Tree Examples

Jasmin Christian Blanchette   Andrei Popescu
Dmitriy Traytel

January 30, 2015

Contents

1 Sum

definition *pls* :: *tree* ⇒ *tree* ⇒ *tree* **where**
pls xs ys = *dtor-corec-J* (λ(*xs*, *ys*). (*val xs* + *val ys*, *map Inr* (*zip* (*sub xs*) (*sub ys*)))) (*xs*, *ys*)

lemma *val-pls*[*simp*]: *val* (*pls t u*) = *val t* + *val u*
unfolding *pls-def J.dtor-corec map-pre-J-def BNF-Comp.id-bnf-comp-def* **by** *simp*

lemma *sub-pls*[*simp*]: *sub* (*pls t u*) = *map* (*split pls*) (*zip* (*sub t*) (*sub u*))
unfolding *pls-def*[*abs-def*] *J.dtor-corec map-pre-J-def BNF-Comp.id-bnf-comp-def* **by** *simp*

lemma *pls-code*[*code*]: *pls t u* = *Node* (*val t* + *val u*) (*map* (*split pls*) (*zip* (*sub t*) (*sub u*)))
by (*metis J.ctor-dtor prod.collapse val-pls sub-pls*)

lemma *pls-uniform*: *pls t u* = *alg*ϱ*1* (*t*, *u*)
unfolding *pls-def*
apply (*rule fun-cong*[*OF sym*[*OF J.dtor-corec-unique*]])
unfolding *alg*ϱ*1*
by (*simp add*: *map-pre-J-def BNF-Comp.id-bnf-comp-def fun-eq-iff convol-def* ϱ*1-def alg*ϱ*1-def*)

2 Shuffle product

definition *prd* :: *tree* ⇒ *tree* ⇒ *tree* **where**
prd t u = *corecUU1* (λ(*t*, *u*). *GUARD1* (*val t* ∗ *val u*,
map (λ(*t*′, *u*′). *PLS1* (*CONT1* (*t*, *u*′), *CONT1* (*t*′, *u*))) (*zip* (*sub t*) (*sub u*))))
(*t*, *u*)

lemma *val-prd*[*simp*]: *val* (*prd t u*) = *val t* ∗ *val u*
unfolding *prd-def corecUU1*
by (*simp add*: *map-pre-J-def BNF-Comp.id-bnf-comp-def J.dtor-ctor eval1-leaf1*′)

lemma *sub-prd*[*simp*]:
sub (*prd t u*) = *map* (λ(*t*′, *u*′). *pls* (*prd t u*′) (*prd t*′ *u*)) (*zip* (*sub t*) (*sub u*))
apply (*subst prd-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 alg*Λ*1-Inr o-eq-dest*[*OF Abs-*Σ*1-natural*] *pls-uniform prd-def split-beta*)

lemma *prd-code*[*code*]: *prd t u* =
Node (*val t* ∗ *val u*) (*map* (λ(*t*′, *u*′). *pls* (*prd t u*′) (*prd t*′ *u*)) (*zip* (*sub t*) (*sub*
u)))
by (*metis J.ctor-dtor prod.collapse val-prd sub-prd*)

lemma *prd-uniform*: *prd t u* = *alg*ϱ*2* (*t*, *u*)
unfolding *prd-def*
apply (*rule fun-cong*[*OF sym*[*OF corecUU1-unique*]])
apply (*rule iffD1*[*OF dtor-J-o-inj*])
unfolding *alg*ϱ*2*
apply (*simp add*: *map-pre-J-def BNF-Comp.id-bnf-comp-def fun-eq-iff J.dtor-ctor*
ϱ*2-def Let-def convol-def eval2-*𝔬𝔭*2 eval1-*𝔬𝔭*1 eval1-leaf1*′
o-eq-dest[*OF Abs-*Σ*1-natural*] *o-eq-dest*[*OF Abs-*Σ*2-natural*] *alg*Λ*2-Inl alg*ϱ*2-def*
split-beta)
done

3 Coinduction Up-To Congruence

lemma *Node-uniform*: *Node x ts* = *eval0* (*gg0* (*x*, *map leaf0 ts*))
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-Node*: ⟦*x1* = *x2*; *list-all2* (*genCngdd0 R*) *ts1 ts2*⟧ ⟹
genCngdd0 R (*Node x1 ts1*) (*Node x2 ts2*)
unfolding *Node-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 rel-funD*[*OF map-transfer*], *rotated*])

apply (*rule leaf0-transfer*)
apply *assumption*
done

lemma *genCngdd0-genCngdd1*: *genCngdd0 R xs ys* $\Longrightarrow$ *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-Node*: $[\![$*x1* = *x2*; *list-all2* (*genCngdd1 R*) *ts1 ts2*$]\!]$ $\Longrightarrow$
genCngdd1 R (*Node x1 ts1*) (*Node x2 ts2*)
apply (*subst I1.idem-Cl*[*symmetric*])
apply (*rule genCngdd0-genCngdd1*)
apply (*rule genCngdd0-Node*)
apply (*auto intro*: *predicate2D*[*OF list.rel-mono*])
done

lemma *genCngdd1-pls*: $[\![$*genCngdd1 R xs1 xs2*; *genCngdd1 R ys1 ys2*$]\!]$ $\Longrightarrow$
genCngdd1 R (*pls xs1 ys1*) (*pls xs2 ys2*)
unfolding *pls-uniform alg*ϱ*1-def o-apply*
apply (*rule genCngdd1-eval1*)
apply (*rule rel-funD*[*OF K1-as-*$\Sigma\Sigma$*1-transfer*])
apply *simp*
done

lemma *genCngdd1-genCngdd2*: *genCngdd1 R xs ys* $\Longrightarrow$ *genCngdd2 R xs ys*
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-Node*: $[\![$*x1* = *x2*; *list-all2* (*genCngdd2 R*) *ts1 ts2*$]\!]$ $\Longrightarrow$
genCngdd2 R (*Node x1 ts1*) (*Node x2 ts2*)
apply (*subst I2.idem-Cl*[*symmetric*])
apply (*rule genCngdd1-genCngdd2*)
apply (*rule genCngdd1-Node*)
apply (*auto intro*: *predicate2D*[*OF list.rel-mono*])
done

lemma *genCngdd2-pls*: ⟦*genCngdd2 R xs1 xs2*; *genCngdd2 R ys1 ys2*⟧ $\Longrightarrow$
genCngdd2 R (*pls xs1 ys1*) (*pls xs2 ys2*)
apply (*subst I2.idem-Cl*[*symmetric*])
apply (*rule genCngdd1-genCngdd2*)
apply (*rule genCngdd1-pls*)
apply *auto*
done

lemma *genCngdd2-prd*: ⟦*genCngdd2 R xs1 xs2*; *genCngdd2 R ys1 ys2*⟧ $\Longrightarrow$
genCngdd2 R (*prd xs1 ys1*) (*prd xs2 ys2*)
unfolding *prd-uniform alg*ϱ*2-def o-apply*
apply (*rule genCngdd2-eval2*)
apply (*rule rel-funD*[*OF K2-as-*$\Sigma\Sigma$*2-transfer*])
apply *simp*
done

lemma *tree-coinduct*[*case-names Eq-tree*, *case-conclusion Eq-tree val sub*]:
assumes $R\ s\ s'$ $\bigwedge s\ s'.\ R\ s\ s' \Longrightarrow val\ s = val\ s' \wedge$ *list-all2 R* (*sub s*) (*sub s'*)
shows $s = s'$
using *assms*(*1*) **proof** (*rule mp*[*OF J.dtor-coinduct*, *rotated*], *safe*)
fix *a b*
assume *R a b*
from *assms*(*2*)[*OF this*] **show** *F-rel R* (*dtor-J a*) (*dtor-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 *tree-coinduct0*[*case-names Eq-tree*, *case-conclusion Eq-tree val sub*]:
assumes $R\ s\ s'$ $\bigwedge s\ s'.\ R\ s\ s' \Longrightarrow val\ s = val\ s' \wedge$ *list-all2* (*genCngdd0 R*) (*sub*
s) (*sub s'*)
shows $s = s'$
using *assms*(*1*) **proof** (*rule mp*[*OF coinductionU-genCngdd0*, *rotated*], *safe*)
fix *a b*
assume *R a b*
from *assms*(*2*)[*OF this*] **show** *F-rel* (*genCngdd0 R*) (*dtor-J a*) (*dtor-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 *tree-coinduct1*[*case-names Eq-tree*, *case-conclusion Eq-tree val sub*]:
assumes $R\ s\ s'$ $\bigwedge s\ s'.\ R\ s\ s' \Longrightarrow val\ s = val\ s' \wedge$ *list-all2* (*genCngdd1 R*) (*sub*
s) (*sub s'*)
shows $s = s'$
using *assms*(*1*) **proof** (*rule mp*[*OF coinductionU-genCngdd1*, *rotated*], *safe*)
fix *a b*
assume *R a b*
from *assms*(*2*)[*OF this*] **show** *F-rel* (*genCngdd1 R*) (*dtor-J a*) (*dtor-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 *tree-coinduct2*[*case-names Eq-tree*, *case-conclusion Eq-tree val sub*]:
assumes $R\ s\ s'$ $\bigwedge s\ s'.\ R\ s\ s' \Longrightarrow val\ s = val\ s' \wedge list\text{-}all2\ (genCngdd2\ R)\ (sub\ s)\ (sub\ s')$
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-rel* (*genCngdd2 R*) (*dtor-J a*) (*dtor-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

4 Proofs by Coinduction Up-To Congruence

lemma *pls-assoc*: *pls* (*pls t u*) *zs* = *pls t* (*pls u zs*)
by (*coinduction arbitrary*: *t u zs rule*: *tree-coinduct*) (*force simp*: *list-all2-iff in-set-zip*)

lemma *pls-commute*: *pls t u* = *pls u t*
by (*coinduction arbitrary*: *t u rule*: *tree-coinduct*) (*force simp*: *list-all2-iff in-set-zip*)

lemma *pls-commute-assoc*: *pls t* (*pls u zs*) = *pls u* (*pls t zs*)
by (*metis pls-assoc pls-commute*)

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

lemma *prd-commute*: *prd t u* = *prd u t*
proof (*coinduction arbitrary*: *t u rule*: *tree-coinduct1*)
case *Eq-tree*
have *?sub* **unfolding** *sub-prd*
by (*subst pls-commute*) (*fastforce simp*: *list-all2-iff in-set-zip intro*!: *genCngdd1-pls*)
then show *?case* **by** *simp*
qed

lemma *prd-distribL*: *prd xs* (*pls ys zs*) = *pls* (*prd xs ys*) (*prd xs zs*)
proof (*coinduction arbitrary*: *xs ys zs rule*: *tree-coinduct1*)
case *Eq-tree*
have $\bigwedge a\ b\ c\ d.$ *pls* (*pls a b*) (*pls c d*) = *pls* (*pls a c*) (*pls b d*) **by** (*metis pls-assoc pls-commute*)
then have *?sub* **by** (*fastforce simp*: *list-all2-iff in-set-zip 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*: *tree-coinduct1*)
case *Eq-tree*

have $\bigwedge$*a b c d. pls* (*pls a b*) (*pls c d*) = *pls* (*pls a c*) (*pls b d*) **by** (*metis pls-assoc pls-commute*)
  then have *?sub* **by** (*fastforce simp*: *list-all2-iff in-set-zip 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*: *tree-coinduct1*)
  case *Eq-tree*
  have *?sub* **unfolding** *sub-prd zip-map1 zip-map2 list.map-comp*
    by (*fastforce simp*: *list-all2-iff in-set-zip pls-ac-simps prd-distribL prd-distribR 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*