) σ
$\leq$
(*map-iterator-dom o it′*) *s* (*dres-case False False c*) (λ*kv s. s* $\gg=$ *f′ kv*) σ

apply (*rule proper-it-mono-dres*)
apply (*rule icf-proper-iteratorI*)
by *fact+*

lemmas *proper-it-monos* =
proper-it-mono-dres-pair proper-it-mono-dres proper-it-mono-dres-dom

attribute-setup *proper-it* = ⟪
Scan.succeed (*Thm.declaration-attribute* (*fn thm => fn context =>*
let
val mono-thms = map-filter (*try* (*curry op RS thm*)) @{*thms proper-it-monos*}
(**val mono-thms = map* (*fn mt => thm RS mt*) @{*thms proper-it-monos*}*)
val context = fold Refine-Misc.refine-mono.add-thm mono-thms context
in
context
end
))
⟫
Proper iterator declaration

end

3.6 Generic Map Algorithms

```
theory Gen-Map
imports ../Intf/Intf-Map   Gen-Iterator
begin

  lemma foldli-add: det-fold-map X
    (λ-. True) (λ(k,v) m. op-map-update k v m) m (op ++ m)
  proof
    case (goal1 l) thus ?case
      apply (induct l arbitrary: m)
      apply (auto simp: map-of-distinct-upd[symmetric])
      done
  qed

  definition gen-add
    :: ('s2 ⇒ -) ⇒ ('k ⇒ 'v ⇒ 's1 ⇒ 's1) ⇒ 's1 ⇒ 's2 ⇒ 's1
    where
    gen-add it upd A B ≡ it B (λ-. True) (λ(k,v) m. upd k v m) A

  lemma gen-add[autoref-rules-raw]:
    assumes PRIO-TAG-GEN-ALGO
   assumes UPD: GEN-OP ins op-map-update (Rk→Rv→⟨Rk,Rv⟩Rs1→⟨Rk,Rv⟩Rs1)
    assumes IT: SIDE-GEN-ALGO (is-map-to-list Rk Rv Rs2 tsl)
    shows (gen-add (foldli o tsl) ins,op ++)
      ∈ (⟨Rk,Rv⟩Rs1) → (⟨Rk,Rv⟩Rs2) → (⟨Rk,Rv⟩Rs1)
    apply (intro fun-relI)
    unfolding gen-add-def comp-def
    apply (rule det-fold-map[OF foldli-add IT[unfolded autoref-tag-defs]])
    apply (parametricity add: UPD[unfolded autoref-tag-defs])+
    done

  lemma foldli-restrict: det-fold-map X (λ-. True)
    (λ(k,v) m. if P (k,v) then op-map-update k v m else m) Map.empty
      (op-map-restrict P ) (is det-fold-map - - ?f - -)
  proof −
    {
      fix l m
      have distinct (map fst l) ⟹
        foldli l (λ-. True) ?f m = m ++ op-map-restrict P (map-of l)
      proof (induction l arbitrary: m)
        case Nil thus ?case by simp
      next
        case (Cons kv l)
        obtain k v where [simp]: kv = (k,v) by fastforce
        from Cons.prems have
          DL: distinct (map fst l) and KNI: k ∉ set (map fst l)
          by auto
```

```
      show ?case proof (cases P (k,v))
        case True[simp]
        have foldli (kv#l) (λ-. True) ?f m = foldli l (λ-. True) ?f (m(k↦v))
          by simp
        also from Cons.IH[OF DL] have
          ... = m(k↦v) ++ op-map-restrict P (map-of l) .
        also have ... = m ++ op-map-restrict P (map-of (kv#l))
          using KNI
          by (auto
            split: option.splits
            intro!: ext
            simp: Map.restrict-map-def Map.map-add-def
            simp: map-of-eq-None-iff [symmetric])
        finally show ?thesis .
      next
        case False[simp]
        have foldli (kv#l) (λ-. True) ?f m = foldli l (λ-. True) ?f m
          by simp
        also from Cons.IH[OF DL] have
          ... = m ++ op-map-restrict P (map-of l) .
        also have ... = m ++ op-map-restrict P (map-of (kv#l))
          using KNI
          by (auto
            intro!: ext
            simp: Map.restrict-map-def Map.map-add-def
            simp: map-of-eq-None-iff [symmetric]
          )
        finally show ?thesis .
      qed
    qed
  }
  from this[of - Map.empty] show ?thesis
    by (auto intro!: det-fold-mapI)
qed

definition gen-restrict :: ('s1 ⇒ -) ⇒ -
  where gen-restrict it upd emp P m
  ≡ it m (λ-. True) (λ(k,v) m. if P (k,v) then upd k v m else m) emp

lemma gen-restrict[autoref-rules-raw]:
  assumes PRIO-TAG-GEN-ALGO
  assumes IT: SIDE-GEN-ALGO (is-map-to-list Rk Rv Rs1 tsl)
  assumes INS:
    GEN-OP upd op-map-update (Rk→Rv→⟨Rk,Rv⟩Rs2→⟨Rk,Rv⟩Rs2)
  assumes EMPTY:
    GEN-OP emp Map.empty (⟨Rk,Rv⟩Rs2)
  shows (gen-restrict (foldli o tsl) upd emp,op-map-restrict)
  ∈ (⟨Rk,Rv⟩prod-rel → Id) → (⟨Rk,Rv⟩Rs1) → (⟨Rk,Rv⟩Rs2)
  apply (intro fun-relI)
```

```
  unfolding gen-restrict-def comp-def
  apply (rule det-fold-map[OF foldli-restrict IT[unfolded autoref-tag-defs]])
  using INS EMPTY unfolding autoref-tag-defs
  apply (parametricity)+
  done

lemma fold-map-of:
  fold (λ(k,v) s. op-map-update k v s) (rev l) Map.empty = map-of l
proof −
  {
    fix m
    have fold (λ(k,v) s. s(k↦v)) (rev l) m = m ++ map-of l
      apply (induct l arbitrary: m)
      apply auto
      done
  } thus ?thesis by simp
qed

definition gen-map-of :: 'm ⇒ ('k⇒'v⇒'m⇒'m) ⇒ - where
  gen-map-of emp upd l ≡ fold (λ(k,v) s. upd k v s) (rev l) emp


lemma gen-map-of[autoref-rules-raw]:
  assumes PRIO-TAG-GEN-ALGO
 assumes UPD: GEN-OP upd op-map-update (Rk→Rv→⟨Rk,Rv⟩Rm→⟨Rk,Rv⟩Rm)
  assumes EMPTY: GEN-OP emp Map.empty (⟨Rk,Rv⟩Rm)
 shows (gen-map-of emp upd,map-of) ∈ ⟨⟨Rk,Rv⟩prod-rel⟩list-rel → ⟨Rk,Rv⟩Rm
  using assms
  apply (intro fun-relI)
  unfolding gen-map-of-def[abs-def]
  unfolding autoref-tag-defs
  apply (subst fold-map-of[symmetric])
  apply parametricity
  done

lemma foldli-ball-aux:
  distinct (map fst l) ⟹ foldli l (λx. x) (λx -. P x) b
  ⟷ b ∧ op-map-ball (map-of l) P
  apply (induct l arbitrary: b)
  apply simp
  apply (force simp: map-to-set-map-of image-def)
  done

lemma foldli-ball:
  det-fold-map X (λx. x) (λx -. P x) True (λm. op-map-ball m P)
  apply rule
  using foldli-ball-aux[where b=True] by auto

definition gen-ball :: ('m ⇒ -) ⇒ - where
```

gen-ball it m P ≡ *it m* (λ*x*. *x*) (λ*x* -. *P x*) *True*

lemma *gen-ball*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-map-to-list Rk Rv Rm tsl*)
shows (*gen-ball* (*foldli o tsl*),*op-map-ball*)
∈ ⟨*Rk*,*Rv*⟩*Rm* → (⟨*Rk*,*Rv*⟩*prod-rel* → *Id*) → *Id*
apply (*intro fun-relI*)
unfolding *gen-ball-def comp-def*
apply (*rule det-fold-map*[*OF foldli-ball IT*[*unfolded autoref-tag-defs*]])
apply (*parametricity*)+
done

lemma *foldli-bex-aux*:
distinct (*map fst l*) ⟹ *foldli l* (λ*x*. ¬*x*) (λ*x* -. *P x*) *b*
⟷ *b* ∨ *op-map-bex* (*map-of l*) *P*
apply (*induct l arbitrary*: *b*)
apply *simp*
apply (*force simp*: *map-to-set-map-of image-def*)
done

lemma *foldli-bex*:
det-fold-map X (λ*x*. ¬*x*) (λ*x* -. *P x*) *False* (λ*m*. *op-map-bex m P*)
apply *rule*
using *foldli-bex-aux*[**where** *b*=*False*] **by** *auto*

definition *gen-bex* :: (′*m* ⇒ -) ⇒ - **where**
gen-bex it m P ≡ *it m* (λ*x*. ¬*x*) (λ*x* -. *P x*) *False*

lemma *gen-bex*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-map-to-list Rk Rv Rm tsl*)
shows (*gen-bex* (*foldli o tsl*),*op-map-bex*)
∈ ⟨*Rk*,*Rv*⟩*Rm* → (⟨*Rk*,*Rv*⟩*prod-rel* → *Id*) → *Id*
apply (*intro fun-relI*)
unfolding *gen-bex-def comp-def*
apply (*rule det-fold-map*[*OF foldli-bex IT*[*unfolded autoref-tag-defs*]])
apply (*parametricity*)+
done

lemma *ball-isEmpty*: *op-map-isEmpty m* = *op-map-ball m* (λ-. *False*)
apply (*auto intro*!: *ext*)
by (*metis map-to-set-simps*(*7*) *option.exhaust*)

definition *gen-isEmpty ball m* ≡ *ball m* (λ-. *False*)

lemma *gen-isEmpty*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *BALL*:

GEN-OP ball op-map-ball (⟨*Rk*,*Rv*⟩*Rm*→(⟨*Rk*,*Rv*⟩*prod-rel*→*Id*) → *Id*)
shows (*gen-isEmpty ball*,*op-map-isEmpty*)
∈ ⟨*Rk*,*Rv*⟩*Rm* → *Id*
apply (*intro fun-relI*)
unfolding *gen-isEmpty-def* **using** *assms*
unfolding *autoref-tag-defs*
apply −
apply (*subst ball-isEmpty*)
apply *parametricity*+
done

lemma *foldli-size-aux*: *distinct* (*map fst l*)
⟹ *foldli l* (λ-. *True*) (λ- *n*. *Suc n*) *n* = *n* + *op-map-size* (*map-of l*)
apply (*induct l arbitrary*: *n*)
apply (*auto simp*: *dom-map-of-conv-image-fst*)
done

lemma *foldli-size*: *det-fold-map X* (λ-. *True*) (λ- *n*. *Suc n*) *0 op-map-size*
apply *rule*
using *foldli-size-aux*[**where** *n*=*0*] **by** *simp*

definition *gen-size* :: (′*m* ⇒ -) ⇒ -
where *gen-size it m* ≡ *it m* (λ-. *True*) (λ- *n*. *Suc n*) *0*

lemma *gen-size*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-map-to-list Rk Rv Rm tsl*)
shows (*gen-size* (*foldli o tsl*),*op-map-size*) ∈ ⟨*Rk*,*Rv*⟩*Rm* → *Id*
apply (*intro fun-relI*)
unfolding *gen-size-def comp-def*
apply (*rule det-fold-map*[*OF foldli-size IT*[*unfolded autoref-tag-defs*]])
apply (*parametricity*)+
done

lemma *foldli-size-abort-aux*:
⟦*n0*≤*m*; *distinct* (*map fst l*)⟧ ⟹
foldli l (λ*n*. *n*<*m*) (λ- *n*. *Suc n*) *n0* = *min m* (*n0* + *card* (*dom* (*map-of l*)))
apply (*induct l arbitrary*: *n0*)
apply (*auto simp*: *dom-map-of-conv-image-fst*)
done

lemma *foldli-size-abort*:
det-fold-map X (λ*n*. *n*<*m*) (λ- *n*. *Suc n*) *0* (*op-map-size-abort m*)
apply *rule*
using *foldli-size-abort-aux*[**where** *?n0.0*=*0*]
by *simp*

definition *gen-size-abort* :: (′*s* ⇒ -) ⇒ - **where**
gen-size-abort it m s ≡ *it s* (λ*n*. *n*<*m*) (λ- *n*. *Suc n*) *0*

lemma *gen-size-abort*[*autoref-rules-raw*]:
  assumes *PRIO-TAG-GEN-ALGO*
  assumes *IT*: *SIDE-GEN-ALGO* (*is-map-to-list Rk Rv Rm tsl*)
  shows (*gen-size-abort* (*foldli o tsl*),*op-map-size-abort*)
    $\in$ *Id* $\rightarrow$ $\langle$*Rk*,*Rv*$\rangle$*Rm* $\rightarrow$ *Id*
  apply (*intro fun-relI*)
  unfolding *gen-size-abort-def comp-def*
  apply (*rule det-fold-map*[*OF foldli-size-abort*
    IT[*unfolded autoref-tag-defs*]])
  apply (*parametricity*)+
  done

lemma *size-abort-isSng*: *op-map-isSng s* $\longleftrightarrow$ *op-map-size-abort 2 s* = *1*
  by (*auto simp*: *dom-eq-singleton-conv min-def dest*!: *card-eq-SucD*)

definition *gen-isSng* :: (*nat* $\Rightarrow$ *'s* $\Rightarrow$ *nat*) $\Rightarrow$ - **where**
  gen-isSng sizea s $\equiv$ *sizea 2 s* = *1*

lemma *gen-isSng*[*autoref-rules-raw*]:
  assumes *PRIO-TAG-GEN-ALGO*
  assumes *GEN-OP sizea op-map-size-abort* (*Id* $\rightarrow$ ($\langle$*Rk*,*Rv*$\rangle$*Rm*) $\rightarrow$ *Id*)
  shows (*gen-isSng sizea*,*op-map-isSng*)
  $\in$ $\langle$*Rk*,*Rv*$\rangle$*Rm* $\rightarrow$ *Id*
  apply (*intro fun-relI*)
  unfolding *gen-isSng-def* **using** *assms*
  unfolding *autoref-tag-defs*
  apply $-$
  apply (*subst size-abort-isSng*)
  apply *parametricity*
  done

end

3.7 Generic Set Algorithms

theory *Gen-Set*
imports ../*Intf*/*Intf-Set Gen-Iterator*
begin

lemma *foldli-union*: *det-fold-set X* (λ-. *True*) *insert a* (*op* $\cup$ *a*)
proof
  case (*goal1 l*) **thus** *?case*
    by (*induct l arbitrary*: *a*) *auto*
qed

definition *gen-union*
  :: - $\Rightarrow$ (*'k* $\Rightarrow$ *'s2* $\Rightarrow$ *'s2*)

⇒ 's1 ⇒ 's2 ⇒ 's2
where
gen-union it ins A B ≡ *it A* (λ-. *True*) *ins B*

lemma *gen-union*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *INS*: *GEN-OP ins Set.insert* (*Rk*→⟨*Rk*⟩*Rs2*→⟨*Rk*⟩*Rs2*)
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs1 tsl*)
shows (*gen-union* (λ*x*. *foldli* (*tsl x*)) *ins*,*op* ∪)
∈ (⟨*Rk*⟩*Rs1*) → (⟨*Rk*⟩*Rs2*) → (⟨*Rk*⟩*Rs2*)
apply (*intro fun-relI*)
apply (*subst Un-commute*)
unfolding *gen-union-def*
apply (*rule det-fold-set*[*OF*
foldli-union IT[*unfolded autoref-tag-defs*]])
using *INS*
unfolding *autoref-tag-defs*
apply (*parametricity*)+
done

lemma *foldli-inter*: *det-fold-set X* (λ-. *True*)
(λ*x s*. *if x*∈*a then insert x s else s*) {} (λ*s*. *s*∩*a*)
(**is** *det-fold-set* - - *?f* - -)
proof −
{
fix *l s0*
have *foldli l* (λ-. *True*)
(λ*x s*. *if x*∈*a then insert x s else s*) *s0* = *s0* ∪ (*set l* ∩ *a*)
by (*induct l arbitrary*: *s0*) *auto*
}
from *this*[*of* - {}] **show** *?thesis* **apply** − **by** *rule simp*
qed

definition *gen-inter* :: - ⇒
('*k* ⇒ '*s2* ⇒ *bool*) ⇒ -
where *gen-inter it1 memb2 ins3 empty3 s1 s2*
≡ *it1 s1* (λ-. *True*)
(λ*x s*. *if memb2 x s2 then ins3 x s else s*) *empty3*

lemma *gen-inter*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs1 tsl*)
assumes *MEMB*:
GEN-OP memb2 op ∈ (*Rk* → ⟨*Rk*⟩*Rs2* → *Id*)
assumes *INS*:
GEN-OP ins3 Set.insert (*Rk*→⟨*Rk*⟩*Rs3*→⟨*Rk*⟩*Rs3*)
assumes *EMPTY*:
GEN-OP empty3 {} (⟨*Rk*⟩*Rs3*)
shows (*gen-inter* (λ*x*. *foldli* (*tsl x*)) *memb2 ins3 empty3*,*op* ∩)

∈ (⟨*Rk*⟩*Rs1*) → (⟨*Rk*⟩*Rs2*) → (⟨*Rk*⟩*Rs3*)
apply (*intro fun-relI*)
unfolding *gen-inter-def*
apply (*rule det-fold-set*[*OF foldli-inter IT*[*unfolded autoref-tag-defs*]])
using *MEMB INS EMPTY*
unfolding *autoref-tag-defs*
apply (*parametricity*)+
done

lemma *foldli-diff*:
det-fold-set X (λ-. *True*) (λ*x s*. *op-set-delete x s*) *s* (*op* − *s*)
proof
case (*goal1 l*) **thus** *?case*
by (*induct l arbitrary*: *s*) *auto*
qed

definition *gen-diff* :: (′*k*⇒′*s1*⇒′*s1*) ⇒ - ⇒ ′*s2* ⇒ -
where *gen-diff del1 it2 s1 s2*
≡ *it2 s2* (λ-. *True*) (λ*x s*. *del1 x s*) *s1*

lemma *gen-diff*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *DEL*:
GEN-OP del1 op-set-delete (*Rk* → ⟨*Rk*⟩*Rs1* → ⟨*Rk*⟩*Rs1*)
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs2 it2*)
shows (*gen-diff del1* (λ*x*. *foldli* (*it2 x*)),*op* −)
∈ (⟨*Rk*⟩*Rs1*) → (⟨*Rk*⟩*Rs2*) → (⟨*Rk*⟩*Rs1*)
apply (*intro fun-relI*)
unfolding *gen-diff-def*
apply (*rule det-fold-set*[*OF foldli-diff IT*[*unfolded autoref-tag-defs*]])
using *DEL*
unfolding *autoref-tag-defs*
apply (*parametricity*)+
done

lemma *foldli-ball-aux*:
foldli l (λ*x*. *x*) (λ*x* -. *P x*) *b* ⟷ *b* ∧ *Ball* (*set l*) *P*
by (*induct l arbitrary*: *b*) *auto*

lemma *foldli-ball*: *det-fold-set X* (λ*x*. *x*) (λ*x* -. *P x*) *True* (λ*s*. *Ball s P*)
apply *rule* **using** *foldli-ball-aux*[**where** *b*=*True*] **by** *simp*

definition *gen-ball* :: - ⇒ ′*s* ⇒ (′*k* ⇒ *bool*) ⇒ -
where *gen-ball it s P* ≡ *it s* (λ*x*. *x*) (λ*x* -. *P x*) *True*

lemma *gen-ball*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs it*)
shows (*gen-ball* (λ*x*. *foldli* (*it x*)),*Ball*) ∈ ⟨*Rk*⟩*Rs* → (*Rk*→*Id*) → *Id*

apply (*intro fun-relI*)
unfolding *gen-ball-def*
apply (*rule det-fold-set*[*OF foldli-ball IT*[*unfolded autoref-tag-defs*]])
apply (*parametricity*)+
done

lemma *foldli-bex-aux*: *foldli l* ($\lambda x.\ \neg x$) (λx -. *P x*) *b* $\longleftrightarrow$ *b* $\vee$ *Bex* (*set l*) *P*
by (*induct l arbitrary*: *b*) *auto*

lemma *foldli-bex*: *det-fold-set X* ($\lambda x.\ \neg x$) (λx -. *P x*) *False* ($\lambda s.$ *Bex s P*)
apply *rule* **using** *foldli-bex-aux*[**where** *b*=*False*] **by** *simp*

definition *gen-bex* :: - $\Rightarrow$ *'s* $\Rightarrow$ (*'k* $\Rightarrow$ *bool*) $\Rightarrow$ -
where *gen-bex it s P* $\equiv$ *it s* ($\lambda x.\ \neg x$) (λx -. *P x*) *False*

lemma *gen-bex*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs it*)
shows (*gen-bex* ($\lambda x.$ *foldli* (*it x*)),*Bex*) $\in$ $\langle Rk\rangle Rs$ $\rightarrow$ ($Rk \rightarrow Id$) $\rightarrow$ *Id*
apply (*intro fun-relI*)
unfolding *gen-bex-def*
apply (*rule det-fold-set*[*OF foldli-bex IT*[*unfolded autoref-tag-defs*]])
apply (*parametricity*)+
done

lemma *ball-subseteq*:
(*Ball s1* ($\lambda x.\ x \in s2$)) $\longleftrightarrow$ *s1* $\subseteq$ *s2*
by *blast*

definition *gen-subseteq*
:: (*'s1* $\Rightarrow$ (*'k* $\Rightarrow$ *bool*) $\Rightarrow$ *bool*) $\Rightarrow$ (*'k* $\Rightarrow$ *'s2* $\Rightarrow$ *bool*) $\Rightarrow$ -
where *gen-subseteq ball1 mem2 s1 s2* $\equiv$ *ball1 s1* ($\lambda x.$ *mem2 x s2*)

lemma *gen-subseteq*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP ball1 Ball* ($\langle Rk\rangle Rs1$ $\rightarrow$ ($Rk \rightarrow Id$) $\rightarrow$ *Id*)
assumes *GEN-OP mem2 op* $\in$ (*Rk* $\rightarrow$ $\langle Rk\rangle Rs2$ $\rightarrow$ *Id*)
shows (*gen-subseteq ball1 mem2*,*op* $\subseteq$) $\in$ $\langle Rk\rangle Rs1$ $\rightarrow$ $\langle Rk\rangle Rs2$ $\rightarrow$ *Id*
apply (*intro fun-relI*)
unfolding *gen-subseteq-def* **using** *assms*
unfolding *autoref-tag-defs*
apply −
apply (*subst ball-subseteq*[*symmetric*])
apply *parametricity*
done

definition *gen-equal ss1 ss2 s1 s2* $\equiv$ *ss1 s1 s2* $\wedge$ *ss2 s2 s1*

lemma *gen-equal*[*autoref-rules-raw*]:

assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP ss1 op* $\subseteq$ ($\langle$*Rk*$\rangle$*Rs1* $\rightarrow$ $\langle$*Rk*$\rangle$*Rs2* $\rightarrow$ *Id*)
assumes *GEN-OP ss2 op* $\subseteq$ ($\langle$*Rk*$\rangle$*Rs2* $\rightarrow$ $\langle$*Rk*$\rangle$*Rs1* $\rightarrow$ *Id*)
shows (*gen-equal ss1 ss2*, *op* =) $\in$ $\langle$*Rk*$\rangle$*Rs1* $\rightarrow$ $\langle$*Rk*$\rangle$*Rs2* $\rightarrow$ *Id*
apply (*intro fun-relI*)
unfolding *gen-equal-def* **using** *assms*
unfolding *autoref-tag-defs*
apply −
apply (*subst set-eq-subset*)
apply (*parametricity*)
done

lemma *foldli-card-aux*: *distinct l* $\Longrightarrow$ *foldli l* (λ-. *True*)
(λ- *n*. *Suc n*) *n* = *n* + *card* (*set l*)
apply (*induct l arbitrary*: *n*)
apply *auto*
done

lemma *foldli-card*: *det-fold-set X* (λ-. *True*) (λ- *n*. *Suc n*) *0 card*
apply *rule* **using** *foldli-card-aux*[**where** *n=0*] **by** *simp*

definition *gen-card* **where**
gen-card it s $\equiv$ *it s* (λ*x*. *True*) (λ- *n*. *Suc n*) *0*

lemma *gen-card*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs it*)
shows (*gen-card* (λ*x*. *foldli* (*it x*)),*card*) $\in$ $\langle$*Rk*$\rangle$*Rs* $\rightarrow$ *Id*
apply (*intro fun-relI*)
unfolding *gen-card-def*
apply (*rule det-fold-set*[*OF foldli-card IT*[*unfolded autoref-tag-defs*]])
apply (*parametricity*)+
done

lemma *fold-set*: *fold Set.insert l s* = *s* $\cup$ *set l*
by (*induct l arbitrary*: *s*) *auto*

definition *gen-set* :: *'s* $\Rightarrow$ (*'k* $\Rightarrow$ *'s* $\Rightarrow$ *'s*) $\Rightarrow$ - **where**
gen-set emp ins l = *fold ins l emp*

lemma *gen-set*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *EMPTY*:
GEN-OP emp {} ($\langle$*Rk*$\rangle$*Rs*)
assumes *INS*:
GEN-OP ins Set.insert (*Rk*$\rightarrow$$\langle$*Rk*$\rangle$*Rs*$\rightarrow$$\langle$*Rk*$\rangle$*Rs*)
shows (*gen-set emp ins*,*set*)$\in$$\langle$*Rk*$\rangle$*list-rel*$\rightarrow$$\langle$*Rk*$\rangle$*Rs*
apply (*intro fun-relI*)
unfolding *gen-set-def* **using** *assms*

unfolding *autoref-tag-defs*
apply −
apply (*subst fold-set*[**where** *s*={},*simplified*,*symmetric*])
apply *parametricity*
done

lemma *ball-isEmpty*: *op-set-isEmpty s* = (∀ *x*∈*s*. *False*)
by *auto*

definition *gen-isEmpty* :: (′*s* ⇒ (′*k* ⇒ *bool*) ⇒ *bool*) ⇒ ′*s* ⇒ *bool* **where**
gen-isEmpty ball s ≡ *ball s* (λ-. *False*)

lemma *gen-isEmpty*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP ball Ball* (⟨*Rk*⟩*Rs* → (*Rk*→*Id*) → *Id*)
shows (*gen-isEmpty ball*,*op-set-isEmpty*) ∈ ⟨*Rk*⟩*Rs* → *Id*
apply (*intro fun-relI*)
unfolding *gen-isEmpty-def* **using** *assms*
unfolding *autoref-tag-defs*
apply −
apply (*subst ball-isEmpty*)
apply *parametricity*
done

lemma *foldli-size-abort-aux*:
⟦*n0*≤*m*; *distinct l*⟧ ⟹
foldli l (λ*n*. *n*<*m*) (λ- *n*. *Suc n*) *n0* = *min m* (*n0* + *card* (*set l*))
apply (*induct l arbitrary*: *n0*)
apply *auto*
done

lemma *foldli-size-abort*:
det-fold-set X (λ*n*. *n*<*m*) (λ- *n*. *Suc n*) *0* (*op-set-size-abort m*)
apply *rule*
using *foldli-size-abort-aux*[**where** *?n0.0*=*0*]
by *simp*

definition *gen-size-abort* **where**
gen-size-abort it m s ≡ *it s* (λ*n*. *n*<*m*) (λ- *n*. *Suc n*) *0*

lemma *gen-size-abort*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs it*)
shows (*gen-size-abort* (λ*x*. *foldli* (*it x*)),*op-set-size-abort*)
∈ *Id* → ⟨*Rk*⟩*Rs* → *Id*
apply (*intro fun-relI*)
unfolding *gen-size-abort-def*
apply (*rule det-fold-set*[*OF foldli-size-abort IT*[*unfolded autoref-tag-defs*]])
apply (*parametricity*)+

```
  done

lemma size-abort-isSng: op-set-isSng s ⟷ op-set-size-abort 2 s = 1
  by auto

definition gen-isSng :: (nat ⇒ 's ⇒ nat) ⇒ - where
  gen-isSng sizea s ≡ sizea 2 s = 1

lemma gen-isSng[autoref-rules-raw]:
  assumes PRIO-TAG-GEN-ALGO
  assumes GEN-OP sizea op-set-size-abort (Id → (⟨Rk⟩Rs) → Id)
  shows (gen-isSng sizea,op-set-isSng) ∈ ⟨Rk⟩Rs → Id
  apply (intro fun-relI)
  unfolding gen-isSng-def using assms
  unfolding autoref-tag-defs
  apply −
  apply (subst size-abort-isSng)
  apply parametricity
  done

lemma foldli-disjoint-aux:
  foldli l1 (λx. x) (λx -. ¬x∈s2) b ⟷ b ∧ op-set-disjoint (set l1) s2
  by (induct l1 arbitrary: b) auto

lemma foldli-disjoint:
  det-fold-set X (λx. x) (λx -. ¬x∈s2) True (λs1. op-set-disjoint s1 s2)
  apply rule using foldli-disjoint-aux[where b=True] by simp

definition gen-disjoint
  :: - ⇒ ('k⇒'s2⇒bool) ⇒ -
  where gen-disjoint it1 mem2 s1 s2
  ≡ it1 s1 (λx. x) (λx -. ¬mem2 x s2) True

lemma gen-disjoint[autoref-rules-raw]:
  assumes PRIO-TAG-GEN-ALGO
  assumes IT: SIDE-GEN-ALGO (is-set-to-list Rk Rs1 it1)
  assumes MEM: GEN-OP mem2 op ∈ (Rk → ⟨Rk⟩Rs2 → Id)
  shows (gen-disjoint (λx. foldli (it1 x)) mem2,op-set-disjoint)
  ∈ ⟨Rk⟩Rs1 → ⟨Rk⟩Rs2 → Id
  apply (intro fun-relI)
  unfolding gen-disjoint-def
  apply (rule det-fold-set[OF foldli-disjoint IT[unfolded autoref-tag-defs]])
  using MEM unfolding autoref-tag-defs
  apply (parametricity)+
  done

lemma foldli-filter-aux:
  foldli l (λ-. True) (λx s. if P x then insert x s else s) s0
  = s0 ∪ op-set-filter P (set l)
```

by (*induct l arbitrary*: *s0*) *auto*

lemma *foldli-filter*:
det-fold-set X (λ-. *True*) (λ*x s*. *if P x then insert x s else s*) {}
(*op-set-filter P*)
apply *rule* **using** *foldli-filter-aux*[**where** *?s0.0*={}] **by** *simp*

definition *gen-filter*
where *gen-filter it1 emp2 ins2 P s1* ≡
it1 s1 (λ-. *True*) (λ*x s*. *if P x then ins2 x s else s*) *emp2*

lemma *gen-filter*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs1 it1*)
assumes *INS*:
GEN-OP ins2 Set.insert (*Rk*→⟨*Rk*⟩*Rs2*→⟨*Rk*⟩*Rs2*)
assumes *EMPTY*:
GEN-OP empty2 {} (⟨*Rk*⟩*Rs2*)
shows (*gen-filter* (λ*x*. *foldli* (*it1 x*)) *empty2 ins2*,*op-set-filter*)
∈ (*Rk*→*Id*) → (⟨*Rk*⟩*Rs1*) → (⟨*Rk*⟩*Rs2*)
apply (*intro fun-relI*)
unfolding *gen-filter-def*
apply (*rule det-fold-set*[*OF foldli-filter IT*[*unfolded autoref-tag-defs*]])
using *INS EMPTY* **unfolding** *autoref-tag-defs*
apply (*parametricity*)+
done

lemma *foldli-image-aux*:
foldli l (λ-. *True*) (λ*x s*. *insert* (*f x*) *s*) *s0*
= *s0* ∪ *f*‘(*set l*)
by (*induct l arbitrary*: *s0*) *auto*

lemma *foldli-image*:
det-fold-set X (λ-. *True*) (λ*x s*. *insert* (*f x*) *s*) {}
(*op* ‘ *f*)
apply *rule* **using** *foldli-image-aux*[**where** *?s0.0*={}] **by** *simp*

definition *gen-image*
where *gen-image it1 emp2 ins2 f s1* ≡
it1 s1 (λ-. *True*) (λ*x s*. *ins2* (*f x*) *s*) *emp2*

lemma *gen-image*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *IT*: *SIDE-GEN-ALGO* (*is-set-to-list Rk Rs1 it1*)
assumes *INS*:
GEN-OP ins2 Set.insert (*Rk'*→⟨*Rk'*⟩*Rs2*→⟨*Rk'*⟩*Rs2*)
assumes *EMPTY*:
GEN-OP empty2 {} (⟨*Rk'*⟩*Rs2*)
shows (*gen-image* (λ*x*. *foldli* (*it1 x*)) *empty2 ins2*,*op* ‘)

```
  ∈ (Rk→Rk') → (⟨Rk⟩Rs1) → (⟨Rk'⟩Rs2)
  apply (intro fun-relI)
  unfolding gen-image-def
  apply (rule det-fold-set[OF foldli-image IT[unfolded autoref-tag-defs]])
  using INS EMPTY unfolding autoref-tag-defs
  apply (parametricity)+
  done

lemma foldli-pick:
  assumes l≠[]
  obtains x where x∈set l
  and (foldli l (option-case True (λ-. False)) (λx -. Some x) None) = Some x
  using assms by (cases l) auto

definition gen-pick where
  gen-pick it s ≡
    (the (it s (option-case True (λ-. False)) (λx -. Some x) None))

lemma gen-pick[autoref-rules-raw]:
  assumes PRIO-TAG-GEN-ALGO
  assumes IT: SIDE-GEN-ALGO (is-set-to-list Rk Rs it)
  assumes SV: PREFER single-valued Rk
  assumes NE: SIDE-PRECOND (s'≠{})
  assumes SREF: (s,s')∈⟨Rk⟩Rs
  shows (RETURN (gen-pick (λx. foldli (it x)) s),
  (OP op-set-pick ::: ⟨Rk⟩Rs→⟨Rk⟩nres-rel)$s')∈⟨Rk⟩nres-rel
proof −
  obtain tsl' where
    [param]: (it s,tsl') ∈ ⟨Rk⟩list-rel
    and IT': RETURN tsl' ≤ it-to-sorted-list (λ- -. True) s'
    using IT[unfolded autoref-tag-defs is-set-to-list-def] SREF
    by (rule is-set-to-sorted-listE)

  from IT' NE have tsl'≠[] and [simp]: s'=set tsl'
    unfolding it-to-sorted-list-def by simp-all
  then obtain x where x∈s' and
    (foldli tsl' (option-case True (λ-. False)) (λx -. Some x) None) = Some x
    (is ?fld = -)
    by (blast elim: foldli-pick)
  moreover
  have (RETURN (gen-pick (λx. foldli (it x)) s), RETURN (the ?fld))
    ∈ ⟨Rk⟩nres-rel
    unfolding gen-pick-def
    using SV[unfolded autoref-tag-defs]
    apply (parametricity add: the-paramR)
    using ‹?fld = Some x›
    by simp
  ultimately show ?thesis
    apply (simp add: nres-rel-def)
```

```
    apply (erule ref-two-step)
    by simp
qed


term Sigma

definition gen-Sigma
  where gen-Sigma it1 it2 empX insX s1 f2 ≡
    it1 s1 (λ-. True) (λx s.
      it2 (f2 x) (λ-. True) (λy s. insX (x,y) s) s
    ) empX


lemma foldli-Sigma-aux:
  fixes s :: 's1-impl and s':: 'k set
  fixes f :: 'k-impl ⇒ 's2-impl and f':: 'k ⇒ 'l set
  fixes s0 :: 'kl-impl and s0' :: ('k×'l) set
  assumes IT1: is-set-to-list Rk Rs1 it1
  assumes IT2: is-set-to-list Rl Rs2 it2
  assumes INS:
    (insX, Set.insert) ∈
      (⟨Rk,Rl⟩prod-rel→⟨⟨Rk,Rl⟩prod-rel⟩Rs3→⟨⟨Rk,Rl⟩prod-rel⟩Rs3)
  assumes S0R: (s0, s0') ∈ ⟨⟨Rk,Rl⟩prod-rel⟩Rs3
  assumes SR: (s, s') ∈ ⟨Rk⟩Rs1
  assumes FR: (f, f') ∈ Rk → ⟨Rl⟩Rs2
  shows (foldli (it1 s) (λ-. True) (λx s.
      foldli (it2 (f x)) (λ-. True) (λy s. insX (x,y) s) s
    ) s0,s0' ∪ Sigma s' f')
    ∈ ⟨⟨Rk,Rl⟩prod-rel⟩Rs3
proof −
  have S: ⋀x s f. Sigma (insert x s) f = ({x}×f x) ∪ Sigma s f
    by auto

  obtain l' where
    IT1L: (it1 s,l')∈⟨Rk⟩list-rel
    and SL: s' = set l'
    apply (rule
      is-set-to-sorted-listE[OF IT1[unfolded is-set-to-list-def] SR])
    by (auto simp: it-to-sorted-list-def)

  show ?thesis
    unfolding SL
    using IT1L S0R
  proof (induct arbitrary: s0 s0' rule: list-rel-induct)
    case Nil thus ?case by simp
  next
    case (Cons x x' l l')
```

```
    obtain l2' where
      IT2L: (it2 (f x),l2')∈⟨Rl⟩list-rel
      and FXL: f' x' = set l2'
      apply (rule
        is-set-to-sorted-listE[
          OF IT2[unfolded is-set-to-list-def], of f x    f' x'
        ])
      apply (parametricity add: Cons.hyps(1) FR)
      by (auto simp: it-to-sorted-list-def)

    have (foldli (it2 (f x)) (λ-. True) (λy. insX (x, y)) s0,
      s0' ∪ {x'}×f' x') ∈ ⟨⟨Rk,Rl⟩prod-rel⟩Rs3
      unfolding FXL
      using IT2L ‹(s0, s0') ∈ ⟨⟨Rk, Rl⟩prod-rel⟩Rs3›
      apply (induct  arbitrary: s0 s0' rule: list-rel-induct)
      apply simp
      apply simp
      apply (subst Un-insert-left[symmetric])
      apply (rprems)
      apply (parametricity add: INS ‹(x,x')∈Rk›)
      done

    show ?case
      apply simp
      apply (subst S)
      apply (subst Un-assoc[symmetric])
      apply (rule Cons.hyps)
      apply fact
      done
  qed
qed


lemma gen-Sigma[autoref-rules-raw]:
  assumes PRIO-TAG-GEN-ALGO
  assumes IT1: SIDE-GEN-ALGO (is-set-to-list Rk Rs1 it1)
  assumes IT2: SIDE-GEN-ALGO (is-set-to-list Rl Rs2 it2)
  assumes EMPTY:
    GEN-OP empX {} (⟨⟨Rk,Rl⟩prod-rel⟩Rs3)
  assumes INS:
    GEN-OP insX Set.insert
      (⟨Rk,Rl⟩prod-rel→⟨⟨Rk,Rl⟩prod-rel⟩Rs3→⟨⟨Rk,Rl⟩prod-rel⟩Rs3)
  shows (gen-Sigma (λx. foldli (it1 x)) (λx. foldli (it2 x)) empX insX,Sigma)
    ∈ (⟨Rk⟩Rs1) → (Rk → ⟨Rl⟩Rs2) → ⟨⟨Rk,Rl⟩prod-rel⟩Rs3
  apply (intro fun-relI)
  unfolding gen-Sigma-def
  using foldli-Sigma-aux[OF
    IT1[unfolded autoref-tag-defs]
    IT2[unfolded autoref-tag-defs]
```

INS[*unfolded autoref-tag-defs*]
EMPTY[*unfolded autoref-tag-defs*]
]
by *simp*

end

3.8 Generic Map To Set Converter

theory *Gen-Map2Set*
imports
../Intf/Intf-Map
../Intf/Intf-Set
../Intf/Intf-Comp
Gen-Iterator
begin

lemma *map-fst-unit-distinct-eq*[*simp*]:
fixes *l* :: ('*k*×*unit*) *list*
shows *distinct* (*map fst l*) ⟷ *distinct l*
by (*induct l*) *auto*

definition
map2set-rel ::
(('*ki*×'*k*) *set* ⇒ (*unit*×*unit*) *set* ⇒ ('*mi*×('*k*⇀*unit*))*set*) ⇒
('*ki*×'*k*) *set* ⇒
('*mi*×('*k set*)) *set*
where
map2set-rel-def-internal:
map2set-rel R Rk ≡ ⟨*Rk*,*Id*::(*unit*×-) *set*⟩*R O* {(*m*,*dom m*)| *m*. *True*}

lemma *map2set-rel-def*: ⟨*Rk*⟩(*map2set-rel R*)
= ⟨*Rk*,*Id*::(*unit*×-) *set*⟩*R O* {(*m*,*dom m*)| *m*. *True*}
unfolding *map2set-rel-def-internal*[*abs-def*] **by** (*simp add*: *relAPP-def*)

lemma *map2set-relI*:
assumes (*s*,*m'*)∈⟨*Rk*,*Id*⟩*R* **and** *s'*=*dom m'*
shows (*s*,*s'*)∈⟨*Rk*⟩*map2set-rel R*
using *assms* **unfolding** *map2set-rel-def* **by** *blast*

lemma *map2set-relE*:
assumes (*s*,*s'*)∈⟨*Rk*⟩*map2set-rel R*
obtains *m'* **where** (*s*,*m'*)∈⟨*Rk*,*Id*⟩*R* **and** *s'*=*dom m'*
using *assms* **unfolding** *map2set-rel-def* **by** *blast*

lemma *map2set-rel-sv*[*relator-props*]:
single-valued (⟨*Rk*,*Id*⟩*Rm*) ⟹ *single-valued* (⟨*Rk*⟩*map2set-rel Rm*)
unfolding *map2set-rel-def*

by (*auto intro*: *single-valuedI dest*: *single-valuedD*)

lemma *map2set-empty*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP e op-map-empty* (⟨*Rk*,*Id*⟩*R*)
shows (*e*,{})∈⟨*Rk*⟩*map2set-rel R*
using *assms*
unfolding *map2set-rel-def*
by *auto*

lemmas [*autoref-rel-intf*] =
REL-INTFI[*of map2set-rel R i-set*, *standard*]

definition *map2set-insert i k s* ≡ *i k* () *s*
lemma *map2set-insert*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP i op-map-update* (*Rk* → *Id* → ⟨*Rk*,*Id*⟩*R* → ⟨*Rk*,*Id*⟩*R*)
shows
(*map2set-insert i*,*Set.insert*)∈*Rk*→⟨*Rk*⟩*map2set-rel R* → ⟨*Rk*⟩*map2set-rel R*
using *assms*
unfolding *map2set-rel-def map2set-insert-def*[*abs-def*]
by (*force dest*: *fun-relD*)

definition *map2set-memb l k s* ≡ *case l k s of None* ⇒ *False* | *Some* - ⇒ *True*
lemma *map2set-memb*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP l op-map-lookup* (*Rk* → ⟨*Rk*,*Id*⟩*R* → ⟨*Id*⟩*option-rel*)
shows (*map2set-memb l* ,*op*∈)
∈ *Rk*→⟨*Rk*⟩*map2set-rel R*→*Id*
using *assms*
unfolding *map2set-rel-def map2set-memb-def*[*abs-def*]
by (*force dest*: *fun-relD split*: *option.splits*)

lemma *map2set-delete*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *GEN-OP d op-map-delete* (*Rk*→⟨*Rk*,*Id*⟩*R*→⟨*Rk*,*Id*⟩*R*)
shows (*d*,*op-set-delete*)∈*Rk*→⟨*Rk*⟩*map2set-rel R*→⟨*Rk*⟩*map2set-rel R*
using *assms*
unfolding *map2set-rel-def*
by (*force dest*: *fun-relD*)

lemma *map2set-to-sorted-list*[*autoref-ga-rules*]:

fixes *it* :: ′*m* ⇒ (′*k*×*unit*) *list*
assumes *A*: *GEN-ALGO-tag* (*is-map-to-sorted-list ordR Rk Id R it*)
shows *is-set-to-sorted-list ordR Rk* (*map2set-rel R*)
(*it-to-list* (*map-iterator-dom o* (*foldli o it*)))
proof −

```
  {
    fix l::('k×unit) list
    have ⋀l0. foldli l (λ-. True) (λx σ. σ @ [fst x]) l0 = l0@map fst l
      by (induct l) auto
  }
  hence S: it-to-list (map-iterator-dom o (foldli o it)) = map fst o it
    unfolding it-to-list-def[abs-def] map-iterator-dom-def[abs-def]
      set-iterator-image-def set-iterator-image-filter-def
    by (auto)
  show ?thesis
    unfolding S
    using assms
    unfolding is-map-to-sorted-list-def is-set-to-sorted-list-def
    apply clarsimp
    apply (erule map2set-relE)
    apply (drule spec, drule spec)
    apply (drule (1) mp)
    apply (elim exE conjE)
    apply (rule-tac x=map fst l' in exI)
    apply (rule conjI)
    apply parametricity

    unfolding it-to-sorted-list-def
    apply (simp add: map-to-set-dom)
    apply (simp add: sorted-by-rel-map key-rel-def[abs-def])
    done
qed

lemma map2set-to-list[autoref-ga-rules]:
  fixes it :: 'm ⇒ ('k×unit) list
  assumes A: GEN-ALGO-tag (is-map-to-list Rk Id R it)
  shows is-set-to-list Rk (map2set-rel R)
    (it-to-list (map-iterator-dom o (foldli o it)))
  using assms unfolding is-set-to-list-def is-map-to-list-def
  by (rule map2set-to-sorted-list)
```

Transfering also non-basic operations results in specializations of map-algorithms to also be used for sets

```
lemma map2set-union[autoref-rules-raw]:
  assumes MINOR-PRIO-TAG −9
  assumes GEN-OP u op ++ (⟨Rk,Id⟩R→⟨Rk,Id⟩R→⟨Rk,Id⟩R)
  shows (u,op ∪)∈⟨Rk⟩map2set-rel R→⟨Rk⟩map2set-rel R→⟨Rk⟩map2set-rel R
  using assms
  unfolding map2set-rel-def
  by (force dest: fun-relD)

lemmas [autoref-ga-rules] = cmp-unit-eq-linorder
lemmas [autoref-rules-raw] = param-cmp-unit
```

```
lemma cmp-lex-zip-unit[simp]:
  cmp-lex (cmp-prod cmp cmp-unit) (map (λk. (k, ())) l)
          (map (λk. (k, ())) m) =
         cmp-lex cmp l m
  apply (induct cmp l m rule: cmp-lex.induct)
  apply (auto split: comp-res.split)
  done

lemma cmp-img-zip-unit[simp]:
  cmp-img (λm. map (λk. (k,())) (f m)) (cmp-lex (cmp-prod cmp1 cmp-unit))
    = cmp-img f (cmp-lex cmp1)
  unfolding cmp-img-def[abs-def]
  apply (intro ext)
  apply simp
  done



lemma map2set-finite[relator-props]:
  assumes finite-map-rel (⟨Rk,Id⟩R)
  shows finite-set-rel (⟨Rk⟩map2set-rel R)
  using assms
  unfolding map2set-rel-def finite-set-rel-def finite-map-rel-def
  by auto

lemma map2set-cmp[autoref-rules-raw]:
  assumes ELO: SIDE-GEN-ALGO (eq-linorder cmpk)
  assumes MPAR:
    GEN-OP cmp (cmp-map cmpk cmp-unit) (⟨Rk,Id⟩R → ⟨Rk,Id⟩R → Id)
  assumes FIN: PREFER finite-map-rel (⟨Rk, Id⟩R)
  shows (cmp,cmp-set cmpk)∈⟨Rk⟩map2set-rel R → ⟨Rk⟩map2set-rel R → Id
proof −
  interpret linorder comp2le cmpk    comp2lt cmpk
    using ELO by (simp add: eq-linorder-class-conv)

  show ?thesis
    using MPAR
    unfolding cmp-map-def cmp-set-def
    apply simp
    apply parametricity
    apply (drule cmp-extend-paramD)
    apply (insert FIN, fastforce simp add: finite-map-rel-def) []
    apply (simp add: sorted-list-of-map-def[abs-def])
    apply (auto simp: map2set-rel-def cmp-img-def[abs-def] dest: fun-relD) []

    apply (insert map2set-finite[OF FIN[unfolded autoref-tag-defs]],
      fastforce simp add: finite-set-rel-def)
    done
qed
```

end

3.9 List Based Maps

theory *Impl-List-Map*
imports
../Lib/Proper-Iterator
../Gen/Gen-Iterator
../Gen/Gen-Map
../Intf/Intf-Comp
../Intf/Intf-Map
List
begin

type-synonym (*'k*,*'v*) *list-map* = (*'k*×*'v*) *list*

definition *list-map-invar* = *distinct o map fst*

definition *list-map-rel-internal-def*:
list-map-rel Rk Rv ≡ ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel O br map-of list-map-invar*

lemma *list-map-rel-def*:
⟨*Rk*,*Rv*⟩*list-map-rel* = ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel O br map-of list-map-invar*
unfolding *list-map-rel-internal-def*[*abs-def*] **by** (*simp add*: *relAPP-def*)

lemma *list-rel-Range*:
∀ *x'*∈*set l'*. *x'* ∈ *Range R* ⟹ *l'* ∈ *Range* (⟨*R*⟩*list-rel*)
proof (*induction l'*)
case *Nil* **thus** *?case* **by** *force*
next
case (*Cons x' xs'*)
then obtain *xs* **where** (*xs*,*xs'*) ∈ ⟨*R*⟩ *list-rel* **by** *force*
moreover from *Cons.prems* **obtain** *x* **where** (*x*,*x'*) ∈ *R* **by** *force*
ultimately have (*x#xs*, *x'#xs'*) ∈ ⟨*R*⟩ *list-rel* **by** *simp*
thus *?case* **..**
qed

All finite maps can be represented

lemma *list-set-rel-range*:
Range (⟨*Rk*,*Rv*⟩*list-map-rel*) =
{*m*. *finite* (*dom m*) ∧ *dom m* ⊆ *Range Rk* ∧ *ran m* ⊆ *Range Rv*}
(**is** *?A* = *?B*)
proof (*intro equalityI subsetI*)
fix *m'* **assume** *m'* ∈ *?A*
then obtain *l l'* **where** *A*: (*l*,*l'*) ∈ ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel* **and**
B: *m'* = *map-of l'* **and** *C*: *list-map-invar l'*
unfolding *list-map-rel-def br-def* **by** *blast*

```
  {
    fix x′ y′ assume m′ x′ = Some y′
    with B have (x′,y′) ∈ set l′ by (fast dest: map-of-SomeD)
    hence x′ ∈ Range Rk and y′ ∈ Range Rv
      by (induction rule: list-rel-induct[OF A], auto)
  }
  with B show m′ ∈ ?B by (force dest: map-of-SomeD simp: ran-def)

next
  fix m′ assume m′ ∈ ?B
  hence A: finite (dom m′) and B: dom m′ ⊆ Range Rk and
        C: ran m′ ⊆ Range Rv by simp-all
  from A have finite (map-to-set m′) by (simp add: finite-map-to-set)
  from finite-distinct-list[OF this]
      obtain l′ where l′-props: distinct l′   set l′ = map-to-set m′ by blast
  hence distinct (map fst l′)
      by (force simp: distinct-map inj-on-def map-to-set-def)
  moreover from map-of-map-to-set[OF this] and l′-props
      have  map-of l′ = m′ by simp
  ultimately have (l′,m′) ∈ br map-of list-map-invar
      unfolding br-def list-map-invar-def o-def  by simp

  moreover from B and C and l′-props
      have ∀ x ∈ set l′. x ∈ Range (⟨Rk,Rv⟩prod-rel)
      unfolding map-to-set-def ran-def prod-rel-def by force
  from list-rel-Range[OF this] obtain l where
      (l,l′) ∈ ⟨⟨Rk,Rv⟩prod-rel⟩list-rel by force

  ultimately show m′ ∈ ?A unfolding list-map-rel-def by blast
qed


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

  lemma list-map-rel-finite[autoref-ga-rules]:
    finite-map-rel (⟨Rk,Rv⟩list-map-rel)
    unfolding finite-map-rel-def list-map-rel-def
    by (auto simp: br-def)

  lemma list-set-rel-sv[relator-props]:
    single-valued Rk ⟹ single-valued Rv ⟹
        single-valued (⟨Rk,Rv⟩list-map-rel)
    unfolding list-map-rel-def
    by tagged-solver
```

3.9.1 Implementation

```
primrec list-map-lookup ::
    ('k ⇒ 'k ⇒ bool) ⇒ 'k ⇒ ('k,'v) list-map ⇒ 'v option where
```

list-map-lookup eq - [] = None |
list-map-lookup eq k (y#ys) =
  (*if eq (fst y) k then Some (snd y) else list-map-lookup eq k ys*)

primrec *list-map-update-aux* :: ('*k* ⇒ '*k* ⇒ *bool*) ⇒ '*k* ⇒ '*v* ⇒
  ('*k*,'*v*) *list-map* ⇒ ('*k*,'*v*) *list-map* ⇒ ('*k*,'*v*) *list-map***where**
list-map-update-aux eq k v [] accu = (k,v) # accu |
list-map-update-aux eq k v (x#xs) accu =
  (*if eq (fst x) k*
    then (k,v) # xs @ accu
    else list-map-update-aux eq k v xs (x#accu))

definition *list-map-update eq k v m* ≡
  list-map-update-aux eq k v m []

primrec *list-map-delete-aux* :: ('*k* ⇒ '*k* ⇒ *bool*) ⇒ '*k* ⇒
  ('*k*, '*v*) *list-map* ⇒ ('*k*, '*v*) *list-map* ⇒ ('*k*, '*v*) *list-map* **where**
list-map-delete-aux eq k [] accu = accu |
list-map-delete-aux eq k (x#xs) accu =
  (*if eq (fst x) k*
    then xs @ accu
    else list-map-delete-aux eq k xs (x#accu))

definition *list-map-delete eq k m* ≡ *list-map-delete-aux eq k m []*

definition *list-map-isEmpty* :: ('*k*,'*v*) *list-map* ⇒ *bool*
  where *list-map-isEmpty* ≡ *List.null*

definition *list-map-isSng* :: ('*k*,'*v*) *list-map* ⇒ *bool*
  where *list-map-isSng m = (case m of [x] ⇒ True | - ⇒ False)*

definition *list-map-size* :: ('*k*,'*v*) *list-map* ⇒ *nat*
  where *list-map-size* ≡ *length*

definition *list-map-iteratei* :: ('*k*,'*v*) *list-map* ⇒ ('*b* ⇒ *bool*) ⇒
  (('*k*×'*v*) ⇒ '*b* ⇒ '*b*) ⇒ '*b* ⇒ '*b*
  where *list-map-iteratei* ≡ *foldli*

definition *list-map-to-list* :: ('*k*,'*v*) *list-map* ⇒ ('*k*×'*v*) *list*
  where *list-map-to-list = id*

3.9.2 Parametricity

lemma *list-map-autoref-empty*[*autoref-rules*]:
  ([], *op-map-empty*)∈⟨*Rk*,*Rv*⟩*list-map-rel*
  by (*auto simp*: *list-map-rel-def br-def list-map-invar-def*)

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

```
  (list-map-lookup,list-map-lookup) ∈ (Rk → Rk → bool-rel) →
      Rk → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel → ⟨Rv⟩option-rel
unfolding list-map-lookup-def[abs-def] by parametricity

lemma list-map-autoref-lookup-aux:
  assumes eq: GEN-OP eq op= (Rk→Rk→Id)
  assumes K: (k, k′) ∈ Rk
  assumes M: (m, m′) ∈ ⟨⟨Rk, Rv⟩prod-rel⟩list-rel
  shows (list-map-lookup eq k m, op-map-lookup k′ (map-of m′))
            ∈ ⟨Rv⟩option-rel
unfolding op-map-lookup-def
proof (induction rule: list-rel-induct[OF M, case-names Nil Cons])
  case Nil
    show ?case by simp
next
  case (Cons x x′ xs xs′)
    from eq have eq′: (eq,op=) ∈ Rk → Rk → Id by simp
    with eq′[param-fo] and K and Cons
      show ?case by (force simp: prod-rel-def)
qed

lemma list-map-autoref-lookup[autoref-rules]:
  assumes GEN-OP eq op= (Rk→Rk→Id)
  shows (list-map-lookup eq, op-map-lookup) ∈
      Rk → ⟨Rk,Rv⟩list-map-rel → ⟨Rv⟩option-rel
   by (force simp: list-map-rel-def br-def
            dest: list-map-autoref-lookup-aux[OF assms])


lemma param-list-map-update-aux[param]:
  (list-map-update-aux,list-map-update-aux) ∈ (Rk → Rk → bool-rel) →
      Rk → Rv → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel
       → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel
unfolding list-map-update-aux-def[abs-def] by parametricity

lemma param-list-map-update[param]:
  (list-map-update,list-map-update) ∈ (Rk → Rk → bool-rel) →
      Rk → Rv → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel
unfolding list-map-update-def[abs-def] by parametricity


lemma list-map-autoref-update-aux1:
  assumes eq: (eq,op=) ∈ Rk → Rk → Id
  assumes K: (k, k′) ∈ Rk
  assumes V: (v, v′) ∈ Rv
  assumes A: (accu, accu′) ∈ ⟨⟨Rk, Rv⟩prod-rel⟩list-rel
  assumes M: (m, m′) ∈ ⟨⟨Rk, Rv⟩prod-rel⟩list-rel
  shows (list-map-update-aux eq k v m accu,
```

list-map-update-aux op= k′ v′ m′ accu′)
$\in$ ⟨⟨*Rk*, *Rv*⟩*prod-rel*⟩*list-rel*
proof (*insert A*, *induction arbitrary*: *accu accu′*
rule: *list-rel-induct*[*OF M*, *case-names Nil Cons*])
case *Nil*
thus *?case* **by** (*simp add*: *K V*)
next
case (*Cons x x′ xs xs′*)
from *eq* **have** *eq′*: (*eq*,*op=*) $\in$ *Rk* $\to$ *Rk* $\to$ *Id* **by** *simp*
from *eq′*[*param-fo*] *Cons*(*1*) *K*
have [*simp*]: (*eq* (*fst x*) *k*) $\longleftrightarrow$ ((*fst x′*) = *k′*)
by (*force simp*: *prod-rel-def*)
show *?case*
proof (*cases eq* (*fst x*) *k*)
case *False*
from *Cons.prems* **and** *Cons.hyps* **have** (*x* # *accu*, *x′* # *accu′*) $\in$
⟨⟨*Rk*, *Rv*⟩*prod-rel*⟩*list-rel* **by** *parametricity*
from *Cons.IH*[*OF this*] **and** *False* **show** *?thesis* **by** *simp*
next
case *True*
from *Cons.prems* **and** *Cons.hyps* **have** (*xs* @ *accu*, *xs′* @ *accu′*) $\in$
⟨⟨*Rk*, *Rv*⟩*prod-rel*⟩*list-rel* **by** *parametricity*
with *K* **and** *V* **and** *True* **show** *?thesis* **by** *simp*
qed
qed

lemma *list-map-autoref-update1*[*param*]:
assumes *eq*: (*eq*,*op=*) $\in$ *Rk* $\to$ *Rk* $\to$ *Id*
shows (*list-map-update eq*, *list-map-update op=*) $\in$ *Rk* $\to$ *Rv* $\to$
⟨⟨*Rk*, *Rv*⟩*prod-rel*⟩*list-rel* $\to$ ⟨⟨*Rk*, *Rv*⟩*prod-rel*⟩*list-rel*
unfolding *list-map-update-def*[*abs-def*]
by (*intro fun-relI*, *erule* (*1*) *list-map-autoref-update-aux1*[*OF eq*],
simp-all)

lemma *map-add-sng-right*: *m* ++ [*k*↦*v*] = *m*(*k* ↦ *v*)
unfolding *map-add-def* **by** *force*
lemma *map-add-sng-right′*:
m ++ (λ*a*. *if a* = *k then Some v else None*) = *m*(*k* ↦ *v*)
unfolding *map-add-def* **by** *force*

lemma *list-map-autoref-update-aux2*:
assumes *K*: (*k*, *k′*) $\in$ *Id*
assumes *V*: (*v*, *v′*) $\in$ *Id*
assumes *A*: (*accu*, *accu′*) $\in$ *br map-of list-map-invar*
assumes *A1*: *distinct* (*map fst* (*m* @ *accu*))
assumes *A2*: *k* $\notin$ *set* (*map fst accu*)
assumes *M*: (*m*, *m′*) $\in$ *br map-of list-map-invar*

```
  shows (list-map-update-aux op= k v m accu,
          accu' ++ op-map-update k' v' m')
              ∈ br map-of list-map-invar (is (?f m accu, -) ∈ -)
using M A A1 A2
proof (induction m arbitrary: accu accu' m')
  case Nil
    with K V show ?case by (auto simp: br-def list-map-invar-def
        map-add-sng-right')
next
  case (Cons x xs accu accu' m')
    from Cons.prems have A: m' = map-of (x#xs)    accu' = map-of accu
        unfolding br-def by simp-all
    show ?case
    proof (cases (fst x) = k)
      case True
        hence ((k, v) # xs @ accu, accu' ++ op-map-update k' v' m')
                ∈ br map-of list-map-invar
            using K V Cons.prems(3,4) unfolding br-def
            by (force simp add: A list-map-invar-def)
        also from True have (k,v) # xs @ accu = ?f (x # xs) accu by simp
        finally show ?thesis .
    next
      case False
        from Cons.prems(1) have B: (xs, map-of xs) ∈ br map-of
            list-map-invarby (simp add: br-def list-map-invar-def)
        from Cons.prems(2,3) have C: (x#accu, map-of (x#accu)) ∈ br map-of
            list-map-invar by (simp add: br-def list-map-invar-def)
        from Cons.prems(3) have D: distinct (map fst (xs @ x # accu))
            by simp
        from Cons.prems(4) and False have E: k ∉ set (map fst (x # accu))
            by simp
        note Cons.IH[OF B C D E]
        also from False have ?f xs (x#accu) = ?f (x#xs) accu by simp
        also from distinct-map-fstD[OF D]
            have F: ⋀z. (fst x, z) ∈ set xs ⟹ z = snd x by force
        have map-of (x # accu) ++ op-map-update k' v' (map-of xs) =
                accu' ++ op-map-update k' v' m'
            by (intro ext, auto simp: A F map-add-def
                    dest: map-of-SomeD split: option.split)
        finally show ?thesis .
    qed
qed

lemma list-map-autoref-update2[param]:
  shows (list-map-update op=, op-map-update) ∈ Id → Id →
            br map-of list-map-invar → br map-of list-map-invar
unfolding list-map-update-def[abs-def]
apply (intro fun-relI)
apply (drule list-map-autoref-update-aux2
```

```
                [where accu=[] and accu'=Map.empty])
apply (auto simp: br-def list-map-invar-def)
done

lemma list-map-autoref-update[autoref-rules]:
  assumes eq: GEN-OP eq op= (Rk→Rk→Id)
  shows (list-map-update eq, op-map-update) ∈
      Rk → Rv → ⟨Rk,Rv⟩list-map-rel → ⟨Rk,Rv⟩list-map-rel
unfolding list-map-rel-def
apply (intro fun-relI, elim relcompE, intro relcompI, clarsimp)
apply (erule (2) list-map-autoref-update1[param-fo, OF eq[simplified]])
apply (rule list-map-autoref-update2[param-fo], simp-all)
done

lemma list-map-autoref-update-dj[autoref-rules]:
    assumes PRIO-TAG-OPTIMIZATION
    assumes new: SIDE-PRECOND-OPT (k' ∉ dom m')
    assumes K: (k,k')∈Rk and V: (v,v')∈Rv
    assumes M: (l,m')∈⟨Rk, Rv⟩list-map-rel
    defines R-annot ≡ Rk → Rv → ⟨Rk,Rv⟩list-map-rel → ⟨Rk,Rv⟩list-map-rel
    shows
      ((k, v)#l,
        (OP op-map-update:::R-annot)$k'$v'$m')
      ∈ ⟨Rk,Rv⟩list-map-rel
proof −
  from M obtain l' where A: (l,l') ∈ ⟨⟨Rk, Rv⟩prod-rel⟩list-rel and
      B: (l',m') ∈ br map-of list-map-invar
      unfolding list-map-rel-def by blast
  hence ((k,v)#l, (k',v')#l')∈⟨⟨Rk, Rv⟩prod-rel⟩list-rel
      and ((k',v')#l', m'(k' ↦ v')) ∈ br map-of list-map-invar
      using assms unfolding br-def list-map-invar-def
          by (simp-all add: dom-map-of-conv-image-fst)
  thus ?thesis
    unfolding autoref-tag-defs
    by (force simp: list-map-rel-def)
qed


lemma param-list-map-delete-aux[param]:
  (list-map-delete-aux,list-map-delete-aux) ∈ (Rk → Rk → bool-rel) →
       Rk → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel
        → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel
unfolding list-map-delete-aux-def[abs-def] by parametricity

lemma param-list-map-delete[param]:
  (list-map-delete,list-map-delete) ∈ (Rk → Rk → bool-rel) →
       Rk → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel → ⟨⟨Rk,Rv⟩prod-rel⟩list-rel
unfolding list-map-delete-def[abs-def] by parametricity
```

```
lemma list-map-autoref-delete-aux1:
  assumes eq: (eq,op=) ∈ Rk → Rk → Id
  assumes K: (k, k') ∈ Rk
  assumes A: (accu, accu') ∈ ⟨⟨Rk, Rv⟩prod-rel⟩list-rel
  assumes M: (m, m') ∈ ⟨⟨Rk, Rv⟩prod-rel⟩list-rel
  shows (list-map-delete-aux eq k m accu,
         list-map-delete-aux op= k' m' accu')
             ∈ ⟨⟨Rk, Rv⟩prod-rel⟩list-rel
proof (insert A, induction arbitrary: accu accu'
          rule: list-rel-induct[OF M, case-names Nil Cons])
  case Nil
    thus ?case by (simp add: K)
next
  case (Cons x x' xs xs')
    from eq have eq': (eq,op=) ∈ Rk → Rk → Id by simp
    from eq'[param-fo] Cons(1) K
      have [simp]: (eq (fst x) k) ⟷ ((fst x') = k')
      by (force simp: prod-rel-def)
    show ?case
    proof (cases eq (fst x) k)
      case False
        from Cons.prems and Cons.hyps have (x # accu, x' # accu') ∈
          ⟨⟨Rk, Rv⟩prod-rel⟩list-rel by parametricity
        from Cons.IH[OF this] and False show ?thesis by simp
    next
      case True
        from Cons.prems and Cons.hyps have (xs @ accu, xs' @ accu') ∈
          ⟨⟨Rk, Rv⟩prod-rel⟩list-rel by parametricity
        with K and True show ?thesis by simp
  qed
qed

lemma list-map-autoref-delete1[param]:
  assumes eq: (eq,op=) ∈ Rk → Rk → Id
  shows (list-map-delete eq, list-map-delete op=) ∈ Rk →
          ⟨⟨Rk, Rv⟩prod-rel⟩list-rel → ⟨⟨Rk, Rv⟩prod-rel⟩list-rel
unfolding list-map-delete-def[abs-def]
  by (intro fun-relI, erule list-map-autoref-delete-aux1[OF eq],
    simp-all)


lemma list-map-autoref-delete-aux2:
  assumes K: (k, k') ∈ Id
  assumes A: (accu, accu') ∈ br map-of list-map-invar
  assumes A1: distinct (map fst (m @ accu))
  assumes A2: k ∉ set (map fst accu)
  assumes M: (m, m') ∈ br map-of list-map-invar
  shows (list-map-delete-aux op= k m accu,
         accu' ++ op-map-delete k' m')
```

```
        ∈ br map-of list-map-invar (is (?f m accu, -) ∈ -)
using M A A1 A2
proof (induction m arbitrary: accu accu' m')
  case Nil
    with K show ?case by (auto simp: br-def list-map-invar-def
        map-add-sng-right')
next
  case (Cons x xs accu accu' m')
    from Cons.prems have A: m' = map-of (x#xs)    accu' = map-of accu
        unfolding br-def by simp-all
    show ?case
    proof (cases (fst x) =  k)
      case True
        with Cons.prems(3) have map-of xs (fst x) = None
          by (induction xs, simp-all)
        with fun-upd-triv[of map-of xs    fst x]
          have map-of xs |‘ (− {fst x}) = map-of xs by simp
        with True have(xs @ accu, accu' ++ op-map-delete k' m')
                    ∈ br map-of list-map-invar
          using K Cons.prems unfolding br-def
          by (auto simp add: A list-map-invar-def)
        thus ?thesis using True by simp
    next
      case False
        from False and K have [simp]: fst x ≠ k' by simp
        from Cons.prems(1) have B: (xs, map-of xs) ∈ br map-of
          list-map-invarby (simp add: br-def list-map-invar-def)
        from Cons.prems(2,3) have C: (x#accu, map-of (x#accu)) ∈ br map-of
          list-map-invar by (simp add: br-def list-map-invar-def)
        from Cons.prems(3) have D: distinct (map fst (xs @ x # accu))
          by simp
        from Cons.prems(4) and False have E: k ∉ set (map fst (x # accu))
          by simp
        note Cons.IH[OF B C D E]
        also from False have ?f xs (x#accu) = ?f (x#xs) accu by simp
        also from distinct-map-fstD[OF D]
          have F: ⋀z. (fst x, z) ∈ set xs ⟹ z = snd x by force

        from Cons.prems(3) have map-of xs (fst x) = None
          by (induction xs, simp-all)
        hence map-of (x # accu) ++ op-map-delete k' (map-of xs) =
              accu' ++ op-map-delete k' m'
          apply (intro ext, simp add: map-add-def A
                        split: option.split)
          apply (intro conjI impI allI)
          apply (auto simp: restrict-map-def)
          done
        finally show ?thesis .
    qed
```

qed

lemma *list-map-autoref-delete2* [*param*]:
  shows (*list-map-delete op*=, *op-map-delete*) ∈ *Id* →
    br map-of list-map-invar → *br map-of list-map-invar*
unfolding *list-map-delete-def* [*abs-def*]
apply (*intro fun-relI*)
apply (*drule list-map-autoref-delete-aux2*
    [**where** *accu*=[] **and** *accu'*=*Map.empty*])
apply (*auto simp*: *br-def list-map-invar-def*)
done

lemma *list-map-autoref-delete* [*autoref-rules*]:
  assumes *eq*: *GEN-OP eq op*= (*Rk*→*Rk*→*Id*)
  shows (*list-map-delete eq*, *op-map-delete*) ∈
    Rk → ⟨*Rk*,*Rv*⟩*list-map-rel* → ⟨*Rk*,*Rv*⟩*list-map-rel*
unfolding *list-map-rel-def*
apply (*intro fun-relI*, *elim relcompE*, *intro relcompI*, *clarsimp*)
apply (*erule* (*1*) *list-map-autoref-delete1* [*param-fo*, *OF eq*[*simplified*]])
apply (*rule list-map-autoref-delete2* [*param-fo*], *simp-all*)
done

lemma *list-map-autoref-isEmpty* [*autoref-rules*]:
  shows (*list-map-isEmpty*, *op-map-isEmpty*) ∈
    ⟨*Rk*,*Rv*⟩*list-map-rel* → *bool-rel*
unfolding *list-map-isEmpty-def op-map-isEmpty-def* [*abs-def*]
  list-map-rel-def br-def List.null-def [*abs-def*] **by** *force*

lemma *param-list-map-isSng* [*param*]:
  assumes (*l*,*l'*) ∈ ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*
  shows (*list-map-isSng l*, *list-map-isSng l'*) ∈ *bool-rel*
unfolding *list-map-isSng-def* **using** *assms* **by** *parametricity*

lemma *list-map-autoref-isSng-aux*:
  assumes (*l'*,*m'*) ∈ *br map-of list-map-invar*
  shows (*list-map-isSng l'*, *op-map-isSng m'*) ∈ *bool-rel*
using *assms*
unfolding *list-map-isSng-def op-map-isSng-def br-def list-map-invar-def*
apply (*clarsimp split*: *list.split*)
apply (*intro conjI impI allI*)
apply (*metis map-upd-nonempty*)
apply *blast*
apply (*simp*, *metis fun-upd-apply option.distinct*(*1*))
done

lemma *list-map-autoref-isSng* [*autoref-rules*]:
  (*list-map-isSng*, *op-map-isSng*) ∈ ⟨*Rk*,*Rv*⟩*list-map-rel* → *bool-rel*
  using *assms* **unfolding** *list-map-rel-def*

```
  by (blast dest!: param-list-map-isSng list-map-autoref-isSng-aux)

lemma list-map-autoref-size-aux:
  assumes distinct (map fst x)
  shows card (dom (map-of x)) = length x
proof−
  have card (dom (map-of x)) = card (map-to-set (map-of x))
     by (simp add: card-map-to-set)
  also from assms have ... = card (set x)
     by (simp add: map-to-set-map-of)
  also from assms have ... = length x
     by (force simp: distinct-card dest!: distinct-mapI)
  finally show ?thesis .
qed

lemma param-list-map-size[param]:
  (list-map-size, list-map-size) ∈ ⟨⟨Rk,Rv⟩prod-rel⟩list-rel → nat-rel
  unfolding list-map-size-def[abs-def] by parametricity

lemma list-map-autoref-size[autoref-rules]:
  shows (list-map-size, op-map-size) ∈
           ⟨Rk,Rv⟩list-map-rel → nat-rel
unfolding list-map-size-def[abs-def] op-map-size-def[abs-def]
    list-map-rel-def br-def list-map-invar-def
    by (force simp: list-map-autoref-size-aux list-rel-imp-same-length)


lemma autoref-list-map-is-iterator[autoref-ga-rules]:
  shows is-map-to-list Rk Rv list-map-rel list-map-to-list
unfolding is-map-to-list-def is-map-to-sorted-list-def
proof (clarify)
  fix l m′
  assume (l,m′) ∈ ⟨Rk,Rv⟩list-map-rel
  then obtain l′ where (l,l′)∈⟨⟨Rk,Rv⟩prod-rel⟩list-rel
     and (l′,m′)∈br map-of list-map-invar
     unfolding list-map-rel-def by blast
  moreover from this have RETURN l′ ≤ it-to-sorted-list
                          (key-rel (λ- -. True)) (map-to-set m′)
     unfolding it-to-sorted-list-def
     apply (intro refine-vcg)
     unfolding br-def list-map-invar-def key-rel-def[abs-def]
     apply (auto intro: distinct-mapI simp: map-to-set-map-of)
     done
  ultimately show
     ∃ l′. (list-map-to-list l, l′) ∈ ⟨⟨Rk, Rv⟩prod-rel⟩list-rel ∧
          RETURN l′ ≤ it-to-sorted-list (key-rel (λ- -. True))
                          (map-to-set m′)
      unfolding list-map-to-list-def by force
qed
```

lemma *pi-list-map*[*icf-proper-iteratorI*]:
proper-it (*list-map-iteratei m*) (*list-map-iteratei m*)
unfolding *proper-it-def list-map-iteratei-def* **by** *blast*

lemma *pi′-list-map*[*icf-proper-iteratorI*]:
proper-it′ list-map-iteratei list-map-iteratei
by (*rule proper-it′I*, *rule pi-list-map*)

end

3.10 Red-Black Tree based Maps

theory *Impl-RBT-Map*
imports
~~/src/HOL/Library/RBT-Impl
../Lib/RBT-add
../../Autoref/Autoref
../Gen/Gen-Iterator
../Intf/Intf-Comp
../Intf/Intf-Map
begin

3.10.1 Standard Setup

inductive-set *color-rel* **where**
(*color.R*,*color.R*) ∈ *color-rel*
| (*color.B*,*color.B*) ∈ *color-rel*

inductive-cases *color-rel-elims*:
(*x*,*color.R*) ∈ *color-rel*
(*x*,*color.B*) ∈ *color-rel*
(*color.R*,*y*) ∈ *color-rel*
(*color.B*,*y*) ∈ *color-rel*

thm *color-rel-elims*

lemma *param-color*[*param*]:
(*color.R*,*color.R*)∈*color-rel*
(*color.B*,*color.B*)∈*color-rel*
(*color-case*,*color-case*)∈*R* → *R* → *color-rel* → *R*
by (*auto*
intro: *color-rel.intros*
elim: *color-rel.cases*
split: *color.split*)

inductive-set *rbt-rel-aux* **for** *Ra Rb* **where**
(*rbt.Empty*,*rbt.Empty*)∈*rbt-rel-aux Ra Rb*

```
| ⟦ (c,c')∈color-rel;
    (l,l')∈rbt-rel-aux Ra Rb; (a,a')∈Ra; (b,b')∈Rb;
    (r,r')∈rbt-rel-aux Ra Rb ⟧
  ⟹ (rbt.Branch c l a b r, rbt.Branch c' l' a' b' r')∈rbt-rel-aux Ra Rb

inductive-cases rbt-rel-aux-elims:
  (x,rbt.Empty)∈rbt-rel-aux Ra Rb
  (rbt.Empty,x')∈rbt-rel-aux Ra Rb
  (rbt.Branch c l a b r,x')∈rbt-rel-aux Ra Rb
  (x,rbt.Branch c' l' a' b' r')∈rbt-rel-aux Ra Rb

definition rbt-rel ≡ rbt-rel-aux
lemma rbt-rel-aux-fold: rbt-rel-aux Ra Rb ≡ ⟨Ra,Rb⟩rbt-rel
  by (simp add: rbt-rel-def relAPP-def)

lemmas rbt-rel-intros = rbt-rel-aux.intros[unfolded rbt-rel-aux-fold]
lemmas rbt-rel-cases = rbt-rel-aux.cases[unfolded rbt-rel-aux-fold]
lemmas rbt-rel-induct[induct set]
  = rbt-rel-aux.induct[unfolded rbt-rel-aux-fold]
lemmas rbt-rel-elims = rbt-rel-aux-elims[unfolded rbt-rel-aux-fold]

lemma param-rbt1[param]:
  (rbt.Empty,rbt.Empty) ∈ ⟨Ra,Rb⟩rbt-rel
  (rbt.Branch,rbt.Branch) ∈
    color-rel → ⟨Ra,Rb⟩rbt-rel → Ra → Rb → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
  by (auto intro: rbt-rel-intros)

lemma param-rbt-case[param]:
  (rbt-case,rbt-case) ∈
    Ra → (color-rel → ⟨Rb,Rc⟩rbt-rel → Rb → Rc → ⟨Rb,Rc⟩rbt-rel → Ra)
      → ⟨Rb,Rc⟩rbt-rel → Ra
  apply clarsimp
  apply (erule rbt-rel-cases)
  apply simp
  apply simp
  apply parametricity
  done

lemma param-rbt-rec[param]: (rbt-rec, rbt-rec) ∈
  Ra → (color-rel → ⟨Rb,Rc⟩rbt-rel → Rb → Rc → ⟨Rb,Rc⟩rbt-rel
   → Ra → Ra → Ra) → ⟨Rb,Rc⟩rbt-rel → Ra
proof (intro fun-relI)
  case (goal1 s s' f f' t t') from goal1(3,1,2) show ?case
    apply (induct arbitrary: s s')
    apply simp
    apply simp
    apply parametricity
    done
qed
```

```
lemma param-paint[param]:
  (paint,paint)∈color-rel → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
  unfolding paint-def
  by parametricity

lemma param-balance[param]:
  shows (balance,balance) ∈
    ⟨Ra,Rb⟩rbt-rel → Ra → Rb → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
proof (intro fun-relI)
  case (goal1 t1 t1' a a' b b' t2 t2')
  thus ?case
    apply (induct t1' a' b' t2' arbitrary: t1 a b t2 rule: balance.induct)
    apply (elim-all rbt-rel-elims color-rel-elims)
    apply (simp-all only: balance.simps)
    apply (parametricity)+
    done
qed


lemma param-rbt-ins[param]:
  fixes less
  assumes param-less[param]: (less,less') ∈ Ra → Ra → Id
  shows (ord.rbt-ins less,ord.rbt-ins less') ∈
          (Ra→Rb→Rb→Rb) → Ra → Rb → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
proof (intro fun-relI)
  case (goal1 f f' a a' b b' t t')
  thus ?case
    apply (induct f' a' b' t' arbitrary: f a b t rule: ord.rbt-ins.induct)
    apply (elim-all rbt-rel-elims color-rel-elims)
    apply (simp-all only: ord.rbt-ins.simps rbt-ins.simps)
    apply parametricity+
    done
qed

term rbt-insert
lemma param-rbt-insert[param]:
  fixes less
  assumes param-less[param]: (less,less') ∈ Ra → Ra → Id
  shows (ord.rbt-insert less,ord.rbt-insert less') ∈
    Ra → Rb → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
  unfolding rbt-insert-def ord.rbt-insert-def
  unfolding rbt-insert-with-key-def[abs-def]
    ord.rbt-insert-with-key-def[abs-def]
  by parametricity

lemma param-rbt-lookup[param]:
  fixes less
  assumes param-less[param]: (less,less') ∈ Ra → Ra → Id
```

```
  shows (ord.rbt-lookup less,ord.rbt-lookup less') ∈
        ⟨Ra,Rb⟩rbt-rel → Ra → ⟨Rb⟩option-rel
  unfolding rbt-lookup-def ord.rbt-lookup-def
  by parametricity

term balance-left
lemma param-balance-left[param]:
  (balance-left, balance-left) ∈
    ⟨Ra,Rb⟩rbt-rel → Ra → Rb → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
proof (intro fun-relI)
  case (goal1 l l' a a' b b' r r')
  thus ?case
    apply (induct l a b r arbitrary: l' a' b' r' rule: balance-left.induct)
    apply (elim-all rbt-rel-elims color-rel-elims)
    apply (simp-all only: balance-left.simps)
    apply parametricity+
    done
qed

term balance-right
lemma param-balance-right[param]:
  (balance-right, balance-right) ∈
    ⟨Ra,Rb⟩rbt-rel → Ra → Rb → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
proof (intro fun-relI)
  case (goal1 l l' a a' b b' r r')
  thus ?case
    apply (induct l a b r arbitrary: l' a' b' r' rule: balance-right.induct)
    apply (elim-all rbt-rel-elims color-rel-elims)
    apply (simp-all only: balance-right.simps)
    apply parametricity+
    done
qed

lemma param-combine[param]:
  (combine,combine)∈⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
proof (intro fun-relI)
  case (goal1 t1 t1' t2 t2')
  thus ?case
    apply (induct t1 t2 arbitrary: t1' t2' rule: combine.induct)
    apply (elim-all rbt-rel-elims color-rel-elims)
    apply (simp-all only: combine.simps)
    apply parametricity+
    done
qed

lemma ih-aux1: ⟦ (a',b)∈R; a'=a ⟧ ⟹ (a,b)∈R by auto
lemma is-eq: a=b ⟹ a=b .

lemma param-rbt-del-aux:
```

```
  fixes br
  fixes less
  assumes param-less[param]: (less,less') ∈ Ra → Ra → Id
  shows
  ⟦ (ak1,ak1')∈Ra; (al,al')∈⟨Ra,Rb⟩rbt-rel; (ak,ak')∈Ra;
    (av,av')∈Rb; (ar,ar')∈⟨Ra,Rb⟩rbt-rel
  ⟧ ⟹ (ord.rbt-del-from-left less ak1 al ak av ar,
    ord.rbt-del-from-left less' ak1' al' ak' av' ar')
  ∈ ⟨Ra,Rb⟩rbt-rel
  ⟦ (bk1,bk1')∈Ra; (bl,bl')∈⟨Ra,Rb⟩rbt-rel; (bk,bk')∈Ra;
    (bv,bv')∈Rb; (br,br')∈⟨Ra,Rb⟩rbt-rel
  ⟧ ⟹ (ord.rbt-del-from-right less bk1 bl bk bv br,
    ord.rbt-del-from-right less' bk1' bl' bk' bv' br')
  ∈ ⟨Ra,Rb⟩rbt-rel
  ⟦ (ck,ck')∈Ra; (ct,ct')∈⟨Ra,Rb⟩rbt-rel ⟧
    ⟹ (ord.rbt-del less ck ct, ord.rbt-del less' ck' ct') ∈ ⟨Ra,Rb⟩rbt-rel
  apply (induct
    ak1' al' ak' av' ar' and bk1' bl' bk' bv' br' and ck' ct'
    arbitrary: ak1 al ak av ar and bk1 bl bk bv br and ck ct
    rule: ord.rbt-del-from-left-rbt-del-from-right-rbt-del.induct)

  apply (assumption
    | elim rbt-rel-elims color-rel-elims
    | simp (no-asm-use) only: rbt-del.simps ord.rbt-del.simps
        rbt-del-from-left.simps ord.rbt-del-from-left.simps
        rbt-del-from-right.simps ord.rbt-del-from-right.simps
    | parametricity
    | rule rbt-rel-intros
    | hypsubst
    | (simp, rule ih-aux1, rprems)
    | (rule is-eq, simp)
  ) +
  done

lemma param-rbt-del[param]:
  fixes less
  assumes param-less: (less,less') ∈ Ra → Ra → Id
  shows
  (ord.rbt-del-from-left less, ord.rbt-del-from-left less') ∈
    Ra → ⟨Ra,Rb⟩rbt-rel → Ra → Rb → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
  (ord.rbt-del-from-right less, ord.rbt-del-from-right less') ∈
    Ra → ⟨Ra,Rb⟩rbt-rel → Ra → Rb → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
  (ord.rbt-del less,ord.rbt-del less') ∈
    Ra → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
  by (intro fun-relI, blast intro: param-rbt-del-aux[OF param-less])+

lemma param-rbt-delete[param]:
  fixes less
  assumes param-less[param]: (less,less') ∈ Ra → Ra → Id
```

```
  shows (ord.rbt-delete less, ord.rbt-delete less')
    ∈ Ra → ⟨Ra,Rb⟩rbt-rel → ⟨Ra,Rb⟩rbt-rel
  unfolding rbt-delete-def[abs-def] ord.rbt-delete-def[abs-def]
  by parametricity

term ord.rbt-insert-with-key

abbreviation compare-rel :: (RBT-Impl.compare × -) set
  where compare-rel ≡ Id

lemma param-compare[param]:
  (RBT-Impl.LT,RBT-Impl.LT)∈compare-rel
  (RBT-Impl.GT,RBT-Impl.GT)∈compare-rel
  (RBT-Impl.EQ,RBT-Impl.EQ)∈compare-rel
 (RBT-Impl.compare-case,RBT-Impl.compare-case)∈R→R→R→compare-rel→R
  by (auto split: RBT-Impl.compare.split)

lemma param-rbtreeify-aux[param]:
  ⟦n≤length kvs; (n,n')∈nat-rel; (kvs,kvs')∈⟨⟨Ra,Rb⟩prod-rel⟩list-rel⟧
  ⟹ (rbtreeify-f n kvs,rbtreeify-f n' kvs')
    ∈ ⟨⟨Ra,Rb⟩rbt-rel, ⟨⟨Ra,Rb⟩prod-rel⟩list-rel⟩prod-rel
  ⟦n≤Suc (length kvs); (n,n')∈nat-rel; (kvs,kvs')∈⟨⟨Ra,Rb⟩prod-rel⟩list-rel⟧
  ⟹ (rbtreeify-g n kvs,rbtreeify-g n' kvs')
    ∈ ⟨⟨Ra,Rb⟩rbt-rel, ⟨⟨Ra,Rb⟩prod-rel⟩list-rel⟩prod-rel
  apply (induct n kvs and n kvs
    arbitrary: n' kvs' and n' kvs'
    rule: rbtreeify-induct)


  apply (simp only: pair-in-Id-conv)
  apply (simp (no-asm-use) only: rbtreeify-f-simps rbtreeify-g-simps)
  apply parametricity

  apply (elim list-relE prod-relE)
  apply (simp only: pair-in-Id-conv)
  apply hypsubst
  apply (simp (no-asm-use) only: rbtreeify-f-simps rbtreeify-g-simps)
  apply parametricity

  apply clarsimp
  apply (subgoal-tac (rbtreeify-f n kvs, rbtreeify-f n kvs'a)
    ∈ ⟨⟨Ra, Rb⟩rbt-rel, ⟨⟨Ra, Rb⟩prod-rel⟩list-rel⟩prod-rel)
  apply (clarsimp elim!: list-relE prod-relE)
  apply parametricity
  apply (rule refl)
  apply rprems
  apply (rule refl)
  apply assumption
```

```
  apply clarsimp
  apply (subgoal-tac (rbtreeify-f n kvs, rbtreeify-f n kvs'a)
    ∈ ⟨⟨Ra, Rb⟩rbt-rel, ⟨⟨Ra, Rb⟩prod-rel⟩list-rel⟩prod-rel)
  apply (clarsimp elim!: list-relE prod-relE)
  apply parametricity
  apply (rule refl)
  apply rprems
  apply (rule refl)
  apply assumption

  apply simp
  apply parametricity

  apply clarsimp
  apply parametricity

  apply clarsimp
  apply (subgoal-tac (rbtreeify-g n kvs, rbtreeify-g n kvs'a)
    ∈ ⟨⟨Ra, Rb⟩rbt-rel, ⟨⟨Ra, Rb⟩prod-rel⟩list-rel⟩prod-rel)
  apply (clarsimp elim!: list-relE prod-relE)
  apply parametricity
  apply (rule refl)
  apply parametricity
  apply (rule refl)

  apply clarsimp
  apply (subgoal-tac (rbtreeify-f n kvs, rbtreeify-f n kvs'a)
    ∈ ⟨⟨Ra, Rb⟩rbt-rel, ⟨⟨Ra, Rb⟩prod-rel⟩list-rel⟩prod-rel)
  apply (clarsimp elim!: list-relE prod-relE)
  apply parametricity
  apply (rule refl)
  apply parametricity
  apply (rule refl)
  done

lemma param-rbtreeify[param]:
  (rbtreeify, rbtreeify) ∈ ⟨⟨Ra,Rb⟩prod-rel⟩list-rel → ⟨Ra,Rb⟩rbt-rel
  unfolding rbtreeify-def[abs-def]
  apply parametricity
  by simp

lemma param-sunion-with[param]:
  fixes less
  shows ⟦ (less,less') ∈ Ra → Ra → Id;
    (f,f')∈(Ra→Rb→Rb→Rb); (a,a')∈⟨⟨Ra,Rb⟩prod-rel⟩list-rel;
    (b,b')∈⟨⟨Ra,Rb⟩prod-rel⟩list-rel ⟧
  ⟹ (ord.sunion-with less f a b, ord.sunion-with less' f' a' b') ∈
    ⟨⟨Ra,Rb⟩prod-rel⟩list-rel
  apply (induct f' a' b' arbitrary: f a b
```

rule: *ord.sunion-with.induct*[*of less'*])
apply (*elim-all list-relE prod-relE*)
apply (*simp-all only*: *ord.sunion-with.simps*)
apply *parametricity*
apply *simp-all*
done

lemma *skip-red-alt*:
RBT-Impl.skip-red t = (*case t of*
(*Branch color.R l k v r*) ⇒ *l*
| - ⇒ *t*)
by (*auto split*: *rbt.split color.split*)

function *compare-height* ::
('*a*, '*b*) *RBT-Impl.rbt* ⇒ ('*a*, '*b*) *RBT-Impl.rbt* ⇒ ('*a*, '*b*) *RBT-Impl.rbt* ⇒ ('*a*, '*b*) *RBT-Impl.rbt* ⇒ *RBT-Impl.compare*
where
compare-height sx s t tx =
(*case* (*RBT-Impl.skip-red sx*, *RBT-Impl.skip-red s*, *RBT-Impl.skip-red t*, *RBT-Impl.skip-red tx*) *of*
(*Branch - sx' - - -*, *Branch - s' - - -*, *Branch - t' - - -*, *Branch - tx' - - -*) ⇒
compare-height (*RBT-Impl.skip-black sx'*) *s' t'* (*RBT-Impl.skip-black tx'*)
| (-, *rbt.Empty*, -, *Branch - - - - -*) ⇒ *RBT-Impl.LT*
| (*Branch - - - - -*, -, *rbt.Empty*, -) ⇒ *RBT-Impl.GT*
| (*Branch - sx' - - -*, *Branch - s' - - -*, *Branch - t' - - -*, *rbt.Empty*) ⇒
compare-height (*RBT-Impl.skip-black sx'*) *s' t' rbt.Empty*
| (*rbt.Empty*, *Branch - s' - - -*, *Branch - t' - - -*, *Branch - tx' - - -*) ⇒
compare-height rbt.Empty s' t' (*RBT-Impl.skip-black tx'*)
| - ⇒ *RBT-Impl.EQ*)
by *pat-completeness auto*

lemma *skip-red-size*: *size* (*RBT-Impl.skip-red b*) ≤ *size b*
by (*auto simp add*: *skip-red-alt split*: *rbt.split color.split*)

lemma *skip-black-size*: *size* (*RBT-Impl.skip-black b*) ≤ *size b*
unfolding *RBT-Impl.skip-black-def*
apply (*auto*
simp add: *Let-def*
split: *rbt.split color.split*
)
using *skip-red-size*[*of b*]
apply *auto*
done

termination
apply (*relation*
measure (λ(*a*, *b*, *c*, *d*). *size a* + *size b* + *size c* + *size d*))
apply *rule*
apply (*auto*

```
    simp: Let-def
    split: rbt.splits color.splits
  )
  apply (smt rbt.size(4) skip-black-size skip-red-size)
  apply (smt rbt.size(4) skip-black-size skip-red-size)
  apply (smt rbt.size(4) skip-black-size skip-red-size)
  done

lemmas [simp del] = compare-height.simps

lemma compare-height-alt:
  RBT-Impl.compare-height sx s t tx = compare-height sx s t tx
  apply (induct sx s t tx rule: compare-height.induct)
  apply (subst RBT-Impl.compare-height.simps)
  apply (subst compare-height.simps)
  apply (auto split: rbt.split)

  apply (rprems, (intro conjI, (rule refl)+)+)+
  done

term RBT-Impl.skip-red
lemma param-skip-red[param]: (RBT-Impl.skip-red,RBT-Impl.skip-red)
  ∈ ⟨Rk,Rv⟩rbt-rel → ⟨Rk,Rv⟩rbt-rel
  unfolding skip-red-alt[abs-def] by parametricity

lemma param-skip-black[param]: (RBT-Impl.skip-black,RBT-Impl.skip-black)
  ∈ ⟨Rk,Rv⟩rbt-rel → ⟨Rk,Rv⟩rbt-rel
  unfolding RBT-Impl.skip-black-def[abs-def] by parametricity

term rbt-case
lemma param-rbt-case':
  assumes (t,t')∈⟨Rk,Rv⟩rbt-rel
  assumes ⟦t=rbt.Empty; t'=rbt.Empty⟧ ⟹ (fl,fl')∈R
  assumes ⋀c l k v r c' l' k' v' r'. ⟦
    t = Branch c l k v r; t' = Branch c' l' k' v' r';
    (c,c')∈color-rel;
    (l,l')∈⟨Rk,Rv⟩rbt-rel; (k,k')∈Rk; (v,v')∈Rv; (r,r')∈⟨Rk,Rv⟩rbt-rel
  ⟧ ⟹ (fb c l k v r, fb' c' l' k' v' r') ∈ R
  shows (rbt-case fl fb t, rbt-case fl' fb' t') ∈ R
  using assms by (auto split: rbt.split elim: rbt-rel-elims)

lemma compare-height-param-aux[param]:
  ⟦ (sx,sx')∈⟨Rk,Rv⟩rbt-rel; (s,s')∈⟨Rk,Rv⟩rbt-rel;
    (t,t')∈⟨Rk,Rv⟩rbt-rel; (tx,tx')∈⟨Rk,Rv⟩rbt-rel ⟧
  ⟹ (compare-height sx s t tx, compare-height sx' s' t' tx') ∈ compare-rel
  apply (induct sx' s' t' tx' arbitrary: sx s t tx
    rule: compare-height.induct)
  apply (subst (2) compare-height.simps)
  apply (subst compare-height.simps)
```

apply (*parametricity add*: *param-prod-case′ param-rbt-case′*,
(*simp only*: *Pair-eq*, *intro conjI*, (*rule refl*)+)+) []
done

lemma *compare-height-param*[*param*]:
(*RBT-Impl.compare-height*,*RBT-Impl.compare-height*) ∈
⟨*Rk*,*Rv*⟩*rbt-rel* → ⟨*Rk*,*Rv*⟩*rbt-rel* → ⟨*Rk*,*Rv*⟩*rbt-rel* → ⟨*Rk*,*Rv*⟩*rbt-rel*
→ *compare-rel*
unfolding *compare-height-alt*[*abs-def*]
by *parametricity*

lemma *param-rbt-union*[*param*]:
fixes *less*
assumes *param-less*[*param*]: (*less*,*less′*) ∈ *Ra* → *Ra* → *Id*
shows (*ord.rbt-union less*, *ord.rbt-union less′*)
∈ ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel* → ⟨*Ra*,*Rb*⟩*rbt-rel*
unfolding *ord.rbt-union-def*[*abs-def*] *ord.rbt-union-with-key-def*[*abs-def*]
ord.rbt-insert-with-key-def[*abs-def*]
unfolding *RBT-Impl.fold-def RBT-Impl.entries-def*
by *parametricity*

term *rm-iterateoi*
lemma *param-rm-iterateoi*[*param*]: (*rm-iterateoi*,*rm-iterateoi*)
∈ ⟨*Ra*,*Rb*⟩*rbt-rel* → (*Rc*→*Id*) → (⟨*Ra*,*Rb*⟩*prod-rel* → *Rc* → *Rc*) → *Rc* → *Rc*
unfolding *rm-iterateoi-def*
by (*parametricity*)

lemma *param-rm-reverse-iterateoi*[*param*]:
(*rm-reverse-iterateoi*,*rm-reverse-iterateoi*)
∈ ⟨*Ra*,*Rb*⟩*rbt-rel* → (*Rc*→*Id*) → (⟨*Ra*,*Rb*⟩*prod-rel* → *Rc* → *Rc*) → *Rc* → *Rc*
unfolding *rm-reverse-iterateoi-def*
by (*parametricity*)

lemma *param-color-eq*[*param*]:
(*op* =, *op* =)∈*color-rel*→*color-rel*→*Id*
by (*auto elim*: *color-rel.cases*)

lemma *param-color-of*[*param*]:
(*color-of*, *color-of*)∈⟨*Rk*,*Rv*⟩*rbt-rel*→*color-rel*
unfolding *color-of-def*
by *parametricity*

term *bheight*
lemma *param-bheight*[*param*]:
(*bheight*,*bheight*)∈⟨*Rk*,*Rv*⟩*rbt-rel*→*Id*
unfolding *bheight-def*
by (*parametricity*)

```
lemma inv1-param[param]: (inv1,inv1)∈⟨Rk,Rv⟩rbt-rel→Id
  unfolding inv1-def
  by (parametricity)

lemma inv2-param[param]: (inv2,inv2)∈⟨Rk,Rv⟩rbt-rel→Id
  unfolding inv2-def
  by (parametricity)

term ord.rbt-less
lemma rbt-less-param[param]: (ord.rbt-less,ord.rbt-less) ∈
  (Rk→Rk→Id) → Rk → ⟨Rk,Rv⟩rbt-rel → Id
  unfolding ord.rbt-less-prop[abs-def]
  apply (parametricity add: param-list-ball)
  unfolding RBT-Impl.keys-def RBT-Impl.entries-def
  apply (parametricity)
  done

term ord.rbt-greater
lemma rbt-greater-param[param]: (ord.rbt-greater,ord.rbt-greater) ∈
  (Rk→Rk→Id) → Rk → ⟨Rk,Rv⟩rbt-rel → Id
  unfolding ord.rbt-greater-prop[abs-def]
  apply (parametricity add: param-list-ball)
  unfolding RBT-Impl.keys-def RBT-Impl.entries-def
  apply (parametricity)
  done

lemma rbt-sorted-param[param]:
  (ord.rbt-sorted,ord.rbt-sorted)∈(Rk→Rk→Id)→⟨Rk,Rv⟩rbt-rel→Id
  unfolding ord.rbt-sorted-def[abs-def]
  by (parametricity)

lemma is-rbt-param[param]: (ord.is-rbt,ord.is-rbt) ∈
  (Rk→Rk→Id) → ⟨Rk,Rv⟩rbt-rel → Id
  unfolding ord.is-rbt-def[abs-def]
  by (parametricity)

definition rbt-map-rel′ lt = br (ord.rbt-lookup lt) (ord.is-rbt lt)

lemma (in linorder) rbt-map-impl:
  (rbt.Empty,Map.empty) ∈ rbt-map-rel′ op <
  (rbt-insert,λk v m. m(k↦v))
    ∈ Id → Id → rbt-map-rel′ op < → rbt-map-rel′ op <
  (rbt-lookup,λm k. m k) ∈ rbt-map-rel′ op < → Id → ⟨Id⟩option-rel
  (rbt-delete,λk m. m|‘(−{k})) ∈ Id → rbt-map-rel′ op < → rbt-map-rel′ op <
  (rbt-union,op ++)
    ∈ rbt-map-rel′ op < → rbt-map-rel′ op < → rbt-map-rel′ op <
  by (auto simp add:
    rbt-lookup-rbt-insert rbt-lookup-rbt-delete rbt-lookup-rbt-union
    rbt-union-is-rbt
```

rbt-map-rel′-def br-def)

lemma *sorted-by-rel-keys-true*[*simp*]: *sorted-by-rel* (λ(-,-) (-,-). *True*) *l*
apply (*induct l*)
apply *auto*
done

definition *rbt-map-rel-def-internal*:
rbt-map-rel lt Rk Rv ≡ ⟨*Rk*,*Rv*⟩*rbt-rel O rbt-map-rel′ lt*

lemma *rbt-map-rel-def*:
⟨*Rk*,*Rv*⟩*rbt-map-rel lt* ≡ ⟨*Rk*,*Rv*⟩*rbt-rel O rbt-map-rel′ lt*
by (*simp add*: *rbt-map-rel-def-internal relAPP-def*)

lemma (**in** *linorder*) *autoref-gen-rbt-empty*:
(*rbt.Empty*,*Map.empty*) ∈ ⟨*Rk*,*Rv*⟩*rbt-map-rel op* <
by (*auto simp*: *rbt-map-rel-def*
intro!: *rbt-map-impl rbt-rel-intros*)

lemma (**in** *linorder*) *autoref-gen-rbt-insert*:
fixes *less-impl*
assumes *param-less*: (*less-impl*,*op* <) ∈ *Rk* → *Rk* → *Id*
shows (*ord.rbt-insert less-impl*,λ*k v m*. *m*(*k*↦*v*)) ∈
Rk → *Rv* → ⟨*Rk*,*Rv*⟩*rbt-map-rel op* < → ⟨*Rk*,*Rv*⟩*rbt-map-rel op* <
apply (*intro fun-relI*)
unfolding *rbt-map-rel-def*
apply (*auto intro*!: *relcomp.intros*)
apply (*rule param-rbt-insert*[*OF param-less*, *param-fo*])
apply *assumption*+
apply (*rule rbt-map-impl*[*param-fo*])
apply (*rule IdI* | *assumption*)+
done

lemma (**in** *linorder*) *autoref-gen-rbt-lookup*:
fixes *less-impl*
assumes *param-less*: (*less-impl*,*op* <) ∈ *Rk* → *Rk* → *Id*
shows (*ord.rbt-lookup less-impl*, λ*m k*. *m k*) ∈
⟨*Rk*,*Rv*⟩*rbt-map-rel op* < → *Rk* → ⟨*Rv*⟩*option-rel*
unfolding *rbt-map-rel-def*
apply (*intro fun-relI*)
apply (*elim relcomp.cases*)
apply *hypsubst*
apply (*subst R-O-Id*[*symmetric*])
apply (*rule relcompI*)
apply (*rule param-rbt-lookup*[*OF param-less*, *param-fo*])

apply *assumption+*
apply (*subst option-rel-id-simp*[*symmetric*])
apply (*rule rbt-map-impl*[*param-fo*])
apply *assumption*
apply (*rule IdI*)
done

lemma (**in** *linorder*) *autoref-gen-rbt-delete*:
fixes *less-impl*
assumes *param-less*: (*less-impl*,*op* <) ∈ *Rk* → *Rk* → *Id*
shows (*ord.rbt-delete less-impl*, λ*k m*. *m* |‘(−{*k*})) ∈
Rk → ⟨*Rk*,*Rv*⟩*rbt-map-rel op* < → ⟨*Rk*,*Rv*⟩*rbt-map-rel op* <
unfolding *rbt-map-rel-def*
apply (*intro fun-relI*)
apply (*elim relcomp.cases*)
apply *hypsubst*
apply (*rule relcompI*)
apply (*rule param-rbt-delete*[*OF param-less*, *param-fo*])
apply *assumption+*
apply (*rule rbt-map-impl*[*param-fo*])
apply (*rule IdI*)
apply *assumption*
done

lemma (**in** *linorder*) *autoref-gen-rbt-union*:
fixes *less-impl*
assumes *param-less*: (*less-impl*,*op* <) ∈ *Rk* → *Rk* → *Id*
shows (*ord.rbt-union less-impl*, *op* ++) ∈
⟨*Rk*,*Rv*⟩*rbt-map-rel op* < → ⟨*Rk*,*Rv*⟩*rbt-map-rel op* < → ⟨*Rk*,*Rv*⟩*rbt-map-rel*
op <
unfolding *rbt-map-rel-def*
apply (*intro fun-relI*)
apply (*elim relcomp.cases*)
apply *hypsubst*
apply (*rule relcompI*)
apply (*rule param-rbt-union*[*OF param-less*, *param-fo*])
apply *assumption+*
apply (*rule rbt-map-impl*[*param-fo*])
apply *assumption+*
done

3.10.2 A linear ordering on red-black trees

abbreviation *rbt-to-list t* ≡ *it-to-list rm-iterateoi t*

lemma (**in** *linorder*) *rbt-to-list-correct*:
assumes *SORTED*: *rbt-sorted t*
shows *rbt-to-list t* = *sorted-list-of-map* (*rbt-lookup t*) (**is** *?tl* = -)
proof −

from *map-it-to-list-linord-correct*[**where** *it=rm-iterateoi*, *OF*
rm-iterateoi-correct[*OF SORTED*]
] **have**
M: *map-of ?tl = rbt-lookup t*
and *D*: *distinct* (*map fst ?tl*)
and *S*: *sorted* (*map fst ?tl*)
by (*simp-all*)

from *the-sorted-list-of-map*[*OF D S*] *M* **show** *?thesis*
by *simp*
qed

definition
cmp-rbt cmpk cmpv ≡ *cmp-img rbt-to-list* (*cmp-lex* (*cmp-prod cmpk cmpv*))

lemma (**in** *linorder*) *param-rbt-sorted-list-of-map*[*param*]:
shows (*rbt-to-list*, *sorted-list-of-map*) ∈
⟨*Rk*, *Rv*⟩*rbt-map-rel op* < → ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*
apply (*auto simp*: *rbt-map-rel-def rbt-map-rel′-def br-def*
rbt-to-list-correct[*symmetric*]
)
by (*parametricity*)

lemma *param-rbt-sorted-list-of-map′*[*param*]:
assumes *ELO*: *eq-linorder cmp′*
shows (*rbt-to-list*,*linorder.sorted-list-of-map* (*comp2le cmp′*)) ∈
⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*) → ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*
proof −
interpret *linorder comp2le cmp′ comp2lt cmp′*
using *ELO* **by** (*simp add*: *eq-linorder-class-conv*)
show *?thesis*
by *parametricity*
qed

lemma *rbt-linorder-impl*:
assumes *ELO*: *eq-linorder cmp′*
assumes [*param*]: (*cmp*,*cmp′*)∈*Rk*→*Rk*→*Id*
shows
(*cmp-rbt cmp*, *cmp-map cmp′*) ∈
(*Rv*→*Rv*→*Id*)
→ ⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*)
→ ⟨*Rk*,*Rv*⟩*rbt-map-rel* (*comp2lt cmp′*) → *Id*
proof −
interpret *linorder comp2le cmp′ comp2lt cmp′*
using *ELO* **by** (*simp add*: *eq-linorder-class-conv*)

show *?thesis*
unfolding *cmp-map-def*[*abs-def*] *cmp-rbt-def*[*abs-def*]
apply (*parametricity add*: *param-cmp-extend param-cmp-img*)

```
    unfolding rbt-map-rel-def[abs-def] rbt-map-rel′-def br-def
    by auto
qed

lemma color-rel-sv[relator-props]: single-valued color-rel
  by (auto intro!: single-valuedI elim: color-rel.cases)

lemma rbt-rel-sv-aux:
  assumes SK: single-valued Rk
  assumes SV: single-valued Rv
  assumes I1: (a,b)∈(⟨Rk, Rv⟩rbt-rel)
  assumes I2: (a,c)∈(⟨Rk, Rv⟩rbt-rel)
  shows b=c
  using I1 I2
  apply (induct arbitrary: c)
  apply (elim rbt-rel-elims)
  apply simp
  apply (elim rbt-rel-elims)
  apply (simp add: single-valuedD[OF color-rel-sv]
    single-valuedD[OF SK] single-valuedD[OF SV])
  done

lemma rbt-rel-sv[relator-props]:
  assumes SK: single-valued Rk
  assumes SV: single-valued Rv
  shows single-valued (⟨Rk, Rv⟩rbt-rel)
  by (auto intro: single-valuedI rbt-rel-sv-aux[OF SK SV])

lemma rbt-map-rel-sv[relator-props]:
  ⟦single-valued Rk; single-valued Rv⟧
  ⟹ single-valued (⟨Rk,Rv⟩rbt-map-rel lt)
  apply (auto simp: rbt-map-rel-def rbt-map-rel′-def)
  apply (rule single-valued-relcomp)
  apply (rule rbt-rel-sv, assumption+)
  apply (rule br-sv)
  done

lemmas [autoref-rel-intf] = REL-INTFI[of rbt-map-rel x i-map, standard]
```

3.10.3 Second Part: Binding

```
lemma autoref-rbt-empty[autoref-rules]:
  assumes ELO: SIDE-GEN-ALGO (eq-linorder cmp′)
  assumes [simplified,param]: GEN-OP cmp cmp′ (Rk→Rk→Id)
  shows (rbt.Empty,op-map-empty) ∈
    ⟨Rk,Rv⟩rbt-map-rel (comp2lt cmp′)
proof −
  interpret linorder comp2le cmp′   comp2lt cmp′
    using ELO by (simp add: eq-linorder-class-conv)
```

```
  show ?thesis
    by (simp) (rule autoref-gen-rbt-empty)
qed

lemma autoref-rbt-update[autoref-rules]:
  assumes ELO: SIDE-GEN-ALGO (eq-linorder cmp')
  assumes [simplified,param]: GEN-OP cmp cmp' (Rk→Rk→Id)
  shows (ord.rbt-insert (comp2lt cmp),op-map-update) ∈
    Rk→Rv→⟨Rk,Rv⟩rbt-map-rel (comp2lt cmp')
    → ⟨Rk,Rv⟩rbt-map-rel (comp2lt cmp')
proof −
  interpret linorder comp2le cmp'   comp2lt cmp'
    using ELO by (simp add: eq-linorder-class-conv)
  show ?thesis
    unfolding op-map-update-def[abs-def]
    apply (rule autoref-gen-rbt-insert)
    unfolding comp2lt-def[abs-def]
    by (parametricity)
qed

lemma autoref-rbt-lookup[autoref-rules]:
  assumes ELO: SIDE-GEN-ALGO (eq-linorder cmp')
  assumes [simplified,param]: GEN-OP cmp cmp' (Rk→Rk→Id)
  shows (λk t. ord.rbt-lookup (comp2lt cmp) t k, op-map-lookup) ∈
    Rk → ⟨Rk,Rv⟩rbt-map-rel (comp2lt cmp') → ⟨Rv⟩option-rel
proof −
  interpret linorder comp2le cmp'   comp2lt cmp'
    using ELO by (simp add: eq-linorder-class-conv)
  show ?thesis
    unfolding op-map-lookup-def[abs-def]
    apply (intro fun-relI)
    apply (rule autoref-gen-rbt-lookup[param-fo])
    apply (unfold comp2lt-def[abs-def]) []
    apply (parametricity)
    apply assumption+
    done
qed

lemma autoref-rbt-delete[autoref-rules]:
  assumes ELO: SIDE-GEN-ALGO (eq-linorder cmp')
  assumes [simplified,param]: GEN-OP cmp cmp' (Rk→Rk→Id)
  shows (ord.rbt-delete (comp2lt cmp),op-map-delete) ∈
    Rk → ⟨Rk,Rv⟩rbt-map-rel (comp2lt cmp')
      → ⟨Rk,Rv⟩rbt-map-rel (comp2lt cmp')
proof −
  interpret linorder comp2le cmp'   comp2lt cmp'
    using ELO by (simp add: eq-linorder-class-conv)
  show ?thesis
    unfolding op-map-delete-def[abs-def]
```

```
    apply (intro fun-relI)
    apply (rule autoref-gen-rbt-delete[param-fo])
    apply (unfold comp2lt-def[abs-def]) []
    apply (parametricity)
    apply assumption+
    done
qed

lemma autoref-rbt-union[autoref-rules]:
  assumes ELO: SIDE-GEN-ALGO (eq-linorder cmp′)
  assumes [simplified,param]: GEN-OP cmp cmp′ (Rk→Rk→Id)
  shows (ord.rbt-union (comp2lt cmp),op ++) ∈
    ⟨Rk,Rv⟩rbt-map-rel (comp2lt cmp′) → ⟨Rk,Rv⟩rbt-map-rel (comp2lt cmp′)
      → ⟨Rk,Rv⟩rbt-map-rel (comp2lt cmp′)
proof −
  interpret linorder comp2le cmp′    comp2lt cmp′
    using ELO by (simp add: eq-linorder-class-conv)
  show ?thesis
    apply (intro fun-relI)
    apply (rule autoref-gen-rbt-union[param-fo])
    apply (unfold comp2lt-def[abs-def]) []
    apply (parametricity)
    apply assumption+
    done
qed

lemma autoref-rbt-is-iterator[autoref-ga-rules]:
  assumes ELO: GEN-ALGO-tag (eq-linorder cmp′)

  shows is-map-to-sorted-list (comp2le cmp′) Rk Rv (rbt-map-rel (comp2lt cmp′))
    rbt-to-list
proof −
  interpret linorder comp2le cmp′    comp2lt cmp′
    using ELO by (simp add: eq-linorder-class-conv)

  show ?thesis
    unfolding is-map-to-sorted-list-def
      it-to-sorted-list-def
    apply auto
  proof −
    fix r m′
    assume (r, m′) ∈ ⟨Rk, Rv⟩rbt-map-rel (comp2lt cmp′)
    then obtain r′ where R1: (r,r′)∈⟨Rk,Rv⟩rbt-rel
      and R2: (r′,m′) ∈ rbt-map-rel′ (comp2lt cmp′)
      unfolding rbt-map-rel-def by blast

    from R2 have is-rbt r′ and M′: m′ = rbt-lookup r′
      unfolding rbt-map-rel′-def
      by (simp-all add: br-def)
```

```
    hence SORTED: rbt-sorted r'
      by (simp add: is-rbt-def)

    from map-it-to-list-linord-correct[where it = rm-iterateoi, OF
      rm-iterateoi-correct[OF SORTED]
    ] have
        M: map-of (rbt-to-list r') = rbt-lookup r'
        and D: distinct (map fst (rbt-to-list r'))
        and S: sorted (map fst (rbt-to-list r'))
      by (simp-all)

    show ∃ l'. (rbt-to-list r, l') ∈ ⟨⟨Rk, Rv⟩prod-rel⟩list-rel ∧
            distinct l' ∧
            map-to-set m' = set l' ∧
            sorted-by-rel (key-rel (comp2le cmp')) l'
    proof (intro exI conjI)
      from D show distinct (rbt-to-list r') by (rule distinct-mapI)
      from S show sorted-by-rel (key-rel (comp2le cmp')) (rbt-to-list r')
        unfolding key-rel-def[abs-def]
        by simp
      show (rbt-to-list r, rbt-to-list r') ∈ ⟨⟨Rk, Rv⟩prod-rel⟩list-rel
        by (parametricity add: R1)
      from M show map-to-set m' = set (rbt-to-list r')
        by (simp add: M' map-of-map-to-set[OF D])
    qed
  qed
qed


lemmas [autoref-ga-rules] = class-to-eq-linorder

lemma (in linorder) dflt-cmp-id:
  (dflt-cmp op ≤ op <, dflt-cmp op ≤ op <)∈Id→Id→Id
  by auto

lemmas [autoref-rules] = dflt-cmp-id

lemma rbt-linorder-autoref[autoref-rules]:
  assumes SIDE-GEN-ALGO (eq-linorder cmpk')
  assumes SIDE-GEN-ALGO (eq-linorder cmpv')
  assumes GEN-OP cmpk cmpk' (Rk→Rk→Id)
  assumes GEN-OP cmpv cmpv' (Rv→Rv→Id)
  shows
  (cmp-rbt cmpk cmpv, cmp-map cmpk' cmpv') ∈
      ⟨Rk,Rv⟩rbt-map-rel (comp2lt cmpk')
    → ⟨Rk,Rv⟩rbt-map-rel (comp2lt cmpk') → Id
  apply (intro fun-relI)
  apply (rule rbt-linorder-impl[param-fo])
```

```
  using assms
  apply simp-all
  done


lemma map-linorder-impl[autoref-ga-rules]:
  assumes GEN-ALGO-tag (eq-linorder cmpk)
  assumes GEN-ALGO-tag (eq-linorder cmpv)
  shows eq-linorder (cmp-map cmpk cmpv)
  using assms apply simp-all
  using map-ord-eq-linorder .

lemma set-linorder-impl[autoref-ga-rules]:
  assumes GEN-ALGO-tag (eq-linorder cmpk)
  shows eq-linorder (cmp-set cmpk)
  using assms apply simp-all
  using set-ord-eq-linorder .

lemma (in linorder) rbt-map-rel-finite-aux:
  finite-map-rel (⟨Rk,Rv⟩rbt-map-rel op <)
  unfolding finite-map-rel-def
  by (auto simp: rbt-map-rel-def rbt-map-rel'-def br-def)

lemma rbt-map-rel-finite[relator-props]:
  assumes ELO: GEN-ALGO-tag (eq-linorder cmpk)
  shows finite-map-rel (⟨Rk,Rv⟩rbt-map-rel (comp2lt cmpk))
proof −
  interpret linorder comp2le cmpk   comp2lt cmpk
    using ELO by (simp add: eq-linorder-class-conv)
  show ?thesis
    using rbt-map-rel-finite-aux .
qed

abbreviation
  dflt-rm-rel ≡ rbt-map-rel (comp2lt (dflt-cmp op ≤ op <))

lemmas [autoref-post-simps] = dflt-cmp-inv2 dflt-cmp-2inv

lemma [simp,autoref-post-simps]: ord.rbt-ins op < = rbt-ins
proof (intro ext)
  case goal1 thus ?case
    apply (induct x xa xb xc rule: rbt-ins.induct)
    apply (simp-all add: ord.rbt-ins.simps)
    done
qed

lemma [simp,autoref-post-simps]:
  ord.rbt-insert-with-key op < = rbt-insert-with-key
  ord.rbt-insert op < = rbt-insert
```

unfolding
ord.rbt-insert-with-key-def [*abs-def*] *rbt-insert-with-key-def* [*abs-def*]
ord.rbt-insert-def [*abs-def*] *rbt-insert-def* [*abs-def*]
by *simp-all*

lemma *autoref-comp2eq*[*autoref-rules-raw*]:
assumes *PRIO-TAG-GEN-ALGO*
assumes *ELC*: *SIDE-GEN-ALGO* (*eq-linorder cmp′*)
assumes [*simplified*,*param*]: *GEN-OP cmp cmp′* (*R→R→Id*)
shows (*comp2eq cmp*, *op* =) ∈ *R→R→Id*
proof −
from *ELC* **have** *1*: *eq-linorder cmp′* **by** *simp*
show *?thesis*
apply (*subst eq-linorder-comp2eq-eq*[*OF 1*,*symmetric*])
by *parametricity*
qed

lemma *pi′-rm*[*icf-proper-iteratorI*]:
proper-it′ rm-iterateoi rm-iterateoi
proper-it′ rm-reverse-iterateoi rm-reverse-iterateoi
apply (*rule proper-it′I*)
apply (*rule pi-rm*)
apply (*rule proper-it′I*)
apply (*rule pi-rm-rev*)
done

declare *pi′-rm*[*proper-it*]

lemmas *autoref-rbt-rules* =
autoref-rbt-empty
autoref-rbt-lookup
autoref-rbt-update
autoref-rbt-delete
autoref-rbt-union

lemmas *autoref-rbt-rules-linorder*[*autoref-rules-raw*] =
autoref-rbt-rules[**where** *Rk*=*Rk*::(-×-::*linorder*) *set*, *standard*]

end

Arrays with in-place updates **theory** *Diff-Array* **imports**
Assoc-List
../../Lib/Misc
../../Autoref/Autoref
../Intf/Intf-Comp
begin

datatype *'a array = Array 'a list*

3.10.4 primitive operations

definition *new-array* :: *'a ⇒ nat ⇒ 'a array*
where *new-array a n = Array (replicate n a)*

primrec *array-length* :: *'a array ⇒ nat*
where *array-length (Array a) = length a*

primrec *array-get* :: *'a array ⇒ nat ⇒ 'a*
where *array-get (Array a) n = a ! n*

primrec *array-set* :: *'a array ⇒ nat ⇒ 'a ⇒ 'a array*
where *array-set (Array A) n a = Array (A[n := a])*

definition *array-of-list* :: *'a list ⇒ 'a array*
where *array-of-list = Array*

— Grows array by *inc* elements initialized to value *x*.
primrec *array-grow* :: *'a array ⇒ nat ⇒ 'a ⇒ 'a array*
where *array-grow (Array A) inc x = Array (A @ replicate inc x)*

— Shrinks array to new size *sz*. Undefined if *sz > array-length*
primrec *array-shrink* :: *'a array ⇒ nat ⇒ 'a array*
where *array-shrink (Array A) sz = (*
if (sz > length A) then
undefined
else
Array (take sz A)
)

3.10.5 Derived operations

primrec *list-of-array* :: *'a array ⇒ 'a list*
where *list-of-array (Array a) = a*

primrec *assoc-list-of-array* :: *'a array ⇒ (nat × 'a) list*
where *assoc-list-of-array (Array a) = zip [0..<length a] a*

function *assoc-list-of-array-code* :: *'a array ⇒ nat ⇒ (nat × 'a) list*
where [*simp del*]:
assoc-list-of-array-code a n =
(if array-length a ≤ n then []
else (n, array-get a n) # assoc-list-of-array-code a (n + 1))
by *pat-completeness auto*
termination *assoc-list-of-array-code*
by(*relation measure (λp. (array-length (fst p) − snd p))) auto*

definition *array-map* :: *(nat ⇒ 'a ⇒ 'b) ⇒ 'a array ⇒ 'b array*

where $array\text{-}map\ f\ a = array\text{-}of\text{-}list\ (map\ (\lambda(i,\ v).\ f\ i\ v)\ (assoc\text{-}list\text{-}of\text{-}array\ a))$

definition $array\text{-}foldr :: (nat \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ array \Rightarrow 'b \Rightarrow 'b$
where $array\text{-}foldr\ f\ a\ b = foldr\ (\lambda(k,\ v).\ f\ k\ v)\ (assoc\text{-}list\text{-}of\text{-}array\ a)\ b$

definition $array\text{-}foldl :: (nat \Rightarrow 'b \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ array \Rightarrow 'b$
where $array\text{-}foldl\ f\ b\ a = foldl\ (\lambda b\ (k,\ v).\ f\ k\ b\ v)\ b\ (assoc\text{-}list\text{-}of\text{-}array\ a)$

3.10.6 Lemmas

lemma *array-length-new-array* [*simp*]:
$array\text{-}length\ (new\text{-}array\ a\ n) = n$
by(*simp add*: *new-array-def*)

lemma *array-length-array-set* [*simp*]:
$array\text{-}length\ (array\text{-}set\ a\ i\ e) = array\text{-}length\ a$
by(*cases a*) *simp*

lemma *array-get-new-array* [*simp*]:
$i < n \Longrightarrow array\text{-}get\ (new\text{-}array\ a\ n)\ i = a$
by(*simp add*: *new-array-def*)

lemma *array-get-array-set-same* [*simp*]:
$n < array\text{-}length\ A \Longrightarrow array\text{-}get\ (array\text{-}set\ A\ n\ a)\ n = a$
by(*cases A*) *simp*

lemma *array-get-array-set-other*:
$n \neq n' \Longrightarrow array\text{-}get\ (array\text{-}set\ A\ n\ a)\ n' = array\text{-}get\ A\ n'$
by(*cases A*) *simp*

lemma *list-of-array-grow* [*simp*]:
$list\text{-}of\text{-}array\ (array\text{-}grow\ a\ inc\ x) = list\text{-}of\text{-}array\ a\ @\ replicate\ inc\ x$
by (*cases a*) (*simp*)

lemma *array-grow-length* [*simp*]:
$array\text{-}length\ (array\text{-}grow\ a\ inc\ x) = array\text{-}length\ a + inc$
by (*cases a*)(*simp add*: *array-of-list-def*)

lemma *array-grow-get* [*simp*]:
$i < array\text{-}length\ a \Longrightarrow array\text{-}get\ (array\text{-}grow\ a\ inc\ x)\ i = array\text{-}get\ a\ i$
$[\![\ i \geq array\text{-}length\ a;\ \ i < array\text{-}length\ a + inc]\!] \Longrightarrow array\text{-}get\ (array\text{-}grow\ a\ inc\ x)\ i = x$
by (*cases a*, *simp add*: *nth-append*)+

lemma *list-of-array-shrink* [*simp*]:
$[\![\ s \leq array\text{-}length\ a]\!] \Longrightarrow list\text{-}of\text{-}array\ (array\text{-}shrink\ a\ s) = take\ s\ (list\text{-}of\text{-}array\ a)$
by (*cases a*) *simp*

lemma *array-shrink-get* [*simp*]:
  $[\![i < s; s \leq \textit{array-length } a]\!] \Longrightarrow \textit{array-get } (\textit{array-shrink } a\ s)\ i = \textit{array-get } a\ i$
by (*cases a*) (*simp*)

lemma *list-of-array-id* [*simp*]: *list-of-array* (*array-of-list l*) = *l*
by (*cases l*)(*simp-all add*: *array-of-list-def*)

lemma *map-of-assoc-list-of-array*:
  map-of (*assoc-list-of-array a*) *k* = (*if k* < *array-length a then Some* (*array-get a k*) *else None*)
by(*cases a*, *cases k* < *array-length a*)(*force simp add*: *set-zip*)+

lemma *length-assoc-list-of-array* [*simp*]:
  length (*assoc-list-of-array a*) = *array-length a*
by(*cases a*) *simp*

lemma *distinct-assoc-list-of-array*:
  distinct (*map fst* (*assoc-list-of-array a*))
by(*cases a*)(*auto*)

lemma *array-length-array-map* [*simp*]:
  array-length (*array-map f a*) = *array-length a*
by(*simp add*: *array-map-def array-of-list-def*)

lemma *array-get-array-map* [*simp*]:
  i < *array-length a* $\Longrightarrow$ *array-get* (*array-map f a*) *i* = *f i* (*array-get a i*)
by(*cases a*)(*simp add*: *array-map-def map-ran-conv-map array-of-list-def*)

lemma *array-foldr-foldr*:
  array-foldr (λ*n*. *f*) (*Array a*) *b* = *foldr f a b*
by(*simp add*: *array-foldr-def foldr-snd-zip*)

lemma *assoc-list-of-array-code-induct*:
  assumes *IH*: $\bigwedge n.\ (n < \textit{array-length } a \Longrightarrow P\ (\textit{Suc } n)) \Longrightarrow P\ n$
  shows *P n*
proof −
  have $a = a \longrightarrow P\ n$
    by(*rule assoc-list-of-array-code.induct*[**where** $P{=}\lambda a'\ n.\ a = a' \longrightarrow P\ n$])(*auto intro*: *IH*)
  thus *?thesis* **by** *simp*
qed

lemma *assoc-list-of-array-code* [*code*]:
  assoc-list-of-array a = *assoc-list-of-array-code a 0*
proof(*cases a*)
  case (*Array A*)
  { fix *n*
    have *zip* [*n*..<*length A*] (*drop n A*) = *assoc-list-of-array-code* (*Array A*) *n*
    proof(*induct n taking*: *Array A rule*: *assoc-list-of-array-code-induct*)

```
      case (1 n)
      show ?case
      proof(cases n < array-length (Array A))
        case False
        thus ?thesis by(simp add: assoc-list-of-array-code.simps)
      next
        case True
         hence zip [Suc n..<length A] (drop (Suc n) A) = assoc-list-of-array-code
(Array A) (Suc n)
          by(rule 1)
        moreover from True have n < length A by simp
        moreover then obtain a A′ where A: drop n A = a # A′ by(cases drop
n A) auto
        moreover with ‹n < length A› have [simp]: a = A ! n
       by(subst append-take-drop-id[symmetric, where n=n])(simp add: nth-append
min-def)
        moreover from A have drop (Suc n) A = A′
          by(induct A arbitrary: n)(simp-all add: drop-Cons split: nat.split-asm)
      ultimately show ?thesis by(subst upt-rec)(simp add: assoc-list-of-array-code.simps)
      qed
    qed }
  note this[of 0]
  with Array show ?thesis by simp
qed

lemma list-of-array-code [code]:
  list-of-array a = array-foldr (λn. Cons) a []
by(cases a)(simp add: array-foldr-foldr foldr-Cons)

lemma array-foldr-cong [fundef-cong]:
  ⟦ a = a′; b = b′;
    ⋀i b. i < array-length a ⟹ f i (array-get a i) b = g i (array-get a i) b ⟧
  ⟹ array-foldr f a b = array-foldr g a′ b′
by(cases a)(auto simp add: array-foldr-def set-zip intro!: foldr-cong)

lemma array-foldl-foldl:
  array-foldl (λn. f) b (Array a) = foldl f b a
by(simp add: array-foldl-def foldl-snd-zip)

lemma array-map-conv-foldl-array-set:
  assumes len: array-length A = array-length a
 shows array-map f a = foldl (λA (k, v). array-set A k (f k v)) A (assoc-list-of-array
a)
proof(cases a)
  case (Array xs)
  obtain ys where [simp]: A = Array ys by(cases A)
  with Array len have length xs ≤ length ys by simp
  hence foldr (λx y. array-set y (fst x) (f (fst x) (snd x)))
               (rev (zip [0..<length xs] xs)) (Array ys) =
```

Array (map (λx. f (fst x) (snd x)) (zip [0..<length xs] xs) @ drop (length xs) ys)
proof(induct xs arbitrary: ys rule: rev-induct)
case Nil **thus** ?case **by** simp
next
case (snoc x xs ys)
from ‹length (xs @ [x]) ≤ length ys› **have** length xs ≤ length ys **by** simp
hence foldr (λx y. array-set y (fst x) (f (fst x) (snd x)))
(rev (zip [0..<length xs] xs)) (Array ys) =
Array (map (λx. f (fst x) (snd x)) (zip [0..<length xs] xs) @ drop (length xs) ys)
by(rule snoc)
moreover from ‹length (xs @ [x]) ≤ length ys›
obtain y ys′ **where** ys: drop (length xs) ys = y # ys′
by(cases drop (length xs) ys) auto
moreover hence drop (Suc (length xs)) ys = ys′ **by**(auto dest: drop-eq-ConsD)
ultimately show ?case **by**(simp add: list-update-append)
qed
thus ?thesis **using** Array len
by(simp add: array-map-def split-beta array-of-list-def foldl-conv-foldr)
qed

3.10.7 Lemmas about empty arrays

lemma array-length-eq-0 [simp]:
array-length a = 0 ⟷ a = Array []
by(cases a) simp

lemma new-array-0 [simp]: new-array v 0 = Array []
by(simp add: new-array-def)

lemma array-of-list-Nil [simp]:
array-of-list [] = Array []
by(simp add: array-of-list-def)

lemma array-map-Nil [simp]:
array-map f (Array []) = Array []
by(simp add: array-map-def)

lemma array-foldl-Nil [simp]:
array-foldl f b (Array []) = b
by(simp add: array-foldl-def)

lemma array-foldr-Nil [simp]:
array-foldr f (Array []) b = b
by(simp add: array-foldr-def)

lemma prod-foldl-conv:
(foldl f a xs, foldl g b xs) = foldl (λ(a, b) x. (f a x, g b x)) (a, b) xs

by(*induct xs arbitrary*: *a b*) *simp-all*

lemma *prod-array-foldl-conv*:
(*array-foldl f b a*, *array-foldl g c a*) = *array-foldl* (λ*h* (*b*, *c*) *v*. (*f h b v*, *g h c v*)) (*b*, *c*) *a*
by(*cases a*)(*simp add*: *array-foldl-def foldl-map prod-foldl-conv split-def*)

lemma *array-foldl-array-foldr-comm*:
comp-fun-commute (λ(*h*, *v*) *b*. *f h b v*) ⟹ *array-foldl f b a* = *array-foldr* (λ*h v b*. *f h b v*) *a b*
by(*cases a*)(*simp add*: *array-foldl-def array-foldr-def split-def comp-fun-commute.foldr-conv-foldl*)

lemma *array-map-conv-array-foldl*:
array-map f a = *array-foldl* (λ*h a v*. *array-set a h* (*f h v*)) *a a*
proof(*cases a*)
case (*Array xs*)
def *a* == *xs*
hence *length xs* ≤ *length a* **by** *simp*
hence *foldl* (λ*a* (*k*, *v*). *array-set a k* (*f k v*))
(*Array a*) (*zip* [*0*..<*length xs*] *xs*)
= *Array* (*map* (λ(*k*, *v*). *f k v*) (*zip* [*0*..<*length xs*] *xs*) @ *drop* (*length xs*) *a*)
proof(*induct xs rule*: *rev-induct*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*snoc x xs*)
have *foldl* (λ*a* (*k*, *v*). *array-set a k* (*f k v*)) (*Array a*) (*zip* [*0*..<*length* (*xs* @ [*x*])] (*xs* @ [*x*])) =
array-set (*foldl* (λ*a* (*k*, *v*). *array-set a k* (*f k v*)) (*Array a*) (*zip* [*0*..<*length xs*] *xs*))
(*length xs*) (*f* (*length xs*) *x*) **by** *simp*
also from ‹*length* (*xs* @ [*x*]) ≤ *length a*› **have** *length xs* ≤ *length a* **by** *simp*
hence *foldl* (λ*a* (*k*, *v*). *array-set a k* (*f k v*)) (*Array a*) (*zip* [*0*..<*length xs*] *xs*) =
Array (*map* (λ(*k*, *v*). *f k v*) (*zip* [*0*..<*length xs*] *xs*) @ *drop* (*length xs*) *a*)
by(*rule snoc*)
also note *array-set.simps*
also have (*map* (λ(*k*, *v*). *f k v*) (*zip* [*0*..<*length xs*] *xs*) @ *drop* (*length xs*) *a*) [*length xs* := *f* (*length xs*) *x*] =
map (λ(*k*, *v*). *f k v*) (*zip* [*0*..<*length xs*] *xs*) @ (*drop* (*length xs*) *a*[*0* := *f* (*length xs*) *x*])
by(*simp add*: *list-update-append*)
also from ‹*length* (*xs* @ [*x*]) ≤ *length a*›
have *drop* (*length xs*) *a*[*0* := *f* (*length xs*) *x*] =
f (*length xs*) *x* # *drop* (*Suc* (*length xs*)) *a*
by(*simp add*: *upd-conv-take-nth-drop*)
also have *map* (λ(*k*, *v*). *f k v*) (*zip* [*0*..<*length xs*] *xs*) @ *f* (*length xs*) *x* # *drop* (*Suc* (*length xs*)) *a* =
(*map* (λ(*k*, *v*). *f k v*) (*zip* [*0*..<*length xs*] *xs*) @ [*f* (*length xs*) *x*]) @ *drop* (*Suc* (*length xs*)) *a* **by** *simp*

also have ... = *map* (λ(*k*, *v*). *f k v*) (*zip* [*0*..<*length* (*xs* @ [*x*])] (*xs* @ [*x*])) @ *drop* (*length* (*xs* @ [*x*])) *a*
by(*simp*)
finally show *?case* **.**
qed
with *a-def Array* **show** *?thesis*
by(*simp add*: *array-foldl-def array-map-def array-of-list-def*)
qed

lemma *array-foldl-new-array*:
array-foldl f b (*new-array a n*) = *foldl* (λ*b* (*k*, *v*). *f k b v*) *b* (*zip* [*0*..<*n*] (*replicate n a*))
by(*simp add*: *new-array-def array-foldl-def*)

3.10.8 Parametricity lemmas

lemma *array-rec-is-case*[*simp*]: *array-rec*=*array-case*
apply (*intro ext*)
apply (*auto split*: *array.split*)
done

definition *array-rel-def-internal*:
array-rel R ≡
{(*Array xs*, *Array ys*)|*xs ys*. (*xs*,*ys*) ∈ ⟨*R*⟩*list-rel*}

lemma *array-rel-def*:
⟨*R*⟩*array-rel* ≡ {(*Array xs*, *Array ys*)|*xs ys*. (*xs*,*ys*) ∈ ⟨*R*⟩*list-rel*}
unfolding *array-rel-def-internal relAPP-def* **.**

lemma *array-relD*:
(*Array l*, *Array l′*) ∈ ⟨*R*⟩*array-rel* ⟹ (*l*,*l′*) ∈ ⟨*R*⟩*list-rel*
by (*simp add*: *array-rel-def*)

lemma *array-rel-alt*:
⟨*R*⟩*array-rel* =
{ (*Array l*, *l*) | *l*. *True* }
O ⟨*R*⟩*list-rel*
O {(*l*,*Array l*) | *l*. *True*}
by (*auto simp*: *array-rel-def*)

lemma *array-rel-sv*[*relator-props*]:
shows *single-valued R* ⟹ *single-valued* (⟨*R*⟩*array-rel*)
unfolding *array-rel-alt*
apply (*intro relator-props*)
apply (*auto intro*: *single-valuedI*)
done

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

apply (*intro fun-relI*)
apply (*simp add*: *array-rel-def*)
done

lemma *param-array-rec*[*param*]:
(*array-rec*,*array-rec*) ∈ (⟨*Ra*⟩*list-rel* → *Rb*) → ⟨*Ra*⟩*array-rel* → *Rb*
apply (*intro fun-relI*)
apply (*rename-tac f f′ a a′*, *case-tac a*, *case-tac a′*)
apply (*auto dest*: *fun-relD array-relD*)
done

lemma *param-array-case*[*param*]:
(*array-case*,*array-case*) ∈ (⟨*Ra*⟩*list-rel* → *Rb*) → ⟨*Ra*⟩*array-rel* → *Rb*
apply (*clarsimp split*: *array.split*)
apply (*drule array-relD*)
by *parametricity*

lemma *param-array-case1′*:
assumes (*a*,*a′*)∈⟨*Ra*⟩*array-rel*
assumes ⋀*l l′*. ⟦ *a*=*Array l*; *a′*=*Array l′*; (*l*,*l′*)∈⟨*Ra*⟩*list-rel* ⟧
⟹ (*f l*,*f′ l′*) ∈ *Rb*
shows (*array-case f a*,*array-case f′ a′*) ∈ *Rb*
using *assms*
apply (*clarsimp split*: *array.split*)
apply (*drule array-relD*)
apply *parametricity*
by (*rule refl*)+

lemmas *param-array-case2′* = *param-array-case1′*[*folded array-rec-is-case*]

lemmas *param-array-case′* = *param-array-case1′ param-array-case2′*

lemma *param-array-length*[*param*]:
(*array-length*,*array-length*) ∈ ⟨*Rb*⟩*array-rel* → *nat-rel*
unfolding *array-length-def*
by *parametricity*

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

lemma *array-rel-imp-same-length*:
(*a*, *a′*) ∈ ⟨*R*⟩*array-rel* ⟹ *array-length a* = *array-length a′*
apply (*cases a*, *cases a′*)
apply (*auto simp add*: *list-rel-imp-same-length dest*!: *array-relD*)
done

lemma *param-array-get*[*param*]:
assumes *I*: *i*<*array-length a*

```
  assumes IR: (i,i')∈nat-rel
  assumes AR: (a,a')∈⟨R⟩array-rel
  shows (array-get a i, array-get a' i') ∈ R
proof −
  obtain l l' where [simp]: a = Array l    a' = Array l'
      by (cases a, cases a', simp-all)
  from AR have LR: (l,l') ∈ ⟨R⟩list-rel by (force dest!: array-relD)
  thus ?thesis using assms
    unfolding array-get-def
    apply (auto intro!: param-nth[param-fo] dest: list-rel-imp-same-length)
    done
qed

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

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

lemma param-array-shrink[param]:
  assumes N: array-length a ≥ n
  assumes NR: (n,n')∈nat-rel
  assumes AR: (a,a')∈⟨R⟩array-rel
  shows (array-shrink a n, array-shrink a' n') ∈ ⟨R⟩ array-rel
proof−
  obtain l l' where [simp]: a = Array l    a' = Array l'
      by (cases a, cases a', simp-all)
  from AR have LR: (l,l') ∈ ⟨R⟩list-rel
    by (auto dest: array-relD)
  with assms show ?thesis by (auto intro:
      param-Array[param-fo] param-take[param-fo]
      dest: array-rel-imp-same-length
    )
qed

lemma param-assoc-list-of-array[param]:
  (assoc-list-of-array, assoc-list-of-array) ∈
      ⟨R⟩ array-rel → ⟨⟨nat-rel,R⟩prod-rel⟩list-rel
  unfolding assoc-list-of-array-def[abs-def] by parametricity

lemma param-array-map[param]:
  (array-map, array-map) ∈
      (nat-rel → Ra → Rb) → ⟨Ra⟩array-rel → ⟨Rb⟩array-rel
  unfolding array-map-def[abs-def] by parametricity

lemma param-array-foldr[param]:
  (array-foldr, array-foldr) ∈
```

(*nat-rel* → *Ra* → *Rb* → *Rb*) → ⟨*Ra*⟩*array-rel* → *Rb* → *Rb*
unfolding *array-foldr-def*[*abs-def*] **by** *parametricity*

lemma *param-array-foldl*[*param*]:
(*array-foldl*, *array-foldl*) ∈
(*nat-rel* → *Rb* → *Ra* → *Rb*) → *Rb* → ⟨*Ra*⟩*array-rel* → *Rb*
unfolding *array-foldl-def*[*abs-def*] **by** *parametricity*

3.10.9 Code Generator Setup

Code generator setup for Haskell

code-type *array*
(*Haskell Array.ArrayType/ -*)

code-reserved *Haskell array-of-list*

code-include *Haskell Array* ⟨⟨

```
--import qualified Data.Array.Diff as Arr;
import qualified Data.Array as Arr;
import Data.Array.IArray;
import Nat;

instance Ix Nat where {
    range (Nat a, Nat b) = map Nat (range (a, b));
    index (Nat a, Nat b) (Nat c) = index (a,b) c;
    inRange (Nat a, Nat b) (Nat c) = inRange (a, b) c;
    rangeSize (Nat a, Nat b) = rangeSize (a, b);
};

--type ArrayType = Arr.DiffArray Nat;
type ArrayType = Arr.Array Nat;

-- we need to start at 1 and not 0, because the empty array
-- is modelled by having s > e for (s,e) = bounds
-- and as we are in Nat, 0 is the smallest number

array-of-size :: Nat -> [e] -> ArrayType e;
array-of-size n = Arr.listArray (1, n);

new-array :: e -> Nat -> ArrayType e;
new-array a n = array-of-size n (repeat a);

array-length :: ArrayType e -> Nat;
array-length a = let (s, e) = bounds a in if s > e then 0 else e - s + 1;
-- the `if` is actually needed, because in Nat we have s > e --> e - s + 1 = 1

array-get :: ArrayType e -> Nat -> e;
array-get a i = a ! (i + 1);
```

array-set :: *ArrayType e* −> *Nat* −> *e* −> *ArrayType e*;
array-set a i e = *a* // [(*i* + *1*, *e*)];

array-of-list :: [*e*] −> *ArrayType e*;
array-of-list xs = *array-of-size* (*fromInteger* (*toInteger* (*length xs* − *1*))) *xs*;

array-grow :: *ArrayType e* −> *Nat* −> *e* −> *ArrayType e*;
array-grow a i x = *let* (*s*, *e*) = *bounds a in Arr.listArray* (*s*, *e*+*i*) (*Arr.elems a* ++ *repeat x*);

array-shrink :: *ArrayType e* −> *Nat* −> *ArrayType e*;
array-shrink a sz = *if sz* > *array-length a then undefined else array-of-size sz* (*Arr.elems a*);
⟫

code-const *Array* (*Haskell Array.array′-of′-list*)
code-const *new-array* (*Haskell Array.new′-array*)
code-const *array-length* (*Haskell Array.array′-length*)
code-const *array-get* (*Haskell Array.array′-get*)
code-const *array-set* (*Haskell Array.array′-set*)
code-const *array-of-list* (*Haskell Array.array′-of′-list*)
code-const *array-grow* (*Haskell Array.array′-grow*)
code-const *array-shrink* (*Haskell Array.array′-shrink*)

Code Generator Setup For SML

We have the choice between single-threaded arrays, that raise an exception if an old version is accessed, and truly functional arrays, that update the array in place, but store undo-information to restore old versions.

code-include *SML STArray*
⟪*structure STArray* = *struct*

datatype ′a Cell = *Invalid* | *Value of ′a array*;

exception AccessedOldVersion;

type ′a array = *′a Cell Unsynchronized.ref*;

fun fromList l = *Unsynchronized.ref* (*Value* (*Array.fromList l*));
fun array (*size*, *v*) = *Unsynchronized.ref* (*Value* (*Array.array* (*size*,*v*)));
fun tabulate (*size*, *f*) = *Unsynchronized.ref* (*Value* (*Array.tabulate*(*size*, *f*)));
fun sub (*Unsynchronized.ref Invalid*, *idx*) = *raise AccessedOldVersion* |
  sub (*Unsynchronized.ref* (*Value a*), *idx*) = *Array.sub* (*a*,*idx*);
fun update (*aref*,*idx*,*v*) =
 case aref of
  (*Unsynchronized.ref Invalid*) => *raise AccessedOldVersion* |
  (*Unsynchronized.ref* (*Value a*)) => (
   aref := *Invalid*;
   Array.update (*a*,*idx*,*v*);

```
    Unsynchronized.ref (Value a)
  );

fun length (Unsynchronized.ref Invalid) = raise AccessedOldVersion |
  length (Unsynchronized.ref (Value a)) = Array.length a

fun grow (aref, i, x) = case aref of
 (Unsynchronized.ref Invalid) => raise AccessedOldVersion |
 (Unsynchronized.ref (Value a)) => (
  let val len=Array.length a;
      val na = Array.array (len+i,x)
  in
    aref := Invalid;
    Array.copy {src=a, dst=na, di=0};
    Unsynchronized.ref (Value na)
  end
  );

fun shrink (aref, sz) = case aref of
 (Unsynchronized.ref Invalid) => raise AccessedOldVersion |
 (Unsynchronized.ref (Value a)) => (
  if sz > Array.length a then
    raise Size
  else (
    aref:=Invalid;
    Unsynchronized.ref (Value (Array.tabulate (sz,fn i => Array.sub (a,i))))
  )
 );

structure IsabelleMapping = struct
type 'a ArrayType = 'a array;

fun new-array (a:'a) (n:int) = array (n, a);

fun array-length (a:'a ArrayType) = length a;

fun array-get (a:'a ArrayType) (i:int) = sub (a, i);

fun array-set (a:'a ArrayType) (i:int) (e:'a) = update (a, i, e);

fun array-of-list (xs:'a list) = fromList xs;

fun array-grow (a:'a ArrayType) (i:int) (x:'a) = grow (a, i, x);

fun array-shrink (a:'a ArrayType) (sz:int) = shrink (a,sz);

end;

end;
```

```
structure FArray = struct
  datatype 'a Cell = Value of 'a Array.array | Upd of (int*'a*'a Cell Unsynchro-
nized.ref);

  type 'a array = 'a Cell Unsynchronized.ref;

  fun array (size,v) = Unsynchronized.ref (Value (Array.array (size,v)));
  fun tabulate (size, f) = Unsynchronized.ref (Value (Array.tabulate(size, f)));
  fun fromList l = Unsynchronized.ref (Value (Array.fromList l));

  fun sub (Unsynchronized.ref (Value a), idx) = Array.sub (a,idx) |
      sub (Unsynchronized.ref (Upd (i,v,cr)),idx) =
        if i=idx then v
        else sub (cr,idx);

  fun length (Unsynchronized.ref (Value a)) = Array.length a |
      length (Unsynchronized.ref (Upd (i,v,cr))) = length cr;

  fun realize-aux (aref, v) =
    case aref of
      (Unsynchronized.ref (Value a)) => (
        let
          val len = Array.length a;
          val a' = Array.array (len,v);
        in
          Array.copy {src=a, dst=a', di=0};
          Unsynchronized.ref (Value a')
        end
      ) |
      (Unsynchronized.ref (Upd (i,v,cr))) => (
        let val res=realize-aux (cr,v) in
          case res of
            (Unsynchronized.ref (Value a)) => (Array.update (a,i,v); res)
        end
      );

  fun realize aref =
    case aref of
      (Unsynchronized.ref (Value -)) => aref |
      (Unsynchronized.ref (Upd (i,v,cr))) => realize-aux(aref,v);

  fun update (aref,idx,v) =
    case aref of
      (Unsynchronized.ref (Value a)) => (
        let val nref=Unsynchronized.ref (Value a) in
          aref := Upd (idx,Array.sub(a,idx),nref);
          Array.update (a,idx,v);
          nref
```

```
        end
      ) |
      (Unsynchronized.ref (Upd -)) =>
        let val ra = realize-aux(aref,v) in
          case ra of
            (Unsynchronized.ref (Value a)) => Array.update (a,idx,v);
          ra
        end
      ;

  fun grow (aref, inc, x) = case aref of
    (Unsynchronized.ref (Value a)) => (
      let val len=Array.length a;
          val na = Array.array (len+inc,x)
      in
        Array.copy {src=a, dst=na, di=0};
        Unsynchronized.ref (Value na)
      end
      )
  | (Unsynchronized.ref (Upd -)) => (
    grow (realize aref, inc, x)
  );

  fun shrink (aref, sz) = case aref of
    (Unsynchronized.ref (Value a)) => (
      if sz > Array.length a then
        raise Size
      else (
        Unsynchronized.ref (Value (Array.tabulate (sz,fn i => Array.sub (a,i))))
      )
    ) |
    (Unsynchronized.ref (Upd -)) => (
      shrink (realize aref,sz)
    );

structure IsabelleMapping = struct
type 'a ArrayType = 'a array;

fun new-array (a:'a) (n:int) = array (n, a);

fun array-length (a:'a ArrayType) = length a;

fun array-get (a:'a ArrayType) (i:int) = sub (a, i);

fun array-set (a:'a ArrayType) (i:int) (e:'a) = update (a, i, e);

fun array-of-list (xs:'a list) = fromList xs;

fun array-grow (a:'a ArrayType) (i:int) (x:'a) = grow (a, i, x);
```

fun array-shrink (*a*:′*a ArrayType*) (*sz*:*int*) = *shrink* (*a*,*sz*);

end;
end;

⟫

code-type *array*
(*SML* -/ *FArray.IsabelleMapping.ArrayType*)

code-const *Array* (*SML FArray.IsabelleMapping.array*′*-of*′*-list*)
code-const *new-array* (*SML FArray.IsabelleMapping.new*′*-array*)
code-const *array-length* (*SML FArray.IsabelleMapping.array*′*-length*)
code-const *array-get* (*SML FArray.IsabelleMapping.array*′*-get*)
code-const *array-set* (*SML FArray.IsabelleMapping.array*′*-set*)
code-const *array-grow* (*SML FArray.IsabelleMapping.array*′*-grow*)
code-const *array-shrink* (*SML FArray.IsabelleMapping.array*′*-shrink*)
code-const *array-of-list* (*SML FArray.IsabelleMapping.array*′*-of*′*-list*)

end

3.11 Array-Based Maps with Natural Number Keys

theory *Impl-Array-Map*
imports
../../Autoref/Autoref
../Lib/Diff-Array
../Gen/Gen-Iterator
../Gen/Gen-Map
../Intf/Intf-Comp
../Intf/Intf-Map
begin

type-synonym ′*v iam* = ′*v option array*

3.11.1 Definitions

definition *iam-α* :: ′*v iam* ⇒ *nat* ⇀ ′*v* **where**
iam-α a i ≡ *if i* < *array-length a then array-get a i else None*

abbreviation *iam-invar* :: ′*v iam* ⇒ *bool* **where** *iam-invar* ≡ λ-. *True*

definition *iam-empty* :: *unit* ⇒ ′*v iam*
where *iam-empty* ≡ λ-::*unit*. *array-of-list* []

definition *iam-lookup* :: *nat* ⇒ ′*v iam* ⇀ ′*v*

where *iam-lookup k a* ≡ *iam-α a k*

definition *iam-increment* (*l*::*nat*) *idx* ≡
max (*idx* + *1* − *l*) (*2* ∗ *l* + *3*)

lemma *incr-correct*: ¬ *idx* < *l* ⟹ *idx* < *l* + *iam-increment l idx*
unfolding *iam-increment-def* **by** *auto*

definition *iam-update* :: *nat* ⇒ ′*v* ⇒ ′*v iam* ⇒ ′*v iam*
where *iam-update k v a* ≡ *let*
l = *array-length a*;
a = *if k* < *l then a else array-grow a* (*iam-increment l k*) *None*
in
array-set a k (*Some v*)

definition *iam-delete* :: *nat* ⇒ ′*v iam* ⇒ ′*v iam*
where *iam-delete k a* ≡
if k<*array-length a then array-set a k None else a*

primrec *iam-iteratei-aux*
:: *nat* ⇒ (′*v iam*) ⇒ (′σ⇒*bool*) ⇒ (*nat* × ′*v*⇒′σ⇒′σ) ⇒ ′σ ⇒ ′σ
where
iam-iteratei-aux 0 a c f σ = σ
| *iam-iteratei-aux* (*Suc i*) *a c f* σ = (
if c σ *then*
iam-iteratei-aux i a c f (
case array-get a i of None ⇒ σ | *Some x* ⇒ *f* (*i*, *x*) σ
)
else σ)

definition *iam-iteratei* :: ′*v iam* ⇒ (*nat* × ′*v*,′σ) *set-iterator* **where**
iam-iteratei a = *iam-iteratei-aux* (*array-length a*) *a*

3.11.2 Parametricity

definition *iam-rel-def-internal*:
iam-rel R ≡ ⟨⟨*R*⟩ *option-rel*⟩ *array-rel*
lemma *iam-rel-def*: ⟨*R*⟩ *iam-rel* = ⟨⟨*R*⟩ *option-rel*⟩ *array-rel*
by (*simp add*: *iam-rel-def-internal relAPP-def*)

lemma *iam-rel-sv*[*relator-props*]:
single-valued Rv ⟹ *single-valued* (⟨*Rv*⟩*iam-rel*)
unfolding *iam-rel-def*
by *tagged-solver*

lemma *param-iam-α*[*param*]:
(*iam-α*, *iam-α*) ∈ ⟨*R*⟩ *iam-rel* → *nat-rel* → ⟨*R*⟩ *option-rel*
unfolding *iam-α-def*[*abs-def*] *iam-rel-def* **by** *parametricity*

lemma *param-iam-invar*[*param*]:
(*iam-invar*, *iam-invar*) ∈ ⟨*R*⟩ *iam-rel* → *bool-rel*
unfolding *iam-rel-def* **by** *parametricity*

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

lemma *param-iam-lookup*[*param*]:
(*iam-lookup*, *iam-lookup*) ∈ *nat-rel* → ⟨*R*⟩*iam-rel* → ⟨*R*⟩*option-rel*
unfolding *iam-lookup-def*[*abs-def*]
by *parametricity*

lemma *param-iam-increment*[*param*]:
(*iam-increment*, *iam-increment*) ∈ *nat-rel* → *nat-rel* → *nat-rel*
unfolding *iam-increment-def*[*abs-def*]
by *simp*

lemma *param-iam-update*[*param*]:
(*iam-update*, *iam-update*) ∈ *nat-rel* → *R* → ⟨*R*⟩*iam-rel* → ⟨*R*⟩*iam-rel*
unfolding *iam-update-def*[*abs-def*] *iam-rel-def Let-def*
apply *parametricity*
done

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

lemma *param-iam-iteratei-aux*[*param*]:
assumes *I*: *i* ≤ *array-length a*
assumes *IR*: (*i*,*i'*) ∈ *nat-rel*
assumes *AR*: (*a*,*a'*) ∈ ⟨*Ra*⟩*iam-rel*
assumes *CR*: (*c*,*c'*) ∈ *Rb* → *bool-rel*
assumes *FR*: (*f*,*f'*) ∈ ⟨*nat-rel*,*Ra*⟩*prod-rel* → *Rb* → *Rb*
assumes *σR*: (*σ*,*σ'*) ∈ *Rb*
shows (*iam-iteratei-aux i a c f σ*, *iam-iteratei-aux i' a' c' f' σ'*) ∈ *Rb*
using *assms*
unfolding *iam-rel-def*
apply (*induct i' arbitrary*: *i σ σ'*)
apply (*simp-all only*: *pair-in-Id-conv iam-iteratei-aux.simps*)
apply *parametricity*
apply *simp-all*
done

lemma *param-iam-iteratei*[*param*]:
(*iam-iteratei*,*iam-iteratei*) ∈ ⟨*Ra*⟩*iam-rel* → (*Rb* → *bool-rel*) →
((⟨*nat-rel*,*Ra*⟩*prod-rel* → *Rb* → *Rb*) → *Rb* → *Rb*

unfolding *iam-iteratei-def* [*abs-def*]
by *parametricity* (*simp-all add*: *iam-rel-def*)

3.11.3 Correctness

definition *iam-rel′* ≡ *br iam-α iam-invar*

lemma *iam-empty-correct*:
(*iam-empty* (), *Map.empty*) ∈ *iam-rel′*
by (*simp add*: *iam-rel′-def br-def iam-α-def* [*abs-def*] *iam-empty-def*)

lemma *iam-update-correct*:
(*iam-update*,*op-map-update*) ∈ *nat-rel* → *Id* → *iam-rel′* → *iam-rel′*
by (*auto simp*: *iam-rel′-def br-def Let-def array-get-array-set-other*
incr-correct iam-α-def [*abs-def*] *iam-update-def*)

lemma *iam-lookup-correct*:
(*iam-lookup*,*op-map-lookup*) ∈ *Id* → *iam-rel′* → ⟨*Id*⟩*option-rel*
by (*auto simp*: *iam-rel′-def br-def iam-lookup-def* [*abs-def*])

lemma *array-get-set-iff*: *i*<*array-length a* ⟹
array-get (*array-set a i x*) *j* = (*if i*=*j then x else array-get a j*)
by (*auto simp*: *array-get-array-set-other*)

lemma *iam-delete-correct*:
(*iam-delete*,*op-map-delete*) ∈ *Id* → *iam-rel′* → *iam-rel′*
unfolding *iam-α-def* [*abs-def*] *iam-delete-def* [*abs-def*] *iam-rel′-def br-def*
by (*auto simp*: *Let-def array-get-set-iff*)

definition *iam-map-rel-def-internal*:
iam-map-rel Rk Rv ≡
if Rk=*nat-rel then* ⟨*Rv*⟩*iam-rel O iam-rel′ else* {}
lemma *iam-map-rel-def*:
⟨*nat-rel*,*Rv*⟩*iam-map-rel* ≡ ⟨*Rv*⟩*iam-rel O iam-rel′*
unfolding *iam-map-rel-def-internal relAPP-def* **by** *simp*

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

lemma *iam-map-rel-sv* [*relator-props*]:
single-valued Rv ⟹ *single-valued* (⟨*nat-rel*,*Rv*⟩*iam-map-rel*)
unfolding *iam-map-rel-def iam-rel′-def* **by** *tagged-solver*

lemma *iam-empty-impl*:
(*iam-empty* (), *op-map-empty*) ∈ ⟨*nat-rel*,*R*⟩*iam-map-rel*
unfolding *iam-map-rel-def op-map-empty-def*
apply (*intro relcompI*)
apply (*rule param-iam-empty* [*THEN fun-relD*], *simp*)

```
  apply (rule iam-empty-correct)
  done


lemma iam-lookup-impl:
    (iam-lookup, op-map-lookup)
  ∈ nat-rel → ⟨nat-rel,R⟩iam-map-rel → ⟨R⟩option-rel
unfolding iam-map-rel-def
apply (intro fun-relI)
apply (elim relcompE)
apply (frule iam-lookup-correct[param-fo], assumption)
apply (frule param-iam-lookup[param-fo], assumption)
apply simp
done

lemma iam-update-impl:
   (iam-update, op-map-update) ∈
     nat-rel → R → ⟨nat-rel,R⟩iam-map-rel → ⟨nat-rel,R⟩iam-map-rel
  unfolding iam-map-rel-def
  apply (intro fun-relI, elim relcompEpair, intro relcompI)
  apply (erule (2) param-iam-update[param-fo])
  apply (rule iam-update-correct[param-fo])
  apply simp-all
  done

lemma iam-delete-impl:
    (iam-delete, op-map-delete) ∈
        nat-rel → ⟨nat-rel,R⟩iam-map-rel → ⟨nat-rel,R⟩iam-map-rel
  unfolding iam-map-rel-def
  apply (intro fun-relI, elim relcompEpair, intro relcompI)
  apply (erule (1) param-iam-delete[param-fo])
  apply (rule iam-delete-correct[param-fo])
  by simp-all

lemmas iam-map-impl =
  iam-empty-impl
  iam-lookup-impl
  iam-update-impl
  iam-delete-impl

declare iam-map-impl[autoref-rules]
```

3.11.4 Iterator proofs

```
abbreviation iam-to-list a ≡ it-to-list iam-iteratei a

lemma distinct-iam-to-list-aux:
  shows ⟦distinct xs; ∀(i,-)∈set xs. i ≥ n⟧ ⟹
        distinct (iam-iteratei-aux n a
```

```
            (λ-. True) (λx y. y @ [x]) xs)
  (is ⟦-;-⟧ ⟹ distinct (?iam-to-list-aux n xs))
proof (induction n arbitrary: xs)
  case (0 xs) thus ?case by simp
next
  case (Suc i xs)
    show ?case
    proof (cases array-get a i)
      case None
        with Suc.IH[OF Suc.prems(1)] Suc.prems(2)
            show ?thesis by force
    next
      case (Some x)
        let ?xs' = xs @ [(i,x)]
        from Suc.prems have distinct ?xs' and
            ∀ (i',x)∈set ?xs'. i' ≥ i by force+
        from Some and Suc.IH[OF this] show ?thesis by simp
  qed
qed

lemma distinct-iam-to-list:
  distinct (iam-to-list a)
unfolding it-to-list-def iam-iteratei-def
  by (force intro: distinct-iam-to-list-aux)

lemma iam-to-list-set-correct-aux:
  assumes (a, m) ∈ iam-rel'
  shows ⟦n ≤ array-length a; map-to-set m − {(k,v). k < n} = set xs⟧
        ⟹ map-to-set m =
            set (iam-iteratei-aux n a (λ-. True) (λx y. y @ [x]) xs)
proof (induction n arbitrary: xs)
  case (0 xs)
    thus ?case by simp
next
  case (Suc n xs)
    with assms have [simp]: array-get a n = m n
        unfolding iam-rel'-def br-def iam-α-def[abs-def] by simp
    show ?case
    proof (cases m n)
      case None
        with Suc.prems(2) have map-to-set m − {(k,v). k < n} = set xs
        unfolding map-to-set-def by (fastforce simp: less-Suc-eq)
        from None and Suc.IH[OF - this] and Suc.prems(1)
            show ?thesis by simp
    next
      case (Some x)
        let ?xs' = xs @ [(n,x)]
        from Some and Suc.prems(2)
            have map-to-set m − {(k,v). k < n} = set ?xs'
```

```
        unfolding map-to-set-def by (fastforce simp: less-Suc-eq)
      from Some and Suc.IH[OF - this] and Suc.prems(1)
        show ?thesis by simp
  qed
qed

lemma iam-to-list-set-correct:
 assumes (a, m) ∈ iam-rel′
 shows map-to-set m = set (iam-to-list a)
proof−
 from assms
   have A: map-to-set m − {(k, v). k < array-length a} = set []
   unfolding map-to-set-def iam-rel′-def br-def iam-α-def[abs-def]
   by (force split: split-if-asm)
 with iam-to-list-set-correct-aux[OF assms - A] show ?thesis
  unfolding it-to-list-def iam-iteratei-def by simp
qed

lemma iam-iteratei-aux-append:
 n ≤ length xs =⇒ iam-iteratei-aux n (Array (xs @ ys)) =
       iam-iteratei-aux n (Array xs)
apply (induction n)
apply force
apply (intro ext, auto split: option.split simp: nth-append)
done

lemma iam-iteratei-append:
 iam-iteratei (Array (xs @ [None])) c f σ =
    iam-iteratei (Array xs) c f σ
 iam-iteratei (Array (xs @ [Some x])) c f σ =
    iam-iteratei (Array xs) c f
    (if c σ then (f (length xs, x) σ) else σ)
unfolding iam-iteratei-def
apply (cases length xs)
apply (simp add: iam-iteratei-aux-append)
apply (force simp: nth-append iam-iteratei-aux-append) []
apply (cases length xs)
apply (simp add: iam-iteratei-aux-append)
apply (force split: option.split
       simp: nth-append iam-iteratei-aux-append) []
done


lemma iam-iteratei-aux-Cons:
 n < array-length a =⇒
   iam-iteratei-aux n a (λ-. True) (λx l. l @ [x]) (x#xs) =
   x # iam-iteratei-aux n a (λ-. True) (λx l. l @ [x]) xs
  by (induction n arbitrary: xs, auto split: option.split)
```

```
lemma iam-to-list-append:
  iam-to-list (Array (xs @ [None])) = iam-to-list (Array xs)
  iam-to-list (Array (xs @ [Some x])) =
      (length xs, x) # iam-to-list (Array xs)
unfolding it-to-list-def iam-iteratei-def
apply (simp add: iam-iteratei-aux-append)
apply (simp add: iam-iteratei-aux-Cons)
apply (simp add: iam-iteratei-aux-append)
done

lemma autoref-iam-is-iterator[autoref-ga-rules]:
  shows is-map-to-list nat-rel Rv iam-map-rel iam-to-list
  unfolding is-map-to-list-def is-map-to-sorted-list-def
proof (clarify)
  fix a m'
  assume (a,m') ∈ ⟨nat-rel,Rv⟩iam-map-rel
  then obtain a' where [param]: (a,a')∈⟨Rv⟩iam-rel
    and (a',m')∈iam-rel' unfolding iam-map-rel-def by blast

  have (iam-to-list a, iam-to-list a')
          ∈ ⟨⟨nat-rel, Rv⟩prod-rel⟩list-rel by parametricity

  moreover from distinct-iam-to-list and
              iam-to-list-set-correct[OF ‹(a',m')∈iam-rel'›]
     have RETURN (iam-to-list a') ≤ it-to-sorted-list
            (key-rel (λ- -. True)) (map-to-set m')
     unfolding it-to-sorted-list-def key-rel-def[abs-def]
         by (force intro: refine-vcg)

  ultimately show ∃l'. (iam-to-list a, l') ∈
                        ⟨⟨nat-rel, Rv⟩prod-rel⟩list-rel
                   ∧ RETURN l' ≤ it-to-sorted-list (
                       key-rel (λ- -. True)) (map-to-set m') by blast
qed


lemmas [autoref-ga-rules] =
  autoref-iam-is-iterator[unfolded is-map-to-list-def]

lemma iam-iteratei-altdef:
    iam-iteratei a = foldli (iam-to-list a)
     (is ?f a = ?g (iam-to-list a))
proof−
  obtain l where a = Array l by (cases a)
  have ?f (Array l) = ?g (iam-to-list (Array l))
  proof (induction length l arbitrary: l)
    case (0 l)
      thus ?case by (intro ext,
          simp add: iam-iteratei-def it-to-list-def)
```

```
  next
    case (Suc n l)
      hence l ≠ [] and [simp]: length l = Suc n by force+
      with append-butlast-last-id have [simp]:
          butlast l @ [last l] = l by simp
      with Suc have [simp]: length (butlast l) = n by simp
      note IH = Suc(1)[OF this[symmetric]]
      let ?l' = iam-to-list (Array l)

      show ?case
      proof (cases last l)
        case None
          have ?f (Array l) =
              ?f (Array (butlast l @ [last l])) by simp
          also have ... = ?g (iam-to-list (Array (butlast l)))
              by (force simp: None iam-iteratei-append IH)
          also have iam-to-list (Array (butlast l)) =
              iam-to-list (Array (butlast l @ [last l]))
              using None by (simp add: iam-to-list-append)
          finally show ?f (Array l) = ?g ?l' by simp
      next
        case (Some x)
          have ?f (Array l) =
              ?f (Array (butlast l @ [last l])) by simp
          also have ... = ?g (iam-to-list
              (Array (butlast l @ [last l])))
              by (force simp: IH iam-iteratei-append
                      iam-to-list-append Some)
          finally show ?thesis by simp
      qed
  qed
  thus ?thesis by (simp add: ‹a = Array l›)
qed


lemma pi-iam[icf-proper-iteratorI]:
  proper-it (iam-iteratei a) (iam-iteratei a)
unfolding proper-it-def by (force simp: iam-iteratei-altdef)

lemma pi'-iam[icf-proper-iteratorI]:
  proper-it' iam-iteratei iam-iteratei
  by (rule proper-it'I, rule pi-iam)

end
```

3.12 The hashable Typeclass

theory *HashCode*

imports *Main*
begin

In this theory a typeclass of hashable types is established. For hashable types, there is a function *hashcode*, that maps any entity of this type to an integer value.

This theory defines the hashable typeclass and provides instantiations for a couple of standard types.

type-synonym
  hashcode = *nat*

class *hashable* =
  fixes *hashcode* :: *'a* ⇒ *hashcode*
  and *bounded-hashcode* :: *nat* ⇒ *'a* ⇒ *hashcode*
  and *def-hashmap-size* :: *'a itself* ⇒ *nat*
  assumes *bounded-hashcode-bounds*: *1* < *n* ⟹ *bounded-hashcode n a* < *n*
  and *def-hashmap-size*: *1* < *def-hashmap-size TYPE*(*'a*)

instantiation *unit* :: *hashable*
begin
  definition [*simp*]: *hashcode* (*u* :: *unit*) = *0*
  definition [*simp*]: *bounded-hashcode n* (*u* :: *unit*) = *0*
  definition *def-hashmap-size* = (λ- :: *unit itself*. *2*)
  instance by(*intro-classes*)(*simp-all add*: *def-hashmap-size-unit-def*)
end

instantiation *bool* :: *hashable*
begin
  definition [*simp*]: *hashcode* (*b* :: *bool*) = (*if b then 1 else 0*)
  definition [*simp*]: *bounded-hashcode n* (*b* :: *bool*) = (*if b then 1 else 0*)
  definition *def-hashmap-size* = (λ- :: *bool itself*. *2*)
  instance by(*intro-classes*)(*simp-all add*: *def-hashmap-size-bool-def*)
end

instantiation *int* :: *hashable*
begin
  definition [*simp*]: *hashcode* (*i* :: *int*) = *nat* (*abs i*)
  definition [*simp*]: *bounded-hashcode n* (*i* :: *int*) = *nat* (*abs i*) *mod n*
  definition *def-hashmap-size* = (λ- :: *int itself*. *16*)
  instance by(*intro-classes*)(*simp-all add*: *def-hashmap-size-int-def*)
end

instantiation *nat* :: *hashable*
begin
  definition [*simp*]: *hashcode* (*n* :: *nat*) = *n*
  definition [*simp*]: *bounded-hashcode n'* (*n* :: *nat*) == *n mod n'*
  definition *def-hashmap-size* = (λ- :: *nat itself*. *16*)
  instance by(*intro-classes*)(*simp-all add*: *def-hashmap-size-nat-def*)
end

```
fun num-of-nibble :: nibble ⇒ nat
  where
  num-of-nibble Nibble0 = 0 |
  num-of-nibble Nibble1 = 1 |
  num-of-nibble Nibble2 = 2 |
  num-of-nibble Nibble3 = 3 |
  num-of-nibble Nibble4 = 4 |
  num-of-nibble Nibble5 = 5 |
  num-of-nibble Nibble6 = 6 |
  num-of-nibble Nibble7 = 7 |
  num-of-nibble Nibble8 = 8 |
  num-of-nibble Nibble9 = 9 |
  num-of-nibble NibbleA = 10 |
  num-of-nibble NibbleB = 11 |
  num-of-nibble NibbleC = 12 |
  num-of-nibble NibbleD = 13 |
  num-of-nibble NibbleE = 14 |
  num-of-nibble NibbleF = 15

instantiation nibble :: hashable
begin
  definition [simp]: hashcode (c :: nibble) = num-of-nibble c
  definition [simp]: bounded-hashcode n c == num-of-nibble c mod n
  definition def-hashmap-size = (λ- :: nibble itself. 16)
  instance by(intro-classes)(simp-all add: def-hashmap-size-nibble-def)
end

instantiation char :: hashable
begin
  fun hashcode-of-char :: char ⇒ hashcode where
    hashcode-of-char (Char a b) = num-of-nibble a * 16 + num-of-nibble b

  definition [simp]: hashcode c == hashcode-of-char c
  definition [simp]: bounded-hashcode n c == hashcode-of-char c mod n
  definition def-hashmap-size = (λ- :: char itself. 32)
  instance by(intro-classes)(simp-all add: def-hashmap-size-char-def)
end

instantiation prod :: (hashable, hashable) hashable
begin
  definition hashcode x == (hashcode (fst x) * 33 + hashcode (snd x))
 definition bounded-hashcode n x == (bounded-hashcode n (fst x) * 33 + bounded-hashcode
n (snd x)) mod n
 definition def-hashmap-size = (λ- :: ('a × 'b) itself. def-hashmap-size TYPE('a)
+ def-hashmap-size TYPE('b))
 instance using def-hashmap-size[where ?'a='a] def-hashmap-size[where ?'a='b]
  by(intro-classes)(simp-all add: bounded-hashcode-prod-def def-hashmap-size-prod-def)
end
```

```
instantiation sum :: (hashable, hashable) hashable
begin
  definition hashcode x == (case x of Inl a ⇒ 2 ∗ hashcode a | Inr b ⇒ 2 ∗
hashcode b + 1)
  definition bounded-hashcode n x == (case x of Inl a ⇒ bounded-hashcode n a |
Inr b ⇒ (n − 1 − bounded-hashcode n b))
 definition def-hashmap-size = (λ- :: ('a + 'b) itself. def-hashmap-size TYPE('a)
+ def-hashmap-size TYPE('b))
 instance using def-hashmap-size[where ?'a='a] def-hashmap-size[where ?'a='b]
  by(intro-classes)(simp-all add: bounded-hashcode-sum-def bounded-hashcode-bounds
def-hashmap-size-sum-def split: sum.split)
end

instantiation list :: (hashable) hashable
begin
  definition hashcode = foldl (λh x. h ∗ 33 + hashcode x) 5381
  definition bounded-hashcode n = foldl (λh x. (h ∗ 33 + bounded-hashcode n x)
mod n) (5381 mod n)
 definition def-hashmap-size = (λ- :: 'a list itself. 2 ∗ def-hashmap-size TYPE('a))
  instance
  proof
    fix n :: nat and xs :: 'a list
    assume 1 < n
    thus bounded-hashcode n xs < n unfolding bounded-hashcode-list-def
      by(cases xs rule: rev-cases) simp-all
  next
    from def-hashmap-size[where ?'a = 'a]
    show 1 < def-hashmap-size TYPE('a list)
      by(simp add: def-hashmap-size-list-def)
  qed
end

instantiation option :: (hashable) hashable
begin
  definition hashcode opt = (case opt of None ⇒ 0 | Some a ⇒ hashcode a + 1)
   definition bounded-hashcode n opt = (case opt of None ⇒ 0 | Some a ⇒
(bounded-hashcode n a + 1) mod n)
 definition def-hashmap-size = (λ- :: 'a option itself. def-hashmap-size TYPE('a)
+ 1)
  instance using def-hashmap-size[where ?'a = 'a]
  by(intro-classes)(simp-all add: bounded-hashcode-option-def def-hashmap-size-option-def
split: option.split)
end

lemma hashcode-option-simps [simp]:
  hashcode None = 0
  hashcode (Some a) = 1 + hashcode a
  by(simp-all add: hashcode-option-def)
```

lemma *bounded-hashcode-option-simps* [*simp*]:
  bounded-hashcode n None = 0
  bounded-hashcode n (Some a) = (bounded-hashcode n a + 1) mod n
  by(*simp-all add*: *bounded-hashcode-option-def*)

end

3.13 Hashable Interface

theory *Intf-Hash*
imports
    Main
    ../Lib/HashCode
  ../../Parametricity/Param-HOL
  ../../Autoref/Autoref-Bindings-HOL
begin

type-synonym *'a eq = 'a ⇒ 'a ⇒ bool*
type-synonym *'k bhc = nat ⇒ 'k ⇒ nat*

3.13.1 Abstract and concrete hash functions

definition *is-hashcode* :: (*'k ⇒ nat*) ⇒ *bool*
  where *is-hashcode - = True*

lemma *hashable-hc-is-hc*[*intro*]:
    is-hashcode hashcode
    unfolding *is-hashcode-def* **..**

definition *is-bounded-hashcode* :: *'c eq ⇒ 'c bhc ⇒ bool*
  where *is-bounded-hashcode eq bhc ≡*
      (∀ *n x y*. *eq x y ⟶ bhc n x = bhc n y*) ∧
      (∀ *n x*. *1 < n ⟶ bhc n x < n*)
definition *abstract-bounded-hashcode* :: (*'c×'a*) *set ⇒ 'c bhc ⇒ 'a bhc*
  where *abstract-bounded-hashcode Rk bhc n x' ≡*
      if x' ∈ Range Rk
        then THE c. ∃ *x*. (*x,x'*) ∈ *Rk* ∧ *bhc n x = c*
        else 0

lemma *is-bounded-hashcodeI*[*intro*]:
  (⋀*x y n*. *eq x y ⟹ bhc n x = bhc n y*) ⟹
   (⋀*x n*. *1 < n ⟹ bhc n x < n*) ⟹ *is-bounded-hashcode eq bhc*
  unfolding *is-bounded-hashcode-def* **by** *force*

lemma *is-bounded-hashcodeD*[*dest*]:
  assumes *is-bounded-hashcode eq bhc*

```
  shows ⋀n x y. eq x y ⟹ bhc n x = bhc n y and
      ⋀n x. 1 < n ⟹ bhc n x < n
  using assms unfolding is-bounded-hashcode-def by simp-all

lemma bounded-hashcode-welldefined:
  assumes (eq, op=) ∈ Rk → Rk → bool-rel and
          is-bounded-hashcode eq bhc and
          (x1,x') ∈ Rk and (x2,x') ∈ Rk
  shows bhc n x1 = bhc n x2
proof−
  from assms(1,3,4) have eq x1 x2 by (force dest: fun-relD)
  thus ?thesis using assms(2) by blast
qed

lemma abstract-bhc-correct[intro]:
  assumes (eq, op=) ∈ Rk → Rk → bool-rel
  assumes is-bounded-hashcode eq bhc
  shows (bhc, abstract-bounded-hashcode Rk bhc) ∈
      nat-rel → Rk → nat-rel (is (bhc, ?bhc') ∈ -)
proof (intro fun-relI)
  fix n n' x x'
  assume A: (n,n') ∈ nat-rel and B: (x,x') ∈ Rk
  hence C: n = n' and D: x' ∈ Range Rk by auto
  have ?bhc' n' x' = bhc n x
      unfolding abstract-bounded-hashcode-def
      apply (simp add: C D, rule)
      apply (intro exI conjI, fact B, rule refl)
      apply (elim exE conjE, hypsubst,
          erule bounded-hashcode-welldefined[OF assms - B])
      done
  thus (bhc n x, ?bhc' n' x') ∈ nat-rel by simp
qed

lemma abstract-bhc-is-bhc[intro]:
  fixes Rk :: ('c×'a) set
  assumes eq: (eq, op=) ∈ Rk → Rk → bool-rel
  assumes bhc: is-bounded-hashcode eq bhc
  shows is-bounded-hashcode op= (abstract-bounded-hashcode Rk bhc)
      (is is-bounded-hashcode op= ?bhc')
proof
  fix x'::'a and y'::'a and n'::nat assume x' = y'
  thus ?bhc' n' x' = ?bhc' n' y' by simp
next
  fix x'::'a and n'::nat assume 1 < n'
  from abstract-bhc-correct[OF eq bhc] show ?bhc' n' x' < n'
  proof (cases x' ∈ Range Rk)
    case False
      with ‹1 < n'› show ?thesis
          unfolding abstract-bounded-hashcode-def by simp
```

```
  next
    case True
      then obtain x where (x,x') ∈ Rk ..
      have (n',n') ∈ nat-rel ..
      from abstract-bhc-correct[OF assms] have ?bhc' n' x' = bhc n' x
        apply −
        apply (drule fun-relD[OF - ‹(n',n') ∈ nat-rel›],
            drule fun-relD[OF - ‹(x,x') ∈ Rk›], simp)
        done
      also from ‹1 < n'› and bhc have ... < n' by blast
      finally show ?bhc' n' x' < n' .
  qed
qed

lemma hashable-bhc-is-bhc[autoref-ga-rules]:
  STRUCT-EQ-tag eq op= ⟹ is-bounded-hashcode eq bounded-hashcode
  unfolding is-bounded-hashcode-def
  by (simp add: bounded-hashcode-bounds)
```

3.13.2 Default hash map size

```
definition is-valid-def-hm-size :: 'k itself ⇒ nat ⇒ bool
    where is-valid-def-hm-size type n ≡ n > 1

lemma hashable-def-size-is-def-size[autoref-ga-rules]:
  shows is-valid-def-hm-size TYPE('k::hashable) (def-hashmap-size TYPE('k))
    unfolding is-valid-def-hm-size-def by (fact def-hashmap-size)

end

theory idx-iteratei
imports
  Diff-Array
  ../Gen/Gen-Iterator
  ../Intf/Intf-Comp
begin
```

iteratei (by get, size)

```
fun idx-iteratei-aux
  :: ('s ⇒ nat ⇒ 'a) ⇒ nat ⇒ nat ⇒ 's ⇒ ('σ⇒bool) ⇒ ('a ⇒'σ ⇒ 'σ) ⇒ 'σ ⇒
'σ
where
  idx-iteratei-aux get sz i l c f σ = (
    if i=0 ∨ ¬ c σ then σ
    else idx-iteratei-aux get sz (i − 1) l c f (f (get l (sz−i)) σ)
  )

declare idx-iteratei-aux.simps[simp del]
```

```
lemma idx-iteratei-aux-simps[simp]:
  i=0 ⟹ idx-iteratei-aux get sz i l c f σ = σ
  ¬c σ ⟹ idx-iteratei-aux get sz i l c f σ = σ
  ⟦i≠0; c σ⟧ ⟹ idx-iteratei-aux get sz i l c f σ = idx-iteratei-aux get sz (i − 1) l
c f (f (get l (sz−i)) σ)
  apply −
  apply (subst idx-iteratei-aux.simps, simp)+
  done

definition idx-iteratei get sz l c f σ == idx-iteratei-aux get (sz l) (sz l) l c f σ

lemma idx-iteratei-eq-foldli[autoref-rules]:
  assumes sz: (sz, length) ∈ arel → nat-rel
  assumes get: (get, op!) ∈ arel → nat-rel → Id
  assumes (s,s') ∈ arel
  shows (idx-iteratei get sz s, foldli s') ∈ Id
proof−
  have size-correct: ⋀s s'. (s,s') ∈ arel ⟹ sz s = length s'
    using sz[param-fo] by simp
  have get-correct: ⋀s s' n. (s,s') ∈ arel ⟹ get s n = s' ! n
    using get[param-fo] by simp
  {
    fix n l
    assume A: Suc n ≤ length l
    hence B: length l − Suc n < length l by simp
    from A have [simp]: Suc (length l − Suc n) = length l − n by simp
    from nth-drop'[OF B, simplified] have
      drop (length l − Suc n) l = l!(length l − Suc n)#drop (length l − n) l
      by simp
  } note drop-aux=this

  {
    fix s s' c f σ i
    assume (s,s') ∈ arel   i≤sz s
    hence idx-iteratei-aux get (sz s) i s c f σ = foldli (drop (sz s − i) s') c f σ
    proof (induct i arbitrary: σ)
      case 0 with size-correct[of s] show ?case by simp
    next
      case (Suc n)
      note S = Suc.prems(1)
      show ?case proof (cases c σ)
        case False thus ?thesis by simp
      next
        case True[simp, intro!]
        show ?thesis using Suc
          by (simp add: size-correct[OF S] get-correct[OF S] drop-aux)
      qed
    qed
  } note aux=this
```

```
  show ?thesis
    unfolding idx-iteratei-def[abs-def]
    by (simp, intro ext, simp add: aux[OF ‹(s,s′) ∈ arel›])
qed
```

Misc.

```
lemma idx-iteratei-aux-array-get-Array-conv-nth:
  idx-iteratei-aux array-get sz i (Array xs) c f σ =
   idx-iteratei-aux op ! sz i xs c f σ
apply(induct get≡op ! :: ′b list ⇒ nat ⇒ ′b sz i xs c f σ rule: idx-iteratei-aux.induct)
apply(subst (1 2) idx-iteratei-aux.simps)
apply simp
done

lemma idx-iteratei-array-get-Array-conv-nth:
  idx-iteratei array-get array-length (Array xs) = idx-iteratei nth length xs
by(simp add: idx-iteratei-def fun-eq-iff idx-iteratei-aux-array-get-Array-conv-nth)

lemma idx-iteratei-aux-nth-conv-foldli-drop:
  fixes xs :: ′b list
  assumes i ≤ length xs
  shows idx-iteratei-aux op ! (length xs) i xs c f σ = foldli (drop (length xs − i)
xs) c f σ
using assms
proof(induct get≡op ! :: ′b list ⇒ nat ⇒ ′b sz≡   length xs i xs c f σ rule:
idx-iteratei-aux.induct)
  case (1 i l c f σ)
  show ?case
  proof(cases i = 0 ∨ ¬ c σ)
    case True thus ?thesis
      by(subst idx-iteratei-aux.simps)(auto)
  next
    case False
    hence i: i > 0 and c: c σ by auto
    hence idx-iteratei-aux op ! (length l) i l c f σ = idx-iteratei-aux op ! (length l)
(i − 1) l c f (f (l ! (length l − i)) σ)
      by(subst idx-iteratei-aux.simps) simp
    also have ... = foldli (drop (length l − (i − 1)) l) c f (f (l ! (length l − i))
σ)
      using ‹i ≤ length l› i c by −(rule 1, auto)
    also from ‹i ≤ length l› i
    have drop (length l − i) l = (l ! (length l − i)) # drop (length l − (i − 1)) l
     by(subst nth-drop′[symmetric])(simp-all, metis Suc-eq-plus1-left add-diff-assoc)
    hence foldli (drop (length l − (i − 1)) l) c f (f (l ! (length l − i)) σ) = foldli
(drop (length l − i) l) c f σ
      using c by simp
    finally show ?thesis .
  qed
```

qed

lemma *idx-iteratei-nth-length-conv-foldli*: *idx-iteratei nth length* = *foldli*
by(*rule ext*)+(*simp add*: *idx-iteratei-def idx-iteratei-aux-nth-conv-foldli-drop*)

end

3.14 Array Based Hash-Maps

theory *Impl-Array-Hash-Map* **imports**
../../Autoref/Autoref
../Lib/Proper-Iterator
../Gen/Gen-Iterator
../Gen/Gen-Map
../Intf/Intf-Hash
../Intf/Intf-Map
../Lib/HashCode
../Lib/Diff-Array
../Lib/idx-iteratei
Impl-List-Map
begin

3.14.1 Type definition and primitive operations

definition *load-factor* :: *nat* — in percent
where *load-factor* = *75*

datatype ('*key*, '*val*) *hashmap* =
HashMap ('*key*,'*val*) *list-map array* *nat*

3.14.2 Operations

definition *new-hashmap-with* :: *nat* ⇒ ('*k*, '*v*) *hashmap*
where ⋀*size*. *new-hashmap-with size* =
HashMap (*new-array* [] *size*) *0*

definition *ahm-empty* :: *nat* ⇒ ('*k*, '*v*) *hashmap*
where *ahm-empty def-size* ≡ *new-hashmap-with def-size*

definition *bucket-ok* :: '*k bhc* ⇒ *nat* ⇒ *nat* ⇒ ('*k*×'*v*) *list* ⇒ *bool*
where *bucket-ok bhc len h kvs* = (∀ *k* ∈ *fst* ‘ *set kvs*. *bhc len k* = *h*)

definition *ahm-invar-aux* :: '*k bhc* ⇒ *nat* ⇒ ('*k*×'*v*) *list array* ⇒ *bool*
where
ahm-invar-aux bhc n a ⟷
(∀ *h*. *h* < *array-length a* ⟶ *bucket-ok bhc* (*array-length a*) *h*
(*array-get a h*) ∧ *list-map-invar* (*array-get a h*)) ∧
array-foldl (λ- *n kvs*. *n* + *size kvs*) *0 a* = *n* ∧
array-length a > *1*

primrec *ahm-invar* :: $'k$ *bhc* $\Rightarrow$ $('k,\ 'v)$ *hashmap* $\Rightarrow$ *bool*
where *ahm-invar bhc* (*HashMap a n*) = *ahm-invar-aux bhc n a*

definition *ahm-lookup-aux* :: $'k$ *eq* $\Rightarrow$ $'k$ *bhc* $\Rightarrow$
$'k \Rightarrow ('k,\ 'v)$ *list-map array* $\Rightarrow$ $'v$ *option*
where [*simp*]: *ahm-lookup-aux eq bhc k a* = *list-map-lookup eq k* (*array-get a* (*bhc* (*array-length a*) *k*))

primrec *ahm-lookup* **where**
ahm-lookup eq bhc k (*HashMap a* -) = *ahm-lookup-aux eq bhc k a*

definition *ahm-*α *bhc m* $\equiv$ λk. *ahm-lookup op= bhc k m*

definition *ahm-map-rel-def-internal*:
ahm-map-rel Rk Rv $\equiv$ {(*HashMap a n*, *HashMap a' n*)| *a a' n n'*.
(*a*,*a'*) $\in$ $\langle\langle\langle$*Rk*,*Rv*$\rangle$*prod-rel*$\rangle$*list-rel*$\rangle$*array-rel* $\wedge$ (*n*,*n'*) $\in$ *Id*}

lemma *ahm-map-rel-def*: $\langle$*Rk*,*Rv*$\rangle$ *ahm-map-rel* $\equiv$
{(*HashMap a n*, *HashMap a' n*)| *a a' n n'*.
(*a*,*a'*) $\in$ $\langle\langle\langle$*Rk*,*Rv*$\rangle$*prod-rel*$\rangle$*list-rel*$\rangle$*array-rel* $\wedge$ (*n*,*n'*) $\in$ *Id*}
unfolding *relAPP-def ahm-map-rel-def-internal* **.**

definition *ahm-map-rel'-def*:
ahm-map-rel' bhc $\equiv$ *br* (*ahm-*α *bhc*) (*ahm-invar bhc*)

definition *ahm-rel-def-internal*: *ahm-rel bhc Rk Rv* =
$\langle$*Rk*,*Rv*$\rangle$ *ahm-map-rel O ahm-map-rel'* (*abstract-bounded-hashcode Rk bhc*)
lemma *ahm-rel-def*: $\langle$*Rk*, *Rv*$\rangle$ *ahm-rel bhc* $\equiv$
$\langle$*Rk*,*Rv*$\rangle$ *ahm-map-rel O ahm-map-rel'* (*abstract-bounded-hashcode Rk bhc*)
unfolding *relAPP-def ahm-rel-def-internal* **.**
lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of ahm-rel bhc i-map*, *standard*]

abbreviation *dflt-ahm-rel* $\equiv$ *ahm-rel bounded-hashcode*

primrec *ahm-iteratei-aux* :: (($'k \times 'v$) *list array*) $\Rightarrow$ ($'k \times 'v$, $'\sigma$) *set-iterator*
where *ahm-iteratei-aux* (*Array xs*) *c f* = *foldli* (*concat xs*) *c f*

primrec *ahm-iteratei* :: (($'k$, $'v$) *hashmap*) $\Rightarrow$ (($'k \times 'v$), $'\sigma$) *set-iterator*
where
ahm-iteratei (*HashMap a n*) = *ahm-iteratei-aux a*

definition *ahm-rehash-aux'* :: $'k$ *bhc* $\Rightarrow$ *nat* $\Rightarrow$ $'k \times 'v$ $\Rightarrow$
($'k \times 'v$) *list array* $\Rightarrow$ ($'k \times 'v$) *list array*
where
ahm-rehash-aux' bhc n kv a =
(*let h* = *bhc n* (*fst kv*)
in array-set a h (*kv* # *array-get a h*))

```
definition ahm-rehash-aux :: 'k bhc ⇒ ('k×'v) list array ⇒ nat ⇒
  ('k×'v) list array
where
  ahm-rehash-aux bhc a sz = ahm-iteratei-aux a (λx. True)
      (ahm-rehash-aux' bhc sz) (new-array [] sz)

primrec ahm-rehash :: 'k bhc ⇒ ('k,'v) hashmap ⇒ nat ⇒ ('k,'v) hashmap
where ahm-rehash bhc (HashMap a n) sz = HashMap (ahm-rehash-aux bhc a sz)
n

primrec hm-grow :: ('k,'v) hashmap ⇒ nat
where hm-grow (HashMap a n) = 2 * array-length a + 3

primrec ahm-filled :: ('k,'v) hashmap ⇒ bool
where ahm-filled (HashMap a n) = (array-length a * load-factor ≤ n * 100)

primrec ahm-update-aux :: 'k eq ⇒ 'k bhc ⇒ ('k,'v) hashmap ⇒
  'k ⇒ 'v ⇒ ('k, 'v) hashmap
where
  ahm-update-aux eq bhc (HashMap a n) k v =
  (let h = bhc (array-length a) k;
      m = array-get a h;
      insert = list-map-lookup eq k m = None
   in HashMap (array-set a h (list-map-update eq k v m))
      (if insert then n + 1 else n))

definition ahm-update :: 'k eq ⇒ 'k bhc ⇒ 'k ⇒ 'v ⇒
  ('k, 'v) hashmap ⇒ ('k, 'v) hashmap
where
  ahm-update eq bhc k v hm =
   (let hm' = ahm-update-aux eq bhc hm k v
    in (if ahm-filled hm' then ahm-rehash bhc hm' (hm-grow hm') else hm'))

primrec ahm-delete :: 'k eq ⇒ 'k bhc ⇒ 'k ⇒
  ('k,'v) hashmap ⇒ ('k,'v) hashmap
where
  ahm-delete eq bhc k (HashMap a n) =
  (let h = bhc (array-length a) k;
      m = array-get a h;
      deleted = (list-map-lookup eq k m ≠ None)
   in HashMap (array-set a h (list-map-delete eq k m)) (if deleted then n − 1 else
n))

primrec ahm-isEmpty :: ('k,'v) hashmap ⇒ bool where
  ahm-isEmpty (HashMap - n) = (n = 0)

primrec ahm-isSng :: ('k,'v) hashmap ⇒ bool where
  ahm-isSng (HashMap - n) = (n = 1)
```

primrec *ahm-size* :: ('*k*,'*v*) *hashmap* ⇒ *nat* **where**
ahm-size (*HashMap* - *n*) = *n*

lemma *hm-grow-gt-1* [*iff*]:
Suc 0 < *hm-grow hm*
by(*cases hm*)(*simp*)

lemma *bucket-ok-Nil* [*simp*]: *bucket-ok bhc len h* [] = *True*
by(*simp add*: *bucket-ok-def*)

lemma *bucket-okD*:
⟦ *bucket-ok bhc len h xs*; (*k*, *v*) ∈ *set xs* ⟧
⟹ *bhc len k* = *h*
by(*auto simp add*: *bucket-ok-def*)

lemma *bucket-okI*:
(⋀*k*. *k* ∈ *fst* ‘ *set kvs* ⟹ *bhc len k* = *h*) ⟹ *bucket-ok bhc len h kvs*
by(*simp add*: *bucket-ok-def*)

3.14.3 Parametricity

lemma *param-HashMap*[*param*]: (*HashMap*, *HashMap*) ∈
⟨⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*⟩*array-rel* → *nat-rel* → ⟨*Rk*,*Rv*⟩*ahm-map-rel*
unfolding *ahm-map-rel-def* **by** *force*

lemma *param-hashmap-case*[*param*]: (*hashmap-case*, *hashmap-case*) ∈
(⟨⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*⟩*array-rel* → *nat-rel* → *R*) →
⟨*Rk*,*Rv*⟩*ahm-map-rel* → *R*
unfolding *ahm-map-rel-def*[*abs-def*]
by (*force split*: *hashmap.split dest*: *fun-relD*)

lemma *hashmap-rec-is-case*[*simp*]: *hashmap-rec* = *hashmap-case*
by (*intro ext*, *simp split*: *hashmap.split*)

3.14.4 *ahm-invar*

lemma *ahm-invar-auxD*:
assumes *ahm-invar-aux bhc n a*
shows ⋀*h*. *h* < *array-length a* ⟹
bucket-ok bhc (*array-length a*) *h* (*array-get a h*) **and**
⋀*h*. *h* < *array-length a* ⟹
list-map-invar (*array-get a h*) **and**
n = *array-foldl* (λ- *n kvs*. *n* + *length kvs*) *0 a* **and**
array-length a > *1*
using *assms* **unfolding** *ahm-invar-aux-def* **by** *auto*

lemma *ahm-invar-auxE*:
assumes *ahm-invar-aux bhc n a*

```
  obtains ∀h. h < array-length a ⟶
      bucket-ok bhc (array-length a) h (array-get a h) ∧
      list-map-invar (array-get a h) and
  n = array-foldl (λ- n kvs. n + length kvs) 0 a and
  array-length a > 1
using assms unfolding ahm-invar-aux-def by blast

lemma ahm-invar-auxI:
  ⟦ ⋀h. h < array-length a ⟹
        bucket-ok bhc (array-length a) h (array-get a h);
    ⋀h. h < array-length a ⟹ list-map-invar (array-get a h);
    n = array-foldl (λ- n kvs. n + length kvs) 0 a; array-length a > 1 ⟧
  ⟹ ahm-invar-aux bhc n a
unfolding ahm-invar-aux-def by blast

lemma ahm-invar-distinct-fst-concatD:
  assumes inv: ahm-invar-aux bhc n (Array xs)
  shows distinct (map fst (concat xs))
proof −
  { fix h
    assume h < length xs
    with inv have bucket-ok bhc (length xs) h (xs ! h) and
               list-map-invar (xs ! h)
      by(simp-all add: ahm-invar-aux-def) }
  note no-junk = this

  show ?thesis unfolding map-concat
  proof(rule distinct-concat')
    have distinct [x←xs . x ≠ []] unfolding distinct-conv-nth
    proof(intro allI ballI impI)
      fix i j
      assume i < length [x←xs . x ≠ []]   j < length [x←xs . x ≠ []]   i ≠ j
      from filter-nth-ex-nth[OF ‹i < length [x←xs . x ≠ []]›]
     obtain i' where i' ≥ i   i' < length xs and ith: [x←xs . x ≠ []] ! i = xs ! i'
        and eqi: [x←take i' xs . x ≠ []] = take i [x←xs . x ≠ []] by blast
      from filter-nth-ex-nth[OF ‹j < length [x←xs . x ≠ []]›]
     obtain j' where j' ≥ j   j' < length xs and jth: [x←xs . x ≠ []] ! j = xs ! j'
        and eqj: [x←take j' xs . x ≠ []] = take j [x←xs . x ≠ []] by blast
      show [x←xs . x ≠ []] ! i ≠ [x←xs . x ≠ []] ! j
      proof
        assume [x←xs . x ≠ []] ! i = [x←xs . x ≠ []] ! j
        hence eq: xs ! i' = xs ! j' using ith jth by simp
        from ‹i < length [x←xs . x ≠ []]›
        have [x←xs . x ≠ []] ! i ∈ set [x←xs . x ≠ []] by(rule nth-mem)
        with ith have xs ! i' ≠ [] by simp
      then obtain kv where kv ∈ set (xs ! i') by(fastforce simp add: neq-Nil-conv)
        with no-junk[OF ‹i' < length xs›] have bhc (length xs) (fst kv) = i'
          by(simp add: bucket-ok-def)
        moreover from eq ‹kv ∈ set (xs ! i')› have kv ∈ set (xs ! j') by simp
```

with *no-junk*[*OF* ‹$j' <$ *length xs*›] **have** *bhc* (*length xs*) (*fst kv*) $= j'$
by(*simp add*: *bucket-ok-def*)
ultimately have [*simp*]: $i' = j'$ **by** *simp*
from ‹$i <$ *length* [$x \leftarrow xs$. $x \neq$ []]› **have** $i =$ *length* (*take* i [$x \leftarrow xs$. $x \neq$ []]) **by** *simp*
also from *eqi eqj* **have** *take* i [$x \leftarrow xs$. $x \neq$ []] $=$ *take* j [$x \leftarrow xs$. $x \neq$ []] **by** *simp*
finally show *False* **using** ‹$i \neq j$› ‹$j <$ *length* [$x \leftarrow xs$. $x \neq$ []]› **by** *simp*
qed
qed
moreover have *inj-on* (*map fst*) {$x \in$ *set xs*. $x \neq$ []}
proof(*rule inj-onI*)
fix x y
assume $x \in$ {$x \in$ *set xs*. $x \neq$ []} $y \in$ {$x \in$ *set xs*. $x \neq$ []} *map fst* $x =$ *map fst* y
hence $x \in$ *set xs* $y \in$ *set xs* $x \neq$ [] $y \neq$ [] **by** *auto*
from ‹$x \in$ *set xs*› **obtain** i **where** *xs* ! $i = x$ $i <$ *length xs* **unfolding** *set-conv-nth* **by** *fastforce*
from ‹$y \in$ *set xs*› **obtain** j **where** *xs* ! $j = y$ $j <$ *length xs* **unfolding** *set-conv-nth* **by** *fastforce*
from ‹$x \neq$ []› **obtain** k v x' **where** $x = (k, v)$ # x' **by**(*cases* x) *auto*
with *no-junk*[*OF* ‹$i <$ *length xs*›] ‹*xs* ! $i = x$›
have *bhc* (*length xs*) $k = i$ **by**(*auto simp add*: *bucket-ok-def*)
moreover from ‹*map fst* $x =$ *map fst* y› ‹$x = (k, v)$ # x'› **obtain** v' **where** $(k, v') \in$ *set* y **by** *fastforce*
with *no-junk*[*OF* ‹$j <$ *length xs*›] ‹*xs* ! $j = y$›
have *bhc* (*length xs*) $k = j$ **by**(*auto simp add*: *bucket-ok-def*)
ultimately have $i = j$ **by** *simp*
with ‹*xs* ! $i = x$› ‹*xs* ! $j = y$› **show** $x = y$ **by** *simp*
qed
ultimately show *distinct* [$ys \leftarrow$ *map* (*map fst*) *xs* . $ys \neq$ []]
by(*simp add*: *filter-map o-def distinct-map*)
next
fix *ys*
have A: $\bigwedge xs$. *distinct* (*map fst xs*) $=$ *list-map-invar xs*
by (*simp add*: *list-map-invar-def*)
assume $ys \in$ *set* (*map* (*map fst*) *xs*)
thus *distinct ys*
by(*clarsimp simp add*: *set-conv-nth* A) (*erule no-junk*(*2*))
next
fix *ys zs*
assume $ys \in$ *set* (*map* (*map fst*) *xs*) $zs \in$ *set* (*map* (*map fst*) *xs*) $ys \neq zs$
then obtain ys' zs' **where** [*simp*]: $ys =$ *map fst* ys' $zs =$ *map fst* zs'
and $ys' \in$ *set xs* $zs' \in$ *set xs* **by** *auto*
have *fst* ‘ *set* $ys' \cap$ *fst* ‘ *set* $zs' = \{\}$
proof(*rule equals0I*)
fix k
assume $k \in$ *fst* ‘ *set* $ys' \cap$ *fst* ‘ *set* zs'
then obtain v v' **where** $(k, v) \in$ *set* ys' $(k, v') \in$ *set* zs' **by**(*auto*)

from ‹*ys′* ∈ *set xs*› **obtain** *i* **where** *xs* ! *i* = *ys′* *i* < *length xs* **unfolding** *set-conv-nth* **by** *fastforce*
with ‹(*k*, *v*) ∈ *set ys′*› **have** *bhc* (*length xs*) *k* = *i* **by**(*auto dest*: *no-junk bucket-okD*)
moreover
from ‹*zs′* ∈ *set xs*› **obtain** *j* **where** *xs* ! *j* = *zs′* *j* < *length xs* **unfolding** *set-conv-nth* **by** *fastforce*
with ‹(*k*, *v′*) ∈ *set zs′*› **have** *bhc* (*length xs*) *k* = *j* **by**(*auto dest*: *no-junk bucket-okD*)
ultimately have *i* = *j* **by** *simp*
with ‹*xs* ! *i* = *ys′*› ‹*xs* ! *j* = *zs′*› **have** *ys′* = *zs′* **by** *simp*
with ‹*ys* ≠ *zs*› **show** *False* **by** *simp*
qed
thus *set ys* ∩ *set zs* = {} **by** *simp*
qed
qed

3.14.5 *ahm-α*

lemma *list-map-lookup-is-map-of*:
list-map-lookup op= *k l* = *map-of l k*
using *list-map-autoref-lookup-aux*[**where** *eq*=*op*= **and** *Rk*=*Id* **and** *Rv*=*Id*] **by** *force*
definition *ahm-α-aux bhc a* ≡
(λ*k*. *ahm-lookup-aux op*= *bhc k a*)
lemma *ahm-α-aux-def2*: *ahm-α-aux bhc a* = (λ*k*. *map-of* (*array-get a* (*bhc* (*array-length a*) *k*)) *k*)
unfolding *ahm-α-aux-def ahm-lookup-aux-def*
by (*simp add*: *list-map-lookup-is-map-of*)
lemma *ahm-α-def2*: *ahm-α bhc* (*HashMap a n*) = *ahm-α-aux bhc a*
unfolding *ahm-α-def ahm-α-aux-def* **by** *simp*

lemma *finite-dom-ahm-α-aux*:
assumes *is-bounded-hashcode op*= *bhc* *ahm-invar-aux bhc n a*
shows *finite* (*dom* (*ahm-α-aux bhc a*))
proof −
have *dom* (*ahm-α-aux bhc a*) ⊆ (⋃*h* ∈ *range* (*bhc* (*array-length a*) :: ′*a* ⇒ *nat*). *dom* (*map-of* (*array-get a h*)))
unfolding *ahm-α-aux-def2*
by(*force simp add*: *dom-map-of-conv-image-fst dest*: *map-of-SomeD*)
moreover have *finite* . . .
proof(*rule finite-UN-I*)
from ‹*ahm-invar-aux bhc n a*› **have** *array-length a* > *1* **by**(*simp add*: *ahm-invar-aux-def*)
hence *range* (*bhc* (*array-length a*) :: ′*a* ⇒ *nat*) ⊆ {*0*..<*array-length a*}
using *assms* **by** *force*
thus *finite* (*range* (*bhc* (*array-length a*) :: ′*a* ⇒ *nat*))
by(*rule finite-subset*) *simp*
qed(*rule finite-dom-map-of*)
ultimately show *?thesis* **by**(*rule finite-subset*)

```
qed

lemma ahm-α-aux-new-array[simp]:
  assumes bhc: is-bounded-hashcode op= bhc   1 < sz
  shows ahm-α-aux bhc (new-array [] sz) k = None
  using is-bounded-hashcodeD(2)[OF assms]
  unfolding ahm-α-aux-def ahm-lookup-aux-def by simp

lemma ahm-α-aux-conv-map-of-concat:
  assumes bhc: is-bounded-hashcode op= bhc
  assumes inv: ahm-invar-aux bhc n (Array xs)
  shows ahm-α-aux bhc (Array xs) = map-of (concat xs)
proof
  fix k
  show ahm-α-aux bhc (Array xs) k = map-of (concat xs) k
  proof(cases map-of (concat xs) k)
    case None
    hence k ∉ fst ‘ set (concat xs) by(simp add: map-of-eq-None-iff)
    hence k ∉ fst ‘ set (xs ! bhc (length xs) k)
    proof(rule contrapos-nn)
      assume k ∈ fst ‘ set (xs ! bhc (length xs) k)
      then obtain v where (k, v) ∈ set (xs ! bhc (length xs) k) by auto
      moreover from inv have bhc (length xs) k < length xs
        using bhc by (force simp: ahm-invar-aux-def)
      ultimately show k ∈ fst ‘ set (concat xs)
        by (force intro: rev-image-eqI)
    qed
    thus ?thesis unfolding None ahm-α-aux-def2
        by (simp add: map-of-eq-None-iff)
  next
    case (Some v)
    hence (k, v) ∈ set (concat xs) by(rule map-of-SomeD)
    then obtain ys where ys ∈ set xs   (k, v) ∈ set ys
      unfolding set-concat by blast
    from ‹ys ∈ set xs› obtain i j where i < length xs   xs ! i = ys
      unfolding set-conv-nth by auto
    with inv ‹(k, v) ∈ set ys›
    show ?thesis unfolding Some
      unfolding ahm-α-aux-def2
      by(force dest: bucket-okD simp add: ahm-invar-aux-def list-map-invar-def)
  qed
qed

lemma ahm-invar-aux-card-dom-ahm-α-auxD:
  assumes bhc: is-bounded-hashcode op= bhc
  assumes inv: ahm-invar-aux bhc n a
  shows card (dom (ahm-α-aux bhc a)) = n
proof(cases a)
  case (Array xs)[simp]
```

from *inv* **have** *card* (*dom* (*ahm-α-aux bhc* (*Array xs*))) = *card* (*dom* (*map-of* (*concat xs*)))
by(*simp add*: *ahm-α-aux-conv-map-of-concat*[*OF bhc*])
also from *inv* **have** *distinct* (*map fst* (*concat xs*))
by(*simp add*: *ahm-invar-distinct-fst-concatD*)
hence *card* (*dom* (*map-of* (*concat xs*))) = *length* (*concat xs*)
by(*rule card-dom-map-of*)
also have *length* (*concat xs*) = *foldl op* + *0* (*map length xs*)
by (*simp add*: *length-concat foldl-conv-fold add-commute fold-plus-listsum-rev*)
also from *inv*
have ... = *n* **unfolding** *foldl-map* **by**(*simp add*: *ahm-invar-aux-def array-foldl-foldl*)
finally show *?thesis* **by**(*simp*)
qed

lemma *finite-dom-ahm-α*:
assumes *is-bounded-hashcode op= bhc* *ahm-invar bhc hm*
shows *finite* (*dom* (*ahm-α bhc hm*))
using *assms* **by** (*cases hm*, *force intro*: *finite-dom-ahm-α-aux*
simp: *ahm-α-def2*)

3.14.6 *ahm-empty*

lemma *ahm-invar-aux-new-array*:
assumes *n* > *1*
shows *ahm-invar-aux bhc 0* (*new-array* [] *n*)
proof −
have *foldl* (λ*b* (*k*, *v*). *b* + *length v*) *0* (*zip* [*0*..<*n*] (*replicate n* [])) = *0*
by(*induct n*)(*simp-all add*: *replicate-Suc-conv-snoc del*: *replicate-Suc*)
with *assms* **show** *?thesis* **by**(*simp add*: *ahm-invar-aux-def array-foldl-new-array list-map-invar-def*)
qed

lemma *ahm-invar-new-hashmap-with*:
n > *1* ⟹ *ahm-invar bhc* (*new-hashmap-with n*)
by(*auto simp add*: *ahm-invar-def new-hashmap-with-def intro*: *ahm-invar-aux-new-array*)

lemma *ahm-α-new-hashmap-with*:
assumes *is-bounded-hashcode op= bhc* **and** *n* > *1*
shows *Map.empty* = *ahm-α bhc* (*new-hashmap-with n*)
unfolding *new-hashmap-with-def ahm-α-def*
using *is-bounded-hashcodeD*(*2*)[*OF assms*] **by** *force*

lemma *ahm-empty-impl*:
assumes *bhc*: *is-bounded-hashcode op= bhc*
assumes *def-size*: *def-size* > *1*
shows (*ahm-empty def-size*, *Map.empty*) ∈ *ahm-map-rel′ bhc*
proof−
from *def-size* **and** *ahm-α-new-hashmap-with*[*OF bhc def-size*] **and**
ahm-invar-new-hashmap-with[*OF def-size*]

```
  show ?thesis unfolding ahm-empty-def ahm-map-rel'-def br-def by force
qed

lemma param-ahm-empty[param]:
  assumes def-size: (def-size, def-size') ∈ nat-rel
  shows (ahm-empty def-size ,ahm-empty def-size') ∈
      ⟨Rk,Rv⟩ahm-map-rel
unfolding ahm-empty-def[abs-def] new-hashmap-with-def[abs-def]
    new-array-def[abs-def]
using assms by parametricity

lemma autoref-ahm-empty[autoref-rules]:
  fixes Rk :: ('kc×'ka) set
  assumes eq: GEN-OP eq op= (Rk → Rk → bool-rel)
  assumes bhc: SIDE-GEN-ALGO (is-bounded-hashcode eq bhc)
  assumes def-size: SIDE-GEN-ALGO (is-valid-def-hm-size TYPE('kc) def-size)
  shows (ahm-empty def-size, op-map-empty) ∈ ⟨Rk, Rv⟩ahm-rel bhc
proof−
  from eq have eq': (eq,op=) ∈ Rk → Rk → bool-rel by simp
  with bhc have is-bounded-hashcode op=
      (abstract-bounded-hashcode Rk bhc)
    unfolding autoref-tag-defs
    by blast
  thus ?thesis using assms
    unfolding op-map-empty-def
    unfolding ahm-rel-def is-valid-def-hm-size-def autoref-tag-defs
    apply (intro relcompI)
    apply (rule param-ahm-empty[of def-size def-size], simp)
    apply (blast intro: ahm-empty-impl)
    done
qed
```

3.14.7 *ahm-lookup*

```
lemma param-ahm-lookup[param]:
  assumes eq: GEN-OP eq op= (Rk → Rk → bool-rel)
  assumes bhc: is-bounded-hashcode eq bhc
  defines bhc'-def: bhc' ≡ abstract-bounded-hashcode Rk bhc
  assumes inv: ahm-invar bhc' m'
  assumes K: (k,k') ∈ Rk
  assumes M: (m,m') ∈ ⟨Rk,Rv⟩ahm-map-rel
  shows (ahm-lookup eq bhc k m, ahm-lookup op= bhc' k' m') ∈
          ⟨Rv⟩option-rel
proof−
  from eq have eq': (eq,op=) ∈ Rk → Rk → bool-rel by simp
  moreover from abstract-bhc-correct[OF eq' bhc]
      have bhc': (bhc,bhc') ∈ nat-rel → Rk → nat-rel unfolding bhc'-def .
  moreover from M obtain a a' n n' where
      [simp]: m = HashMap a n and [simp]: m' = HashMap a' n' and
```

```
    A: (a,a') ∈ ⟨⟨⟨Rk,Rv⟩prod-rel⟩list-rel⟩array-rel and N: (n,n') ∈ Id
      by (cases m, cases m', unfold ahm-map-rel-def, auto)
  moreover from inv and array-rel-imp-same-length[OF A]
    have array-length a > 1 by (simp add: ahm-invar-aux-def)
  with abstract-bhc-is-bhc[OF eq' bhc]
    have bhc' (array-length a) k' < array-length a
    unfolding bhc'-def by blast
  with bhc'[param-fo, OF - K]
    have bhc (array-length a) k < array-length a by simp
  ultimately show ?thesis using K
    unfolding ahm-lookup-def[abs-def] hashmap-rec-is-case
    by (simp, parametricity)
qed


lemma ahm-lookup-impl:
  assumes bhc: is-bounded-hashcode op= bhc
  shows (ahm-lookup op= bhc, op-map-lookup) ∈ Id → ahm-map-rel' bhc → Id
unfolding ahm-map-rel'-def br-def ahm-α-def by force

lemma autoref-ahm-lookup[autoref-rules]:
  assumes eq: GEN-OP eq op= (Rk → Rk → bool-rel)
  assumes
    bhc[unfolded autoref-tag-defs]: SIDE-GEN-ALGO (is-bounded-hashcode eq bhc)
  shows (ahm-lookup eq bhc, op-map-lookup) ∈
           Rk → ⟨Rk,Rv⟩ahm-rel bhc → ⟨Rv⟩option-rel
proof (intro fun-relI)
  let ?bhc' = abstract-bounded-hashcode Rk bhc
  fix k k' a m'
  assume K: (k,k') ∈ Rk
  assume M: (a,m') ∈ ⟨Rk,Rv⟩ahm-rel bhc
  from eq have (eq,op=) ∈ Rk → Rk → bool-rel by simp
  with bhc have bhc': is-bounded-hashcode op= ?bhc'
    by blast

  from M obtain a' where M1: (a,a') ∈ ⟨Rk,Rv⟩ahm-map-rel and
    M2: (a',m') ∈ ahm-map-rel' ?bhc' unfolding ahm-rel-def by blast
  hence inv: ahm-invar ?bhc' a'
    unfolding ahm-map-rel'-def br-def by simp

  from relcompI[OF param-ahm-lookup[OF eq bhc inv K M1]
                ahm-lookup-impl[param-fo, OF bhc' - M2]]
  show (ahm-lookup eq bhc k a, op-map-lookup k' m') ∈ ⟨Rv⟩option-rel
    by simp
qed
```

3.14.8 *ahm-iteratei*

```
abbreviation ahm-to-list ≡ it-to-list ahm-iteratei
```

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

lemma *param-ahm-iteratei-aux*[*param*]:
(*ahm-iteratei-aux*,*ahm-iteratei-aux*) ∈ ⟨⟨*Ra*⟩*list-rel*⟩*array-rel* →
(*Rb* → *bool-rel*) → (*Ra* → *Rb* → *Rb*) → *Rb* → *Rb*
unfolding *ahm-iteratei-aux-def*[*abs-def*] **by** *parametricity*

lemma *param-ahm-iteratei*[*param*]:
(*ahm-iteratei*,*ahm-iteratei*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel* →
(*Rb* → *bool-rel*) → (⟨*Rk*,*Rv*⟩*prod-rel* → *Rb* → *Rb*) → *Rb* → *Rb*
unfolding *ahm-iteratei-def*[*abs-def*] *hashmap-rec-is-case* **by** *parametricity*

lemma *param-ahm-to-list*[*param*]:
(*ahm-to-list*,*ahm-to-list*) ∈
⟨*Rk*, *Rv*⟩*ahm-map-rel* → ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*
unfolding *it-to-list-def*[*abs-def*] **by** *parametricity*

lemma *ahm-to-list-distinct*[*simp*,*intro*]:
assumes *bhc*: *is-bounded-hashcode op= bhc*
assumes *inv*: *ahm-invar bhc m*
shows *distinct* (*ahm-to-list m*)
proof−
obtain *n a* **where** [*simp*]: *m* = *HashMap a n* **by** (*cases m*)
obtain *l* **where** [*simp*]: *a* = *Array l* **by** (*cases a*)
from *inv* **show** *distinct* (*ahm-to-list m*) **unfolding** *it-to-list-def*
by (*force intro*: *distinct-mapI dest*: *ahm-invar-distinct-fst-concatD*)
qed

lemma *set-ahm-to-list*:
assumes *bhc*: *is-bounded-hashcode op= bhc*
assumes *ref*: (*m*,*m'*) ∈ *ahm-map-rel' bhc*
shows *map-to-set m'* = *set* (*ahm-to-list m*)
proof−
obtain *n a* **where** [*simp*]: *m* = *HashMap a n* **by** (*cases m*)
obtain *l* **where** [*simp*]: *a* = *Array l* **by** (*cases a*)
from *ref* **have** α[*simp*]: *m'* = *ahm-α bhc m* **and**
inv: *ahm-invar bhc m*
unfolding *ahm-map-rel'-def br-def* **by** *auto*

from *inv* **have** *length*: *length l* > *1*
unfolding *ahm-invar-def ahm-invar-aux-def* **by** *force*
from *inv* **have** *buckets-ok*: ⋀*h x*. *h* < *length l* ⟹ *x* ∈ *set* (*l*!*h*) ⟹
bhc (*length l*) (*fst x*) = *h*

```
    ⋀h. h < length l ⟹ distinct (map fst (l!h))
    by (simp-all add: ahm-invar-def ahm-invar-aux-def
                bucket-ok-def list-map-invar-def)

  show ?thesis unfolding it-to-list-def α ahm-α-def ahm-iteratei-def
    apply (simp add: list-map-lookup-is-map-of)
  proof (intro equalityI subsetI)
   case (goal1 x)
    let ?m = λk. map-of (l ! bhc (length l) k) k
    obtain k v where [simp]: x = (k, v) by (cases x)
    from goal1 have set-to-map (map-to-set ?m) k = Some v
      by (simp add: set-to-map-simp inj-on-fst-map-to-set)
    also note map-to-set-inverse
    finally have map-of (l ! bhc (length l) k) k = Some v .
    hence (k,v) ∈ set (l ! bhc (length l) k)
      by (simp add: map-of-is-SomeD)
    moreover have bhc (length l) k < length l using bhc length ..
    ultimately show ?case by force
  next
   case (goal2 x)
    obtain k v where [simp]: x = (k, v) by (cases x)
    from goal2 obtain h where h-props: (k,v) ∈ set (l!h)   h < length l
      by (force simp: set-conv-nth)
    moreover from h-props and buckets-ok
      have bhc (length l) k = h   distinct (map fst (l!h)) by auto
    ultimately have map-of (l ! bhc (length l) k) k = Some v
      by (force intro: map-of-is-SomeI)
    thus ?case by simp
  qed
qed


lemma ahm-iteratei-aux-impl:
  assumes inv: ahm-invar-aux bhc n a
  and bhc: is-bounded-hashcode op= bhc
  shows map-iterator (ahm-iteratei-aux a) (ahm-α-aux bhc a)
proof (cases a, rule)
  fix xs assume [simp]: a = Array xs
  show ahm-iteratei-aux a = foldli (concat xs)
    by (intro ext, simp)
  from ahm-invar-distinct-fst-concatD and inv
    show distinct (map fst (concat xs)) by simp
  from ahm-α-aux-conv-map-of-concat and assms
    show ahm-α-aux bhc a = map-of (concat xs) by simp
qed

lemma ahm-iteratei-impl:
  assumes inv: ahm-invar bhc m
```

and *bhc*: *is-bounded-hashcode op= bhc*
shows *map-iterator* (*ahm-iteratei m*) (*ahm-α bhc m*)
by (*insert assms*, *cases m*, *simp add*: *ahm-α-def2*,
erule (*1*) *ahm-iteratei-aux-impl*)

lemma *autoref-ahm-is-iterator*[*autoref-ga-rules*]:
assumes *eq*: *GEN-OP-tag* ((*eq*,*OP op* = ::: (*Rk* → *Rk* → *bool-rel*)) ∈ (*Rk* → *Rk* → *bool-rel*))
assumes *bhc*: *GEN-ALGO-tag* (*is-bounded-hashcode eq bhc*)
shows *is-map-to-list Rk Rv* (*ahm-rel bhc*) *ahm-to-list*
unfolding *is-map-to-list-def is-map-to-sorted-list-def*
proof (*intro allI impI*)
let *?bhc′* = *abstract-bounded-hashcode Rk bhc*
fix *a m′* **assume** *M*: (*a*,*m′*) ∈ ⟨*Rk*,*Rv*⟩*ahm-rel bhc*
from *eq* **and** *bhc* **have** *bhc′*: *is-bounded-hashcode op= ?bhc′*
unfolding *autoref-tag-defs*
apply (*rule-tac abstract-bhc-is-bhc*)
by *simp-all*

from *M* **obtain** *a′* **where** *M1*: (*a*,*a′*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel* **and**
M2: (*a′*,*m′*) ∈ *ahm-map-rel′ ?bhc′* **unfolding** *ahm-rel-def* **by** *blast*
hence *inv*: *ahm-invar ?bhc′ a′*
unfolding *ahm-map-rel′-def br-def* **by** *simp*

let *?l′* = *ahm-to-list a′*
from *param-ahm-to-list*[*param-fo*, *OF M1*]
have (*ahm-to-list a*, *?l′*) ∈ ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel* **.**
moreover from *ahm-to-list-distinct*[*OF bhc′ inv*]
have *distinct* (*ahm-to-list a′*) **.**
moreover from *set-ahm-to-list*[*OF bhc′ M2*]
have *map-to-set m′* = *set* (*ahm-to-list a′*) **.**
ultimately show ∃ *l′*. (*ahm-to-list a*, *l′*) ∈ ⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel* ∧
RETURN l′ ≤ *it-to-sorted-list*
(*key-rel* (λ- -. *True*)) (*map-to-set m′*)
by (*force simp*: *it-to-sorted-list-def key-rel-def*[*abs-def*])
qed

lemma *ahm-iteratei-aux-code*[*code*]:
ahm-iteratei-aux a c f σ = *idx-iteratei array-get array-length a c*
(λ*x*. *foldli x c f*) *σ*
proof(*cases a*)
case (*Array xs*)[*simp*]
have *ahm-iteratei-aux a c f σ* = *foldli* (*concat xs*) *c f σ* **by** *simp*
also have ... = *foldli xs c* (λ*x*. *foldli x c f*) *σ* **by** (*simp add*: *foldli-concat*)
also have ... = *idx-iteratei op* ! *length xs c* (λ*x*. *foldli x c f*) *σ*
by (*simp add*: *idx-iteratei-nth-length-conv-foldli*)
also have ... = *idx-iteratei array-get array-length a c* (λ*x*. *foldli x c f*) *σ*
by(*simp add*: *idx-iteratei-array-get-Array-conv-nth*)

finally show *?thesis* **.**
qed

3.14.9 *ahm-rehash*

lemma *array-length-ahm-rehash-aux′*:
array-length (*ahm-rehash-aux′ bhc n kv a*) = *array-length a*
by(*simp add*: *ahm-rehash-aux′-def Let-def*)

lemma *ahm-rehash-aux′-preserves-ahm-invar-aux*:
assumes *inv*: *ahm-invar-aux bhc n a*
and *bhc*: *is-bounded-hashcode op= bhc*
and *fresh*: $k \notin$ *fst* ‘ *set* (*array-get a* (*bhc* (*array-length a*) *k*))
shows *ahm-invar-aux bhc* (*Suc n*) (*ahm-rehash-aux′ bhc* (*array-length a*) (*k*, *v*) *a*)
(**is** *ahm-invar-aux bhc - ?a*)
proof(*rule ahm-invar-auxI*)
note *invD* = *ahm-invar-auxD*[*OF inv*]
let *?l* = *array-length a*
fix *h*
assume *h* < *array-length ?a*
hence *hlen*: *h* < *?l* **by**(*simp add*: *array-length-ahm-rehash-aux′*)
from *invD*(*1*,*2*)[*OF this*] **have** *bucket*: *bucket-ok bhc ?l h* (*array-get a h*)
and *dist*: *distinct* (*map fst* (*array-get a h*))
by (*simp-all add*: *list-map-invar-def*)
let *?h* = *bhc* (*array-length a*) *k*
from *hlen bucket* **show** *bucket-ok bhc* (*array-length ?a*) *h* (*array-get ?a h*)
by(*cases h* = *?h*)(*auto simp add*: *ahm-rehash-aux′-def Let-def array-length-ahm-rehash-aux′ array-get-array-set-other dest*: *bucket-okD intro*!: *bucket-okI*)
from *dist hlen fresh*
show *list-map-invar* (*array-get ?a h*)
unfolding *list-map-invar-def*
by(*cases h* = *?h*)(*auto simp add*: *ahm-rehash-aux′-def Let-def array-get-array-set-other*)
next
let *?f* = λ*n kvs*. *n* + *length kvs*
{ fix *n* :: *nat* **and** *xs* :: (*′a* × *′b*) *list list*
have *foldl ?f n xs* = *n* + *foldl ?f 0 xs*
by(*induct xs arbitrary*: *rule*: *rev-induct*) *simp-all* **}**
note *fold* = *this*
let *?h* = *bhc* (*array-length a*) *k*

obtain *xs* **where** *a* [*simp*]: *a* = *Array xs* **by**(*cases a*)
from *inv* **and** *bhc* **have** [*simp*]: *bhc* (*length xs*) *k* < *length xs*
by (*force simp add*: *ahm-invar-aux-def*)
have *xs*: *xs* = *take ?h xs* @ (*xs* ! *?h*) # *drop* (*Suc ?h*) *xs* **by**(*simp add*: *nth-drop′*)
from *inv* **have** *n* = *array-foldl* (λ- *n kvs*. *n* + *length kvs*) *0 a*
by(*auto elim*: *ahm-invar-auxE*)
hence *n* = *foldl ?f 0* (*take ?h xs*) + *length* (*xs* ! *?h*) + *foldl ?f 0* (*drop* (*Suc ?h*) *xs*)

by(*simp add*: *array-foldl-foldl*)(*subst xs*, *simp*, *subst* (*1 2 3 4*) *fold*, *simp*)
thus *Suc n* = *array-foldl* (λ- *n kvs*. *n* + *length kvs*) *0 ?a*
by(*simp add*: *ahm-rehash-aux′-def Let-def array-foldl-foldl foldl-list-update*)(*subst* (*1 2 3 4*) *fold*, *simp*)
next
from *inv* **have** *1* < *array-length a* **by**(*auto elim*: *ahm-invar-auxE*)
thus *1* < *array-length ?a* **by**(*simp add*: *array-length-ahm-rehash-aux′*)
qed

lemma *ahm-rehash-aux-correct*:
fixes *a* :: (′*k*×′*v*) *list array*
assumes *bhc*: *is-bounded-hashcode op= bhc*
and *inv*: *ahm-invar-aux bhc n a*
and *sz* > *1*
shows *ahm-invar-aux bhc n* (*ahm-rehash-aux bhc a sz*) (**is** *?thesis1*)
and *ahm-α-aux bhc* (*ahm-rehash-aux bhc a sz*) = *ahm-α-aux bhc a* (**is** *?thesis2*)
proof −
thm *ahm-rehash-aux′-def*
let *?a* = *ahm-rehash-aux bhc a sz*
def *I* ≡ λ*it a′*.
ahm-invar-aux bhc (*n* − *card it*) *a′*
∧ *array-length a′* = *sz*
∧ (∀ *k*. *if k* ∈ *it then*
ahm-α-aux bhc a′ k = *None*
else ahm-α-aux bhc a′ k = *ahm-α-aux bhc a k*)

note *iterator-rule* = *map-iterator-no-cond-rule-P*[
OF ahm-iteratei-aux-impl[*OF inv bhc*],
of I new-array [] *sz* *ahm-rehash-aux′ bhc sz* *I* {}]

from *inv* **have** *I* {} *?a* **unfolding** *ahm-rehash-aux-def*
proof(*intro iterator-rule*)
from *ahm-invar-aux-card-dom-ahm-α-auxD*[*OF bhc inv*]
have *card* (*dom* (*ahm-α-aux bhc a*)) = *n* **.**
moreover from *ahm-invar-aux-new-array*[*OF* ‹*1* < *sz*›]
have *ahm-invar-aux bhc 0* (*new-array* ([]::(′*k*×′*v*) *list*) *sz*) **.**
moreover {
fix *k*
assume *k* ∉ *dom* (*ahm-α-aux bhc a*)
hence *ahm-α-aux bhc a k* = *None* **by** *auto*
hence *ahm-α-aux bhc* (*new-array* [] *sz*) *k* = *ahm-α-aux bhc a k*
using *assms* **by** *simp*
}
ultimately show *I* (*dom* (*ahm-α-aux bhc a*)) (*new-array* [] *sz*)
using *assms* **by** (*simp add*: *I-def*)
next

```
fix k :: 'k
  and v :: 'v
  and it a'
assume k ∈ it    ahm-α-aux bhc a k = Some v
  and it-sub: it ⊆ dom (ahm-α-aux bhc a)
  and I: I it a'
from I have inv': ahm-invar-aux bhc (n − card it) a'
  and a'-eq-a: ⋀k. k ∉ it ⟹ ahm-α-aux bhc a' k = ahm-α-aux bhc a k
  and a'-None: ⋀k. k ∈ it ⟹ ahm-α-aux bhc a' k = None
  and [simp]: sz = array-length a'
  by (auto split: split-if-asm simp: I-def)
from it-sub finite-dom-ahm-α-aux[OF bhc inv]
    have finite it by(rule finite-subset)
moreover with ‹k ∈ it› have card it > 0 by (auto simp add: card-gt-0-iff)
moreover from finite-dom-ahm-α-aux[OF bhc inv] it-sub
    have card it ≤ card (dom (ahm-α-aux bhc a)) by (rule card-mono)
moreover have ... = n using inv
    by(simp add: ahm-invar-aux-card-dom-ahm-α-auxD[OF bhc])
ultimately have n − card (it − {k}) = (n − card it) + 1
    using ‹k ∈ it› by auto
moreover from ‹k ∈ it› have ahm-α-aux bhc a' k = None by (rule a'-None)
hence k ∉ fst ‘ set (array-get a' (bhc (array-length a') k))
    by (simp add: ahm-α-aux-def2 map-of-eq-None-iff)
ultimately have ahm-invar-aux bhc (n − card (it − {k}))
    (ahm-rehash-aux' bhc sz (k, v) a')
    using ahm-rehash-aux'-preserves-ahm-invar-aux[OF inv' bhc] by simp
moreover have array-length (ahm-rehash-aux' bhc sz (k, v) a') = sz
    by (simp add: array-length-ahm-rehash-aux')
moreover {
  fix k'
  assume k' ∈ it − {k}
  with is-bounded-hashcodeD(2)[OF bhc ‹1 < sz›, of k'] a'-None[of k']
  have ahm-α-aux bhc (ahm-rehash-aux' bhc sz (k, v) a') k' = None
      unfolding ahm-α-aux-def2
      by (cases bhc sz k = bhc sz k') (simp-all add:
              array-get-array-set-other ahm-rehash-aux'-def Let-def)
} moreover {
  fix k'
  assume k' ∉ it − {k}
  with is-bounded-hashcodeD(2)[OF bhc ‹1 < sz›, of k]
       is-bounded-hashcodeD(2)[OF bhc ‹1 < sz›, of k']
       a'-eq-a[of k'] ‹k ∈ it›
  have ahm-α-aux bhc (ahm-rehash-aux' bhc sz (k, v) a') k' =
           ahm-α-aux bhc a k'
      unfolding ahm-rehash-aux'-def Let-def
      using ‹ahm-α-aux bhc a k = Some v›
      unfolding ahm-α-aux-def2
    by(cases bhc sz k = bhc sz k') (case-tac [!]    k' = k,
       simp-all add: array-get-array-set-other)
```

```
    }
    ultimately show I (it − {k}) (ahm-rehash-aux′ bhc sz (k, v) a′)
      unfolding I-def by simp
  qed simp-all
  thus ?thesis1 ?thesis2 unfolding ahm-rehash-aux-def I-def by auto
qed

lemma ahm-rehash-correct:
  fixes hm :: ('k, 'v) hashmap
  assumes bhc: is-bounded-hashcode op= bhc
  and inv: ahm-invar bhc hm
  and sz > 1
  shows ahm-invar bhc (ahm-rehash bhc hm sz)
      ahm-α bhc (ahm-rehash bhc hm sz) = ahm-α bhc hm
proof−
  obtain a n where [simp]: hm = HashMap a n by (cases hm)
  from inv have ahm-invar-aux bhc n a by simp
  from ahm-rehash-aux-correct[OF bhc this ‹sz > 1›]
    show ahm-invar bhc (ahm-rehash bhc hm sz) and
        ahm-α bhc (ahm-rehash bhc hm sz) = ahm-α bhc hm
    by (simp-all add: ahm-α-def2)
qed
```

3.14.10 *ahm-update*

```
lemma param-mult[param]:
   (op∗, op∗) ∈ nat-rel → nat-rel → nat-rel by blast
lemma param-hm-grow[param]:
  (hm-grow, hm-grow) ∈ ⟨Rk,Rv⟩ahm-map-rel → nat-rel
unfolding hm-grow-def[abs-def] hashmap-rec-is-case by parametricity

lemma param-ahm-rehash-aux′[param]:
  assumes is-bounded-hashcode eq bhc
  assumes 1 < n
  assumes (bhc,bhc′) ∈ nat-rel → Rk → nat-rel
  assumes (n,n′) ∈ nat-rel and n = array-length a
  assumes (kv,kv′) ∈ ⟨Rk,Rv⟩prod-rel
  assumes (a,a′) ∈ ⟨⟨⟨Rk,Rv⟩prod-rel⟩list-rel⟩array-rel
  shows (ahm-rehash-aux′ bhc n kv a, ahm-rehash-aux′ bhc′ n′ kv′ a′) ∈
          ⟨⟨⟨Rk,Rv⟩prod-rel⟩list-rel⟩array-rel
proof−
  from assms have bhc n (fst kv) < array-length a by force
  thus ?thesis unfolding ahm-rehash-aux′-def[abs-def]
    hashmap-rec-is-case Let-def using assms by parametricity
qed


lemma param-new-array[param]:
  (new-array, new-array) ∈ R → nat-rel → ⟨R⟩array-rel
```

```
unfolding new-array-def[abs-def] by parametricity


lemma param-foldli-induct:
  assumes l: (l,l') ∈ ⟨Ra⟩list-rel
  assumes c: (c,c') ∈ Rb → bool-rel
  assumes σ: (σ,σ') ∈ Rb
  assumes Pσ: P σ σ'
  assumes f: ⋀a a' b b'. (a,a')∈Ra ⟹ (b,b')∈Rb ⟹ c b ⟹ c' b' ⟹
                     P b b' ⟹ (f a b, f' a' b') ∈ Rb ∧
                    P (f a b) (f' a' b')
  shows (foldli l c f σ, foldli l' c' f' σ') ∈ Rb
using c σ Pσ f by (induction arbitrary: σ σ' rule: list-rel-induct[OF l],
               auto dest!: fun-relD)

lemma param-foldli-induct-nocond:
  assumes l: (l,l') ∈ ⟨Ra⟩list-rel
  assumes σ: (σ,σ') ∈ Rb
  assumes Pσ: P σ σ'
  assumes f: ⋀a a' b b'. (a,a')∈Ra ⟹ (b,b')∈Rb ⟹ P b b' ⟹
              (f a b, f' a' b') ∈ Rb ∧ P (f a b) (f' a' b')
  shows (foldli l (λ-. True) f σ, foldli l' (λ-. True) f' σ') ∈ Rb
  using assms by (blast intro: param-foldli-induct)

lemma param-ahm-rehash-aux[param]:
  assumes eq: (eq,op=) ∈ Rk → Rk → bool-rel
  assumes bhc: is-bounded-hashcode eq bhc
  assumes bhc-rel: (bhc,bhc') ∈ nat-rel → Rk → nat-rel
  assumes A: (a,a') ∈ ⟨⟨⟨Rk,Rv⟩prod-rel⟩list-rel⟩array-rel
  assumes N: (n,n') ∈ nat-rel    1 < n
  shows (ahm-rehash-aux bhc a n, ahm-rehash-aux bhc' a' n') ∈
      ⟨⟨⟨Rk,Rv⟩prod-rel⟩list-rel⟩array-rel
proof−
  obtain l l' where [simp]: a = Array l    a' = Array l'
    by (cases a, cases a')
  from A have L: (l,l') ∈ ⟨⟨⟨Rk,Rv⟩prod-rel⟩list-rel⟩list-rel
    unfolding array-rel-def by simp
  hence L': (concat l, concat l') ∈ ⟨⟨Rk,Rv⟩prod-rel⟩list-rel
    by parametricity
  let ?P = λa a'. n = array-length a

  note induct-rule = param-foldli-induct-nocond[OF L', where P=?P]

  show ?thesis unfolding ahm-rehash-aux-def
    by (simp, induction rule: induct-rule, insert N bhc bhc-rel,
      auto intro: param-new-array[param-fo]
                param-ahm-rehash-aux'[param-fo]
      simp: array-length-ahm-rehash-aux')
qed
```

lemma *param-ahm-rehash*[*param*]:
  assumes *eq*: (*eq*,*op=*) ∈ *Rk* → *Rk* → *bool-rel*
  assumes *bhc*: *is-bounded-hashcode eq bhc*
  assumes *bhc-rel*: (*bhc*,*bhc′*) ∈ *nat-rel* → *Rk* → *nat-rel*
  assumes *M*: (*m*,*m′*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel*
  assumes *N*: (*n*,*n′*) ∈ *nat-rel*   *1* < *n*
  shows (*ahm-rehash bhc m n*, *ahm-rehash bhc′ m′ n′*) ∈
      ⟨*Rk*,*Rv*⟩*ahm-map-rel*
proof−
  obtain *a a′ k k′* **where** [*simp*]: *m* = *HashMap a k*   *m′* = *HashMap a′ k′*
    by (*cases m*, *cases m′*)
  hence *K*: (*k*,*k′*) ∈ *nat-rel* **and**
      A: (*a*,*a′*) ∈ ⟨⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*⟩*array-rel*
    using *M* **unfolding** *ahm-map-rel-def* **by** *simp-all*
  show *?thesis* **unfolding** *ahm-rehash-def*
    by (*simp*, *insert K A assms*, *parametricity*)
qed

lemma *param-load-factor*[*param*]:
  (*load-factor*, *load-factor*) ∈ *nat-rel*
  unfolding *load-factor-def* **by** *simp*

lemma *param-ahm-filled*[*param*]:
   (*ahm-filled*, *ahm-filled*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel* → *bool-rel*
  unfolding *ahm-filled-def*[*abs-def*] *hashmap-rec-is-case*
  by *parametricity*

lemma *param-ahm-update-aux*[*param*]:
  assumes *eq*: (*eq*, *op=*) ∈ *Rk* → *Rk* → *bool-rel*
  assumes *bhc*: *is-bounded-hashcode eq bhc*
  assumes *bhc-rel*: (*bhc*,*bhc′*) ∈ *nat-rel* → *Rk* → *nat-rel*
  assumes *inv*: *ahm-invar bhc′ m′*
  assumes *K*: (*k*,*k′*) ∈ *Rk*
  assumes *V*: (*v*,*v′*) ∈ *Rv*
  assumes *M*: (*m*,*m′*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel*
  shows (*ahm-update-aux eq bhc m k v*,
      ahm-update-aux op= bhc′ m′ k′ v′) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel*
proof−
  obtain *a a′ n n′* **where**
    [*simp*]: *m* = *HashMap a n* **and** [*simp*]: *m′* = *HashMap a′ n′*
    by (*cases m*, *cases m′*)
  from *M* **have** *A*: (*a*,*a′*) ∈ ⟨⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*⟩*array-rel* **and**
      N: (*n*,*n′*) ∈ *nat-rel*
    unfolding *ahm-map-rel-def* **by** *simp-all*

  from *inv* **have** *1* < *array-length a′*
    unfolding *ahm-invar-def ahm-invar-aux-def* **by** *force*

```
  hence 1 < array-length a
    by (simp add: array-rel-imp-same-length[OF A])
  with bhc have bhc-range: bhc (array-length a) k < array-length a by blast

  have option-compare: ⋀a a'. (a,a') ∈ ⟨Rv⟩option-rel ⟹
                       (a = None,a' = None) ∈ bool-rel by force
  have (array-get a (bhc (array-length a) k),
        array-get a' (bhc' (array-length a') k')) ∈
        ⟨⟨Rk,Rv⟩prod-rel⟩list-rel
    using A K bhc-rel bhc-range by parametricity
  note cmp = option-compare[OF param-list-map-lookup[param-fo, OF eq K this]]

  show ?thesis apply simp
   unfolding ahm-update-aux-def Let-def hashmap-rec-is-case
   using assms A N bhc-range cmp by parametricity
qed


lemma param-ahm-update[param]:
  assumes eq: (eq, op=) ∈ Rk → Rk → bool-rel
  assumes bhc: is-bounded-hashcode eq bhc
  assumes bhc-rel: (bhc,bhc') ∈ nat-rel → Rk → nat-rel
  assumes inv: ahm-invar bhc' m'
  assumes K: (k,k') ∈ Rk
  assumes V: (v,v') ∈ Rv
  assumes M: (m,m') ∈ ⟨Rk,Rv⟩ahm-map-rel
  shows (ahm-update eq bhc k v m, ahm-update op= bhc' k' v' m') ∈
          ⟨Rk,Rv⟩ahm-map-rel
proof−
  have 1 < hm-grow (ahm-update-aux eq bhc m k v) by simp
  with assms show ?thesis unfolding ahm-update-def[abs-def] Let-def
    by parametricity
qed



lemma length-list-map-update:
  length (list-map-update op= k v xs) =
   (if list-map-lookup op= k xs = None then Suc (length xs) else length xs)
     (is ?l-new = -)
proof (cases list-map-lookup op= k xs, simp-all)
  case None
   hence k ∉ dom (map-of xs) by (force simp: list-map-lookup-is-map-of)
   hence ⋀a. list-map-update-aux op= k v xs a = (k,v) # rev xs @ a
     by (induction xs, auto)
   thus ?l-new = Suc (length xs) unfolding list-map-update-def by simp
next
  case (Some v')
   hence (k,v') ∈ set xs unfolding list-map-lookup-is-map-of
```

```
      by (rule map-of-is-SomeD)
    hence ⋀a. length (list-map-update-aux op= k v xs a) =
        length xs + length a by (induction xs, auto)
    thus ?l-new = length xs unfolding list-map-update-def by simp
qed

lemma length-list-map-delete:
  length (list-map-delete op= k xs) =
    (if list-map-lookup op= k xs = None then length xs else length xs − 1)
        (is ?l-new = -)
proof (cases list-map-lookup op= k xs, simp-all)
  case None
    hence k ∉ dom (map-of xs) by (force simp: list-map-lookup-is-map-of)
    hence ⋀a. list-map-delete-aux op= k xs a = rev xs @ a
        by (induction xs, auto)
    thus ?l-new = length xs unfolding list-map-delete-def by simp
next
  case (Some v′)
    hence (k,v′) ∈ set xs unfolding list-map-lookup-is-map-of
        by (rule map-of-is-SomeD)
    hence ⋀a. k ∉ fst‘set a ⟹ length (list-map-delete-aux op= k xs a) =
        length xs + length a − 1 by (induction xs, auto)
    thus ?l-new = length xs − Suc 0 unfolding list-map-delete-def by simp
qed



lemma ahm-update-impl:
  assumes bhc: is-bounded-hashcode op= bhc
  shows (ahm-update op= bhc, op-map-update) ∈ (Id::(′k×′k) set) →
            (Id::(′v×′v) set) → ahm-map-rel′ bhc → ahm-map-rel′ bhc
proof (intro fun-relI, clarsimp)
  fix k::′k and v::′v and hm::(′k,′v) hashmap and  m::′k⇀′v
  assume (hm,m) ∈ ahm-map-rel′ bhc
  hence α: m = ahm-α bhc hm and inv: ahm-invar bhc hm
      unfolding ahm-map-rel′-def br-def by simp-all
  obtain a n where [simp]: hm = HashMap a n by (cases hm)

  have K: (k,k) ∈ Id and V: (v,v) ∈ Id by simp-all

  from inv have [simp]: 1 < array-length a
      unfolding ahm-invar-def ahm-invar-aux-def by simp
  hence bhc-range[simp]: ⋀k. bhc (array-length a) k < array-length a
      using bhc by blast

  let ?l = array-length a
  let ?h = bhc (array-length a) k
  let ?a′ = array-set a ?h (list-map-update op= k v (array-get a ?h))
  let ?n′ = if list-map-lookup op= k (array-get a ?h) = None
```

```
          then n + 1 else n

let ?list = array-get a (bhc ?l k)
let ?list′ = map-of ?list
have L: (?list, ?list′) ∈ br map-of list-map-invar
  using inv unfolding ahm-invar-def ahm-invar-aux-def br-def by simp
hence list: list-map-invar ?list by (simp-all add: br-def)
let ?list-new = list-map-update op= k v ?list
let ?list-new′ = op-map-update k v (map-of (?list))
from list-map-autoref-update2[param-fo, OF K V L]
  have list-updated: map-of ?list-new = ?list-new′
    list-map-invar ?list-new unfolding br-def by simp-all

have ahm-invar bhc (HashMap ?a′ ?n′) unfolding ahm-invar.simps
proof(rule ahm-invar-auxI)
 fix h
 assume h < array-length ?a′
 hence h-in-range: h < array-length a by simp
 with inv have bucket-ok: bucket-ok bhc ?l h (array-get a h)
   by(auto elim: ahm-invar-auxD)
 thus bucket-ok bhc (array-length ?a′) h (array-get ?a′ h)
  proof (cases h = bhc (array-length a) k)
   case False
    with bucket-ok show ?thesis
     by (intro bucket-okI, force simp add:
       array-get-array-set-other dest: bucket-okD)
  next
   case True
    show ?thesis
    proof (insert True, simp, intro bucket-okI)
     case (goal1 k′)
      show ?case
      proof (cases k = k′)
       case False
        from goal1 have k′ ∈ dom ?list-new′
          by (simp only: dom-map-of-conv-image-fst
            list-updated(1)[symmetric])
        hence k′ ∈ fst‘set ?list using False
          by (simp add: dom-map-of-conv-image-fst)
        from imageE[OF this] obtain x where
          fst x = k′ and x ∈ set ?list by force
        then obtain v′ where (k′,v′) ∈ set ?list
           by (cases x, simp)
        with bucket-okD[OF bucket-ok] and
          ‹h = bhc (array-length a) k›
          show ?thesis by simp
     qed simp
    qed
  qed
```

from ‹*h* < *array-length a*› *inv* **have** *list-map-invar* (*array-get a h*)
by(*auto dest*: *ahm-invar-auxD*)
with ‹*h* < *array-length a*›
show *list-map-invar* (*array-get ?a′ h*)
by (*cases h* = *?h*, *simp-all add*:
list-updated array-get-array-set-other)
next

obtain *xs* **where** *a* [*simp*]: *a* = *Array xs* **by**(*cases a*)

let *?f* = λ*n kvs*. *n* + *length kvs*
{ fix *n* :: *nat* **and** *xs* :: (′*a* × ′*b*) *list list*
have *foldl ?f n xs* = *n* + *foldl ?f 0 xs*
by(*induct xs arbitrary*: *rule*: *rev-induct*) *simp-all* **}**
note *fold* = *this*

from *inv* **have** [*simp*]: *bhc* (*length xs*) *k* < *length xs*
using *bhc-range* **by** *simp*
have *xs*: *xs* = *take ?h xs* @ (*xs* ! *?h*) # *drop* (*Suc ?h*) *xs*
by(*simp add*: *nth-drop′*)
from *inv* **have** *n* = *array-foldl* (λ- *n kvs*. *n* + *length kvs*) *0 a*
by (*force dest*: *ahm-invar-auxD*)
hence *n* = *foldl ?f 0* (*take ?h xs*) + *length* (*xs* ! *?h*) + *foldl ?f 0* (*drop* (*Suc ?h*) *xs*)
by(*simp add*: *array-foldl-foldl*)(*subst xs*, *simp*, *subst* (*1 2 3 4*) *fold*, *simp*)
thus *?n′* = *array-foldl* (λ- *n kvs*. *n* + *length kvs*) *0 ?a′*
apply(*simp add*: *ahm-rehash-aux′-def Let-def array-foldl-foldl foldl-list-update map-of-eq-None-iff*)
apply(*subst* (*1 2 3 4 5 6 7 8*) *fold*)
apply(*simp add*: *length-list-map-update*)
done
next
from *inv* **have** *1* < *array-length a* **by**(*auto elim*: *ahm-invar-auxE*)
thus *1* < *array-length ?a′* **by** *simp*
next
qed

moreover have *ahm-α bhc* (*ahm-update-aux op= bhc hm k v*) =
ahm-α bhc hm(*k* ↦ *v*)
proof
fix *k′*
show *ahm-α bhc* (*ahm-update-aux op= bhc hm k v*) *k′* = (*ahm-α bhc hm*(*k* ↦ *v*)) *k′*
proof (*cases bhc ?l k* = *bhc ?l k′*)
case *False*
thus *?thesis* **by** (*force simp add*: *Let-def*
ahm-α-def array-get-array-set-other)
next
case *True*

hence *bhc ?l k'* = *bhc ?l k* **by** *simp*
thus *?thesis* **by** (*auto simp add*: *Let-def ahm-α-def*
list-map-lookup-is-map-of list-updated)
qed
qed

ultimately have *ref*: (*ahm-update-aux op= bhc hm k v*,
m(*k* ↦ *v*)) ∈ *ahm-map-rel' bhc* (**is** (*?hm'*,-)∈-)
unfolding *ahm-map-rel'-def br-def* **using** α **by** (*auto simp*: *Let-def*)

show (*ahm-update op= bhc k v hm*, *m*(*k* ↦ *v*))
∈ *ahm-map-rel' bhc*
proof (*cases ahm-filled ?hm'*)
case *False*
with *ref* **show** *?thesis* **unfolding** *ahm-update-def*
by (*simp del*: *ahm-update-aux.simps*)
next
case *True*
from *ref* **have** (*ahm-rehash bhc ?hm'* (*hm-grow ?hm'*), *m*(*k* ↦ *v*)) ∈
ahm-map-rel' bhc **unfolding** *ahm-map-rel'-def br-def*
by (*simp del*: *ahm-update-aux.simps*
add: *ahm-rehash-correct*[*OF bhc*])
thus *?thesis* **unfolding** *ahm-update-def* **using** *True*
by (*simp del*: *ahm-update-aux.simps add*: *Let-def*)
qed
qed

lemma *autoref-ahm-update*[*autoref-rules*]:
assumes *eq*: *GEN-OP eq op=* (*Rk* → *Rk* → *bool-rel*)
assumes *bhc*[*unfolded autoref-tag-defs*]:
SIDE-GEN-ALGO (*is-bounded-hashcode eq bhc*)
shows (*ahm-update eq bhc*, *op-map-update*) ∈
Rk → *Rv* → ⟨*Rk*,*Rv*⟩*ahm-rel bhc* → ⟨*Rk*,*Rv*⟩*ahm-rel bhc*
proof (*intro fun-relI*)
let *?bhc'* = *abstract-bounded-hashcode Rk bhc*
fix *k k' v v' a m'*
assume *K*: (*k*,*k'*) ∈ *Rk* **and** *V*: (*v*,*v'*) ∈ *Rv*
assume *M*: (*a*,*m'*) ∈ ⟨*Rk*,*Rv*⟩*ahm-rel bhc*
from *eq* **have** *eq'*: (*eq*,*op=*) ∈ *Rk* → *Rk* → *bool-rel* **by** *simp*
with *bhc* **have** *bhc'*: *is-bounded-hashcode op= ?bhc'* **by** *blast*
from *abstract-bhc-correct*[*OF eq' bhc*]
have *bhc-rel*: (*bhc*,*?bhc'*) ∈ *nat-rel* → *Rk* → *nat-rel* **.**

from *M* **obtain** *a'* **where** *M1*: (*a*,*a'*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel* **and**
M2: (*a'*,*m'*) ∈ *ahm-map-rel' ?bhc'* **unfolding** *ahm-rel-def* **by** *blast*
hence *inv*: *ahm-invar ?bhc' a'*
unfolding *ahm-map-rel'-def br-def* **by** *simp*

from *relcompI*[*OF param-ahm-update*[*OF eq′ bhc bhc-rel inv K V M1*]
ahm-update-impl[*param-fo*, *OF bhc′ - - M2*]]
show (*ahm-update eq bhc k v a*, *op-map-update k′ v′ m′*) ∈
⟨*Rk*,*Rv*⟩*ahm-rel bhc* **unfolding** *ahm-rel-def* **by** *simp*
qed

3.14.11 *ahm-delete*

lemma *param-ahm-delete*[*param*]:
assumes *eq*: (*eq*,*op*=) ∈ *Rk* → *Rk* → *bool-rel*
assumes *isbhc*: *is-bounded-hashcode eq bhc*
assumes *bhc*: (*bhc*,*bhc′*) ∈ *nat-rel* → *Rk* → *nat-rel*
assumes *inv*: *ahm-invar bhc′ m′*
assumes *K*: (*k*,*k′*) ∈ *Rk*
assumes *M*: (*m*,*m′*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel*
shows
(*ahm-delete eq bhc k m*, *ahm-delete op*= *bhc′ k′ m′*) ∈
⟨*Rk*,*Rv*⟩*ahm-map-rel*
proof−
obtain *a a′ n n′* **where**
[*simp*]: *m* = *HashMap a n* **and** [*simp*]: *m′* = *HashMap a′ n′*
by (*cases m*, *cases m′*)
from *M* **have** *A*: (*a*,*a′*) ∈ ⟨⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*⟩*array-rel* **and**
N: (*n*,*n′*) ∈ *nat-rel*
unfolding *ahm-map-rel-def* **by** *simp-all*

from *inv* **have** *1* < *array-length a′*
unfolding *ahm-invar-def ahm-invar-aux-def* **by** *force*
hence *1* < *array-length a*
by (*simp add*: *array-rel-imp-same-length*[*OF A*])
with *isbhc* **have** *bhc-range*: *bhc* (*array-length a*) *k* < *array-length a* **by** *blast*

have *option-compare*: ⋀*a a′*. (*a*,*a′*) ∈ ⟨*Rv*⟩*option-rel* ⟹
(*a* = *None*,*a′* = *None*) ∈ *bool-rel* **by** *force*
have (*array-get a* (*bhc* (*array-length a*) *k*),
array-get a′ (*bhc′* (*array-length a′*) *k′*)) ∈
⟨⟨*Rk*,*Rv*⟩*prod-rel*⟩*list-rel*
using *A K bhc bhc-range* **by** *parametricity*
note *cmp* = *option-compare*[*OF param-list-map-lookup*[*param-fo*, *OF eq K this*]]

show *?thesis* **unfolding** ‹*m* = *HashMap a n*› ‹*m′* = *HashMap a′ n′*›
by (*simp only*: *ahm-delete.simps Let-def*,
insert eq isbhc bhc K A N bhc-range cmp, *parametricity*)
qed

lemma *ahm-delete-impl*:
assumes *bhc*: *is-bounded-hashcode op*= *bhc*
shows (*ahm-delete op*= *bhc*, *op-map-delete*) ∈ (*Id*::(′*k*×′*k*) *set*) →

```
          ahm-map-rel' bhc → ahm-map-rel' bhc
proof (intro fun-relI, clarsimp)
  fix k::'k and hm::('k,'v) hashmap and  m::'k⇀'v
  assume (hm,m) ∈ ahm-map-rel' bhc
  hence α: m = ahm-α bhc hm and inv: ahm-invar bhc hm
     unfolding ahm-map-rel'-def br-def by simp-all
  obtain a n where [simp]: hm = HashMap a n by (cases hm)

  have K: (k,k) ∈ Id by simp

  from inv have [simp]: 1 < array-length a
     unfolding ahm-invar-def ahm-invar-aux-def by simp
  hence bhc-range[simp]: ⋀k. bhc (array-length a) k < array-length a
     using bhc by blast

  let ?l = array-length a
  let ?h = bhc ?l k
  let ?a' = array-set a ?h (list-map-delete op= k (array-get a ?h))
  let ?n' = if list-map-lookup op= k (array-get a ?h) = None then n else n − 1
  let ?list = array-get a ?h let ?list' = map-of ?list
  let ?list-new = list-map-delete op= k ?list
  let ?list-new' = ?list' |` (−{k})
  from inv have (?list, ?list') ∈ br map-of list-map-invar
     unfolding br-def ahm-invar-def ahm-invar-aux-def by simp
  from list-map-autoref-delete2[param-fo, OF K this]
     have list-updated: map-of ?list-new = ?list-new'
         list-map-invar ?list-new by (simp-all add: br-def)

  have [simp]: array-length ?a' = ?l by simp

  have ahm-invar-aux bhc ?n' ?a'
  proof(rule ahm-invar-auxI)
    fix h
    assume h < array-length ?a'
    hence h-in-range[simp]: h < array-length a by simp
    with inv have inv': bucket-ok bhc ?l h (array-get a h)    1 < ?l
      list-map-invar (array-get a h) by (auto elim: ahm-invar-auxE)

    show bucket-ok bhc (array-length ?a') h (array-get ?a' h)
      proof (cases h = bhc ?l k)
        case False thus ?thesis using inv'
            by (simp add: array-get-array-set-other)
      next
        case True thus ?thesis
        proof (simp, intro bucket-okI)
          case (goal1 k')
              show ?case
              proof (cases k = k')
                case False
```

from *goal1* **have** $k' \in$ *dom ?list-new*$'$
by (*simp only*: *dom-map-of-conv-image-fst*
list-updated(*1*)[*symmetric*])
hence $k' \in$ *fst*'*set ?list* **using** *False*
by (*simp add*: *dom-map-of-conv-image-fst*)
from *imageE*[*OF this*] **obtain** x **where**
fst $x = k'$ **and** $x \in$ *set ?list* **by** *force*
then obtain v' **where** $(k',v') \in$ *set ?list*
by (*cases* x, *simp*)
with *bucket-okD*[*OF inv*$'$(*1*)] **and**
⟨$h =$ *bhc* (*array-length a*) k⟩
show *?thesis* **by** *blast*
qed *simp*
qed
qed

from *inv*$'$(*3*) ⟨$h <$ *array-length a*⟩
show *list-map-invar* (*array-get ?a*$'$ *h*)
by (*cases* $h =$ *?h*, *simp-all add*:
list-updated array-get-array-set-other)
next
obtain *xs* **where** *a* [*simp*]: *a* = *Array xs* **by**(*cases a*)

let *?f* = λn *kvs*. n + *length* (*kvs*::($'k \times 'v$) *list*)
{ fix n :: *nat* **and** *xs* :: ($'k \times 'v$) *list list*
have *foldl ?f n xs* = n + *foldl ?f 0 xs*
by(*induct xs arbitrary*: *rule*: *rev-induct*) *simp-all* **}**
note *fold* = *this*

from *bhc-range* **have** [*simp*]: *bhc* (*length xs*) $k <$ *length xs* **by** *simp*
moreover
have *xs*: *xs* = *take ?h xs* @ (*xs* ! *?h*) # *drop* (*Suc ?h*) *xs* **by**(*simp add*: *nth-drop*$'$)
from *inv* **have** n = *array-foldl* (λ- n *kvs*. n + *length kvs*) *0 a*
by(*auto elim*: *ahm-invar-auxE*)
hence n = *foldl ?f 0* (*take ?h xs*) + *length* (*xs* ! *?h*) + *foldl ?f 0* (*drop* (*Suc* *?h*) *xs*)
by(*simp add*: *array-foldl-foldl*)(*subst xs*, *simp*, *subst* (*1 2 3 4*) *fold*, *simp*)
moreover have $\bigwedge v$ *a b*. *list-map-lookup op*= k (*xs* ! *?h*) = *Some v*
$\Longrightarrow$ *a* + (*length* (*xs* ! *?h*) − *1*) + *b* = *a* + *length* (*xs* ! *?h*) + *b* − *1*
by (*cases xs* ! *?h*, *simp-all*)
ultimately show *?n*$'$ = *array-foldl* (λ- n *kvs*. n + *length kvs*) *0 ?a*$'$
apply(*simp add*: *array-foldl-foldl foldl-list-update map-of-eq-None-iff*)
apply(*subst* (*1 2 3 4 5 6 7 8*) *fold*)
apply (*intro conjI impI*)
apply(*auto simp add*: *length-list-map-delete*)
done
next

from *inv* **show** *1* < *array-length ?a*$'$ **by**(*auto elim*: *ahm-invar-auxE*)

```
  qed
  hence ahm-invar bhc (HashMap ?a' ?n') by simp

  moreover have ahm-α-aux bhc ?a' = ahm-α-aux bhc a |` (− {k})
  proof
    fix k' :: 'k
    show ahm-α-aux bhc ?a' k' = (ahm-α-aux bhc a |` (− {k})) k'
    proof (cases bhc ?l k' = ?h)
      case False
        hence k ≠ k' by force
        thus ?thesis using False unfolding ahm-α-aux-def
            by (simp add: array-get-array-set-other
                          list-map-lookup-is-map-of)
    next
      case True
        thus ?thesis unfolding ahm-α-aux-def
            by (simp add: list-map-lookup-is-map-of
                       list-updated(1) restrict-map-def)
    qed
  qed
  hence ahm-α bhc (HashMap ?a' ?n') = op-map-delete k m
      unfolding op-map-delete-def by (simp add: ahm-α-def2 α)

  ultimately have (HashMap ?a' ?n', op-map-delete k m) ∈ ahm-map-rel' bhc
      unfolding ahm-map-rel'-def br-def by simp

  thus (ahm-delete op= bhc k hm, m |` (−{k})) ∈ ahm-map-rel' bhc
      by (simp only: ‹hm = HashMap a n› ahm-delete.simps Let-def
                op-map-delete-def, force)
qed

lemma autoref-ahm-delete[autoref-rules]:
  assumes eq: GEN-OP eq op= (Rk → Rk → bool-rel)
  assumes bhc[unfolded autoref-tag-defs]:
    SIDE-GEN-ALGO (is-bounded-hashcode eq bhc)
  shows (ahm-delete eq bhc, op-map-delete) ∈
              Rk → ⟨Rk,Rv⟩ahm-rel bhc → ⟨Rk,Rv⟩ahm-rel bhc
proof (intro fun-relI)
  let ?bhc' = abstract-bounded-hashcode Rk bhc
  fix k k' a m'
  assume K: (k,k') ∈ Rk
  assume M: (a,m') ∈ ⟨Rk,Rv⟩ahm-rel bhc
  from eq have eq': (eq,op=) ∈ Rk → Rk → bool-rel by simp
  with bhc have bhc': is-bounded-hashcode op= ?bhc' by blast
  from abstract-bhc-correct[OF eq' bhc]
      have bhc-rel: (bhc,?bhc') ∈ nat-rel → Rk → nat-rel .

  from M obtain a' where M1: (a,a') ∈ ⟨Rk,Rv⟩ahm-map-rel and
      M2: (a',m') ∈ ahm-map-rel' ?bhc' unfolding ahm-rel-def by blast
```

hence *inv*: *ahm-invar ?bhc′ a′*
unfolding *ahm-map-rel′-def br-def* **by** *simp*

from *relcompI*[*OF param-ahm-delete*[*OF eq′ bhc bhc-rel inv K M1*]
ahm-delete-impl[*param-fo*, *OF bhc′ - M2*]]
show (*ahm-delete eq bhc k a*, *op-map-delete k′ m′*) ∈
⟨*Rk*,*Rv*⟩*ahm-rel bhc* **unfolding** *ahm-rel-def* **by** *simp*
qed

3.14.12 Various simple operations

lemma *param-ahm-isEmpty*[*param*]:
(*ahm-isEmpty*, *ahm-isEmpty*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel* → *bool-rel*
unfolding *ahm-isEmpty-def*[*abs-def*] *hashmap-rec-is-case*
by *parametricity*

lemma *param-ahm-isSng*[*param*]:
(*ahm-isSng*, *ahm-isSng*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel* → *bool-rel*
unfolding *ahm-isSng-def*[*abs-def*] *hashmap-rec-is-case*
by *parametricity*

lemma *param-ahm-size*[*param*]:
(*ahm-size*, *ahm-size*) ∈ ⟨*Rk*,*Rv*⟩*ahm-map-rel* → *nat-rel*
unfolding *ahm-size-def*[*abs-def*] *hashmap-rec-is-case*
by *parametricity*

lemma *ahm-isEmpty-impl*:
assumes *is-bounded-hashcode op= bhc*
shows (*ahm-isEmpty*, *op-map-isEmpty*) ∈ *ahm-map-rel′ bhc* → *bool-rel*
proof (*intro fun-relI*)
fix *hm m* **assume** *rel*: (*hm*,*m*) ∈ *ahm-map-rel′ bhc*
obtain *a n* **where** [*simp*]: *hm* = *HashMap a n* **by** (*cases hm*)
from *rel* **have** α: *m* = *ahm-α-aux bhc a* **and** *inv*: *ahm-invar-aux bhc n a*
unfolding *ahm-map-rel′-def br-def* **by** (*simp-all add*: *ahm-α-def2*)
from *ahm-invar-aux-card-dom-ahm-α-auxD*[*OF assms inv*,*symmetric*] **and**
finite-dom-ahm-α-aux[*OF assms inv*]
show (*ahm-isEmpty hm*, *op-map-isEmpty m*) ∈ *bool-rel*
unfolding *ahm-isEmpty-def op-map-isEmpty-def*
by (*simp add*: α *card-eq-0-iff*)
qed

lemma *ahm-isSng-impl*:
assumes *is-bounded-hashcode op= bhc*
shows (*ahm-isSng*, *op-map-isSng*) ∈ *ahm-map-rel′ bhc* → *bool-rel*
proof (*intro fun-relI*)
fix *hm m* **assume** *rel*: (*hm*,*m*) ∈ *ahm-map-rel′ bhc*
obtain *a n* **where** [*simp*]: *hm* = *HashMap a n* **by** (*cases hm*)
from *rel* **have** α: *m* = *ahm-α-aux bhc a* **and** *inv*: *ahm-invar-aux bhc n a*

```
    unfolding ahm-map-rel'-def br-def by (simp-all add: ahm-α-def2)
  note n-props[simp] = ahm-invar-aux-card-dom-ahm-α-auxD[OF assms inv,symmetric]
  note finite-dom[simp] = finite-dom-ahm-α-aux[OF assms inv]
  show (ahm-isSng hm, op-map-isSng m) ∈ bool-rel
    by (force simp add: α[symmetric] dom-eq-singleton-conv
          dest!: card-eq-SucD)
qed

lemma ahm-size-impl:
  assumes is-bounded-hashcode op= bhc
  shows (ahm-size, op-map-size) ∈ ahm-map-rel' bhc → nat-rel
proof (intro fun-relI)
  fix hm m assume rel: (hm,m) ∈ ahm-map-rel' bhc
  obtain a n where [simp]: hm = HashMap a n by (cases hm)
  from rel have α: m = ahm-α-aux bhc a and inv: ahm-invar-aux bhc n a
    unfolding ahm-map-rel'-def br-def by (simp-all add: ahm-α-def2)
  from ahm-invar-aux-card-dom-ahm-α-auxD[OF assms inv,symmetric]
    show (ahm-size hm, op-map-size m) ∈ nat-rel
      unfolding ahm-isEmpty-def op-map-isEmpty-def
      by (simp add: α card-eq-0-iff)
qed


lemma autoref-ahm-isEmpty[autoref-rules]:
  assumes eq: GEN-OP eq op= (Rk → Rk → bool-rel)
  assumes bhc[unfolded autoref-tag-defs]:
    SIDE-GEN-ALGO (is-bounded-hashcode eq bhc)
  shows (ahm-isEmpty, op-map-isEmpty) ∈ ⟨Rk,Rv⟩ahm-rel bhc → bool-rel
proof (intro fun-relI)
  let ?bhc' = abstract-bounded-hashcode Rk bhc
  fix k k' a m'
  assume M: (a,m') ∈ ⟨Rk,Rv⟩ahm-rel bhc
  from eq have (eq,op=) ∈ Rk → Rk → bool-rel by simp
  with bhc have bhc': is-bounded-hashcode op= ?bhc'
   by blast

  from M obtain a' where M1: (a,a') ∈ ⟨Rk,Rv⟩ahm-map-rel and
    M2: (a',m') ∈ ahm-map-rel' ?bhc' unfolding ahm-rel-def by blast

  from relcompI[OF param-ahm-isEmpty[param-fo, OF M1]
                 ahm-isEmpty-impl[param-fo, OF bhc' M2]]
  show (ahm-isEmpty a, op-map-isEmpty m') ∈ bool-rel by simp
qed

lemma autoref-ahm-isSng[autoref-rules]:
  assumes eq: GEN-OP eq op= (Rk → Rk → bool-rel)
  assumes bhc[unfolded autoref-tag-defs]:
    SIDE-GEN-ALGO (is-bounded-hashcode eq bhc)
  shows (ahm-isSng, op-map-isSng) ∈ ⟨Rk,Rv⟩ahm-rel bhc → bool-rel
```

```
proof (intro fun-relI)
  let ?bhc' = abstract-bounded-hashcode Rk bhc
  fix k k' a m'
  assume M: (a,m') ∈ ⟨Rk,Rv⟩ahm-rel bhc
  from eq have (eq,op=) ∈ Rk → Rk → bool-rel by simp
  with bhc have bhc': is-bounded-hashcode op= ?bhc'
    by blast

  from M obtain a' where M1: (a,a') ∈ ⟨Rk,Rv⟩ahm-map-rel and
      M2: (a',m') ∈ ahm-map-rel' ?bhc' unfolding ahm-rel-def by blast

  from relcompI[OF param-ahm-isSng[param-fo, OF M1]
                ahm-isSng-impl[param-fo, OF bhc' M2]]
  show (ahm-isSng a, op-map-isSng m') ∈ bool-rel by simp
qed

lemma autoref-ahm-size[autoref-rules]:
  assumes eq: GEN-OP eq op= (Rk → Rk → bool-rel)
  assumes bhc[unfolded autoref-tag-defs]:
      SIDE-GEN-ALGO (is-bounded-hashcode eq bhc)
  shows (ahm-size, op-map-size) ∈ ⟨Rk,Rv⟩ahm-rel bhc → nat-rel
proof (intro fun-relI)
  let ?bhc' = abstract-bounded-hashcode Rk bhc
  fix k k' a m'
  assume M: (a,m') ∈ ⟨Rk,Rv⟩ahm-rel bhc
  from eq have (eq,op=) ∈ Rk → Rk → bool-rel by simp
  with bhc have bhc': is-bounded-hashcode op= ?bhc'
    by blast

  from M obtain a' where M1: (a,a') ∈ ⟨Rk,Rv⟩ahm-map-rel and
      M2: (a',m') ∈ ahm-map-rel' ?bhc' unfolding ahm-rel-def by blast

  from relcompI[OF param-ahm-size[param-fo, OF M1]
                ahm-size-impl[param-fo, OF bhc' M2]]
  show (ahm-size a, op-map-size m') ∈ nat-rel by simp
qed

lemma ahm-map-rel-sv[relator-props]:
  assumes SK: single-valued Rk
  assumes SV: single-valued Rv
  shows single-valued (⟨Rk, Rv⟩ahm-map-rel)
proof -
  from SK SV have 1: single-valued (⟨⟨⟨Rk, Rv⟩prod-rel⟩list-rel⟩array-rel)
    by (tagged-solver)

  thus ?thesis
    unfolding ahm-map-rel-def
    by (auto intro: single-valuedI dest: single-valuedD[OF 1])
qed
```

lemma *ahm-rel-sv*[*relator-props*]:
  ⟦*single-valued Rk*; *single-valued Rv*⟧
  ⟹ *single-valued* (⟨*Rk*,*Rv*⟩*ahm-rel bhc*)
  unfolding *ahm-rel-def ahm-map-rel′-def*
  by (*tagged-solver* (*keep*))

lemma *rbt-map-rel-finite*[*relator-props*]:
  assumes *A*[*simplified*]: *GEN-ALGO-tag* (*is-bounded-hashcode eq bhc*)
  assumes *eq*[*unfolded GEN-OP-tag-def*]:
    GEN-OP-tag ((*eq*, *op*=) ∈ (*Rk* → *Rk* → *bool-rel*))
  shows *finite-map-rel* (⟨*Rk*,*Rv*⟩*ahm-rel bhc*)
  unfolding *ahm-rel-def finite-map-rel-def ahm-map-rel′-def br-def*
  apply *auto*
  apply (*case-tac y*)
  apply (*auto simp*: *ahm-α-def2*)
  thm *finite-dom-ahm-α-aux*
  apply (*rule finite-dom-ahm-α-aux*)
  apply (*rule abstract-bhc-is-bhc*)
  apply (*rule eq*)
  apply (*rule A*)
  apply *assumption*
  done

3.14.13 Proper iterator proofs

lemma *pi-ahm*[*icf-proper-iteratorI*]:
  proper-it (*ahm-iteratei m*) (*ahm-iteratei m*)
proof−
  obtain *a n* **where** [*simp*]: *m* = *HashMap a n* **by** (*cases m*)
  then obtain *l* **where** [*simp*]: *a* = *Array l* **by** (*cases a*)
  thus *?thesis*
    unfolding *proper-it-def list-map-iteratei-def*
    by (*simp add*: *ahm-iteratei-aux-def*, *blast*)
qed

lemma *pi′-ahm*[*icf-proper-iteratorI*]:
  proper-it′ ahm-iteratei ahm-iteratei
  by (*rule proper-it′I*, *rule pi-ahm*)

lemmas *autoref-ahm-rules* =
  autoref-ahm-empty
  autoref-ahm-lookup
  autoref-ahm-update
  autoref-ahm-delete

```
  autoref-ahm-isEmpty
  autoref-ahm-isSng
  autoref-ahm-size

lemmas autoref-ahm-rules-hashable[autoref-rules-raw]
  = autoref-ahm-rules[where Rk=Rk::(-×-::hashable) set, standard]


end
```

3.15 List Based Sets

```
theory Impl-List-Set
imports
  ../Gen/Gen-Iterator
  ../Intf/Intf-Set
  ../Lib/Proper-Iterator
begin

  lemma list-all2-refl-conv:
    list-all2 P xs xs ⟷ (∀ x∈set xs. P x x)
    by (induct xs) auto

  primrec glist-member :: ('a⇒'a⇒bool) ⇒ 'a ⇒ 'a list ⇒ bool where
    glist-member eq x [] ⟷ False
  | glist-member eq x (y#ys) ⟷ eq x y ∨ glist-member eq x ys

  lemma param-glist-member[param]:
    (glist-member,glist-member)∈(Ra→Ra→Id) → Ra → ⟨Ra⟩list-rel → Id
    unfolding glist-member-def
    by (parametricity)

  lemma list-member-alt: List.member = (λl x. glist-member op = x l)
  proof (intro ext)
    fix x l
    show List.member l x = glist-member op = x l
      by (induct l) (auto simp: List.member-rec)
  qed

  thm List.insert-def
  definition
    glist-insert eq x xs = (if glist-member eq x xs then xs else x#xs)

  lemma param-glist-insert[param]:
    (glist-insert, glist-insert) ∈ (R→R→Id) → R → ⟨R⟩list-rel → ⟨R⟩list-rel
    unfolding glist-insert-def[abs-def]
    by (parametricity)
```

```
primrec glist-delete-aux1 :: ('a⇒'a⇒bool) ⇒ 'a ⇒ 'a list ⇒ 'a list where
  glist-delete-aux1 eq x [] = []
| glist-delete-aux1 eq x (y#ys) = (
    if eq x y then
      ys
    else y#glist-delete-aux1 eq x ys)

primrec rev-append where
  rev-append [] ac = ac
| rev-append (x#xs) ac = rev-append xs (x#ac)

lemma rev-append-eq: rev-append l ac = rev l @ ac
  by (induct l arbitrary: ac) auto

primrec glist-delete-aux2 :: ('a⇒'a⇒-) ⇒ - where
  glist-delete-aux2 eq ac x [] = ac
| glist-delete-aux2 eq ac x (y#ys) = (if eq x y then rev-append ys ac else
    glist-delete-aux2 eq (y#ac) x ys
  )

lemma glist-delete-aux2-eq1:
  glist-delete-aux2 eq ac x l = rev (glist-delete-aux1 eq x l) @ ac
  by (induct l arbitrary: ac) (auto simp: rev-append-eq)

definition glist-delete eq x l = glist-delete-aux2 eq [] x l

lemma param-glist-delete[param]:
  (glist-delete, glist-delete) ∈ (R→R→Id) → R → ⟨R⟩list-rel → ⟨R⟩list-rel
  unfolding glist-delete-def[abs-def]
    glist-delete-aux2-def
    rev-append-def
  by (parametricity)

definition
  list-set-rel-internal-def: list-set-rel R ≡ ⟨R⟩list-rel O br set distinct

lemma list-rel-Range:
  ∀x'∈set l'. x' ∈ Range R ⟹ l' ∈ Range (⟨R⟩list-rel)
proof (induction l')
  case Nil thus ?case by force
next
  case (Cons x' xs')
    then obtain xs where (xs,xs') ∈ ⟨R⟩ list-rel by force
    moreover from Cons.prems obtain x where (x,x') ∈ R by force
    ultimately have (x#xs, x'#xs') ∈ ⟨R⟩ list-rel by simp
    thus ?case ..
qed

  lemma list-set-rel-def: ⟨R⟩list-set-rel = ⟨R⟩list-rel O br set distinct
```

unfolding *list-set-rel-internal-def*[*abs-def*] **by** (*simp add*: *relAPP-def*)

All finite sets can be represented

lemma *list-set-rel-range*:
Range (⟨*R*⟩*list-set-rel*) = { *S*. *finite S* ∧ *S*⊆*Range R* }
(**is** *?A* = *?B*)
proof (*intro equalityI subsetI*)
fix *s'* **assume** *s'* ∈ *?A*
then obtain *l l'* **where** *A*: (*l*,*l'*) ∈ ⟨*R*⟩*list-rel* **and**
B: *s'* = *set l'* **and** *C*: *distinct l'*
unfolding *list-set-rel-def br-def* **by** *blast*
moreover have *set l'* ⊆ *Range R*
by (*induction rule*: *list-rel-induct*[*OF A*], *auto*)
ultimately show *s'* ∈ *?B* **by** *simp*
next
fix *s'* **assume** *A*: *s'* ∈ *?B*
then obtain *l'* **where** *B*: *set l'* = *s'* **and** *C*: *distinct l'*
using *finite-distinct-list* **by** *blast*
hence (*l'*,*s'*) ∈ *br set distinct* **by** (*simp add*: *br-def*)

moreover from *A* **and** *B* **have** ∀ *x*∈*set l'*. *x* ∈ *Range R* **by** *blast*
from *list-rel-Range*[*OF this*] **obtain** *l*
where (*l*,*l'*) ∈ ⟨*R*⟩*list-rel* **by** *blast*

ultimately show *s'* ∈ *?A* **unfolding** *list-set-rel-def* **by** *fast*
qed

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

lemma *list-set-rel-finite*[*autoref-ga-rules*]:
finite-set-rel (⟨*R*⟩*list-set-rel*)
unfolding *finite-set-rel-def list-set-rel-def*
by (*auto simp*: *br-def*)

lemma *list-set-rel-sv*[*relator-props*]:
single-valued R ⟹ *single-valued* (⟨*R*⟩*list-set-rel*)
unfolding *list-set-rel-def*
by *tagged-solver*

lemma *Id-comp-Id*: *Id O Id* = *Id* **by** *simp*

lemma *glist-member-id-impl*:
(*glist-member op* =, *op* ∈) ∈ *Id* → *br set distinct* → *Id*
proof (*intro fun-relI*)
case (*goal1 x x' l s'*) **thus** *?case*
by (*induct l arbitrary*: *s'*) (*auto simp*: *br-def*)
qed

lemma *glist-insert-id-impl*:
  (*glist-insert op* =, *Set.insert*) ∈ *Id* → *br set distinct* → *br set distinct*
proof −
  have *IC*: ⋀*x s*. *insert x s* = (*if x*∈*s then s else insert x s*) **by** *auto*

  show *?thesis*
    apply (*intro fun-relI*)
    apply (*subst IC*)
    unfolding *glist-insert-def*
    apply (*parametricity add*: *glist-member-id-impl*)
    apply (*auto simp*: *br-def*)
    done
qed

lemma *glist-delete-id-impl*:
  (*glist-delete op* =, λ*x s*. *s*−{*x*})
  ∈ *Id*→*br set distinct* → *br set distinct*
proof (*intro fun-relI*)
  case (*goal1 x x′ l s′*) **thus** *?case*
    apply (*simp add*: *glist-delete-aux2-eq1 glist-delete-def*)
    apply (*induct l arbitrary*: *s′*)
    apply (*auto simp add*: *br-def*)
    done
qed

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

lemma *list-set-autoref-member*[*autoref-rules*]:
  assumes *GEN-OP eq op*= (*R*→*R*→*Id*)
  shows (*glist-member eq*,*op* ∈) ∈ *R* → ⟨*R*⟩*list-set-rel* → *Id*
  using *assms*
  apply (*intro fun-relI*)
  unfolding *list-set-rel-def*
  apply (*erule relcompE*)
  apply (*simp del*: *pair-in-Id-conv*)
  apply (*subst Id-comp-Id*[*symmetric*])
  apply (*rule relcompI*[*rotated*])
  apply (*rule glist-member-id-impl*[*param-fo*])
  apply (*rule IdI*)
  apply *assumption*
  apply *parametricity*
  done

lemma *list-set-autoref-insert*[*autoref-rules*]:
  assumes *GEN-OP eq op*= (*R*→*R*→*Id*)
  shows (*glist-insert eq*,*Set.insert*)
    ∈ *R* → ⟨*R*⟩*list-set-rel* → ⟨*R*⟩*list-set-rel*

```
  using assms
  apply (intro fun-relI)
  unfolding list-set-rel-def
  apply (erule relcompE)
  apply (simp del: pair-in-Id-conv)
  apply (rule relcompI[rotated])
  apply (rule glist-insert-id-impl[param-fo])
  apply (rule IdI)
  apply assumption
  apply parametricity
  done

lemma list-set-autoref-delete[autoref-rules]:
  assumes GEN-OP eq op= (R→R→Id)
  shows (glist-delete eq,op-set-delete)
    ∈ R → ⟨R⟩list-set-rel → ⟨R⟩list-set-rel
  using assms
  apply (intro fun-relI)
  unfolding list-set-rel-def
  apply (erule relcompE)
  apply (simp del: pair-in-Id-conv)
  apply (rule relcompI[rotated])
  apply (rule glist-delete-id-impl[param-fo])
  apply (rule IdI)
  apply assumption
  apply parametricity
  done

lemma list-set-autoref-to-list[autoref-ga-rules]:
  shows is-set-to-list R list-set-rel id
  unfolding is-set-to-list-def is-set-to-sorted-list-def
    it-to-sorted-list-def list-set-rel-def br-def
  by auto

lemma list-set-it-simp[iterator-simps]:
  foldli (id l) = foldli l by simp

lemma glist-insert-dj-id-impl:
  ⟦ x∉s; (l,s)∈br set distinct ⟧ ⟹ (x#l,insert x s)∈br set distinct
  by (auto simp: br-def)

lemma list-set-autoref-insert-dj[autoref-rules]:
  assumes PRIO-TAG-OPTIMIZATION
  assumes SIDE-PRECOND-OPT (x'∉s')
  assumes (x,x')∈R
  assumes (s,s')∈⟨R⟩list-set-rel
  shows (x#s,
    (OP Set.insert ::: R → ⟨R⟩list-set-rel → ⟨R⟩list-set-rel) $ x' $ s')
    ∈ ⟨R⟩list-set-rel
```

using *assms*
unfolding *autoref-tag-defs*
unfolding *list-set-rel-def*
apply −
apply (*erule relcompE*)
apply (*simp del*: *pair-in-Id-conv*)
apply (*rule relcompI*[*rotated*])
apply (*rule glist-insert-dj-id-impl*)
apply *assumption*
apply *assumption*
apply *parametricity*
done

end

Chapter 4

Entry Points

Entry points to the Autoref-Bundle.

4.1 Default Setup

theory *Refine-Dflt*
imports
  Monadic/Autoref-Monadic
  Collections/Impl/Impl-List-Set
  Collections/Impl/Impl-List-Map
  Collections/Impl/Impl-RBT-Map
  Collections/Impl/Impl-Array-Map
  Collections/Impl/Impl-Array-Hash-Map
  Collections/Gen/Gen-Set
  Collections/Gen/Gen-Map
  Collections/Gen/Gen-Map2Set
  Collections/Gen/Gen-Comp
begin

Useful Abbreviations

abbreviation *dflt-rs-rel* ≡ *map2set-rel dflt-rm-rel*
abbreviation *iam-set-rel* ≡ *map2set-rel iam-map-rel*
abbreviation *dflt-ahs-rel* ≡ *map2set-rel dflt-ahm-rel*

Some standard configurations

lemmas [*autoref-tyrel*] =
  ty-REL[**where** *'a=nat set* **and** *R=⟨Id⟩dflt-rs-rel*]
  ty-REL[**where** *'a=bool set* **and** *R=⟨Id⟩list-set-rel*]
  ty-REL[**where** *R=⟨nat-rel,Rv⟩dflt-rm-rel*, *standard*]

declaration ⟪ *let open Autoref-Fix-Rel in fn phi =>*
  I
  #> declare-prio Gen−AHM−map−hashable

```
  @{cpat ⟨?Rk::(-×-::hashable) set,?Rv⟩ahm-rel ?bhc} PR-LAST phi
 #> declare-prio Gen−RBT−map−linorder
  @{cpat ⟨?Rk::(-×-::linorder) set,?Rv⟩rbt-map-rel ?lt} PR-LAST phi
 #> declare-prio Gen−AHM−map @{cpat ⟨?Rk,?Rv⟩ahm-rel ?bhc} PR-LAST
phi
 #> declare-prio Gen−RBT−map @{cpat ⟨?Rk,?Rv⟩rbt-map-rel ?lt} PR-LAST
phi
end ⟫
```

ML-val ⟪
 let open Autoref-Debug in
  print-thm-pairs-matching @{context} @{cpat op-map-lookup}
 end
⟫

end

4.2 Entry Point with genCF and original ICF

theory *Refine-Dflt-ICF*
imports
 Monadic/Autoref-Monadic
 Collections/ICF/Autoref-Binding-ICF
 Collections/Impl/Impl-List-Set
 Collections/Impl/Impl-List-Map
 Collections/Impl/Impl-RBT-Map
 Collections/Impl/Impl-Array-Map
 Collections/Impl/Impl-Array-Hash-Map
 Collections/Gen/Gen-Set
 Collections/Gen/Gen-Map
 Collections/Gen/Gen-Map2Set
 Collections/Gen/Gen-Comp
begin

Contains the Monadic Refinement Framework, the generic collection framework and the original Isabelle Collection Framework

Useful Abbreviations

abbreviation *dflt-rs-rel* ≡ *map2set-rel dflt-rm-rel*
abbreviation *iam-set-rel* ≡ *map2set-rel iam-map-rel*
abbreviation *dflt-ahs-rel* ≡ *map2set-rel dflt-ahm-rel*

declaration ⟪ *let open Autoref-Fix-Rel in fn phi =>*
```
 I
 #> declare-prio Gen−RBT−set @{cpat ⟨?R⟩dflt-rs-rel} PR-LAST phi
 #> declare-prio RBT−set @{cpat ⟨?R⟩rs.rel} PR-LAST phi
```

```
  #> declare-prio Hash−set @{cpat ⟨?R⟩hs.rel} PR-LAST phi
  #> declare-prio List−set @{cpat ⟨?R⟩lsi.rel} PR-LAST phi
end ⟩⟩
```

declaration ⟨⟨ *let open Autoref-Fix-Rel in fn phi =>*

```
  I
  #> declare-prio Gen−RBT−map @{cpat ⟨?R⟩dflt-rm-rel} PR-LAST phi
  #> declare-prio RBT−map @{cpat ⟨?Rk,?Rv⟩rm.rel} PR-LAST phi
  #> declare-prio Hash−map @{cpat ⟨?Rk,?Rv⟩hm.rel} PR-LAST phi
  #> declare-prio List−map @{cpat ⟨?Rk,?Rv⟩lmi.rel} PR-LAST phi
end ⟩⟩
```

lemmas [*autoref-tyrel*] =
ty-REL[**where** *'a=nat* **and** *R=nat-rel*]
ty-REL[**where** *'a=int* **and** *R=int-rel*]
ty-REL[**where** *'a=bool* **and** *R=bool-rel*]
ty-REL[**where** *'a=nat set* **and** *R=⟨Id⟩rs.rel*]
ty-REL[**where** *'a=int set* **and** *R=⟨Id⟩rs.rel*]
ty-REL[**where** *'a=bool set* **and** *R=⟨Id⟩lsi.rel*]

end

4.3 Entry Point with only the ICF

theory *Refine-Dflt-Only-ICF*
imports
Monadic/Autoref-Monadic
Collections/ICF/Autoref-Binding-ICF
begin

Contains the Monadic Refinement Framework and the original Isabelle Collection Framework. The generic collection framework is not contained

declaration ⟨⟨ *let open Autoref-Fix-Rel in fn phi =>*

```
  I
  #> declare-prio RBT−set @{cpat ⟨?R⟩rs.rel} PR-LAST phi
  #> declare-prio Hash−set @{cpat ⟨?R⟩hs.rel} PR-LAST phi
  #> declare-prio List−set @{cpat ⟨?R⟩lsi.rel} PR-LAST phi
end ⟩⟩
```

declaration ⟨⟨ *let open Autoref-Fix-Rel in fn phi =>*

```
  I
  #> declare-prio RBT−map @{cpat ⟨?Rk,?Rv⟩rm.rel} PR-LAST phi
  #> declare-prio Hash−map @{cpat ⟨?Rk,?Rv⟩hm.rel} PR-LAST phi
  #> declare-prio List−map @{cpat ⟨?Rk,?Rv⟩lmi.rel} PR-LAST phi
end ⟩⟩
```

end

Chapter 5

Case Studies

5.1 Nested DFS (HPY improvement)

theory *Nested-DFS*
imports
../Refine-Dflt
Succ-Graph
begin

Implementation of a nested DFS algorithm for accepting cycle detection using the refinement framework. The algorithm uses the improvement of [HPY96], i.e., it reports a cycle if the inner DFS finds a path back to the stack of the outer DFS.

The algorithm returns a witness in case that an accepting cycle is detected.

5.1.1 Tools for DFS Algorithms

Invariants

definition *gen-dfs-pre E U S V u0* ≡
$E``U \subseteq U$ (∗ *Upper bound is closed under transitions* ∗)
$\wedge$ *finite U* (∗ *Upper bound is finite* ∗)
$\wedge\ V \subseteq U$ (∗ *Visited set below upper bound* ∗)
$\wedge\ u0 \in U$ (∗ *Start node in upper bound* ∗)
$\wedge\ E``(V-S) \subseteq V$ (∗ *Visited nodes closed under reachability, or on stack* ∗)
$\wedge\ u0 \notin V$ (∗ *Start node not yet visited* ∗)
$\wedge\ S \subseteq V$ (∗ *Stack is visited* ∗)
$\wedge\ (\forall v \in S.\ (v,u0) \in (E \cap S \times UNIV)^*)$ (∗ *u0 reachable from stack, only over stack* ∗)

definition *gen-dfs-var U* ≡ *finite-psupset U*

definition *gen-dfs-fe-inv E U S V0 u0 it V brk* ≡
$(\neg brk \longrightarrow E``(V-S) \subseteq V)$ (∗ *Visited set closed under reachability* ∗)

∧ $E``\{u0\} - it \subseteq V$ (∗ *Successors of u0 visited* ∗)
∧ $V0 \subseteq V$ (∗ *Visited set increasing* ∗)
∧ $V \subseteq V0 \cup (E - UNIV \times S)^*$ `` $(E``\{u0\} - it - S)$ (∗ *All visited nodes reachable* ∗)

definition *gen-dfs-post E U S V0 u0 V brk* ≡
($\neg brk \longrightarrow E``(V-S) \subseteq V$) (∗ *Visited set closed under reachability* ∗)
∧ $u0 \in V$ (∗ *u0 visited* ∗)
∧ $V0 \subseteq V$ (∗ *Visited set increasing* ∗)
∧ $V \subseteq V0 \cup (E - UNIV \times S)^*$ `` $\{u0\}$ (∗ *All visited nodes reachable* ∗)

Invariant Preservation

lemma *gen-dfs-pre-initial*:
assumes *finite* $(E^*``\{v0\})$
assumes $v0 \in U$
shows *gen-dfs-pre E* $(E^*``\{v0\})$ {} {} *v0*
using *assms* **unfolding** *gen-dfs-pre-def*
apply *auto*
done

lemma *gen-dfs-pre-imp-wf*:
assumes *gen-dfs-pre E U S V u0*
shows *wf* (*gen-dfs-var U*)
using *assms* **unfolding** *gen-dfs-pre-def gen-dfs-var-def* **by** *auto*

lemma *gen-dfs-pre-imp-fin*:
assumes *gen-dfs-pre E U S V u0*
shows *finite* (E `` $\{u0\}$)
apply (*rule finite-subset*[**where** *B=U*])
using *assms* **unfolding** *gen-dfs-pre-def*
by *auto*

Inserted *u0* on stack and to visited set

lemma *gen-dfs-pre-imp-fe*:
assumes *gen-dfs-pre E U S V u0*
shows *gen-dfs-fe-inv E U* (*insert u0 S*) (*insert u0 V*) *u0*
($E``\{u0\}$) (*insert u0 V*) *False*
using *assms* **unfolding** *gen-dfs-pre-def gen-dfs-fe-inv-def*
apply *auto*
done

lemma *gen-dfs-fe-inv-pres-visited*:
assumes *gen-dfs-pre E U S V u0*
assumes *gen-dfs-fe-inv E U* (*insert u0 S*) (*insert u0 V*) *u0 it V′ False*
assumes $t \in it$ $it \subseteq E``\{u0\}$ $t \in V'$
shows *gen-dfs-fe-inv E U* (*insert u0 S*) (*insert u0 V*) *u0* $(it - \{t\})$ *V′ False*

using *assms* **unfolding** *gen-dfs-fe-inv-def*
apply *auto*
done

lemma *gen-dfs-upper-aux*:
assumes $(x,y) \in E'^*$
assumes $(u0,x) \in E$
assumes $u0 \in U$
assumes $E' \subseteq E$
assumes $E``U \subseteq U$
shows $y \in U$
using *assms*
by *induct auto*

lemma *gen-dfs-fe-inv-imp-var*:
assumes *gen-dfs-pre E U S V u0*
assumes *gen-dfs-fe-inv E U (insert u0 S) (insert u0 V) u0 it V′ False*
assumes $t \in it$ $it \subseteq E``\{u0\}$ $t \notin V'$
shows $(V',V) \in$ *gen-dfs-var U*
using *assms* **unfolding** *gen-dfs-fe-inv-def gen-dfs-pre-def gen-dfs-var-def*
apply (*clarsimp simp add*: *finite-psupset-def*)
apply (*blast dest*: *gen-dfs-upper-aux*)
done

lemma *gen-dfs-fe-inv-imp-pre*:
assumes *gen-dfs-pre E U S V u0*
assumes *gen-dfs-fe-inv E U (insert u0 S) (insert u0 V) u0 it V′ False*
assumes $t \in it$ $it \subseteq E``\{u0\}$ $t \notin V'$
shows *gen-dfs-pre E U (insert u0 S) V′ t*
using *assms* **unfolding** *gen-dfs-fe-inv-def gen-dfs-pre-def*
apply *clarsimp*
apply (*intro conjI*)
apply (*blast dest*: *gen-dfs-upper-aux*)
apply *blast*
apply *blast*
apply *blast*
apply *clarsimp*
apply (*rule rtrancl-into-rtrancl*[**where** *b=u0*])
apply (*auto intro*: *set-rev-mp*[*OF - rtrancl-mono*[**where** $r=E \cap S \times UNIV$]]) []
apply *blast*
done

lemma *gen-dfs-post-imp-fe-inv*:
assumes *gen-dfs-pre E U S V u0*
assumes *gen-dfs-fe-inv E U (insert u0 S) (insert u0 V) u0 it V′ False*
assumes $t \in it$ $it \subseteq E``\{u0\}$ $t \notin V'$
assumes *gen-dfs-post E U (insert u0 S) V′ t V″ cyc*
shows *gen-dfs-fe-inv E U (insert u0 S) (insert u0 V) u0* $(it - \{t\})$ *V″ cyc*

```
  using assms unfolding gen-dfs-fe-inv-def gen-dfs-post-def gen-dfs-pre-def
  apply clarsimp
  apply (intro conjI)
  apply blast
  apply blast
  apply blast
  apply (erule order-trans)
  apply simp
  apply (rule conjI)
    apply (erule order-trans[
      where y=insert u0 (V ∪ (E − UNIV × insert u0 S)*
        `` (E `` {u0} − it − insert u0 S))])
    apply blast

    apply (cases cyc)
      apply simp
      apply blast

      apply simp
      apply blast
  done

lemma gen-dfs-post-aux:
  assumes 1: (u0,x)∈E
  assumes 2: (x,y)∈(E − UNIV × insert u0 S)*
  assumes 3: S⊆V   x∉V
  shows (u0, y) ∈ (E − UNIV × S)*
proof −
  from 1 3 have (u0,x)∈(E − UNIV × S) by blast
  also have (x,y)∈(E − UNIV × S)*
    apply (rule-tac set-rev-mp[OF 2 rtrancl-mono])
    by auto
  finally show ?thesis .
qed

lemma gen-dfs-fe-imp-post-brk:
  assumes gen-dfs-pre E U S V u0
  assumes gen-dfs-fe-inv E U (insert u0 S) (insert u0 V) u0 it V' True
  assumes it⊆E``{u0}
  shows gen-dfs-post E U S V u0 V' True
  using assms unfolding gen-dfs-pre-def gen-dfs-fe-inv-def gen-dfs-post-def
  apply clarify
  apply (intro conjI)
  apply simp
  apply simp
  apply simp
  apply clarsimp
  apply (blast intro: gen-dfs-post-aux)
  done
```

lemma *gen-dfs-fe-inv-imp-post*:
assumes *gen-dfs-pre E U S V u0*
assumes *gen-dfs-fe-inv E U* (*insert u0 S*) (*insert u0 V*) *u0* {} *V′ cyc*
assumes *cyc*⟶*cyc′*
shows *gen-dfs-post E U S V u0 V′ cyc′*
using *assms* **unfolding** *gen-dfs-pre-def gen-dfs-fe-inv-def gen-dfs-post-def*
apply *clarsimp*
apply (*intro conjI*)
apply *blast*
apply (*auto intro*: *gen-dfs-post-aux*) []
done

lemma *gen-dfs-pre-imp-post-brk*:
assumes *gen-dfs-pre E U S V u0*
shows *gen-dfs-post E U S V u0* (*insert u0 V*) *True*
using *assms* **unfolding** *gen-dfs-pre-def gen-dfs-post-def*
apply *auto*
done

Consequences of Postcondition

lemma *gen-dfs-post-imp-reachable*:
assumes *gen-dfs-pre E U S V0 u0*
assumes *gen-dfs-post E U S V0 u0 V brk*
shows $V \subseteq V0 \cup E^{*}``\{u0\}$
using *assms* **unfolding** *gen-dfs-post-def gen-dfs-pre-def*
apply *clarsimp*
apply (*blast intro*: *set-rev-mp*[*OF - rtrancl-mono*])
done

lemma *gen-dfs-post-imp-complete*:
assumes *gen-dfs-pre E U* {} *V0 u0*
assumes *gen-dfs-post E U* {} *V0 u0 V False*
shows $V0 \cup E^{*}``\{u0\} \subseteq V$
using *assms* **unfolding** *gen-dfs-post-def gen-dfs-pre-def*
apply *clarsimp*
apply (*blast dest*: *Image-closed-trancl*)
done

lemma *gen-dfs-post-imp-eq*:
assumes *gen-dfs-pre E U* {} *V0 u0*
assumes *gen-dfs-post E U* {} *V0 u0 V False*
shows $V = V0 \cup E^{*}``\{u0\}$
using *gen-dfs-post-imp-reachable*[*OF assms*] *gen-dfs-post-imp-complete*[*OF assms*]
by *blast*

lemma *gen-dfs-post-imp-below-U*:

```
assumes gen-dfs-pre E U S V0 u0
assumes gen-dfs-post E U S V0 u0 V False
shows V ⊆ U
using assms unfolding gen-dfs-pre-def gen-dfs-post-def
apply clarsimp
apply (blast intro: set-rev-mp[OF - rtrancl-mono] dest: Image-closed-trancl)
done
```

5.1.2 Abstract Algorithm

Inner (red) DFS

A witness of the red algorithm is a node on the stack and a path to this node

```
type-synonym 'v red-witness = ('v list × 'v) option
```

Prepend node to red witness

```
fun prep-wit-red :: 'v ⇒ 'v red-witness ⇒ 'v red-witness where
  prep-wit-red v None = None
| prep-wit-red v (Some (p,u)) = Some (v#p,u)
```

Initial witness for node u with onstack successor v

```
definition red-init-witness :: 'v ⇒ 'v ⇒ 'v red-witness where
  red-init-witness u v = Some ([u],v)
```

```
definition red-dfs where
  red-dfs E onstack V u ≡
    REC_T (λD (V,u). do {
      let V=insert u V;

      (* Check whether we have a successor on stack *)
      brk ← FOREACH_C (E``{u}) (λbrk. brk=None)
       (λt -. if t∈onstack then RETURN (red-init-witness u t) else RETURN None)
        None;

      (* Recurse for successors *)
      case brk of
        None ⇒
          FOREACH_C (E``{u}) (λ(V,brk). brk=None)
            (λt (V,-).
              if t∉V then do {
                (V,brk) ← D (V,t);
                RETURN (V,prep-wit-red u brk)
              } else RETURN (V,None))
            (V,None)
      | - ⇒ RETURN (V,brk)
    }) (V,u)
```

A witness of the blue DFS may be in two different phases, the *REACH* phase is before the node on the stack has actually been popped, and the *CIRC* phase is after the node on the stack has been popped.
REACH v p u p′:

v accepting node

p path from *v* to *u*

u node on stack

p′ path from current node to *v*

CIRC v pc pr:

v accepting node

pc path from *v* to *v*

pr path from current node to *v*

```
datatype 'v blue-witness =
  NO-CYC
| REACH 'v   'v list   'v   'v list
| CIRC 'v   'v list   'v list
```

Prepend node to witness

```
primrec prep-wit-blue :: 'v ⇒ 'v blue-witness ⇒ 'v blue-witness where
  prep-wit-blue u0 NO-CYC = NO-CYC
| prep-wit-blue u0 (REACH v p u p') = (
  if u0=u then
    CIRC v (p@u#p') (u0#p')
  else
    REACH v p u (u0#p')
  )
| prep-wit-blue u0 (CIRC v pc pr) = CIRC v pc (u0#pr)
```

Initialize blue witness

```
fun init-wit-blue :: 'v ⇒ 'v red-witness ⇒ 'v blue-witness where
  init-wit-blue u0 None = NO-CYC
| init-wit-blue u0 (Some (p,u)) = (
  if u=u0 then
    CIRC u0 p []
  else REACH u0 p u [])
```

Extract result from witness

```
definition extract-res cyc
  ≡ (case cyc of CIRC v pc pr ⇒ Some (v,pc,pr) | - ⇒ None)
```

Outer (Blue) DFS

```
definition blue-dfs
  :: ('a×'a) set ⇒ 'a set ⇒ 'a ⇒ ('a×'a list×'a list) option nres
  where
  blue-dfs E A s ≡ do {
    (-,-,-,cyc) ← REC_T (λD (blues,reds,onstack,s). do {
      let blues=insert s blues;
      let onstack=insert s onstack;
      (blues,reds,onstack,cyc) ←
      FOREACH_C (E``{s}) (λ(-,-,-,cyc). cyc=NO-CYC)
        (λt (blues,reds,onstack,cyc).
          if t∉blues then do {
            (blues,reds,onstack,cyc) ← D (blues,reds,onstack,t);
            RETURN (blues,reds,onstack,(prep-wit-blue s cyc))
          } else RETURN (blues,reds,onstack,cyc))
        (blues,reds,onstack,NO-CYC);

      (reds,cyc) ←
      if cyc=NO-CYC ∧ s∈A then do {
        (reds,rcyc) ← red-dfs E onstack reds s;
        RETURN (reds, init-wit-blue s rcyc)
      } else RETURN (reds,cyc);

      let onstack=onstack − {s};
      RETURN (blues,reds,onstack,cyc)
    }) ({},{},{},s);
    RETURN (extract-res cyc)
  }
```

5.1.3 Correctness

Specification of a reachable accepting cycle:

definition *has-acc-cycle E A v0* ≡ $\exists\, v{\in}A.\ (v0,v){\in}E^* \wedge (v,v){\in}E^+$

Paths

```
inductive path :: ('v×'v) set ⇒ 'v ⇒ 'v list ⇒ 'v ⇒ bool for E where
  path0: path E u [] u
| path-prepend: ⟦ (u,v)∈E; path E v l w ⟧ ⟹ path E u (u#l) w

lemma path1: (u,v)∈E ⟹ path E u [u] v
  by (auto intro: path.intros)

lemma path-simps[simp]:
  path E u [] v ⟷ u=v
  by (auto intro: path0 elim: path.cases)
```

lemma *path-conc*:
  assumes *P1*: *path E u la v*
  assumes *P2*: *path E v lb w*
  shows *path E u* (*la*@*lb*) *w*
  using *P1 P2* **apply** *induct*
  by (*auto intro*: *path.intros*)

lemma *path-append*:
  ⟦ *path E u l v*; (*v*,*w*)∈*E* ⟧ ⟹ *path E u* (*l*@[*v*]) *w*
  using *path-conc*[*OF* - *path1*] **.**

lemma *path-unconc*:
  assumes *path E u* (*la*@*lb*) *w*
  obtains *v* **where** *path E u la v* **and** *path E v lb w*
  using *assms*
  thm *path.induct*
  apply (*induct u la*@*lb w arbitrary*: *la lb rule*: *path.induct*)
  apply (*auto intro*: *path.intros elim*!: *list-Cons-eq-append-cases*)
  done

lemma *path-uncons*:
  assumes *path E u* (*u′*#*l*) *w*
  obtains *v* **where** *u′*=*u* **and** (*u*,*v*)∈*E* **and** *path E v l w*
  apply (*rule path-unconc*[*of E u* [*u′*] *l w*, *simplified*, *OF assms*])
  apply (*auto elim*: *path.cases*)
  done

lemma *path-is-rtrancl*:
  assumes *path E u l v*
  shows (*u*,*v*)∈E^*
  using *assms*
  by *induct auto*

lemma *rtrancl-is-path*:
  assumes (*u*,*v*)∈E^*
  obtains *l* **where** *path E u l v*
  using *assms*
  by *induct* (*auto intro*: *path0 path-append*)

lemma *path-is-trancl*:
  assumes *path E u l v*
  and *l*≠[]
  shows (*u*,*v*)∈E^+
  using *assms*
  apply *induct*
  apply *auto* []
  apply (*case-tac l*)
  apply *auto*
  done

lemma *trancl-is-path*:
  assumes $(u,v) \in E^+$
  obtains *l* **where** $l \neq []$ **and** *path E u l v*
  using *assms*
  by *induct* (*auto intro*: *path0 path-append*)

Specification of witness for accepting cycle

definition *is-acc-cycle E A v0 v r c*
  $\equiv v \in A \wedge$ *path E v0 r v* $\wedge$ *path E v c v* $\wedge c \neq []$

Specification is compatible with existence of accepting cycle

lemma *is-acc-cycle-eq*:
  has-acc-cycle E A v0 $\longleftrightarrow$ ($\exists$ *v r c*. *is-acc-cycle E A v0 v r c*)
  unfolding *has-acc-cycle-def is-acc-cycle-def*
  by (*auto elim*!: *rtrancl-is-path trancl-is-path*
    intro: *path-is-rtrancl path-is-trancl*)

Additional invariant to be maintained between calls of red dfs

definition *red-dfs-inv E U reds onstack* $\equiv$
  $E``U \subseteq U$ (* *Upper bound is closed under transitions* *)
  $\wedge$ *finite U* (* *Upper bound is finite* *)
  $\wedge$ *reds* $\subseteq U$ (* *Red set below upper bound* *)
  $\wedge$ $E``$*reds* $\subseteq$ *reds* (* *Red nodes closed under reachability* *)
  $\wedge$ $E``$*reds* $\cap$ *onstack* $= \{\}$ (* *No red node with edge to stack* *)

lemma *red-dfs-inv-initial*:
  assumes *finite* $(E^* `` \{v0\})$
  shows *red-dfs-inv E* $(E^* `` \{v0\})$ $\{\}$ $\{\}$
  using *assms* **unfolding** *red-dfs-inv-def*
  apply *auto*
  done

Correctness of the red DFS.

theorem *red-dfs-correct*:
  fixes *v0 u0* :: *'v*
  assumes *PRE*:
    red-dfs-inv E U reds onstack
    $u0 \in U$
    $u0 \notin$ *reds*
  shows *red-dfs E onstack reds u0*
    $\leq$ *SPEC* (λ(*reds'*,*cyc*). *case cyc of*
      Some $(p,v) \Rightarrow v \in$ *onstack* $\wedge$ $p \neq []$ $\wedge$ *path E u0 p v*
    | *None* $\Rightarrow$
        red-dfs-inv E U reds' onstack
        $\wedge$ $u0 \in$ *reds'*
        $\wedge$ *reds'* $\subseteq$ *reds* $\cup$ E^* `` $\{u0\}$

```
  )
proof -
  let ?dfs-red =
    REC_T (λD (V,u). do {
      let V=insert u V;

      (* Check whether we have a successor on stack *)
      brk ← FOREACH_C (E``{u}) (λbrk. brk=None)
        (λt -. if t∈onstack then
            RETURN (red-init-witness u t)
          else RETURN None)
        None;

      (* Recurse for successors *)
      case brk of
        None ⇒
          FOREACH_C (E``{u}) (λ(V,brk). brk=None)
            (λt (V,-).
              if t∉V then do {
                (V,brk) ← D (V,t);
                RETURN (V,prep-wit-red u brk)
              } else RETURN (V,None))
            (V,None)
      | - ⇒ RETURN (V,brk)
    }) (V,u)

  let REC_T ?body ?init =    ?dfs-red

  def pre ≡ λS (V,u0). gen-dfs-pre E U S V u0 ∧ E``V ∩ onstack = {}
  def post ≡ λS (V0,u0) (V,cyc). gen-dfs-post E U S V0 u0 V (cyc≠None)
    ∧ (case cyc of None ⇒ E``V ∩ onstack = {}
      | Some (p,v) ⇒ v∈onstack ∧ p≠[] ∧ path E u0 p v)


  def fe-inv ≡ λS V0 u0 it (V,cyc).
    gen-dfs-fe-inv E U S V0 u0 it V (cyc≠None)
    ∧ (case cyc of None ⇒ E``V ∩ onstack = {}
      | Some (p,v) ⇒ v∈onstack ∧ p≠[] ∧ path E u0 p v)



  from PRE have GENPRE: gen-dfs-pre E U {} reds u0
    unfolding red-dfs-inv-def gen-dfs-pre-def
    by auto
  with PRE have PRE': pre {} (reds,u0)
    unfolding pre-def red-dfs-inv-def
    by auto

  have IMP-POST: SPEC (post {} (reds,u0))
```

```
  ≤ SPEC (λ(reds′,cyc). case cyc of
    Some (p,v) ⇒ v∈onstack ∧ p≠[] ∧ path E u0 p v
  | None ⇒
      red-dfs-inv E U reds′ onstack
      ∧ u0∈reds′
      ∧ reds′ ⊆ reds ∪ E* `` {u0})
  apply (clarsimp split: option.split)
  apply (intro impI conjI allI)
  apply simp-all
proof −
  fix reds′ p v
  assume post {} (reds,u0) (reds′,Some (p,v))
  thus v∈onstack and p≠[] and path E u0 p v
    unfolding post-def by auto
next
  fix reds′
  assume post {} (reds, u0) (reds′, None)
  hence GPOST: gen-dfs-post E U {} reds u0 reds′ False
    and NS: E``reds′ ∩ onstack = {}
    unfolding post-def by auto

  from GPOST show u0∈reds′ unfolding gen-dfs-post-def by auto

  show red-dfs-inv E U reds′ onstack
    unfolding red-dfs-inv-def
    apply (intro conjI)
    using GENPRE[unfolded gen-dfs-pre-def]
    apply (simp-all) [2]
    apply (rule gen-dfs-post-imp-below-U[OF GENPRE GPOST])
    using GPOST[unfolded gen-dfs-post-def] apply simp
    apply fact
    done

  from GPOST show reds′ ⊆ reds ∪ E* `` {u0}
    unfolding gen-dfs-post-def by auto
qed

{
  fix σ S
  assume INV0: pre S σ
  have REC_T ?body σ
    ≤ SPEC (post S σ)

    apply (rule RECT-rule-arb[where
      Φ=pre and
      V=gen-dfs-var U <*lex*> {} and
      arb=S
      ])
```

```
apply refine-mono

using INV0[unfolded pre-def] apply (auto intro: gen-dfs-pre-imp-wf) []

apply fact

apply (rename-tac D S u)
apply (intro refine-vcg)


apply (rule-tac I=λit cyc.
  (case cyc of None ⇒ (E``{b} − it) ∩ onstack = {}
    | Some (p,v) ⇒ (v∈onstack ∧ p≠[] ∧ path E b p v))
  in FOREACHc-rule)
apply (auto simp add: pre-def gen-dfs-pre-imp-fin) []
apply auto []
apply (auto
  split: option.split
  simp: red-init-witness-def intro: path1) []

apply (intro refine-vcg)


apply (rule-tac I=fe-inv (insert b S) (insert b a) b in
  FOREACHc-rule
)
apply (auto simp add: pre-def gen-dfs-pre-imp-fin) []

apply (auto simp add: pre-def fe-inv-def gen-dfs-pre-imp-fe) []

apply (intro refine-vcg)


apply (rule order-trans)
apply (rprems)
apply (clarsimp simp add: pre-def fe-inv-def)
apply (rule gen-dfs-fe-inv-imp-pre, assumption+) []
apply (auto simp add: pre-def fe-inv-def intro: gen-dfs-fe-inv-imp-var) []

apply (clarsimp simp add: pre-def post-def fe-inv-def
  split: option.split-asm prod.split-asm
  ) []
  apply (blast intro: gen-dfs-post-imp-fe-inv)
  apply (blast intro: gen-dfs-post-imp-fe-inv path-prepend)

apply (auto simp add: pre-def post-def fe-inv-def
  intro: gen-dfs-fe-inv-pres-visited) []

apply (auto simp add: pre-def post-def fe-inv-def
```

```
      intro: gen-dfs-fe-inv-imp-post) []

    apply (auto simp add: pre-def post-def fe-inv-def
      intro: gen-dfs-fe-imp-post-brk) []

    apply (auto simp add: pre-def post-def fe-inv-def
      intro: gen-dfs-pre-imp-post-brk) []

    apply (auto simp add: pre-def post-def fe-inv-def
      intro: gen-dfs-pre-imp-post-brk) []

    done
  } note GEN=this

  note GEN[OF PRE']
  also note IMP-POST
  finally show ?thesis
    unfolding red-dfs-def .
qed
```

Main theorem: Correctness of the blue DFS

```
theorem blue-dfs-correct:
  fixes v0 :: 'v
  assumes FIN[simp,intro!]: finite (E* `` {v0})
  shows blue-dfs E A v0 ≤ SPEC (λr.
    case r of None ⇒ ¬has-acc-cycle E A v0
  | Some (v,pc,pv) ⇒ is-acc-cycle E A v0 v pv pc)
proof -
  let ?ndfs =
  do {
    (-,-,-,cyc) ← REC_T (λD (blues,reds,onstack,s). do {
      let blues=insert s blues;
      let onstack=insert s onstack;
      (blues,reds,onstack,cyc) ←
      FOREACH_C (E``{s}) (λ(-,-,-,cyc). cyc=NO-CYC)
        (λt (blues,reds,onstack,cyc).
          if t∉blues then do {
            (blues,reds,onstack,cyc) ← D (blues,reds,onstack,t);
            RETURN (blues,reds,onstack,(prep-wit-blue s cyc))
          } else RETURN (blues,reds,onstack,cyc))
        (blues,reds,onstack,NO-CYC);

      (reds,cyc) ←
      if cyc=NO-CYC ∧ s∈A then do {
        (reds,rcyc) ← red-dfs E onstack reds s;
        RETURN (reds, init-wit-blue s rcyc)
      } else RETURN (reds,cyc);

      let onstack=onstack - {s};
```

```
    RETURN (blues,reds,onstack,cyc)
  }) ({},{},{},s);
  RETURN (case cyc of NO-CYC ⇒ None | CIRC v pc pr ⇒ Some (v,pc,pr))
}
let do {- ← REC_T ?body ?init; -} =     ?ndfs

let ?U = E* ``{v0}

def add-inv ≡ λblues reds onstack.
  ¬(∃ v∈(blues−onstack)∩A. (v,v)∈E⁺)  (* No cycles over finished,
                                         accepting states *)
  ∧ reds ⊆ blues                        (* Red nodes are also blue *)
  ∧ reds ∩ onstack = {}                    (* No red nodes on stack *)
  ∧ red-dfs-inv E ?U reds onstack

def cyc-post ≡ λblues reds onstack u0 cyc. (case cyc of
    NO-CYC ⇒ add-inv blues reds onstack
  | REACH v p u p' ⇒ v∈A ∧ u∈onstack−{u0} ∧ p≠[]
    ∧ path E v p u ∧ path E u0 p' v
  | CIRC v pc pr ⇒ v∈A ∧ pc≠[] ∧ path E v pc v ∧ path E u0 pr v
  )

def pre ≡ λ(blues,reds,onstack,u).
  gen-dfs-pre E ?U onstack blues u ∧ add-inv blues reds onstack

def post ≡ λ(blues0,reds0::'v set,onstack0,u0) (blues,reds,onstack,cyc).
  onstack = onstack0
  ∧ gen-dfs-post E ?U onstack0 blues0 u0 blues (cyc≠NO-CYC)
  ∧ cyc-post blues reds onstack u0 cyc

def fe-inv ≡ λblues0 u0 onstack0 it (blues,reds,onstack,cyc).
  onstack=onstack0
  ∧ gen-dfs-fe-inv E ?U onstack0 blues0 u0 it blues (cyc≠NO-CYC)
  ∧ cyc-post blues reds onstack u0 cyc

have GENPRE: gen-dfs-pre E ?U {} {} v0
  apply (auto intro: gen-dfs-pre-initial)
  done
hence PRE': pre ({},{},{},v0)
  unfolding pre-def add-inv-def
  apply (auto intro: red-dfs-inv-initial)
  done

{
  fix blues reds onstack cyc
  assume post ({},{},{},v0) (blues,reds,onstack,cyc)
  hence case cyc of NO-CYC ⇒ ¬has-acc-cycle E A v0
    | REACH - - - - ⇒ False
    | CIRC v pc pr ⇒ is-acc-cycle E A v0 v pr pc
```

```
    unfolding post-def cyc-post-def
    apply (cases cyc)
    using gen-dfs-post-imp-eq[OF GENPRE, of blues]
    apply (auto simp: add-inv-def has-acc-cycle-def) []
    apply auto []
    apply (auto simp: is-acc-cycle-def) []
    done
} note IMP-POST = this

{
  fix onstack blues u0 reds
  assume pre (blues,reds,onstack,u0)
  hence fe-inv (insert u0 blues) u0 (insert u0 onstack) (E``{u0})
    (insert u0 blues,reds,insert u0 onstack,NO-CYC)
    unfolding fe-inv-def add-inv-def cyc-post-def
    apply clarsimp
    apply (intro conjI)
    apply (simp add: pre-def gen-dfs-pre-imp-fe)
    apply (auto simp: pre-def add-inv-def) []
    apply (auto simp: pre-def add-inv-def) []
    defer
    apply (auto simp: pre-def add-inv-def) []
    apply (unfold pre-def add-inv-def red-dfs-inv-def gen-dfs-pre-def) []
    apply clarsimp
    apply blast

    apply (auto simp: pre-def add-inv-def gen-dfs-pre-def) []
    done
} note PRE-IMP-FE = this

have [simp]: ⋀u cyc. prep-wit-blue u cyc = NO-CYC ⟷ cyc=NO-CYC
  by (case-tac cyc) auto

{
  fix blues0 reds0 onstack0 u0
    blues reds onstack blues' reds' onstack'
    cyc it t
  assume PRE: pre (blues0,reds0,onstack0,u0)
  assume FEI: fe-inv (insert u0 blues0) u0 (insert u0 onstack0)
      it (blues,reds,onstack,NO-CYC)
  assume IT: t∈it    it⊆E``{u0}    t∉blues
  assume POST: post (blues,reds,onstack, t) (blues',reds',onstack',cyc)
  note [simp del] = path-simps
  have fe-inv (insert u0 blues0) u0 (insert u0 onstack0) (it−{t})
    (blues',reds',onstack',prep-wit-blue u0 cyc)
    unfolding fe-inv-def
    using PRE FEI IT POST
    unfolding fe-inv-def post-def pre-def
    apply (clarsimp)
```

```
    apply (intro allI impI conjI)
    apply (blast intro: gen-dfs-post-imp-fe-inv)
    unfolding cyc-post-def
    apply (auto split: blue-witness.split-asm intro: path-conc path-prepend)
    done
} note FE-INV-PRES=this

{
  fix blues reds onstack u0
  assume pre (blues,reds,onstack,u0)
  hence (v0,u0)∈E*
    unfolding pre-def gen-dfs-pre-def by auto
} note PRE-IMP-REACH = this

{
  fix blues0 reds0 onstack0 u0 blues reds onstack
  assume A: pre (blues0,reds0,onstack0,u0)
    fe-inv (insert u0 blues0) u0 (insert u0 onstack0)
     {} (blues,reds,onstack,NO-CYC)
    u0∈A
  have u0∉reds using A
    unfolding fe-inv-def add-inv-def pre-def cyc-post-def
    apply auto
    done
} note FE-IMP-RED-PRE = this

{
  fix blues0 reds0 onstack0 u0 blues reds onstack rcyc reds'
  assume PRE: pre (blues0,reds0,onstack0,u0)
  assume FEI: fe-inv (insert u0 blues0) u0 (insert u0 onstack0)
     {} (blues,reds,onstack,NO-CYC)
  assume ACC: u0∈A
  assume SPECR: case rcyc of
    Some (p,v) ⇒ v∈onstack ∧ p≠[] ∧ path E u0 p v
  | None ⇒
      red-dfs-inv E ?U reds' onstack
      ∧ u0∈reds'
      ∧ reds' ⊆ reds ∪ E* `` {u0}
  have post (blues0,reds0,onstack0,u0)
    (blues,reds',onstack − {u0},init-wit-blue u0 rcyc)
    unfolding post-def add-inv-def cyc-post-def
    apply (clarsimp)
    apply (intro conjI)
  proof −
    from PRE FEI show OS0[symmetric]: onstack − {u0} = onstack0
      by (auto simp: pre-def fe-inv-def add-inv-def gen-dfs-pre-def) []

    from PRE FEI have u0∈onstack
      unfolding pre-def gen-dfs-pre-def fe-inv-def gen-dfs-fe-inv-def
```

```
  by auto

from PRE FEI
show POST: gen-dfs-post E (E* `` {v0}) onstack0 blues0 u0 blues
  (init-wit-blue u0 rcyc ≠ NO-CYC)
  by (auto simp: pre-def fe-inv-def intro: gen-dfs-fe-inv-imp-post)

from FEI have [simp]: onstack=insert u0 onstack0
  unfolding fe-inv-def by auto
from FEI have u0∈blues unfolding fe-inv-def gen-dfs-fe-inv-def by auto

case goal3 show ?case
  apply (cases rcyc)
  apply (simp-all add: split-paired-all)
proof -
  assume [simp]: rcyc=None
  show (∀ v∈(blues − (onstack0 − {u0})) ∩ A. (v, v) ∉ E⁺) ∧
    reds' ⊆ blues ∧
    reds' ∩ (onstack0 − {u0}) = {} ∧
    red-dfs-inv E (E* `` {v0}) reds' (onstack0 − {u0})
  proof (intro conjI)
    from SPECR have RINV: red-dfs-inv E ?U reds' onstack
      and u0∈reds'
      and REDS'R: reds' ⊆ reds ∪ E*``{u0}
      by auto

    from RINV show
      RINV': red-dfs-inv E (E* `` {v0}) reds' (onstack0 − {u0})
      unfolding red-dfs-inv-def by auto

    from RINV'[unfolded red-dfs-inv-def] have
      REDS'CL: E``reds' ⊆ reds'
      and DJ': E `` reds' ∩ (onstack0 − {u0}) = {} by auto

    from RINV[unfolded red-dfs-inv-def] have
      DJ: E `` reds' ∩ (onstack) = {} by auto

    show reds' ⊆ blues
    proof
      fix v assume v∈reds'
      with REDS'R have v∈reds ∨ (u0,v)∈E* by blast
      thus v∈blues proof
        assume v∈reds
        moreover with FEI have reds⊆blues
          unfolding fe-inv-def add-inv-def cyc-post-def by auto
        ultimately show ?thesis ..
      next
        from POST[unfolded gen-dfs-post-def OS0] have
          CL: E `` (blues − (onstack0 − {u0})) ⊆ blues and u0∈blues
```

```
      by auto
    from PRE FEI have onstack0 ⊆ blues
      unfolding pre-def fe-inv-def gen-dfs-pre-def gen-dfs-fe-inv-def
      by auto

    assume (u0,v)∈E^*
    thus v∈blues
    proof (cases rule: rtrancl-last-visit[where S=onstack − {u0}])
      case no-visit
      thus v∈blues using ‹u0∈blues› CL
        by induct (auto elim: rtranclE)
    next
      case (last-visit-point u)
      then obtain uh where (u0,uh)∈E^* and (uh,u)∈E
        by (metis tranclD2)
      with REDS'CL DJ' ‹u0∈reds'› have uh∈reds'
        by (auto dest: Image-closed-trancl)
      with DJ' ‹(uh,u)∈E› ‹u ∈ onstack − {u0}› have False
        by simp blast
      thus ?thesis ..
    qed
  qed
qed

show ∀ v∈(blues − (onstack0 − {u0})) ∩ A. (v, v) ∉ E^+
proof
  fix v
  assume A: v ∈ (blues − (onstack0 − {u0})) ∩ A
  show (v,v)∉E^+ proof (cases v=u0)
    assume v≠u0
    with A have v∈(blues − (insert u0 onstack)) ∩ A by auto
    with FEI show ?thesis
      unfolding fe-inv-def add-inv-def cyc-post-def by auto
  next
    assume [simp]: v=u0
    show ?thesis proof
      assume (v,v)∈E^+
      then obtain uh where (u0,uh)∈E^* and (uh,u0)∈E
        by (auto dest: tranclD2)
      with REDS'CL DJ ‹u0∈reds'› have uh∈reds'
        by (auto dest: Image-closed-trancl)
      with DJ ‹(uh,u0)∈E› ‹u0 ∈ onstack› show False by blast
    qed
  qed
qed

show reds' ∩ (onstack0 − {u0}) = {}
proof (rule ccontr)
  assume reds' ∩ (onstack0 − {u0}) ≠ {}
```

```
      then obtain v where v∈reds' and v∈onstack0 and v≠u0 by auto

      from ‹v∈reds'› REDS'R have v∈reds ∨ (u0,v)∈E*
        by auto
      thus False proof
        assume v∈reds
        with FEI[unfolded fe-inv-def add-inv-def cyc-post-def]
          ‹v∈onstack0›
        show False by auto
      next
        assume (u0,v)∈E*
        with ‹v≠u0› obtain uh where (u0,uh)∈E* and (uh,v)∈E
          by (auto elim: rtranclE)
        with REDS'CL DJ ‹u0∈reds'› have uh∈reds'
          by (auto dest: Image-closed-trancl)
        with DJ ‹(uh,v)∈E› ‹v ∈ onstack0› show False by simp blast
      qed
    qed
  qed
next
  fix u p
  assume [simp]: rcyc = Some (p,u)
  show (u = u0 ⟶ u0 ∈ A ∧ p ≠ [] ∧ path E u0 p u0) ∧
    (u ≠ u0 ⟶ u0 ∈ A ∧ u ∈ onstack0 ∧ p ≠ [] ∧ path E u0 p u)
  proof (intro conjI impI)
    show u0∈A by fact
    show u0∈A by fact
    from SPECR show
      u≠u0 ⟹ u∈onstack0
      p≠[]
      p≠[]
      path E u0 p u
      u=u0 ⟹ path E u0 p u0
      by auto
  qed
 qed
qed
} note RED-IMP-POST = this

{
 fix blues0 reds0 onstack0 u0 blues reds onstack and cyc :: 'v blue-witness
 assume PRE: pre (blues0,reds0,onstack0,u0)
 and FEI: fe-inv (insert u0 blues0) u0 (insert u0 onstack0)
    {} (blues,reds,onstack,NO-CYC)
 and FC[simp]: cyc=NO-CYC
 and NCOND: u0∉A

 from PRE FEI have OS0: onstack0 = onstack − {u0}
   by (auto simp: pre-def fe-inv-def add-inv-def gen-dfs-pre-def) []
```

```
    from PRE FEI have u0∈onstack
      unfolding pre-def gen-dfs-pre-def fe-inv-def gen-dfs-fe-inv-def
      by auto
    with OS0 have OS1: onstack = insert u0 onstack0 by auto

    have post (blues0,reds0,onstack0,u0) (blues,reds,onstack − {u0},NO-CYC)
      apply (clarsimp simp: post-def cyc-post-def) []
      apply (intro conjI impI)
      apply (simp add: OS0)
      using PRE FEI apply (auto
        simp: pre-def fe-inv-def intro: gen-dfs-fe-inv-imp-post) []

      using FEI[unfolded fe-inv-def cyc-post-def] unfolding add-inv-def
      apply clarsimp
      apply (intro conjI)
      using NCOND apply auto []
      apply auto []
      apply (clarsimp simp: red-dfs-inv-def, blast) []
      done
  } note NCOND-IMP-POST=this

  {
    fix blues0 reds0 onstack0 u0 blues reds onstack it
      and cyc :: 'v blue-witness
    assume PRE: pre (blues0,reds0,onstack0,u0)
    and FEI: fe-inv (insert u0 blues0) u0 (insert u0 onstack0)
        it (blues,reds,onstack,cyc)
    and NC: cyc≠NO-CYC
    and IT: it⊆E``{u0}
    from PRE FEI have OS0: onstack0 = onstack − {u0}
      by (auto simp: pre-def fe-inv-def add-inv-def gen-dfs-pre-def) []

    from PRE FEI have u0∈onstack
      unfolding pre-def gen-dfs-pre-def fe-inv-def gen-dfs-fe-inv-def
      by auto
    with OS0 have OS1: onstack = insert u0 onstack0 by auto

    have post (blues0,reds0,onstack0,u0) (blues,reds,onstack − {u0},cyc)
      apply (clarsimp simp: post-def) []
      apply (intro conjI impI)
      apply (simp add: OS0)
      using PRE FEI IT NC apply (auto
        simp: pre-def fe-inv-def intro: gen-dfs-fe-imp-post-brk) []
      using FEI[unfolded fe-inv-def] NC
      unfolding cyc-post-def
      apply (auto split: blue-witness.split simp: OS1) []
      done
  } note BREAK-IMP-POST = this
```

```
{
 fix σ
 assume INV0: pre σ
 have REC_T ?body σ
   ≤ SPEC (post σ)

   apply (intro refine-vcg
     RECT-rule[where Φ=pre
     and V=gen-dfs-var ?U <*lex*> {}]
   )
   apply refine-mono
   apply (blast intro!: gen-dfs-pre-imp-wf[OF GENPRE])
   apply (rule INV0)


   apply (rule-tac
     I=fe-inv (insert bb a) bb (insert bb ab)
     in FOREACHc-rule')

   apply (auto simp add: pre-def gen-dfs-pre-imp-fin) []

   apply (blast intro: PRE-IMP-FE)

   apply (intro refine-vcg)


   apply (rule order-trans)
   apply (rprems)
   apply (clarsimp simp add: pre-def fe-inv-def cyc-post-def)
   apply (rule gen-dfs-fe-inv-imp-pre, assumption+) []
   apply (auto simp add: pre-def fe-inv-def intro: gen-dfs-fe-inv-imp-var) []

   apply (auto intro: FE-INV-PRES) []

   apply (auto simp add: pre-def post-def fe-inv-def
     intro: gen-dfs-fe-inv-pres-visited) []

   apply (intro refine-vcg)


   apply (rule order-trans)
   apply (rule red-dfs-correct[where U=E* `` {v0}])
   apply (auto simp add: fe-inv-def add-inv-def cyc-post-def) []
   apply (auto intro: PRE-IMP-REACH) []
   apply (auto dest: FE-IMP-RED-PRE) []

   apply (intro refine-vcg)
   apply clarsimp
```

apply (*rule RED-IMP-POST*, *assumption+*) []

apply (*clarsimp*, *blast intro*: *NCOND-IMP-POST*) []

apply (*intro refine-vcg*)
apply *simp*

apply (*clarsimp*, *blast intro*: *BREAK-IMP-POST*) []
done
} **note** *GEN=this*

show *?thesis*
unfolding *blue-dfs-def extract-res-def*
apply (*intro refine-vcg*)
apply (*rule order-trans*)
apply (*rule GEN*)
apply *fact*
apply (*intro refine-vcg*)
apply *clarsimp*
apply (*drule IMP-POST*)
apply (*simp split*: *blue-witness.split-asm*)
done
qed

5.1.4 Refinement

Setup for Custom Datatypes

This effort can be automated, but currently, such an automation is not yet implemented

abbreviation *red-wit-rel* ≡ ⟨⟨⟨*nat-rel*⟩*list-rel*,*nat-rel*⟩*prod-rel*⟩*option-rel*
abbreviation *wit-res-rel* ≡
⟨⟨*nat-rel*,⟨⟨*nat-rel*⟩*list-rel*,⟨*nat-rel*⟩*list-rel*⟩*prod-rel*⟩*prod-rel*⟩*option-rel*
abbreviation *i-red-wit* ≡ $\langle\langle\langle$*i-nat*$\rangle_i$*i-list*,*i-nat*$\rangle_i$*i-prod*$\rangle_i$*i-option*
abbreviation *i-res* ≡
$\langle\langle$*i-nat*,$\langle\langle$*i-nat*$\rangle_i$*i-list*,$\langle$*i-nat*$\rangle_i$*i-list*$\rangle_i$*i-prod*$\rangle_i$*i-prod*$\rangle_i$*i-option*

abbreviation *blue-wit-rel* ≡ (*Id*::(*nat blue-witness* × -) *set*)
consts *i-blue-wit* :: *interface*

term *extract-res*

lemma [*autoref-itype*]:
NO-CYC $::_i$ *i-blue-wit*
op = $::_i$ *i-blue-wit* $\rightarrow_i$ *i-blue-wit* $\rightarrow_i$ *i-bool*
init-wit-blue $::_i$ *i-nat* $\rightarrow_i$ *i-red-wit* $\rightarrow_i$ *i-blue-wit*
prep-wit-blue $::_i$ *i-nat* $\rightarrow_i$ *i-blue-wit* $\rightarrow_i$ *i-blue-wit*
red-init-witness $::_i$ *i-nat* $\rightarrow_i$ *i-nat* $\rightarrow_i$ *i-red-wit*
prep-wit-red $::_i$ *i-nat* $\rightarrow_i$ *i-red-wit* $\rightarrow_i$ *i-red-wit*

extract-res $::_i$ *i-blue-wit* $\rightarrow_i$ *i-res*
by *auto*

lemma [*autoref-op-pat*]: *NO-CYC* $\equiv$ *OP NO-CYC* $:::_i$ *i-blue-wit* **by** *simp*

lemma [*autoref-rules-raw*]:
(*NO-CYC*,*NO-CYC*)∈*blue-wit-rel*
(*op* =, *op* =) ∈ *blue-wit-rel* → *blue-wit-rel* → *bool-rel*
(*init-wit-blue*, *init-wit-blue*) ∈ *nat-rel* → *red-wit-rel* → *blue-wit-rel*
(*prep-wit-blue*,*prep-wit-blue*)∈*nat-rel* → *blue-wit-rel* → *blue-wit-rel*
(*red-init-witness*, *red-init-witness*) ∈ *nat-rel*→*nat-rel*→*red-wit-rel*
(*prep-wit-red*,*prep-wit-red*) ∈ *nat-rel* → *red-wit-rel* → *red-wit-rel*
(*extract-res*,*extract-res*) ∈ *blue-wit-rel* → *wit-res-rel*
by *simp-all*

Actual Refinement

schematic-lemma *red-dfs-impl-refine-aux*:
notes [[*goals-limit* = *1*]]
fixes *u'*::*nat* **and** *V'*::*nat set*
assumes [*autoref-rules*]:
(*u*,*u'*)∈*nat-rel*
(*V*,*V'*)∈⟨*nat-rel*⟩*dflt-rs-rel*
(*onstack*,*onstack'*)∈⟨*nat-rel*⟩*dflt-rs-rel*
(*E*,*E'*)∈⟨*nat-rel*⟩*slg-rel*
shows (*RETURN* (*?f*::*?'c*), *red-dfs E' onstack' V' u'*) ∈ *?R*
apply −
unfolding *red-dfs-def*
apply (*autoref-monadic*)
done

concrete-definition *red-dfs-impl* **uses** *red-dfs-impl-refine-aux*
prepare-code-thms *red-dfs-impl-def*
declare *red-dfs-impl.refine*[*autoref-higher-order-rule*, *autoref-rules*]

schematic-lemma *ndfs-impl-refine-aux*:
fixes *s*::*nat*
assumes [*autoref-rules*]:
(*succi*,*E*)∈⟨*nat-rel*⟩*slg-rel*
(*Ai*,*A*)∈⟨*nat-rel*⟩*dflt-rs-rel*
notes [*autoref-rules*] = *IdI*[*of s*]
shows (*RETURN* (*?f*::*?'c*), *blue-dfs E A s*) ∈ ⟨*?R*⟩*nres-rel*
unfolding *blue-dfs-def*
apply (*autoref-monadic* (*trace*))
done

concrete-definition *ndfs-impl* **for** *succi Ai s* **uses** *ndfs-impl-refine-aux*
prepare-code-thms *ndfs-impl-def*
export-code *ndfs-impl* **in** *SML* **file** −

schematic-lemma *ndfs-impl-refine-aux-old*:
  fixes *s::nat*
  assumes [*autoref-rules*]:
    (*succi*,*E*)∈⟨*nat-rel*⟩*slg-rel*
    (*Ai*,*A*)∈⟨*nat-rel*⟩*dflt-rs-rel*
  notes [*autoref-rules*] = *IdI*[*of s*]
  shows (*RETURN* (*?f::?'c*), *blue-dfs E A s*) ∈ ⟨*?R*⟩*nres-rel*
  unfolding *blue-dfs-def red-dfs-def*
  using [[*autoref-trace*]]
  apply (*autoref-monadic*)
  done

end

5.2 Simple DFS Algorithm

theory *Simple-DFS*
imports
  ../Refine-Dflt
begin

This example presents the usage of the recursion combinator *RECT*. The usage of the partial correct version *REC* is similar.

We define a simple DFS-algorithm, prove a simple correctness property, and do data refinement to an efficient implementation.

5.2.1 Definition

hide-const *Zorn.succ*

Recursive DFS-Algorithm. *E* is the edge relation of the graph, *vd* the node to search for, and *v0* the start node. Already explored nodes are stored in *V*.

definition *dfs* :: (*'a ⇒'a set*) ⇒ *'a* ⇒ *'a* ⇒ *bool nres*
  where
  dfs succ vd v0 ≡ REC_T (λ*D* (*V*,*v*).
    if v=vd then RETURN True
    else if v∈V then RETURN False
    else do {
      let V=insert v V;
      $FOREACH_C$ (*succ v*) (*op* = *False*) (λ*v'* -. *D* (*V*,*v'*)) *False* }
) ({},*v0*)

5.2.2 Correctness

As simple correctness property, we show: If the algorithm returns true, then *vd* is reachable from *v0*.

lemma *dfs-sound*:
  fixes *succ*
  defines *E* ≡ {(*v*,*v*′). *v*′∈*succ v*}
  assumes *F*: *finite* {*v*. (*v0*,*v*)∈*E**}
  shows *dfs succ vd v0* ≤ *SPEC* (λ*r*. *r* ⟶ (*v0*,*vd*)∈*E**)
proof −
  have *S*: ⋀*v*. *succ v* = *E*‘‘{*v*}
    by (*auto simp*: *E-def*)

  from *F* **show** *?thesis*
    unfolding *dfs-def S*
    apply (*refine-rcg refine-vcg impI*
      RECT-rule[**where**
        Φ=λ(*V*,*v*). (*v0*,*v*)∈*E** ∧ *V*⊆{*v*. (*v0*,*v*)∈*E**} **and**
        V=*finite-psupset* ({*v*. (*v0*,*v*)∈*E**}) <*lex*> {}]
      FOREACHc-rule[**where** *I*=λ- *r*. *r* ⟶ (*v0*, *vd*) ∈ *E**]
)
    apply (*auto intro*: *finite-subset*[*of* - {*v*′. (*v0*,*v*′)∈*E**}])
    apply *rprems*
    apply (*auto simp*: *finite-psupset-def*)
    done
qed

5.2.3 Data Refinement and Determinization

Next, we use automatic data refinement and transfer to generate an executable algorithm. The edges function is refined to a successor function returning a list-set.

schematic-lemma *dfs-impl-refine-aux*:
  fixes *succi* **and** *succ* :: *nat* ⇒ *nat set* **and** *vd v0* :: *nat*
  assumes [*autoref-rules*]: (*succi*,*succ*)∈*Id*→⟨*Id*⟩*list-set-rel*
  notes [*autoref-rules*] = *IdI*[*of v0*] *IdI*[*of vd*]
  shows (*?f*::*?′c*, *dfs succ vd v0*)∈*?R*
  unfolding *dfs-def*[*abs-def*]
  apply (*autoref-monadic*)
  done

We can configure our tool to use different implementations. Here, we use lists for sets of natural numbers.

schematic-lemma *dfs-impl-refine-aux2*:
  fixes *succi* **and** *succ* :: *nat* ⇒ *nat set* **and** *vd v0* :: *nat*
  assumes [*autoref-rules*]: (*succi*,*succ*)∈*Id*→⟨*Id*⟩*dflt-rs-rel*
  notes [*autoref-rules*] = *IdI*[*of v0*] *IdI*[*of vd*]
  notes [*autoref-tyrel*] = *ty-REL*[**where** ′*a*=*nat set* **and** *R*=⟨*Id*⟩*list-set-rel*]

shows (*?f*::*?'c*, *dfs succ vd v0*)∈*?R*
unfolding *dfs-def*[*abs-def*]
apply (*autoref-monadic*)
done

We can also leave the type of the nodes and its implementation unspecified. However, the implementation relation must be single-valued, and we need a comparison operator on nodes

schematic-lemma *dfs-impl-refine-aux3*:
fixes *succi* **and** *succ* :: *'a*::*linorder* ⇒ *'a set*
and *Rv* :: (*'ai*×*'a*) *set*
assumes [*relator-props*]: *single-valued Rv*
assumes [*autoref-rules-raw*]: (*cmpk*, *dflt-cmp op* ≤ *op* <)∈(*Rv*→*Rv*→*Id*)
notes [*autoref-tyrel*] = *ty-REL*[**where** *'a*=*'a set* **and** *R*=⟨*Rv*⟩*dflt-rs-rel*]
assumes *P-REF*[*autoref-rules*]:
(*succi*,*succ*)∈*Rv*→⟨*Rv*⟩*list-set-rel*
(*vdi*,*vd*::*'a*)∈*Rv*
(*v0i*,*v0*)∈*Rv*
shows (*?f*::*?'c*, *dfs succ vd v0*)∈*?R*
unfolding *dfs-def*[*abs-def*]
by *autoref-monadic*

Next, we extract constants from the refinement lemmas, and prepare them for code-generation

concrete-definition *dfs-impl* **for** *succi vd ?v0.0* **uses** *dfs-impl-refine-aux*
prepare-code-thms *dfs-impl-def*
concrete-definition *dfs-impl2* **for** *succi vd ?v0.0* **uses** *dfs-impl-refine-aux2*
prepare-code-thms *dfs-impl2-def*
concrete-definition *dfs-impl3* **for** *succi vd ?v0.0* **uses** *dfs-impl-refine-aux3*
prepare-code-thms *dfs-impl3-def*

Finally, we export code using the code-generator

export-code *dfs-impl dfs-impl2 dfs-impl3* **in** *SML* **file** −
export-code *dfs-impl dfs-impl2 dfs-impl3* **in** *OCaml* **file** −
export-code *dfs-impl dfs-impl2 dfs-impl3* **in** *Haskell* **file** −
export-code *dfs-impl dfs-impl2 dfs-impl3* **in** *Scala* **file** −

Derived correctness lemma for the generated function

lemma *dfs-impl-correct*:
fixes *succi succ*
defines $E \equiv \{(s, s').\ s' \in succ\ s\}$
assumes *S*: (*succi*,*succ*) ∈ *Id* → ⟨*Id*⟩*list-set-rel*
assumes *F*: *finite* (E^* ‘‘{*v0*})
assumes *R*: *dfs-impl succi vd v0*
shows (*v0*,*vd*)∈E^*
proof −
note *dfs-impl.refine*[*OF S*, *of vd v0*, *THEN nres-relD*]
also

```
  have F': finite {v. (v0, v) ∈ {(v, v'). v' ∈ succ v}*}
    using F
    apply (fo-rule back-subst, assumption)
    by (auto simp: E-def)
  note dfs-sound[OF F']
  finally show ?thesis using R
    by (auto simp: E-def)

qed

end

theory Preorder-Equiv-Classes
imports ../../../Refine-Dflt
begin

definition rel-α R ≡ {(x,y). ∃ Rx. R x = Some Rx ∧ y ∈ Rx}

definition preord-eqclasses-map-invar S R it m ≡
  S − it ⊆ dom m ∧ dom m ⊆ S ∧ ran m ⊆ S − it ∧
  (∀ s∈dom m. ∀ t∈S. m s = m t ⟷ ((s,t) ∈ R ∧ (t,s) ∈ R))

lemma preord-eqclasses-map-invarI[intro]:
  assumes S − it ⊆ dom m    dom m ⊆ S    ran m ⊆ S − it
  assumes ⋀s t. s∈dom m ⟹ t∈S ⟹ m s = m t ⟷
             ((s,t) ∈ R ∧ (t,s) ∈ R)
  shows preord-eqclasses-map-invar S R it m
  using assms unfolding preord-eqclasses-map-invar-def by simp

lemma preord-eqclasses-map-invarD[dest]:
  assumes preord-eqclasses-map-invar S R it m
  shows S − it ⊆ dom m and dom m ⊆ S    ran m ⊆ S − it
    and ⋀s t. s∈dom m ⟹ t∈S ⟹ m s = m t ⟷
           ((s,t) ∈ R ∧ (t,s) ∈ R)
  using assms unfolding preord-eqclasses-map-invar-def by simp-all



definition preord-eqclasses-map where
preord-eqclasses-map S R ≡ do {
  ASSUME (finite S);
  ASSUME (preorder-on S R);
  FOREACH^(preord-eqclasses-map-invar S R) S (λs m.
    case m s of
      Some - ⇒ RETURN m |
      None ⇒ RETURN (λx. if (s,x)∈R ∧ (x,s)∈R then Some s else m x)
  ) Map.empty
}
```

```
definition is-preord-eqclasses-map S R m ≡ dom m = S ∧
   (∀ s∈S. ∀ t∈S. m s = m t ⟷ ((s,t)∈R ∧ (t,s)∈R))

lemma is-preord-eqclasses-mapI[intro]:
  assumes dom m = S
  assumes ⋀s t. s∈S ⟹ t∈S ⟹ m s = m t ⟷
            (s,t)∈R ∧ (t,s)∈R
  shows is-preord-eqclasses-map S R m
  using assms unfolding is-preord-eqclasses-map-def by simp

lemma is-preord-is-eqclasses-mapD[dest]:
  assumes is-preord-eqclasses-map S R m
  shows dom m = S
    and ⋀s t. s∈S ⟹ t∈S ⟹ m s = m t ⟷
          (s,t)∈R ∧ (t,s)∈R
  using assms unfolding is-preord-eqclasses-map-def by simp-all


lemma preord-eqclasses-map-correct:
  preord-eqclasses-map S R ≤ SPEC (is-preord-eqclasses-map S R)
  unfolding preord-eqclasses-map-def
proof (intro refine-vcg FOREACH-rule)
  assume finite S thus finite S .
next
  show preord-eqclasses-map-invar S R S Map.empty
    by (intro preord-eqclasses-map-invarI, simp-all)
next
  case (goal3 s it m)
    hence s ∈ S by blast
    note inv = preord-eqclasses-map-invarD[OF goal3(5)]
    from inv(3) and ‹s ∈ it› have [dest!]:
        ⋀x. m x = Some s ⟹ False by (blast intro: ranI)
    hence [dest!]: ⋀x. Some s = m x ⟹ False by force

    from ‹preorder-on S R› have R-in-S: R ⊆ S×S
        unfolding preorder-on-def refl-on-def by simp
    from ‹preorder-on S R› have
         refl: ⋀x. x∈S ⟹ (x,x) ∈ R and
         trans: ⋀x y z. (x,y)∈R ⟹ (y,z)∈R ⟹ (x,z)∈R
        unfolding preorder-on-def by (blast dest: refl-onD transD)+

    let ?m′ = λx. if (s, x) ∈ R ∧ (x, s) ∈ R then Some s else m x
    have new-dom: dom ?m′ = dom m ∪ {x. (s,x) ∈ R ∧ (x,s) ∈ R} by auto
    show ?case
    proof (intro preord-eqclasses-map-invarI)
      have s ∈ {x. (s,x) ∈ R ∧ (x,s) ∈ R}
          using ‹s ∈ S› refl by blast
      thus S − (it − {s}) ⊆ dom ?m′
          using inv(1) by (subst new-dom, blast)
```

```
    next
      show dom ?m' ⊆ S using inv(2) R-in-S by (subst new-dom, blast)
    next
      show ran ?m' ⊆ S − (it − {s}) using inv(3) ‹s ∈ S›
          by (auto simp: ran-def)
    next
      fix s' t assume s'-in-dom: s' ∈ dom ?m' and t: t ∈ S
      hence s': s' ∈ S using R-in-S and inv(2) by (auto simp: new-dom)

      show ?m' s' = ?m' t ⟷ (s',t) ∈ R ∧ (t,s') ∈ R
      proof (cases (s,s') ∈ R ∧ (s',s) ∈ R)
        case True
          thus ?thesis by (force intro: trans)
      next
        case False
          hence s' ∈ dom m using s'-in-dom
              by (simp only: new-dom, blast)
          from inv(4)[OF this t]
               show ?thesis by (force intro: trans)
      qed
  qed
next
  case (goal4 s it m)
     thus ?case by (intro preord-eqclasses-map-invarI, auto)
next
  case (goal5 m)
    thus ?case by blast
qed
```

```
definition preord-eqclasses-map-impl1-loop-invar S R s m it m' ≡
    (∀x. m' x = (if (s,x) ∈ R ∧ (x,s) ∈ R ∧ x ∉ it
                then Some s else m x))

definition preord-eqclasses-map-impl1-loop S R s m ≡
    FOREACH^{preord-eqclasses-map-impl1-loop-invar S R s m}
        {x. (s,x) ∈ R} (λt m. if (t,s) ∈ R
            then RETURN (m(t ↦ s))
            else RETURN m) m

definition preord-eqclasses-map-impl1 where
preord-eqclasses-map-impl1 S R ≡ do {
  ASSUME (finite S);
  ASSUME (preorder-on S R);
  FOREACH S (λs m.
    case m s of
      Some - ⇒ RETURN m |
      None ⇒ preord-eqclasses-map-impl1-loop S R s m
```

```
  ) Map.empty
}

lemma preord-eqclasses-map-impl1-loop-correct:
  assumes fin: finite S and preord: preorder-on S R
      and inv: preord-eqclasses-map-invar S R it m
  shows preord-eqclasses-map-impl1-loop S R s m ≤
      SPEC (λm'. m' = (λx. if (s, x) ∈ R ∧ (x, s) ∈ R
                          then Some s else m x))
unfolding preord-eqclasses-map-impl1-loop-def
proof (intro refine-vcg FOREACH-rule)
  from preord have R ⊆ S × S
      by (simp add: preorder-on-def refl-on-def)
  hence {x. (s, x) ∈ R} ⊆ S by blast
  thus finite {x. (s, x) ∈ R} using fin finite-subset by blast
next
  show preord-eqclasses-map-impl1-loop-invar S R s m {x. (s, x) ∈ R} m
      unfolding preord-eqclasses-map-impl1-loop-invar-def by force
qed (unfold preord-eqclasses-map-impl1-loop-invar-def, auto)

lemma preord-eqclasses-map-impl1-loop-correct':
  assumes fin: finite S and preord: preorder-on S R
      and inv: preord-eqclasses-map-invar S R it' m'
      and s' = id s    (m,m') ∈ Id
  shows preord-eqclasses-map-impl1-loop S R s m ≤
      SPEC (λm''. (m'',(λx. if (s', x) ∈ R ∧ (x, s') ∈ R
                          then Some s' else m' x)) ∈ Id)
proof−
  from assms have A: s' = s    m' = m by simp-all
  with preord-eqclasses-map-impl1-loop-correct[OF fin preord inv]
    show ?thesis by (simp only: A, simp)
qed


lemma preord-eqclasses-map-impl1-refine:
  shows preord-eqclasses-map-impl1 S R ≤ ⇓Id (preord-eqclasses-map S R)
unfolding preord-eqclasses-map-impl1-def preord-eqclasses-map-def
by (refine-rcg inj-on-id, simp, simp, simp,
       erule (4) preord-eqclasses-map-impl1-loop-correct')


definition preord-eqclasses-map-impl2-loop S R s m ≡
  case R s of
    None ⇒ RETURN m |
    Some Rs ⇒ FOREACH Rs (λt m. RETURN (
        let ts = case R t of None ⇒ False |
                           Some Rt ⇒ s ∈ Rt
        in if ts then m(t ↦ s) else m)
      ) m
```

```
definition preord-eqclasses-map-impl2 where
preord-eqclasses-map-impl2 S R ≡ FOREACH S (λs m.
    case m s of
      Some - ⇒ RETURN m |
      None ⇒ preord-eqclasses-map-impl2-loop S R s m
      ) Map.empty

lemma preord-eqclasses-map-impl2-loop-refine:
  fixes s::'a and R'::('a×'a) set
  assumes s ∈ S    preorder-on S R'    R' = rel-α R
  shows preord-eqclasses-map-impl2-loop S R s m ≤
            ⇓Id (preord-eqclasses-map-impl1-loop S R' s m)
proof−
  let ?cond = λt. case R t of None ⇒ False |
                              Some x ⇒ s ∈ x
  have cond-simp:
      ⋀t it. t ∈ it ⟹ it ⊆ {x. (s,x) ∈ R'} ⟹
           ?cond t = ((t,s) ∈ R')
      using assms unfolding rel-α-def
      by (force split: option.split-asm)
  from assms have (s,s) ∈ R'
      unfolding preorder-on-def refl-on-def by simp
  hence R s = Some {x. (s,x) ∈ R'}
      using assms unfolding rel-α-def by force
  hence preord-eqclasses-map-impl2-loop S R s m =
            FOREACH {x. (s,x) ∈ R'} (λt m.
                RETURN (if ?cond t then m(t ↦ s) else m)) m
      unfolding preord-eqclasses-map-impl2-loop-def by simp
  also have ... ≤ ⇓Id (preord-eqclasses-map-impl1-loop S R' s m)
      unfolding preord-eqclasses-map-impl1-loop-def
      by (refine-rcg inj-on-id, simp-all add: cond-simp)
  finally show ?thesis .
qed

lemma preord-eqclasses-map-impl2-loop-refine':
  fixes s::'a and R'::('a×'a) set
  assumes preorder-on S R'    R' = rel-α R
      and s ∈ it    it ⊆ S    s' = id s    (m,m') ∈ Id
  shows preord-eqclasses-map-impl2-loop S R s m ≤
            ⇓Id (preord-eqclasses-map-impl1-loop S R' s' m')
proof−
  from assms(3,4) have s ∈ S by blast
  from preord-eqclasses-map-impl2-loop-refine[OF this assms(1,2)] assms
      show ?thesis by simp
qed

lemma preord-eqclasses-map-impl2-refine:
  assumes fin: finite S and preord: preorder-on S R'
```

and *R*: R' = *rel-α R*
shows *preord-eqclasses-map-impl2 S R* ≤
⇓*Id* (*preord-eqclasses-map-impl1 S R′*)
unfolding *preord-eqclasses-map-impl2-def*
preord-eqclasses-map-impl1-def
using *assms* **by** (*refine-rcg inj-on-id*, *simp*, *simp*, *simp*)
(*erule* (*3*) *preord-eqclasses-map-impl2-loop-refine′*[*OF preord R*])

abbreviation *preord-eqclasses-map-impl* ≡ *preord-eqclasses-map-impl2*
lemmas *preord-eqclasses-map-impl-def* = *preord-eqclasses-map-impl2-def*

lemma *preord-eqclasses-map-impl-correct*:
assumes *finite S* **and** *preorder-on S R′*
assumes (*R*,*R′*) ∈ *br rel-α* (λ-. *True*)
shows *preord-eqclasses-map-impl S R* ≤ *SPEC* (*is-preord-eqclasses-map S R′*)
proof−
from *assms*(*3*) **have** *R′* = *rel-α R* **unfolding** *br-def* **by** *simp*
from *preord-eqclasses-map-impl2-refine*[*OF assms*(*1*,*2*) *this*]
have *preord-eqclasses-map-impl2 S R* ≤
(*preord-eqclasses-map-impl1 S R′*) **by** *simp*
also from *preord-eqclasses-map-impl1-refine*
have *preord-eqclasses-map-impl1 S R′* ≤
(*preord-eqclasses-map S R′*) **by** *simp*
also from *preord-eqclasses-map-correct*
have *preord-eqclasses-map S R′* ≤
SPEC (*is-preord-eqclasses-map S R′*) **.**
finally show *?thesis* **.**
qed

schematic-lemma *preord-eqclasses-map-code-refine*:
assumes [*autoref-rules*]: (*S*,*S′*) ∈ ⟨*nat-rel*⟩*dflt-rs-rel*
assumes [*autoref-rules*]: (*R*,*R′*) ∈ ⟨*nat-rel*,⟨*nat-rel*⟩*dflt-rs-rel*⟩*dflt-rm-rel*
shows (*?f*::*?′c*, *preord-eqclasses-map-impl S′ R′*) ∈ *?R*
unfolding *preord-eqclasses-map-impl-def*
preord-eqclasses-map-impl2-loop-def
using *assms* **by** *autoref-monadic*

concrete-definition *preord-eqclasses-map-code* **uses** *preord-eqclasses-map-code-refine*

end

theory *NFA-Refine*
imports *Main NFA ../../Refine-Dflt*
begin

fun *𝒬-impl* **where** *𝒬-impl* (*Q*,*S*,*D*,*I*,*F*) = *Q*
fun *Σ-impl* **where** *Σ-impl* (*Q*,*S*,*D*,*I*,*F*) = *S*

fun Δ-*impl* **where** Δ-*impl* $(Q,S,D,I,F) = D$
fun $\mathcal{I}$-*impl* **where** $\mathcal{I}$-*impl* $(Q,S,D,I,F) = I$
fun $\mathcal{F}$-*impl* **where** $\mathcal{F}$-*impl* $(Q,S,D,I,F) = F$

definition
NFA-rel :: - ⇒ - ⇒ - ⇒ - ⇒ - ⇒ - ⇒ - ⇒ (-×(-,-)*NFA-rec*) *set*
where
NFA-rel-internal-def: *NFA-rel Rqs Rss Rds Ris Rfs R*$\mathcal{Q}$ *R*Σ ≡
{ ((*Q*,*S*,*D*,*I*,*F*),$\mathcal{A}$) .
NFA $\mathcal{A}$ ∧
(*Q*,$\mathcal{Q}$ $\mathcal{A}$)∈⟨*R*$\mathcal{Q}$⟩*Rqs* ∧
(*S*,Σ $\mathcal{A}$)∈⟨*R*Σ⟩*Rss* ∧
(*D*,Δ $\mathcal{A}$)∈⟨⟨*R*$\mathcal{Q}$,⟨*R*Σ,*R*$\mathcal{Q}$⟩*prod-rel*⟩*prod-rel*⟩*Rds* ∧
(*I*,$\mathcal{I}$ $\mathcal{A}$)∈⟨*R*$\mathcal{Q}$⟩*Ris* ∧
(*F*,$\mathcal{F}$ $\mathcal{A}$)∈⟨*R*$\mathcal{Q}$⟩*Rfs*}

lemma *NFA-rel-def*: ⟨*R*$\mathcal{Q}$,*R*Σ⟩*NFA-rel Rqs Rss Rds Ris Rfs* ≡ { ((*Q*,*S*,*D*,*I*,*F*),$\mathcal{A}$)
.
NFA $\mathcal{A}$ ∧
(*Q*,$\mathcal{Q}$ $\mathcal{A}$)∈⟨*R*$\mathcal{Q}$⟩*Rqs* ∧
(*S*,Σ $\mathcal{A}$)∈⟨*R*Σ⟩*Rss* ∧
(*D*,Δ $\mathcal{A}$)∈⟨⟨*R*$\mathcal{Q}$,⟨*R*Σ,*R*$\mathcal{Q}$⟩*prod-rel*⟩*prod-rel*⟩*Rds* ∧
(*I*,$\mathcal{I}$ $\mathcal{A}$)∈⟨*R*$\mathcal{Q}$⟩*Ris* ∧
(*F*,$\mathcal{F}$ $\mathcal{A}$)∈⟨*R*$\mathcal{Q}$⟩*Rfs*}
unfolding *NFA-rel-internal-def*[*abs-def*] *relAPP-def* **.**

lemma *NFA-rel-sv*[*relator-props*]:
assumes *single-valued* (⟨*R*$\mathcal{Q}$⟩*Rqs*)
assumes *single-valued* (⟨*R*Σ⟩*Rss*)
assumes *single-valued* (⟨⟨*R*$\mathcal{Q}$,⟨*R*Σ,*R*$\mathcal{Q}$⟩*prod-rel*⟩*prod-rel*⟩*Rds*)
assumes *single-valued* (⟨*R*$\mathcal{Q}$⟩*Ris*)
assumes *single-valued* (⟨*R*$\mathcal{Q}$⟩*Rfs*)
shows *single-valued* (⟨*R*$\mathcal{Q}$,*R*Σ⟩*NFA-rel Rqs Rss Rds Ris Rfs*)
apply (*intro single-valuedI allI*)
apply (*auto simp add*: *NFA-rel-def*)
apply (*case-tac y*)
apply (*case-tac z*)
apply (*auto dest*: *assms*[*THEN single-valuedD*])
done

consts *i-NFA* :: *interface* ⇒ *interface* ⇒ *interface*

lemmas [*autoref-rel-intf*] =
REL-INTFI[*of NFA-rel Rqs Rss Rds Ris Rfs i-NFA*, *standard*]

lemma $\mathcal{Q}$-*autoref*[*autoref-rules*]:
($\mathcal{Q}$-*impl*,$\mathcal{Q}$)∈⟨*R*$\mathcal{Q}$,*R*Σ⟩*NFA-rel Rqs Rss Rds Ris Rfs* → ⟨*R*$\mathcal{Q}$⟩*Rqs*
unfolding *NFA-rel-def* **by** *auto*
lemma Σ-*autoref*[*autoref-rules*]:

(Σ-*impl*,Σ)∈⟨*R𝒬*,*R*Σ⟩*NFA-rel Rqs Rss Rds Ris Rfs* → ⟨*R*Σ⟩*Rss*
unfolding *NFA-rel-def* **by** *auto*
lemma Δ-*autoref* [*autoref-rules*]:
(Δ-*impl*,Δ)∈⟨*R𝒬*,*R*Σ⟩*NFA-rel Rqs Rss Rds Ris Rfs*
→ ⟨⟨*R𝒬*,⟨*R*Σ,*R𝒬*⟩*prod-rel*⟩*prod-rel*⟩*Rds*
unfolding *NFA-rel-def* **by** *auto*
lemma ℐ-*autoref* [*autoref-rules*]:
(ℐ-*impl*,ℐ)∈⟨*R𝒬*,*R*Σ⟩*NFA-rel Rqs Rss Rds Ris Rfs* → ⟨*R𝒬*⟩*Ris*
unfolding *NFA-rel-def* **by** *auto*
lemma ℱ-*autoref* [*autoref-rules*]:
(ℱ-*impl*,ℱ)∈⟨*R𝒬*,*R*Σ⟩*NFA-rel Rqs Rss Rds Ris Rfs* → ⟨*R𝒬*⟩*Rfs*
unfolding *NFA-rel-def* **by** *auto*

fun *NFA-reverse-impl* **where**
NFA-reverse-impl D-img (*Q*,*S*,*D*,*I*,*F*) =
(*Q*, *S*, *D-img* (λ(*q1*,*l*,*q2*). (*q2*,*l*,*q1*)) *D*, *F*, *I*)

lemma *NFA-reverse-alt*: *NFA-reverse* 𝒜 =
⦇𝒬 = 𝒬 𝒜, Σ = Σ 𝒜, Δ = (λ(*q*, σ, *p*). (*p*, σ, *q*)) ‘Δ 𝒜, ℐ = ℱ 𝒜, ℱ = ℐ 𝒜⦈
unfolding *NFA-reverse-def*
by (*force simp*: *image-def split*: *prod.splits*)

lemma *NFA-reverse-autoref* [*autoref-rules*]:
fixes *R𝒬 R*Σ
defines [*simp*]: *trip-rel* ≡ ⟨*R𝒬*,⟨*R*Σ,*R𝒬*⟩*prod-rel*⟩*prod-rel*
assumes [*unfolded autoref-tag-defs trip-rel-def*, *param*]:
GEN-OP D-img op ‘ ((*trip-rel* → *trip-rel*) → ⟨*trip-rel*⟩*Rds* → ⟨*trip-rel*⟩*Rds*)
shows (*NFA-reverse-impl D-img*, *NFA-reverse*) ∈
⟨*R𝒬*,*R*Σ⟩*NFA-rel Rqs Rss Rds Rifs Rifs*
→ ⟨*R𝒬*,*R*Σ⟩*NFA-rel Rqs Rss Rds Rifs Rifs*
apply (*rule fun-relI*)
unfolding *NFA-rel-def*
apply *clarsimp*
apply (*rule conjI*)
apply (*erule NFA-reverse---is-well-formed*)
unfolding *NFA-reverse-alt*
apply *clarsimp*
apply *parametricity*
done

fun *NFA-rename-states-impl* **where**
NFA-rename-states-impl Q-img D-img I-img F-img (*Q*,*S*,*D*,*I*,*F*) *f* =
(*Q-img f Q*, *S*, *D-img* (λ(*u*,*c*,*v*). (*f u*, *c*, *f v*)) *D*, *I-img f I*, *F-img f F*)

thm *NFA-rename-states-def SemiAutomaton-rename-states-ext-def*

lemma *NFA-rename-states-alt*: *NFA-rename-states* 𝒜 *f* =

```
(|Q = f`Q A, Σ = Σ A,
  Δ = (λ(p, σ, q). (f p, σ, f q))`Δ A, I = f`I A, F = f`F A
|)
unfolding NFA-rename-states-def SemiAutomaton-rename-states-ext-def
by (force simp: image-def split: prod.splits)

lemma NFA-rename-states-autoref[autoref-rules]:
  fixes RQ RΣ
  defines [simp]: trip-rel ≡ ⟨RQ,⟨RΣ,RQ⟩prod-rel⟩prod-rel
  assumes [unfolded autoref-tag-defs trip-rel-def, param]:
    GEN-OP Q-img op ` ((RQ → RQ) → ⟨RQ⟩Rqs → ⟨RQ⟩Rqs)
    GEN-OP D-img op ` ((trip-rel → trip-rel) → ⟨trip-rel⟩Rds → ⟨trip-rel⟩Rds)
    GEN-OP I-img op ` ((RQ → RQ) → ⟨RQ⟩Ris → ⟨RQ⟩Ris)
    GEN-OP F-img op ` ((RQ → RQ) → ⟨RQ⟩Rfs → ⟨RQ⟩Rfs)
  shows (NFA-rename-states-impl Q-img D-img I-img F-img, NFA-rename-states)
∈
    ⟨RQ,RΣ⟩NFA-rel Rqs Rss Rds Ris Rfs → (RQ → RQ)
    → ⟨RQ,RΣ⟩NFA-rel Rqs Rss Rds Ris Rfs
  apply (rule fun-relI)
  unfolding NFA-rel-def
  apply clarsimp
  apply (intro conjI)
  apply (erule NFA-rename-states---is-well-formed)
  apply parametricity
  unfolding NFA-rename-states-alt
  apply clarsimp
  apply parametricity
  apply parametricity
  apply parametricity
  done


abbreviation dflt-NFA-rel
  ≡ NFA-rel dflt-rs-rel dflt-rs-rel dflt-rs-rel dflt-rs-rel dflt-rs-rel

lemma dflt-NFA-rel-sv[relator-props]:
  assumes [relator-props]: single-valued RQ    single-valued RΣ
  shows single-valued (⟨RQ,RΣ⟩dflt-NFA-rel)
  by tagged-solver
```

5.2.4 Tests

```
schematic-lemma
  assumes [autoref-rules]: (Aimpl,A)∈⟨nat-rel,nat-rel⟩dflt-NFA-rel
  shows (?f::?'c, NFA-reverse A) ∈ ?R
  apply (autoref (keep-goal))
  done

schematic-lemma
```

assumes [*autoref-rules*]: ($\mathcal{A}$*impl*,$\mathcal{A}$)∈⟨*nat-rel*,*nat-rel*⟩*dflt-NFA-rel*
shows (*?f::?'c*, *RETURN* (*NFA-rename-states* $\mathcal{A}$ ($\lambda x.\ x+1$))) ∈ *?R*
apply (*autoref-monadic*)
done

end

5.3 The algorithm by Ilie, Navarro and Yu

theory *NFA-Simulations-INY*
imports *Main NFA-Simulations Lib/Preorder-Equiv-Classes NFA-Refine ../../Refine-Dflt*

begin

We verify the algorithm by Ilie, Navarro and Yu for computation of simulation preorders in nondeterministic finite automata. We use the Refinement Framework to produce efficiently executable code.

context *NFA*
begin

5.3.1 Preliminary definitions

The complement relation of a relation $\mathcal{S}$ over states of the automaton $\mathcal{A}$

abbreviation *compl* **where** *compl* $\mathcal{S} \equiv \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A} - \mathcal{S}$

The complement relation of a simulation contains all pairs $(u,\ v)$ where $u \in \mathcal{F}\ \mathcal{A}$ and $v \notin \mathcal{F}\ \mathcal{A}$, as final states can only be simulated by other final states

lemma *sim-compl-subset*[*intro*]:
is-sim $\mathcal{S} \Longrightarrow \mathcal{F}\ \mathcal{A} \times (\mathcal{Q}\ \mathcal{A} - \mathcal{F}\ \mathcal{A}) \subseteq$ *compl* $\mathcal{S}$
using $\mathcal{F}$*-consistent* **unfolding** *is-sim-def* **by** *blast*

the complement function is its own inverse function

lemma *compl-compl*[*simp*,*intro*]: $x \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A} \Longrightarrow$ *compl* (*compl* x) = x
using $\mathcal{F}$*-consistent* **by** *blast*

Some lemmata about how set inequalities between two relations reverse when taking the complements of both sides

lemma *compl-subseteq-reverse*: ⟦$x \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$; $y \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$; $x \subseteq$ *compl* y⟧
$\Longrightarrow y \subseteq$ *compl* x **using** $\mathcal{F}$*-consistent* **by** *blast*
lemma *compl-subseteq*: ⟦$x \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$; $y \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$; $x \subseteq y$⟧
$\Longrightarrow$ *compl* $y \subseteq$ *compl* x **using** $\mathcal{F}$*-consistent* **by** *blast*
lemma *compl-subset-reverse*: ⟦$x \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$; $y \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$; $x \subset$ *compl* y⟧
$\Longrightarrow y \subset$ *compl* x **using** $\mathcal{F}$*-consistent* **by** *blast*
lemma *compl-subset*: ⟦$x \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$; $y \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$; $x \subset y$⟧
$\Longrightarrow$ *compl* $y \subset$ *compl* x **using** $\mathcal{F}$*-consistent* **by** *blast*

5.3.2 Abstract algorithm

The invariant of the WHILE loop in the algorithm It consists of five parts: 1. ω may only relate states of the automaton 2. ω must relate all final states with all nonfinal states 3. $\mathcal{C}$ must be a subset of ω 4. ω must not relate two states that are also related by a simulation (i.e. ω must be disjoint from all simulation relations) This means that ω never gets "too large" 5. if two states u and v are not related by ω, each successor u' of u must have a matching successor v' of v that is either not in ω or still in $\mathcal{C}$. This means that after each loop iteration, ω is "large enough" w.r.t. the state pairs processed so far.

definition *INY-abstr1-invar* **where**
INY-abstr1-invar $\equiv \lambda(\omega, \mathcal{C})$.
$\omega \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}\ \wedge\ \mathcal{F}\ \mathcal{A} \times (\mathcal{Q}\ \mathcal{A} - \mathcal{F}\ \mathcal{A}) \subseteq \omega\ \wedge\ \mathcal{C} \subseteq \omega\ \wedge \mathcal{S}_{\mathcal{A}} \cap \omega = \{\}\ \wedge$
$(\forall u\ v\ u'\ c.\ (u,v) \in compl\ \omega \wedge (u,c,u') \in \Delta\ \mathcal{A} \longrightarrow$
$(\exists v'.\ (v,c,v') \in \Delta\ \mathcal{A} \wedge (u',v') \in compl\ (\omega - \mathcal{C})))$

lemma *INY-abstr1-invarI*[*intro*]:
assumes $\omega \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$ **and**
$\mathcal{F}\ \mathcal{A} \times (\mathcal{Q}\ \mathcal{A} - \mathcal{F}\ \mathcal{A}) \subseteq \omega$ **and** $\mathcal{C} \subseteq \omega$ **and** $\mathcal{S}_{\mathcal{A}} \cap \omega = \{\}$ **and**
$\bigwedge u\ v\ u'\ c.\ [\![(u,v) \in compl\ \omega;\ (u,c,u') \in \Delta\ \mathcal{A}]\!] \Longrightarrow$
$\exists v'.\ (v,c,v') \in \Delta\ \mathcal{A} \wedge (u',v') \in compl\ (\omega - \mathcal{C})$
shows *INY-abstr1-invar* $(\omega, \mathcal{C})$ **unfolding** *INY-abstr1-invar-def*
apply (*clarify*, *intro conjI*)
using *assms* **apply** (*blast*, *blast*, *blast*, *blast*) **apply** (*blast intro*: *assms*(*5*))
done

lemma *INY-abstr1-invarD*:
assumes *INY-abstr1-invar* $(\omega, \mathcal{C})$
shows $\omega \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$ **and** $\mathcal{F}\ \mathcal{A} \times (\mathcal{Q}\ \mathcal{A} - \mathcal{F}\ \mathcal{A}) \subseteq \omega$ **and** $\mathcal{C} \subseteq \omega$ **and** $\mathcal{S}_{\mathcal{A}} \cap \omega = \{\}$
$\bigwedge u\ v\ u'\ c.\ [\![(u,v) \in compl\ \omega;\ (u,c,u') \in \Delta\ \mathcal{A}]\!] \Longrightarrow$
$\exists v'.\ (v,c,v') \in \Delta\ \mathcal{A} \wedge (u',v') \in compl\ (\omega - \mathcal{C})$
using *assms* **unfolding** *INY-abstr1-invar-def* **by** *blast+*

lemma *INY-abstr1-invar-emptyD*:
assumes *INY-abstr1-invar* $(\omega, \{\})$
shows $\omega \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$ **and** $\mathcal{F}\ \mathcal{A} \times (\mathcal{Q}\ \mathcal{A} - \mathcal{F}\ \mathcal{A}) \subseteq \omega$ **and** $\mathcal{S}_{\mathcal{A}} \cap \omega = \{\}$ **and**
$\bigwedge u\ v\ u'\ c.\ [\![(u,v) \in compl\ \omega;\ (u,c,u') \in \Delta\ \mathcal{A}]\!] \Longrightarrow$
$\exists v'.\ (v,c,v') \in \Delta\ \mathcal{A} \wedge (u',v') \in compl\ \omega$
apply *clarify*
using *INY-abstr1-invarD*[*OF assms*] **apply** (*blast*, *blast*, *blast*)
using *INY-abstr1-invarD*(*5*)[*OF assms*] **apply** *blast*
done

The initial ω. These are all pairs of states (u, v) of which we know that v does not simulate u from the start. This can be because one of the following reasons: - u is a final state and v is not - u has a successor w.r.t. a character

c whereas v does not. Note that the second case is not taken into account in the paper by Ilie et al. This is a mistake, as it causes some state pairs in non-total NFAs to be erroneously marked as simulating.

definition *INY-initial* **where**
INY-initial $= \mathcal{F}\ \mathcal{A} \times (\mathcal{Q}\ \mathcal{A} - \mathcal{F}\ \mathcal{A}) \cup$
$\{(u,v).\ u \in \mathcal{Q}\ \mathcal{A} \wedge v \in \mathcal{Q}\ \mathcal{A} \wedge (\exists\, c\ u'.\ (u,c,u') \in \Delta\ \mathcal{A} \wedge \neg(\exists\, v'.\ (v,c,v') \in \Delta\ \mathcal{A}))\}$

The while loop invariant holds initially

lemma *INY-abstr1-invar-initial*:
INY-abstr1-invar (*INY-initial*, *INY-initial*)
apply (*rule INY-abstr1-invarI*)
using $\mathcal{F}$*-consistent INY-initial-def* **apply** (*fast*, *simp*, *simp*)
unfolding *INY-initial-def* **using** $\mathcal{S}_{\mathcal{A}}$*-is-largest-sim* **apply** *blast*
using Δ*-consistent* **apply** *blast*
done

The new entries that have to be added to ω in one interation, i.e. the state pairs $(u,\ v)$ that become unsimulatable if we now know that $(u',\ v')$ is not simulatable.

definition *INY-abstr1-set* **where**
INY-abstr1-set $\omega\ \mathcal{C}\ u'\ v' =$
$\{(u,v)\ |u\ v\ c.\ (u,v) \in compl\ \omega \wedge (u,c,u') \in \Delta\ \mathcal{A} \wedge (v,c,v') \in \Delta\ \mathcal{A} \wedge$
$(\forall\, v''.\ (v,c,v'') \in \Delta\ \mathcal{A} \longrightarrow (u',v'') \in \omega - \mathcal{C})\}$

lemma *INY-abstr1-set-subset-QQ*[*simp*]:
INY-abstr1-set $\omega\ \mathcal{C}\ u'\ v' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$
unfolding *INY-abstr1-set-def* **by** *blast*

if a pair $(u,\ v)$ is in these entries, we know that it is not a simulating pair, but it is also not yet in ω.

lemma *INY-abstr1-set-memE*:
assumes $(u,v) \in$ *INY-abstr1-set* $\omega\ \mathcal{C}\ u'\ v'$
obtains c **where** $(u,v) \in compl\ \omega \wedge (u,c,u') \in \Delta\ \mathcal{A} \wedge (v,c,v') \in \Delta\ \mathcal{A} \wedge$
$(\forall\, v''.\ (v,c,v'') \in \Delta\ \mathcal{A} \longrightarrow (u',v'') \in \omega - \mathcal{C})$
using *assms* **unfolding** *INY-abstr1-set-def* **by** *blast*

if a pair $(u,\ v)$ is not in these entries, we know that it must fulfil the simulation criteria to the best of our current knowledge

lemma *INY-abstr1-set-notmemE*:
assumes $(u,\ v) \notin$ *INY-abstr1-set* $\omega\ \mathcal{C}\ u'\ v'$
$(u,\ v) \in compl\ \omega \quad (u,c,u') \in \Delta\ \mathcal{A} \quad (v,c,v') \in \Delta\ \mathcal{A}$
obtains v'' **where** $(v,c,v'') \in \Delta\ \mathcal{A} \wedge (u',v'') \in compl\ (\omega - \mathcal{C})$
using *assms* Δ*-consistent* **unfolding** *INY-abstr1-set-def* **by** *blast*

None of the state pairs that are to be added in every step of the while loop are in the simulation preorder.

lemma *INY-abstr1-set-disjoint-*$\mathcal{S}_{\mathcal{A}}$:

assumes *I*: *INY-abstr1-invar* $(\omega, \mathcal{C})$ **and** *uv-in-$\mathcal{C}$*: $(u',v')\in\mathcal{C}$
shows $\mathcal{S}_{\mathcal{A}} \cap$ *INY-abstr1-set* ω $(\mathcal{C}-\{(u',v')\})$ u' v' $= \{\}$ (**is** $\mathcal{S}_{\mathcal{A}} \cap$ *?T* $= \{\}$)
proof (*intro equalityI subsetI*, *elim IntE*, *simp-all*, *clarify*)
note *invar* = *INY-abstr1-invarD*[*OF I*]
note *ω-disjoint-$\mathcal{S}_{\mathcal{A}}$* = *invar*(*4*)
fix *u v uv* **assume** $(u,v) \in$ *?T* **and** $(u,v) \in \mathcal{S}_{\mathcal{A}}$
from *INY-abstr1-set-memE*[*OF this*(*1*)] **guess** *c* **.**
moreover with ‹$(u,v) \in \mathcal{S}_{\mathcal{A}}$› **have** $\exists\, v'.\ (u',v') \in \mathcal{S}_{\mathcal{A}} \wedge (v,c,v')\in\Delta\ \mathcal{A}$
using *$\mathcal{S}_{\mathcal{A}}$-is-largest-sim* **by** *blast*
ultimately show *False* **using** *ω-disjoint-$\mathcal{S}_{\mathcal{A}}$* *invar*(*3*) *uv-in-$\mathcal{C}$* **by** *blast*
qed

The conditions that the new values for ω and $\mathcal{C}$ need to fulfil in order for the algorithm to work.

definition *INY-abstr1-is-valid-ω'$\mathcal{C}'$* **where**
INY-abstr1-is-valid-ω'$\mathcal{C}'$ $\omega\ \mathcal{C}\ u'\ v' \equiv \lambda(\omega',\mathcal{C}').\ \omega' \subseteq \mathcal{Q}\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}\ \wedge$
$\omega'=\omega\cup(\omega'-\omega) \wedge \mathcal{C}'=\mathcal{C}\cup(\omega'-\omega) \wedge$ *INY-abstr1-set* $\omega\ \mathcal{C}\ u'\ v' \subseteq \omega' \wedge$
$\omega' \cap \mathcal{S}_{\mathcal{A}} = \{\}$

lemma *INY-abstr1-is-valid-ω'$\mathcal{C}'$I*:
assumes $\omega' \subseteq \mathcal{Q}\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}$ **and** $\omega \subseteq \omega'$ **and** $\mathcal{C}' = \mathcal{C} \cup (\omega'-\omega)$ **and**
INY-abstr1-set $\omega\ \mathcal{C}\ u'\ v' \subseteq \omega'$ **and** $\mathcal{S}_{\mathcal{A}} \cap \omega' = \{\}$
shows *INY-abstr1-is-valid-ω'$\mathcal{C}'$* $\omega\ \mathcal{C}\ u'\ v'\ (\omega',\mathcal{C}')$
using *assms* **unfolding** *INY-abstr1-is-valid-ω'$\mathcal{C}'$-def* **by** *auto*

lemma *INY-abstr1-is-valid-ω'$\mathcal{C}'$D*:
assumes *INY-abstr1-is-valid-ω'$\mathcal{C}'$* $\omega\ \mathcal{C}\ u'\ v'\ (\omega',\mathcal{C}')$
shows $\omega' \subseteq \mathcal{Q}\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}$ **and** $\omega \subseteq \omega'$ **and** $\mathcal{C}' = \mathcal{C} \cup (\omega'-\omega)$ **and**
INY-abstr1-set $\omega\ \mathcal{C}\ u'\ v' \subseteq \omega'$ **and** $\mathcal{S}_{\mathcal{A}} \cap \omega' = \{\}$
using *assms* **unfolding** *INY-abstr1-is-valid-ω'$\mathcal{C}'$-def* **by** *auto*

lemma *INY-abstr1-is-valid-ω'$\mathcal{C}'$D2*:
assumes *INY-abstr1-is-valid-ω'$\mathcal{C}'$* $\omega\ (\mathcal{C}-\{(u',v')\})\ u'\ v'\ (\omega',\mathcal{C}')$ **and**
I: *INY-abstr1-invar* $(\omega, \mathcal{C})$
shows $\omega \subseteq \omega'$ $\quad \mathcal{C}-\{(u',v')\} \subseteq \mathcal{C}'$ $\quad \mathcal{C}' = \mathcal{C} - \{(u',v')\} \cup (\omega'-\omega)$
$\omega' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$ $\quad \mathcal{C}' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$ $\quad \mathcal{C}' \subseteq \omega'$
INY-abstr1-set $\omega\ (\mathcal{C}-\{(u',v')\})\ u'\ v' \subseteq \omega'$ $\quad \mathcal{S}_{\mathcal{A}} \cap \omega' = \{\}$
using *INY-abstr1-invarD*[*OF I*] *INY-abstr1-is-valid-ω'$\mathcal{C}'$D*[*OF assms*(*1*)] **by** *auto*

The algorithm in its most abstract form: while there are unprocessed state pairs, we pick one pair at random and process it. The variables are as follows: - ω is the relation that shall be the complement of $\leq_{\mathcal{A}}$ in the end - $\mathcal{C}$ is the set of elements in ω that have not yet been processed - u' and v' are the two states being processed in one interation

definition *INY-abstr1* **where**
INY-abstr1 $\equiv$ *WHILE*$_T$$^{\textit{INY-abstr1-invar}}$ $(\lambda(\omega, \mathcal{C}).\ \mathcal{C}\neq\{\})$ $(\lambda(\omega, \mathcal{C}).$ *do* {
$(u',v') \leftarrow$ *SPEC* $(\lambda(u',v').\ (u',v') \in \mathcal{C})$;
let $\mathcal{C} = \mathcal{C} - \{(u',v')\}$;

```
    (ω, 𝒞) ← SPEC (INY-abstr1-is-valid-ω'𝒞' ω 𝒞 u' v');
    RETURN (ω, 𝒞)
}) (INY-initial, INY-initial)
```

the termination measure for the WHILE loop. The basic idea is that in each iteration, one pair (u', v') is processed, each pair is processed at most once and there is a finite number of pairs. $\omega - \mathcal{C}$ is the set of pairs that are known to be non-simulating and of whose non-simulatability has been propagated as well.

```
definition INY-abstr-measure::((('q×'q) set×('q×'q) set)) ⇒ nat
where INY-abstr-measure ≡ (λ(ω,𝒞). card (compl (ω−𝒞)))
```

The measure decreases in every iteration of the while loop.

```
lemma INY-measure-decreases:
  assumes INY-abstr1-invar (ω, 𝒞) and (u',v')∈𝒞
      and INY-abstr1-is-valid-ω'𝒞' ω (𝒞−{(u',v')}) u' v' (ω',𝒞')
  shows INY-abstr-measure (ω', 𝒞') < INY-abstr-measure (ω, 𝒞)
proof−
  let ?A = compl (ω−𝒞) and ?A' = compl (ω'−𝒞')
  note invar = INY-abstr1-invarD[OF assms(1)]
  note valid-ω'𝒞' = INY-abstr1-is-valid-ω'𝒞'D[OF assms(3)]
  have finite ?A using finite-𝒬 by blast
  moreover have ?A' ⊂ ?A using assms invar(1,3) valid-ω'𝒞'(1−3) by blast
  ultimately show ?thesis unfolding INY-abstr-measure-def
      by (simp add: psubset-card-mono)
qed
```

```
lemma INY-abstr1-invar5-preserved:
  assumes INY-abstr1-invar (ω, 𝒞) and
      INY-abstr1-is-valid-ω'𝒞' ω (𝒞−{(u',v')}) u' v' (ω',𝒞') and
      (u',v')∈𝒞 and uv-notin-ω': (u,v) ∈ compl ω' and (u,c,u'')∈Δ 𝒜
  shows ∃ v''. (v,c,v'')∈Δ 𝒜 ∧ (u'',v'') ∈ compl (ω'−(𝒞'−{(u',v')}))
proof−
  note invar = INY-abstr1-invarD[OF assms(1)]
  note valid-ω'𝒞' = INY-abstr1-is-valid-ω'𝒞'D2[OF assms(2,1)]
  let ?T = ω' − ω and ?T' = INY-abstr1-set ω (𝒞−{(u',v')}) u' v'
  have uv-properties: (u,v) ∉ ?T'    (u,v) ∈ compl ω
      using valid-ω'𝒞'(1,7) uv-notin-ω' by blast+
  then obtain v'' where
      v''-properties: (u'',v'') ∈ compl (ω − 𝒞) ∧ (v, c, v'')∈Δ 𝒜
      using invar(5) ‹(u,c,u'')∈Δ 𝒜› ‹(u',v')∈𝒞› by blast
    show ?thesis

    — case distinction: is the pair (u'',v'') we are looking at the
    — same as the one we are processing at the moment
    proof (cases u'=u'' ∧ v'=v'')
      case True
        — yes, (u'',v'')=(u',v'), therefore, we cannot use
```

— (u'',v'') as it will no longer be in $\omega - \mathcal{C}$ after
— this loop iteration.
hence $(u,c,u')\in\Delta\ \mathcal{A}$ **and** $(v,c,v')\in\Delta\ \mathcal{A}$
  using ‹$(u,c,u'')\in\Delta\ \mathcal{A}$› *v''-properties* **by** *blast+*
from *INY-abstr1-set-notmemE[OF uv-properties this]* **guess** v''' **.**
— this new v"' is definitely distinct from v'.
moreover with *invar(3)* ‹$(u',v')\in\mathcal{C}$› **have** $v''' \neq v'$ **by** *blast*
ultimately show *?thesis* **using** *True valid-ω'𝒞'(3)* **by** *blast*

next
 case *False*
 — no, $(u'',v'') \neq (u', v')$ therefore we can just use
 — (u'',v'') itself.
 thus *?thesis* **using** *v''-properties valid-ω'𝒞'(3)* **by** *blast*
qed
qed

The while loop invariant is not violated by a loop iteration

lemma *INY-abstr1-invar-preserved*:
 assumes *INY-abstr1-invar* $(\omega, \mathcal{C})$ **and** $(u',v')\in\mathcal{C}$ **and**
  INY-abstr1-is-valid-ω'𝒞' ω $(\mathcal{C}-\{(u',v')\})$ u' v' $(\omega',\mathcal{C}')$
 shows *INY-abstr1-invar* $(\omega', \mathcal{C}')$
apply (*intro INY-abstr1-invarI*)
using *INY-abstr1-invarD(2)[OF assms(1)] INY-abstr1-is-valid-ω'𝒞'D2[OF assms(3,1)]*
apply (*blast, fast, blast*)
using *INY-abstr1-is-valid-ω'𝒞'D2(4,8)[OF assms(3,1)]* **apply** *blast*
using *INY-abstr1-invar5-preserved[OF assms(1,3,2)]* **apply** *blast*
done

If the invariant still holds and we have processed all elements (i.e. $\mathcal{C}$ is empty), ω is now the complement of the simulation preorder.

lemma *INY-abstr1-invar-imp-goal*:
 assumes *INY-abstr1-invar* $(\omega, \{\})$ **shows** *compl* $\omega = \mathcal{S}_\mathcal{A}$
proof−
 note *invar = INY-abstr1-invar-emptyD[OF assms(1)]*
 have *is-largest-sim (compl ω)*
  apply (*intro is-largest-simI is-simI*)
  using *invar(2,4)* **apply** (*fast, fast, fast*)
  apply (*subgoal-tac* $\bigwedge\mathcal{S}$. *is-sim* $\mathcal{S} \Longrightarrow \mathcal{S} \cap \omega = \{\}$)
  using *compl-subseteq-reverse[OF invar(1)]* **apply** *blast*
  using *invar(3)* $\mathcal{S}_\mathcal{A}$*-is-largest-sim* **apply** *blast*
  done
 thus *compl* $\omega = \mathcal{S}_\mathcal{A}$ **using** *is-largest-sim-unique* **by** *simp*
qed

The abstract algorithm is correct.

theorem *INY-abstr1-correct*:
  shows *INY-abstr1* $\leq$ *SPEC* $(\lambda(\omega,\text{-}).\ \textit{compl}\ \omega = \mathcal{S}_\mathcal{A})$
  unfolding *INY-abstr1-def*

apply (*intro refine-vcg*)
apply (*rule wf-measure*[*of INY-abstr-measure*])
apply (*fact INY-abstr1-invar-initial*)
apply (*simp-all add*: *INY-abstr1-invar-preserved INY-measure-decreases*)[*2*]
apply (*clarsimp simp add*: *INY-abstr1-invar-imp-goal*)
done

The set to be added to ω in one iteration of the first for each loop.

definition *INY-abstr2-set* **where**
$INY\text{-}abstr2\text{-}set\ \omega\ \mathcal{C}\ u'\ v'\ c \equiv$
$\{(u,v)\ |u\ v.\ (u,v)\notin\omega \wedge (u,c,u')\in\Delta\ \mathcal{A} \wedge (v,c,v') \in \Delta\ \mathcal{A} \wedge$
$(\forall v''.\ (v,c,v'')\in\Delta\ \mathcal{A} \longrightarrow (u',v'') \in\omega-\mathcal{C})\}$

The conditions the new values for ω and $\mathcal{C}$ need to fulfil in order for the first for each loop to work.

definition *INY-abstr2-is-valid-ω'C'* **where**
$INY\text{-}abstr2\text{-}is\text{-}valid\text{-}\omega'\mathcal{C}'\ \omega\ \mathcal{C}\ u'\ v'\ c \equiv \lambda(\omega',\mathcal{C}').\ \omega'\subseteq\mathcal{Q}\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A} \wedge$
$\omega'=\omega\cup(\omega'-\omega) \wedge \mathcal{C}'=\mathcal{C}\cup(\omega'-\omega) \wedge INY\text{-}abstr2\text{-}set\ \omega\ \mathcal{C}\ u'\ v'\ c \subseteq \omega' \wedge \mathcal{S}_{\mathcal{A}} \cap \omega'$
$= \{\}$

lemma *INY-abstr2-is-valid-ω'C'I*:
assumes $\omega'=\omega\cup(\omega'-\omega)$ $\mathcal{C}'=\mathcal{C}\cup(\omega'-\omega)$ $INY\text{-}abstr2\text{-}set\ \omega\ \mathcal{C}\ u'\ v'\ c \subseteq \omega'$
$\omega'\subseteq\mathcal{Q}\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}$ $\mathcal{S}_{\mathcal{A}} \cap \omega' = \{\}$
shows $INY\text{-}abstr2\text{-}is\text{-}valid\text{-}\omega'\mathcal{C}'\ \omega\ \mathcal{C}\ u'\ v'\ c\ (\omega',\mathcal{C}')$
unfolding *INY-abstr2-is-valid-ω'C'-def* **using** *assms* **by** *blast*

lemma *INY-abstr2-is-valid-ω'C'D*:
assumes $INY\text{-}abstr2\text{-}is\text{-}valid\text{-}\omega'\mathcal{C}'\ \omega\ \mathcal{C}\ u'\ v'\ c\ (\omega',\mathcal{C}')$
shows $\omega'=\omega\cup(\omega'-\omega)$ $\mathcal{C}'=\mathcal{C}\cup(\omega'-\omega)$ $INY\text{-}abstr2\text{-}set\ \omega\ \mathcal{C}\ u'\ v'\ c \subseteq \omega'$
$\omega' \subseteq \mathcal{Q}\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}$ $\mathcal{S}_{\mathcal{A}} \cap \omega' = \{\}$
using *assms* **unfolding** *INY-abstr2-is-valid-ω'C'-def* **by** *blast+*

definition *INY-abstr2-loopc-invar* **where**
$INY\text{-}abstr2\text{-}loopc\text{-}invar\ \omega\ \mathcal{C}\ u'\ v'\ \Sigma' \equiv \lambda(\omega',\ \mathcal{C}').$
$let\ T = \omega'-\omega\ in\ \omega' = \omega \cup T \wedge \mathcal{C}' = \mathcal{C} \cup T \wedge \mathcal{C} \subseteq \omega \wedge$
$\omega' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A} \wedge \mathcal{C}' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A} \wedge$
$(\bigcup c\in(\Sigma\ \mathcal{A}-\Sigma').\ INY\text{-}abstr2\text{-}set\ \omega\ \mathcal{C}\ u'\ v'\ c) \subseteq \omega' \wedge$
$\mathcal{S}_{\mathcal{A}} \cap \omega' = \{\} \wedge (u',v') \in \omega \wedge (u',v') \notin \mathcal{C}'$

lemma *INY-abstr2-loopc-invarI*[*intro*]: **fixes** $\omega\ \omega'\ \mathcal{C}\ \mathcal{C}'\ \Sigma'\ u'\ v'$
assumes $\omega' = \omega \cup (\omega'-\omega)$ $\mathcal{C}' = \mathcal{C} \cup (\omega'-\omega)$ $\mathcal{C} \subseteq \omega$
$\omega' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$ $\mathcal{C}' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$
$(\bigcup c\in\Sigma\ \mathcal{A}-\Sigma'.\ INY\text{-}abstr2\text{-}set\ \omega\ \mathcal{C}\ u'\ v'\ c) \subseteq \omega'$ $\mathcal{S}_{\mathcal{A}} \cap \omega' = \{\}$
$(u',v') \in \omega$ $(u',v') \notin \mathcal{C}'$
shows $INY\text{-}abstr2\text{-}loopc\text{-}invar\ \omega\ \mathcal{C}\ u'\ v'\ \Sigma'\ (\omega',\ \mathcal{C}')$
unfolding *INY-abstr2-loopc-invar-def* **using** *assms* **by** (*simp-all add*: *Let-def*)

lemma *INY-abstr2-loopc-invarD*[*dest*]: **fixes** $\omega\ \omega'\ \mathcal{C}\ \mathcal{C}'\ \Sigma'\ u'\ v'$
assumes $INY\text{-}abstr2\text{-}loopc\text{-}invar\ \omega\ \mathcal{C}\ u'\ v'\ \Sigma'\ (\omega',\ \mathcal{C}')$

shows $\omega' = \omega \cup (\omega' - \omega)$ $\quad \mathcal{C}' = \mathcal{C} \cup (\omega' - \omega)$ $\quad \mathcal{C} \subseteq \omega$
$\omega' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$ $\quad \mathcal{C}' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$
$(\bigcup c{\in}\Sigma\ \mathcal{A}{-}\Sigma'.\ \textit{INY-abstr2-set}\ \omega\ \mathcal{C}\ u'\ v'\ c) \subseteq \omega'$
$\mathcal{S}_{\mathcal{A}} \cap \omega' = \{\}$ $\quad (u',v') \in \omega$ $\quad (u',v') \notin \mathcal{C}'$
using *assms* **unfolding** *INY-abstr2-loopc-invar-def* **by** (*auto simp add*: *Let-def*)

The for each loop that iterates over all $c \in \Sigma$.

definition *INY-abstr2-loopc* **where**
INY-abstr2-loopc $\omega\ \mathcal{C}\ u'\ v' \equiv$ *FOREACH*$^{\textit{INY-abstr2-loopc-invar}\ \omega\ \mathcal{C}\ u'\ v'}$
$(\Sigma\ \mathcal{A})\ (\lambda c\ (\omega, \mathcal{C}).\ do\ \{$
$(\omega', \mathcal{C}') \leftarrow$ *SPEC* (*INY-abstr2-is-valid-ω'C'* $\omega\ \mathcal{C}\ u'\ v'\ c$);
RETURN $(\omega', \mathcal{C}')$
$\})\ (\omega, \mathcal{C})$

The first for each loop works correctly, i.e. it returns valid values for ω and $\mathcal{C}$.

lemma *INY-abstr2-loopc-correct*: **fixes** $\omega\ \mathcal{C}\ u'\ v'$
assumes *I*: *INY-abstr1-invar* $(\omega,\mathcal{C} \cup \{(u',v')\})$ **and** $(u',v') \notin \mathcal{C}$
shows *INY-abstr2-loopc* $\omega\ \mathcal{C}\ u'\ v' \leq$ *SPEC* (*INY-abstr1-is-valid-ω'C'* $\omega\ \mathcal{C}\ u'\ v'$)
unfolding *INY-abstr2-loopc-def*
proof (*rule FOREACHi-rule*)
show *finite* $(\Sigma\ \mathcal{A})$ **using** *finite-Σ* **.**
next
note *invar-outer* = *INY-abstr1-invarD*[*OF I*]
show *INY-abstr2-loopc-invar* $\omega\ \mathcal{C}\ u'\ v'\ (\Sigma\ \mathcal{A})\ (\omega, \mathcal{C})$
using *invar-outer assms*(*2*) **by** (*intro INY-abstr2-loopc-invarI*, *blast+*)
next
case (*goal3 c* Σ')
thus *?case*
proof (*intro refine-vcg*, *clarify*)
case (*goal1* $\omega'\ \mathcal{C}'$ - - $\omega''\ \mathcal{C}''$)
note *invar* = *INY-abstr2-loopc-invarD*[*OF goal1*(*3*)]
note *valid* = *INY-abstr2-is-valid-ω'C'D*[*OF goal1*(*4*)]
let $?\Sigma'' = \Sigma' - \{c\}$
show *INY-abstr2-loopc-invar* $\omega\ \mathcal{C}\ u'\ v'\ ?\Sigma''\ (\omega'', \mathcal{C}'')$
proof (*intro INY-abstr2-loopc-invarI*)
have $\omega \cup$ *INY-abstr2-set* $\omega\ \mathcal{C}\ u'\ v'\ c \subseteq \omega' \cup$ *INY-abstr2-set* $\omega'\ \mathcal{C}'\ u'\ v'\ c$
unfolding *INY-abstr2-set-def* **using** *invar*(*1*−*2*) **by** *blast*
moreover have *INY-abstr2-set* $\omega\ \mathcal{C}\ u'\ v'\ c \cap \omega = \{\}$
unfolding *INY-abstr2-set-def* **by** *blast*
ultimately show $(\bigcup c{\in}\Sigma\ \mathcal{A} - ?\Sigma''.\ \textit{INY-abstr2-set}\ \omega\ \mathcal{C}\ u'\ v'\ c) \subseteq$
ω'' **using** *invar*(*6*) *valid*(*1*,*3*) **by** *blast*
next
case *goal5* **thus** *?case* **using** *invar*(*4*,*5*) *valid*(*1*,*2*,*4*) **by** *blast*
next
from *INY-abstr1-invarD*(*3*)[*OF I*] **show** $(u',v') \in \omega$ **by** *simp*
next
from *invar valid*(*2*) **show** $(u',v') \notin \mathcal{C}''$ **by** *blast*
qed (*insert assms*(*2*) *invar*(*1*−*3*) *valid*(*1*,*2*,*4*,*5*), *auto*)

```
  qed

next
  case (goal4 ω'C') thus ?case
    proof (cases ω'C', simp)
      fix ω' C' assume I: INY-abstr2-loopc-invar ω C u' v' {} (ω', C')
      note invar = INY-abstr2-loopc-invarD[OF I]
      have (⋃ c∈Σ A. INY-abstr2-set ω C u' v' c) = INY-abstr1-set ω C u' v'
          unfolding INY-abstr2-set-def INY-abstr1-set-def
          using Δ-consistent by blast
     thus INY-abstr1-is-valid-ω'C' ω C u' v' (ω', C')
         apply (intro INY-abstr1-is-valid-ω'C'I)
         using invar apply (blast, blast, blast, blast, simp)
         done
    qed
qed

lemma INY-abstr2-loopc-correct':
  assumes INY-abstr1-invar (ω, C) and (u',v') ∈ C
  shows   INY-abstr2-loopc ω (C − {(u',v')}) u' v' ≤
            SPEC (INY-abstr1-is-valid-ω'C' ω (C − {(u',v')}) u' v')
proof−
  from assms have C − {(u',v')} ∪ {(u',v')} = C by blast
  with INY-abstr2-loopc-correct assms show ?thesis by simp
qed
```

The entire algorithm, now with a more concrete implementation of the computation of new ω and $\mathcal{C}$. Since all steps from here on are analogous to this, they will not be commented any further.

```
definition INY-abstr2 where
INY-abstr2 ≡ WHILE_T^{INY-abstr1-invar} (λ(ω, C). C≠{}) (λ(ω, C). do {
    (u',v') ← SPEC (λ(u', v'). (u',v') ∈ C);
    let C = C − {(u',v')};
    (ω, C) ← (INY-abstr2-loopc ω C u' v');
    RETURN (ω, C)
}) (INY-initial, INY-initial)

lemma INY-abstr2-correct: INY-abstr2 ≤ ⇓Id INY-abstr1
  unfolding INY-abstr2-def INY-abstr1-def
  by (refine-rcg, simp-all add: INY-abstr2-loopc-correct')

definition INY-abstr3-set where
INY-abstr3-set ω C u' v' c v ≡
    {(u,v) |u. (u,v)∉ω ∧ (u,c,u') ∈ Δ A ∧ (v,c,v') ∈ Δ A ∧
                (∀ v''. (v,c,v'')∈Δ A ⟶ (u',v'')∈ω−C)}

lemma INY-abstr3-set-simp:
    (v,c,v')∈Δ A ⟹ (if (∀ v''. (v,c,v'')∈Δ A ⟶ (u',v'')∈ω−C) then
    {(u,v) |u. (u,v)∉ω ∧ (u,c,u') ∈ Δ A} else {}) =
```

$INY\text{-}abstr3\text{-}set\ \omega\ \mathcal{C}\ u'\ v'\ c\ v$
unfolding *INY-abstr3-set-def* **by** *simp*

definition *INY-abstr3-is-valid-ω′C′* **where**
$INY\text{-}abstr3\text{-}is\text{-}valid\text{-}\omega'\mathcal{C}'\ \omega\ \mathcal{C}\ u'\ v'\ c\ v \equiv \lambda(\omega',\mathcal{C}').\ \omega' \subseteq \mathcal{Q}\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}\ \wedge$
$\omega'=\omega\cup(\omega'-\omega)\ \wedge\ \mathcal{C}'=\mathcal{C}\cup(\omega'-\omega)\ \wedge\ INY\text{-}abstr3\text{-}set\ \omega\ \mathcal{C}\ u'\ v'\ c\ v \subseteq \omega'\ \wedge\ \mathcal{S}_{\mathcal{A}} \cap \omega' = \{\}$

lemma *INY-abstr3-is-valid-ω′C′I*:
assumes $\omega'=\omega\cup(\omega'-\omega)$ $\quad \mathcal{C}'=\mathcal{C}\cup(\omega'-\omega)$ $\quad INY\text{-}abstr3\text{-}set\ \omega\ \mathcal{C}\ u'\ v'\ c\ v \subseteq \omega'$
$\omega' \subseteq \mathcal{Q}\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}$ $\quad \mathcal{S}_{\mathcal{A}} \cap \omega' = \{\}$
shows $INY\text{-}abstr3\text{-}is\text{-}valid\text{-}\omega'\mathcal{C}'\ \omega\ \mathcal{C}\ u'\ v'\ c\ v\ (\omega',\mathcal{C}')$
unfolding *INY-abstr3-is-valid-ω′C′-def* **using** *assms* **by** *blast*

lemma *INY-abstr3-is-valid-ω′C′D*:
assumes $INY\text{-}abstr3\text{-}is\text{-}valid\text{-}\omega'\mathcal{C}'\ \omega\ \mathcal{C}\ u'\ v'\ c\ v\ (\omega',\mathcal{C}')$
shows $\omega'=\omega\cup(\omega'-\omega)$ $\quad \mathcal{C}'=\mathcal{C}\cup(\omega'-\omega)$ $\quad INY\text{-}abstr3\text{-}set\ \omega\ \mathcal{C}\ u'\ v'\ c\ v \subseteq \omega'$
$\omega' \subseteq \mathcal{Q}\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}$ $\quad \mathcal{S}_{\mathcal{A}} \cap \omega' = \{\}$
using *assms* **unfolding** *INY-abstr3-is-valid-ω′C′-def* **by** *blast+*

definition *INY-abstr3-loopv-invar* **where**
$INY\text{-}abstr3\text{-}loopv\text{-}invar\ \omega\ \mathcal{C}\ u'\ v'\ c\ V' \equiv \lambda(\omega',\ \mathcal{C}').$
$let\ T = \omega'-\omega\ in\ \omega' = \omega \cup T \wedge \mathcal{C}' = \mathcal{C} \cup T \wedge \mathcal{C} \subseteq \omega\ \wedge$
$\omega' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A} \wedge \mathcal{C}' \subseteq \mathcal{Q}\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}\ \wedge$
$(\bigcup v\in(\{v.\ (v,c,v')\in\Delta\ \mathcal{A}\}-V').\ INY\text{-}abstr3\text{-}set\ \omega\ \mathcal{C}\ u'\ v'\ c\ v) \subseteq \omega'\ \wedge$
$\mathcal{S}_{\mathcal{A}} \cap \omega' = \{\} \wedge (u',v') \in \omega \wedge (u',v') \notin \mathcal{C}'$

lemma *INY-abstr3-loopv-invarI*[*intro*]:
assumes $\omega' = \omega \cup (\omega'-\omega)$ $\quad \mathcal{C}' = \mathcal{C} \cup (\omega'-\omega)$ $\quad \mathcal{C} \subseteq \omega$
$\omega'\subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$ $\quad \mathcal{C}' \subseteq \mathcal{Q}\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}$
$(\bigcup v\in\{v.\ (v,c,v')\in\Delta\ \mathcal{A}\}-V'.\ INY\text{-}abstr3\text{-}set\ \omega\ \mathcal{C}\ u'\ v'\ c\ v) \subseteq \omega'$
$\mathcal{S}_{\mathcal{A}} \cap \omega' = \{\}$ $\quad (u',v') \in \omega$ $\quad (u',v') \notin \mathcal{C}'$
shows $INY\text{-}abstr3\text{-}loopv\text{-}invar\ \omega\ \mathcal{C}\ u'\ v'\ c\ V'\ (\omega',\ \mathcal{C}')$
using *assms* **unfolding** *INY-abstr3-loopv-invar-def* **by** (*auto simp add*: *Let-def*)

lemma *INY-abstr3-loopv-invarD*[*dest*]: **fixes** $\omega\ \omega'\ \mathcal{C}\ \mathcal{C}'\ \Sigma'\ u'\ v'\ c$
assumes $INY\text{-}abstr3\text{-}loopv\text{-}invar\ \omega\ \mathcal{C}\ u'\ v'\ c\ V'\ (\omega',\ \mathcal{C}')$
shows $\omega' = \omega \cup (\omega'-\omega)$ $\quad \mathcal{C}' = \mathcal{C} \cup (\omega'-\omega)$ **and** $\mathcal{C} \subseteq \omega$
$\omega' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$ **and** $\mathcal{C}' \subseteq \mathcal{Q}\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}$ **and**
$(\bigcup v\in\{v.\ (v,c,v')\in\Delta\ \mathcal{A}\}-V'.\ INY\text{-}abstr3\text{-}set\ \omega\ \mathcal{C}\ u'\ v'\ c\ v) \subseteq \omega'$ **and**
$\mathcal{S}_{\mathcal{A}} \cap \omega' = \{\}$ $\quad (u',v') \in \omega$ $\quad (u',v') \notin \mathcal{C}'$
using *assms* **unfolding** *INY-abstr3-loopv-invar-def* **by** (*auto simp add*: *Let-def*)

definition *INY-abstr3-loopv* **where**
$INY\text{-}abstr3\text{-}loopv\ \omega\ \mathcal{C}\ u'\ v'\ c \equiv$
$FOREACH^{INY\text{-}abstr3\text{-}loopv\text{-}invar\ \omega\ \mathcal{C}\ u'\ v'\ c}$
$\{v.\ (v,c,v') \in \Delta\ \mathcal{A}\}\ (\lambda v\ (\omega,\ \mathcal{C}).$
$if\ (\forall v''.\ (v,c,v'')\in\Delta\ \mathcal{A} \longrightarrow (u',v'')\in\omega-\mathcal{C})\ then\ do\ \{$
$(\omega',\ \mathcal{C}') \leftarrow SPEC\ (INY\text{-}abstr3\text{-}is\text{-}valid\text{-}\omega'\mathcal{C}'\ \omega\ \mathcal{C}\ u'\ v'\ c\ v);$

```
      RETURN (ω′, C′)
    } else
      RETURN (ω, C)
  ) (ω, C)

lemma INY-abstr3-loopv-correct:
  assumes I1: INY-abstr2-loopc-invar ω C u′ v′ Σ′ (ω′,C′)
  shows INY-abstr3-loopv ω′ C′ u′ v′ c ≤
      SPEC (INY-abstr2-is-valid-ω′C′ ω′ C′ u′ v′ c)
unfolding INY-abstr3-loopv-def
proof (rule FOREACHi-rule)
  have {v. (v,c,v′)∈Δ A} ⊆ Q A using Δ-consistent by blast
  thus finite {v. (v,c,v′)∈Δ A} using rev-finite-subset[OF finite-Q] by simp
next
  note invar-loopc = INY-abstr2-loopc-invarD[OF I1]
  show INY-abstr3-loopv-invar ω′ C′ u′ v′ c {v. (v, c, v′) ∈ Δ A} (ω′, C′)
    apply (rule INY-abstr3-loopv-invarI)
    using invar-loopc apply (blast, blast, blast, blast, blast, blast)
    using invar-loopc(7) apply blast
    using invar-loopc apply (blast, blast)
    done
next
  case (goal3 v V′)
  hence v′-succ-v: (v,c,v′) ∈ Δ A by blast
  show ?case using goal3(3)
  proof (intro refine-vcg, clarsimp)
    case (goal1 ω′′ C′′ ω′′′ C′′′)
    note invar = INY-abstr3-loopv-invarD[OF goal1(1)]
    note valid = INY-abstr3-is-valid-ω′C′D[OF goal1(3)]
    let ?V′′ = V′ − {v}
    show INY-abstr3-loopv-invar ω′ C′ u′ v′ c ?V′′ (ω′′′, C′′′)
    proof (intro INY-abstr3-loopv-invarI)
      have ω′∪INY-abstr3-set ω′ C′ u′ v′ c v ⊆
          ω′′∪INY-abstr3-set ω′′ C′′ u′ v′ c v
          unfolding INY-abstr3-set-def using invar(1,2) by blast
      moreover have INY-abstr3-set ω′ C′ u′ v′ c v ∩ ω′ = {}
          unfolding INY-abstr3-set-def by blast
      ultimately show (⋃v∈{v.(v,c,v′)∈Δ A}−?V′′.
          INY-abstr3-set ω′ C′ u′ v′ c v) ⊆ ω′′′
         using invar(6) valid(1,3) by blast
    next
      case goal5 thus ?case using invar(4,5) valid(1,2,4) by blast
    next
      from invar(8) show (u′, v′) ∈ ω′ .
    next
      from invar valid(2) show (u′, v′) ∉ C′′′ by blast
    qed (insert invar(1−3) valid(1,2,4,5), fast, fast, force, force)
  next
    case (goal2 ω′′ C′′)
```

```
    have INY-abstr3-loopv-invar ω' C' u' v' c V' (ω'', C'')
      using goal2(1,2) by simp
    note invar = INY-abstr3-loopv-invarD[OF this]
    let ?V'' = V' − {v}
    from goal2(3) have A: ¬ (∀ v''. (v, c, v'') ∈ Δ A ⟶ (u', v'') ∈ ω'−C')
     using invar(1,2) by blast
    have B: INY-abstr3-set ω' C' u' v' c v = {} using A
      by (subst INY-abstr3-set-simp[OF v'-succ-v, of u' ω' C', symmetric],
         auto)
    thus INY-abstr3-loopv-invar ω' C' u' v' c ?V'' (ω'', C'')
     apply (intro INY-abstr3-loopv-invarI)
     using invar(1−5) apply (fast, fast, force, force, force)
     using invar(6) B apply blast
     using invar(7) apply force
     using invar(8,9) apply (simp, simp)
     done
  qed

next
  case (goal4 ω''C'') thus ?case
    proof (cases ω''C'', simp)
     fix ω'' C''
     assume I: INY-abstr3-loopv-invar ω' C' u' v' c {} (ω'', C'')
     note invar = INY-abstr3-loopv-invarD[OF I]
     note invar-loopc = INY-abstr2-loopc-invarD[OF I1]
     have (⋃ v∈{v. (v,c,v')∈Δ A}. INY-abstr3-set ω' C' u' v' c v) =
        INY-abstr2-set ω' C' u' v' c
        unfolding INY-abstr3-set-def INY-abstr2-set-def by blast
     thus INY-abstr2-is-valid-ω'C' ω' C' u' v' c (ω'', C'')
       using invar by(intro INY-abstr2-is-valid-ω'C'I, simp-all)
    qed
qed


definition INY-abstr3-loopc where
INY-abstr3-loopc ω C u' v' ≡ FOREACH^{INY-abstr2-loopc-invar ω C u' v'}
    (Σ A) (λc (ω, C). do {
      (ω', C') ← INY-abstr3-loopv ω C u' v' c;
      RETURN (ω', C')
    }) (ω, C)

definition INY-abstr3 where
INY-abstr3 ≡ WHILE_T^{INY-abstr1-invar} (λ(ω, C). C≠{}) (λ(ω, C). do {
    (u',v') ← SPEC (λ(u', v'). (u',v') ∈ C);
    let C = C − {(u',v')};
    (ω, C) ← (INY-abstr3-loopc ω C u' v');
    RETURN (ω, C)
}) (INY-initial, INY-initial)
```

lemma *INY-abstr3-loopc-correct*:
assumes $(\omega_1', \omega_2') \in Id$ **and** $(\mathcal{C}_1', \mathcal{C}_2') \in Id$
and $(u_1', u_2') \in Id$ **and** $(v_1', v_2') \in Id$
shows *INY-abstr3-loopc* $\omega_1'\ \mathcal{C}_1'\ u_1'\ v_1' \leq \Downarrow Id$ (*INY-abstr2-loopc* $\omega_2'\ \mathcal{C}_2'\ u_2'\ v_2'$)
unfolding *INY-abstr3-loopc-def INY-abstr2-loopc-def*
using *assms* **by** (*refine-rcg inj-on-id*, *simp-all add*: *INY-abstr3-loopv-correct*)

lemma *INY-abstr3-correct*: *INY-abstr3* $\leq \Downarrow Id$ *INY-abstr2*
unfolding *INY-abstr3-def INY-abstr2-def*
by (*refine-rcg INY-abstr3-loopc-correct*, *simp-all*)

definition *INY-abstr4-set* **where**
INY-abstr4-set $\omega\ \mathcal{C}\ u'\ v'\ c\ v\ u \equiv$ (*if* $(u,v) \notin \omega$ *then* $\{(u,v)\}$ *else* $\{\}$)

definition *INY-abstr4-is-valid-*$\omega'\mathcal{C}'$ **where**
INY-abstr4-is-valid-$\omega'\mathcal{C}'\ \omega\ \mathcal{C}\ u'\ v'\ c\ v\ u \equiv \lambda(\omega', \mathcal{C}').\ \omega' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}\ \wedge$
$\omega' = \omega \cup (\omega' - \omega) \wedge \mathcal{C}' = \mathcal{C} \cup (\omega' - \omega) \wedge$ *INY-abstr4-set* $\omega\ \mathcal{C}\ u'\ v'\ c\ v\ u \subseteq \omega' \wedge$
$\mathcal{S}_{\mathcal{A}} \cap \omega' = \{\}$

lemma *INY-abstr4-is-valid-*$\omega'\mathcal{C}'I$:
assumes $\omega' = \omega \cup (\omega' - \omega)$ $\quad \mathcal{C}' = \mathcal{C} \cup (\omega' - \omega)$ $\quad$ *INY-abstr4-set* $\omega\ \mathcal{C}\ u'\ v'\ c\ v\ u \subseteq \omega'$
$\omega' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$ $\quad \mathcal{S}_{\mathcal{A}} \cap \omega' = \{\}$
shows *INY-abstr4-is-valid-*$\omega'\mathcal{C}'\ \omega\ \mathcal{C}\ u'\ v'\ c\ v\ u\ (\omega', \mathcal{C}')$
unfolding *INY-abstr4-is-valid-*$\omega'\mathcal{C}'$*-def* **using** *assms* **by** *blast*

lemma *INY-abstr4-is-valid-*$\omega'\mathcal{C}'E$:
assumes *INY-abstr4-is-valid-*$\omega'\mathcal{C}'\ \omega\ \mathcal{C}\ u'\ v'\ c\ v\ u\ (\omega', \mathcal{C}')$
shows $\omega' = \omega \cup (\omega' - \omega)$ $\quad \mathcal{C}' = \mathcal{C} \cup (\omega' - \omega)$ $\quad$ *INY-abstr4-set* $\omega\ \mathcal{C}\ u'\ v'\ c\ v\ u \subseteq \omega'$
$\omega' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$ $\quad \mathcal{S}_{\mathcal{A}} \cap \omega' = \{\}$
using *assms* **unfolding** *INY-abstr4-is-valid-*$\omega'\mathcal{C}'$*-def* **by** *blast+*

definition *INY-abstr4-loopu-invar* **where**
INY-abstr4-loopu-invar $\omega\ \mathcal{C}\ u'\ v'\ c\ v\ U' \equiv \lambda(\omega', \mathcal{C}')$.
let $T = \omega' - \omega$ *in* $\omega' = \omega \cup T \wedge \mathcal{C}' = \mathcal{C} \cup T \wedge \mathcal{C} \subseteq \omega\ \wedge$
$\omega' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A} \wedge \mathcal{C}' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}\ \wedge$
$(\bigcup u \in (\{u.\ (u,c,u') \in \Delta\ \mathcal{A}\} - U')$. *INY-abstr4-set* $\omega\ \mathcal{C}\ u'\ v'\ c\ v\ u) \subseteq \omega'\ \wedge$
$\mathcal{S}_{\mathcal{A}} \cap \omega' = \{\} \wedge (u', v') \in \omega \wedge (u', v') \notin \mathcal{C}'$

lemma *INY-abstr4-loopu-invarI*[*intro*]:
assumes $\omega' = \omega \cup \omega'$ $\quad \mathcal{C}' = \mathcal{C} \cup (\omega' - \omega)$ $\quad \mathcal{C} \subseteq \omega$
$\omega' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$ $\quad \mathcal{C}' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$
$(\bigcup u \in (\{u.\ (u,c,u') \in \Delta\ \mathcal{A}\} - U')$. *INY-abstr4-set* $\omega\ \mathcal{C}\ u'\ v'\ c\ v\ u) \subseteq \omega'$
$\mathcal{S}_{\mathcal{A}} \cap \omega' = \{\}$ $\quad (u', v') \in \omega$ $\quad (u', v') \notin \mathcal{C}'$
shows *INY-abstr4-loopu-invar* $\omega\ \mathcal{C}\ u'\ v'\ c\ v\ U'\ (\omega', \mathcal{C}')$
using *assms* **unfolding** *INY-abstr4-loopu-invar-def* **by** (*auto simp add*: *Let-def*)

lemma *INY-abstr4-loopu-invarD*[*dest*]:
assumes *INY-abstr4-loopu-invar* $\omega\ \mathcal{C}\ u'\ v'\ c\ v\ U'\ (\omega', \mathcal{C}')$

shows $\omega' = \omega \cup \omega' \quad \mathcal{C}' = \mathcal{C} \cup (\omega' - \omega) \quad \mathcal{C} \subseteq \omega$
$\omega' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A} \quad \mathcal{C}' \subseteq \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$
$(\bigcup u \in (\{u.\ (u,c,u') \in \Delta\ \mathcal{A}\} - U').\ \textit{INY-abstr4-set}\ \omega\ \mathcal{C}\ u'\ v'\ c\ v\ u) \subseteq \omega'$
$\mathcal{S}_{\mathcal{A}} \cap \omega' = \{\} \quad (u',v') \in \omega \quad (u',v') \notin \mathcal{C}$
using *assms* **unfolding** *INY-abstr4-loopu-invar-def* **by** (*auto simp add*: *Let-def*)

definition *INY-abstr4-loopu* **where**
INY-abstr4-loopu $\omega\ \mathcal{C}\ u'\ v'\ c\ v \equiv$
$\textit{FOREACH}^{\textit{INY-abstr4-loopu-invar}\ \omega\ \mathcal{C}\ u'\ v'\ c\ v}$
$\{u.\ (u,c,u') \in \Delta\ \mathcal{A}\}\ (\lambda u\ (\omega, \mathcal{C}).$
if $(u,v) \notin \omega$ *then do* {
ASSERT $((u,v) \notin \omega)$;
ASSERT $((u,v) \notin \mathcal{C})$;
RETURN (*insert* $(u,v)\ \omega$, *insert* $(u,v)\ \mathcal{C}$)
} *else*
RETURN $(\omega, \mathcal{C})$
) $(\omega, \mathcal{C})$

lemma *INY-abstr4-loopu-correct*:
assumes *I1*: *INY-abstr3-loopv-invar* $\omega\ \mathcal{C}\ u'\ v'\ c\ V'\ (\omega',\mathcal{C}')$ **and**
v'-succ-v: $(v,c,v') \in \Delta\ \mathcal{A}$ **and**
v'-properties: $\forall v''.\ (v,c,v'') \in \Delta\ \mathcal{A} \longrightarrow (u',v'') \in \omega' - \mathcal{C}' \cup \{(u',v')\}$
shows *INY-abstr4-loopu* $\omega'\ \mathcal{C}'\ u'\ v'\ c\ v \leq$
SPEC (*INY-abstr3-is-valid-ω'C'* $\omega'\ \mathcal{C}'\ u'\ v'\ c\ v$)
unfolding *INY-abstr4-loopu-def*
proof (*rule FOREACHi-rule*)
have $\{u.\ (u,c,u') \in \Delta\ \mathcal{A}\} \subseteq \mathcal{Q}\ \mathcal{A}$ **using** Δ*-consistent* **by** *blast*
thus *finite* $\{u.\ (u,c,u') \in \Delta\ \mathcal{A}\}$ **using** *rev-finite-subset*[*OF finite-*$\mathcal{Q}$] **by** *simp*
next
note *invar-loopv* = *INY-abstr3-loopv-invarD*[*OF I1*]
show *INY-abstr4-loopu-invar* $\omega'\ \mathcal{C}'\ u'\ v'\ c\ v\ \{u.\ (u,c,u') \in \Delta\ \mathcal{A}\}\ (\omega', \mathcal{C}')$
using *invar-loopv* **by** *auto*
next
case (*goal3 u U'*)
hence *u'-succ-u*: $(u,c,u') \in \Delta\ \mathcal{A}$ **by** *blast*
show *?case* **using** *goal3*(*3*)
apply (*intro refine-vcg*)
apply (*unfold INY-abstr4-loopu-invar-def*, *auto*) [*2*]
apply *clarify*
proof −
case (*goal1* $\omega''\ \mathcal{C}''$)
note *invar* = *INY-abstr4-loopu-invarD*[*OF goal1*(*1*)]
note *uv-notin-ω''* = *goal1*(*2*)
let $?U'' = U' - \{u\}$ **and** $?\omega''' = \textit{insert}\ (u,\ v)\ \omega''$ **and**
$?\mathcal{C}''' = \textit{insert}\ (u,\ v)\ \mathcal{C}''$
show *INY-abstr4-loopu-invar* $\omega'\ \mathcal{C}'\ u'\ v'\ c\ v\ ?U''\ (?\omega''', ?\mathcal{C}''')$
proof (*intro INY-abstr4-loopu-invarI*)
have $\omega' \cup \textit{INY-abstr4-set}\ \omega'\ \mathcal{C}'\ u'\ v'\ c\ v\ u \subseteq$
$\omega'' \cup \textit{INY-abstr4-set}\ \omega''\ \mathcal{C}''\ u'\ v'\ c\ v\ u$

```
        unfolding INY-abstr4-set-def using invar(1) by auto
    moreover have INY-abstr4-set ω' 𝒞' u' v' c v u ∩ ω' = {}
        unfolding INY-abstr4-set-def using uv-notin-ω'' invar(1) by force
    moreover have ⋀a b u.(a, b) ∈ INY-abstr4-set ω' 𝒞' u' v' c v u
        ⟹ (a, b) = (u, v) unfolding INY-abstr4-set-def
      by (auto split: split-if-asm)
    ultimately show (⋃u∈{u.(u,c,u')∈Δ 𝒜}−?U''.
        INY-abstr4-set ω' 𝒞' u' v' c v u) ⊆ ?ω''' using invar(6) by auto
  next

    {
      assume (u,v)∈𝒮_𝒜
       hence ∃v'. (u',v') ∈ 𝒮_𝒜 ∧ (v,c,v')∈Δ 𝒜
          using u'-succ-u v'-succ-v ‹(u,v)∈𝒮_𝒜› 𝒮_𝒜-is-largest-sim by blast
       moreover have  ∀v''. (v, c, v'') ∈ Δ 𝒜 ⟶ (u', v'') ∈ ω''
           using v'-properties ‹(u',v')∈ω'› invar(1) by blast
       ultimately have False using invar(1,7) v'-properties by blast
    }
    thus 𝒮_𝒜 ∩ (insert (u,v) ω'') = {} using invar(7) by blast
  qed (insert invar uv-notin-ω'' u'-succ-u v'-succ-v Δ-consistent, auto)
 next

  case (goal2 ω'' 𝒞'')
  let ?U'' = U' − {u}
  have INY-abstr4-loopu-invar ω' 𝒞' u' v' c v U' (ω'', 𝒞'')
      using goal2(1,2) by simp
  note invar = INY-abstr4-loopu-invarD[OF this]
  note invar-loopv = INY-abstr3-loopv-invarD[OF I1]
  hence A: INY-abstr4-set ω 𝒞 u' v' c v u ⊆ ω'' using ‹¬(u,v) ∉ ω''›
     unfolding INY-abstr4-set-def by (auto split: split-if-asm)
  have B: (⋃u∈{u. (u,c,u')∈Δ 𝒜} − ?U''.
      INY-abstr4-set ω' 𝒞' u' v' c v u) = (⋃u∈{u. (u,c,u')∈Δ 𝒜} − U'.
      INY-abstr4-set ω' 𝒞' u' v' c v u) ∪ INY-abstr4-set ω' 𝒞' u' v' c v u
      using u'-succ-u by blast
  thus INY-abstr4-loopu-invar ω' 𝒞' u' v' c v ?U'' (ω'', 𝒞'')
    apply (intro INY-abstr4-loopu-invarI)
    using invar(1−5) apply (blast, blast, blast, blast, blast)
    apply (subst B) unfolding INY-abstr4-set-def
    using invar(6) ‹¬(u,v)∉ω''› apply auto[1]
    using invar apply simp-all[3]
    done
 qed

next
 case (goal4 ω''𝒞'') thus ?case
  proof (cases ω''𝒞'', simp)
    fix ω'' 𝒞''
    assume I: INY-abstr4-loopu-invar ω' 𝒞' u' v' c v {} (ω'', 𝒞'')
    note invar = INY-abstr4-loopu-invarD[OF I]
```

note *invar-loopv* = *INY-abstr3-loopv-invarD*[*OF I1*]
have $\bigwedge v''$. $(v,c,v'') \in \Delta\ \mathcal{A} \Longrightarrow (u',v'') \in \omega' - \mathcal{C}'$
using *v′-properties invar-loopv*(*1*) *invar*(*8*,*9*) **by** *blast*
hence $(\bigcup u{\in}\{u.\ (u,c,u'){\in}\Delta\ \mathcal{A}\}.$ *INY-abstr4-set* $\omega'\ \mathcal{C}'\ u'\ v'\ c\ v\ u) =$
INY-abstr3-set $\omega'\ \mathcal{C}'\ u'\ v'\ c\ v$ **using** *v′-succ-v*
unfolding *INY-abstr4-set-def INY-abstr3-set-def*
by (*auto split*: *split-if-asm*)
thus *INY-abstr3-is-valid-ω′C′* $\omega'\ \mathcal{C}'\ u'\ v'\ c\ v\ (\omega'', \mathcal{C}'')$
apply (*intro INY-abstr3-is-valid-ω′C′I*)
using *invar*(*1*−*3*) **apply** *auto*[*2*]
using *invar* **apply** *auto*
done
qed
qed

definition *INY-abstr4-loopv* **where**
INY-abstr4-loopv $\omega\ \mathcal{C}\ u'\ v'\ c \equiv$
$FOREACH^{INY\text{-}abstr3\text{-}loopv\text{-}invar\ \omega\ \mathcal{C}\ u'\ v'\ c}$
$\{v.\ (v,c,v'){\in}\Delta\ \mathcal{A}\}\ (\lambda v\ (\omega, \mathcal{C}).$
if $(\forall v''.\ (v,c,v''){\in}\Delta\ \mathcal{A} \longrightarrow (u',v'') \in \omega{-}\mathcal{C})$ *then do* {
$(\omega', \mathcal{C}') \leftarrow$ *INY-abstr4-loopu* $\omega\ \mathcal{C}\ u'\ v'\ c\ v$;
RETURN $(\omega', \mathcal{C}')$
} *else*
RETURN $(\omega, \mathcal{C})$
) $(\omega, \mathcal{C})$

lemma *INY-abstr4-loopv-correct*:
shows *INY-abstr4-loopv* $\omega'\ \mathcal{C}'\ u'\ v'\ c \leq$
$\Downarrow$*Id* (*INY-abstr3-loopv* $\omega'\ \mathcal{C}'\ u'\ v'\ c$)
unfolding *INY-abstr4-loopv-def INY-abstr3-loopv-def*
using *assms* **apply** (*refine-rcg inj-on-id*)
using *assms* **apply** *simp-all*[*4*]
apply *simp*
apply (*rule INY-abstr4-loopu-correct*, *simp*, *blast*)
apply *simp-all*
done

definition *INY-abstr4-loopc* **where**
INY-abstr4-loopc $\omega\ \mathcal{C}\ u'\ v' \equiv FOREACH^{INY\text{-}abstr2\text{-}loopc\text{-}invar\ \omega\ \mathcal{C}\ u'\ v'}$
$(\Sigma\ \mathcal{A})\ (\lambda c\ (\omega, \mathcal{C}).$ *do* {
$(\omega', \mathcal{C}') \leftarrow$ *INY-abstr4-loopv* $\omega\ \mathcal{C}\ u'\ v'\ c$;
RETURN $(\omega', \mathcal{C}')$
}) $(\omega, \mathcal{C})$

lemma *INY-abstr4-loopc-correct*:
assumes $(\omega_1{}',\omega_2{}') \in Id$ **and** $(\mathcal{C}_1{}',\mathcal{C}_2{}') \in Id$ **and**
$(u_1{}',u_2{}') \in Id$ **and** $(v_1{}',v_2{}') \in Id$

shows *INY-abstr4-loopc* $\omega_1{}' \, \mathcal{C}_1{}' \, u_1{}' \, v_1{}' \leq \Downarrow Id$ (*INY-abstr3-loopc* $\omega_2{}' \, \mathcal{C}_2{}' \, u_2{}' \, v_2{}'$)
unfolding *INY-abstr4-loopc-def INY-abstr3-loopc-def* **using** *assms*
apply (*refine-rcg inj-on-id INY-abstr4-loopv-correct*)
apply *simp-all*[*4*]
apply (*rule INY-abstr4-loopv-correct*)
apply *simp-all*
done

definition *INY-abstr4* **where**
INY-abstr4 $\equiv$ $WHILE_T{}^{INY\text{-}abstr1\text{-}invar}$ $(\lambda(\omega, \mathcal{C}).\ \mathcal{C}\neq\{\})$ $(\lambda(\omega, \mathcal{C}).$ *do* {
  $(u',v') \leftarrow SPEC\ (\lambda(u', v').\ (u',v') \in \mathcal{C})$;
  let $\mathcal{C} = \mathcal{C} - \{(u',v')\}$;
  $(\omega, \mathcal{C}) \leftarrow$ (*INY-abstr4-loopc* $\omega\ \mathcal{C}\ u'\ v'$);
  RETURN $(\omega, \mathcal{C})$
}) (*INY-initial*, *INY-initial*)

lemma *INY-abstr4-correct*:
  INY-abstr4 $\leq \Downarrow Id$ *INY-abstr3*
  unfolding *INY-abstr4-def INY-abstr3-def*
  by (*refine-rcg inj-on-id INY-abstr4-loopc-correct*, *simp-all*)

5.3.3 Optimisation with caching

We now introduce the counter N and the constant data structures d and δ^r. $N(c,u',v)$ stores the number of successors of v that are known to be unable to simulate v' and have been processed. $d(v,c)$ stores the number of successors of v w.r.t. c, i.e. $|\delta(v,c)|$ (or $|\{v'.\ (v,c,v') \in \Delta\}|$)

The counter's domain must be $\Sigma\times\mathcal{Q}\times\mathcal{Q}$ and initially, all values must be zero.

definition *INY-is-valid-initial-counter*::
  $(('a\times'q\times'q) \rightharpoonup nat) \Rightarrow bool$ **where**
INY-is-valid-initial-counter $N \equiv (dom\ N = \Sigma\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}) \wedge$
  $(\forall\,(c,u,v)\in dom\ N.\ N\ (c,u,v) = Some\ (card\ \{v'.\ (v,c,v') \in \Delta\ \mathcal{A}\}))$

Increments the counter value for $(c,\ u',\ v)$. This is used when we know that another successor of v is unable to simulate u'.

definition *INY-dec-counter* **where**
INY-dec-counter $N\ c\ u'\ v =$
  $(case\ N\ (c,u',v)\ of\ Some\ n \Rightarrow (let\ n = n - 1\ in$
    $(N((c,u',v) \mapsto n),\ n = 0))\ |\ None \Rightarrow (N,\ True))$

lemma *INY-dec-counter-correct*: **assumes** $N\ (c2,u'2,v2) = Some\ n$
  shows $(c,u',v)=(c2,u'2,v2) \Longrightarrow$
      $fst\ (INY\text{-}dec\text{-}counter\ N\ c\ u'\ v)\ (c2,u'2,v2) = Some\ (n-(1::nat))$
    $(c,u',v)\neq(c2,u'2,v2) \Longrightarrow$
      $fst\ (INY\text{-}dec\text{-}counter\ N\ c\ u'\ v)\ (c2,u'2,v2) = Some\ n$
    $dom\ (fst\ (INY\text{-}dec\text{-}counter\ N\ c\ u'\ v)) = dom\ N$

$N\ (c,u',v) = Some\ n \Longrightarrow snd\ (INY\text{-}dec\text{-}counter\ N\ c\ u'\ v) = (n - 1 = 0)$
using *assms* **by** (*auto simp*: *INY-dec-counter-def*
split: *option.split split-if-asm*)

The *dec-counter* function does not change the counter's domain.

lemma *INY-dec-counter-dom-unchanged*[*simp*]:
$dom\ (fst\ (INY\text{-}dec\text{-}counter\ N\ c\ u'\ v)) = dom\ N$
unfolding *INY-dec-counter-def dom-def* **by** (*auto split*: *option.split simp*: *Let-def*)

Only the incremented value changes.

lemma *INY-dec-counter-unaffected*: $(c,u',v) \neq (c2,u'2,v2) \Longrightarrow$
$fst\ (INY\text{-}dec\text{-}counter\ N\ c\ u'\ v)\ (c2,u'2,v2) = N\ (c2,u'2,v2)$
by (*auto simp*: *assms INY-dec-counter-def split*: *option.split simp*: *Let-def*)

d is the data structure that stores $|\delta(v,c)|$ for any $v \in \mathcal{Q}$, $c \in \Sigma$.

definition *INY-abstr5-d-correct* **where**
$INY\text{-}abstr5\text{-}d\text{-}correct\ d \equiv (dom\ d = \mathcal{Q}\ \mathcal{A} \times \Sigma\ \mathcal{A}) \wedge$
$(\forall (v,c) \in dom\ d.\ d\ (v,c) = Some\ (card\ \{v'.\ (v,c,v') \in \Delta\ \mathcal{A}\}))$

lemma *INY-abstr5-d-correctD*[*dest*]:
assumes *INY-abstr5-d-correct d* **and** $(v,c) \in \mathcal{Q}\ \mathcal{A} \times \Sigma\ \mathcal{A}$
shows $d\ (v,c) = Some\ (card\ \{v'.\ (v,c,v') \in \Delta\ \mathcal{A}\})$
using *assms* **unfolding** *INY-abstr5-d-correct-def* **by** *blast*

δ^r is the data structure that stores the predecessors of v w.r.t. c for any $v \in \mathcal{Q}$, $c \in \Sigma$.

definition *INY-abstr5-δ^r-correct* **where**
$INY\text{-}abstr5\text{-}\delta^r\text{-}correct\ d \equiv (dom\ d = \mathcal{Q}\ \mathcal{A} \times \Sigma\ \mathcal{A}) \wedge$
$(\forall (v,c) \in dom\ d.\ d\ (v,c) = Some\ \{v'.\ (v',c,v) \in \Delta\ \mathcal{A}\})$

definition *INY-abstr5-$N\delta^r$-correct* **where**
$INY\text{-}abstr5\text{-}N\delta^r\text{-}correct\ \omega\ \mathcal{C}\ N\ \delta^r \equiv (dom\ N = \Sigma\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}) \wedge$
$(\forall (c,u',v) \in dom\ N.\ N\ (c,u',v) =$
$Some\ (card\ \{v''.\ (v,c,v'') \in \Delta\ \mathcal{A} \wedge (u',v'') \notin \omega - \mathcal{C}\})) \wedge$
$INY\text{-}abstr5\text{-}\delta^r\text{-}correct\ \delta^r$

lemma *INY-abstr5-$N\delta^r$-correctI*:
assumes $dom\ N = \Sigma\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$
$\bigwedge c\ u'\ v.\ [\![c \in \Sigma\ \mathcal{A};\ u' \in \mathcal{Q}\ \mathcal{A};\ v \in \mathcal{Q}\ \mathcal{A}]\!] \Longrightarrow$
$N\ (c,u',v) = Some\ (card\ \{v''.\ (v,c,v'') \in \Delta\ \mathcal{A} \wedge (u',v'') \notin \omega - \mathcal{C}\})$
$INY\text{-}abstr5\text{-}\delta^r\text{-}correct\ \delta^r$
shows $INY\text{-}abstr5\text{-}N\delta^r\text{-}correct\ \omega\ \mathcal{C}\ N\ \delta^r$
unfolding *INY-abstr5-$N\delta^r$-correct-def* **using** *assms* **by** *blast*

lemma *INY-abstr5-$N\delta^r$-correctD*[*dest*]:
assumes $INY\text{-}abstr5\text{-}N\delta^r\text{-}correct\ \omega\ \mathcal{C}\ N\ \delta^r$
shows $dom\ N = \Sigma\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A} \times \mathcal{Q}\ \mathcal{A}$
$\bigwedge c\ u'\ v.\ [\![c \in \Sigma\ \mathcal{A};\ u' \in \mathcal{Q}\ \mathcal{A};\ v \in \mathcal{Q}\ \mathcal{A}]\!] \Longrightarrow$

$N\ (c,u',v) = Some\ (card\ \{v''.\ (v,c,v'')\in\Delta\ \mathcal{A} \wedge (u',v'')\notin\omega - \mathcal{C}\})$
INY-abstr5-δ^r-correct δ^r
using *assms* **unfolding** *INY-abstr5-Nδ^r-correct-def* **by** *blast+*

definition *INY-abstr5-invar* **where**
INY-abstr5-invar $\delta^r \equiv \lambda(\omega,\ \mathcal{C},\ N).$
INY-abstr1-invar $(\omega,\ \mathcal{C}) \wedge$ *INY-abstr5-Nδ^r-correct* $\omega\ \mathcal{C}\ N\ \delta^r$

lemma *INY-abstr5-invarI*:
assumes *INY-abstr1-invar* $(\omega,\ \mathcal{C})$ **and** *INY-abstr5-Nδ^r-correct* $\omega\ \mathcal{C}\ N\ \delta^r$
shows *INY-abstr5-invar* $\delta^r\ (\omega,\ \mathcal{C},\ N)$
unfolding *INY-abstr5-invar-def* **using** *assms* **by** *blast*

lemma *INY-abstr5-invarD*[*dest*]:
assumes *INY-abstr5-invar* $\delta^r\ (\omega,\ \mathcal{C},\ N)$
shows *INY-abstr1-invar* $(\omega,\ \mathcal{C})$ *INY-abstr5-Nδ^r-correct* $\omega\ \mathcal{C}\ N\ \delta^r$
using *assms* **unfolding** *INY-abstr5-invar-def* **by** *blast+*

definition *INY-abstr5-loopc-Nδ^r-correct* **where**
INY-abstr5-loopc-Nδ^r-correct $\omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'\ \Sigma'\ N' \equiv$
$(dom\ N' = \Sigma\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}) \wedge$
$(\forall (c,u'',v)\in dom\ N'.\ N'\ (c,u'',v) = (if\ c\in\Sigma'$
$then\ N\ (c,u'',v)$
$else\ Some\ (card\ \{v''.\ (v,c,v'')\in\Delta\ \mathcal{A} \wedge (u'',v'')\notin(\omega - \mathcal{C})\})$
$)) \wedge$ *INY-abstr5-δ^r-correct* δ^r

lemma *INY-abstr5-loopc-Nδ^r-correctI*:
assumes $dom\ N' = \Sigma\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}$
$\bigwedge c\ u''\ v.\ [\![c\in\Sigma\ \mathcal{A};\ c\in\Sigma';\ u''\in\mathcal{Q}\ \mathcal{A};\ v\in\mathcal{Q}\ \mathcal{A}]\!] \Longrightarrow$
$N'\ (c,u'',v) = N\ (c,u'',v)$ **and**
$\bigwedge c\ u''\ v.\ [\![c\in\Sigma\ \mathcal{A};\ c\notin\Sigma';\ u''\in\mathcal{Q}\ \mathcal{A};\ v\in\mathcal{Q}\ \mathcal{A}]\!] \Longrightarrow$
$N'\ (c,u'',v) = Some\ (card\ \{v''.\ (v,c,v'')\in\Delta\ \mathcal{A} \wedge$
$(u'',v'')\notin(\omega - \mathcal{C})\})$
INY-abstr5-δ^r-correct δ^r
shows *INY-abstr5-loopc-Nδ^r-correct* $\omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'\ \Sigma'\ N'$
unfolding *INY-abstr5-loopc-Nδ^r-correct-def* **using** *assms* **by** *auto*

lemma *INY-abstr5-loopc-Nδ^r-correctD*[*dest*]:
assumes *INY-abstr5-loopc-Nδ^r-correct* $\omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'\ \Sigma'\ N'$
shows $dom\ N' = \Sigma\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}$
$\bigwedge c\ u''\ v.\ [\![c\in\Sigma\ \mathcal{A};\ c\in\Sigma';\ u''\in\mathcal{Q}\ \mathcal{A};\ v\in\mathcal{Q}\ \mathcal{A}]\!] \Longrightarrow$
$N'\ (c,u'',v) = N\ (c,u'',v)$ **and**
$\bigwedge c\ u''\ v.\ [\![c\in\Sigma\ \mathcal{A};\ c\notin\Sigma';\ u''\in\mathcal{Q}\ \mathcal{A};\ v\in\mathcal{Q}\ \mathcal{A}]\!] \Longrightarrow$
$N'\ (c,u'',v) = Some\ (card\ \{v''.\ (v,c,v'')\in\Delta\ \mathcal{A} \wedge$
$(u'',v'')\notin(\omega - \mathcal{C})\})$
INY-abstr5-δ^r-correct δ^r
using *assms* **unfolding** *INY-abstr5-loopc-Nδ^r-correct-def* **by** *auto*

definition *INY-abstr5-loopc-invar* **where**

$INY\text{-}abstr5\text{-}loopc\text{-}invar\ \omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'\ \Sigma' \equiv \lambda(\omega',\ \mathcal{C}',\ N').$
$\quad INY\text{-}abstr2\text{-}loopc\text{-}invar\ \omega\ \mathcal{C}\ u'\ v'\ \Sigma'\ (\omega',\ \mathcal{C}') \wedge$
$\quad INY\text{-}abstr5\text{-}loopc\text{-}N\delta^r\text{-}correct\ \omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'\ \Sigma'\ N'$

lemma $INY\text{-}abstr5\text{-}loopc\text{-}invarI$:
assumes $INY\text{-}abstr2\text{-}loopc\text{-}invar\ \omega\ \mathcal{C}\ u'\ v'\ \Sigma'\ (\omega',\ \mathcal{C}')$ **and**
$\quad INY\text{-}abstr5\text{-}loopc\text{-}N\delta^r\text{-}correct\ \omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'\ \Sigma'\ N'$
shows $INY\text{-}abstr5\text{-}loopc\text{-}invar\ \omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'\ \Sigma'\ (\omega',\ \mathcal{C}',\ N')$
unfolding *INY-abstr5-loopc-invar-def* **using** *assms* **by** *blast*

lemma $INY\text{-}abstr5\text{-}loopc\text{-}invarD$:
assumes $INY\text{-}abstr5\text{-}loopc\text{-}invar\ \omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'\ \Sigma'\ (\omega',\ \mathcal{C}',\ N')$
shows $INY\text{-}abstr2\text{-}loopc\text{-}invar\ \omega\ \mathcal{C}\ u'\ v'\ \Sigma'\ (\omega',\ \mathcal{C}')$ **and**
$\quad INY\text{-}abstr5\text{-}loopc\text{-}N\delta^r\text{-}correct\ \omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'\ \Sigma'\ N'$
using *assms* **unfolding** *INY-abstr5-loopc-invar-def* **by** *blast+*

definition $INY\text{-}abstr5\text{-}loopv\text{-}N\delta^r\text{-}correct$ **where**
$INY\text{-}abstr5\text{-}loopv\text{-}N\delta^r\text{-}correct\ \omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'\ c'\ V'\ N' \equiv$
$\quad (dom\ N' = \Sigma\ \mathcal{A}{\times}\mathcal{Q}\ \mathcal{A}{\times}\mathcal{Q}\ \mathcal{A}) \wedge$
$\quad\quad (\forall (c,u'',v){\in}dom\ N'.\ N'\ (c,u'',v) = (if\ c{\neq}c' \vee v{\in}V'$
$\quad\quad\quad then\ N\ (c,u'',v)$
$\quad\quad\quad else\ Some\ (card\ \{v''.\ (v,c,v''){\in}\Delta\ \mathcal{A} \wedge (u'',v''){\notin}(\omega{-}\mathcal{C})\})$
$\quad)) \wedge INY\text{-}abstr5\text{-}\delta^r\text{-}correct\ \delta^r$

lemma $INY\text{-}abstr5\text{-}loopv\text{-}N\delta^r\text{-}correctI$:
assumes $dom\ N' = \Sigma\ \mathcal{A}{\times}\mathcal{Q}\ \mathcal{A}{\times}\mathcal{Q}\ \mathcal{A}$
$\quad \bigwedge c\ u''\ v.\ [\![c{\in}\Sigma\ \mathcal{A};\ c{\neq}c';\ u''{\in}\mathcal{Q}\ \mathcal{A};\ v{\in}\mathcal{Q}\ \mathcal{A}]\!] \Longrightarrow$
$\quad\quad N'\ (c,u'',v) = N\ (c,u'',v)$ **and**
$\quad \bigwedge c\ u''\ v.\ [\![c{\in}\Sigma\ \mathcal{A};\ v{\in}V';\ u''{\in}\mathcal{Q}\ \mathcal{A};\ v{\in}\mathcal{Q}\ \mathcal{A}]\!] \Longrightarrow$
$\quad\quad N'\ (c,u'',v) = N\ (c,u'',v)$
$\quad \bigwedge c\ u''\ v.\ [\![c{\in}\Sigma\ \mathcal{A};\ c{=}c';\ v{\notin}V';\ u''{\in}\mathcal{Q}\ \mathcal{A};\ v{\in}\mathcal{Q}\ \mathcal{A}]\!] \Longrightarrow$
$\quad\quad N'\ (c,u'',v) = Some\ (card\ \{v''.\ (v,c,v''){\in}\Delta\ \mathcal{A} \wedge$
$\quad\quad\quad (u'',v''){\notin}(\omega{-}\mathcal{C})\})$
$\quad INY\text{-}abstr5\text{-}\delta^r\text{-}correct\ \delta^r$
shows $INY\text{-}abstr5\text{-}loopv\text{-}N\delta^r\text{-}correct\ \omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'\ c'\ V'\ N'$
unfolding *INY-abstr5-loopv-Nδ^r-correct-def* **using** *assms* **by** *auto*

lemma $INY\text{-}abstr5\text{-}loopv\text{-}N\delta^r\text{-}correctD$:
assumes $INY\text{-}abstr5\text{-}loopv\text{-}N\delta^r\text{-}correct\ \omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'\ c'\ V'\ N'$
shows $dom\ N' = \Sigma\ \mathcal{A}{\times}\mathcal{Q}\ \mathcal{A}{\times}\mathcal{Q}\ \mathcal{A}$
$\quad \bigwedge c\ u''\ v.\ [\![c{\in}\Sigma\ \mathcal{A};\ c{\neq}c';\ u''{\in}\mathcal{Q}\ \mathcal{A};\ v{\in}\mathcal{Q}\ \mathcal{A}]\!] \Longrightarrow$
$\quad\quad N'\ (c,u'',v) = N\ (c,u'',v)$ **and**
$\quad \bigwedge c\ u''\ v.\ [\![c{\in}\Sigma\ \mathcal{A};\ v{\in}V';\ u''{\in}\mathcal{Q}\ \mathcal{A};\ v{\in}\mathcal{Q}\ \mathcal{A}]\!] \Longrightarrow$
$\quad\quad N'\ (c,u'',v) = N\ (c,u'',v)$
$\quad \bigwedge c\ u''\ v.\ [\![c{\in}\Sigma\ \mathcal{A};\ c{=}c';\ v{\notin}V';\ u''{\in}\mathcal{Q}\ \mathcal{A};\ v{\in}\mathcal{Q}\ \mathcal{A}]\!] \Longrightarrow$
$\quad\quad N'\ (c,u'',v) = Some\ (card\ \{v''.\ (v,c,v''){\in}\Delta\ \mathcal{A} \wedge$

$(u'',v'') \notin (\omega - \mathcal{C})\})$
INY-abstr5-δ^r-correct δ^r
using *assms* **unfolding** *INY-abstr5-loopv-Nδ^r-correct-def* **by** *auto*

definition *INY-abstr5-loopv-invar* **where**
INY-abstr5-loopv-invar ω $\mathcal{C}$ N δ^r u' v' c V' $\equiv \lambda(\omega', \mathcal{C}', N')$.
INY-abstr3-loopv-invar ω $\mathcal{C}$ u' v' c V' $(\omega', \mathcal{C}') \wedge$
INY-abstr5-loopv-Nδ^r-correct ω $\mathcal{C}$ N δ^r u' v' c V' N'

lemma *INY-abstr5-loopv-invarI*:
assumes *INY-abstr3-loopv-invar* ω $\mathcal{C}$ u' v' c V' $(\omega', \mathcal{C}')$ **and**
INY-abstr5-loopv-Nδ^r-correct ω $\mathcal{C}$ N δ^r u' v' c V' N'
shows *INY-abstr5-loopv-invar* ω $\mathcal{C}$ N δ^r u' v' c V' $(\omega', \mathcal{C}', N')$
unfolding *INY-abstr5-loopv-invar-def* **using** *assms* **by** *blast*

lemma *INY-abstr5-loopv-invarD*[*intro*]:
assumes *INY-abstr5-loopv-invar* ω $\mathcal{C}$ N δ^r u' v' c V' $(\omega', \mathcal{C}', N')$
shows *INY-abstr3-loopv-invar* ω $\mathcal{C}$ u' v' c V' $(\omega', \mathcal{C}')$ **and**
INY-abstr5-loopv-Nδ^r-correct ω $\mathcal{C}$ N δ^r u' v' c V' N'
using *assms* **unfolding** *INY-abstr5-loopv-invar-def* **by** *blast+*

The new, optimised loops of the algorithm using the cache variables we have just introduces.

definition *INY-abstr5-loopu* **where**
INY-abstr5-loopu ω $\mathcal{C}$ δ^r u' v' c v $\equiv$
FOREACH$^{\textit{INY-abstr4-loopu-invar}\ \omega\ \mathcal{C}\ u'\ v'\ c\ v}$
(*case* $\delta^r(u',c)$ *of None* $\Rightarrow$ {} | *Some s* $\Rightarrow$ *s*) (λu $(\omega, \mathcal{C})$.
if $(u,v) \notin \omega$ *then do* {
ASSERT $((u,v) \notin \omega)$;
ASSERT $((u,v) \notin \mathcal{C})$;
RETURN (*insert* (u,v) ω, *insert* (u,v) $\mathcal{C}$)
} *else*
RETURN $(\omega, \mathcal{C})$
) $(\omega, \mathcal{C})$

lemma *INY-abstr5-δ^r-correct*:
assumes *INY-abstr2-loopc-invar* ω $\mathcal{C}$ u' v' Σ' $(\omega',\mathcal{C}')$
INY-abstr5-Nδ^r-correct ω $\mathcal{C}_2$ N δ^r
$c \in \Sigma'$ $\quad \Sigma' \subseteq \Sigma\ \mathcal{A}$
shows $(\delta^r(u',c))$ = *Some* $\{u.\ (u,c,u') \in \Delta\ \mathcal{A}\}$ (**is** *?A*) **and**
$(\delta^r(v',c))$ = *Some* $\{v.\ (v,c,v') \in \Delta\ \mathcal{A}\}$ (**is** *?B*)
proof−
from *INY-abstr2-loopc-invarD*(*1*,*4*,*8*)[*OF assms*(*1*)]
have $u' \in \mathcal{Q}\ \mathcal{A}$ $\quad v' \in \mathcal{Q}\ \mathcal{A}$ **by** *blast+*
moreover from ‹$c \in \Sigma'$› **and** ‹$\Sigma' \subseteq \Sigma\ \mathcal{A}$› **have** $c \in \Sigma\ \mathcal{A}$ **by** *blast*
moreover note *INY-abstr5-Nδ^r-correctD*(*3*)[*OF assms*(*2*)]
ultimately show *?A* **and** *?B* **unfolding** *INY-abstr5-δ^r-correct-def* **by** *auto*
qed

lemma *INY-abstr5-loopu-correct*:
assumes $\mathit{INY\text{-}abstr2\text{-}loopc\text{-}invar}\ \omega\ \mathcal{C}\ u'\ v'\ \Sigma'\ (\omega',\mathcal{C}')$
$\mathit{INY\text{-}abstr5\text{-}N\delta^r\text{-}correct}\ \omega\ (\mathcal{C}\cup\{(u',v')\})\ N\ \delta^r$
$c\in\Sigma' \quad \Sigma'\subseteq\Sigma\ \mathcal{A}$
shows $\mathit{INY\text{-}abstr5\text{-}loopu}\ \omega''\ \mathcal{C}''\ \delta^r\ u'\ v'\ c\ v \leq \Downarrow\mathit{Id}$
$(\mathit{INY\text{-}abstr4\text{-}loopu}\ \omega''\ \mathcal{C}''\ u'\ v'\ c\ v)$
unfolding *INY-abstr5-loopu-def INY-abstr4-loopu-def*
thm $\mathit{INY\text{-}abstr5\text{-}\delta^r\text{-}correct}[\mathit{OF\ assms}(1)]$
by $(\mathit{refine\text{-}rcg\ inj\text{-}on\text{-}id},\ \mathit{simp\text{-}all\ add}\colon \mathit{INY\text{-}abstr5\text{-}\delta^r\text{-}correct}[\mathit{OF\ assms}])$

definition *INY-abstr5-loopv* **where**
$\mathit{INY\text{-}abstr5\text{-}loopv}\ \omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'\ c \equiv$
$\mathit{FOREACH}^{\mathit{INY\text{-}abstr5\text{-}loopv\text{-}invar}\ \omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'\ c}$
$(\mathit{case}\ \delta^r(v',c)\ \mathit{of\ None} \Rightarrow \{\}\ |\ \mathit{Some}\ s \Rightarrow s)\ (\lambda v\ (\omega,\ \mathcal{C},\ N).\ \mathit{do}\ \{$
$\mathit{let}\ (N,\ \mathit{iszero}) = \mathit{INY\text{-}dec\text{-}counter}\ N\ c\ u'\ v;$
$\mathit{if\ iszero\ then\ do}\ \{$
$(\omega',\ \mathcal{C}') \leftarrow \mathit{INY\text{-}abstr5\text{-}loopu}\ \omega\ \mathcal{C}\ \delta^r\ u'\ v'\ c\ v;$
$\mathit{RETURN}\ (\omega',\ \mathcal{C}',\ N)$
$\}\ \mathit{else}$
$\mathit{RETURN}\ (\omega,\ \mathcal{C},\ N)$
$\})\ (\omega,\ \mathcal{C},\ N)$

If $v \notin \delta^{-1}(v',)$, we don't have to do any updates on v, it is not affected by the new information about $(u',\ v')$.

lemma *INY-abstr5-loopv-N-unaffected*:
assumes $(v,c,v')\notin\Delta\ \mathcal{A}$
shows $\mathit{card}\ \{v''.(v,c,v'')\in\Delta\ \mathcal{A} \wedge (u'',v'')\notin\omega-\mathcal{C}\} =$
$\mathit{card}\ \{v''.\ (v,c,v'')\in\Delta\ \mathcal{A} \wedge (u'',v'')\notin\omega-(\mathcal{C}\cup\{(u',v')\})\}$
(**is** $\mathit{card}\ ?U = \mathit{card}\ ?V$)
by $(\mathit{subgoal\text{-}tac}\ ?U{=}?V,\ \mathit{simp},\ \mathit{insert\ assms},\ \mathit{blast})$

If the outer invariant holds, the counters have the correct values initially.

lemma *INY-abstr5-loopv-Nδ^r-correct-initial*:
assumes $\mathit{INY\text{-}abstr5\text{-}loopc\text{-}N\delta^r\text{-}correct}\ \omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'\ \Sigma'\ N'$
$\mathit{INY\text{-}abstr2\text{-}loopc\text{-}invar}\ \omega\ \mathcal{C}\ u'\ v'\ \Sigma'\ (\omega',\mathcal{C}') \quad c\in\Sigma'$
$\mathit{INY\text{-}abstr5\text{-}N\delta^r\text{-}correct}\ \omega\ (\mathcal{C} \cup \{(u',v')\})\ N\ \delta^r$
shows $\mathit{INY\text{-}abstr5\text{-}loopv\text{-}N\delta^r\text{-}correct}\ \omega\ \mathcal{C}\ N'\ \delta^r\ u'\ v'\ c\ \{v.\ (v,c,v') \in \Delta\ \mathcal{A}\}\ N'$
apply $(\mathit{rule\ INY\text{-}abstr5\text{-}loopv\text{-}N\delta^r\text{-}correctI})$
defer *4*
apply $(\mathit{fact\ INY\text{-}abstr5\text{-}loopc\text{-}N\delta^r\text{-}correctD}(1)[\mathit{OF\ assms}(1)])$
apply *simp*
apply *blast*
apply $(\mathit{fact\ INY\text{-}abstr5\text{-}N\delta^r\text{-}correctD}(3)[\mathit{OF\ assms}(4)])$
proof−
case $(\mathit{goal1}\ c'\ u''\ v)$
note $\mathit{invar\text{-}loopc} = \mathit{INY\text{-}abstr2\text{-}loopc\text{-}invarD}[\mathit{OF\ assms}(2)]$
have $(v,c',v')\notin\Delta\ \mathcal{A}$ **using** $\mathit{goal1}(2,3)$ **by** *blast*

```
    have c'∈Σ' using goal1(2) assms(3) by simp
   note INY-abstr5-loopc-Nδ^r-correctD(2)[OF assms(1) goal1(1) this goal1(4,5)]
    also note INY-abstr5-Nδ^r-correctD(2)[OF assms(4) goal1(1,4,5)]
    also note INY-abstr5-loopv-N-unaffected[OF ‹(v,c',v')∉Δ 𝒜›, symmetric]
    finally show ?case .
qed
```

The original if condition (v has no successor w.r.t. c that can simulate u') and the new one (the counter for (c, u', v) is at its maximum) are equivalent.

```
lemma INY-abstr5-loopv-N-eq-0-iff:
  assumes I1: INY-abstr3-loopv-invar ω' 𝒞' u' v' c V' (ω'', 𝒞'') and
          C1: INY-abstr5-loopv-Nδ^r-correct ω 𝒞 N' δ^r u' v' c V' N'' and
          I2: INY-abstr2-loopc-invar ω 𝒞 u' v' Σ' (ω',𝒞') and
          C2: INY-abstr5-loopc-Nδ^r-correct ω 𝒞 N δ^r u' v' Σ' N' and
          C3: INY-abstr5-Nδ^r-correct ω (𝒞∪{(u',v')}) N δ^r and v∈V' and
             V'⊆{v. (v,c,v')∈Δ 𝒜} and c∈Σ' and Σ'⊆Σ 𝒜
  shows snd (INY-dec-counter N'' c u' v) =
          (∀v''. (v,c,v'')∈Δ 𝒜 ⟶ (u',v'') ∈ ω'' − 𝒞'')
proof−
  note correct = INY-abstr5-loopv-Nδ^r-correctD[OF C1]
  note correct-loopc = INY-abstr5-loopc-Nδ^r-correctD[OF C2]
  note correct-while = INY-abstr5-Nδ^r-correctD[OF C3]
  note invar = INY-abstr3-loopv-invarD[OF I1]
  note invar-loopc = INY-abstr2-loopc-invarD[OF I2]

  let ?X1 = {v''.(v,c,v'')∈Δ 𝒜 ∧ (u',v'')∉ω − (𝒞 ∪ {(u',v')})}
  let ?X2 = {v''.(v,c,v'')∈Δ 𝒜 ∧ (u',v'')∉ω − 𝒞}
  let ?X3 = {v''.(v,c,v'')∈Δ 𝒜 ∧ (u',v'')∉ω'− 𝒞'}

  from ‹v∈V'› and ‹V'⊆{v. (v,c,v')∈Δ 𝒜}› have (v,c,v')∈Δ 𝒜   v∈𝒬 𝒜
     using Δ-consistent by blast+
  from ‹c∈Σ'› ‹Σ'⊆Σ 𝒜› have c∈Σ 𝒜 by blast
  have u'∈𝒬 𝒜 using invar-loopc(1,4,8) by blast

  have ?X1 ⊆ 𝒬 𝒜   ?X2 ⊆ 𝒬 𝒜 using Δ-consistent by blast+
  hence fin: finite ?X1   finite ?X2 using finite-𝒬 finite-Σ
    by (blast intro: finite-subset)+

  have ?X2 = ?X1 − {v'} and v'∈?X1 using ‹(v,c,v')∈Δ 𝒜›
     invar-loopc(2,8,9) by auto
  hence new-card: card ?X2 = card ?X1 − 1 using fin invar-loopc(8) by simp

  have N'' (c,u',v) = N' (c,u',v)
     using correct(3)[OF ‹c∈Σ 𝒜› - ‹u'∈𝒬 𝒜› ‹v∈𝒬 𝒜›] ‹v∈V'› by simp
  also have ... = N (c,u',v)
     using correct-loopc(2)[OF ‹c∈Σ 𝒜› ‹c∈Σ'› ‹u'∈𝒬 𝒜› ‹v∈𝒬 𝒜›] .
  also have ... = Some (card ?X1)
     using correct-while(2)[OF ‹c∈Σ 𝒜› ‹u'∈𝒬 𝒜› ‹v∈𝒬 𝒜›] .
  finally have snd (INY-dec-counter N'' c u' v) ⟷ (card ?X2 = 0)
```

using *INY-dec-counter-correct*(4)[*of* N'' c u' v *card* *?X1*] *new-card*
by *simp*
hence *snd* (*INY-dec-counter* N'' c u' v) = (*?X2*={})
using *fin* **by** *simp*
thus *?thesis* **using** *invar*(1,2) *correct*(2) *invar-loopc*(1,2) **by** *blast*
qed

The counter is updated correctly, i.e. after an iteration, it will correctly reflect the fact that v' is a successor of v that cannot simulate u'.

lemma *INY-abstr5-loopv-N-correctness-preserved*:
assumes *I1*: *INY-abstr3-loopv-invar* ω' $\mathcal{C}'$ u' v' c V' (ω'', $\mathcal{C}''$) **and**
C1: *INY-abstr5-loopv-N*δ^r*-correct* ω $\mathcal{C}$ N' δ^r u' v' c V' N'' **and**
I2: *INY-abstr2-loopc-invar* ω $\mathcal{C}$ u' v' Σ' (ω',$\mathcal{C}'$) **and**
C2: *INY-abstr5-loopc-N*δ^r*-correct* ω $\mathcal{C}$ N δ^r u' v' Σ' N' **and**
I3: *INY-abstr1-invar* (ω,$\mathcal{CC}$ $\cup$ {(u',v')}) **and**
C3: *INY-abstr5-N*δ^r*-correct* ω ($\mathcal{C}$$\cup${($u'$,$v'$)}) N δ^r **and** $v \in V'$ **and**
$V' \subseteq \{v.\ (v,c,v') \in \Delta\ \mathcal{A}\}$ **and** $c \in \Sigma'$
shows *INY-abstr5-loopv-N*δ^r*-correct* ω $\mathcal{C}$ N' δ^r u' v' c (V' − {v})
(*fst* (*INY-dec-counter* N'' c u' v))
proof−
from ‹$v \in V'$› **and** ‹$V' \subseteq \{v.\ (v,c,v') \in \Delta\ \mathcal{A}\}$› **have** $(v,c,v') \in \Delta\ \mathcal{A}$ **by** *blast*
note *correct* = *INY-abstr5-loopv-N*δ^r*-correctD*[*OF C1*]
note *correct-loopc* = *INY-abstr5-loopc-N*δ^r*-correctD*[*OF C2*]
note *correct-while* = *INY-abstr5-N*δ^r*-correctD*[*OF C3*]
note *invar* = *INY-abstr3-loopv-invarD*[*OF I1*]
note *invar-loopc* = *INY-abstr2-loopc-invarD*[*OF I2*]
note *invar-while* = *INY-abstr1-invarD*[*OF I3*]
from *invar-loopc*(1,4,8) **have** $u' \in \mathcal{Q}\ \mathcal{A}$ **by** *blast*
show *?thesis*
apply (*rule INY-abstr5-loopv-N*δ^r*-correctI*)
using *correct*(1) **apply** *simp*
using *INY-dec-counter-unaffected*[*of* c u' v - - - N'']
correct(2) **apply** *auto* []
using *INY-dec-counter-unaffected*[*of* c u' v - - - N'']
correct(3) **apply** *auto* []
defer
apply (*fact correct*(5))

proof−
case (*goal1* c' u'' v'')
hence $c' \in \Sigma'$ **using** ‹$c \in \Sigma'$› **by** *simp*
show *?case* **proof**(*cases* v''=v)
case *False*
hence $(c,u',v) \neq (c',u'',v'')$ **by** *blast*
note *INY-dec-counter-unaffected*[*OF this*, *of* N'']
moreover have $v'' \notin V'$ **using** *goal1*(3) ‹$v'' \neq v$› **by** *simp*
note *correct*(4)[*OF goal1*(1,2) *this goal1*(4,5)]
moreover have $\omega' - (\mathcal{C}' - \{(u',v')\}) = \omega - (\mathcal{C} - \{(u',v')\})$
using *invar-loopc*(1−3,8) *invar*(1,2) **by** *blast*

```
      ultimately show ?thesis by simp
  next

    case True
      hence v∈𝒬 𝒜 and v''∈V' using ‹v∈V'› and ‹(v,c,v')∈Δ 𝒜›
          Δ-consistent by blast+
      thus ?thesis proof (cases u''=u')
        case False
          let ?X1 = {v'''.(v'',c',v''')∈Δ 𝒜 ∧ (u'',v''')∉ω −
              (𝒞 ∪ {(u',v')})}
          let ?X2 = {v'''.(v'',c',v''')∈Δ 𝒜 ∧ (u'',v''')∉ω−𝒞}
          from False have set-unchanged: ?X1 = ?X2 by blast

          from False have (c,u',v) ≠ (c',u'',v'') by blast
          note INY-dec-counter-unaffected[OF this, of N'']
          also note correct(3)[OF goal1(1) ‹v''∈V'›goal1(4) ‹v''∈𝒬 𝒜›]
          also note correct-loopc(2)[OF goal1(1) ‹c'∈Σ'›goal1(4,5)]
          also note correct-while(2)[OF goal1(1,4,5)]
          finally have fst (INY-dec-counter N'' c u' v) (c',u'',v'') =
              Some (card ?X1) using ‹v''=v› by simp
          hence fst (INY-dec-counter N'' c u' v) (c',u'',v'') =
              Some (card ?X2) using set-unchanged by simp
          moreover have ω−(𝒞−{(u',v')})=ω'−(𝒞'−{(u',v')})
            using invar-loopc(1−3,8) invar(1−3) by blast
          ultimately show ?thesis by simp
        next

        case True
          let ?X1 = {v'''.(v'',c',v''')∈Δ 𝒜 ∧ (u'',v''')∉ω−
              (𝒞 ∪ {(u',v')})}
          let ?X2 = {v'''.(v'',c',v''')∈Δ 𝒜 ∧ (u'',v''')∉ω−𝒞}
          have ?X1 ⊆ 𝒬 𝒜    ?X2 ⊆ 𝒬 𝒜 using Δ-consistent by blast+
          hence fin: finite ?X1    finite ?X2 using finite-𝒬
              using rev-finite-subset by blast+
          have ?X2 = ?X1 − {v'} and v' ∈ ?X1 using invar(2,8)
              ‹u'' = u'› ‹v'' = v›‹(v,c,v')∈Δ 𝒜› ‹c'=c›
              invar-loopc(2,8,9) by auto

          hence new-card: card ?X2 = card ?X1 − 1
              using fin invar-loopc(8) by simp

          from True have param-eq: (c,u',v) = (c',u'',v'')
              using ‹v''=v› ‹c'=c› ‹c∈Σ'› by simp
          note correct(3)[OF goal1(1) ‹v∈V'› ‹u'∈𝒬 𝒜› ‹v∈𝒬 𝒜›]
          also note correct-loopc(2)[OF goal1(1) ‹c'∈Σ'› ‹u'∈𝒬 𝒜› ‹v∈𝒬 𝒜›]
          also note correct-while(2)[OF goal1(1) ‹u'∈𝒬 𝒜› ‹v∈𝒬 𝒜›]
          finally have N'' (c',u'',v'') = Some (card ?X1)
              using ‹u''=u'› ‹v''=v› by simp
          moreover note INY-dec-counter-correct(1)[OF -
```

param-eq, *of* N'' *card ?X1*]
ultimately have *fst* (*INY-dec-counter* N'' *c* u' *v*) (c',u'',v'') =
Some (*card ?X2*) **using** *new-card* **by** *simp*
moreover have $\omega-(\mathcal{C}-\{(u',v')\})=\omega'-(\mathcal{C}'-\{(u',v')\})$
using *invar-loopc*(*1−3*,*8*) *invar*(*1−3*) **by** *blast*
ultimately show *?thesis* **using** ‹$u''=u'$›‹$v''=v$› **by** *simp*
qed
qed
qed
qed

lemma *INY-abstr5-loopv-invarI2*:
assumes *INY-abstr2-loopc-invar* ω $\mathcal{C}$ u' v' Σ' $(\omega',\mathcal{C}')$ **and**
INY-abstr3-loopv-invar ω' $\mathcal{C}'$ u' v' *c* V' $(\omega'', \mathcal{C}'')$ **and**
INY-abstr5-loopv-Nδ^r*-correct* ω $\mathcal{C}$ N' δ^r u' v' *c* V' N''
shows *INY-abstr5-loopv-invar* ω' $\mathcal{C}'$ N' δ^r u' v' *c* V' $(\omega'',\mathcal{C}'',N'')$
apply (*intro INY-abstr5-loopv-invarI*)
using *assms*(*2*) **apply** *simp*
apply (*subgoal-tac* $\omega'-\mathcal{C}' = \omega-\mathcal{C}$)
using *assms*(*3*) **unfolding** *INY-abstr5-loopv-N*δ^r*-correct-def* **apply** *clarsimp*
using *INY-abstr2-loopc-invarD*(*1−3*,*8*)[*OF assms*(*1*)] **apply** *blast*
done

abbreviation *INY-abstr5-refrel-loopv-it* **where**
INY-abstr5-refrel-loopv-it ω $\mathcal{C}$ *N* δ^r u' v' *c* ≡
br $(\lambda(it,(\omega',\mathcal{C}',N')).\ (it,(\omega',\mathcal{C}')))$
$(\lambda(V',(\omega',\mathcal{C}',N')).$ *INY-abstr5-loopv-N*δ^r*-correct* ω $\mathcal{C}$ *N* δ^r u' v' *c* V' N')

abbreviation *INY-abstr5-refrel-loopv* **where**
INY-abstr5-refrel-loopv ω $\mathcal{C}$ *N* δ^r u' v' Σ' *c* ≡
br $(\lambda(\omega',\mathcal{C}',N').\ (\omega',\mathcal{C}'))$ $(\lambda(\omega',\mathcal{C}',N').$
INY-abstr5-loopc-Nδ^r*-correct* ω $\mathcal{C}$ *N* δ^r u' v' $(\Sigma'-\{c\})$ N')

lemma *INY-abstr5-loopv-N*δ^r*-correct-transfer*:
assumes *INY-abstr2-loopc-invar* ω $\mathcal{C}$ u' v' Σ' $(\omega',\mathcal{C}')$
INY-abstr5-loopv-Nδ^r*-correct* ω $\mathcal{C}$ N' δ^r u' v' *c it b*
shows *INY-abstr5-loopv-N*δ^r*-correct* ω' $\mathcal{C}'$ N' δ^r u' v' *c it b*
apply (*rule INY-abstr5-loopv-N*δ^r*-correctI*)
using *INY-abstr5-loopv-N*δ^r*-correctD*[*OF assms*(*2*)] **apply** (*simp*, *simp*, *simp*)
apply (*subgoal-tac* $\omega'-\mathcal{C}' = \omega-\mathcal{C}$)
using *INY-abstr5-loopv-N*δ^r*-correctD*[*OF assms*(*2*)] **apply** *simp*
using *INY-abstr2-loopc-invarD*(*1−3*,*8*)[*OF assms*(*1*)] **apply** *blast*
using *INY-abstr5-loopv-N*δ^r*-correctD*[*OF assms*(*2*)] **apply** *simp*
done

lemma *INY-abstr5-loopv-N*δ^r*-correct-lift*:
assumes *INY-abstr5-loopc-N*δ^r*-correct* ω $\mathcal{C}$ *N* δ^r u' v' Σ' N'
INY-abstr5-loopv-Nδ^r*-correct* ω $\mathcal{C}$ N' δ^r u' v' *c* {} *b*

shows *INY-abstr5-loopc-N*δ^r*-correct* ω $\mathcal{C}$ N δ^r u' v' $(\Sigma' - \{c\})$ b
apply (*rule INY-abstr5-loopc-N*δ^r*-correctI*)
using *INY-abstr5-loopv-N*δ^r*-correctD*(*1*)[*OF assms*(*2*)] **apply** *simp*
apply (*rename-tac* c' u'' v, *case-tac* $c{=}c'$)
using *INY-abstr5-loopv-N*δ^r*-correctD*(*2*)[*OF assms*(*2*)]
INY-abstr5-loopc-Nδ^r*-correctD*(*2*)[*OF assms*(*1*)] **apply** *simp-all*[*2*]
apply (*rename-tac* c' u'' v, *case-tac* $c{=}c'$)
using *INY-abstr5-loopv-N*δ^r*-correctD*(*2*,*4*,*5*)[*OF assms*(*2*)]
INY-abstr5-loopc-Nδ^r*-correctD*[*OF assms*(*1*)] **apply** *simp-all*[*3*]
done

lemma *INY-abstr5-loopv-correct*:
notes [*simp*] = *br-def*
assumes *INY-abstr2-loopc-invar* ω $\mathcal{C}$ u' v' Σ' $(\omega',\mathcal{C}')$
INY-abstr5-loopc-Nδ^r*-correct* ω $\mathcal{C}$ N δ^r u' v' Σ' N'
INY-abstr1-invar $(\omega,\mathcal{C} \cup \{(u',v')\})$
INY-abstr5-Nδ^r*-correct* ω $(\mathcal{C}\cup\{(u',v')\})$ N δ^r $c\in\Sigma'$ $\Sigma'\subseteq\Sigma$ $\mathcal{A}$
shows *INY-abstr5-loopv* ω' $\mathcal{C}'$ N' δ^r u' v' c $\leq$
$\Downarrow$(*INY-abstr5-refrel-loopv* ω $\mathcal{C}$ N δ^r u' v' Σ' c)
(*INY-abstr4-loopv* ω' $\mathcal{C}'$ u' v' c)
unfolding *INY-abstr5-loopv-def INY-abstr4-loopv-def*
apply (*refine-rcg*
inj-on-id FOREACHi-refine-genR[**where**
R = *INY-abstr5-refrel-loopv-it* ω $\mathcal{C}$ N' δ^r u' v' c]
)
using *INY-abstr5-*δ^r*-correct*(*2*)[*OF assms*(*1*,*4*,*5*,*6*)] **apply** *simp*
apply (*simp add*: *INY-abstr5-*δ^r*-correct*(*2*)[*OF assms*(*1*,*4*,*5*,*6*)]
INY-abstr5-loopv-Nδ^r*-correct-initial*[*OF assms*(*2*,*1*,*5*,*4*)])
using *INY-abstr5-loopv-N*δ^r*-correct-transfer*[*OF assms*(*1*)]
apply (*clarsimp*, *unfold INY-abstr5-loopv-invar-def*, *simp*)
using *assms* **apply** (*force dest*!: *INY-abstr5-loopv-N-eq-0-iff*)
apply *clarsimp*
apply (*rule INY-abstr5-loopu-correct*[*OF assms*(*1*,*4*,*5*,*6*)])
apply (*simp add*: *single-valued-def*)
using *assms* **apply** (*force dest*!: *INY-abstr5-loopv-N-correctness-preserved*)
apply (*simp add*: *single-valued-def*)
using *assms* **apply** (*force dest*!: *INY-abstr5-loopv-N-correctness-preserved*)
using *INY-abstr5-loopv-N*δ^r*-correct-lift*[*OF assms*(*2*)] **apply** *clarsimp*
apply (*simp add*: *single-valued-def*)
done

lemma *INY-abstr5-loopc-N*δ^r*-correct-initial*:
assumes *INY-abstr5-N*δ^r*-correct* ω $(\mathcal{C}\cup\{(u',v')\})$ N δ^r
shows *INY-abstr5-loopc-N*δ^r*-correct* ω $\mathcal{C}$ N δ^r u' v' $(\Sigma\ \mathcal{A})$ N
apply (*rule INY-abstr5-loopc-N*δ^r*-correctI*)
using *INY-abstr5-N*δ^r*-correctD*[*OF assms*] **apply** *simp-all*
done

definition *INY-abstr5-loopc* **where**
INY-abstr5-loopc $\omega\ \mathcal{C}\ N\ \delta^r\ u'\ v' \equiv$ *FOREACH*$^{\textit{INY-abstr5-loopc-invar}\ \omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'}$

$(\Sigma\ \mathcal{A})\ (\lambda c\ (\omega,\ \mathcal{C},\ N).\ do\ \{$
$(\omega',\mathcal{C}',N') \leftarrow$ *INY-abstr5-loopv* $\omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'\ c;$
RETURN $(\omega',\mathcal{C}',N')$
$\})\ (\omega,\ \mathcal{C},\ N)$

abbreviation *INY-abstr5-refrel-loopc-it* **where**
INY-abstr5-refrel-loopc-it $\omega\ \mathcal{C}\ N\ \delta^r\ u'\ v' \equiv$
br $(\lambda(it,(\omega',\mathcal{C}',N')).\ (it,(\omega',\mathcal{C}')))$
$(\lambda(\Sigma',(\omega',\mathcal{C}',N'))$. *INY-abstr5-loopc-N*$\delta^r$*-correct* $\omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'\ \Sigma'\ N')$

abbreviation *INY-abstr5-refrel-loopc* **where**
INY-abstr5-refrel-loopc $\omega\ \mathcal{C}\ N\ \delta^r\ u'\ v' \equiv$
br $(\lambda(\omega',\mathcal{C}',N').\ (\omega',\mathcal{C}'))\ (\lambda(\omega',\mathcal{C}',N')$.
INY-abstr5-Nδ^r*-correct* $\omega'\ \mathcal{C}'\ N'\ \delta^r)$

lemma *INY-abstr5-loopc-N*δ^r*-correct-lift*[*intro*]:
assumes *INY-abstr5-N*δ^r*-correct* $\omega\ (\mathcal{C}\cup\{(u',v')\})\ N\ \delta^r$
INY-abstr5-loopc-Nδ^r*-correct* $\omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'\ \{\}\ N'$
shows *INY-abstr5-N*δ^r*-correct* $\omega\ \mathcal{C}\ N'\ \delta^r$
using *INY-abstr5-loopc-N*δ^r*-correctD*[*OF assms*(*2*)]
INY-abstr5-Nδ^r*-correctD*(*2*)[*OF assms*(*1*)]
by (*intro INY-abstr5-N*δ^r*-correctI*, *simp-all*)

lemma *INY-abstr5-N*δ^r*-correct-transfer*:
assumes *INY-abstr2-loopc-invar* $\omega\ \mathcal{C}\ u'\ v'\ \Sigma'\ (\omega',\mathcal{C}')$
INY-abstr5-Nδ^r*-correct* $\omega\ \mathcal{C}\ N'\ \delta^r$
shows *INY-abstr5-N*δ^r*-correct* $\omega'\ \mathcal{C}'\ N'\ \delta^r$
apply (*rule INY-abstr5-N*δ^r*-correctI*)
using *INY-abstr5-N*δ^r*-correctD*[*OF assms*(*2*)] **apply** *simp*
apply (*subgoal-tac* $\omega{-}\mathcal{C} = \omega'{-}\mathcal{C}'$)
using *INY-abstr5-N*δ^r*-correctD*[*OF assms*(*2*)] **apply** *simp*
using *INY-abstr2-loopc-invarD*(*1*−*3*,*8*)[*OF assms*(*1*)] **apply** *blast*
using *INY-abstr5-N*δ^r*-correctD*[*OF assms*(*2*)] **apply** *simp*
done

lemma *INY-abstr5-loopc-correct*:
notes [*simp*] = *br-def*
assumes *INY-abstr1-invar* $(\omega,\mathcal{C}\cup\{(u',v')\})$
INY-abstr5-Nδ^r*-correct* $\omega\ (\mathcal{C}\cup\{(u',v')\})\ N\ \delta^r$
shows *INY-abstr5-loopc* $\omega\ \mathcal{C}\ N\ \delta^r\ u'\ v' \le$
$\Downarrow$(*INY-abstr5-refrel-loopc* $\omega\ \mathcal{C}\ N\ \delta^r\ u'\ v'$)
(*INY-abstr4-loopc* $\omega\ \mathcal{C}\ u'\ v'$)
unfolding *INY-abstr5-loopc-def INY-abstr4-loopc-def*
apply (*refine-rcg*

inj-on-id FOREACHi-refine-genR[**where** R =
INY-abstr5-refrel-loopc-it ω $\mathcal{C}$ N δ^r u' v']
)
apply *simp*
using *assms INY-abstr5-loopc-Nδ^r-correct-initial* **apply** *simp*
apply (*clarsimp simp add*: *INY-abstr5-loopc-invarI*)
apply *clarsimp*
apply (*rule INY-abstr5-loopv-correct*[*OF - - assms*(*1*,*2*)], *assumption*+)
apply (*simp add*: *single-valued-def*)
using *INY-abstr5-loopc-Nδ^r-correct-lift*[*OF assms*(*2*)] **apply** *clarsimp*
using *INY-abstr5-loopc-Nδ^r-correct-lift*[*OF assms*(*2*)]
INY-abstr5-Nδ^r-correct-transfer **apply** *clarsimp*
apply (*simp add*: *single-valued-def*)
done

lemma *INY-abstr5-loopc-correct'*:
assumes *INY-abstr1-invar* (ω,$\mathcal{C}$)
INY-abstr5-Nδ^r-correct ω $\mathcal{C}$ N δ^r (u',v') $\in \mathcal{C}$
shows *INY-abstr5-loopc* ω ($\mathcal{C}-\{(u',v')\}$) N δ^r u' v' $\leq$
$\Downarrow$(*INY-abstr5-refrel-loopc* ω ($\mathcal{C}-\{(u',v')\}$) N δ^r u' v')
(*INY-abstr4-loopc* ω ($\mathcal{C}-\{(u',v')\}$) u' v')
proof−
have $\mathcal{C} - \{(u',v')\} \cup \{(u',v')\} = \mathcal{C}$ **using** *assms*(*3*) **by** *blast*
thus *?thesis* **using** *INY-abstr5-loopc-correct assms* **by** *simp*
qed

definition *INY-abstr5'* **where**
INY-abstr5' ω $\mathcal{C}$ N δ^r $\equiv$ $WHILE_T{}^{INY\text{-}abstr5\text{-}invar\ \delta^r}$ (λ(ω, $\mathcal{C}$, N). $\mathcal{C}\neq\{\}$)
(λ(ω, $\mathcal{C}$, N). *do* {
ASSERT ($\mathcal{C}\neq\{\}$);
(u',v') $\leftarrow$ *SPEC* (λ(u',v'). (u',v')$\in\mathcal{C}$);
let $\mathcal{C} = \mathcal{C} - \{(u',v')\}$;
(ω,$\mathcal{C}$,N) $\leftarrow$ *INY-abstr5-loopc* ω $\mathcal{C}$ N δ^r u' v';
RETURN (ω, $\mathcal{C}$, N)
}) (ω, $\mathcal{C}$, N)

abbreviation *INY-abstr5-refrel* **where**
INY-abstr5-refrel δ^r $\equiv$ *br* (λ(ω,$\mathcal{C}$,-). (ω,$\mathcal{C}$)) (λ(ω,$\mathcal{C}$,N).
INY-abstr5-Nδ^r-correct ω $\mathcal{C}$ N δ^r)

lemma *INY-abstr5'-correct*:
notes [*simp*] = *br-def*
assumes *INY-is-valid-initial-counter* N
INY-abstr5-δ^r-correct δ^r ω = *INY-initial*
shows *INY-abstr5'* ω ω N δ^r $\leq$ $\Downarrow$(*INY-abstr5-refrel* δ^r) *INY-abstr4*
unfolding *INY-abstr5'-def INY-abstr4-def*
apply (*refine-rcg*)

using *assms* **unfolding** *INY-is-valid-initial-counter-def*
  apply (*simp*, *intro INY-abstr5-N*δ^r*-correctI*, *simp*, *simp*, *fast*, *simp*)
apply *simp*
using *INY-abstr5-invarI* **apply** *clarsimp*
apply *simp*
apply *simp*
apply *simp*
apply *clarsimp*
apply (*rule INY-abstr5-loopc-correct'*, *assumption+*)
apply *simp*
done

abbreviation *INY-is-empty-d* **where**
INY-is-empty-d d $\equiv$ *dom d* $=$ $\mathcal{Q}$ $\mathcal{A}\times\Sigma$ $\mathcal{A}$ $\wedge$ ($\forall\,(q,a)\in$*dom d*. *d*(q,a) $=$ *Some* (*0*::*nat*))
abbreviation *INY-is-empty-*δ^r **where**
INY-is-empty-δ^r δ^r $\equiv$ *dom* δ^r $=$ $\mathcal{Q}$ $\mathcal{A}\times\Sigma$ $\mathcal{A}$ $\wedge$
  ($\forall\,(q,a)\in$*dom* δ^r. $\delta^r(q,a)$ $=$ *Some* ({}::'*q set*))

definition *INY-abstr5* **where**
INY-abstr5 $\equiv$ *do* {
  (d,δ^r) $\leftarrow$ *SPEC* ($\lambda(d,\delta^r)$. *INY-is-empty-d d* $\wedge$ *INY-is-empty-*δ^r δ^r);
  (d,δ^r) $\leftarrow$ *SPEC* ($\lambda(d,\delta^r)$. *INY-abstr5-d-correct d* $\wedge$ *INY-abstr5-*δ^r*-correct* δ^r);
  N $\leftarrow$ *SPEC* (λN. *INY-is-valid-initial-counter N*);
  $(\omega,\mathcal{C})$ $\leftarrow$ *SPEC* ($\lambda(\omega,\mathcal{C})$. ω $=$ *INY-initial* $\wedge$ $\mathcal{C}$ $=$ *INY-initial*);
  $(\omega,\mathcal{C},N)$ $\leftarrow$ *INY-abstr5'* ω $\mathcal{C}$ N δ^r;
  RETURN $(\omega,\mathcal{C})$
}

lemma *INY-abstr5-correct*: *INY-abstr5* $\leq$ $\Downarrow$(*Id*) *INY-abstr4*
  unfolding *INY-abstr5-def*
  apply *refine-rcg*
  using *INY-abstr5'-correct*
  apply (*simp add*: *pw-le-iff refine-pw-simps br-def*)
  apply *force*
  done

5.3.4 Implementation of the initialisation

INY-abstr6-empty-Ndδ^r returns N and d filled with 0 and δ^r filled with the empty set for each value in their respective domains.

abbreviation *INY-abstr6-empty-d*δ^r*-invar-loopc* **where**
INY-abstr6-empty-dδ^r*-invar-loopc* Σ' $\equiv$ $\lambda(d,\delta^r)$.
  dom d $=$ $\mathcal{Q}$ $\mathcal{A}\times(\Sigma$ $\mathcal{A}-\Sigma')$ $\wedge$ ($\forall\,x\in$*dom d*. *d x* $=$ *Some* (*0*::*nat*)) $\wedge$
  dom δ^r $=$ $\mathcal{Q}$ $\mathcal{A}\times(\Sigma$ $\mathcal{A}-\Sigma')$ $\wedge$ ($\forall\,x\in$*dom* δ^r. δ^r *x* $=$ *Some* ({}::'*q set*))

abbreviation *INY-abstr6-empty-d*δ^r*-invar-loopu* **where**
INY-abstr6-empty-dδ^r*-invar-loopu d* δ^r *c U'* $\equiv$ $\lambda(d',\delta^{r}{}')$.

dom d' = dom d $\cup$ ($\mathcal{Q}$ $\mathcal{A}$$-$*U'*)$\times${*c*} $\wedge$ ($\forall$*x*$\in$*dom d'*. *d' x* = *Some* (*0::nat*)) $\wedge$
dom $\delta^r{}'$ = *dom* δ^r $\cup$ ($\mathcal{Q}$ $\mathcal{A}$$-$*U'*)$\times${*c*} $\wedge$ ($\forall$*x*$\in$*dom* $\delta^r{}'$. $\delta^r{}'$ *x* = *Some* ({}::*'q set*))

definition *INY-abstr6-empty-d*δ^r **where**
INY-abstr6-empty-dδ^r $\equiv$ *FOREACH*$^{\textit{INY-abstr6-empty-d}\delta^r\textit{-invar-loopc}}$ (Σ $\mathcal{A}$) (λ*c* (*d*,δ^r).
FOREACH$^{\textit{INY-abstr6-empty-d}\delta^r\textit{-invar-loopu d } \delta^r \textit{ c}}$ ($\mathcal{Q}$ $\mathcal{A}$) (λ*u* (*d*,δ^r).
RETURN (*d*((*u*,*c*) $\mapsto$ *0::nat*), δ^r((*u*,*c*) $\mapsto$ {}::*'q set*))
) (*d*,δ^r)
) (*Map.empty*, *Map.empty*)

lemma *INY-abstr6-empty-d*δ^r*-correct*:
INY-abstr6-empty-dδ^r $\leq$ *SPEC* (λ(*d*,δ^r). *INY-is-empty-d d* $\wedge$
INY-is-empty-δ^r δ^r)
unfolding *INY-abstr6-empty-d*δ^r*-def*
apply (*intro refine-vcg*)
apply (*simp-all add*: *finite-*$\mathcal{Q}$ *finite-*Σ)[*5*]
apply (*clarsimp*, *blast*)
apply (*clarsimp*, *blast*)
apply (*simp add*: *INY-is-valid-initial-counter-def*)
done

INY-abstr6-init-dδ^r fills $d(u,c)$ with $|\delta(u,c)|$ for all $u\in\mathcal{Q}$, $c\in\Sigma$ and $\delta^r(u,c)$ with all $\delta^{-1}(u,c)$ for all $u\in\mathcal{Q}$, $c\in\Sigma$, i.e. the set $\{u'.\ (u',c,u)\in\Delta\}$.

definition *INY-abstr6-init-d*δ^r*-invar* **where**
INY-abstr6-init-dδ^r*-invar* Δ' $\equiv$ λ(*d*,δ^r). (*dom d* = $\mathcal{Q}$ $\mathcal{A}$$\times$$\Sigma$ $\mathcal{A}$) $\wedge$
($\forall$(*v*,*c*)$\in$*dom d*. *d* (*v*,*c*) = *Some* (*card* {*v'*. (*v*,*c*,*v'*)$\in$$\Delta$ $\mathcal{A}$$-$$\Delta'$})) $\wedge$
(*dom* δ^r = $\mathcal{Q}$ $\mathcal{A}$$\times$$\Sigma$ $\mathcal{A}$) $\wedge$
($\forall$(*v*,*c*)$\in$*dom* δ^r. δ^r (*v*,*c*) = *Some* {*v'*. (*v'*,*c*,*v*)$\in$$\Delta$ $\mathcal{A}$$-$$\Delta'$})

lemma *INY-abstr6-init-d*δ^r*-invarI*[*intro*]:
assumes *dom d* = $\mathcal{Q}$ $\mathcal{A}$$\times$$\Sigma$ $\mathcal{A}$
$\bigwedge$*v c*. (*v*,*c*) $\in$ $\mathcal{Q}$ $\mathcal{A}$$\times$$\Sigma$ $\mathcal{A}$ $\Longrightarrow$ *d* (*v*,*c*) =
Some (*card* {*v'*. (*v*,*c*,*v'*)$\in$$\Delta$ $\mathcal{A}$$-$$\Delta'$})
dom δ^r = $\mathcal{Q}$ $\mathcal{A}$$\times$$\Sigma$ $\mathcal{A}$
$\bigwedge$*v c*. (*v*,*c*) $\in$ $\mathcal{Q}$ $\mathcal{A}$$\times$$\Sigma$ $\mathcal{A}$ $\Longrightarrow$ δ^r (*v*,*c*) = *Some* {*v'*. (*v'*,*c*,*v*)$\in$$\Delta$ $\mathcal{A}$$-$$\Delta'$}
shows *INY-abstr6-init-d*δ^r*-invar* Δ' (*d*,δ^r)
using *assms* **unfolding** *INY-abstr6-init-d*δ^r*-invar-def* **by** *simp*

lemma *INY-abstr6-init-d*δ^r*-invarD*:
assumes *INY-abstr6-init-d*δ^r*-invar* Δ' (*d*,δ^r)
shows *dom d* = $\mathcal{Q}$ $\mathcal{A}$$\times$$\Sigma$ $\mathcal{A}$
$\bigwedge$*v c*. (*v*,*c*) $\in$ $\mathcal{Q}$ $\mathcal{A}$$\times$$\Sigma$ $\mathcal{A}$ $\Longrightarrow$ *d* (*v*,*c*) = *Some* (*card* {*v'*. (*v*,*c*,*v'*)$\in$$\Delta$ $\mathcal{A}$$-$$\Delta'$})
dom δ^r = $\mathcal{Q}$ $\mathcal{A}$$\times$$\Sigma$ $\mathcal{A}$
$\bigwedge$*v c*. (*v*,*c*) $\in$ $\mathcal{Q}$ $\mathcal{A}$$\times$$\Sigma$ $\mathcal{A}$ $\Longrightarrow$ δ^r (*v*,*c*) = *Some* {*v'*. (*v'*,*c*,*v*)$\in$$\Delta$ $\mathcal{A}$$-$$\Delta'$}
using *assms* **unfolding** *INY-abstr6-init-d*δ^r*-invar-def* **by** *blast+*

abbreviation *INY-abstr6-inc-d* **where**

```
INY-abstr6-inc-d d v c ≡
  (case d (v,c) of Some n ⇒ d((v,c) ↦ n+(1::nat)) | None ⇒ d)

abbreviation INY-abstr6-update-δʳ where
INY-abstr6-update-δʳ δʳ v' c v ≡
  (case δʳ (v,c) of Some A ⇒ δʳ((v,c) ↦ insert v' A) | None ⇒ δʳ)
```

The initialisation of d and δ^r

```
definition INY-abstr6-init-dδʳ where
INY-abstr6-init-dδʳ d δʳ = FOREACH^{INY-abstr6-init-dδʳ-invar} (Δ 𝒜)
  (λ(v,c,v') (d,δʳ). RETURN (INY-abstr6-inc-d d v c,
                            INY-abstr6-update-δʳ δʳ v c v'))
  (d, δʳ)

lemma INY-abstr6-init-dδʳ-correct:
  assumes dom d = 𝒬 𝒜×Σ 𝒜 ∧ (∀x∈dom d. d x = Some 0)
          dom δʳ = 𝒬 𝒜×Σ 𝒜 ∧ (∀x∈dom δʳ. δʳ x = Some {})
  shows INY-abstr6-init-dδʳ d δʳ ≤
      SPEC (λ(d,δʳ). INY-abstr5-d-correct d ∧ INY-abstr5-δʳ-correct δʳ)
unfolding INY-abstr6-init-dδʳ-def
proof (intro refine-vcg)
  show finite (Δ 𝒜) using finite-𝒬 finite-Σ Δ-consistent
      rev-finite-subset[of 𝒬 𝒜×Σ 𝒜×𝒬 𝒜   Δ 𝒜] by force
next
  show INY-abstr6-init-dδʳ-invar (Δ 𝒜) (d,δʳ)
      using assms by (intro INY-abstr6-init-dδʳ-invarI, simp-all)
next
  case (goal3 vcv' Δ' dδʳ)
    thus ?case proof (cases vcv', cases dδʳ, clarsimp)
    case (goal1 v c v' d δʳ)
    note invar = INY-abstr6-init-dδʳ-invarD[OF goal1(2)]
    let ?d' = INY-abstr6-inc-d d v c
    let ?δʳ' = INY-abstr6-update-δʳ δʳ v c v'

    let ?X1 = {v''. (v,c,v'')∈Δ 𝒜−Δ'}
    let ?X2 = {v''. (v,c,v'')∈Δ 𝒜−(Δ'−{(v,c,v')})}
    have ?X2 = ?X1 ∪ {v'}    v'∉ ?X1 using goal1 by blast+
    moreover have ?X2 ⊆ 𝒬 𝒜 using Δ-consistent by blast+
    hence finite ?X2 using rev-finite-subset[OF finite-𝒬] by blast+
    ultimately have new-card: card ?X2 = card ?X1 + 1 by simp

    let ?Y1 = {v''. (v'',c,v')∈Δ 𝒜−Δ'}
    let ?Y2 = {v''. (v'',c,v')∈Δ 𝒜−(Δ'−{(v,c,v')})}
    have new-set: ?Y2 = ?Y1 ∪ {v} using goal1(1,3) by blast

    have d (v,c) = Some (card {v'. (v,c,v')∈Δ 𝒜−Δ'})
      using goal1(1,3) invar(2) Δ-consistent by blast
    moreover have δʳ (v',c) = Some {v''. (v'',c,v')∈Δ 𝒜−Δ'}
      using goal1(1,3) invar(4) Δ-consistent by blast
```

ultimately show *INY-abstr6-init-dδ^r-invar* $(\Delta'-\{(v,c,v')\})$ $(?d',?\delta^r{}')$
using *goal1*(*1*,*2*) **apply** (*intro INY-abstr6-init-dδ^r-invarI*)
using *invar*(*1*) *Δ-consistent* **apply** (*simp*, *blast*)
using *invar*(*2*) *new-card* **apply** *fastforce*
using *invar*(*3*) *Δ-consistent* **apply** (*simp*, *blast*)
using *invar*(*4*) *new-set* **apply** *fastforce*
done
qed

next
case *goal4* **thus** *?case*
proof (*clarsimp*)
fix d δ^r **assume** *INY-abstr6-init-dδ^r-invar* $\{\}$ (d,δ^r)
thus *INY-abstr5-d-correct* d $\wedge$ *INY-abstr5-δ^r-correct* δ^r
unfolding *INY-abstr5-d-correct-def INY-abstr5-δ^r-correct-def*
INY-abstr6-init-dδ^r-invar-def **by** *simp*
qed
qed

The initialisation of N.

abbreviation *INY-abstr6-init-N-invar-loopc* **where**
INY-abstr6-init-N-invar-loopc Σ' $N \equiv (dom\ N = (\Sigma\ \mathcal{A}-\Sigma')\times\mathcal{Q}\ \mathcal{A}\times\mathcal{Q}\ \mathcal{A}\ \wedge$
$(\forall (c,u,v)\in dom\ N.\ N(c,u,v) = Some\ (card\ \{v'.\ (v,c,v')\in\Delta\ \mathcal{A}\})))$

abbreviation *INY-abstr6-init-N-invar-loopu* **where**
INY-abstr6-init-N-invar-loopu N c U' $N' \equiv$
$(dom\ N' = dom\ N \cup \{c\}\times(\mathcal{Q}\ \mathcal{A}-U')\times\mathcal{Q}\ \mathcal{A}\ \wedge$
$(\forall (c,u,v)\in dom\ N'.\ N'(c,u,v) = Some\ (card\ \{v'.\ (v,c,v')\in\Delta\ \mathcal{A}\})))$

abbreviation *INY-abstr6-init-N-invar-loopv* **where**
INY-abstr6-init-N-invar-loopv N c u V' $N' \equiv (dom\ N' =$
$dom\ N \cup \{c\}\times\{u\}\times(\mathcal{Q}\ \mathcal{A}-V')\ \wedge$
$(\forall (c,u,v)\in dom\ N'.\ N'(c,u,v) = Some\ (card\ \{v'.\ (v,c,v')\in\Delta\ \mathcal{A}\})))$

definition *INY-abstr6-init-N* **where**
INY-abstr6-init-N $d \equiv FOREACH^{INY\text{-}abstr6\text{-}init\text{-}N\text{-}invar\text{-}loopc}\ (\Sigma\ \mathcal{A})\ (\lambda c\ N.$
$FOREACH^{INY\text{-}abstr6\text{-}init\text{-}N\text{-}invar\text{-}loopu\ N\ c}\ (\mathcal{Q}\ \mathcal{A})\ (\lambda u\ N.$
$FOREACH^{INY\text{-}abstr6\text{-}init\text{-}N\text{-}invar\text{-}loopv\ N\ c\ u}\ (\mathcal{Q}\ \mathcal{A})\ (\lambda v\ N.\ do\ \{$
$ASSERT\ (d(v,c) \neq None);$
$RETURN\ (N((c,u,v) \mapsto (the\ (d(v,c)))::nat))$
$\}$
$)\ N$
$)\ N$
$)\ Map.empty$

thm *INY-abstr5-d-correctD*
lemma *INY-abstr5-d-correctD-ne-None-aux*:
assumes C: *INY-abstr5-d-correct* d
assumes M: $v\in\mathcal{Q}\ \mathcal{A}$ $\quad c\in\Sigma\ \mathcal{A}$

shows $d\ (v,c) \neq None$
using *INY-abstr5-d-correctD*[*OF C*] *assms*
by *blast*

lemma *INY-abstr6-init-N-correct*:
assumes *INY-abstr5-d-correct d*
shows *INY-abstr6-init-N d* $\leq$ *SPEC INY-is-valid-initial-counter*
unfolding *INY-abstr6-init-N-def*
apply (*intro refine-vcg*)
apply (*simp-all add*: *finite-*Σ *finite-*$\mathcal{Q}$)[*6*]
apply (*rule INY-abstr5-d-correctD-ne-None-aux*[*OF assms*], (*erule* (*1*) *set-mp*)+)
[]
defer
using *INY-abstr5-d-correctD*[*OF assms*] **apply** *auto*[*2*]
apply (*simp add*: *INY-is-valid-initial-counter-def*)
proof−
thm *INY-abstr5-d-correctD*[*OF assms*, *simplified*, *rule-format*]
case ($goal8\ c\ \Sigma'\ N\ u\ U'\ N'\ v\ V'\ N''$)
def $N''' == N''((c,\ u,\ v) \mapsto the\ (d\ (v,\ c)))$
note *this*[*THEN meta-eq-to-obj-eq*]
also have $(v,c) \in \mathcal{Q}\ \mathcal{A} \times \Sigma\ \mathcal{A}$ **using** *goal8*(*1*,*2*,*7*,*8*) **by** *blast*
hence $d\ (v,\ c) = Some\ (card\ \{v'.\ (v,c,v')\in\Delta\ \mathcal{A}\})$
using *INY-abstr5-d-correctD*[*OF assms*] **by** *simp*
finally have A: $N''' = N''((c,u,v)\mapsto(card\ \{v'.\ (v,c,v')\in\Delta\ \mathcal{A}\}))$ **by** *force*
have $dom\ N''' = dom\ N' \cup \{c\} \times \{u\} \times (\mathcal{Q}\ \mathcal{A} - (V' - \{v\}))$
using *goal8*(*1*−*9*) **by** (*subst A*, *auto*)
moreover have $\bigwedge n\ c\ u\ v.\ N''\ (c,u,v) = Some\ n \Longrightarrow$
$N''\ (c,u,v) = Some\ (card\ \{v'.\ (v,c,v')\in\Delta\ \mathcal{A}\})$ **using** *goal8*(*9*) **by** *fast*
hence C: $\bigwedge n\ c\ u\ v.\ N''\ (c,u,v) = Some\ n \Longrightarrow$
$n = (card\ \{v'.\ (v,c,v')\in\Delta\ \mathcal{A}\})$ **by** *simp*
have $\forall (c,\ u,\ v)\in dom\ N'''.\ N'''\ (c,\ u,\ v) =$
$Some\ (card\ \{v'.\ (v,\ c,\ v') \in \Delta\ \mathcal{A}\})$ **by** (*clarsimp simp add*: *A C*)
ultimately show *?case* **by** (*simp add*: N'''*-def*)
qed

Now we compute the nontrivial part of the initial ω, i.e. $\{(u,v)|\ \exists\, c.\ \delta(u,c) \neq \{\} \wedge \delta(v,c) = \{\}\}$ For this, we iterate over all c, u, v with for each loops, checking the $\delta(u,c)$ / $\delta(v,c)$ conditions as soon as possible.

definition *INY-abstr6-init-*$\omega\mathcal{C}$*-loopc-invar* **where**
INY-abstr6-init-$\omega\mathcal{C}$*-loopc-invar* $(d::('q\times'a)\rightharpoonup nat)\ \Sigma' \equiv \lambda(\omega,\mathcal{C}).$
$(\omega = \{(u,v)\ |u\ v\ c.\ u\in\mathcal{Q}\ \mathcal{A} \wedge v\in\mathcal{Q}\ \mathcal{A} \wedge c\in(\Sigma\ \mathcal{A}-\Sigma') \wedge$
$d\ (u,c)\neq Some\ 0 \wedge d\ (v,c)=Some\ 0\} \wedge \mathcal{C} = \omega)$

definition *INY-abstr6-init-*$\omega\mathcal{C}$*-loopu-invar* **where**
INY-abstr6-init-$\omega\mathcal{C}$*-loopu-invar* $(d::('q\times'a)\rightharpoonup nat)\ c\ \omega\ U' \equiv \lambda(\omega',\mathcal{C}').$
$(\omega' = \omega \cup \{(u,v)\ |u\ v.\ u\in\mathcal{Q}\ \mathcal{A}-U' \wedge v\in\mathcal{Q}\ \mathcal{A} \wedge$
$d\ (u,c)\neq Some\ 0 \wedge d\ (v,c)=Some\ 0\} \wedge \mathcal{C}' = \omega')$

definition *INY-abstr6-init-*$\omega\mathcal{C}$*-loopv-invar* **where**

INY-abstr6-init-ωC-loopv-invar $(d::('q \times 'a) \rightharpoonup nat)\ c\ u\ \omega\ V' \equiv \lambda(\omega',\mathcal{C}').$
$(\omega' = \omega \cup \{(u,v)\ |v.\ v \in \mathcal{Q}\ \mathcal{A} - V' \wedge d\ (v,c) = Some\ 0\} \wedge \mathcal{C}' = \omega')$

definition *INY-abstr6-init-ωC-loopv* **where**
INY-abstr6-init-ωC-loopv $d\ c\ u\ \omega\ \mathcal{C} \equiv$
$FOREACH^{INY\text{-}abstr6\text{-}init\text{-}\omega\mathcal{C}\text{-}loopv\text{-}invar\ d\ c\ u\ \omega}\ (\mathcal{Q}\ \mathcal{A})$
$(\lambda v\ (\omega',\mathcal{C}').$ *if* $d\ (v,\ c) = Some\ 0$ *then*
RETURN $(insert\ (u,v)\ \omega',\ insert\ (u,v)\ \mathcal{C}')$
else RETURN $(\omega',\mathcal{C}'))\ (\omega,\mathcal{C})$

definition *INY-abstr6-init-ωC-loopu* **where**
INY-abstr6-init-ωC-loopu $d\ c\ \omega\ \mathcal{C} \equiv$
$FOREACH^{INY\text{-}abstr6\text{-}init\text{-}\omega\mathcal{C}\text{-}loopu\text{-}invar\ d\ c\ \omega}\ (\mathcal{Q}\ \mathcal{A})$
$(\lambda u\ (\omega',\mathcal{C}').$ *if* $d\ (u,\ c) \neq Some\ 0$
then INY-abstr6-init-ωC-loopv $d\ c\ u\ \omega'\ \mathcal{C}'$
else RETURN $(\omega',\mathcal{C}'))$
$(\omega,\mathcal{C})$

definition *INY-abstr6-init-ωC-loopc* **where**
INY-abstr6-init-ωC-loopc $d \equiv$
$FOREACH^{INY\text{-}abstr6\text{-}init\text{-}\omega\mathcal{C}\text{-}loopc\text{-}invar\ d}\ (\Sigma\ \mathcal{A})$
$(\lambda c\ (\omega,\mathcal{C}).$ *INY-abstr6-init-ωC-loopu* $d\ c\ \omega\ \mathcal{C})$
$(\{\},\ \{\})$

This computes the initial ω and $\mathcal{C}$ using the precomputed $|\delta(u,c)|$ values.

definition *INY-abstr6-init-ωC* **where**
INY-abstr6-init-ωC $d \equiv$ *do* {
$(\omega,\mathcal{C}) \leftarrow$ *INY-abstr6-init-ωC-loopc* d;
let $FN = \mathcal{F}\ \mathcal{A} \times (\mathcal{Q}\ \mathcal{A} - \mathcal{F}\ \mathcal{A})$;
ASSERT $(\omega \cup FN = $ *INY-initial*$)$;
ASSERT $(\mathcal{C} \cup FN = $ *INY-initial*$)$;
RETURN $(\omega \cup FN,\ \mathcal{C} \cup FN)$
}

lemma *INY-abstr6-init-ωC-loopv-correct*:
assumes $u \in U'$ $\quad U' \subseteq \mathcal{Q}\ \mathcal{A}$ $\quad$ *INY-abstr6-init-ωC-loopu-invar* $d\ c\ \omega\ U'\ (\omega',\mathcal{C}')$
$d\ (u,c) \neq Some\ 0$
shows *INY-abstr6-init-ωC-loopv* $d\ c\ u\ \omega'\ \mathcal{C}' \leq SPEC$
(*INY-abstr6-init-ωC-loopu-invar* $d\ c\ \omega\ (U' - \{u\}))$
proof−
let $?T = \{(u,v)\ |v.\ v \in \mathcal{Q}\ \mathcal{A} \wedge d(v,c) = Some\ 0\}$
from *assms(3)* **have** [*simp*]: $\mathcal{C}' = \omega'$
unfolding *INY-abstr6-init-ωC-loopu-invar-def* **by** *simp*
hence *INY-abstr6-init-ωC-loopv* $d\ c\ u\ \omega'\ \mathcal{C}' \leq$
$SPEC\ (\lambda(\omega'',\mathcal{C}'').\ \omega'' = \omega' \cup ?T \wedge \mathcal{C}'' = \omega'')$
unfolding *INY-abstr6-init-ωC-loopv-def*
by (*intro refine-vcg*, *auto simp add*:
INY-abstr6-init-ωC-loopv-invar-def finite-Q)
also have *INY-abstr6-init-ωC-loopu-invar* $d\ c\ \omega\ (U' - \{u\})\ (\omega' \cup ?T,\ \mathcal{C}' \cup ?T)$

using *assms* **unfolding** *INY-abstr6-init-ωC-loopu-invar-def* **by** *blast*
hence *SPEC* $(\lambda(\omega'',\mathcal{C}'').\ \omega'' = \omega' \cup ?T \wedge \mathcal{C}'' = \omega'') \leq$ *SPEC* (
INY-abstr6-init-ωC-loopu-invar d c ω $(U'-\{u\})$) **using** *SPEC-rule* **by** *force*
finally show *?thesis* **.**
qed

lemma *INY-abstr6-init-ωC-loopu-correct*:
assumes $c \in \Sigma'$ $\Sigma' \subseteq \Sigma\ \mathcal{A}$ *INY-abstr6-init-ωC-loopc-invar d* Σ' $(\omega',\mathcal{C}')$
shows *INY-abstr6-init-ωC-loopu d c* $\omega'\ \mathcal{C}' \leq$ *SPEC*
(*INY-abstr6-init-ωC-loopc-invar d* $(\Sigma'-\{c\})$)
proof−
let $?T = \{(u,v)\ |u\ v.\ u \in \mathcal{Q}\ \mathcal{A} \wedge v \in \mathcal{Q}\ \mathcal{A} \wedge d(u,c) \neq Some\ 0 \wedge d(v,c) = Some\ 0\}$
from *assms*(*3*) **have** [*simp*]: $\mathcal{C}' = \omega'$
unfolding *INY-abstr6-init-ωC-loopc-invar-def* **by** *simp*
have *INY-abstr6-init-ωC-loopu d c* $\omega'\ \mathcal{C}' \leq$
SPEC $(\lambda(\omega'',\mathcal{C}'').\ \omega'' = \omega' \cup ?T \wedge \mathcal{C}'' = \omega'')$
unfolding *INY-abstr6-init-ωC-loopu-def*
apply (*intro refine-vcg finite-Q*)
apply (*simp add*: *INY-abstr6-init-ωC-loopu-invar-def*)
apply (*intro INY-abstr6-init-ωC-loopv-correct*)
apply (*auto simp add*: *INY-abstr6-init-ωC-loopu-invar-def*)
done
also have *INY-abstr6-init-ωC-loopc-invar d* $(\Sigma' - \{c\})$ $(\omega' \cup ?T,\ \mathcal{C}' \cup ?T)$
using *assms* **unfolding** *INY-abstr6-init-ωC-loopc-invar-def* **by** *blast*
hence *SPEC* $(\lambda(\omega'',\mathcal{C}'').\ \omega'' = \omega' \cup ?T \wedge \mathcal{C}'' = \omega'') \leq$
SPEC (*INY-abstr6-init-ωC-loopc-invar d* $(\Sigma'-\{c\})$) **using** *SPEC-rule* **by** *auto*
finally show *?thesis* **.**
qed

lemma *INY-abstr6-init-ω-loopc-correct*:
INY-abstr6-init-ωC-loopc d $\leq$ *SPEC* $(\lambda(\omega,\mathcal{C}).\ \omega = \{(u,v)\ |c\ u\ v.$
$u \in \mathcal{Q}\ \mathcal{A} \wedge v \in \mathcal{Q}\ \mathcal{A} \wedge c \in \Sigma\ \mathcal{A} \wedge d(u,c) \neq Some\ 0 \wedge d(v,c) = Some\ 0\} \wedge \mathcal{C} = \omega)$
unfolding *INY-abstr6-init-ωC-loopc-def INY-initial-def*
apply (*intro refine-vcg finite-Σ*)
apply (*simp add*: *INY-abstr6-init-ωC-loopc-invar-def*)
apply (*intro INY-abstr6-init-ωC-loopu-correct*)
apply (*auto simp*: *INY-abstr6-init-ωC-loopc-invar-def*)
done

lemma *INY-abstr6-init-ωC-loopc-correct-aux*:
assumes *INY-abstr5-d-correct d*
shows $\mathcal{F}\ \mathcal{A} \times (\mathcal{Q}\ \mathcal{A} - \mathcal{F}\ \mathcal{A}) \cup \{(u,v)\ |c\ u\ v.\ u \in \mathcal{Q}\ \mathcal{A} \wedge v \in \mathcal{Q}\ \mathcal{A} \wedge$
$c \in \Sigma\ \mathcal{A} \wedge d(u,c) \neq Some\ 0 \wedge d(v,c) = Some\ 0\}$ = *INY-initial*
(**is** $?A \cup ?B$ = *INY-initial*)
proof−
let $?C = \{(u,v).\ u \in \mathcal{Q}\ \mathcal{A} \wedge v \in \mathcal{Q}\ \mathcal{A} \wedge$
$(\exists\ c\ u'.\ (u,c,u') \in \Delta\ \mathcal{A} \wedge \neg(\exists\ v'.(v,c,v') \in \Delta\ \mathcal{A}))\}$
{
fix *q c* **assume** $q \in \mathcal{Q}\ \mathcal{A}$ $c \in \Sigma\ \mathcal{A}$

hence $d(q,c) = Some\ (card\ \{q'.\ (q,c,q') \in \Delta\ \mathcal{A}\})$ **using** *assms*
unfolding *INY-abstr5-d-correct-def* **by** *blast*
also have $\{q'.\ (q,c,q') \in \Delta\ \mathcal{A}\} \subseteq \mathcal{Q}\ \mathcal{A}$ **using** Δ-*consistent* **by** *blast*
hence *finite* $\{q'.\ (q,c,q') \in \Delta\ \mathcal{A}\}$
using *rev-finite-subset finite-*$\mathcal{Q}$ **by** *blast*
hence $Some\ (card\ \{q'.\ (q,c,q') \in \Delta\ \mathcal{A}\}) = Some\ 0 \longleftrightarrow$
$\neg(\exists\, q'.\ (q,c,q') \in \Delta\ \mathcal{A})$ **by** *simp*
finally have $(d\ (q,c) = Some\ 0) = (\neg(\exists\, q'.\ (q,c,q') \in \Delta\ \mathcal{A}))$ **.**
}

with Δ-*consistent* **have** $?B = ?C$ **by** *blast*
thus *?thesis* **unfolding** *INY-initial-def* **by** *simp*
qed

lemma *INY-abstr6-init-ωC-loopc-correct′*:
assumes *INY-abstr5-d-correct d*
shows *INY-abstr6-init-ωC-loopc d* $\leq$
$SPEC\ (\lambda(\omega,\mathcal{C}).\ \mathcal{F}\ \mathcal{A} \times (\mathcal{Q}\ \mathcal{A} - \mathcal{F}\ \mathcal{A}) \cup \omega = INY\text{-}initial \wedge \mathcal{C} = \omega)$
proof−
let $?\omega_1{}' = \mathcal{F}\ \mathcal{A} \times (\mathcal{Q}\ \mathcal{A} - \mathcal{F}\ \mathcal{A})$ **and**
$?\omega_2{}' = \{(u,v)\ |c\ u\ v.\ u \in \mathcal{Q}\ \mathcal{A} \wedge v \in \mathcal{Q}\ \mathcal{A} \wedge c \in \Sigma\ \mathcal{A} \wedge$
$d(u,\ c) \neq Some\ 0 \wedge d(v,c) = Some\ 0\}$
note *INY-abstr6-init-ω-loopc-correct*[*of d*]
also have $SPEC\ (\lambda(\omega,\mathcal{C}).\ \omega = ?\omega_2{}' \wedge \mathcal{C} = \omega) \leq$
$SPEC\ (\lambda(\omega,\mathcal{C}).\ ?\omega_1{}' \cup \omega = ?\omega_1{}' \cup ?\omega_2{}' \wedge \mathcal{C} = \omega)$
by (*rule SPEC-rule*, *force*)
also note *INY-abstr6-init-ωC-loopc-correct-aux*[*OF assms*]
finally show *?thesis* **.**
qed

lemma *INY-abstr6-init-ωC-correct*:
assumes *INY-abstr5-d-correct d*
shows *INY-abstr6-init-ωC d* $\leq$
$SPEC\ (\lambda(\omega,\mathcal{C}).\ \omega = INY\text{-}initial \wedge \mathcal{C} = INY\text{-}initial)$
unfolding *INY-abstr6-init-ωC-def*
apply (*intro refine-vcg*)
apply (*rule order-trans*[*OF INY-abstr6-init-ωC-loopc-correct′*[*OF assms*]])
apply (*intro refine-vcg*)
apply (*auto simp*: *Let-def*)
done

The final version of the abstract algorithm in which all operations have been implemented. The only SPEC remaining is the one that obtains a $(u',\ v')$ from $\mathcal{C}$.

definition *INY-abstr6* **where**
INY-abstr6 ≡ *do* {
$(d,\delta^r) \leftarrow INY\text{-}abstr6\text{-}empty\text{-}d\delta^r$;
$(d,\delta^r) \leftarrow INY\text{-}abstr6\text{-}init\text{-}d\delta^r\ d\ \delta^r$;

N ← *INY-abstr6-init-N d*;
(ω,𝒞) ← *INY-abstr6-init-ω𝒞 d*;
(ω,𝒞,*N*) ← *INY-abstr5′ ω 𝒞 N* δ^r;
RETURN (ω,𝒞)
}

lemma *INY-abstr6-correct*: *INY-abstr6* ≤ ⇓*Id INY-abstr5*
unfolding *INY-abstr6-def INY-abstr5-def*
apply (*refine-rcg*)
apply (*rule INY-abstr6-empty-d*δ^r*-correct*)
apply (*rule INY-abstr6-init-d*δ^r*-correct*, *simp*, *simp*)
apply (*rule INY-abstr6-init-N-correct*, *simp*)
apply (*rule INY-abstr6-init-ω𝒞-correct*, *simp*)
apply (*simp*, *rule Id-refine*)
apply *simp*
done

Refinement of ω from (′*q* × ′*q*) *set* to ′*q* ⇀ ′*q set*

abbreviation *ω-α* ≡ *rel-α*
lemmas *ω-α-def* = *rel-α-def*

definition *ω-insert* :: (′*q*×′*q*) ⇒ (′*q*⇀′*q set*) ⇒ ′*q*⇀′*q set***where**
ω-insert ≡ (λ(*x*,*y*) ω. *case* ω *x of*
None ⇒ ω(*x* ↦ {*y*}) | *Some ωx* ⇒ ω(*x* ↦ *insert y ωx*))

definition *ω-member* :: (′*q*×′*q*) ⇒ (′*q*⇀′*q set*) ⇒ *bool* **where**
ω-member = (λ(*x*,*y*) ω. *case* ω *x of*
None ⇒ *False* | *Some ωx* ⇒ *y* ∈ *ωx*)

abbreviation *ω-union-invar* ω *S* ≡ (λ*it* ω′. *ω-α* ω′ ∪ *it* = *ω-α* ω ∪ *S*)

definition *ω-union* :: (′*q*⇀′*q set*) ⇒ (′*q*×′*q*) *set* ⇒ (′*q*⇀′*q set*) *nres* **where**
ω-union ω *S* ≡ *FOREACH*$^{\textit{ω-union-invar ω S}}$ *S*
(λ*xy* ω. *RETURN* (*ω-insert xy* ω)) ω

lemma *ω-insert-correct*[*simp*]: *ω-α* (*ω-insert xy* ω) = *insert xy* (*ω-α* ω)
unfolding *ω-α-def ω-insert-def*
by (*cases xy*, *force split*: *option.split-asm option.split split-if-asm*)

lemma *ω-member-correct*[*simp*]: *ω-member xy* ω = (*xy* ∈ *ω-α* ω)
unfolding *ω-α-def ω-member-def*
by (*cases xy*, *force split*: *option.split*)

lemma *ω-union-correct*:
assumes *finite S*
shows *ω-union* ω *S* ≤ ⇓(*br ω-α* (λ-. *True*)) (*RETURN* (*ω-α* ω ∪ *S*))
unfolding *ω-union-def*
by (*refine-rcg*, *intro refine-vcg assms*, *auto simp*: *br-def*)

definition *INY-abstr7-init-ωC-loopv* **where**
INY-abstr7-init-ωC-loopv d c u ω C ≡
FOREACH (Q A)
(λv (ω′,C′). if d (v, c) = Some 0 then
RETURN (ω-insert (u,v) ω′, insert (u,v) C′)
else RETURN (ω′,C′)) (ω,C)

definition *INY-abstr7-init-ωC-loopu* **where**
INY-abstr7-init-ωC-loopu d c ω C ≡
FOREACH (Q A)
(λu (ω′,C′). if d (u, c) ≠ Some 0
then INY-abstr7-init-ωC-loopv d c u ω′ C′
else RETURN (ω′,C′))
(ω,C)

definition *INY-abstr7-init-ωC-loopc* **where**
INY-abstr7-init-ωC-loopc d ≡
FOREACH (Σ A)
(λc (ω,C). INY-abstr7-init-ωC-loopu d c ω C)
(Map.empty, {})

definition *INY-abstr7-init-ωC* **where**
INY-abstr7-init-ωC d ≡ do {
(ω,C) ← INY-abstr7-init-ωC-loopc d;
let FN = F A × (Q A − F A);
ω′ ← ω-union ω FN;
RETURN (ω′, C ∪ FN)
}

abbreviation *ωC-rel ≡ br (λ(ω,C). (ω-α ω, C)) (λ-. True)*
lemma *ωC-rel-sv: single-valued ωC-rel* **by** *(fact br-sv)*

lemma *finite-FN: finite (F A × (Q A − F A))*
by *(intro finite-Diff finite-SigmaI finite-F finite-Q)*

lemma *nofail-ω-union: finite S ⟹ nofail (ω-union ω S)*
by *(drule ω-union-correct, force simp add: pw-le-iff pw-conc-nofail)*

lemma *inres-ω-union:*
assumes *finite S inres (ω-union ω S) ω′*
shows *ω-α ω′ = ω-α ω ∪ S*
using *ω-union-correct[OF assms(1)] assms(2)*
by *(simp add: pw-ref-sv-iff[OF br-sv] nofail-ω-union[OF assms(1)])*
(simp add: br-def)

lemma *INY-abstr7-init-ωC-refine:*
notes *[[goals-limit = 1]]*
shows *INY-abstr7-init-ωC d ≤ ⇓ωC-rel (INY-abstr6-init-ωC d)*

unfolding *INY-abstr7-init-ωC-def INY-abstr6-init-ωC-def*
INY-abstr7-init-ωC-loopc-def INY-abstr6-init-ωC-loopc-def
INY-abstr7-init-ωC-loopu-def INY-abstr6-init-ωC-loopu-def
INY-abstr7-init-ωC-loopv-def INY-abstr6-init-ωC-loopv-def
by (*refine-rcg inj-on-id ωC-rel-sv*, *simp*, *simp add*: *ω-α-def*,
  simp-all add: *pw-le-iff refine-pw-simps br-def*
    nofail-ω-union[*OF finite-FN*] *inres-ω-union*[*OF finite-FN*])

definition *INY-abstr7-loopu* **where**
INY-abstr7-loopu ω $\mathcal{C}$ δ^r u' v' c v $\equiv$
  FOREACH (*case* $\delta^r(u',c)$ *of None* $\Rightarrow$ {} | *Some s* $\Rightarrow$ *s*) (λu (ω, $\mathcal{C}$).
    if $\neg\omega$*-member* (*u*,*v*) ω *then*
      RETURN (ω*-insert* (*u*,*v*) ω, *insert* (*u*,*v*) $\mathcal{C}$)
    else
      RETURN (ω, $\mathcal{C}$)
) (ω, $\mathcal{C}$)

abbreviation $\omega\mathcal{C}$-α $\equiv$ ($\lambda(\omega,\mathcal{C})$. ($\omega$-$\alpha$ ω,$\mathcal{C}$))
abbreviation $\omega\mathcal{C}N$-α $\equiv$ ($\lambda(\omega,\mathcal{C},N)$. ($\omega$-$\alpha$ ω,$\mathcal{C}$,N))
abbreviation $\omega\mathcal{C}N$-*rel* $\equiv$ *br* $\omega\mathcal{C}N$-α (λ-. *True*)
lemma $\omega\mathcal{C}N$-*rel-sv*: *single-valued* $\omega\mathcal{C}N$-*rel* **by** (*fact br-sv*)

lemma *INY-abstr7-loopu-refine*:
  notes [[*goals-limit* = *1*]]
  shows *INY-abstr7-loopu* ω $\mathcal{C}$ δ^r u' v' c v $\leq\Downarrow\omega\mathcal{C}$-*rel*
    (*INY-abstr5-loopu* (ω-α ω) $\mathcal{C}$ δ^r u' v' c v)
unfolding *INY-abstr7-loopu-def INY-abstr5-loopu-def*
  by (*refine-rcg inj-on-id* $\omega\mathcal{C}N$-*rel-sv*, *simp-all add*: *br-def*)

definition *INY-abstr7-loopv* **where**
INY-abstr7-loopv ω $\mathcal{C}$ N δ^r u' v' c $\equiv$
  FOREACH (*case* $\delta^r(v',c)$ *of None* $\Rightarrow$ {} | *Some s* $\Rightarrow$ *s*) (λv (ω, $\mathcal{C}$, N). *do* {
    let (N, *iszero*) = *INY-dec-counter* N c u' v;
    if iszero then do {
      (ω', $\mathcal{C}'$) $\leftarrow$ *INY-abstr7-loopu* ω $\mathcal{C}$ δ^r u' v' c v;
      RETURN (ω', $\mathcal{C}'$, N)
    } *else*
      RETURN (ω, $\mathcal{C}$, N)
  }) (ω, $\mathcal{C}$, N)

lemma *INY-abstr7-loopv-refine*:
  notes [[*goals-limit* = *1*]]
  shows *INY-abstr7-loopv* ω $\mathcal{C}$ N δ^r u' v' c $\leq\Downarrow\omega\mathcal{C}N$-*rel*
    (*INY-abstr5-loopv* (ω-α ω) $\mathcal{C}$ N δ^r u' v' c)
unfolding *INY-abstr7-loopv-def INY-abstr5-loopv-def*
apply (*refine-rcg inj-on-id* $\omega\mathcal{C}N$-*rel-sv*)
apply (*simp-all add*: *br-def*) [*3*]
apply (*force simp*: *br-def intro*!: *INY-abstr7-loopu-refine*)

apply (*simp-all add*: *br-def*)
done

definition *INY-abstr7-loopc* **where**
INY-abstr7-loopc ω $\mathcal{C}$ N δ^r u' v' $\equiv$ *FOREACH*
  (Σ $\mathcal{A}$) (λc (ω, $\mathcal{C}$, N). *do* {
    (ω',$\mathcal{C}'$,N') $\leftarrow$ *INY-abstr7-loopv* ω $\mathcal{C}$ N δ^r u' v' c;
    RETURN (ω',$\mathcal{C}'$,N')
  }) (ω, $\mathcal{C}$, N)

lemma *INY-abstr7-loopc-refine*:
 notes [[*goals-limit* = *1*]]
 shows *INY-abstr7-loopc* ω $\mathcal{C}$ N δ^r u' v' $\leq\Downarrow\omega\mathcal{C}N$*-rel*
   (*INY-abstr5-loopc* (ω-α ω) $\mathcal{C}$ N δ^r u' v')
unfolding *INY-abstr7-loopc-def INY-abstr5-loopc-def*
apply (*refine-rcg inj-on-id* $\omega\mathcal{C}N$*-rel-sv*)
apply (*simp-all add*: *br-def*) [*2*]
apply (*force simp*: *br-def intro*!: *INY-abstr7-loopv-refine*)
apply (*simp-all add*: *br-def*)
done

definition *INY-abstr7'* **where**
INY-abstr7' ω $\mathcal{C}$ N δ^r $\equiv$ *WHILE*$_T$ (λ(ω, $\mathcal{C}$, N). $\mathcal{C}\neq\{\}$)
 (λ(ω, $\mathcal{C}$, N). *do* {
   ASSERT ($\mathcal{C}\neq\{\}$);
   (u',v') $\leftarrow$ *SPEC* (λ(u',v'). (u',v')$\in\mathcal{C}$);
   let $\mathcal{C}$ = $\mathcal{C}$ − {(u',v')};
   (ω,$\mathcal{C}$,N) $\leftarrow$ *INY-abstr7-loopc* ω $\mathcal{C}$ N δ^r u' v';
   RETURN (ω, $\mathcal{C}$, N)
 }) (ω, $\mathcal{C}$, N)

lemma *INY-abstr7'-refine*:
 notes [[*goals-limit* = *1*]]
 shows *INY-abstr7'* ω $\mathcal{C}$ N δ^r $\leq\Downarrow\omega\mathcal{C}N$*-rel*
   (*INY-abstr5'* (ω-α ω) $\mathcal{C}$ N δ^r)
unfolding *INY-abstr7'-def INY-abstr5'-def*
apply (*refine-rcg inj-on-id* $\omega\mathcal{C}N$*-rel-sv*)
apply (*simp-all add*: *br-def*) [*4*]
apply (*force simp*: *br-def intro*!: *INY-abstr7-loopc-refine*)
apply (*simp-all add*: *br-def*)
done

definition *INY-abstr7* **where**
INY-abstr7 $\equiv$ *do* {
 (d,δ^r) $\leftarrow$ *INY-abstr6-empty-d*δ^r;
 (d,δ^r) $\leftarrow$ *INY-abstr6-init-d*δ^r d δ^r;

N ← *INY-abstr6-init-N d*;
(ω,$\mathcal{C}$) ← *INY-abstr7-init-ω$\mathcal{C}$ d*;
(ω,$\mathcal{C}$,*N*) ← *INY-abstr7′ ω $\mathcal{C}$ N δ^r*;
RETURN (ω,$\mathcal{C}$)
}
lemma *INY-abstr7-correct*:
notes [[*goals-limit* = *1*]]
shows *INY-abstr7* ≤⇓ω$\mathcal{C}$*-rel INY-abstr6*
unfolding *INY-abstr7-def INY-abstr6-def*
by (*refine-rcg inj-on-id* ω$\mathcal{C}$*-rel-sv*)
(*auto simp*: *br-def intro*!: *Id-refine*
INY-abstr7-init-ω$\mathcal{C}$-refine INY-abstr7′-refine)

Summarize the definitions of all the internal constants that occur in *INY-abstr6*

abbreviation *ω-compl-invar ω it ω′* ≡
(∀ *x*. ω′ *x* = (*if x*∈*it* ∨ *x*∉$\mathcal{Q}$ $\mathcal{A}$ *then None else*
Some {*y*. (*x*,*y*) ∈ *compl* (*ω-α ω*)}))

definition *ω-compl* :: (′*q*⇀′*q set*) ⇒ (′*q*⇀′*q set*) *nres* **where**
ω-compl ω = *FOREACH*$^{\omega\text{-compl-invar }\omega}$ ($\mathcal{Q}$ $\mathcal{A}$)
(λ*q ω′*. *case ω q of*
None ⇒ *RETURN* (ω′(*q* ↦ $\mathcal{Q}$ $\mathcal{A}$)) |
Some ωq ⇒ *RETURN* (ω′(*q* ↦ $\mathcal{Q}$ $\mathcal{A}$ − *ωq*))
) *Map.empty*

lemma *ω-compl-correct*:
ω-compl ω ≤⇓(*br ω-α* (λ-. *True*)) (*RETURN* (*compl* (*ω-α ω*)))
unfolding *ω-compl-def*
by (*refine-rcg*, *intro refine-vcg finite-$\mathcal{Q}$*, *unfold ω-α-def br-def*,
auto split: *split-if-asm*)

lemma *nofail-ω-compl*: *nofail* (*ω-compl ω*)
by (*insert ω-compl-correct*, *force simp add*: *pw-le-iff pw-conc-nofail*)

lemma *inres-ω-compl*:
assumes *inres* (*ω-compl ω*) *ω′*
shows *ω-α ω′* = *compl* (*ω-α ω*)
using *ω-compl-correct assms*
by (*simp add*: *pw-ref-sv-iff* [*OF br-sv*] *nofail-ω-compl*)
(*simp add*: *br-def*)

definition *compute-simrel* ≡ *do* {
(ω,-) ← *INY-abstr7*;
ω-compl ω
}

lemma *compute-simrel-correct*:

compute-simrel $\leq\Downarrow$(*br* ω-α (λ-. *True*)) (*SPEC* ($\lambda s.$ s=$\mathcal{S}_{\mathcal{A}}$))
proof –
note *INY-abstr7-correct*
also note *INY-abstr6-correct*
also note *INY-abstr5-correct*
also note *INY-abstr4-correct*
also note *INY-abstr3-correct*
also note *INY-abstr2-correct*
also note *INY-abstr1-correct*
finally have *A7*: *INY-abstr7* $\leq\Downarrow\omega\mathcal{C}$-*rel* (*SPEC* ($\lambda(\omega$, -). *compl* $\omega = \mathcal{S}_{\mathcal{A}}$)) **.**

have *nofail*: *nofail INY-abstr7*
by (*insert A7*, *force simp add*: *pw-le-iff pw-conc-nofail*)
with *A7* **have** *inres*: $\bigwedge\omega$ $\mathcal{C}$. *inres INY-abstr7* (ω,$\mathcal{C}$) $\Longrightarrow$
compl (ω-α ω) = $\mathcal{S}_{\mathcal{A}}$
by (*simp add*: *pw-ref-sv-iff* [*OF br-sv*] *nofail-*ω*-compl*)
(*auto simp add*: *br-def*)

show *?thesis*
unfolding *compute-simrel-def*
by (*auto simp add*: *pw-le-iff refine-pw-simps br-def*
nofail inres nofail-ω*-compl inres-*ω*-compl*)
qed

lemmas *INY-defs* =
INY-abstr7-def
INY-abstr7'-def
INY-abstr7-init-$\omega\mathcal{C}$*-def*
INY-abstr7-init-$\omega\mathcal{C}$*-loopc-def*
INY-abstr7-init-$\omega\mathcal{C}$*-loopu-def*
INY-abstr7-init-$\omega\mathcal{C}$*-loopv-def*
INY-abstr7-loopc-def
INY-abstr7-loopv-def
INY-abstr7-loopu-def
INY-abstr6-def
INY-abstr6-empty-dδ^r*-def*
INY-abstr6-init-dδ^r*-def*
INY-abstr6-init-N-def
INY-abstr6-init-$\omega\mathcal{C}$*-def*
INY-abstr5'-def
INY-abstr6-init-$\omega\mathcal{C}$*-loopc-def*
INY-abstr5-loopc-def
INY-abstr6-init-$\omega\mathcal{C}$*-loopu-def*
INY-abstr5-loopv-def
INY-dec-counter-def
INY-abstr5-loopu-def
INY-abstr6-init-$\omega\mathcal{C}$*-loopv-def*
INY-initial-def
ω*-insert-def*

ω-member-def
ω-union-def
ω-compl-def

NFA reduction

abbreviation *map-option f x* ≡ *case f x of None* ⇒ *x* | *Some y* ⇒ *y*

definition *NFA-reduce* **where**
NFA-reduce ≡ *do* {
  $\mathcal{S}_\mathcal{A}$ ← *SPEC* (λ$\mathcal{S}$. $\mathcal{S}$ = $\mathcal{S}_\mathcal{A}$);
  f ← *SPEC* (*is-preord-eqclasses-map* ($\mathcal{Q}$ $\mathcal{A}$) $\mathcal{S}_\mathcal{A}$);
  RETURN (*NFA-rename-states* $\mathcal{A}$ (*map-option f* :: ′*q* ⇒ ′*q*))
}

lemma *preord-eqclasses-map-is-rename-fun*:
assumes *is-preord-eqclasses-map* ($\mathcal{Q}$ $\mathcal{A}$) $\mathcal{S}_\mathcal{A}$ *f*
shows *NFA-is-equivalence-rename-fun* $\mathcal{A}$ (*map-option f*)
proof−
  let *?eq* = λ*u v*. (*u*, *v*) ∈ $\mathcal{S}_\mathcal{A}$ ∧ (*v*, *u*) ∈ $\mathcal{S}_\mathcal{A}$
  from *assms* **have** *f-props*:
    ⋀*u*. *u*∈$\mathcal{Q}$ $\mathcal{A}$ ⟹ ∃ *v*. *f u* = *Some v*
    ⋀*u v*. *u*∈$\mathcal{Q}$ $\mathcal{A}$ ⟹ *v* ∈ $\mathcal{Q}$ $\mathcal{A}$ ⟹
        (*f u* = *f v*) ⟷ *?eq u v*
    unfolding *is-preord-eqclasses-map-def* **by** *auto*

  {
   fix *u*::′*q* **and** *v*::′*q*
   assume *u*∈$\mathcal{Q}$ $\mathcal{A}$ *v*∈$\mathcal{Q}$ $\mathcal{A}$
   from *f-props*(*1*)[*OF this*(*1*)] *f-props*(*1*)[*OF this*(*2*)]
    obtain *u*′ *v*′ **where** [*simp*]: *f u* = *Some u*′ *f v* = *Some v*′ **by** *auto*
   from *f-props*(*2*)[*OF* ‹*u* ∈ $\mathcal{Q}$ $\mathcal{A}$› ‹*v* ∈ $\mathcal{Q}$ $\mathcal{A}$›]
    have (*map-option f u* = *map-option f v*) ⟷ *?eq u v* **by** *simp*
   also have ... ⟶ *u* $=_R$ *v* **using** *sim-imp-$\mathcal{L}$-right-subset* **by** *blast*
   finally have *map-option f u* = *map-option f v* ⟶ *u* $=_R$ *v* **by** *simp*
  }
  thus *?thesis*
   unfolding *NFA-is-equivalence-rename-fun-def* **by** *blast*
qed

lemma *NFA-reduce-correct*:
  NFA-reduce ≤ *SPEC* (λ$\mathcal{A}$′. $\mathcal{L}$ $\mathcal{A}$′ = $\mathcal{L}$ $\mathcal{A}$)
unfolding *NFA-reduce-def*
by (*intro refine-vcg*, *simp add*: *$\mathcal{L}$-rename-iff*
    preord-eqclasses-map-is-rename-fun)

definition *NFA-reduce-impl* **where**
NFA-reduce-impl ≡ *do* {
  $\mathcal{S}_\mathcal{A}$ ← *compute-simrel*;

$f \leftarrow$ *preord-eqclasses-map-impl* $(\mathcal{Q}\ \mathcal{A})\ \mathcal{S}_{\mathcal{A}}$;
RETURN (*NFA-rename-states* $\mathcal{A}$ (*map-option* f :: $'q \Rightarrow 'q$))
}

lemma *NFA-reduce-impl-refine*:
NFA-reduce-impl $\leq\Downarrow$*Id NFA-reduce*
unfolding *NFA-reduce-impl-def NFA-reduce-def*
apply (*refine-rcg br-sv*[*of* ω-α λ-. *True*])
apply (*rule compute-simrel-correct*)
apply (*rule preord-eqclasses-map-impl-correct*[*OF finite-*$\mathcal{Q}$])
apply (*simp add*: *simrel-preorder*)
apply (*simp add*: ω-α-*def*[*abs-def*] *rel-*α-*def*[*abs-def*])
apply *simp*
done

lemma *NFA-reduce-impl-correct*:
NFA-reduce-impl $\leq$ *SPEC* ($\lambda\mathcal{A}'$. $\mathcal{L}\ \mathcal{A}' = \mathcal{L}\ \mathcal{A}$)
by (*rule order-trans*, *rule NFA-reduce-impl-refine*,
simp add: *NFA-reduce-correct*)

definition *rev-simrel* ($\mathcal{S}_{\mathcal{A}}{}^{-1}$)
where *rev-simrel* $\equiv$ *NFA*.$\mathcal{S}_{\mathcal{A}}$ (*NFA-reverse* $\mathcal{A}$)

definition *NFA-reduce-rev* **where**
NFA-reduce-rev $\equiv$ *do* {
$\mathcal{A}' \leftarrow$ *SPEC* ($\lambda\mathcal{A}'$. $\mathcal{A}'$ = *NFA-reverse* $\mathcal{A}$);
$\mathcal{S}_{\mathcal{A}} \leftarrow$ *SPEC* ($\lambda\mathcal{S}$. $\mathcal{S} = \mathcal{S}_{\mathcal{A}}{}^{-1}$);
$f \leftarrow$ *SPEC* (*is-preord-eqclasses-map* $(\mathcal{Q}\ \mathcal{A})\ \mathcal{S}_{\mathcal{A}}{}^{-1}$);
RETURN (*NFA-rename-states* $\mathcal{A}$ (*the o f* :: $'q \Rightarrow 'q$))
}

definition *NFA-reduce-rev-impl* **where**
NFA-reduce-rev-impl $\equiv$ *do* {
$\mathcal{A}' \leftarrow$ *SPEC* ($\lambda\mathcal{A}'$. $\mathcal{A}'$ = *NFA-reverse* $\mathcal{A}$);
$\mathcal{S} \leftarrow$ *NFA.compute-simrel* $\mathcal{A}'$;
$f \leftarrow$ *preord-eqclasses-map-impl* $(\mathcal{Q}\ \mathcal{A})\ \mathcal{S}$;
RETURN (*NFA-rename-states* $\mathcal{A}$ (*the o f* :: $'q \Rightarrow 'q$))
}

Renaming states and taking the reverse automaton can be performed in arbitrary order without changing the result.

lemma *NFA-rename-reverse-commute*:
NFA-rename-states (*NFA-reverse* $\mathcal{A}$) f = *NFA-reverse* (*NFA-rename-states* $\mathcal{A}$ f)
unfolding *NFA-rename-states-def SemiAutomaton-rename-states-ext-def*
NFA-reverse-def **by** *auto*

States with the same left language can be merged without changing the acceptance behaviour of the automaton. Note: $\mathcal{L}$-*right* of individual states may change. The renaming is therefore not an "equivalence rename function" as defined in the NFA theory.

lemma *NFA-left-equiv-rename*:
  assumes $\forall\, q{\in}\mathcal{Q}\ \mathcal{A}.\ \forall\, q'{\in}\mathcal{Q}\ \mathcal{A}.\ (f\ q = f\ q') \longrightarrow \mathcal{L}\text{-}left\ \mathcal{A}\ q = \mathcal{L}\text{-}left\ \mathcal{A}\ q'$
  shows $\mathcal{L}$ (*NFA-rename-states* $\mathcal{A}$ *f*) = $\mathcal{L}$ $\mathcal{A}$
proof−
  have [*simp*]: $\bigwedge A$. *NFA-reverse* (*NFA-reverse* *A*) = *A*
    unfolding *NFA-reverse-def* **by** *simp*

  let ?$\mathcal{A}'$ = *NFA-reverse* $\mathcal{A}$ **and** ?$\mathcal{A}''$ = *NFA-reverse* (*NFA-rename-states* $\mathcal{A}$ *f*)
  {
   fix $q\ q'$ **assume** $q{\in}\mathcal{Q}\ \mathcal{A}$ $\quad q'{\in}\mathcal{Q}\ \mathcal{A}$ $\quad f\ q = f\ q'$
   with *assms* **have** $\mathcal{L}$-*left* $\mathcal{A}$ *q* = $\mathcal{L}$-*left* $\mathcal{A}$ *q'* **by** *blast*
   hence $\mathcal{L}$-*right* ?$\mathcal{A}'$ *q* = $\mathcal{L}$-*right* ?$\mathcal{A}'$ *q'*
    using *NFA-reverse---*$\mathcal{L}$*-in-state*[*of* $\mathcal{A}$ *q*]
      NFA-reverse---$\mathcal{L}$*-in-state*[*of* $\mathcal{A}$ *q'*] **by** *simp*
  }
  moreover have $\mathcal{Q}$?$\mathcal{A}'$ = $\mathcal{Q}$ $\mathcal{A}$ **by** *simp*
  ultimately have *NFA-is-equivalence-rename-fun* ?$\mathcal{A}'$ *f*
    unfolding *NFA-is-equivalence-rename-fun-def* **by** *blast*
 with *NFA.*$\mathcal{L}$*-rename-iff*[*OF NFA-reverse---is-well-formed*[*OF NFA-axioms*], *symmetric*]
    have $\mathcal{L}$?$\mathcal{A}'$ = $\mathcal{L}$ (*NFA-rename-states* ?$\mathcal{A}'$ *f*) **.**
  also have *NFA-rename-states* ?$\mathcal{A}'$ *f* = ?$\mathcal{A}''$
    using *NFA.NFA-rename-reverse-commute*[*OF NFA-axioms*] **.**
  finally have $\mathcal{L}$?$\mathcal{A}'$ = $\mathcal{L}$?$\mathcal{A}''$ **.**
  thus *?thesis* **by** (*force simp*: *NFA-reverse---*$\mathcal{L}$)
qed

lemma *rev-simrel-eqclasses-map-is-rename-fun*:
assumes *is-preord-eqclasses-map* ($\mathcal{Q}$ $\mathcal{A}$) $\mathcal{S}_{\mathcal{A}}{}^{-1}$ *f*
shows $\mathcal{L}$ (*NFA-rename-states* $\mathcal{A}$ (*the* ∘ *f*)) = $\mathcal{L}$ $\mathcal{A}$
proof (*rule NFA-left-equiv-rename*, *intro ballI impI*)
  note *wf-rev* = *NFA.NFA-reverse---is-well-formed*[*OF NFA-axioms*]
  fix *u v* **assume** *A*: $u \in \mathcal{Q}\ \mathcal{A}$ $\quad v \in \mathcal{Q}\ \mathcal{A}$ $\quad$ (*the* ∘ *f*) *u* = (*the* ∘ *f*) *v*
  from *assms* **have** *f-props*:
    $\bigwedge u.\ u{\in}\mathcal{Q}\ \mathcal{A} \Longrightarrow \exists\, v.\ f\ u = Some\ v$
    $\bigwedge u\ v.\ u{\in}\mathcal{Q}\ \mathcal{A} \Longrightarrow v \in \mathcal{Q}\ \mathcal{A} \Longrightarrow$
      $(f\ u = f\ v) \longrightarrow u =_L v$
    unfolding *is-preord-eqclasses-map-def*
    unfolding *rev-simrel-def NFA.in-*$\mathcal{S}_{\mathcal{A}}$*-iff-simulated*[*OF wf-rev*]
    by (*auto dest*!: *sim-reverse-imp-*$\mathcal{L}$*-left-subset*)
  from *f-props*(*1*)[*OF A*(*1*)] *f-props*(*1*)[*OF A*(*2*)]
    obtain *u' v'* **where** [*simp*]: *f u* = *Some u'* $\quad$ *f v* = *Some v'* **by** *auto*
  from *A*(*3*) **have** *u'* = *v'* **by** *simp*
  with *f-props*(*2*)[*OF A*(*1*,*2*)] **show** $u =_L v$ **by** *simp*
qed

lemma *NFA-reduce-rev-correct*:

NFA-reduce-rev ≤ *SPEC* (λ𝒜′. ℒ 𝒜′ = ℒ 𝒜)
unfolding *NFA-reduce-rev-def*
by (*intro refine-vcg*, *simp add*: *rev-simrel-eqclasses-map-is-rename-fun*)

lemma *NFA-reduce-rev-impl-refine*:
NFA-reduce-rev-impl ≤⇓*Id NFA-reduce-rev*
unfolding *NFA-reduce-rev-impl-def NFA-reduce-rev-def*
apply (*refine-rcg br-sv*[*of* ω-α λ-. *True*])
using *NFA.compute-simrel-correct*[*OF NFA.NFA-reverse---is-well-formed*[*OF NFA-axioms*]]
apply (*simp add*: *rev-simrel-def*)
apply (*rule preord-eqclasses-map-impl-correct*[*OF finite-𝒬*])
using *NFA.simrel-preorder*[*OF NFA.NFA-reverse---is-well-formed*[*OF NFA-axioms*]]
apply (*simp add*: *rev-simrel-def*)
apply (*simp add*: ω-α-*def*[*abs-def*] *rel*-α-*def*[*abs-def*])
apply *simp*
done

lemma *NFA-reduce-rev-impl-correct*:
NFA-reduce-rev-impl ≤ *SPEC* (λ𝒜′. ℒ 𝒜′ = ℒ 𝒜)
apply (*rule order-trans*)
apply (*rule NFA-reduce-rev-impl-refine*)
apply (*simp add*: *NFA-reduce-rev-correct*)
done

lemmas *NFA-reduce-impl-defs* =
preord-eqclasses-map-impl-def
preord-eqclasses-map-impl2-loop-def

end

5.3.5 Refinement and Code Generation

Test

lemmas (**in** *NFA*) *foo-uc* = *INY-abstr5-loopu-def*[*unfolded INY-defs*]
concrete-definition *foo* **uses** *NFA.foo-uc*

schematic-lemma
notes [[*goals-limit* = *1*]]
assumes [*autoref-rules*]: (𝒜*impl*,𝒜)∈⟨*nat-rel*,*nat-rel*⟩*dflt-NFA-rel*
shows (*?f*::*?′c*, *foo* 𝒜)∈*?R*
unfolding *foo-def*[*abs-def*]
apply (*autoref* (*keep-goal*,*trace*))
done

preord-eqclasses-map

lemmas (**in** *NFA*) *pecm-uc* = *preord-eqclasses-map-impl-def*[
unfolded preord-eqclasses-map-impl2-loop-def

]

concrete-definition *pecm-ex* **uses** *NFA.pecm-uc*

schematic-lemma *pecm-impl*:
  notes [[*goals-limit* = *1*]]
  assumes [*autoref-rules*]: (*Qimpl*,*Q*)∈⟨*nat-rel*⟩*dflt-rs-rel*
  assumes [*autoref-rules*]: (*Simpl*,*S*)∈⟨*nat-rel*,⟨*nat-rel*⟩*dflt-rs-rel*⟩*dflt-rm-rel*
  shows (*?f*::*?'c*, *pecm-ex Q S*)∈*?R*
  unfolding *pecm-ex-def*[*abs-def*]
  apply (*autoref-monadic* (*trace*))
  done

concrete-definition *pecm-impl* **uses** *pecm-impl*

declare *pecm-impl.refine*[*autoref-higher-order-rule*, *autoref-rules*]

compute-simrel

We need to extract the definition from the NFA-locale, and unfold the definitions of the internally used constants

lemmas (**in** *NFA*) *compute-simrel-unfold-complete*
  = *compute-simrel-def*[*unfolded INY-defs*]

concrete-definition *compute-simrel-ex* **uses** *NFA.compute-simrel-unfold-complete*

schematic-lemma *compute-simrel-impl*:
  notes [[*goals-limit* = *1*]]
  assumes [*autoref-rules*]: (𝒜*impl*,𝒜)∈⟨*nat-rel*,*nat-rel*⟩*dflt-NFA-rel*
  shows (*?f*,*compute-simrel-ex* 𝒜)∈(*?R*::(*?'c*×-) *set*)
  unfolding *compute-simrel-ex-def*[*abs-def*]
  apply (*autoref-monadic* (*trace*))
  done

concrete-definition *compute-simrel-impl* **uses** *compute-simrel-impl*
thm *compute-simrel-impl.refine*[*no-vars*]
lemma *compute-simrel-impl-refine*[*autoref-rules*]:
  (λ𝒜*impl*. *RETURN* (*compute-simrel-impl* 𝒜*impl*),*compute-simrel-ex*)∈
    ⟨*nat-rel*, *nat-rel*⟩*dflt-NFA-rel*
  → ⟨⟨*nat-rel*,⟨*nat-rel*⟩*dflt-rs-rel*⟩*dflt-rm-rel*⟩*nres-rel*
  by (*parametricity add*: *compute-simrel-impl.refine*)

Reduce

lemma (**in** *NFA*) *is-NFA*: *NFA* 𝒜 **by** *unfold-locales*

context *NFA* **begin**

lemmas *pecm-unfold* = *pecm-ex.refine*[*OF is-NFA*]

lemmas *compute-simrel-unfold* = *compute-simrel-ex.refine*[*OF is-NFA*]

lemmas *reduce-unfold* = *NFA-reduce-impl-def*[
unfolded pecm-unfold, *unfolded compute-simrel-unfold*]

end

concrete-definition *NFA-reduce-ex* **uses** *NFA.reduce-unfold*
print-theorems

declare [[*autoref-trace-intf-unif*]]
declare [[*autoref-trace-failed-id*]]

schematic-lemma *NFA-reduce-impl*:
notes [[*goals-limit* = *1*]]
assumes [*autoref-rules*]: $(\mathcal{A}impl,\mathcal{A})\in\langle nat\text{-}rel,nat\text{-}rel\rangle dflt\text{-}NFA\text{-}rel$
shows $(?f::?'c,NFA\text{-}reduce\text{-}ex\ \mathcal{A})\in ?R$
unfolding *NFA-reduce-ex-def*
apply (*autoref-monadic* (*trace*))
done

concrete-definition *NFA-reduce-impl* **uses** *NFA-reduce-impl*

lemma *NFA-reduce-impl-refine*[*autoref-rules*]:
$(\lambda\mathcal{A}impl.\ RETURN\ (NFA\text{-}reduce\text{-}impl\ \mathcal{A}impl),NFA\text{-}reduce\text{-}ex)\in$
$\langle nat\text{-}rel,\ nat\text{-}rel\rangle dflt\text{-}NFA\text{-}rel$
$\to \langle\langle nat\text{-}rel,\ nat\text{-}rel\rangle dflt\text{-}NFA\text{-}rel\rangle nres\text{-}rel$
by (*parametricity add*: *NFA-reduce-impl.refine*)

export-code
compute-simrel-impl
NFA-reduce-impl
in *SML* **file** −

Correctness lemmas for the constants we generated code from

definition *simrel-rel-def-internal*:
$simrel\text{-}rel\ R \equiv \langle R,\langle R\rangle dflt\text{-}rs\text{-}rel\rangle dflt\text{-}rm\text{-}rel\ O\ br\ rel\text{-}\alpha\ (\lambda\text{-}.\ True)$
lemma *simrel-rel-def*:
$\langle R\rangle simrel\text{-}rel \equiv \langle R,\langle R\rangle dflt\text{-}rs\text{-}rel\rangle dflt\text{-}rm\text{-}rel\ O\ br\ rel\text{-}\alpha\ (\lambda\text{-}.\ True)$
unfolding *relAPP-def simrel-rel-def-internal* **.**

lemma *compute-simrel-correct*:
shows $(compute\text{-}simrel\text{-}impl,\ NFA.\mathcal{S}_{\mathcal{A}})$
$\in \langle nat\text{-}rel,nat\text{-}rel\rangle dflt\text{-}NFA\text{-}rel \to \langle nat\text{-}rel\rangle simrel\text{-}rel$
proof (*intro fun-relI*)
fix $\mathcal{A}impl\ \mathcal{A}$
assume *A*: $(\mathcal{A}impl,\mathcal{A})\in\langle nat\text{-}rel,nat\text{-}rel\rangle dflt\text{-}NFA\text{-}rel$

```
  from A interpret NFA 𝒜 by (auto simp add: NFA-rel-def)

  note compute-simrel-impl.refine[OF A, THEN nres-relD]
  also note compute-simrel-ex.refine[OF is-NFA, symmetric, THEN meta-eq-to-obj-eq]
  also note compute-simrel-correct
  also note conc-fun-chain
  finally have RETURN (compute-simrel-impl 𝒜impl)
    ≤ ⇓ (⟨nat-rel, ⟨nat-rel⟩dflt-rs-rel⟩dflt-rm-rel O br rel-α (λ-. True))
         (SPEC (λs. s = NFA.𝒮𝒜 𝒜))
    by rprems tagged-solver
  thus (compute-simrel-impl 𝒜impl, NFA.𝒮𝒜 𝒜) ∈ ⟨nat-rel⟩simrel-rel
    unfolding simrel-rel-def
    apply (rule RETURN-ref-SPECD)
    by simp
qed

lemma NFA-reduce-impl-correct:
  assumes A: (𝒜impl,𝒜)∈⟨nat-rel,nat-rel⟩dflt-NFA-rel
  shows RETURN (NFA-reduce-impl 𝒜impl)
  ≤ ⇓(⟨nat-rel,nat-rel⟩dflt-NFA-rel) (SPEC (λ𝒜'. ℒ 𝒜' = ℒ 𝒜))
proof –
  from A interpret NFA 𝒜 by (auto simp add: NFA-rel-def)

  note NFA-reduce-impl.refine[OF A, THEN nres-relD]
  also note NFA-reduce-ex.refine[OF is-NFA, symmetric, THEN meta-eq-to-obj-eq]
  also note NFA-reduce-impl-correct
  finally show ?thesis .
qed

hide-const NFA.compl

end

Regular expressions theory Regular-Exp
imports Regular-Set
begin

datatype 'a rexp =
  Zero |
  One |
  Atom 'a |
  Plus ('a rexp)   ('a rexp) |
  Times ('a rexp)   ('a rexp) |
  Star ('a rexp)

primrec lang :: 'a rexp => 'a lang where
lang Zero = {} |
lang One = {[]} |
lang (Atom a) = {[a]} |
```

lang (*Plus r s*) = (*lang r*) *Un* (*lang s*) |
lang (*Times r s*) = *conc* (*lang r*) (*lang s*) |
lang (*Star r*) = *star*(*lang r*)

primrec *atoms* :: *'a rexp* ⇒ *'a set*
where
atoms Zero = {} |
atoms One = {} |
atoms (*Atom a*) = {*a*} |
atoms (*Plus r s*) = *atoms r* ∪ *atoms s* |
atoms (*Times r s*) = *atoms r* ∪ *atoms s* |
atoms (*Star r*) = *atoms r*

fun *rexp-simp* :: *'a rexp* ⇒ *'a rexp*
where
rexp-simp (*Plus a Zero*) = *rexp-simp a* |
rexp-simp (*Plus Zero a*) = *rexp-simp a* |
rexp-simp (*Times a Zero*) = *Zero* |
rexp-simp (*Times Zero a*) = *Zero* |
rexp-simp (*Times a One*) = *rexp-simp a* |
rexp-simp (*Times One a*) = *rexp-simp a* |
rexp-simp (*Star Zero*) = *One* |
rexp-simp (*Star One*) = *One* |
rexp-simp (*Star* (*Star a*)) = *Star* (*rexp-simp a*) |
rexp-simp (*Plus a b*) = *Plus* (*rexp-simp a*) (*rexp-simp b*) |
rexp-simp (*Times a b*) = *Times* (*rexp-simp a*) (*rexp-simp b*) |
rexp-simp (*Star a*) = *Star* (*rexp-simp a*) |
rexp-simp a = *a*

lemma *simplify-correct*: *lang* (*rexp-simp a*) = *lang a*
  by (*induction rule*: *rexp-simp.induct*, *simp-all*)

end

5.4 Conversion of NFAs to Regular Expressions

theory *nfa-to-rexp*
imports
  Main
  NFA
  ../../Refine-Dflt
  ./Regular−Sets/Regular-Set
  ./Regular−Sets/Regular-Exp
begin

We verify an algorithm to convert NFAs to regular expressions. The Algorithm works by iteratively contracting the edges of the NFA to regular expressions. The Refinement Framework is used to refine the algorithm to efficiently executable code.

5.4.1 Basic Definitions

datatype *'q gnfastate = Start | End | State 'q*

instantiation *gnfastate* :: (*hashable*) *hashable*
begin
  definition [*simp*]: *hashcode q* ≡
    (*case q of Start* ⇒ *0* | *End* ⇒ *1* | *State q'* ⇒ *hashcode q'*)
  definition [*simp*]: *bounded-hashcode n q* ≡
    (*case q of Start* ⇒ *0* | *End* ⇒ *1* | *State q'* ⇒ (*2* + *hashcode q'*) *mod n*)
  definition *def-hashmap-size* = (λ- :: (- *gnfastate*) *itself*. *16*)
  instance by(*intro-classes*, *simp-all split*: *gnfastate.split*
    add: *def-hashmap-size-gnfastate-def*)
end

instantiation *gnfastate* :: (*linorder*) *linorder*
begin
  definition *less-eq-def* [*simp*]: *less-eq x y* ≡
   case x of
    Start ⇒ *True* |
    End ⇒ *y* ≠ *Start* |
    State x' ⇒ *case y of*
     State y' ⇒ *x'* ≤ *y'* |
     - ⇒ *False*

  definition *less-def* [*simp*]: *less x y* ≡
   case x of
    Start ⇒ *y* ≠ *Start* |
    End ⇒ *y* ≠ *Start* ∧ *y* ≠ *End* |
    State x' ⇒ *case y of*
     State y' ⇒ *x'* < *y'* |
     - ⇒ *False*

  instance apply (*intro-classes*)
   unfolding *less-eq-def less-def*
   apply (*auto split*: *gnfastate.split gnfastate.split-asm*)
   done
end

record (*'q*,*'a*) *GNFA-rec* =
  𝒬 :: *'q gnfastate set* — The set of states
  δ :: *'q gnfastate* ⇒ *'q gnfastate* ⇒ *'a lang* — The transition function

locale *GNFA* =
  fixes *𝒜*::(*'q*,*'a*) *GNFA-rec*
  assumes *finite-𝒬*: *finite* (*𝒬 𝒜*) **and**

start-correct: $Start \in \mathcal{Q}\ \mathcal{A}$ $\quad \bigwedge q.\ \delta\ \mathcal{A}\ q\ Start = \{\}$ **and**
end-correct: $End \in \mathcal{Q}\ \mathcal{A}$ $\quad \bigwedge q.\ \delta\ \mathcal{A}\ End\ q = \{\}$ **and**
no-epsilon: $\bigwedge u\ v.\ [\![u \neq Start;\ v \neq End]\!] \Longrightarrow [] \notin \delta\ \mathcal{A}\ u\ v$
begin
lemmas *GNFA-wf* = *finite-Q start-correct end-correct no-epsilon*
end

5.4.2 Reachability and paths in GNFAs

inductive *gnfa-is-reachable* **where**
gnfa-eps[*intro*]: $u \in \mathcal{Q}\ \mathcal{A} \Longrightarrow$ *gnfa-is-reachable* $\mathcal{A}\ u\ []\ u$ |
gnfa-step[*intro*]: $[\![x = x1\ @\ x2;$ *gnfa-is-reachable* $\mathcal{A}\ u\ x1\ v;\ w \in \mathcal{Q}\ \mathcal{A};$
$x2 \in \delta\ \mathcal{A}\ v\ w]\!] \Longrightarrow$ *gnfa-is-reachable* $\mathcal{A}\ u\ x\ w$

inductive-cases *gnfa-is-reachableE*[*elim*]: *gnfa-is-reachable* $\mathcal{A}\ u\ x\ v$

lemma *gnfa-is-reachable-trans*[*trans*, *dest*]:
$[\![$*gnfa-is-reachable* $\mathcal{A}\ u\ x1\ v$; *gnfa-is-reachable* $\mathcal{A}\ v\ x2\ w]\!]$
$\Longrightarrow$ *gnfa-is-reachable* $\mathcal{A}\ u\ (x1@x2)\ w$
proof (*rotate-tac*, *induction rule*: *gnfa-is-reachable.induct*)
case (*gnfa-step x2 x21 x22 v1 v2 w*)
have $x1\ @\ x2 = (x1\ @\ x21)\ @\ x22$ **using** ‹$x2 = x21\ @\ x22$› **by** *simp*
thus *?case* **using** *gnfa-step* **by** *blast*
qed *simp*

lemma *gnfa-is-reachable-step*[*intro*, *dest*]:
$[\![u \in \mathcal{Q}\ \mathcal{A};\ v \in \mathcal{Q}\ \mathcal{A};\ x \in \delta\ \mathcal{A}\ u\ v]\!] \Longrightarrow$ *gnfa-is-reachable* $\mathcal{A}\ u\ x\ v$ **by** *blast*

lemma *gnfa-is-reachable-imp-in-Q*[*dest*, *simp*]:
assumes *gnfa-is-reachable* $\mathcal{A}\ u\ x\ v$
shows $u \in \mathcal{Q}\ \mathcal{A}$ **and** $v \in \mathcal{Q}\ \mathcal{A}$ **using** *assms*
by (*induction u x v rule*: *gnfa-is-reachable.induct*, *simp-all*)

lemma *gnfa-is-reachable-rev*:
assumes $x = x1\ @\ x2$ $\quad x1 \in \delta\ \mathcal{A}\ u\ v$ $\quad u \in \mathcal{Q}\ \mathcal{A}$ $\quad$ *gnfa-is-reachable* $\mathcal{A}\ v\ x2\ w$
shows *gnfa-is-reachable* $\mathcal{A}\ u\ x\ w$
proof−
from *assms* **have** *gnfa-is-reachable* $\mathcal{A}\ u\ x1\ v$ **by** *blast*
also note ‹*gnfa-is-reachable* $\mathcal{A}\ v\ x2\ w$›
finally show *?thesis* **using** ‹$x = x1\ @\ x2$› **by** *simp*
qed

lemma *gnfa-is-reachable-revE*:
assumes *gnfa-is-reachable* $\mathcal{A}\ u\ x\ w$
obtains $x = []$ $\quad w{=}u$ $\quad u \in \mathcal{Q}\ \mathcal{A}$ |
x1 x2 v **where** $x = x1\ @\ x2$ $\quad x1 \in \delta\ \mathcal{A}\ u\ v$ $\quad v \in \mathcal{Q}\ \mathcal{A}$
gnfa-is-reachable $\mathcal{A}\ v\ x2\ w$
proof−

```
  case goal1
  from assms have (x=[] ∧ w=u ∧ u ∈ Q 𝒜) ∨ (∃x1 x2 v. x = x1 @ x2 ∧
      gnfa-is-reachable 𝒜 v x2 w ∧ v ∈ Q 𝒜 ∧ x1 ∈ δ 𝒜 u v) (is ?P ∨ ?Q)
  proof (induction 𝒜 u x w rule: gnfa-is-reachable.induct)
    case (gnfa-step x x1 x2 𝒜 u v w)
      from gnfa-step.IH show ?case
      proof (rule disjE)
        assume x1 = [] ∧ v = u ∧ u ∈ GNFA-rec.Q 𝒜
        hence x = x2 @ [] ∧ gnfa-is-reachable 𝒜 w [] w ∧ w ∈ Q 𝒜 ∧
            x2 ∈ δ 𝒜 u w using gnfa-step by auto
        thus ?thesis by blast
      next
        assume ∃x11 x12 v'. x1 = x11 @ x12 ∧ gnfa-is-reachable 𝒜 v' x12 v ∧
            v' ∈ Q 𝒜 ∧ x11 ∈ δ 𝒜 u v'
        then guess x11 x12 v' by (elim exE conjE)
        moreover from this have gnfa-is-reachable 𝒜 v' (x12 @ x2) w
            using gnfa-step and gnfa-is-reachable-trans by blast
        ultimately have x = x11 @ (x12 @ x2) ∧
            gnfa-is-reachable 𝒜 v' (x12 @ x2) w ∧ v' ∈ Q 𝒜 ∧ x11 ∈ δ 𝒜 u v'
            using gnfa-step by force
        thus ?thesis by blast
      qed
  qed blast
  thus ?case using goal1 by auto
qed

lemma gnfa-is-reachable-revE2:
  assumes gnfa-is-reachable 𝒜 u x w and u ≠ w
  obtains x1 x2 v where x = x1 @ x2    x1 ∈ δ 𝒜 u v    v ∈ Q 𝒜
                   gnfa-is-reachable 𝒜 v x2 w
  by (rule gnfa-is-reachable-revE[OF assms(1)], insert assms(2), simp-all)

inductive gnfa-is-reachable-in where
gnfa-in-eps[intro]: u ∈ Q 𝒜 ⟹ gnfa-is-reachable-in 𝒜 u [] u 0 |
gnfa-in-step[intro]: ⟦x = x1 @ x2; gnfa-is-reachable-in 𝒜 u x1 v n;
    w ∈ Q 𝒜; x2 ∈ δ 𝒜 v w⟧ ⟹ gnfa-is-reachable-in 𝒜 u x w (Suc n)

inductive-cases gnfa-is-reachable-inE[elim]:
    gnfa-is-reachable-in 𝒜 u x v n
inductive-cases gnfa-is-reachable-in-0E[elim!]:
    gnfa-is-reachable-in 𝒜 u x v 0
inductive-cases gnfa-is-reachable-in-SucE[elim]:
    gnfa-is-reachable-in 𝒜 u x v (Suc n)

lemma gnfa-is-reachable-in-imp-in-Q[dest]:
  assumes gnfa-is-reachable-in 𝒜 u x v n
  shows u ∈ Q 𝒜 and v ∈ Q 𝒜 using assms
  by (induction rule: gnfa-is-reachable-in.induct, simp-all)
```

lemma *gnfa-is-reachable-in-step*:
  assumes $u \in \mathcal{Q}\ \mathcal{A}$ **and** $v \in \mathcal{Q}\ \mathcal{A}$ **and** $x \in \delta\ \mathcal{A}\ u\ v$
  shows *gnfa-is-reachable-in* $\mathcal{A}$ *u x v* (*Suc 0*)
using *assms* **by** *blast*

lemma *gnfa-is-reachable-iff-is-reachable-in*:
  gnfa-is-reachable $\mathcal{A}\ u\ x\ v \longleftrightarrow (\exists\, n.$ *gnfa-is-reachable-in* $\mathcal{A}\ u\ x\ v\ n)$
apply (*rule iffI*)
apply (*induction u x v rule*: *gnfa-is-reachable.induct*, *blast*, *blast*)
apply (*elim exE*, *induct-tac rule*: *gnfa-is-reachable-in.induct*, *blast*+)
done

5.4.3 Operations on GNFAs

definition *gnfa-add-transitions* **where**
gnfa-add-transitions $\mathcal{A}\ \delta' \equiv (\!|\ \mathcal{Q} = \mathcal{Q}\ \mathcal{A},\ \delta = (\lambda u\ v.\ \delta\ \mathcal{A}\ u\ v \cup \delta'\ u\ v)\ |\!)$

lemma (**in** *GNFA*) *gnfa-add-transitions-wf*[*simp*,*intro*]:
  assumes $\bigwedge q.\ \delta'\ q$ *Start* $= \{\}$ **and** $\bigwedge q.\ \delta'$ *End* $q = \{\}$ **and**
    $\bigwedge u\ v.\ [\![u \neq$ *Start*; $v \neq$ *End*$]\!] \Longrightarrow [] \notin \delta'\ u\ v$
  shows *GNFA* (*gnfa-add-transitions* $\mathcal{A}\ \delta'$)
  using *assms* **and** *GNFA-wf* **unfolding** *gnfa-add-transitions-def GNFA-def*
  by *simp-all*

lemma *gnfa-add-transitions-*$\mathcal{Q}$[*simp*]: $\mathcal{Q}$ (*gnfa-add-transitions* $\mathcal{A}\ \delta'$) $= \mathcal{Q}\ \mathcal{A}$
  unfolding *gnfa-add-transitions-def* **by** *simp*

lemma *gnfa-add-transitions-*δ[*simp*]:
  δ (*gnfa-add-transitions* $\mathcal{A}\ \delta'$) $= (\lambda u\ v.\ \delta\ \mathcal{A}\ u\ v \cup \delta'\ u\ v)$
  unfolding *gnfa-add-transitions-def* **by** *simp*

lemma *gnfa-add-transitions-ge*:
  shows *gnfa-is-reachable* $\mathcal{A}\ u\ x\ v \Longrightarrow$
    gnfa-is-reachable (*gnfa-add-transitions* $\mathcal{A}\ \delta'$) *u x v*
proof (*induction u x v rule*: *gnfa-is-reachable.induct*)
  case (*gnfa-step x x1 x2* $\mathcal{A}$ *u v w*)
    let $?\mathcal{A}'$ = *gnfa-add-transitions* $\mathcal{A}\ \delta'$
    from *gnfa-step.hyps* **have** $w \in \mathcal{Q}\ ?\mathcal{A}'$ **and** $x2 \in \delta\ ?\mathcal{A}'\ v\ w$ **by** *simp-all*
    with ‹$x = x1$ @ $x2$› **and** *gnfa-step.IH* **show** *?case*
      using *gnfa-is-reachable.intros*(2) **by** *blast*
qed *fastforce*

Computes, for all u, w, all the indirect transitions from u to w that can be made by going through the intermediate state v.

definition *gnfa-subsumed-transitions* **where**
gnfa-subsumed-transitions $\mathcal{A}\ v \equiv \lambda u\ w.$ *if* $u \in \mathcal{Q}\ \mathcal{A} - \{v\} \wedge w \in \mathcal{Q}\ \mathcal{A} - \{v\}$ *then*
  (*let* $\mathcal{L}_1 = \delta\ \mathcal{A}\ u\ v$; $\mathcal{L}_2 = \delta\ \mathcal{A}\ v\ v$; $\mathcal{L}_3 = \delta\ \mathcal{A}\ v\ w$
  in $\mathcal{L}_1$ @@ *star* $\mathcal{L}_2$ @@ $\mathcal{L}_3$) *else* $\{\}$

```
lemma gnfa-is-reachable-loop:
  ⟦x ∈ star (δ 𝒜 q q); q ∈ 𝒬 𝒜⟧ ⟹ gnfa-is-reachable 𝒜 q x q
proof (induction rule: star-induct)
  case Nil thus ?case by blast
next
  case (append u v)
    hence gnfa-is-reachable 𝒜 q v q using append by simp
    from gnfa-is-reachable-rev[OF - append.hyps(1) append.prems this]
        show ?case by simp
qed

lemma gnfa-subsumed-transitions-is-reachable:
  assumes x ∈ gnfa-subsumed-transitions 𝒜 v u w and
      u ∈ 𝒬 𝒜 − {v} and v ∈ 𝒬 𝒜 and w ∈ 𝒬 𝒜 − {v}
  shows gnfa-is-reachable 𝒜 u x w
proof−
  from assms have u ≠ v and w ≠ v and u ∈ 𝒬 𝒜 and w ∈ 𝒬 𝒜 by simp-all
  with assms(1) obtain x1 x2 x3 where x-split[simp]:
      x = x1 @ x2 @ x3    x1 ∈ δ 𝒜 u v
      x2 ∈ star (δ 𝒜 v v)    x3 ∈ δ 𝒜 v w
      by (force simp: gnfa-subsumed-transitions-def elim!: concE)
  have gnfa-is-reachable 𝒜 u x1 v
      using gnfa-is-reachable-step[OF ‹u ∈ 𝒬 𝒜› ‹v ∈ 𝒬 𝒜›] by simp
  also have gnfa-is-reachable 𝒜 v x2 v
      using gnfa-is-reachable-loop[OF - ‹v ∈ 𝒬 𝒜›] by simp
  also have gnfa-is-reachable 𝒜 v x3 w
      using gnfa-is-reachable-step[OF ‹v ∈ 𝒬 𝒜› ‹w ∈ 𝒬 𝒜›] by simp
  finally show ?thesis by simp
qed

abbreviation gnfa-add-subsumed-transitions 𝒜 q ≡
    gnfa-add-transitions 𝒜 (gnfa-subsumed-transitions 𝒜 q)

lemma (in GNFA) gnfa-add-subsumed-transitions-wf[simp,intro]:
  assumes q ≠ Start and q ≠ End
  shows GNFA (gnfa-add-subsumed-transitions 𝒜 q)
  by (rule, unfold gnfa-subsumed-transitions-def, auto simp: GNFA-wf assms)

lemma gnfa-add-subsumed-transitions-le:
  assumes q ∈ 𝒬 𝒜 and
         gnfa-is-reachable
             (gnfa-add-subsumed-transitions 𝒜 q) u x w
         (is gnfa-is-reachable ?𝒜′ - - -)
  shows gnfa-is-reachable 𝒜 u x w
using assms(2)
proof (induction ?𝒜′ u x w rule: gnfa-is-reachable.induct, simp-all)
  fix u assume u ∈ 𝒬 𝒜 thus gnfa-is-reachable 𝒜 u [] u by blast
next
  fix x x1 x2 u v w
```

```
  assume x = x1 @ x2 and w ∈ Q A and
     u-to-v: gnfa-is-reachable A u x1 v and
     v-to-w-trans: x2 ∈ δ A v w ∨ x2 ∈ gnfa-subsumed-transitions A q v w
  thus gnfa-is-reachable A u (x1 @ x2) w
  proof (cases rule: disjE[OF v-to-w-trans])


    assume x2 ∈ δ A v w
    from gnfa-is-reachable.intros(2)[OF ‹x = x1 @ x2› u-to-v ‹w ∈ Q A› this]
       show gnfa-is-reachable A u (x1 @ x2) w using ‹x = x1 @ x2› by simp
  next


    assume in-subsumed: x2 ∈ gnfa-subsumed-transitions A q v w
    moreover from this have v ≠ q and w ≠ q
       unfolding gnfa-subsumed-transitions-def by auto
    moreover from u-to-v have v ∈ Q A by simp-all
    ultimately have gnfa-is-reachable A v x2 w using ‹q ∈ Q A› ‹w ∈ Q A›
       gnfa-subsumed-transitions-is-reachable by fast
    thus gnfa-is-reachable A u (x1 @ x2) w using u-to-v by blast
  qed
qed

lemma gnfa-add-subsumed-transitions-equiv[simp]:
  assumes q ∈ Q A
  shows gnfa-is-reachable (gnfa-add-transitions A
     (gnfa-subsumed-transitions A q)) u x w = gnfa-is-reachable A u x w
  by (rule iffI, fact gnfa-add-subsumed-transitions-le[OF assms],
    fact gnfa-add-transitions-ge)


definition gnfa-remove-state where
gnfa-remove-state A q ≡ (| Q = Q A − {q}, δ = δ A |)

lemma gnfa-remove-state-Q[simp]: Q (gnfa-remove-state A q) = Q A − {q}
   unfolding gnfa-remove-state-def by simp
lemma gnfa-remove-state-δ[simp]: δ (gnfa-remove-state A q) = δ A
   unfolding gnfa-remove-state-def by simp

lemma (in GNFA) gnfa-remove-state-wf[simp,intro]:
  assumes q ≠ Start and q ≠ End
  shows GNFA (gnfa-remove-state A q)
  by (unfold-locales, insert assms, simp-all add: GNFA-wf)


lemma gnfa-remove-state-le:
  assumes gnfa-is-reachable (gnfa-remove-state A q) u x v
       (is gnfa-is-reachable ?A' - - -)
  shows gnfa-is-reachable A u x v
```

```
by (insert assms, induction ?𝒜' u x v rule: gnfa-is-reachable.induct, auto)
```

For a sequence of steps $u \to \ldots \to w$ in the automaton with $u \neq w$, this partitions the sequence into $u \to \ldots \to v \to w \to w \to \ldots \to w \to w$ where $v \neq w$.

```
lemma gnfa-is-reachable-in-last-decompose:
  assumes gnfa-is-reachable-in 𝒜 u x w n and u ≠ w
  obtains v x1 x2 x3 n' where n' < n and x = x1 @ x2 @ x3 and v ≠ w and
      gnfa-is-reachable-in 𝒜 u x1 v n' and x2 ∈ δ 𝒜 v w and
      x3 ∈ star (δ 𝒜 w w)
proof−
  case goal1
  have ∃ v x1 x2 x3 n'. n' < n ∧ x = x1 @ x2 @ x3 ∧ v ≠ w ∧
      gnfa-is-reachable-in 𝒜 u x1 v n' ∧ x2 ∈ δ 𝒜 v w ∧ x3 ∈ star (δ 𝒜 w w)
  proof (insert assms, induction rule: gnfa-is-reachable-in.induct)
    case (gnfa-in-step x x1 x2 𝒜 u v n w) thus ?case
    proof (cases v = w)
      case False
        moreover have x = x1 @ x2 @ [] using gnfa-in-step by simp
        ultimately show ?thesis using gnfa-in-step by blast

    next
      case True
        hence u ≠ v using gnfa-in-step by simp
        from gnfa-in-step.IH[OF this] and ‹v = w›
            obtain v' x11 x12 x13 n' where
                    decomposition: n' < n    x1 = x11 @ x12 @ x13    v' ≠ w
                  gnfa-is-reachable-in 𝒜 u x11 v' n'    x12 ∈ δ 𝒜 v' w
                  x13 ∈ star (δ 𝒜 w w) by blast
        with ‹x2 ∈ GNFA-rec.δ 𝒜 v w› and ‹v = w›
            have x13 @ x2 ∈ star (δ 𝒜 w w) by simp
        moreover have x = x11 @ x12 @ (x13 @ x2) and n' < Suc n
            using gnfa-in-step and decomposition by simp-all
        ultimately show ?thesis using decomposition by blast
    qed
  qed simp
  with goal1 show ?thesis by blast
qed


lemma gnfa-remove-redundant-state-ge-helper:
  assumes gnfa-is-reachable-in 𝒜 u x w n and u ≠ q and w ≠ q and
        ⋀u w. ⟦u ∈ 𝒬 𝒜 − {q}; w ∈ 𝒬 𝒜 − {q}⟧ ⟹
            gnfa-subsumed-transitions 𝒜 q u w ⊆
                δ (gnfa-remove-state 𝒜 q) u w
  shows ∃ n. gnfa-is-reachable-in (gnfa-remove-state 𝒜 q) u x w n
      (is ∃ n. gnfa-is-reachable-in ?𝒜' - - - -)
proof (insert assms, induction n arbitrary: x w rule: less-induct)
  case (less n)
    show ?case
```

```
  proof (cases rule: gnfa-is-reachable-inE[OF less.prems(1)])
    case 1
      with less.prems have gnfa-is-reachable-in ?𝒜′ u x w 0 by fastforce
      thus ?thesis ..
  next
    case (2 x1 x2 v n′)
      show ?thesis
      proof (cases v = q)
        case False
          from ‹v ≠ q› and less and 2 obtain n″
              where gnfa-is-reachable-in ?𝒜′ u x1 v n″ by blast
          moreover from 2 and less.prems
              have x2 ∈ δ ?𝒜′ v w and w ∈ 𝒬 ?𝒜′ by simp-all
          ultimately show ?thesis using 2 by blast
      next

        case True
          hence u ≠ v using less.prems by simp
          from gnfa-is-reachable-in-last-decompose[OF
              ‹gnfa-is-reachable-in 𝒜 u x1 v n′› this]
          guess v′ x11 x12 x13 n″ .
          note decomposition = this
          with ‹x2 ∈ δ 𝒜 v w› and ‹v = q› and ‹w ∈ 𝒬 𝒜› and ‹w ≠ q›
              have subsumed: x12 @ x13 @ x2 ∈
                    gnfa-subsumed-transitions 𝒜 q v′ w
              unfolding gnfa-subsumed-transitions-def by (force simp: Let-def)
          have v′ ∈ 𝒬 𝒜 − {q}    w ∈ 𝒬 𝒜 − {q}    u ≠ q    v′ ≠ q
              using decomposition(3,4) ‹v = q› less.prems by blast+
          from assms(4)[OF this(1,2)] and subsumed
              have x12 @ x13 @ x2 ∈ δ ?𝒜′ v′ w by blast

          moreover from ‹n″ < n′› and ‹n = Suc n′› have n″ < n by simp
          from less.IH[OF this decomposition(4) ‹u ≠ q› ‹v′ ≠ q› assms(4)]
            obtain n‴ where gnfa-is-reachable-in ?𝒜′ u x11 v′ n‴
                by blast

          ultimately have gnfa-is-reachable-in ?𝒜′ u x w (Suc n‴)
              using ‹w ∈ 𝒬 𝒜› ‹w ≠ q› decomposition(2)
              ‹x = x1 @ x2› by (force intro!: gnfa-in-step)
          thus ?thesis ..
      qed
  qed
qed

lemma gnfa-remove-redundant-state-ge:
  assumes gnfa-is-reachable 𝒜 u x w and u ≠ q and w ≠ q and
        ⋀u w. ⟦u ∈ 𝒬 𝒜 − {q}; w ∈ 𝒬 𝒜 − {q}⟧ ⟹
            gnfa-subsumed-transitions 𝒜 q u w ⊆
                δ (gnfa-remove-state 𝒜 q) u w
```

shows *gnfa-is-reachable* (*gnfa-remove-state* $\mathcal{A}$ *q*) *u x w*
(**is** *gnfa-is-reachable ?*$\mathcal{A}'$ - - -)
proof−
from *assms*(*1*) **and** *gnfa-is-reachable-iff-is-reachable-in*[*of* $\mathcal{A}$ *u x w*]
obtain *n* **where** *gnfa-is-reachable-in* $\mathcal{A}$ *u x w n* **by** *blast*
from *gnfa-remove-redundant-state-ge-helper*[*OF this assms*(*2*−)]
have $\exists$ *n*. *gnfa-is-reachable-in ?*$\mathcal{A}'$ *u x w n* **by** *blast*
thus *?thesis* **using** *gnfa-is-reachable-iff-is-reachable-in*[*of ?*$\mathcal{A}'$ *u x w*]
by *simp*
qed

lemma *gnfa-remove-redundant-state-equiv*:
assumes $u \neq q$ **and** $w \neq q$ **and** $\bigwedge u\ w.\ [\![u \in \mathcal{Q}\ \mathcal{A} - \{q\};\ w \in \mathcal{Q}\ \mathcal{A} - \{q\}]\!] \Longrightarrow$
gnfa-subsumed-transitions $\mathcal{A}$ *q u w* $\subseteq$
δ (*gnfa-remove-state* $\mathcal{A}$ *q*) *u w*
shows *gnfa-is-reachable* (*gnfa-remove-state* $\mathcal{A}$ *q*) *u x w* =
gnfa-is-reachable $\mathcal{A}$ *u x w*
by (*rule iffI*, *fact gnfa-remove-state-le*,
insert gnfa-remove-redundant-state-ge[*OF - assms*], *blast*)

Contraction of the automaton by "short-circuiting" ingoing transitions, loops and outgoing transitions of the given state q and then removing it.

abbreviation *gnfa-contract* $\mathcal{A}$ *q* $\equiv$ *gnfa-remove-state* (
gnfa-add-subsumed-transitions $\mathcal{A}$ *q*) *q*

lemma *gnfa-contract-def*: *gnfa-contract* $\mathcal{A}$ *q* =
$(\!|\mathcal{Q} = \mathcal{Q}\ \mathcal{A} - \{q\},\ \delta = \lambda u\ v.\ \delta\ \mathcal{A}\ u\ v \cup$ *gnfa-subsumed-transitions* $\mathcal{A}$ *q u v*$|\!)$
unfolding *gnfa-remove-state-def gnfa-add-transitions-def* **by** *simp*

lemma (**in** *GNFA*) *gnfa-contract-wf*[*simp*,*intro*]:
assumes $q \neq$ *Start* **and** $q \neq$ *End*
shows *GNFA* (*gnfa-contract* $\mathcal{A}$ *q*)
using *assms* **by** (*intro GNFA.gnfa-remove-state-wf*, *blast*)

lemma *gnfa-subsumed-transitions-remove-state*[*simp*]:
gnfa-subsumed-transitions (*gnfa-remove-state* $\mathcal{A}$ *q*) *q* =
gnfa-subsumed-transitions $\mathcal{A}$ *q*
by (*intro ext*, *simp add*: *gnfa-subsumed-transitions-def gnfa-remove-state-def*)

lemma *gnfa-contract-correct*:
assumes $q \in \mathcal{Q}\ \mathcal{A}$ **and** $u \in \mathcal{Q}\ \mathcal{A} - \{q\}$ **and** $v \in \mathcal{Q}\ \mathcal{A} - \{q\}$
shows *gnfa-is-reachable* (*gnfa-contract* $\mathcal{A}$ *q*) *u x v* $\longleftrightarrow$
gnfa-is-reachable $\mathcal{A}$ *u x v*
proof−
let *?P* = $\lambda\mathcal{A}$. *gnfa-is-reachable* $\mathcal{A}$ *u x v*
let *?*$\mathcal{A}'$ = *gnfa-add-subsumed-transitions* $\mathcal{A}$ *q*
let *?*$\mathcal{A}''$ = *gnfa-contract* $\mathcal{A}$ *q*

from *gnfa-add-subsumed-transitions-equiv*[*OF assms*(*1*)]

    have $?P\ \mathcal{A} \longleftrightarrow ?P\ ?\mathcal{A}'$ **..**
  also have $\bigwedge u\ v.\ [\![u \in \mathcal{Q}\ ?\mathcal{A}' - \{q\};\ v \in \mathcal{Q}\ ?\mathcal{A}' - \{q\}]\!]$
    $\Longrightarrow$ *gnfa-subsumed-transitions* $?\mathcal{A}'\ q\ u\ v \subseteq \delta\ ?\mathcal{A}''\ u\ v$
    unfolding *gnfa-add-transitions-def gnfa-subsumed-transitions-def* **by** *simp*
  from *assms* **and** *gnfa-remove-redundant-state-equiv*[*OF - - this*]
    have $?P\ ?\mathcal{A}' \longleftrightarrow ?P\ ?\mathcal{A}''$ **by** *simp*
  finally show *?thesis* **..**
qed

The language that is accepted by the automaton.

definition *gnfa-*$\mathcal{L}$ **where** *gnfa-*$\mathcal{L}$ $\mathcal{A} = \{x.$ *gnfa-is-reachable* $\mathcal{A}$ *Start x End*$\}$

Converts an NFA into an equivalent GNFA.

definition *nfa-to-gnfa* **where**
nfa-to-gnfa $A = (\!|\ \mathcal{Q} = \{Start,\ End\} \cup State$ ‘ *SemiAutomaton*.$\mathcal{Q}\ A$,
  $\delta = \lambda u\ v.$ *case u of*
      Start $\Rightarrow$ (*case v of*
        State v $\Rightarrow$ *if* $v \in \mathcal{I}\ A$ *then* $\{[]\}$ *else* $\{\}$ |
        - $\Rightarrow$ $\{\}$) |
      End $\Rightarrow$ $\{\}$ |
      State u $\Rightarrow$ (*case v of*
        Start $\Rightarrow$ $\{\}$ |
        End $\Rightarrow$ *if* $u \in \mathcal{F}\ A$ *then* $\{[]\}$ *else* $\{\}$ |
        State v $\Rightarrow$ $\{[c]\ |c.\ (u,c,v) \in \Delta\ A\}$
) $|\!)$

lemma *nfa-to-gnfa-*$\mathcal{Q}$[*simp*]:
  $\mathcal{Q}$ (*nfa-to-gnfa* $\mathcal{A}$) $= \{Start,\ End\} \cup State$ ‘ *SemiAutomaton*.$\mathcal{Q}\ \mathcal{A}$
  unfolding *nfa-to-gnfa-def* **by** *simp*

lemma *nfa-to-gnfa-*δ[*simp*]:
  δ (*nfa-to-gnfa* $\mathcal{A}$) *u Start* $= \{\}$
  δ (*nfa-to-gnfa* $\mathcal{A}$) *Start End* $= \{\}$
  $v' \in \mathcal{I}\ \mathcal{A} \Longrightarrow \delta$ (*nfa-to-gnfa* $\mathcal{A}$) *Start* (*State* v') $= \{[]\}$
  $v' \notin \mathcal{I}\ \mathcal{A} \Longrightarrow \delta$ (*nfa-to-gnfa* $\mathcal{A}$) *Start* (*State* v') $= \{\}$
  δ (*nfa-to-gnfa* $\mathcal{A}$) *End v* $= \{\}$
  $u' \in \mathcal{F}\ \mathcal{A} \Longrightarrow \delta$ (*nfa-to-gnfa* $\mathcal{A}$) (*State* u') *End* $= \{[]\}$
  $u' \notin \mathcal{F}\ \mathcal{A} \Longrightarrow \delta$ (*nfa-to-gnfa* $\mathcal{A}$) (*State* u') *End* $= \{\}$
  δ (*nfa-to-gnfa* $\mathcal{A}$) (*State* u') (*State* v') $= \{[c]\ |c.\ (u',c,v') \in \Delta\ \mathcal{A}\}$
unfolding *nfa-to-gnfa-def* **by** (*cases u*, *simp-all*)

lemma (**in** *NFA*) *nfa-to-gnfa-wf*[*simp*,*intro*]:
  GNFA (*nfa-to-gnfa* $\mathcal{A}$) (**is** *GNFA* $?\mathcal{A}'$)
proof (*unfold-locales*)
  fix u::'q *gnfastate* **and** v::'q *gnfastate*
  assume $u \neq Start$ **and** $v \neq End$
  thus $[] \notin \delta\ ?\mathcal{A}'\ u\ v$ **by** (*cases u*, *simp*, *simp*, *cases v*, *simp-all*)
qed (*simp-all add*: *finite-*$\mathcal{Q}$)

If the automaton consists only of Start and End, its entire language is in

GNFA-rec.δ 𝒜 Start end.

```
lemma (in GNFA) gnfa-ℒ-Start-End:
  assumes 𝒬 𝒜 = {Start, End}
  shows gnfa-ℒ 𝒜 = δ 𝒜 Start End
unfolding gnfa-ℒ-def
proof (intro equalityI subsetI, simp-all)
  fix x assume gnfa-is-reachable 𝒜 Start x End
  thus x ∈ δ 𝒜 Start End
  proof (rule gnfa-is-reachableE)
    fix x1 x2 v
    assume x = x1 @ x2 and gnfa-is-reachable 𝒜 Start x1 v and
        x2 ∈ δ 𝒜 v End
    moreover from this have v = Start using assms and end-correct by fast
    ultimately have x = x2 using start-correct by blast
    thus x ∈ δ 𝒜 Start End using ‹v = Start› and ‹x2 ∈ δ 𝒜 v End› by simp
  qed simp
next
  fix x assume x ∈ δ 𝒜 Start End
  thus gnfa-is-reachable 𝒜 Start x End
      using start-correct and end-correct by blast
qed
```

```
lemma (in NFA) LTS-is-reachable-iff-gnfa-is-reachable:
  assumes u ∈ SemiAutomaton.𝒬 𝒜 and v ∈ SemiAutomaton.𝒬 𝒜
  shows LTS-is-reachable (Δ 𝒜) u x v ⟷
      gnfa-is-reachable (nfa-to-gnfa 𝒜) (State u) x (State v)
   (is - ⟷ gnfa-is-reachable ?𝒜′ - - -)
proof (rule iffI)
  assume LTS-is-reachable (Δ 𝒜) u x v
  thus gnfa-is-reachable ?𝒜′ (State u) x (State v) using assms(1,2)
  proof (induction x arbitrary: u)
    case (Cons c x)
      then obtain v′ where v′-props: (u,c,v′) ∈ Δ 𝒜
          LTS-is-reachable (Δ 𝒜) v′ x v by force
      hence v′ ∈ SemiAutomaton.𝒬 𝒜
          using Δ-consistent by simp-all
      with Cons and v′-props
          have gnfa-is-reachable ?𝒜′ (State v′) x (State v) by simp
      moreover from v′-props have [c] ∈ δ ?𝒜′ (State u) (State v′) by simp
      moreover have c # x = [c] @ x and State u ∈ 𝒬 ?𝒜′
          using Cons by simp-all
      ultimately show ?case by (blast intro: gnfa-is-reachable-rev)
  qed force

next

  assume gnfa-is-reachable ?𝒜′ (State u) x (State v)
  thus LTS-is-reachable (Δ 𝒜) u x v
```

proof (*induction* $?\mathcal{A}'$ (*State u*) *x* (*State v*)
arbitrary: *v* *rule*: *gnfa-is-reachable.induct*)
case (*gnfa-step x x1 x2* v')
from ‹$x2 \in \delta\ ?\mathcal{A}'\ v'$ (*State v*)› **have** $v' \neq$ *End* **by** *fastforce*
moreover from ‹*gnfa-is-reachable* $?\mathcal{A}'$ (*State u*) *x1* v'›
have $v' \neq$ *Start* **by** (*rule gnfa-is-reachableE*, *simp*, *fastforce*)
ultimately obtain v'' **where** $v' =$ *State* v'' **by** (*cases* v', *simp-all*)
with *gnfa-step* **have** *LTS-is-reachable* ($\Delta\ \mathcal{A}$) *u x1* v'' **by** *simp*
moreover obtain *c* **where** $(v'',c,v) \in \Delta\ \mathcal{A}$ **and** $x2 = [c]$
using ‹$x2 \in \delta\ ?\mathcal{A}'\ v'$ (*State v*)› **and** ‹$v' =$ *State* v''› **by** *force*
ultimately show *LTS-is-reachable* ($\Delta\ \mathcal{A}$) *u x v*
using ‹$x = x1$ @ $x2$› **by** *fastforce*
qed *simp*
qed

If a word is in the language of a GNFA obtained from an NFA, this obtains the corresponding initial state and finals state in the original NFA.

lemma *nfa-to-gnfa-Start-EndE*:
assumes $x \in$ *gnfa-*$\mathcal{L}$ (*nfa-to-gnfa* $\mathcal{A}$)
obtains *u v* **where** *gnfa-is-reachable* (*nfa-to-gnfa* $\mathcal{A}$) (*State u*) *x* (*State v*)
and $u \in \mathcal{I}\ \mathcal{A}$ **and** $v \in \mathcal{F}\ \mathcal{A}$
proof−
case *goal1*
let $?\mathcal{A}' =$ *nfa-to-gnfa* $\mathcal{A}$
from *assms* **obtain** v' *x1 x2* **where** *gnfa-is-reachable* $?\mathcal{A}'$ *Start x1* v' **and**
$x2 \in \delta\ ?\mathcal{A}'\ v'$ *End* **and** $x = x1$ @ $x2$
unfolding *gnfa-*$\mathcal{L}$*-def* **by** *blast*
moreover from ‹$x2 \in \delta\ ?\mathcal{A}'\ v'$ *End*› **obtain** *v* **where** $v' =$ *State v* **and**
$x2 = []$ **and** $v \in \mathcal{F}\ \mathcal{A}$ **unfolding** *nfa-to-gnfa-def*
by (*cases* v', *auto split*: *split-if-asm*)
ultimately have *gnfa-is-reachable* $?\mathcal{A}'$ *Start x* (*State v*) **and**
Start $\neq$ *State v* **by** *simp-all*
from *gnfa-is-reachable-revE2*[*OF this*]
obtain u' *x1 x2* **where** $x = x1$ @ $x2$ **and** $x1 \in \delta\ ?\mathcal{A}'$ *Start* u' **and**
gnfa-is-reachable $?\mathcal{A}'\ u'$ *x2* (*State v*) **.**
moreover from ‹$x1 \in \delta\ ?\mathcal{A}'$ *Start* u'› **obtain** *u* **where** $u' =$ *State u* **and**
$x1 = []$ **and** $u \in \mathcal{I}\ \mathcal{A}$ **unfolding** *nfa-to-gnfa-def*
by (*cases* u', *auto split*: *split-if-asm*)
ultimately have *gnfa-is-reachable* $?\mathcal{A}'$ (*State u*) *x* (*State v*) **by** *simp*
with ‹$u \in \mathcal{I}\ \mathcal{A}$› **and** ‹$v \in \mathcal{F}\ \mathcal{A}$› **and** *goal1* **show** *?thesis* **by** *simp*
qed

lemma (**in** *NFA*) *nfa-to-gnfa-correct*[*simp*]:
gnfa-$\mathcal{L}$ (*nfa-to-gnfa* $\mathcal{A}$) $= \mathcal{L}\ \mathcal{A}$ (**is** *gnfa-*$\mathcal{L}$ $?\mathcal{A}' =$ -)
proof (*intro equalityI subsetI*)
fix *x* **assume** $x \in$ *gnfa-*$\mathcal{L}$ $?\mathcal{A}'$
then obtain *u v* **where** *uv-props*: *gnfa-is-reachable* $?\mathcal{A}'$ (*State u*) *x* (*State v*)
$u \in \mathcal{I}\ \mathcal{A}$ $v \in \mathcal{F}\ \mathcal{A}$ **by** (*blast elim*: *nfa-to-gnfa-Start-EndE*)
hence $u \in$ *SemiAutomaton.*$\mathcal{Q}\ \mathcal{A}$ **and** $v \in$ *SemiAutomaton.*$\mathcal{Q}\ \mathcal{A}$

using $\mathcal{I}$*-consistent* $\mathcal{F}$*-consistent* **by** *blast+*
from *LTS-is-reachable-iff-gnfa-is-reachable*[*OF this assms*] **and** *uv-props*(*1*)
have *LTS-is-reachable* (Δ $\mathcal{A}$) *u x v* **..**
thus $x \in \mathcal{L}\ \mathcal{A}$ **using** *uv-props*(*2*,*3*)
unfolding $\mathcal{L}$*-def NFA-accept-def*[*abs-def*] **by** *blast*
next

interpret *GNFA ?*$\mathcal{A}'$ **using** *nfa-to-gnfa-wf*[*OF assms*] **.**
fix *x* **assume** $x \in \mathcal{L}\ \mathcal{A}$
then obtain *u v* **where** *uv-props*: $u \in \mathcal{I}\ \mathcal{A}$ $\quad v \in \mathcal{F}\ \mathcal{A}$
LTS-is-reachable (Δ $\mathcal{A}$) *u x v*
unfolding $\mathcal{L}$*-def NFA-accept-def*[*abs-def*] **by** *blast*
have *gnfa-is-reachable ?*$\mathcal{A}'$ *Start* [] (*State u*)
using *start-correct uv-props* $\mathcal{I}$*-consistent* **by** *force*
also from *uv-props* **have** $u \in$ *SemiAutomaton.*$\mathcal{Q}$ $\mathcal{A}$ **and** $v \in$ *SemiAutomaton.*$\mathcal{Q}$ $\mathcal{A}$
using $\mathcal{I}$*-consistent* $\mathcal{F}$*-consistent* **by** *blast+*
from *LTS-is-reachable-iff-gnfa-is-reachable*[*OF this assms*] **and** *uv-props*
have *gnfa-is-reachable ?*$\mathcal{A}'$ (*State u*) *x* (*State v*) **by** *simp*
also have *gnfa-is-reachable ?*$\mathcal{A}'$ (*State v*) [] *End*
using *end-correct uv-props* $\mathcal{F}$*-consistent* **by** *force*
finally show $x \in$ *gnfa-*$\mathcal{L}$ *?*$\mathcal{A}'$ **unfolding** *gnfa-*$\mathcal{L}$*-def* **by** *simp*
qed

5.4.4 Conversion from NFA to RExp

The invariant of the conversion algorithms main loop. The GNFA is well-formed, it is a subautomaton of teh original GNFA and all remaining states are connected with the same languages as initially.

definition *nfa-to-rexp-invar* **where**
nfa-to-rexp-invar $\mathcal{A}\ \mathcal{A}' \equiv$ *GNFA* $\mathcal{A}' \wedge \mathcal{Q}\ \mathcal{A}' \subseteq \mathcal{Q}\ \mathcal{A} \wedge$
($\forall u\ v\ x.\ u \in \mathcal{Q}\ \mathcal{A}' \wedge v \in \mathcal{Q}\ \mathcal{A}' \longrightarrow$
(*gnfa-is-reachable* $\mathcal{A}'$ *u x v* $\longleftrightarrow$ *gnfa-is-reachable* $\mathcal{A}$ *u x v*))

lemma *nfa-to-rexp-invarI*[*intro*]:
assumes *GNFA* $\mathcal{A}'$ **and** $\mathcal{Q}\ \mathcal{A}' \subseteq \mathcal{Q}\ \mathcal{A}$ **and** $\bigwedge u\ v\ x.\ [\![u \in \mathcal{Q}\ \mathcal{A}';\ v \in \mathcal{Q}\ \mathcal{A}']\!] \Longrightarrow$
gnfa-is-reachable $\mathcal{A}'$ *u x v* $\longleftrightarrow$ *gnfa-is-reachable* $\mathcal{A}$ *u x v*
shows *nfa-to-rexp-invar* $\mathcal{A}\ \mathcal{A}'$
unfolding *nfa-to-rexp-invar-def* **by** (*intro conjI allI impI*,
fact assms(*1*), *fact assms*(*2*), *clarify*, *fact assms*(*3*))

lemma *nfa-to-rexp-invarD*[*dest*]:
assumes *nfa-to-rexp-invar* $\mathcal{A}\ \mathcal{A}'$
shows *GNFA* $\mathcal{A}'$ **and** $\mathcal{Q}\ \mathcal{A}' \subseteq \mathcal{Q}\ \mathcal{A}$ **and** $\bigwedge u\ v\ x.\ [\![u \in \mathcal{Q}\ \mathcal{A}';\ v \in \mathcal{Q}\ \mathcal{A}']\!] \Longrightarrow$
gnfa-is-reachable $\mathcal{A}'$ *u x v* $\longleftrightarrow$ *gnfa-is-reachable* $\mathcal{A}$ *u x v*
using *assms* **unfolding** *nfa-to-rexp-invar-def* **by** *blast+*

Abstract algorithm that computes the language of a given NFA. This is, of course, not executable and will later be refined to use regular expressions as a concrete representation of these languages.

```
definition nfa-to-rexp-abstr where
nfa-to-rexp-abstr 𝒜 ≡ do {
  𝒜 ← SPEC (λ𝒜'. 𝒜' = nfa-to-gnfa 𝒜);
  𝒜 ←
    WHILE_T^(nfa-to-rexp-invar 𝒜) (λ𝒜. 𝒬 𝒜 ≠ {Start, End}) (λ𝒜. do {
      q ← SPEC (λq. q ∈ 𝒬 𝒜 − {Start, End});
      𝒜 ← SPEC (λ𝒜'. 𝒜' = gnfa-contract 𝒜 q);
      RETURN 𝒜
    }) 𝒜;
  RETURN (δ 𝒜 Start End)
}
```

The algorithm returns the language of the NFA

```
lemma (in NFA) nfa-to-rexp-abstr-correct:
  nfa-to-rexp-abstr 𝒜 ≤ SPEC (λL. L = ℒ 𝒜)
unfolding nfa-to-rexp-abstr-def
proof (intro refine-vcg)

  show wf (measure (card ∘ 𝒬)) by simp
next

  fix 𝒜' assume 𝒜' = nfa-to-gnfa 𝒜
  with assms show nfa-to-rexp-invar 𝒜' 𝒜' by blast
next

  case (goal3 𝒜' 𝒜'' q 𝒜''' )

    note inv = nfa-to-rexp-invarD[OF goal3(2)]
    then interpret GNFA 𝒜'' by simp
    from goal3 have [simp]: 𝒬 𝒜''' = 𝒬 𝒜'' − {q}
        unfolding gnfa-contract-def by simp


    have nfa-to-rexp-invar 𝒜' 𝒜'''
    proof (intro nfa-to-rexp-invarI)
      from goal3 and wf show GNFA 𝒜''' by blast
      from inv(2) show 𝒬 𝒜''' ⊆ 𝒬 𝒜' by (auto simp: gnfa-contract-def)
      fix u v x assume uv-assms: u ∈ 𝒬 𝒜'''    v ∈ 𝒬 𝒜'''
      with ‹q ∈ 𝒬 𝒜'' − {Start, End}› and gnfa-contract-correct
          have gnfa-is-reachable (gnfa-contract 𝒜'' q) u x v ⟷
              gnfa-is-reachable 𝒜'' u x v
          by (intro gnfa-contract-correct, simp-all)
      also have gnfa-contract 𝒜'' q = 𝒜''' using goal3 by simp
      also from uv-assms have u ∈ 𝒬 𝒜'' and v ∈ 𝒬 𝒜''
          by (auto simp add: gnfa-contract-def)
      note inv(3)[OF this]
      finally show gnfa-is-reachable 𝒜''' u x v =
          gnfa-is-reachable 𝒜' u x v .
```

qed

moreover have *card* ($\mathcal{Q}$ $\mathcal{A}'''$) < *card* ($\mathcal{Q}$ $\mathcal{A}''$) **using** *goal3* **and** *card-Diff1-less*[*OF finite-$\mathcal{Q}$*] **by** *simp*

ultimately show *nfa-to-rexp-invar* $\mathcal{A}'$ $\mathcal{A}'''$ ∧ ($\mathcal{A}'''$,$\mathcal{A}''$) ∈ *measure* (*card* ∘ $\mathcal{Q}$) **using** *goal3* **by** *simp*

next

case (*goal4* $\mathcal{A}'$ $\mathcal{A}''$)
note *inv* = *nfa-to-rexp-invarD*[*OF goal4*(*2*)]
from *inv*(*1*) **interpret** *GNFA* $\mathcal{A}''$ **.**
from *gnfa-$\mathcal{L}$-Start-End* **and** *goal4* **have** *GNFA-rec.δ* $\mathcal{A}''$ *Start End* = *gnfa-$\mathcal{L}$* $\mathcal{A}''$ **by** *simp*
also from *inv*(*3*) **have** *gnfa-$\mathcal{L}$* $\mathcal{A}''$ = *gnfa-$\mathcal{L}$* $\mathcal{A}'$
unfolding *gnfa-$\mathcal{L}$-def* **using** *start-correct*(*1*) **and** *end-correct*(*1*) **by** *blast*
also have *gnfa-$\mathcal{L}$* $\mathcal{A}'$ = $\mathcal{L}$ $\mathcal{A}$ **using** *assms* **and** *goal4* **by** *simp*
finally show *GNFA-rec.δ* $\mathcal{A}''$ *Start End* = $\mathcal{L}$ $\mathcal{A}$ **.**
qed

Refinement step 1

Implementation of *gnfa-contract* and *nfa-to-gnfa*

definition *gnfa-contract-invar1* **where**
gnfa-contract-invar1 $\mathcal{A}$ *q* *Q'* $\mathcal{A}'$ ≡ ($\mathcal{Q}$ $\mathcal{A}'$ = $\mathcal{Q}$ $\mathcal{A}$) ∧ (∀ *u v*.
($u \in Q' \longrightarrow \delta\ \mathcal{A}'\ u\ v = \delta\ \mathcal{A}\ u\ v$) ∧
($u \notin Q' \longrightarrow \delta\ \mathcal{A}'\ u\ v = \delta\ \mathcal{A}\ u\ v \cup$ *gnfa-subsumed-transitions* $\mathcal{A}$ *q u v*))

lemma *gnfa-contract-invar1I*[*intro*]:
assumes $\mathcal{Q}$ $\mathcal{A}'$ = $\mathcal{Q}$ $\mathcal{A}$ **and**
$\bigwedge u\ v.\ u \in Q' \Longrightarrow \delta\ \mathcal{A}'\ u\ v = \delta\ \mathcal{A}\ u\ v$ **and**
$\bigwedge u\ v.\ u \notin Q' \Longrightarrow \delta\ \mathcal{A}'\ u\ v = \delta\ \mathcal{A}\ u\ v \cup$
gnfa-subsumed-transitions $\mathcal{A}$ *q u v*
shows *gnfa-contract-invar1* $\mathcal{A}$ *q Q'* $\mathcal{A}'$
using *assms* **unfolding** *gnfa-contract-invar1-def* **by** *simp*

lemma *gnfa-contract-invar1D*[*dest*]:
assumes *gnfa-contract-invar1* $\mathcal{A}$ *q Q'* $\mathcal{A}'$
shows $\mathcal{Q}$ $\mathcal{A}'$ = $\mathcal{Q}$ $\mathcal{A}$ **and**
$\bigwedge u\ v.\ u \in Q' \Longrightarrow \delta\ \mathcal{A}'\ u\ v = \delta\ \mathcal{A}\ u\ v$ **and**
$\bigwedge u\ v.\ u \notin Q' \Longrightarrow \delta\ \mathcal{A}'\ u\ v = \delta\ \mathcal{A}\ u\ v \cup$
gnfa-subsumed-transitions $\mathcal{A}$ *q u v*
using *assms* **unfolding** *gnfa-contract-invar1-def* **by** *simp-all*

definition *gnfa-contract-invar2* **where**

gnfa-contract-invar2 $\equiv \lambda\mathcal{A}\ \mathcal{A}'\ q\ u\ Q'\ \mathcal{A}''.\ \mathcal{Q}\ \mathcal{A}'' = \mathcal{Q}\ \mathcal{A}\ \wedge$
$(\forall v.\ (\forall u'.\ u' \neq u \longrightarrow \delta\ \mathcal{A}''\ u'\ v = \delta\ \mathcal{A}'\ u'\ v)\ \wedge$
$(v \in Q' \longrightarrow \delta\ \mathcal{A}''\ u\ v = \delta\ \mathcal{A}'\ u\ v)\ \wedge$
$(v \notin Q' \longrightarrow \delta\ \mathcal{A}''\ u\ v = \delta\ \mathcal{A}'\ u\ v \cup$ *gnfa-subsumed-transitions* $\mathcal{A}\ q\ u\ v))$

lemma *gnfa-contract-invar2I*[*intro*]:
assumes $\mathcal{Q}\ \mathcal{A}'' = \mathcal{Q}\ \mathcal{A}$ **and**
$\bigwedge u'\ v.\ u' \neq u \Longrightarrow \delta\ \mathcal{A}''\ u'\ v = \delta\ \mathcal{A}'\ u'\ v$ **and**
$\bigwedge v.\ v \in Q' \Longrightarrow \delta\ \mathcal{A}''\ u\ v = \delta\ \mathcal{A}'\ u\ v$ **and**
$\bigwedge v.\ v \notin Q' \Longrightarrow \delta\ \mathcal{A}''\ u\ v = \delta\ \mathcal{A}'\ u\ v \cup$
gnfa-subsumed-transitions $\mathcal{A}\ q\ u\ v$
shows *gnfa-contract-invar2* $\mathcal{A}\ \mathcal{A}'\ q\ u\ Q'\ \mathcal{A}''$
using *assms* **unfolding** *gnfa-contract-invar2-def* **by** *simp*

lemma *gnfa-contract-invar2D*[*dest*]:
assumes *gnfa-contract-invar2* $\mathcal{A}\ \mathcal{A}'\ q\ u\ Q'\ \mathcal{A}''$
shows $\mathcal{Q}\ \mathcal{A}'' = \mathcal{Q}\ \mathcal{A}$ **and**
$\bigwedge u'\ v.\ u' \neq u \Longrightarrow \delta\ \mathcal{A}''\ u'\ v = \delta\ \mathcal{A}'\ u'\ v$ **and**
$\bigwedge v.\ v \in Q' \Longrightarrow \delta\ \mathcal{A}''\ u\ v = \delta\ \mathcal{A}'\ u\ v$ **and**
$\bigwedge v.\ v \notin Q' \Longrightarrow \delta\ \mathcal{A}''\ u\ v = \delta\ \mathcal{A}'\ u\ v \cup$
gnfa-subsumed-transitions $\mathcal{A}\ q\ u\ v$
using *assms* **unfolding** *gnfa-contract-invar2-def* **by** *simp-all*

abbreviation *gnfa-contract-impl-update-δ* $\equiv \lambda\mathcal{A}\ u\ q\ v\ u'\ v'.$
(*if* $u'{=}u \wedge v'{=}v$ *then* $\delta\ \mathcal{A}\ u\ v \cup \delta\ \mathcal{A}\ u\ q$ @@ *star* $(\delta\ \mathcal{A}\ q\ q)$ @@ $\delta\ \mathcal{A}\ q\ v$
else $\delta\ \mathcal{A}\ u'\ v'$)

definition *gnfa-contract-impl* **where**
gnfa-contract-impl $\mathcal{A}\ q \equiv$ *do* {
$\mathcal{A} \leftarrow$ *SPEC* $(\lambda\mathcal{A}'.\ \mathcal{A}' =$ *gnfa-remove-state* $\mathcal{A}\ q)$;
$\mathcal{A} \leftarrow$ *FOREACH*$^{\textit{gnfa-contract-invar1}\ \mathcal{A}\ q}$ $\{u{\in}\mathcal{Q}\ \mathcal{A}.\ \delta\ \mathcal{A}\ u\ q \neq \{\}\}$ $(\lambda u\ \mathcal{A}'.$
FOREACH$^{\textit{gnfa-contract-invar2}\ \mathcal{A}\ \mathcal{A}'\ q\ u}$ $\{v{\in}\mathcal{Q}\ \mathcal{A}.\ \delta\ \mathcal{A}\ q\ v \neq \{\}\}$ $(\lambda v\ \mathcal{A}'.$
RETURN $(\!|\ \mathcal{Q} = \mathcal{Q}\ \mathcal{A}',\ \delta =$ *gnfa-contract-impl-update-δ* $\mathcal{A}'\ u\ q\ v\ |\!))\ \mathcal{A}'$
$)\ \mathcal{A}$;
ASSERT (*GNFA* $\mathcal{A}$);
RETURN $\mathcal{A}$
}

lemma (**in** *GNFA*) *gnfa-contract-impl-correct*:
assumes $q \neq$ *Start* **and** $q \neq$ *End*
shows *gnfa-contract-impl* $\mathcal{A}\ q \leq$ *SPEC* $(\lambda\mathcal{A}'.\ \mathcal{A}' =$ *gnfa-contract* $\mathcal{A}\ q)$
unfolding *gnfa-contract-impl-def*
proof (*intro refine-vcg*)
case *goal1* **thus** *?case* **using** *finite-*$\mathcal{Q}$ **by** *simp*
next
case *goal2* **show** *?case* **by** (*rule gnfa-contract-invar1I*, *simp*, *simp*,

```
    unfold gnfa-subsumed-transitions-def, clarsimp)
next
  case goal3 thus ?case using finite-𝒬 by simp
next
  case goal4
    note inv = gnfa-contract-invar1D(1)[OF goal4(4)]
    thus ?case by (intro gnfa-contract-invar2I, simp, simp, simp,
        unfold gnfa-subsumed-transitions-def, clarsimp)
next
  case (goal5 𝒜 u it_u 𝒜′ v it_v 𝒜′′ )

    from goal5(2,3,5,6) have uv-in-Q: u ∈ 𝒬 𝒜    v ∈ 𝒬 𝒜  by auto
    note invu = gnfa-contract-invar1D[OF goal5(4)]
    note invv = gnfa-contract-invar2D[OF goal5(7)]
    from goal5 have q ∉ it_u    q ≠ u    q ∉ it_v    q ≠ v by auto
    with invu invv ‹u ∈ it_u› ‹v ∈ it_v› have q-props: δ 𝒜′′ u q = δ 𝒜 u q
        δ 𝒜′′ q q = δ 𝒜 q q    ⋀v′. δ 𝒜′′ q v′ = δ 𝒜 q v′
        by (simp-all add: gnfa-subsumed-transitions-def gnfa-remove-state-def)
    show ?case proof
      fix v′ assume v′-props: v′ ∉ it_v − {v}
      show δ (| 𝒬 = 𝒬 𝒜′′, δ = gnfa-contract-impl-update-δ 𝒜′′ u q v |) u v′
          = δ 𝒜′ u v′ ∪ gnfa-subsumed-transitions 𝒜 q u v′
        apply (cases v = v′, insert uv-in-Q goal5(1,5) v′-props invv)
        apply (simp-all add: gnfa-subsumed-transitions-def q-props)
        done
    qed (simp-all add: invv)
next
  case (goal6 𝒜 u it_u 𝒜′ 𝒜′′)
    note invu = gnfa-contract-invar1D[OF goal6(4)]
    note invv = gnfa-contract-invar2D[OF goal6(5)]
    show ?case proof
      fix u′ v′ assume u′ ∈ it_u − {u}
      with invu(2) invv(2) show δ 𝒜′′ u′ v′ = δ 𝒜 u′ v′ by simp
    next
      fix u′ v′ assume u′-props: u′ ∉ it_u − {u}
      thus δ 𝒜′′ u′ v′ = δ 𝒜 u′ v′ ∪ gnfa-subsumed-transitions 𝒜 q u′ v′
        apply (cases u′ = u)
        apply (insert invu(2) invv(4) ‹u ∈ it_u›,
            auto simp: gnfa-subsumed-transitions-def)[1]
        apply (simp add: u′-props invu(3) invv(2))
      done
    qed (simp add: invu invv)
next
  case (goal7 𝒜′ 𝒜′′)
    note inv = gnfa-contract-invar1D(1,3)[OF goal7(2)]
    with goal7 have δ 𝒜′′ = (λu v. δ 𝒜 u v ∪
            gnfa-subsumed-transitions 𝒜 q u v)
        by (intro ext, simp)
    with inv and goal7 have 𝒜′′ = gnfa-contract 𝒜 q by simp
```

```
  thus ?case using assms by simp
next
 case (goal8 𝒜' 𝒜'')
  note inv = gnfa-contract-invar1D(1,3)[OF goal8(2)]
  with goal8(1) have δ 𝒜'' = (λu v. δ 𝒜 u v ∪
          gnfa-subsumed-transitions 𝒜 q u v)
    by (intro ext, simp)
  with inv and goal8 show ?case by simp
qed

lemma gnfa-contract-impl-correct':
 fixes q₁::'q gnfastate and 𝒜₁::('q,'a,-) GNFA-rec-scheme
 assumes GNFA 𝒜₁ and q₁ ≠ Start and q₁ ≠ End and
     (q₁,q₂) ∈ Id and (𝒜₁,𝒜₂) ∈ Id
 shows gnfa-contract-impl 𝒜₁ q₁ ≤ SPEC (λ𝒜'. 𝒜' = gnfa-contract 𝒜₂ q₂)
 using assms and GNFA.gnfa-contract-impl-correct by blast

definition nfa-to-gnfa-invar1 where
nfa-to-gnfa-invar1 𝒜 Q ≡ λI' 𝒜'. (𝒬 𝒜' = Q ∧ δ 𝒜' = (λu v. (case u of
        Start ⇒ (case v of
         State v ⇒ if v ∈ ℐ 𝒜 − I' then {[]} else {} |
         - ⇒ {}) |
        End ⇒ {} |
        State u ⇒ {})))

definition nfa-to-gnfa-invar2 where
nfa-to-gnfa-invar2 𝒜 Q F' 𝒜' ≡ (𝒬 𝒜' = Q ∧ δ 𝒜' = (λu v. (case u of
        Start ⇒ (case v of
         State v ⇒ if v ∈ ℐ 𝒜 then {[]} else {} |
         - ⇒ {}) |
        End ⇒ {} |
        State u ⇒ (case v of
         Start ⇒ {} |
         End ⇒ if u ∈ ℱ 𝒜 − F' then {[]} else {} |
         State v ⇒ {}
        ))))

definition nfa-to-gnfa-invar3 where
nfa-to-gnfa-invar3 𝒜 Q Δ' 𝒜' ≡ (𝒬 𝒜' = Q ∧ δ 𝒜' = (λu v. (case u of
        Start ⇒ (case v of
         State v ⇒ if v ∈ ℐ 𝒜 then {[]} else {} |
         - ⇒ {}) |
        End ⇒ {} |
        State u ⇒ (case v of
         Start ⇒ {} |
         End ⇒ if u ∈ ℱ 𝒜 then {[]} else {} |
         State v ⇒ {[c] |c. (u,c,v) ∈ Δ 𝒜 − Δ'}
        ))))
```

```
definition nfa-to-gnfa-impl where
nfa-to-gnfa-impl 𝒜 = do {
  𝒜' ← SPEC (λ𝒜'. 𝒜' = (| 𝒬 = {Start, End} ∪ State ` SemiAutomaton.𝒬 𝒜,
      δ = λu v. {} |));
  𝒜'' ← FOREACH^{nfa-to-gnfa-invar1 𝒜 (𝒬 𝒜')} (ℐ 𝒜) (λv 𝒜'.
        RETURN (| 𝒬 = 𝒬 𝒜', δ = λu' v'.
           if u'=Start ∧ v'=State v then {[]} else δ 𝒜' u' v' |)) 𝒜';
  𝒜'' ← FOREACH^{nfa-to-gnfa-invar2 𝒜 (𝒬 𝒜')} (ℱ 𝒜) (λu 𝒜'.
        RETURN (| 𝒬 = 𝒬 𝒜', δ = λu' v'.
           if u'=State u ∧ v'=End then {[]} else δ 𝒜' u' v' |)) 𝒜'';
  𝒜'' ← FOREACH^{nfa-to-gnfa-invar3 𝒜 (𝒬 𝒜')} (Δ 𝒜) (λ(u,c,v) 𝒜'.
        RETURN (| 𝒬 = 𝒬 𝒜', δ = λu' v'. if u'=State u ∧ v'=State v then
           insert ([c]) (δ 𝒜' u' v') else δ 𝒜' u' v' |)) 𝒜'';
  RETURN 𝒜''
}
```

```
lemma (in NFA) nfa-to-gnfa-impl-correct:
  nfa-to-gnfa-impl 𝒜 ≤ SPEC (λ𝒜'. 𝒜' = nfa-to-gnfa 𝒜)
unfolding nfa-to-gnfa-impl-def
proof (intro refine-vcg)
  case goal1 show ?case using finite-ℐ . next
  case goal2 thus ?case unfolding nfa-to-gnfa-invar1-def
      by (simp, intro ext, simp split: gnfastate.split) next
  case goal3 thus ?case unfolding nfa-to-gnfa-invar1-def
      by (simp, intro ext, auto split: gnfastate.split) next
  case goal4 show ?case using finite-ℱ . next
  case goal5 thus ?case unfolding nfa-to-gnfa-invar1-def nfa-to-gnfa-invar2-def
      by (simp, intro ext, simp split: gnfastate.split) next
  case goal6 thus ?case unfolding nfa-to-gnfa-invar2-def
      by (simp, intro ext, auto split: gnfastate.split) next
  case goal7 show ?case using finite-Δ . next
  case goal8 thus ?case unfolding nfa-to-gnfa-invar2-def nfa-to-gnfa-invar3-def
      by (simp, intro ext, simp split: gnfastate.split) next
  case goal9 thus ?case unfolding nfa-to-gnfa-invar3-def[abs-def]
      by (simp split: prod.split, clarsimp, intro ext,
          auto split: gnfastate.split)
next
  case (goal10 d - - 𝒜' )
    hence 𝒬 𝒜' = 𝒬 (nfa-to-gnfa 𝒜)
        unfolding nfa-to-gnfa-invar3-def by simp
    moreover have δ 𝒜' = δ (nfa-to-gnfa 𝒜) using goal10(4)
        unfolding nfa-to-gnfa-def nfa-to-gnfa-invar3-def
        by (intro ext, simp split: gnfastate.split)
    ultimately show ?case using goal10 by simp
qed
```

definition *nfa-to-rexp-impl* :: $('q,'a,\text{-})$ *NFA-rec-scheme* $\Rightarrow$ *'a lang nres* **where**
nfa-to-rexp-impl $\mathcal{A} \equiv$ *do* {
  $\mathcal{A} \leftarrow$ *nfa-to-gnfa-impl* $\mathcal{A}$;
  $\mathcal{A} \leftarrow$
    $WHILE_T$ $(\lambda\mathcal{A}.\ \exists\, q{\in}\mathcal{Q}\ \mathcal{A}.\ q \neq Start \wedge q \neq End)$ $(\lambda\mathcal{A}.$ *do* {
      $q \leftarrow SPEC\ (\lambda q.\ q \in \mathcal{Q}\ \mathcal{A} - \{Start,\ End\})$;
      $\mathcal{A} \leftarrow$ *gnfa-contract-impl* $\mathcal{A}\ q$;
      RETURN $\mathcal{A}$
    }) $\mathcal{A}$;
  RETURN $(\delta\ \mathcal{A}\ Start\ End)$
}

lemma (**in** *NFA*) *nfa-to-rexp-impl-refine*:
  nfa-to-rexp-impl $\mathcal{A} \leq \Downarrow Id$ (*nfa-to-rexp-abstr* $\mathcal{A}$)
unfolding *nfa-to-rexp-impl-def nfa-to-rexp-abstr-def*
apply (*refine-rcg Id-refine single-valued-Id*)
apply (*fact nfa-to-gnfa-impl-correct*)
apply *simp*
apply (*blast intro*: *GNFA.GNFA-wf*)
apply *simp*
apply (*drule nfa-to-rexp-invarD*(*1*), *blast intro*: *gnfa-contract-impl-correct'*)[]
apply *simp-all*
done

Refinement step 2

concretisation of *GNFA* to (Q,δ,P,S), where P and S are predecessor and successor maps

definition *gnfa-α* $\equiv \lambda(Q,\delta,\text{-},\text{-}).\ (\!|\ \mathcal{Q} = \{Start,\ End\} \cup State\text{`}Q,\ \delta = \delta\ |\!)$
definition *gnfa-invar* $\equiv \lambda(Q,\delta,P,S).$ *let* $\mathcal{A} =$ *gnfa-α* (Q,δ,P,S) *in*
    $GNFA\ \mathcal{A} \wedge (\forall\, q{\in}\mathcal{Q}\ \mathcal{A}.\ P\ q = Some\ \{u{\in}\mathcal{Q}\ \mathcal{A}.\ \delta\ u\ q \neq \{\}\} \wedge$
        $S\ q = Some\ \{v{\in}\mathcal{Q}\ \mathcal{A}.\ \delta\ q\ v \neq \{\}\})$

lemma *gnfa-invarI*[*intro*]:
  fixes δ
  assumes *GNFA* (*gnfa-α* (Q,δ,P,S)) **and**
    $\bigwedge q.\ q{\in}\mathcal{Q}$ (*gnfa-α* (Q,δ,P,S)) $\Longrightarrow$
      $P\ q = Some\ \{u{\in}\mathcal{Q}$ (*gnfa-α* (Q,δ,P,S)). $\delta\ u\ q \neq \{\}\}$
    $\bigwedge q.\ q{\in}\mathcal{Q}$ (*gnfa-α* (Q,δ,P,S)) $\Longrightarrow$
      $S\ q = Some\ \{v{\in}\mathcal{Q}$ (*gnfa-α* (Q,δ,P,S)). $\delta\ q\ v \neq \{\}\}$
  shows *gnfa-invar* (Q,δ,P,S)
  using *assms* **unfolding** *gnfa-invar-def* **by** *simp*

abbreviation *gnfa-refrel* $\equiv$ *br gnfa-α gnfa-invar*

lemma *single-valued-gnfa-refrel*: *single-valued gnfa-refrel*
  by (*fact br-sv*)

lemma *gnfa-refrel-imp-GNFA*[*simp*,*dest*]:

assumes $(\mathcal{A}_1,\mathcal{A}_2) \in$ *gnfa-refrel*
shows *GNFA* $\mathcal{A}_2$
using *assms* **unfolding** *gnfa-invar-def br-def* **by** (*cases* $\mathcal{A}_1$, *simp add*: *Let-def*)

lemma *GNFA-PS-correct*:
fixes $\mathcal{A}$::('q,'a,-) *GNFA-rec-scheme* **and** δ
assumes $((Q,\delta,P,S), \mathcal{A}) \in$ *gnfa-refrel* **and** $q \in \mathcal{Q}\ \mathcal{A}$
shows $P\ q = Some\ \{u{\in}\mathcal{Q}\ \mathcal{A}.\ GNFA\text{-}rec.\delta\ \mathcal{A}\ u\ q \neq \{\}\}$
$S\ q = Some\ \{v{\in}\mathcal{Q}\ \mathcal{A}.\ GNFA\text{-}rec.\delta\ \mathcal{A}\ q\ v \neq \{\}\}$
using *assms* **unfolding** *gnfa-*α*-def gnfa-invar-def* **by** (*auto simp add*: *br-def*)

definition *gnfa-remove-state-invar1* **where**
gnfa-remove-state-invar1 $\equiv \lambda Q\ \delta\ P\ q\ it\ P'.\ (\forall v{\in}\{Start,End\} \cup State`Q.$
$(v \in it \longrightarrow P'\ v = P\ v) \wedge$
$(v \notin it \longrightarrow P'\ v = Some\ \{u{\in}\{Start,End\}{\cup}State`Q{-}\{q\}.\ \delta\ u\ v \neq\{\}\}))$

lemma *gnfa-remove-state-invar1I*[*intro*]:
fixes δ
assumes $\bigwedge v.\ v{\in}\{Start,End\} \cup State`Q \Longrightarrow v{\in}it \Longrightarrow P'\ v = P\ v$ **and**
$\bigwedge v.\ v{\in}\{Start,End\} \cup State`Q \Longrightarrow v{\notin}it \Longrightarrow P'\ v =$
$Some\ \{u{\in}\{Start,End\}{\cup}State`Q{-}\{q\}.\ \delta\ u\ v \neq\{\}\}$
shows *gnfa-remove-state-invar1* $Q\ \delta\ P\ q\ it\ P'$
using *assms* **unfolding** *gnfa-remove-state-invar1-def* **by** *simp*

lemma *gnfa-remove-state-invar1D*[*dest*]:
fixes δ
assumes *gnfa-remove-state-invar1* $Q\ \delta\ P\ q\ it\ P'$
shows $\bigwedge v.\ v{\in}\{Start,End\} \cup State`Q \Longrightarrow v{\in}it \Longrightarrow P'\ v = P\ v$ **and**
$\bigwedge v.\ v{\in}\{Start,End\} \cup State`Q \Longrightarrow v{\notin}it \Longrightarrow P'\ v =$
$Some\ \{u{\in}\{Start,End\}{\cup}State`Q{-}\{q\}.\ \delta\ u\ v \neq\{\}\}$
using *assms* **unfolding** *gnfa-remove-state-invar1-def* **by** *blast+*

definition *gnfa-remove-state-invar2* **where**
gnfa-remove-state-invar2 $\equiv \lambda Q\ \delta\ S\ q\ it\ S'.\ (\forall u{\in}\{Start,End\} \cup State`Q.$
$(u \in it \longrightarrow S'\ u = S\ u) \wedge$
$(u \notin it \longrightarrow S'\ u = Some\ \{v{\in}\{Start,End\}{\cup}State`Q{-}\{q\}.\ \delta\ u\ v \neq\{\}\}))$

lemma *gnfa-remove-state-invar2I*[*intro*]:
fixes δ
assumes $\bigwedge u.\ u{\in}\{Start,End\} \cup State`Q \Longrightarrow u{\in}it \Longrightarrow S'\ u = S\ u$ **and**
$\bigwedge u.\ u{\in}\{Start,End\} \cup State`Q \Longrightarrow u{\notin}it \Longrightarrow S'\ u =$
$Some\ \{v{\in}\{Start,End\}{\cup}State`Q{-}\{q\}.\ \delta\ u\ v \neq\{\}\}$
shows *gnfa-remove-state-invar2* $Q\ \delta\ S\ q\ it\ S'$
using *assms* **unfolding** *gnfa-remove-state-invar2-def* **by** *simp*

lemma *gnfa-remove-state-invar2D*[*dest*]:
fixes δ

assumes *gnfa-remove-state-invar2* $Q\ \delta\ S\ q\ it\ S'$
shows $\bigwedge u.\ u \in \{Start, End\} \cup State`Q \Longrightarrow u \in it \Longrightarrow S'\ u = S\ u$ **and**
$\bigwedge u.\ u \in \{Start, End\} \cup State`Q \Longrightarrow u \notin it \Longrightarrow S'\ u =$
$Some\ \{v \in \{Start, End\} \cup State`Q - \{q\}.\ \delta\ u\ v \neq \{\}\}$
using *assms* **unfolding** *gnfa-remove-state-invar2-def* **by** *blast+*

definition $PS\text{-}add\ M\ u\ v \equiv case\ M\ u\ of\ None \Rightarrow M\ |$
$Some\ Mu \Rightarrow M(u \mapsto insert\ v\ Mu)$
definition $PS\text{-}remove\ M\ u\ v \equiv case\ M\ u\ of\ None \Rightarrow M\ |$
$Some\ Mu \Rightarrow M(u \mapsto Mu - \{v\})$
definition $PS\text{-}the\ M\ u \equiv case\ M\ u\ of\ None \Rightarrow \{\}\ |\ Some\ Mu \Rightarrow Mu$

definition *gnfa-remove-state-impl2* **where**
$gnfa\text{-}remove\text{-}state\text{-}impl2 \equiv \lambda(Q,\delta,P,S)\ q.$
$case\ q\ of\ Start \Rightarrow RETURN\ (Q,\delta,P,S)\ |\ End \Rightarrow RETURN\ (Q,\delta,P,S)\ |$
$State\ q' \Rightarrow do\ \{$
$P' \leftarrow FOREACH^{gnfa\text{-}remove\text{-}state\text{-}invar1\ Q\ \delta\ P\ q}\ (PS\text{-}the\ S\ q)$
$(\lambda v\ P.\ RETURN\ (PS\text{-}remove\ P\ v\ q))\ P;$
$S' \leftarrow FOREACH^{gnfa\text{-}remove\text{-}state\text{-}invar2\ Q\ \delta\ S\ q}\ (PS\text{-}the\ P\ q)$
$(\lambda u\ S.\ RETURN\ (PS\text{-}remove\ S\ u\ q))\ S;$
$RETURN\ (Q - \{q'\},\ \delta,\ P',\ S')$
$\}$

abbreviation $gnfa\text{-}state\text{-}refrel \equiv br\ State\ (\lambda\text{-}.\ True)$
lemma *single-valued-gnfa-state-refrel*:
single-valued gnfa-state-refrel **by** (*fact br-sv*)
lemma *gnfa-state-refrel-simp*[*simp*]:
$(q', q) \in gnfa\text{-}state\text{-}refrel \longleftrightarrow q = State\ q'$ **by** (*simp add*: *br-def*)
lemma *gnfa-state-refrelD*[*dest*]:
$(q', q) \in gnfa\text{-}state\text{-}refrel \Longrightarrow q = State\ q'$ **by** *simp*

lemma *gnfa-remove-state-impl2-correct*:
fixes $\mathcal{A}_1 :: ('q, 'a, \text{-})\ GNFA\text{-}rec\text{-}scheme$ **and**
$\delta :: 'q\ gnfastate \Rightarrow 'q\ gnfastate \Rightarrow 'a\ lang$
assumes $(q', q) \in gnfa\text{-}state\text{-}refrel$ **and** $q' \in Q$ **and**
$((Q,\delta,P,S), \mathcal{A}_2) \in gnfa\text{-}refrel$
shows $gnfa\text{-}remove\text{-}state\text{-}impl2\ (Q,\delta,P,S)\ (State\ q') \leq$
$\Downarrow gnfa\text{-}refrel\ (SPEC\ (\lambda\mathcal{A}'.\ \mathcal{A}' = gnfa\text{-}remove\text{-}state\ \mathcal{A}_2\ q))$
unfolding *gnfa-remove-state-impl2-def*
using *assms* **apply** (*simp add*: *br-def*)
apply (*refine-rcg*, *simp add*: *single-valued-def*, *simp*)
proof (*intro refine-vcg*)
case *goal1*
interpret *GNFA* $\mathcal{A}_2$ **using** *assms*(*3*) **by** *blast*
from *goal1 GNFA-PS-correct*(*2*)[*OF assms*(*3*)] *finite-Q assms*

```
      show ?case by (simp add: gnfa-α-def PS-the-def)
next
  case goal2
    from assms have q ∈ Q A₂ unfolding gnfa-α-def by (simp add: br-def)
    with GNFA-PS-correct[OF assms(3)]
      have PS-props: S q = Some {v∈Q A₂. GNFA-rec.δ A₂ q v ≠ {}}
        ⋀v. v ∈ Q A₂ ⟹ P v = Some {u∈Q A₂. GNFA-rec.δ A₂ u v ≠ {}}
        by simp-all
    {
      fix v assume v-props: v ∈ Q A₂    v ∉ the (S q)
      hence q ∉ the (P v) using PS-props assms unfolding gnfa-α-def by simp
      hence P v = Some {u∈Q A₂ − {q}. GNFA-rec.δ A₂ u v ≠{}}
        using PS-props(2)[OF  v-props(1)] by auto
    }
    thus ?case using assms unfolding gnfa-remove-state-invar1-def
      gnfa-α-def PS-the-def gnfa-invar-def by (auto simp: br-def)
next
  case (goal3 v it P′ )
    let ?P″=(PS-remove P′ v (State q′))
    from GNFA-PS-correct[OF assms(3)] assms
      have S-props: S q = Some {v∈Q A₂. GNFA-rec.δ A₂ q v ≠ {}}
      unfolding gnfa-α-def by (simp add: br-def)
    note inv = gnfa-remove-state-invar1D[OF goal3(6)]
    have v-props: v ∈ {Start,End} ∪ State‘Q
      using goal3(4,5) S-props assms(1,3)
      unfolding gnfa-α-def PS-the-def by (auto simp: br-def)
    with GNFA-PS-correct[OF assms(3)] assms
      have P-props: P v = Some {u∈Q A₂. GNFA-rec.δ A₂ u v ≠ {}}
      unfolding gnfa-α-def by (simp add: br-def)
    hence P″-props: ?P″ v = Some ({u∈Q A₂−{State q′}.
        GNFA-rec.δ A₂ u v ≠ {}})
      using  inv(1)[OF v-props ‹v ∈ it›]
      by (auto simp add: PS-remove-def)
    show ?case
      apply (intro gnfa-remove-state-invar1I)
      apply (insert  inv, simp add: PS-remove-def
        split: option.split) []
      apply (rename-tac v′, case-tac v′ = v)
      apply (insert assms, simp add: P″-props gnfa-α-def br-def) []
      apply (insert  inv, simp add: PS-remove-def br-def
        split: option.split) []
      done
next
  case goal4
    interpret GNFA A₂ using assms(3) by blast
    from goal4 GNFA-PS-correct(1)[OF assms(3)] finite-Q assms
      show ?case by (simp add: gnfa-α-def PS-the-def)
next
  case goal5
```

```
    from assms have q ∈ 𝒬 𝒜₂ unfolding gnfa-α-def by (simp add: br-def)
    with GNFA-PS-correct[OF assms(3)]
      have PS-props: P q = Some {u∈𝒬 𝒜₂. GNFA-rec.δ 𝒜₂ u q ≠ {}}
        ⋀u. u ∈ 𝒬 𝒜₂ ⟹ S u = Some {v∈𝒬 𝒜₂. GNFA-rec.δ 𝒜₂ u v ≠ {}}
        by simp-all
    {
     fix u assume u-props: u ∈ 𝒬 𝒜₂    u ∉ the (P q)
     hence q ∉ the (S u) using PS-props assms unfolding gnfa-α-def by simp
     hence S u = Some {v∈𝒬 𝒜₂ − {q}. GNFA-rec.δ 𝒜₂ u v ≠{}}
       using PS-props(2)[OF  u-props(1)] by auto
    }
    thus ?case using assms unfolding gnfa-remove-state-invar2-def
      gnfa-α-def PS-the-def gnfa-invar-def by (auto simp: br-def)
next
  case (goal6 - u it S′ )
    let ?S′′ = (PS-remove S′ u (State q′))
    from GNFA-PS-correct[OF assms(3)] assms
      have P-props: P q = Some {u∈𝒬 𝒜₂. GNFA-rec.δ 𝒜₂ u q ≠ {}}
      unfolding gnfa-α-def by (simp add: br-def)
    note inv = gnfa-remove-state-invar2D[OF goal6(7)]
    have u-props: u ∈ {Start,End} ∪ State‘Q
      using goal6(5,6) P-props assms(1,3)
      unfolding gnfa-α-def PS-the-def by (auto simp: br-def)
    with GNFA-PS-correct[OF assms(3)] assms
      have S-props: S u = Some {v∈𝒬 𝒜₂. GNFA-rec.δ 𝒜₂ u v ≠ {}}
      unfolding gnfa-α-def by (simp add: br-def)
    hence S′′-props: ?S′′ u = Some ({v∈𝒬 𝒜₂−{State q′}.
        GNFA-rec.δ 𝒜₂ u v ≠ {}})
      using  inv(1)[OF u-props ‹u ∈ it›]
      by (auto simp: PS-remove-def)
    show ?case
     apply (intro gnfa-remove-state-invar2I)
     apply (insert  inv, simp add: PS-remove-def
        split: option.split) []
     apply (rename-tac u′, case-tac u′ = u)
     apply (insert assms, simp add: S′′-props gnfa-α-def br-def) []
     apply (insert  inv, simp add: PS-remove-def
        split: option.split) []
     done
next
  case (goal7 P′ S′ )
    let ?𝒜₁′ = (Q − {q′}, δ, P′, S′)

    interpret GNFA 𝒜₂ using assms(3) by blast
    note invP = gnfa-remove-state-invar1D(2)[OF goal7(4)]
    note invS = gnfa-remove-state-invar2D(2)[OF goal7(5)]
    from goal7 have A: gnfa-remove-state 𝒜₂ (State q′) = gnfa-α ?𝒜₁′
      unfolding gnfa-remove-state-def gnfa-α-def by fastforce
    moreover from assms have GNFA (gnfa-α (Q, δ, P, S))
```

```
      unfolding gnfa-invar-def by (simp add: Let-def br-def)
   from GNFA.gnfa-remove-state-wf[OF this]
      have GNFA (gnfa-remove-state 𝒜₂ (State q')) by simp
   ultimately have B: GNFA (gnfa-α ?𝒜₁') by simp
   {fix q assume q∈𝒬 (gnfa-α ?𝒜₁')
    hence q ∈ {Start,End} ∪ State`Q using assms(3)
       unfolding gnfa-α-def by auto
    from invP[OF this] invS[OF this]
       have P' q = Some {u∈𝒬 (gnfa-α ?𝒜₁'). δ u q ≠ {}}
            S' q = Some {v∈𝒬 (gnfa-α ?𝒜₁'). δ q v ≠ {}}
       unfolding gnfa-α-def by auto
   } note P'S'-props = this

   show ?case apply (intro conjI)
      apply (insert A goal7(3), simp) []
      apply ( rule gnfa-invarI)
      apply (insert B , simp) []
      apply (insert P'S'-props, simp-all)
      done
qed
```

```
definition gnfa-contract-update-δ-impl2 where
gnfa-contract-update-δ-impl2 ≡ λδ u q v u' v'.
    (if u'=u ∧ v'=v then δ u v ∪ δ u q @@ star (δ q q) @@ δ q v
                    else δ u' v')
```

```
definition gnfa-contract-impl2 where
gnfa-contract-impl2 ≡ λ(Q,δ,P,S) q. do {
  let Pq = PS-the P q − {q}; let Sq = PS-the S q − {q};
  (Q,δ,P,S) ← gnfa-remove-state-impl2 (Q,δ,P,S) q;
  (Q,δ,P,S) ← FOREACH Pq (λu (Q',δ',P',S').
    FOREACH Sq (λv (Q',δ',P',S').
        RETURN (Q', gnfa-contract-update-δ-impl2 δ' u q v,
            PS-add P' v u, PS-add S' u v))
      (Q',δ',P',S')
  ) (Q,δ,P,S);
  ASSERT (GNFA (gnfa-α (Q,δ,P,S)));
  RETURN (Q,δ,P,S)
}
```

```
lemma gnfa-contract-impl2-refine:
  fixes 𝒜::('q,'a,-) GNFA-rec-scheme and δ
  assumes (q',q) ∈ gnfa-state-refrel and q' ∈ Q and
        ((Q,δ,P,S), 𝒜) ∈ gnfa-refrel
  shows gnfa-contract-impl2 (Q,δ,P,S) (State q') ≤
      ⇓gnfa-refrel (gnfa-contract-impl 𝒜 q)
```

unfolding *gnfa-contract-impl2-def gnfa-contract-impl-def Let-def*
apply (*refine-rcg single-valued-gnfa-refrel inj-on-id*)
apply (*clarify*, *rule gnfa-remove-state-impl2-correct*[*OF assms*(*1*−*3*)])
apply (*insert GNFA-PS-correct*(*1*)[*OF assms*(*3*)] *assms*, *unfold gnfa-α-def*,
  auto simp: *PS-the-def br-def split*: *option.split*) []
apply *simp*
apply (*insert GNFA-PS-correct*(*2*)[*OF assms*(*3*)] *assms*, *unfold gnfa-α-def*,
  auto simp: *br-def PS-the-def split*: *option.split*) []
apply *simp*
defer
apply (*simp add*: *br-def*)
apply (*clarsimp simp*: *br-def*)
proof−
 case (*goal1 Q1 δ1 P1 S1 u* it_u *Q2 δ2 P2 S2 v* it_v *Q3 δ3 P3 S3*)
  note *inv1* = *gnfa-contract-invar1D*[*OF goal1*(*5*)]
  note *inv2* = *gnfa-contract-invar2D*[*OF goal1*(*9*)]

  let *?𝒜1* = *gnfa-α* (*Q1*, *δ1*, *P1*, *S1*)
  let *?𝒜2* = *gnfa-α* (*Q2*, *δ2*, *P2*, *S2*)
  let *?𝒜3* = *gnfa-α* (*Q3*, *δ3*, *P3*, *S3*)
  from *goal1*(*10*) **have** *𝒜1-GNFA*: *GNFA ?𝒜1*
    unfolding *gnfa-invar-def Let-def* **by** *simp*
  from *goal1*(*12*) **have** *𝒜3-GNFA*: *GNFA ?𝒜3*
    unfolding *gnfa-invar-def Let-def* **by** *simp*

  from *inv2*(*1*) *goal1*(*1*) **have** {*Start*, *End*} ∪
     State‘Q1 = {*Start*, *End*} ∪ *State‘Q3*
    unfolding *gnfa-α-def gnfa-remove-state-def* **by** *simp*
  hence *𝒬 ?𝒜1* = *𝒬 ?𝒜3* **using** *inv2*(*1*) **unfolding** *gnfa-α-def* **by** *simp*
  moreover have *uv-in-Q1*: *u* ∈ *𝒬 ?𝒜1* *v* ∈ *𝒬 ?𝒜1* **using** *goal1*
    unfolding *gnfa-α-def* **by** *auto*
  ultimately have *uv-in-Q3*[*simp*]: *u* ∈ *𝒬 ?𝒜3* *v* ∈ *𝒬 ?𝒜3* **by** *simp-all*

  have *GNFA-rec.δ 𝒜* = *δ1* **using** *goal1*(*1*)
    unfolding *gnfa-α-def gnfa-remove-state-def* **by** *simp*
  hence *A*: (|*GNFA-rec.𝒬* = *𝒬 ?𝒜3*,
   δ = *gnfa-contract-impl-update-δ ?𝒜3 u q v*|) =
   gnfa-α (*Q3*, *gnfa-contract-update-δ-impl2 δ3 u*
   (*State q*′) *v*, *PS-add P3 v u*, *PS-add S3 u v*)
  unfolding *gnfa-α-def gnfa-remove-state-def*
   gnfa-contract-update-δ-impl2-def **using** *assms*(*1*) **by** *fastforce*

  have *gnfa-contract-update-δ-impl2 δ3 u q v* =
    gnfa-contract-impl-update-δ ?𝒜3 u q v
    unfolding *gnfa-α-def gnfa-contract-update-δ-impl2-def* **by** *force*
  hence *B*: *GNFA* (|*GNFA-rec.𝒬* = *insert Start* (*insert End* (*State* ‘ *Q3*)),
    δ = *gnfa-contract-update-δ-impl2 δ3 u* (*State q*′) *v*|)
     using *GNFA.GNFA-wf*[*OF 𝒜1-GNFA*] *GNFA.GNFA-wf*[*OF 𝒜3-GNFA*]
assms(*1*)

```
      by (auto simp: GNFA-def gnfa-α-def)

  have PS-props:
    PS-the P q − {q} = {u ∈ Q ?A1. δ1 u q ≠ {}}
    PS-the S q − {q} = {v ∈ Q ?A1. δ1 q v ≠ {}}
    using assms goal1(1) unfolding gnfa-invar-def Let-def gnfa-α-def
    gnfa-remove-state-def PS-the-def by (auto simp: br-def)

  have P3S3-props:
    ⋀q. q ∈Q ?A3 ⟹ P3 q = Some {u ∈ Q ?A3. δ3 u q ≠ {}}
    ⋀q. q ∈Q ?A3 ⟹ S3 q = Some {v ∈ Q ?A3. δ3 q v ≠ {}}
    using goal1(12) unfolding gnfa-invar-def Let-def by simp-all

  from goal1 have q ∉ it_u   q ≠ u   q ∉ it_v by auto
  from inv2(2)[OF this(2)] inv1(3)[OF this(1)] inv2(4)[OF this(3)]
    inv1(2)[OF ‹u ∈ it_u›] goal1(1)
  have δ3-q-props: δ3 u q = δ1 u q   δ3 q v = δ1 q v  by (simp-all add:
     gnfa-α-def gnfa-subsumed-transitions-def gnfa-remove-state-def)

  have δ-uq-qv-nonempty: δ1 u q ≠ {}   δ1 q v ≠ {}
    using assms(1) goal1(2,3,6,7) PS-props by auto
  with δ3-q-props have δ3-uq-qv-nonempty: δ3 u q ≠ {}   δ3 q v ≠ {}
    by simp-all
  hence C: δ3 u q @@ star (δ3 q q) @@ δ3 q v ≠ {} by blast

  show ?case
   apply (intro conjI)
   apply (rule A)
   apply rule
   apply (unfold gnfa-α-def, insert B, simp) []
   unfolding gnfa-contract-update-δ-impl2-def
   apply (rename-tac v', case-tac v' = v,
     insert assms(1) P3S3-props(1) C uv-in-Q3 δ-uq-qv-nonempty,
     unfold gnfa-α-def PS-the-def PS-add-def, force, simp) []
   apply (rename-tac u', case-tac u' = u,
     insert assms(1) P3S3-props(2) C uv-in-Q3 δ-uq-qv-nonempty,
     unfold gnfa-α-def PS-the-def PS-add-def, force, simp) []
   done
qed


definition gnfa-initial-invar where
gnfa-initial-invar A it M ≡
   (∀ q ∈ State‘(SemiAutomaton.Q A − it). M q = Some {})

definition gnfa-initial-impl2 where
gnfa-initial-impl2 A ≡ do {
```

$M \leftarrow FOREACH^{gnfa\text{-}initial\text{-}invar}\ \mathcal{A}\ (SemiAutomaton.\mathcal{Q}\ \mathcal{A})$
$(\lambda q\ M.\ RETURN\ (M(State\ q \mapsto \{\})))\ Map.empty;$
$let\ M = M(Start \mapsto \{\},\ End \mapsto \{\});$
$d \leftarrow SPEC\ (\lambda d.\ \forall\, u\ v.\ d\ u\ v = \{\});$
$RETURN\ (SemiAutomaton.\mathcal{Q}\ \mathcal{A},\ d,\ M,\ M)$
}

lemma (**in** *NFA*) *gnfa-initial-impl2-correct*:
$gnfa\text{-}initial\text{-}impl2\ \mathcal{A} \le \Downarrow gnfa\text{-}refrel\ (SPEC\ (\lambda\mathcal{A}'.\ \mathcal{A}' =$
$(\!|\ \mathcal{Q} = \{Start, End\} \cup State\ `\ SemiAutomaton.\mathcal{Q}\ \mathcal{A},\ \delta = \lambda u\ v.\ \{\}\ |\!)))$
unfolding *gnfa-initial-impl2-def*
apply (*simp add*: *br-def*, *refine-rcg single-valued-gnfa-refrel*)
apply (*simp add*: *single-valued-def*)
apply (*intro refine-vcg*)
apply (*fact finite-$\mathcal{Q}$*)
apply (*simp add*: *gnfa-initial-invar-def*)
apply (*force simp*: *gnfa-initial-invar-def*)
unfolding *gnfa-initial-invar-def gnfa-invar-def Let-def*
apply (*force simp add*: *GNFA-def gnfa-α-def finite-$\mathcal{Q}$*)
done

definition *nfa-to-gnfa-impl2* **where**
$nfa\text{-}to\text{-}gnfa\text{-}impl2\ \mathcal{A} = do\ \{$
$\mathcal{A}' \leftarrow gnfa\text{-}initial\text{-}impl2\ \mathcal{A};$
$\mathcal{A}'' \leftarrow FOREACH\ (\mathcal{I}\ \mathcal{A})\ (\lambda v\ (Q,\ \delta,\ P,\ S).$
$RETURN\ (Q,\ \lambda u'\ v'.\ if\ u'=Start \wedge v'=State\ v\ then\ \{[]\}\ else\ \delta\ u'\ v',$
$P(State\ v \mapsto \{Start\}),\ PS\text{-}add\ S\ Start\ (State\ v)))\ \mathcal{A}';$
$\mathcal{A}'' \leftarrow FOREACH\ (\mathcal{F}\ \mathcal{A})\ (\lambda u\ (Q,\ \delta,\ P,\ S).$
$RETURN\ (Q,\ \lambda u'\ v'.\ if\ u'=State\ u \wedge v'=End\ then\ \{[]\}\ else\ \delta\ u'\ v',$
$PS\text{-}add\ P\ End\ (State\ u),\ S(State\ u \mapsto \{End\})))\ \mathcal{A}'';$
$\mathcal{A}'' \leftarrow FOREACH\ (\Delta\ \mathcal{A})\ (\lambda(u,c,v)\ (Q,\ \delta,\ P,\ S).$
$RETURN\ (Q,\ \lambda u'\ v'.\ if\ u'=State\ u \wedge v'=State\ v\ then$
$insert\ [c]\ (\delta\ u'\ v')\ else\ \delta\ u'\ v',$
$PS\text{-}add\ P\ (State\ v)\ (State\ u),\ PS\text{-}add\ S\ (State\ u)\ (State\ v)))\ \mathcal{A}'';$
$RETURN\ \mathcal{A}''$
}

lemma (**in** *NFA*) *nfa-to-gnfa-impl2-aux1*:
fixes δ
assumes $v \in it$ $\quad it \subseteq \mathcal{I}\ \mathcal{A}$ $\quad nfa\text{-}to\text{-}gnfa\text{-}invar1\ \mathcal{A}$
$(insert\ Start\ (insert\ End\ (State`SemiAutomaton.\mathcal{Q}\ \mathcal{A})))\ it$
$(\!|\mathcal{Q}=insert\ Start\ (insert\ End\ (State`Q)),\ GNFA\text{-}rec.\delta = \delta\ |\!)$
$gnfa\text{-}invar\ (Q,\ \delta,\ P,\ S)$
shows $gnfa\text{-}invar\ (Q,\ \lambda u'\ v'.\ if\ u'=Start \wedge v'=State\ v\ then$
$\{[]\}\ else\ \delta\ u'\ v',\ P(State\ v \mapsto \{Start\}),\ PS\text{-}add\ S\ Start\ (State\ v))$
(**is** $gnfa\text{-}invar\ (Q,\ ?\delta',\ ?P',\ ?S')$)
proof
let $?\mathcal{A} = gnfa\text{-}\alpha\ (Q,\delta,P,S)$ **and** $?\mathcal{A}' = gnfa\text{-}\alpha\ (Q,?\delta',?P',?S')$

have [*simp*]: $\mathcal{Q}$ *?*$\mathcal{A}'$ = $\mathcal{Q}$ *?*$\mathcal{A}$ **unfolding** *gnfa-α-def* **by** *simp*
from *assms* **have** *GNFA ?*$\mathcal{A}$ **unfolding** *gnfa-invar-def Let-def* **by** *simp*
thus *GNFA ?*$\mathcal{A}'$ **by** (*simp-all add*: *GNFA-def gnfa-α-def*)

hence [*simp*, *intro*]: *Start* ∈ $\mathcal{Q}$ *?*$\mathcal{A}$ **unfolding** *GNFA-def* **by** *simp*

fix *q* **assume** *q* ∈ $\mathcal{Q}$ *?*$\mathcal{A}'$ **hence** *q* ∈ $\mathcal{Q}$ *?*$\mathcal{A}$ **by** *simp*
with *assms* **have** *P q* = *Some* {*u*∈$\mathcal{Q}$ *?*$\mathcal{A}$. δ *u q* ≠ {}}
  unfolding *gnfa-invar-def* **by** (*simp add*: *Let-def*)
moreover from *assms* **have** ⋀*u v*. *u* ≠ *Start* ⟹ δ *u v* = {}
  unfolding *nfa-to-gnfa-invar1-def gnfa-α-def Let-def*
  by (*simp split*: *gnfastate.split*)
ultimately show *?P*′ *q* = *Some* {*u*∈$\mathcal{Q}$ *?*$\mathcal{A}'$. *?*δ′ *u q* ≠ {}} **by** *simp*

from *assms* **have** [*simp*]:*v* ∈ *Q* **using** $\mathcal{I}$*-consistent*
  by (*auto simp*: *nfa-to-gnfa-invar1-def*)
from ‹*q* ∈ $\mathcal{Q}$ *?*$\mathcal{A}'$› **and** *assms* **have** *S q* = *Some* {*v*∈$\mathcal{Q}$ *?*$\mathcal{A}$. δ *q v* ≠ {}}
  unfolding *gnfa-invar-def* **by** (*simp add*: *Let-def*)
thus *?S*′ *q* = *Some* {*v*∈$\mathcal{Q}$ *?*$\mathcal{A}'$. *?*δ′ *q v* ≠ {}} **using** *assms*
  by (*force simp*: *gnfa-α-def PS-add-def split*: *option.split*)
qed

lemma (**in** *NFA*) *nfa-to-gnfa-impl2-aux2*:
fixes δ
assumes *u* ∈ *it*   *it* ⊆ $\mathcal{F}$ $\mathcal{A}$ *nfa-to-gnfa-invar2* $\mathcal{A}$
  (*insert Start* (*insert End* (*State*‘*SemiAutomaton*.$\mathcal{Q}$ $\mathcal{A}$))) *it*
  (|$\mathcal{Q}$=*insert Start* (*insert End* (*State*‘*Q*)), *GNFA-rec*.δ = δ |)
  gnfa-invar (*Q*, δ, *P*, *S*)
shows *gnfa-invar* (*Q*, λ*u*′ *v*′. *if u*′=*State u*∧*v*′=*End then*
  {[]} *else* δ *u*′ *v*′, *PS-add P End* (*State u*), *S*(*State u* ↦ {*End*}))
  (**is** *gnfa-invar* (*Q*, *?*δ′, *?P*′, *?S*′))
proof
let *?*$\mathcal{A}$ = *gnfa-α* (*Q*,δ,*P*,*S*) **and** *?*$\mathcal{A}'$ = *gnfa-α* (*Q*,*?*δ′,*?P*′,*?S*′)
have [*simp*]: $\mathcal{Q}$ *?*$\mathcal{A}'$ = $\mathcal{Q}$ *?*$\mathcal{A}$ **unfolding** *gnfa-α-def* **by** *simp*
from *assms* **have** *GNFA ?*$\mathcal{A}$ **unfolding** *gnfa-invar-def Let-def* **by** *simp*
thus *GNFA ?*$\mathcal{A}'$ **by** (*simp-all add*: *GNFA-def gnfa-α-def*)

hence [*simp*, *intro*]: *End* ∈ $\mathcal{Q}$ *?*$\mathcal{A}$ **unfolding** *GNFA-def* **by** *simp*

from *assms* **have** [*simp*]:*u* ∈ *Q* **using** $\mathcal{F}$*-consistent*
  by (*auto simp*: *nfa-to-gnfa-invar2-def*)

fix *q* **assume** *q* ∈ $\mathcal{Q}$ *?*$\mathcal{A}'$ **hence** *q* ∈ $\mathcal{Q}$ *?*$\mathcal{A}$ **by** *simp*
with *assms* **have** *S q* = *Some* {*v*∈$\mathcal{Q}$ *?*$\mathcal{A}$. δ *q v* ≠ {}}
  unfolding *gnfa-invar-def* **by** (*simp add*: *Let-def*)
moreover from *assms* **have** ⋀*u v*. *v* ≠ *End* ⟹ δ (*State u*) *v* = {}
  unfolding *nfa-to-gnfa-invar2-def gnfa-α-def Let-def*
  by (*simp split*: *gnfastate.split*)

ultimately show *?S′ q = Some {v∈𝒬 ?𝒜′. ?δ′ q v ≠ {}}* **by** *simp*

from ‹*q ∈ 𝒬 ?𝒜′*› **and** *assms* **have** *P q = Some {u∈𝒬 ?𝒜. δ u q ≠ {}}*
unfolding *gnfa-invar-def* **by** (*simp add*: *Let-def*)
thus *?P′ q = Some {u∈𝒬 ?𝒜′. ?δ′ u q ≠ {}}* **using** *assms*
by (*force simp*: *gnfa-α-def PS-add-def split*: *option.split*)
qed

lemma (**in** *NFA*) *nfa-to-gnfa-impl2-aux3*:
fixes *δ*
assumes *ucv ∈ it* *it ⊆ Δ 𝒜* *ucv = (u,c,v)* *nfa-to-gnfa-invar3 𝒜*
(*insert Start* (*insert End* (*State‘SemiAutomaton.𝒬 𝒜*))) *it*
(|*Q=insert Start* (*insert End* (*State‘Q*)), *GNFA-rec.δ = δ* |)
gnfa-invar (*Q*, *δ*, *P*, *S*)
shows *gnfa-invar* (*Q*, *λu′ v′. if u′=State u∧v′=State v then*
insert [*c*] (*δ u′ v′*) *else δ u′ v′*,
PS-add P (*State v*) (*State u*), *PS-add S* (*State u*) (*State v*))
(**is** *gnfa-invar* (*Q*, *?δ′*, *?P′*, *?S′*))
proof
let *?𝒜 = gnfa-α* (*Q,δ,P,S*) **and** *?𝒜′ = gnfa-α* (*Q,?δ′,?P′,?S′*)
have [*simp*]: *𝒬 ?𝒜′ = 𝒬 ?𝒜* **unfolding** *gnfa-α-def* **by** *simp*
from *assms* **have** *GNFA ?𝒜* **unfolding** *gnfa-invar-def Let-def* **by** *simp*
thus *GNFA ?𝒜′* **by** (*simp-all add*: *GNFA-def gnfa-α-def*)

from *assms*(*1−3*) **and** *Δ-consistent*
have *u ∈ SemiAutomaton.𝒬 𝒜* *v ∈ SemiAutomaton.𝒬 𝒜* **by** *auto*
hence [*simp*, *intro*]: *State u ∈ 𝒬 ?𝒜* *State v ∈ 𝒬 ?𝒜* *u ∈ Q* *v ∈ Q*
using *assms*(*4*) **by** (*auto simp*: *nfa-to-gnfa-invar3-def gnfa-α-def*)

fix *q* **assume** *q ∈ 𝒬 ?𝒜′* **hence** *q ∈ 𝒬 ?𝒜* **by** *simp*
with *assms* **have** *A*: *S q = Some {v∈𝒬 ?𝒜. δ q v ≠ {}}* **and**
B: *P q = Some {u∈𝒬 ?𝒜. δ u q ≠ {}}*
unfolding *gnfa-invar-def* **by** (*simp-all add*: *Let-def*)
thus *?S′ q = Some {v∈𝒬 ?𝒜′. ?δ′ q v ≠ {}}*
?P′ q = Some {u∈𝒬 ?𝒜′. ?δ′ u q ≠ {}}
by (*auto simp*: *gnfa-α-def PS-add-def split*: *option.split*)
qed

lemma (**in** *NFA*) *nfa-to-gnfa-impl2-refine*:
nfa-to-gnfa-impl2 𝒜 ≤ ⇓gnfa-refrel (*nfa-to-gnfa-impl 𝒜*)
unfolding *nfa-to-gnfa-impl2-def nfa-to-gnfa-impl-def*
apply (*refine-rcg single-valued-gnfa-refrel inj-on-id*)
apply (*fact gnfa-initial-impl2-correct*)
apply (*auto simp*: *gnfa-α-def br-def intro*!: *ext nfa-to-gnfa-impl2-aux1*) [*3*]
apply (*auto simp*: *gnfa-α-def br-def intro*!: *ext nfa-to-gnfa-impl2-aux2*) [*3*]
apply (*auto simp*: *gnfa-α-def br-def intro*!: *ext nfa-to-gnfa-impl2-aux3*) []

done

definition *nfa-to-rexp-impl2* **where**
nfa-to-rexp-impl2 $\mathcal{A} \equiv$ *do* {
  $(Q,\delta,P,S) \leftarrow$ *nfa-to-gnfa-impl2* $\mathcal{A}$;
  $(Q,\delta,P,S) \leftarrow$
    $WHILE_T$ $(\lambda(Q,\delta,P,S).\ Q \neq \{\})$ $(\lambda(Q,\delta,P,S).$ *do* {
      $q \leftarrow$ *SPEC* $(\lambda q.\ q \in Q)$;
      $(Q,\delta,P,S) \leftarrow$ *gnfa-contract-impl2* (Q,δ,P,S) (*State* q);
      RETURN (Q,δ,P,S)
    }) (Q,δ,P,S);
  RETURN (δ *Start End*)
}

lemma (**in** *NFA*) *nfa-to-rexp-impl2-refine*:
  nfa-to-rexp-impl2 $\mathcal{A} \leq \Downarrow$*Id* (*nfa-to-rexp-impl* $\mathcal{A}$)
unfolding *nfa-to-rexp-impl2-def nfa-to-rexp-impl-def*
apply (*refine-rcg single-valued-Id single-valued-gnfa-refrel*
    single-valued-gnfa-state-refrel)
apply (*rule nfa-to-gnfa-impl2-refine*)
apply (*simp add*: *br-def*)
apply (*force simp*: *gnfa-α-def br-def*)
apply (*rule SPEC-refine-sv*[*OF single-valued-gnfa-state-refrel SPEC-rule*],
    simp add: *gnfa-α-def br-def*)
apply (*blast intro*!: *gnfa-contract-impl2-refine*)
apply *simp*
apply (*simp add*: *gnfa-α-def br-def*)
done

abbreviation *gnfa-δ-lookup d u v* $\equiv$
  (*case d u of None* $\Rightarrow$ *Zero* | *Some du* $\Rightarrow$
    (*case du v of None* $\Rightarrow$ *Zero* | *Some r* $\Rightarrow$ *r*))

abbreviation *gnfa-δ-α2 d* $\equiv \lambda u\ v.$
    lang (*gnfa-δ-lookup d u v*)

abbreviation *gnfa-δ-refrel2* $\equiv$ *br gnfa-δ-α2* (λ-. *True*)
lemma *single-valued-gnfa-δ-refrel2*:
  single-valued gnfa-δ-refrel2 **by** (*fact br-sv*)

abbreviation *rprod* $\equiv$ *prod-rel*

abbreviation *gnfa-refrel2* $\equiv \langle$*Id*, $\langle$*gnfa-δ-refrel2*, $\langle$*Id*, *Id*$\rangle$*rprod*$\rangle$*rprod*$\rangle$*rprod*
lemma *single-valued-gnfa-refrel2*:
  single-valued gnfa-refrel2
    by (*intro prod-rel-sv single-valued-Id single-valued-gnfa-δ-refrel2*)

definition *gnfa-initial-impl3* :: ('q, 'c, 'e) *SemiAutomaton-rec-scheme*
  ⇒ ('q *set* ×
    ('q *gnfastate* ⇒ ('q *gnfastate* ⇒ 'c *rexp option*) *option*) ×
    ('q *gnfastate* ⇒ ('q *gnfastate*) *set option*) ×
    ('q *gnfastate* ⇒ ('q *gnfastate*) *set option*)) *nres* **where**
gnfa-initial-impl3 $\mathcal{A}$ ≡ *do* {
  M ← *FOREACH* (*SemiAutomaton*.$\mathcal{Q}$ $\mathcal{A}$)
    (λ*q M*. *RETURN* (*M*(*State q* ↦ {}))) *Map.empty*;
  let M = *M*(*Start* ↦ {}, *End* ↦ {});
  d ← *RETURN Map.empty*;
  RETURN (*SemiAutomaton*.$\mathcal{Q}$ $\mathcal{A}$, *d*, *M*, *M*)
}

lemma *gnfa-initial-impl3-δ-correct*:
 RETURN Map.empty ≤ ⇓*gnfa-δ-refrel2*
   (*SPEC* (λ*d*. ∀ *u v*. *d u v* = {}))
 by (*rule SPEC-refine*, *simp add*: *br-def*)

lemma *gnfa-initial-impl3-refine*:
 gnfa-initial-impl3 $\mathcal{A}$ ≤ ⇓*gnfa-refrel2*
   (*gnfa-initial-impl2* $\mathcal{A}$)
unfolding *gnfa-initial-impl3-def gnfa-initial-impl2-def*
apply (*refine-rcg single-valued-gnfa-refrel2*
  single-valued-Id Id-refine inj-on-id)
apply *simp-all*[*3*]
apply (*rule gnfa-initial-impl3-δ-correct*)
apply (*simp add*: *br-def*)
done

definition *gnfa-δ-update d u v r* ≡
 case d u of
  None ⇒ *d*(*u* ↦ [*v* ↦ *r*]) |
  Some du ⇒ *d*(*u* ↦ *du*(*v* ↦ *r*))

definition *gnfa-δ-insert d u v r* ≡
 case d u of
  None ⇒ *d*(*u* ↦ [*v* ↦ *r*]) |
  Some du ⇒ *let duv'* = (
    case du v of
     None ⇒ *r* |
     Some duv ⇒ *Plus duv r*)
   in d(*u* ↦ *du*(*v* ↦ *duv'*))

lemma *gnfa-δ-update-correct*[*simp*]:
 gnfa-δ-α2 (*gnfa-δ-update d u v r*) = (λ*u' v'*.
   (*if u'*=*u* ∧ *v'*=*v then lang r else gnfa-δ-α2 d u' v'*))
 unfolding *gnfa-δ-update-def*

```
  by (intro ext, auto split: option.split)

lemma gnfa-δ-insert-correct[simp]:
    gnfa-δ-α2 (gnfa-δ-insert d u v r) = (λu′ v′.
     (if u′=u ∧ v′=v then gnfa-δ-α2 d u′ v′ ∪ lang r
        else gnfa-δ-α2 d u′ v′))
  unfolding gnfa-δ-insert-def
  by (intro ext, auto split: option.split)


definition nfa-to-gnfa-impl3 where
nfa-to-gnfa-impl3 𝒜 = do {
  𝒜′ ← gnfa-initial-impl3 𝒜;
  𝒜′′ ← FOREACH (ℐ 𝒜) (λv (Q, δ, P, S).
       RETURN (Q, gnfa-δ-update δ Start (State v) rexp.One,
          P(State v ↦ {Start}), PS-add S Start (State v))) 𝒜′;
  𝒜′′ ← FOREACH (ℱ 𝒜) (λu (Q, δ, P, S).
       RETURN (Q, gnfa-δ-update δ (State u) End rexp.One,
          PS-add P End (State u), S(State u ↦ {End}))) 𝒜′′;
  𝒜′′ ← FOREACH (Δ 𝒜) (λ(u,c,v) (Q, δ, P, S).
       RETURN (Q, gnfa-δ-insert δ (State u) (State v) (Atom c),
          PS-add P (State v) (State u), PS-add S (State u) (State v))) 𝒜′′;
  RETURN 𝒜′′
}


lemma nfa-to-gnfa-impl3-refine:
  nfa-to-gnfa-impl3 𝒜 ≤ ⇓gnfa-refrel2 (nfa-to-gnfa-impl2 𝒜)
unfolding nfa-to-gnfa-impl3-def nfa-to-gnfa-impl2-def
apply (refine-rcg single-valued-gnfa-refrel2 inj-on-id)
apply (rule gnfa-initial-impl3-refine)
apply simp
apply simp
apply simp
apply (simp add: br-def, intro ext, simp)
apply simp
apply simp
apply (simp add: br-def, intro ext, simp)
apply simp
apply simp
apply (simp add: br-def, intro ext, auto) []
done


definition gnfa-remove-state-impl3 where
gnfa-remove-state-impl3 ≡ λ(Q,δ,P,S) q.
  case q of Start ⇒ RETURN (Q,δ,P,S) | End ⇒ RETURN (Q,δ,P,S) |
     State q′ ⇒ do {
```

$P' \leftarrow$ *FOREACH* (*PS-the S q*)
  (λv *P*. *RETURN* (*PS-remove P v q*)) *P*;
$S' \leftarrow$ *FOREACH* (*PS-the P q*)
  (λu *S*. *RETURN* (*PS-remove S u q*)) *S*;
RETURN ($Q - \{q'\}$, δ, P', S')
}

lemma *gnfa-remove-state-impl3-refine*:
fixes δ
assumes $((Q,\delta,P,S), (Q',\delta',P',S')) \in$ *gnfa-refrel2* **and** $(q,q') \in Id$
shows *gnfa-remove-state-impl3* (Q,δ,P,S) $q \leq \Downarrow$*gnfa-refrel2*
  (*gnfa-remove-state-impl2* (Q',δ',P',S') q')
unfolding *gnfa-remove-state-impl3-def gnfa-remove-state-impl2-def*
apply (*simp add*: *br-def prod-rel-def split*: *gnfastate.split*)
apply (*intro conjI impI allI*)
apply (*insert assms*)
apply (*refine-rcg*)
apply (*simp-all add*: *single-valued-def br-def*)[*5*]
apply (*rule RETURN-refine-sv*)
apply (*simp add*: *single-valued-def*)
apply (*force simp*: *single-valued-def br-def*)
apply (*refine-rcg*, (*simp-all add*: *single-valued-def br-def*)[*4*])
apply (*refine-rcg inj-on-id single-valued-gnfa-refrel2*)
apply (*simp add*: *br-def prod-rel-def*)
prefer *2*
apply (*rule single-valued-Id*)
apply (*simp-all add*: *br-def prod-rel-def*)[*4*]
prefer *2*
apply (*rule single-valued-Id*)
apply (*simp-all add*: *br-def prod-rel-def*)[*3*]
apply (*simp add*: *single-valued-def*)
apply (*simp add*: *prod-rel-def br-def*)
done

definition *rexp-simped-concat r s* $\equiv$
(*if r* = *rexp.One then s*
  else if s = *rexp.One then r else Times r s*)

lemma *rexp-simped-concat-correct*[*simp*]:
lang (*rexp-simped-concat r s*) = *lang r* @@ *lang s*
unfolding *rexp-simped-concat-def* **by** *simp*

definition *rexp-simped-contract r s t* $\equiv$
(*if s* = *Zero then rexp-simped-concat r t*
  else rexp-simped-concat r
    (*rexp-simped-concat* (*Star s*) *t*))

```
lemma rexp-simped-contract-correct[simp]:
  lang (rexp-simped-contract r1 r2 r3) =
    lang r1 @@ star (lang r2) @@ lang r3
  unfolding rexp-simped-contract-def by simp

definition gnfa-contract-impl3-update-δ where
gnfa-contract-impl3-update-δ ≡ λδ u q v.
    let r1 = gnfa-δ-lookup δ u q;
        r2 = gnfa-δ-lookup δ q q;
        r3 = gnfa-δ-lookup δ q v
    in gnfa-δ-insert δ u v (rexp-simped-contract r1 r2 r3)


lemma gnfa-δ-insert-correct′:
  gnfa-δ-α2 (gnfa-δ-insert d u v r) u′ v′ =
    (if u′ = u ∧ v′ = v then gnfa-δ-α2 d u′ v′ ∪ lang r
        else gnfa-δ-α2 d u′ v′)
  by (simp add: gnfa-δ-insert-def split: option.split)

lemma gnfa-contract-impl3-update-δ-correct:
  assumes (δ1, δ2) ∈ gnfa-δ-refrel2    (q,q′) ∈ Id
  shows gnfa-contract-update-δ-impl2 δ2 u q′ v =
      gnfa-δ-α2 (gnfa-contract-impl3-update-δ δ1 u q v)
          (is ?g = gnfa-δ-α2 ?f)
proof−
  {fix u′ v′
    from assms have gnfa-δ-α2 ?f u′ v′ = ?g u′ v′
    unfolding gnfa-contract-impl3-update-δ-def
        gnfa-contract-update-δ-impl2-def
      by (auto simp: Let-def br-def gnfa-δ-insert-correct′)
  }
  hence gnfa-δ-α2 ?f = ?g by (intro ext)
  thus ?thesis by simp
qed


definition gnfa-contract-impl3 where
gnfa-contract-impl3 ≡ λ(Q,δ,P,S) q. do {
  let Pq = PS-the P q − {q}; let Sq = PS-the S q − {q};
  (Q,δ,P,S) ← gnfa-remove-state-impl3 (Q,δ,P,S) q;
  (Q,δ,P,S) ← FOREACH Pq (λu (Q′,δ′,P′,S′).
    FOREACH Sq (λv (Q′,δ′,P′,S′).
        RETURN (Q′, gnfa-contract-impl3-update-δ δ′ u q v,
            PS-add P′ v u, PS-add S′ u v))
      (Q′,δ′,P′,S′)
  ) (Q,δ,P,S);
  RETURN (Q,δ,P,S)
}
```

lemma *gnfa-contract-impl3-refine*:
  fixes δ
  assumes $((Q,\delta,P,S),\ (Q',\delta',P',S')) \in$ *gnfa-refrel2* **and**
    $(q,q') \in Id$
  shows *gnfa-contract-impl3* $(Q,\delta,P,S)\ q \leq \Downarrow$*gnfa-refrel2*
    (*gnfa-contract-impl2* $(Q',\delta',P',S')\ q'$)
unfolding *gnfa-contract-impl3-def gnfa-contract-impl2-def*
apply (*refine-rcg single-valued-gnfa-refrel2 inj-on-id*
  single-valued-Id)
thm *gnfa-remove-state-impl3-refine*
apply (*insert assms*, *rule gnfa-remove-state-impl3-refine*, *simp-all*) [*2*]
apply (*insert assms*, *simp add*: *br-def*) []
apply *simp*
apply (*insert assms*, *simp add*: *br-def*) []
apply *simp*
apply (*clarsimp simp add*: *br-def*
  gnfa-contract-impl3-update-δ-correct[*OF - assms*(*2*)])
apply *simp*
done

definition *nfa-to-rexp-impl3* **where**
nfa-to-rexp-impl3 $\mathcal{A} \equiv$ *do* {
  $(Q,\delta,P,S) \leftarrow$ *nfa-to-gnfa-impl3* $\mathcal{A}$;
  $(Q,\delta,P,S) \leftarrow$
    $WHILE_T$ $(\lambda(Q,\delta,P,S).\ Q \neq \{\})$ $(\lambda(Q,\delta,P,S).$ *do* {
      ASSERT $(Q\neq\{\})$;
      $q \leftarrow$ *SPEC* $(\lambda q.\ q \in Q)$;
      $(Q,\delta,P,S) \leftarrow$ *gnfa-contract-impl3* (Q,δ,P,S) (*State* q);
      RETURN (Q,δ,P,S)
    }) (Q,δ,P,S);
  RETURN (*gnfa-δ-lookup* δ *Start End*)
}

abbreviation *rexp-refrel* $\equiv$ *br lang* $(\lambda$-. *True*)

lemma *nfa-to-rexp-impl3-refine*:
  nfa-to-rexp-impl3 $\mathcal{A} \leq \Downarrow$*rexp-refrel* (*nfa-to-rexp-impl2* $\mathcal{A}$)
unfolding *nfa-to-rexp-impl3-def nfa-to-rexp-impl2-def*
apply (*refine-rcg single-valued-gnfa-refrel2 inj-on-id*)
apply (*rule nfa-to-gnfa-impl3-refine*)
apply (*simp-all add*: *br-def prod-rel-def*)[*4*]
apply (*rule gnfa-contract-impl3-refine*)
apply (*simp-all add*: *br-def prod-rel-def*)
done

5.4.5 Implementation of NFAs

abbreviation *rexp-rel* ≡ *Id* :: (*nat rexp*×*nat rexp*) *set*
consts *i-rexp* :: *interface*

lemmas *rexp-rel-def* = *TrueI*

lemmas [*autoref-rel-intf*] =
REL-INTFI[*of rexp-rel i-rexp*, *standard*]

lemma *rexp-rel-sv*[*relator-props*]: *single-valued rexp-rel*
unfolding *rexp-rel-def* **by** *simp*

lemma *Zero-param*[*param*,*autoref-rules*]:
(*Zero*,*Zero*) ∈ *rexp-rel* **unfolding** *rexp-rel-def* **by** *simp*

lemma *One-param*[*param*,*autoref-rules*]:
(*One*,*One*) ∈ *rexp-rel* **unfolding** *rexp-rel-def* **by** *simp*

lemma *Atom-param*[*param*,*autoref-rules*]:
(*Atom*,*Atom*) ∈ *nat-rel* → *rexp-rel* **unfolding** *rexp-rel-def* **by** *simp*

lemma *Plus-param*[*param*,*autoref-rules*]:
(*Plus*,*Plus*) ∈ *rexp-rel* → *rexp-rel* → *rexp-rel*
unfolding *rexp-rel-def* **by** *simp*

lemma *Times-param*[*param*,*autoref-rules*]:
(*Times*,*Times*) ∈ *rexp-rel* → *rexp-rel* → *rexp-rel*
unfolding *rexp-rel-def* **by** *simp*

lemma *Star-param*[*param*,*autoref-rules*]:
(*Star*,*Star*) ∈ *rexp-rel* → *rexp-rel*
unfolding *rexp-rel-def* **by** *simp*

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

consts *i-gnfastate* :: *interface*

lemmas [*autoref-rel-intf*] =
REL-INTFI[*of gnfastate-rel i-gnfastate*, *standard*]

lemma *param-State*[*param*,*autoref-rules*]:
(*State*, *State*) ∈ *nat-rel* → *gnfastate-rel*
by *simp*

lemma *param-Start*[*param*,*autoref-rules*]:
(*Start*, *Start*) ∈ *gnfastate-rel* **by** *simp*

lemma *param-End*[*param*,*autoref-rules*]:
  (*End*, *End*) ∈ *gnfastate-rel* **by** *simp*

lemma *param-gnfastate-case*[*param*,*autoref-rules*]:
  (*gnfastate-case*, *gnfastate-case*) ∈
    R → *R* → (*nat-rel* → *R*) → *gnfastate-rel* → *R*

 by (*force split*: *gnfastate.split dest*: *fun-relD*)

lemma *param-gnfastate-eq*[*autoref-rules*]:
 (*op* =, *op* =) ∈ *gnfastate-rel* → *gnfastate-rel* → *bool-rel*
 by *simp*

lemma *gnfastate-rec-is-gnfastate-case*[*simp*]:
  gnfastate-rec = *gnfastate-case*
  by (*force split*: *gnfastate.split*)

fun $\mathcal{Q}$*-impl* **where** $\mathcal{Q}$*-impl* (*Q*,*S*,*D*,*I*,*F*) = *Q*
fun Σ*-impl* **where** Σ*-impl* (*Q*,*S*,*D*,*I*,*F*) = *S*
fun Δ*-impl* **where** Δ*-impl* (*Q*,*S*,*D*,*I*,*F*) = *D*
fun $\mathcal{I}$*-impl* **where** $\mathcal{I}$*-impl* (*Q*,*S*,*D*,*I*,*F*) = *I*
fun $\mathcal{F}$*-impl* **where** $\mathcal{F}$*-impl* (*Q*,*S*,*D*,*I*,*F*) = *F*

definition *NFA-rel-internal-def*: *NFA-rel Rqs Rss Rds Ris Rfs R*$\mathcal{Q}$ *R*Σ ≡
 { ((*Q*,*S*,*D*,*I*,*F*),$\mathcal{A}$) .
  NFA $\mathcal{A}$ ∧
  (*Q*,*SemiAutomaton*.$\mathcal{Q}$ $\mathcal{A}$)∈⟨*R*$\mathcal{Q}$⟩*Rqs* ∧
  (*S*,Σ $\mathcal{A}$)∈⟨*R*Σ⟩*Rss* ∧
  (*D*,Δ $\mathcal{A}$)∈⟨⟨*R*$\mathcal{Q}$,⟨*R*Σ,*R*$\mathcal{Q}$⟩*prod-rel*⟩*prod-rel*⟩*Rds* ∧
  (*I*,$\mathcal{I}$ $\mathcal{A}$)∈⟨*R*$\mathcal{Q}$⟩*Ris* ∧
  (*F*,$\mathcal{F}$ $\mathcal{A}$)∈⟨*R*$\mathcal{Q}$⟩*Rfs*}

lemma *NFA-rel-def*: ⟨*R*$\mathcal{Q}$,*R*Σ⟩*NFA-rel Rqs Rss Rds Ris Rfs* ≡ { ((*Q*,*S*,*D*,*I*,*F*),$\mathcal{A}$)
.
  NFA $\mathcal{A}$ ∧
  (*Q*,*SemiAutomaton*.$\mathcal{Q}$ $\mathcal{A}$)∈⟨*R*$\mathcal{Q}$⟩*Rqs* ∧
  (*S*,Σ $\mathcal{A}$)∈⟨*R*Σ⟩*Rss* ∧
  (*D*,Δ $\mathcal{A}$)∈⟨⟨*R*$\mathcal{Q}$,⟨*R*Σ,*R*$\mathcal{Q}$⟩*prod-rel*⟩*prod-rel*⟩*Rds* ∧
  (*I*,$\mathcal{I}$ $\mathcal{A}$)∈⟨*R*$\mathcal{Q}$⟩*Ris* ∧
  (*F*,$\mathcal{F}$ $\mathcal{A}$)∈⟨*R*$\mathcal{Q}$⟩*Rfs*}
 unfolding *NFA-rel-internal-def*[*abs-def*] *relAPP-def* **.**

consts *i-NFA* :: *interface* ⇒ *interface* ⇒ *interface*

lemmas [*autoref-rel-intf*] =
 REL-INTFI[*of NFA-rel Rqs Rss Rds Ris Rfs i-NFA*, *standard*]

lemma $\mathcal{Q}$*-autoref*[*autoref-rules*]:

($\mathcal{Q}$-*impl*,*SemiAutomaton*.$\mathcal{Q}$)∈⟨$R\mathcal{Q}$,$R\Sigma$⟩*NFA-rel Rqs Rss Rds Ris Rfs* → ⟨$R\mathcal{Q}$⟩*Rqs*
unfolding *NFA-rel-def* **by** *auto*
lemma Σ-*autoref*[*autoref-rules*]:
(Σ-*impl*,Σ)∈⟨$R\mathcal{Q}$,$R\Sigma$⟩*NFA-rel Rqs Rss Rds Ris Rfs* → ⟨$R\Sigma$⟩*Rss*
unfolding *NFA-rel-def* **by** *auto*
lemma Δ-*autoref*[*autoref-rules*]:
(Δ-*impl*,Δ)∈⟨$R\mathcal{Q}$,$R\Sigma$⟩*NFA-rel Rqs Rss Rds Ris Rfs*
→ ⟨⟨$R\mathcal{Q}$,⟨$R\Sigma$,$R\mathcal{Q}$⟩*prod-rel*⟩*prod-rel*⟩*Rds*
unfolding *NFA-rel-def* **by** *auto*
lemma $\mathcal{I}$-*autoref*[*autoref-rules*]:
($\mathcal{I}$-*impl*,$\mathcal{I}$)∈⟨$R\mathcal{Q}$,$R\Sigma$⟩*NFA-rel Rqs Rss Rds Ris Rfs* → ⟨$R\mathcal{Q}$⟩*Ris*
unfolding *NFA-rel-def* **by** *auto*
lemma $\mathcal{F}$-*autoref*[*autoref-rules*]:
($\mathcal{F}$-*impl*,$\mathcal{F}$)∈⟨$R\mathcal{Q}$,$R\Sigma$⟩*NFA-rel Rqs Rss Rds Ris Rfs* → ⟨$R\mathcal{Q}$⟩*Rfs*
unfolding *NFA-rel-def* **by** *auto*

abbreviation *dflt-NFA-rel*
≡ *NFA-rel dflt-rs-rel dflt-rs-rel dflt-rs-rel dflt-rs-rel dflt-rs-rel*

lemmas *nfa-to-rexp-unfold-complete* =
nfa-to-rexp-impl3-def[*unfolded nfa-to-gnfa-impl3-def gnfa-initial-impl3-def*
gnfa-δ-update-def gnfa-contract-impl3-def gnfa-contract-impl3-update-δ-def
rexp-simped-contract-def rexp-simped-concat-def PS-add-def
gnfa-remove-state-impl3-def PS-the-def gnfa-δ-insert-def PS-remove-def]

concrete-definition *nfa-to-rexp* **uses** *nfa-to-rexp-unfold-complete*

lemma (**in** *transfer*) *transfer-gnfastate*[*refine-transfer*]:
assumes α *fs* ≤ *Fs*
assumes α *fe* ≤ *Fe*
assumes ⋀*q*. α (*fq q*) ≤ *Fq q*
shows α (*gnfastate-case fs fe fq x*) ≤ *gnfastate-case Fs Fe Fq x*
using *assms* **by** (*auto split*: *gnfastate.split*)

lemma *gnfastate-ne-bot*[*refine-transfer*]:
⋀*fs fe fq x*.
⟦ *fs*≠*dSUCCEED*; *fe*≠*dSUCCEED*; ⋀*v*. *fq v* ≠ *dSUCCEED* ⟧
⟹ *gnfastate-case fs fe fq x* ≠ *dSUCCEED*
by (*auto split*: *gnfastate.split*)

schematic-lemma *nfa-to-rexp-impl*:
notes [[*goals-limit* = *1*]]

assumes [*autoref-rules*]: ($\mathcal{A}$,$\mathcal{A}'$)∈⟨*nat-rel*,*nat-rel*⟩*dflt-NFA-rel*
shows (*?f*::*?'c*, *nfa-to-rexp* $\mathcal{A}'$)∈*?R*
using *assms*
unfolding *nfa-to-rexp-def*
apply (*autoref-monadic* (*trace*))
done

concrete-definition *nfa-to-rexp-code* **uses** *nfa-to-rexp-impl*

export-code *nfa-to-rexp-code* **in** *SML* **file** −

theorem *nfa-to-rexp-code-correct*:
assumes *A*: ($\mathcal{A}$*impl*,$\mathcal{A}$)∈⟨*nat-rel*,*nat-rel*⟩*dflt-NFA-rel*
shows *lang* (*nfa-to-rexp-code* $\mathcal{A}$*impl*) = $\mathcal{L}$ $\mathcal{A}$ (**is** *lang ?r* = -)
proof −
interpret *NFA* $\mathcal{A}$ **using** *A*[*unfolded NFA-rel-def*] **by** *auto*

note *nfa-to-rexp-code.refine*[*OF A*, *THEN nres-relD*]
also note *nfa-to-rexp.refine*[*symmetric*, *THEN meta-eq-to-obj-eq*]
also note *nfa-to-rexp-impl3-refine*
also note *nfa-to-rexp-impl2-refine*
also note *nfa-to-rexp-impl-refine*
also note *nfa-to-rexp-abstr-correct*
finally show *?thesis* **by** (*elim RETURN-ref-SPECD*, *simp add*: *br-def*)
qed

end

Various Examples for the Autoref-Tool **theory** *Testbench*
imports
Examples/Coll-Test
Examples/Nested-DFS
Examples/Simple-DFS
Examples/NFA/NFA-Simulations-INY
Examples/NFA/nfa-to-rexp
Examples/ICF-Test
Examples/ICF-Only-Test
begin

end