Combinatory Logic Blocks

Contents

1 Multi-Dimensional Combinators

We introduce several convenient definitions for families of (arity-extended variants of) combinators. In the spirit of Schönfinkel [3], our aim is to provide building blocks for "point-free" representations of mathematical knowledge (rather than reducing our vocabulary to its base minimum).

theory *combinators*
  imports *Main*
begin

We aggregate theory-related definitions to be unfolded on demand. Here for combinators.

named_theorems *comb_defs*

1.1 Traditional Combinators

1.1.1 Identity and Appliers

The convenient all-purpose identity combinator.

definition *I_comb :: "'a ⇒ 'a" ("I")*
  where "I ≡ λx. x"

The family of combinators A_m are called "appliers". They take m+1 arguments, and return the application of the first argument (an m-ary function) to the remaining ones.

abbreviation *(input) A0_comb :: "'a ⇒ 'a" ("A_0")*
  where "A_0 ≡ I" — degenerate case (m = 0) corresponds to identity I
definition *A1_comb :: "('a ⇒ 'b) ⇒ 'a ⇒ 'b" ("A_1")*
  where "A_1 ≡ λf x. f x" — (unary) function application (@); cf. reverse-pipe (<|)
definition *A2_comb :: "('a ⇒ 'b ⇒ 'c) ⇒ 'a ⇒ 'b ⇒ 'c" ("A_2")*
  where "A_2 ≡ λf x_1 x_2. f x_1 x_2" — function application (binary case)
definition *A3_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ 'a ⇒ 'b ⇒ 'c ⇒ 'd" ("A_3")*
  where "A_3 ≡ λf x_1 x_2 x_3. f x_1 x_2 x_3" — function application (ternary case)
— ... and so on

notation *A1_comb ("A")*

The identity combinator I (suitably typed) generalizes all A_m combinators (via polymorphism and η-conversion).

lemma "A_1 = I"
lemma "A_2 = I"
lemma "A_3 = I"

It is convenient to introduce a family T_m of "reversed appliers" as abbreviations I.

abbreviation *T1_comb::"'b ⇒ ('b ⇒ 'a) ⇒ 'a" ("T_1")*
  where "T_1 x f ≡ A_1 f x" — unary case
abbreviation *T2_comb::"'b ⇒ 'c ⇒ ('b ⇒ 'c ⇒ 'a) ⇒ 'a" ("T_2")*
  where "T_2 x y f ≡ A_2 f x y" — binary case
abbreviation *T3_comb ("T_3")*
  where "T_3 x y z f ≡ A_3 f x y z" — ternary case
— ... and so on

Special notation for unary and binary cases.

notation *T1_comb ("T")* — cf. "Let"; Smullyan's "thrush" [4]
notation *T2_comb ("V")* — cf. "pairing" in λ-calculus; Smullyan's "vireo" [4]

Convenient "pipe" notation for A_1 and its reverse T_1 in their role as function application.

notation *(input) A1_comb* (**infixr** *"<|" 54)*
notation *(input) T1_comb* (**infixl** *"|>" 54)*

declare *I_comb_def[comb_defs] A1_comb_def[comb_defs] A2_comb_def[comb_defs] A3_comb_def[comb_defs]*

Do some notation checks.

lemma "a |> f = f a"
lemma "f <| a = f a"
lemma "a |> f |> g = g (f a)"
lemma "g <| f <| a = g (f a)"
lemma "(a |> f) <| b = f a b"

1.1.2 Compositors

The family of combinators B_N are called "compositors" (with N an m-sized sequence of arities). They compose their first argument f (an m-ary function) with m functions $g_{i \leq m}$ (each of arity N_i). Thus, the returned function has arity: $\Sigma_{i \leq m}\ N_i$.

abbreviation *(input) B0_comb :: "('a ⇒ 'b) ⇒ 'a ⇒ 'b" ("$\mathtt{B}_0$")*
  where "$\mathtt{B}_0 \equiv \mathtt{A}_1$" — composing with a nullary function corresponds to (unary) function application
definition *B1_comb :: "('a ⇒ 'b) ⇒ ('c ⇒ 'a) ⇒ 'c ⇒ 'b" ("$\mathtt{B}_1$")*
  where "$\mathtt{B}_1 \equiv \lambda f\ g\ x.\ f\ (g\ x)$" — the traditional B combinator
definition *B2_comb :: "('a ⇒ 'b) ⇒ ('c ⇒ 'd ⇒ 'a) ⇒ 'c ⇒ 'd ⇒ 'b" ("$\mathtt{B}_2$")*
  where "$\mathtt{B}_2 \equiv \lambda f\ g\ x\ y.\ f\ (g\ x\ y)$" — cf. Smullyan's "blackbird" combinator [4]
definition *B3_comb :: "('a ⇒ 'b) ⇒ ('c ⇒ 'd ⇒ 'e ⇒ 'a) ⇒ 'c ⇒ 'd ⇒ 'e ⇒ 'b" ("$\mathtt{B}_3$")*
  where "$\mathtt{B}_3 \equiv \lambda f\ g\ x\ y\ z.\ f\ (g\ x\ y\ z)$"
definition *B4_comb :: "('a ⇒ 'b) ⇒ ('c ⇒ 'd ⇒ 'e ⇒ 'f ⇒ 'a) ⇒ 'c ⇒ 'd ⇒ 'e ⇒ 'f ⇒ 'b" ("$\mathtt{B}_4$")*
  where "$\mathtt{B}_4 \equiv \lambda f\ g\ x\ y\ z\ w.\ f\ (g\ x\ y\ z\ w)$"
— ... and so on
abbreviation *(input) B00_comb :: "('a ⇒ 'b ⇒ 'c) ⇒ 'a ⇒ 'b ⇒ 'c" ("$\mathtt{B}_{00}$")*
  where "$\mathtt{B}_{00} \equiv \mathtt{A}_2$" — composing with two nullary functions corresponds to binary function application
definition *B01_comb :: "('a ⇒ 'b ⇒ 'c) ⇒ 'a ⇒ ('d ⇒ 'b) ⇒ 'd ⇒ 'c" ("$\mathtt{B}_{01}$")*
  where "$\mathtt{B}_{01} \equiv \lambda f\ g_1\ g_2\ x_2.\ f\ g_1\ (g_2\ x_2)$" — D combinator (cf. Smullyan's "dove"[4])
definition *B02_comb :: "('a ⇒ 'b ⇒ 'c) ⇒ 'a ⇒ ('d ⇒ 'e ⇒ 'b) ⇒ 'd ⇒ 'e ⇒ 'c" ("$\mathtt{B}_{02}$")*
  where "$\mathtt{B}_{02} \equiv \lambda f\ g_1\ g_2\ x_2\ y_2.\ f\ g_1\ (g_2\ x_2\ y_2)$" — E combinator (cf. Smullyan's "eagle"[4])
definition *B03_comb :: "('a ⇒ 'b ⇒ 'c) ⇒ 'a ⇒ ('d ⇒ 'e ⇒ 'f ⇒ 'b) ⇒ 'd ⇒ 'e ⇒ 'f ⇒ 'c" ("$\mathtt{B}_{03}$")*
  where "$\mathtt{B}_{03} \equiv \lambda f\ g_1\ g_2\ x_2\ y_2\ z_2.\ f\ g_1\ (g_2\ x_2\ y_2\ z_2)$"
— ... and so on
definition *B10_comb :: "('a ⇒ 'b ⇒ 'c) ⇒ ('d ⇒ 'a) ⇒ 'b ⇒ 'd ⇒ 'c" ("$\mathtt{B}_{10}$")*
  where "$\mathtt{B}_{10} \equiv \lambda f\ g_1\ g_2\ x_1.\ f\ (g_1\ x_1)\ g_2$"
definition *B11_comb :: "('a ⇒ 'b ⇒ 'c) ⇒ ('d ⇒ 'a) ⇒ ('e ⇒ 'b) ⇒ 'd ⇒ 'e ⇒ 'c" ("$\mathtt{B}_{11}$")*
  where "$\mathtt{B}_{11} \equiv \lambda f\ g_1\ g_2\ x_1\ x_2.\ f\ (g_1\ x_1)\ (g_2\ x_2)$"
definition *B12_comb :: "('a ⇒ 'b ⇒ 'c) ⇒ ('d ⇒ 'a) ⇒ ('e ⇒ 'f ⇒ 'b) ⇒ 'd ⇒ 'e ⇒ 'f ⇒ 'c" ("$\mathtt{B}_{12}$")*
  where "$\mathtt{B}_{12} \equiv \lambda f\ g_1\ g_2\ x_1\ x_2\ y_2.\ f\ (g_1\ x_1)\ (g_2\ x_2\ y_2)$"
definition *B13_comb :: "('a ⇒ 'b ⇒ 'c) ⇒ ('d ⇒ 'a) ⇒ ('e ⇒ 'f ⇒ 'g ⇒ 'b)*
⇒ 'd ⇒ 'e ⇒ 'f ⇒ 'g ⇒ 'c" ("$\mathtt{B}_{13}$")
  where "$\mathtt{B}_{13} \equiv \lambda f\ g_1\ g_2\ x_1\ x_2\ y_2\ z_2.\ f\ (g_1\ x_1)\ (g_2\ x_2\ y_2\ z_2)$"
— ... and so on
definition *B20_comb :: "('a ⇒ 'b ⇒ 'c) ⇒ ('d ⇒ 'e ⇒ 'a) ⇒ 'b ⇒ 'd ⇒ 'e ⇒ 'c" ("$\mathtt{B}_{20}$")*
  where "$\mathtt{B}_{20} \equiv \lambda f\ g_1\ g_2\ x_1\ y_1.\ f\ (g_1\ x_1\ y_1)\ g_2$"
definition *B21_comb :: "('a ⇒ 'b ⇒ 'c) ⇒ ('d ⇒ 'e ⇒ 'a) ⇒ ('f ⇒ 'b) ⇒ 'd ⇒ 'f ⇒ 'e ⇒ 'c" ("$\mathtt{B}_{21}$")*
  where "$\mathtt{B}_{21} \equiv \lambda f\ g_1\ g_2\ x_1\ x_2\ y_1.\ f\ (g_1\ x_1\ y_1)\ (g_2\ x_2)$"
definition *B22_comb :: "('a ⇒ 'b ⇒ 'c) ⇒ ('d ⇒ 'e ⇒ 'a) ⇒ ('f ⇒ 'g ⇒ 'b)*
⇒ 'd ⇒ 'f ⇒ 'e ⇒ 'g ⇒ 'c" ("$\mathtt{B}_{22}$")
  where "$\mathtt{B}_{22} \equiv \lambda f\ g_1\ g_2\ x_1\ x_2\ y_1\ y_2.\ f\ (g_1\ x_1\ y_1)\ (g_2\ x_2\ y_2)$"
— ... and so on
definition *B30_comb :: "('a ⇒ 'b ⇒ 'c) ⇒ ('d ⇒ 'e ⇒ 'f ⇒ 'a) ⇒ 'b ⇒ 'd ⇒ 'e ⇒ 'f ⇒ 'c" ("$\mathtt{B}_{30}$")*
  where "$\mathtt{B}_{30} \equiv \lambda f\ g_1\ g_2\ x_1\ y_1\ z_1.\ f\ (g_1\ x_1\ y_1\ z_1)\ g_2$"
— ... and so on
abbreviation *(input) B000_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ 'a ⇒ 'b ⇒ 'c ⇒ 'd"("$\mathtt{B}_{000}$")*
  where "$\mathtt{B}_{000} \equiv \mathtt{A}_3$" — composing with three nullary functions corresponds to ternary function application
— ... and so on
definition *B111_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ ('e ⇒ 'a) ⇒ ('f ⇒ 'b) ⇒ ('g ⇒ 'c)*
⇒ 'e ⇒ 'f ⇒ 'g ⇒ 'd" ("$\mathtt{B}_{111}$")
  where "$\mathtt{B}_{111} \equiv \lambda f\ g_1\ g_2\ g_3\ x_1\ x_2\ x_3.\ f\ (g_1\ x_1)\ (g_2\ x_2)\ (g_3\ x_3)$"
definition *B112_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ ('e ⇒ 'a) ⇒ ('f ⇒ 'b) ⇒ ('g ⇒ 'h ⇒ 'c)*
⇒ 'e ⇒ 'f ⇒ 'g ⇒ 'h ⇒ 'd" ("$\mathtt{B}_{112}$")

where "$\mathtt{B}_{112} \equiv \lambda f\ g_1\ g_2\ g_3\ x_1\ x_2\ x_3\ y_3.\ f\ (g_1\ x_1)\ (g_2\ x_2)\ (g_3\ x_3\ y_3)$"
— ... and so on
definition *B222_comb* :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ ('e ⇒ 'f ⇒ 'a) ⇒ ('g ⇒ 'h ⇒ 'b) ⇒ ('i ⇒ 'j ⇒ 'c) ⇒ 'e ⇒ 'g ⇒ 'i ⇒ 'f ⇒ 'h ⇒ 'j ⇒ 'd" ("$\mathtt{B}_{222}$")
where "$\mathtt{B}_{222} \equiv \lambda f\ g_1\ g_2\ g_3\ x_1\ x_2\ x_3\ y_1\ y_2\ y_3.\ f\ (g_1\ x_1\ y_1)\ (g_2\ x_2\ y_2)\ (g_3\ x_3\ y_3)$"
definition *B223_comb* :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ ('e ⇒ 'f ⇒ 'a) ⇒ ('g ⇒ 'h ⇒ 'b) ⇒ ('i ⇒ 'j ⇒ 'k ⇒ 'c) ⇒ 'e ⇒ 'g ⇒ 'i ⇒ 'f ⇒ 'h ⇒ 'j ⇒ 'k ⇒ 'd" ("$\mathtt{B}_{223}$")
where "$\mathtt{B}_{223} \equiv \lambda f\ g_1\ g_2\ g_3\ x_1\ x_2\ x_3\ y_1\ y_2\ y_3\ z_3.\ f\ (g_1\ x_1\ y_1)\ (g_2\ x_2\ y_2)\ (g_3\ x_3\ y_3\ z_3)$"
— ... and so on
definition *B333_comb* :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ ('e ⇒ 'f ⇒ 'g ⇒ 'a) ⇒ ('h ⇒ 'i ⇒ 'j ⇒ 'b) ⇒ ('k ⇒ 'l ⇒ 'm ⇒ 'c) ⇒ 'e ⇒ 'h ⇒ 'k ⇒ 'f ⇒ 'i ⇒ 'l ⇒ 'g ⇒ 'j ⇒ 'm ⇒ 'd" ("$\mathtt{B}_{333}$")
where "$\mathtt{B}_{333} \equiv \lambda f\ g_1\ g_2\ g_3\ x_1\ x_2\ x_3\ y_1\ y_2\ y_3\ z_1\ z_2\ z_3.\ f\ (g_1\ x_1\ y_1\ z_1)\ (g_2\ x_2\ y_2\ z_2)\ (g_3\ x_3\ y_3\ z_3)$"
— ... and so on
definition *B1111_comb* :: "('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e) ⇒ ('f ⇒ 'a) ⇒ ('g ⇒ 'b) ⇒ ('h ⇒ 'c) ⇒ ('i ⇒ 'd) ⇒ 'f ⇒ 'g ⇒ 'h ⇒ 'i ⇒ 'e" ("$\mathtt{B}_{1111}$")
where "$\mathtt{B}_{1111} \equiv \lambda f\ g_1\ g_2\ g_3\ g_4\ x_1\ x_2\ x_3\ x_4.\ f\ (g_1\ x_1)\ (g_2\ x_2)\ (g_3\ x_3)\ (g_4\ x_4)$"
— ... and so on

We introduce a convenient infix notation for the $\mathtt{B}_n$ family of combinators (and their transposes) in their role as arity-extended versions of composition, and write $\mathtt{B}_n\ f\ g$ as $f\ \circ_n\ g$.

notation *B1_comb* (**infixl** "$\circ_1$" 55)
notation *B2_comb* (**infixl** "$\circ_2$" 55)
notation *B3_comb* (**infixl** "$\circ_3$" 55)
notation *B4_comb* (**infixl** "$\circ_4$" 55)
abbreviation(*input*) *B1_comb_t* (**infixr** "$;_1$" 55)
where "$f\ ;_1\ g \equiv g\ \circ_1\ f$"
abbreviation(*input*) *B2_comb_t* (**infixr** "$;_2$" 55)
where "$f\ ;_2\ g \equiv g\ \circ_2\ f$"
abbreviation(*input*) *B3_comb_t* (**infixr** "$;_3$" 55)
where "$f\ ;_3\ g \equiv g\ \circ_3\ f$"
abbreviation(*input*) *B4_comb_t* (**infixr** "$;_4$" 55)
where "$f\ ;_4\ g \equiv g\ \circ_4\ f$"

Convenient default notation.

notation *B1_comb* ("B")
notation *B1_comb* (**infixl** "∘" 55)
abbreviation(*input*) *B1_comb_t'* (**infixr** ";" 55)
where "$f\ ;\ g \equiv g \circ f$"

Alternative notations for some known compositors in the literature.

notation *B01_comb* ("D") — aliasing $\mathtt{B}_{01}$ as D (cf. Smullyan's "dove" combinator)
notation *B02_comb* ("E") — aliasing $\mathtt{B}_{02}$ as E (cf. Smullyan's "eagle" combinator)

declare *B1_comb_def[comb_defs] B2_comb_def[comb_defs] B3_comb_def[comb_defs] B4_comb_def[comb_defs] B01_comb_def[comb_defs] B02_comb_def[comb_defs] B03_comb_def[comb_defs] B10_comb_def[comb_defs] B11_comb_def[comb_defs] B12_comb_def[comb_defs] B13_comb_def[comb_defs] B20_comb_def[comb_defs] B21_comb_def[comb_defs] B22_comb_def[comb_defs] B30_comb_def[comb_defs] B111_comb_def[comb_defs] B112_comb_def[comb_defs] B222_comb_def[comb_defs] B223_comb_def[comb_defs] B333_comb_def[comb_defs] B1111_comb_def[comb_defs]*

Notation checks.

lemma "$f \circ g \circ h$ = h ; g ; f"
lemma "$f \circ_2 g$ = g ;$_2$ f"
lemma "a |> f |> g = a |> f ; g"
lemma "a |> f |> g = g ∘ f <| a"
lemma "a |> f |> g = f ; g <| a"

Composing compositors. In the following cases, we have that $\mathtt{B}_{ab} \circ \mathtt{B}_{cd} = \mathtt{B}_{(a+b)(c+d)}$.

lemma "$\mathtt{B}_{11} \circ \mathtt{B}_{00} = \mathtt{B}_{11}$"
lemma "$\mathtt{B}_{10} \circ \mathtt{B}_{01} = \mathtt{B}_{11}$"
lemma "$\mathtt{B}_{12} \circ \mathtt{B}_{00} = \mathtt{B}_{12}$"
lemma "$\mathtt{B}_{11} \circ \mathtt{B}_{01} = \mathtt{B}_{12}$"
lemma "$\mathtt{B}_{10} \circ \mathtt{B}_{10} = \mathtt{B}_{20}$"
lemma "$\mathtt{B}_{11} \circ \mathtt{B}_{10} = \mathtt{B}_{21}$"
lemma "$\mathtt{B}_{11} \circ \mathtt{B}_{11} = \mathtt{B}_{22}$"

Similarly, below we have that $\mathtt{B}_{abc} \circ \mathtt{B}_{def} = \mathtt{B}_{(a+d)(b+e)(c+f)}$.

lemma "$\mathtt{B}_{000} \circ \mathtt{B}_{111} = \mathtt{B}_{111}$"
lemma "$\mathtt{B}_{111} \circ \mathtt{B}_{111} = \mathtt{B}_{222}$"
lemma "$\mathtt{B}_{111} \circ \mathtt{B}_{112} = \mathtt{B}_{223}$"
lemma "$\mathtt{B}_{111} \circ \mathtt{B}_{222} = \mathtt{B}_{333}$"
lemma "$\mathtt{B}_{222} \circ \mathtt{B}_{111} = \mathtt{B}_{333}$"

Note, however, that:

proposition "$\mathtt{B}_{01} \circ \mathtt{B}_{10} = \mathtt{B}_{11}$" **nitpick** — countermodel found
proposition "$\mathtt{B}_{01} \circ \mathtt{B}_{11} = \mathtt{B}_{12}$" **nitpick** — countermodel found
proposition "$\mathtt{B}_{112} \circ \mathtt{B}_{111} = \mathtt{B}_{223}$" **nitpick** — countermodel found

1.1.3 Permutators

The family of combinators $\mathtt{C}_N$ are called "permutators", where N an m-sized sequence of (different) numbers indicating a permutation on the arguments of the first argument (an m-ary function).

abbreviation *(input) C12_comb :: "('a ⇒ 'b ⇒ 'c) ⇒ 'a ⇒ 'b ⇒ 'c" ("$\mathtt{C}_{12}$")*
  where "$\mathtt{C}_{12} \equiv \mathtt{A}_2$" — trivial case (no permutation): binary function application
definition *C21_comb :: "('a ⇒ 'b ⇒ 'c) ⇒ 'b ⇒ 'a ⇒ 'c" ("$\mathtt{C}_{21}$")*
  where "$\mathtt{C}_{21} \equiv \lambda f\ x_1\ x_2.\ f\ x_2\ x_1$"
— Ternary permutators (6 in total).
abbreviation *(input) C123_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ 'a ⇒ 'b ⇒ 'c ⇒ 'd" ("$\mathtt{C}_{123}$")*

  where "$\mathtt{C}_{123} \equiv \mathtt{A}_3$" — trivial case (no permutation): ternary function application
abbreviation *(input) C213_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ 'b ⇒ 'a ⇒ 'c ⇒ 'd" ("$\mathtt{C}_{213}$")*

  where "$\mathtt{C}_{213} \equiv \mathtt{C}_{21}$" — permutation $\mathtt{C}_{213}$ corresponds to $\mathtt{C}_{21}$ (flipping the first two arguments)
definition *C132_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ 'a ⇒ 'c ⇒ 'b ⇒ 'd" ("$\mathtt{C}_{132}$")*
  where "$\mathtt{C}_{132} \equiv \lambda f\ x_1\ x_2\ x_3.\ f\ x_1\ x_3\ x_2$"
definition *C231_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ 'c ⇒ 'a ⇒ 'b ⇒ 'd" ("$\mathtt{C}_{231}$")*
  where "$\mathtt{C}_{231} \equiv \lambda f\ x_1\ x_2\ x_3.\ f\ x_2\ x_3\ x_1$"
definition *C312_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ 'b ⇒ 'c ⇒ 'a ⇒ 'd" ("$\mathtt{C}_{312}$")*
  where "$\mathtt{C}_{312} \equiv \lambda f\ x_1\ x_2\ x_3.\ f\ x_3\ x_1\ x_2$"
definition *C321_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ 'c ⇒ 'b ⇒ 'a ⇒ 'd" ("$\mathtt{C}_{321}$")*
  where "$\mathtt{C}_{321} \equiv \lambda f\ x_1\ x_2\ x_3.\ f\ x_3\ x_2\ x_1$"
— Quaternary permutators (24 in total) we define some below (the rest are added on demand).
abbreviation *(input) C2134_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e) ⇒ 'b ⇒ 'a ⇒ 'c ⇒ 'd ⇒ 'e" ("$\mathtt{C}_{2134}$")*
  where "$\mathtt{C}_{2134} \equiv \mathtt{C}_{21}$" — permutation $\mathtt{C}_{2134}$ corresponds to $\mathtt{C}_{21}$ (flipping the first two arguments)
definition *C1243_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e) ⇒ 'a ⇒ 'b ⇒ 'd ⇒ 'c ⇒ 'e" ("$\mathtt{C}_{1243}$")*
  where "$\mathtt{C}_{1243} \equiv \lambda f\ x_1\ x_2\ x_3\ x_4.\ f\ x_1\ x_2\ x_4\ x_3$"
definition *C1324_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e) ⇒ 'a ⇒ 'c ⇒ 'b ⇒ 'd ⇒ 'e" ("$\mathtt{C}_{1324}$")*

where "$\mathtt{C}_{1324} \equiv \lambda f\ x_1\ x_2\ x_3\ x_4.\ f\ x_1\ x_3\ x_2\ x_4$"
definition *C1423_comb* :: "('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e) ⇒ 'a ⇒ 'c ⇒ 'd ⇒ 'b ⇒ 'e" ("$\mathtt{C}_{1423}$")
where "$\mathtt{C}_{1423} \equiv \lambda f\ x_1\ x_2\ x_3\ x_4.\ f\ x_1\ x_4\ x_2\ x_3$"
definition *C2143_comb* :: "('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e) ⇒ 'b ⇒ 'a ⇒ 'd ⇒ 'c ⇒ 'e"("$\mathtt{C}_{2143}$")
where "$\mathtt{C}_{2143} \equiv \lambda f\ x_1\ x_2\ x_3\ x_4.\ f\ x_2\ x_1\ x_4\ x_3$"
definition *C2314_comb* :: "('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e) ⇒ 'c ⇒ 'a ⇒ 'b ⇒ 'd ⇒ 'e" ("$\mathtt{C}_{2314}$")
where "$\mathtt{C}_{2314} \equiv \lambda f\ x_1\ x_2\ x_3\ x_4.\ f\ x_2\ x_3\ x_1\ x_4$"
definition *C3142_comb* :: "('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e) ⇒ 'b ⇒ 'd ⇒ 'a ⇒ 'c ⇒ 'e" ("$\mathtt{C}_{3142}$")
where "$\mathtt{C}_{3142} \equiv \lambda f\ x_1\ x_2\ x_3\ x_4.\ f\ x_3\ x_1\ x_4\ x_2$"
definition *C3412_comb* :: "('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e) ⇒ 'c ⇒ 'd ⇒ 'a ⇒ 'b ⇒ 'e" ("$\mathtt{C}_{3412}$")
where "$\mathtt{C}_{3412} \equiv \lambda f\ x_1\ x_2\ x_3\ x_4.\ f\ x_3\ x_4\ x_1\ x_2$" — note that arguments are flipped pairwise
— ... and so on

Introduce some convenient combinator notations.

notation *C21_comb* ("C") — the traditional flip/transposition (C) combinator is $\mathtt{C}_{21}$
notation *C3412_comb* ("$\mathtt{C}_2$") — pairwise flip/transposition of the arguments of a quaternary function
notation *C231_comb* ("R") — right (counterclockwise) rotation of a ternary function
notation *C312_comb* ("L") — left (counterclockwise) rotation of a ternary function
notation *C321_comb* ("C''") — Full reversal of the arguments of a ternary function

declare *C21_comb_def[comb_defs]*
C132_comb_def[comb_defs] C231_comb_def[comb_defs] C312_comb_def[comb_defs]
C321_comb_def[comb_defs] C1243_comb_def[comb_defs] C1324_comb_def[comb_defs]
C1423_comb_def[comb_defs] C2143_comb_def[comb_defs] C2314_comb_def[comb_defs]
C3142_comb_def[comb_defs] C3412_comb_def[comb_defs]

Composing rotation combinators (identity, left and right) works as expected.

lemma "R ∘ L = L ∘ R"
lemma "R = L ∘ L"
lemma "L = R ∘ R"
lemma "I = L ∘ L ∘ L"
lemma "I = R ∘ R ∘ R"

1.1.4 Cancellators

The next family of combinators $\mathtt{K}_{mn}$ are called "cancellators". They take m arguments and return the n-th one (thus ignoring or "cancelling" all others).

abbreviation*(input) K11_comb::"'a ⇒ 'a" ("$\mathtt{K}_{11}$")*
where "$\mathtt{K}_{11} \equiv$ I" — trivial/degenerate case m = 1: identity combinator I
definition *K21_comb::"'a ⇒ 'b ⇒ 'a" ("$\mathtt{K}_{21}$")*
where "$\mathtt{K}_{21} \equiv \lambda x\ y.\ x$" — the traditional K combinator
definition *K22_comb::"'a ⇒ 'b ⇒ 'b" ("$\mathtt{K}_{22}$")*
where "$\mathtt{K}_{22} \equiv \lambda x\ y.\ y$"
definition *K31_comb::"'a ⇒ 'b ⇒ 'c ⇒ 'a" ("$\mathtt{K}_{31}$")*
where "$\mathtt{K}_{31} \equiv \lambda x\ y\ z.\ x$"
definition *K32_comb::"'a ⇒ 'b ⇒ 'c ⇒ 'b" ("$\mathtt{K}_{32}$")*
where "$\mathtt{K}_{32} \equiv \lambda x\ y\ z.\ y$"
definition *K33_comb::"'a ⇒ 'b ⇒ 'c ⇒ 'c" ("$\mathtt{K}_{33}$")*
where "$\mathtt{K}_{33} \equiv \lambda x\ y\ z.\ z$"
— ... and so on

notation *K21_comb* ("K") — aliasing $\mathtt{K}_{21}$ as K

declare *K21_comb_def[comb_defs] K22_comb_def[comb_defs]*
K31_comb_def[comb_defs] K32_comb_def[comb_defs] K33_comb_def[comb_defs]

1.1.5 Contractors

abbreviation*(input) W11_comb :: "('a ⇒ 'b) ⇒ 'a ⇒ 'b" ("$\mathtt{W}_{11}$")*

where "$W_{11} \equiv A_1$" — for the degenerate case m = 1: $W_{1n} = A_n$
abbreviation *(input) W12_comb :: "('a ⇒ 'b ⇒ 'c) ⇒ 'a ⇒ 'b ⇒ 'c" ("W_{12}")*
where "$W_{12} \equiv A_2$"
abbreviation *(input) W13_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ 'a ⇒ 'b ⇒ 'c ⇒ 'd" ("W_{13}")*
where "$W_{13} \equiv A_3$"
— ... and so on
definition *W21_comb :: "('a ⇒ 'a ⇒ 'b) ⇒ 'a ⇒ 'b" ("W_{21}")*
where "$W_{21} \equiv \lambda f\ x.\ f\ x\ x$"
definition *W22_comb :: "('a ⇒ 'a ⇒ 'b ⇒ 'b ⇒ 'c) ⇒ 'a ⇒ 'b ⇒ 'c" ("W_{22}")*
where "$W_{22} \equiv \lambda f\ x\ y.\ f\ x\ x\ y\ y$"
definition *W23_comb :: "('a ⇒ 'a ⇒ 'b ⇒ 'b ⇒ 'c ⇒ 'c ⇒ 'd) ⇒ 'a ⇒ 'b ⇒ 'c ⇒ 'd" ("W_{23}")*
where "$W_{23} \equiv \lambda f\ x\ y\ z.\ f\ x\ x\ y\ y\ z\ z$"
— ... and so on
definition *W31_comb :: "('a ⇒ 'a ⇒ 'a ⇒ 'b) ⇒ 'a ⇒ 'b" ("W_{31}")*
where "$W_{31} \equiv \lambda f\ x.\ f\ x\ x\ x$"
definition *W32_comb :: "('a ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ⇒ 'b ⇒ 'c) ⇒ 'a ⇒ 'b ⇒ 'c" ("W_{32}")*
where "$W_{32} \equiv \lambda f\ x\ y.\ f\ x\ x\ x\ y\ y\ y$"
definition *W33_comb :: "('a ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ⇒ 'b ⇒ 'c ⇒ 'c ⇒ 'c ⇒ 'd)
⇒ 'a ⇒ 'b ⇒ 'c ⇒ 'd"
("W_{33}")*
where "$W_{33} \equiv \lambda f\ x\ y\ z.\ f\ x\ x\ x\ y\ y\ y\ z\ z\ z$"
— ... and so on

notation *W21_comb ("W")* — the traditional W combinator corresponds to W_{21}

declare *W21_comb_def[comb_defs] W31_comb_def[comb_defs]
W22_comb_def[comb_defs] W23_comb_def[comb_defs]
W32_comb_def[comb_defs] W33_comb_def[comb_defs]*

1.1.6 Fusers

The families S_{mn} (resp. Σ_{mn}) generalize the combinator S (resp. its evil twin Σ) towards higher arities.

definition *S11_comb :: "('a ⇒ 'b ⇒ 'c) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'c" ("S_{11}")*
where "$S_{11} \equiv \lambda f\ g\ x.\ f\ x\ (g\ x)$" — aka. S (same as B$\Sigma$C)
definition *S12_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ 'a ⇒ 'b ⇒ 'd" ("S_{12}")*
where "$S_{12} \equiv \lambda f\ g\ x\ y.\ f\ x\ y\ (g\ x\ y)$"
definition *S13_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e) ⇒ ('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ 'a ⇒ 'b
⇒ 'c ⇒ 'e" ("S_{13}")*
where "$S_{13} \equiv \lambda f\ g\ x\ y\ z.\ f\ x\ y\ z\ (g\ x\ y\ z)$"
— ... and so on
definition *S21_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ ('a ⇒ 'b) ⇒ ('a ⇒ 'c) ⇒ 'a ⇒ 'd" ("S_{21}")*
where "$S_{21} \equiv \lambda f\ g_1\ g_2\ x.\ f\ x\ (g_1\ x)\ (g_2\ x)$"
definition *S22_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e) ⇒ ('a ⇒ 'b ⇒ 'c)
⇒ ('a ⇒ 'b ⇒ 'd) ⇒ 'a ⇒ 'b ⇒ 'e"
("S_{22}")*
where "$S_{22} \equiv \lambda f\ g_1\ g_2\ x\ y.\ f\ x\ y\ (g_1\ x\ y)\ (g_2\ x\ y)$"
definition *S23_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e ⇒ 'f ⇒ 'g) ⇒ ('a ⇒ 'b ⇒ 'c ⇒ 'd)
⇒ ('a ⇒ 'b ⇒ 'c ⇒ 'e) ⇒ ('a ⇒ 'b ⇒ 'c ⇒ 'f) ⇒ 'a ⇒ 'b ⇒
'c ⇒ 'g" ("S_{23}")*
where "$S_{23} \equiv \lambda f\ g_1\ g_2\ g_3\ x\ y\ z.\ f\ x\ y\ z\ (g_1\ x\ y\ z)\ (g_2\ x\ y\ z)\ (g_3\ x\ y\ z)$"
— ... and so on
definition *Σ11_comb :: "('a ⇒ 'b ⇒ 'c) ⇒ ('b ⇒ 'a) ⇒ 'b ⇒ 'c" ("Σ_{11}")*
where "$\Sigma_{11} \equiv \lambda f\ g\ x.\ f\ (g\ x)\ x$ " — aka. Σ (same as BSC)
definition *Σ12_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ ('b ⇒ 'c ⇒ 'a) ⇒ 'b ⇒ 'c ⇒ 'd" ("Σ_{12}")*
where "$\Sigma_{12} \equiv \lambda f\ g\ x\ y.\ f\ (g\ x\ y)\ x\ y$" — same as $BS_{12}L$
definition *Σ13_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e) ⇒ ('b ⇒ 'c ⇒ 'd ⇒ 'a) ⇒ 'b ⇒ 'c
⇒ 'd ⇒ 'e" ("Σ_{13}")*
where "$\Sigma_{13} \equiv \lambda f\ g\ x\ y\ z.\ f\ (g\ x\ y\ z)\ x\ y\ z$"

— ... and so on
definition Σ*21_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ ('c ⇒ 'a) ⇒ ('c ⇒ 'b) ⇒ 'c ⇒ 'd" ("*$\mathbf{\Sigma}_{21}$*")*
  where *"*$\mathbf{\Sigma}_{21} \equiv \lambda f\ g_1\ g_2\ x.\ f\ (g_1\ x)\ (g_2\ x)\ x$*"*
definition Σ*22_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e) ⇒ ('c ⇒ 'd ⇒ 'a) ⇒ ('c ⇒ 'd ⇒ 'b) ⇒ 'c ⇒ 'd ⇒ 'e" ("*$\mathbf{\Sigma}_{22}$*")*
  where *"*$\mathbf{\Sigma}_{22} \equiv \lambda f\ g_1\ g_2\ x\ y.\ f\ (g_1\ x\ y)\ (g_2\ x\ y)\ x\ y$*"*
definition Σ*23_comb :: "('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e ⇒ 'f ⇒ 'g) ⇒ ('d ⇒ 'e ⇒ 'f ⇒ 'a) ⇒ ('d ⇒ 'e ⇒ 'f ⇒ 'b) ⇒ ('d ⇒ 'e ⇒ 'f ⇒ 'c) ⇒ 'd ⇒ 'e ⇒ 'f ⇒ 'g" ("*$\mathbf{\Sigma}_{23}$*")*
  where *"*$\mathbf{\Sigma}_{23} \equiv \lambda f\ g_1\ g_2\ g_3\ x\ y\ z.\ f\ (g_1\ x\ y\ z)\ (g_2\ x\ y\ z)\ (g_3\ x\ y\ z)\ x\ y\ z$*"*

notation *S11_comb ("S")*
notation Σ*11_comb ("*$\mathbf{\Sigma}$*")*

declare *S11_comb_def[comb_defs] S12_comb_def[comb_defs] S13_comb_def[comb_defs] S21_comb_def[comb_defs] S22_comb_def[comb_defs] S23_comb_def[comb_defs]* Σ*11_comb_def[comb_defs]* Σ*12_comb_def[comb_defs]* Σ*13_comb_def[comb_defs]*

`S`/$\mathbf{\Sigma}$ can be defined in terms of other combinators.

lemma "S = B (B (B W) C) (B B)"
lemma "S = B (B W)(B B C)"
lemma "$\mathbf{\Sigma}$ = B (B W) B"
lemma "S = B $\mathbf{\Sigma}$ C"
lemma "$\mathbf{\Sigma}$ = B S C"
lemma "S = B (T C) B $\mathbf{\Sigma}$"
lemma "$\mathbf{\Sigma}$ = B (T C) B S"
lemma "$\mathbf{\Sigma}_{12}$ = B $\mathtt{S}_{12}$ L"

1.2 Further Combinators

1.2.1 Preprocessors

The family of $\mathbf{\Psi}_m$ combinators below are special cases of compositors. They take an m-ary function *f* and prepend to each of its m inputs a given unary function *g* (acting as a "preprocessor").

abbreviation*(input)* Ψ*1_comb :: "('a ⇒ 'b) ⇒ ('c ⇒ 'a) ⇒ 'c ⇒ 'b" ("*$\mathbf{\Psi}_1$*")*
  where *"*$\mathbf{\Psi}_1 \equiv$ B*"* — trivial case m = 1 corresponds to the `B` combinator
definition Ψ*2_comb :: "('a ⇒ 'a ⇒ 'b) ⇒ ('c ⇒ 'a) ⇒ 'c ⇒ 'c ⇒ 'b" ("*$\mathbf{\Psi}_2$*")*
  where *"*$\mathbf{\Psi}_2 \equiv \lambda f\ g\ x\ y.\ f\ (g\ x)\ (g\ y)$*"* — cf. "Ψ" in [1]; "on" in Haskell Data.Function
definition Ψ*3_comb :: "('a ⇒ 'a ⇒ 'a ⇒ 'b) ⇒ ('c ⇒ 'a) ⇒ 'c ⇒ 'c ⇒ 'c ⇒ 'b" ("*$\mathbf{\Psi}_3$*")*
  where *"*$\mathbf{\Psi}_3 \equiv \lambda f\ g\ x\ y\ z.\ f\ (g\ x)\ (g\ y)\ (g\ z)$*"*
definition Ψ*4_comb :: "('a ⇒ 'a ⇒ 'a ⇒ 'a ⇒ 'b) ⇒ ('c ⇒ 'a) ⇒ 'c ⇒ 'c ⇒ 'c ⇒ 'c ⇒ 'b" ("*$\mathbf{\Psi}_4$*")*
  where *"*$\mathbf{\Psi}_4 \equiv \lambda f\ g\ x\ y\ z\ u.\ f\ (g\ x)\ (g\ y)\ (g\ z)\ (g\ u)$*"*
— ... and so on

declare Ψ*2_comb_def[comb_defs]* Ψ*3_comb_def[comb_defs]* Ψ*4_comb_def[comb_defs]*

1.2.2 Imitators

The combinators $\mathbf{\Phi}_{mn}$ are called "imitators". They compose a m-ary function *f* with m functions $g_{i \leq m}$ (having arity n each) by sharing their input arguments, so as to return an n-ary function. They can be seen as a kind of "input-merging compositors". These combinators are quite convenient and appear often in the literature, e.g., as "trains" in array languages like APL and BQN, and in "imitation bindings" in higher-order (pre-)unification algorithms (from where they get their name).

Conveniently introduce a (degenerate) case m = 0 as abbreviation, where $\mathbf{\Phi}_{0n}$ corresponds to $\mathtt{K}_{(n+1)1}$.

abbreviation *(input)* Φ*01_comb* :: "'a ⇒ 'b ⇒ 'a" ("$\mathbf{\Phi}_{01}$")
  where "$\mathbf{\Phi}_{01} \equiv \mathtt{K}_{21}$"
abbreviation *(input)* Φ*02_comb* :: "'a ⇒ 'b ⇒ 'c ⇒ 'a" ("$\mathbf{\Phi}_{02}$")
  where "$\mathbf{\Phi}_{02} \equiv \mathtt{K}_{31}$"
— ...and so on

Each combinator $\mathbf{\Phi}_{1n}$ corresponds in fact to $\mathtt{B}_n$.

abbreviation *(input)* Φ*11_comb* :: "('a ⇒ 'b) ⇒ ('c ⇒ 'a) ⇒ 'c ⇒ 'b" ("$\mathbf{\Phi}_{11}$")
  where "$\mathbf{\Phi}_{11} \equiv \mathtt{B}_1$"
abbreviation *(input)* Φ*12_comb* :: "('a ⇒ 'b) ⇒ ('c ⇒ 'd ⇒ 'a) ⇒ 'c ⇒ 'd ⇒ 'b" ("$\mathbf{\Phi}_{12}$")

  where "$\mathbf{\Phi}_{12} \equiv \mathtt{B}_2$"
abbreviation *(input)* Φ*13_comb* :: "('a ⇒ 'b) ⇒ ('c ⇒ 'd ⇒ 'e ⇒ 'a) ⇒ 'c ⇒ 'd ⇒ 'e ⇒ 'b" ("$\mathbf{\Phi}_{13}$")
  where "$\mathbf{\Phi}_{13} \equiv \mathtt{B}_3$"
abbreviation *(input)* Φ*14_comb* :: "('a ⇒ 'b) ⇒ ('c ⇒ 'd ⇒ 'e ⇒ 'f ⇒ 'a) ⇒ 'c ⇒ 'd ⇒ 'e ⇒ 'f ⇒ 'b" ("$\mathbf{\Phi}_{14}$")
  where "$\mathbf{\Phi}_{14} \equiv \mathtt{B}_4$"
— ...and so on

Combinators $\mathbf{\Phi}_{mn}$ with m > 1 have their idiosyncratic definition.

definition Φ*21_comb* :: "('a ⇒ 'b ⇒ 'c) ⇒ ('d ⇒ 'a) ⇒ ('d ⇒ 'b) ⇒ 'd ⇒ 'c" ("$\mathbf{\Phi}_{21}$")
  where "$\mathbf{\Phi}_{21} \equiv \lambda f\ g_1\ g_2\ x.\ f\ (g_1\ x)\ (g_2\ x)$" — cf. "$\Phi_1$" in [1]; "liftA2" in Haskell; "monadic fork" in APL)
definition Φ*22_comb* :: "('a ⇒ 'b ⇒ 'c) ⇒ ('d ⇒ 'e ⇒ 'a) ⇒ ('d ⇒ 'e ⇒ 'b) ⇒ 'd ⇒ 'e ⇒ 'c" ("$\mathbf{\Phi}_{22}$")
  where "$\mathbf{\Phi}_{22} \equiv \lambda f\ g_1\ g_2\ x\ y.\ f\ (g_1\ x\ y)\ (g_2\ x\ y)$" — cf. "$\Phi_2$" in [1]; "dyadic fork" in APL
— ...and so on
definition Φ*31_comb* :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ ('e ⇒ 'a) ⇒ ('e ⇒ 'b) ⇒ ('e ⇒ 'c) ⇒ 'e ⇒ 'd" ("$\mathbf{\Phi}_{31}$")
  where "$\mathbf{\Phi}_{31} \equiv \lambda f\ g_1\ g_2\ g_3\ x.\ f\ (g_1\ x)\ (g_2\ x)\ (g_3\ x)$"
definition Φ*32_comb* :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ ('e ⇒ 'f ⇒ 'a) ⇒ ('e ⇒ 'f ⇒ 'b)
⇒ ('e ⇒ 'f ⇒ 'c) ⇒ 'e ⇒ 'f ⇒ 'd" ("$\mathbf{\Phi}_{32}$")
  where "$\mathbf{\Phi}_{32} \equiv \lambda f\ g_1\ g_2\ g_3\ x\ y.\ f\ (g_1\ x\ y)\ (g_2\ x\ y)\ (g_3\ x\ y)$"
definition Φ*33_comb* :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ ('e ⇒ 'f ⇒ 'g ⇒ 'a) ⇒ ('e ⇒ 'f ⇒ 'g ⇒ 'b)
⇒ ('e ⇒ 'f ⇒ 'g ⇒ 'c) ⇒ 'e ⇒ 'f ⇒ 'g ⇒ 'd" ("$\mathbf{\Phi}_{33}$")
  where "$\mathbf{\Phi}_{33} \equiv \lambda f\ g_1\ g_2\ g_3\ x\ y\ z.\ f\ (g_1\ x\ y\ z)\ (g_2\ x\ y\ z)\ (g_3\ x\ y\ z)$"
— ...and so on

declare Φ*21_comb_def[comb_defs]* Φ*22_comb_def[comb_defs]*
  Φ*31_comb_def[comb_defs]* Φ*32_comb_def[comb_defs]* Φ*33_comb_def[comb_defs]*

— $\mathbf{\Phi}_{m(i+j)}$ can be defined as: $\mathbf{\Phi}_{mi} \circ \mathbf{\Phi}_{mj}$.
lemma "$\mathbf{\Phi}_{12} = \mathbf{\Phi}_{11} \circ \mathbf{\Phi}_{11}$"
lemma "$\mathbf{\Phi}_{13} = \mathbf{\Phi}_{11} \circ \mathbf{\Phi}_{12}$"
lemma "$\mathbf{\Phi}_{13} = \mathbf{\Phi}_{12} \circ \mathbf{\Phi}_{11}$"
lemma "$\mathbf{\Phi}_{22} = \mathbf{\Phi}_{21} \circ \mathbf{\Phi}_{21}$"
lemma "$\mathbf{\Phi}_{32} = \mathbf{\Phi}_{31} \circ \mathbf{\Phi}_{31}$"
lemma "$\mathbf{\Phi}_{33} = \mathbf{\Phi}_{31} \circ \mathbf{\Phi}_{32}$"
lemma "$\mathbf{\Phi}_{33} = \mathbf{\Phi}_{32} \circ \mathbf{\Phi}_{31}$"

Moreover, $\mathbf{\Phi}_{mn}$ is definable by composing $\mathtt{W}_{mn}$ and $\mathtt{B}_N$, via the following schema: $\mathbf{\Phi}_{mn} = \mathtt{W}_{mn} \circ_{m+1} \mathtt{B}_N$ (where N is an m-sized array of ns).

lemma "$\mathbf{\Phi}_{11} = \mathtt{W}_{11} \circ_2 \mathtt{B}_1$"
lemma "$\mathbf{\Phi}_{12} = \mathtt{W}_{12} \circ_2 \mathtt{B}_2$"
lemma "$\mathbf{\Phi}_{13} = \mathtt{W}_{13} \circ_2 \mathtt{B}_3$"

lemma "$\mathbf{\Phi}_{21}$ = W_{21} $\circ_3$ B_{11}"
lemma "$\mathbf{\Phi}_{22}$ = W_{22} $\circ_3$ B_{22}"
lemma "$\mathbf{\Phi}_{31}$ = W_{31} $\circ_4$ B_{111}"
lemma "$\mathbf{\Phi}_{32}$ = W_{32} $\circ_4$ B_{222}"
lemma "$\mathbf{\Phi}_{33}$ = W_{33} $\circ_4$ B_{333}"

1.2.3 Projectors

The family of projectors $\mathbf{\Pi}_{lmn}$ features three parameters: *l* = total number of arguments; *m* ($\leq$ *l*) = the index of the projection; *n* = the arity of the (projected) m-th argument. They are used to construct "projection bindings" in higher-order (pre-)unification algorithms.

abbreviation *(input)* Π*110_comb* :: "'a ⇒ 'a" ("$\mathbf{\Pi}_{110}$")
  where "$\mathbf{\Pi}_{110}$ ≡ I" — trivial case corresponds to the identity combinator I
definition Π*111_comb* :: "(('a ⇒ 'b) ⇒ 'a) ⇒ ('a ⇒ 'b) ⇒ 'b" ("$\mathbf{\Pi}_{111}$") — Smullyan's "owl" [4]
  where "$\mathbf{\Pi}_{111}$ ≡ λh x. x (h x)"
definition Π*112_comb* :: "(('a ⇒ 'b ⇒ 'c) ⇒ 'a) ⇒ (('a ⇒ 'b ⇒ 'c) ⇒ 'b) ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ 'c" ("$\mathbf{\Pi}_{112}$")
  where "$\mathbf{\Pi}_{112}$ ≡ λh_1 h_2 x. x (h_1 x) (h_2 x)"
definition Π*113_comb* :: "(('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ 'a) ⇒ (('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ 'b)
    ⇒ (('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ 'c) ⇒ ('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ 'd" ("$\mathbf{\Pi}_{113}$")
  where "$\mathbf{\Pi}_{113}$ ≡ λh_1 h_2 h_3 x. x (h_1 x) (h_2 x) (h_3 x)"
— ...and so on
abbreviation *(input)* Π*210_comb* :: "'a ⇒ 'b ⇒ 'a" ("$\mathbf{\Pi}_{210}$")
  where "$\mathbf{\Pi}_{210}$ ≡ K_{21}" — trivial case corresponds to the combinator K_{21} (i.e. K)
definition Π*211_comb* :: "(('a ⇒ 'b) ⇒ 'c ⇒ 'a) ⇒ ('a ⇒ 'b) ⇒ 'c ⇒ 'b" ("$\mathbf{\Pi}_{211}$")
  where "$\mathbf{\Pi}_{211}$ ≡ λh x y. x (h x y)"
definition Π*212_comb* :: "(('a ⇒ 'b ⇒ 'c) ⇒ 'd ⇒ 'a)
    ⇒ (('a ⇒ 'b ⇒ 'c) ⇒ 'd ⇒ 'b) ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ 'd ⇒ 'c" ("$\mathbf{\Pi}_{212}$")
  where "$\mathbf{\Pi}_{212}$ ≡ λh_1 h_2 x y. x (h_1 x y) (h_2 x y)"
— ...and so on
abbreviation *(input)* Π*220_comb* :: "'a ⇒ 'b ⇒ 'b"("$\mathbf{\Pi}_{220}$")
  where "$\mathbf{\Pi}_{220}$ ≡ K_{22}" — trivial case corresponds to the combinator K_{22}
definition Π*221_comb* :: "('a ⇒ ('b ⇒ 'c) ⇒ 'b) ⇒ 'a ⇒ ('b ⇒ 'c) ⇒ 'c" ("$\mathbf{\Pi}_{221}$")
  where "$\mathbf{\Pi}_{221}$ ≡ λh x y. y (h x y)"
definition Π*222_comb* :: "('a ⇒ ('b ⇒ 'c ⇒ 'd) ⇒ 'b)
    ⇒ ('a ⇒ ('b ⇒ 'c ⇒ 'd) ⇒ 'c) ⇒ 'a ⇒ ('b ⇒ 'c ⇒ 'd) ⇒ 'd" ("$\mathbf{\Pi}_{222}$")
  where "$\mathbf{\Pi}_{222}$ ≡ λh_1 h_2 x y. y (h_1 x y) (h_2 x y)"
— ...and so on

declare Π*111_comb_def[comb_defs]* Π*112_comb_def[comb_defs]* Π*113_comb_def[comb_defs]*
  Π*211_comb_def[comb_defs]* Π*212_comb_def[comb_defs]*
  Π*221_comb_def[comb_defs]* Π*222_comb_def[comb_defs]*

notation Π*111_comb* ("O") — aliasing $\mathbf{\Pi}_{111}$ as O (cf. Smullyan's "owl" combinator)

Projectors $\mathbf{\Pi}_{lmn}$ can be defined as: S_{nl} K_{lm}

lemma "$\mathbf{\Pi}_{111}$ = S_{11} K_{11}"
lemma "$\mathbf{\Pi}_{112}$ = S_{21} K_{11}"
lemma "$\mathbf{\Pi}_{211}$ = S_{12} K_{21}"
lemma "$\mathbf{\Pi}_{212}$ = S_{22} K_{21}"
lemma "$\mathbf{\Pi}_{221}$ = S_{12} K_{22}"
lemma "$\mathbf{\Pi}_{222}$ = S_{22} K_{22}"

1.3 Combinator Interrelations

lemma "B = S (K S) K"
lemma "C = S (S (K (S (K S) K)) S) (K K)"
lemma "C = S (B B S) (K K)"

```
lemma "I = S K K"
lemma "W = S S (S K)"
lemma "W = C S I"
lemma "I = W K"
lemma "T = S (K (S (S K K))) K"
lemma "O = S I"
lemma "S = Φ21 I"
lemma "Φ21 = B (B S) B"
lemma "Σ = B2 W B"
lemma "W = Σ I"
lemma "W31 = W ∘ W"

lemma "B A = I"
lemma "C B2 A = B"
lemma "B C K  = B K "
lemma "C (C x) = x"
lemma "W f = S f I"
lemma "W f = Σ f I"
lemma "T = C I"
lemma "T = C A"

lemma "V = L A2"
lemma "V = L I"
lemma "I = R V"
lemma "R V = I"
lemma "V = L I"
lemma "L V = R I"
lemma "A2 = L(R I)"
lemma "L (C I) = C (R I)"
lemma "C (L I) = R (C I)"

end
```

2 Bridge with Isabelle/HOL Logic

This theory provides a bridge or "wrapper" for logic-based developments in Isabelle/HOL.

```
theory logic_bridge
  imports combinators
begin
```

2.1 Custom Type Notation

2.1.1 Basic Types

Classical HOL systems come with a built-in boolean type, for which we introduce convenient notation alias.

```
type_notation bool ("o")
```

The creation of a functional type (starting with a type `'a`) can be seen from two complementary perspectives: Environmentalization (aka. indexation or contextualization) and valuation (e.g. classification, coloring, etc.).

```
type_synonym ('e,'a)Env = "'e ⇒ 'a" ("_-Env'(_')" [1000])
type_synonym ('v,'a)Val = "'a ⇒ 'v" ("_-Val'(_')" [1000])
```

2.1.2 Pairs and Sets

Starting with the boolean type, we immediately obtain endopairs resp. sets via indexation resp. valuation.

type_synonym *('a)EPair = "o-Env('a)" ("EPair'(_')")* — an endopair is encoded as a boolean-index
type_synonym *('a)Set = "o-Val('a)" ("Set'(_')")* — a set is encoded as a boolean-valuation (boolean classifier)

term *"((P :: EPair('a)):: 'a-Val(o)) :: o ⇒ 'a"*
term *"((S :: Set('a)):: 'a-Env(o)) :: 'a ⇒ o"*

Sets of endopairs correspond to (directed) graphs (which are isomorphic to relations via currying).

type_synonym *('a)Graph = "Set(EPair('a))" ("Graph'(_')")*
term *"(G :: Graph('a)) :: (o ⇒ 'a) ⇒ o"*

Spaces (sets of sets) are the playground of mathematicians, so they deserve a special type notation.

type_synonym *('a)Space = "Set(Set('a))" ("Space'(_')")*
term *"(S :: Space('a)) :: ('a ⇒ o) ⇒ o"*

2.1.3 Relations

Valuations can be made binary (useful e.g. for classifying pairs of objects or encoding their "distance").

type_synonym *('v,'a,'b)Val2 = "'a ⇒ 'b ⇒ 'v" ("_-Val$_2$'(_,_')" [1000])*

Binary valuations can also be seen as indexed (unary) valuations.

term *"((F :: 'v-Val$_2$('a,'b)) :: 'a-Env('v-Val('b))) :: 'a ⇒ 'b ⇒ 'v"*

In fact (heterogeneous) relations correspond to o-valued binary functions/valuations.

type_synonym *('a,'b)Rel = "o-Val$_2$('a,'b)" ("Rel'(_,_')")*

They can also be seen as set-valued functions/valuations or as indexed (families of) sets.

term *"(((R :: Rel('a,'b)) :: Set('b)-Val('a)) :: 'a-Env(Set('b))) :: 'a ⇒ 'b ⇒ o"*

Ternary relations are seen as set-valued binary valuations (partial and non-deterministic binary functions).

type_synonym *('a,'b,'c)Rel3 = "Set('c)-Val$_2$('a,'b)" ("Rel$_3$'(_,_,_')")*

They can also be seen as indexed binary relations (e.g. an indexed family of programs or (a group of) agents).

term *"((R::Rel$_3$('a,'b,'c)) :: 'a-Env(Rel('b,'c))) :: 'a ⇒ 'b ⇒ 'c ⇒ o"*

In general, we can encode n+1-ary relations as indexed n-ary relations.

type_synonym *('a,'b,'c,'d)Rel4 = "'a-Env(Rel$_3$('b,'c,'d))" ("Rel$_4$'(_,_,_,_')")*

Convenient notation for the particular case where the relata have all the same type.

type_synonym *('a)ERel = "Rel('a,'a)" ("ERel'(_')")* — (binary) endorelations
type_synonym *('a)ERel$_3$ = "Rel$_3$('a,'a,'a)" ("ERel$_3$'(_')")* — ternary endorelations
type_synonym *('a)ERel$_4$ = "Rel$_4$('a,'a,'a,'a)" ("ERel$_4$'(_')")* — quaternary endorelations

term *"(R :: ERel('a)) :: 'a ⇒ 'a ⇒ o"*
term *"(R :: ERel$_3$('a)) :: 'a ⇒ 'a ⇒ 'a ⇒ o"*
term *"(R :: ERel$_4$('a)) :: 'a ⇒ 'a ⇒ 'a ⇒ 'a ⇒ o"*

2.1.4 Operations

As a convenient mathematical abstraction, we introduce the notion of "operation". In mathematical phraseology, operations are said to "operate" on (one or more) "operands". Operations can be seen as (curried) functions whose arguments have all the same type.

Unary case: (endo)operations just correspond to (endo)functions.

type_synonym *('a,'b)Op1 = "'a ⇒ 'b" ("Op'(_,_')")*
type_synonym *('a)EOp1 = "Op('a,'a)" ("EOp'(_')")* — same as: *'a ⇒ 'a*

Binary case: (endo)bi-operations correspond to curried (endo)bi-functions.

type_synonym *('a,'b)Op2 = "'a ⇒ 'a ⇒ 'b" ("Op$_2$'(_,_')")*
type_synonym *('a)EOp2 = "Op$_2$('a,'a)" ("EOp$_2$'(_')")* — same as: *'a ⇒ ('a ⇒ 'a)*

Arbitrary case: generalized (endo)operations correspond to (endo)functions on sets.

type_synonym *('a,'b)OpG = "Op(Set('a),'b)" ("Op$_G$'(_,_')")*
type_synonym *('a)EOpG = "Op$_G$('a,'a)" ("EOp$_G$'(_')")* — same as: *Set('a) ⇒ 'a*

Convenient type aliases for (endo)operations on sets.

type_synonym *('a,'b)SetOp = "Op(Set('a),Set('b))" ("SetOp'(_,_')")*
type_synonym *('a)SetEOp = "SetOp('a,'a)" ("SetEOp'(_')")* — same as: *Set('a) ⇒ Set('a)*

Binary case: (endo)bi-operations correspond to curried (endo)bi-functions

type_synonym *('a,'b)SetOp2 = "Set('a) ⇒ Set('a) ⇒ Set('b)" ("SetOp$_2$'(_,_')")*
type_synonym *('a)SetEOp2 = "SetOp$_2$('a,'a)" ("SetEOp$_2$'(_')")* — same as: *Set('a) ⇒ Set('a) ⇒ Set('a)*

2.1.5 Products of Boolean Types

Now consider the following equivalent type notations.

term *"((S :: Set(o)) :: EPair(o)) :: o ⇒ o"*
term *"((R :: ERel(o)) :: EOp$_2$(o)) :: o ⇒ (o ⇒ o)"*
term *"(((S :: Space(o)) :: Graph(o)) :: EOp$_G$(o)) :: (o ⇒ o) ⇒ o"*

We can make good sense of them by considering a new type having four inhabitants.

type_synonym *four = "o ⇒ o" ("oo")*
term *"((P :: oo) :: EPair(o)) :: Set(o)"*

Using the new type we can seamlessly define types for (endo)quadruples and 4-valued sets.

type_synonym *('a)EQuad = "oo ⇒ 'a" ("EQuad'(_')")*
type_synonym *('a)Set4 = "'a ⇒ oo" ("Set4'(_')")*

The following two types have each 16 elements (we show a bijection between their elements later on).

type_synonym *sixteen = "o ⇒ oo" ("ooo")* — *4^2 = (2^2)^2*
type_synonym *sixteen' = "oo ⇒ o" ("ooo''")* — *2^4 = 2^(2^2)*

So we can have that the following type notations are in fact identical (not just isomorphic).

term *"(((S :: Set(o)) :: EPair(o)) :: o ⇒ o) :: oo"*
term *"(((((R :: ERel(o)) :: EOp$_2$(o)) :: EPair(oo)) :: Set4(o)) :: o ⇒ o ⇒ o) :: ooo"*
term *"((((((S :: Space(o)) :: Graph(o)) :: EOp$_G$(o)) :: Set(oo)) :: EQuad(o)) :: (o ⇒ o) ⇒ o) :: ooo'"*

We can continue producing types (we stop giving them special notation after the magic number 64).

type_synonym *sixtyfour = "oo ⇒ oo" ("oooo")* — *4^4 = (2^2)^(2^2) = 64*
type_synonym *n256   = "o ⇒ ooo"* — *16^2 = 256*
type_synonym *n65536 = "oo ⇒ ooo"* — *16^4 = 65536*

```
type_synonym n65536' = "ooo ⇒ o" — 2^16 = 65536
type_synonym n4294967296 = "ooo ⇒ oo" — 4^16 = 4294967296
— and so on...
```

Continuations (with result type 'r) take inputs of type 'a

Unary case:

```
type_synonym ('a,'r)Cont1 = "'r-Val(Op('a,'r))" ("Cont'(_,_')") — same as: ('a ⇒ 'r) ⇒ 'r
```

Binary case:

```
type_synonym ('a,'r)Cont2 = "'r-Val(Op₂('a,'r))" ("Cont₂'(_,_')") — same as: ('a ⇒ 'a ⇒ 'r) ⇒ 'r
```

2.2 Custom Term Notation

Convenient combinator-like symbols 𝒬 resp. 𝒟 to be used instead of (=) resp. (≠).

```
notation HOL.eq ("𝒬") and HOL.not_equal ("𝒟")
```

Alternative (more concise) notation for boolean constants.

```
notation HOL.True ("𝒯") and HOL.False ("ℱ")
```

Add (binder) notation for indefinite descriptions (aka. Hilbert's epsilon or choice operator).

```
notation Hilbert_Choice.Eps ("ε") and Hilbert_Choice.Eps (binder "ε" 10)
```

Introduce a convenient "dual" to Hilbert's epsilon operator (adds variable-binding notation).

```
definition Delta ("δ")
  where "δ ≡ λA. ε (λx. ¬A x)"
notation Delta (binder "δ" 10)
```

Add (binder) notation for definite descriptions (incl. binder notation).

```
notation HOL.The ("ι") and HOL.The (binder "ι" 10)
```

We introduce (pedagogically convenient) notation for HOL logical constants.

```
notation HOL.All ("∀")
notation HOL.Ex  ("∃")
abbreviation Empty ("∄")
  where "∄A ≡ ¬∃A"

notation HOL.implies (infixr "→" 25)  — convenient alternative notation
notation HOL.iff (infixr "↔" 25) — convenient alternative notation
```

Add convenient logical connectives.

```
abbreviation(input) seilpmi (infixl "←" 25) — reversed implication
  where "A ← B ≡ B → A"
abbreviation(input) excludes (infixl "↽" 25) — aka. co-implication
  where "A ↽ B ≡ A ∧ ¬B"
abbreviation(input) sedulcxe (infixr "⇀" 25) — aka. dual-implication
  where "A ⇀ B ≡ B ↽ A"
abbreviation(input) xor (infix "⇌" 25) — aka. symmetric difference
  where "A ⇌ B ≡ A ≠ B"
abbreviation(input) nand (infix "↑" 35) — aka. Sheffer stroke
  where "A ↑ B ≡ ¬(A ∧ B)"
abbreviation(input) nor (infix "↓" 30) — aka. Peirce arrow or Quine dagger
  where "A ↓ B ≡ ¬(A ∨ B)"
```

Check relationships

lemma *disj_impl:* "(A ∨ B) = ((A → B) → B)"
lemma *conj_excl:* "(A ∧ B) = ((A ⇀ B) ⇀ B)"
lemma *xor_excl:* "(A ⇌ B) = (A ↼ B) ∨ (A ⇀ B)"
end

3 Logical Connectives using Primitive Equality

Via positiva: equality (notation: 𝒬, infix =) is all you can tell.

theory *connectives_equality*
  imports *logic_bridge*
begin

3.1 Basic Connectives

3.1.1 Verum

Since any function is self-identical, the following serves as definition of verum/true.

lemma *true_defQ:* "𝒯 = 𝒬 𝒬 𝒬"
lemma "𝒯 = (𝒬 = 𝒬)"

3.1.2 Identity (for booleans)

In fact, the identity function (for booleans) is also definable from equality alone.

lemma *id_defQ:* "I = 𝒬 𝒯"
lemma "I = 𝒬 (𝒬 𝒬 𝒬)"

3.1.3 Falsum

Asserting that two different functions are equal is a good way to encode falsum.

lemma *false_defQ:* "ℱ = 𝒬 I (K 𝒯)"
lemma "ℱ = (I = K 𝒯)"
lemma "ℱ = 𝒬(𝒬(𝒬 𝒬 𝒬))(K(𝒬 𝒬 𝒬))"

3.1.4 Negation

We can negate a proposition P by asserting that 'P is absurd' (i.e. P is equal to falsum).

lemma *not_defQ:* "(¬) = 𝒬 ℱ"
lemma "(¬) = (λP. P = ℱ)"
lemma "(¬) = 𝒬(𝒬(𝒬(𝒬 𝒬 𝒬))(K(𝒬 𝒬 𝒬)))"

3.1.5 Disequality

Using negation we can define disequality for any type (not only boolean).

lemma *diseq_defQ:* "𝒟 = (¬) $\circ_2$ 𝒬"
lemma "𝒟 = (λA B. ¬(A = B))"

named_theorems *eq_defs*
declare *true_defQ [eq_defs] id_defQ [eq_defs]*
        false_defQ [eq_defs] not_defQ [eq_defs] diseq_defQ [eq_defs]

end

4 Logical Connectives using Primitive Disequality

Via negativa: disequality (notation: 𝒟, infix ≠) is all you can tell.

```
theory connectives_disequality
  imports logic_bridge
begin
```

4.1 Basic Connectives

4.1.1 Falsum

Since no function is non-self-identical, the following serves as definition of falsum/false.

```
lemma false_defD: "𝓕 = 𝓓 𝓓 𝓓"
lemma "𝓕 = (𝓓 ≠ 𝓓)"
```

4.1.2 Identity (for booleans)

In fact, the identity function (for booleans) is also definable from disequality alone.

```
lemma id_defD: "I = 𝓓 𝓕"
lemma "I = 𝓓 (𝓓 𝓓 𝓓)"
```

4.1.3 Verum

Asserting that two different functions are different is a good way to encode verum.

```
lemma true_defD: "𝓣 = 𝓓 I (K 𝓕)"
lemma "𝓣 = (I ≠ K 𝓕)"
lemma "𝓣 = 𝓓(𝓓(𝓓 𝓓 𝓓))(K(𝓓 𝓓 𝓓))"
```

4.1.4 Negation

We can negate a proposition P by asserting that "P is not true" (i.e. P is not equal to verum).

```
lemma not_defD: "(¬) = 𝓓 𝓣"
lemma "(¬) = (λP. P ≠ 𝓣)"
lemma "(¬) = 𝓓(𝓓(𝓓(𝓓 𝓓 𝓓))(K(𝓓 𝓓 𝓓)))"
```

4.1.5 Equality

Using negation we can define equality for any type (not only boolean).

```
lemma eq_defD: "𝓠 = (¬) ∘₂ 𝓓"
lemma "𝓠 = (λA B. ¬(A ≠ B))"

named_theorems diseq_defs
declare false_defD [diseq_defs] id_defD [diseq_defs]
        true_defD [diseq_defs] not_defD [diseq_defs] eq_defD [diseq_defs]

end
```

4.2 Defined connectives

We illustrate how the logical connectives could have been defined in terms of equality resp. disequality. (We actually work with them as they are provided by Isabelle/HOL (with the notational changes).

```
theory connectives
imports connectives_equality      — via positiva
        connectives_disequality — via negativa
begin
```

4.2.1 Biconditional (aka. iff, double-implication)

Biconditional is just equality (for booleans).

```
lemma iff_def: "(↔) = 𝒬"
lemma "(↔) = (λA B. A = B)"
```

4.2.2 XOR (aka. symmetric difference)

XOR is just disequality (for booleans).

```
lemma xor_def: "(⇌) = 𝒟"
lemma "(⇌) = (λA B. A ≠ B)"
```

4.2.3 Conjunction, disjunction, and (co)implication

We can encode them by their truth tables.

```
lemma and_def:  "(∧) = B₂₀ (𝒬::ERel(Set(ERel(o)))) V (V 𝒯 𝒯)"
lemma or_def:   "(∨) = B₂₀ (𝒟::ERel(Set(ERel(o)))) V (V ℱ ℱ)"
lemma impl_def: "(→) = B₂₀ (𝒟::ERel(Set(ERel(o)))) V (V 𝒯 ℱ)"
lemma excl_def: "(↼) = B₂₀ (𝒬::ERel(Set(ERel(o)))) V (V 𝒯 ℱ)"

lemma "(∧)  = (λA B. (λr::ERel(o). r A B) = (λr. r 𝒯 𝒯))"
lemma "(∨)  = (λA B. (λr::ERel(o). r A B) ≠ (λr. r ℱ ℱ))"
lemma "(→) = (λA B. (λr::ERel(o). r A B) ≠ (λr. r 𝒯 ℱ))"
lemma "(↼) = (λA B. (λr::ERel(o). r A B) = (λr. r 𝒯 ℱ))"
```

We add to both the equality and disequality definition bags:

```
declare iff_def [eq_defs] xor_def [eq_defs]
        and_def [eq_defs] or_def [eq_defs] impl_def [eq_defs] excl_def [eq_defs]
declare iff_def [diseq_defs] xor_def [diseq_defs]
        and_def [diseq_defs] or_def [diseq_defs] impl_def [diseq_defs] excl_def [diseq_defs]
```

4.3 Quantifiers and co.

Quantifiers can also be defined using equality/disequality.

```
lemma ex_defQ:  "∃ = 𝒟 (K ℱ)"
lemma all_defQ: "∀ = 𝒬 (K 𝒯)"

declare ex_defQ [eq_defs] all_defQ [eq_defs]

lemma "∃φ = (φ ≠ (λx. ℱ))"
lemma "∀φ = (φ = (λx. 𝒯))"
```

Moreover, they are also definable using indefinite descriptions ε resp. δ and the $\mathbf{\Pi}_{111}$/O combinator.

```
lemma ex_defEps:  "∃ = O ε"
lemma all_defEps: "∀ = O δ"

lemma "∃φ = φ(ε x.  φ x)"
lemma "∀φ = φ(ε x. ¬φ x)"
```

We introduce convenient arity-extended versions of the quantifiers.

```
abbreviation(input) All2 ("∀²")
  where "∀²R ≡ ∀a b. R a b"
abbreviation(input) All3 ("∀³")
  where "∀³R ≡ ∀a b c. R a b c"
— ... and so on
abbreviation(input) Ex2 ("∃²")
  where "∃²R ≡ ∃a b. R a b"
```

```
abbreviation(input) Ex3 ("∃^3")
  where "∃^3R ≡ ∃a b c. R a b c"
```
— ... and so on

```
abbreviation NotEx2 ("∄^2")
  where "∄^2R ≡ ¬∃^2R"
abbreviation NotEx3 ("∄^3")
  where "∄^3R ≡ ¬∃^3R"
```
— ... and so on

4.4 Definite description (for booleans)

Henkin (1963) also defines `ι::(o⇒o)⇒o` via equality, namely as: `𝒬 I`. Note, however, that in Isabelle/HOL the term `ι::(o⇒o)⇒o` is not introduced as a definition. Instead, `ι::(o⇒o)⇒o` is an instance of `ι::('a⇒o)⇒'a`, which is an axiomatized (polymorphic) constant.

```
proposition "ι = 𝒬 I" nitpick
```
— countermodel found

```
end
```

5 Endopairs

```
theory endopairs
  imports logic_bridge
begin

named_theorems endopair_defs and endopair_simps
```

5.1 Definitions

Term constructor: making an endopair out of two given objects.

```
definition mkEndopair::"'a ⇒ 'a ⇒ EPair('a)" ("<_,_>")
  where "mkEndopair ≡ L If"

declare mkEndopair_def[endopair_defs]
```

With syntactic sugar the above definition looks like:

```
lemma "<x,y> = (λb. if b then x else y)"
```

Under the hood, the term constructor `mkEndopair` is built in terms of definite descriptions.

```
lemma mkEndopair_def2: "<x,y> = (λb. ι z. (b → z = x) ∧ (¬b → z = y))"
```

Incidentally, (endo)pairs of booleans have an alternative, simpler representation.

```
lemma mkEndopair_bool_simp: "<x,y> = (λb. (b ∧ x) ∨ (¬b ∧ y))"
```

Componentwise equality comparison between endopairs (added as convenient simplification rule).

```
lemma mkEndopair_equ_simp: "(<x_1,x_2> = <y_1,y_2>) = (x_1 = y_1 ∧ x_2 = y_2)"
```

We conveniently add the previous lemmata as a simplification rules.

```
declare mkEndopair_bool_simp[endopair_simps] and mkEndopair_equ_simp[endopair_simps]
```

Now, observe that:

```
lemma "<x,y> 𝒯 = x"
lemma "<x,y> ℱ = y"
```

This motivates the introduction of the following projection/extraction functions.

definition *proj1::"EPair('a)* $\Rightarrow$ *'a" ("*π_1*")*
  where "$\pi_1 \equiv \texttt{T}\ \mathcal{T}$"
definition *proj2::"EPair('a)* $\Rightarrow$ *'a" ("*π_2*")*
  where "$\pi_2 \equiv \texttt{T}\ \mathcal{F}$"

declare *proj1_def[endopair_defs] proj2_def[endopair_defs]*

lemma "$\pi_1 = (\lambda P.\ P\ \mathcal{T})$"

lemma "$\pi_2 = (\lambda P.\ P\ \mathcal{F})$"

The following lemmata (aka. "product laws") verify that the previous definitions work as intended.

lemma *proj1_simp:* "$\pi_1 <x,y> = x$"
lemma *proj2_simp:* "$\pi_2 <x,y> = y$"
lemma *mkEndopair_simp:* "$<\pi_1\ P,\ \pi_2\ P> = P$"

We conveniently add them as simplification rules.

declare *proj1_simp[endopair_simps] proj2_simp[endopair_simps] mkEndopair_simp[endopair_simps]*

Let's now add a useful "swap" (endo)operation on endopairs.

definition *swap::"EOp(EPair('a))"*
  where "$swap \equiv \texttt{C}\ \texttt{B}\ (\neg)$"

declare *swap_def[endopair_defs]*

lemma "$swap\ p = p \circ (\neg)$"
lemma "$swap\ p = (\lambda b.\ p\ (\neg b))$"

We conveniently prove and add some useful simplification rules.

lemma *swap_simp1:* "$swap <a,b> = <b,a>$"
lemma *swap_simp2:* "$<\pi_2\ p,\ \pi_1\ p> = swap\ p$"

declare *swap_simp1[endopair_simps] swap_simp2[endopair_simps]*

5.2 Currying

The morphisms that convert between unary operations on endopairs and (curried) binary operations.

definition *curry::"Op(EPair('a),'b)* $\Rightarrow$ *Op*$_2$*('a,'b)" ("*$\lfloor\_\rfloor$*")*
  where "$curry \equiv \texttt{C}\ \texttt{B}_2\ mkEndopair$"
definition *uncurry::"Op*$_2$*('a,'b)* $\Rightarrow$ *Op(EPair('a),'b)" ("*$\lceil\_\rceil$*")*
  where "$uncurry \equiv \texttt{L}\ \mathbf{\Phi}_{21}\ \pi_1\ \pi_2$"

declare *curry_def[endopair_defs] uncurry_def[endopair_defs]*

Some sanity checks:

lemma "$curry\ f = \texttt{B}_2\ f\ mkEndopair$"
lemma "$curry\ f = (\lambda x\ y.\ f <x,y>)$"
lemma "$uncurry\ f = \mathbf{\Phi}_{21}\ f\ \pi_1\ \pi_2$"
lemma "$uncurry\ f = (\lambda P.\ f\ (\pi_1\ P)\ (\pi_2\ P))$"

Both morphisms constitute an isomorphism (we add them as simplification rules too)

lemma *curry_simp1:* "$\lfloor\lceil f \rceil\rfloor$ = f"
lemma *curry_simp2:* "$\lceil\lfloor f \rfloor\rceil$ = f"

declare *curry_simp1[endopair_simps] curry_simp2[endopair_simps]*

end

6 Functions and Sets

We introduce several convenient definitions and lemmata for working with functions and sets.

theory *func_sets*
imports *connectives*
begin

named_theorems *func_defs*

6.1 Basic Functional Notions

6.1.1 Monoid Structure

Functions feature a monoidal structure. The identity function is a nullary operation (i.e. a "constant"). It corresponds to the I combinator. Function composition is the main binary operation between functions and corresponds to the B combinator.

lemma "$f \circ g \circ h$ = (λx. f (g (h x)))"
lemma "f ; g ; h = (λx. h(g (f x)))"

Composition and identity satisfy the monoid conditions.

lemma "$(f \circ g) \circ h = f \circ (g \circ h)$"
lemma "$\mathtt{I} \circ f = f$"
lemma "$f \circ \mathtt{I} = f$"

6.1.2 Fixed-Points

The set of pre- resp. post-fixed-points of an endofunction *f* wrt an endorelation *R*, are those points sent by *f* backwards resp. forward wrt *R*. Note that if *R* is symmetric then both notions coincide.

definition *preFixedPoint::"ERel('a) $\Rightarrow$ EOp('a) $\Rightarrow$ Set('a)" ("_-preFP")*
  where "*preFixedPoint* $\equiv$ $\mathbf{\Sigma}$"
definition *postFixedPoint::"ERel('a) $\Rightarrow$ EOp('a) $\Rightarrow$ Set('a)" ("_-postFP")*
  where "*postFixedPoint* $\equiv$ S"

declare *preFixedPoint_def[func_defs] postFixedPoint_def[func_defs]*

lemma "R-preFP f = (λA. R (f A) A)"
lemma "R-postFP f = (λA. R A (f A))"

The set of weak pre-/post-fixed-points of endooperation wrt. an endorelation.

definition *weakPreFixedPoint::"ERel('a) $\Rightarrow$ EOp('a) $\Rightarrow$ Set('a)" ("_-wPreFP")*
  where "*weakPreFixedPoint* $\equiv$ L $\mathbf{\Phi}_{22}$ (W B) A"
definition *weakPostFixedPoint::"ERel('a) $\Rightarrow$ EOp('a) $\Rightarrow$ Set('a)" ("_-wPostFP")*
  where "*weakPostFixedPoint* $\equiv$ L $\mathbf{\Phi}_{22}$ A (W B)"

declare *weakPreFixedPoint_def[func_defs] weakPostFixedPoint_def[func_defs]*

lemma "R-wPreFP φ = (λA. R (φ(φ A)) (φ A))"
lemma "R-wPostFP φ = (λA. R (φ A) (φ (φ A)))"

The (non-)fixed-points of an endofunction are just the pre/post-fixed points wrt (dis)equality.

```
definition fixedPoint::"('a ⇒ 'a) ⇒ Set('a)" ("FP")
  where "FP  ≡ 𝒬-postFP"
definition nonFixedPoint::"('a ⇒ 'a) ⇒ Set('a)" ("nFP")
  where "nFP ≡ 𝒟-postFP"

declare fixedPoint_def[func_defs] nonFixedPoint_def[func_defs]

lemma "FP f x = (x = f x)"
lemma "nFP f x = (x ≠ f x)"

lemma fixedPoint_defT: "FP = 𝒬-preFP"
lemma nonFixedPoint_defT: "nFP = 𝒟-preFP"
```

An endooperation can be said to be (weakly) expansive resp. contractive wrt an endorelation when all of its points are (weak) pre-fixed-points resp. (weak) post-fixed-points.

```
definition expansive::"ERel('a) ⇒ Set(EOp('a))" ("_-EXPN")
  where "R-EXPN ≡ ∀ ∘ R-postFP"
definition contractive::"ERel('a) ⇒ Set(EOp('a))" ("_-CNTR")
  where "R-CNTR ≡ ∀ ∘ R-preFP"
definition weaklyExpansive::"ERel('a) ⇒ Set(EOp('a))" ("_-wEXPN")
  where "R-wEXPN ≡ ∀ ∘ R-wPostFP"
definition weaklyContractive::"ERel('a) ⇒ Set(EOp('a))" ("_-wCNTR")
  where "R-wCNTR ≡ ∀ ∘ R-wPreFP"

declare expansive_def[func_defs] contractive_def[func_defs]
        weaklyExpansive_def[func_defs] weaklyContractive_def[func_defs]

lemma "R-EXPN f = (∀A. R A (f A))"
lemma "R-CNTR f = (∀A. R (f A) A)"
lemma "R-wEXPN f = (∀A. R (f A) (f (f A)))"
lemma "R-wCNTR f = (∀A. R (f (f A)) (f A))"
```

6.1.3 Type-lifting - General Case: Environment (aka. Reader) Monad

We can conceive of functional types of the form `'a ⇒ 'b` as arising via an "environmentalization", or "indexation" of the type `'b` by the type `'a`, i.e. as `'a-Env('b)` using our type notation. This type constructor comes with a monad structure (and is thus an applicative and a functor too).

```
abbreviation(input) unit_env::"'a ⇒ 'e-Env('a)"
  where "unit_env  ≡ K"
abbreviation(input) fmap_env::"('a ⇒ 'b) ⇒ 'e-Env('a) ⇒ 'e-Env('b)"
  where "fmap_env  ≡ B"
abbreviation(input) join_env::"'e-Env('e-Env('a)) ⇒ 'e-Env('a)"
  where "join_env  ≡ W"
abbreviation(input) ap_env::"'e-Env('a ⇒ 'b) ⇒ 'e-Env('a) ⇒ 'e-Env('b)"
  where "ap_env    ≡ S"
abbreviation(input) rbind_env::"('a ⇒ 'e-Env('b)) ⇒ 'e-Env('a) ⇒ 'e-Env('b)"
  where "rbind_env ≡ Σ" — reversed-bind
```

We define the customary bind operation as "flipped" rbind (which seems more intuitive).

```
abbreviation(input) bind_env::"'e-Env('a) ⇒ ('a ⇒ 'e-Env('b)) ⇒ 'e-Env('b)"
  where "bind_env ≡ C rbind_env"
```

But we could have also given it a direct alternative definition.

```
lemma "bind_env = W ∘₂ (C B)"
```

Some properties of monads in general

```
lemma "rbind_env = join_env ∘₂ fmap_env"
lemma "join_env = rbind_env I"
```

Some properties of this particular monad

lemma *"ap_env = rbind_env ∘ C"*

The so-called "monad laws". They guarantee that monad-related term operations compose reliably.

abbreviation *(input) "monadLaw1 unit bind ≡ ∀f a. (bind (unit a) f) = (f a)"* — left identity
abbreviation *(input) "monadLaw2 unit bind ≡ ∀A. (bind A unit) = A"* — right identity
abbreviation *(input) "monadLaw3 bind ≡ ∀A f g. (bind A (λa. bind (f a) g)) = bind (bind A f) g"* — associativity

Verifies compliance with the monad laws.

lemma *"monadLaw1 unit_env bind_env"*
lemma *"monadLaw2 unit_env bind_env"*
lemma *"monadLaw3 bind_env"*

6.1.4 Type-lifting - Digression: On Higher Arities

Note that $\mathbf{\Phi}_{mn}$ combinators can be used to index (or "environmentalize") a given m-ary function n-times.

term *"($\mathbf{\Phi}_{01}$ (f::'a)) :: 'e-Env('a)"*
term *"($\mathbf{\Phi}_{11}$ (f::'a ⇒ 'b)) :: 'e-Env('a) ⇒ 'e-Env('b)"*
term *"($\mathbf{\Phi}_{12}$ (f::'a ⇒ 'b)) :: 'e$_2$-Env('e$_1$-Env('a)) ⇒ 'e$_2$-Env('e$_1$-Env('b))"*
— ...and so on
term *"($\mathbf{\Phi}_{21}$ (g::'a ⇒ 'b ⇒ 'c)) :: 'e-Env('a) ⇒ 'e-Env('b) ⇒ 'e-Env('c)"*
term *"($\mathbf{\Phi}_{22}$ (g::'a ⇒ 'b ⇒ 'c)) :: 'e$_2$-Env('e$_1$-Env('a)) ⇒ 'e$_2$-Env('e$_1$-Env('b)) ⇒ 'e$_2$-Env('e$_1$-Env(*
— ...and so on

Hence the $\mathbf{\Phi}_{mn}$ combinators can play the role of (n-times iterated) functorial "lifters".

lemma *"(unit_env::'a ⇒ 'e-Env('a)) = $\mathbf{\Phi}_{01}$"*
lemma *"(fmap_env::('a ⇒ 'b) ⇒ ('e-Env('a) ⇒ 'e-Env('b))) = $\mathbf{\Phi}_{11}$"*
abbreviation *(input) fmap2_env::"('a ⇒ 'b ⇒ 'c) ⇒ ('e-Env('a) ⇒ 'e-Env('b) ⇒ 'e-Env('c))"*
  where *"fmap2_env ≡ $\mathbf{\Phi}_{21}$"* — cf. Haskell's `liftA2`
— ...and so on

In the same spirit, we can employ the combinator families $\mathtt{S}_{mn}$ resp. $\mathbf{\Sigma}_{mn}$ as (n-times iterated) m-ary applicative resp. monadic "lifters".

abbreviation *(input) ap2_env::"'e-Env('a ⇒ 'b ⇒ 'c) ⇒ ('e-Env('a) ⇒ 'e-Env('b) ⇒ 'e-Env('c))"*
  where *"ap2_env ≡ $\mathtt{S}_{21}$"*
abbreviation *(input) rbind2_env::"('a ⇒ 'b ⇒ 'e-Env('c)) ⇒ ('e-Env('a) ⇒ 'e-Env('b) ⇒ 'e-Env('c)*
  where *"rbind2_env ≡ $\mathbf{\Sigma}_{21}$"*
— ...and so on

6.1.5 Type-lifting - Base Case: Identity Monad

Finally, we consider the (degenerate) base case arising from an identity type constructor

abbreviation *(input) unit_id::"'a ⇒ 'a"*
  where *"unit_id ≡ I"*
abbreviation *(input) fmap_id::"('a ⇒ 'b) ⇒ ('a ⇒ 'b)"*
  where *"fmap_id ≡ A"*
abbreviation *(input) fmap2_id::"('a ⇒ 'a ⇒ 'b) ⇒ ('a ⇒ 'a ⇒ 'b)"*
  where *"fmap2_id ≡ $\mathtt{A}_2$"*
abbreviation *(input) join_id::"'a ⇒ 'a"*
  where *"join_id ≡ I"*
abbreviation *(input) ap_id::"('a ⇒ 'b) ⇒ ('a ⇒ 'b)"*
  where *"ap_id ≡ A"*
abbreviation *(input) rbind_id::"('a ⇒ 'b) ⇒ ('a ⇒ 'b)"*
  where *"rbind_id ≡ A"*

abbreviation *(input) bind_id::"'a ⇒ ('a ⇒ 'b) ⇒ 'b"*
where *"bind_id ≡ T"*

lemma *"monadLaw1 unit_id bind_id"*
lemma *"monadLaw2 unit_id bind_id"*
lemma *"monadLaw3 bind_id"*

6.1.6 Type-lifting - Relations

Relations can be seen (and thus type-lifted) from two equivalent perspectives:

1. As unary functions (with set codomain), or equivalently, as indexed families of sets.
2. As binary functions (with a boolean codomain).

term *"(R :: Rel('a,'b)) :: 'a-Env(Set('b))"*
term *"(R :: Rel('a,'b)) :: 'a ⇒ 'b ⇒ o"*

Note that when "lifting" relations as binary functions (via $\mathbf{\Phi}_{21}$) what we obtain is not quite a relation.

term *"$\mathbf{\Phi}_{21}$ (R :: Rel('a,'b)) :: 'e-Env('a) ⇒ 'e-Env('b) ⇒ Set('e)"*

We introduce two convenient ways to lift a given relation to obtain its "indexed" counterpart.

definition *relLiftEx :: "Rel('a,'b) ⇒ Rel('c-Env('a),'c-Env('b))" ("$\mathbf{\Phi}_\exists$")* — existential lifting
where *"$\mathbf{\Phi}_\exists$ ≡ ∃ $\circ_3$ $\mathbf{\Phi}_{21}$"*
definition *relLiftAll :: "Rel('a,'b) ⇒ Rel('c-Env('a),'c-Env('b))" ("$\mathbf{\Phi}_\forall$")* — universal lifting
where *"$\mathbf{\Phi}_\forall$ ≡ ∀ $\circ_3$ $\mathbf{\Phi}_{21}$"*

declare *relLiftEx_def[func_defs] relLiftAll_def[func_defs]*

6.2 Basic Set Notions

6.2.1 Set-Operations

Note that sets of As can be faithfully encoded as A-indexed booleans (aka. "characteristic functions").

term *"(S :: Set('a)) :: 'a-Env(o)"*

Thus the usual set operations arise via "indexation" of HOL's boolean connectives (via $\mathbf{\Phi}_{m1}$ combinators). This explains, among others, why sets come with a Boolean algebra structure (cf. Stone representation).

definition *universe::"Set('a)" ("$\mathfrak{U}$")*
where *"$\mathfrak{U}$ ≡ $\mathbf{\Phi}_{01}$ $\mathcal{T}$"* — the universal set: the nullary connective/constant $\mathcal{T}$ lifted once
definition *emptyset::"Set('a)" ("∅")*
where *"∅ ≡ $\mathbf{\Phi}_{01}$ $\mathcal{F}$"* — the empty set: the nullary connective/constant $\mathcal{F}$ lifted once
definition *compl::"EOp(Set('a))" ("−")*
where ‹− ≡ $\mathbf{\Phi}_{11}$(¬)›

set complement: the unary ¬ connective lifted once

definition *inter::"EOp$_2$(Set('a))" (**infixr** "∩" 54)*
where *"(∩) ≡ $\mathbf{\Phi}_{21}$(∧)"* — set intersection: the binary ∧ connective lifted once
definition *union::"EOp$_2$(Set('a))" (**infixr** "∪" 53)*
where *"(∪) ≡ $\mathbf{\Phi}_{21}$(∨)"* — set union
definition *diff::"EOp$_2$(Set('a))" (**infixl** "\" 51)*
where *"(\) ≡ $\mathbf{\Phi}_{21}$(↼)"* — set difference
definition *impl::"EOp$_2$(Set('a))" (**infixr** "⇒" 51)*
where *"(⇒) ≡ $\mathbf{\Phi}_{21}$(→)"* — set implication

definition *dimpl::"EOp$_2$(Set('a))"* **(infix** *"⇔" 51)*
  where *"(⇔) ≡ $\mathbf{\Phi}_{21}$(↔)"* — set double-implication
definition *sdiff::"EOp$_2$(Set('a))"* **(infix** *"△" 51)*
  where *"(△) ≡ $\mathbf{\Phi}_{21}$(⇌)"* — set symmetric-difference (aka. xor)

Reversed implication as convenient syntactic sugar.

abbreviation *(input) lpmi::"EOp$_2$(Set('a))"* **(infixl** *"⇐" 51)*
  where *"A ⇐ B ≡ B ⇒ A"*

declare *universe_def[func_defs] emptyset_def[func_defs]*
  compl_def[func_defs] inter_def[func_defs] union_def[func_defs]
  impl_def[func_defs] dimpl_def[func_defs] diff_def[func_defs] sdiff_def[func_defs]

Double-check point-based definitions.

lemma *"𝔘 = (λx. 𝒯)"*
lemma *"∅ = (λx. ℱ)"*
lemma *"−A = (λx. ¬A x)"*
lemma *"A ∩ B = (λx. A x ∧ B x)"*
lemma *"A ∪ B = (λx. A x ∨ B x)"*
lemma *"A \ B = (λx. A x ↼ B x)"*
lemma *"A ⇒ B = (λx. A x → B x)"*
lemma *"A ⇐ B = (λx. A x ← B x)"*
lemma *"A ⇔ B = (λx. A x ↔ B x)"*
lemma *"A △ B = (λx. A x ⇌ B x)"*

Double-check some well known properties.

lemma *compl_involutive: "−(−S) = S"*
lemma *compl_deMorgan1: "−(−A ∪ −B) = (A ∩ B)"*
lemma *compl_deMorgan2: "−(−A ∩ −B) = (A ∪ B)"*
lemma *compl_fixedpoint: "nFP = − ∘ FP"*
lemma *"nFP f = −(FP f)"*

6.2.2 Dual-composition of Unary Set-Operations

Clearly, functional composition can be seamlessly applied to set-operations too.

lemma **fixes** *F::"Set('b) ⇒ Set('c)"* **and** *G::"Set('a) ⇒ Set('b)"*
  shows *"F ∘ G = (λx. F (G x))"*

Moreover, we can conveniently introduce a dual for the (functional) composition of set-operations.

definition *compDual::"SetOp('a,'b) ⇒ SetOp('c,'a) ⇒ SetOp('c,'b)"* **(infixl** *"·" 55)*
  where *"(·) ≡ λf g. λx. f (−(g x))"*
abbreviation *(input) compDual_t* **(infixr** *":" 55)*
  where *"f : g ≡ g · f"*

declare *compDual_def[func_defs]*

lemma *compDuality1: "(f · g) = − ∘ ((− ∘ f) ∘ (− ∘ g))"*
lemma *compDuality2: "(f · g) = (f ∘ (− ∘ g))"*
lemma *compDuality3: "(f ∘ g) = (f · (− ∘ g))"*

6.2.3 Set Orderings

In the previous section we applied a kind of "functional lifting" to the boolean HOL operations in order to encode the corresponding operations on sets. Here we encode sets' (lattice) order structure via a "relational lifting" of the ordering of HOL's truth-values.

We start by noting that HOL's binary boolean operations can also be seen as (endo)relations.

term *"(∧) :: ERel(o)"*

term `"(∨) :: ERel(o)"`
term `"(→) :: ERel(o)"` — the customary ordering on truth-values (where $\mathcal{F} \to \mathcal{T}$)

The algebra of sets is thus naturally ordered via the subset endorelation (via 'relational lifting').

definition `subset::"ERel(Set('a))"` **(infixr** `"⊆" 51)`
  where `"(⊆) ≡ Φ∀ (→)"`

declare `subset_def[func_defs]`

lemma `"A ⊆ B = (∀x. A x → B x)"`
lemma `"A ⊆ B = ∀(A ⇒ B)"`

lemma `subset_setdef: "(⊆) = ∀ ∘₂ (⇒)"`

abbreviation `(input) superset::"ERel(Set('a))"` **(infixr** `"⊇" 51)`
  where `"B ⊇ A ≡ A ⊆ B"`

The powerset operation corresponds in fact to (partial application of) superset relation.

abbreviation `(input) powerset::"Set('a) ⇒ Set(Set('a))" ("℘")`
  where `"℘ ≡ (⊇)"`

lemma `"℘A = (λB. B ⊆ A)"`

Alternative characterizations of the sub/super-set orderings in terms of fixed-points.

lemma `subset_defFP: "(⊆) = FP ∘ (∪)"`
lemma `superset_defFP: "(⊇) = FP ∘ (∩)"`
lemma `"(A ⊆ B) = (B = A ∪ B)"`
lemma `"(B ⊇ A) = (A = B ∩ A)"`

Subset is antisymmetric.

lemma `subset_antisym: "R ⊆ T ⟹ R ⊇ T ⟹ R = T"`

In the same spirit, we conveniently provide the following related endorelations:

Two sets are said to "overlap" (or "intersect") if their intersection is non-empty.

definition `overlap::"ERel(Set('a))"` **(infix** `"⊓" 52)`
  where `"(⊓) ≡ Φ∃ (∧)"`

dually, two sets form a "cover" if every element belongs to one or the other.

definition `cover::"ERel(Set('a))"` **(infix** `"⊔" 53)`
  where `"(⊔) ≡ Φ∀ (∨)"`

declare `overlap_def[func_defs] cover_def[func_defs]`

Convenient notation: Two sets are said to be "incompatible" if they don't overlap.

abbreviation `(input) incompat::"ERel(Set('a))"` **(infix** `"⊥" 52)`
  where `"(⊥) ≡ (¬) ∘₂ (⊓)"`

lemma `cover_setdef: "(⊔) = ∀ ∘₂ (∪)"`
lemma `overlap_setdef: "(⊓) = ∃ ∘₂ (∩)"`
lemma `"A ⊔ B = ∀(A ∪ B)"`
lemma `"A ⊓ B = ∃(A ∩ B)"`
lemma `"A ⊥ B = ∄(A ∩ B)"`

Subset, overlap and cover are interrelated as expected.

lemma `"A ⊆ B = −A ⊔ B"`
lemma `"A ⊆ B = A ⊥ −B"`
lemma `"¬(A ⊆ B) = A ⊓ −B"`
lemma `"¬(A ⊆ B) = A ⊓ −B"`

lemma *"A ⊔ B = −A ⊆ B"*
lemma *"A ⊓ B = (¬(A ⊆ −B))"*
lemma *"A ⊥ B = A ⊆ −B"*

6.2.4 Constructing Sets

abbreviation *(input) insert :: "'a ⇒ Set('a) ⇒ Set('a)"*
  where *"insert a S ≡ 𝒬 a ∪ S"*
abbreviation *(input) remove :: "'a ⇒ Set('a) ⇒ Set('a)"*
  where *"remove a S ≡ 𝒟 a ∩ S"*

The previous functions in terms of combinators.

lemma *"insert = C (B_{10} (∪) 𝒬)"*
lemma *"remove = C (B_{10} (∩) 𝒟)"*

syntax
  "_finiteSet" :: "args ⇒ Set('a)"   ("{(_)}")
  "_finiteCoset" :: "args ⇒ Set('a)" ("⦃(_)⦄")
translations
  "{x, xs}" ⇌ "CONST insert x (_finiteSet xs)"
  "⦃x, xs⦄" ⇌ "CONST remove x (_finiteCoset xs)"
  "{x}" ⇀ "𝒬 x" — aka. "singleton"
  "⦃x⦄" ⇀ "𝒟 x" — aka. "cosingleton")

Some syntax checks.

lemma *"{a} = 𝒬 a"*
lemma *"{a,b} = {a} ∪ {b}"*
lemma *"{a,b,c} = {a} ∪ {b,c}"*
lemma *"{a,b,c} = {a} ∪ {b} ∪ {c}"*
lemma *"⦃a⦄ = 𝒟 a"*
lemma *"⦃a,b⦄ = ⦃a⦄ ∩ ⦃b⦄"*
lemma *"⦃a,b,c⦄ = ⦃a⦄ ∩ ⦃b,c⦄"*
lemma *"⦃a,b,c⦄ = ⦃a⦄ ∩ ⦃b⦄ ∩ ⦃c⦄"*
lemma *"⦃{a,b,c}, {d,e}⦄ = ⦃{a} ∪ {b} ∪ {c}⦄ ∩ ⦃{d} ∪ {e}⦄"*

Sets and cosets are related via set-complement as expected.

lemma *"⦃a⦄ = −{a}"*
lemma *"⦃a,b⦄ = −{a,b}"*
lemma *"⦃a,b,c⦄ = −{a,b,c}"*

HOL quantifiers can be seen as sets of sets (or equivalently as "properties" of sets).

term *"∀::Set(Set('a))"* — *∀A* means that the set A contains all elements
term *"∃::Set(Set('a))"* — *∃A* means that A contains at least one element, i.e. A is nonempty
term *"∄::Set(Set('a))"* — *∃A* means that A is empty

We conveniently add a couple more.

definition *unique::"Set(Set('a))"*
  where *‹unique A ≡ ∀x y. A x ∧ A y → x = y›* — A contains at most one element (it may be empty)
definition *singleton::"Set(Set('a))" ("∃!")*
  where *‹∃!A ≡ ∃x. A x ∧ (∀y. A y → x = y)›* — A contains exactly one element

declare *unique_def[func_defs] singleton_def[func_defs]*

6.2.5 Infinitary Set-Operations

Union and intersection can be generalized to operate on arbitrary sets of sets (aka. "infinitary" operations).

definition *biginter::"EOp$_G$(Set('a))" ("$\bigcap$")*
  where "$\bigcap$ ≡ ∀ ∘$_2$ (B$_{11}$ (⇒) I T)"
definition *bigunion::"EOp$_G$(Set('a))" ("$\bigcup$")*
  where "$\bigcup$ ≡ ∃ ∘$_2$ (B$_{11}$ (∩) I T)"

lemma "$\bigcap S\ x = (\forall A.\ S\ A \rightarrow A\ x)$"
lemma "$\bigcup S\ x = (\exists A.\ S\ A \wedge A\ x)$"

declare *biginter_def[func_defs] bigunion_def[func_defs]*

We say of a set of sets that it "overlaps" (or "intersects") if there exists a shared element.

abbreviation *(input) bigoverlap::"Set(Set(Set('a)))" ("$\bigsqcap$")*
  where "$\bigsqcap$ ≡ ∃ ∘ $\bigcap$"

Dually, a set of sets forms a "cover" if every element is contained in at least one of the sets.

abbreviation *(input) bigcover::"Set(Set(Set('a)))" ("$\bigsqcup$")*
  where "$\bigsqcup$ ≡ ∀ ∘ $\bigcup$"

lemma "$\bigsqcap S = \exists (\bigcap S)$"
lemma "$\bigsqcup S = \forall (\bigcup S)$"

6.3 Function Transformations

6.3.1 Inverse and Range

The inverse of a function *f* is the relation that assigns to each object *b* in its codomain the set of elements in its domain mapped to *b* (i.e. the preimage of *b* under *f*).

definition *inverse::"('a ⇒ 'b) ⇒ Rel('b,'a)"*
  where "*inverse* ≡ B$_{10}$ $\mathcal{Q}$"

lemma "*inverse f b* = (λa. f a = b)"

declare *inverse_def[func_defs]*

An alternative combinator-based definition (by commutativity of $\mathcal{Q}$).

lemma *inverse_def2:* "*inverse* = C (D $\mathcal{Q}$)"

We introduce some convenient superscript notation.

notation *(input) inverse ("_$^{-1}$")* **notation** *(output) inverse ("'(_')$^{-1}$")*

The related notion of inverse-function of a (bijective) function can be written as:

term "$(\iota \circ f^{-1})$ ::('a ⇒ 'b) ⇒ ('b ⇒ 'a)" — Beware: well-behaved for bijective functions only!

Given a function *f* we can obtain its range as the set of those objects *b* in the codomain that are the image of some object *a* (i.e. have a non-empty preimage) under the function *f*.

definition *range::"('a ⇒ 'b) ⇒ Set('b)"*
  where "*range* ≡ ∃ ∘$_2$ *inverse*"

declare *range_def[func_defs]*

lemma "*range f* = ∃ ∘ f^{-1}"
lemma "*range f b* = (∃a. f a = b)"

More generally, the inverse of an n-ary function *f* is the n+1-ary relation that relates to each object *c* in f's codomain those ("curried" tuples of) elements in the domain are become mapped to *c* under *f* (i.e. the "preimage" of *c* under *f*). We use this to define the range of an n-ary function too.

definition *inverse2 :: "('a ⇒ 'b ⇒ 'c) ⇒ Rel$_3$('c,'a,'b)" ("inverse$_2$")*

where *"inverse*$_2$ $\equiv$ B$_{20}$ $\mathcal{Q}$*"*
definition *inverse3 :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ Rel*$_4$*('d,'a,'b,'c)" ("inverse*$_3$*")*
where *"inverse*$_3$ $\equiv$ B$_{30}$ $\mathcal{Q}$*"*
— ... *inverse*$_n$ $\equiv$ B$_{n0}$ $\mathcal{Q}$

definition *range2::"('a ⇒ 'b ⇒ 'c) ⇒ Set('c)" ("range*$_2$*")*
where *"range*$_2$ $\equiv$ $\exists^2$ $\circ_2$ *inverse*$_2$*"*
definition *range3::"('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ Set('d)" ("range*$_3$*")*
where *"range*$_3$ $\equiv$ $\exists^3$ $\circ_2$ *inverse*$_3$*"*
— ... *range*$_n$ $\equiv$ $\exists^n$ $\circ_2$ *inverse*$_n$

declare *inverse2_def[func_defs] inverse3_def[func_defs] range2_def[func_defs] range3_def[func_defs]*

lemma *"inverse*$_2$ *f c = (λa b. f a b = c)"*
lemma *"inverse*$_3$ *f d = (λa b c. f a b c = d)"*

lemma *"range*$_2$ *f c = (∃a b. f a b = c)"*
lemma *"range*$_3$ *f d = (∃a b c. f a b c = d)"*

6.3.2 Kernel of a Function

The "kernel" of a function relates those elements in its domain that get assigned the same value.

definition *kernel::"('a ⇒ 'b) ⇒ ERel('a)"*
where *"kernel* $\equiv$ $\mathbf{\Psi}_2$ $\mathcal{Q}$*"*

lemma *"kernel f = (λx y. f x = f y)"*

declare *kernel_def[func_defs]*

We add convenient superscript notation.

notation*(input) kernel ("_*$^=$*")* **notation***(***output***) kernel ("'(_')*$^=$*")*

6.3.3 Pullback and Equalizer of a Pair of Functions

The pullback (aka. fiber product) of two functions *f* and *g* (sharing the same codomain), relates those pairs of elements that get assigned the same value by *f* and *g* respectively.

definition *pullback :: "('a ⇒ 'c) ⇒ ('b ⇒ 'c) ⇒ Rel('a,'b)"*
where *"pullback* $\equiv$ B$_{11}$ $\mathcal{Q}$*"*

lemma *"pullback f g = (λx y. f x = g y)"*

declare *pullback_def[func_defs]*

Pullback can be said to be "symmetric" in the following sense.

lemma *pullback_symm: "pullback =* C$_{2143}$ *pullback"*
lemma *pullback_symm': "pullback f g x y = pullback g f y x"*
lemma *"pullback =* C $\circ_2$ *(*C *pullback)"*

Inverse and kernel of a function can be easily stated in terms of pullback.

lemma *"inverse = pullback* I*"*
lemma *"kernel =* W *pullback"*

The equalizer of two functions *f* and *g* (sharing the same domain and codomain) is the set of elements in their (common) domain that get assigned the same value by both *f* and *g*.

definition *equalizer :: "('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ Set('a)"*
where *"equalizer* $\equiv$ $\mathbf{\Phi}_{21}$ $\mathcal{Q}$*"*

lemma *"equalizer f g = (λx. f x = g x)"*

declare *equalizer_def[func_defs]*

In fact, the equalizer of two functions can be stated in terms of pullback.

lemma "*equalizer* = W $\circ_2$ *pullback*"

Note that we can swap the roles of "points" and "functions" in the above definitions using permutators.

lemma "R *equalizer x* = (λ*f g. f x = g x*)"
lemma "C_2 *pullback x y* = (λ*f g. f x = g y*)"

6.3.4 Pushout and Coequalizer of a Pair of Functions

The pushout (aka. fiber coproduct) of two functions *f* and *g* (sharing the same domain), relates pairs of elements (in their codomains) whose preimages under *f* resp. *g* intersect.

definition *pushout* :: "('c $\Rightarrow$ 'a) $\Rightarrow$ ('c $\Rightarrow$ 'b) $\Rightarrow$ *Rel*('a,'b)"
  where "*pushout* $\equiv$ B_{22} ($\sqcap$) *inverse inverse*"

lemma "*pushout f g* = (λ*x y.* f^{-1} *x* $\sqcap$ g^{-1} *y*)"

declare *pushout_def[func_defs]*

Pushout can be said to be "symmetric" in the following sense.

lemma *pushout_symm*: "*pushout* = C_{2143} *pushout*"
lemma *pushout_symm'*: "*pushout f g x y* = *pushout g f y x*"
lemma "*pushout* = C $\circ_2$ (C *pushout*)"

The equations below don't work as definitions since they unduly restrict types ("inverse" appears only once).

lemma "*pushout* = W (B_{22} ($\sqcap$)) *inverse*"
lemma "*pushout* = Ψ_2 (B_{11} ($\sqcap$)) *inverse*"

The coequalizer of two functions *f* and *g* (sharing the same domain and codomain) is the set of elements in their (common) codomain whose preimage under *f* resp. *g* intersect.

definition *coequalizer* :: "('a $\Rightarrow$ 'b) $\Rightarrow$ ('a $\Rightarrow$ 'b) $\Rightarrow$ *Set*('b)"
  where "*coequalizer* $\equiv$ W $\circ_2$ (Ψ_2 (B_{11} ($\sqcap$)) *inverse*)"

lemma "*coequalizer f g* = Φ_{21} ($\sqcap$) (f^{-1}) (g^{-1})"
lemma "*coequalizer f g* = (λ*x.* (f^{-1}) *x* $\sqcap$ (g^{-1}) *x*)"

declare *coequalizer_def[func_defs]*

The coequalizer of two functions can be stated in terms of pushout.

lemma "*coequalizer* = W $\circ_2$ *pushout*"

6.3.5 Image and Preimage

We can "lift" functions to act on sets via the image operator. The term *image f* denotes a set-operation that takes a set *A* and returns the set of elements whose *f*-preimage intersects *A*.

definition *image*::"('a $\Rightarrow$ 'b) $\Rightarrow$ *SetOp*('a,'b)"
  where "*image* $\equiv$ C (B_{20} ($\sqcap$) *inverse*)"

lemma "*image f A* = (λ*b.* f^{-1} *b* $\sqcap$ *A*)"
lemma "*image f A b* = ($\exists$*x.* f^{-1} *b x* $\wedge$ *A x*)"

Analogously, the term *preimage f* denotes a set-operation that takes a set *B* and returns the set of those elements which *f* maps to some element in *B*.

definition *preimage::"('a ⇒ 'b) ⇒ SetOp('b,'a)"*
  where *"preimage ≡* C B*"* — i.e. (;)

lemma *"preimage f B = f ; B"*
lemma *"preimage f B = (λa. B (f a))"*

declare *image_def[func_defs] preimage_def[func_defs]*

Introduce convenient notation.

notation*(input) image ("⦇_ _⦈") **and** preimage ("⦇_ _⦈$^{-1}$")*
notation*(***output***) image ("⦇'(_') '(_')⦈") **and** preimage ("⦇'(_') '(_')⦈$^{-1}$")*

term *"⦇f A⦈"* — read "the image of A under f"
term *"⦇f B⦈$^{-1}$ = (λa. B (f a))"* — read "the image of A under f"

Range can be defined in terms of image as expected.

lemma *range_def2: "range =* C *image* 𝔘*"*

term *"preimage (f::'a⇒'b) ∘ image f"*
term *"image (f::'a⇒'b) ∘ preimage f"*

lemma *"preimage f ∘ image f = (λA. λa. f$^{=}$ a ⊓ A)"*
lemma *"image f ∘ preimage f = (λB. λb. f^{-1} b ⊓ preimage f B)"*

Preservation/reversal of monoidal structure under set-operations.

lemma *image_morph1: "image (f ∘ g) = image f ∘ image g"*
lemma *image_morph2: "image* I *=* I*"*
lemma *preimage_morph1: "preimage (f ∘ g) = preimage g ∘ preimage f"*

lemma *preimage_morph2: "preimage* I *=* I*"*

Random-looking simplification(?) rule that becomes useful later on.

lemma *image_simp1: "image ((G ∘ R) a) ∘ image (*T *a) = image (*T *a) ∘ image (*S *(G ∘ R))"*

6.4 Miscellaneous

Function "update" or "override" at a point.

definition *update :: "('a ⇒ 'b) ⇒ 'a ⇒ 'b ⇒ 'a ⇒ 'b" ("_[_↦_]")*
  where *"f[a ↦ b] ≡ λx. if x = a then b else f x"*

declare *update_def[func_defs]*

A set S can be closed under a n-ary endooperation, a generalized endooperation, or a set endooperation.

definition *op1_closed::"EOp('a) ⇒ Set(Set('a))" ("_-closed$_1$")*
  where *"f-closed$_1$ ≡ λS. ∀x. S x → S(f x)"*
definition *op2_closed::"EOp$_2$('a) ⇒ Set(Set('a))" ("_-closed$_2$")*
  where *"g-closed$_2$ ≡ λS. ∀x y. S x → S y → S(g x y)"*
definition *opG_closed::"EOp$_G$('a) ⇒ Set(Set('a))" ("_-closed$_G$")*
  where *"F-closed$_G$ ≡ λS. ∀X. X ⊆ S → S(F X)"*
definition *setop_closed::"SetEOp('a) ⇒ Set(Set('a))" ("_-closed$_S$")*
  where *"φ-closed$_S$ ≡ λS. ∀X. X ⊆ S → φ X ⊆ S"*

declare *op1_closed_def[func_defs] op2_closed_def[func_defs]* *opG_closed_def[func_defs] setop_closed_def[func_defs]*

Closure under n-ary endooperations can be reduced to closure under (n-1)-ary endooperations.

lemma *op2_closed_def2: "g-closed$_2$ = (λS. (∀x. S x ⟶ (g x)-closed$_1$ S))"*
lemma *"(λS. ∀x y z. S x → S y → S z → S(g x y z)) = (λS. (∀x. S x ⟶ (g x)-closed$_2$ S))"*

The set of elements inductively generated by *G* by using a sequence of constructors, as indicated.

definition *inductiveSet1 :: "Set('a) ⇒ EOp('a) ⇒ Set('a)" ("indSet$_1$")*
where *"indSet$_1$ G f ≡ ⋂(λS. G ⊆ S ∧ f-closed$_1$ S)"* — one unary constructor
definition *inductiveSet2 :: "Set('a) ⇒ EOp$_2$('a) ⇒ Set('a)" ("indSet$_2$")*
where *"indSet$_2$ G g ≡ ⋂(λS. G ⊆ S ∧ g-closed$_2$ S)"* — one binary constructor
— and so on ...
definition *inductiveSet11 :: "Set('a) ⇒ EOp('a) ⇒ EOp('a) ⇒ Set('a)" ("indSet$_{11}$")*
where *"indSet$_{11}$ G f$_1$ f$_2$ ≡ ⋂(λS. G ⊆ S ∧ f$_1$-closed$_1$ S ∧ f$_2$-closed$_1$ S)"* — two unary constructors
definition *inductiveSet12 :: "Set('a) ⇒ EOp('a) ⇒ EOp$_2$('a) ⇒ Set('a)" ("indSet$_{12}$")*
where *"indSet$_{12}$ G f g ≡ ⋂(λS. G ⊆ S ∧ f-closed$_1$ S ∧ g-closed$_2$ S)"* — a unary and a binary constructor
— and so on ...

declare *inductiveSet1_def[func_defs] inductiveSet2_def[func_defs] inductiveSet11_def[func_defs] inductiveSet12_def[func_defs]*

A convenient special case when the set of generators *G* is a singleton *{g}*.

lemma *inductiveSet1_singleton: "indSet$_1$ {g} f = ⋂(λS. S g ∧ f-closed$_1$ S)"*

The set of all powers (via iterated composition) for a given endofunction (including I).

definition *funPower::"ERel(EOp('a))"*
where *"funPower ≡ B (indSet$_1$ (Q I)) B"*

declare *funPower_def[func_defs]*

lemma *"funPower f = indSet$_1$ {I} (λh. f ∘ h)"*
lemma *funPower_def2: "funPower f g = (∀S. (∀h. S h → S (f ∘ h)) → S I → S g)"*

Definition works as expected:

lemma *"funPower f I"*
lemma *"funPower f f"*
lemma *"funPower f (f∘f)"*
lemma *"funPower f (f∘f∘f∘f∘f∘f∘f∘f∘f∘f∘f∘f)"*
lemma *funPower_ind: "funPower f g ⟹ funPower f (f ∘ g)"*

lemma *"(∃g. funPower f g ∧ ∃!(FP g)) → ∃(FP f)"* — proof by external provers

end

7 Relations

A theory of (heterogeneous) relations as set-valued functions. Relations inherit the structures of both sets and functions and enrich them in manifold ways.

theory *relations*
imports *func_sets*

begin

named_theorems *rel_defs* **and** *rel_simps*

7.1 Constructing Relations

7.1.1 Product and Sum

Relations can also be constructed out of pairs of sets, via (cartesian) product and (disjoint) sum.

definition *product::"Set('a) ⇒ Set('b) ⇒ Rel('a,'b)"* (**infixl** "×" 90)
  where "(×) ≡ B_{11} (∧)"
definition *sum::"Set('a) ⇒ Set('b) ⇒ Rel('a,'b)"* (**infixl** "⊎" 90)
  where "(⊎) ≡ B_{11} (∨)"

declare *product_def[rel_defs] sum_def[rel_defs]*

lemma "A × B = (λx y. A x ∧ B y)"
lemma "A ⊎ B = (λx y. A x ∨ B y)"

7.1.2 Pairs and Copairs

A (co)atomic-like relation can be constructed out of two elements.

definition *pair::"'a ⇒ 'b ⇒ Rel('a,'b)"* ("⟨_,_⟩")
  where ‹pair ≡ B_{22} (∧) $\mathcal{Q}$ $\mathcal{Q}$› — relational counterpart of 'singleton'
definition *copair::"'a ⇒ 'b ⇒ Rel('a,'b)"* ("⦉_,_⦊")
  where ‹copair ≡ B_{22} (∨) $\mathcal{D}$ $\mathcal{D}$› — relational counterpart of 'cosingleton'

declare *pair_def[rel_defs] copair_def[rel_defs]*

lemma "⟨a,b⟩ = (λx y. a = x ∧ b = y)"
lemma "⦉a,b⦊ = (λx y. a ≠ x ∨ b ≠ y)"

Recalling that

lemma "B_{22} = B_{11} ∘ B_{11}"

We have that pair and copair can be defined in terms of (×) and (⊎) directly.

lemma "pair = B_{11} (×) $\mathcal{Q}$ $\mathcal{Q}$"
lemma "copair = B_{11} (⊎) $\mathcal{D}$ $\mathcal{D}$"
lemma "⟨a,b⟩ = {a} × {b}"
lemma "⦉a,b⦊ = ⦃a⦄ ⊎ ⦃b⦄"

We conveniently extrapolate the definitions of unique/singleton from sets to relations.

definition *uniqueR::"Set(Rel('a,'b))"* ("unique^2") — R holds of at most one pair of elements (R may hold of none)
  where ‹unique^2 R ≡ ∀a b x y. (R a b ∧ R x y) → (a = x ∧ b = y)›
definition *singletonR::"Set(Rel('a,'b))"* ("∃!^{2}") — R holds of exactly one pair of elements, i.e. R is a 'singleton relation'
  where ‹∃!^{2} R ≡ ∃x y. R x y ∧ (∀a b. R a b → (a = x ∧ b = y))›

declare *uniqueR_def[rel_defs] singletonR_def[rel_defs]*

lemma *uniqueR_def2:* "unique^2 = ∄^{2} ∪ ∃!^{2}"
lemma *singletonR_def2:* "∃!^{2} = ∃^{2} ∩ unique^2"

lemma *pair_singletonR:* "∃!^{2} ⟨a,b⟩"
lemma *singletonR_def3:* "∃!^{2} R = (∃a b. R = ⟨a,b⟩)"

7.2 Boolean Algebraic Structure

7.2.1 Boolean Operations

As we have seen, relations correspond to indexed (families of) sets. Hence it is not surprising that they inherit their boolean algebraic structure. Moreover, we saw previously how boolean set operations arise via "indexation" of HOL's boolean connectives (via $\mathbf{\Phi}_{m1}$ combinators). The relational boolean operations arise analogously by "double-indexation" of HOL's counterparts (via $\mathbf{\Phi}_{m2}$ combinators), or, equivalently, by "indexation" of the corresponding set counterparts, as shown below.

definition *univR::"Rel('a,'b)" ("$\mathfrak{U}^r$")*
  where "$\mathfrak{U}^r \equiv \mathbf{\Phi}_{01}\ \mathfrak{U}$" — the universal relation
definition *emptyR::"Rel('a,'b)" ("$\emptyset^r$")*
  where "$\emptyset^r \equiv \mathbf{\Phi}_{01}\ \emptyset$" — the empty relation
definition *complR::"EOp(Rel('a,'b))" ("$-^r$")*
  where ‹$-^r \equiv \mathbf{\Phi}_{11}\ -$› — relation complement
definition *interR::"EOp$_2$(Rel('a,'b))"* (**infixl** "$\cap^r$" 54)
  where "$(\cap^r) \equiv \mathbf{\Phi}_{21}\ (\cap)$" — relation intersection
definition *unionR::"EOp$_2$(Rel('a,'b))"* (**infixl** "$\cup^r$" 53)
  where "$(\cup^r) \equiv \mathbf{\Phi}_{21}\ (\cup)$" — relation union
definition *diffR:: "EOp$_2$(Rel('a,'b))"* (**infixl** "$\backslash^r$" 51)
  where "$(\backslash^r) \equiv \mathbf{\Phi}_{21}\ (\backslash)$" — relation difference
definition *implR::"EOp$_2$(Rel('a,'b))"* (**infixr** "$\Rightarrow^r$" 51)
  where "$(\Rightarrow^r) \equiv \mathbf{\Phi}_{21}\ (\Rightarrow)$" — relation implication
definition *dimplR::"EOp$_2$(Rel('a,'b))"* (**infix** "$\Leftrightarrow^r$" 51)
  where "$(\Leftrightarrow^r) \equiv \mathbf{\Phi}_{21}(\Leftrightarrow)$" — relation double-implication
definition *sdiffR::"EOp$_2$(Rel('a,'b))"* (**infix** "$\triangle^r$" 51)
  where "$(\triangle^r) \equiv \mathbf{\Phi}_{21}(\triangle)$" — relation symmetric difference (aka. xor)

Convenient notation for reversed implication.

abbreviation*(input) lpmiR::"EOp$_2$(Rel('a,'b))"* (**infixl** "$\Leftarrow^r$" 51)
  where "$A \Leftarrow^r B \equiv B \Rightarrow^r A$"

declare *univR_def[rel_defs] emptyR_def[rel_defs]*
  complR_def[rel_defs] interR_def[rel_defs] unionR_def[rel_defs]
  implR_def[rel_defs] dimplR_def[rel_defs] diffR_def[rel_defs] sdiffR_def[rel_defs]

We add a convenient superscript notation, as commonly found in the literature.

notation *(input) complR ("$(\_)^-$")* **notation(output)** *complR ("'(_')$^-$")*

Point-based definitions

lemma "$\mathfrak{U}^r = \mathbf{\Phi}_{02}\ \mathcal{T}$"
lemma "$\mathfrak{U}^r = (\lambda x\ y.\ \mathcal{T})$"
lemma "$\emptyset^r = \mathbf{\Phi}_{02}\ \mathcal{F}$"
lemma "$\emptyset^r = (\lambda x\ y.\ \mathcal{F})$"
lemma "$-^r = \mathbf{\Phi}_{12}(\neg)$"
lemma "$-^r R = (\lambda x\ y.\ \neg R\ x\ y)$"
lemma "$(\cap^r) = \mathbf{\Phi}_{22}(\wedge)$"
lemma "$R \cap^r T = (\lambda x\ y.\ R\ x\ y \wedge T\ x\ y)$"
lemma "$(\cup^r) = \mathbf{\Phi}_{22}(\vee)$"
lemma "$R \cup^r T = (\lambda x\ y.\ R\ x\ y \vee T\ x\ y)$"

Product and sum satisfy the corresponding DeMorgan dualities.

lemma *prodSum_simp1:* "$-^r(A \times B) = -A \uplus -B$"
lemma *prodSum_simp2:* "$-^r(A \uplus B) = -A \times -B$"
lemma *prodSum_simp1':* "$-^r((-A) \times (-B)) = A \uplus B$"
lemma *prodSum_simp2':* "$-^r((-A) \uplus (-B)) = A \times B$"

Pairs and copairs are related via relation-complement as expected.

lemma *copair_simp:* $"-^r \llparenthesis a,b \rrparenthesis = \langle a,b \rangle"$

declare *prodSum_simp1 [rel_simps] prodSum_simp2 [rel_simps] prodSum_simp1' [rel_simps] prodSum_simp2' [rel_simps]*

7.2.2 Ordering Structure

Similarly, relations also inherit the ordering structure of sets.

Analogously to the notion of "equalizer" of two functions, we have the "orderer" or two relations:

definition *orderer::"Rel('a,'b) ⇒ Rel('a,'b) ⇒ Set('a)"* (**infixr** *"⊑" 51)*
where $"(\sqsubseteq) \equiv \mathbf{\Phi}_{21}\ (\subseteq)"$

declare *orderer_def[rel_defs]*

lemma $"R \sqsubseteq T = (\lambda x.\ R\ x \subseteq T\ x)"$

We encode the notion of sub-/super-relation building upon the set counterparts.

definition *subrel::"ERel(Rel('a,'b))"* (**infixr** $"\subseteq^r"$ *51)*
where $"(\subseteq^r) \equiv \mathbf{\Phi}_{\forall}\ (\subseteq)"$

declare *subrel_def[rel_defs]*

lemma *subrel_setdef:* $"R \subseteq^r T = (\forall x.\ R\ x \subseteq T\ x)"$
lemma $"R \subseteq^r T = (\forall x\ y.\ R\ x\ y \rightarrow T\ x\ y)"$
lemma $"R \subseteq^r T = \forall^2 (R \Rightarrow^r T)"$
lemma *subrel_def2:* $"(\subseteq^r) = \forall\ \circ_2\ (\sqsubseteq)"$
lemma *subrel_reldef:* $"(\subseteq^r) = \forall^2\ \circ_2\ (\Rightarrow^r)"$

abbreviation *(input) superrel::"ERel(Rel('a,'b))"* (**infixr** $"\supseteq^r"$ *51)*
where $"B \supseteq^r A \equiv A \subseteq^r B"$

The "power-relation" operation corresponds to the (partial) application of superrel.

abbreviation *(input) powerrel::"Rel('a,'b) ⇒ Set(Rel('a,'b))"* $("\wp^r")$
where $"\wp^r \equiv (\supseteq^r)"$

lemma $"\wp^r A = (\lambda B.\ B \subseteq^r A)"$

Alternative characterizations of the sub/super-rel orderings in terms of fixed-points.

lemma *subrel_defFP:* $"(\subseteq^r) = FP \circ (\cup^r)"$
lemma *superrel_defFP:* $"(\supseteq^r) = FP \circ (\cap^r)"$
lemma $"(R \subseteq^r T) = (T = R \cup^r T)"$
lemma $"(T \supseteq^r R) = (R = T \cap^r R)"$

Sub-relation is antisymmetric

lemma *subrel_antisym:* $"R \subseteq^r T \Longrightarrow R \supseteq^r T \Longrightarrow R = T"$

Two relations are said to "overlap" (or "intersect") if their intersection is non-empty

definition *overlapR::"ERel(Rel('a,'b))"* (**infix** $"\sqcap^r"$ *52)*
where $"(\sqcap^r) \equiv \mathbf{\Phi}_{\exists}\ (\sqcap)"$

Dually, two relations form a "cover" if every pair belongs to one or the other.

definition *coverR::"ERel(Rel('a,'b))"* (**infix** $"\sqcup^r"$ *53)*
where $"(\sqcup^r) \equiv \mathbf{\Phi}_{\forall}\ (\sqcup)"$

declare *overlapR_def[rel_defs] coverR_def[rel_defs]*

Convenient notation: Two relations can also be said to be "incompatible" analogously to sets.

abbreviation *(input) incompatR::"ERel(Rel('a,'b))"* **(infix** *"$\bot^r$" 52)*
  where *"($\bot^r$) $\equiv$ $\nexists^2$ $\circ_2$ ($\cap^r$)"*

lemma *coverR_reldef: "($\sqcup^r$) = $\forall^2$ $\circ_2$ ($\cup^r$)"*
lemma *overlapR_reldef: "($\sqcap^r$) = $\exists^2$ $\circ_2$ ($\cap^r$)"*
lemma *"A $\sqcup^r$ B = $\forall^2$(A $\cup^r$ B)"*
lemma *"A $\sqcap^r$ B = $\exists^2$(A $\cap^r$ B)"*
lemma *"A $\bot^r$ B = $\nexists^2$(A $\cap^r$ B)"*

7.2.3 Infinitary Operations

We can also generalize union and intersection to the infinitary case.

definition *biginterR::"EOp$_G$(Rel('a,'b))" ("$\bigcap^r$")*
  where *"$\bigcap^r$ $\equiv$ $\bigcap$ $\circ_2$ (B$_{10}$ image T)"*
definition *bigunionR::"EOp$_G$(Rel('a,'b))" ("$\bigcup^r$")*
  where *"$\bigcup^r$ $\equiv$ $\bigcup$ $\circ_2$ (B$_{10}$ image T)"*

declare *biginterR_def[rel_defs] bigunionR_def[rel_defs]*

lemma *"$\bigcap^r$S a = $\bigcap$⦇(λR. R a) S⦈"*
lemma *"$\bigcup^r$S a = $\bigcup$⦇(λR. R a) S⦈"*

Alternative definitions in terms of quantifiers directly.

lemma *biginterR_def2: "$\bigcap^r$S = (λa b. $\forall$R. S R $\rightarrow$ R a b)"*
lemma *bigunionR_def2: "$\bigcup^r$S = (λa b. $\exists$R. S R $\wedge$ R a b)"*

We say of a set of relations that it "overlaps" (or "intersects") if there exists a shared pair.

abbreviation *(input) bigoverlapR::"Set(Set(Rel('a,'b)))" ("$\bigsqcap^r$")*
  where *"$\bigsqcap^r$ $\equiv$ $\exists^2$ $\circ$ $\bigcap^r$"*

Dually, a set of relations forms a "cover" if every pair is contained in at least one of the relations.

abbreviation *(input) bigcoverR::"Set(Set(Rel('a,'b)))" ("$\bigsqcup^r$")*
  where *"$\bigsqcup^r$ $\equiv$ $\forall^2$ $\circ$ $\bigcup^r$"*

lemma *"$\bigsqcap^r$S = $\exists^2$($\bigcap^r$S)"*
lemma *"$\bigsqcup^r$S = $\forall^2$($\bigcup^r$S)"*

7.3 Function-like Structure I

We have seen the shared (boolean) algebraic structure between sets and relations. We now explore their shared structure with functions.

We start by noting that, given a relation *R* of type *Rel('a,'b)*, we refer to the semantic domain of type *'a* as R's "source" domain, which is identical to R's domain when seen as a (set-valued) function. Analogously, we refer to the semantic domain for type *'b* as R's "target" domain, which is in fact different from its codomain when seen as a (set-valued) function (corresponding to the type *'b $\Rightarrow$ o*).

7.3.1 Range and Cylindrification

We define the left- (right-) range of a relation as the set of those objects in the source (target) domain that reach to (are reached by) some element in the target (source) domain.

definition *leftRange::"Rel('a,'b) $\Rightarrow$ Set('a)"*
  where *"leftRange $\equiv$ $\exists$ $\circ_2$ A"*
definition *rightRange::"Rel('a,'b) $\Rightarrow$ Set('b)"*
  where *"rightRange $\equiv$ $\exists$ $\circ_2$ C"*

```
lemma "leftRange R a = (∃x. R a x)"
lemma "rightRange R b = (∃x. R x b)"
```

Dually, the left- (right-) dual-range of a relation is the set of those objects in the source (target) domain that reach to (are reached by) all elements in the target (source) domain.

```
definition leftDualRange::"Rel('a,'b) ⇒ Set('a)"
  where "leftDualRange ≡ ∀ ∘₂ A"
definition rightDualRange::"Rel('a,'b) ⇒ Set('b)"
  where "rightDualRange ≡ ∀ ∘₂ C"

lemma "leftDualRange R a = (∀x. R a x)"
lemma "rightDualRange R b = (∀x. R x b)"

declare leftRange_def[rel_defs] rightRange_def[rel_defs]
        leftDualRange_def[rel_defs] rightDualRange_def[rel_defs]
```

Both pairs of definitions are "dual" wrt. complement.

```
lemma "rightDualRange R = −(rightRange R⁻)"
lemma "leftDualRange R = −(leftRange R⁻)"
```

For the left we have in fact that ranges are obtained directly by composition with ∃ and ∀.

```
lemma leftRange_def2: "leftRange = B ∃"
lemma leftDualRange_def2: "leftDualRange = B ∀"
```

The operations below perform what is known as "cylindrification" in the literature on relation algebra.

```
definition leftCylinder::"Set('b) ⇒ Rel('a,'b)"
  where "leftCylinder ≡ K"
definition rightCylinder::"Set('a) ⇒ Rel('a,'b)"
  where "rightCylinder ≡ B K"

declare leftCylinder_def[rel_defs] rightCylinder_def[rel_defs]

lemma "leftCylinder  S = (λa b. S b)"
lemma "rightCylinder S = (λa b. S a)"
```

Alternative formulation in terms of cartesian product.

```
lemma leftCylinder_def2:  "leftCylinder  A = 𝔘 × A"
lemma rightCylinder_def2: "rightCylinder A = A × 𝔘"
```

They act inverse to (right and left) range by transforming sets into (left and right-ideal) relations.

```
lemma "rightRange (leftCylinder A) = A"
lemma "leftRange (rightCylinder A) = A"
```

Also note that:

```
lemma "R ⊆ʳ rightCylinder (leftRange R)"
lemma "R ⊆ʳ leftCylinder (rightRange R)"
proposition "rightCylinder (leftRange R) ⊆ʳ R"  nitpick — countermodel found
proposition "leftCylinder (rightRange R) ⊆ʳ R"  nitpick — countermodel found
```

Source and target restrictions (as relation-operations) can be encoded in terms of cylindrification.

```
definition sourceRestriction::"Set('a) ⇒ Rel('a,'b) ⇒ Rel('a,'b)" ("_↲_")
  where "sourceRestriction ≡ B₁₁ (∩ʳ) rightCylinder I"
definition targetRestriction::"Set('b) ⇒ Rel('a,'b) ⇒ Rel('a,'b)" ("_↳_")
  where "targetRestriction ≡ B₁₁ (∩ʳ) leftCylinder I"
```

declare *sourceRestriction_def[rel_defs] targetRestriction_def[rel_defs]*

lemma *"A⇂R = rightCylinder A* $\cap^r$ *R"*
lemma *"B⇃R = leftCylinder B* $\cap^r$ *R"*
lemma *"A⇂R = (λa b. A a ∧ R a b)"*
lemma *"B⇃R = (λa b. B b ∧ R a b)"*

7.3.2 Uniqueness and Determinism

By composition with *unique*, we obtain the set of deterministic (or "univalent") elements. They get assigned at most one value under the relation (which then behaves deterministically on them)

definition *deterministic::"Rel('a,'b) ⇒ Set('a)"*
  where *"deterministic ≡* B *unique"*

Also, by composition with *∃!*, we obtain the set of total(ly) deterministic elements. They get assigned precisely one value under the relation (which then behaves as a function on them)

definition *totalDeterministic::"Rel('a,'b) ⇒ Set('a)"*
  where *"totalDeterministic ≡* B *∃!"*

declare *deterministic_def[rel_defs] totalDeterministic_def[rel_defs]*

lemma *totalDeterministic_def2: "totalDeterministic R = deterministic R ∩ leftRange R"*

Right- resp. left-unique relations; aka. univalent/(partial-)functional resp. injective relations.

definition *rightUnique::"Set(Rel('a,'b))"*
  where *"rightUnique ≡ ∀ ∘ deterministic"*
definition *leftUnique::"Set(Rel('a,'b))"*
  where *"leftUnique ≡ ∀ ∘ deterministic ∘* C*"*

declare *rightUnique_def [rel_defs] leftUnique_def [rel_defs]*

Further names for special kinds of relations, also common in the literature.

abbreviation *(input) "one_to_one R ≡ leftUnique R ∧ rightUnique R"* — injective and functional
abbreviation *(input) "one_to_many R ≡ leftUnique R ∧ ¬rightUnique R"* — injective and not functional
abbreviation *(input) " many_to_one R ≡ ¬leftUnique R ∧ rightUnique R"* — functional and not injective
abbreviation *(input) "many_to_many R ≡ ¬leftUnique R ∧ ¬rightUnique R"* — neither injective nor functional

Pairs are both right-unique and left-unique, i.e. one-to-one.

lemma *"singletonR ⊆ one_to_one"*
proposition *"one_to_one ⊆ singletonR"* **nitpick** — counterexample: e.g. empty relation

In fact, any relation can also be generated by its right- resp. left-unique subrelations.

lemma *rightUnique_gen: "R =* $\bigcup^r$ *(*$\wp^r$ *R ∩ rightUnique)"* — proof by external provers

lemma *leftUnique_gen: "R =* $\bigcup^r$ *(*$\wp^r$ *R ∩ leftUnique)"* — proof by external provers

7.3.3 Totality

Right- resp. left-unique relations; aka. surjective resp. total/serial/multi-functional relations.

definition *rightTotal::"Set(Rel('a,'b))"*
  where *"rightTotal ≡ ∀ ∘ rightRange"*
definition *leftTotal::"Set(Rel('a,'b))"*

```
  where "leftTotal ≡ ∀ ∘ leftRange"

declare rightTotal_def[rel_defs] leftTotal_def[rel_defs]
```

A relation that relates each element in its source to precisely one element in its target corresponds to a (total) function. They can also be characterized as being both total and functional (i.e. left-total and right-unique) relations.

```
definition totalFunction::"Set(Rel('a,'b))"
  where "totalFunction ≡ ∀ ∘ totalDeterministic"

declare totalFunction_def[rel_defs]

lemma totalFunction_def2: "totalFunction R = (leftTotal R ∧ rightUnique R)"
```

The inverse of a function (qua relation) is always left-unique and right-total.

```
lemma "leftUnique f⁻¹"
lemma "rightTotal f⁻¹"
```

7.4 Transformations between Relations and (Sets of) Functions

7.4.1 From (Sets of) Functions to Relations

A given function can be disguised as a relation.

```
definition asRel::"('a ⇒ 'b) ⇒ Rel('a,'b)" ("asRel")
  where "asRel ≡ B 𝒬"

declare asRel_def[rel_defs]

lemma "asRel f = 𝒬 ∘ f"
lemma "asRel f = (λa. 𝒬 (f a))"
lemma "asRel f = (λa. (λb. 𝒬 (f a) b))"
lemma "asRel f = (λa b. f a = b)"
```

Alternative characterization:

```
lemma asRel_def2: "asRel = C ∘ inverse"
lemma "asRel f = C (f⁻¹)"
```

Relations corresponding to lifted functions are always left-total and right-unique (i.e. functions).

```
lemma "totalFunction (asRel f)"
```

A given set of functions can be transformed (or "aggregated") into a relation.

```
definition intoRel::"Set('a ⇒ 'b) ⇒ Rel('a,'b)" ("intoRel")
  where "intoRel ≡ C (image ∘ T)"

declare intoRel_def[rel_defs]

lemma "intoRel = (λS a. ⦇(T a) S⦈)"
lemma "intoRel S a = ⦇(λf. f a) S⦈"
```

Alternative characterization (in terms of relational bigunion):

```
lemma intoRel_def2: "intoRel = ⋃ʳ ∘ (image asRel)"
lemma "intoRel S = ⋃ʳ⦇asRel S⦈"
```

7.4.2 From Relations to (Sets of) Functions

A given relation can be disguised as a function (and go unnoticed under certain circumstances).

```
definition asFun::"Rel('a,'b) ⇒ ('a ⇒ 'b)" ("asFun")
```

where "asFun ≡ B ε"

declare *asFun_def[rel_defs]*

lemma "asFun R = ε ∘ R"
lemma "asFun R = (λa. ε(R a))"
lemma "asFun R = (λa. ε b. R a b)"

lemma *asFun_def2:* "totalFunction R ⟹ asFun R = ι ∘ R" — alternative definition for total functions

Transforming (or 'decomposing') a given relation into a set of functions.

definition *intoFunSet::*"Rel('a,'b) ⇒ Set('a ⇒ 'b)" ("intoFunSet")
where "intoFunSet ≡ C ((⊆^{r}) ∘ asRel)"

declare *intoFunSet_def[rel_defs]*

lemma "intoFunSet R = (λf. asRel f ⊆^{r} R)"
lemma "intoFunSet R = (λf. ∀a b. f a = b → R a b)"

Another perspective:

lemma *intoFunSet_def2:* "intoFunSet = B_{11} ℘^{r} I asRel"

7.4.3 Back-and-Forth Translation Conditions

Disguising a function as a relation, and back as a function, gives back the original function.

lemma *funRel_trans:* "asFun (asRel f) = f"

However, disguising a relation as a function, and back as a relation, does not give anything recognizable.

proposition "asRel (asFun R) = R" **nitpick** — countermodel found

In case of left-total relations, what we get back is a strict subrelation.

lemma *relFun_trans1:* "leftTotal R ⟹ asRel (asFun R) ⊆^{r} R"
proposition "leftTotal R ⟹ R ⊆^{r} asRel (asFun R)" **nitpick** — countermodel found

In case of right-unique relations, what we get back is a strict superrelation.

lemma *relFun_trans2:* "rightUnique R ⟹ R ⊆^{r} asRel (asFun R)"
proposition "rightUnique R ⟹ asRel (asFun R) ⊆^{r} R" **nitpick** — countermodel found

Indeed, we get back the original relation when it is a total-function.

lemma *relFun_trans:* "totalFunction R ⟹ asRel (asFun R) = R"

Transforming a set of functions into a relation, and back to a set of functions, gives a strict superset.

lemma *funsetRel_trans1:* "S ⊆ intoFunSet (intoRel S)"
proposition "intoFunSet (intoRel S) ⊆ S" **nitpick** — countermodel found

We get the original set in those cases where it corresponds already to a transformed relation.

lemma *funsetRel_trans2:*"let S = intoFunSet R in intoFunSet (intoRel S) ⊆ S"

Transforming a relation into a set of functions, and back to a relation, gives a strict subrelation.

lemma *relFunSet_trans1:* "intoRel (intoFunSet R) ⊆^{r} R"
proposition "R ⊆^{r} intoRel (intoFunSet R)" **nitpick** — countermodel found

In fact, we get the original relation in case it is left-total.

lemma *leftTotal_auxsimp:* "leftTotal R $\Longrightarrow$ R a b = (let f = (asFun R)[a $\mapsto$ b] in (f a = b $\wedge$ (asRel f) $\subseteq^r$ R))"
lemma *relFunSet_trans2:* "leftTotal R $\Longrightarrow$ R $\subseteq^r$ intoRel (intoFunSet R)"
lemma *relFunSet_simp:* "leftTotal R $\Longrightarrow$ intoRel (intoFunSet R) = R"

7.5 Transpose and Cotranspose

Relations come with two further idiosyncratic unary operations. The first one is transposition (aka. "converse" or "reverse"), which naturally arises by seeing relations as binary operations (with boolean codomain), and corresponds to the C combinator. The second one, which we call "cotransposition", corresponds to the transpose/converse of the complement (which is in fact identical to the complement of the transpose).

definition *transpose::"Rel('a,'b) $\Rightarrow$ Rel('b,'a)" ("$\smile$")*
  where "$\smile$ $\equiv$ C"
definition *cotranspose::"Rel('a,'b) $\Rightarrow$ Rel('b,'a)" ("$\sim$")*
  where "$\sim$ $\equiv$ $-^r$ $\circ$ C"

declare *transpose_def[rel_defs] cotranspose_def[rel_defs]*

Most of the time we will employ the following superscript notation (analogously to complement).

notation*(input) transpose ("$(\_)^\smile$")* **and** *cotranspose ("$(\_)^\sim$")*
notation*(***output***) transpose ("$'(\_')^\smile$")* **and** *cotranspose ("$'(\_')^\sim$")*

lemma "$R^\sim$ = $R^{\smile -}$"
lemma "$R^\sim$ = $R^{- \smile}$"

lemma *transpose_involutive:* "$R^{\smile\smile}$ = R"
lemma *cotranspose_involutive:* "$R^{\sim\sim}$ = R"
lemma *complement_involutive:* "R^{--} = R"

Clearly, (co)transposition (co)distributes over union and intersection.

lemma "$(R \cup^r T)^\smile$ = $(R^\smile) \cup^r (T^\smile)$"
lemma "$(R \cap^r T)^\smile$ = $(R^\smile) \cap^r (T^\smile)$"
lemma "$(R \cup^r T)^\sim$ = $(R^\sim) \cap^r (T^\sim)$"
lemma "$(R \cap^r T)^\sim$ = $(R^\sim) \cup^r (T^\sim)$"

The inverse of a function corresponds to its converse when seen as a relation.

lemma ‹f^{-1} = $(\mathit{asRel}\ f)^\smile$›

Relational "lifting" commutes with transpose.

lemma *relLiftEx_trans:* "$\mathbf{\Phi}_\exists$ $(R^\smile)$ = $(\mathbf{\Phi}_\exists\ R)^\smile$"
lemma *relLiftAll_trans:* "$\mathbf{\Phi}_\forall$ $(R^\smile)$ = $(\mathbf{\Phi}_\forall\ R)^\smile$"

And "dually commutes" with co-transpose.

lemma *relLiftEx_cotrans:* "$\mathbf{\Phi}_\exists$ $(R^\sim)$ = $(\mathbf{\Phi}_\forall\ R)^\sim$"
lemma *relLiftAll_cotrans:* "$\mathbf{\Phi}_\forall$ $(R^\sim)$ = $(\mathbf{\Phi}_\exists\ R)^\sim$"

Using transpose, we can encode a convenient notion of "interpolants" (wrt. two relations) as the set of elements that "bridge" between two given points (belonging each to one of the relations), as follows.

definition *interpolants :: "Rel('a,'c) $\Rightarrow$ Rel('c,'b) $\Rightarrow$ 'a $\Rightarrow$ 'b $\Rightarrow$ Set('c)"*
  where "interpolants $\equiv$ B_{22} ($\cap$) A $\smile$"

And, since we are at it, we add a convenient dual notion.

definition *dualInterpolants :: "Rel('a,'c) $\Rightarrow$ Rel('c,'b) $\Rightarrow$ 'a $\Rightarrow$ 'b $\Rightarrow$ Set('c)"*
  where "dualInterpolants $\equiv$ B_{22} ($\cup$) A $\smile$"

declare *interpolants_def[rel_defs] dualInterpolants_def[rel_defs]*

lemma "interpolants R_1 R_2 a b = R_1 a $\cap$ $R_2{}^\smile$ b"
lemma "dualInterpolants R_1 R_2 a b = R_1 a $\cup$ $R_2{}^\smile$ b"
lemma "interpolants R_1 R_2 a b = (λc. R_1 a c $\wedge$ R_2 c b)"
lemma "dualInterpolants R_1 R_2 a b = (λc. R_1 a c $\vee$ R_2 c b)"

7.6 Structure Preservation and Reflection

The function f preserves the relational structure of R into T.

abbreviation *(input) preserving::"ERel('a) $\Rightarrow$ ERel('b) $\Rightarrow$ Set(Op('a,'b))" ("_,_-preserving")*
where *"R,T-preserving f $\equiv$ $\forall$X Y. R X Y $\to$ T (f X) (f Y)"*

The function f reflects the relational structure of T into R.

abbreviation *(input) reflecting::"ERel('a) $\Rightarrow$ ERel('b) $\Rightarrow$ Set(Op('a,'b))" ("_,_-reflecting")*
where *"R,T-reflecting f $\equiv$ $\forall$X Y. R X Y $\leftarrow$ T (f X) (f Y)"*

This generalizes the notion of order-embedding to (endo)relations in general.

abbreviation *(input) embedding::"ERel('a) $\Rightarrow$ ERel('b) $\Rightarrow$ Set(Op('a,'b))" ("_,_-embedding")*
where *"R,T-embedding f $\equiv$ $\forall$X Y. R X Y = T (f X) (f Y)"*

Clearly, a function is an embedding iff it is both preserving and reflecting.

lemma *"R,T-embedding f = (R,T-preserving f $\wedge$ R,T-reflecting f)"*

An endofunction f is said to be monotonic resp. anti(mono)tonic wrt an endorelation R when it is R-preserving resp. R-reversing

definition *monotonic::"ERel('a) $\Rightarrow$ Set(EOp('a))" ("_-MONO")*
where *"R-MONO $\equiv$ R,R-preserving"*
definition *antitonic::"ERel('a) $\Rightarrow$ Set(EOp('a))" ("_-ANTI")*
where *"R-ANTI $\equiv$ R,R$^\smile$-preserving"*

declare *monotonic_def[rel_defs] antitonic_def[rel_defs]*

lemma *"R-MONO f = ($\forall$A B. R A B $\longrightarrow$ R (f A) (f B))"*
lemma *"R-ANTI f = ($\forall$A B. R A B $\longrightarrow$ R (f B) (f A))"*
lemma *"($\subseteq^r$)-MONO f = ($\forall$A B. A $\subseteq^r$ B $\longrightarrow$ f A $\subseteq^r$ f B)"*
lemma *"($\subseteq^r$)-ANTI f = ($\forall$A B. A $\subseteq^r$ B $\longrightarrow$ f B $\subseteq^r$ f A)"*

Monotonic endofunctions are called "closure/interior operators" when they satisfy particular properties.

definition *closure ("_-CLOSURE")*
where *"R-CLOSURE φ $\equiv$ R-MONO φ $\wedge$ R-EXPN φ $\wedge$ R-wCNTR φ"*
definition *interior ("_-INTERIOR")*
where *"R-INTERIOR φ $\equiv$ R-MONO φ $\wedge$ R-CNTR φ $\wedge$ R-wEXPN φ"*

declare *closure_def[rel_defs] interior_def[rel_defs]*

lemma *closure_setprop: "($\subseteq$)-CLOSURE f = ($\forall$A B. (A $\subseteq$ f B) $\leftrightarrow$ (f A $\subseteq$ f B))"*

7.7 Function-like Structure II

7.7.1 Monoidal Structure (composition and its dual)

In analogy to functions, relations can also be composed, as follows:

definition *relComp::"Rel('a,'b) $\Rightarrow$ Rel('b,'c) $\Rightarrow$ Rel('a,'c)"* (**infixr** *";^{r}" 55)*
where *"(;^{r}) = $\mathbf{B}_{22}$ ($\sqcap$) $\mathbf{A}$ $\smile$ "*

Again, we can in fact define an operator that acts as a "dual" to relation-composition:

definition *relDualComp::"Rel('c,'a)* $\Rightarrow$ *Rel('a,'b)* $\Rightarrow$ *Rel('c,'b)"* **(infixr** "$\dagger^r$" *55)*
  where "$(\dagger^r) \equiv \mathtt{B}_{22}\ (\sqcup)\ \mathtt{A}\ {}^{\smile}$"

declare *relDualComp_def[rel_defs] relComp_def[rel_defs]*

lemma "$R_1\ ;^r\ R_2 = (\lambda a\ b.\ R_1\ a \sqcap R_2{}^{\smile}\ b)$"
lemma "$R_1\ \dagger^r\ R_2 = (\lambda a\ b.\ R_1\ a \sqcup R_2{}^{\smile}\ b)$"
lemma "$R_1\ ;^r\ R_2 = (\lambda a\ b.\ \exists c.\ R_1\ a\ c \wedge R_2\ c\ b)$"
lemma "$R_1\ \dagger^r\ R_2 = (\lambda a\ b.\ \forall c.\ R_1\ a\ c \vee R_2\ c\ b)$"
lemma "$R_1\ ;^r\ R_2 = (\lambda a\ b.\ \exists(\mathit{interpolants}\ R_1\ R_2\ a\ b))$"
lemma "$R_1\ \dagger^r\ R_2 = (\lambda a\ b.\ \forall(\mathit{dualInterpolants}\ R_1\ R_2\ a\ b))$"
lemma "$R_1\ ;^r\ R_2 = (\exists\ \circ_2\ (\mathit{interpolants}\ R_1\ R_2))$"
lemma "$R_1\ \dagger^r\ R_2 = (\forall\ \circ_2\ (\mathit{dualInterpolants}\ R_1\ R_2))$"
lemma *relComp_def2:* "$(;^r) = \exists\ \circ_4\ \mathit{interpolants}$"
lemma *relDualComp_def2:* "$(\dagger^r) = \forall\ \circ_4\ \mathit{dualInterpolants}$"

We introduce convenient "flipped" notations for (dual-)composition (analogous to those for functions).

abbreviation *(input) relComp_t::"Rel('b,'c)* $\Rightarrow$ *Rel('a,'b)* $\Rightarrow$ *Rel('a,'c)"* **(infixl** "$\circ^r$" *55)*
  where "$R\ \circ^r\ T \equiv T\ ;^r\ R$"
abbreviation *(input) relDualComp_t::"Rel('c,'b)* $\Rightarrow$ *Rel('a,'c)* $\Rightarrow$ *Rel('a,'b)"* **(infixl** "$\bullet^r$" *55)*
  where "$R\ \bullet^r\ T \equiv T\ \dagger^r\ R$"

Unsurprisingly, (relational) composition and dual-composition are dual wrt. (relational) complement.

lemma *relCompDuality1:* "$R\ \bullet^r\ T = ((R^-)\ \circ^r\ (T^-))^-$"
lemma *relCompDuality2:* "$R\ \circ^r\ T = ((R^-)\ \bullet^r\ (T^-))^-$"

Moreover, relation (dual)composition and (dis)equality satisfy the monoid conditions

lemma *relComp_assoc:* "$(R\ \circ^r\ T)\ \circ^r\ V = R\ \circ^r\ (T\ \circ^r\ V)$" — associativity
lemma *relComp_id1:* "$\mathcal{Q}\ \circ^r\ R = R$" — identity 1
lemma *relComp_id2:* "$R\ \circ^r\ \mathcal{Q} = R$" — identity 2
lemma *relCompDual_assoc:* "$(R\ \bullet^r\ T)\ \bullet^r\ V = R\ \bullet^r\ (T\ \bullet^r\ V)$" — associativity
lemma *relCompDual_id1:* "$\mathcal{D}\ \bullet^r\ R = R$" — identity 1
lemma *relCompDual_id2:* "$R\ \bullet^r\ \mathcal{D} = R$" — identity 2

Transpose acts as an "antihomomorphism" wrt. composition as well as its dual.

lemma *relComp_antihom:* "$(R\ \circ^r\ T)^{\smile} = ((T^{\smile})\ \circ^r\ (R^{\smile}))$"
lemma *relCompDual_antihom:* "$(R\ \bullet^r\ T)^{\smile} = ((T^{\smile})\ \bullet^r\ (R^{\smile}))$"

In a similar spirit, we have:

lemma "$(R\ \circ^r\ T)^{\sim} = ((T^{\sim})\ \bullet^r\ (R^{\sim}))$"
lemma "$(R\ \bullet^r\ T)^{\sim} = ((T^{\sim})\ \circ^r\ (R^{\sim}))$"

7.7.2 Residuals

Introduce residuals (on the left resp. right) wrt. composition taken as $(;^r)$.

definition *residualOnRight::"Rel('c,'a)* $\Rightarrow$ *Rel('c,'b)* $\Rightarrow$ *Rel('a,'b)"* **(infix** "$\triangleright^r$" *99)*
  where "$R\ \triangleright^r\ S \equiv (R^{\sim})\ \dagger^r\ S$"
definition *residualOnLeft::"Rel('a,'c)* $\Rightarrow$ *Rel('b,'c)* $\Rightarrow$ *Rel('a,'b)"* **(infix** "$\triangleleft^r$" *99)*
  where "$R\ \triangleleft^r\ S \equiv R\ \dagger^r\ (S^{\sim})$"

declare *residualOnRight_def[rel_defs] residualOnLeft_def[rel_defs]*

Residuals can alternatively be defined using converse and complement.

lemma *residualOnRight_def2:* "$R\ \triangleright^r\ S = ((R^{\smile})\ ;^r\ (S^-))^-$"

lemma *residualOnLeft_def2:* "$R \lhd^r S = ((R^-) ;^r (S^\smile))^-$"

We verify that they work as residuals wrt. $(;^r)$ in the expected way.

lemma *residual_simp1:* "$(R ;^r S \subseteq^r T) = (S \subseteq^r R \rhd^r T)$"
lemma *residual_simp2:* "$(R ;^r S \subseteq^r T) = (R \subseteq^r T \lhd^r S)$"

Introduce some convenient reversed notation for the corresponding residuals wrt. $(\circ^r)$.

abbreviation *(input) residualOnRight_t* (**infix** "$\lhd^r$" *99)*
where "$R \lhd^r S \equiv S \rhd^r R$"
abbreviation *(input) residualOnLeft_t* (**infix** "$\unrhd^r$" *99)*
where "$R \unrhd^r S \equiv S \unlhd^r R$"

Check alternative characterization.

lemma "$R \unrhd^r S = ((R^\smile) \circ^r (S^-))^-$"
lemma "$R \lhd^r S = ((R^-) \circ^r (S^\smile))^-$"

Verify that they work as residuals wrt. $(\circ^r)$ in the expected way.

lemma "$(R \circ^r S \subseteq^r T) = (S \subseteq^r R \unrhd^r T)$"
lemma "$(R \circ^r S \subseteq^r T) = (R \subseteq^r T \lhd^r S)$"

7.7.3 Ideal Elements

A related property of relations is that of (generating a) left- resp. right ideal.

definition *leftIdeal::"Set(Rel('a,'b))"*
where "$leftIdeal \equiv FP\ ((;^r)\ \mho^r)$"
definition *rightIdeal::"Set(Rel('a,'b))"*
where "$rightIdeal \equiv FP\ ((\circ^r)\ \mho^r)$"

declare *leftIdeal_def[rel_defs] rightIdeal_def[rel_defs]*

lemma "$leftIdeal\ R = (R = \mho^r ;^r R)$"
lemma "$rightIdeal\ R = (R = R ;^r \mho^r)$"

An alternative, equivalent definition also common in the literature (e.g. on semirings).

lemma *leftIdeal_def2:* "$leftIdeal\ R = (\forall T.\ R \circ^r T \subseteq^r R)$"
lemma *rightIdeal_def2:* "$rightIdeal\ R = (\forall T.\ R ;^r T \subseteq^r R)$"

In fact, the left/right-cylindrification operations discussed previously return left/right-ideal (generating) relations. Moreover, all left/right-ideal relations can be generated this way.

lemma "*rightIdeal = range rightCylinder*"
lemma "*leftIdeal = range leftCylinder*"

7.7.4 Kernel of a Relation

The *kernel* of a relation relates those elements in its source domain that are related to some same value (i.e. whose images overlap).

definition *relKernel::"Rel('a,'b) ⇒ ERel('a)"*
where "$relKernel \equiv \mathbf{\Psi}_2\ (\sqcap)$"

declare *relKernel_def[rel_defs]*

lemma "$relKernel\ R = (\lambda x\ y.\ R\ x \sqcap R\ y)$"

The notion of kernel for relations corresponds to (and generalizes) the functional counterpart.

lemma "*relKernel (asRel f) = kernel f*"
lemma "$totalFunction\ R \Longrightarrow kernel\ (asFun\ R) = relKernel\ R$"

7.7.5 Pullback and Equalizer of a Pair of Relations

The pullback (aka. fiber product) of two relations R and T (sharing the same target), relates those pairs of elements that get assigned some same value by R and T respectively.

definition *relPullback :: "Rel('a,'c) ⇒ Rel('b,'c) ⇒ Rel('a,'b)"*
  where *"relPullback ≡* $\mathtt{B}_{11}$ *(⊓)"*

declare *relPullback_def[rel_defs]*

lemma *"relPullback R T = (λx y. R x ⊓ T y)"*

Pullback can be said to be "symmetric" in the following sense.

lemma *relPullback_symm: "relPullback =* $\mathtt{C}_{2143}$ *relPullback"*
lemma *relPullback_symm': "relPullback R T x y = relPullback T R y x"*
lemma *"relPullback =* C $\circ_2$ *(*C *relPullback)"*

The notion of pullback for relations corresponds to (and generalizes) the functional counterpart.

lemma *"relPullback (asRel f) (asRel g) = pullback f g"*
lemma *"totalFunction R ⟹ totalFunction T ⟹ pullback (asFun R) (asFun T) = relPullback R T"*

Converse and kernel of a relation can be easily stated in terms of relation-pullback.

lemma *"*C *= relPullback* $\mathcal{Q}$*"*
lemma *"relKernel =* W *relPullback"*

The equalizer of two relations *R* and *T* (sharing the same source and target) is the set of elements *x* in their (common) source that are related to some same value (i.e. *R x* and *T x* intersect).

definition *relEqualizer :: "Rel('a,'b) ⇒ Rel('a,'b) ⇒ Set('a)"*
  where *"relEqualizer ≡* $\mathbf{\Phi}_{21}$ *(⊓)"*

declare *relEqualizer_def[rel_defs]*

lemma *"relEqualizer R T = (λx. R x ⊓ T x)"*

In fact, the equalizer of two relations can be stated in terms of their pullback.

lemma *"relEqualizer =* W $\circ_2$ *relPullback"*

Note that we can swap the roles of "points" and "functions" in the above definitions using permutators.

lemma *"*R *relEqualizer x = (λR T. R x ⊓ T x)"*
lemma *"*$\mathtt{C}_2$ *relPullback x y = (λR T. R x ⊓ T y)"*

The notion of equalizer for relations corresponds to (and generalizes) the functional counterpart.

lemma *"relEqualizer (asRel f) (asRel g) = equalizer f g"*
lemma *"totalFunction R ⟹ totalFunction T ⟹ equalizer (asFun R) (asFun T) = relEqualizer R T"*

7.7.6 Pushout and Coequalizer of a Pair of Relations

The pushout (aka. fiber coproduct) of two relations R and T (sharing the same source), relates pairs of elements (in their targets) whose preimages under R resp. T intersect.

abbreviation *relPushout :: "Rel('a,'b) ⇒ Rel('a,'c) ⇒ Rel('b,'c)"*
  where *"relPushout R T ≡ relPullback R⌣ T⌣"*

lemma *"relPushout R T = (λx y. R⌣ x ⊓ T⌣ y)"*

The notion of pushout for relations corresponds to (and generalizes) the functional counterpart.

lemma *"relPushout (asRel f) (asRel g) = pushout f g"*
lemma *"totalFunction R ⟹ totalFunction T ⟹ pushout (asFun R) (asFun T) = relPushout R T"*

The coequalizer of two relations R and T (sharing the same source and target) is the set of elements in their (common) target whose preimage under R resp. T intersect.

abbreviation *relCoequalizer :: "Rel('a,'b) ⇒ Rel('a,'b) ⇒ Set('b)"*
  where *"relCoequalizer R T ≡ relEqualizer $R^{\smile}$ $T^{\smile}$"*

lemma *"relCoequalizer R T = (λx. $R^{\smile}$ x ⊓ $T^{\smile}$ x)"*

The coequalizer of two relations can be stated in terms of pushout.

lemma *"relCoequalizer = W $\circ_2$ relPushout"*

The notion of coequalizer for relations corresponds to (and generalizes) the functional counterpart.

lemma *"relCoequalizer (asRel f) (asRel g) = coequalizer f g"*
lemma *"totalFunction R ⟹ totalFunction T ⟹ coequalizer (asFun R) (asFun T) = relCoequalizer R T"*

7.7.7 Diagonal Elements

The notion of diagonal (aka. reflexive) elements of an endorelation is the relational counterpart to the notion of fixed-points of an endofunction. It corresponds to the W combinator.

abbreviation *(input) diagonal::"ERel('a) ⇒ Set('a)" ("Δ")*
  where *"Δ ≡ W"*

lemma *"Δ R x = R x x"*

lemma *"Δ (asRel f) = FP f"*
lemma *"totalFunction R ⟹ FP (asFun R) = Δ R"*

Analogously, the notion of anti-diagonal (aka. irreflexive) elements of an endorelation (notation: Δ^-) is the relational counterpart to the notion of non-fixed-points of an endofunction.

lemma *"Δ^- = $-^r\Delta$"*
lemma *"Δ^- = Δ ∘ $-^r$"*
lemma *"Δ^- R x = (¬R x x)"*
lemma *"Δ^- = − ∘ Δ"*
lemma *"Δ^- R = −(Δ R)"*

lemma *"Δ^- (asRel f) = nFP f"*
lemma *"totalFunction R ⟹ nFP (asFun R) = Δ^- R"*

7.8 Relation-based Set-Operations

We can extend the definitions of the (pre)image set-operator from functions to relations together with their "dual" counterparts.

definition *rightImage::"Rel('a,'b) ⇒ SetOp('a,'b)"*
  where *"rightImage ≡ C (B_{20} (⊓) C)"*
definition *leftImage::"Rel('a,'b) ⇒ SetOp('b,'a)"*
  where *"leftImage ≡ C (B_{20} (⊓) A)"*
definition *rightDualImage::"Rel('a,'b) ⇒ SetOp('a,'b)"*

```
  where "rightDualImage ≡ C (B₂₀ (⊆) C)"
definition leftDualImage::"Rel('a,'b) ⇒ SetOp('b,'a)"
  where "leftDualImage ≡ C (B₂₀ (⊆) A)"

declare rightImage_def[rel_defs] leftImage_def[rel_defs] rightDualImage_def[rel_defs] leftDualImage_c


notation(input) rightImage ("_-rightImage") and  leftImage ("_-leftImage") and
                rightDualImage ("_-rightDualImage") and  leftDualImage ("_-leftDualImage")

lemma "R-rightImage A = (λb. R˘ b ⊓ A)"
lemma "R-leftImage B = (λa. R a ⊓ B)"
lemma "R-rightDualImage A = (λb. R˘ b ⊆ A)"
lemma "R-leftDualImage B = (λa. R a ⊆ B)"

lemma "R-rightImage A b = (∃a. R a b ∧ A a)"
lemma "R-leftImage B a = (∃b. R a b ∧ B b)"
lemma "R-rightDualImage A b = (∀a. R a b → A a)"
lemma "R-leftDualImage B a = (∀b. R a b → B b)"
```

Convenient characterizations in terms of big-union and big-intersection.

```
lemma rightImage_def2: "rightImage = ⋃ ∘₂ image"
lemma leftImage_def2: "leftImage = ⋃ ∘₂ (image ∘ ˘)"
lemma rightDualImage_def2: "rightDualImage = ⋂ ∘₂ (B₁₁ image −ʳ −)"
lemma leftDualImage_def2: "leftDualImage = ⋂ ∘₂ (B₁₁ image ∼ −)"

lemma "R-rightImage A = ⋃⦇R A⦈"
lemma "R-leftImage B = ⋃⦇R˘ B⦈"
lemma "R-rightDualImage A =  ⋂⦇R⁻ −A⦈"
lemma "R-leftDualImage B =  ⋂⦇R∼ −B⦈"
```

As expected, relational right- resp. left-image correspond to functional image resp. preimage.

```
lemma "rightImage (asRel f) = image f"
lemma "leftImage (asRel f) = preimage f"
lemma "totalFunction R ⟹ image (asFun R) = rightImage R"
lemma "totalFunction R ⟹ preimage (asFun R) = leftImage R"
```

Clearly, each direction (right/left) uniquely determines the other (its transpose).

```
lemma rightImage_defT: "R-rightImage = R˘-leftImage"
lemma leftImage_defT: "R-leftImage = R˘-rightImage"
lemma rightDualImage_defT: "R-rightDualImage = R˘-leftDualImage"
lemma leftDualImage_defT: "R-leftDualImage = R˘-rightDualImage"
```

Following operators (aka. "polarities") are inspired by (and generalize) the notion of upper/lower bounds of a set wrt. an ordering. They are defined here for relations in general.

```
definition rightBound::"Rel('a,'b) ⇒ SetOp('a,'b)"
  where "rightBound ≡ C (B₂₀ (⊇) C)"
definition leftBound::"Rel('a,'b) ⇒ SetOp('b,'a)"
  where "leftBound ≡ C (B₂₀ (⊇) A)"
definition rightDualBound::"Rel('a,'b) ⇒ SetOp('a,'b)"
  where  "rightDualBound ≡ C (B₂₀ (Ψ₂ (⊓) −) C)"
definition leftDualBound::"Rel('a,'b) ⇒ SetOp('b,'a)"
  where  "leftDualBound ≡ C (B₂₀ (Ψ₂ (⊓) −) A)"

declare rightBound_def[rel_defs] leftBound_def[rel_defs] rightDualBound_def[rel_defs] leftDualBound_c

notation(input) rightBound ("_-rightBound") and  leftBound ("_-leftBound") and
                rightDualBound ("_-rightDualBound") and  leftDualBound ("_-leftDualBound")
```

lemma *"R-rightBound A = (λb. A ⊆ R⌣ b)"*
lemma *"R-leftBound B = (λa. B ⊆ R a)"*
lemma *"R-rightDualBound A = (λb. −(R⌣ b) ⊓ −A)"*
lemma *"R-leftDualBound B = (λa. −(R a) ⊓ −B)"*

lemma *"R-rightBound A = (λb. ∀a. A a → R a b)"*
lemma *"R-leftBound B = (λa. ∀b. B b → R a b)"*
lemma *"R-rightDualBound A = (λb. ∃a. ¬R a b ∧ ¬A a)"*
lemma *"R-leftDualBound B = (λa. ∃b. ¬R a b ∧ ¬B b)"*

Alternative (more insightful?) definitions for dual-bounds.

lemma *rightDualBound_def': "rightDualBound = $-^r$ ∘ (C (B_{20} (⊔) C))"*
lemma *leftDualBound_def': "leftDualBound = $-^r$ ∘ (C (B_{20} (⊔) A))"*

lemma *"R-rightDualBound A = −(λb. R⌣ b ⊔ A)"*
lemma *"R-leftDualBound B = −(λa. R a ⊔ B)"*

Convenient characterizations in terms of big-union and big-intersection.

lemma *rightBound_def2: "rightBound = ⋂ $\circ_2$ image"*
lemma *leftBound_def2: "leftBound = ⋂ $\circ_2$ (image ∘ ⌣)"*
lemma *rightDualBound_def2: "rightDualBound = ⋃ $\circ_2$ (B_{11} image $-^r$ −)"*
lemma *leftDualBound_def2: "leftDualBound = ⋃ $\circ_2$ (B_{11} image ∼ −)"*

lemma *"R-rightBound A = ⋂⦇R A⦈"*
lemma *"R-leftBound B = ⋂⦇R⌣ B⦈"*
lemma *"R-rightDualBound A = ⋃⦇R^- −A⦈"*
lemma *"R-leftDualBound B = ⋃⦇$R^\sim$ −B⦈"*

Some particular properties of right and left bounds.

lemma *right_dual_hom: "R-rightBound(⋃S) = ⋂⦇R-rightBound S⦈"*
lemma *left_dual_hom: "R-leftBound(⋃S) = ⋂⦇R-leftBound S⦈"*

Note, however:

proposition *"R-rightBound(⋂S) = ⋃⦇R-rightBound S⦈"* **nitpick** — countermodel found
proposition *"R-leftBound(⋂S) = ⋃⦇R-leftBound S⦈"* **nitpick** — countermodel found

We have, rather:

lemma *"R-rightBound(⋂S) ⊇ ⋃⦇R-rightBound S⦈"*
lemma *"R-leftBound(⋂S) ⊇ ⋃⦇R-leftBound S⦈"*

Clearly, each direction (right/left) uniquely determines the other (its transpose).

lemma *rightBound_defT: "R-rightBound = R⌣-leftBound"*
lemma *leftBound_defT: "R-leftBound = R⌣-rightBound"*
lemma *rightBoundDual_defT: "R-rightDualBound = R⌣-leftDualBound"*
lemma *leftBoundDual_defT: "R-leftDualBound = R⌣-rightDualBound"*

In fact, there exists a particular "relational duality" between images and bounds, as follows:

lemma *rightImage_dualR: "R-rightImage = (R^--rightBound)$^-$"*
lemma *leftImage_dualR: "R-leftImage = (R^--leftBound)$^-$"*
lemma *rightDualImage_dualR: "R-rightDualImage = (R^--rightDualBound)$^-$"*
lemma *leftDualImage_dualR: "R-leftDualImage = (R^--leftDualBound)$^-$"*
lemma *rightBound_dualR: "R-rightBound = (R^--rightImage)$^-$"*
lemma *leftBound_dualR: "R-leftBound = (R^--leftImage)$^-$"*
lemma *rightDualBound_dualR: "R-rightDualBound = (R^--rightDualImage)$^-$"*
lemma *leftDualBound_dualR: "R-leftDualBound = (R^--leftDualImage)$^-$"*

Finally, ranges can be expressed in terms of images and bounds.

lemma *leftRange_simp: "leftImage R $\mathfrak{U}$ = leftRange R"*
lemma *rightRange_simp: "rightImage R $\mathfrak{U}$ = rightRange R"*
lemma *leftDualRange_simp: "leftBound R $\mathfrak{U}$ = leftDualRange R"*
lemma *rightDualRange_simp: "rightBound R $\mathfrak{U}$ = rightDualRange R"*

declare *leftRange_simp[rel_simps] rightRange_simp[rel_simps]*
leftDualRange_simp[rel_simps] rightDualRange_simp[rel_simps]

7.9 Type-lifting and Monads

7.9.1 Set Monad

We can conceive of types of form *Set('a)*, i.e. *'a* $\Rightarrow$ *o*, as arising via an "environmentalization" (or "indexation") of the boolean type *o* by the type *'a* (i.e. as an instance of the environment monad discussed previously). Furthermore, we can adopt an alternative perspective and consider a constructor that returns the type of boolean "valuations" (or "classifiers") for objects of type *'a*. This type constructor comes with a monad structure too (and is also an applicative and a functor).

abbreviation *(input) unit_set::"'a $\Rightarrow$ Set('a)"*
  where *"unit_set $\equiv$ $\mathcal{Q}$"*
abbreviation *(input) fmap_set::"('a $\Rightarrow$ 'b) $\Rightarrow$ Set('a) $\Rightarrow$ Set('b)"*
  where *"fmap_set $\equiv$ image"*
abbreviation *(input) join_set::"Set(Set('a)) $\Rightarrow$ Set('a)"*
  where *"join_set $\equiv$ $\bigcup$"*
abbreviation *(input) ap_set::"Set('a $\Rightarrow$ 'b) $\Rightarrow$ Set('a) $\Rightarrow$ Set('b)"*
  where *"ap_set $\equiv$ rightImage $\circ$ intoRel"*
abbreviation *(input) rbind_set::"('a $\Rightarrow$ Set('b)) $\Rightarrow$ Set('a) $\Rightarrow$ Set('b)"*
  where *"rbind_set $\equiv$ rightImage"* — reversed bind

We define the customary bind operation as "flipped" *rbind* (which seems more intuitive).

abbreviation *bind_set::"Set('a) $\Rightarrow$ ('a $\Rightarrow$ Set('b)) $\Rightarrow$ Set('b)"*
  where *"bind_set $\equiv$ C rbind_set"*

Some properties of monads in general.

lemma *"rbind_set = join_set $\circ_2$ fmap_set"*
lemma *"join_set = rbind_set I"*

Some properties of this particular monad.

lemma *"ap_set = $\bigcup^r$ $\circ$ (image image)"*

Verifies compliance with the monad laws.

lemma *"monadLaw1 unit_set bind_set"*
lemma *"monadLaw2 unit_set bind_set"*
lemma *"monadLaw3 bind_set"*

7.9.2 Relation Monad

In fact, the *Rel('a,'b)* type constructor also comes with a monad structure, which can be seen as a kind of "monad composition" of the environment monad with the set monad.

abbreviation *(input) unit_rel::"'a $\Rightarrow$ Rel('b,'a)"*
  where *"unit_rel $\equiv$ K $\circ$ $\mathcal{Q}$"*
abbreviation *(input) fmap_rel::"('a $\Rightarrow$ 'b) $\Rightarrow$ Rel('c,'a) $\Rightarrow$ Rel('c,'b)"*
  where *"fmap_rel $\equiv$ B $\circ$ image"*
abbreviation *(input) join_rel::"Rel('c,Rel('c,'a)) $\Rightarrow$ Rel('c,'a)"*
  where *"join_rel $\equiv$ W $\circ$ (B $\bigcup^r$)"*
abbreviation *(input) ap_rel::"Rel('c, 'a $\Rightarrow$ 'b) $\Rightarrow$ Rel('c,'a) $\Rightarrow$ Rel('c,'b)"*
  where *"ap_rel $\equiv$ $\mathbf{\Phi}_{21}$ (rightImage $\circ$ intoRel)"*

abbreviation *(input) rbind_rel::"('a ⇒ Rel('c,'b)) ⇒ Rel('c,'a) ⇒ Rel('c,'b)"*
where *"rbind_rel ≡ (*Φ_{21} *rightImage) ∘* C*"* — reversed bind

Again, we define the bind operation as "flipped" rbind

abbreviation *bind_rel::"Rel('c,'a) ⇒ ('a ⇒ Rel('c,'b)) ⇒ Rel('c,'b)"*
where *"bind_rel ≡* C *rbind_rel"*

Some properties of monads in general.

lemma *"rbind_rel = join_rel* $\circ_2$ *fmap_rel"*
lemma *"join_rel = rbind_rel* I*"*

Note that for the relation monad we have:

lemma *"unit_rel =* B *unit_env unit_set"*
lemma *"fmap_rel =* B *fmap_env fmap_set"*
lemma *"ap_rel =* Φ_{21} *ap_set"*
lemma *"rbind_rel =* B *(*C B C*)* Φ_{21} *rbind_set"*

Finally, verify compliance with the monad laws.

lemma *"monadLaw1 unit_rel bind_rel"*
lemma *"monadLaw2 unit_rel bind_rel"*
lemma *"monadLaw3 bind_rel"*

end

8 Endorelations

Endorelations are particular cases of relations where the relata have the same type.

theory *endorelations*
imports *relations*
begin

named_theorems *endorel_defs*

8.1 Intervals and Powers

8.1.1 Intervals

We now conveniently encode a notion of "interval" (wrt given relation R) as the set of elements that lie between or "interpolate" a given pair of points (seen as "boundaries").

definition *interval::"ERel('a) ⇒ 'a ⇒ 'a ⇒ Set('a)" ("_-interval")*
where *"interval ≡* W *interpolants"*

And also introduce a convenient dual notion.

definition *dualInterval::"ERel('a) ⇒ 'a ⇒ 'a ⇒ Set('a)" ("_-dualInterval")*
where *"dualInterval ≡* W *dualInterpolants"*

declare *interval_def[endorel_defs] dualInterval_def[endorel_defs]*

lemma *"R-interval a b = (λc. R a c ∧ R c b)"*
lemma *"R-dualInterval a b = (λc. R a c ∨ R c b)"*

8.1.2 Powers

The set of all powers (via iterated composition) for a given endorelation can be defined in two ways, depending whether we want to include the "zero-power" (i.e. $R^0 = \mathcal{Q}$) or not.

definition *relPower::"ERel(ERel('a))"*
where *"relPower ≡* Φ_{21} *indSet*$_1$ $\mathcal{Q}$ *(*$\circ^r$*)"*

definition *relPower0::"ERel(ERel('a))"*
  where *"relPower0 ≡ B (indSet$_1$ (𝒬 𝒬)) (∘^{r})"*

declare *relPower_def[endorel_defs] relPower0_def[endorel_defs]*

lemma *"relPower R = indSet$_1$ {R} ((∘^{r}) R)"*
lemma *relPower_def2: "relPower R T = (∀S. (∀H. S H → S (R ∘^{r} H)) → S R → S T)"*

lemma *"relPower0 R = indSet$_1$ {𝒬} ((∘^{r}) R)"*
lemma *relPower0_def2: "relPower0 R T = (∀S. (∀H. S H → S (R ∘^{r} H)) → S 𝒬 → S T)"*

Definitions work as intended:

proposition *"relPower R 𝒬"* **nitpick** — countermodel found
lemma *"relPower R R"*
lemma *"relPower R (R ∘^{r} R)"*
lemma *"relPower R (R ∘^{r} R ∘^{r} R ∘^{r} R ∘^{r} R ∘^{r} R ∘^{r} R)"*
lemma *"relPower0 R 𝒬"*
lemma *"relPower0 R R"*
lemma *"relPower0 R (R ∘^{r} R)"*
lemma *"relPower0 R (R ∘^{r} R ∘^{r} R ∘^{r} R ∘^{r} R ∘^{r} R ∘^{r} R)"*

lemma *relPower_ind: "relPower R T ⟹ relPower R (R ∘^{r} T)"*
lemma *relPower0_ind: "relPower0 R T ⟹ relPower0 R (R ∘^{r} T)"*

8.2 Properties and Operations

8.2.1 Reflexivity and Irreflexivity

Relations are called reflexive (aka. diagonal) resp. irreflexive (aka. antidiagonal) when they are larger than identity/equality resp. smaller than difference/disequality.

definition *reflexive::"Set(ERel('a))"*
  where ‹*reflexive ≡ (⊆^{r}) 𝒬*›
definition *irreflexive::"Set(ERel('a))"*
  where ‹*irreflexive ≡ (⊇^{r}) 𝒟*›

declare *reflexive_def[endorel_defs] irreflexive_def[endorel_defs]*

lemma ‹*reflexive R = 𝒬 ⊆^{r} R*›
lemma ‹*irreflexive R = R ⊆^{r} 𝒟*›

Both properties are "complementary" in the expected way.

lemma *reflexive_compl: "reflexive R$^-$ = irreflexive R"*
lemma *irreflexive_compl: "irreflexive R$^-$ = reflexive R"*

An alternative pair of definitions.

lemma *reflexive_def2: "reflexive = ∀ ∘ Δ"*
lemma *irreflexive_def2: "irreflexive = ∄ ∘ Δ"*
lemma *"reflexive R = (∀a. R a a)"*
lemma *"irreflexive R = (∀a. ¬R a a)"*

We can naturally obtain a reflexive resp. irreflexive relations via the following operators.

definition *reflexiveClosure::"ERel('a) ⇒ ERel('a)"*
  where *"reflexiveClosure ≡ (∪^{r}) 𝒬"*
definition *irreflexiveInterior::"ERel('a) ⇒ ERel('a)"*
  where *"irreflexiveInterior ≡ (∩^{r}) 𝒟"*

declare *reflexiveClosure_def[endorel_defs] irreflexiveInterior_def[endorel_defs]*

lemma *"reflexiveClosure R = (R ∪^{r} 𝒬)"*

lemma *"irreflexiveInterior R = (R $\cap^r$ $\mathcal{D}$)"*

The operators reflexive closure and irreflexive interior are duals wrt. relation-complement.

lemma *"irreflexiveInterior (R^-) = (reflexiveClosure R)$^-$"*
lemma *"reflexiveClosure (R^-) = (irreflexiveInterior R)$^-$"*

All reflexive resp. irreflexive relations arise via their corresponding closure resp. interior operator.

lemma *reflexive_def3: "reflexive = range reflexiveClosure"*
lemma *irreflexive_def3: "irreflexive = range irreflexiveInterior"*

We now check that these unary relation-operators are indeed closure resp. interior operators.

lemma *‹($\subseteq^r$)-CLOSURE reflexiveClosure›*
lemma *‹($\subseteq^r$)-INTERIOR irreflexiveInterior›*

Thus, reflexive resp. irreflexive relations are the fixed points of the corresponding operators.

lemma *reflexive_def4: ‹reflexive = FP reflexiveClosure›*
lemma *irreflexive_def4: ‹irreflexive = FP irreflexiveInterior›*

The smallest reflexive super-relation resp. largest irreflexive subrelation.

lemma *"reflexiveClosure R = $\bigcap^r$(λT. R $\subseteq^r$ T $\wedge$ reflexive T)"* — proof by external provers
lemma *"irreflexiveInterior R = $\bigcup^r$(λT. T $\subseteq^r$ R $\wedge$ irreflexive T)"* — proof by external provers

8.2.2 Strong-identity, Weak-difference, and Tests

We call relations strong-identities (aka. coreflexive, "tests") resp. weak-differences when they are smaller than identity/equality resp. larger than difference/disequality.

definition *strongIdentity::"Set(ERel('a))"*
  where *"strongIdentity $\equiv$ ($\supseteq^r$) $\mathcal{Q}$"*
definition *weakDifference::"Set(ERel('a))"*
  where *"weakDifference $\equiv$ ($\subseteq^r$) $\mathcal{D}$"*

declare *strongIdentity_def[endorel_defs] weakDifference_def[endorel_defs]*

lemma *‹strongIdentity R = R $\subseteq^r$ $\mathcal{Q}$›*
lemma *‹weakDifference R = $\mathcal{D}$ $\subseteq^r$ R›*

Elements in strong-identities are only related to themselves (may be related to none).

lemma *strongIdentity_def2: "strongIdentity R = ($\forall$a. R a $\subseteq$ {a})"*

Elements in weak-differences are related to (at least) everyone else (may be also related to themselves).

lemma *weakDifference_def2: "weakDifference R = ($\forall$a. $\{\!|a|\!\}$ $\subseteq$ R a)"*

They are "weaker" than identity resp. difference since they may feature anti-diagonal resp. diagonal elements.

proposition *"strongIdentity R $\wedge$ $\neg$R a a"* **nitpick***[satisfy]* — satisfying model found
proposition *"weakDifference R $\wedge$ R a a"* **nitpick***[satisfy]* — satisfying model found

We can naturally obtain strong-identities resp. weak-differences via the following operators.

definition *strongIdentityInterior::"ERel('a) $\Rightarrow$ ERel('a)" ("(_)$^!$")*
  where *"strongIdentityInterior $\equiv$ ($\cap^r$) $\mathcal{Q}$"*
definition *weakDifferenceClosure::"ERel('a) $\Rightarrow$ ERel('a)" ("(_)$^?$")*
  where *"weakDifferenceClosure $\equiv$ ($\cup^r$) $\mathcal{D}$"*

declare *weakDifferenceClosure_def[endorel_defs] strongIdentityInterior_def[endorel_defs]*

lemma *"strongIdentityInterior R = (R $\cap^r$ $\mathcal{Q}$)"*

lemma "*weakDifferenceClosure R = (R* $\cup^r$ $\mathcal{D}$*)*"

The notions of strong-identity-interior and weak-difference-closure are duals wrt. relation-complement.

lemma "$R^{-?}$ = $R^{!-}$"
lemma "$R^{-!}$ = $R^{?-}$"

All strong-identity resp. weak-difference relations arise via their corresponding interior resp. closure operator.

lemma *strongIdentity_def3: "strongIdentity = range strongIdentityInterior"*
lemma *weakDifference_def3: "weakDifference = range weakDifferenceClosure"*

We now check that these unary relation-operators are indeed closure resp. interior operators.

lemma ‹($\subseteq^r$)*-INTERIOR strongIdentityInterior*›
lemma ‹($\subseteq^r$)*-CLOSURE weakDifferenceClosure*›

Thus, strong-identity resp. weak-difference relations are the fixed points of the corresponding operators.

lemma *strongIdentity_def4:* ‹*strongIdentity = FP strongIdentityInterior*›
lemma *weakDifference_def4:* ‹*weakDifference = FP weakDifferenceClosure*›

The largest strong-identity sub-relation resp. smallest weak-difference super-relation.

lemma "$R^!$ = $\bigcup^r(\lambda T.\ T \subseteq^r R \wedge$ *strongIdentity T*)" — proof by external provers
lemma "$R^?$ = $\bigcap^r(\lambda T.\ R \subseteq^r T \wedge$ *weakDifference T*)" — proof by external provers

A convenient way of disguising sets as endorelations (cf. dynamic logics and program algebras).

definition *test::"Set('a)* $\Rightarrow$ *ERel('a)"*
  where "*test* $\equiv$ *strongIdentityInterior* $\circ$ K"
definition *dualtest::"Set('a)* $\Rightarrow$ *ERel('a)"*
  where "*dualtest* $\equiv$ *weakDifferenceClosure* $\circ$ K"

declare *test_def[endorel_defs] dualtest_def[endorel_defs]*

lemma *test_def2: "test = strongIdentityInterior* $\circ$ (W ($\times$))"
lemma "*test A* = $(A \times A)^!$"
lemma "*test A* = $\mathcal{Q} \cap^r (A \times A)$"

lemma *dualtest_def2: "dualtest = weakDifferenceClosure* $\circ$ (W ($\times$))"
lemma "*dualtest A* = $(A \times A)^?$"
lemma "*dualtest A* = $\mathcal{D} \cup^r (A \times A)$"

lemma *test_def3: "test = strongIdentityInterior* $\circ$ *leftCylinder"*
lemma *dualtest_def3: "dualtest = weakDifferenceClosure* $\circ$ *leftCylinder"*

lemma *test_def4: "test = strongIdentityInterior* $\circ$ *rightCylinder"*
lemma *dualtest_def4: "dualtest = weakDifferenceClosure* $\circ$ *rightCylinder"*

Both are duals wrt relation/set complement, as expected.

lemma *test_dual1:* "$(test\ A)^-$ = *dualtest* $(-A)$"
lemma *test_dual2:* "$(dualtest\ A)^-$ = *test* $(-A)$"

Both test resp. dual-test act as (full) inverses of diagonal (assuming strong-identity resp. weak-difference)

lemma "Δ (*test A*) = *A*"
lemma "Δ (*dualtest A*) = *A*"
lemma "*strongIdentity A* $\Longrightarrow$ *test* (Δ *A*) = *A*"
lemma "*weakDifference A* $\Longrightarrow$ *dualtest* (Δ *A*) = *A*"

In fact, all strong-identities resp. weak-differences arise via the test resp- dual-test operators (applied to some set).

lemma *strongIdentity_def5: "strongIdentity = range test"*
lemma *weakDifference_def5: "weakDifference = range dualtest"*

8.2.3 Seriality and Quasireflexivity

Following usual practice, we shal call "serial" those endorelations that are left-total.

abbreviation*(input) serial::"Set(ERel('a))"*
where *"serial ≡ leftTotal"*

The following "weakening" of reflexivity does not imply seriality (i.e. left-totality).

definition *quasireflexive::"Set(ERel('a))"*
where *"quasireflexive ≡ leftRange ⊑ Δ"*

declare *quasireflexive_def[endorel_defs]*

lemma *"quasireflexive R = leftRange R ⊆ Δ R"*
lemma *"quasireflexive R = (∀x. ∃(R x) → R x x)"*

We have in fact that:

lemma *reflexive_def5: "reflexive R = (serial R ∧ quasireflexive R)"*

The quasireflexive closure of a relation: elements related to someone else become related to themselves.

definition *quasireflexiveClosure::"ERel('a) ⇒ ERel('a)"*
where *"quasireflexiveClosure ≡ W ((∪^{r}) ∘ ((∩^{r}) Q) ∘ ((×) 𝔘) ∘ leftRange)"*

The serial extension of a relation: elements not related to anyone else become related to themselves.

definition *serialExtension::"ERel('a) ⇒ ERel('a)"*
where *"serialExtension ≡ W ((∪^{r}) ∘ ((∩^{r}) Q) ∘ ((×) 𝔘) ∘ − ∘ leftRange)"*

declare *serialExtension_def[endorel_defs] quasireflexiveClosure_def[endorel_defs]*

lemma *"quasireflexiveClosure R = (R ∪^{r} (Q ∩^{r} (𝔘 × (leftRange R))))"*
lemma *"serialExtension R = (R ∪^{r} (Q ∩^{r} (𝔘 × −(leftRange R))))"*

lemma *"serial (serialExtension R)"*
lemma *"quasireflexive (quasireflexiveClosure R)"*

8.2.4 Symmetry, Connectedness, and co.

We introduce two ways of "symmetrizing" a given relation R: The symmetric interior and closure operations. The intuition is that the symmetric interior/closure of R intersects/merges R with its converse, thus generating R's largest/smallest symmetric sub/super-relation.

definition *symmetricInterior::"ERel('a) ⇒ ERel('a)"*
where *"symmetricInterior ≡ S (∩^{r}) ⌣"* — aka. symmetric part of R
definition *symmetricClosure::"ERel('a) ⇒ ERel('a)"*
where *"symmetricClosure ≡ S (∪^{r}) ⌣"*

declare *symmetricInterior_def[endorel_defs] symmetricClosure_def[endorel_defs]*

lemma *"symmetricInterior R = R ∩^{r} (R$^⌣$)"*
lemma *"symmetricClosure R = R ∪^{r} (R$^⌣$)"*

```
lemma "symmetricInterior R = (λx y. R x y ∧ R y x)"
lemma "symmetricClosure R  = (λx y. R x y ∨ R y x)"

lemma symmetricInterior_def2: "symmetricInterior = W ∘ interval"
lemma symmetricClosure_def2:  "symmetricClosure  = W ∘ dualInterval"

lemma "symmetricInterior R a = (λx. R-interval a a x)"
lemma "symmetricClosure R a = (λx. R-dualInterval a a x)"
```

The notions of symmetric closure and symmetric interior are duals wrt. relation-complement.

```
lemma "symmetricInterior (R^-) = (symmetricClosure R)^-"
lemma "symmetricClosure (R^-) = (symmetricInterior R)^-"
```

The properties of (ir)reflexivity and co(ir)reflexivity are preserved by symmetric interior and closure.

```
lemma reflexive_si: ‹reflexive R = reflexive (symmetricInterior R)›
lemma weakDifference_si: ‹weakDifference R = weakDifference (symmetricInterior R)›
lemma strongIdentity_sc: ‹strongIdentity R = strongIdentity (symmetricClosure R)›
lemma irreflexive_sc: ‹irreflexive R = irreflexive (symmetricClosure R)›
```

A relation is symmetric when it is a fixed-point of the symmetric interior or closure.

```
definition symmetric::"Set(ERel('a))"
  where ‹symmetric ≡ FP symmetricInterior›

lemma symmetric_defT: "symmetric = FP symmetricClosure"

declare symmetric_def[endorel_defs]

lemma symmetric_def2:  ‹symmetric = S (⊆^r) ⌣›
lemma symmetric_defT2: ‹symmetric = S (⊇^r) ⌣›

lemma symmetric_reldef:   ‹symmetric R = R  ⊆^r R⌣›
lemma symmetric_reldefT:  ‹symmetric R = R⌣ ⊆^r R›
lemma ‹symmetric R = (∀a b. R a b → R b a)›

lemma "symmetricInterior R = ⋃^r(λT. T ⊆^r R ∧ symmetric T)" — proof by external provers
lemma "symmetricClosure R  = ⋂^r(λT. R ⊆^r T ∧ symmetric T)" — proof by external provers

lemma "symmetric R^- = symmetric R"
```

All symmetric relations arise via their interior or closure operator.

```
lemma symmetric_def3:  "symmetric = range symmetricInterior"
lemma symmetric_defT3: "symmetric = range symmetricClosure"
```

The following operation takes a relation R and returns its "strict" part, which is always an asymmetric sub-relation (though not a maximal one in general).

```
definition asymmetricContraction::"ERel('a) ⇒ ERel('a)" ("(_)^#")
  where "asymmetricContraction ≡ S (∩^r) ∼"
```

Analogously, this extends a relation R towards a connected super-relation (not minimal in general).

```
definition connectedExpansion::"ERel('a) ⇒ ERel('a)" ("(_)^♭")
  where "connectedExpansion ≡ S (∪^r) ∼"

declare asymmetricContraction_def[endorel_defs] connectedExpansion_def[endorel_defs]

lemma "R^# = R ∩^r (R^∼)"
lemma "R^# = (λa b. R a b ∧ ¬R b a)"
```

lemma "$R^\flat$ = R $\cup^r$ ($R^\sim$)"
lemma "$R^\flat$ = (λa b. R a b ∨ ¬R b a)"

definition *asymmetric::"Set(ERel('a))"*
  where "asymmetric ≡ FP asymmetricContraction"
definition *connected::"Set(ERel('a))"*
  where ‹connected ≡ FP connectedExpansion› — aka. "linear" or "total" in order theory

declare *asymmetric_def[endorel_defs] connected_def[endorel_defs]*

lemma *asymmetric_def2:* ‹asymmetric = S ($\subseteq^r$) ∼›
lemma *asymmetric_reldef:* ‹asymmetric R = R $\subseteq^r$ $R^\sim$›
lemma "asymmetric R = (∀a b. R a b → ¬R b a)"

lemma *connected_def2:* ‹connected = S ($\supseteq^r$) ∼›
lemma *connected_reldef:* ‹connected R = $R^\sim$ $\subseteq^r$ R›
lemma ‹connected R = (∀a b. ¬R b a → R a b)›

lemma "connected R^- = asymmetric R"
lemma "asymmetric R^- = connected R"

Connectedness resp. asymmetry entail reflexivity resp. irreflexivity.

lemma "connected R ⟹ reflexive R"
lemma "asymmetric R ⟹ irreflexive R"

lemma *connected_def3:* "connected R = $\forall^2$(symmetricClosure R)"
lemma *asymmetric_def3:* "asymmetric R = $\nexists^2$(symmetricInterior R)"

All asymmetric resp. connected relations arise via their corresponding interior resp. closure operator.

lemma *asymmetric_def4:* "asymmetric = range asymmetricContraction"
lemma *connected_def4:* "connected = range connectedExpansion"

An alternative (more intuitive?) definition of connectedness.

lemma *connected_def5:* ‹connected = S ($\sqcup^r$) $\smile$›
lemma *connected_reldef5:* ‹connected R = R $\sqcup^r$ $R^\smile$›
lemma ‹connected R = (∀a b. R b a ∨ R a b)›

The asymmetric-contraction and connected-expansion operators are duals wrt. relation-complement.

lemma "$R^{\flat-}$ = $R^{-\#}$"
lemma "$R^{\#-}$ = $R^{-\flat}$"

8.2.5 Antisymmetry, Semiconnectedness, and co.

definition *antisymmetric::"Set(ERel('a))"*
  where "antisymmetric ≡ strongIdentity ∘ symmetricInterior"
definition *semiconnected::"Set(ERel('a))"*
  where "semiconnected ≡ weakDifference ∘ symmetricClosure"

declare *antisymmetric_def[endorel_defs] semiconnected_def[endorel_defs]*

lemma ‹antisymmetric R = strongIdentity (symmetricInterior R)›
lemma ‹antisymmetric R = symmetricInterior R $\subseteq^r$ Q›
lemma *antisymmetric_reldef:* ‹antisymmetric R = R $\cap^r$ ($R^\smile$) $\subseteq^r$ Q›
lemma ‹antisymmetric R = (∀a b. R a b ∧ R b a ⟶ a = b)›

lemma ‹semiconnected R = weakDifference (symmetricClosure R)›

lemma ‹*semiconnected R =* $\mathcal{D} \subseteq^r$ *symmetricClosure R*›
lemma *semiconnected_reldef: "semiconnected R =* $\mathcal{D} \subseteq^r R \cup^r (R^{\smile})$*"*
lemma *"semiconnected R = (*$\forall$ *a b. a* $\neq$ *b* $\to$ *R a b* $\vee$ *R b a)"*

A relation is antisymmetric/semiconnected iff its complement is semiconnected/antisymmetric.

lemma *antisymmetric_defN: "antisymmetric R = semiconnected* R^-*"*
lemma *semiconnected_defN: "semiconnected R = antisymmetric* R^-*"*

lemma *asymmetric_def5: "asymmetric R = (irreflexive R* $\wedge$ *antisymmetric R)"*

A relation is called (co)skeletal when its symmetric interior (closure) is the (dis)equality relation, inspired by category theory where categories are skeletal when isomorphic objects are identical.

definition *skeletal::"Set(ERel('a))"*
  where ‹*skeletal* $\equiv$ *(*$\mathcal{Q}$ $\mathcal{Q}$*)* $\circ$ *symmetricInterior*›
definition *coskeletal::"Set(ERel('a))"*
  where ‹*coskeletal* $\equiv$ *(*$\mathcal{Q}$ $\mathcal{D}$*)* $\circ$ *symmetricClosure*›

declare *skeletal_def[endorel_defs] coskeletal_def[endorel_defs]*

lemma *"skeletal R = (*$\mathcal{Q}$ *= symmetricInterior R)"*
lemma *"coskeletal R = (*$\mathcal{D}$ *= symmetricClosure R)"*

lemma *"skeletal R = coskeletal* R^-*"*
lemma *"coskeletal R = skeletal* R^-*"*

Alternative definitions in terms of other relational properties.

lemma *skeletal_def2: "skeletal R = (antisymmetric R* $\wedge$ *reflexive R)"*
lemma *coskeletal_def2: "coskeletal R = (semiconnected R* $\wedge$ *irreflexive R)"*

8.2.6 Transitivity, Denseness, Quasitransitivity, and co.

Every pair of elements x and y that can be connected by an element z in between are (un)related.

definition *transitive::"Set(ERel('a))"*
  where ‹*transitive* $\equiv$ S *(*$\supseteq^r$*)* (W *(*$\circ^r$*))*›
definition *antitransitive::"Set(ERel('a))"*
  where ‹*antitransitive* $\equiv$ $\mathbf{\Phi}_{21}$ *(*$\supseteq^r$*)* $-^r$ (W *(*$\circ^r$*))*›

declare *transitive_def[endorel_defs] antitransitive_def[endorel_defs]*

lemma *transitive_reldef:* ‹*transitive R = (R* $\circ^r$ *R)* $\subseteq^r$ *R*›
lemma *antitransitive_reldef:* ‹*antitransitive R = (R* $\circ^r$ *R)* $\subseteq^r$ R^-›

Alternative convenient definitions.

lemma *transitive_def2:* ‹*transitive R = (*$\forall$ *a b c. R a c* $\wedge$ *R c b* $\to$ *R a b)*›
lemma *antitransitive_def2:* ‹*antitransitive R = (*$\forall$ *a b c. R a c* $\wedge$ *R c b* $\to$ $\neg$*R a b)*›

Relationship between antitransitivity and irreflexivity.

lemma *"antitransitive R* $\Longrightarrow$ *irreflexive R"*
lemma *"leftUnique R* $\vee$ *rightUnique R* $\Longrightarrow$ *antitransitive R = irreflexive R"*

Every pair of (un)related elements x and y can be connected by an element z in between.

definition *dense::"Set(ERel('a))"*
  where ‹*dense* $\equiv$ S *(*$\subseteq^r$*)* (W *(*$\circ^r$*))*›
definition *pseudoClique::"Set(ERel('a))"* — i.e. a graph with diameter 2 (where cliques have diameter 1)

where ‹*pseudoClique* ≡ **Φ**$_{21}$ *(⊆^{r}) −^{r} (W (∘^{r}))*›

declare *dense_def[endorel_defs] pseudoClique_def[endorel_defs]*

lemma *dense_reldef:* ‹*dense R = R ⊆^{r} (R ∘^{r} R)*›
lemma *pseudoClique_reldef:* ‹*pseudoClique R = R$^-$ ⊆^{r} (R ∘^{r} R)*›

The above properties are preserved by transposition:

lemma *transitive_defT: "transitive R = transitive (R$^\smile$)"*
lemma *antitransitive_defT: "antitransitive R = antitransitive (R$^\smile$)"*
lemma *quasiDense_defT: "dense R = dense (R$^\smile$)"*
lemma *quasiClique_defT: "pseudoClique R = pseudoClique (R$^\smile$)"*

The above properties can be stated for the complemented relations in an analogous fashion.

lemma *transitive_compl_reldef:* ‹*transitive R$^-$ = R ⊆^{r} (R •^{r} R)*›
lemma *dense_compl_reldef:* ‹*dense R$^-$ = (R •^{r} R) ⊆^{r} R*›
lemma *antitransitive_compl_reldef:* ‹*antitransitive R$^-$ = R$^-$ ⊆^{r} (R •^{r} R)*›
lemma *pseudoClique_compl_reldef:* ‹*pseudoClique R$^-$ = (R •^{r} R) ⊆^{r} R$^-$*›

We can provide alternative definitions for the above relational properties in terms of intervals.

lemma ‹*transitive R = (∀a b. ∃(R-interval a b) → R a b)*›
lemma ‹*antitransitive R = (∀a b. ∃(R-interval a b) → R$^-$ a b)*›
lemma ‹*dense R = (∀a b. R a b → ∃(R-interval a b))*›
lemma ‹*pseudoClique R = (∀a b. R$^-$ a b → ∃(R-interval a b))*›

The following notions are often discussed in the literature (applied to strict relations/orderings).

abbreviation *(input)* ‹*quasiTransitive ≡ transitive ∘ asymmetricContraction*›
abbreviation *(input)* ‹*quasiAntitransitive ≡ antitransitive ∘ asymmetricContraction*›

lemma ‹*quasiTransitive R = (∀a b. ∃(R$^\#$-interval a b) → R$^\#$ a b)*›
lemma ‹*quasiAntitransitive R = (∀a b. ∃(R$^\#$-interval a b) → R$^{\#-}$ a b)*›

The "quasi" variants are weaker than their counterparts.

lemma *"transitive R ⟹ quasiTransitive R"*
lemma *"antitransitive R ⟹ quasiAntitransitive R"*

However, both variants coincide under the right conditions.

lemma *"antisymmetric R ⟹ quasiTransitive R = transitive R"*
lemma *"asymmetric R ⟹ quasiAntitransitive R = antitransitive R"*

lemma *quasiTransitive_defT: "quasiTransitive R = quasiTransitive (R$^\smile$)"*
lemma *quasiAntitransitive_defT: "quasiAntitransitive R = quasiAntitransitive (R$^\smile$)"*

lemma *quasitransitive_defN: "quasiTransitive R = quasiTransitive (R$^-$)"*
lemma *quasiintransitive_defN: "quasiAntitransitive R = quasiAntitransitive (R$^-$)"*

Symmetry entails both quasi-transitivity and quasi-antitransitivity.

lemma *"symmetric R ⟹ quasiTransitive R"*
lemma *"symmetric R ⟹ quasiAntitransitive R"*

The property of transitivity is closed under arbitrary infima (i.e. it is a "closure system").

lemma *"$\bigcap^r$-closed$_G$ transitive"*

Natural ways to obtain transitive relations resp. preorders.

definition *transitiveClosure::"ERel('a) ⇒ ERel('a)" ("_$^+$")*
where *"transitiveClosure ≡ $\bigcup^r$ ∘ relPower"*
definition *preorderClosure::"ERel('a) ⇒ ERel('a)" ("_*")* — aka. reflexive-transitive closure

where "*preoderClosure* ≡ $\bigcup^r$ ∘ *relPower0*"

declare *transitiveClosure_def [endorel_defs] preorderClosure_def [endorel_defs]*

lemma "R^+ = $\bigcup^r$(*relPower R*)"
lemma "R^* = $\bigcup^r$(*relPower0 R*)"

lemma *transitiveClosure_char*: "R^+ = $\bigcap^r$(λT. *transitive T* ∧ R $\subseteq^r$ T)" — proof by external provers

lemma "R^* = *reflexiveClosure* (R^+)" — proof by external provers

8.2.7 Euclideanness and co.

The relational properties of left-/right- euclideanness.

definition ‹*rightEuclidean* ≡ **S** ($\supseteq^r$) (**S** ($\circ^r$) $\smile$)›
definition ‹*leftEuclidean* ≡ **S** ($\supseteq^r$) (**Σ** ($\circ^r$) $\smile$)›

lemma *rightEuclidean_reldef*: "*rightEuclidean R* = R $\circ^r$ ($R^\smile$) $\subseteq^r$ R"
lemma *leftEuclidean_reldef*: "*leftEuclidean R* = ($R^\smile$) $\circ^r$ R $\subseteq^r$ R"

declare *rightEuclidean_def[endorel_defs] leftEuclidean_def[endorel_defs]*

lemma "*rightEuclidean R* = (∀ a b. (∃ c. R c a ∧ R c b) → R a b)"
lemma "*leftEuclidean R* = (∀ a b. (∃ c. R a c ∧ R b c) → R a b)"

lemma "*leftEuclidean R* = *rightEuclidean* $R^\smile$"

lemma "*symmetric R* ⟹ *rightEuclidean R* = *leftEuclidean R*"

Alternative convenient definitions.

lemma *rightEuclidean_def2*: ‹*rightEuclidean R* = (∀ a b c. R c a ∧ R c b → R a b)›
lemma *leftEuclidean_def2*: ‹*leftEuclidean R* = (∀ a b c. R a c ∧ R b c → R a b)›

Some interrelationships.

lemma "*leftEuclidean R* ⟹ *quasiTransitive R*"
lemma "*rightEuclidean R* ⟹ *quasiTransitive R*"
lemma "*connected R* ⟹ *rightEuclidean R* ⟹ *transitive R*"
lemma "*connected R* ⟹ *leftEuclidean R* ⟹ *transitive R*"
lemma "*symmetric R* ⟹ *leftEuclidean R* = *transitive R*"
lemma "*symmetric R* ⟹ *rightEuclidean R* = *transitive R*"
lemma "*reflexive R* ⟹ *rightEuclidean R* ⟹ *transitive R*"
lemma "*reflexive R* ⟹ *leftEuclidean R* ⟹ *transitive R*"
lemma "*leftEuclidean R* ⟹ *leftUnique R* = *antisymmetric R*"
lemma "*rightEuclidean R* ⟹ *rightUnique R* = *antisymmetric R*"

8.2.8 Equivalence, Equality and co.

Equivalence relations are their own kernels (when seen as set-valued functions).

definition "*equivalence* ≡ *FP kernel*"

lemma *equivalence_reldef*: "*equivalence R* = (R = $R^=$)"

declare *equivalence_def[endorel_defs]*

lemma "*equivalence R* = (∀ a b. R a b = (R a = R b))"

Alternative, traditional characterization in terms of other relational properties.

lemma *equivalence_char: "equivalence R = (reflexive R ∧ transitive R ∧ symmetric R)"*

In fact, equality $\mathcal{Q}$ is an equivalence relation (which means that $\mathcal{Q}$ is identical to its own kernel).

lemma *"equivalence $\mathcal{Q}$"*

This gives a kind of recursive definition of equality (of which we can make a simplification rule).

lemma *eq_kernel_simp: "$\mathcal{Q}^{=}$ = $\mathcal{Q}$"*

Equality has other alternative definitions. We can also make simplification rules out of them:

The intersection of all reflexive relations.

lemma *eq_refl_simp: "$\bigcap^r$ reflexive = $\mathcal{Q}^{=}$"*

Leibniz principle of identity of indiscernibles.

lemma *eq_leibniz_simp1: "(λa b. ∀P. P a ↔ P b) = $\mathcal{Q}^{=}$"* — symmetric version
lemma *eq_leibniz_simp2: "(λa b. ∀P. P a → P b) = $\mathcal{Q}^{=}$"* — simplified version

By extensionality, the above equation can be written as follows.

lemma *eq_filt_simp1: "(λa b. (λk. k a) ⊆ (λc. c b)) = $\mathcal{Q}^{=}$"*

Equality also corresponds to identity of generated principal filters.

lemma *eq_filt_simp2: "(λa b. (λk::Set(Set('a)). k a) = (λc. c b)) = $\mathcal{Q}^{=}$"*

Or, in terms of combinators

lemma *eq_filt_simp3: "(T::'a ⇒ Set(Set('a)))$^{=}$ = $\mathcal{Q}^{=}$"*

Finally, note that:

lemma *"(∀y::'a ⇒ o. y a = y b) ⟹ (∀y::'a ⇒ 'b. y a = y b)"* — external provers find a proof
proposition *"(∀y::'a ⇒ 'b. y a = y b) ⟹ (∀y::'a ⇒ o. y a = y b)"* **nitpick** — counterexample found

8.2.9 Orderings

definition *"preorder R ≡ reflexive R ∧ transitive R"*
definition *"partial_order R ≡ preorder R ∧ antisymmetric R"*

declare *preorder_def [endorel_defs] partial_order_def [endorel_defs]*

lemma *preorder_def2: "preorder R = (∀a b. R a b = (∀x. R b x → R a x))"*

lemma *partial_order_def2: "partial_order R = (skeletal R ∧ transitive R)"*

lemma *reflexive_symm: "reflexive R$^{\smile}$ = reflexive R"*
lemma *transitive_symm: "transitive R$^{\smile}$ = transitive R"*
lemma *antisymmetric_symm: "antisymmetric R$^{\smile}$ = antisymmetric R"*
lemma *skeletal_symm: "skeletal R$^{\smile}$ = skeletal R"*
lemma *preorder_symm: "preorder R$^{\smile}$ = preorder R"*
lemma *partial_order_symm: "partial_order R$^{\smile}$ = partial_order R"*

The subset and subrelation relations are partial orders.

lemma *subset_partial_order: "partial_order (⊆)"*
lemma *subrel_partial_order: "partial_order (⊆^{r})"*

Functional-power is a preorder.

lemma *funPower_preorder: "preorder funPower"* — proof by external provers

Relational-power is a preorder

lemma *relPower_preorder: "preorder relPower"*
lemma *relPower0_preorder: "preorder relPower0"*

However, relational-power is not antisymmetric (and thus not partially ordered), because we have:

proposition *"R = T ∘^{r} T ⟹ T = R ∘^{r} R ⟹ R = T"* **nitpick** *[card 'a=3]* — countermodel found

8.3 Endorelation-based Set-Operations

When talking about endorelations (orderings in particular) it is customary to employ the expressions "up" and "down" instead of "right" and "left" respectively. Similarly, we use expressions like "maximal/greatest" and "minimal/least" to mean "rightmost" and "leftmost" respectively.

We conveniently introduce the following alternative names for left resp. right bounds/images

notation *(input) leftBound ("lowerBound")* **and** *leftBound ("_-lowerBound")*
and *rightBound ("upperBound")* **and** *rightBound ("_-upperBound")*
and *leftImage ("downImage")* **and** *leftImage ("_-downImage")*
and *rightImage ("upImage")* **and** *rightImage ("_-upImage")*

8.3.1 Least and Greatest Elements

The set of least (leftmost) resp. greatest (rightmost) elements of a set wrt. an endorelation.

definition *least::"ERel('a) ⇒ SetEOp('a)"*
where *‹least ≡ (S (∩)) ∘ lowerBound›*
definition *greatest::"ERel('a) ⇒ SetEOp('a)"*
where *‹greatest ≡ (S (∩)) ∘ upperBound›*

notation *(input) least ("_-least")* **and** *greatest ("_-greatest")*

lemma *"R-least A = (λm. A m ∧ (∀x. A x → R m x))"*
lemma *"R-greatest A = (λm. A m ∧ (∀x. A x → R x m))"*

declare *least_def[endorel_defs] greatest_def[endorel_defs]*

lemma *greatest_defT: ‹R-greatest = R$^{\smile}$-least›*
lemma *least_defT: ‹R-least = R$^{\smile}$-greatest›*

8.3.2 Maximal and Minimal Elements

The set of minimal (resp. maximal) elements of a set A wrt. a relation R.

definition *min::"ERel('a) ⇒ SetEOp('a)"*
where *‹min ≡ least ∘ connectedExpansion›*
definition *max::"ERel('a) ⇒ SetEOp('a)"*
where *‹max ≡ greatest ∘ connectedExpansion›*

notation *(input) min ("_-min")* **and** *max ("_-max")*

lemma *"R-min A = (λm. A m ∧ (∀x. A x → R$^♭$ m x))"*

```
lemma "R-max A = (λm. A m ∧ (∀x. A x → R♭ x m))"

lemma ‹R-min = (λA. λm. A m ∧ (∀x. A x → R x m → R m x))›
lemma ‹R-max = (λA. λm. A m ∧ (∀x. A x → R m x → R x m))›

declare min_def[endorel_defs] max_def[endorel_defs]

lemma max_defT: ‹R-max = R˘-min›
lemma min_defT: ‹R-min = R˘-max›
```

Minimal and maximal elements generalize least and greatest elements respectively.

```
lemma "R-least A ⊆ R-min A"
lemma "R-greatest A ⊆ R-max A"
```

8.3.3 Least Upper- and Greatest Lower-Bounds

The (set of) least upper-bound(s) and greatest lower-bound(s) for a given set.

```
definition lub::"ERel('a) ⇒ SetEOp('a)"
  where "lub ≡ Φ₂₁ B least upperBound"
definition glb::"ERel('a) ⇒ SetEOp('a)"
  where "glb ≡ Φ₂₁ B greatest lowerBound"

notation(input) lub ("_-lub") and  glb ("_-glb")

lemma "R-lub =    (R-least) ∘ (R-upperBound)"
lemma "R-glb = (R-greatest) ∘ (R-lowerBound)"

declare lub_def[endorel_defs] glb_def[endorel_defs]

lemma lub_defT: "R-lub = R˘-glb"
lemma glb_defT: "R-glb = R˘-lub"
```

Moreover, when it comes to upper/lower bounds, least/greatest and glb/lub elements coincide.

```
lemma lub_def3: "R-lub S = R-glb (R-upperBound S)"
lemma glb_def3: "R-glb S = R-lub (R-lowerBound S)"

lemma lub_prop: "S ⊆ R-lowerBound (R-lub S)"
lemma glb_prop: "S ⊆ R-upperBound (R-glb S)"
```

Big-union resp. big-intersection of sets and relations corresponds in fact to the lub resp. glb.

```
lemma bigunion_lub: "(⊆)-lub S (⋃S)"
lemma biginter_glb: "(⊆)-glb S (⋂S)"
lemma bigunionR_lub:"(⊆ʳ)-lub S (⋃ʳS)"
lemma biginterR_glb: "(⊆ʳ)-glb S (⋂ʳS)"
```

8.4 Existence and Uniqueness under Antisymmetry

The following properties hold under the assumption that the given relation R is antisymmetric.

There can be at most one least/greatest element in a set.

```
lemma antisymm_least_unique: "antisymmetric R ⟹ unique(R-least S)"
lemma antisymm_greatest_unique: "antisymmetric R ⟹ unique(R-greatest S)"
```

If (the) least/greatest elements exist then they are identical to (the) min/max elements.

```
lemma antisymm_least_min: "antisymmetric R ⟹ ∃(R-least S) ⟹ (R-least S) = (R-min S)"
```

lemma *antisymm_greatest_max: "antisymmetric R* $\Longrightarrow$ $\exists$*(R-greatest S)* $\Longrightarrow$ *(R-greatest S) = (R-max S)"*

If (the) least/greatest elements of a set exist then they are identical to (the) glb/lub.

lemma *antisymm_least_glb: "antisymmetric R* $\Longrightarrow$ $\exists$*(R-least S)* $\Longrightarrow$ *(R-least S) = (R-glb S)"*
lemma *antisymm_greatest_lub: "antisymmetric R* $\Longrightarrow$ $\exists$*(R-greatest S)* $\Longrightarrow$ *(R-greatest S) = (R-lub S)"*

8.5 Further Properties of Endorelations

8.5.1 Well-ordering and Well-foundedness

The property of being a well-founded/ordered relation.

definition *wellOrdered::"Set(ERel('a))" ("wellOrdered")*
  where *"wellOrdered* $\equiv$ *((*$\subseteq$*)* $\exists$*)* $\circ$ *(B* $\exists$*)* $\circ$ *least"*
definition *wellFounded::"Set(ERel('a))" ("wellFounded")*
  where *"wellFounded* $\equiv$ *((*$\subseteq$*)* $\exists$*)* $\circ$ *(B* $\exists$*)* $\circ$ *min"*

declare *wellFounded_def[endorel_defs] wellOrdered_def[endorel_defs]*

lemma *"wellOrdered R = (*$\forall$*D.* $\exists$*D* $\rightarrow$ $\exists$*(R-least D))"*
lemma *"wellFounded R = (*$\forall$*D.* $\exists$*D* $\rightarrow$ $\exists$*(R-min D))"*

8.5.2 Limit-completeness

Limit-completeness is an important property of endorelations (orderings in particular). Famously, this is the property that characterizes the ordering of real numbers (in contrast to the rationals).

Note that existence of lubs for all sets entails existence of glbs for all sets (and viceversa).

lemma *"*$\forall$*S.* $\exists$*(R-lub S)* $\Longrightarrow$ $\forall$*S.* $\exists$*(R-glb S)"*
lemma *"*$\forall$*S.* $\exists$*(R-glb S)* $\Longrightarrow$ $\forall$*S.* $\exists$*(R-lub S)"*

The above results motivate the following definition: An endorelation R is called limit-complete when lubs/glbs (wrt R) exist for every set S (note that they must not be necessarily contained in S).

definition *limitComplete::"Set(ERel('a))"*
  where *"limitComplete* $\equiv$ $\forall$ $\circ$ *(*$\exists$ $\circ_2$ *lub)"*

lemma *"limitComplete R = (*$\forall$*S.* $\exists$*(R-lub S))"*

proposition *"limitComplete R* $\Longrightarrow$ *(R-lub S)* $\subseteq$ *S"* **nitpick** — countermodel found

Transpose/converse definitions.

lemma *limitComplete_def2: "limitComplete =* $\forall$ $\circ$ *(*$\exists$ $\circ_2$ *glb)"*
lemma *"limitComplete R = (*$\forall$*S.* $\exists$*(R-glb S))"*

lemma *limitComplete_defT: "limitComplete R*$^\smile$ *= limitComplete R"*

declare *limitComplete_def[endorel_defs]*

The subset and subrelation relations are indeed limit-complete.

lemma *subset_limitComplete: "limitComplete (*$\subseteq$*)"*
lemma *subrel_limitComplete: "limitComplete (*$\subseteq^r$*)"*

end

9 Graphs

Graphs are sets of endopairs and end up being isomorphic to endorelations (via currying). We replicate some of the theory of endorelations for illustration (exploiting currying).

```
theory graphs
imports endopairs endorelations
begin
```

9.1 Intervals and Powers

9.1.1 Intervals

An interval (wrt. given graph G) is the set of points that lie between given pair (of "boundaries").

```
abbreviation interval::"Graph('a) ⇒ 'a ⇒ 'a ⇒ Set('a)" ("_-interval")
  where "G-interval a b ≡ ⌊G⌋-interval a b "

lemma "G-interval a b = (λc. G <a,c> ∧ G <c,b>)"
```

9.1.2 Powers

We can extrapolate the notion of (relational) powers to graphs using currying.

```
abbreviation graphPower::"ERel(Graph('a))"
  where "graphPower G ≡ ⦇uncurry (relPower ⌊G⌋)⦈"
```

9.2 Properties and Operations

Properties of endorelations can be seamlessly transported to the world of graphs via currying.

9.2.1 Reflexivity and Irreflexivity

```
abbreviation reflexive::"Set(Graph('a))"
  where ‹reflexive G  ≡ endorelations.reflexive ⌊G⌋›
abbreviation irreflexive::"Set(Graph('a))"
  where ‹irreflexive G ≡ endorelations.irreflexive ⌊G⌋›

lemma "reflexive G = (∀x. G <x,x>)"
```

...and so on

9.2.2 Symmetry, Connectedness, and co.

```
abbreviation symmetric::"Set(Graph('a))"
  where ‹symmetric G  ≡ endorelations.symmetric ⌊G⌋›
abbreviation connected::"Set(Graph('a))"
  where ‹connected G ≡ endorelations.connected ⌊G⌋›

lemma "symmetric G = (∀a b. G <a,b> → G <b,a>)"
lemma "connected G = (∀a b. G <a,b> ∨ G <b,a>)"
```

...and so on

9.2.3 Transitivity, Denseness, Quasitransitivity, and co.

```
abbreviation transitive::"Set(Graph('a))"
  where ‹transitive G  ≡ endorelations.transitive ⌊G⌋›
abbreviation antitransitive::"Set(Graph('a))"
  where ‹antitransitive G ≡ endorelations.antitransitive ⌊G⌋›
abbreviation dense::"Set(Graph('a))"
  where ‹dense G ≡ endorelations.dense ⌊G⌋›

lemma ‹transitive G = (∀a b c. G <a,c> ∧ G <c,b> → G <a,b>)›
lemma ‹antitransitive G = (∀a b c. G <a,c> ∧ G <c,b> → ¬G <a,b>)›
lemma ‹dense G = (∀a b. G <a,b> → (∃c. G <a,c> ∧ G <c,b>))›
```

...and so on

9.2.4 Euclideanness and co.

```
abbreviation rightEuclidean::"Set(Graph('a))"
  where ‹rightEuclidean G  ≡ endorelations.rightEuclidean ⌊G⌋›
abbreviation leftEuclidean::"Set(Graph('a))"
  where ‹leftEuclidean G ≡ endorelations.leftEuclidean ⌊G⌋›

lemma ‹rightEuclidean G = (∀a b c. G <a,b> ∧ G <a,c> → G <b,c>)›
lemma ‹leftEuclidean G = (∀a b c. G <a,b> ∧ G <c,b> → G <a,c>)›
```

...and so on

end

10 Commutative diagrams

Commutative diagrams are convenient tools in mathematics that show how different paths of functions or maps between objects lead to the same result.

```
theory diagrams
 imports relations
begin
```

10.1 Basic Diagrams

10.1.1 For Functions

A commutative triangle states that a function "factorizes" as a composition of two given functions.

```
definition triangle :: "('a ⇒ 'b) ⇒ ('a ⇒ 'c) ⇒ ('c ⇒ 'b) ⇒ o" ("_-FACTOR")
  where "triangle ≡ B₁₂ 𝒬 I (;)"
```

Commutative triangles are often read as "h factors through f and g", and diagrammatically represented as:

```
lemma "h-FACTOR f g = (h = f ; g)"
abbreviation(input) triangleDiagram (" ∙ −_→ ∙ // \ ↓_ // _→ ∙")
  where "∙ −f→ ∙
          \      ↓g
            h→ ∙    ≡ h-FACTOR f g"

declare triangle_def[func_defs]
```

We say that an endofunction is idempotent when it is identical to the composition with itself, or, in other words, when it factors through itself.

```
definition idempotent::"Set('a ⇒ 'a)"
  where "idempotent ≡ W31 triangle"

declare idempotent_def[func_defs]

lemma "idempotent f =  ▪ —f→ ▪
                        \      ↓f
                          f→ ▪  "

lemma "idempotent f = (∀x. f x = f (f x))"
lemma "idempotent f = (f = (f ; f))"
lemma "idempotent = W (𝒬 ∘ (W B))"


definition square :: "('a ⇒ 'b) ⇒ ('a ⇒ 'c) ⇒ ('b ⇒ 'd) ⇒ ('c ⇒ 'd) ⇒ o"
  where "square ≡ B22 𝒬 (;) (;)"

abbreviation(input) squareDiagram (" ▪ —_→ ▪ // _↓ ↓_ // ▪ —_→ ▪")
  where " ▪ —j→ ▪
         i↓      ↓l
          ▪ —k→ ▪   ≡ square i j k l"

declare square_def[func_defs]

lemma " ▪ —j→ ▪
       i↓      ↓l
        ▪ —k→ ▪   = (i ; k = j ; l)"

lemma " ▪ —j→ ▪
       i↓      ↓l
        ▪ —k→ ▪   =  ▪ — j → ▪
                      \      ↓l
                        i;k→ ▪  "
```

10.1.2 For Relations

A commutative triangle states that a relation "factorizes" as a composition of two given relations.

```
definition relTriangle::"Rel('a,'b) ⇒ Rel('a,'c) ⇒ Rel('c,'b) ⇒ o" ("_-FACTORʳ")
  where "relTriangle ≡ B12 𝒬 I (;ʳ)"
```

Analogously to functions, we can represent this as a diagram (read as "H factors through F and G".)

```
lemma "H-FACTORʳ F G = (H = F ;ʳ G)"
abbreviation(input) relTriangleDiagram (" ▪ —_→ ▪ // \ ↓_ // _→ ▪")
  where "▪ —F→ ▪
          \     ↓G
            H→ ▪   ≡ H-FACTORʳ F G"

declare relTriangle_def[rel_defs]
```

Functional and relational triangle diagrams correspond as expected.

```
lemma triangle_funrel: "totalFunction i ⟹ totalFunction j ⟹ totalFunction k ⟹
                          triangle (asFun i) (asFun j) (asFun k) = relTriangle i j k"
lemma triangle_relfun: "relTriangle (asRel i) (asRel j) (asRel k) =    triangle i j k"
```

We say that an endorelation is idempotent when it is identical to the composition with itself, or, in other words, when it factors through itself.

```
definition relIdempotent::"Set(ERel('a))"
  where "relIdempotent ≡ W31 relTriangle"
```

declare *relIdempotent_def[rel_defs]*

lemma *"relIdempotent R =* $\cdot\ -R\to\ \cdot$
$\quad\backslash \qquad \downarrow R$
$\qquad R\to\ \cdot$ *"*

lemma *"relIdempotent R =* $(\forall a\ c.\ R\ a\ c \leftrightarrow (\exists b.\ R\ a\ b \wedge R\ b\ c))$*"*
lemma *"relIdempotent R = (R = (R* $;^r$ *R))"*
lemma *"relIdempotent = W (*$\mathcal{Q}$ $\circ$ *(W (*$\circ^r$*)))"*

The relational notions correspond to their functional counterparts as expected.

lemma *idempotent_funRel: "idempotent f = relIdempotent (asRel f)"*
lemma *idempotent_relFun: "totalFunction R* $\Longrightarrow$ *relIdempotent R = idempotent (asFun R)"*

definition *relSquare::"Rel('a,'b)* $\Rightarrow$ *Rel('a,'c)* $\Rightarrow$ *Rel('b,'d)* $\Rightarrow$ *Rel('c,'d)* $\Rightarrow$ *o"*
where *"relSquare* $\equiv$ $\mathtt{B}_{22}$ $\mathcal{Q}$ *(*$;^r$*) (*$;^r$*)"*

abbreviation *(input) relSquareDiagram ("* $\cdot$ *−_→* $\cdot$ *// _↓ ↓_ //* $\cdot$ *−_→* $\cdot$*")*
where *"* $\cdot\ -j\to\ \cdot$
$i\downarrow \qquad \downarrow l$
$\cdot\ -k\to\ \cdot$ $\quad\equiv$ *relSquare i j k l"*

declare *relSquare_def[rel_defs]*

lemma *"* $\cdot\ -j\to\ \cdot$
$i\downarrow \qquad \downarrow l$
$\cdot\ -k\to\ \cdot$ *= (i* $;^r$ *k = j* $;^r$ *l)"*

lemma *"* $\cdot\ -j\to\ \cdot$
$i\downarrow \qquad \downarrow l$
$\cdot\ -k\to\ \cdot$ *=* $\cdot\ -\ j\ \to\ \cdot$
$\qquad\backslash \qquad \downarrow l$
$\qquad i\ ;^r\ k\to\ \cdot$ *"*

Beware: Composition in relational squares must always be read along the principal (NWSE) diagonal!

lemma *"relSquare i j k l = (i* $;^r$ *k = j* $;^r$ *l)"*
lemma *"relSquare i j k l = ((k*$^\smile$*)* $;^r$ *(i*$^\smile$*) = (l*$^\smile$*)* $;^r$ *(j*$^\smile$*))"*
proposition *"relSquare i j k l* $\Longrightarrow$ *((i*$^\smile$*)* $;^r$ *j = k* $;^r$ *(l*$^\smile$*))"* **nitpick** — countermodel: wrong diagonal!
proposition *"relSquare i j k l* $\Longrightarrow$ *((j*$^\smile$*)* $;^r$ *i = l* $;^r$ *(k*$^\smile$*))"* **nitpick** — countermodel: wrong diagonal!

An alternative definition in terms of pullbacks.

lemma *relSquare_def2: "relSquare =* $\mathtt{C}_{3412}$ *(*$\mathtt{B}_{22}$ $\mathcal{Q}$ *(relPullback* $\circ$ *transpose) (relPullback* $\circ$ *transpose)*

Relational and functional squares correspond as expected.

lemma *square_funrel: "totalFunction i* $\Longrightarrow$ *totalFunction j* $\Longrightarrow$ *totalFunction k* $\Longrightarrow$ *totalFunction l* $\Longrightarrow$
square (asFun i) (asFun j) (asFun k) (asFun l) = relSquare i j k l"
lemma *square_relfun: "relSquare (asRel i) (asRel j) (asRel k) (asRel l) = square i j k l"*

10.2 Splittings

10.2.1 For Functions

We say of two functions f and g that they form a splitting (of the identity `I`) when g "undoes the effect" of f. In some literature, g (f) is called a left (right) inverse of f (g). We adopt another common (arguably less confusing) wording by referring to f (g) as the section (retraction) of g (f).

```
definition splitting::"Rel(('a ⇒ 'b),('b ⇒ 'a))"
  where "splitting ≡ I-FACTOR"

declare splitting_def[func_defs]
```

A section f followed by the corresponding retraction g takes us back where we started.

```
lemma "splitting f g = · −f→ ·
                        \     ↓g
                         I→ ·  "
```

We say that an endofunction is involutive when it is self-inverse (i.e. it forms a splitting with itself).

```
definition involutive::"Set('a ⇒ 'a)"
  where "involutive ≡ W splitting"

declare involutive_def[func_defs]

lemma "involutive f = (∀x. x = f (f x))"
lemma "involutive f = (I = f ; f)"
lemma "involutive = (Q I) ∘ (W B)"
```

Identity is the only function which is both involutive and idempotent.

```
lemma "(involutive f ∧ idempotent f) = (f = I)"
```

10.2.2 For Relations

Analogously to functions, we can say of two relations S and R that they form a splitting (of the identity $\mathcal{Q}$). Similarly, we call S (R) the section (retraction) of R (S).

```
definition relSplitting::"Rel(Rel('a,'b),Rel('b,'a))"
  where "relSplitting ≡ Q-FACTORʳ"

declare relSplitting_def[rel_defs]
```

A section S followed by the corresponding retraction R takes us back where we started.

```
lemma "relSplitting S R = · −S→ ·
                           \     ↓R
                            Q → ·  "
```

If a relation R (S) has a section (retraction) then it is right (left) total.

```
lemma "∃(relSplitting˘ R) ⟹ rightTotal R"
lemma "∃(relSplitting  S) ⟹  leftTotal S"
```

If a relation is both right-total and right-unique (surjective partial function) then it always has a section, and moreover, when it has a retraction then that retraction is unique

```
lemma exist_section:     "rightTotal R ⟹ rightUnique R ⟹ ∃(relSplitting˘ R)"
lemma unique_retraction: "rightTotal S ⟹ rightUnique S ⟹ unique(relSplitting S)"
```

If a relation is both left-total and left-unique (injective nondeterministic function) then it has a retraction, and moreover, when it has a section it is unique.

```
lemma exist_retraction: "leftTotal S ⟹ leftUnique S ⟹ ∃(relSplitting S)"
```

lemma *unique_section:* "$\mathit{leftTotal}\ R \Longrightarrow \mathit{leftUnique}\ R \Longrightarrow \mathit{unique}(\mathit{relSplitting}^{\smile}\ R)$"

lemma *splitting_trans:* "$\mathit{relSplitting}\ R\ T \Longrightarrow \mathit{relSplitting}\ (T^{\smile})\ (R^{\smile})$"

We say that an endorelation is involutive when it is self-inverse (i.e. it forms a splitting with itself).

definition *relInvolutive::*"$\mathit{Set}(\mathit{ERel}('a))$"
  where "$\mathit{relInvolutive} \equiv \mathtt{W}\ \mathit{relSplitting}$"

declare *relInvolutive_def[rel_defs]*

lemma "$\mathit{relInvolutive}\ R = (\forall a\ c.\ (a = c) \leftrightarrow (\exists b.\ R\ a\ b \wedge R\ b\ c))$"
lemma "$\mathit{relInvolutive}\ R = (\mathcal{Q} = R\ ;^r\ R)$"
lemma "$\mathit{relInvolutive} = (\mathcal{Q}\ \mathcal{Q}) \circ (\mathtt{W}\ (\circ^r))$"

Equality is the only relation which is both involutive and idempotent.

lemma "$(\mathit{relInvolutive}\ R \wedge \mathit{relIdempotent}\ R) = (R = \mathcal{Q})$"

Relational and functional notions correspond as expected.

lemma *involutive_funRel:* "$\mathit{involutive}\ f = \mathit{relInvolutive}\ (\mathit{asRel}\ f)$"
lemma *involutive_relFun:* "$\mathit{totalFunction}\ R \Longrightarrow \mathit{relInvolutive}\ R = \mathit{involutive}\ (\mathit{asFun}\ R)$"

10.3 Duality

We encode (relational) duality as a relation between functions (relations). It arises by fixing two of the arguments of a (relational) square as parameters (which we refer to as n_1 and n_2).

10.3.1 For Functions

Two functions f and g are said to be dual wrt. to a pair of functions n_1 and n_2 (as parameters).

definition *dual::*"$('a_1 \Rightarrow 'a_2) \Rightarrow ('b_1 \Rightarrow 'b_2) \Rightarrow \mathit{Rel}('a_1 \Rightarrow 'b_1,\ 'a_2 \Rightarrow 'b_2)$" ("_,_-DUAL")
  where "$n_1,n_2\text{-}\mathit{DUAL}\ f\ g \equiv$

$$\begin{array}{ccc} \cdot & -f\rightarrow & \cdot \\ n_1\downarrow & & \downarrow n_2 \\ \cdot & -g\rightarrow & \cdot \end{array}$$
"

We can also lift the previous notion of duality to apply to n-ary functions.

definition *dual2::*"$('a_1 \Rightarrow 'a_2) \Rightarrow ('b_1 \Rightarrow 'b_2) \Rightarrow \mathit{Rel}('e \Rightarrow 'a_1 \Rightarrow 'b_1,\ 'e \Rightarrow 'a_2 \Rightarrow 'b_2)$" ("$\_,\_\text{-}\mathit{DUAL}_2$")
  where "$n_1,n_2\text{-}\mathit{DUAL}_2 \equiv \mathbf{\Phi}_\forall\ (n_1,n_2\text{-}\mathit{DUAL})$"
definition *dual3::*"$('a_1 \Rightarrow 'a_2) \Rightarrow ('b_1 \Rightarrow 'b_2) \Rightarrow \mathit{Rel}('e_1 \Rightarrow 'e_2 \Rightarrow 'a_1 \Rightarrow 'b_1,\ 'e_1 \Rightarrow 'e_2 \Rightarrow 'a_2 \Rightarrow 'b_2)$" ("$\_,\_\text{-}\mathit{DUAL}_3$")
  where "$n_1,n_2\text{-}\mathit{DUAL}_3 \equiv \mathbf{\Phi}_\forall\ (n_1,n_2\text{-}\mathit{DUAL}_2)$"
— ... $n_1,n_2\text{-}\mathit{DUAL}_n \equiv \mathbf{\Phi}_\forall\ n_1,n_2\text{-}\mathit{DUAL}_{n-1}$

declare *dual_def[func_defs] dual2_def[func_defs] dual3_def[func_defs]*

lemma "$n_1,n_2\text{-}\mathit{DUAL}_2\ f\ g = (\forall x\ y.\ g\ x\ (n_1\ y) = n_2\ (f\ x\ y))$"
lemma "$n_1,n_2\text{-}\mathit{DUAL}_3\ f\ g = (\forall x\ y\ z.\ g\ x\ y\ (n_1\ z) = n_2\ (f\ x\ y\ z))$"

Note that if both n_1 and n_2 are involutive, then the dual relation is symmetric.

lemma *dual_symm:* "$\mathit{involutive}\ n_1 \Longrightarrow \mathit{involutive}\ n_2 \Longrightarrow n_1,n_2\text{-}\mathit{DUAL}\ f\ g = n_1,n_2\text{-}\mathit{DUAL}\ f\ g$"
lemma *dual2_symm:* "$\mathit{involutive}\ n_1 \Longrightarrow \mathit{involutive}\ n_2 \Longrightarrow n_1,n_2\text{-}\mathit{DUAL}_2\ f\ g = n_1,n_2\text{-}\mathit{DUAL}_2\ f\ g$"

This notion does NOT correspond with the so-called "De Morgan duality" (although they are not unrelated).

proposition "$\neg,\neg\text{-}\mathit{DUAL}_2\ (\wedge)\ (\vee)$" **nitpick** — countermodel found

We add a (convenient?) diagram for duality of binary functions (for unary functions it is just the square).

```
abbreviation(input) dual2Diagram (" • =_→ • // _↓ ↓_ // • =_→ •")
  where "  • = f → •
        n₁↓         ↓n₂
          • = g → •    ≡ n₁,n₂-DUAL₂ f g"
```

Some examples of dual pairs of binary operations (recall that negation and complement are involutive).

```
lemma " • = (∧) → •
      ¬↓          ↓¬
        • = (→) → •  "

lemma " • = (∨) → •
      ¬↓          ↓¬
        • = (⇀) → •  "

lemma " • = (⇒) → •
      −↓           ↓−
        • = (∩) → •  "

lemma " • =  (\)  →  •
      −↓             ↓−
        • =(−∘₂(∩))→ •  "
```

10.3.2 For Relations

Two relations R and T are said to be dual wrt. to a pair of relations n_1 and n_2 (as parameters).

```
definition relDual::"Rel('a₁,'a₂) ⇒ Rel('b₁,'b₂) ⇒ Rel(Rel('a₁,'b₁), Rel('a₂,'b₂))" ("_,_-DUALʳ")
  where "n₁,n₂-DUALʳ R T ≡   • −R→ •
                           n₁↓      ↓n₂
                             • −T→ •   "

declare  relDual_def[rel_defs]

lemma "n₁,n₂-DUAL  f g = (n₁ ; g = f ; n₂)"
lemma "n₁,n₂-DUALʳ R T = (n₁ ;ʳ T = R ;ʳ n₂)"
```

The notion of dual for relations corresponds to the counterpart notion for functions.

```
lemma "n₁,n₂-DUAL f g = (asRel n₁),(asRel n₂)-DUALʳ (asRel f) (asRel g)"
lemma "totalFunction n₁ ⟹ totalFunction n₂ ⟹ totalFunction R ⟹ totalFunction T
          ⟹ n₁,n₂-DUALʳ R T = (asFun n₁),(asFun n₂)-DUAL (asFun R) (asFun T)"
```

Existence of sections and retractions influences existence and uniqueness of duals. As a corollary, if n_1 (resp. n_2) is involutive, then the dual relation is well-defined (exists and is unique).

```
lemma "relSplitting n₁ m ⟹ n₁,n₂-DUALʳ R (m ;ʳ R ;ʳ n₂)"
lemma "relSplitting m n₂ ⟹ n₁,n₂-DUALʳ (n₁ ;ʳ T ;ʳ m) T"
lemma "∃m. relSplitting m n₁ ⟹ unique(n₁,n₂-DUALʳ R)"
lemma "∃m. relSplitting n₂ m ⟹ unique((n₁,n₂-DUALʳ)˘ T)"
```

Moreover, if both n_1 and n_2 are involutive, then the dual relation is symmetric.

```
lemma relDual_symm: "relInvolutive n₁ ⟹ relInvolutive n₂ ⟹ n₁,n₂-DUALʳ R T = n₁,n₂-DUALʳ
T R"

end
```

11 Adjunctions

The term "adjunction" is quite overloaded in the literature. Here we consider two flavors:

1. Galois-connections (aka. dual-adjunctions or Galois-adjunctions), which are contravariant.
2. Adjunctions (aka. residuations), which are covariant. We refer to them just as "adjunctions" (simpliciter). We will focus on Galois connections, since (covariant) adjunctions can easily defined in terms of them.

theory *adjunctions*
 imports *diagrams endorelations*
begin

named_theorems *adj_defs*

Galois-connections (aka. Galois- or dual-adjunctions) relate pairs of functions (having flipped domain-codomain) wrt. a pair of endorelations (usually orderings on the functions' domains). We focus in this section on the traditional notion of "contravariant" Galois-connection wrt. a pair of arbitrary relations R_1 and R_2. Note that a "covariant" version, aka. adjunction (simpliciter), can always be defined by reversing R_2 below.

11.1 For Relations

We introduce a convenient notion of "relational Galois-connection" relating a given pair of relations F and G wrt. another pair of relations R_1 and R_2 (as parameters). This generalizes the traditional "functional" notion, while sidestepping the use of descriptions and their associated existence/uniqueness assumptions.

definition *relGalois::"Rel('a,'b) ⇒ Rel('c,'d) ⇒ Rel('a,'d) ⇒ Rel('c,'b) ⇒ o"* ("_,_-GAL^r")
  where "R_1,R_2-GAL^r F G ≡
$$\begin{array}{ccc} \bullet & -R_2\rightarrow & \bullet \\ G\downarrow & & \downarrow F^{\smile} \\ \bullet & -R_1{}^{\smile}\rightarrow & \bullet \end{array}$$
"

declare *relGalois_def[adj_defs]*

We get a more intuitive representation for Galois-connections by rotating the above (square) diagram by *90*° clockwise. Note that in such "Galois diagrams" composition is read along the SWNE diagonal!

abbreviation *(input) relGaloisDiagram (" • ←_– • // _↑ ↓_ // • –_→ •")*
  where "
$$\begin{array}{ccc} \bullet & \leftarrow G- & \bullet \\ R_1\uparrow & & \downarrow R_2 \\ \bullet & -F\rightarrow & \bullet \end{array} \quad \equiv R_1,R_2\text{-}GAL^r\ F\ G$$
"

lemma *relGalois_def2:* "
$$\begin{array}{ccc} \bullet & \leftarrow G- & \bullet \\ R_1\uparrow & & \downarrow R_2 \\ \bullet & -F\rightarrow & \bullet \end{array} \quad = (F\ ;^r\ (R_2{}^{\smile}) = R_1\ ;^r\ (G^{\smile}))$$
"

An alternative definition:

lemma *relGalois_altdef:* "relGalois = C (B_{22} $\mathcal{Q}$ relPullback (C relPullback))"
lemma "R_1,R_2-GAL^r = (λF G. $\mathcal{Q}$ (relPullback R_2 F) (C relPullback R_1 G))"
lemma "R_1,R_2-GAL^r = (λF G. relPullback R_2 F = relPullback G R_1)"
lemma "R_1,R_2-GAL^r = (λF G. ∀b a. relPullback R_2 F b a ↔ relPullback G R_1 b a)"
lemma *relGalois_setdef:* "R_1,R_2-GAL^r = (λF G. ∀a b. (R_2 b ⊓ F a) ↔ (G b ⊓ R_1 a))"

Galois-connections are clearly "symmetric" in the following sense:

lemma *relGalois_symm:* "R_1,R_2-GAL^r F G = R_2,R_1-GAL^r G F"

Galois-connections and dualities are intertranslatable in several ways.

lemma $''R_1,R_2\text{-}GAL^r\ \ F\ G = R_1,R_2{}^{\smile}\text{-}DUAL^r\ F\ (G^{\smile})''$
lemma $''n_1,n_2\text{-}DUAL^r\ R\ T = n_1,n_2{}^{\smile}\text{-}GAL^r\ \ R\ (T^{\smile})''$
lemma $''R_1,R_2\text{-}GAL^r\ \ F\ G = \ F,G^{\smile}\text{-}DUAL^r\ R_1\ (R_2{}^{\smile})''$
lemma $''n_1,n_2\text{-}DUAL^r\ R\ T = \ R,T^{\smile}\text{-}GAL^r\ \ n_1\ (n_2{}^{\smile})''$

Drawing upon the above, we can sketch solutions to the problem of finding a right resp. left adjoint to a given relation, for those particular cases where R_1 resp. R_2 have sections or retractions.

lemma $''relSplitting\ R_1\ m \Longrightarrow R_1,R_2\text{-}GAL^r\ F\ (R_2\ ;^r\ (F^{\smile})\ ;^r\ (m^{\smile}))''$
lemma $''relSplitting\ R_2\ m \Longrightarrow R_1,R_2\text{-}GAL^r\ (R_1\ ;^r\ (G^{\smile})\ ;^r\ (m^{\smile}))\ G''$

For the (very common) particular case where R_1 and R_2 are endorelations (possibly on different types), we can introduce the following operation (parameterized with R_1 and R_2) that given a relation F returns another relation G, its Galois "adjoint", so that F and G form a Galois-connection (wrt. R_1 and R_2).

definition $relAdjoint{::}''ERel('a) \Rightarrow ERel('b) \Rightarrow Rel('a,'b) \Rightarrow Rel('b,'a)''\ (''\_,\_\text{-}adj^r'')$
  where $''relAdjoint \equiv \mathtt{B}_{11}\ \mathtt{I}\ (\mathtt{E}\ lub)\ relPullback''$

declare $relAdjoint\_def[adj\_defs]$

lemma $''R_1,R_2\text{-}adj^r = (\mathtt{E}\ lub\ R_1)\ (relPullback\ R_2)''$
lemma $relAdjoint\_setdef{:}\ ''R_1,R_2\text{-}adj^r\ F = (\lambda b.\ (R_1\text{-}lub\ (\lambda a.\ R_2\ b \sqcap F\ a)))''$

Some useful things can be said of adjoints already in this general (relational) case

lemma $''antisymmetric\ R_1 \Longrightarrow rightUnique\ F \Longrightarrow rightUnique\ (R_1,R_2\text{-}adj^r\ F)''$ — right-uniqueness preservation

An interesting question is that of determining minimal conditions under which the previous definition behaves as expected. A partial solution is provided below for illustration, where it remains to find out under which conditions a relation F has a Galois adjoint that is a total function. A real answer for the general case is left as exercise (solving for particular cases will be enough later on).

lemma $relGalois\_prop{:}\ ''skeletal\ R_1 \Longrightarrow \exists(R_1,R_2\text{-}GAL^r\ F \cap totalFunction)$
$\Longrightarrow R_1,R_2\text{-}GAL^r\ F\ (R_1,R_2\text{-}adj^r\ F)''$

The related question of uniqueness of Galois adjoints (when they exist) is simpler.

lemma $relGalois\_rightUnique{:}\ ''skeletal\ R_1 \Longrightarrow unique((R_1,R_2\text{-}GAL^r\ F) \cap rightUnique)''$ — proof by external provers

11.2 For Functions

We now move towards the notion of (functional) Galois-connections, still slightly generalized, such that it relates pairs of functions f and g wrt a pair of arbitrary relations R_1 and R_2. We encode this notion as a particular case of the relational Galois-connection discussed above.

definition $galois{::}''Rel('a,'b) \Rightarrow Rel('c,'d) \Rightarrow Rel(('a \Rightarrow 'd),('c \Rightarrow 'b))''\ (''\_,\_\text{-}GAL'')$
  where $''galois \equiv \mathtt{B}_{1111}\ relGalois\ \mathtt{I}\ \mathtt{I}\ asRel\ asRel''$

declare $galois\_def[adj\_defs]$

lemma $''R_1,R_2\text{-}GAL\ f\ g = R_1,R_2\text{-}GAL^r\ \ (asRel\ f)\ (asRel\ g)''$
lemma $''R_1,R_2\text{-}GAL\ f\ g = (\forall b\ a.\ R_2\ b \sqcap \{f\ a\} = \{g\ b\} \sqcap R_1\ a)''$

We also introduce a convenient diagram notation for functional Galois connections.

```
abbreviation(input) galoisDiagram (" ∙ ←_− ∙ // _↑ ↓_ // ∙ −_→ ∙")
  where  " ∙ ←g− ∙
         R₁↑      ↓R₂
           ∙ −f→ ∙    ≡ R₁,R₂-GAL f g"

lemma galois_def2: " ∙ ←g− ∙
                   R₁↑      ↓R₂
                     ∙ −f→ ∙    = (∀a b. R₂ b (f a) ↔ R₁ a (g b))"
```

An alternative definition:

```
lemma galois_altdef: "galois = C (B₂₂ 𝒬 (B₁₁ I) (C ∘₂ (B₁₁ I)))"
lemma "R₁,R₂-GAL f g = ((B₁₁ I) R₂ f = (C ∘₂ (B₁₁ I)) R₁ g)"
lemma "R₁,R₂-GAL f g = (∀a b. (B₁₁ I) R₂ f a b ↔ (C ∘₂ (B₁₁ I)) R₁ g a b)"
lemma "R₁,R₂-GAL f g = (relPullback R₂ (asRel f) = relPullback (asRel g) R₁)"
```

Again, Galois-connections are "symmetric" in the following sense:

```
lemma galois_symm: "R₁,R₂-GAL f g = R₂,R₁-GAL g f"
```

For the (very common) particular case where R_1 and R_2 are endorelations (possibly on different types), we can introduce the following operation (parameterized with R_1 and R_2) that given a function f returns another relation g, its "adjoint", so that f and g form a Galois-connection (wrt. R_1 and R_2).

```
definition adjoint::"ERel('a) ⇒ ERel('b) ⇒ Op('a,'b) ⇒ Op('b,'a)" ("_,_-adj")
  where "adjoint ≡ (B₃ ι) ∘ ((B₁₃ I lub) (B₁₁ I)) "

declare adjoint_def[adj_defs]

lemma "R₁,R₂-adj f = (λb. ι (R₁-lub (λa. R₂ b (f a))))"
```

As mentioned previously, (covariant) adjunctions can be encoded by reversing the parameter R_2.

```
abbreviation(input) adjunction::"ERel('a) ⇒ ERel('b) ⇒ Rel(Op('a,'b),Op('b,'a))" ("_,_-ADJ")
  where "R₁,R₂-ADJ ≡ R₁,R₂˘-GAL"
```

We also introduce a convenient diagram notation for adjunctions (with a reversed right arrow).

```
abbreviation(input) adjunctionDiagram (" ∙ ←_− ∙ // _↑ ↑_ // ∙ −_→ ∙")
  where  " ∙ ←g− ∙
         R₁↑      ↑R₂
           ∙ −f→ ∙    ≡ R₁,R₂-ADJ f g"

lemma adjunction_def2: " ∙ ←g− ∙
                       R₁↑      ↑R₂
                         ∙ −f→ ∙    = (∀a b. R₂ (f a) b ↔ R₁ a (g b))"
```

Note that the (covariant) adjunction is not "symmetric" in the sense the Galois-connection is.

```
proposition "R₁,R₂-ADJ f g = R₂,R₁-ADJ g f" nitpick — countermodel found
```

A possible explanation for the adjectives "covariant" and "contravariant".

```
lemma "preorder R ⟹ R,R-ADJ f g ⟹ R-MONO g"
lemma "preorder R ⟹ R,R-GAL f g ⟹ R-ANTI g"
```

Hence, when working with (covariant) adjunctions we need to introduce two operations (parameterized with R_1 and R_2) which when given functions f resp. g return their "right" resp. "left" adjoint.

```
abbreviation(input) rightAdjoint::"ERel('a) ⇒ ERel('b) ⇒ Op('a,'b) ⇒ Op('b,'a)" ("_,_-rightAdj")
```

where "R_1,R_2-*rightAdj* ≡ R_1,$R_2{}^{\smile}$-*adj*"
abbreviation(*input*) *leftAdjoint*::"*ERel('a) ⇒ ERel('b) ⇒ Op('b,'a) ⇒ Op('a,'b)*" ("_,_-*leftAdj*")
where "R_1,R_2-*leftAdj* ≡ $R_2{}^{\smile}$,R_1-*adj*"

lemma "R_1,R_2-*rightAdj f* = (λb. ι (R_1-*lub* (λa. R_2 (*f* a) b)))"
lemma "R_1,R_2-*leftAdj g* = (λa. ι (R_2-*glb* (λb. R_1 a (*g* b))))"

lemma "R_1,R_2-*leftAdj* = $R_2{}^{\smile}$,$R_1{}^{\smile}$-*rightAdj*"

Our adjoint operator behaves as expected for those functions that have indeed some adjoint (again, we still have to find out under which minimal conditions such adjoints exist for the general case).

lemma *galois_prop*: "*skeletal* R_1 ⟹ ∃(R_1,R_2-*GAL f*) ⟹ R_1,R_2-*GAL f* (R_1,R_2-*adj f*)" — proof by external provers

We can conveniently extend the previous definitions towards indexed functions (e.g. binary endooperations).

definition *galois2*::"*ERel('a) ⇒ ERel('b) ⇒ Rel('e-Env(Op('a,'b)),'e-Env(Op('b,'a)))*" ("_,_-*GAL*$_2$")
where "R_1,R_2-*GAL*$_2$ ≡ $\mathbf{\Phi}_\forall$ (R_1,R_2-*GAL*)"
abbreviation(*input*) *adjunction2*::"*ERel('a) ⇒ ERel('b) ⇒ Rel('e-Env(Op('a,'b)),'e-Env(Op('b,'a)))*" ("_,_-*ADJ*$_2$")
where "R_1,R_2-*ADJ*$_2$ ≡ R_1,$R_2{}^{\smile}$-*GAL*$_2$"

declare *galois2_def[adj_defs]*

lemma "R_1,R_2-*GAL*$_2$ *f g* = (∀x. R_1,R_2-*GAL* (*f* x) (*g* x))"
lemma "R_1,R_2-*ADJ*$_2$ *f g* = (∀x. R_1,R_2-*ADJ* (*f* x) (*g* x))"
lemma "R_1,R_2-*GAL*$_2$ *f g* = (∀a b c. R_2 b (*f* c a) ↔ R_1 a (*g* c b))"
lemma "R_1,R_2-*ADJ*$_2$ *f g* = (∀a b c. R_2 (*f* c a) b ↔ R_1 a (*g* c b))"
lemma "(⊆),(⊆)-*GAL*$_2$ *f g* = (∀a b c. b ⊆ (*f* c) a ↔ a ⊆ (*g* c) b)" — proof by external provers

lemma "(⊆),(⊆)-*ADJ*$_2$ *f g* = (∀a b c. (*f* c) a ⊆ b ↔ a ⊆ (*g* c) b)" — proof by external provers

A convenient "lifting" rule for (Galois) adjunctions (and for any arities).

lemma *galois_lift1*: "R_1,R_2-*GAL f g* ⟹ ($\mathbf{\Phi}_\forall$ R_1),($\mathbf{\Phi}_\forall$ R_2)-*GAL* ($\mathbf{\Phi}_{11}$ *f*) ($\mathbf{\Phi}_{11}$ *g*)"
lemma *adjunction_lift1*: "R_1,R_2-*ADJ f g* ⟹ ($\mathbf{\Phi}_\forall$ R_1),($\mathbf{\Phi}_\forall$ R_2)-*ADJ* ($\mathbf{\Phi}_{11}$ *f*) ($\mathbf{\Phi}_{11}$ *g*)"

lemma *galois_lift2*: "R_1,R_2-*GAL*$_2$ *f g* ⟹ ($\mathbf{\Phi}_\forall$ R_1),($\mathbf{\Phi}_\forall$ R_2)-*GAL*$_2$ ($\mathbf{\Phi}_{21}$ *f*) ($\mathbf{\Phi}_{21}$ *g*)"
lemma *adjunction_lift2*: "R_1,R_2-*ADJ*$_2$ *f g* ⟹ ($\mathbf{\Phi}_\forall$ R_1),($\mathbf{\Phi}_\forall$ R_2)-*ADJ*$_2$ ($\mathbf{\Phi}_{21}$ *f*) ($\mathbf{\Phi}_{21}$ *g*)"

11.3 Concrete examples

Integer addition and substraction form a Galois-connection wrt equality and an adjunction wrt. inequality.

lemma "$\mathcal{Q}$,$\mathcal{Q}$-*GAL* (λx::*int*. x + a) (λx. x - a)"
lemma "(≤),(≤)-*ADJ* (λx::*int*. x + a) (λx. x - a)"

Symmetric difference is self-adjoint wrt. equality (but not wrt inequality).

lemma "$\mathcal{Q}$,$\mathcal{Q}$-*GAL* ((△) a) ((△) a)"
proposition "(⊆),(⊆)-*GAL* ((△) a) ((△) a)" **nitpick** — countermodel found
proposition "(⊆),(⊆)-*ADJ* ((△) a) ((△) a)" **nitpick** — countermodel found

lemma "(≤)-*MONO* (*f*::*int⇒int*) ⟹ $\mathcal{Q}$,$\mathcal{Q}$-*GAL f g* ⟹ (≤),(≤)-*ADJ f g*"
lemma "(≤),(≤)-*ADJ* (*f*::*int⇒int*) (*g*::*int⇒int*) ⟹ $\mathcal{Q}$,$\mathcal{Q}$-*GAL f g*" — proof by external provers

The relation-based right- and left-bound operators form a Galois-connection.

lemma *"(⊆),(⊆)-GAL R-rightBound R-leftBound"*

The relation-based right-image and left-dualimage operators form a (covariant) adjunction.

lemma *"(⊆),(⊆)-ADJ R-rightImage R-leftDualImage"*

The usual "residuation" properties of boolean connectives (recall that → is an ordering on *{𝒯,ℱ}*).

lemma *and_impl_adj: "(→),(→)-ADJ$_2$ (∧) (→)"*
lemma *dimpl_or_adj: "(→),(→)-ADJ$_2$ (⇀) (∨)"*

Note that we can use the "adjunction lifting" rule to prove adjunctions on lifted (indexed) operations.

lemma *"(⊆),(⊆)-ADJ$_2$ (∩) (⇒)"*
lemma *"(⊆^{r}),(⊆^{r})-ADJ$_2$ (∩^{r}) (⇒^{r})"*

end

12 Entailment and Validity

theory *entailment*
imports *adjunctions*
begin

12.1 Special Case (for Modal Logicians and co.)

Modal logics model propositions as sets (of "worlds") and are primarily concerned with "validity" of propositions. We encode below the set of valid (resp. unsatisfiable, satisfiable) propositions.

definition *valid::"Set(Set('a))" ("⊨ _")*
where *"valid ≡ ∀"*
definition *satisfiable::"Set(Set('a))" ("⊨^{sat} _")*
where *"satisfiable ≡ ∃"*
definition *unsatisfiable::"Set(Set('a))" ("⊨^{unsat} _")*
where *"unsatisfiable ≡ ∄"*

lemma *"⊨ P = (∀w. P w)"*
lemma *"⊨^{sat} P = (∃w. P w)"*
lemma *"⊨^{unsat} P = (¬(∃w. P w))"*

In modal logic, logical consequence/entailment usually comes in two flavours: "local" and "global". The local variant is the default one (i.e. the one employed in most sources). Semantically, it corresponds to the subset relation (assumptions are aggregated using conjunction/intersection).

abbreviation *(input) localEntailment::"ERel(Set('a))"* (**infixr** *"⊨$_l$" 99)*
where *"a ⊨$_l$ c ≡ a ⊆ c"*
abbreviation *(input) localEntailment2::"Set('a) ⇒ ERel(Set('a))" ("_,_ ⊨$_l$ _")*
where *"a$_1$, a$_2$ ⊨$_l$ c ≡ (a$_1$ ∩ a$_2$) ⊨$_l$ c"* — syntax sugar for two (or more) premises
— ...and so on

Clearly, validity is a special case of local entailment.

lemma *"⊨ c ↔ 𝔘 ⊨$_l$ c"*

In fact, local entailment can also be stated in terms of validity via the so-called "deduction (meta-)theorem", which follows as a particular case of the following fact (aka. "residuation law").

lemma *local_residuation:* "$(\models_l),(\models_l)$-ADJ$_2$ $(\cap)$ $(\Rightarrow)$"
lemma "$a,\ b \models_l c \leftrightarrow b \models_l (a \Rightarrow c)$"

Which produces the "deduction meta-theorem" as a particular case (with b = $\mathfrak{U}$).

lemma *DMT:* "$\models a \Rightarrow c \leftrightarrow a \models_l c$ "

Global entailment is sometimes discussed, mostly for theoretical purposes (e.g. in algebraic logic).

abbreviation *(input) globalEntailment::"Rel(Set(Set('a)),Set('a))"* (**infixr** "$\models_g$" 99)
where "$A \models_g c \equiv (\forall a.\ A\ a \rightarrow \models a) \rightarrow \models c$"

Again, validity is clearly a special case of global entailment.

lemma "$\models c \leftrightarrow \{\mathfrak{U}\} \models_g c$"

Local entailment is stronger than global entailment.

lemma "$a \models_l c \Longrightarrow \{a\} \models_g c$"
lemma "$a,b \models_l c \Longrightarrow \{a,b\} \models_g c$"
lemma "$\{a\} \models_g c \Longrightarrow a \models_l c$" **nitpick**

The "deduction meta-theorem" does not hold for global entailment.

lemma "$\{a\} \models_g c \Longrightarrow (\models a \Rightarrow c)$" **nitpick**
lemma "$\{a,b\} \models_g c \Longrightarrow \{b\} \models_g (a \Rightarrow c)$" **nitpick**
lemma "$\models (a \Rightarrow c) \Longrightarrow \{a\} \models_g c$"
lemma "$\{b\} \models_g (a \Rightarrow c) \Longrightarrow \{a,b\} \models_g c$"

12.2 General Case (for Algebraic and Many-valued/Fuzzy Logicians)

We introduce an "entailment" operation (for denotations) that corresponds to the semantic counterpart of the notion of consequence (for formulas). We refer to the literature on algebraic logic for detailed explanations, in particular [2] for an overview, and references therein.

We encode below a general notion of logical entailment as discussed in the algebraic logic literature; cf. "ramified matrices" (e.g. [5]) and "generalized matrices" (e.g. [2]). Entailment becomes parameterized with a class *TT* of "truth-sets". We say that a set of assumptions *A* entails the conclusion *c* iff when all *A*s are in *T* then *c* is in *T* too, for all truth-sets *T* in *TT*.

definition *entailment::"Set(Set('a))* $\Rightarrow$ *SetEOp('a)"* ("$\mathcal{E}$")
where "$\mathcal{E}\ TT \equiv \lambda A.\ \lambda c.\ \forall T.\ TT\ T \longrightarrow A \subseteq T \longrightarrow T\ c$"

notation *(input) entailment* ("[_|_ $\models$ _]")
lemma "[TT| A $\models$ c] = $\mathcal{E}$ TT A c"

Alternative definition: *c* is in the intersection of all truth-sets containing *A*

lemma *entailment_def2:* "[TT| A $\models$ c] = $\bigcap(TT \cap (\subseteq)\ A)\ c$"

It is worth noting that when the class of truth-sets — TT is closed under arbitrary intersections (aka. "closure system") then entailment becomes a closure (aka. hull) operator.

lemma *entailment_closure:* "$\forall X.\ X \subseteq TT \longrightarrow TT\ (\bigcap X) \Longrightarrow (\subseteq)$-CLOSURE $(\mathcal{E}\ TT)$"

One special case of the definition above occurs when *TT* is a singleton *{T}*. This corresponds to the traditional notion of logical consequence associated to "logical matrices" in algebraic logic, and which is characterized by the principle of truth(-value) preservation ("truth-preserving consequence" in [2]). We thus refer to its encoding below as "(truth-)value-preserving entailment".

definition *valueEntailment::"Set('a)* $\Rightarrow$ *SetEOp('a)"* ("$\mathcal{E}_v$")
where "$\mathcal{E}_v\ T \equiv \mathcal{E}\ \{T\}$"

notation *(input) valueEntailment ("[_|_ $\models_v$ _]")*
lemma *"[T| A $\models_v$ c] = $\mathcal{E}_v$ T A c"*

Alternative definition: if all As are in T (true) then c is also in T (true).

lemma *valueEntailment_def2: "[T| A $\models_v$ c] = (A $\subseteq$ T $\longrightarrow$ T c)"*

Value-preserving entailment is a closure operator too.

lemma *ValueConsequence_closure: "($\subseteq$)-CLOSURE ($\mathcal{E}_v$ T)"*

Back to the general notion of entailment, now observe that it satisfies the following properties:

lemma *entailment_prop1: "transitive R $\Longrightarrow$ R-glb A m $\Longrightarrow$ [range R | A $\models$ c] = R m c"*
lemma *entailment_prop2: "preorder R $\Longrightarrow$ [range R | {a} $\models$ c] = R a c"*

The properties above justify the following special case, in which the class of truth-sets is given as the (functional) range of a relation (qua set-valued function). Following [2] we speak of "(truth-)degree-preserving" entailment.

definition *degreeEntailment::"ERel('a) $\Rightarrow$ SetEOp('a)" ("$\mathcal{E}_d$")*
where *"$\mathcal{E}_d$ R $\equiv$ $\mathcal{E}$ (range R)"*

notation *(input) degreeEntailment ("[_|_ $\models_d$ _]")*
lemma *"[R| A $\models_d$ c] = $\mathcal{E}_d$ R A c"*

Alternative definitions for transitive relations resp. preorders.

lemma *degreeEntailment_def2: "transitive R $\Longrightarrow$ R-glb A h $\Longrightarrow$ [R| A $\models_d$ c] = R h c"*
lemma *degreeEntailment_def3: "preorder R $\Longrightarrow$ [R| {a} $\models_d$ c] = R a c"*

Degree-preserving entailment is a closure operator.

lemma *degreeEntailment_closure: "($\subseteq$)-CLOSURE ($\mathcal{E}_d$ R)"*

It is worth mentioning that for semantics based on algebras of sets (e.g. modal algebras/Kripke models) the usual notion of logical consequence ("local consequence") corresponds to the "degree-preserving" entailment presented here, when instantiated with the subset relation.

lemma *degreeEntailment_local: "[($\subseteq$)| A $\models_d$ c] = $\bigcap$A $\models_l$ c"*

Similarly, the notion of "global consequence" (e.g. in modal logic) corresponds to "value-preserving" consequence instantiated with *T = {$\mathfrak{U}$}* where $\mathfrak{U}$ is the universe of all points (or "worlds").

lemma *valueEntailment_global: "[{$\mathfrak{U}$}| A $\models_v$ c] = A $\models_g$ c"*

end

13 General Theory of Relation-based Operators

It is well known that (n+1-ary) relations give rise to (n-ary) operations on sets (called "operators"). We explore some basic algebraic properties of relation-based set-operators.

theory *operators*
imports *adjunctions*
begin

13.1 Set-Operators from Binary Relations

This is the (non-trivial) base case. It is very common in logic, so it gets an special treatment.

Add some convenient (arguably less visually-cluttering) notation, reminiscent of logical operations.

notation *(input) leftImage ("_-$\Diamond_{\leftarrow}$")* **and** *leftDualImage ("_-$\Box_{\leftarrow}$")* **and**
rightImage ("_-$\Diamond_{\rightarrow}$") **and** *rightDualImage ("_-$\Box_{\rightarrow}$")* **and**
leftBound ("_-$\ominus_{\leftarrow}$") **and** *leftDualBound ("_-$\oslash_{\leftarrow}$")* **and**
rightBound ("_-$\ominus_{\rightarrow}$") **and** *rightDualBound ("_-$\oslash_{\rightarrow}$")*

— and extend this notation to the transformations themselves

notation *(input) leftImage ("$\Diamond_{\leftarrow}$")* **and** *leftDualImage ("$\Box_{\leftarrow}$")* **and**
rightImage ("$\Diamond_{\rightarrow}$") **and** *rightDualImage ("$\Box_{\rightarrow}$")* **and**
leftBound ("$\ominus_{\leftarrow}$") **and** *leftDualBound ("$\oslash_{\leftarrow}$")* **and**
rightBound ("$\ominus_{\rightarrow}$") **and** *rightDualBound ("$\oslash_{\rightarrow}$")*

13.1.1 Order Embedding

This is a good moment to recall that unary operations on sets (set-operations) are also relations...

term *"(F :: SetOp('a,'b)) :: Rel(Set('a),'b)"*

... and thus can be ordered as such. Thus read $F \subseteq^r G$ as: "F is a sub-operation of G".

lemma fixes *F::"SetOp('a,'b)"* **and** *G::"SetOp('a,'b)"*
shows *"F $\subseteq^r$ G = ($\forall$A. F A $\subseteq$ G A)"*

Operators are (dual) embeddings between the sub-relation and the (converse of) sub-operation ordering.

lemma *rightImage_embedding: "($\subseteq^r$),($\subseteq^r$)-embedding $\Diamond_{\rightarrow}$"*
lemma *leftImage_embedding: "($\subseteq^r$),($\subseteq^r$)-embedding $\Diamond_{\leftarrow}$"*
lemma *rightDualImage_embedding: "($\subseteq^r$),($\supseteq^r$)-embedding $\Box_{\rightarrow}$"*
lemma *leftDualImage_embedding: "($\subseteq^r$),($\supseteq^r$)-embedding $\Box_{\leftarrow}$"*
lemma *rightBound_embedding: "($\subseteq^r$),($\subseteq^r$)-embedding $\ominus_{\rightarrow}$"*
lemma *leftBound_embedding: "($\subseteq^r$),($\subseteq^r$)-embedding $\ominus_{\leftarrow}$"*
lemma *rightDualBound_embedding: "($\subseteq^r$),($\supseteq^r$)-embedding $\oslash_{\rightarrow}$"*
lemma *leftDualBound_embedding: "($\subseteq^r$),($\supseteq^r$)-embedding $\oslash_{\leftarrow}$"*

13.1.2 Homomorphisms

Operators are also (dual) homomorphisms from the monoid of relations to the monoid of (set-)operators.

First of all, they map the relational unit $\mathcal{Q}$ (resp. its dual $\mathcal{D}$) to the functional unit I (resp. its dual $-$).

lemma *rightImage_hom_id: "$\mathcal{Q}$-$\Diamond_{\rightarrow}$ = I"*
lemma *leftImage_hom_id: "$\mathcal{Q}$-$\Diamond_{\leftarrow}$ = I"*
lemma *rightDualImage_hom_id: "$\mathcal{Q}$-$\Box_{\rightarrow}$ = I"*
lemma *leftDualImage_hom_id: "$\mathcal{Q}$-$\Box_{\leftarrow}$ = I"*
lemma *rightBound_hom_id: "$\mathcal{D}$-$\ominus_{\rightarrow}$ = $-$"*
lemma *leftBound_hom_id: "$\mathcal{D}$-$\ominus_{\leftarrow}$ = $-$"*
lemma *rightDualBound_hom_id: "$\mathcal{D}$-$\oslash_{\rightarrow}$ = $-$"*
lemma *leftDualBound_hom_id: "$\mathcal{D}$-$\oslash_{\leftarrow}$ = $-$"*

Moreover, they map the relational composition $\circ^r$ (resp. its dual $\bullet^r$) to their functional counterparts.

lemma *rightImage_hom_comp: "(A $\circ^r$ B)-$\Diamond_{\rightarrow}$ = (A-$\Diamond_{\rightarrow}$) $\circ$ (B-$\Diamond_{\rightarrow}$)"*
lemma *leftImage_hom_comp: "(A $\circ^r$ B)-$\Diamond_{\leftarrow}$ = (B-$\Diamond_{\leftarrow}$) $\circ$ (A-$\Diamond_{\leftarrow}$)"*

lemma *rightDualImage_hom_comp:* "$(A \circ^r B)\text{-}\square_{\rightarrow} = (A\text{-}\square_{\rightarrow}) \circ (B\text{-}\square_{\rightarrow})$"
lemma *leftDualImage_hom_comp:* "$(A \circ^r B)\text{-}\square_{\leftarrow} = (B\text{-}\square_{\leftarrow}) \circ (A\text{-}\square_{\leftarrow})$"
lemma *rightBound_hom_comp:* "$(A \cdot^r B)\text{-}\ominus_{\rightarrow} = (A\text{-}\ominus_{\rightarrow}) \cdot (B\text{-}\ominus_{\rightarrow})$"
lemma *leftBound_hom_comp:* "$(A \cdot^r B)\text{-}\ominus_{\leftarrow} = (B\text{-}\ominus_{\leftarrow}) \cdot (A\text{-}\ominus_{\leftarrow})$"
lemma *rightDualBound_hom_comp:* "$(A \cdot^r B)\text{-}\oslash_{\rightarrow} = (A\text{-}\oslash_{\rightarrow}) \cdot (B\text{-}\oslash_{\rightarrow})$"
lemma *leftDualBound_hom_comp:* "$(A \cdot^r B)\text{-}\oslash_{\leftarrow} = (B\text{-}\oslash_{\leftarrow}) \cdot (A\text{-}\oslash_{\leftarrow})$"

Operators are also (dual) lattice homomorphisms from the BA of relations to the BA of set-operators.

lemma *rightImage_hom_join:* "$(R_1 \cup^r R_2)\text{-}\Diamond_{\rightarrow} = R_1\text{-}\Diamond_{\rightarrow} \cup^r R_2\text{-}\Diamond_{\rightarrow}$"
lemma *leftImage_hom_join:* "$(R_1 \cup^r R_2)\text{-}\Diamond_{\leftarrow} = R_1\text{-}\Diamond_{\leftarrow} \cup^r R_2\text{-}\Diamond_{\leftarrow}$"
lemma *rightBound_hom_meet:* "$(R_1 \cap^r R_2)\text{-}\ominus_{\rightarrow} = R_1\text{-}\ominus_{\rightarrow} \cap^r R_2\text{-}\ominus_{\rightarrow}$"
lemma *leftBound_hom_meet:* "$(R_1 \cap^r R_2)\text{-}\ominus_{\leftarrow} = R_1\text{-}\ominus_{\leftarrow} \cap^r R_2\text{-}\ominus_{\leftarrow}$"
lemma *rightDualImage_hom_join:* "$(R_1 \cup^r R_2)\text{-}\square_{\rightarrow} = R_1\text{-}\square_{\rightarrow} \cap^r R_2\text{-}\square_{\rightarrow}$"
lemma *leftDualImage_hom_join:* "$(R_1 \cup^r R_2)\text{-}\square_{\leftarrow} = R_1\text{-}\square_{\leftarrow} \cap^r R_2\text{-}\square_{\leftarrow}$"
lemma *rightDualBound_hom_meet:* "$(R_1 \cap^r R_2)\text{-}\oslash_{\rightarrow} = R_1\text{-}\oslash_{\rightarrow} \cup^r R_2\text{-}\oslash_{\rightarrow}$"
lemma *leftDualBound_hom_meet:* "$(R_1 \cap^r R_2)\text{-}\oslash_{\leftarrow} = R_1\text{-}\oslash_{\leftarrow} \cup^r R_2\text{-}\oslash_{\leftarrow}$"

As for complement, we have a particular morphism property between images and bounds (cf. dualities below).

lemma *rightImage_hom_compl:* "$(R^-)\text{-}\Diamond_{\rightarrow} = (R\text{-}\ominus_{\rightarrow})^-$"
lemma *leftImage_hom_compl:* "$(R^-)\text{-}\Diamond_{\leftarrow} = (R\text{-}\ominus_{\leftarrow})^-$"
lemma *rightDualImage_hom_compl:* "$(R^-)\text{-}\square_{\rightarrow} = (R\text{-}\oslash_{\rightarrow})^-$"
lemma *leftDualImage_hom_compl:* "$(R^-)\text{-}\square_{\leftarrow} = (R\text{-}\oslash_{\leftarrow})^-$"
lemma *rightBound_hom_compl:* "$(R^-)\text{-}\ominus_{\rightarrow} = (R\text{-}\Diamond_{\rightarrow})^-$"
lemma *leftBound_hom_compl:* "$(R^-)\text{-}\ominus_{\leftarrow} = (R\text{-}\Diamond_{\leftarrow})^-$"
lemma *rightDualBound_hom_compl:* "$(R^-)\text{-}\oslash_{\rightarrow} = (R\text{-}\square_{\rightarrow})^-$"
lemma *leftDualBound_hom_compl:* "$(R^-)\text{-}\oslash_{\leftarrow} = (R\text{-}\square_{\leftarrow})^-$"

13.1.3 Dualities (illustrated with diagrams)

Dualities between some pairs of relation-based set-operators.

lemma *leftImage_dual:* "$-,-\text{-}DUAL\ (R\text{-}\Diamond_{\leftarrow})\ (R\text{-}\square_{\leftarrow})$"
lemma " $\cdot\ {-R\text{-}\Diamond_{\leftarrow}} \rightarrow \cdot$
$-\!\downarrow \qquad\qquad \downarrow\!-$
$\cdot\ {-R\text{-}\square_{\leftarrow}} \rightarrow \cdot$ "
lemma *rightImage_dual:* "$-,-\text{-}DUAL\ (R\text{-}\Diamond_{\rightarrow})\ (R\text{-}\square_{\rightarrow})$"
lemma " $\cdot\ {-R\text{-}\Diamond_{\rightarrow}} \rightarrow \cdot$
$-\!\downarrow \qquad\qquad \downarrow\!-$
$\cdot\ {-R\text{-}\square_{\rightarrow}} \rightarrow \cdot$ "

lemma *leftBound_dual:* "$-,-\text{-}DUAL\ (R\text{-}\ominus_{\leftarrow})\ (R\text{-}\oslash_{\leftarrow})$"
lemma " $\cdot\ {-R\text{-}\ominus_{\leftarrow}} \rightarrow \cdot$
$-\!\downarrow \qquad\qquad \downarrow\!-$
$\cdot\ {-R\text{-}\oslash_{\leftarrow}} \rightarrow \cdot$ "
lemma *rightBound_dual:* "$-,-\text{-}DUAL\ (R\text{-}\ominus_{\rightarrow})\ (R\text{-}\oslash_{\rightarrow})$"
lemma " $\cdot\ {-R\text{-}\ominus_{\rightarrow}} \rightarrow \cdot$
$-\!\downarrow \qquad\qquad \downarrow\!-$
$\cdot\ {-R\text{-}\oslash_{\rightarrow}} \rightarrow \cdot$ "

Recall that set-operators are also relations (and thus can be ordered as such). We thus have following dualities between the transformations themselves (cf. morphisms wrt. complement discussed above).

lemma *leftImageBound_dual:* "$-^r,-^r\text{-}DUAL\ \Diamond_{\leftarrow}\ \ominus_{\leftarrow}$"
lemma " $\cdot\ {-\Diamond_{\leftarrow}} \rightarrow \cdot$

$-^r\downarrow$ $\downarrow -^r$
▪ −⊖← → ▪ "
lemma *rightImageBound_dual:* "$-^r$,$-^r$-DUAL ◊→ ⊖→"
lemma " ▪ −◊→ → ▪
$-^r\downarrow$ $\downarrow -^r$
▪ −⊖→ → ▪ "

lemma *leftDualImageBound_dual:* "$-^r$,$-^r$-DUAL □← ⊘←"
lemma " ▪ −□← → ▪
$-^r\downarrow$ $\downarrow -^r$
▪ −⊘← → ▪ "
lemma *rightDualImageBound_dual:* "$-^r$,$-^r$-DUAL □→ ⊘→"
lemma " ▪ −□→ → ▪
$-^r\downarrow$ $\downarrow -^r$
▪ −⊘→ → ▪ "

13.1.4 Adjunctions (illustrated with diagrams)

In order theory it is not uncommon to refer to a (covariant) adjunction as a "residuation".

lemma *leftImage_residuation:* "(⊆),(⊆)-ADJ (R-◊←) (R-□→)"
lemma " ▪ ←R-□→ − ▪
(⊆)↑ ↑(⊆)
▪ −R-◊← → ▪ "
lemma *rightImage_residuation:* "(⊆),(⊆)-ADJ (R-◊→) (R-□←)"
lemma " ▪ ←R-□← − ▪
(⊆)↑ ↑(⊆)
▪ −R-◊→ → ▪ "

We may refer to a residuation between complements of operators as a "co-residuation" (between the operators).

lemma *leftBound_coresiduation:* "(⊆),(⊆)-ADJ (R-⊖←)$^-$ (R-⊘→)$^-$"
lemma " ▪ ←(R-⊘→)$^-$ − ▪
(⊆)↑ ↑(⊆)
▪ −(R-⊖←)$^-$ → ▪ "
lemma *rightBound_coresiduation:* "(⊆),(⊆)-ADJ (R-⊖→)$^-$ (R-⊘←)$^-$"
lemma " ▪ ←(R-⊘←)$^-$ − ▪
(⊆)↑ ↑(⊆)
▪ −(R-⊖→)$^-$ → ▪ "

There is a Galois connection between the right and left bounds.

lemma *rightBound_galois:* "(⊆),(⊆)-GAL (R-⊖→) (R-⊖←)"
lemma " ▪ ←R-⊖← − ▪
(⊆)↑ ↓(⊆)
▪ −R-⊖→ → ▪ "

We shall refer to a Galois connection with reversed orderings as a "dual-Galois-connection".

lemma *leftDualBound_dualgalois:* "(⊇),(⊇)-GAL (R-⊘←) (R-⊘→)"
lemma " ▪ ←R-⊘→ − ▪
(⊇)↑ ↓(⊇)
▪ −R-⊘← → ▪ "

We also refer to a (dual) Galois connection between complements of operators as "(dual) conjugation".

lemma *rightImage_conjugation:* "(⊆),(⊆)-GAL (R-◊→)$^-$ (R-◊←)$^-$"
lemma " ▪ ←(R-◊←)$^-$ − ▪
(⊆)↑ ↓(⊆)
▪ −(R-◊→)$^-$ → ▪ "
lemma *leftDualImage_dualconjugation:* "(⊇),(⊇)-GAL (R-□←)$^-$ (R-□→)$^-$"
lemma " ▪ ←(R-□→)$^-$ − ▪

$(\supseteq)\uparrow \qquad\qquad\qquad \downarrow(\supseteq)$
$\cdot - (R\text{-}\Box_{\leftarrow})^{-} \rightarrow \cdot$ "

13.2 Set-Operators from n-ary Relations

13.2.1 Images and Preimages of n-ary Functions

We shall begin by extending the notions of image and preimage from unary to n-ary functions.

Recall that for unary functions we obtain a unary image set-operation as:

term *"image :: ('a ⇒ 'b) ⇒ Set('a) ⇒ Set('b)"*
lemma *"image f A = (λb. ∃a. f a = b ∧ A a)"*

We now generalize the previous notion towards higher arities to obtain n-ary set-operations.

definition *image2 :: "('a ⇒ 'b ⇒ 'c) ⇒ Set('a) ⇒ Set('b) ⇒ Set('c)" ("$image_2$")*
  where *"$image_2$ f A B ≡ (λc. ∃a b. f a b = c ∧ A a ∧ B b)"*
definition *image3 :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ Set('a) ⇒ Set('b) ⇒ Set('c) ⇒ Set('d)" ("$image_3$")*
  where *"$image_3$ f A B C ≡ (λd. ∃a b c. f a b c = d ∧ A a ∧ B b ∧ C c)"*
— ... $image_n$ f A_1 ...A_n ≡ (λx. ∃a_1 ... a_n. f a_1 ... a_n = x ∧ A_1 a_1 ∧ ... A_n a_n)

declare *image2_def[func_defs] image3_def[func_defs]*

lemma *"$image_2$ f A B = (λc. $inverse_2$ f c $\sqcap^r$ (A × B))"*

The same move can be done for the notion of preimage.

definition *preimage2 :: "('a ⇒ 'b ⇒ 'c) ⇒ Set('c) ⇒ Rel('a,'b)" ("$preimage_2$")*
  where *"$preimage_2$ f C ≡ f $;_2$ C"*
definition *preimage3 :: "('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ Set('d) ⇒ Rel_3('a,'b,'c)" ("$preimage_3$")*
  where *"$preimage_3$ f D ≡ f $;_3$ D"*
— ... $preimage_n$ f X ≡ f $;_n$ X

declare *preimage2_def[func_defs] preimage3_def[func_defs]*

13.2.2 Images and Bounds of n-ary Relations

Let us start by recalling that images and bounds are two sides of the same dual coin.

lemma *"$-^r$,$-^r$-DUAL $\Diamond_{\leftarrow}$ $\ominus_{\leftarrow}$"*
lemma *"$-^r$,$-^r$-DUAL $\Diamond_{\rightarrow}$ $\ominus_{\rightarrow}$"*

Recall that by seeing binary relations as generalized (partial and non-deterministic) functions, the notions of function's (direct) image becomes generalized as relation's right-image, which corresponds to.

lemma *"rightImage = $\bigcup$ $\circ_2$ image"*

We extend this notion of direct (i.e. right) image to n+1-ary relations, thus obtaining n-ary set-operations.

definition *rightImage2 ::"Rel_3('a,'b,'c) ⇒ Set('a) ⇒ Set('b) ⇒ Set('c)" ("$rightImage_2$")*
  where *"$rightImage_2$ ≡ $\bigcup$ $\circ_3$ $image_2$"*
— ... $rightImage_n$ ≡ $\bigcup$ $\circ_{n+1}$ $image_n$

declare *rightImage2_def[rel_defs]*

Or, alternatively:

lemma *rightImage2_def2: "$rightImage_2$ = (B_{111} (($\sqcap^r$) $\circ_2$ (×)) I I) ∘ R"*

Recall that for binary relations the analogous of preimage is the left-image operator, definable as the right-image of their converse. We now "lift" this idea to higher arities, noting that we must now consider six permutations, so we have to come up with a richer naming scheme. In the ternary case, we conveniently use a numbering scheme, related to the family of $\mathtt{C}_{abc}$ combinators (permutators).

abbreviation *(input) image123::"Rel*$_3$*('a,'b,'c) ⇒ Set('a) ⇒ Set('b) ⇒ Set('c)" ("*$\Diamond_{123}$*")*
where "$\Diamond_{123} \equiv$ *rightImage*$_2 \circ \mathtt{C}_{123}$" — $\mathtt{C}_{123}$ as identity permutation is its own inverse (involutive)
abbreviation *(input) image132::"Rel*$_3$*('a,'b,'c) ⇒ Set('a) ⇒ Set('c) ⇒ Set('b)" ("*$\Diamond_{132}$*")*
where "$\Diamond_{132} \equiv$ *rightImage*$_2 \circ \mathtt{C}_{132}$" — $\mathtt{C}_{132}$ is its own inverse
abbreviation *(input) image213::"Rel*$_3$*('a,'b,'c) ⇒ Set('b) ⇒ Set('a) ⇒ Set('c)" ("*$\Diamond_{213}$*")*
where "$\Diamond_{213} \equiv$ *rightImage*$_2 \circ \mathtt{C}_{213}$" — $\mathtt{C}_{213}$ is its own inverse
abbreviation *(input) image231::"Rel*$_3$*('a,'b,'c) ⇒ Set('b) ⇒ Set('c) ⇒ Set('a)" ("*$\Diamond_{231}$*")*
where "$\Diamond_{231} \equiv$ *rightImage*$_2 \circ \mathtt{C}_{312}$" — $\mathtt{C}_{312}$/L is the inverse of $\mathtt{C}_{231}$/R
abbreviation *(input) image312::"Rel*$_3$*('a,'b,'c) ⇒ Set('c) ⇒ Set('a) ⇒ Set('b)" ("*$\Diamond_{312}$*")*
where "$\Diamond_{312} \equiv$ *rightImage*$_2 \circ \mathtt{C}_{231}$" — $\mathtt{C}_{231}$/R is the inverse of $\mathtt{C}_{312}$/L
abbreviation *(input) image321::"Rel*$_3$*('a,'b,'c) ⇒ Set('c) ⇒ Set('b) ⇒ Set('a)" ("*$\Diamond_{321}$*")*
where "$\Diamond_{321} \equiv$ *rightImage*$_2 \circ \mathtt{C}_{321}$" — $\mathtt{C}_{321}$ is its own inverse

notation *(input) image123 ("_-*$\Diamond_{123}$*")* **and** *image132 ("_-*$\Diamond_{132}$*")* **and**
image213 ("_-$\Diamond_{213}$*")* **and** *image231 ("_-*$\Diamond_{231}$*")* **and**
image312 ("_-$\Diamond_{312}$*")* **and** *image321 ("_-*$\Diamond_{321}$*")*

lemma "$R\text{-}\Diamond_{123} = (\lambda A\ B.\ \lambda c.\ \exists a\ b.\ R\ a\ b\ c \wedge A\ a \wedge B\ b)$"
lemma "$R\text{-}\Diamond_{132} = (\lambda A\ C.\ \lambda b.\ \exists a\ c.\ R\ a\ b\ c \wedge A\ a \wedge C\ c)$"
lemma "$R\text{-}\Diamond_{213} = (\lambda B\ A.\ \lambda c.\ \exists b\ a.\ R\ a\ b\ c \wedge B\ b \wedge A\ a)$"
lemma "$R\text{-}\Diamond_{231} = (\lambda B\ C.\ \lambda a.\ \exists b\ c.\ R\ a\ b\ c \wedge B\ b \wedge C\ c)$"
lemma "$R\text{-}\Diamond_{312} = (\lambda C\ A.\ \lambda b.\ \exists c\ a.\ R\ a\ b\ c \wedge C\ c \wedge A\ a)$"
lemma "$R\text{-}\Diamond_{321} = (\lambda C\ B.\ \lambda a.\ \exists c\ b.\ R\ a\ b\ c \wedge C\ c \wedge B\ b)$"

Note that the images (in general: all operators) of a relation can be interrelated via permutation.

lemma "$R\text{-}\Diamond_{123} = (\mathtt{C}_{132}\ R)\text{-}\Diamond_{132}$"
lemma "$R\text{-}\Diamond_{123} = (\mathtt{C}_{213}\ R)\text{-}\Diamond_{213}$"
lemma "$R\text{-}\Diamond_{123} = (\mathtt{C}_{231}\ R)\text{-}\Diamond_{231}$"
lemma "$R\text{-}\Diamond_{123} = (\mathtt{C}_{312}\ R)\text{-}\Diamond_{312}$"
lemma "$R\text{-}\Diamond_{123} = (\mathtt{C}_{321}\ R)\text{-}\Diamond_{321}$"
lemma "$R\text{-}\Diamond_{132} = (\mathtt{C}_{231}\ R)\text{-}\Diamond_{213}$"
lemma "$R\text{-}\Diamond_{132} = (\mathtt{C}_{321}\ R)\text{-}\Diamond_{312}$"
lemma "$R\text{-}\Diamond_{213} = (\mathtt{C}_{132}\ R)\text{-}\Diamond_{312}$"
lemma "$R\text{-}\Diamond_{213} = (\mathtt{C}_{231}\ R)\text{-}\Diamond_{321}$"
lemma "$R\text{-}\Diamond_{231} = (\mathtt{C}_{132}\ R)\text{-}\Diamond_{321}$"
lemma "$R\text{-}\Diamond_{231} = (\mathtt{C}_{231}\ R)\text{-}\Diamond_{312}$"
lemma "$R\text{-}\Diamond_{312} = (\mathtt{C}_{213}\ R)\text{-}\Diamond_{321}$"
lemma "$R\text{-}\Diamond_{312} = (\mathtt{C}_{231}\ R)\text{-}\Diamond_{123}$"
lemma "$R\text{-}\Diamond_{321} = (\mathtt{C}_{213}\ R)\text{-}\Diamond_{312}$"
lemma "$R\text{-}\Diamond_{321} = (\mathtt{C}_{321}\ R)\text{-}\Diamond_{123}$"

Now, recall that for binary relations we have that:

lemma "*rightBound* = $\bigcap \circ_2$ *image*"

Again, we extend this notion towards n+1-ary relations to obtain n-ary set-operations

definition *rightBound2 ::"Rel*$_3$*('a,'b,'c) ⇒ Set('a) ⇒ Set('b) ⇒ Set('c)" ("rightBound*$_2$*")*
where "*rightBound*$_2 \equiv \bigcap \circ_3$ *image*$_2$"
— ... *rightBound*$_n \equiv \bigcap \circ_{n+1}$ *image*$_n$

declare *rightBound2_def[rel_defs]*

Or, alternatively:

lemma *rightBound2_def2:* "*rightBound*$_2$ = $(\mathtt{B}_{111}\ ((\sqcup^r) \circ_2 (\mathbf{\Psi}_2\ (\uplus)\ -))\ \mathtt{I}\ \mathtt{I}) \circ \mathtt{R}$"

Analogously as the case for images/preimages, when we "lift" the notion of bounds to higher arities we consider several permutations, and come up with a numbering scheme based on permutations.

abbreviation *(input) bound123::"Rel*$_3$*('a,'b,'c) ⇒ Set('a) ⇒ Set('b) ⇒ Set('c)" ("*$\ominus_{123}$*")*
  where *"*$\ominus_{123}$ ≡ *rightBound*$_2$ ∘ C$_{123}$*"* — C$_{123}$ being identity is its own inverse (involutive)
abbreviation *(input) bound132::"Rel*$_3$*('a,'b,'c) ⇒ Set('a) ⇒ Set('c) ⇒ Set('b)" ("*$\ominus_{132}$*")*
  where *"*$\ominus_{132}$ ≡ *rightBound*$_2$ ∘ C$_{132}$*"* — C$_{132}$ is its own inverse
abbreviation *(input) bound213::"Rel*$_3$*('a,'b,'c) ⇒ Set('b) ⇒ Set('a) ⇒ Set('c)" ("*$\ominus_{213}$*")*
  where *"*$\ominus_{213}$ ≡ *rightBound*$_2$ ∘ C$_{213}$*"* — C$_{213}$ is its own inverse
abbreviation *(input) bound231::"Rel*$_3$*('a,'b,'c) ⇒ Set('b) ⇒ Set('c) ⇒ Set('a)" ("*$\ominus_{231}$*")*
  where *"*$\ominus_{231}$ ≡ *rightBound*$_2$ ∘ C$_{312}$*"* — C$_{312}$/L is the inverse of C$_{231}$/R
abbreviation *(input) bound312::"Rel*$_3$*('a,'b,'c) ⇒ Set('c) ⇒ Set('a) ⇒ Set('b)" ("*$\ominus_{312}$*")*
  where *"*$\ominus_{312}$ ≡ *rightBound*$_2$ ∘ C$_{231}$*"* — C$_{231}$/R is the inverse of C_{312}/L
abbreviation *(input) bound321::"Rel*$_3$*('a,'b,'c) ⇒ Set('c) ⇒ Set('b) ⇒ Set('a)" ("*$\ominus_{321}$*")*
  where *"*$\ominus_{321}$ ≡ *rightBound*$_2$ ∘ C$_{321}$*"* — C$_{321}$ is its own inverse

notation *(input) bound123 ("_-*$\ominus_{123}$*")* **and** *bound132 ("_-*$\ominus_{132}$*")* **and**
  bound213 ("_-$\ominus_{213}$*")* **and** *bound231 ("_-*$\ominus_{231}$*")* **and**
  bound312 ("_-$\ominus_{312}$*")* **and** *bound321 ("_-*$\ominus_{321}$*")*

lemma *"R-*$\ominus_{123}$ *= (λA B. λc. ∀a b. A a → B b → R a b c)"*
lemma *"R-*$\ominus_{132}$ *= (λA C. λb. ∀a c. A a → C c → R a b c)"*
lemma *"R-*$\ominus_{213}$ *= (λB A. λc. ∀b a. B b → A a → R a b c)"*
lemma *"R-*$\ominus_{231}$ *= (λB C. λa. ∀b c. B b → C c → R a b c)"*
lemma *"R-*$\ominus_{312}$ *= (λC A. λb. ∀c a. C c → A a → R a b c)"*
lemma *"R-*$\ominus_{321}$ *= (λC B. λa. ∀c b. C c → B b → R a b c)"*

Again, note that the different bound operators can be similarly interrelated by permutation.

lemma *"R-*$\ominus_{123}$ = (C$_{132}$ *R)-*$\ominus_{132}$*"*
lemma *"R-*$\ominus_{123}$ = (C$_{213}$ *R)-*$\ominus_{213}$*"*
lemma *"R-*$\ominus_{123}$ = (C$_{231}$ *R)-*$\ominus_{231}$*"*
lemma *"R-*$\ominus_{123}$ = (C$_{312}$ *R)-*$\ominus_{312}$*"*
lemma *"R-*$\ominus_{123}$ = (C$_{321}$ *R)-*$\ominus_{321}$*"*
lemma *"R-*$\ominus_{132}$ = (C$_{231}$ *R)-*$\ominus_{213}$*"*
lemma *"R-*$\ominus_{132}$ = (C$_{321}$ *R)-*$\ominus_{312}$*"*
lemma *"R-*$\ominus_{213}$ = (C$_{132}$ *R)-*$\ominus_{312}$*"*
lemma *"R-*$\ominus_{213}$ = (C$_{231}$ *R)-*$\ominus_{321}$*"*
lemma *"R-*$\ominus_{231}$ = (C$_{132}$ *R)-*$\ominus_{321}$*"*
lemma *"R-*$\ominus_{231}$ = (C$_{231}$ *R)-*$\ominus_{312}$*"*
lemma *"R-*$\ominus_{312}$ = (C$_{213}$ *R)-*$\ominus_{321}$*"*
lemma *"R-*$\ominus_{312}$ = (C$_{231}$ *R)-*$\ominus_{123}$*"*
lemma *"R-*$\ominus_{321}$ = (C$_{213}$ *R)-*$\ominus_{312}$*"*
lemma *"R-*$\ominus_{321}$ = (C$_{321}$ *R)-*$\ominus_{123}$*"*

13.2.3 Dual-Images and Dual-Bounds

As for the dual images, we take this as starting point.

definition *rightDualImage2::"Rel*$_3$*('a,'b,'c) ⇒ Set('a) ⇒ Set('b) ⇒ Set('c)" ("rightDualImage*$_2$*")*
  where *"rightDualImage*$_2$ *R ≡ λA B. λc. ∀a b. R a b c → A a → B b"*

declare *rightDualImage2_def[rel_defs]*

As in the case of binary relations, (left-, right-, ...) image-operators have their duals too.

abbreviation *(input) dualImage123::"Rel*$_3$*('a,'b,'c) ⇒ Set('a) ⇒ Set('b) ⇒ Set('c)" ("*$\square_{123}$*")*
  where *"*$\square_{123}$ ≡ *rightDualImage*$_2$ ∘ C$_{123}$*"* — C$_{123}$ being identity is its own inverse (involutive)
abbreviation *(input) dualImage132::"Rel*$_3$*('a,'b,'c) ⇒ Set('a) ⇒ Set('c) ⇒ Set('b)" ("*$\square_{132}$*")*
  where *"*$\square_{132}$ ≡ *rightDualImage*$_2$ ∘ C$_{132}$*"* — C$_{132}$ is its own inverse
abbreviation *(input) dualImage213::"Rel*$_3$*('a,'b,'c) ⇒ Set('b) ⇒ Set('a) ⇒ Set('c)" ("*$\square_{213}$*")*

where "$\Box_{213} \equiv$ $rightDualImage_2 \circ C_{213}$" — C_{213} is its own inverse
abbreviation(*input*) *dualImage231*::"Rel_3('a,'b,'c) ⇒ Set('b) ⇒ Set('c) ⇒ Set('a)" ("$\Box_{231}$")
where "$\Box_{231} \equiv$ $rightDualImage_2 \circ C_{312}$" — C_{312}/L is the inverse of C_{231}/R
abbreviation(*input*) *dualImage312*::"Rel_3('a,'b,'c) ⇒ Set('c) ⇒ Set('a) ⇒ Set('b)" ("$\Box_{312}$")
where "$\Box_{312} \equiv$ $rightDualImage_2 \circ C_{231}$" — C_{231}/R is the inverse of C_{312}/L
abbreviation(*input*) *dualImage321*::"Rel_3('a,'b,'c) ⇒ Set('c) ⇒ Set('b) ⇒ Set('a)" ("$\Box_{321}$")
where "$\Box_{321} \equiv$ $rightDualImage_2 \circ C_{321}$" — C_{321} is its own inverse

notation(*input*) *dualImage123* ("_-$\Box_{123}$") **and** *dualImage132* ("_-$\Box_{132}$") **and**
dualImage213 ("_-$\Box_{213}$") **and** *dualImage231* ("_-$\Box_{231}$") **and**
dualImage312 ("_-$\Box_{312}$") **and** *dualImage321* ("_-$\Box_{321}$")

lemma "R-$\Box_{123}$ = $(\lambda A\ B.\ \lambda c.\ \forall a\ b.\ R\ a\ b\ c \to A\ a \to B\ b)$"
lemma "R-$\Box_{132}$ = $(\lambda A\ C.\ \lambda b.\ \forall a\ c.\ R\ a\ b\ c \to A\ a \to C\ c)$"
lemma "R-$\Box_{213}$ = $(\lambda B\ A.\ \lambda c.\ \forall b\ a.\ R\ a\ b\ c \to B\ b \to A\ a)$"
lemma "R-$\Box_{231}$ = $(\lambda B\ C.\ \lambda a.\ \forall b\ c.\ R\ a\ b\ c \to B\ b \to C\ c)$"
lemma "R-$\Box_{321}$ = $(\lambda C\ B.\ \lambda a.\ \forall c\ b.\ R\ a\ b\ c \to C\ c \to B\ b)$"
lemma "R-$\Box_{312}$ = $(\lambda C\ A.\ \lambda b.\ \forall c\ a.\ R\ a\ b\ c \to C\ c \to A\ a)$"

Again, note that the dual-images of a relation can be similarly interrelated by permutation.

lemma "R-$\Box_{123}$ = (C_{132} R)-$\Box_{132}$"
lemma "R-$\Box_{123}$ = (C_{213} R)-$\Box_{213}$"
lemma "R-$\Box_{123}$ = (C_{231} R)-$\Box_{231}$"
lemma "R-$\Box_{123}$ = (C_{312} R)-$\Box_{312}$"
lemma "R-$\Box_{123}$ = (C_{321} R)-$\Box_{321}$"
lemma "R-$\Box_{132}$ = (C_{231} R)-$\Box_{213}$"
lemma "R-$\Box_{132}$ = (C_{321} R)-$\Box_{312}$"
lemma "R-$\Box_{213}$ = (C_{132} R)-$\Box_{312}$"
lemma "R-$\Box_{213}$ = (C_{231} R)-$\Box_{321}$"
lemma "R-$\Box_{231}$ = (C_{132} R)-$\Box_{321}$"
lemma "R-$\Box_{231}$ = (C_{231} R)-$\Box_{312}$"
lemma "R-$\Box_{312}$ = (C_{213} R)-$\Box_{321}$"
lemma "R-$\Box_{312}$ = (C_{231} R)-$\Box_{123}$"
lemma "R-$\Box_{321}$ = (C_{213} R)-$\Box_{312}$"
lemma "R-$\Box_{321}$ = (C_{321} R)-$\Box_{123}$"

Check dualities.

lemma *image123_dual*: "−,−-$DUAL_2$ (R-$\Diamond_{123}$) (R-$\Box_{123}$)"
lemma *image132_dual*: "−,−-$DUAL_2$ (R-$\Diamond_{132}$) (R-$\Box_{132}$)"
lemma *image213_dual*: "−,−-$DUAL_2$ (R-$\Diamond_{213}$) (R-$\Box_{213}$)"
lemma *image231_dual*: "−,−-$DUAL_2$ (R-$\Diamond_{231}$) (R-$\Box_{231}$)"
lemma *image312_dual*: "−,−-$DUAL_2$ (R-$\Diamond_{312}$) (R-$\Box_{312}$)"
lemma *image321_dual*: "−,−-$DUAL_2$ (R-$\Diamond_{321}$) (R-$\Box_{321}$)"

For the dual-bounds, we take the following as starting point.

definition *rightDualBound2*::"Rel_3('a,'b,'c) ⇒ Set('a) ⇒ Set('b) ⇒ Set('c)" ("$rightDualBound_2$")
where "$rightDualBound_2\ R\ \equiv \lambda A\ B.\ \lambda c.\ \exists a\ b.\ A\ a \leftharpoondown B\ b \leftharpoondown R\ a\ b\ c$"

declare *rightDualBound2_def[rel_defs]*

abbreviation(*input*) *dualBound123*::"Rel_3('a,'b,'c) ⇒ Set('a) ⇒ Set('b) ⇒ Set('c)" ("$\oslash_{123}$")
where "$\oslash_{123} \equiv$ $rightDualBound_2 \circ C_{123}$" — C_{123} as identity permutation is its own inverse (involutive)
abbreviation(*input*) *dualBound132*::"Rel_3('a,'b,'c) ⇒ Set('a) ⇒ Set('c) ⇒ Set('b)" ("$\oslash_{132}$")
where "$\oslash_{132} \equiv$ $rightDualBound_2 \circ C_{132}$" — C_{132} is its own inverse
abbreviation(*input*) *dualBound213*::"Rel_3('a,'b,'c) ⇒ Set('b) ⇒ Set('a) ⇒ Set('c)" ("$\oslash_{213}$")
where "$\oslash_{213} \equiv$ $rightDualBound_2 \circ C_{213}$" — C_{213} is its own inverse
abbreviation(*input*) *dualBound231*::"Rel_3('a,'b,'c) ⇒ Set('b) ⇒ Set('c) ⇒ Set('a)" ("$\oslash_{231}$")
where "$\oslash_{231} \equiv$ $rightDualBound_2 \circ C_{312}$" — C_{312}/L is the inverse of C_{231}/R

abbreviation$(input)$ $dualBound312::$"$Rel_3(\text{'a},\text{'b},\text{'c}) \Rightarrow Set(\text{'c}) \Rightarrow Set(\text{'a}) \Rightarrow Set(\text{'b})$" ("$\oslash_{312}$")
  where "$\oslash_{312} \equiv rightDualBound_2 \circ \mathtt{C}_{231}$" — $\mathtt{C}_{231}$/R is the inverse of $\mathtt{C}_{312}$/L
abbreviation$(input)$ $dualBound321::$"$Rel_3(\text{'a},\text{'b},\text{'c}) \Rightarrow Set(\text{'c}) \Rightarrow Set(\text{'b}) \Rightarrow Set(\text{'a})$" ("$\oslash_{321}$")
  where "$\oslash_{321} \equiv rightDualBound_2 \circ \mathtt{C}_{321}$" — $\mathtt{C}_{321}$ is its own inverse

notation$(input)$ $dualBound123$ ("_-$\oslash_{123}$") **and** $dualBound132$ ("_-$\oslash_{132}$") **and**
$dualBound213$ ("_-$\oslash_{213}$") **and** $dualBound231$ ("_-$\oslash_{231}$") **and**
$dualBound312$ ("_-$\oslash_{312}$") **and** $dualBound321$ ("_-$\oslash_{321}$")

lemma "$R\text{-}\oslash_{123} = (\lambda A\ B.\ \lambda c.\ \exists a\ b.\ A\ a \leftharpoonup B\ b \leftharpoonup R\ a\ b\ c)$"
lemma "$R\text{-}\oslash_{132} = (\lambda A\ C.\ \lambda b.\ \exists a\ c.\ A\ a \leftharpoonup C\ c \leftharpoonup R\ a\ b\ c)$"
lemma "$R\text{-}\oslash_{213} = (\lambda B\ A.\ \lambda c.\ \exists b\ a.\ B\ b \leftharpoonup A\ a \leftharpoonup R\ a\ b\ c)$"
lemma "$R\text{-}\oslash_{231} = (\lambda B\ C.\ \lambda a.\ \exists b\ c.\ B\ b \leftharpoonup C\ c \leftharpoonup R\ a\ b\ c)$"
lemma "$R\text{-}\oslash_{312} = (\lambda C\ A.\ \lambda b.\ \exists c\ a.\ C\ c \leftharpoonup A\ a \leftharpoonup R\ a\ b\ c)$"
lemma "$R\text{-}\oslash_{321} = (\lambda C\ B.\ \lambda a.\ \exists c\ b.\ C\ c \leftharpoonup B\ b \leftharpoonup R\ a\ b\ c)$"

Similarly, dual-bounds can also be similarly interrelated by permutation.

lemma "$R\text{-}\oslash_{123} = (\mathtt{C}_{132}\ R)\text{-}\oslash_{132}$"
lemma "$R\text{-}\oslash_{123} = (\mathtt{C}_{213}\ R)\text{-}\oslash_{213}$"
lemma "$R\text{-}\oslash_{123} = (\mathtt{C}_{231}\ R)\text{-}\oslash_{231}$"
lemma "$R\text{-}\oslash_{123} = (\mathtt{C}_{312}\ R)\text{-}\oslash_{312}$"
lemma "$R\text{-}\oslash_{123} = (\mathtt{C}_{321}\ R)\text{-}\oslash_{321}$"
lemma "$R\text{-}\oslash_{132} = (\mathtt{C}_{231}\ R)\text{-}\oslash_{213}$"
lemma "$R\text{-}\oslash_{132} = (\mathtt{C}_{321}\ R)\text{-}\oslash_{312}$"
lemma "$R\text{-}\oslash_{213} = (\mathtt{C}_{132}\ R)\text{-}\oslash_{312}$"
lemma "$R\text{-}\oslash_{213} = (\mathtt{C}_{231}\ R)\text{-}\oslash_{321}$"
lemma "$R\text{-}\oslash_{231} = (\mathtt{C}_{132}\ R)\text{-}\oslash_{321}$"
lemma "$R\text{-}\oslash_{231} = (\mathtt{C}_{231}\ R)\text{-}\oslash_{312}$"
lemma "$R\text{-}\oslash_{312} = (\mathtt{C}_{213}\ R)\text{-}\oslash_{321}$"
lemma "$R\text{-}\oslash_{312} = (\mathtt{C}_{231}\ R)\text{-}\oslash_{123}$"
lemma "$R\text{-}\oslash_{321} = (\mathtt{C}_{213}\ R)\text{-}\oslash_{312}$"
lemma "$R\text{-}\oslash_{321} = (\mathtt{C}_{321}\ R)\text{-}\oslash_{123}$"

Check dualities.

lemma $bound123\_dual$: "$-,--DUAL_2\ (R\text{-}\ominus_{123})\ (R\text{-}\oslash_{123})$"
lemma $bound132\_dual$: "$-,--DUAL_2\ (R\text{-}\ominus_{132})\ (R\text{-}\oslash_{132})$"
lemma $bound213\_dual$: "$-,--DUAL_2\ (R\text{-}\ominus_{213})\ (R\text{-}\oslash_{213})$"
lemma $bound231\_dual$: "$-,--DUAL_2\ (R\text{-}\ominus_{231})\ (R\text{-}\oslash_{231})$"
lemma $bound312\_dual$: "$-,--DUAL_2\ (R\text{-}\ominus_{312})\ (R\text{-}\oslash_{312})$"
lemma $bound321\_dual$: "$-,--DUAL_2\ (R\text{-}\ominus_{321})\ (R\text{-}\oslash_{321})$"

13.3 Transformations

We can always make a unary image/bound operator out of its binary counterpart as follows.

lemma "$R\text{-}\Diamond_{\rightarrow}\ A = (\mathtt{K}\ R)\text{-}\Diamond_{123}\ A\ A$"
lemma "$R\text{-}\ominus_{\rightarrow}\ A = (\mathtt{K}\ R)\text{-}\ominus_{123}\ A\ A$"
lemma "$R\text{-}\Diamond_{\leftarrow}\ A = (\mathtt{K}\ R^{\smile})\text{-}\Diamond_{123}\ A\ A$"
lemma "$R\text{-}\ominus_{\leftarrow}\ A = (\mathtt{K}\ R^{\smile})\text{-}\ominus_{123}\ A\ A$"

And the same holds for the dual variants.

lemma "$R\text{-}\Box_{\rightarrow}\ A = (\mathtt{K}\ R)\text{-}\Box_{123}\ (-A)\ A$"
lemma "$R\text{-}\oslash_{\rightarrow}\ A = (\mathtt{K}\ R)\text{-}\oslash_{123}\ (-A)\ A$"
lemma "$R\text{-}\Box_{\leftarrow}\ A = (\mathtt{K}\ R^{\smile})\text{-}\Box_{123}\ (-A)\ A$"
lemma "$R\text{-}\oslash_{\leftarrow}\ A = (\mathtt{K}\ R^{\smile})\text{-}\oslash_{123}\ (-A)\ A$"

The converse conversion is not possible in general:

proposition "$\forall T.\ \exists R.\ \forall A.\ (T\text{-}\Diamond_{123}\ A\ A) = (R\text{-}\Diamond_{\rightarrow}\ A)$" **nitpick** — countermodel found

In particular, this does not hold (for arbitrary T)

proposition "$(T\text{-}\Diamond_{123}\ A\ A) = ((\lambda a\ b.\ T\ a\ a\ b)\text{-}\Diamond_{\rightarrow}\ A)$" **nitpick** — countermodel found

13.4 Adjunctions

Check that similar adjunction conditions obtain among binary set-operators as for their unary counterparts.

13.4.1 Residuation and Coresiduation

Residuation (coresiduation) between $\Diamond$ and $\Box$ ($\ominus$ and $\oslash$) obtains when swapping second and third parameters.

lemma *image123_residuation:* "$(\subseteq),(\subseteq)$-ADJ_2 (R-$\Diamond_{123}$) (R-$\Box_{132}$)"
lemma *image132_residuation:* "$(\subseteq),(\subseteq)$-ADJ_2 (R-$\Diamond_{132}$) (R-$\Box_{123}$)"
lemma *image213_residuation:* "$(\subseteq),(\subseteq)$-ADJ_2 (R-$\Diamond_{213}$) (R-$\Box_{231}$)"
lemma *image231_residuation:* "$(\subseteq),(\subseteq)$-ADJ_2 (R-$\Diamond_{231}$) (R-$\Box_{213}$)"
lemma *image312_residuation:* "$(\subseteq),(\subseteq)$-ADJ_2 (R-$\Diamond_{312}$) (R-$\Box_{321}$)"
lemma *image321_residuation:* "$(\subseteq),(\subseteq)$-ADJ_2 (R-$\Diamond_{321}$) (R-$\Box_{312}$)"

lemma *bound123_coresiduation:* "$(\subseteq),(\subseteq)$-ADJ_2 ($- \circ_2$ (R-$\ominus_{123}$)) ($- \circ_2$ (R-$\oslash_{132}$))"
lemma *bound132_coresiduation:* "$(\subseteq),(\subseteq)$-ADJ_2 ($- \circ_2$ (R-$\ominus_{132}$)) ($- \circ_2$ (R-$\oslash_{123}$))"
lemma *bound213_coresiduation:* "$(\subseteq),(\subseteq)$-ADJ_2 ($- \circ_2$ (R-$\ominus_{213}$)) ($- \circ_2$ (R-$\oslash_{231}$))"
lemma *bound231_coresiduation:* "$(\subseteq),(\subseteq)$-ADJ_2 ($- \circ_2$ (R-$\ominus_{231}$)) ($- \circ_2$ (R-$\oslash_{213}$))"
lemma *bound312_coresiduation:* "$(\subseteq),(\subseteq)$-ADJ_2 ($- \circ_2$ (R-$\ominus_{312}$)) ($- \circ_2$ (R-$\oslash_{321}$))"
lemma *bound321_coresiduation:* "$(\subseteq),(\subseteq)$-ADJ_2 ($- \circ_2$ (R-$\ominus_{321}$)) ($- \circ_2$ (R-$\oslash_{312}$))"

13.4.2 Galois-connection and its Dual

(Dual)Galois-connections for pairs of $\ominus$ ($\oslash$) also obtain when swapping second and third parameters.

lemma *bound123_galois:* "$(\subseteq),(\subseteq)$-GAL_2 (R-$\ominus_{123}$) (R-$\ominus_{132}$)"
lemma *bound213_galois:* "$(\subseteq),(\subseteq)$-GAL_2 (R-$\ominus_{213}$) (R-$\ominus_{231}$)"
lemma *bound312_galois:* "$(\subseteq),(\subseteq)$-GAL_2 (R-$\ominus_{312}$) (R-$\ominus_{321}$)"

lemma *dualBound123_dualgalois:* "$(\supseteq),(\supseteq)$-GAL_2 (R-$\oslash_{123}$) (R-$\oslash_{132}$)"
lemma *dualBound213_dualgalois:* "$(\supseteq),(\supseteq)$-GAL_2 (R-$\oslash_{213}$) (R-$\oslash_{231}$)"
lemma *dualBound312_dualgalois:* "$(\supseteq),(\supseteq)$-GAL_2 (R-$\oslash_{312}$) (R-$\oslash_{321}$)"

13.4.3 Conjugation and its Dual

Similarly, (dual)conjugations for pairs of $\Diamond$ ($\Box$) obtain when swapping second and third parameters.

lemma *image123_conjugation:* "$(\subseteq),(\subseteq)$-GAL_2 ($- \circ_2$ (R-$\Diamond_{123}$)) ($- \circ_2$ (R-$\Diamond_{132}$))"
lemma *image213_conjugation:* "$(\subseteq),(\subseteq)$-GAL_2 ($- \circ_2$ (R-$\Diamond_{213}$)) ($- \circ_2$ (R-$\Diamond_{231}$))"
lemma *image312_conjugation:* "$(\subseteq),(\subseteq)$-GAL_2 ($- \circ_2$ (R-$\Diamond_{312}$)) ($- \circ_2$ (R-$\Diamond_{321}$))"

lemma *dualImage123_dualconjugation:* "$(\supseteq),(\supseteq)$-GAL_2 ($- \circ_2$ (R-$\Box_{123}$)) ($- \circ_2$ (R-$\Box_{132}$))"
lemma *dualImage213_dualconjugation:* "$(\supseteq),(\supseteq)$-GAL_2 ($- \circ_2$ (R-$\Box_{213}$)) ($- \circ_2$ (R-$\Box_{231}$))"
lemma *dualImage312_dualconjugation:* "$(\supseteq),(\supseteq)$-GAL_2 ($- \circ_2$ (R-$\Box_{312}$)) ($- \circ_2$ (R-$\Box_{321}$))"

end

14 Spaces

Spaces are sets of sets (of ... "points"). They are the main playground of mathematicians.

theory *spaces*
imports *endorelations*
begin

named_theorems *space_defs*

14.1 Spaces as Quantifiers and co.

Quantifiers are particular kinds of spaces.

term *"$\forall$ :: Space('a)"* — $\forall$ is the space that contains only the universe
lemma *All_simp1:"$\{\mathfrak{U}\}$ = $\forall$ "*
lemma *All_simp2:"$(\subseteq)$ $\mathfrak{U}$ = $\forall$ "*

term *"$\exists$ :: Space('a)"* — $\exists$ is the space that contains all but the empty set
lemma *Ex_simp1: "$\{\!\!\{\emptyset\}\!\!\}$ = $\exists$ "*
lemma *Ex_simp2: "$(\supseteq)$ $\emptyset$ = $\nexists$ "*

term *"$\nexists$ :: Space('a)"* — $\nexists$ is the space that contains only the empty set
lemma *nonEx_simp: "$\{\emptyset\}$ = $\nexists$ "*

In general, any property of sets corresponds to a space. For instance:

term *"unique :: Space('a)"* — unique is the space that contains all and only univalent sets (having at most one element)
term *"$\exists!$ :: Space('a)"* — $\exists!$ is the space that contains all and only singleton sets

lemma *unique_def2: "unique = $\nexists$ $\cup$ $\exists!$"*
lemma *singleton_def2: "$\exists!$ = $\exists$ $\cap$ unique"*
lemma *singleton_def3: "$\exists!A$ = ($\exists a$. A = {a})"*

Further convenient instances of spaces.

definition *upair::"Space('a)" ("$\exists_{\leq 2}$")* — $\exists_{\leq 2}$ contains the unordered pairs (sets with one or two elements)
  where ‹$\exists_{\leq 2} \equiv \exists^2 \circ (\mathbf{\Phi}_{21}\ (\cap^r)\ (\mathtt{W}\ (\times))\ (\mathtt{R}\ \mathtt{E}\ (\mathbf{\Psi}_2\ (\cup)\ \mathcal{Q})\ (\subseteq)))$›
definition *doubleton::"Space('a)" ("$\exists!_2$")* — $\exists!_2$ contains the doubletons (sets with two (different) elements)
  where ‹$\exists!_2 \equiv \exists_{\leq 2} \setminus \exists!$›

declare *unique_def[space_defs] singleton_def[space_defs] doubleton_def[space_defs] upair_def[space_de*

lemma *"$\exists_{\leq 2}A = \exists^2((A \times A) \cap^r (\lambda x\ y.\ A \subseteq \{x,y\}))$"*
lemma *doubleton_def2: "$\exists_{\leq 2}A = (\exists x\ y.\ A\ x \wedge A\ y \wedge (\forall z.\ A\ z \to (z = x \vee z = y)))$"*

lemma ‹$\exists!_2\ A = (\exists x\ y.\ x \neq y \wedge A\ x \wedge A\ y \wedge (\forall z.\ A\ z \to (z = x \vee z = y)))$›
lemma *upair_def2: "$\exists_{\leq 2} = \exists! \cup \exists!_2$"*

lemma *doubleton_def3: "$\exists!_2A = (\exists a\ b.\ a \neq b \wedge A = \{a,b\})$"*
lemma *upair_def3: "$\exists_{\leq 2}A = (\exists a\ b.\ A = \{a,b\})$"*

Convenient abbreviation for sets that have 2 or more elements.

abbreviation *(input) nonUnique::"Space('a)" ("$\exists_{\geq 2}$")*
  where *"$\exists_{\geq 2}A \equiv \neg$(unique A)"*

Sets, in general, are the bigunions of their contained singletons.

lemma *singleton_gen: "$S = \bigcup(\wp S \cap \exists!)$"*

Sets with more than one element are the bigunions of their contained doubletons.

lemma *doubleton_gen: "$\exists_{\geq 2}\ S \Longrightarrow S = \bigcup(\wp S \cap \exists!_2)$"*

Sets, in general, are the bigunions of their contained unordered pairs.

lemma *upair_gen: "$S = \bigcup(\wp S \cap \exists_{\leq 2})$"*

Some useful equations:

lemma *singleton_prop: "$(\forall D.\ D \subseteq S \to \exists!D \to P\ D) = (\forall x.\ S\ x \to P\ \{x\})$"*
lemma *doubleton_prop: "$(\forall D.\ D \subseteq S \to \exists!_2D \to P\ D) = (\forall x\ y.\ S\ x \wedge S\ y \to x \neq y \to P\ \{x,y\})$"*
lemma *upair_prop: "$(\forall D.\ D \subseteq S \to \exists_{\leq 2}D \to P\ D) = (\forall x\ y.\ S\ x \wedge S\ y \to P\ \{x,y\})$"*

14.2 Spaces via Closure under Operations

We obtain spaces by considering the set of sets closed under the given (n-ary) operation.

term *"f-closed$_1$ :: Space('a)"*
term *"g-closed$_2$ :: Space('a)"*
term *"F-closed$_G$ :: Space('a)"*
term *"φ-closed$_S$ :: Space('a)"*

14.3 Spaces from Endorelations

The following definitions correspond to functions that take an endorelation R and return the space of those sets satisfying a particular property wrt. R.

14.3.1 Lub- and Glb-related Definitions

These definitions generalize the "complete join/meet-semilattice" property (existence of suprema resp. infima).

definition *lubComplete::"ERel('a) $\Rightarrow$ Space('a)" ("_-lubComplete")*
where *"R-lubComplete $\equiv$ $\mathbf{\Phi}_{21}$ ($\subseteq$) $\wp$ ((R D (R-lub)) ($\sqcap$))"*
definition *glbComplete::"ERel('a) $\Rightarrow$ Space('a)" ("_-glbComplete")*
where *"R-glbComplete $\equiv$ $\mathbf{\Phi}_{21}$ ($\subseteq$) $\wp$ ((R D (R-glb)) ($\sqcap$))"* — all of S-subsets have a glb (wrt R) in S

declare *lubComplete_def[space_defs] glbComplete_def[space_defs]*

All of S-subsets have a lub (wrt R) in S.

lemma *"R-lubComplete = (λS. $\wp$ S $\subseteq$ ((R D (R-lub)) ($\sqcap$) S))"*
lemma *lubComplete_def2: "R-lubComplete = (λS. $\forall$D. D $\subseteq$ S $\longrightarrow$ (R-lub D $\sqcap$ S))"*

All of S-subsets have a glb (wrt R) in S.

lemma *"R-glbComplete = (λS. $\wp$ S $\subseteq$ ((R D (R-glb)) ($\sqcap$) S))"*
lemma *glbComplete_def2: "R-glbComplete = (λS. $\forall$D. D $\subseteq$ S $\longrightarrow$ (R-glb D $\sqcap$ S))"*

lemma *lubComplete_defT: "R-lubComplete = R$^{\smile}$-glbComplete"*
lemma *glbComplete_defT: "R-glbComplete = R$^{\smile}$-lubComplete"*

Limit-completeness of a relation can be expressed in terms of either lub- or glb-completeness.

lemma *limitComplete_def3: "limitComplete R = R-lubComplete $\mathfrak{U}$"*
lemma *limitComplete_def4: "limitComplete R = R-glbComplete $\mathfrak{U}$"*

Note that lub/glb-completeness is neither monotonic nor antitonic, for instance:

proposition *"A $\subseteq$ B $\Longrightarrow$ R-lubComplete A $\Longrightarrow$ R-lubComplete B"* **nitpick** — countermodel found
proposition *"A $\subseteq$ B $\Longrightarrow$ R-lubComplete B $\Longrightarrow$ R-lubComplete A"* **nitpick** — countermodel found

The following related propertes correspond to closure under the lub resp. glb set-operation wrt R.

definition *lubClosed::"ERel('a) $\Rightarrow$ Space('a)" ("_-lubClosed")*
where *"R-lubClosed $\equiv$ (R-lub)-closed$_S$"*
definition *glbClosed::"ERel('a) $\Rightarrow$ Space('a)" ("_-glbClosed")*
where *"R-glbClosed $\equiv$ (R-glb)-closed$_S$"*

declare *lubClosed_def[space_defs] glbClosed_def[space_defs]*

lemma *lubClosed_defT: "R-lubClosed = R$^{\smile}$-glbClosed"*
lemma *glbClosed_defT: "R-glbClosed = R$^{\smile}$-lubClosed"*

Recalling that antisymmetry entails uniqueness of lub/glb (when they exist), we have in fact.

lemma *lubComplete_lubClosed: "antisymmetric R $\Longrightarrow$ R-lubComplete S $\Longrightarrow$ R-lubClosed S"*
lemma *glbComplete_glbClosed: "antisymmetric R $\Longrightarrow$ R-glbComplete S $\Longrightarrow$ R-glbClosed S"*

However, being closed under lub/glb does not entail existence of lub/glb.

proposition *"$\exists$ S $\Longrightarrow$ R-lubClosed S $\Longrightarrow$ R-lubComplete S"* **nitpick** — countermodel found
proposition *"$\exists$ S $\Longrightarrow$ R-glbClosed S $\Longrightarrow$ R-glbComplete S"* **nitpick** — countermodel found

In fact, for limit-complete relations, closure under lub/glb does entail existence of lub/glb.

lemma *lubClosed_lubComplete: "limitComplete R $\Longrightarrow$ R-lubClosed S $\Longrightarrow$ R-lubComplete S"*
lemma *glbClosed_glbComplete: "limitComplete R $\Longrightarrow$ R-glbClosed S $\Longrightarrow$ R-glbComplete S"*

lemma *lubClosed_def2: "antisymmetric R $\Longrightarrow$ limitComplete R $\Longrightarrow$ R-lubComplete = R-lubClosed"*

lemma *glbClosed_def2: "antisymmetric R $\Longrightarrow$ limitComplete R $\Longrightarrow$ R-glbComplete = R-glbClosed"*

14.3.2 Upwards- and Downwards-Closure

definition *upwardsClosed::"ERel('a) $\Rightarrow$ Space('a)" ("_-upwardsClosed")*
  where *"R-upwardsClosed $\equiv$ (R-upImage)-closed$_S$"*
definition *downwardsClosed::"ERel('a) $\Rightarrow$ Space('a)" ("_-downwardsClosed")*
  where *"R-downwardsClosed $\equiv$ (R-downImage)-closed$_S$"*

declare *upwardsClosed_def[space_defs] downwardsClosed_def[space_defs]*

lemma *upwardsClosed_defT: "R-upwardsClosed = R$^\smile$-downwardsClosed"*
lemma *downwardsClosed_defT: "R-downwardsClosed = R$^\smile$-upwardsClosed"*

lemma *upwardsClosed_def2: "R-upwardsClosed S = ($\forall$x y. R x y $\longrightarrow$ S x $\longrightarrow$ S y)"*
lemma *downwardsClosed_def2: "R-downwardsClosed S = ($\forall$x y. R x y $\longrightarrow$ S y $\longrightarrow$ S x)"*

lemma *upwardsClosed_def3: "skeletal R $\Longrightarrow$ R-upwardsClosed S = ($\forall$D. $\exists$(R-glb D) $\longrightarrow$ (R-glb D) $\subseteq$ S $\longrightarrow$ D $\subseteq$ S)"*
lemma *downwardsClosed_def3: "skeletal R $\Longrightarrow$ R-downwardsClosed S = ($\forall$D. $\exists$(R-lub D) $\longrightarrow$ (R-lub D) $\subseteq$ S $\longrightarrow$ D $\subseteq$ S)"*

lemma *upwardsClosed_def4: "skeletal R $\Longrightarrow$ limitComplete R $\Longrightarrow$ R-upwardsClosed S = ($\forall$D. (R-glb D) $\subseteq$ S $\longrightarrow$ D $\subseteq$ S)"*
lemma *downwardsClosed_def4: "skeletal R $\Longrightarrow$ limitComplete R $\Longrightarrow$ R-downwardsClosed S = ($\forall$D. (R-lub D) $\subseteq$ S $\longrightarrow$ D $\subseteq$ S)"*

14.3.3 Existence of Greatest- and Least-Elements

Another interesting property is existence of greatest resp. least elements.

definition *greatestExist::"ERel('a) $\Rightarrow$ Space('a)" ("_-greatestExist")*
  where *"R-greatestExist $\equiv$ $\exists$ $\circ$ R-greatest"*
definition *leastExist::"ERel('a) $\Rightarrow$ Space('a)" ("_-leastExist")*
  where *"R-leastExist $\equiv$ $\exists$ $\circ$ R-least"*

declare *greatestExist_def[space_defs] leastExist_def[space_defs]*

In fact, recalling that:

lemma *"R-greatest S = (S $\cap$ R-upperBound S)"*
lemma *"R-least S = (S $\cap$ R-lowerBound S)"*

lemma *greatestExist_defT: "R-greatestExist = R$^\smile$-leastExist"*

lemma *leastExist_defT: "R-leastExist = R$^\smile$-greatestExist"*

We have that existence of greatest/least elements for S is equivalent to every S-subset having upper/lower bounds (wrt R) in S. (Note that this is a strong variant of up/downwards directedness.)

lemma *greatestExist_def2: "R-greatestExist S = (∀D. D ⊆ S ⟶ ∃(S ∩ R-upperBound D))"*
lemma *leastExist_def2: "R-leastExist S = (∀D. D ⊆ S ⟶ ∃(S ∩ R-lowerBound D))"*

Or, alternatively:

lemma *greatestExist_def3: "R-greatestExist S = (∃S ∧ (∀D. D ⊆ S ⟶ ∃D ⟶ ∃(S ∩ R-upperBound D)))"*
lemma *leastExist_def3: "R-leastExist S = (∃S ∧ (∀D. D ⊆ S ⟶ ∃D ⟶ ∃(S ∩ R-lowerBound D)))"*

In fact, existence of greatest/least-elements is a strictly weaker property than lub/glb-completeness.

lemma *glbComplete_least: "R-glbComplete ⊆ R-leastExist"*
lemma *lubComplete_greatest: "R-lubComplete ⊆ R-greatestExist"*
proposition *"R-greatestExist ⊆ R-lubComplete"* **nitpick** — countermodel found
proposition *"R-leastExist ⊆ R-glbComplete"* **nitpick** — countermodel found

lemma *greatestExist_lubClosed: "R-downwardsClosed S ⟹ R-greatestExist S ⟹ R-lubClosed S"*
lemma *leastExist_glbClosed: "R-upwardsClosed S ⟹ R-leastExist S ⟹ R-glbClosed S"*

lemma *greatestExist_def4: "limitComplete R ⟹ R-downwardsClosed S ⟹ R-greatestExist S = R-lubClosed S"*
lemma *leastExist_def4: "limitComplete R ⟹ R-upwardsClosed S ⟹ R-leastExist S = R-glbClosed S"*

14.3.4 Upwards- and Downwards-Directedness

The property of a set being "up/downwards directed" wrt. an endorelation: Every pair of S-elements has upper/lower-bounds (wrt R) in S.

definition *upwardsDirected::"ERel('a) ⇒ Space('a)" ("_-upwardsDirected")*
  where *"R-upwardsDirected ≡ $\mathbf{\Phi}_{21}$ (⊆^{r}) (W (×)) (R E ($\mathbf{\Psi}_2$ (∩) R) (⊓))"*
definition *downwardsDirected::"ERel('a) ⇒ Space('a)" ("_-downwardsDirected")*
  where *"R-downwardsDirected ≡ $\mathbf{\Phi}_{21}$ (⊆^{r}) (W (×)) (R E ($\mathbf{\Psi}_2$ (∩) (C R)) (⊓))"*

declare *upwardsDirected_def[space_defs] downwardsDirected_def[space_defs]*

lemma *"R-upwardsDirected = (λS. (S × S) ⊆^{r} (λa b. S ⊓ ($\mathbf{\Psi}_2$ (∩) R a b)))"*
lemma *"R-upwardsDirected = (λS. (S × S) ⊆^{r} (λa b. S ⊓ (R a ∩ R b)))"*
lemma *"R-upwardsDirected = (λS. ∀a b. S a ∧ S b ⟶ (∃c. S c ∧ R a c ∧ R b c))"*

lemma *"R-downwardsDirected = (λS. (S × S) ⊆^{r} (λa b. S ⊓ ($\mathbf{\Psi}_2$ (∩) (C R) a b)))"*
lemma *"R-downwardsDirected = (λS. (S × S) ⊆^{r} (λa b. S ⊓ (R$^\smile$ a ∩ R$^\smile$ b)))"*
lemma *"R-downwardsDirected = (λS. ∀a b. S a ∧ S b ⟶ (∃c. S c ∧ R c a ∧ R c b))"*

lemma *upwardsDirected_defT: "R-upwardsDirected = R$^\smile$-downwardsDirected"*
lemma *downwardsDirected_defT: "R-downwardsDirected = R$^\smile$-upwardsDirected"*

The definition above can be rephrased as:

lemma *upwardsDirected_def2: "R-upwardsDirected S = (∀a b. S a ∧ S b ⟶ ∃(S ∩ R-upperBound {a,b}))"*
lemma *downwardsDirected_def2: "R-downwardsDirected S = (∀a b. S a ∧ S b ⟶ ∃(S ∩ R-lowerBound {a,b}))"*

or, alternatively:

lemma *upwardsDirected_def3:* "R-upwardsDirected S = ($\forall$D. D $\subseteq$ S $\longrightarrow$ $\exists_{\leq 2}$D $\longrightarrow$ $\exists$(S $\cap$ R-upperBound D))"
lemma *downwardsDirected_def3:* "R-downwardsDirected S = ($\forall$D. D $\subseteq$ S $\longrightarrow$ $\exists_{\leq 2}$D $\longrightarrow$ $\exists$(S $\cap$ R-lowerBou D))"

Note that up/downwards directedness does not entail non-emptyness of S.

proposition "R-upwardsDirected S $\longrightarrow$ $\exists$S" **nitpick** — countermodel found
proposition "R-downwardsDirected S $\longrightarrow$ $\exists$S" **nitpick** — countermodel found

14.3.5 Join- and Meet-Closure

Convenient abbreviations for joins resp. meets (lub resp. glb of sets with 2 elements).

abbreviation(input) join ("_-join")
  where "R-join a b $\equiv$ R-lub {a,b}"
abbreviation(input) meet ("_-meet")
  where "R-meet a b $\equiv$ R-glb {a,b}"

Some platitudes about meets and joins.

lemma *join_prop1:* "R-lowerBound (R-join a b) a"
lemma *join_prop2:* "R-lowerBound (R-join a b) b"
lemma *meet_prop1:* "R-upperBound (R-meet a b) a"
lemma *meet_prop2:* "R-upperBound (R-meet a b) b"

The following are weaker versions of lub/glb-closure customarily used in the literature.

definition *joinClosed::"ERel('a) $\Rightarrow$ Space('a)" ("_-joinClosed")*
  where "R-joinClosed $\equiv$ $\mathbf{\Phi}_{21}$ ($\subseteq^r$) (W ($\times$)) (**R** **E** (R-join) $\wp$)"
definition *meetClosed::"ERel('a) $\Rightarrow$ Space('a)" ("_-meetClosed")*
  where "R-meetClosed $\equiv$ $\mathbf{\Phi}_{21}$ ($\subseteq^r$) (W ($\times$)) (**R** **E** (R-meet) $\wp$)"

declare *joinClosed_def[space_defs] meetClosed_def[space_defs]*

lemma "R-joinClosed = (λS. (S $\times$ S) $\subseteq^r$ (**R** **E** (R-join) $\wp$ S))"
lemma "R-joinClosed = (λS. (S $\times$ S) $\subseteq^r$ (λa b. R-join a b $\subseteq$ S))"
lemma "R-joinClosed = (λS. $\forall$a b. S a $\wedge$ S b $\longrightarrow$ R-join a b $\subseteq$ S)"

lemma "R-meetClosed = (λS. (S $\times$ S) $\subseteq^r$ (**R** **E** (R-meet) $\wp$ S))"
lemma "R-meetClosed = (λS. (S $\times$ S) $\subseteq^r$ (λa b. R-meet a b $\subseteq$ S))"
lemma "R-meetClosed = (λS. $\forall$a b. S a $\wedge$ S b $\longrightarrow$ R-meet a b $\subseteq$ S)"

lemma *joinClosed_defT:* "R-joinClosed = R$^{\smile}$-meetClosed"
lemma *meetClosed_defT:* "R-meetClosed = R$^{\smile}$-joinClosed"

lemma *joinClosed_def2:* "joinClosed R S = ($\forall$p. p $\subseteq$ S $\rightarrow$ $\exists_{\leq 2}$ p $\rightarrow$ (R-lub p) $\subseteq$ S)"
lemma *meetClosed_def2:* "meetClosed R S = ($\forall$p. p $\subseteq$ S $\rightarrow$ $\exists_{\leq 2}$ p $\rightarrow$ (R-glb p) $\subseteq$ S)"

lemma *joinClosed_upwardsDirected:* "limitComplete R $\Longrightarrow$ R-joinClosed S $\Longrightarrow$ R-upwardsDirected S"
lemma *meetClosed_downwardsDirected:* "limitComplete R $\Longrightarrow$ R-meetClosed S $\Longrightarrow$ R-downwardsDirected S"

Thus we have:

lemma *greatestExist_upwardsDirected:* "R-greatestExist S $\Longrightarrow$ R-upwardsDirected S"
lemma *leastExist_downwardsDirected:* "R-leastExist S $\Longrightarrow$ R-downwardsDirected S"

Note, however:

proposition "$\exists S \Longrightarrow$ *R-upwardsDirected S* $\Longrightarrow$ *R-greatestExist S*" **nitpick** — countermodel found
proposition "$\exists S \Longrightarrow$ *R-downwardsDirected S* $\Longrightarrow$ *R-leastExist S*" **nitpick** — countermodel found

lemma *downwardsDirected_meetClosed:* "*R-upwardsClosed S* $\Longrightarrow$ *R-downwardsDirected S* $\Longrightarrow$ *R-meetClosed S*"
lemma *upwardsDirected_joinClosed:* "*R-downwardsClosed S* $\Longrightarrow$ *R-upwardsDirected S* $\Longrightarrow$ *R-joinClosed S*"

lemma *downwardsDirected_def4:* "*limitComplete R* $\Longrightarrow$ *R-upwardsClosed S* $\Longrightarrow$ *R-downwardsDirected S = R-meetClosed S*"
lemma *upwardsDirected_def4:* "*limitComplete R* $\Longrightarrow$ *R-downwardsClosed S* $\Longrightarrow$ *R-upwardsDirected S = R-joinClosed S*"

14.3.6 Ideals and Filters

definition *pseudoFilter::"ERel('a) $\Rightarrow$ Space('a)" ("_-pseudoFilter")*
  where "*R-pseudoFilter* $\equiv \mathbf{\Phi}_{21}$ ($\subseteq^r$) (R E *R-meet* $\wp$) (W ($\times$))"
definition *pseudoIdeal::"ERel('a) $\Rightarrow$ Space('a)" ("_-pseudoIdeal")*
  where "*R-pseudoIdeal* $\equiv \mathbf{\Phi}_{21}$ ($\subseteq^r$) (R E *R-join* $\wp$) (W ($\times$))"

declare *pseudoFilter_def[space_defs] pseudoIdeal_def[space_defs]*

lemma "*R-pseudoFilter* = (λS. (R E *R-meet* $\wp$ *S*) $\subseteq^r$ (*S* $\times$ *S*))"
lemma "*R-pseudoFilter* = (λS. $\forall$ *a b. R-meet a b* $\subseteq$ *S* $\longrightarrow$ (*S a* $\wedge$ *S b*))"

lemma "*R-pseudoIdeal* = (λS. (R E *R-join* $\wp$ *S*) $\subseteq^r$ (*S* $\times$ *S*))"
lemma "*R-pseudoIdeal* = (λS. $\forall$ *a b. R-join a b* $\subseteq$ *S* $\longrightarrow$ (*S a* $\wedge$ *S b*))"

lemma *pseudoFilter_defT:* "*R-pseudoFilter* = $R^{\smile}$*-pseudoIdeal*"
lemma *pseudoIdeal_defT:* "*R-pseudoIdeal* = $R^{\smile}$*-pseudoFilter*"

lemma *pseudoFilter_upwardsClosed:* "*skeletal R* $\Longrightarrow$ *R-pseudoFilter S* $\Longrightarrow$ *R-upwardsClosed S*"

lemma *pseudoIdeal_downwardsClosed:* "*skeletal R* $\Longrightarrow$ *R-pseudoIdeal S* $\Longrightarrow$ *R-downwardsClosed S*"

lemma *upwardsClosed_pseudoFilter:* "*limitComplete R* $\Longrightarrow$ *R-upwardsClosed S* $\Longrightarrow$ *R-pseudoFilter S*"
lemma *downwardsClosed_pseudoIdeal:* "*limitComplete R* $\Longrightarrow$ *R-downwardsClosed S* $\Longrightarrow$ *R-pseudoIdeal S*"

lemma *pseudoFilter_def2:* "*skeletal R* $\Longrightarrow$ *limitComplete R* $\Longrightarrow$ *R-pseudoFilter S = R-upwardsClosed S*"
lemma *pseudoIdeal_def2:* "*skeletal R* $\Longrightarrow$ *limitComplete R* $\Longrightarrow$ *R-pseudoIdeal S = R-downwardsClosed S*"

The following notions abstract the order-theoretical property of filter/ideal towards relations in general: S is a filter/ideal when all and only pairs of S-elements have their meet/join (wrt R) in S.

abbreviation *(input) filter::"ERel('a) $\Rightarrow$ Space('a)" ("_-filter")*
  where "*R-filter S* $\equiv$ *R-pseudoFilter S* $\wedge$ *R-meetClosed S*"
abbreviation *(input) ideal::"ERel('a) $\Rightarrow$ Space('a)" ("_-ideal")*
  where "*R-ideal S* $\equiv$ *R-pseudoIdeal S* $\wedge$ *R-joinClosed S*"

lemma *filter_defT:* "*R-filter S* = $R^{\smile}$*-ideal S*"
lemma *ideal_defT:* "*R-ideal S* = $R^{\smile}$*-filter S*"

lemma *filter_char:* "*R-filter S* = ($\forall$ *a b. R-meet a b* $\subseteq$ *S* $\longleftrightarrow$ *S a* $\wedge$ *S b*)"

lemma *ideal_char: "R-ideal S = (∀ a b. R-join a b ⊆ S ⟷ S a ∧ S b)"*

lemma *filter_prop1: "limitComplete R ⟹ R-upwardsClosed S ⟹ R-downwardsDirected S ⟹ R-filter S"*
lemma *filter_prop2: "limitComplete R ⟹ R-filter S ⟹ R-downwardsDirected S"*
lemma *filter_prop3: "partial_order R ⟹ limitComplete R ⟹ R-filter S ⟹ R-upwardsClosed S"*

lemma *ideal_prop1: "limitComplete R ⟹ R-downwardsClosed S ⟹ R-upwardsDirected S ⟹ R-ideal S"*
lemma *ideal_prop2: "limitComplete R ⟹ R-ideal S ⟹ R-upwardsDirected S"*
lemma *ideal_prop3: "partial_order R ⟹ limitComplete R ⟹ R-ideal S ⟹ R-downwardsClosed S"*

lemma *filter_def2: "partial_order R ⟹ limitComplete R ⟹ R-filter = (R-upwardsClosed ∩ R-downwardsDirected)"*
lemma *ideal_def2: "partial_order R ⟹ limitComplete R ⟹ R-ideal = (R-downwardsClosed ∩ R-upwardsDirected)"*

14.3.7 Well-Founded- and Well-Ordered-Sets

Well-foundedness of sets wrt. a given relation (as in "Nat is well-founded wrt. < ").

definition *wellFoundedSet::"ERel('a) ⇒ Space('a)" ("_-wellFoundedSet")*
  where *"wellFoundedSet ≡ B₁₁ (⊇) (∃ ∘₂ min) (((∩) ∃) ∘ (⊇))"*
definition *wellOrderedSet::"ERel('a) ⇒ Space('a)" ("_-wellOrderedSet")*
  where *"wellOrderedSet ≡ B₁₁ (⊇) (∃ ∘₂ least) (((∩) ∃) ∘ (⊇))"*

declare *wellFoundedSet_def[endorel_defs] wellOrderedSet_def[endorel_defs]*

Every non-empty S-subset S has min elements (in D).

lemma *wellFoundedSet_def2: "R-wellFoundedSet S = (∀D. D ⊆ S ⟶ ∃D ⟶ ∃(R-min D))"*

Every non-empty S-subset D has least elements (in D).

lemma *wellOrderedSet_def2: "R-wellOrderedSet S = (∀D. D ⊆ S ⟶ ∃D ⟶ ∃(R-least D))"*

As expected, we have:

lemma *"wellFounded R = R-wellFoundedSet 𝔘"*
lemma *"wellOrdered R = R-wellOrderedSet 𝔘"*

For non-empty sets, well-orderedness entails existence of least elements (but not the other way round).

lemma *"∃S ⟹ R-wellOrderedSet S ⟹ R-leastExist S"*
proposition *"∃S ⟹ R-leastExist S ⟹ R-wellOrderedSet S"* **nitpick** — countermodel found

lemma *"(⊆)-wellFoundedSet {{1::nat},{2},{1,2}}"*
lemma *"¬ (⊆)-wellOrderedSet {{1::nat},{2},{1,2}}"*

end

References

[1] H. B. Curry. The universal quantifier in combinatory logic. *Annals of Mathematics*, 32(1):154–180, 1931.

[2] J. M. Font. On substructural logics preserving degrees of truth. *Bulletin of the Section of Logic*, 36(3/4):117–129, 2007.

[3] M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische annalen*, 92(3):305–316, 1924.

[4] R. M. Smullyan. *To Mock a Mockingbird: and other logic puzzles including an amazing adventure in combinatory logic.* Oxford University Press, USA, 2000.

[5] R. Wójcicki. Some remarks on the consequence operation in sentential logics. *Fundamenta Mathematicae*, 68(1), 1970.