

Stream Examples

Jasmin Christian Blanchette Andrei Popescu
Dmitriy Traytel

January 30, 2015

Contents

1	Sum	1
2	Onetwo	2
3	Shuffle product	2
4	Exponentiation	3
5	Supremum	4
6	Skewed product	4
7	Coinduction Up-To Congruence	5
8	Proofs by Coinduction Up-To Congruence	12
9	Two Examples of Fibonacci streams	13
10	Streams of Factorials	14
11	Mixed Recursion-Corecursion	17

1 Sum

definition $pls :: stream \Rightarrow stream \Rightarrow stream$ **where**

$pls\ xs\ ys = dtor-corec-J\ (\lambda(xs, ys). (head\ xs + head\ ys, Inr\ (tail\ xs, tail\ ys)))$
 (xs, ys)

lemma $head-pls[simp]: head\ (pls\ xs\ ys) = head\ xs + head\ ys$

unfolding $pls-def\ J.dtor-corec\ map-pre-J-def\ BNF-Comp.id-bnf-comp-def$ **by**
 $simp$

lemma $tail-pls[simp]: tail\ (pls\ xs\ ys) = pls\ (tail\ xs)\ (tail\ ys)$

unfolding *pls-def J.dtor-corec map-pre-J-def BNF-Comp.id-bnf-comp-def* **by** *simp*

lemma *pls-code[code]: pls xs ys = SCons (head xs + head ys) (pls (tail xs) (tail ys))*
by (*metis J.ctor-dtor prod.collapse head-pls tail-pls*)

lemma *pls-uniform: pls xs ys = alg₀1 (xs, ys)*
unfolding *pls-def*
apply (*rule fun-cong[OF sym[OF J.dtor-corec-unique]]*)
unfolding *alg₀1*
by (*simp add: map-pre-J-def BNF-Comp.id-bnf-comp-def fun-eq-iff convol-def₀1-def alg₀1-def*)

2 Onetwo

definition *onetwo :: stream where*
onetwo = corecUU0 (λ-. GUARD0 (1, SCONS0 (2, CONT0 ()))) ()

lemma *onetwo-code[code]: onetwo = SCons 1 (SCons 2 onetwo)*
apply (*subst onetwo-def*)
unfolding *corecUU0*
by (*simp add: map-pre-J-def BNF-Comp.id-bnf-comp-def J.dtor-ctor eval0-leaf0' o-eq-dest[OF eval0-gg0] o-eq-dest[OF gg0-natural] onetwo-def*)

definition *onetwo' :: stream where*
onetwo' = corecUU0 (λ-. SCONS0 (1, GUARD0 (2, CONT0 ()))) ()

lemma *onetwo'-code[code]: onetwo' = SCons 1 (SCons 2 onetwo')*
apply (*subst onetwo'-def*)
unfolding *corecUU0*
by (*simp add: map-pre-J-def BNF-Comp.id-bnf-comp-def J.dtor-ctor eval0-leaf0' o-eq-dest[OF eval0-gg0] o-eq-dest[OF gg0-natural] onetwo'-def*)

definition *stutter :: stream where*
stutter = corecUU1 (λ-. SCONS1 (1, GUARD1 (1, PLS1 (CONT1 ()), CONT1 ()))) ()

lemma *stutter-code[code]: stutter = SCons 1 (SCons 1 (pls stutter stutter))*
apply (*subst stutter-def*)
unfolding *corecUU1 prod.case*
by (*simp add: map-pre-J-def BNF-Comp.id-bnf-comp-def J.dtor-ctor eval1-leaf1' eval1-_{op}1 alg₁1-Inr o-eq-dest[OF Abs- Σ 1-natural] o-eq-dest[OF eval1-gg1] o-eq-dest[OF gg1-natural] pls-uniform stutter-def*)

3 Shuffle product

definition *prd :: stream \Rightarrow stream \Rightarrow stream where*

$prd\ xs\ ys = corecUU1\ (\lambda(xs, ys).\ GUARD1\ (head\ xs * head\ ys,$
 $PLS1\ (CONT1\ (xs, tail\ ys), CONT1\ (tail\ xs, ys))))\ (xs, ys)$

lemma $head\text{-}prd[simp]$: $head\ (prd\ xs\ ys) = head\ xs * head\ ys$
unfolding $prd\text{-}def\ corecUU1$
by ($simp\ add$: $map\text{-}pre\text{-}J\text{-}def\ BNF\text{-}Comp.id\text{-}bnf\text{-}comp\text{-}def\ J.dtor\text{-}ctor\ eval1\text{-}leaf1'$)

lemma $tail\text{-}prd[simp]$: $tail\ (prd\ xs\ ys) = pls\ (prd\ xs\ (tail\ ys))\ (prd\ (tail\ xs)\ ys)$
apply ($subst\ prd\text{-}def$)
unfolding $corecUU1\ prod.case$
by ($simp\ add$: $map\text{-}pre\text{-}J\text{-}def\ BNF\text{-}Comp.id\text{-}bnf\text{-}comp\text{-}def\ J.dtor\text{-}ctor\ eval1\text{-}leaf1'$
 $eval1\text{-}op1\ alg\Lambda1\text{-}Inr\ o\text{-}eq\text{-}dest[OF\ Abs\text{-}\Sigma1\text{-}natural]\ pls\text{-}uniform\ prd\text{-}def$)

lemma $prd\text{-}code[code]$: $prd\ xs\ ys = SCons\ (head\ xs * head\ ys)\ (pls\ (prd\ xs\ (tail\ ys))\ (prd\ (tail\ xs)\ ys))$
by ($metis\ J.ctor\text{-}dtor\ prod.collapse\ head\text{-}prd\ tail\text{-}prd$)

lemma $prd\text{-}uniform$: $prd\ xs\ ys = alg\varrho2\ (xs, ys)$
unfolding $prd\text{-}def$
apply ($rule\ fun\text{-}cong[OF\ sym[OF\ corecUU1\text{-}unique]]$)
apply ($rule\ iffD1[OF\ dtor\text{-}J\text{-}o\text{-}inj]$)
unfolding $alg\varrho2$
apply ($simp\ add$: $map\text{-}pre\text{-}J\text{-}def\ BNF\text{-}Comp.id\text{-}bnf\text{-}comp\text{-}def\ fun\text{-}eq\text{-}iff\ J.dtor\text{-}ctor\ \varrho2\text{-}def\ Let\text{-}def\ convol\text{-}def\ eval2\text{-}op2\ eval1\text{-}op1\ eval1\text{-}leaf1'$
 $o\text{-}eq\text{-}dest[OF\ Abs\text{-}\Sigma1\text{-}natural]\ o\text{-}eq\text{-}dest[OF\ Abs\text{-}\Sigma2\text{-}natural]\ alg\Lambda2\text{-}Inl\ alg\varrho2\text{-}def$)
done

abbreviation $scale\ n\ s \equiv prd\ (sconst\ n)\ s$

4 Exponentiation

definition $Exp :: stream \Rightarrow stream$ **where**

$Exp = corecUU2\ (\lambda xs.\ GUARD2\ (exp\ (head\ xs), PRD2\ (END2\ (tail\ xs), CONT2\ xs)))$

lemma $head\text{-}Exp[simp]$: $head\ (Exp\ xs) = exp\ (head\ xs)$
unfolding $Exp\text{-}def\ corecUU2$
by ($simp\ add$: $map\text{-}pre\text{-}J\text{-}def\ BNF\text{-}Comp.id\text{-}bnf\text{-}comp\text{-}def\ J.dtor\text{-}ctor\ eval2\text{-}leaf2'$)

lemma $tail\text{-}Exp[simp]$: $tail\ (Exp\ xs) = prd\ (tail\ xs)\ (Exp\ xs)$
apply ($subst\ Exp\text{-}def$)
unfolding $corecUU2$
by ($simp\ add$: $map\text{-}pre\text{-}J\text{-}def\ BNF\text{-}Comp.id\text{-}bnf\text{-}comp\text{-}def\ J.dtor\text{-}ctor\ eval2\text{-}leaf2'$
 $eval2\text{-}op2\ alg\Lambda2\text{-}Inr\ o\text{-}eq\text{-}dest[OF\ Abs\text{-}\Sigma2\text{-}natural]\ prd\text{-}uniform\ Exp\text{-}def$)

lemma $Exp\text{-}code[code]$: $Exp\ xs = SCons\ (exp\ (head\ xs))\ (prd\ (tail\ xs)\ (Exp\ xs))$
by ($metis\ J.ctor\text{-}dtor\ prod.collapse\ head\text{-}Exp\ tail\text{-}Exp$)

lemma $Exp\text{-}uniform$: $Exp\ xs = alg\varrho3\ (K3.I\ xs)$

```

unfolding Exp-def
apply (rule fun-cong[OF sym[OF corecUU2-unique]])
apply (rule iffD1[OF dtor-J-o-inj])
unfolding alg $\varrho$ 3 o-def[symmetric] o-assoc
apply (simp add: map-pre-J-def BNF-Comp.id-bnf-comp-def fun-eq-iff J.dtor-ctor
   $\varrho$ 3-def Let-def convol-def eval3-op3 eval2-op2 eval2-leaf2' eval3-leaf3'
  o-eq-dest[OF Abs- $\Sigma$ 2-natural] o-eq-dest[OF Abs- $\Sigma$ 3-natural] alg $\Lambda$ 3-Inl alg $\varrho$ 3-def)
done

```

5 Supremum

definition $\text{sup} :: \text{stream fset} \Rightarrow \text{stream}$ **where**
 $\text{sup} = \text{dtor-corec-J } (\lambda F. (\text{fMax } (\text{head } |' | F), \text{Inr } (\text{tail } |' | F)))$

lemma $\text{head-sup}[simp]: \text{head } (\text{sup } F) = \text{fMax } (\text{head } |' | F)$
unfolding sup-def J.dtor-corec map-pre-J-def BNF-Comp.id-bnf-comp-def **by** simp

lemma $\text{tail-sup}[simp]: \text{tail } (\text{sup } F) = \text{sup } (\text{tail } |' | F)$
unfolding sup-def J.dtor-corec map-pre-J-def BNF-Comp.id-bnf-comp-def **by** simp

lemma $\text{sup-code}[code]: \text{sup } F = \text{SCons } (\text{fMax } (\text{head } |' | F)) (\text{sup } (\text{tail } |' | F))$
by (metis J.ctor-dtor prod.collapse head-sup tail-sup)

lemma $\text{sup-uniform}: \text{sup } F = \text{alg}\varrho_4 F$
unfolding sup-def
apply (rule fun-cong[OF sym[OF J.dtor-corec-unique]])
unfolding alg ϱ 4
by (simp add: map-pre-J-def BNF-Comp.id-bnf-comp-def fun-eq-iff convol-def
 ϱ 4-def alg ϱ 4-def o-def)

6 Skewed product

definition $\text{prd}' :: \text{stream} \Rightarrow \text{stream} \Rightarrow \text{stream}$ **where**
 $\text{prd}' xs ys = \text{corecUU5 } (\lambda(xs, ys). \text{GUARD5 } (\text{head } xs * \text{head } ys,$
 $\text{PRD5 } (\text{CONT5 } (xs, \text{tail } ys), \text{PLS5 } (\text{END5 } (\text{tail } xs), \text{END5 } ys)))) (xs, ys)$

lemma $\text{prd}'\text{-uniform}: \text{prd}' xs ys = \text{alg}\varrho_5 (xs, ys)$
unfolding prd'-def
apply (rule fun-cong[OF sym[OF corecUU5-unique]])
apply (rule iffD1[OF dtor-J-o-inj])
unfolding alg ϱ 5
apply (simp add: map-pre-J-def BNF-Comp.id-bnf-comp-def fun-eq-iff J.dtor-ctor
 ϱ 5-def Let-def convol-def eval5-op5 eval4-op4 eval3-op3 eval2-op2 eval1-op1
eval5-leaf5'
o-eq-dest[OF Abs- Σ 1-natural] o-eq-dest[OF Abs- Σ 2-natural] o-eq-dest[OF Abs- Σ 3-natural]
o-eq-dest[OF Abs- Σ 4-natural] o-eq-dest[OF Abs- Σ 5-natural] alg Λ 5-Inl alg ϱ 5-def)

done

lemma *head-prd'*[simp]: *head* (*prd' xs ys*) = *head xs* * *head ys*
unfolding *prd'-def corecUU5*
by (*simp add: map-pre-J-def BNF-Comp.id-bnf-comp-def J.dtor-ctor eval5-leaf5'*)

lemma *tail-prd'*[simp]: *tail* (*prd' xs ys*) = *prd'* (*prd' xs* (*tail ys*)) (*pls* (*tail xs*) *ys*)
apply (*subst prd'-def, subst (2) prd'-uniform*)
unfolding *corecUU5 prod.case*
by (*simp add: map-pre-J-def BNF-Comp.id-bnf-comp-def J.dtor-ctor*
eval5-op5 eval4-op4 eval3-op3 eval2-op2 eval1-op1 eval5-leaf5'
algΛ5-Inr algΛ5-Inl algΛ4-Inl algΛ3-Inl algΛ2-Inl algΛ1-Inr
o-eq-dest[OF Abs-Σ5-natural] o-eq-dest[OF Abs-Σ4-natural]
o-eq-dest[OF Abs-Σ3-natural] o-eq-dest[OF Abs-Σ2-natural] o-eq-dest[OF
Abs-Σ1-natural]
pls-uniform prd'-def)

lemma *prd'-code*[code]:
prd' xs ys = *SCons* (*head xs* * *head ys*) (*prd'* (*prd' xs* (*tail ys*)) (*pls* (*tail xs*) *ys*))
by (*metis J.ctor-dtor prod.collapse head-prd' tail-prd'*)

7 Coinduction Up-To Congruence

lemma *SCons-uniform*: *SCons x s* = *eval0* (*gg0* (*x*, *leaf0 s*))
by (*rule iffD1[OF J.dtor-inject]*)
(simp add: map-pre-J-def BNF-Comp.id-bnf-comp-def J.dtor-ctor o-eq-dest[OF
eval0-gg0] eval0-leaf0')

lemma *genCngdd0-SCons*: $\llbracket x1 = x2; \text{genCngdd0 } R \text{ } xs1 \text{ } xs2 \rrbracket \implies$
genCngdd0 R (*SCons x1 xs1*) (*SCons x2 xs2*)
unfolding *SCons-uniform*
apply (*rule genCngdd0-eval0*)
apply (*rule rel-funD[OF gg0-transfer]*)
unfolding *rel-pre-J-def BNF-Comp.id-bnf-comp-def vimage2p-def*
apply (*rule rel-funD[OF rel-funD[OF Pair-transfer], rotated]*)
apply (*erule rel-funD[OF leaf0-transfer]*)
apply *assumption*
done

lemma *genCngdd0-genCngdd1*: *genCngdd0 R xs ys* \implies *genCngdd1 R xs ys*
unfolding *genCngdd0-def cngdd0-def cptdd0-def genCngdd1-def cngdd1-def cptdd1-def*
eval1-embL1[symmetric]
apply (*intro allI impI*)
apply (*erule conjE*) +
apply (*drule spec*)
apply (*erule mp conjI*) +
apply (*erule rel-funD[OF rel-funD[OF comp-transfer]]*)
apply (*rule embL1-transfer*)
done

```

lemma genCngdd1-SCons:  $\llbracket x1 = x2; \text{genCngdd1 } R \text{ } x1 \text{ } x2 \rrbracket \implies$ 
  genCngdd1 R (SCons x1 x1) (SCons x2 x2)
apply (subst I1.idem-Cl[symmetric])
apply (rule genCngdd0-genCngdd1)
apply (rule genCngdd0-SCons)
apply auto
done

lemma genCngdd1-pls:  $\llbracket \text{genCngdd1 } R \text{ } x1 \text{ } x2; \text{genCngdd1 } R \text{ } y1 \text{ } y2 \rrbracket \implies$ 
  genCngdd1 R (pls x1 y1) (pls x2 y2)
unfolding pls-uniform algo1-def o-apply
apply (rule genCngdd1-eval1)
apply (rule rel-funD[OF K1-as- $\Sigma\Sigma 1$ -transfer])
apply simp
done

lemma genCngdd1-genCngdd2:  $\text{genCngdd1 } R \text{ } x \text{ } y \implies \text{genCngdd2 } R \text{ } x \text{ } y$ 
unfolding genCngdd1-def cngdd1-def cptdd1-def genCngdd2-def cngdd2-def cptdd2-def
  eval2-embL2[symmetric]
apply (intro allI impI)
apply (erule conjE)+
apply (drule spec)
apply (erule mp conjI)+
apply (erule rel-funD[OF rel-funD[OF comp-transfer]])
apply (rule embL2-transfer)
done

lemma genCngdd2-SCons:  $\llbracket x1 = x2; \text{genCngdd2 } R \text{ } x1 \text{ } x2 \rrbracket \implies$ 
  genCngdd2 R (SCons x1 x1) (SCons x2 x2)
apply (subst I2.idem-Cl[symmetric])
apply (rule genCngdd1-genCngdd2)
apply (rule genCngdd1-SCons)
apply auto
done

lemma genCngdd2-pls:  $\llbracket \text{genCngdd2 } R \text{ } x1 \text{ } x2; \text{genCngdd2 } R \text{ } y1 \text{ } y2 \rrbracket \implies$ 
  genCngdd2 R (pls x1 y1) (pls x2 y2)
apply (subst I2.idem-Cl[symmetric])
apply (rule genCngdd1-genCngdd2)
apply (rule genCngdd1-pls)
apply auto
done

lemma genCngdd2-prd:  $\llbracket \text{genCngdd2 } R \text{ } x1 \text{ } x2; \text{genCngdd2 } R \text{ } y1 \text{ } y2 \rrbracket \implies$ 
  genCngdd2 R (prd x1 y1) (prd x2 y2)
unfolding prd-uniform algo2-def o-apply
apply (rule genCngdd2-eval2)
apply (rule rel-funD[OF K2-as- $\Sigma\Sigma 2$ -transfer])

```

```

apply simp
done

lemma genCngdd2-genCngdd3: genCngdd2 R xs ys  $\implies$  genCngdd3 R xs ys
  unfolding genCngdd2-def cngdd2-def cptdd2-def genCngdd3-def cngdd3-def cptdd3-def
eval3-embL3[symmetric]
  apply (intro allI impI)
  apply (erule conjE) +
  apply (drule spec)
  apply (erule mp conjI) +
  apply (erule rel-funD[OF rel-funD[OF comp-transfer]])
  apply (rule embL3-transfer)
done

lemma genCngdd3-SCons:  $\llbracket x1 = x2; \text{genCngdd3 } R \text{ } xs1 \text{ } xs2 \rrbracket \implies$ 
  genCngdd3 R (SCons x1 xs1) (SCons x2 xs2)
  apply (subst I3.idem-CI[symmetric])
  apply (rule genCngdd2-genCngdd3)
  apply (rule genCngdd2-SCons)
  apply auto
done

lemma genCngdd3-pls:  $\llbracket \text{genCngdd3 } R \text{ } xs1 \text{ } xs2; \text{genCngdd3 } R \text{ } ys1 \text{ } ys2 \rrbracket \implies$ 
  genCngdd3 R (pls xs1 ys1) (pls xs2 ys2)
  apply (subst I3.idem-CI[symmetric])
  apply (rule genCngdd2-genCngdd3)
  apply (rule genCngdd2-pls)
  apply auto
done

lemma genCngdd3-prd:  $\llbracket \text{genCngdd3 } R \text{ } xs1 \text{ } xs2; \text{genCngdd3 } R \text{ } ys1 \text{ } ys2 \rrbracket \implies$ 
  genCngdd3 R (prd xs1 ys1) (prd xs2 ys2)
  apply (subst I3.idem-CI[symmetric])
  apply (rule genCngdd2-genCngdd3)
  apply (rule genCngdd2-prd)
  apply auto
done

lemma genCngdd3-Exp: genCngdd3 R xs ys  $\implies$ 
  genCngdd3 R (Exp xs) (Exp ys)
  unfolding Exp-uniform algQ3-def o-apply
  apply (rule genCngdd3-eval3)
  apply (rule rel-funD[OF K3-as-ΣΣ3-transfer])
  apply simp
done

lemma genCngdd3-genCngdd4: genCngdd3 R xs ys  $\implies$  genCngdd4 R xs ys
  unfolding genCngdd3-def cngdd3-def cptdd3-def genCngdd4-def cngdd4-def cptdd4-def
eval4-embL4[symmetric]

```

```

apply (intro allI impI)
apply (erule conjE)+
apply (drule spec)
apply (erule mp conjI)+
apply (erule rel-funD[OF rel-funD[OF comp-transfer]])
apply (rule embL4-transfer)
done

lemma genCngdd4-SCons:  $\llbracket x1 = x2; \text{genCngdd4 } R \text{ } x1 \text{ } x2 \rrbracket \implies$ 
  genCngdd4 R (SCons x1 x1) (SCons x2 x2)
apply (subst I4.idem-Cl[symmetric])
apply (rule genCngdd3-genCngdd4)
apply (rule genCngdd3-SCons)
apply auto
done

lemma genCngdd4-pls:  $\llbracket \text{genCngdd4 } R \text{ } x1 \text{ } x2; \text{genCngdd4 } R \text{ } y1 \text{ } y2 \rrbracket \implies$ 
  genCngdd4 R (pls x1 y1) (pls x2 y2)
apply (subst I4.idem-Cl[symmetric])
apply (rule genCngdd3-genCngdd4)
apply (rule genCngdd3-pls)
apply auto
done

lemma genCngdd4-prd:  $\llbracket \text{genCngdd4 } R \text{ } x1 \text{ } x2; \text{genCngdd4 } R \text{ } y1 \text{ } y2 \rrbracket \implies$ 
  genCngdd4 R (prd x1 y1) (prd x2 y2)
apply (subst I4.idem-Cl[symmetric])
apply (rule genCngdd3-genCngdd4)
apply (rule genCngdd3-prd)
apply auto
done

lemma genCngdd4-Exp: genCngdd4 R xs ys  $\implies$ 
  genCngdd4 R (Exp xs) (Exp ys)
apply (subst I4.idem-Cl[symmetric])
apply (rule genCngdd3-genCngdd4)
apply (rule genCngdd3-Exp)
apply auto
done

lemma genCngdd4-sup: rel-fset (genCngdd4 R) xs ys  $\implies$ 
  genCngdd4 R (sup xs) (sup ys)
unfolding sup-uniform algo4-def o-apply
apply (rule genCngdd4-eval4)
apply (rule rel-funD[OF K4-as- $\Sigma\Sigma$ 4-transfer])
apply simp
done

lemma genCngdd4-genCngdd5: genCngdd4 R xs ys  $\implies$  genCngdd5 R xs ys

```



```

unfolding genCngdd4-def cngdd4-def cptdd4-def genCngdd5-def cngdd5-def cptdd5-def
eval5-embL5[symmetric]
apply (intro allI impI)
apply (erule conjE)+
apply (drule spec)
apply (erule mp conjI)+
apply (erule rel-funD[OF rel-funD[OF comp-transfer]])
apply (rule embL5-transfer)
done

```

```

lemma genCngdd5-SCons:  $\llbracket x1 = x2; \text{genCngdd5 } R \text{ } x1 \text{ } x2 \rrbracket \implies$ 
genCngdd5 R (SCons x1 x1) (SCons x2 x2)
apply (subst I5.idem-CI[symmetric])
apply (rule genCngdd4-genCngdd5)
apply (rule genCngdd4-SCons)
apply auto
done

```

```

lemma genCngdd5-pls:  $\llbracket \text{genCngdd5 } R \text{ } x1 \text{ } x2; \text{genCngdd5 } R \text{ } y1 \text{ } y2 \rrbracket \implies$ 
genCngdd5 R (pls x1 y1) (pls x2 y2)
apply (subst I5.idem-CI[symmetric])
apply (rule genCngdd4-genCngdd5)
apply (rule genCngdd4-pls)
apply auto
done

```

```

lemma genCngdd5-prd:  $\llbracket \text{genCngdd5 } R \text{ } x1 \text{ } x2; \text{genCngdd5 } R \text{ } y1 \text{ } y2 \rrbracket \implies$ 
genCngdd5 R (prd x1 y1) (prd x2 y2)
apply (subst I5.idem-CI[symmetric])
apply (rule genCngdd4-genCngdd5)
apply (rule genCngdd4-prd)
apply auto
done

```

```

lemma genCngdd5-Exp: genCngdd5 R xs ys  $\implies$ 
genCngdd5 R (Exp xs) (Exp ys)
apply (subst I5.idem-CI[symmetric])
apply (rule genCngdd4-genCngdd5)
apply (rule genCngdd4-Exp)
apply auto
done

```

```

lemma genCngdd5-sup: rel-fset (genCngdd5 R) xs ys  $\implies$ 
genCngdd5 R (sup xs) (sup ys)
apply (subst I5.idem-CI[symmetric])
apply (rule genCngdd4-genCngdd5)
apply (rule genCngdd4-sup)
apply (auto intro: predicate2D[OF fset.rel-mono])
done

```

```

lemma genCngdd5-prd':  $\llbracket \text{genCngdd5 } R \text{ } xs1 \text{ } xs2; \text{genCngdd5 } R \text{ } ys1 \text{ } ys2 \rrbracket \implies$ 
  genCngdd5 R (prd' xs1 ys1) (prd' xs2 ys2)
unfolding prd'-uniform alg5-def o-apply
apply (rule genCngdd5-eval5)
apply (rule rel-funD[OF K5-as-ΣΣ5-transfer])
apply simp
done

lemma stream-coinduct[case-names Eq-stream, case-conclusion Eq-stream head tail]:
  assumes  $R \text{ } s \text{ } s' \wedge s \text{ } s' \implies \text{head } s = \text{head } s' \wedge R \text{ } (\text{tail } s) \text{ } (\text{tail } s')$ 
  shows  $s = s'$ 
using assms(1) proof (rule mp[OF J.dtor-coinduct, rotated], safe)
  fix a b
  assume  $R \text{ } a \text{ } b$ 
  from assms(2)[OF this] show  $F\text{-rel } R \text{ } (\text{dtor-}J \text{ } a) \text{ } (\text{dtor-}J \text{ } b)$ 
  by (cases dtor-J a dtor-J b rule: prod.exhaust[case-product prod.exhaust])
    (auto simp: rel-pre-J-def vimage2p-def BNF-Comp.id-bnf-comp-def)
qed

lemma stream-coinduct0[case-names Eq-stream, case-conclusion Eq-stream head tail]:
  assumes  $R \text{ } s \text{ } s' \wedge s \text{ } s' \implies \text{head } s = \text{head } s' \wedge \text{genCngdd0 } R \text{ } (\text{tail } s) \text{ } (\text{tail } s')$ 
  shows  $s = s'$ 
using assms(1) proof (rule mp[OF coinductionU-genCngdd0, rotated], safe)
  fix a b
  assume  $R \text{ } a \text{ } b$ 
  from assms(2)[OF this] show  $F\text{-rel } (\text{genCngdd0 } R) \text{ } (\text{dtor-}J \text{ } a) \text{ } (\text{dtor-}J \text{ } b)$ 
  by (cases dtor-J a dtor-J b rule: prod.exhaust[case-product prod.exhaust])
    (auto simp: rel-pre-J-def vimage2p-def BNF-Comp.id-bnf-comp-def)
qed

lemma stream-coinduct1[case-names Eq-stream, case-conclusion Eq-stream head tail]:
  assumes  $R \text{ } s \text{ } s' \wedge s \text{ } s' \implies \text{head } s = \text{head } s' \wedge \text{genCngdd1 } R \text{ } (\text{tail } s) \text{ } (\text{tail } s')$ 
  shows  $s = s'$ 
using assms(1) proof (rule mp[OF coinductionU-genCngdd1, rotated], safe)
  fix a b
  assume  $R \text{ } a \text{ } b$ 
  from assms(2)[OF this] show  $F\text{-rel } (\text{genCngdd1 } R) \text{ } (\text{dtor-}J \text{ } a) \text{ } (\text{dtor-}J \text{ } b)$ 
  by (cases dtor-J a dtor-J b rule: prod.exhaust[case-product prod.exhaust])
    (auto simp: rel-pre-J-def vimage2p-def BNF-Comp.id-bnf-comp-def)
qed

lemma stream-coinduct2[case-names Eq-stream, case-conclusion Eq-stream head tail]:
  assumes  $R \text{ } s \text{ } s' \wedge s \text{ } s' \implies \text{head } s = \text{head } s' \wedge \text{genCngdd2 } R \text{ } (\text{tail } s) \text{ } (\text{tail } s')$ 

```

```

s')
  shows  $s = s'$ 
using assms(1) proof (rule mp[OF coinductionU-genCngdd2, rotated], safe)
  fix a b
  assume  $R a b$ 
  from assms(2)[OF this] show  $F\text{-rel } (genCngdd2 R) (dtor\text{-}J a) (dtor\text{-}J b)$ 
    by (cases dtor-J a dtor-J b rule: prod.exhaust[case-product prod.exhaust])
      (auto simp: rel-pre-J-def vimage2p-def BNF-Comp.id-bnf-comp-def)
qed

lemma stream-coinduct3[case-names Eq-stream, case-conclusion Eq-stream head
tail]:
  assumes  $R s s' \wedge s s'. R s s' \implies head s = head s' \wedge genCngdd3 R (tail s) (tail s')$ 
  shows  $s = s'$ 
using assms(1) proof (rule mp[OF coinductionU-genCngdd3, rotated], safe)
  fix a b
  assume  $R a b$ 
  from assms(2)[OF this] show  $F\text{-rel } (genCngdd3 R) (dtor\text{-}J a) (dtor\text{-}J b)$ 
    by (cases dtor-J a dtor-J b rule: prod.exhaust[case-product prod.exhaust])
      (auto simp: rel-pre-J-def vimage2p-def BNF-Comp.id-bnf-comp-def)
qed

lemma stream-coinduct4[case-names Eq-stream, case-conclusion Eq-stream head
tail]:
  assumes  $R s s' \wedge s s'. R s s' \implies head s = head s' \wedge genCngdd4 R (tail s) (tail s')$ 
  shows  $s = s'$ 
using assms(1) proof (rule mp[OF coinductionU-genCngdd4, rotated], safe)
  fix a b
  assume  $R a b$ 
  from assms(2)[OF this] show  $F\text{-rel } (genCngdd4 R) (dtor\text{-}J a) (dtor\text{-}J b)$ 
    by (cases dtor-J a dtor-J b rule: prod.exhaust[case-product prod.exhaust])
      (auto simp: rel-pre-J-def vimage2p-def BNF-Comp.id-bnf-comp-def)
qed

lemma stream-coinduct5[case-names Eq-stream, case-conclusion Eq-stream head
tail]:
  assumes  $R s s' \wedge s s'. R s s' \implies head s = head s' \wedge genCngdd5 R (tail s) (tail s')$ 
  shows  $s = s'$ 
using assms(1) proof (rule mp[OF coinductionU-genCngdd5, rotated], safe)
  fix a b
  assume  $R a b$ 
  from assms(2)[OF this] show  $F\text{-rel } (genCngdd5 R) (dtor\text{-}J a) (dtor\text{-}J b)$ 
    by (cases dtor-J a dtor-J b rule: prod.exhaust[case-product prod.exhaust])
      (auto simp: rel-pre-J-def vimage2p-def BNF-Comp.id-bnf-comp-def)
qed

```

8 Proofs by Coinduction Up-To Congruence

```

lemma pls-commute: pls xs ys = pls ys xs
  by (coinduction arbitrary: xs ys rule: stream-coinduct) auto

lemma prd-commute: prd xs ys = prd ys xs
proof (coinduction arbitrary: xs ys rule: stream-coinduct1)
  case Eq-stream
  then show ?case unfolding tail-prd
    by (subst pls-commute) (auto intro: genCngdd1-pls)
qed

lemma pls-assoc: pls (pls xs ys) zs = pls xs (pls ys zs)
  by (coinduction arbitrary: xs ys zs rule: stream-coinduct) auto

lemma pls-commute-assoc: pls xs (pls ys zs) = pls ys (pls xs zs)
  by (metis pls-assoc pls-commute)

lemmas pls-ac-simps = pls-assoc pls-commute pls-commute-assoc

lemma onetwo = onetwo'
  by (coinduction rule: stream-coinduct0)
    (auto simp: arg-cong[OF onetwo-code, of head] arg-cong[OF onetwo'-code, of head] J.dtor-ctor
      arg-cong[OF onetwo-code, of tail] arg-cong[OF onetwo'-code, of tail] intro: genCngdd0-SCons)

lemma prd-distribL: prd xs (pls ys zs) = pls (prd xs ys) (prd xs zs)
proof (coinduction arbitrary: xs ys zs rule: stream-coinduct1)
  case Eq-stream
  have  $\bigwedge a\ b\ c\ d. \text{pls (pls a b) (pls c d) = pls (pls a c) (pls b d)}$  by (metis pls-assoc pls-commute)
  then have ?tail by (auto intro!: genCngdd1-pls)
  then show ?case by (simp add: algebra-simps)
qed

lemma prd-distribR: prd (pls xs ys) zs = pls (prd xs zs) (prd ys zs)
proof (coinduction arbitrary: xs ys zs rule: stream-coinduct1)
  case Eq-stream
  have  $\bigwedge a\ b\ c\ d. \text{pls (pls a b) (pls c d) = pls (pls a c) (pls b d)}$  by (metis pls-assoc pls-commute)
  then have ?tail by (auto intro!: genCngdd1-pls)
  then show ?case by (simp add: algebra-simps)
qed

lemma prd-assoc: prd (prd xs ys) zs = prd xs (prd ys zs)
proof (coinduction arbitrary: xs ys zs rule: stream-coinduct1)
  case Eq-stream
  have ?tail unfolding tail-prd pls-ac-simps prd-distribL prd-distribR by (auto

```

```

intro!: genCngdd1-pls)
  then show ?case by simp
qed

lemma prd-commute-assoc: prd xs (prd ys zs) = prd ys (prd xs zs)
  by (metis prd-assoc prd-commute)

lemmas prd-ac-simps = prd-assoc prd-commute prd-commute-assoc

lemma sconst-0[simp]: same 0 = sconst 0
  by (coinduction rule: stream-coinduct0) auto

lemma pls-sconst-0L[simp]: pls (sconst 0) s = s
  by (coinduction arbitrary: s rule: stream-coinduct) auto

lemma pls-sconst-0R[simp]: pls s (sconst 0) = s
  by (coinduction arbitrary: s rule: stream-coinduct) auto

lemma scale-0[simp]: scale 0 s = sconst 0
  apply (coinduction arbitrary: s rule: stream-coinduct1)
  apply simp
  apply (subst (5) pls-sconst-0L[of sconst 0, symmetric])
  apply (rule genCngdd1-pls)
  apply auto
done

lemma scale-Suc: scale (Suc n) s = pls s (scale n s)
  by (coinduction arbitrary: s rule: stream-coinduct1) auto

lemma scale-add: scale (m + n) s = pls (scale m s) (scale n s)
  by (induct m) (auto simp: scale-Suc pls-assoc)

lemma scale-mult: scale (m * n) s = scale m (scale n s)
  by (induct m) (auto simp: scale-Suc scale-add)

lemma sup-empty: sup {} = sconst 0
  by (coinduction rule: stream-coinduct1) (auto simp: fMax-def)

lemma Exp-pls: Exp (pls xs ys) = prd (Exp xs) (Exp ys)
  by (coinduction arbitrary: xs ys rule: stream-coinduct2)
  (auto simp: exp-def power-add prd-distribR pls-commute prd-assoc prd-commute-assoc[of
Exp x for x]
  intro!: genCngdd2-pls genCngdd2-prd)

```

9 Two Examples of Fibonacci streams

definition *fibA* :: *stream* **where**

$fibA = corecUU1 (\lambda xs. GUARD1 (0, PLS1 (SCONS1 (1, CONT1 xs), CONT1 xs))) ()$

lemma *head-fibA[simp]*: $head\ fibA = 0$
unfolding *fibA-def corecUU1*
by (*simp add: map-pre-J-def BNF-Comp.id-bnf-comp-def J.dtor-ctor eval1-leaf1'*)

lemma *tail-fibA[simp]*: $tail\ fibA = pls\ (SCons\ 1\ fibA)\ fibA$
apply (*subst fibA-def*)
unfolding *corecUU1*
by (*simp add: map-pre-J-def BNF-Comp.id-bnf-comp-def J.dtor-ctor eval1-leaf1' eval1- \mathfrak{op} 1 alg Λ 1-Inr o-eq-dest[OF Abs- Σ 1-natural] o-eq-dest[OF gg1-natural] o-eq-dest[OF eval1-gg1] pls-uniform fibA-def*)

lemma *fibA-code[code]*: $fibA = SCons\ 0\ (pls\ (SCons\ 1\ fibA)\ fibA)$
by (*metis J.ctor-dtor prod.collapse head-fibA tail-fibA*)

definition *fibB :: stream where*
 $fibB = corecUU1 (\lambda xs. PLS1 (GUARD1 (0, (SCONS1 (1, CONT1 xs))), GUARD1 (0, CONT1 xs))) ()$

lemma *fibB-code[code]*: $fibB = pls\ (SCons\ 0\ (SCons\ 1\ fibB))\ (SCons\ 0\ fibB)$
apply (*subst fibB-def*)
unfolding *corecUU1*
by (*simp add: map-pre-J-def BNF-Comp.id-bnf-comp-def J.dtor-ctor eval1-leaf1' eval1- \mathfrak{op} 1 alg Λ 1-Inr o-eq-dest[OF Abs- Σ 1-natural] o-eq-dest[OF gg1-natural] o-eq-dest[OF eval1-gg1] pls-uniform fibB-def*)

lemma *fibA = fibB*
proof (*coinduction rule: stream-coinduct1*)
case *Eq-stream*
have *?head* **by** (*subst fibB-code*) (*simp add: J.dtor-ctor*)
moreover
have *?tail* **by** (*subst (2) fibB-code*) (*auto simp add: J.dtor-ctor intro: genCngdd1-pls genCngdd1-SCons*)
ultimately show *?case ..*
qed

10 Streams of Factorials

definition *facA = corecUU2* ($\lambda xs. PRD2 (GUARD2 (1, CONT2 xs), GUARD2 (1, CONT2 xs))) ()$

lemma *facA-code[code]*: $facA = prd\ (SCons\ 1\ facA)\ (SCons\ 1\ facA)$
apply (*subst facA-def*)
unfolding *corecUU2*

by (*simp* *add*: *map-pre-J-def* *BNF-Comp.id-bnf-comp-def* *J.dtor-ctor* *eval2-leaf2'*
eval2- \mathbf{op} 2 *alg Λ 2-Inr* *o-eq-dest*[*OF* *Abs- Σ 2-natural*] *o-eq-dest*[*OF* *gg2-natural*]
o-eq-dest[*OF* *eval2-gg2*] *prd-uniform* *facA-def*)

definition *facB* = *corecUU3* ($\lambda xs.$ *EXP3* (*K3.I* (*GUARD3* (*0*, *CONT3* *xs*))))
 ()

lemma *facB-code*[*code*]: *facB* = *Exp* (*SCons* *0* *facB*)
apply (*subst* *facB-def*)
unfolding *corecUU3*
by (*simp* *add*: *map-pre-J-def* *BNF-Comp.id-bnf-comp-def* *J.dtor-ctor* *eval3-leaf3'*
eval3- \mathbf{op} 3 *alg Λ 3-Inr* *o-eq-dest*[*OF* *Abs- Σ 3-natural*] *o-eq-dest*[*OF* *gg3-natural*]
o-eq-dest[*OF* *eval3-gg3*] *Exp-uniform* *facB-def*)

lemma *head-facB*[*simp*]: *head* *facB* = *1*
by (*subst* *facB-code*) (*simp* *add*: *J.dtor-ctor* *exp-def*)

lemma *tail-facB*[*simp*]: *tail* *facB* = *prd* *facB* *facB*
by (*subst* *facB-code*, *subst* *tail-Exp*) (*simp* *add*: *J.dtor-ctor* *facB-code*[*symmetric*])

lemma *facA-facB*: *SCons* *1* *facA* = *facB*
proof (*coinduction* *rule*: *stream-coinduct3*)
case *Eq-stream*
have *?head* **by** (*subst* *facA-code*) (*simp* *add*: *J.dtor-ctor* *exp-def*)
moreover
have *?tail* **by** (*subst* (*2*) *facA-code*) (*auto* *intro!*: *genCngdd3-prd* *simp*: *J.dtor-ctor*)
ultimately show *?case* ..
qed

fun *facC_{rec}* **where**
facC_{rec} (*n*, *fn*, *i*) =
 (*if* *i* = *0* *then* *GUARD0* (*fn*, *CONT0* (*n* + *1*, *1*, *n* + *1*)) *else* *facC_{rec}* (*n*, *fn*
 * *i*, *i* - *1*))

definition *facC* = *corecUU0* *facC_{rec}* (*1*, *1*, *1*)

lemma *factsD_{rec}-code*:
corecUU0 *facC_{rec}* (*n*, *fn*, *i*) =
 (*if* *i* = *0* *then* *SCons* *fn* (*corecUU0* *facC_{rec}* (*n* + *1*, *1*, *n* + *1*))
else *corecUU0* *facC_{rec}* (*n*, *fn* * *i*, *i* - *1*))
by (*subst* *corecUU0*, *subst* *facC_{rec}.simps*)
 (*simp* *del*: *facC_{rec}.simps* *add*: *map-pre-J-def* *BNF-Comp.id-bnf-comp-def* *eval0-leaf0'*
corecUU0)

definition *fromN* = *dtor-corec-J* ($\lambda n.$ (*n*, *Inr* (*Suc* *n*)))

lemma *head-fromN*[*simp*]: *head* (*fromN* *n*) = *n*
unfolding *fromN-def* *J.dtor-corec* *map-pre-J-def* *BNF-Comp.id-bnf-comp-def* **by**
simp

lemma *tail-fromN[simp]*: $\text{tail } (\text{fromN } n) = \text{fromN } (\text{Suc } n)$
unfolding *fromN-def J.dtor-corec map-pre-J-def BNF-Comp.id-bnf-comp-def* **by** *simp*

abbreviation *facD* $n \equiv \text{smap } \text{fact } (\text{fromN } n)$

primrec *prds* **where**
 $\text{prds } 0 \ s = s$
 $|\ \text{prds } (\text{Suc } n) \ s = \text{prd } s \ (\text{prds } n \ s)$

lemma *head-prds[simp]*: $\text{head } (\text{prds } n \ s) = \text{head } s \ ^\wedge (\text{Suc } n)$
by *(induct n) auto*

lemma *tail-prds-fac[simp]*: $\text{tail } (\text{prds } n \ \text{facD } n) = \text{scale } (\text{Suc } n) (\text{prds } (\text{Suc } n) \ \text{facD } n)$
by *(induct n) (auto simp: scale-Suc, auto simp: prd-distribL pls-ac-simps prd-ac-simps)*

lemma *facD-facD*: $\text{facD } n = \text{scale } (\text{fact } n) (\text{prds } n \ \text{facD } n)$
proof *(coinduction arbitrary: n rule: stream-coinduct3)*
case *Eq-stream*
have *?head* **by** *(subst facD-code) (simp add: J.dtor-ctor exp-def)*
moreover
have *?tail* **by** *(subst (2) facD-code) (auto simp add: J.dtor-ctor facD-code[symmetric] scale-mult[symmetric] trans[OF mult.commute fact-Suc[symmetric]] simp del: mult-Suc-right mult-Suc fact-Suc prds.simps)*
ultimately show *?case ..*
qed

corollary *facA = facD 1*
unfolding *facD-facD facA-facD[symmetric]* **by** *(subst facA-code) (simp add: scale-Suc)*

corollary *facB = facD 0*
unfolding *facD-facD* **by** *(simp add: scale-Suc)*

primrec *ffac* **where**
 $\text{ffac } fn \ 0 = fn$
 $|\ \text{ffac } fn \ (\text{Suc } i) = \text{ffac } (fn * \text{Suc } i) \ i$

lemma *ffac-fact*: $\text{ffac } m \ n = m * \text{fact } n$
by *(induct n arbitrary: m) (auto simp: algebra-simps)*

lemma *ffac-fact-Suc*: $\text{ffac } (\text{Suc } n) \ n = \text{fact } (\text{Suc } n)$
unfolding *ffac-fact fact-Suc ..*

lemma *factsD_{rec}-facD*: $\text{corecUU0 } \text{factsC}_{\text{rec}} \ (n, fn, i) = \text{SCons } (\text{ffac } fn \ i) \ (\text{factsD } (n + 1))$
proof *(coinduction arbitrary: n fn i rule: stream-coinduct)*
case *Eq-stream*


```

have ?head
proof (induct i arbitrary: fn)
  case 0 then show ?case by (subst factsDrec-code) (simp add: J.dtor-ctor)
next
  case (Suc i) then show ?case by (subst factsDrec-code) simp
qed
moreover have ?tail
proof (induct i arbitrary: fn)
  case 0
  have factsD (Suc n) = SCons (ffac (Suc n) n) (factsD (Suc (Suc n)))
  by (coinduction rule: stream-coinduct0) (auto simp: J.dtor-ctor ffac-fact-Suc)
  then show ?case by (subst factsDrec-code) (force simp: J.dtor-ctor)
next
  case (Suc i)
  then show ?case by (subst factsDrec-code) simp
qed
ultimately show ?case by blast
qed

lemma factsC-factsD: factsC = factsD 1
  unfolding factsC-def factsDrec-factsD by (subst (2) smap-code) auto

```

11 Mixed Recursion-Corecursion

```

function primesrec :: (nat * nat) ⇒ (stream + nat * nat) ΣΣ0 F ΣΣ0 where
  primesrec (m, n) =
    (if (m = 0 ∧ n > 1) ∨ coprime m n then GUARD0 (n, CONT0 (m * n, Suc
n))
    else (primesrec (m, Suc n)))
by pat-completeness auto
termination
  apply (relation measure (λ(m, n).
    if n = 0 then 1 else if coprime m n then 0 else m - n mod m))
  apply (auto simp: mod-Suc intro: Suc-lessI)
  apply (metis One-nat-def coprime-Suc-nat gcd-nat.commute gcd-red-nat)
  apply (metis diff-less-mono2 lessI mod-less-divisor)
done

```

```

definition primes :: nat ⇒ nat ⇒ stream where
  primes = curry (corecUU0 primesrec)

```

```

lemma primes-code:
  primes m n =
    (if (m = 0 ∧ n > 1) ∨ coprime m n then SCons n (primes (m * n) (Suc n))
    else primes m (Suc n))
  unfolding primes-def curry-def
  by (subst corecUU0, subst primesrec.simps)

```

(simp del: primes_{rec}.simps add: map-pre-J-def BNF-Comp.id-bnf-comp-def
eval0-leaf0' corecUU0)

lemma primes: primes 1 2 = SCons 2 (primes 2 3)
by (subst primes-code) auto

fun catalan_{rec} :: nat ⇒ (stream + nat) ΣΣ1 F ΣΣ1 **where**
catalan_{rec} n =
(if n > 0 then PLS1 (catalan_{rec} (n - 1), GUARD1 (0, CONT1 (n+1))) else
GUARD1 (1, CONT1 1))

definition catalan :: nat ⇒ stream **where**
catalan = corecUU1 catalan_{rec}

lemma catalan-code:
catalan n =
(if n > 0 then pls (catalan (n - 1)) (SCons 0 (catalan (n + 1)))
else SCons 1 (catalan 1))
unfolding catalan-def
by (subst corecUU1, subst catalan_{rec}.simps)
(simp del: catalan_{rec}.simps add: map-pre-J-def BNF-Comp.id-bnf-comp-def
eval1-**op**1 eval1-leaf1' algΛ1-Inr o-eq-dest[OF Abs-Σ1-natural] corecUU1
pls-uniform)