Contents

1 Sequences as Lists

theory *Sequences*
imports *Main*
begin

locale *Sequences*
begin

We reverse the order of application of *op #* and *op @* because we think that it is easier to think of sequences as growing to the right.

no-notation *Cons* (**infixr** # *65*)
abbreviation *Append* (**infixl** # *65*)
  where *Append xs x* ≡ *Cons x xs*
no-notation *append* (**infixr** @ *65*)
abbreviation *Concat* (**infixl** @ *65*)
  where *Concat xs ys* ≡ *append ys xs*

end

end

2 I/O Automata with Finite-Trace Semantics

theory *IOA*
imports *Main Sequences*
begin

This theory is inspired by the IOA theory of Olaf Mller

locale *IOA* = *Sequences*

record *'a signature* =
  inputs::*'a set*
  outputs::*'a set*
  internals::*'a set*

context *IOA*
begin

2.1 Signatures

definition *actions* :: *'a signature* ⇒ *'a set* **where**
  actions asig ≡ *inputs asig* ∪ *outputs asig* ∪ *internals asig*

definition *externals* :: *'a signature* ⇒ *'a set* **where**
  externals asig ≡ *inputs asig* ∪ *outputs asig*

definition *locals* :: *'a signature* ⇒ *'a set* **where**
  locals asig ≡ *internals asig* ∪ *outputs asig*

definition *is-asig* :: *'a signature* ⇒ *bool* **where**
is-asig triple ≡
inputs triple ∩ *outputs triple* = {} ∧
outputs triple ∩ *internals triple* = {} ∧
inputs triple ∩ *internals triple* = {}

lemma *internal-inter-external*:
assumes *is-asig sig*
shows *internals sig* ∩ *externals sig* = {}
⟨*proof*⟩

definition *hide-asig* **where**
hide-asig asig actns ≡
(|*inputs* = *inputs asig* − *actns*, *outputs* = *outputs asig* − *actns*,
internals = *internals asig* ∪*actns*|)

end

2.2 I/O Automata

type-synonym
('*s*,'*a*) *transition* = '*s* × '*a* × '*s*

record ('*s*,'*a*) *ioa* =
asig::'*a signature*
start::'*s set*
trans::('*s*,'*a*)*transition set*

context *IOA*
begin

abbreviation *act A* ≡ *actions* (*asig A*)
abbreviation *ext A* ≡ *externals* (*asig A*)
abbreviation *int* **where** *int A* ≡ *internals* (*asig A*)
abbreviation *inp A* ≡ *inputs* (*asig A*)
abbreviation *out A* ≡ *outputs* (*asig A*)
abbreviation *local A* ≡ *locals* (*asig A*)

definition *is-ioa*::('*s*,'*a*) *ioa* ⇒ *bool* **where**
is-ioa A ≡ *is-asig* (*asig A*)
∧ (∀ *triple* ∈ *trans A* . (*fst o snd*) *triple* ∈ *act A*)

definition *hide* **where**
hide A actns ≡ *A*(|*asig* := *hide-asig* (*asig A*) *actns*|)

definition *is-trans*::'*s* ⇒ '*a* ⇒ ('*s*,'*a*)*ioa* ⇒ '*s* ⇒ *bool* **where**
is-trans s1 a A s2 ≡ (*s1*,*a*,*s2*) ∈ *trans A*

notation
is-trans (- −-−-⟶ - [*81*,*81*,*81*,*81*] *100*)

definition *rename-set* **where**
rename-set A ren ≡ {*b*. ∃ *x* ∈ *A* . *ren b* = *Some x*}

definition *rename* **where**
rename A ren ≡
(⦇*asig* = (⦇*inputs* = *rename-set* (*inp A*) *ren*,
outputs = *rename-set* (*out A*) *ren*,
internals = *rename-set* (*int A*) *ren*⦈),
start = *start A*,
trans = {*tr*. ∃ *x* . *ren* (*fst* (*snd tr*)) = *Some x* ∧ (*fst tr*) −*x*−*A*⟶ (*snd* (*snd tr*))}⦈

Reachable states and invariants

inductive
reachable :: ('*s*,'*a*) *ioa* ⇒ '*s* ⇒ *bool*
for *A* :: ('*s*,'*a*) *ioa*
where
reachable-0: *s* ∈ *start A* ⟹ *reachable A s*
| *reachable-n*: ⟦ *reachable A s*; *s* −*a*−*A*⟶ *t* ⟧ ⟹ *reachable A t*

definition *invariant* **where**
invariant A P ≡ (∀ *s* . *reachable A s* ⟶ *P*(*s*))

theorem *invariantI*:
fixes *A P*
assumes ⋀ *s* . *s* ∈ *start A* ⟹ *P s*
and ⋀ *s t a* . ⟦*reachable A s*; *P s*; *s* −*a*−*A*⟶ *t*⟧ ⟹ *P t*
shows *invariant A P*
⟨*proof*⟩

end

2.3 Composition of Families of I/O Automata

record ('*id*, '*a*) *family* =
ids :: '*id set*
memb :: '*id* ⇒ '*a*

context *IOA*

begin

definition *is-ioa-fam* **where**
is-ioa-fam fam $\equiv \forall\ i \in ids\ fam\ .\ is\text{-}ioa\ (memb\ fam\ i)$

definition *compatible2* **where**
compatible2 A B $\equiv$
$out\ A \cap out\ B = \{\} \wedge$
$int\ A \cap act\ B = \{\} \wedge$
$int\ B \cap act\ A = \{\}$

definition *compatible*::('*id*, ('*s*,'*a*)*ioa*) *family* $\Rightarrow$ *bool* **where**
compatible fam $\equiv$ *finite* (*ids fam*) $\wedge$
$(\forall\ i \in ids\ fam\ .\ \forall\ j \in ids\ fam\ .\ i \neq j \longrightarrow$
compatible2 (*memb fam i*) (*memb fam j*))

definition *asig-comp2* **where**
asig-comp2 A B $\equiv$
$(\!|inputs = (inputs\ A \cup inputs\ B) - (outputs\ A \cup outputs\ B),$
$outputs = outputs\ A \cup outputs\ B,$
$internals = internals\ A \cup internals\ B|\!)$

definition *asig-comp*::('*id*, ('*s*,'*a*)*ioa*) *family* $\Rightarrow$ '*a signature* **where**
asig-comp fam $\equiv$
$(\!|\ inputs = \bigcup i{\in}(ids\ fam).\ inp\ (memb\ fam\ i)$
$- (\bigcup i{\in}(ids\ fam).\ out\ (memb\ fam\ i)),$
$outputs = \bigcup i{\in}(ids\ fam).\ out\ (memb\ fam\ i),$
$internals = \bigcup i{\in}(ids\ fam).\ int\ (memb\ fam\ i)\ |\!)$

definition *par2* (**infixr** $\|$ *10*) **where**
$A \parallel B \equiv$
$(\!|asig = asig\text{-}comp2\ (asig\ A)\ (asig\ B),$
$start = \{pr.\ fst\ pr \in start\ A \wedge snd\ pr \in start\ B\},$
$trans = \{tr.$
$let\ s = fst\ tr;\ a = fst\ (snd\ tr);\ t = snd\ (snd\ tr)$
$in\ (a \in act\ A \vee a \in act\ B)$
$\wedge\ (if\ a \in act\ A$
$then\ fst\ s\ -a-A\longrightarrow fst\ t$
$else\ fst\ s = fst\ t)$
$\wedge\ (if\ a \in act\ B$
$then\ snd\ s\ -a-B\longrightarrow snd\ t$
$else\ snd\ s = snd\ t)\ \}|\!)$

definition *par*::('*id*, ('*s*,'*a*)*ioa*) *family* $\Rightarrow$ ('*id* $\Rightarrow$ '*s*,'*a*)*ioa* **where**

par fam ≡ *let ids* = *ids fam*; *memb* = *memb fam in*
(| *asig* = *asig-comp fam*,
start = {*s* . ∀ *i*∈*ids* . *s i* ∈ *start* (*memb i*)},
trans = { (*s*, *a*, *s'*) .
(∃ *i*∈*ids* . *a* ∈ *act* (*memb i*))
∧ (∀ *i*∈*ids* .
if a ∈ *act* (*memb i*)
then s i −*a*−(*memb i*)⟶ *s' i*
else s i = (*s' i*)) } |)

lemmas *asig-simps* = *hide-asig-def is-asig-def locals-def externals-def actions-def hide-def compatible-def asig-comp-def*
lemmas *ioa-simps* = *rename-def rename-set-def is-trans-def is-ioa-def par-def*

end

2.4 Executions and Traces

type-synonym
('*s*,'*a*)*pairs* = ('*a*×'*s*) *list*
type-synonym
('*s*,'*a*)*execution* = '*s* × ('*s*,'*a*)*pairs*
type-synonym
'*a trace* = '*a list*

record ('*s*,'*a*)*execution-module* =
execs::('*s*,'*a*)*execution set*
asig::'*a signature*

record '*a trace-module* =
traces::'*a trace set*
asig::'*a signature*

context *IOA*
begin

fun *is-exec-frag-of*::('*s*,'*a*)*ioa* ⇒ ('*s*,'*a*)*execution* ⇒ *bool* **where**
is-exec-frag-of A (*s*,(*ps*#*p'*)#*p*) =
(*snd p'* −*fst p*−*A*⟶ *snd p* ∧ *is-exec-frag-of A* (*s*, (*ps*#*p'*)))
| *is-exec-frag-of A* (*s*, [*p*]) = *s* −*fst p*−*A*⟶ *snd p*
| *is-exec-frag-of A* (*s*, []) = *True*

definition *is-exec-of*::('*s*,'*a*)*ioa* ⇒ ('*s*,'*a*)*execution* ⇒ *bool* **where**
is-exec-of A e ≡ *fst e* ∈ *start A* ∧ *is-exec-frag-of A e*

definition *filter-act* **where**
filter-act ≡ *map fst*

definition *schedule* **where**
schedule ≡ *filter-act o snd*

definition *trace* **where**
trace sig ≡ *filter* (λ *a* . *a* ∈ *externals sig*) *o schedule*

definition *is-schedule-of* **where**
is-schedule-of A sch ≡
(∃ *e* . *is-exec-of A e* ∧ *sch* = *filter-act* (*snd e*))

definition *is-trace-of* **where**
is-trace-of A tr ≡
(∃ *sch* . *is-schedule-of A sch* ∧ *tr* = *filter* (λ *a*. *a* ∈ *ext A*) *sch*)

definition *traces* **where**
traces A ≡ {*tr*. *is-trace-of A tr*}

lemma *traces-alt*:
shows *traces A* = {*tr* . ∃ *e* . *is-exec-of A e*
∧ *tr* = *trace* (*ioa.asig A*) *e*}
⟨*proof*⟩

lemmas *trace-simps* = *traces-def is-trace-of-def is-schedule-of-def filter-act-def is-exec-of-def trace-def schedule-def*

definition *proj-trace*::'*a trace* ⇒ ('*a signature*) ⇒ '*a trace* (**infixr** | *12*) **where**
proj-trace t sig ≡ *filter* (λ *a* . *a* ∈ *actions sig*) *t*

definition *ioa-implements* :: ('*s1*,'*a*)*ioa* ⇒ ('*s2*,'*a*)*ioa* ⇒ *bool* (**infixr** =<| *12*)
where
A =<| *B* ≡ *inp A* = *inp B* ∧ *out A* = *out B* ∧ *traces A* ⊆ *traces B*

2.5 Operations on Executions

definition *cons-exec* **where**
cons-exec e p ≡ (*fst e*, (*snd e*)#*p*)

definition *append-exec* **where**
append-exec e e' ≡ (*fst e*, (*snd e*)@(*snd e'*))

fun *last-state* **where**
  last-state $(s,[]) = s$
| *last-state* $(s,ps\#p) = snd\ p$

lemma *last-state-reachable*:
  fixes $A\ e$
  assumes *is-exec-of* $A\ e$
  shows *reachable* A (*last-state* e) ⟨*proof*⟩

lemma *trans-from-last-state*:
  assumes *is-exec-frag-of* $A\ e$ **and** (*last-state* e)$-a-A\longrightarrow s'$
  shows *is-exec-frag-of* A (*cons-exec* $e\ \ (a,s')$)
    ⟨*proof*⟩

lemma *exec-frag-prefix*:
  fixes $A\ p\ ps$
  assumes *is-exec-frag-of* A (*cons-exec* $e\ p$)
  shows *is-exec-frag-of* $A\ e$
    ⟨*proof*⟩

lemma *trace-same-ext*:
  fixes $A\ B\ e$
  assumes *ext* A = *ext* B
  shows *trace* (*ioa.asig* A) e = *trace* (*ioa.asig* B) e
  ⟨*proof*⟩

lemma *trace-append-is-append-trace*:
  fixes $e\ e'$ *sig*
  shows *trace sig* (*append-exec* $e'\ e$) = *trace sig* e' @ *trace sig* e
  ⟨*proof*⟩

lemma *append-exec-frags-is-exec-frag*:
  fixes $e\ e'\ A\ as$
  assumes *is-exec-frag-of* $A\ e$ **and** *last-state* e = *fst* e'
  and *is-exec-frag-of* $A\ e'$
  shows *is-exec-frag-of* A (*append-exec* $e\ e'$)
⟨*proof*⟩

lemma *last-state-of-append*:
  fixes $e\ e'$
  assumes *fst* e' = *last-state* e
  shows *last-state* (*append-exec* $e\ e'$) = *last-state* e'
  ⟨*proof*⟩

end

end

3 Definition and Soundness of Refinement Mappings, Forward Simulations and Backward Simulations

theory *Simulations*
imports *IOA*
begin

context *IOA*
begin

definition *refines* **where**
refines e s a t A f ≡ *fst e* = *f s* ∧ *last-state e* = *f t* ∧ *is-exec-frag-of A e*
∧ (*let tr* = *trace* (*ioa.asig A*) *e in*
if a ∈ *ext A then tr* = [*a*] *else tr* = [])
definition
is-ref-map :: (*'s1* ⇒ *'s2*) ⇒ (*'s1*,*'a*)*ioa* ⇒ (*'s2*,*'a*)*ioa* ⇒ *bool* **where**
is-ref-map f B A ≡
(∀ *s* ∈ *start B* . *f s* ∈ *start A*) ∧ (∀ *s t a*. *reachable B s* ∧ *s* −*a*−*B*⟶ *t*
⟶ (∃ *e* . *refines e s a t A f*))

definition
is-forward-sim :: (*'s1* ⇒ (*'s2 set*)) ⇒ (*'s1*,*'a*)*ioa* ⇒ (*'s2*,*'a*)*ioa* ⇒ *bool* **where**
is-forward-sim f B A ≡
(∀ *s* ∈ *start B* . *f s* ∩ *start A* ≠ {})
∧ (∀ *s s' t a*. *s'* ∈ *f s* ∧ *s* −*a*−*B*⟶ *t* ∧ *reachable B s*
⟶ (∃ *e* . *fst e* = *s'* ∧ *last-state e* ∈ *f t* ∧ *is-exec-frag-of A e*
∧ (*let tr* = *trace* (*ioa.asig A*) *e in*
if a ∈ *ext A then tr* = [*a*] *else tr* = [])))

definition
is-backward-sim :: (*'s1* ⇒ (*'s2 set*)) ⇒ (*'s1*,*'a*)*ioa* ⇒ (*'s2*,*'a*)*ioa* ⇒ *bool* **where**
is-backward-sim f B A ≡
(∀ *s* . *f s* ≠ {}) (∗ *Restricting this to reachable states would suffice* ∗)
∧ (∀ *s* ∈ *start B* . *f s* ⊆ *start A*)
∧ (∀ *s t a t'*. *t'* ∈ *f t* ∧ *s* −*a*−*B*⟶ *t* ∧ *reachable B s*
⟶ (∃ *e* . *fst e* ∈ *f s* ∧ *last-state e* = *t'* ∧ *is-exec-frag-of A e*
∧ (*let tr* = *trace* (*ioa.asig A*) *e in*
if a ∈ *ext A then tr* = [*a*] *else tr* = [])))

3.1 A series of lemmas that will be useful in the soundness proofs

lemma *step-eq-traces*:
fixes *e-B′ A e e-A′ a t*
defines *e-A* ≡ *append-exec e-A′ e* **and** *e-B* ≡ *cons-exec e-B′* (*a*,*t*)
and *tr* ≡ *trace* (*ioa.asig A*) *e*
assumes *1*:*trace* (*ioa.asig A*) *e-A′* = *trace* (*ioa.asig A*) *e-B′*
and *2*:*if a* ∈ *ext A then tr* = [*a*] *else tr* = []
shows *trace* (*ioa.asig A*) *e-A* = *trace* (*ioa.asig A*) *e-B*
⟨*proof*⟩

lemma *exec-inc-imp-trace-inc*:
fixes *A B*
assumes *ext B* = *ext A*
and ⋀ *e-B* . *is-exec-of B e-B*
⟹ ∃ *e-A* . *is-exec-of A e-A* ∧ *trace* (*ioa.asig A*) *e-A* = *trace* (*ioa.asig A*) *e-B*
shows *traces B* ⊆ *traces A*
⟨*proof*⟩

3.2 Soundness of Refinement Mappings

lemma *ref-map-execs*:
fixes *A*::(*'sA*,*'a*)*ioa* **and** *B*::(*'sB*,*'a*)*ioa* **and** *f*::*'sB* ⇒ *'sA* **and** *e-B*
assumes *is-ref-map f B A* **and** *is-exec-of B e-B*
shows ∃ *e-A* . *is-exec-of A e-A*
∧ *trace* (*ioa.asig A*) *e-A* = *trace* (*ioa.asig A*) *e-B*
⟨*proof*⟩

theorem *ref-map-soundness*:
fixes *A*::(*'sA*,*'a*)*ioa* **and** *B*::(*'sB*,*'a*)*ioa* **and** *f*::*'sB* ⇒ *'sA*
assumes *is-ref-map f B A* **and** *ext A* = *ext B*
shows *traces B* ⊆ *traces A*
⟨*proof*⟩

3.3 Soundness of Forward Simulations

lemma *forward-sim-execs*:
fixes *A*::(*'sA*,*'a*)*ioa* **and** *B*::(*'sB*,*'a*)*ioa* **and** *f*::*'sB* ⇒ *'sA set* **and** *e-B*
assumes *is-forward-sim f B A* **and** *is-exec-of B e-B*
shows ∃ *e-A* . *is-exec-of A e-A*
∧ *trace* (*ioa.asig A*) *e-A* = *trace* (*ioa.asig A*) *e-B*
⟨*proof*⟩

theorem *forward-sim-soundness*:

fixes A::$('sA,'a)ioa$ **and** B::$('sB,'a)ioa$ **and** f::$'sB \Rightarrow 'sA\ set$
assumes *is-forward-sim f B A* **and** *ext A = ext B*
shows *traces B* $\subseteq$ *traces A*
⟨*proof*⟩

3.4 Soundness of Backward Simulations

lemma *backward-sim-execs*:
fixes A::$('sA,'a)ioa$ **and** B::$('sB,'a)ioa$ **and** f::$'sB \Rightarrow 'sA\ set$ **and** *e-B*
assumes *is-backward-sim f B A* **and** *is-exec-of B e-B*
shows $\exists$ *e-A . is-exec-of A e-A*
$\wedge$ *trace* (*ioa.asig A*) *e-A* = *trace* (*ioa.asig A*) *e-B*
⟨*proof*⟩

theorem *backward-sim-soundness*:
fixes A::$('sA,'a)ioa$ **and** B::$('sB,'a)ioa$ **and** f::$'sB \Rightarrow 'sA\ set$
assumes *is-backward-sim f B A* **and** *ext A = ext B*
shows *traces B* $\subseteq$ *traces A*
⟨*proof*⟩

end

end

4 Recoverable Data Types

theory *RDR*
imports *Main Sequences*
begin

4.1 The pre-RDR locale

locale *pre-RDR* = *Sequences* +
fixes δ::$'a \Rightarrow ('b \times 'c) \Rightarrow 'a$ (**infix** $\cdot$ *65*)
and γ::$'a \Rightarrow ('b \times 'c) \Rightarrow 'd$
and *bot*::$'a$ ($\perp$)
begin

fun *exec*::$'a \Rightarrow ('b \times 'c)list \Rightarrow 'a$ (**infix** $\star$ *65*) **where**
exec s Nil = *s*
| *exec s* (*rs*#*r*) = (*exec s rs*) $\cdot$ *r*

definition *less-eq* (**infix** $\preceq$ *50*) **where**
less-eq s s' $\equiv \exists$ *rs . s'* = (*s*$\star$*rs*)

definition *less* (**infix** $\prec$ *50*) **where**
less s s′ $\equiv$ *less-eq s s′* $\wedge$ $s \neq s'$

definition *is-lb* **where**
is-lb s s1 s2 $\equiv$ $s \preceq$ *s2* $\wedge$ $s \preceq$ *s1*

definition *is-glb* **where**
is-glb s s1 s2 $\equiv$ *is-lb s s1 s2* $\wedge$ $(\forall\ s'\ .$ *is-lb s′ s1 s2* $\longrightarrow s' \preceq s)$

definition *contains* **where**
contains s r $\equiv \exists\ rs\ .\ r \in$ *set rs* $\wedge\ s = (\bot \star rs)$

definition *inf* (**infix** $\sqcap$ *65*) **where**
inf s1 s2 $\equiv$ *THE s . is-glb s s1 s2*

4.2 Useful Lemmas in the pre-RDR locale

lemma *exec-cons*:
$s \star (rs \ \#\ r) = (s \star rs) \cdot r$ ⟨*proof*⟩

lemma *exec-append*:
$(s \star rs) \star rs' = s \star (rs @ rs')$
⟨*proof*⟩

lemma *trans*:
assumes *s1* $\preceq$ *s2* **and** *s2* $\preceq$ *s3*
shows *s1* $\preceq$ *s3* ⟨*proof*⟩

lemma *contains-star*:
fixes *s r rs*
assumes *contains s r*
shows *contains* $(s \star rs)$ *r*
⟨*proof*⟩

lemma *preceq-star*: $s \star (rs \# r) \preceq s' \Longrightarrow s \star rs \preceq s'$
⟨*proof*⟩

end

4.3 The RDR locale

locale *RDR* = *pre-RDR* +
assumes *idem1*:*contains s r* $\Longrightarrow s \cdot r = s$
and *idem2*:$\bigwedge$ *s r r′ . fst r* $\neq$ *fst r′* $\Longrightarrow \gamma\ s\ r = \gamma\ ((s \cdot r) \cdot r')\ r$
and *antisym*:$\bigwedge$ *s1 s2 . s1* $\preceq$ *s2* $\wedge$ *s2* $\preceq$ *s1* $\Longrightarrow$ *s1* $=$ *s2*

and *glb-exists*:$\bigwedge$ *s1 s2* . $\exists$ *s* . *is-glb s s1 s2*
and *consistency*:$\bigwedge$*s1 s2 s3 rs* . *s1* $\preceq$ *s2* $\Longrightarrow$ *s2* $\preceq$ *s3* $\Longrightarrow$ *s3* = *s1* $\star$ *rs*
$\Longrightarrow \exists$ *rs′ rs″* . *s2* = *s1* $\star$ *rs′* $\wedge$ *s3* = *s2* $\star$ *rs″*
$\wedge$ *set rs′* $\subseteq$ *set rs* $\wedge$ *set rs″* $\subseteq$ *set rs*
and *bot*:$\bigwedge$ *s* . $\bot \preceq$ *s*

begin

lemma *inf-glb*:*is-glb* (*s1* $\sqcap$ *s2*) *s1 s2*
⟨*proof*⟩

sublocale *ordering less-eq less*
⟨*proof*⟩

sublocale *semilattice-set inf*
⟨*proof*⟩

sublocale *semilattice-order-set inf less-eq less*
⟨*proof*⟩

notation *F* ($\prod$ - [*99*])

4.4 Some useful lemmas

lemma *idem-star*:
fixes *r s rs*
assumes *contains s r*
shows *s* $\star$ *rs* = *s* $\star$ (*filter* (λ *x* . *x* $\neq$ *r*) *rs*)
⟨*proof*⟩

lemma *idem-star2*:
fixes *s rs′*
shows $\exists$ *rs′* . *s* $\star$ *rs* = *s* $\star$ *rs′* $\wedge$ *set rs′* $\subseteq$ *set rs*
$\wedge$ ($\forall$ *r* $\in$ *set rs′* . $\neg$ *contains s r*)
⟨*proof*⟩

lemma *idem2-star*:
assumes *contains s r*
and $\bigwedge$ *r′* . *r′* $\in$ *set rs* $\Longrightarrow$ *fst r′* $\neq$ *fst r*
shows γ *s r* = γ (*s* $\star$ *rs*) *r* ⟨*proof*⟩

lemma *glb-common*:

fixes *s1 s2 s rs1 rs2*
assumes *s1* = *s* ⋆ *rs1* **and** *s2* = *s* ⋆ *rs2*
shows ∃ *rs* . *s1* ⊓ *s2* = *s* ⋆ *rs* ∧ *set rs* ⊆ *set rs1* ∪ *set rs2*
⟨*proof*⟩

lemma *glb-common-set*:
fixes *ss s0 rset*
assumes *finite ss* **and** *ss* ≠ {}
and ⋀ *s* . *s* ∈ *ss* ⟹ ∃ *rs* . *s* = *s0* ⋆ *rs* ∧ *set rs* ⊆ *rset*
shows ∃ *rs* . ⨅ *ss* = *s0* ⋆ *rs* ∧ *set rs* ⊆ *rset*
⟨*proof*⟩

end

end

5 The SLin Automata specification

theory *SLin*
imports *IOA RDR*
begin

datatype ('*a*,'*b*,'*c*,'*d*)*SLin-action* =
— The nat component is the instance number
  Invoke nat '*b* '*c*
| *Response nat* '*b* '*d*
| *Switch nat* '*b* '*c* '*a*
| *Recover nat*
| *Linearize nat*

datatype *SLin-status* = *Sleep* | *Pending* | *Ready* | *Aborted*

record ('*a*,'*b*,'*c*)*SLin-state* =
  pending :: '*b* ⇒ '*b* × '*c*
  initVals :: '*a set*
  abortVals :: '*a set*
  status :: '*b* ⇒ *SLin-status*
  dstate :: '*a*
  initialized :: *bool*

locale *SLin* = *RDR* + *IOA*
begin

definition

```
  asig :: nat ⇒ nat ⇒ ('a,'b,'c,'d)SLin-action signature
  — The first instance has number 0
  where
  asig i j ≡ (|
    inputs = {act . ∃ p c iv i' .
      (i ≤ i' ∧ i' < j ∧ act = Invoke i' p c) ∨ (i > 0 ∧ act = Switch i p c iv)},
    outputs = {act . ∃ p c av i' outp .
      (i ≤ i' ∧ i' < j ∧ act = Response i' p outp) ∨ act = Switch j p c av},
    internals = {act. ∃ i' . i ≤ i' ∧ i' < j
      ∧ (act = Linearize i' ∨ act = Recover i')} |)

definition pendingReqs :: ('a,'b,'c)SLin-state ⇒ ('b×'c) set
  where
  pendingReqs s ≡ {r . ∃ p .
     r = pending s p
     (*∧ ¬ contains (dstate s) r*)
     ∧ status s p ∈ {Pending, Aborted}}

definition Inv :: 'b ⇒ 'c
  ⇒ ('a,'b,'c)SLin-state ⇒ ('a,'b,'c)SLin-state ⇒ bool
where
  Inv p c s s' ≡
    status s p = Ready
    ∧ s' = s(|pending := (pending s)(p := (p,c)),
        status := (status s)(p := Pending)|)

definition pendingSeqs where
  pendingSeqs s ≡ {rs . set rs ⊆ pendingReqs s}

definition Lin :: ('a,'b,'c)SLin-state ⇒ ('a,'b,'c)SLin-state ⇒ bool
where
  Lin s s' ≡ ∃ rs ∈ pendingSeqs s .
      initialized s
      ∧ (∀ av ∈ abortVals s . (dstate s) ⋆ rs ≼ av)
      ∧ s' = s(|dstate := (dstate s) ⋆ rs|)

definition initSets where
  initSets s ≡ {ivs . ivs ≠ {} ∧ ivs ⊆ initVals s}

definition safeInits where
  safeInits s ≡ if initVals s = {} then {}
    else {d . ∃ ivs ∈ initSets s . ∃ rs ∈ pendingSeqs s .
      d = ⨅ ivs ⋆ rs ∧ (∀ av ∈ abortVals s . d ≼ av)}

definition initAborts where
```

```
  initAborts s ≡ { d .dstate s ≼ d
    ∧ ((∃ rs ∈ pendingSeqs s . d = dstate s ⋆ rs)
      ∨ (∃ ivs ∈ initSets s . dstate s ≼ ⨅ ivs
        ∧ (∃ rs ∈ pendingSeqs s . d = ⨅ ivs ⋆ rs))) }
definition uninitAborts where
  uninitAborts s ≡ { d .
    ∃ ivs ∈ initSets s . ∃ rs ∈ pendingSeqs s .
      d = ⨅ ivs ⋆ rs }

definition safeAborts::('a,'b,'c)SLin-state ⇒ 'a set where
  safeAborts s ≡ if initialized s then initAborts s
    else uninitAborts s

definition Reco :: ('a,'b,'c)SLin-state ⇒ ('a,'b,'c)SLin-state ⇒ bool
where
  Reco s s' ≡
      (∃ p . status s p ≠ Sleep)
      ∧ ¬ initialized s
      ∧ (∃ d ∈ safeInits s .
        s' = s(|dstate := d, initialized := True|))

definition Resp :: 'b ⇒ 'd ⇒ ('a,'b,'c)SLin-state ⇒ ('a,'b,'c)SLin-state ⇒ bool
where
  Resp p ou s s' ≡
      status s p = Pending
      ∧ initialized s
      ∧ contains (dstate s) (pending s p)
      ∧ ou = γ (dstate s) (pending s p)
      ∧ s' = s (|status := (status s)(p := Ready)|)

definition Init :: 'b ⇒ 'c ⇒ 'a
  ⇒ ('a,'b,'c)SLin-state ⇒ ('a,'b,'c)SLin-state ⇒ bool
where
  Init p c iv s s' ≡
    status s p = Sleep
    ∧ s' = s (|initVals := {iv} ∪ (initVals s),
        status := (status s)(p := Pending),
        pending := (pending s)(p := (p,c))|)

definition Abort :: 'b ⇒ 'c ⇒ 'a
  ⇒ ('a,'b,'c)SLin-state ⇒ ('a,'b,'c)SLin-state ⇒ bool
where
  Abort p c av s s' ≡
     status s p = Pending ∧ pending s p = (p,c)
     ∧ av ∈ safeAborts s
```

$\wedge\ s' = s(\!|status := (status\ s)(p := Aborted),$
$abortVals := (abortVals\ s \cup \{av\})|\!)$

definition *trans* **where**
trans i j $\equiv$ { (s,a,s') . *case a of*
Invoke i' p c $\Rightarrow$ $i \leq i' \wedge i < j \wedge$ *Inv p c s s'*
| *Response i' p ou* $\Rightarrow$ $i \leq i' \wedge i < j \wedge$ *Resp p ou s s'*
| *Switch i' p c v* $\Rightarrow$ $(i > 0 \wedge i' = i \wedge$ *Init p c v s s'*$)$
$\vee\ (i' = j \wedge$ *Abort p c v s s'*$)$
| *Linearize i'* $\Rightarrow$ $i' = i \wedge$ *Lin s s'*
| *Recover i'* $\Rightarrow$ $i > 0 \wedge i' = i \wedge$ *Reco s s'* }

definition *start* **where**
start i $\equiv$ { *s* .
$\forall$ *p* . *status s p* = (*if i* > *0 then Sleep else Ready*)
$\wedge$ *dstate s* $= \perp$
$\wedge$ (*if i* > *0 then* $\neg$ *initialized s else initialized s*)
$\wedge$ *initVals s* $= \{\}$
$\wedge$ *abortVals s* $= \{\}\}$

definition *ioa* **where**
ioa i j $\equiv$
$(\!|$*ioa.asig* = *asig i j* ,
start = *start i*,
trans = *trans i j*$|\!)$

end

end

6 The Consensus Data Type

theory *Consensus*
imports *RDR*
begin

This theory provides a model for the RDR locale, thus showing that the assumption of the RDR locale are consistent.

typedecl *proc*
typedecl *val*

locale *Consensus*
— To avoid name clashes
begin

fun δ::*val option* $\Rightarrow$ (*proc* $\times$ *val*) $\Rightarrow$ *val option* (**infix** $\cdot$ *65*) **where**
  δ *None r* = *Some* (*snd r*)
| δ (*Some v*) *r* = *Some v*

fun γ::*val option* $\Rightarrow$ (*proc* $\times$ *val*) $\Rightarrow$ *val* **where**
  γ *None r* = *snd r*
| γ (*Some v*) *r* = *v*

interpretation *pre-RDR* δ γ *None* ⟨*proof*⟩
notation *exec* (**infix** $\star$ *65*)
notation *less-eq* (**infix** $\preceq$ *50*)
notation *None* ($\bot$)

lemma *single-use*:
  fixes *r rs*
  shows $\bot \star ([r]@rs)$ = *Some* (*snd r*)
⟨*proof*⟩

lemma *bot*: $\exists$ *rs* . *s* = $\bot \star$ *rs*
⟨*proof*⟩

lemma *prec-eq-None-or-equal*:
fixes *s1 s2*
assumes *s1* $\preceq$ *s2*
shows *s1* = *None* $\vee$ *s1* = *s2* ⟨*proof*⟩

interpretation *RDR* δ γ $\bot$
⟨*proof*⟩

end

end

7 Idempotence of the SLin I/O automaton

theory *Idempotence*
imports *SLin Simulations*
begin

locale *Idempotence* = *SLin* +
  fixes *id1 id2* :: *nat*
  assumes *id1*:*0* < *id1* **and** *id2*:*id1* < *id2*
begin

lemmas *ids = id1 id2*

definition *composition* **where**
composition ≡
hide ((*ioa 0 id1*) ∥ (*ioa id1 id2*))
{*act . EX p c av . act = Switch id1 p c av* }

lemmas *comp-simps = hide-def composition-def ioa-def par2-def is-trans-def start-def actions-def asig-def trans-def*

lemmas *trans-defs = Inv-def Lin-def Resp-def Init-def Abort-def Reco-def*

declare *split-if-asm* [*split*]

7.1 A case rule for decomposing the transition relation of the composition of two SLins

declare *comp-simps* [*simp*]
lemma *trans-elim*:
fixes *s t a s′ t′ P*
assumes (*s*,*t*) −*a*−*composition*⟶ (*s′*,*t′*)
obtains
(*Invoke1*) *i p c*
where *Inv p c s s′* ∧ *t = t′*
and *i < id1* **and** *a = Invoke i p c*
| (*Invoke2*) *i p c*
where *Inv p c t t′* ∧ *s = s′*
and *id1* ≤ *i* ∧ *i < id2* **and** *a = Invoke i p c*
| (*Switch1*) *p c av*
where *Abort p c av s s′* ∧ *Init p c av t t′*
and *a = Switch id1 p c av*
| (*Switch2*) *p c av*
where *s = s′* ∧ *Abort p c av t t′*
and *a = Switch id2 p c av*
| (*Response1*) *i p ou*
where *Resp p ou s s′*∧ *t = t′*
and *i < id1* **and** *a = Response i p ou*
| (*Response2*) *i p ou*
where *Resp p ou t t′* ∧ *s = s′*
and *id1* ≤ *i* ∧ *i < id2* **and** *a = Response i p ou*
| (*Lin1*) *Lin s s′* ∧ *t = t′* **and** *a = Linearize 0*
| (*Lin2*) *Lin t t′* ∧ *s = s′* **and** *a = Linearize id1*
| (*Reco2*) *Reco t t′* ∧ *s = s′* **and** *a = Recover id1*

declare *comp-simps* [*simp del*]

7.2 Definition of the Refinement Mapping

fun $f :: (('a,'b,'c)SLin\text{-}state * ('a,'b,'c)SLin\text{-}state) \Rightarrow ('a,'b,'c)SLin\text{-}state$
where
$f\ (s1,\ s2) =$
$(\!|pending = \lambda\ p.\ (if\ status\ s1\ p \neq Aborted\ then\ pending\ s1\ p\ else\ pending\ s2\ p),$
$initVals = \{\},$
$abortVals = abortVals\ s2,$
$status = \lambda\ p.\ (if\ status\ s1\ p \neq Aborted\ then\ status\ s1\ p\ else\ status\ s2\ p),$
$dstate = (if\ dstate\ s2 = \bot\ then\ dstate\ s1\ else\ dstate\ s2),$
$initialized = True|\!)$

7.3 Invariants

declare
trans-defs [*simp*]

fun *P1* **where**
$P1\ (s1,s2) = (\forall\ p\ .\ status\ s1\ p \in \{Pending,\ Aborted\}$
$\longrightarrow fst\ (pending\ s1\ p) = p)$

fun *P2* **where**
$P2\ (s1,s2) = (\forall\ p\ .\ status\ s2\ p \neq Sleep \longrightarrow fst\ (pending\ s2\ p) = p)$

fun *P3* **where**
$P3\ (s1,s2) = (\forall\ p\ .\ (status\ s2\ p = Ready \longrightarrow initialized\ s2))$

fun *P4* **where**
$P4\ (s1,s2) = ((\forall\ p\ .\ status\ s2\ p = Sleep) = (initVals\ s2 = \{\}))$

fun *P5* **where**
$P5\ (s1,s2) = (\forall\ p\ .\ status\ s1\ p \neq Sleep \wedge initialized\ s1 \wedge initVals\ s1 = \{\})$

fun *P6* **where**
$P6\ (s1,s2) = (\forall\ p\ .\ (status\ s1\ p \neq Aborted) = (status\ s2\ p = Sleep))$

fun *P7* **where**
$P7\ (s1,s2) = (\forall\ c\ .\ status\ s1\ c = Aborted \wedge \neg\ initialized\ s2$
$\longrightarrow (pending\ s2\ c = pending\ s1\ c \wedge status\ s2\ c \in \{Pending,\ Aborted\}))$

fun *P8* **where**
P8 (*s1*,*s2*) = (∀ *iv* ∈ *initVals s2* . ∃ *rs* ∈ *pendingSeqs s1* .
iv = *dstate s1* ⋆ *rs*)

fun *P8a* **where**
P8a (*s1*,*s2*) = (∀ *ivs* ∈ *initSets s2* . ∃ *rs* ∈ *pendingSeqs s1* .
⨅ *ivs* = *dstate s1* ⋆ *rs*)

fun *P9* **where**
P9 (*s1*,*s2*) = (*initialized s2* ⟶ *dstate s1* ⪯ *dstate s2*)

fun *P10* **where**
P10 (*s1*,*s2*) = ((¬ *initialized s2*) ⟶ (*dstate s2* = ⊥))

fun *P11* **where**
P11 (*s1*,*s2*) = (*initVals s2* = *abortVals s1*)

fun *P12* **where**
P12 (*s1*,*s2*) = (*initialized s2* ⟶ ⨅ (*initVals s2*) ⪯ *dstate s2*)

fun *P13* **where**
P13 (*s1*,*s2*) = (*finite* (*initVals s2*)
∧ *finite* (*abortVals s1*) ∧ *finite* (*abortVals s2*))

fun *P14* **where**
P14 (*s1*,*s2*) = (*initialized s2* ⟶ *initVals s2* ≠ {})

fun *P15* **where**
P15 (*s1*,*s2*) = (∀ *av* ∈ *abortVals s1* . *dstate s1* ⪯ *av*)

fun *P16* **where**
P16 (*s1*,*s2*) = (*dstate s2* ≠ ⊥ ⟶ *initialized s2*)

fun *P17* **where**
— For the Response1 case of the refinement proof, in case a response is produced in the first instance and the second instance is already initialized
P17 (*s1*,*s2*) = (*initialized s2*
⟶ (∀ *p* .
((*status s1 p* = *Ready*
∨ (*status s1 p* = *Pending* ∧ *contains* (*dstate s1*) (*pending s1 p*)))
⟶ (∃ *rs* . *dstate s2* = *dstate s1* ⋆ *rs* ∧ (∀ *r* ∈ *set rs* . *fst r* ≠ *p*)))
∧ ((*status s1 p* = *Pending* ∧ ¬ *contains* (*dstate s1*) (*pending s1 p*))
⟶ (∃ *rs* . *dstate s2* = *dstate s1* ⋆ *rs* ∧ (∀ *r* ∈ *set rs* .

$fst\ r = p \longrightarrow r = pending\ s1\ p)))))$

fun *P18* **where**
$P18\ (s1,s2) = (abortVals\ s2 \neq \{\} \longrightarrow (\exists\ p\ .\ status\ s2\ p \neq Sleep))$

fun *P19* **where**
$P19\ (s1,s2) = (abortVals\ s2 \neq \{\} \longrightarrow abortVals\ s1 \neq \{\})$

fun *P20* **where**
$P20\ (s1,s2) = (\forall\ av \in abortVals\ s2\ .\ dstate\ s2 \preceq av)$

fun *P21* **where**
$P21\ (s1,s2) = (\forall\ av \in abortVals\ s2\ .\ \bigsqcap(abortVals\ s1) \preceq av)$

fun *P22* **where**
$P22\ (s1,s2) = (initialized\ s2 \longrightarrow dstate\ (f\ (s1,s2)) = dstate\ s2)$

fun *P23* **where**
$P23\ (s1,s2) = ((\neg\ initialized\ s2) \longrightarrow$
$pendingSeqs\ s1 \subseteq pendingSeqs\ (f\ (s1,s2)))$

fun *P25* **where**
$P25\ (s1,s2) = (\forall\ ivs\ .\ (ivs \in initSets\ s2 \wedge initialized\ s2$
$\wedge\ dstate\ s2 \preceq \bigsqcap ivs)$
$\longrightarrow (\exists\ rs' \in pendingSeqs\ (f\ (s1,s2))\ .\ \bigsqcap ivs = dstate\ s2 \star rs'))$

fun *P26* **where**
$P26\ (s1,s2) = (\forall\ p\ .\ (status\ s1\ p = Aborted$
$\wedge \neg\ contains\ (dstate\ s2)\ (pending\ s1\ p))$
$\longrightarrow (status\ s2\ p \in \{Pending,Aborted\}$
$\wedge\ pending\ s1\ p = pending\ s2\ p))$

lemma *P1-invariant*:
shows *invariant* (*composition*) *P1*
⟨*proof*⟩

lemma *P2-invariant*:
shows *invariant* (*composition*) *P2*
⟨*proof*⟩

lemma *P16-invariant*:
shows *invariant* (*composition*) *P16*

⟨*proof*⟩

lemma *P3-invariant*:
shows *invariant* (*composition*) *P3*
⟨*proof*⟩

lemma *P4-invariant*:
shows *invariant* (*composition*) *P4*
⟨*proof*⟩

lemma *P5-invariant*:
shows *invariant* (*composition*) *P5*
⟨*proof*⟩

lemma *P13-invariant*:
shows *invariant* (*composition*) *P13*
⟨*proof*⟩

lemma *P20-invariant*:
shows *invariant* (*composition*) *P20*
⟨*proof*⟩

lemma *P18-invariant*:
shows *invariant* (*composition*) *P18*
⟨*proof*⟩

lemma *P14-invariant*:
shows *invariant* (*composition*) *P14*
⟨*proof*⟩

lemma *P15-invariant*:
shows *invariant* (*composition*) *P15*
⟨*proof*⟩

lemma *P6-invariant*:
shows *invariant* (*composition*) *P6*
⟨*proof*⟩

lemma *P7-invariant*:
shows *invariant* (*composition*) *P7*
⟨*proof*⟩

lemma *P10-invariant*:
shows *invariant* (*composition*) *P10*

⟨*proof*⟩

lemma *P11-invariant*:
shows *invariant* (*composition*) *P11*
⟨*proof*⟩

lemma *P8-invariant*:
shows *invariant* (*composition*) *P8*
⟨*proof*⟩

lemma *P8a-invariant*:
shows *invariant* (*composition*) *P8a*
⟨*proof*⟩

lemma *P12-invariant*:
shows *invariant* (*composition*) *P12*
⟨*proof*⟩

lemma *P19-invariant*:
shows *invariant* (*composition*) *P19*
⟨*proof*⟩

lemma *P9-invariant*:
shows *invariant* (*composition*) *P9*
⟨*proof*⟩

lemma *P17-invariant*:
shows *invariant* (*composition*) *P17*
⟨*proof*⟩

lemma *P21-invariant*:
shows *invariant* (*composition*) *P21*
⟨*proof*⟩

lemma *P22-invariant*:
shows *invariant* (*composition*) *P22*
⟨*proof*⟩

lemma *P23-invariant*:
shows *invariant* (*composition*) *P23*
⟨*proof*⟩

lemma *P26-invariant*:
shows *invariant* (*composition*) *P26*
⟨*proof*⟩

lemma *P25-invariant*:
shows *invariant* (*composition*) *P25*
⟨*proof*⟩

7.4 Proof of the Idempotence Theorem

theorem *idempotence*:
  shows ((*composition*) =<| (*ioa 0 id2*))
⟨*proof*⟩

end

end