syber-x-project

Xuwj

May 12, 2021

Contents

1 Security Model of Separation Kernels

theory *YX-SecurityModel*
imports *Main*
begin

1.1 Security State Machine

locale *SM* =
  fixes *s0* :: ′*s*
  fixes *step* :: ′*e* ⇒ (′*s* × ′*s*) *set*
  fixes *domain* :: ′*s* ⇒ ′*e* ⇒ (′*d option*)
  fixes *sched* :: ′*d*
  fixes *vpeq* :: ′*s* ⇒ ′*d* ⇒ ′*s* ⇒ *bool* ((- ∼ - ∼ -))
  fixes *interference* :: ′*d* ⇒ ′*d* ⇒ *bool* ((- -))
  fixes *audit-policy* :: ′*e* ⇒ *bool*
  fixes *access-control* :: ′*d* ⇒ ′*obj* ⇒ ′*p* ⇒ *bool*
  fixes *issysPartition* :: ′*d* ⇒ *bool*
  fixes *issysEvent* :: ′*e* ⇒ *bool*
  fixes *audit-info* :: ′*s* ⇒ ′*e* ⇒ ′*a*
  assumes
    vpeq-transitive-lemma : ∀ *s t r d*. (*s* ∼ *d* ∼ *t*) ∧ (*t* ∼ *d* ∼ *r*) ⟶ (*s* ∼ *d* ∼ *r*)
and
    vpeq-symmetric-lemma : ∀ *s t d*. (*s* ∼ *d* ∼ *t*) ⟶ (*t* ∼ *d* ∼ *s*) **and**
    vpeq-reflexive-lemma : ∀ *s d*. (*s* ∼ *d* ∼ *s*) **and**
    sched-vpeq : ∀ *s t a*. (*s* ∼ *sched* ∼ *t*) ⟶ (*domain s a*) = (*domain t a*) **and**
    sched-intf-all : ∀ *d*. (*sched*   *d*) **and**
    no-intf-sched : ∀ *d*. (*d*   *sched*) ⟶ *d* = *sched* **and**
    interf-reflexive: ∀ *d*. (*d*   *d*) **and**
    api-protection: ∀ *e s* . (*issysEvent e*) ⟶ *issysPartition* (*the* (*domain s e*)) **and**
    audit-stable : ∀ *e s t*. *audit-policy e* ∧ (*s*,*t*) ∈ *step e*⟶ (∃ *a*. *a* = *audit-info t e*)

begin

  definition *non-interference* :: ′*d* ⇒ ′*d* ⇒ *bool* ((- \ -))

```
  where (u \  v) ≡ ¬ (u  v)

definition ivpeq :: 's ⇒ 'd set ⇒ 's ⇒ bool ((- ≈ - ≈ -))
where ivpeq s D t ≡ ∀ d ∈ D. (s ∼ d ∼ t)

primrec run :: 'e list ⇒ ('s × 's) set
  where run-Nil: run [] = Id |
        run-Cons: run (a#as) = step a O run as

definition next-states :: 's ⇒ 'e ⇒ 's set
  where next-states s a ≡ {Q. (s,Q)∈step a}

term  (run as)

definition execute  :: 'e list ⇒ 's ⇒ 's set
  where execute as s = Image (run as) {s}

definition reachable :: 's ⇒ 's ⇒ bool ((- ↪ -) [70,71] 60) where
  reachable s1 s2 ≡  (∃ as. (s1,s2) ∈ run as)

definition reachable0 :: 's ⇒ bool  where
  reachable0 s ≡ reachable s0 s

lemma reachable-s0 : reachable0 s0
 by (metis SM.reachable-def SM-axioms pair-in-Id-conv reachable0-def run.simps(1))


lemma reachable-self : reachable s s
  using reachable-def run.simps(1) by fastforce

lemma reachable-step : (s,s')∈step a ⟹ reachable s s'
  proof−
    assume a0: (s,s')∈step a
    then have (s,s')∈run [a] by simp

    then show ?thesis using reachable-def by blast
  qed

  lemma run-trans : ∀ C T V as bs. (C,T)∈run as ∧ (T,V)∈run bs ⟶
(C,V)∈run (as@bs)
  proof −
  {
    fix T V as bs
    have ∀ C. (C,T)∈run as ∧ (T,V)∈run bs ⟶ (C,V)∈run (as@bs)
      proof(induct as)
        case Nil show ?case by simp
      next
        case (Cons c cs)
         assume a0: ∀ C. (C, T) ∈ run cs ∧ (T, V) ∈ run bs ⟶ (C, V) ∈
```

```
run (cs @ bs)
          show ?case
            proof−
            {
              fix C
              have (C, T) ∈ run (c # cs) ∧ (T, V) ∈ run bs ⟶ (C, V) ∈ run
((c # cs) @ bs)
                proof
                  assume b0: (C, T) ∈ run (c # cs) ∧ (T, V) ∈ run bs
                   from b0 obtain C' where b2: (C,C')∈step c ∧ (C',T)∈run cs
by auto
                  with a0 b0 have (C',V)∈run (cs@bs) by blast
                  with b2 show (C, V) ∈ run ((c # cs) @ bs)
                    using append-Cons relcomp.relcompI run-Cons by auto
                qed
            }
            then show ?thesis by auto
            qed
        qed
    }
    then show ?thesis by auto
    qed

  lemma reachable-trans : ⟦reachable C T; reachable T V⟧ ⟹ reachable C V
    proof−
      assume a0: reachable C T
      assume a1: reachable T V
     from a0 have C = T ∨ (∃ as. (C,T)∈run as) using reachable-def by simp
      then show ?thesis
        proof
          assume b0: C = T
          show ?thesis
            proof −
              from a1 have T = V ∨ (∃ as. (T,V)∈run as) using reachable-def
by simp
              then show ?thesis
                proof
                  assume c0: T=V
                  with a0 show ?thesis by simp
                next
                  assume c0: (∃ as. (T,V)∈run as)
                  then show ?thesis by (simp add: a1 b0)
                qed
            qed
        next
          assume b0: ∃ as. (C,T)∈run as
          show ?thesis
            proof −
              from a1 have T = V ∨ (∃ as. (T,V)∈run as) using reachable-def
```

```
by simp
              then show ?thesis
                proof
                  assume c0: T=V
                  then show ?thesis using a0 by auto
                next
                  assume c0: (∃ as. (T,V)∈run as)
                  from b0 obtain as where d0: (C,T)∈run as by auto
                  from c0 obtain bs where d1: (T,V)∈run bs by auto
                  then show ?thesis using d0 reachable-def run-trans by blast
                qed
            qed
        qed
    qed

  lemma reachableStep : ⟦reachable0 C; (C,C')∈step a⟧ ⟹ reachable0 C'
    proof −
      assume a0: reachable0 C
      assume a1: (C,C')∈step a
     from a0 have (C = s0) ∨ (∃ as. (s0,C) ∈ run as) unfolding reachable0-def
reachable-def by auto
      then show reachable0 C'
        proof
          assume b0: C = s0
          show reachable0 C'
            using a1 b0 reachable0-def reachable-step by auto
        next
          assume b0: ∃ as. (s0,C) ∈ run as
          show reachable0 C'
            using a0 a1 reachable0-def reachable-step reachable-trans by blast
        qed
    qed

  lemma reachable0-reach : ⟦reachable0 C; reachable C C'⟧ ⟹ reachable0 C'
    using reachable0-def reachable-trans by blast

end
```

1.2 Information flow security properties

```
locale SM-enabled = SM s0 step domain sched vpeq interference
  for s0 :: 's and
      step :: 'e ⇒ ('s × 's) set and
      domain :: 's ⇒ 'e ⇒ ('d option) and
      sched :: 'd and
      vpeq :: 's ⇒ 'd ⇒ 's ⇒ bool  ((- ∼ - ∼ -)) and
      interference :: 'd ⇒ 'd ⇒ bool ((-  -))
  +
  assumes enabled0: ∀ s a. reachable0 s ⟶ (∃ s'. (s,s') ∈ step a)
```

```
begin
  lemma enabled : reachable0 s ⟹ (∃ s′. (s,s′) ∈ step a)
    using enabled0 by simp

  lemma enabled-ex: ∀ s es. reachable0 s ⟶ (∃ s′. s′ ∈ execute es s)
    proof −
    {
      fix es
      have ∀ s. reachable0 s ⟶ (∃ s′. s′ ∈ execute es s)
        proof(induct es)
          case Nil show ?case by (simp add: execute-def)
        next
          case (Cons a as)
          assume a0: ∀ s. reachable0 s ⟶ (∃ s′. s′ ∈ execute as s)
          show ?case
            proof−
            {
              fix s
              assume b0: reachable0 s
                   have b1: ∃ s1. (s,s1)∈step a using enabled b0 by (simp ad-
d:next-states-def)
              then obtain s1 where b2: (s,s1)∈step a by auto
              with a0 b0 have b3: ∃ s′. s′ ∈ execute as s1
                using reachableStep by blast
              then obtain s2 where b4: s2∈execute as s1 by auto
              then have s2 ∈ execute (a # as) s
             using Image-singleton-iff SM.execute-def SM-axioms b2 relcomp.simps
run-Cons by fastforce
              then have ∃ s′. s′ ∈ execute (a # as) s by auto
            }
            then show ?thesis by auto
            qed
        qed
    }
    then show ?thesis by auto
    qed

  lemma enabled-ex2: reachable0 s ⟹ (∃ s′. s′ ∈ execute es s)
    using enabled-ex by auto

  primrec sources :: ′e list ⇒ ′s ⇒ ′d ⇒ ′d set where
    sources-Nil:sources [] s d = {d} |
    sources-Cons:sources (a # as) s d = (⋃{sources as s′ d| s′. (s,s′) ∈ step a})
∪
                          {w . w = the (domain s a) ∧ (∃ v s′. (w   v) ∧
                               (s,s′)∈step a ∧ v∈sources as s′ d)}
  declare sources-Nil [simp del]
  declare sources-Cons [simp del]
```

primrec *ipurge* :: $'e\ list \Rightarrow 'd \Rightarrow 's\ set \Rightarrow 'e\ list$ **where**
ipurge-Nil: $ipurge\ []\ u\ ss = []\ |$
ipurge-Cons: $ipurge\ (a\#as)\ u\ ss = (if\ \exists s{\in}ss.\ the\ (domain\ s\ a) \in (sources\ (a\#as)\ s\ u)\ then$
$a\ \#\ ipurge\ as\ u\ (\bigcup s{\in}ss.\ \{s'.\ (s,s') \in step\ a\})$
$else$
$ipurge\ as\ u\ ss$
$)$

definition *observ-equivalence* :: $'s \Rightarrow 'e\ list \Rightarrow 's \Rightarrow$
$'e\ list \Rightarrow 'd \Rightarrow bool\ ((\text{-}\ \ \text{-} \cong \text{-}\ \ \text{-}\ @\ \text{-}))$
where $observ\text{-}equivalence\ s\ as\ t\ bs\ d \equiv$
$\forall s'\ t'.\ ((s,s'){\in}\ run\ as \wedge (t,t'){\in}\ run\ bs) \longrightarrow (s' \sim d \sim t')$

lemma *observ-equiv-sym*:
$(s\ \ as \cong t\ \ bs\ @\ d) \Longrightarrow (t\ \ bs \cong s\ \ as\ @\ d)$
using *observ-equivalence-def vpeq-symmetric-lemma* **by** *blast*

lemma *observ-equiv-trans*:
$[\![reachable0\ t;\ (s\ \ as \cong t\ \ bs\ @\ d);\ (t\ \ bs \cong x\ \ cs\ @\ d)]\!] \Longrightarrow (s\ \ as \cong x\ \ cs\ @\ d)$
apply(*clarsimp simp*: *observ-equivalence-def*)
apply(*cut-tac* $s{=}t$ **and** $es{=}bs$ **in** *enabled-ex2*)
apply *simp*
by (*metis* (*no-types*, *hide-lams*) *Image-singleton-iff execute-def vpeq-transitive-lemma*)

lemma *exec-equiv-leftI*:
$[\![reachable0\ C;\ \forall\ C'.\ (C,C'){\in}step\ a \longrightarrow (C'\ \ as \cong D\ \ bs\ @\ d)]\!] \Longrightarrow (C\ \ (a\ \#\ as) \cong D\ \ bs\ @\ d)$
proof −
assume *a0*: $reachable0\ C$
assume *a1*: $\forall\ C'.\ (C,C'){\in}step\ a \longrightarrow (C'\ \ as \cong D\ \ bs\ @\ d)$
have $\forall S'\ T'.\ ((C,S'){\in}\ run\ (a\#as) \wedge (D,T'){\in}\ run\ bs) \longrightarrow (S' \sim d \sim T')$
proof −
{
fix $S'\ T'$
assume *b0*: $(C,\ S') \in run\ (a\ \#\ as) \wedge (D,\ T') \in run\ bs$
then obtain C' **where** *b1*: $(C,C'){\in}step\ a \wedge (C',S'){\in}run\ as$
using *relcompEpair run-Cons* **by** *auto*
with *a1* **have** *b2*: $(C'\ \ as \cong D\ \ bs\ @\ d)$ **by** *simp*
with *b0 b1* **have** $S' \sim d \sim T'$ **by** (*simp add*: *observ-equivalence-def*)
}
then show *?thesis* **by** *auto*
qed
then show *?thesis* **using** *observ-equivalence-def* **by** *blast*
qed

lemma *exec-equiv-both*:

$[\![$*reachable0 C1*; *reachable0 C2*; $\forall$ *C1' C2'*. (*C1*,*C1'*)$\in$*step a* $\wedge$(*C2*,*C2'*)$\in$*step b*$\longrightarrow$ (*C1'* *as* $\cong$ *C2'* *bs* @ *u*)$]\!]$
$\Longrightarrow$ (*C1* (*a* # *as*) $\cong$ *C2* (*b* # *bs*) @ *u*)
proof −
assume *a0*: *reachable0 C1*
assume *a1*: *reachable0 C2*
assume *a2*: $\forall$ *C1' C2'*. (*C1*,*C1'*)$\in$*step a* $\wedge$(*C2*,*C2'*)$\in$*step b*$\longrightarrow$ (*C1'* *as* $\cong$ *C2'* *bs* @ *u*)
have $\forall$ *S' T'*. ((*C1*,*S'*)$\in$ *run* (*a* # *as*) $\wedge$ (*C2*,*T'*)$\in$ *run* (*b* # *bs*)) $\longrightarrow$(*S'* $\sim$ *u* $\sim$ *T'*)
proof −
{
fix *S' T'*
assume *b0*: (*C1*, *S'*) $\in$ *run* (*a* # *as*) $\wedge$ (*C2*, *T'*) $\in$ *run* (*b* # *bs*)
then obtain *C1'* **where** *b1*: (*C1*,*C1'*)$\in$*step a* $\wedge$ (*C1'*,*S'*)$\in$*run as*
using *relcompEpair run-Cons* **by** *auto*
from *b0* **obtain** *C2'* **where** *b2*: (*C2*,*C2'*)$\in$*step b* $\wedge$ (*C2'*,*T'*)$\in$*run bs*
using *relcompEpair run-Cons* **by** *auto*
with *a2 b1* **have** *b3*: (*C1'* *as* $\cong$ *C2'* *bs* @ *u*) **by** *blast*
with *b0 b1 b2* **have** *S'* $\sim$ *u* $\sim$ *T'* **by** (*simp add*: *observ-equivalence-def*)
}
then show *?thesis* **by** *auto*
qed
then show *?thesis* **by** (*simp add*: *observ-equivalence-def*)
qed

lemma *sources-refl*:*reachable0 s* $\Longrightarrow$ *u* $\in$ *sources as s u*
apply(*induct as arbitrary*: *s*)
apply(*simp add*: *sources-Nil*)
apply(*simp add*: *sources-Cons*)
using *enabled reachableStep* **by** *metis*

lemma *scheduler-in-sources-Cons*:
reachable0 s $\Longrightarrow$ *the* (*domain s a*) = *sched* $\Longrightarrow$ *the* (*domain s a*) $\in$ *sources* (*a*#*as*) *s u*
apply(*unfold sources-Cons*)
apply(*erule ssubst*)
apply(*rule UnI2*)
apply(*clarsimp*)
apply(*rule-tac x=u* **in** *exI*)
apply(*safe*)
apply (*simp add*: *sched-intf-all*)
using *enabled reachableStep sources-refl* **by** *blast*

definition *noninterference-r* :: *bool*
where *noninterference-r* $\equiv$ $\forall$ *d as s*. *reachable0 s* $\longrightarrow$ (*s* *as* $\cong$ *s* (*ipurge as d* {*s*}) @ *d*)

definition *noninterference* :: *bool*
where *noninterference* $\equiv \forall\, d\ as.\ (s0\ \ as \cong s0\ \ (ipurge\ as\ d\ \{s0\})\ @\ d)$

definition *weak-noninterference* :: *bool*
where *weak-noninterference* $\equiv \forall\, d\ as\ bs.\ ipurge\ as\ d\ \{s0\} = ipurge\ bs\ d\ \{s0\}$
$\longrightarrow (s0\ \ as \cong s0\ \ bs\ @\ d)$

definition *weak-noninterference-r* :: *bool*
where *weak-noninterference-r* $\equiv \forall\, d\ as\ bs\ s.\ reachable0\ s \wedge ipurge\ as\ d\ \{s\} = ipurge\ bs\ d\ \{s\}$
$\longrightarrow (s\ \ as \cong s\ \ bs\ @\ d)$

definition *noninfluence*::*bool*
where *noninfluence* $\equiv \forall\ d\ as\ s\ t\ .\ reachable0\ s \wedge reachable0\ t \wedge (s \approx (sources\ as\ s\ d) \approx t)$
$\wedge\ (s \sim sched \sim t) \longrightarrow (s\ \ as \cong t\ \ (ipurge\ as\ d\ \{t\})\ @\ d)$

definition *noninfluence-gen*::*bool*
where *noninfluence-gen* $\equiv \forall\ d\ as\ s\ ts\ .\ reachable0\ s \wedge (\forall\, t{\in}ts.\ reachable0\ t)$
$\wedge\ (\forall\, t{\in}ts.\ (s \approx (sources\ as\ s\ d) \approx t))$
$\wedge\ (\forall\, t{\in}ts.\ (s \sim sched \sim t))$
$\longrightarrow (\forall\, t{\in}ts.\ (s\ \ as \cong t\ \ (ipurge\ as\ d\ ts)\ @\ d))$

definition *weak-noninfluence* ::*bool*
where *weak-noninfluence* $\equiv \forall\ d\ as\ bs\ s\ t\ .\ reachable0\ s \wedge reachable0\ t \wedge (s \approx (sources\ as\ s\ d) \approx t)$
$\wedge\ (s \sim sched \sim t) \wedge ipurge\ as\ d\ \{s\} = ipurge\ bs\ d\ \{s\}$
$\longrightarrow (s\ \ as \cong t\ \ bs\ @\ d)$

definition *weak-noninfluence2* ::*bool*
where *weak-noninfluence2* $\equiv \forall\ d\ as\ bs\ s\ t\ .\ reachable0\ s \wedge reachable0\ t \wedge (s \approx (sources\ as\ s\ d) \approx t)$
$\wedge\ (s \sim sched \sim t) \wedge ipurge\ as\ d\ \{s\} = ipurge\ bs\ d\ \{t\}$
$\longrightarrow (s\ \ as \cong t\ \ bs\ @\ d)$

definition *nonleakage* :: *bool*
where *nonleakage* $\equiv \forall\, d\ as\ s\ t.\ reachable0\ s \wedge reachable0\ t \wedge (s \sim sched \sim t)$
$\wedge\ (s \approx (sources\ as\ s\ d) \approx t) \longrightarrow (s\ \ as \cong t\ \ as\ @\ d)$

1.3 Unwinding conditions

definition *step-consistent* :: *bool* **where**
step-consistent $\equiv \forall\, a\ d\ s\ t.\ reachable0\ s \wedge reachable0\ t \wedge (s \sim d \sim t) \wedge (s \sim sched \sim t) \wedge$
$(((the\ (domain\ s\ a))\ \ d) \longrightarrow (s \sim (the\ (domain\ s\ a)) \sim t)) \longrightarrow$

$(\forall\ s'\ t'.\ (s,s') \in step\ a \wedge (t,t') \in step\ a \longrightarrow (s' \sim d \sim t'))$

definition *weak-step-consistent* :: *bool* **where**
weak-step-consistent $\equiv \forall\ a\ d\ s\ t.\ reachable0\ s \wedge reachable0\ t \wedge (s \sim d \sim t) \wedge (s \sim sched \sim t) \wedge$
$((the\ (domain\ s\ a))\ \ d) \wedge (s \sim (the\ (domain\ s\ a)) \sim t) \longrightarrow$
$(\forall\ s'\ t'.\ (s,s') \in step\ a \wedge (t,t') \in step\ a \longrightarrow (s' \sim d \sim t'))$

definition *step-consistent-e* :: $'e \Rightarrow bool$ **where**
step-consistent-e $a \equiv \forall\ d\ s\ t.\ reachable0\ s \wedge reachable0\ t \wedge (s \sim d \sim t) \wedge (s \sim sched \sim t) \wedge$
$(((the\ (domain\ s\ a))\ \ d) \longrightarrow (s \sim (the\ (domain\ s\ a)) \sim t)) \longrightarrow$
$(\forall\ s'\ t'.\ (s,s') \in step\ a \wedge (t,t') \in step\ a \longrightarrow (s' \sim d \sim t'))$

definition *weak-step-consistent-e* :: $'e \Rightarrow bool$ **where**
weak-step-consistent-e $a \equiv \forall\ d\ s\ t.\ reachable0\ s \wedge reachable0\ t \wedge (s \sim d \sim t) \wedge (s \sim sched \sim t) \wedge$
$((the\ (domain\ s\ a))\ \ d) \wedge (s \sim (the\ (domain\ s\ a)) \sim t) \longrightarrow$
$(\forall\ s'\ t'.\ (s,s') \in step\ a \wedge (t,t') \in step\ a \longrightarrow (s' \sim d \sim t'))$

definition *local-respect* :: *bool* **where**
local-respect $\equiv \forall\ a\ d\ s\ s'.$
$reachable0\ s \wedge ((the\ (domain\ s\ a)) \setminus d) \wedge (s,s') \in step\ a \longrightarrow (s \sim d \sim s')$

definition *local-respect-e* :: $'e \Rightarrow bool$ **where**
local-respect-e $a \equiv \forall\ d\ s\ s'.$
$reachable0\ s \wedge ((the\ (domain\ s\ a)) \setminus d) \wedge (s,s') \in step\ a \longrightarrow (s \sim d \sim s')$

lemma *local-respect-all-evt* : *local-respect* = ($\forall$ *a*. *local-respect-e a*)
by (*simp add*: *local-respect-def local-respect-e-def*)

lemma *step-consistent-all-evt* : *step-consistent* = ($\forall$ *a*. *step-consistent-e a*)
by (*simp add*: *step-consistent-def step-consistent-e-def*)

lemma *weak-step-consistent-all-evt* : *weak-step-consistent* = ($\forall$ *a*. *weak-step-consistent-e a*)
by (*simp add*:*weak-step-consistent-def weak-step-consistent-e-def*)

lemma *step-cons-impl-weak* : *step-consistent* $\Longrightarrow$ *weak-step-consistent*
using *step-consistent-def weak-step-consistent-def* **by** *blast*

lemma *weak-with-step-cons*:
assumes *p1*:*weak-step-consistent*
and *p2*:*local-respect*
shows *step-consistent*
proof –

{
fix $d\ a\ s\ t\ s'\ t'$
have *reachable0 s* ∧ *reachable0 t* ⟶ (*s* ∼ *d* ∼ *t*) ∧ (*s* ∼ *sched* ∼ *t*) ∧
(((*the* (*domain s a*)) *d*) ⟶ (*s* ∼ (*the* (*domain s a*)) ∼ *t*)) ⟶
(*s*,*s'*)∈*step a* ∧ (*t*,*t'*)∈*step a*
⟶ (*s'* ∼ *d* ∼ *t'*)
proof −
{
assume *aa*:*reachable0 s* ∧ *reachable0 t*
assume *a0*:*s* ∼ *d* ∼ *t*
assume *a1*:*s* ∼ *sched* ∼ *t*
assume *a2*:((*the* (*domain s a*)) *d*) ⟶ (*s* ∼ (*the* (*domain s a*)) ∼ *t*)
assume *a3*: (*s*,*s'*)∈*step a* ∧ (*t*,*t'*)∈*step a*
have *s'* ∼ *d* ∼ *t'*
proof(*cases* (*the* (*domain s a*)) *d*)
assume *b0*:(*the* (*domain s a*)) *d*
show *?thesis* **using** *aa a0 a1 a2 b0 p1 weak-step-consistent-def a3* **by** *blast*
next
assume *b1*:¬((*the* (*domain s a*)) *d*)
have *b2*:(*domain s a*) = (*domain t a*) **by** (*simp add*: *a1 sched-vpeq*)
with *b1* **have** *b3*:¬((*the* (*domain t a*)) *d*) **by** *auto*
then have *b4*:*s*∼*d*∼*s'* **using** *aa b1 local-respect-def non-interference-def p2 a3* **by** *auto*
then have *b5*:*t*∼*d*∼*t'* **using** *aa b3 local-respect-def non-interference-def p2 a3* **by** *auto*
then show *?thesis* **using** *a0 b4 vpeq-symmetric-lemma vpeq-transitive-lemma* **by** *blast*
qed
}
then show *?thesis* **by** *auto*
qed
}
then show *?thesis* **using** *step-consistent-def* **by** *blast*
qed

1.4 Lemmas for the inference framework

lemma *sched-equiv-preserved*:
assumes *1*:*step-consistent*
and *2*:*s* ∼*sched*∼ *t*
and *3*:(*s*,*s'*)∈*step a*
and *4*:(*t*,*t'*)∈*step a*
and *5*:*reachable0 s* ∧ *reachable0 t*
shows *s'* ∼*sched*∼ *t'*
apply(*case-tac the* (*domain s a*) = *sched*)
using *1 2 3 4 5 step-consistent-def* **apply** *blast*
using *1 2 3 4 5 no-intf-sched step-consistent-def* **by** *blast*

lemma *sched-equiv-preserved-left*:
$[\![$*local-respect*; *reachable0 s*; (*s* $\sim$*sched*$\sim$ *t*); *the* (*domain s a*) $\neq$ *sched*; (*s*,*s*$'$)$\in$ *step a*$]\!]$
$\Longrightarrow$ (*s*$'$ $\sim$*sched*$\sim$ *t*)
using *local-respect-def no-intf-sched non-interference-def vpeq-symmetric-lemma vpeq-transitive-lemma* **by** *blast*

lemma *un-eq*:
$[\![S = S'; T = T']\!] \Longrightarrow S \cup T = S' \cup T'$
apply *auto*
done
lemma *Un-eq*:
$[\![\bigwedge x\ y.\ [\![x \in xs;\ y \in ys]\!] \Longrightarrow P\ x = Q\ y;\ \exists\ x.\ x \in xs;\ \exists\ y.\ y \in ys]\!] \Longrightarrow (\bigcup x{\in}xs.\ P\ x) = (\bigcup y{\in}ys.\ Q\ y)$
apply *auto*
done

lemma *sources-eq0*: *step-consistent* $\wedge$ (*s* $\sim$ *sched* $\sim$ *t*) $\wedge$ *reachable0 s* $\wedge$ *reachable0 t*
$\longrightarrow$ *sources as s d* = *sources as t d*
proof (*induct as arbitrary*: *s t*)
case *Nil* **show** *?case*
by (*simp add*: *sources-Nil*)
next
case (*Cons a as*) **show** *?case*
apply(*clarsimp simp*: *sources-Cons*)
apply(*rule un-eq*)
apply(*simp only*: *Union-eq*, *simp only*: *UNION-eq*[*symmetric*])
apply(*rule Un-eq*, *clarsimp*)
apply(*metis Cons.hyps*[*rule-format*] *sched-equiv-preserved reachableStep*)
using *enabled* **apply** *simp*
using *enabled* **apply** *simp*
apply(*clarsimp simp*: *sched-vpeq*)
apply(*rule Collect-cong*)
apply(*rule conj-cong*, *rule refl*)
apply(*rule iff-exI*)
apply (*metis* (*no-types*, *hide-lams*) *Cons.hyps enabled reachableStep sched-equiv-preserved*)
done
qed

lemma *sources-eq*:
$[\![$*step-consistent*; *s* $\sim$ *sched* $\sim$ *t*; *reachable0 s*; *reachable0 t*$]\!]$ $\Longrightarrow$ *sources as s d* = *sources as t d*
by (*simp add*: *sources-eq0*)

lemma *same-sources-dom*:
$[\![$*s* $\approx$(*sources* (*a*#*as*) *s d*)$\approx$ *t*; (*the* (*domain s a*)) *x*; *x* $\in$ *sources as s*$'$ *d*;

$(s,s')\in step\ a]\!] \Longrightarrow (s \sim(the\ (domain\ s\ a))\sim\ t)$
apply(*simp add:ivpeq-def*)
apply(*erule bspec*)
apply(*subst sources-Cons*)
apply(*rule UnI2*)
apply(*blast*)
done

lemma *sources-step*:
$[\![reachable0\ s;\ (the\ (domain\ s\ a)) \setminus d]\!] \Longrightarrow sources\ [a]\ s\ d = \{d\}$
apply(*auto simp*: *sources-Cons sources-Nil enabled dest*: *enabled*)
by (*simp add*: *non-interference-def*)

lemma *sources-step2*:
$[\![reachable0\ s;\ (the\ (domain\ s\ a))\ \ d]\!] \Longrightarrow sources\ [a]\ s\ d = \{the\ (domain\ s\ a),d\}$
apply(*auto simp*: *sources-Cons sources-Nil enabled dest*: *enabled*)
done

lemma *sources-unwinding-step*:
$[\![s \approx(sources\ (a\#as)\ s\ d)\approx\ t;\ (s\sim sched\sim t);\ step\text{-}consistent;$
$(s,s')\in step\ a;\ (t,t')\in step\ a;\ reachable0\ s;\ reachable0\ t]\!] \Longrightarrow (s' \approx(sources\ as\ s'\ d)\approx\ t')$
apply(*clarsimp simp*: *ivpeq-def sources-Cons*)
apply(*subst* (*asm*) *step-consistent-def*)
apply(*drule-tac x=a* **in** *spec*)
apply(*drule-tac x=da* **in** *spec*)
apply(*drule-tac x=s* **in** *spec*)
apply(*drule-tac x=t* **in** *spec*)
using *UnionI* **by** *blast*

lemma *sources-eq-step*:
$[\![local\text{-}respect;\ step\text{-}consistent;(s,s')\in step\ a;$
$(the\ (domain\ s\ a)) \neq sched;\ reachable0\ s]\!] \Longrightarrow$
$(sources\ as\ s'\ d) = (sources\ as\ s\ d)$
using *reachableStep sched-equiv-preserved-left sources-eq0 vpeq-reflexive-lemma* **by** *blast*

lemma *sources-equiv-preserved-left*: $[\![local\text{-}respect;\ step\text{-}consistent;\ s\sim sched\sim t;$

$the\ (domain\ s\ a) \notin sources\ (a\#as)\ s\ d;\ s \approx sources\ (a\#as)\ s\ d\approx\ t;\ (s,s')\in step\ a;$
$(the\ (domain\ s\ a)) \neq sched;\ reachable0\ s;\ reachable0\ t]\!] \Longrightarrow (s' \approx sources\ as\ s'\ d\approx\ t)$
apply(*clarsimp simp*: *ivpeq-def*)
apply(*rename-tac v*)
apply(*case-tac* (*the* (*domain s a*)) *v*)
apply(*fastforce simp*: *sources-Cons*)
proof –

fix v :: $'d$
assume *a1*: *local-respect*
assume *a2*: *step-consistent*
assume *a3*: $s \sim sched \sim t$
assume *a4*: $\forall\, d{\in}sources\ (a\ \#\ as)\ s\ d.\ (s \sim d \sim t)$
assume *a5*: $(s,\ s') \in step\ a$
assume *a6*: *reachable0 s*
assume *a7*: *reachable0 t*
assume *a8*: $v \in sources\ as\ s'\ d$
assume *a9*: $\neg\ ((the\ (domain\ s\ a))\ \ v)$
obtain ss :: $'s \Rightarrow {}'e \Rightarrow {}'s$ **where**
f10: $\forall\, e.\ (t,\ ss\ t\ e) \in step\ e$
using *a7* **by** (*meson enabled*)
have $\forall\, e.\ domain\ s\ e = domain\ t\ e$
using *a3* **by** (*meson sched-vpeq*)
then have *f11*: $\forall\, d\ sa\ e.\ (t,\ sa) \notin step\ e \vee (t \sim d \sim sa) \vee ((the\ (domain\ s\ e))\ \ d)$
using *a7 a1 local-respect-def non-interference-def* **by** *force*
have $s' \sim v \sim (ss\ t\ a)$
using *f10 a8 a7 a6 a5 a4 a3 a2* **by** (*metis* (*no-types*) *ivpeq-def sources-unwinding-step*)
then show $s' \sim v \sim t$
using *f11 f10 a9* **by** (*meson vpeq-symmetric-lemma vpeq-transitive-lemma*)
qed

lemma *ipurge-eq′-helper*:
$[\![s \in ss;\ the\ (domain\ s\ a) \in sources\ (a\ \#\ as)\ s\ u;\ \forall\, s{\in}ts.\ the\ (domain\ s\ a) \notin sources\ (a\ \#\ as)\ s\ u;$
$(\forall\, s\ t.\ s \in ss \wedge t \in ts \longrightarrow (s \sim sched \sim t) \wedge reachable0\ s \wedge reachable0\ t);\ \ t \in ts;\ step\text{-}consistent]\!] \Longrightarrow$
False
apply(*cut-tac s=s* **and** *t=t* **and** *as=as* **and** *d=u* **in** *sources-eq*, *simp+*)
apply(*clarsimp simp*: *sources-Cons* | *safe*)+
apply(*rename-tac s′*)
apply(*drule-tac x=t* **in** *bspec*, *simp*)
apply *clarsimp*
apply(*cut-tac s=t* **in** *enabled*, *simp*)
apply(*erule exE*, *rename-tac t′*)
apply(*drule-tac x=sources as t′ u* **in** *spec*)
apply(*cut-tac s=s′* **and** *t=t′* **and** *d=u* **in** *sources-eq*, *simp+*)
apply(*fastforce elim*: *sched-equiv-preserved*)
apply(*fastforce intro*: *reachableStep*)
apply(*fastforce intro*: *reachableStep*)
apply(*fastforce simp*: *sched-vpeq*)
apply(*drule-tac x=t* **in** *bspec*, *simp*)
apply *clarsimp*
apply(*rename-tac v s′*)
apply(*drule-tac x=v* **in** *spec*, *erule impE*, *fastforce simp*: *sched-vpeq*)
apply(*cut-tac s=t* **in** *enabled*[**where** *a=a*], *simp*, *clarsimp*, *rename-tac t′*)

apply(*cut-tac s=s′* **and** *t=t′* **and** *d=u* **in** *sources-eq*, *simp+*)
apply(*fastforce elim*: *sched-equiv-preserved*)
apply(*fastforce intro*: *reachableStep*)
apply(*fastforce intro*: *reachableStep*)
apply(*fastforce simp*: *sched-vpeq*)
done

lemma *ipurge-eq′*:
(∀ *s t*. *s*∈*ss* ∧ *t*∈*ts* ⟶ (*s* ∼*sched*∼ *t*) ∧ *reachable0 s* ∧ *reachable0 t*) ∧
(∃ *s*. *s* ∈ *ss*) ∧ (∃ *t*. *t* ∈ *ts*) ∧ *step-consistent* ⟶ *ipurge as u ss* = *ipurge as u ts*
proof (*induct as arbitrary*: *ss ts*)
case *Nil* **show** *?case*
apply(*simp add*: *ipurge-def*)
done
next
case (*Cons a as*) **show** *?case*
apply(*clarsimp simp*: *sched-vpeq*)
apply(*intro conjI impI*)
apply(*rule Cons.hyps*[*rule-format*])
apply *clarsimp*
apply(*metis sched-equiv-preserved reachableStep enabled*)
apply *clarsimp*
apply(*drule ipurge-eq′-helper*, *simp+*)[*1*]
apply *clarsimp*
apply(*drule ipurge-eq′-helper*, (*simp add*: *vpeq-symmetric-lemma*)+)[*1*]
apply(*rule Cons.hyps*[*rule-format*], *auto*)
done
qed

lemma *ipurge-eq*:⟦*step-consistent*; *s* ∼ *sched* ∼ *t*; *reachable0 s* ∧ *reachable0 t*⟧
⟹ *ipurge as d* {*s*} = *ipurge as d* {*t*}
by (*simp add*: *ipurge-eq′*)

1.5 Inference framework of information flow security properties

theorem *nonintf-impl-weak*: *noninterference* ⟹ *weak-noninterference*
by (*metis noninterference-def observ-equiv-sym observ-equiv-trans reachable-s0 weak-noninterference-def*)

theorem *wk-nonintf-r-impl-wk-nonintf*: *weak-noninterference-r* ⟹ *weak-noninterference*
by (*simp add*: *reachable-s0 weak-noninterference-def weak-noninterference-r-def*)

theorem *nonintf-r-impl-noninterf*: *noninterference-r* ⟹ *noninterference*
using *noninterference-def noninterference-r-def reachable-s0* **by** *auto*

theorem *nonintf-r-impl-wk-nonintf-r*: *noninterference-r* $\Longrightarrow$ *weak-noninterference-r*
by (*metis noninterference-r-def observ-equiv-sym observ-equiv-trans weak-noninterference-r-def*)

lemma *noninf-impl-nonintf-r*: *noninfluence* $\Longrightarrow$ *noninterference-r*
by (*simp add*: *ivpeq-def noninfluence-def noninterference-r-def vpeq-reflexive-lemma*)

lemma *noninf-impl-nonlk*: *noninfluence* $\Longrightarrow$ *nonleakage*
using *noninterference-r-def nonleakage-def observ-equiv-sym observ-equiv-trans noninfluence-def noninf-impl-nonintf-r* **by** *blast*

lemma *wk-noninfl-impl-nonlk*: *weak-noninfluence* $\Longrightarrow$ *nonleakage*
by (*simp add*: *weak-noninfluence-def nonleakage-def*)

lemma *wk-noninfl-impl-wk-nonintf-r*: *weak-noninfluence* $\Longrightarrow$ *weak-noninterference-r*
using *ivpeq-def weak-noninfluence-def vpeq-reflexive-lemma weak-noninterference-r-def*
by *blast*

lemma *noninf-gen-impl-noninfl*: *noninfluence-gen* $\Longrightarrow$ *noninfluence*
by (*clarsimp simp*:*noninfluence-gen-def noninfluence-def*)

lemma *nonlk-imp-sc*: *nonleakage* $\Longrightarrow$ *step-consistent*
proof –
assume *p0*: *nonleakage*
then have *p1*[*rule-format*]: $\forall$ *as d s t*. *reachable0 s* $\wedge$ *reachable0 t* $\longrightarrow$ ($s \sim$ *sched* $\sim t$)
$\longrightarrow$ ($s \approx$ (*sources as s d*) $\approx t$) $\longrightarrow$ (s *as* $\cong$ t *as* @ d)
using *nonleakage-def* **by** *simp*

have $\forall$ *a d s t*. *reachable0 s* $\wedge$ *reachable0 t* $\longrightarrow$ ($s \sim d \sim t$) $\wedge$ ($s \sim$ *sched* $\sim t$) $\wedge$
(((*the* (*domain s a*)) d) $\longrightarrow$ ($s \sim$ (*the* (*domain s a*)) $\sim t$))
$\longrightarrow$
($\forall$ s' t'. $(s,s') \in$ *step a* $\wedge$ $(t,t') \in$ *step a* $\longrightarrow$ ($s' \sim d \sim t'$))
proof –
{
fix *a d s t*
assume *a0*: *reachable0 s* $\wedge$ *reachable0 t*
and *a1*: ($s \sim d \sim t$) $\wedge$ ($s \sim$ *sched* $\sim t$)
and *a2*: ((*the* (*domain s a*)) d) $\longrightarrow$ ($s \sim$ (*the* (*domain s a*)) $\sim t$)
have $\forall$ s' t'. $(s,s') \in$ *step a* $\wedge$ $(t,t') \in$ *step a* $\longrightarrow$ ($s' \sim d \sim t'$)
proof –
{
fix s' t'
assume *b0*: $(s,s') \in$ *step a* $\wedge$ $(t,t') \in$ *step a*
have $s' \sim d \sim t'$
proof(*cases* (*the* (*domain s a*)) d)
assume *c0*: (*the* (*domain s a*)) d

```
          with a2 have s ∼ (the (domain s a)) ∼ t by simp
          with a0 a1 c0 have s ≈ (sources [a] s d) ≈ t
            using sources-step2[of s a d] ivpeq-def[of s sources [a] s d t]
              insert-iff singletonD by auto
          then have s  [a] ≅ t  [a] @ d
            using p1[of s t [a] d] a0 a1 by simp
          with b0 show ?thesis
            by (simp add: observ-equivalence-def )
        next
          assume c0: ¬((the (domain s a))  d)
          with a0 a1 have s ≈ (sources [a] s d) ≈ t
            using sources-step[of s a d] ivpeq-def[of s sources [a] s d t]
              non-interference-def insert-iff singletonD by auto
          then have s  [a] ≅ t  [a] @ d
            using p1[of s t [a] d] a0 a1 by simp
          with b0 show ?thesis
            by (simp add: observ-equivalence-def )
        qed
      }
      then show ?thesis by auto
      qed
    }
    then show ?thesis by blast
    qed
  then show step-consistent using step-consistent-def by blast
qed


lemma sc-imp-nonlk: step-consistent =⇒ nonleakage
 proof −
  assume p0: step-consistent
  have ∀ d as s t. reachable0 s ∧ reachable0 t −→ (s ∼ sched ∼ t)
                 −→ (s ≈ (sources as s d) ≈ t) −→ (s  as ≅ t  as @ d)
   proof −
   {
    fix as
    have ∀ d s t. reachable0 s ∧ reachable0 t −→ (s ∼ sched ∼ t)
                 −→ (s ≈ (sources as s d) ≈ t) −→ (s  as ≅ t  as @ d)
     proof(induct as)
      case Nil show ?case
        by (simp add: ivpeq-def observ-equivalence-def sources-refl)
     next
      case (Cons b bs)
      assume a0: ∀ d s t. reachable0 s ∧ reachable0 t −→ (s ∼ sched ∼ t)
                     −→ (s ≈ sources bs s d ≈ t) −→ (s  bs ≅ t  bs @ d)
      show ?case
        proof −
        {
          fix d s t
```

assume *b0*: *reachable0 s* ∧ *reachable0 t*
and *b1*: *s* ∼ *sched* ∼ *t*
and *b2*: *s* ≈ *sources* (*b* # *bs*) *s d* ≈ *t*
then have *s* *b* # *bs* ≅ *t* *b* # *bs* @ *d*
using *exec-equiv-both sources-unwinding-step p0 a0*
by (*meson reachableStep SM-axioms sched-equiv-preserved*)
}
then show *?thesis* **by** *auto*
qed
qed
}
then show *?thesis* **by** *auto*
qed

then show *nonleakage* **using** *nonleakage-def* **by** *blast*
qed

theorem *sc-eq-nonlk*: *step-consistent* = *nonleakage*
using *nonlk-imp-sc sc-imp-nonlk* **by** *auto*

lemma *noninf-imp-lr*: *noninfluence* ⟹ *local-respect*
proof −
assume *p0*: *noninfluence*
then have *p1*[*rule-format*]: ∀ *d as s t* . *reachable0 s* ∧ *reachable0 t* ⟶ (*s* ≈ (*sources as s d*) ≈ *t*)
⟶ (*s* ∼ *sched* ∼ *t*) ⟶ (*s* *as* ≅ *t* (*ipurge as d* {*t*}) @ *d*)
using *noninfluence-def* **by** *simp*

have ∀ *a d s s′*. *reachable0 s* ⟶ ((*the* (*domain s a*)) \ *d*) ∧ (*s*,*s′*)∈*step a* ⟶ (*s* ∼ *d* ∼ *s′*)
proof −
{
fix *a d s s′*
assume *a0*: *reachable0 s*
and *a1*: ((*the* (*domain s a*)) \ *d*) ∧ (*s*,*s′*)∈*step a*
then have *a2*: *the* (*domain s a*) ≠ *d* **using** *non-interference-def interf-reflexive* **by** *auto*
from *a0 a1 p1*[*of s s* [*a*] *d*] **have** *a3*: *s* [*a*] ≅ *s* (*ipurge* [*a*] *d* {*s*}) @ *d*
using *ivpeq-def vpeq-reflexive-lemma* **by** *blast*
from *a0 a1 a2* **have** *ipurge* [*a*] *d* {*s*} = []
using *sources-step SM-enabled-axioms non-interference-def* **by** *fastforce*
with *a1 a3* **have** *s* ∼ *d* ∼ *s′*
by (*metis IdI R-O-Id observ-equiv-sym observ-equivalence-def run-Cons run-Nil*)
}
then show *?thesis* **by** *auto*
qed
then show *local-respect* **using** *local-respect-def* **by** *blast*

```
    qed

  lemma noninf-imp-sc: noninfluence ⟹ step-consistent
    by (simp add: nonlk-imp-sc noninf-impl-nonlk)

 theorem UnwindingTheorem : ⟦step-consistent; local-respect⟧ ⟹ noninfluence-gen
  proof -
    assume p1:step-consistent
    assume p2:local-respect
    {
      fix as d
      have ∀ s ts. reachable0 s ∧ (∀ t∈ts. reachable0 t)
                 ⟶ (∀ t∈ts. (s ≈ (sources as s d) ≈ t))
                 ⟶ (∀ t∈ts. (s ∼ sched ∼ t))
                 ⟶ (∀ t∈ts. (s  as ≅ t  (ipurge as d ts) @ d))
        proof(induct as)
       case Nil show ?case by (simp add: execute-def ivpeq-def observ-equivalence-def
sources-refl)
        next
          case (Cons b bs)
          assume a0: ∀ s ts. reachable0 s ∧ (∀ t∈ts. reachable0 t)
                     ⟶ (∀ t∈ts. (s ≈ (sources bs s d) ≈ t))
                     ⟶ (∀ t∈ts. (s ∼ sched ∼ t))
                     ⟶ (∀ t∈ts. (s  bs ≅ t  (ipurge bs d ts) @ d))
          show ?case
            proof -
            {
              fix s ts
              assume b0: reachable0 s ∧ (∀ t∈ts. reachable0 t)
                and  b1: ∀ t∈ts. (s ≈ (sources (b # bs) s d) ≈ t)
                and  b2: ∀ t∈ts. (s ∼ sched ∼ t)
              {
                fix t
                assume c0: t∈ts
                have c1: sources (b#bs) s d = sources (b#bs) t d
                  by (simp add: b0 b2 c0 p1 sources-eq0)
                have c2: domain s b = domain t b
                  by (simp add: b2 c0 sched-vpeq)
                have s  b # bs ≅ t  ipurge (b # bs) d ts @ d
                  proof(cases the (domain s b) ∈ sources (b#bs) s d)
                    assume d0:the (domain s b) ∈ sources (b#bs) s d
                     have d1: ipurge (b # bs) d ts = b # ipurge bs d (⋃ s∈ts. {s'.
(s,s') ∈ step b})
                      using c0 c1 c2 d0 by auto
                    let ?ts' = ⋃ s∈ts. {s'. (s,s') ∈ step b}
                    let ?bs' = ipurge bs d (⋃ s∈ts. {s'. (s,s') ∈ step b})
                    {
                      fix s' t'
                      assume e0: (s,s')∈ run (b#bs) ∧ (t,t')∈ run (b # ?bs')
```

then have *e1*: $\exists s'' t''$. $(s,s'')\in$*step* $b \wedge (s'',s')\in$*run* *bs* $\wedge$ $(t,t'')\in$*step* $b \wedge (t'',t')\in$*run* *?bs'*
using *relcompEpair run-Cons* **by** *auto*
then obtain s'' **and** t'' **where** *e2*: $(s,s'')\in$*step* $b \wedge (s'',s')\in$*run* *bs* $\wedge$ $(t,t'')\in$*step* $b \wedge (t'',t')\in$*run* *?bs'*
by *auto*
have $\forall t\in$*?ts'*. *reachable0* t **using** *b0 reachableStep* **by** *auto*
moreover
have $\forall t\in$*?ts'*. $(s'' \approx$ (*sources bs* s'' d) $\approx t)$
using *b0 b1 b2 e2 p1 sources-unwinding-step* **by** *fastforce*
moreover
have $\forall t\in$*?ts'*. $(s'' \sim$ *sched* $\sim t)$
using *SM-enabled.sched-equiv-preserved SM-enabled-axioms b0 b2 e2 p1* **by** *fastforce*
ultimately
have *e3*: $\forall t\in$*?ts'*. $(s''$ *bs* $\cong t$ (*ipurge bs d ?ts'*) @ d) **using** *a0*
by (*metis b0 e2 reachableStep*)
then have $s' \sim d \sim t'$
using *UN-iff c0 e2 mem-Collect-eq observ-equivalence-def* **by** *auto*
}
then have $\forall s'\ t'$. $((s,s')\in$ *run* (*b#bs*) $\wedge$ $(t,t')\in$ *run* (*b # ?bs'*)) $\longrightarrow(s' \sim d \sim t')$
by *simp*
with *d1* **show** *?thesis* **by** (*simp add:observ-equivalence-def*)
next
assume *d0*:$\neg$(*the* (*domain s b*) $\in$ *sources* (*b#bs*) *s d*)
have *d1*: *ipurge* (*b # bs*) *d ts* = *ipurge bs d ts*
using *b0 b2 d0 p1 sched-vpeq sources-eq* **by** *auto*
let *?bs'* = *ipurge bs d ts*
{
fix $s'\ t'$
assume *e0*: $(s,s')\in$ *run* (*b#bs*) $\wedge$ $(t,t')\in$ *run* *?bs'*
then have *e1*: $\exists s'' t''$. $(s,s'')\in$*step* $b \wedge (s'',s')\in$*run* *bs*
using *relcompEpair run-Cons* **by** *auto*
then obtain s'' **where** *e2*: $(s,s'')\in$*step* $b \wedge (s'',s')\in$*run* *bs*
by *auto*
have $\forall t\in$*ts*. $(s'' \approx$ (*sources bs* s'' d) $\approx t)$
using *b0 b1 b2 d0 e2 p1 p2 scheduler-in-sources-Cons sources-equiv-preserved-left* **by** *blast*
moreover
have $\forall t\in$*ts*. $(s'' \sim$ *sched* $\sim t)$
using *b0 b2 d0 e2 p2 sched-equiv-preserved-left scheduler-in-sources-Cons* **by** *blast*
ultimately
have *e3*: $\forall t\in$*ts*. $(s''$ *bs* $\cong t$ (*ipurge bs d ts*) @ d) **using** *a0*
by (*metis b0 e2 reachableStep*)
then have $s' \sim d \sim t'$
using *c0 e0 e2 observ-equivalence-def* **by** *blast*

}
then have $\forall s'\ t'.\ ((s,s')\in$ *run* $(b\#bs) \wedge (t,t')\in$ *run* *?bs'*$) \longrightarrow (s'$
$\sim d \sim t')$
by *simp*
with *d1* **show** *?thesis* **by** (*simp add*:*observ-equivalence-def*)
qed

}

}
then show *?thesis* **by** *auto*
qed
qed
}
then show *?thesis* **by** (*simp add*:*noninfluence-gen-def*)
qed

theorem *UnwindingTheorem1* : ⟦*weak-step-consistent*; *local-respect*⟧ $\Longrightarrow$ *noninfluence-gen*
by (*simp add*: *UnwindingTheorem weak-with-step-cons*)

theorem *noninf-eq-noninf-gen*: *noninfluence* = *noninfluence-gen*
using *UnwindingTheorem noninf-imp-lr noninf-imp-sc noninf-gen-impl-noninfl*
by *blast*

theorem *uc-eq-noninf* : (*step-consistent* $\wedge$ *local-respect*) = *noninfluence*
using *UnwindingTheorem1 step-cons-impl-weak noninf-eq-noninf-gen*
noninf-imp-lr noninf-imp-sc **by** *blast*

theorem *noninf-impl-weak*:*noninfluence* $\Longrightarrow$ *weak-noninfluence*
by (*smt observ-equiv-sym observ-equiv-trans ipurge-eq weak-noninfluence-def*
noninterference-r-def noninf-imp-sc noninfluence-def noninf-impl-nonintf-r)

lemma *wk-nonintf-r-and-nonlk-impl-noninfl*: ⟦*weak-noninterference-r*; *nonleak-age*⟧ $\Longrightarrow$ *weak-noninfluence*
proof −
assume *p0*: *weak-noninterference-r*
and *p1*: *nonleakage*
then have *a0*: $\forall$ *d as bs s*. *reachable0 s* $\wedge$ *ipurge as d* $\{s\}$ = *ipurge bs d* $\{s\}$
$\longrightarrow (s \quad as \cong s \quad bs \ @ \ d)$
by (*simp add*:*weak-noninterference-r-def*)
from *p1* **have** *a1*: $\forall$ *d as s t*. *reachable0 s* $\wedge$ *reachable0 t* $\wedge$ $(s \sim sched \sim t)$
$\wedge$ $(s \approx (sources\ as\ s\ d) \approx t) \longrightarrow (s \quad as \cong t \quad as \ @ \ d)$
by (*simp add*:*nonleakage-def*)

then have $\forall$ *d as bs s t* . *reachable0 s* $\wedge$ *reachable0 t* $\wedge$ $(s \approx (sources\ as\ s$
$d) \approx t)$
$\wedge$ $(s \sim sched \sim t)$ $\wedge$ *ipurge as d* $\{s\}$ = *ipurge bs d*
$\{s\}$

$\longrightarrow (s \ \ as \cong t \ \ bs$ @ $d)$

proof –

{

fix *d as bs s t*

assume *b0*: *reachable0 s* $\wedge$ *reachable0 t* $\wedge$ $(s \approx (sources\ as\ s\ d) \approx t)$
$\wedge$ $(s \sim sched \sim t)$ $\wedge$ *ipurge as d* $\{s\}$ = *ipurge bs d* $\{s\}$

with *a1* **have** *b1*: $s \ \ as \cong t \ \ as$ @ d **by** *simp*

from *b0* **have** *b2*: *ipurge as d* $\{s\}$ = *ipurge as d* $\{t\}$
using *ipurge-eq nonlk-imp-sc p1* **by** *blast*

from *b0* **have** *b3*: *ipurge bs d* $\{s\}$ = *ipurge bs d* $\{t\}$
using *ipurge-eq nonlk-imp-sc p1* **by** *blast*

from *a0 b0 b2 b3* **have** *b4*: $s \ \ as \cong s \ \ bs$ @ d **by** *simp*

from *a0 b0 b2 b3* **have** *b5*: $t \ \ as \cong t \ \ bs$ @ d **by** *simp*

from *b1 b4 b5* **have** $s \ \ as \cong t \ \ bs$ @ d
using *b0 observ-equiv-trans* **by** *blast*

}

then show *?thesis* **by** *auto*

qed

then show *?thesis* **by** (*simp add*:*weak-noninfluence-def*)

qed

lemma *nonintf-r-and-nonlk-impl-noninfl*: ⟦*noninterference-r*; *nonleakage*⟧ $\Longrightarrow$ *noninfluence*

proof –

assume *p0*: *noninterference-r*
and *p1*: *nonleakage*

then have *a0*: $\forall$ *d as s*. *reachable0 s* $\longrightarrow$ $(s \ \ as \cong s \ \ (ipurge\ as\ d\ \{s\})$ @ $d)$
by (*simp add*:*noninterference-r-def*)

from *p1* **have** *a1*: $\forall$ *d as s t*. *reachable0 s* $\wedge$ *reachable0 t* $\wedge$ $(s \sim sched \sim t)$
$\wedge$ $(s \approx (sources\ as\ s\ d) \approx t)$ $\longrightarrow$ $(s \ \ as \cong t \ \ as$ @ $d)$
by (*simp add*:*nonleakage-def*)

then have $\forall$ *d as s t* . *reachable0 s* $\wedge$ *reachable0 t* $\wedge$ $(s \approx (sources\ as\ s\ d) \approx t)$
$\wedge$ $(s \sim sched \sim t)$ $\longrightarrow$ $(s \ \ as \cong t \ \ (ipurge\ as\ d\ \{t\})$ @ $d)$

proof –

{

fix *d as bs s t*

assume *b0*: *reachable0 s* $\wedge$ *reachable0 t* $\wedge$ $(s \approx (sources\ as\ s\ d) \approx t)$
$\wedge$ $(s \sim sched \sim t)$

with *a1* **have** *b1*: $s \ \ as \cong t \ \ as$ @ d **by** *simp*

from *b0 a0* **have** *b2*: $s \ \ as \cong s \ \ (ipurge\ as\ d\ \{s\})$ @ d **by** *simp*

from *b0 a0* **have** *b3*: $t \ \ as \cong t \ \ (ipurge\ as\ d\ \{t\})$ @ d **by** *simp*

from *b1 b2 b3* **have** $s \ \ as \cong t \ \ (ipurge\ as\ d\ \{t\})$ @ d
using *b0 observ-equiv-trans* **by** *blast*

}

then show *?thesis* **by** *auto*

qed
then show *?thesis* **by** (*simp add*:*noninfluence-def*)
qed

lemma *noninfl-impl-noninfl2*: *weak-noninfluence* $\Longrightarrow$ *weak-noninfluence2*
using *ipurge-eq wk-noninfl-impl-nonlk weak-noninfluence2-def weak-noninfluence-def nonlk-imp-sc* **by** *fastforce*

lemma *noninf2-imp-lr*: *weak-noninfluence2* $\Longrightarrow$ *local-respect*
proof –
assume *p0*: *weak-noninfluence2*
then have *p1*[*rule-format*]: $\forall$ *d as bs s t* . *reachable0 s* $\wedge$ *reachable0 t* $\wedge$ (*s* $\approx$ (*sources as s d*) $\approx$ *t*)
$\wedge$ (*s* $\sim$ *sched* $\sim$ *t*) $\wedge$ *ipurge as d* {*s*} = *ipurge bs d* {*t*}
$\longrightarrow$ (*s as* $\cong$ *t bs* @ *d*)
using *weak-noninfluence2-def* **by** *simp*

have $\forall$ *a d s s*′. *reachable0 s* $\longrightarrow$ ((*the* (*domain s a*)) $\setminus$ *d*) $\wedge$ (*s*,*s*′)$\in$*step a* $\longrightarrow$ (*s* $\sim$ *d* $\sim$ *s*′)
proof –
{
fix *a d s s*′
assume *a0*: *reachable0 s*
and *a1*: ((*the* (*domain s a*)) $\setminus$ *d*) $\wedge$ (*s*,*s*′)$\in$*step a*
then have *a2*: *the* (*domain s a*) $\neq$ *d* **using** *non-interference-def interf-reflexive* **by** *auto*
from *a0 a1 a2* **have** *ipurge* [*a*] *d* {*s*} = *ipurge* [] *d* {*s*}
using *sources-step SM-enabled-axioms non-interference-def* **by** *fastforce*
with *a0* **have** *s* [*a*] $\cong$ *s* [] @ *d*
using *p1*[*of s s* [*a*] *d* []] *ivpeq-def vpeq-reflexive-lemma* **by** *blast*
with *a1* **have** *s* $\sim$ *d* $\sim$ *s*′
by (*metis IdI R-O-Id observ-equiv-sym observ-equivalence-def run-Cons run-Nil*)
}
then show *?thesis* **by** *auto*
qed
then show *local-respect* **using** *local-respect-def* **by** *blast*
qed

lemma *noninf2-imp-sc*: *weak-noninfluence2* $\Longrightarrow$ *step-consistent*
proof –
assume *p0*: *weak-noninfluence2*
then have *p1*[*rule-format*]: $\forall$ *d as bs s t* . *reachable0 s* $\wedge$ *reachable0 t* $\wedge$ (*s* $\approx$ (*sources as s d*) $\approx$ *t*)
$\wedge$ (*s* $\sim$ *sched* $\sim$ *t*) $\wedge$ *ipurge as d* {*s*} = *ipurge bs d* {*t*}
$\longrightarrow$ (*s as* $\cong$ *t bs* @ *d*)
using *weak-noninfluence2-def* **by** *simp*

```
have ∀ a d s t. reachable0 s ∧ reachable0 t ∧ (s ∼ d ∼ t) ∧ (s ∼ sched ∼ t) ∧
                (((the (domain s a)) ⇝ d) ⟶ (s ∼ (the (domain s a)) ∼ t)) ⟶
                (∀ s' t'. (s,s') ∈ step a ∧ (t,t') ∈ step a ⟶ (s' ∼d∼ t'))
  proof -
  {
    fix a d s t
    assume a0: reachable0 s ∧ reachable0 t
      and a1: (s ∼ d ∼ t) ∧ (s ∼ sched ∼ t)
      and a2: ((the (domain s a)) ⇝ d) ⟶ (s ∼ (the (domain s a)) ∼ t)
    then have a3: domain s a = domain t a by (simp add: sched-vpeq)

    have ∀ s' t'. (s,s') ∈ step a ∧ (t,t') ∈ step a ⟶ (s' ∼d∼ t')
      proof -
      {
        fix s' t'
        assume b0: (s,s') ∈ step a ∧ (t,t') ∈ step a
        have s' ∼d∼ t'
          proof(cases (the (domain s a)) ⇝ d)
            assume c0: (the (domain s a)) ⇝ d
            with a2 have c1: s ∼ (the (domain s a)) ∼ t by simp
            with a0 a1 c0 have c2: s ≈ (sources [a] s d) ≈ t
              using sources-step2[of s a d] ivpeq-def[of s sources [a] s d t]
                insert-iff singletonD by auto
            from a0 c0 a3 have c4: ipurge [a] d {s} = ipurge [a] d {t}
              using sources-step2[of s a d] sources-step2[of t a d]
                ipurge-Cons[of a [] d {s}] ipurge-Cons[of a [] d {t}]
                ipurge-Nil insertI1 by auto
            then have s ⊳ [a] ≅ t ⊳ [a] @ d
              using p1[of s t [a] d] a0 a1 c2 by blast
            with b0 show ?thesis
              by (simp add: observ-equivalence-def )
          next
            assume c0: ¬((the (domain s a)) ⇝ d)
             then have c1: the (domain s a) ≠ d using non-interference-def interf-reflexive by auto
            from c0 a0 a1 have c2: s ≈ (sources [a] s d) ≈ t
              using sources-step[of s a d] ivpeq-def[of s sources [a] s d t]
                non-interference-def insert-iff singletonD by auto
            from a0 c0 c1 a3 have c4: ipurge [a] d {s} = ipurge [a] d {t}
              using sources-step[of s a d] sources-step[of t a d]
                ipurge-Cons[of a [] d {s}] ipurge-Cons[of a [] d {t}]
                ipurge-Nil non-interference-def singletonD by auto
            then have s ⊳ [a] ≅ t ⊳ [a] @ d
              using p1[of s t [a] d] a0 a1 c2 by blast
            with b0 show ?thesis
              by (simp add: observ-equivalence-def )
```

qed
}
then show *?thesis* **by** *auto*
qed
}
then show *?thesis* **by** *blast*
qed
then show *step-consistent* **using** *step-consistent-def* **by** *blast*
qed

theorem *noninfl-eq-noninfl2*: *weak-noninfluence* = *weak-noninfluence2*
using *noninf2-imp-lr noninf2-imp-sc noninf-impl-weak noninfl-impl-noninfl2 uc-eq-noninf* **by** *blast*

theorem *nonintf-r-and-nonlk-eq-strnoninfl*: (*noninterference-r* ∧ *nonleakage*) = *noninfluence*
using *nonintf-r-and-nonlk-impl-noninfl noninf-impl-nonintf-r noninf-impl-nonlk* **by** *blast*

theorem *wk-nonintf-r-and-nonlk-eq-noninfl*: (*weak-noninterference-r* ∧ *nonleakage*) = *weak-noninfluence*
using *wk-noninfl-impl-nonlk wk-noninfl-impl-wk-nonintf-r wk-nonintf-r-and-nonlk-impl-noninfl* **by** *blast*

end
end

2 Index-based manipulation of lists

theory *List-Index* **imports** *Main* **begin**

This theory collects functions for index-based manipulation of lists.

2.1 Findiindexng an index

This subsection defines three functions for finding the index of items in a list:

find-index P xs finds the index of the first element in *xs* that satisfies *P*.

index xs x finds the index of the first occurrence of *x* in *xs*.

last-index xs x finds the index of the last occurrence of *x* in *xs*.

All functions return *length xs* if *xs* does not contain a suitable element.

The argument order of *find-index* follows the function of the same name in the Haskell standard library. For *index* (and *last-index*) the order is intentionally reversed: *index* maps lists to a mapping from elements to their indices, almost the inverse of function *nth*.

fun *find-index* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow nat$ **where**
find-index - $[] = 0$ |
find-index $P\ (x\#xs) = (if\ P\ x\ then\ 0\ else\ find\text{-}index\ P\ xs + 1)$

definition *index* :: $'a\ list \Rightarrow 'a \Rightarrow nat$ **where**
index $xs = (\lambda a.\ find\text{-}index\ (\lambda x.\ x{=}a)\ xs)$

definition *last-index* :: $'a\ list \Rightarrow 'a \Rightarrow nat$ **where**
last-index $xs\ x =$
$(let\ i = index\ (rev\ xs)\ x;\ n = size\ xs$
$\ in\ if\ i = n\ then\ i\ else\ n - (i{+}1))$

lemma *find-index-le-size*: *find-index P xs <= size xs*
by(*induct xs*) *simp-all*

lemma *index-le-size*: *index xs x <= size xs*
by(*simp add*: *index-def find-index-le-size*)

lemma *last-index-le-size*: *last-index xs x <= size xs*
by(*simp add*: *last-index-def Let-def index-le-size*)

lemma *index-Nil*[*simp*]: *index* $[]\ a = 0$
by(*simp add*: *index-def*)

lemma *index-Cons*[*simp*]: *index* $(x\#xs)\ a = (if\ x{=}a\ then\ 0\ else\ index\ xs\ a + 1)$
by(*simp add*: *index-def*)

lemma *index-append*: *index* $(xs\ @\ ys)\ x =$
$(if\ x : set\ xs\ then\ index\ xs\ x\ else\ size\ xs + index\ ys\ x)$
by (*induct xs*) *simp-all*

lemma *index-conv-size-if-notin*[*simp*]: $x \notin set\ xs \Longrightarrow index\ xs\ x = size\ xs$
by (*induct xs*) *auto*

lemma *find-index-eq-size-conv*:
$size\ xs = n \Longrightarrow (find\text{-}index\ P\ xs = n) = (\forall\, x \in set\ xs.\ {\sim}\ P\ x)$
by(*induct xs arbitrary*: *n*) *auto*

lemma *size-eq-find-index-conv*:
$size\ xs = n \Longrightarrow (n = find\text{-}index\ P\ xs) = (\forall\, x \in set\ xs.\ {\sim}\ P\ x)$
by(*metis find-index-eq-size-conv*)

lemma *index-size-conv*: $size\ xs = n \Longrightarrow (index\ xs\ x = n) = (x \notin set\ xs)$
by(*auto simp*: *index-def find-index-eq-size-conv*)

lemma *size-index-conv*: $size\ xs = n \Longrightarrow (n = index\ xs\ x) = (x \notin set\ xs)$
by (*metis index-size-conv*)

lemma *last-index-size-conv*:
  size xs = *n* $\Longrightarrow$ (*last-index xs x* = *n*) = (*x* $\notin$ *set xs*)
apply(*auto simp*: *last-index-def index-size-conv*)
apply(*drule length-pos-if-in-set*)
apply *arith*
done

lemma *size-last-index-conv*:
  size xs = *n* $\Longrightarrow$ (*n* = *last-index xs x*) = (*x* $\notin$ *set xs*)
by (*metis last-index-size-conv*)

lemma *find-index-less-size-conv*:
  (*find-index P xs* < *length xs*) = ($\exists$ *x* $\in$ *set xs*. *P x*)
by (*induct xs*) *auto*

lemma *index-less-size-conv*:
  (*index xs x* < *size xs*) = (*x* $\in$ *set xs*)
by(*auto simp*: *index-def find-index-less-size-conv*)

lemma *last-index-less-size-conv*:
  (*last-index xs x* < *size xs*) = (*x* : *set xs*)
by(*simp add*: *last-index-def Let-def index-size-conv length-pos-if-in-set*
  del:*length-greater-0-conv*)

lemma *index-less*[*simp*]:
  x : *set xs* $\Longrightarrow$ *size xs* <= *n* $\Longrightarrow$ *index xs x* < *n*
apply(*induct xs*) **apply** *auto*
apply (*metis index-less-size-conv less-eq-Suc-le less-trans-Suc*)
done

lemma *last-index-less*[*simp*]:
  x : *set xs* $\Longrightarrow$ *size xs* <= *n* $\Longrightarrow$ *last-index xs x* < *n*
by(*simp add*: *last-index-less-size-conv*[*symmetric*])

lemma *last-index-Cons*: *last-index* (*x*#*xs*) *y* =
  (*if x*=*y then*
    if x $\in$ *set xs then last-index xs y* + *1 else 0*
  else last-index xs y + *1*)
using *index-le-size*[*of rev xs y*]
apply(*auto simp add*: *last-index-def index-append Let-def*)
apply(*simp add*: *index-size-conv*)
done

lemma *last-index-append*: *last-index* (*xs* @ *ys*) *x* =
  (*if x* : *set ys then size xs* + *last-index ys x*
  else if x : *set xs then last-index xs x else size xs* + *size ys*)
by (*induct xs*) (*simp-all add*: *last-index-Cons last-index-size-conv*)

lemma *last-index-Snoc*[*simp*]:
  last-index (*xs* @ [*x*]) *y* =
  (*if x=y then size xs*
   else if y : set xs then last-index xs y else size xs + 1)
by(*simp add*: *last-index-append last-index-Cons*)

lemma *nth-find-index*: *find-index P xs* < *size xs* $\Longrightarrow$ *P*(*xs* ! *find-index P xs*)
by (*induct xs*) *auto*

lemma *nth-index*[*simp*]: *x* $\in$ *set xs* $\Longrightarrow$ *xs* ! *index xs x* = *x*
by (*induct xs*) *auto*

lemma *nth-last-index*[*simp*]: *x* $\in$ *set xs* $\Longrightarrow$ *xs* ! *last-index xs x* = *x*
by(*simp add*:*last-index-def index-size-conv Let-def rev-nth*[*symmetric*])

lemma *index-rev*: $[\![$ *distinct xs*; *x* $\in$ *set xs* $]\!]$ $\Longrightarrow$
  index (*rev xs*) *x* = *length xs* − *index xs x* − *1*
by (*induct xs*) (*auto simp*: *index-append*)

lemma *index-nth-id*:
  $[\![$ *distinct xs*; *n* < *length xs* $]\!]$ $\Longrightarrow$ *index xs* (*xs* ! *n*) = *n*
by (*metis in-set-conv-nth index-less-size-conv nth-eq-iff-index-eq nth-index*)

lemma *index-upt*[*simp*]: *m* $\leq$ *i* $\Longrightarrow$ *i* < *n* $\Longrightarrow$ *index* [*m*..<*n*] *i* = *i*−*m*
by (*induction n*) (*auto simp add*: *index-append*)

lemma *index-eq-index-conv*[*simp*]: *x* $\in$ *set xs* $\vee$ *y* $\in$ *set xs* $\Longrightarrow$
  (*index xs x* = *index xs y*) = (*x* = *y*)
by (*induct xs*) *auto*

lemma *last-index-eq-index-conv*[*simp*]: *x* $\in$ *set xs* $\vee$ *y* $\in$ *set xs* $\Longrightarrow$
  (*last-index xs x* = *last-index xs y*) = (*x* = *y*)
by (*induct xs*) (*auto simp*:*last-index-Cons*)

lemma *inj-on-index*: *inj-on* (*index xs*) (*set xs*)
by (*simp add*:*inj-on-def*)

lemma *inj-on-index2*: *I* $\subseteq$ *set xs* $\Longrightarrow$ *inj-on* (*index xs*) *I*
by (*rule inj-onI*) *auto*

lemma *inj-on-last-index*: *inj-on* (*last-index xs*) (*set xs*)
by (*simp add*:*inj-on-def*)

lemma *index-conv-takeWhile*: *index xs x* = *size*(*takeWhile* (λ*y*. *x*$\neq$*y*) *xs*)
by(*induct xs*) *auto*

lemma *index-take*: *index xs x* >= *i* $\Longrightarrow$ *x* $\notin$ *set*(*take i xs*)
apply(*subst* (*asm*) *index-conv-takeWhile*)
apply(*subgoal-tac set*(*take i xs*) <= *set*(*takeWhile* (($\neq$) *x*) *xs*))

apply(*blast dest*: *set-takeWhileD*)
apply(*metis set-take-subset-set-take takeWhile-eq-take*)
done

lemma *last-index-drop*:
$\textit{last-index xs x} < i \Longrightarrow x \notin \textit{set}(\textit{drop i xs})$
apply(*subgoal-tac* $\textit{set}(\textit{drop i xs}) = \textit{set}(\textit{take} \ (\textit{size xs} - i) \ (\textit{rev xs}))$)
apply(*simp add*: *last-index-def index-take Let-def split*:*if-split-asm*)
apply (*metis rev-drop set-rev*)
done

lemma *set-take-if-index*: **assumes** $\textit{index xs x} < i$ **and** $i \leq \textit{length xs}$
shows $x \in \textit{set} \ (\textit{take i xs})$
proof −
have $\textit{index} \ (\textit{take i xs} \ @ \ \textit{drop i xs}) \ x < i$
using *append-take-drop-id*[*of i xs*] *assms*(*1*) **by** *simp*
thus *?thesis* **using** *assms*(*2*)
by(*simp add*:*index-append del*:*append-take-drop-id split*: *if-splits*)
qed

lemma *index-take-if-index*:
assumes $\textit{index xs x} \leq n$ **shows** $\textit{index} \ (\textit{take n xs}) \ x = \textit{index xs x}$
proof *cases*
assume $x : \textit{set}(\textit{take n xs})$ **with** *assms* **show** *?thesis*
by (*metis append-take-drop-id index-append*)
next
assume $x \notin \textit{set}(\textit{take n xs})$ **with** *assms* **show** *?thesis*
by (*metis order-le-less set-take-if-index le-cases length-take min-def size-index-conv take-all*)
qed

lemma *index-take-if-set*:
$x : \textit{set}(\textit{take n xs}) \Longrightarrow \textit{index} \ (\textit{take n xs}) \ x = \textit{index xs x}$
by (*metis index-take index-take-if-index linear*)

lemma *index-last*[*simp*]:
$xs \neq [] \Longrightarrow \textit{distinct xs} \Longrightarrow \textit{index xs} \ (\textit{last xs}) = \textit{length xs} - 1$
by (*induction xs*) *auto*

lemma *index-update-if-diff2*:
$n < \textit{length xs} \Longrightarrow x \neq xs!n \Longrightarrow x \neq y \Longrightarrow \textit{index} \ (xs[n := y]) \ x = \textit{index xs x}$
by(*subst* (*2*) *id-take-nth-drop*[*of n xs*])
(*auto simp*: *upd-conv-take-nth-drop index-append min-def*)

lemma *set-drop-if-index*: $\textit{distinct xs} \Longrightarrow \textit{index xs x} < i \Longrightarrow x \notin \textit{set}(\textit{drop i xs})$
by (*metis in-set-dropD index-nth-id last-index-drop last-index-less-size-conv nth-last-index*)

lemma *index-swap-if-distinct*: **assumes** *distinct xs* $i < \textit{size xs}$ $j < \textit{size xs}$
shows $\textit{index} \ (xs[i := xs!j, \ j := xs!i]) \ x =$

($if\ x = xs!i\ then\ j\ else\ if\ x = xs!j\ then\ i\ else\ index\ xs\ x$)
proof−
have $distinct(xs[i := xs!j,\ j := xs!i])$ **using** *assms* **by** *simp*
with *assms* **show** *?thesis*
apply (*auto simp*: *swap-def simp del*: *distinct-swap*)
apply (*metis index-nth-id list-update-same-conv*)
apply (*metis* (*erased*, *hide-lams*) *index-nth-id length-list-update list-update-swap nth-list-update-eq*)
apply (*metis index-nth-id length-list-update nth-list-update-eq*)
by (*metis index-update-if-diff2 length-list-update nth-list-update*)
qed

lemma *bij-betw-index*:
$distinct\ xs \Longrightarrow X = set\ xs \Longrightarrow l = size\ xs \Longrightarrow bij\text{-}betw\ (index\ xs)\ X\ \{0..<l\}$
apply *simp*
apply(*rule bij-betw-imageI*[*OF inj-on-index*])
by (*auto simp*: *image-def*) (*metis index-nth-id nth-mem*)

lemma *index-image*: $distinct\ xs \Longrightarrow set\ xs = X \Longrightarrow index\ xs\ `\ X = \{0..<size\ xs\}$
by (*simp add*: *bij-betw-imp-surj-on bij-betw-index*)

lemma *index-map-inj-on*:
$[\![\ inj\text{-}on\ f\ S;\ y \in S;\ set\ xs \subseteq S\ ]\!] \Longrightarrow index\ (map\ f\ xs)\ (f\ y) = index\ xs\ y$
by (*induct xs*) (*auto simp*: *inj-on-eq-iff*)

lemma *index-map-inj*: $inj\ f \Longrightarrow index\ (map\ f\ xs)\ (f\ y) = index\ xs\ y$
by (*simp add*: *index-map-inj-on*[**where** S=*UNIV*])

lemma *myListIndexAux1*: $find\text{-}index\ P\ L = a\ \wedge\ a = length\ L \Longrightarrow (\forall j<a.\ \neg\ P\ (L!j))$
using *find-index-eq-size-conv nth-mem* **by** *fastforce*

lemma *myListIndexAux2*: $find\text{-}index\ P\ L = a\ \Longrightarrow find\text{-}index\ P\ (l\#L) = a+1 \vee find\text{-}index\ P\ (l\#L) = 0$
by (*meson find-index.simps*(*2*))

lemma *myListIndexAux3*: $find\text{-}index\ P\ L = a \wedge a \neq length\ L \Longrightarrow a < length\ L \wedge (\forall j<a.\ \neg\ P\ (L!j)) \wedge P\ (L\ !\ a)$
apply (*induction L arbitrary*: *a*)
apply *fastforce*
using *myListIndexAux2*
by (*smt One-nat-def Suc-less-eq Suc-pred add.right-neutral add-Suc-right find-index.simps*(*2*) *list.size*(*4*) *nat-neq-iff not-less-zero nth-Cons' nth-find-index*)

lemma *myListIndexAux4*: $a < length\ L \wedge (\forall j<a.\ \neg\ P\ (L!j)) \wedge P\ (L\ !\ a) \Longrightarrow find\text{-}index\ P\ L = a \wedge a \neq length\ L$
by (*metis myListIndexAux3 nat-neq-iff nth-mem size-eq-find-index-conv*)

lemma *myListIndexAux5*: $find\text{-}index\ P\ L = a \Longrightarrow a \leq length\ L$

using *find-index-le-size* **by** *blast*

2.2 Map with index

primrec *map-index'* :: *nat* ⇒ (*nat* ⇒ *'a* ⇒ *'b*) ⇒ *'a list* ⇒ *'b list* **where**
map-index' n f [] = []
| *map-index' n f* (*x*#*xs*) = *f n x* # *map-index'* (*Suc n*) *f xs*

lemma *length-map-index'*[*simp*]: *length* (*map-index' n f xs*) = *length xs*
by (*induct xs arbitrary*: *n*) *auto*

lemma *map-index'-map-zip*: *map-index' n f xs* = *map* (*case-prod f*) (*zip* [*n* ..< *n* + *length xs*] *xs*)
proof (*induct xs arbitrary*: *n*)
case (*Cons x xs*)
hence *map-index' n f* (*x*#*xs*) = *f n x* # *map* (*case-prod f*) (*zip* [*Suc n* ..< *n* + *length* (*x* # *xs*)] *xs*) **by** *simp*
also have ... = *map* (*case-prod f*) (*zip* (*n* # [*Suc n* ..< *n* + *length* (*x* # *xs*)]) (*x* # *xs*)) **by** *simp*
also have (*n* # [*Suc n* ..< *n* + *length* (*x* # *xs*)]) = [*n* ..< *n* + *length* (*x* # *xs*)] **by** (*induct xs*) *auto*
finally show *?case* **by** *simp*
qed *simp*

abbreviation *map-index* ≡ *map-index' 0*

lemmas *map-index* = *map-index'-map-zip*[*of 0*, *simplified*]

lemma *take-map-index*: *take p* (*map-index f xs*) = *map-index f* (*take p xs*)
unfolding *map-index* **by** (*auto simp*: *min-def take-map take-zip*)

lemma *drop-map-index*: *drop p* (*map-index f xs*) = *map-index' p f* (*drop p xs*)
unfolding *map-index'-map-zip* **by** (*cases p* < *length xs*) (*auto simp*: *drop-map drop-zip*)

lemma *map-map-index*[*simp*]: *map g* (*map-index f xs*) = *map-index* (λ*n x*. *g* (*f n x*)) *xs*
unfolding *map-index* **by** *auto*

lemma *map-index-map*[*simp*]: *map-index f* (*map g xs*) = *map-index* (λ*n x*. *f n* (*g x*)) *xs*
unfolding *map-index* **by** (*auto simp*: *map-zip-map2*)

lemma *set-map-index*[*simp*]: *x* ∈ *set* (*map-index f xs*) = (∃ *i* < *length xs*. *f i* (*xs* ! *i*) = *x*)
unfolding *map-index* **by** (*auto simp*: *set-zip intro*!: *image-eqI*[*of - case-prod f*])

lemma *set-map-index'*[*simp*]: *x*∈*set* (*map-index' n f xs*)
⟷ (∃ *i*<*length xs*. *f* (*n*+*i*) (*xs*!*i*) = *x*)

unfolding *map-index'-map-zip*
by (*auto simp*: *set-zip intro*!: *image-eqI*[*of* - *case-prod f*])

lemma *nth-map-index*[*simp*]: $p < length\ xs \Longrightarrow map\text{-}index\ f\ xs\ !\ p = f\ p\ (xs\ !\ p)$
unfolding *map-index* **by** *auto*

lemma *map-index-cong*:
$\forall p < length\ xs.\ f\ p\ (xs\ !\ p) = g\ p\ (xs\ !\ p) \Longrightarrow map\text{-}index\ f\ xs = map\text{-}index\ g\ xs$
unfolding *map-index* **by** (*auto simp*: *set-zip*)

lemma *map-index-id*: $map\text{-}index\ (curry\ snd)\ xs = xs$
unfolding *map-index* **by** *auto*

lemma *map-index-no-index*[*simp*]: $map\text{-}index\ (\lambda n\ x.\ f\ x)\ xs = map\ f\ xs$
unfolding *map-index* **by** (*induct xs rule*: *rev-induct*) *auto*

lemma *map-index-congL*:
$\forall p < length\ xs.\ f\ p\ (xs\ !\ p) = xs\ !\ p \Longrightarrow map\text{-}index\ f\ xs = xs$
by (*rule trans*[*OF map-index-cong map-index-id*]) *auto*

lemma *map-index'-is-NilD*: $map\text{-}index'\ n\ f\ xs = [] \Longrightarrow xs = []$
by (*induct xs*) *auto*

declare *map-index'-is-NilD*[*of 0*, *dest*!]

lemma *map-index'-is-ConsD*:
$map\text{-}index'\ n\ f\ xs = y\ \#\ ys \Longrightarrow \exists z\ zs.\ xs = z\ \#\ zs \land f\ n\ z = y \land map\text{-}index'\ (n + 1)\ f\ zs = ys$
by (*induct xs arbitrary*: *n*) *auto*

lemma *map-index'-eq-imp-length-eq*: $map\text{-}index'\ n\ f\ xs = map\text{-}index'\ n\ g\ ys \Longrightarrow length\ xs = length\ ys$
proof (*induct ys arbitrary*: *xs n*)
case (*Cons y ys*) **thus** *?case* **by** (*cases xs*) *auto*
qed (*auto dest*!: *map-index'-is-NilD*)

lemmas *map-index-eq-imp-length-eq* = *map-index'-eq-imp-length-eq*[*of 0*]

lemma *map-index'-comp*[*simp*]: $map\text{-}index'\ n\ f\ (map\text{-}index'\ n\ g\ xs) = map\text{-}index'\ n\ (\lambda n.\ f\ n\ o\ g\ n)\ xs$
by (*induct xs arbitrary*: *n*) *auto*

lemma *map-index'-append*[*simp*]: $map\text{-}index'\ n\ f\ (a\ @\ b)$
$= map\text{-}index'\ n\ f\ a\ @\ map\text{-}index'\ (n + length\ a)\ f\ b$
by (*induct a arbitrary*: *n*) *auto*

lemma *map-index-append*[*simp*]: $map\text{-}index\ f\ (a\ @\ b)$
$= map\text{-}index\ f\ a\ @\ map\text{-}index'\ (length\ a)\ f\ b$
using *map-index'-append*[**where** *n=0*]

by (*simp del*: *map-index′-append*)

2.3 Insert at position

primrec *insert-nth* :: *nat* ⇒ *′a* ⇒ *′a list* ⇒ *′a list* **where**
insert-nth 0 x xs = *x* # *xs*
| *insert-nth* (*Suc n*) *x xs* = (*case xs of* [] ⇒ [*x*] | *y* # *ys* ⇒ *y* # *insert-nth n x ys*)

lemma *insert-nth-take-drop*[*simp*]: *insert-nth n x xs* = *take n xs* @ [*x*] @ *drop n xs*
proof (*induct n arbitrary*: *xs*)
case *Suc* **thus** *?case* **by** (*cases xs*) *auto*
qed *simp*

lemma *length-insert-nth*: *length* (*insert-nth n x xs*) = *Suc* (*length xs*)
by (*induct xs*) *auto*

lemma *set-insert-nth*:
set (*insert-nth i x xs*) = *insert x* (*set xs*)
by (*simp add*: *set-append*[*symmetric*])

lemma *distinct-insert-nth*:
assumes *distinct xs*
assumes $x \notin$ *set xs*
shows *distinct* (*insert-nth i x xs*)
using *assms* **proof** (*induct xs arbitrary*: *i*)
case *Nil*
then show *?case* **by** (*cases i*) *auto*
next
case (*Cons a xs*)
then show *?case*
by (*cases i*) (*auto simp add*: *set-insert-nth simp del*: *insert-nth-take-drop*)
qed

lemma *nth-insert-nth-front*:
assumes $i < j$ $j \le$ *length xs*
shows *insert-nth j x xs* ! *i* = *xs* ! *i*
using *assms* **by** (*simp add*: *nth-append*)

lemma *nth-insert-nth-index-eq*:
assumes $i \le$ *length xs*
shows *insert-nth i x xs* ! *i* = *x*
using *assms* **by** (*simp add*: *nth-append*)

lemma *nth-insert-nth-back*:
assumes $j < i$ $i \le$ *length xs*
shows *insert-nth j x xs* ! *i* = *xs* ! ($i - 1$)
using *assms* **by** (*cases i*) (*auto simp add*: *nth-append min-def*)

lemma *nth-insert-nth*:
  assumes *i* ≤ *length xs j* ≤ *length xs*
  shows *insert-nth j x xs* ! *i* = (*if i* = *j then x else if i* < *j then xs* ! *i else xs* ! (*i*
− *1*))
using *assms* **by** (*simp add*: *nth-insert-nth-front nth-insert-nth-index-eq nth-insert-nth-back*
del: *insert-nth-take-drop*)

lemma *insert-nth-inverse*:
  assumes *j* ≤ *length xs j'* ≤ *length xs'*
  assumes *x* ∉ *set xs x* ∉ *set xs'*
  assumes *insert-nth j x xs* = *insert-nth j' x xs'*
  shows *j* = *j'*
proof −
  from *assms*(*1*,*3*) **have** ∀ *i*≤*length xs*. *insert-nth j x xs* ! *i* = *x* ⟷ *i* = *j*
    by (*auto simp add*: *nth-insert-nth simp del*: *insert-nth-take-drop*)
  moreover from *assms*(*2*,*4*) **have** ∀ *i*≤*length xs'*. *insert-nth j' x xs'* ! *i* = *x* ⟷
i = *j'*
    by (*auto simp add*: *nth-insert-nth simp del*: *insert-nth-take-drop*)
  ultimately show *j* = *j'*
    using *assms*(*1*,*2*,*5*) **by** (*metis dual-order.trans nat-le-linear*)
qed

Insert several elements at given (ascending) positions

lemma *length-fold-insert-nth*:
  length (*fold* (λ(*p*, *b*). *insert-nth p b*) *pxs xs*) = *length xs* + *length pxs*
  by (*induct pxs arbitrary*: *xs*) *auto*

lemma *invar-fold-insert-nth*:
  ⟦∀ *x*∈*set pxs*. *p* < *fst x*; *p* < *length xs*; *xs* ! *p* = *b*⟧ ⟹
    fold (λ(*x*, *y*). *insert-nth x y*) *pxs xs* ! *p* = *b*
  by (*induct pxs arbitrary*: *xs*) (*auto simp*: *nth-append*)

lemma *nth-fold-insert-nth*:
  ⟦*sorted* (*map fst pxs*); *distinct* (*map fst pxs*); ∀ (*p*, *b*) ∈ *set pxs*. *p* < *length xs* +
length pxs;
    i < *length pxs*; *pxs* ! *i* = (*p*, *b*)⟧ ⟹
  fold (λ(*p*, *b*). *insert-nth p b*) *pxs xs* ! *p* = *b*
proof (*induct pxs arbitrary*: *xs i p b*)
  case (*Cons pb pxs*)
  show *?case*
  proof (*cases i*)
    case *0*
    with *Cons.prems* **have** *p* < *Suc* (*length xs*)
    proof (*induct pxs rule*: *rev-induct*)
      case (*snoc pb' pxs*)
      then obtain *p' b'* **where** *pb'* = (*p'*, *b'*) **by** *auto*
      with *snoc.prems* **have** ∀ *p* ∈ *fst* ‘ *set pxs*. *p* < *p' p'* ≤ *Suc* (*length xs* + *length*
pxs)
        by (*auto simp*: *image-iff sorted-append le-eq-less-or-eq*)

```
      with snoc.prems show ?case by (intro snoc(1)) (auto simp: sorted-append)
    qed auto
    with 0 Cons.prems show ?thesis unfolding fold.simps o-apply
   by (intro invar-fold-insert-nth) (auto simp: image-iff le-eq-less-or-eq nth-append)
  next
    case (Suc n) with Cons.prems show ?thesis unfolding fold.simps
      by (auto intro!: Cons(1))
  qed
qed simp
```

2.4 Remove at position

```
fun remove-nth :: nat ⇒ 'a list ⇒ 'a list
where
  remove-nth i [] = []
| remove-nth 0 (x # xs) = xs
| remove-nth (Suc i) (x # xs) = x # remove-nth i xs

lemma remove-nth-take-drop:
  remove-nth i xs = take i xs @ drop (Suc i) xs
proof (induct xs arbitrary: i)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  then show ?case by (cases i) auto
qed

lemma remove-nth-insert-nth:
  assumes i ≤ length xs
  shows remove-nth i (insert-nth i x xs) = xs
using assms proof (induct xs arbitrary: i)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  then show ?case by (cases i) auto
qed

lemma insert-nth-remove-nth:
  assumes i < length xs
  shows insert-nth i (xs ! i) (remove-nth i xs) = xs
using assms proof (induct xs arbitrary: i)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  then show ?case by (cases i) auto
qed
```

```
lemma length-remove-nth:
  assumes i < length xs
  shows length (remove-nth i xs) = length xs − 1
using assms unfolding remove-nth-take-drop by simp

lemma set-remove-nth-subset:
  set (remove-nth j xs) ⊆ set xs
proof (induct xs arbitrary: j)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  then show ?case by (cases j) auto
qed

lemma set-remove-nth:
  assumes distinct xs j < length xs
  shows set (remove-nth j xs) = set xs − {xs ! j}
using assms proof (induct xs arbitrary: j)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  then show ?case by (cases j) auto
qed

lemma distinct-remove-nth:
  assumes distinct xs
  shows distinct (remove-nth i xs)
using assms proof (induct xs arbitrary: i)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  then show ?case
    by (cases i) (auto simp add: set-remove-nth-subset set-rev-mp)
qed

end
theory commdata
  imports Main
begin

definition PAGE-SIZE ≡ nat 4096

definition VM-NUM-MAX ≡ nat 8

definition MAX-CHANNEL-NUM ≡ VM-NUM-MAX ∗ 2
```

type-synonym *region-num* = *nat*

type-synonym *u64* = *nat*
type-synonym *region-size* = *nat*
type-synonym *page-free* = *nat*
type-synonym *page-last* = *nat*

type-synonym *bitmap* = *bool list*

record *mem-region* = *sizeB* :: *region-size*
freeB :: *page-free*

type-synonym *vcpu-idx* = *nat*

type-synonym *INTERRUPT-NUM-MAX* = *nat*
type-synonym *INTERRUPT-NUM-SIZE* = *nat*

type-synonym *interrupt-id* = *nat*
type-synonym *interrupt-src* = *nat*

datatype *handlers* = *irq-handler-t interrupt-id interrupt-src*

datatype *Interrupt-Config* = *BITMAP bitmap interrupt-id handlers list*

record *Interrupts* = *Interrupt-hyper-bitmap*::*bitmap*
Interrupt-glb-bitmap::*bitmap*
Interrupt-handlers::*handlers list*

type-synonym *channel-num* = *nat*

type-synonym *page-num* = *nat*
type-synonym *mem-region-index* = *nat*
type-synonym *bitmap-region* = *page-num* ⇒ *mem-region-index option*

type-synonym *page-index* = *nat*
type-synonym *bitmap-heap* = *page-num* ⇒ *page-index option*
type-synonym *count-heap* = *page-index* ⇒ *page-num* ⇒ *page-num*

datatype *CPU-WORK-STATE* = *CPU-S-INV* | *CPU-S-IDLE* | *CPU-S-RUN*
datatype *VM-WORK-STATE* = *VM-S-INV* | *VM-S-PENDING* | *VM-S-ACT*
datatype *VCPU-WORK-STATE* = *VCPU-S-INV* | *VCPU-S-PEND* | *VCPU-S-ACT*

type-synonym *cpu-id* = *nat*
type-synonym *vcpu-id* = *nat*
type-synonym *vcpu-num* = *nat*
type-synonym *vcpu-pool* = *vcpu-id list*

type-synonym *vm-id* = *nat*
type-synonym *vm-name* = *string*
type-synonym *Alloc-VCPU* = *vcpu-id list*
type-synonym *Alloc-CPU* = *cpu-id list*
type-synonym *BITMAP* = *bool list*

type-synonym *region-idx* = *nat*

type-synonym *port-id* =*nat*

record *pa-region* = *pa-start* :: *u64*
pa-length :: *u64*
offset :: *u64*

record *CPU* = *id* :: *cpu-id*
vcpus :: *vcpu-pool*
poolSize :: *nat*
active-vcpu :: *vcpu-id*
running-num :: *vcpu-num*

record *VCPU* = *id* :: *vcpu-id*
physID :: *cpu-id*
vmID :: *vm-id*
allocated :: *bool*

datatype *vm-port-type* = *RECEIVE* | *SEND*

datatype *vm-type* = *OS* | *BMA*

record *vm-port* = *type* :: *vm-port-type*
idV :: *vm-id*

type-synonym *Port-Config* = (*vm-port* × *vm-id*) *list*

record *VM-INFO* = *vmId* :: *vm-id*
vmName :: *vm-name*

vmType :: *vm-type*
vmState :: *VM-WORK-STATE*

record *VM* = *id* :: *vm-id*
name :: *vm-name*
type :: *vm-type*
vcpus :: *Alloc-VCPU*
cpus :: *Alloc-CPU*
int-bitmap::*BITMAP*
address :: *pa-region list*
port-config :: *Port-Config*

typedecl *MSG*
typedecl *IVC*

type-synonym *channel-id* = *nat*
type-synonym *assigned* = *bool*

record *ports* = *nums* :: *nat*
port :: *vm-port list*

type-synonym *portP* = *vm-id*×*vm-id*

record *channel* = *id* :: *channel-id*
flag :: *assigned*
portSrc :: *vm-port option*
portDes :: *vm-port option*
msg :: *MSG*

record *IVC-State* = *channel-num* :: *nat*
channels :: *channel list*

record *vm-port-config* = *num* :: *nat*
contact-vm :: *vm-id list*
contact-type :: *vm-port-type list*

end
theory *commfunc*
imports *Main commdata*
begin

end
theory *hvc*
imports *Main ../util/List-Index ../common/commfunc*
begin

3 Data type Declaration

3.1 Memory

typedecl *Page*

datatype *Mem-Heap-Config = Heap-Config region-size bitmap-heap count-heap Page list*

record *Mem-Heap = size :: region-size*
freeR :: page-free
mapR :: bitmap-heap
free-countR :: count-heap
pl :: Page list

type-synonym *region-num = nat*
type-synonym *mem-region-index = nat*
type-synonym *vm-mem-position = mem-region-index × page-index*
type-synonym *bitmap-region = page-num ⇒ mem-region-index option*

datatype *Mem-Region-Config = Mem-Region-CFG region-size Page list*
type-synonym *Mem-Config = mem-region-index ⇒ Mem-Region-Config*
datatype *Mem-VM-Config = Mem-VM-Config region-num Mem-Config bitmap-region*

type-synonym *pa-Map = u64 ⇒ region-idx option*
type-synonym *ipa-Map = u64 ⇒ region-idx option*

datatype *VM-REGION-MAPS = RegionMapCFG pa-Map list ipa-Map list*

type-synonym *map-targetN = vcpu-id ⇒ vcpu-id option*
type-synonym *map-targetV = vm-id ⇒ vcpu-id option*
type-synonym *map-isHave = vcpu-id ⇒ nat option*
type-synonym *map-vidx = vcpu-id list ⇒ vcpu-idx option*

datatype *CPU-Related-MAPS* = *CPUMapsCFG map-targetN list map-targetV list map-isHave list map-vidx list*

type-synonym *getChMap* = *vm-id* ⇒ *vm-id* ⇒ *channel-id option*
type-synonym *availChMap* = *vm-port* ⇒ *vm-id* ⇒ *channel-id option*

datatype *IVC-Config* = *ivcCFG channel-num channel list*

datatype *Commu-Config* = *CommuCFG IVC-Config getChMap availChMap*

record *Mem-VM* = *vm-region* :: *mem-region list*
map :: *bitmap-region*

3.2 cpu

datatype *VM-Config* = *VM-CFG vm-id vm-name vm-type Alloc-VCPU Alloc-CPU bitmap pa-region list Port-Config*
datatype *VMS-Config* = *VMS-CFG VM-Config list*

datatype *VCPU-Config* = *VCPU-CFG vcpu-id*
datatype *VCPUS-Config* = *VCPUS-CFG VCPU-Config list*

datatype *CPU-Config* = *CPU-CFG cpu-id*
datatype *CPUS-Config* = *CPUS-CFG CPU-Config list*

3.3 Interrupts

type-synonym *assigned* = *bool*
type-synonym *port-id* = *nat*
type-synonym *channel-id* = *nat*

type-synonym *Port-Config-Pair* = *Port-Config*×*Port-Config*

datatype *Channel-Config* = *Channel-CFG channel-id assigned Port-Config-Pair MSG*

datatype *COMMU-Config* = *COMMU-CFG Channel-Config list vm-id list*

4 Component configuration and relevant event

record *Sys-Config* = *mhc* :: *Mem-Heap-Config*
mvc :: *Mem-VM-Config*
vmc :: *VMS-Config*

regmap :: *VM-REGION-MAPS*
vcpuc :: *VCPUS-Config*
cpuc :: *CPUS-Config*
cpumaps :: *CPU-Related-MAPS*
intc :: *Interrupt-Config*
irqc :: *handlers*
commuc :: *Commu-Config*

datatype *Resource-Type* = *T-CPU* | *T-MEM* | *T-VM* | *T-SYS*

datatype *Event* = *HVC* | *IRQ*

datatype *Event-Result* = *SUCCEED* | *FAILED* | *UNKNOWN*

record *Commu* = *ivc* :: *IVC-State*
getChannel :: *vm-id* ⇒ *vm-id* ⇒ *channel-id option*
availChannel :: *vm-port* ⇒ *vm-id* ⇒ *channel-id option*

record *CPU-MAPS* = *cpu-map-active* :: *map-targetN list*
cpu-map-idByVM :: *map-targetV list*
cpu-is-haveVCPU :: *map-isHave list*
get-vcpuIDX :: *map-vidx list*

record *VM-MAPS* = *get-paIdX* :: *pa-Map list*
get-ipaIdX :: *ipa-Map list*

record *Audit* = *res* :: *Resource-Type*
sub :: *nat*
event :: *Event*
result :: *Event-Result*

record *HV* = *mem-heap* :: *Mem-Heap*
mem-vm :: *Mem-VM*
vm :: *VM list*
vm-wk-st :: *VM-WORK-STATE list*
vm-regionMaps :: *VM-MAPS*
vcpu :: *VCPU list*
vcpu-wk-st :: *VCPU-WORK-STATE list*
cpu :: *CPU list*
cpu-wk-st :: *CPU-WORK-STATE list*
cpu-maps :: *CPU-MAPS*

interrupts::*Interrupts*

commu :: *Commu*
audit :: *Audit list*

4.1 relevant event

datatype *Mem-Heap-Event* = *Heap-Rgion-Init* | *Heap-Alloc page-num* |
Heap-Free page-index page-num | *Heap-Region-Reset*

datatype *Mem-VM-Event* = *Mem-VM-Init* | *VM-Region-Alloc page-num* |
VM-Region-Free mem-region-index page-num

datatype *CPU-Event* = *CPU-Init* | *CPU-Report-State*

datatype *VCPU-Event* = *VCPU-Init* | *VCPU-RUN cpu-id* | *VCPU-SHUTDOWN vcpu-id* |
VCPU-POOL-SWITCH cpu-id

datatype *VM-EVENT* = *VM-Init* | *VMM-SHOWDOWN-VM vm-id*

datatype *VMM-EVENT* = *VMM-SHOWDOWN-VM vm-id*

datatype *IVC-EVENT* = *IVC-SEND-MSG vm-id MSG vm-id*

datatype *SCHED-EVENT* = *CPU-SCHEDULE cpu-id*

datatype *Interrupt-Event* = *Interrupt-init* |
Interrupt-reserve-int interrupt-id handlers |
Interrupt-cpu-enable interrupt-id bool |
Interrupt-cpu-ipi-send cpu-id interrupt-id |
Interrupt-Handler interrupt-id interrupt-src

datatype *HVC-SYS-EVENT* = *SYS-REBOOT* | *SYS-SHUTDOWN*
datatype *HVC-VMM-EVENT* = *VMM-LIST-VM-INFO* | *VMM-SHUTDOWN-VM vm-id* |
VMM-REBOOT-VM vm-id | *VMM-GET-VM-ID vm-name*
datatype *HVC-IVC-EVENT* = *IVC-SEND-MSG vm-id MSG vm-id*
| *IVC-BROADCAST-MSG vm-id*

5 Retrieve Configuration's element

primrec *get-heapSize-HCFG* :: *Mem-Heap-Config* ⇒ *region-size* **where**
get-heapSize-HCFG (*Heap-Config s m c pl0*) = *s*

primrec *get-bitMap-HCFG* :: *Mem-Heap-Config* ⇒ *bitmap-heap* **where**
get-bitMap-HCFG (*Heap-Config s m c pl0*) = *m*

primrec *get-freeCount-HCFG* :: *Mem-Heap-Config* ⇒ *count-heap* **where**
get-freeCount-HCFG (*Heap-Config s m c pl0*) = *c*

primrec *get-pageList-HCFG* :: *Mem-Heap-Config* ⇒ *Page list* **where**
get-pageList-HCFG (*Heap-Config s m c pl0*) = *pl0*

primrec *get-regionNum-VCFG* :: *Mem-VM-Config* ⇒ *region-num* **where**
get-regionNum-VCFG (*Mem-VM-Config n mc br*) = *n*

primrec *get-memConf-VCFG* :: *Mem-VM-Config* ⇒ *Mem-Config* **where**
get-memConf-VCFG (*Mem-VM-Config n mc br*) = *mc*

primrec *get-bitmap-VCFG* :: *Mem-VM-Config* ⇒ *bitmap-region* **where**
get-bitmap-VCFG (*Mem-VM-Config n mc br*) = *br*

primrec *get-regionSize-RCFG* :: *Mem-Region-Config* ⇒ *region-size* **where**
get-regionSize-RCFG (*Mem-Region-CFG s pl0*) = *s*

primrec *get-pageList-RCFG* :: *Mem-Region-Config* ⇒ *Page list* **where**
get-pageList-RCFG (*Mem-Region-CFG s pl0*) = *pl0*

primrec *get-VMCFGList* :: *VMS-Config* ⇒ *VM-Config list* **where**
get-VMCFGList (*VMS-CFG vcs*) = *vcs*

primrec *get-ID-VMCFG* :: *VM-Config* ⇒ *vm-id* **where**
get-ID-VMCFG (*VM-CFG id0 nam typ avc ac - - -*) = *id0*

primrec *get-NAME-VMCFG* :: *VM-Config* ⇒ *vm-name* **where**
get-NAME-VMCFG (*VM-CFG id0 nam typ avc ac - - -*) = *nam*

primrec *get-TYPE-VMCFG* :: *VM-Config* ⇒ *vm-type* **where**
get-TYPE-VMCFG (*VM-CFG id0 nam typ avc ac - - -*) = *typ*

primrec *get-AVC-VMCFG* :: *VM-Config* ⇒ *Alloc-VCPU* **where**
get-AVC-VMCFG (*VM-CFG id0 nam typ avc ac - - -*) = *avc*

primrec *get-AC-VMCFG* :: *VM-Config* ⇒ *Alloc-CPU* **where**
get-AC-VMCFG (*VM-CFG id0 nam typ avc ac - - -*) = *ac*

primrec *get-BITMAP-VMCFG* :: *VM-Config* ⇒ *bitmap* **where**
get-BITMAP-VMCFG (*VM-CFG - - - - - bmp - -*) = *bmp*

primrec *get-ADDR-VMCFG* :: *VM-Config* ⇒ *pa-region list* **where**
get-ADDR-VMCFG (*VM-CFG - - - - - - addr -*) = *addr*

primrec *get-PORTCFG-VMCFG* :: *VM-Config* ⇒ *Port-Config* **where**
get-PORTCFG-VMCFG (*VM-CFG - - - - - - - cfg*) = *cfg*

primrec *get-ID-VCPUCFG* :: *VCPU-Config* ⇒ *vcpu-id* **where**
get-ID-VCPUCFG (*VCPU-CFG id0*) = *id0*

primrec *get-VCPUCFGList* :: *VCPUS-Config* ⇒ *VCPU-Config list* **where**

get-VCPUCFGList (*VCPUS-CFG a*) = *a*

primrec *get-ID-CPUCFG* :: *CPU-Config* ⇒ *cpu-id* **where**
get-ID-CPUCFG (*CPU-CFG id0*) = *id0*

primrec *get-CPUCFGList* :: *CPUS-Config* ⇒ *CPU-Config list* **where**
get-CPUCFGList (*CPUS-CFG a*) = *a*

primrec *get-intIndex-BCFG* :: *Interrupt-Config* ⇒ *interrupt-id* **where**
get-intIndex-BCFG (*BITMAP - i -*) = *i*

primrec *get-Bitmap-BCFG* :: *Interrupt-Config* ⇒ *bitmap* **where**
get-Bitmap-BCFG (*BITMAP bmp - -*) = *bmp*

primrec *get-Handlers-BCFG* :: *Interrupt-Config* ⇒ *handlers list* **where**
get-Handlers-BCFG (*BITMAP - - hdl*) = *hdl*

primrec *get-channelCFGList* :: *COMMU-Config* ⇒ *Channel-Config list* **where**
get-channelCFGList (*COMMU-CFG a b*) = *a*

primrec *get-vmcList* :: *COMMU-Config* ⇒ *vm-id list* **where**
get-vmcList (*COMMU-CFG a b*) = *b*

primrec *get-ID-channelCFG* :: *Channel-Config* ⇒ *channel-id* **where**
get-ID-channelCFG (*Channel-CFG a b c d*) = *a*

primrec *get-assigned-channelCFG* :: *Channel-Config* ⇒ *assigned* **where**
get-assigned-channelCFG (*Channel-CFG a b c d*) = *b*

primrec *get-PortsCFG-channelCFG* :: *Channel-Config* ⇒ *Port-Config-Pair* **where**
get-PortsCFG-channelCFG (*Channel-CFG a b c d*) = *c*

primrec *get-InitalMSG-channelCFG* :: *Channel-Config* ⇒ *MSG* **where**
get-InitalMSG-channelCFG (*Channel-CFG a b c d*) = *d*

primrec *get-paMap-RegionMap* :: *VM-REGION-MAPS* ⇒ *pa-Map list* **where**
get-paMap-RegionMap (*RegionMapCFG a b*) = *a*

primrec *get-ipaMap-RegionMap* :: *VM-REGION-MAPS* ⇒ *ipa-Map list* **where**
get-ipaMap-RegionMap (*RegionMapCFG a b*) = *b*

primrec *get-map1-CPURelatedMap* :: *CPU-Related-MAPS* ⇒ *map-targetN list*
where
get-map1-CPURelatedMap (*CPUMapsCFG a b c d*) = *a*

primrec *get-map2-CPURelatedMap* :: *CPU-Related-MAPS* ⇒ *map-targetV list*

where
  get-map2-CPURelatedMap (*CPUMapsCFG a b c d*) = *b*

primrec *get-map3-CPURelatedMap* :: *CPU-Related-MAPS* ⇒ *map-isHave list*
where
  get-map3-CPURelatedMap (*CPUMapsCFG a b c d*) = *c*

primrec *get-map4-CPURelatedMap* :: *CPU-Related-MAPS* ⇒ *map-vidx list* **where**
  get-map4-CPURelatedMap (*CPUMapsCFG a b c d*) = *d*

primrec *get-nums-IVCCFG* :: *IVC-Config* ⇒ *channel-num* **where**
  get-nums-IVCCFG (*ivcCFG a b*) = *a*

primrec *get-chans-IVCCFG* :: *IVC-Config* ⇒ *channel list* **where**
  get-chans-IVCCFG (*ivcCFG a b*) = *b*

primrec *get-ivcCfg-COMMUCFG* :: *Commu-Config* ⇒ *IVC-Config* **where**
  get-ivcCfg-COMMUCFG (*CommuCFG a b c*) = *a*

primrec *get-map1-COMMUCFG* :: *Commu-Config* ⇒ *getChMap* **where**
  get-map1-COMMUCFG (*CommuCFG a b c*) = *b*

primrec *get-map2-COMMUCFG* :: *Commu-Config* ⇒ *availChMap* **where**
  get-map2-COMMUCFG (*CommuCFG a b c*) = *c*

6 relevant Functions and Definitions

6.1 Asset Initialization

definition *get-natArr* :: *int* ⇒ *nat list* **where**
  get-natArr numX ≡ *List.map* (λ*x*. *nat x*) [*1..numX*]

definition *heap-region-init* :: *Mem-Heap-Config* => *Mem-Heap* **where**
  heap-region-init cfg ≡ *let*
    size0 = *get-heapSize-HCFG cfg*;
    map0 = *get-bitMap-HCFG cfg*;
    count0 = *get-freeCount-HCFG cfg*;
    pl0 = *get-pageList-HCFG cfg*
      in (|*Mem-Heap.size*=*size0*, *freeR*=*size0*, *mapR*=*map0*, *free-countR*=*count0* , *pl*=*pl0*|)

definition *mem-region-init* :: *Mem-Region-Config* ⇒ *mem-region* **where**
  mem-region-init cfg ≡ *let*
    size0 = *get-regionSize-RCFG cfg*
    in (| *mem-region.sizeB*=*size0*, *freeB*=*size0* |)

definition *mem-vm-init* :: *Mem-VM-Config* ⇒ *Mem-VM* **where**

mem-vm-init cfg ≡
let
regionNum0 = *get-regionNum-VCFG cfg*;
regionNum1 = *int regionNum0*;
bitmap0 = *get-bitmap-VCFG cfg*;
memConfList = *List.map* ($\lambda x.$(*get-memConf-VCFG cfg x*)) (*get-natArr regionNum1*);
memRegionList = *List.map* ($\lambda y.$(*mem-region-init y*)) *memConfList*
in ⦇ *Mem-VM.vm-region* = *memRegionList*, *map* = *bitmap0* ⦈

definition *VM-RegionMaps-init* :: *VM-REGION-MAPS* ⇒ *VM-MAPS* **where**
VM-RegionMaps-init cfg ≡
let
map1 = *get-paMap-RegionMap cfg*;
map2 = *get-ipaMap-RegionMap cfg*
in
⦇*get-paIdX* = *map1* , *get-ipaIdX* = *map2* ⦈

definition *CPU-RelatedMaps-init* :: *CPU-Related-MAPS* ⇒ *CPU-MAPS* **where**
CPU-RelatedMaps-init cfg ≡
let
map1 = *get-map1-CPURelatedMap cfg*;
map2 = *get-map2-CPURelatedMap cfg*;
map3 = *get-map3-CPURelatedMap cfg*;
map4 = *get-map4-CPURelatedMap cfg*
in
⦇*cpu-map-active* = *map1* , *cpu-map-idByVM* = *map2*, *cpu-is-haveVCPU* = *map3*, *get-vcpuIDX* = *map4* ⦈

definition *IVC-init* :: *IVC-Config* ⇒ *IVC-State* **where**
IVC-init cfg ≡
let
n = *get-nums-IVCCFG cfg*;
l = *get-chans-IVCCFG cfg*
in
⦇*channel-num* = *n* , *channels* = *l* ⦈

definition *Commus-init* :: *Commu-Config* ⇒ *Commu* **where**
Commus-init cfg ≡
let
ivc0 = *IVC-init* (*get-ivcCfg-COMMUCFG cfg*);
map1 = *get-map1-COMMUCFG cfg*;
map2 = *get-map2-COMMUCFG cfg*
in
⦇*ivc* = *ivc0* , *getChannel* = *map1*, *availChannel* = *map2* ⦈

```
definition VM-init :: VM-Config ⇒ VM where
  VM-init cfg ≡
    let
      id0 = get-ID-VMCFG cfg;
      name0 = get-NAME-VMCFG cfg;
      type0 = get-TYPE-VMCFG cfg;
      avc = get-AVC-VMCFG cfg;
      ac = get-AC-VMCFG cfg;
      bmp = get-BITMAP-VMCFG cfg;
      addr = get-ADDR-VMCFG cfg;
      portCFG = get-PORTCFG-VMCFG cfg
    in (| VM.id=id0, name=name0 , type = type0, vcpus=avc, cpus=ac,
         int-bitmap=bmp,address=addr,port-config=portCFG|)


definition VMS-init :: VMS-Config ⇒ VM list where
  VMS-init cfg ≡
    List.map (λx.(VM-init x)) (get-VMCFGList cfg)

value List.replicate 2 (4::int)

definition VMWorkState-init :: VMS-Config ⇒ VM-WORK-STATE list where
  VMWorkState-init cfg ≡
    List.replicate (length (get-VMCFGList cfg)) VM-S-INV

definition VCPU-init :: VCPU-Config ⇒ VCPU where
  VCPU-init cfg1 ≡ let
                      id0 = get-ID-VCPUCFG cfg1
                    in (| VCPU.id=id0, physID=0, vmID=0, allocated=False |)


definition VCPUS-init :: VCPUS-Config ⇒ VCPU list where
  VCPUS-init cfg1 ≡ List.map (λx.(VCPU-init x )) (get-VCPUCFGList cfg1)

definition VCPUWorkState-init :: VCPUS-Config ⇒ VCPU-WORK-STATE list
where
  VCPUWorkState-init cfg ≡
    List.replicate (length (get-VCPUCFGList cfg)) VCPU-S-PEND

definition CPU-init :: CPU-Config ⇒ CPU where
  CPU-init cfg ≡
    let
        id0 = get-ID-CPUCFG cfg
    in (| CPU.id=id0, vcpus=[], poolSize=0, active-vcpu = 0, running-num =0
|)
```

definition *CPUS-init* :: *CPUS-Config* ⇒ *CPU list* **where**
CPUS-init cfg ≡ *List.map* (*λx.*(*CPU-init x*)) (*get-CPUCFGList cfg*)

definition *CPUWorkState-init* :: *CPUS-Config* ⇒ *CPU-WORK-STATE list* **where**
CPUWorkState-init cfg ≡
List.replicate (*length* (*get-CPUCFGList cfg*)) *CPU-S-IDLE*

definition *interrupt-init* :: *Interrupt-Config* ⇒ *Interrupts* **where**
interrupt-init cfg ≡ *let*
b = (*get-Bitmap-BCFG cfg*);
hl = (*get-Handlers-BCFG cfg*)
in
(|*Interrupt-hyper-bitmap* = *b*, *Interrupt-glb-bitmap* = *b*, *Interrupt-handlers* = *hl*|)

definition *HV-init* :: *Sys-Config* ⇒ *HV* **where**
HV-init sc ≡
let
x1= *heap-region-init* (*mhc sc*) ;
x2= *mem-vm-init* (*mvc sc*) ;
x3= *VMS-init* (*vmc sc*) ;
x4= *VMWorkState-init* (*vmc sc*) ;
x5= *VM-RegionMaps-init* (*regmap sc*);
x6= *VCPUS-init* (*vcpuc sc*) ;
x7= *VCPUWorkState-init* (*vcpuc sc*);
x8= *CPUS-init* (*cpuc sc*) ;
x9= *CPUWorkState-init* (*cpuc sc*);
x10= *CPU-RelatedMaps-init* (*cpumaps sc*);
x11= *interrupt-init* (*intc sc*);
x12 = *Commus-init* (*commuc sc*)
in
(|*mem-heap*=*x1* , *mem-vm*=*x2* ,*vm* =*x3* , *vm-wk-st*=*x4* , *vm-regionMaps* = *x5*,
vcpu = *x6*, *vcpu-wk-st* = *x7*, *cpu* = *x8*, *cpu-wk-st* = *x9*, *cpu-maps*=*x10*,
interrupts= *x11*, *commu*=*x12*, *audit*=*Nil*|)

6.2 Other related functions and definitions

6.3 memory

definition *heap-alloc-req* :: *HV* => *page-num* => *HV* × *bool* **where**
heap-alloc-req hvc numX ≡
if (*numX* = *0* ∨ (*numX* > *Mem-Heap.freeR* (*mem-heap hvc*))) *then*
(*hvc*,*False*)
else
case mapR (*mem-heap hvc*) *numX of*

```
      None ⇒ (hvc,False) |
      Some region-index ⇒
          let
            free0 = Mem-Heap.freeR (mem-heap hvc) − numX;
            mem-heap0 = (mem-heap hvc)⦇Mem-Heap.freeR:=free0⦈
          in
            (hvc⦇mem-heap:=mem-heap0⦈,True)
```

```
definition heap-free-req :: HV ⇒ page-index ⇒ page-num ⇒ HV × bool where
  heap-free-req hvc index0 numX ≡
    if (index0 + numX) < Mem-Heap.size (mem-heap hvc) then
          let
              map0 = free-countR (mem-heap hvc);
              free0 = Mem-Heap.freeR (mem-heap hvc) + map0 index0 numX;
              mem-heap0 = (mem-heap hvc)⦇Mem-Heap.freeR:=free0⦈
          in
              (hvc⦇mem-heap:=mem-heap0⦈,True)
      else (hvc,False)
```

```
definition heap-region-reset :: HV ⇒ HV where
  heap-region-reset hvc ≡
    let
      free0 = Mem-Heap.size (mem-heap hvc);
      mem-heap0 = (mem-heap hvc)⦇Mem-Heap.freeR:=free0⦈
    in
      hvc⦇mem-heap:=mem-heap0⦈
```

```
definition get-memRegion :: HV ⇒ mem-region-index ⇒ mem-region where
  get-memRegion hvc idx ≡ (vm-region (mem-vm hvc))!idx
```

```
definition vm-region-alloc-req :: HV ⇒ page-num ⇒ HV × bool where
  vm-region-alloc-req hvc numX ≡
    case Mem-VM.map (mem-vm hvc) numX of
        None ⇒ (hvc,False) |
        Some idx ⇒
         let
           free0 = mem-region.freeB (get-memRegion hvc idx);
          region0 = ((vm-region (mem-vm hvc))!idx) ⦇freeB:= free0−numX ⦈;
           vm-region0 = (vm-region (mem-vm hvc))[idx:=region0];
           mem-vm0 = (mem-vm hvc)⦇vm-region:=vm-region0⦈
         in
           (hvc⦇mem-vm:=mem-vm0⦈, True)
```

```
definition vm-region-clear :: HV ⇒ mem-region-index ⇒ page-num ⇒ HV ×
bool where
```

vm-region-clear hvc idx numX $\equiv$
if idx $<$ (*length* (*vm-region* (*mem-vm hvc*))) $\wedge$ *numX* $<$ *mem-region.sizeB* ((*get-memRegion hvc idx*)) *then*
let
free0 = *mem-region.freeB* ((*vm-region* (*mem-vm hvc*))!*idx*) + *numX* ;
region0 = ((*vm-region* (*mem-vm hvc*))!*idx*) (|*mem-region.freeB*:=*free0*|);
vm-region0 = (*vm-region* (*mem-vm hvc*))[*idx*:=*region0*];
mem-vm0 = (*mem-vm hvc*)(|*vm-region*:=*vm-region0*|)
in
(*hvc*(|*mem-vm*:=*mem-vm0*|),*True*)
else (*hvc*,*False*)

definition *cpu-report-state* :: *HV* $\Rightarrow$ *CPU-WORK-STATE list* **where**
cpu-report-state hv $\equiv$
cpu-wk-st hv

definition *get-vmid-bycid* :: *HV* $\Rightarrow$ *cpu-id* $\Rightarrow$ *vm-id* **where**
get-vmid-bycid hv cid $\equiv$
let
vcid = *active-vcpu* ((*cpu hv*)!*cid*);
vcpu = (*vcpu hv*)!*vcid*
in vmID vcpu

definition *get-vmid-cid-req* :: *cpu-id* $\Rightarrow$ *HV* $\Rightarrow$ *vm-id* **where**
get-vmid-cid-req cid hv $\equiv$
let
vcid = *active-vcpu* ((*cpu hv*)!*cid*);
vcpu = (*vcpu hv*)!*vcid*
in
vmID vcpu

6.4 VCPU

definition *vcpu-run-req* :: *HV* $\Rightarrow$ *cpu-id* $\Rightarrow$ *HV* **where**
vcpu-run-req hv cid $\equiv$
if running-num((*cpu hv*)!*cid*) $>$ *1 then*
let
vmid = *get-vmid-cid-req cid hv*;
cpuN = (*cpu-wk-st hv*)[*cid*:=*CPU-S-RUN*];
vmN = (*vm-wk-st hv*)[*vmid*:=*VM-S-ACT*]
in
hv(|*cpu-wk-st*:=*cpuN*, *vm-wk-st*:=*vmN*|)
else hv

definition *set-active-vcpu-req* :: *HV* $\Rightarrow$ *cpu-id* $\Rightarrow$ *vcpu-id* $\Rightarrow$ *HV* **where**
set-active-vcpu-req hv cid vcid $\equiv$
let
cpu0 = (*cpu hv*)!*cid*;

actido = *active-vcpu cpu0*;
vcStas = (*vcpu-wk-st hv*)[*actido*:=*VCPU-S-PEND*, *vcid*:=*VCPU-S-ACT*];
cpuN = *cpu0*(|*active-vcpu*:=*vcid*|);
cpus = (*cpu hv*)[*cid*:=*cpuN*]
in hv(|*cpu*:=*cpus*, *vcpu-wk-st*:=*vcStas*|)

definition *vcpu-pool-suspend-req* :: *HV* ⇒ *cpu-id* ⇒ *vcpu-id* ⇒ *HV* × *bool*
where
vcpu-pool-suspend-req hv cid vcid ≡
if (*length* (*CPU.vcpus* ((*cpu hv*)!*cid*)) = *0* ∨ ((*cpu-is-haveVCPU* (*cpu-maps hv*))!*cid*) *vcid* = *None*) *then*
(*hv*,*False*)
else
if ((*vcpu-wk-st hv*)!*vcid*) = *VCPU-S-INV then*
(*hv*,*True*)
else
let
vN = (*vcpu-wk-st hv*)[*vcid*:=*VCPU-S-INV*];
cpu0 = (*cpu hv*)!*cid*;
cpu1 = *cpu0*(|*running-num*:=*running-num cpu0* − *1*|);
cN = (*cpu hv*)[*cid*:=*cpu1*]
in
(*hv*(|*cpu*:=*cN*,*vcpu-wk-st*:=*vN*|),*True*)

definition *vcpu-pool-wakeup-req* :: *HV* ⇒ *cpu-id* ⇒ *vcpu-id* ⇒ *HV* × *bool*
where
vcpu-pool-wakeup-req hv cid vcid ≡
if (*length* (*CPU.vcpus* ((*cpu hv*)!*cid*)) = *0* ∨ ((*cpu-is-haveVCPU* (*cpu-maps hv*))!*cid*) *vcid* = *None*) *then*
(*hv*,*False*)
else
if ¬((*vcpu-wk-st hv*)!*vcid*) = *VCPU-S-INV then*
(*hv*,*True*)
else
let
vN = (*vcpu-wk-st hv*)[*vcid*:=*VCPU-S-PEND*];
cpu0 = (*cpu hv*)!*cid*;
cpu1 = *cpu0*(|*running-num*:=*running-num cpu0* + *1*|);
cN = (*cpu hv*)[*cid*:=*cpu1*]
in
(*hv*(|*cpu*:=*cN*,*vcpu-wk-st*:=*vN*|),*True*)

definition *vcpu-pool-remove-req* :: *HV* ⇒ *cpu-id* ⇒ *vcpu-id* ⇒ *HV* × *bool*
where
vcpu-pool-remove-req hv cid vcid ≡

if (length (CPU.vcpus ((cpu hv)!cid)) = 0 ∨ ((cpu-is-haveVCPU (cpu-maps hv))!cid) vcid = None) then
(hv,False)
else
let
target = the (((cpu-is-haveVCPU (cpu-maps hv))!cid) vcid);
cpu0 = (cpu hv)!cid;
num0 = if ¬((vcpu-wk-st hv)!vcid) = VCPU-S-INV then
running-num cpu0 −1
else
running-num cpu0;
cpu1 = cpu0⦇running-num:=num0,CPU.vcpus:=(CPU.vcpus cpu0)[target:= length (vcpu hv)]⦈;
vN = (vcpu-wk-st hv)[vcid:=VCPU-S-INV]
in
(hv⦇cpu:=(cpu hv)[cid:=cpu1],vcpu-wk-st:=vN⦈,True)

definition *vcpu-pool-switch-req :: HV ⇒ cpu-id ⇒ vcpu-id ⇒ HV* **where**
vcpu-pool-switch-req hv cidx target ≡
let
cpu0 = (cpu hv)!cidx;
targetOld = active-vcpu cpu0;
running-num = running-num ((cpu hv)!cidx);
map0 = ((CPU-MAPS.cpu-map-active (cpu-maps hv))!cidx)
in
if(¬ (target = targetOld ∨ running-num = 0)) then
case map0 target of
None ⇒ hv |
Some vidx ⇒
if (vidx = targetOld) then
set-active-vcpu-req hv cidx vidx
else
hv
else
hv

definition *vcpu-pool-pop-through-vmid-req :: HV ⇒ cpu-id ⇒ vm-id ⇒ HV × bool*
where *vcpu-pool-pop-through-vmid-req hv cid vmid ≡*
let
map0 = (cpu-map-idByVM (cpu-maps hv))!cid;
vcpuPool = CPU.vcpus ((cpu hv)!cid)
in
if(length vcpuPool = 0 ∨ map0 vmid = None) then
(hv,False)

```
      else
        (hv,True)
```

```
definition cpu-idle-req :: HV ⇒ cpu-id ⇒ HV where
  cpu-idle-req hv cid ≡
    let
      wks = (cpu-wk-st hv)[cid:=CPU-S-IDLE]
    in
      hv(|cpu-wk-st:=wks|)
```

```
definition vcpu-shutdown-req :: HV ⇒ cpu-id ⇒ vcpu-id ⇒ HV  where
   vcpu-shutdown-req hv cid vcid ≡
      let
        map1 = (CPU-MAPS.cpu-map-active (cpu-maps hv)) ! cid;
        map2 = (cpu-is-haveVCPU (cpu-maps hv)) ! cid;
        cpu0 = (cpu hv)!cid;
        ret = map2 vcid;
        limit = length (vcpu hv)
      in
        case ret of
            None ⇒ cpu-idle-req hv cid |
            Some idx ⇒
              let
                aN = map1 limit;
                vs = (vcpu-wk-st hv)[vcid:=VCPU-S-INV];
                cpu0 = cpu0(|running-num:=running-num cpu0−1|);
                hv1 = hv(|vcpu-wk-st:=vs,cpu:=(cpu hv)[cid:=cpu0]|)
              in
                if(running-num cpu0 = 0) then
                  cpu-idle-req hv1 cid
                else
                  if(vcid=active-vcpu cpu0) then
                      if(aN = None ∨ (the aN) = active-vcpu cpu0) then
                        hv1
                      else
                        hv1(|vcpu-wk-st:=(vcpu-wk-st hv1)[the aN:=VCPU-S-ACT],
                            cpu:=(cpu hv1)[cid:= cpu0(|active-vcpu:=the aN|)]|)
                  else
                    hv1
```

```
definition get-vmid-cid ::  HV ⇒ cpu-id ⇒ vm-id  where
 get-vmid-cid hv cid ≡
```

```
            let
              vcid = active-vcpu  ((cpu hv)!cid);
              vcpu = (vcpu hv)!vcid
             in  vmID vcpu
```

```
definition get-cid-vmid ::  HV ⇒ vm-id ⇒ cpu-id  where
  get-cid-vmid hv vmid ≡
           let
            cpuS = cpu hv;
            idx = find-index (λx.(get-vmid-cid hv (CPU.id x)) = vmid ) cpuS
           in idx
```

```
definition vmm-shutdown-vm-req :: HV ⇒ cpu-id ⇒HV where
  vmm-shutdown-vm-req hv cidx ≡
   let
     vcidx = active-vcpu ((cpu hv)!cidx);
     vmidx = vmID((vcpu hv)!vcidx);
     ws = vm-wk-st hv;
     hv1 = hv(|vm-wk-st:=ws[vmidx:=VM-S-INV]|) ;
     map1 = (CPU-MAPS.cpu-map-active (cpu-maps hv)) ! cidx;
     map2 = (cpu-is-haveVCPU (cpu-maps hv)) ! cidx;
     cpu0 = (cpu hv)!cidx;
     limit = length (vcpu hv);
     aN = map1 limit;
     vs = (vcpu-wk-st hv)[vcidx:=VCPU-S-INV];
     cpu0 = cpu0(|running-num:=running-num cpu0−1|);
     hv2 = hv1(|vcpu-wk-st:=vs,cpu:=(cpu hv)[cidx:=cpu0]|)
   in
       if(running-num cpu0 = 0) then
         cpu-idle-req hv2 cidx
       else
         if(aN = None ) then
           hv2
         else
           hv2(|vcpu-wk-st:=(vcpu-wk-st hv2)[the aN:=VCPU-S-ACT],
               cpu:=(cpu hv2)[cidx:= cpu0(|active-vcpu:=the aN|)]|)
```

```
definition vmm-reset-vm-req :: HV ⇒ cpu-id ⇒HV where
  vmm-reset-vm-req hv cidx ≡
   let
     vcidx = active-vcpu ((cpu hv)!cidx);
```

```
    vmidx = vmID((vcpu hv)!vcidx);
    vmws = vm-wk-st hv;
    vmwsN = vmws[vmidx:=VM-S-ACT];
    cws = cpu-wk-st hv;
    cwsN = cws[cidx:=CPU-S-RUN]
  in
    hv(|cpu-wk-st:=cwsN,vm-wk-st:=vmwsN|)
```

definition *vm-vcpuid-to-pcpuid-req* :: *HV* ⇒ *vm-id* ⇒ *vcpu-idx* ⇒ *HV* × *cpu-id option* **where**

```
 vm-vcpuid-to-pcpuid-req hv vmid vidx ≡
  let
    vm0 = (vm hv)!vmid;
    vcpus0 = vcpus vm0
  in
    if(vidx<length vcpus0) then
        (hv, Some (physID ((vcpu hv)!(vcpus0 !vidx))))
    else
      (hv,None)
```

definition *vm-pcpuid-to-vcpuid-req* :: *HV* ⇒ *vm-id* ⇒ *cpu-id* ⇒ *HV* × *vcpu-idx option* **where**

```
 vm-pcpuid-to-vcpuid-req hv vmid cid ≡
  let
    vm0 = (vm hv)!vmid;
    vcpus = vcpus vm0;
    map0 = (get-vcpuIDX (cpu-maps hv))!cid
  in
    (hv,map0 vcpus)
```

definition *vm-pa2ipa-req* :: *HV* ⇒ *vm-id* ⇒ *u64* ⇒ *HV* × *u64* **where**

```
  vm-pa2ipa-req hv vmid pa≡
   if (pa = 0) then
    (hv,0)
   else
    let
      regions = address ((vm hv)!vmid);
      ret = ((get-paIdX (vm-regionMaps hv)) ! vmid) pa
    in
      case ret of
         None ⇒ (hv,0)|
```

Some idx ⇒ (*hv*,(*pa* + *offset* (*regions*!*idx*)))

definition *vm-ipa2pa-req* :: *HV* ⇒ *vm-id* ⇒ *u64* ⇒ *HV* × *u64* **where**
vm-ipa2pa-req hv vmid ipa≡
if (*ipa* = *0*) *then*
(*hv*,*0*)
else
let
regions = *address* ((*vm hv*)!*vmid*);
ret = ((*get-ipaIdX* (*vm-regionMaps hv*)) ! *vmid*) *ipa*
in
case ret of
None ⇒ (*hv*,*0*)|
Some idx ⇒ (*hv*,(*ipa* − *offset* (*regions*!*idx*)))

definition *interrupt-is-reserved-req* :: *HV* ⇒ *interrupt-id* ⇒ (*HV*×*bool*) **where**
interrupt-is-reserved-req hv n ≡ (*hv*, (*Interrupt-hyper-bitmap* (*interrupts hv*))!*n*)

definition *interrupt-arch-conflict* :: *bitmap* ⇒ *interrupt-id* ⇒ *bool* **where**
interrupt-arch-conflict bmp n ≡ (*bmp*!*n*)

definition *interrupt-reserve-int-req* :: *HV* ⇒ *interrupt-id* ⇒ *handlers* ⇒ *HV* **where**
interrupt-reserve-int-req hv int-id hdl≡
if int-id < (*length* [(*Interrupts.Interrupt-glb-bitmap* (*interrupts hv*))]) *then*
let
x = (*Interrupts.Interrupt-handlers* (*interrupts hv*));
bp = (*Interrupts.Interrupt-glb-bitmap* (*interrupts hv*));
hp = (*Interrupts.Interrupt-hyper-bitmap* (*interrupts hv*))
in
hv(|*interrupts*:=(|*Interrupt-hyper-bitmap*=*hp* [*int-id*:=*True*],
Interrupt-glb-bitmap=*bp* [*int-id*:=*True*],
Interrupt-handlers=*x* [*int-id* := *hdl*]|)|)
else
hv

definition *interrupt-cpu-enable-req* ::*HV* ⇒ *interrupt-id* ⇒ *bool* ⇒ *HV* **where**
interrupt-cpu-enable-req hv int-id en ≡ *let*
c = (!) (*cpu hv*) *int-id*
in
if en then

hv⦇cpu-wk-st:=((cpu-wk-st hv)[int-id:=CPU-S-RUN])⦈
else
hv⦇cpu-wk-st:=((cpu-wk-st hv)[int-id:=CPU-S-IDLE])⦈

definition *interrupt-cpu-ipi-send-req* :: *HV* ⇒ *cpu-id* ⇒ *interrupt-id* ⇒ *HV* **where**
interrupt-cpu-ipi-send-req hv t ip ≡ *let*
x = (!) (*cpu-wk-st hv*) *t*
in
hv⦇*cpu-wk-st*:=((*cpu-wk-st hv*)[*ip*:=(*CPU-S-IDLE*)])⦈

definition *vm-has-interrupt* :: *VM* ⇒ *interrupt-id* ⇒ *bool* **where**
vm-has-interrupt v n ≡ (*int-bitmap v*)! *n* = *True*

definition *interrupt-vm-inject-req* :: *HV* ⇒ *vm-id* ⇒ *interrupt-id* ⇒ *HV* **where**
interrupt-vm-inject-req hv vi int-id ≡ *let*
v = (*vm hv*)!*vi*;
x = *v*⦇*int-bitmap*:=((*int-bitmap v*)[*int-id*:=*True*])⦈
in
(*hv*⦇*vm*:=((*vm hv*)[*int-id*:=*x*])⦈)

definition *interrupt-vm-register-req* :: *HV* ⇒ *interrupt-src* ⇒ *interrupt-id* ⇒ *HV* ×*bool* **where**
interrupt-vm-register-req hv src int-id ≡
if (*interrupt-arch-conflict* (*Interrupts.Interrupt-glb-bitmap* (*interrupts hv*)) *int-id*) *then*
(*hv* , *False*)
else
let
v = (*vm hv*)!*src*;
iv = *v*⦇*int-bitmap*:=(*int-bitmap v*)[*int-id*:=*True*]⦈;
gh = (*interrupts hv*);
h = *gh*⦇*Interrupt-glb-bitmap*:=(*int-bitmap v*)[*int-id*:=*True*]⦈
in
((*hv*⦇*vm*:=(*vm hv*)[*int-id*:=*iv*], *interrupts*:=*h*⦈), *True*)

definition *interrupt-handler-req* :: *HV* ⇒ *interrupt-id* ⇒ *interrupt-src* ⇒ (*HV* × *bool option*) **where**
interrupt-handler-req hv int-id src ≡
if (*vm-has-interrupt* ((*vm hv*)!*src*) *int-id*) *then*
((*interrupt-vm-inject-req hv* (*get-vmid-cid hv src*) *int-id*), *Some False*)
else
if (*snd* (*interrupt-is-reserved-req hv int-id*)) *then*
let
ix = (*interrupts hv*);
hdl = (*Interrupt-handlers ix*)@[(*irq-handler-t int-id src*)];
hext = [(*irq-handler-t int-id src*)]
in
(*hv*⦇*interrupts*:=*ix*⦇*Interrupt-handlers*:=*hdl*@*hext*⦈⦈, *Some True*)

```
    else (hv, None)


value find-index (λx.(x=2)) [9..15]


definition insert-channel-msg-req :: Commu ⇒ channel-id ⇒ MSG ⇒ Commu
  where insert-channel-msg-req c idx mesg ≡
        let
          ivc0 = ivc c;
          channel0 = ((channels ivc0) ! idx)(|msg:=mesg|);
          channelsN = (channels ivc0)[idx:=channel0];
          ivcN = ivc0(|channels:=channelsN|)
        in
          c(|ivc:=ivcN|)


definition ivc-send-msg-req :: HV ⇒ cpu-id ⇒ vm-id ⇒ MSG ⇒ HV × bool
  where ivc-send-msg-req hv cid vm-tgrt mesg ≡
      if (get-vmid-cid-req cid hv = vm-tgrt) then
        (hv,False)
      else
        let
          commu0 = commu hv ;
          vm-src = get-vmid-cid-req cid hv;
          ret = (getChannel commu0) vm-src vm-tgrt
        in
          if (ret = None) then
            (hv,False)
          else
            let
              cN = insert-channel-msg-req commu0 (the ret) mesg
            in
              (hv(|commu:=cN|),True)


definition vm-init-channel-req :: HV ⇒ vm-port ⇒ vm-id ⇒ HV×bool
  where vm-init-channel-req hv pt vmid ≡
        let
          channels = channels (ivc (commu hv));
          nums = channel-num (ivc (commu hv));
          ret = (availChannel (commu hv)) pt vmid
        in
          if (ret = None) then
            if (nums ≥ MAX-CHANNEL-NUM) then
```

```
          (hv,False)
        else
          let
            idx = nums;
            channel0 = if (vm-port.type pt = RECEIVE) then
                          (channels ! idx)(|flag:=True,portDes:= Some pt|)
                       else
                          (channels ! idx)(|flag:=True,portSrc:= Some pt|);
            channelsN = channels[idx:=channel0];
            numsN = nums+1;
        ivcN = (ivc (commu hv))(|channels:=channelsN,channel-num:=numsN|);
            commuN = (commu hv)(|ivc:=ivcN|)
          in
            (hv(|commu:=commuN|),True)
      else
          let
            idx = the ret;
            channel0 = if (vm-port.type pt = RECEIVE) then
                          (channels ! idx)(|portDes:= Some pt|)
                       else
                          (channels ! idx)(|portSrc:= Some pt|);
            channelsN = channels[idx:=channel0];
            ivcN = (ivc (commu hv))(|channels:=channelsN|);
            commuN = (commu hv)(|ivc:=ivcN|)
          in
            (hv(|commu:=commuN|),True)
```

definition *init-port-in-channel-req* :: $HV \Rightarrow$ *vm-id* $\Rightarrow$ $HV \times nat$

```
  where init-port-in-channel-req hv idx ≡
    let
      channels = channels (ivc (commu hv));
      i = find-index (λx. flag x) channels
    in
      (hv,i)
```

definition *cpu-schedule-req* :: $HV \Rightarrow$ *cpu-id* $\Rightarrow HV$ **where**

```
  cpu-schedule-req hv cid ≡
    let
        cpu0= (cpu hv)!cid;
        map1 = (CPU-MAPS.cpu-map-active (cpu-maps hv)) ! cid;
        limit = length (vcpu hv);
```

```
    aN = map1 limit;
    actOld = active-vcpu cpu0
  in
    if (running-num cpu0)>1 then
    hv⦇vcpu-wk-st:=(vcpu-wk-st hv)[actOld:=VCPU-S-PEND,the aN:=VCPU-S-ACT],
          cpu:=(cpu hv)[cid:= cpu0⦇active-vcpu:=the aN⦈]⦈
    else hv
```

definition *t* :: *int list* **where**

```
t ≡ [1,2,3]
```

definition *audit-append-event* :: *HV* ⇒ *Audit* ⇒ *HV* **where**

```
 audit-append-event hv a ≡
    let
      as = audit hv
    in hv⦇audit:= a#as⦈
```

definition *vmm-list-vm-info-req* :: *HV* ⇒ *HV* × *VM-INFO list* **where**

```
 vmm-list-vm-info-req hv ≡
   let
     vmZ = List.zip (vm hv) (vm-wk-st hv);
     ret = List.map
        (λx. ⦇vmId=VM.id (fst x), vmName=VM.name (fst x), vmType=VM.type
(fst x),
                vmState=snd x⦈)
           vmZ
   in
     (hv,ret)
```

definition *vmm-get-vm-id-req* :: *HV* ⇒ *vm-name* ⇒ *HV* × *vm-id option* **where**

```
  vmm-get-vm-id-req hv name0 ≡
        let
          vms = (vm hv);
          limit = length vms;
          idx = find-index (λx. (name x=name0)) vms
        in
          if idx<limit then
             (hv,Some idx)
          else
             (hv,None)
```

end

theory *Value-Abbreviation*

imports *Main*

keywords *value-abbreviation* :: *thy-decl*

begin

Computing values and saving as abbreviations.

Useful in program verification to handle some configuration constant (e.g. n = 4) which may change. This mechanism can be used to give names (abbreviations) to other related constants (e.g. $2^{\,n}, 2^n{-}1, [1..n], rev[1..n])whichmayappearrepeatedly.$

ML ‹

```
structure Value-Abbreviation = struct
fun value-and-abbreviation mode name expr int ctxt = let
    val decl = (name, NONE, Mixfix.NoSyn)
    val expr = Syntax.read-term ctxt expr
    val eval-expr =Value-Command.value ctxt expr
    val lhs = Free (Binding.name-of name, fastype-of expr)
    val eq = Logic.mk-equals (lhs, eval-expr)
    val ctxt = Specification.abbreviation mode (SOME decl) [] eq int ctxt
    val pretty-eq = Syntax.pretty-term ctxt eq
  in Pretty.writeln pretty-eq; ctxt end

val - =
  Outer-Syntax.local-theory' @{command-keyword value-abbreviation}
    setup abbreviation for evaluated value
    (Parse.syntax-mode -- Parse.binding -- Parse.term
      >> (fn ((mode, name), expr) => value-and-abbreviation mode name expr));

end
```

›

Testing it out. Unfortunately locale/experiment/notepad all won't work here because the code equation setup is all global.

definition
value-abbreviation-test-config-constant-1 = (*24* :: *nat*)

definition
value-abbreviation-test-config-constant-2 = (*5* :: *nat*)

value-abbreviation (*input*)
value-abbreviation-test-important-magic-number
((*2* :: *int*) ^ *value-abbreviation-test-config-constant-1*)

− (2 ^ value-abbreviation-test-config-constant-2)

value-abbreviation (*input*)
value-abbreviation-test-range-of-options
rev [*int value-abbreviation-test-config-constant-2*
.. *int value-abbreviation-test-config-constant-1*]

end

theory *Match-Abbreviation*

imports *Main*

keywords *match-abbreviation* :: *thy-decl*
and *reassoc-thm* :: *thy-decl*

begin

Splicing components of terms and saving as abbreviations. See the example at the bottom for explanation/documentation.

ML ‹

```
structure Match-Abbreviation = struct

fun app-cons-dummy cons x y
  = Const (cons, dummyT) $ x $ y

fun lazy-lam x t = if Term.exists-subterm (fn t' => t' aconv x) t
    then lambda x t else t

fun abs-dig-f ctxt lazy f (Abs (nm, T, t))
  = let
    val (nms, ctxt) = Variable.variant-fixes [nm] ctxt
    val x = Free (hd nms, T)
    val t = betapply (Abs (nm, T, t), x)
    val t' = f ctxt t
  in if lazy then lazy-lam x t' else lambda x t' end
  | abs-dig-f - - - t = raise TERM (abs-dig-f: not abs, [t])

fun find-term1 ctxt get (f $ x)
  = (get ctxt (f $ x) handle Option => (find-term1 ctxt get f
      handle Option => find-term1 ctxt get x))
  | find-term1 ctxt get (a as Abs -)
  = abs-dig-f ctxt true (fn ctxt => find-term1 ctxt get) a
  | find-term1 ctxt get t = get ctxt t

fun not-found pat t = raise TERM (pattern not found, [pat, t])

fun find-term ctxt get pat t = find-term1 ctxt get t
  handle Option => not-found pat t
```

```
fun lambda-frees-vars ctxt ord-t t = let
   fun is-free t = is-Free t andalso not (Variable.is-fixed ctxt (Term.term-name t))
   fun is-it t = is-free t orelse is-Var t
   val get = fold-aterms (fn t => if is-it t then insert (=) t else I)
   val all-vars = get ord-t []
   val vars = get t []
   val ord-vars = filter (member (=) vars) all-vars
  in fold lambda ord-vars t end

fun parse-pat-fixes ctxt fixes pats = let
   val (-, ctxt') = Variable.add-fixes
         (map (fn (b, -, -) => Binding.name-of b) fixes) ctxt
   val read-pats = Syntax.read-terms ctxt' pats
  in Variable.export-terms ctxt' ctxt read-pats end

fun add-reassoc name rhs fixes thms-info ctxt = let
   val thms = Attrib.eval-thms ctxt thms-info
   val rhs-pat = singleton (parse-pat-fixes ctxt fixes) rhs
     |> Thm.cterm-of ctxt
   val rew = Simplifier.rewrite (clear-simpset ctxt addsimps thms) rhs-pat
     |> Thm.symmetric
   val (-, ctxt) = Local-Theory.note ((name, []), [rew]) ctxt
   val pretty-decl = Pretty.block [Pretty.str (Binding.name-of name ^ :\n),
       Thm.pretty-thm ctxt rew]
  in Pretty.writeln pretty-decl; ctxt end

fun dig-f ctxt repeat adj (f $ x) = (adj ctxt (f $ x)
   handle Option => (dig-f ctxt repeat adj f
          $ (if repeat then (dig-f ctxt repeat adj x
             handle Option => x) else x)
      handle Option => f $ dig-f ctxt repeat adj x))
  | dig-f ctxt repeat adj (a as Abs -)
    = abs-dig-f ctxt false (fn ctxt => dig-f ctxt repeat adj) a
  | dig-f ctxt - adj t = adj ctxt t

fun do-rewrite ctxt repeat rew-pair t = let
   val thy = Proof-Context.theory-of ctxt
   fun adj - t = case Pattern.match-rew thy t rew-pair
     of NONE => raise Option | SOME (t', -) => t'
  in dig-f ctxt repeat adj t
   handle Option => not-found (fst rew-pair) t end

fun select-dig ctxt [] f t = f ctxt t
  | select-dig ctxt (p :: ps) f t = let
   val thy = Proof-Context.theory-of ctxt
   fun do-rec ctxt t = if Pattern.matches thy (p, t)
     then select-dig ctxt ps f t else raise Option
  in dig-f ctxt false do-rec t handle Option => not-found p t end
```

```
fun ext-dig-lazy ctxt f (a as Abs -)
  = abs-dig-f ctxt true (fn ctxt => ext-dig-lazy ctxt f) a
  | ext-dig-lazy ctxt f t = f ctxt t

fun report-adjust ctxt nm t = let
    val pretty-decl = Pretty.block [Pretty.str (nm ^ , have:\n),
        Syntax.pretty-term ctxt t]
  in Pretty.writeln pretty-decl; t end

fun do-adjust ctxt (((select, []), [p]), fixes) t = let
    val p = singleton (parse-pat-fixes ctxt fixes) p
    val thy = Proof-Context.theory-of ctxt
    fun get - t = if Pattern.matches thy (p, t) then t else raise Option
    val t = find-term ctxt get p t
  in report-adjust ctxt Selected t end
  | do-adjust ctxt (((retype-consts, []), consts), []) t = let
    fun get-constname (Const (s, -)) = s
      | get-constname (Abs (-, -, t)) = get-constname t
      | get-constname (f $ -) = get-constname f
      | get-constname - = raise Option
    fun get-constname2 t = get-constname t
      handle Option => raise TERM (do-adjust: no constant, [t])
    val cnames = map (get-constname2 o Syntax.read-term ctxt) consts
      |> Symtab.make-set
    fun adj (Const (cn, T)) = if Symtab.defined cnames cn
        then Const (cn, dummyT) else Const (cn, T)
      | adj t = t
    val t = Syntax.check-term ctxt (Term.map-aterms adj t)
  in report-adjust ctxt Adjusted types t end
  | do-adjust ctxt (((r, in-selects), [from, to]), fixes) t = if
        r = rewrite1 orelse r = rewrite then let
    val repeat = r <> rewrite1
    val sel-pats = map (fn (p, fixes) => singleton (parse-pat-fixes ctxt fixes) p)
        in-selects
    val rewrite-pair = case parse-pat-fixes ctxt fixes [from, to]
      of [f, t] => (f, t) | - => error (do-adjust: unexpected length)
    val t = ext-dig-lazy ctxt (fn ctxt => select-dig ctxt sel-pats
        (fn ctxt => do-rewrite ctxt repeat rewrite-pair)) t
  in report-adjust ctxt (if repeat then Rewrote else Rewrote (repeated)) t end
  else error (do-adjust: unexpected:  ^ r)
  | do-adjust - args - = error (do-adjust: unexpected:  ^ @{make-string} args)

fun unvarify-types-same ty = ty
  |> Term-Subst.map-atypsT-same
    (fn TVar ((a, i), S) => TFree (a ^ -var- ^ string-of-int i, S)
      | - => raise Same.SAME)

fun unvarify-types tm = tm
```

```
  |> Same.commit (Term-Subst.map-types-same unvarify-types-same)

fun match-abbreviation mode name init adjusts int ctxt = let
    val init-term = init ctxt
    val init-lambda = lambda-frees-vars ctxt init-term init-term
      |> unvarify-types
      |> Syntax.check-term ctxt
    val decl = (name, NONE, Mixfix.NoSyn)
    val result = fold (do-adjust ctxt) adjusts init-lambda
    val lhs = Free (Binding.name-of name, fastype-of result)
    val eq = Logic.mk-equals (lhs, result)
    val ctxt = Specification.abbreviation mode (SOME decl) [] eq int ctxt
    val pretty-eq = Syntax.pretty-term ctxt eq
  in Pretty.writeln pretty-eq; ctxt end

fun from-thm f thm-info ctxt = let
    val thm = singleton (Attrib.eval-thms ctxt) thm-info
  in f thm end

fun from-term term-str ctxt = Syntax.parse-term ctxt term-str

val init-term-parse = Parse.$$$ in |--
    ((Parse.reserved concl |-- Parse.thm >> from-thm Thm.concl-of)
        || (Parse.reserved thm-prop |-- Parse.thm >> from-thm Thm.prop-of)
        || (Parse.term >> from-term)
    )

val term-to-term = (Parse.term -- (Parse.reserved to |-- Parse.term))
    >> (fn (a, b) => [a, b])

val p-for-fixes = Scan.optional
    (Parse.$$$ ( |-- Parse.for-fixes --| Parse.$$$ )) []

val adjust-parser = Parse.and-list1
     ((Parse.reserved select -- Scan.succeed [] -- (Parse.term >> single) --
p-for-fixes)
        || (Parse.reserved retype-consts -- Scan.succeed []
            -- Scan.repeat Parse.term -- Scan.succeed [])
        || ((Parse.reserved rewrite1 || Parse.reserved rewrite)
            -- Scan.repeat (Parse.$$$ in |-- Parse.term -- p-for-fixes)
            -- term-to-term -- p-for-fixes)
    )

(* install match-abbreviation. see below for examples/docs *)
val - =
  Outer-Syntax.local-theory' @{command-keyword match-abbreviation}
    setup abbreviation for subterm of theorem
    (Parse.syntax-mode -- Parse.binding
        -- init-term-parse -- adjust-parser
```

```
    >> (fn (((mode, name), init), adjusts)
          => match-abbreviation mode name init adjusts));

val - =
  Outer-Syntax.local-theory @{command-keyword reassoc-thm}
    store a reassociate−theorem
    (Parse.binding -- Parse.term -- p-for-fixes -- Scan.repeat Parse.thm
      >> (fn (((name, rhs), fixes), thms)
          => add-reassoc name rhs fixes thms));
end
›
```

The match/abbreviate command. There are examples of all elements below, and an example involving monadic syntax in the theory Match-Abbreviation-Test.

Each invocation is match abbreviation, a syntax mode (e.g. (input)), an abbreviation name, a term specifier, and a list of adjustment specifiers.

A term specifier can be term syntax or the conclusion or proposition of some theorem. Examples below.

Each adjustment is a select, a rewrite, or a constant retype.

The select adjustment picks out the part of the term matching the pattern (examples below). It picks the first match point, ordered in term order with compound terms before their subterms and functions before their arguments.

The rewrite adjustment uses a pattern pair, and rewrites instances of the first pattern into the second. The match points are found in the same order as select. The "in" specifiers (examples below) limit the rewriting to within some matching subterm, specified with pattern in the same way as select. The rewrite1 variant only rewrites once, at the first matching site.

The rewrite mechanism can be used to replace terms with terms of different types. The retype adjustment can then be used to repair the term by resetting the types of all instances of the named constants. This is used below with list constructors, to assemble a new list with a different element type.

experiment begin

Fetching part of the statement of a theorem.

match-abbreviation (*input*) *fixp-thm-bit*
  in *thm-prop fixp-induct-tailrec*
  select $X \equiv Y$ (**for** X Y)

Ditto conclusion.

match-abbreviation (*input*) *rev-simps-bit*
  in *concl rev.simps(2)*
  select X (**for** X)

Selecting some conjuncts and reorienting an equality.

match-abbreviation (*input*) *conjunct-test*
  in ($P \wedge Q \wedge P \wedge P \wedge P \wedge ((1 :: nat) = 2) \wedge Q \wedge Q$, [*Suc 0*, *0*])
  select $Q \wedge Z$ (**for** Z)
  and *rewrite* $x = y$ *to* $y = x$ (**for** $x\ y$)
  and *rewrite* **in** $x = y$ & Z (**for** $x\ y\ Z$)
    $A \wedge B$ *to* A (**for** $A\ B$)

The relevant reassociate theorem, that rearranges a conjunction like the above to group the elements selected.

reassoc-thm *conjunct-test-reassoc*
  conjunct-test $P\ Q \wedge Z$ (**for** $P\ Q\ Z$)
  conj-assoc

Selecting some elements of a list, and then replacing tuples with equalities, and adjusting the type of the list constructors so the new term is type correct.

match-abbreviation (*input*) *list-test*
  in [(*Suc 1*, *Suc 2*), (*4*, *5*), (*6*, *7*), (*8*, *9*), (*10*, *11*), (x, y), (*6*, *7*),
    (*18*, *19*), a, a, a, a, a, a, a]
  select (*4*, V) # *xs* (**for** V *xs*)
  and *rewrite* (x, y) *to* (y, x) (**for** $x\ y$)
  and *rewrite1* **in** (*9*, V) # *xs* (**for** V *xs*) **in** (*7*, V) # *xs* (**for** V *xs*)
    x # *xs* *to* [x] (**for** x *xs*)
  and *rewrite* (x, y) *to* $x = y$ (**for** $x\ y$)
  and *retype-consts Cons Nil*

end

end

theory *Subgoal-Methods*
imports *Main*
begin
ML ‹

```
signature SUBGOAL-METHODS =
sig
  val fold-subgoals: Proof.context -> bool -> thm -> thm
  val unfold-subgoals-tac: Proof.context -> tactic
  val distinct-subgoals: Proof.context -> thm -> thm
end;

structure Subgoal-Methods: SUBGOAL-METHODS =
struct

fun max-common-prefix eq (ls :: lss) =
      let
```

```
        val ls' = tag-list 0 ls;
        fun all-prefix (i,a) =
          forall (fn ls' => if length ls' > i then eq (a, nth ls' i) else false) lss
        val ls'' = take-prefix all-prefix ls'
      in map snd ls'' end
  | max-common-prefix - [] = [];

fun push-outer-params ctxt th =
  let
    val ctxt' = ctxt
      |> Simplifier.empty-simpset
      |> Simplifier.add-simp Drule.norm-hhf-eq;
  in
    Conv.fconv-rule
      (Raw-Simplifier.rewrite-cterm (true, false, false) (K (K NONE)) ctxt') th
  end;

fun fix-schematics ctxt raw-st =
  let
    val ((schematic-types, [st']), ctxt1) = Variable.importT [raw-st] ctxt;
    val ((-, inst), ctxt2) =
      Variable.import-inst true [Thm.prop-of st'] ctxt1;

    val schematic-terms = map (apsnd (Thm.cterm-of ctxt2)) inst;
    val schematics = (schematic-types, schematic-terms);

  in (Thm.instantiate schematics st', ctxt2) end

val strip-params = Term.strip-all-vars;
val strip-prems = Logic.strip-imp-prems o Term.strip-all-body;
val strip-concl = Logic.strip-imp-concl o Term.strip-all-body;



fun fold-subgoals ctxt prefix raw-st =
  if Thm.nprems-of raw-st < 2 then raw-st
  else
    let
      val (st, inner-ctxt) = fix-schematics ctxt raw-st;

      val subgoals = Thm.prems-of st;
      val paramss = map strip-params subgoals;
      val common-params = max-common-prefix (eq-snd (op =)) paramss;

      fun strip-shift subgoal =
        let
          val params = strip-params subgoal;
          val diff = length common-params − length params;
          val prems = strip-prems subgoal;
```

```
      in map (Term.incr-boundvars diff) prems end;

    val premss = map (strip-shift) subgoals;

    val common-prems = max-common-prefix (op aconv) premss;

    val common-params = if prefix then common-params else [];
    val common-prems = if prefix then common-prems else [];

    fun mk-concl subgoal =
      let
        val params = Term.strip-all-vars subgoal;
        val local-params = drop (length common-params) params;
        val prems = strip-prems subgoal;
        val local-prems = drop (length common-prems) prems;
        val concl = strip-concl subgoal;
      in Logic.list-all (local-params, Logic.list-implies (local-prems, concl)) end;

    val goal =
      Logic.list-all (common-params,
      (Logic.list-implies (common-prems,Logic.mk-conjunction-list (map mk-concl
subgoals))));

    val chyp = Thm.cterm-of inner-ctxt goal;

    val (common-params',inner-ctxt') =
      Variable.add-fixes (map fst common-params) inner-ctxt
      |>> map2 (fn (-, T) => fn x => Thm.cterm-of inner-ctxt (Free (x, T)))
common-params;

    fun try-dest rule =
       try (fn () => (@{thm conjunctionD1} OF [rule], @{thm conjunctionD2}
OF [rule])) ();

    fun solve-headgoal rule =
      let
        val rule' = rule
          |> Drule.forall-intr-list common-params'
          |> push-outer-params inner-ctxt';
      in
        (fn st => Thm.implies-elim st rule')
      end;

    fun solve-subgoals rule' st =
      (case try-dest rule' of
        SOME (this, rest) => solve-subgoals rest (solve-headgoal this st)
      | NONE => solve-headgoal rule' st);

    val rule = Drule.forall-elim-list common-params' (Thm.assume chyp);
```

```
    in
      st
      |> push-outer-params inner-ctxt
      |> solve-subgoals rule
      |> Thm.implies-intr chyp
      |> singleton (Variable.export inner-ctxt' ctxt)
    end;

fun distinct-subgoals ctxt raw-st =
  let
    val (st, inner-ctxt) = fix-schematics ctxt raw-st;
    val subgoals = Drule.cprems-of st;
    val atomize = Conv.fconv-rule (Object-Logic.atomize-prems inner-ctxt);

    val rules =
      map (atomize o Raw-Simplifier.norm-hhf inner-ctxt o Thm.assume) subgoals
      |> sort (int-ord o apply2 Thm.nprems-of);

    val st' = st
      |> ALLGOALS (fn i =>
        Object-Logic.atomize-prems-tac inner-ctxt i THEN solve-tac inner-ctxt rules
i)
      |> Seq.hd;

    val subgoals' = subgoals
      |> inter (op aconvc) (Thm.chyps-of st')
      |> distinct (op aconvc);
  in
    Drule.implies-intr-list subgoals' st'
    |> singleton (Variable.export inner-ctxt ctxt)
  end;

(* Variant of filter-prems-tac that recovers premise order *)
fun filter-prems-tac' ctxt pred =
  let
    fun Then NONE tac = SOME tac
      | Then (SOME tac) tac' = SOME (tac THEN' tac');
    fun thins H (tac, n, i) =
      (if pred H then (tac, n + 1, i)
       else (Then tac (rotate-tac n THEN' eresolve-tac ctxt [thin-rl]), 0, i + n));
  in
    SUBGOAL (fn (goal, i) =>
      let val Hs = Logic.strip-assums-hyp goal in
        (case fold thins Hs (NONE, 0, 0) of
          (NONE, -, -) => no-tac
        | (SOME tac, -, n) => tac i THEN rotate-tac (~ n) i)
      end)
  end;
```

```
fun trim-prems-tac ctxt rules =
let
  fun matches (prem,rule) =
  let
    val ((-,prem′),ctxt′) = Variable.focus NONE prem ctxt;
    val rule-prop = Thm.prop-of rule;
  in Unify.matches-list (Context.Proof ctxt′) [rule-prop] [prem′] end;

in filter-prems-tac′ ctxt (not o member matches rules) end;

val adhoc-conjunction-tac = REPEAT-ALL-NEW
  (SUBGOAL (fn (goal, i) =>
    if can Logic.dest-conjunction (Logic.strip-imp-concl goal)
    then resolve0-tac [Conjunction.conjunctionI] i
    else no-tac));

fun unfold-subgoals-tac ctxt =
  TRY (adhoc-conjunction-tac 1)
  THEN (PRIMITIVE (Raw-Simplifier.norm-hhf ctxt));

val - =
  Theory.setup
   (Method.setup @{binding fold-subgoals}
      (Scan.lift (Args.mode prefix) >> (fn prefix => fn ctxt =>
         SIMPLE-METHOD (PRIMITIVE (fold-subgoals ctxt prefix))))
      lift all subgoals over common premises/params #>
    Method.setup @{binding unfold-subgoals}
      (Scan.succeed (fn ctxt => SIMPLE-METHOD (unfold-subgoals-tac ctxt)))
      recover subgoals after folding #>
    Method.setup @{binding distinct-subgoals}
    (Scan.succeed (fn ctxt => SIMPLE-METHOD (PRIMITIVE (distinct-subgoals
ctxt))))
     trim all subgoals to be (logically) distinct #>
    Method.setup @{binding trim}
      (Attrib.thms >> (fn thms => fn ctxt =>
         SIMPLE-METHOD (HEADGOAL (trim-prems-tac ctxt thms))))
     trim all premises that match the given rules);

end;
›
```

end

theory *Rule-By-Method*
imports
  Main
  HOL−Eisbach.Eisbach-Tools
begin

ML ‹

```
signature RULE-BY-METHOD =
sig
  val rule-by-tac: Proof.context -> {vars : bool, prop: bool} ->
    (Proof.context -> tactic) -> (Proof.context -> tactic) list -> Position.T
-> thm
end;


fun atomize ctxt = Conv.fconv-rule (Object-Logic.atomize ctxt);

fun fix-schematics ctxt raw-st =
  let
    val ((schematic-types, [st']), ctxt1) = Variable.importT [raw-st] ctxt;
    fun certify-inst ctxt inst = map (apsnd (Thm.cterm-of ctxt)) (#2 inst)
    val (schematic-terms, ctxt2) =
      Variable.import-inst true [Thm.prop-of st'] ctxt1
      |>> certify-inst ctxt1;
    val schematics = (schematic-types, schematic-terms);
  in (Thm.instantiate schematics st', ctxt2) end

fun curry-asm ctxt st = if Thm.nprems-of st = 0 then Seq.empty else
let

  val prems = Thm.cprem-of st 1 |> Thm.term-of |> Logic.strip-imp-prems;

  val (thesis :: xs,ctxt') = Variable.variant-fixes (thesis :: replicate (length prems)
P) ctxt;

  val rl =
    xs
    |> map (fn x => Thm.cterm-of ctxt' (Free (x, propT)))
    |> Conjunction.mk-conjunction-balanced
    |> (fn xs => Thm.apply (Thm.apply @{cterm Pure.imp} xs) (Thm.cterm-of
ctxt' (Free (thesis,propT))))
    |> Thm.assume
    |> Conjunction.curry-balanced (length prems)
    |> Drule.implies-intr-hyps

  val rl' = singleton (Variable.export ctxt' ctxt) rl;

  in Thm.bicompose (SOME ctxt) {flatten = false, match = false, incremented =
false}
            (false, rl', 1) 1 st end;

val drop-trivial-imp =
let
  val asm =
```

```
    Thm.assume (Drule.protect @{cprop (PROP A ==> PROP A) ==> PROP A})
    |> Goal.conclude;

in
  Thm.implies-elim  asm (Thm.trivial @{cprop PROP A})
  |> Drule.implies-intr-hyps
  |> Thm.generalize ([],[A]) 1
  |> Drule.zero-var-indexes
 end

val drop-trivial-imp' =
let
  val asm =
    Thm.assume (Drule.protect @{cprop (PROP P ==> A) ==> A})
    |> Goal.conclude;

  val asm' = Thm.assume @{cprop PROP P == Trueprop A}

in
  Thm.implies-elim asm (asm' COMP Drule.equal-elim-rule1)
  |> Thm.implies-elim (asm' COMP Drule.equal-elim-rule2)
  |> Drule.implies-intr-hyps
  |> Thm.permute-prems 0 ~1
  |> Thm.generalize ([],[A,P]) 1
  |> Drule.zero-var-indexes
 end

fun atomize-equiv-tac ctxt i =
  Object-Logic.full-atomize-tac ctxt i
  THEN PRIMITIVE (fn st'  =>
  let val (-,[A,-]) = Drule.strip-comb (Thm.cprem-of st' i) in
  if Object-Logic.is-judgment ctxt (Thm.term-of A) then st'
 else error (Failed to fully atomize result:\n ^ (Syntax.string-of-term ctxt (Thm.term-of
A))) end)


structure Data = Proof-Data
(
  type T = thm list * bool;
  fun init - = ([],false);
);

val empty-rule-prems = Data.map (K ([],true));

fun add-rule-prem thm = Data.map (apfst (Thm.add-thm thm));

fun with-rule-prems enabled parse =
  Scan.state :|-- (fn context =>
  let
```

```
    val context' = Context.proof-of context |> Data.map (K ([Drule.free-dummy-thm],enabled))
                  |> Context.Proof
  in Scan.lift (Scan.pass context' parse) end)


fun get-rule-prems ctxt =
  let
    val (thms,b) = Data.get ctxt
  in if (not b) then [] else thms end


fun zip-subgoal assume tac (ctxt,st : thm) = if Thm.nprems-of st = 0 then Se-
q.single (ctxt,st) else
let
  fun bind-prems st' =
  let
    val prems = Drule.cprems-of st';
    val (asms, ctxt') = Assumption.add-assumes prems ctxt;
    val ctxt'' = fold add-rule-prem asms ctxt';
    val st'' = Goal.conclude (Drule.implies-elim-list st' (map Thm.assume prems));
  in (ctxt'',st'') end

  fun defer-prems st' =
  let
    val nprems = Thm.nprems-of st';
    val st'' = Thm.permute-prems 0 nprems (Goal.conclude st');
  in (ctxt,st'') end;


in
  tac ctxt (Goal.protect 1 st)
  |> Seq.map (if assume then bind-prems else defer-prems)  end


fun zip-subgoals assume tacs pos ctxt st =
let
  val nprems = Thm.nprems-of st;
  val - = nprems < length tacs andalso error (More tactics than rule assumptions
^ Position.here pos);
  val tacs' = map (zip-subgoal assume) (tacs @ (replicate (nprems − length tacs)
(K all-tac)));
  val ctxt' = empty-rule-prems ctxt;
in Seq.EVERY tacs' (ctxt',st) end;

fun rule-by-tac' ctxt {vars,prop} tac asm-tacs pos raw-st =
  let
    val (st,ctxt1) = if vars then (raw-st,ctxt) else fix-schematics ctxt raw-st;

    val ([x],ctxt2) = Proof-Context.add-fixes [(Binding.name Auto-Bind.thesisN,NONE,
```

```
NoSyn)] ctxt1;

    val thesis = if prop then Free (x,propT) else Object-Logic.fixed-judgment ctxt2
x;

    val cthesis = Thm.cterm-of ctxt thesis;

    val revcut-rl' = Thm.instantiate' [] ([NONE,SOME cthesis]) @{thm revcut-rl};

    fun is-thesis t = Logic.strip-assums-concl t aconv thesis;

    fun err thm str = error (str ^ Position.here pos ^ \n ^
      (Pretty.string-of (Goal-Display.pretty-goal ctxt thm)));

    fun pop-thesis st =
    let
      val prems = Thm.prems-of st |> tag-list 0;
      val (i,-) = (case filter (is-thesis o snd) prems of
        [] => err st Lost thesis
        | [x] => x
        | - => err st More than one result obtained);
     in st |> Thm.permute-prems 0 i  end

    val asm-st =
    (revcut-rl' OF [st])
    |> (fn st => Goal.protect (Thm.nprems-of st − 1) st)


    val (ctxt3,concl-st) = case Seq.pull (zip-subgoals (not vars) asm-tacs pos ctxt2
asm-st) of
      SOME (x,-) => x
   | NONE => error (Failed to apply tactics to rule assumptions. ^ (Position.here
pos));

    val concl-st-prepped =
      concl-st
      |> Goal.conclude
       |> (fn st => Goal.protect (Thm.nprems-of st) st |> Thm.permute-prems 0
~1 |> Goal.protect 1)

    val concl-st-result = concl-st-prepped
      |> (tac ctxt3
          THEN (PRIMITIVE pop-thesis)
          THEN curry-asm ctxt
            THEN PRIMITIVE (Goal.conclude #> Thm.permute-prems 0 1 #>
Goal.conclude))

    val result = (case Seq.pull concl-st-result of
      SOME (result,-) => singleton (Proof-Context.export ctxt3 ctxt) result
```

```
      | NONE => err concl-st-prepped Failed to apply tactic to rule conclusion:)

    val drop-rule = if prop then drop-trivial-imp else drop-trivial-imp'

    val result' = ((Goal.protect (Thm.nprems-of result -1) result) RS drop-rule)
    |> (if prop then all-tac else
       (atomize-equiv-tac ctxt (Thm.nprems-of result)
       THEN resolve-tac ctxt @{thms Pure.reflexive} (Thm.nprems-of result)))
    |> Seq.hd
    |> Raw-Simplifier.norm-hhf ctxt

  in Drule.zero-var-indexes result' end;

fun rule-by-tac is-closed ctxt args tac asm-tacs pos raw-st =
 let val f = rule-by-tac' ctxt args tac asm-tacs pos
  in
   if is-closed orelse Context-Position.is-really-visible ctxt then SOME (f raw-st)
   else try f raw-st
 end

fun pos-closure (scan : 'a context-parser) :
  (('a * (Position.T * bool)) context-parser) = (fn (context,toks) =>
  let
     val (((context',x),tr-toks),toks') = Scan.trace (Scan.pass context (Scan.state
-- scan)) toks;
    val pos = Token.range-of tr-toks;
    val is-closed = exists (fn t => is-some (Token.get-value t)) tr-toks
  in ((x,(Position.range-position pos, is-closed)),(context',toks')) end)

val parse-flags = Args.mode schematic -- Args.mode raw-prop >> (fn (b,b') =>
{vars = b, prop = b'})

fun tac m ctxt =
  Method.NO-CONTEXT-TACTIC ctxt
    (Method.evaluate-runtime m ctxt []);

(* Declare as a mixed attribute to avoid any partial evaluation *)

fun handle-dummy f (context, thm) =
  case (f context thm) of SOME thm' => (NONE, SOME thm')
  | NONE => (SOME context, SOME Drule.free-dummy-thm)

val (rule-prems-by-method : attribute context-parser) = Scan.lift parse-flags :--
(fn flags =>
  pos-closure (Scan.repeat1
    (with-rule-prems (not (#vars flags)) Method.text-closure ||
      Scan.lift (Args.$$$ - >> (K Method.succeed-text))))) >>
        (fn (flags,(ms,(pos, is-closed))) => handle-dummy (fn context =>
          rule-by-tac is-closed (Context.proof-of context) flags (K all-tac) (map tac
```

```
ms) pos))

val (rule-concl-by-method : attribute context-parser) = Scan.lift parse-flags :−−
(fn flags =>
  pos-closure (with-rule-prems (not (#vars flags)) Method.text-closure)) >>
    (fn (flags,(m,(pos, is-closed))) => handle-dummy (fn context =>
      rule-by-tac is-closed (Context.proof-of context) flags (tac m) [] pos))

val - = Theory.setup
  (Global-Theory.add-thms-dynamic (@{binding rule-prems},
    (fn context => get-rule-prems (Context.proof-of context))) #>
   Attrib.setup @{binding #} rule-prems-by-method
    transform rule premises with method #>
   Attrib.setup @{binding @} rule-concl-by-method
    transform rule conclusion with method #>
   Attrib.setup @{binding atomized}
    (Scan.succeed (Thm.rule-attribute []
      (fn context => fn thm =>
        Conv.fconv-rule (Object-Logic.atomize (Context.proof-of context)) thm
          |> Drule.zero-var-indexes)))
    atomize rule)
›
```

experiment begin

ML ‹

```
  val [att] = @{attributes [@‹erule thin-rl, cut-tac TrueI, fail›]}
  val k = Attrib.attribute @{context} att
  val - = case (try k (Context.Proof @{context}, Drule.dummy-thm)) of
    SOME - => error Should fail
    | - => ()
  ›
```

lemmas *baz* = [[@‹*erule thin-rl*, *rule revcut-rl*[*of* $P \longrightarrow P \wedge P$], *simp*›]] **for** P

lemmas *bazz*[*THEN impE*] = *TrueI*[@‹*erule thin-rl*, *rule revcut-rl*[*of* $P \longrightarrow P \wedge P$], *simp*›] **for** P

lemma $Q \longrightarrow Q \wedge Q$ **by** (*rule baz*)

method *silly-rule* **for** P :: *bool* **uses** *rule* =
  (*rule* [[@‹*erule thin-rl*, *cut-tac rule*, *drule asm-rl*[*of* P]›]])

lemma assumes A **shows** A **by** (*silly-rule* A *rule*: ‹A›)

lemma assumes A[*simp*]: A **shows** A
  apply (*match* **conclusion in** P **for** $P \Rightarrow$
      ‹*rule* [[@‹*erule thin-rl*, *rule revcut-rl*[*of* P], *simp*›]]›)
  done

end

end

theory *Local-Method*
imports *Main*
keywords *supply-local-method* :: *prf-script* % *proof*
begin

See documentation in `Local_Method_Tests.thy`.

ML ‹
```
  structure MethodData = Proof-Data(
    type T = Method.method Symtab.table
    val init = K Symtab.empty);
```
›

method-setup *local-method* = ‹
```
  Scan.lift Parse.liberal-name >>
  (fn name => fn - => fn facts => fn (ctxt, st) =>
    case (ctxt |> MethodData.get |> Symtab.lookup) name of
        SOME method => method facts (ctxt, st)
      | NONE => Seq.succeed (Seq.Error (K (Couldn't find method text named ^
quote name))))
```
›

ML ‹
```
local

val parse-name-text-ranges =
 Scan.repeat1 (Parse.liberal-name --| Parse.!!! @{keyword =} -- Method.parse)

fun supply-method-cmd name-text-ranges ctxt =
  let
    fun add-method ((name, (text, range)), ctxt) =
      let
        val - = Method.report (text, range)
        val method = Method.evaluate text ctxt
      in
        MethodData.map (Symtab.update (name, method)) ctxt
      end
  in
    List.foldr add-method ctxt name-text-ranges
  end

val - =
  Outer-Syntax.command @{command-keyword‹supply-local-method›}
    Add a local method alias to the current proof context
```

```
  (parse-name-text-ranges >> (Toplevel.proof o Proof.map-context o supply-method-cmd))

in end
›
```

end

theory *Eisbach-Methods*
imports
  subgoal-focus/Subgoal-Methods
  HOL−Eisbach.Eisbach-Tools
  Rule-By-Method
  Local-Method
begin

7 Debugging methods

method *print-concl* = (*match* **conclusion in** *P* **for** *P* ⇒ ‹*print-term P*›)

```
method-setup print-raw-goal = ‹Scan.succeed (fn ctxt => fn facts =>
  (fn (ctxt, st) => (Output.writeln (Thm.string-of-thm ctxt st);
    Seq.make-results (Seq.single (ctxt, st)))))›

ML ‹fun method-evaluate text ctxt facts =
  Method.NO-CONTEXT-TACTIC ctxt
    (Method.evaluate-runtime text ctxt facts)›


method-setup print-headgoal =
  ‹Scan.succeed (fn ctxt =>
    fn - => fn (ctxt', thm) =>
    ((SUBGOAL (fn (t,-) =>
     (Output.writeln
     (Pretty.string-of (Syntax.pretty-term ctxt t)); all-tac)) 1 thm);
     (Seq.make-results (Seq.single (ctxt', thm)))))›
```

8 Simple Combinators

```
method-setup defer-tac = ‹Scan.succeed (fn - => SIMPLE-METHOD (defer-tac
1))›
method-setup prefer-last = ‹Scan.succeed (fn - => SIMPLE-METHOD (PRIMITIVE
(Thm.permute-prems 0 ~1)))›

method-setup all =
 ‹Method.text-closure >> (fn m => fn ctxt => fn facts =>
```

```
  let
    fun tac i st' =
      Goal.restrict i 1 st'
      |> method-evaluate m ctxt facts
      |> Seq.map (Goal.unrestrict i)

  in SIMPLE-METHOD (ALLGOALS tac) facts end)
›
```

method-setup *determ* =

```
‹Method.text-closure >> (fn m => fn ctxt => fn facts =>
  let
    fun tac st' = method-evaluate m ctxt facts st'

  in SIMPLE-METHOD (DETERM tac) facts end)
› ‹Run the given method, but only yield the first result›
```

ML ‹

```
fun require-determ (method : Method.method) facts st =
  case method facts st |> Seq.filter-results |> Seq.pull of
    NONE => Seq.empty
  | SOME (r1, rs) =>
      (case Seq.pull rs of
         NONE => Seq.single r1 |> Seq.make-results
       | - => Method.fail facts st);

fun require-determ-method text ctxt =
  require-determ (Method.evaluate-runtime text ctxt);
›
```

method-setup *require-determ* =

```
  ‹Method.text-closure >> require-determ-method›
  ‹Run the given method, but fail if it returns more than one result›
```

method-setup *changed* =

```
‹Method.text-closure >> (fn m => fn ctxt => fn facts =>
  let
    fun tac st' = method-evaluate m ctxt facts st'

  in SIMPLE-METHOD (CHANGED tac) facts end)
›
```

method-setup *timeit* =

```
‹Method.text-closure >> (fn m => fn ctxt => fn facts =>
  let
    fun timed-tac st seq = Seq.make (fn () => Option.map (apsnd (timed-tac st))
      (timeit (fn () => (Seq.pull seq))));
```

```
    fun tac st' =
      timed-tac st' (method-evaluate m ctxt facts st');

  in SIMPLE-METHOD tac [] end)
⟩
```

method-setup *timeout* =

```
⟨Scan.lift Parse.int -- Method.text-closure >> (fn (i,m) => fn ctxt => fn facts
=>
  let
    fun str-of-goal th = Pretty.string-of (Goal-Display.pretty-goal ctxt th);

    fun limit st f x = Timeout.apply (Time.fromSeconds i) f x
      handle Timeout.TIMEOUT - => error (Method timed out:\n ^ (str-of-goal
st));

      fun timed-tac st seq = Seq.make (limit st (fn () => Option.map (apsnd
(timed-tac st))
      (Seq.pull seq)));

    fun tac st' =
      timed-tac st' (method-evaluate m ctxt facts st');

  in SIMPLE-METHOD tac [] end)
⟩
```

method *repeat-new* **methods** *m* = (*m* ; (*repeat-new* ⟨*m*⟩)*?*)

The following *fails* and *succeeds* methods protect the goal from the effect of a method, instead simply determining whether or not it can be applied to the current goal. The *fails* method inverts success, only succeeding if the given method would fail.

method-setup *fails* =

```
⟨Method.text-closure >> (fn m => fn ctxt => fn facts =>
  let
    fun fail-tac st' =
      (case Seq.pull (method-evaluate m ctxt facts st') of
         SOME - => Seq.empty
       | NONE => Seq.single st')

  in SIMPLE-METHOD fail-tac facts end)
⟩
```

method-setup *succeeds* =

```
⟨Method.text-closure >> (fn m => fn ctxt => fn facts =>
  let
    fun can-tac st' =
      (case Seq.pull (method-evaluate m ctxt facts st') of
```

```
    SOME (st'',-) => Seq.single st'
  | NONE => Seq.empty)

 in SIMPLE-METHOD can-tac facts end)
⟩
```

This method wraps up the "focus" mechanic of match without actually doing any matching. We need to consider whether or not there are any assumptions in the goal, as premise matching fails if there are none.

If the *fails* method is removed here, then backtracking will produce a set of invalid results, where only the conclusion is focused despite the presence of subgoal premises.

method *focus-concl* **methods** *m* =
 ((*fails* ⟨*erule thin-rl*⟩, *match* **conclusion in** - ⇒ ⟨*m*⟩)
 | *match* **premises** (*local*) **in** *H*:- (*multi*) ⇒ ⟨*m*⟩)

repeat applies a method a specific number of times, like a bounded version of the '+' combinator.

usage: apply (repeat n *text*)

- Applies the method *text* to the current proof state n times. - Fails if *text* can't be applied n times.

ML ⟨
```
 fun repeat-tac count tactic =
   if count = 0
   then all-tac
   else tactic THEN (repeat-tac (count − 1) tactic)
⟩
```

method-setup *repeat* = ⟨
```
 Scan.lift Parse.nat −− Method.text-closure >> (fn (count, text) => fn ctxt =>
fn facts =>
   let val tactic = method-evaluate text ctxt facts
   in SIMPLE-METHOD (repeat-tac count tactic) facts end)
⟩
```

notepad begin
 fix *A B C*
 assume *assms*: *A B C*

repeat: simple repeated application.

 have *A* ∧ *B* ∧ *C* ∧ *True*

repeat: fails if method can't be applied the specified number of times.

  apply (*fails* ⟨*repeat 4* ⟨*rule conjI*, *rule assms*⟩⟩)
  apply (*repeat 3* ⟨*rule conjI*, *rule assms*⟩)
  by (*rule TrueI*)

repeat: application with subgoals.

have *A* ∧ *A* *B* ∧ *B* *C* ∧ *C*
apply −

We have three subgoals. This *repeat* call consumes two of them.

apply (*repeat 2* ‹*rule conjI*, (*rule assms*)+›)

One subgoal remaining...

apply (*rule conjI*, (*rule assms*)+)
done

end

Literally a copy of the parser for *subgoal-tac* composed with an analogue of **prefer**.

Useful if you find yourself introducing many new facts via ‘subgoal$_{t}ac$‘, $butprefertoprovethemimmedia$

setup ‹
Method.setup ***binding*** ‹*prop-tac*›
(*Args.goal-spec* −− *Scan.lift* (*Scan.repeat1* *Args.embedded-inner-syntax* −− *Parse.for-fixes*) >>
(*fn* (*quant*, (*props*, *fixes*)) => *fn ctxt* =>
(*SIMPLE-METHOD''* *quant*
(*EVERY'* (*map* (*fn prop* => *Rule-Insts.subgoal-tac ctxt prop fixes*) *props*)
THEN'
(*K* (*prefer-tac 2*)))))))
insert prop (*dynamic instantiation*), *introducing prop subgoal first*
›

notepad begin {
fix *xs*
assume *assms*: *list-all even* (*xs* :: *nat list*)

from *assms* **have** *even* (*sum-list xs*)
apply (*induct xs*)
apply *simp*

Inserts the desired proposition as the current subgoal.

apply (*prop-tac list-all even xs*)
subgoal by *simp*

The prop *list-all even xs* is now available as an assumption. Let's add another one.

apply (*prop-tac even* (*sum-list xs*))
subgoal by *simp*

Now that we've proven our introduced props, use them!

apply *clarsimp*

```
  done
}
end
```

9 Advanced combinators

9.1 Protecting goal elements (assumptions or conclusion) from methods

```
context
begin

private definition protect-concl x ≡ ¬ x
private definition protect-false ≡ False

private lemma protect-start: (protect-concl P ⟹ protect-false) ⟹ P
  by (simp add: protect-concl-def protect-false-def) (rule ccontr)

private lemma protect-end: protect-concl P ⟹ P ⟹ protect-false
  by (simp add: protect-concl-def protect-false-def)

method only-asm methods m =
  (match premises in H[thin]:- (multi,cut) ⇒
    ‹rule protect-start,
     match premises in H'[thin]:protect-concl - ⇒
       ‹insert H,m;rule protect-end[OF H']››)

method only-concl methods m = (focus-concl ‹m›)

end

notepad begin
 fix D C
  assume DC:D ⟹ C
  have D ∧ D ⟹ C ∧ C
  apply (only-asm ‹simp›) — stash conclusion before applying method
  apply (only-concl ‹simp add: DC›) — hide premises from method
  by (rule DC)

end
```

9.2 Safe subgoal folding (avoids expanding meta-conjuncts)

Isabelle's goal mechanism wants to aggressively expand meta-conjunctions if they are the top-level connective. This means that *fold-subgoals* will immediately be unfolded if there are no common assumptions to lift over.

To avoid this we simply wrap conjunction inside of conjunction' to hide it from the usual facilities.

context begin

definition
conjunction' :: *prop* ⇒ *prop* ⇒ *prop* (**infixr** &^& *2*) **where**
conjunction' A B ≡ (*PROP A* &&& *PROP B*)

In general the context antiquotation does not work in method definitions.
Here it is fine because Conv.top$_sweep_convisjustover-specifiedtoneedaProof.contextwhenanything$

method *safe-meta-conjuncts* =
raw-tactic
‹*REPEAT-DETERM*
(*CHANGED-PROP*
(*PRIMITIVE*
(*Conv.gconv-rule* ((*Conv.top-sweep-conv* (*K* (*Conv.rewr-conv* @{*thm conjunction'-def*[*symmetric*]})) @{*context*})) *1*)))›

method *safe-fold-subgoals* = (*fold-subgoals* (*prefix*), *safe-meta-conjuncts*)

lemma *atomize-conj'* [*atomize*]: (*A* &^& *B*) == *Trueprop* (*A* & *B*)
by (*simp add*: *conjunction'-def*, *rule atomize-conj*)

lemma *context-conjunction'I*:
PROP P ⟹ (*PROP P* ⟹ *PROP Q*) ⟹ *PROP P* &^& *PROP Q*
apply (*simp add*: *conjunction'-def*)
apply (*rule conjunctionI*)
apply *assumption*
apply (*erule meta-mp*)
apply *assumption*
done

lemma *conjunction'I*:
PROP P ⟹ *PROP Q* ⟹ *PROP P* &^& *PROP Q*
by (*rule context-conjunction'I*; *simp*)

lemma *conjunction'E*:
assumes *PQ*: *PROP P* &^& *PROP Q*
assumes *PQR*: *PROP P* ⟹ *PROP Q* ⟹ *PROP R*
shows
PROP R
apply (*rule PQR*)
apply (*rule PQ*[*simplified conjunction'-def*, *THEN conjunctionD1*])
by (*rule PQ*[*simplified conjunction'-def*, *THEN conjunctionD2*])

end

notepad begin
fix *D C E*

assume *DC*: *D* ∧ *C*

have $D\ C \wedge C$
apply $-$
apply (*safe-fold-subgoals*, *simp*, *atomize* (*full*))
apply (*rule DC*)
done

end

10 Utility methods

10.1 Finding a goal based on successful application of a method

context begin

method-setup *find-goal* =
‹*Method.text-closure* >> (*fn m* => *fn ctxt* => *fn facts* =>
let
fun prefer-first i = *SELECT-GOAL*
(*fn st′* =>
(*case Seq.pull* (*method-evaluate m ctxt facts st′*) *of*
SOME (*st′′*,-) => *Seq.single st′′*
| *NONE* => *Seq.empty*)) *i THEN prefer-tac i*

in SIMPLE-METHOD (*FIRSTGOAL prefer-first*) *facts end*)›

end

notepad begin

fix $A\ B$
assume A: A **and** B: B

have $A\ A\ B$
apply (*find-goal* ‹*match conclusion in B* ⇒ ‹−››)
apply (*rule B*)
by (*rule A*)+

have $A \wedge A\ A \wedge A\ B$
apply (*find-goal* ‹*fails* ‹*simp*››) — find the first goal which cannot be simplified
apply (*rule B*)
by (*simp add*: *A*)+

have $B\ A\ A \wedge A$
apply (*find-goal* ‹*succeeds* ‹*simp*››) — find the first goal which can be simplified (without doing so)
apply (*rule conjI*)
by (*rule A B*)+

end

10.2 Remove redundant subgoals

Tries to solve subgoals by assuming the others and then using the given method. Backtracks over all possible re-orderings of the subgoals.

context begin

definition *protect* (*PROP P*) ≡ *P*

lemma *protectE*: *PROP protect P* ⟹ (*PROP P* ⟹ *PROP R*) ⟹ *PROP R* **by** (*simp add*: *protect-def*)

private lemmas *protect-thin* = *thin-rl*[**where** *V*=*PROP protect P* **for** *P*]

```
private lemma context-conjunction'I-protected:
  assumes P: PROP P
  assumes PQ: PROP protect (PROP P) ⟹ PROP Q
  shows
  PROP P &^& PROP Q
   apply (simp add: conjunction'-def)
   apply (rule P)
  apply (rule PQ)
  apply (simp add: protect-def)
  by (rule P)
```

```
private lemma conjunction'-sym: PROP P &^& PROP Q ⟹ PROP Q &^&
PROP P
  apply (simp add: conjunction'-def)
  apply (frule conjunctionD1)
  apply (drule conjunctionD2)
  apply (rule conjunctionI)
  by assumption+
```

```
private lemmas context-conjuncts'I =
  context-conjunction'I-protected
  context-conjunction'I-protected[THEN conjunction'-sym]
```

```
method distinct-subgoals-strong methods m =
  (safe-fold-subgoals,
   (intro context-conjuncts'I;
     (((elim protectE conjunction'E)?, solves ‹m›)
     | (elim protect-thin)?)))?
```

end

```
method forward-solve methods fwd m =
  (fwd, prefer-last, fold-subgoals, safe-meta-conjuncts, rule conjunction'I,
   defer-tac, ((intro conjunction'I)?; solves ‹m›))[1]
```

method *frule-solve* **methods** m **uses** *rule* = (*forward-solve* ‹*frule rule*› ‹m›)
method *drule-solve* **methods** m **uses** *rule* = (*forward-solve* ‹*drule rule*› ‹m›)

notepad begin
  {
  fix A B C D E
  assume $ABCD$: $A \Longrightarrow B \Longrightarrow C \Longrightarrow D$
  assume ACD: $A \Longrightarrow C \Longrightarrow D$
  assume DE: $D \Longrightarrow E$
  assume B C

  have $A \Longrightarrow D$
  apply (*frule-solve* ‹*simp add*: ‹B› ‹C›› *rule*: $ABCD$)
  apply (*drule-solve* ‹*simp add*: ‹B› ‹C›› *rule*: ACD)
  apply (*match* **premises in** $A \Rightarrow$ ‹*fail*› | - $\Rightarrow$ ‹−›)
  apply *assumption*
  done
  }
end

notepad begin
  {
  fix A B C
  assume A: A
  have A $B \Longrightarrow A$
  apply −
  apply (*distinct-subgoals-strong* ‹*assumption*›)
  by (*rule* A)

  have $B \Longrightarrow A$ A
  by (*distinct-subgoals-strong* ‹*assumption*›, *rule* A) — backtracking required here
  }

  {
  fix A B C

  assume B: B
  assume BC: $B \Longrightarrow C$ $B \Longrightarrow A$
  have A $B \longrightarrow (A \land C)$ B
  apply (*distinct-subgoals-strong* ‹*simp*›, *rule* B) — backtracking required here
  by (*simp add*: BC)

  }
end

11 Attribute methods (for use with rule$_{b}y_{m}ethodattributes$)

method *prove-prop-raw* **for** P :: *prop* **methods** m =
  (*erule thin-rl*, *rule revcut-rl*[*of PROP P*],
    solves ⟨*match conclusion in* - ⇒ ⟨*m*⟩⟩)

method *prove-prop* **for** P :: *prop* = (*prove-prop-raw PROP P* ⟨*auto*⟩)

experiment begin

lemma assumes *A*[*simp*]:*A* **shows** *A* **by** (*rule* [[@⟨*prove-prop A*⟩]])

end

12 Shortcuts for prove$_{p}rop.Notethesearelessefficientthanusingtheraws$ $proveneverytime.$

method *ruleP* **for** P :: *prop* = (*catch* ⟨*rule* [[@⟨*prove-prop PROP P*⟩]]⟩ ⟨*fail*⟩)
method *insertP* **for** P :: *prop* = (*catch* ⟨*insert* [[@⟨*prove-prop PROP P*⟩]]⟩ ⟨*fail*⟩)[*1*]

experiment begin

lemma assumes *A*[*simp*]:*A* **shows** *A* **by** (*ruleP False* | *ruleP A*)
lemma assumes *A*:*A* **shows** *A* **by** (*ruleP* $\bigwedge P.\ P \Longrightarrow P \Longrightarrow P$, *rule A*, *rule A*)

end

context begin

private definition *bool-protect* (*b*::*bool*) ≡ *b*

lemma *bool-protectD*:
  bool-protect P ⟹ *P*
  unfolding *bool-protect-def* **by** *simp*

lemma *bool-protectI*:
  P ⟹ *bool-protect P*
  unfolding *bool-protect-def* **by** *simp*

When you want to apply a rule/tactic to transform a potentially complex goal into another one manually, but want to indicate that any fresh emerging goals are solved by a more brutal method. E.g. apply (solves$_{e}mergingfrule$ *x*=... *in my-rulefastforce s*

method *solves-emerging* **methods** *m1 m2* = (*rule bool-protectD*, (*m1* ; (*rule bool-protectI* | (*m2*; *fail*))))

end

end

theory *Try-Methods*

imports *Eisbach-Methods*

keywords *trym* :: *diag*
 and *add-try-method* :: *thy-decl*

begin

A collection of methods that can be "tried" against subgoals (similar to try, try0 etc). It is easy to add new methods with "add$_{t}ry_{m}ethod$", *althoughtheparsercurrentlysupportso*
Particular subgoals can be tried with "trym 1" etc. By default all subgoals are attempted unless they are coupled to others by shared schematic variables.

ML ‹
```
structure Try-Methods = struct

structure Methods = Theory-Data
(
  type T = Symtab.set;
  val empty = Symtab.empty;
  val extend = I;
  val merge = Symtab.merge (K true);
);

val get-methods-global = Methods.get #> Symtab.keys
val add-method = Methods.map o Symtab.insert-set

(* borrowed from try0 implementation (of course) *)
fun parse-method-name keywords =
  enclose ( )
  #> Token.explode keywords Position.start
  #> filter Token.is-proper
  #> Scan.read Token.stopper Method.parse
  #> (fn SOME (Method.Source src, -) => src | - => raise Fail expected Source);

fun mk-method ctxt = parse-method-name (Thy-Header.get-keywords' ctxt)
  #> Method.method-cmd ctxt
  #> Method.Basic

fun get-methods ctxt = get-methods-global (Proof-Context.theory-of ctxt)
  |> map (mk-method ctxt)

fun try-one-method m ctxt n goal
    = can (Timeout.apply (Time.fromSeconds 5)
        (Goal.restrict n 1 #> Method.NO-CONTEXT-TACTIC ctxt
            (Method.evaluate-runtime m ctxt []))
```

```
        #> Seq.hd
    )) goal

fun msg m-nm n = writeln (method  ^ m-nm ^  succceeded on goal  ^ string-of-int
n)

fun times xs ys = maps (fn x => map (pair x) ys) xs

fun independent-subgoals goal verbose = let
    fun get-vars t = Term.fold-aterms
        (fn (Var v) => Termtab.insert-set (Var v) | - => I)
        t Termtab.empty
    val goals = Thm.prems-of goal
    val goal-vars = map get-vars goals
    val count-vars = fold (fn t1 => fn t2 => Termtab.join (K (+))
        (Termtab.map (K (K 1)) t1, t2)) goal-vars Termtab.empty
    val indep-vars = Termtab.forall (fst #> Termtab.lookup count-vars
        #> (fn n => n = SOME 1))
    val indep = (1 upto Thm.nprems-of goal) ~~ map indep-vars goal-vars
    val - = app (fst #> string-of-int
        #> prefix ignoring non−independent goal  #> warning)
        (filter (fn x => verbose andalso not (snd x)) indep)
  in indep |> filter snd |> map fst end

fun try-methods opt-n ctxt goal = let
    val ms = get-methods-global (Proof-Context.theory-of ctxt)
        ~~ get-methods ctxt
    val ns = case opt-n of
        NONE => independent-subgoals goal true
      | SOME n => [n]
    fun apply ((m-nm, m), n) = if try-one-method m ctxt n goal
      then (msg m-nm n; SOME (m-nm, n)) else NONE
    val results = Par-List.map apply (times ms ns)
  in map-filter I results end

fun try-methods-command opt-n st = let
    val ctxt = #context (Proof.goal st)
        |> Try0.silence-methods false
    val goal = #goal (Proof.goal st)
  in try-methods opt-n ctxt goal; () end

val - = Outer-Syntax.command @{command-keyword trym}
  try methods from a library of specialised strategies
  (Scan.option Parse.int >> (fn opt-n =>
    Toplevel.keep-proof (try-methods-command opt-n o Toplevel.proof-of)))

fun local-check-add-method nm ctxt =
    (mk-method ctxt nm; Local-Theory.background-theory (add-method nm) ctxt)
```

val - = Outer-Syntax.command @{command-keyword add-try-method}
add a method to a library of strategies tried by trym
(Parse.name >> (Toplevel.local-theory NONE NONE o local-check-add-method))

end
›

add-try-method *fastforce*
add-try-method *blast*
add-try-method *metis*

method *auto-metis* = *solves* ‹*auto*; *metis*›
add-try-method *auto-metis*

end

theory *Extract-Conjunct*
imports
Main
Eisbach-Methods
begin

13 Extracting conjuncts in the conclusion

Methods for extracting a conjunct from a nest of conjuncts in the conclusion of a goal, typically by pattern matching.

When faced with a conclusion which is a big conjunction, it is often the case that a small number of conjuncts require special attention, while the rest can be solved easily by *clarsimp*, *auto* or similar. However, sometimes the method that would solve the bulk of the conjuncts would put some of the conjuncts into a more difficult or unsolvable state.

The higher-order methods defined here provide an efficient way to select a conjunct requiring special treatment, so that it can be dealt with first. Once all such conjuncts have been removed, the remaining conjuncts can all be solved together by some automated method.

Each method takes an inner method as an argument, and selects the left-most conjunct for which that inner method succeeds. The methods differ according to what they do with the selected conjunct. See below for more information and some simple examples.

context begin

13.1 Focused conjunct with context

We define a predicate which allows us to identify a particular sub-tree and its context within a nest of conjunctions. We express this sub-tree-with-context using a function which reconstructs the original nest of conjunctions. The context consists of a list of parent contexts, where each parent context consists of a sibling sub-tree, and a tag indicating whether the focused sub-tree is on the left or right. Rebuilding the original tree works from the focused sub-tree up towards the root of the original structure. This sub-tree-with-context is sometimes known as a zipper.

private fun *focus-conj* :: *bool* ⇒ *bool list* ⇒ *bool* **where**
focus-conj current [] = *current*
| *focus-conj current* (*sibling* # *parents*) = *focus-conj* (*current* ∧ *sibling*) *parents*

private definition *focus* ≡ *focus-conj*

private definition *tag t P* ≡ *P*
private lemmas *focus-defs* = *focus-def tag-def*

private abbreviation *left* ≡ *tag Left*
private abbreviation *right* ≡ *tag Right*

private lemma *focus-example*:
focus C [*right B*, *left D*, *left E*, *right A*] ⟷ *A* ∧ ((*B* ∧ *C*) ∧ *D*) ∧ *E*
unfolding *focus-defs* **by** *auto*

13.2 Moving the focus

We now prove some rules which allow us to switch between focused and unfocused structures, and to move the focus around. Some versions of these rules carry an extra conjunct E outside the structure. Once we find the conjunct we want, this E allows to keep track of it while we reassemble the rest of the original structure.

First, we have rules for going between focused and unfocused structures.

private lemma *focus-top-iff*: *E* ∧ *focus P* [] ⟷ *E* ∧ *P*
unfolding *focus-def* **by** *simp*

private lemmas *to-focus* = *focus-top-iff* [**where** *E*=*True*, *simplified*, *THEN iffD1*]
private lemmas *from-focusE* = *focus-top-iff* [*THEN iffD2*]
private lemmas *from-focus* = *from-focusE*[**where** *E*=*True*, *simplified*]

Next, we have rules for moving the focus to and from the left conjunct.

private lemma *focus-left-iff*: *E* ∧ *focus L* (*left R* # *P*) ⟷ *E* ∧ *focus* (*L* ∧ *R*) *P*
unfolding *focus-defs* **by** *simp*

private lemmas *focus-left* = *focus-left-iff* [**where** *E*=*True*, *simplified*, *THEN iffD1*]
private lemmas *unfocusE-left* = *focus-left-iff* [*THEN iffD2*]
private lemmas *unfocus-left* = *unfocusE-left*[**where** *E*=*True*, *simplified*]

Next, we have rules for moving the focus to and from the right conjunct.

private lemma *focus-right-iff*: $E \wedge focus\ R\ (right\ L\ \#\ P) \longleftrightarrow E \wedge focus\ (L \wedge R)\ P$
  unfolding *focus-defs* **using** *conj-commute* **by** *simp*

private lemmas *focus-right* = *focus-right-iff* [**where** *E*=*True*, *simplified*, *THEN iffD1*]
private lemmas *unfocusE-right* = *focus-right-iff* [*THEN iffD2*]
private lemmas *unfocus-right* = *unfocusE-right*[**where** *E*=*True*, *simplified*]

Finally, we have rules for extracting the current focus. The sibling of the extracted focus becomes the new focus of the remaining structure.

private lemma *extract-focus-iff*: $focus\ C\ (tag\ t\ S\ \#\ P) \longleftrightarrow (C \wedge focus\ S\ P)$
  unfolding *focus-defs* **by** (*induct P arbitrary*: *S*) *auto*

private lemmas *extract-focus* = *extract-focus-iff* [*THEN iffD2*]

13.3 Primitive methods for navigating a conjunction

Using these rules as transitions, we implement a machine which navigates a tree of conjunctions, searching from left to right for a conjunct for which a given method will succeed. Once a matching conjunct is found, it is extracted, and the remaining conjuncts are reassembled.

From the current focus, move to the leftmost sub-conjunct.

private method *focus-leftmost* = (*intro focus-left*)?

Find the furthest ancestor for which the current focus is still on the right.

private method *unfocus-rightmost* = (*intro unfocus-right*)?

Move to the immediate-right sibling.

private method *focus-right-sibling* = (*rule unfocus-left*, *rule focus-right*)

Move to the next conjunct in right-to-left ordering.

private method *focus-next-conjunct* = (*unfocus-rightmost*, *focus-right-sibling*, *focus-leftmost*)

Search from current focus toward the right until we find a matching conjunct.

private method *find-match* **methods** *m* = (*rule extract-focus*, *m* | *focus-next-conjunct*, *find-match m*)

Search within nest of conjuncts, leaving remaining structure focused.

private method *extract-match* **methods** *m* = (*rule to-focus*, *focus-leftmost*, *find-match m*)

Move all the way out of focus, keeping track of any extracted conjunct.

private method *unfocusE* = ((*intro unfocusE-right unfocusE-left*)*?*, *rule from-focusE*)
private method *unfocus* = ((*intro unfocus-right unfocus-left*)*?*, *rule from-focus*)

13.4 Methods for selecting the leftmost matching conjunct

See the introduction at the top of this theory for motivation, and below for some simple examples.

Assuming the conclusion of the goal is a nest of conjunctions, method *lift-conjunct* finds the leftmost conjunct for which the given method succeeds, and moves it to the front of the conjunction in the goal.

method *lift-conjunct* **methods** m = (*extract-match* ‹*succeeds* ‹*rule conjI*, m››, *unfocusE*)

Method *extract-conjunct* finds the leftmost conjunct for which the given method succeeds, and splits it into a fresh subgoal, leaving the remaining conjuncts untouched in the second subgoal. It is equivalent to *lift-conjunct* followed by *rule* $[\![?P;\ ?Q]\!] \Longrightarrow ?P \wedge ?Q$.

method *extract-conjunct* **methods** m = (*extract-match* ‹*rule conjI*, *succeeds* m›; *unfocus?*)

Method *apply-conjunct* finds the leftmost conjunct for which the given method succeeds, leaving any subgoals created by the application of that method, and a subgoal containing the remaining conjuncts untouched. It is equivalent to *extract-conjunct* followed by the given method, but more efficient.

method *apply-conjunct* **methods** m = (*extract-match* ‹*rule conjI*, m›; *unfocus?*)

13.5 Examples

Given an inner method based on *match*, which only succeeds on the desired conjunct C, *lift-conjunct* moves the conjunct C to the front. The body of the *match* here is irrelevant, since *lift-conjunct* always discards the effect of the method it is given.

lemma $[\![\ A;\ B;\ [\![\ A;\ B;\ D;\ E\ ]\!] \Longrightarrow C;\ D;\ E\ ]\!] \Longrightarrow A \wedge ((B \wedge C) \wedge D) \wedge E$
  apply (*lift-conjunct* ‹*match conclusion in* $C \Rightarrow$ ‹−››)
  — C as been moved to the front of the conclusion.
  apply (*match* **conclusion in** ‹$C \wedge A \wedge (B \wedge D) \wedge E$› $\Rightarrow$ ‹−›)
  oops

Method *extract-conjunct* works similarly, but peels of the matched conjunct as a separate subgoal. As for *lift-conjunct*, the effect of the given method is discarded, so the body of the *match* is irrelevant.

lemma $[\![\ A;\ B;\ [\![\ A;\ B;\ D;\ E\ ]\!] \Longrightarrow C;\ D;\ E\ ]\!] \Longrightarrow A \wedge ((B \wedge C) \wedge D) \wedge E$
  apply (*extract-conjunct* ‹*match conclusion in* $C \Rightarrow$ ‹−››)

— *extract-conjunct* gives us the matched conjunct C as a separate subgoal.
apply (*match* **conclusion in** $C \Rightarrow \langle - \rangle$)
apply *blast*
— The other subgoal contains the remaining conjuncts untouched.
apply (*match* **conclusion in** $\langle A \wedge (B \wedge D) \wedge E \rangle \Rightarrow \langle - \rangle$)
oops

Method *apply-conjunct* goes one step further, and applies the given method to the extracted subgoal.

lemma $[\![A; B; [\![A; B; D; E]\!] \Longrightarrow C; D; E]\!] \Longrightarrow A \wedge ((B \wedge C) \wedge D) \wedge E$
apply (*apply-conjunct* ⟨*match conclusion in* $C \Rightarrow$ ⟨*match premises in* H: - $\Rightarrow$ ⟨*rule* H⟩⟩⟩)
— We get four subgoals from applying the given method to the matched conjunct C.
apply (*match* **premises in** H: $A \Rightarrow$ ⟨*rule* H⟩)
apply (*match* **premises in** H: $B \Rightarrow$ ⟨*rule* H⟩)
apply (*match* **premises in** H: $D \Rightarrow$ ⟨*rule* H⟩)
apply (*match* **premises in** H: $E \Rightarrow$ ⟨*rule* H⟩)
— The last subgoal contains the remaining conjuncts untouched.
apply (*match* **conclusion in** $\langle A \wedge (B \wedge D) \wedge E \rangle \Rightarrow \langle - \rangle$)
oops

end

end

theory *Eval-Bool*

imports *Try-Methods*

begin

The eval$_{b}oolmethod/simprocusesthecodegeneratorsetuptoreducetermsofbooleantypetoTrueorFals$
$equations.$

Additional simprocs exist to reduce other types.

ML ⟨
structure Eval-Simproc = struct

exception Failure

fun mk-constname-tab ts = fold Term.add-const-names ts []
|> Symtab.make-set

fun is-built-from tab t = case Term.strip-comb t of
(Const (cn, -), ts) => Symtab.defined tab cn
andalso forall (is-built-from tab) ts
| - => false

```
fun eval tab ctxt ct = let
    val t = Thm.term-of ct
    val - = Term.fold-aterms (fn Free - => raise Failure
      | Var - => raise Failure | - => ignore) t ()
    val - = not (is-built-from tab t) orelse raise Failure
    val ev = the (try (Code-Simp.dynamic-conv ctxt) ct)
  in if is-built-from tab (Thm.term-of (Thm.rhs-of ev))
    then SOME ev else NONE end
  handle Failure => NONE | Option => NONE

val eval-bool = eval (mk-constname-tab [@{term True}, @{term False}])
val eval-nat = eval (mk-constname-tab [@{term Suc 0}, @{term Suc 1},
    @{term Suc 9}])
val eval-int = eval (mk-constname-tab [@{term 0 :: int}, @{term 1 :: int},
    @{term 18 :: int}, @{term (−9) :: int}])

val eval-bool-simproc = Simplifier.make-simproc @{context} eval-bool
  { lhss = [@{term b :: bool}], proc = K eval-bool }
val eval-nat-simproc = Simplifier.make-simproc @{context} eval-nat
  { lhss = [@{term n :: nat}], proc = K eval-nat }
val eval-int-simproc = Simplifier.make-simproc @{context} eval-int
  { lhss = [@{term i :: int}], proc = K eval-int }

end
›
```

method-setup *eval-bool* = ‹*Scan.succeed (fn ctxt => SIMPLE-METHOD′*
  (CHANGED o full-simp-tac (clear-simpset ctxt
    addsimprocs [Eval-Simproc.eval-bool-simproc])))›
  use code generator setup to simplify booleans in goals to True or False

method-setup *eval-int-nat* = ‹*Scan.succeed (fn ctxt => SIMPLE-METHOD′*
  (CHANGED o full-simp-tac (clear-simpset ctxt
   addsimprocs [Eval-Simproc.eval-nat-simproc, Eval-Simproc.eval-int-simproc])))›
  use code generator setup to simplify nats and ints in goals to values

add-try-method *eval-bool*

Testing.

definition
 eval-bool-test-seq :: *int list*
where
 eval-bool-test-seq = [*2*, *3*, *4*, *5*, *6*, *7*, *8*]

lemma
 eval-bool-test-seq ! *4* = *6* ∧ (*3* :: *nat*) < *4*
  ∧ *sorted eval-bool-test-seq*
 by *eval-bool*

A related gadget for installing constant definitions from locales as code equations. Useful where locales are being used to "hide" constants from the global state rather than to do anything tricky with interpretations.

Installing the global definitions in this way will allow eval$_bool etc to "see through" the hiding and decide equ

ML ‹
structure Add-Locale-Code-Defs = struct

```
fun get-const-defs thy nm = Sign.consts-of thy
  |> Consts.dest |> #constants
  |> map fst
  |> filter (fn s => case Long-Name.explode s of
       [-, nm', -] => nm' = nm | - => false)
  |> map-filter (try (suffix -def #> Global-Theory.get-thm thy))
  |> filter (Thm.strip-shyps #> Thm.shyps-of #> null)
  |> tap (fn xs => tracing (Installing ^ string-of-int (length xs) ^ code defs))

fun setup nm thy = fold (fn t => Code.add-eqn-global (t, true))
    (get-const-defs thy nm) thy

end
```
›

locale *eval-bool-test-locale* **begin**

definition
  $x == (12 :: int)$

definition
  $y == (13 :: int)$

definition
  $z = (x * y) + x + y$

end

setup ‹*Add-Locale-Code-Defs.setup eval-bool-test-locale*›

setup ‹*Add-Locale-Code-Defs.setup eval-bool-test-locale*›

lemma *eval-bool-test-locale.z > 150*
  by *eval-bool*

end

— MLUtils is a collection of 'basic' ML utilities (kind of like `~~/src/Pure/library.ML`, but maintained by Trustworthy Systems). If you find yourself implementing: - A simple data-structure-shuffling task, - Something that shows up in the standard library of other functional languages, or - Something that's "missing" from the general pattern of an Isabelle ML library, consider adding it here.

theory *MLUtils*
imports *Main*
begin
ML-file *StringExtras.ML*
ML-file *ListExtras.ML*
ML-file *MethodExtras.ML*
ML-file *OptionExtras.ML*
ML-file *ThmExtras.ML*
ML-file *Sum.ML*
end

theory *Apply-Trace*
imports
  Main
  ml−helpers/MLUtils
begin

ML ‹
```
signature APPLY-TRACE =
sig
  val apply-results :
    {silent-fail : bool} ->
    (Proof.context -> thm -> ((string * int option) * term) list -> unit) ->
    Method.text-range -> Proof.state -> Proof.state Seq.result Seq.seq

  (* Lower level interface. *)
  val can-clear : theory -> bool
  val clear-deps : thm -> thm
  val join-deps : thm -> thm -> thm
  val used-facts : Proof.context -> thm -> ((string * int option) * term) list
  val pretty-deps: bool -> (string * Position.T) option -> Proof.context -> thm
->
    ((string * int option) * term) list -> Pretty.T
end

structure Apply-Trace : APPLY-TRACE =
struct

(*TODO: Add more robust oracle without hyp clearing *)
fun thm-to-cterm keep-hyps thm =
let

  val thy = Thm.theory-of-thm thm
```

```
  val pairs = Thm.tpairs-of thm
  val ceqs = map (Thm.global-cterm-of thy o Logic.mk-equals) pairs
  val hyps = Thm.chyps-of thm
  val prop = Thm.cprop-of thm
  val thm' = if keep-hyps then Drule.list-implies (hyps,prop) else prop

in
  Drule.list-implies (ceqs,thm') end


val (-, clear-thm-deps') =
 Context.>>> (Context.map-theory-result (Thm.add-oracle (Binding.name count-cheat,
thm-to-cterm false)));

fun clear-deps thm =
let

  val thm' = try clear-thm-deps' thm
 |> Option.map (fold (fn - => fn t => (@{thm Pure.reflexive} RS t)) (Thm.tpairs-of
thm))

in case thm' of SOME thm' => thm' | NONE => error Can't clear deps here end


fun can-clear thy = Context.subthy(@{theory},thy)

fun join-deps pre-thm post-thm =
let
  val pre-thm' = Thm.flexflex-rule NONE pre-thm |> Seq.hd
    |> Thm.adjust-maxidx-thm (Thm.maxidx-of post-thm + 1)
in
  Conjunction.intr pre-thm' post-thm |> Conjunction.elim |> snd
end

fun get-ref-from-nm' nm =
let
  val exploded = space-explode - nm;
  val base = List.take (exploded, (length exploded) - 1) |> space-implode -
  val idx = List.last exploded |> Int.fromString;
in if is-some idx andalso base <>  then SOME (base, the idx) else NONE end

fun get-ref-from-nm nm = Option.join (try get-ref-from-nm' nm);

fun maybe-nth l = try (curry List.nth l)

fun fact-from-derivation ctxt xnm =
let

  val facts = Proof-Context.facts-of ctxt;
```

```
  (* TODO: Check that exported local fact is equivalent to external one *)

  val idx-result =
    let
      val (name', idx) = get-ref-from-nm xnm |> the;
        val entry = try (Facts.retrieve (Context.Proof ctxt) facts) (name', Posi-
tion.none) |> the;
      val thm = maybe-nth (#thms entry) (idx - 1) |> the;
    in SOME (xnm, thm) end handle Option => NONE;

  fun non-idx-result () =
    let
        val entry = try (Facts.retrieve (Context.Proof ctxt) facts) (xnm, Posi-
tion.none) |> the;
      val thm = try the-single (#thms entry) |> the;
    in SOME (#name entry, thm) end handle Option => NONE;

in
  case idx-result of
    SOME thm => SOME thm
  | NONE => non-idx-result ()
end

fun most-local-fact-of ctxt xnm =
let
  val local-name = try (fn xnm => Long-Name.explode xnm |> tl |> tl |> Long-Name.implode)
xnm |> the;
in SOME (fact-from-derivation ctxt local-name |> the) end handle Option =>
  fact-from-derivation ctxt xnm;

fun thms-of (PBody {thms,...}) = thms

fun proof-body-descend' f get-fact (ident, thm-node) deptab = let
  val nm = Proofterm.thm-node-name thm-node
  val body = Proofterm.thm-node-body thm-node
in
  (if not (f nm) then
     (Inttab.update-new (ident, SOME (nm, get-fact nm |> the)) deptab handle
Inttab.DUP - => deptab)
  else raise Option) handle Option =>
    ((fold (proof-body-descend' f get-fact) (thms-of (Future.join body))
      (Inttab.update-new (ident, NONE) deptab)) handle Inttab.DUP - => deptab)
end

fun used-facts' f get-fact thm =
  let
    val body = thms-of (Thm.proof-body-of thm);

  in fold (proof-body-descend' f get-fact) body Inttab.empty end
```

```
fun used-pbody-facts ctxt thm =
  let
    val nm = Thm.get-name-hint thm;
    val get-fact = most-local-fact-of ctxt;
  in
    used-facts' (fn nm' => nm' =  orelse nm' = nm) get-fact thm
    |> Inttab.dest |> map-filter snd |> map snd |> map (apsnd (Thm.prop-of))
  end

fun raw-primitive-text f = Method.Basic (fn - => ((K (fn (ctxt, thm) => Se-
q.make-results (Seq.single (ctxt, f thm))))))


(*Find local facts from new hyps*)
fun used-local-facts ctxt thm =
let
  val hyps = Thm.hyps-of thm
  val facts = Proof-Context.facts-of ctxt |> Facts.dest-static true []

  fun match-hyp hyp =
  let
    fun get (nm,thms) =
      case (get-index (fn t => if (Thm.prop-of t) aconv hyp then SOME hyp else
NONE) thms)
      of SOME t => SOME (nm,t)
        | NONE => NONE


  in
    get-first get facts
  end

in
  map-filter match-hyp hyps end

fun used-facts ctxt thm =
  let
   val used-from-pbody = used-pbody-facts ctxt thm |> map (fn (nm,t) => ((nm,NONE),t))
      val used-from-hyps = used-local-facts ctxt thm |> map (fn (nm,(i,t)) =>
((nm,SOME i),t))
  in
    (used-from-hyps @ used-from-pbody)
  end

(* Perform refinement step, and run the given stateful function
   against computed dependencies afterwards. *)
fun refine args f text state =
let
```

```
  val ctxt = Proof.context-of state

  val thm = Proof.simple-goal state |> #goal

  fun save-deps deps = f ctxt thm deps


in
  if (can-clear (Proof.theory-of state)) then
      Proof.refine (Method.Combinator (Method.no-combinator-info,Method.Then,
[raw-primitive-text (clear-deps),text,
      raw-primitive-text (fn thm' => (save-deps (used-facts ctxt thm');join-deps thm
thm'))])) state
  else
      (if (#silent-fail args) then (save-deps [];Proof.refine text state) else error
Apply-Trace theory must be imported to trace applies)
end

(* Boilerplate from Proof.ML *)


fun method-error kind pos state =
  Seq.single (Proof-Display.method-error kind pos (Proof.raw-goal state));

fun apply args f text = Proof.assert-backward #> refine args f text #>
 Seq.maps-results (Proof.apply ((raw-primitive-text I),(Position.none, Position.none)));

fun apply-results args f (text, range) =
  Seq.APPEND (apply args f text, method-error  (Position.range-position range));


structure Filter-Thms = Named-Thms
(
  val name = @{binding no-trace}
  val description = thms to be ignored from tracing
)

(* Print out the found dependencies. *)
fun pretty-deps only-names query ctxt thm deps =
let
  (* Remove duplicates. *)
 val deps = sort-distinct (prod-ord (prod-ord string-ord (option-ord int-ord)) Term-Ord.term-ord)
deps

  (* Fetch canonical names and theorems. *)
  val deps = map (fn (ident, term) => ThmExtras.adjust-thm-name ctxt ident
term) deps
```

```
  (* Remove boring theorems. *)
  val deps = subtract (fn (a, ThmExtras.FoundName (-, thm)) => Thm.eq-thm
(thm, a)
                        | - => false) (Filter-Thms.get ctxt) deps

  val deps = case query of SOME (raw-query,pos) =>
    let
      val pos' = perhaps (try (Position.advance-offsets 1)) pos;
      val q = Find-Theorems.read-query pos' raw-query;
       val results = Find-Theorems.find-theorems-cmd ctxt (SOME thm) (SOME
1000000000) false q
                  |> snd
                  |> map ThmExtras.fact-ref-to-name;

      (* Only consider theorems from our query. *)

     val deps = inter (fn (ThmExtras.FoundName (nmidx,-), ThmExtras.FoundName
(nmidx',-)) => nmidx = nmidx'
                                  | - => false) results deps
     in deps end
     | - => deps

in
  if only-names then
    Pretty.block
      (Pretty.separate  (map (ThmExtras.pretty-fact only-names ctxt) deps))
  else
  (* Pretty−print resulting theorems. *)
    Pretty.big-list used theorems:
      (map (Pretty.item o single o ThmExtras.pretty-fact only-names ctxt) deps)

end

val - = Context.>> (Context.map-theory Filter-Thms.setup)

end
⟩
```

end

theory *Apply-Trace-Cmd*
imports *Apply-Trace*
keywords *apply-trace* :: *prf-script*
begin

ML⟨

val - =
Outer-Syntax.command @{*command-keyword apply-trace*} *initial refinement step* (*unstructured*)

(*Args.mode only-names* −− (*Scan.option* (*Parse.position Parse.cartouche*)) −− *Method.parse* >>
(*fn* ((*on*,*query*),*text*) => *Toplevel.proofs* (*Apply-Trace.apply-results* {*silent-fail* = *false*}
(*Pretty.writeln ooo* (*Apply-Trace.pretty-deps on query*)) *text*)));

⟩

lemmas [*no-trace*] = *protectI protectD TrueI Eq-TrueI eq-reflection*

lemma $(a \wedge b) = (b \wedge a)$
apply-trace *auto*
oops

lemma $(a \wedge b) = (b \wedge a)$
apply-trace ⟨*intro*⟩ *auto*
oops

lemma
assumes X: $b = a$
assumes Y: $b = a$
shows
$b = a$
apply-trace (*rule* Y)
oops

locale *Apply-Trace-foo* = **fixes** b a
assumes X: $b = a$
begin

lemma shows $b = a$ $b = a$
apply −
apply-trace (*rule Apply-Trace-foo.X*)
prefer *2*
apply-trace (*rule* X)
oops
end

experiment begin

Example of trace for grouped lemmas

definition *ex* :: *nat set* **where**
ex = {*1*,*2*,*3*,*4*}

lemma *v1*: *1* ∈ *ex* **by** (*simp add*: *ex-def*)
lemma *v2*: *2* ∈ *ex* **by** (*simp add*: *ex-def*)
lemma *v3*: *3* ∈ *ex* **by** (*simp add*: *ex-def*)

Group several lemmas in a single one

lemmas *vs* = *v1 v2 v3*

lemma *2* ∈ *ex*
apply-trace (*simp add*: *vs*)
oops

end
end

theory *Apply-Debug*
imports
Apply-Trace
HOL−Eisbach.Eisbach-Tools
keywords
apply-debug :: *prf-script* % *proof* **and**
continue :: *prf-script* % *proof* **and** *finish* :: *prf-script* % *proof*
begin

ML ‹
val start-max-threads = *Multithreading.max-threads* ();
›

context
begin

private method *put-prems* =
(*match* **premises in** *H*:*PROP* - (*multi*) ⇒ ‹*insert H*›)

ML ‹
fun get-match-prems ctxt =
let

val st = *Goal.init* @{*cterm PROP P*}

fun get-wrapped () =
let
val ((-,*st'*),-) =

```
        Method-Closure.apply-method ctxt @{method put-prems} [] [] [] ctxt [] (ctxt,
st)
          |> Seq.first-result prems;

        val prems =
          Thm.prems-of st' |> hd |> Logic.strip-imp-prems;

      in prems end

     val match-prems = the-default [] (try get-wrapped ());

     val all-prems = Assumption.all-prems-of ctxt;

    in map-filter (fn t => find-first (fn thm => t aconv (Thm.prop-of thm))
all-prems) match-prems end

⟩
```

end

ML ⟨

```
signature APPLY-DEBUG =
sig
type break-opts = { tags : string list, trace : (string * Position.T) option, show-running
: bool }

val break : Proof.context -> string option -> tactic;
val apply-debug : break-opts -> Method.text-range -> Proof.state -> Proof.state;
val continue : int option -> (context-state -> context-state option) option ->
Proof.state -> Proof.state;
val finish : Proof.state -> Proof.state;

val pretty-state: Toplevel.state -> Pretty.T option;

end

structure Apply-Debug : APPLY-DEBUG =
struct
type break-opts = { tags : string list, trace : (string * Position.T) option, show-running
: bool }

fun do-markup range m = Output.report [Markup.markup (Markup.properties (Position.properties-of-range
range) m) ];
fun do-markup-pos pos m = Output.report [Markup.markup (Markup.properties
(Position.properties-of pos) m) ];

type markup-queue = { cur : Position.range option, next : Position.range option,
clear-cur : bool }

fun map-cur f ({cur, next, clear-cur} : markup-queue) =
```

```
  ({cur = f cur, next = next, clear-cur = clear-cur} : markup-queue)

fun map-next f ({cur, next, clear-cur} : markup-queue) =
  ({cur = cur, next = f next, clear-cur = clear-cur} : markup-queue)

fun map-clear-cur f ({cur, next, clear-cur} : markup-queue) =
  ({cur = cur, next = next, clear-cur = f clear-cur} : markup-queue)

type markup-state =
  { running : markup-queue
  }

fun map-running f ({running} : markup-state) =
  {running = f running}


structure Markup-Data = Proof-Data
(
  type T = markup-state Synchronized.var option *
    Position.range option (* latest method location *) *
    Position.range option (* latest breakpoint location *)
  fun init - : T = (NONE, NONE, NONE)
);

val init-queue = ({cur = NONE, next = NONE, clear-cur = false}: markup-queue)
val init-markup-state = ({running = init-queue} : markup-state)

fun set-markup-state id = Markup-Data.map (@{apply 3 (1)} (K id));
fun get-markup-id ctxt = #1 (Markup-Data.get ctxt);

fun set-latest-range range = Markup-Data.map (@{apply 3 (2)} (K (SOME range)));
fun get-latest-range ctxt = #2 (Markup-Data.get ctxt);

fun set-breakpoint-range range = Markup-Data.map (@{apply 3 (3)} (K (SOME
range)));
fun get-breakpoint-range ctxt = #3 (Markup-Data.get ctxt);

val clear-ranges = Markup-Data.map (@{apply 3 (3)} (K NONE) o @{apply 3
(2)} (K NONE));

fun swap-markup queue startm endm =
if is-some (#next queue) andalso #next queue = #cur queue then SOME (map-next
(K NONE) queue) else
let
  fun clear-cur () =
    (case #cur queue of SOME crng =>
        do-markup crng endm
      | NONE => ())
in
```

```
  case #next queue of SOME rng =>
     (clear-cur (); do-markup rng startm; SOME ((map-cur (K (SOME rng)) o
map-next (K NONE)) queue))
    | NONE => if #clear-cur queue then (clear-cur (); SOME ((map-cur (K
NONE) o map-clear-cur (K false)) queue))
             else NONE
end

fun markup-worker (SOME (id : markup-state Synchronized.var)) =
let
  fun main-loop () =
    let val - = Synchronized.guarded-access id (fn e =>
    case swap-markup (#running e) Markup.running Markup.finished of
      SOME queue' => SOME ((),map-running (fn - => queue') e)
    | NONE => NONE)
     in main-loop () end
in main-loop () end
 | markup-worker NONE = (fn () => ())

fun set-gen get set (SOME id) rng =
  let
    val - =
       Synchronized.guarded-access id (fn e =>
         if is-some (#next (get e)) orelse (#clear-cur (get e)) then NONE else
         if (#cur (get e)) = SOME rng then SOME ((), e)
         else (SOME ((),(set (map-next (fn - => SOME rng)) e))))

     val - = Synchronized.guarded-access id (fn e => if is-some (#next (get e))
then NONE else SOME ((),e))
  in () end
 | set-gen - - NONE - = ()


fun clear-gen get set (SOME id) =
  Synchronized.guarded-access id (fn e =>
  if (#clear-cur (get e)) then NONE
  else (SOME ((),(set (map-clear-cur (fn - => true)) e))))
 | clear-gen - - NONE = ()

val set-running = set-gen #running map-running
val clear-running = clear-gen #running map-running


fun traceify-method static-ctxt src =
let
  val range = Token.range-of src;
  val head-range = Token.range-of [hd src];
  val m = Method.method-cmd static-ctxt src;
```

```
in (fn eval-ctxt => fn facts =>
  let
    val eval-ctxt = set-latest-range head-range eval-ctxt;
    val markup-id = get-markup-id eval-ctxt;

    fun traceify seq = Seq.make (fn () =>
        let
          val - = set-running markup-id range;
          val r = Seq.pull seq;
          val - = clear-running markup-id;
        in Option.map (apsnd traceify) r end)

    fun tac (runtime-ctxt,thm) =
        let
          val runtime-ctxt' = set-latest-range head-range runtime-ctxt;
          val - = set-running markup-id range;
          in traceify (m eval-ctxt facts (runtime-ctxt', thm)) end

   in tac end)
end

fun add-debug ctxt (Method.Source src) = (Method.Basic (traceify-method ctxt sr-
c))
  | add-debug ctxt (Method.Combinator (x,y,txts)) = (Method.Combinator (x,y,
map (add-debug ctxt) txts))
  | add-debug - x = x

fun st-eq (ctxt : Proof.context,st) (ctxt',st') =
  pointer-eq (ctxt,ctxt') andalso Thm.eq-thm (st,st')

type result =
  { pre-state : thm,
    post-state : thm,
    context: Proof.context}

datatype final-state = RESULT of (Proof.context * thm) | ERR of (unit ->
string)

type debug-state =
  {results : result list, (* this execution, in order of appearance *)
   prev-results : thm list, (* continuations needed to get thread back to some state*)
   next-state : thm option, (* proof thread blocks waiting for this *)
    break-state : (Proof.context * thm) option, (* state of proof thread just before
blocking *)
   restart : (unit -> unit) * int, (* restart function (how many previous results to
keep), restart requested if non-zero *)
   final : final-state option, (* final result, maybe error *)
   trans-id : int, (* increment on every restart *)
   ignore-breaks: bool}
```

```
val init-state =
  ({results = [],
    prev-results = [],
    next-state = NONE, break-state = NONE,
    final = NONE, ignore-breaks = false, restart = (K (), ~1), trans-id = 0} :
debug-state)

fun map-next-state f ({results, next-state, break-state, final, ignore-breaks, prev-results,
restart, trans-id} : debug-state) =
  ({results = results, next-state = f next-state, break-state = break-state, final =
final, prev-results = prev-results,
   restart = restart, ignore-breaks = ignore-breaks, trans-id = trans-id} : debug-state)

fun map-results f ({results, next-state, break-state, final, ignore-breaks, prev-results,
restart, trans-id} : debug-state) =
  ({results = f results, next-state = next-state, break-state = break-state, final =
final, prev-results = prev-results,
   restart = restart, ignore-breaks = ignore-breaks, trans-id = trans-id} : debug-state)

fun map-prev-results f ({results, next-state, break-state, final, ignore-breaks, prev-results,
restart, trans-id} : debug-state) =
  ({results = results, next-state = next-state, break-state = break-state, final =
final, prev-results = f prev-results,
   restart = restart, ignore-breaks = ignore-breaks, trans-id = trans-id} : debug-state)

fun map-ignore-breaks f ({results, next-state, break-state = break-state, final, ignore-breaks,
prev-results, restart, trans-id} : debug-state) =
 ({results = results, next-state = next-state, break-state = break-state,final = final,
prev-results = prev-results,
   restart = restart, ignore-breaks = f ignore-breaks, trans-id = trans-id} : debug-state)

fun map-final f ({results, next-state, break-state, final, ignore-breaks, prev-results,
restart, trans-id} : debug-state) =
  ({results = results, next-state = next-state, break-state =break-state ,final = f
final, prev-results = prev-results,
   restart = restart, ignore-breaks = ignore-breaks, trans-id = trans-id} : debug-state)

fun map-restart f ({results, next-state, break-state, final, ignore-breaks, prev-results,
restart, trans-id} : debug-state) =
  ({results = results, next-state = next-state, break-state = break-state, final =
final, prev-results = prev-results,
   restart = f restart, ignore-breaks = ignore-breaks, trans-id = trans-id} : debug-state)

fun map-break-state f ({results, next-state, break-state, final, ignore-breaks, prev-results,
restart, trans-id} : debug-state) =
  ({results = results, next-state = next-state, break-state = f break-state, final =
final, prev-results = prev-results,
   restart = restart, ignore-breaks = ignore-breaks, trans-id = trans-id} : debug-state)
```

```
fun map-trans-id f ({results, next-state, break-state, final, ignore-breaks, prev-results,
restart, trans-id} : debug-state) =
  ({results = results, next-state = next-state, break-state = break-state, final =
final, prev-results = prev-results,
  restart = restart, ignore-breaks = ignore-breaks, trans-id = f trans-id} : debug-state)

fun is-restarting ({restart,...} : debug-state) = snd restart > ~1;
fun is-finished ({final,...} : debug-state) = is-some final;

val drop-states = map-break-state (K NONE) o map-next-state (K NONE);

fun add-result ctxt pre post = map-results (cons {pre-state = pre, post-state =
post, context = ctxt}) o drop-states;

fun get-trans-id (id : debug-state Synchronized.var) = #trans-id (Synchronized.value
id);

fun stale-transaction-err trans-id trans-id' =
  error (Stale transaction. Expected  ^ Int.toString trans-id ^  but found  ^ In-
t.toString trans-id')

fun assert-trans-id trans-id (e : debug-state) =
  if trans-id = (#trans-id e) then ()
    else stale-transaction-err trans-id (#trans-id e)

fun guarded-access id f =
  let
    val trans-id = get-trans-id id;
  in
  Synchronized.guarded-access id
    (fn (e : debug-state) =>
     (assert-trans-id trans-id e;
       (case f e of
          NONE => NONE
        | SOME (e', g) => SOME (e', g e))))
   end

fun guarded-read id f =
  let
    val trans-id = get-trans-id id;
  in
  Synchronized.guarded-access id
    (fn (e : debug-state) =>
     (assert-trans-id trans-id e;
      (case f e of
          NONE => NONE
        | SOME e' => SOME (e', e))))
   end
```

```
(∗ Immediate return if there are previous results available or we are ignoring break-
points ∗)

fun pop-state-no-block id ctxt pre = guarded-access id (fn e =>
  if is-finished e then error Attempted to pop state from finished proof else
  if (#ignore-breaks e) then SOME (SOME pre, add-result ctxt pre pre) else
  case #prev-results e of
     [] => SOME (NONE, I)
   | (st :: sts) => SOME (SOME st, add-result ctxt pre st o map-prev-results (fn
- => sts)))

fun pop-next-state id ctxt pre = guarded-access id (fn e =>
  if is-finished e then error Attempted to pop state from finished proof else
  if not (null (#prev-results e)) then error Attempted to pop state when previous
results exist else
    if (#ignore-breaks e) then SOME (pre, add-result ctxt pre pre) else
    (case #next-state e of
            NONE => NONE
          | SOME st => SOME (st, add-result ctxt pre st)))

fun set-next-state id trans-id st = guarded-access id (fn e =>
  (assert-trans-id trans-id e;
   (if is-none (#next-state e) andalso is-some (#break-state e) then
      SOME ((), map-next-state (fn - => SOME st) o map-break-state (fn - =>
NONE))
   else error (Attempted to set next state in inconsistent state ^ (@{make-string}
e)))))

fun set-break-state id st = guarded-access id (fn e =>
  if is-none (#next-state e) andalso is-none (#break-state e) then
    SOME ((), map-break-state (fn - => SOME st))
  else error (Attempted to set break state in inconsistent state ^ (@{make-string}
e)))

fun pop-state id ctxt pre =
  case pop-state-no-block id ctxt pre of SOME st => st
  | NONE =>
  let
    val - = set-break-state id (ctxt, pre); (∗ wait for continue ∗)
  in pop-next-state id ctxt pre end

(∗ block until a breakpoint is hit or method finishes ∗)
fun wait-break-state id trans-id = guarded-read id
  (fn e =>
    (assert-trans-id trans-id e;
     (case (#final e) of SOME st => SOME (st, true) | NONE =>
```

```
    case (#break-state e) of SOME st => SOME (RESULT st, false)
   | NONE => NONE)));

fun debug-print (id : debug-state Synchronized.var) =
  (@{print} (Synchronized.value id));

(* Trigger a restart if an existing nth entry differs from the given one *)
fun maybe-restart id n st =
let
  val gen = guarded-read id (fn e => SOME (#trans-id e));

  val did-restart = guarded-access id (fn e =>
    if is-some (#next-state e) then NONE else
    if not (null (#prev-results e)) then NONE else
    if is-restarting e then NONE (* TODO, what to do if we're already restarting?
*)
    else if length (#results e) > n then
      (SOME (true, map-restart (apsnd (fn - => n))))
    else SOME (false, I))


  val trans-id = Synchronized.guarded-access id
    (fn e => if is-restarting e then NONE else
           if not did-restart orelse gen + 1 = #trans-id e then SOME (#trans-id
e,e) else
           stale-transaction-err (gen + 1) (#trans-id e));
in trans-id end;


fun peek-all-results id = guarded-read id (fn e => SOME (#results e));

fun peek-final-result id =
  guarded-read id (fn e => #final e)

fun poke-error (RESULT st) = st
  | poke-error (ERR e) = error (e ())

fun context-state e = (#context e, #pre-state e);

fun nth-pre-result id i = guarded-read id
  (fn e =>
     if length (#results e) > i then SOME (RESULT (context-state (nth (rev
(#results e)) i)), false) else
    if not (null (#prev-results e)) then NONE else
      (if length (#results e) = i then
         (case #break-state e of SOME st => SOME (RESULT st, false) | NONE
=> NONE) else
         (case #final e of SOME st => SOME (st, true) | NONE => NONE)))
```

```
fun set-finished-result id trans-id st =
  guarded-access id (fn e =>
  (assert-trans-id trans-id e;
   SOME ((), map-final (K (SOME st)))));

fun is-finished-result id = guarded-read id (fn e => SOME (is-finished e));

fun get-finish id =
if is-finished-result id then peek-final-result id else
  let
    val - = guarded-access id
      (fn - => SOME ((), (map-ignore-breaks (fn - => true))))

  in peek-final-result id end


val no-break-opts = ({tags = [], trace = NONE, show-running = false} : break-opts)

structure Debug-Data = Proof-Data
(
  type T = debug-state Synchronized.var option  (* handle on active proof thread
*) *
  int * (* continuation counter *)
  bool * (* currently interactive context *)
  break-opts * (* global break arguments *)
  string option (* latest breakpoint tag *)
  fun init - : T = (NONE,~1, false, no-break-opts, NONE)
);

fun set-debug-ident ident = Debug-Data.map  (@{apply 5 (1)} (fn - => SOME
ident))
val get-debug-ident = #1 o Debug-Data.get;
val get-the-debug-ident = the o get-debug-ident;

fun set-break-opts opts = Debug-Data.map (@{apply 5 (4)} (fn - => opts))
val get-break-opts = #4 o Debug-Data.get;

fun set-last-tag tags = Debug-Data.map (@{apply 5 (5)} (fn - => tags))
val get-last-tag = #5 o Debug-Data.get;

val is-debug-ctxt = is-some o #1 o Debug-Data.get;

fun clear-debug ctxt = ctxt
 |> Debug-Data.map (fn - => (NONE,~1,false, no-break-opts, NONE))
 |> clear-ranges


val get-continuation = #2 o Debug-Data.get;
val get-can-break = #3 o Debug-Data.get;
```

```
(* Maintain pointer equality if possible *)
fun set-continuation i ctxt = if get-continuation ctxt = i then ctxt else
  Debug-Data.map (@{apply 5 (2)} (fn - => i)) ctxt;

fun set-can-break b ctxt = if get-can-break ctxt = b then ctxt else
  Debug-Data.map (@{apply 5 (3)} (fn - => b)) ctxt;

fun has-break-tag (SOME tag) tags = member (=) tags tag
  | has-break-tag NONE - = true;

fun break ctxt tag = (fn thm =>
if not (get-can-break ctxt)
   orelse Method.detect-closure-state thm
   orelse not (has-break-tag tag (#tags (get-break-opts ctxt)))
    then Seq.single thm else
  let
    val id = get-the-debug-ident ctxt;
    val ctxt' = set-last-tag tag ctxt;

    val st' = Seq.make (fn () =>
     SOME (pop-state id ctxt' thm,Seq.empty))

  in st' end)

fun init-interactive ctxt = ctxt
  |> set-can-break false
  |> Config.put Method.closure true;

type static-info =
  {private-dyn-facts : string list, local-facts : (string * thm list) list}

structure Data = Generic-Data
(
  type T = (morphism * Proof.context * static-info) option;
  val empty: T = NONE;
  val extend = K NONE;
  fun merge data : T = NONE;
);

(* Present Eisbach/Match variable binding context as normal context elements.
   Potentially shadows existing facts/binds *)

fun dest-local s =
  let
    val [local,s'] = Long-Name.explode s;
  in SOME s' end handle Bind => NONE

fun maybe-bind st (-,[tok]) ctxt =
```

```
if Method.detect-closure-state st then
  let
    val target = Local-Theory.target-of ctxt
    val local-facts = Proof-Context.facts-of ctxt;
   val global-facts = map (Global-Theory.facts-of) (Context.parents-of (Proof-Context.theory-of
ctxt));
    val raw-facts = Facts.dest-all (Context.Proof ctxt) true global-facts local-facts
|> map fst;

    fun can-retrieve s = can (Facts.retrieve (Context.Proof ctxt) local-facts) (s,
Position.none)

    val private-dyns = raw-facts |>
      (filter (fn s => Facts.is-concealed local-facts s andalso Facts.is-dynamic
local-facts s
                  andalso can-retrieve (Long-Name.base-name s)
                  andalso Facts.intern local-facts (Long-Name.base-name s) = s
                  andalso not (can-retrieve s)) )

   val local-facts = Facts.dest-static true [(Proof-Context.facts-of target)] local-facts;

    val - = Token.assign (SOME (Token.Declaration (fn phi =>
     Data.put (SOME (phi,ctxt, {private-dyn-facts = private-dyns, local-facts =
local-facts}))))) tok;

  in ctxt end
else
  let
    val SOME (Token.Declaration decl) = Token.get-value tok;
    val dummy-ctxt = decl Morphism.identity (Context.Proof ctxt);
   val SOME (phi,static-ctxt,{private-dyn-facts, local-facts}) = Data.get dummy-ctxt;

    val old-facts = Proof-Context.facts-of static-ctxt;
    val cur-priv-facts = map (fn s =>
          Facts.retrieve (Context.Proof ctxt) old-facts (Long-Name.base-name
s,Position.none)) private-dyn-facts;

    val cur-local-facts =
      map (fn (s,fact) => (dest-local s, Morphism.fact phi fact)) local-facts
    |> map-filter (fn (s,fact) => case s of SOME s => SOME (s,fact) | - =>
NONE)

    val old-fixes = (Variable.dest-fixes static-ctxt)

    val local-fixes =
      filter (fn (-,f) =>
        Variable.is-newly-fixed static-ctxt (Local-Theory.target-of static-ctxt) f)
old-fixes
      |> map-filter (fn (n,f) => case Variable.default-type static-ctxt f of SOME
```

```
typ =>
          if typ = dummyT then NONE else SOME (n, Free (f, typ))
        | NONE => NONE)

    val local-binds = (map (apsnd (Morphism.term phi)) local-fixes)

    val ctxt' = ctxt
    |> fold (fn (s,t) =>
        Variable.bind-term ((s,0),t)
      #> Variable.declare-constraints (Var ((s,0),Term.fastype-of t))) local-binds
    |> fold (fn e =>
         Proof-Context.put-thms true (Long-Name.base-name (#name e), SOME
(#thms e))) cur-priv-facts
    |> fold (fn (nm,fact) =>
        Proof-Context.put-thms true (nm, SOME fact)) cur-local-facts
   |> Proof-Context.put-thms true (match-prems, SOME (get-match-prems ctxt));

   in ctxt' end
 | maybe-bind - - ctxt = ctxt

val - = Context.>> (Context.map-theory (Method.setup @{binding #}
 (Scan.lift (Scan.trace (Scan.trace (Args.$$$ break) -- (Scan.option Parse.string)))
>>
  (fn ((b,tag),toks) => fn - => fn - =>
   fn (ctxt,thm) =>
     (let

       val range = Token.range-of toks;
       val ctxt' = ctxt
         |> maybe-bind thm b
         |> set-breakpoint-range range;

     in Seq.make-results (Seq.map (fn thm' => (ctxt',thm')) (break ctxt' tag thm))
end))) ))

fun map-state f state =
     let
      val (r,-) = Seq.first-result map-state (Proof.apply
        (Method.Basic (fn - => fn - => fn st =>
           Seq.make-results (Seq.single (f st))),
         Position.no-range) state)
     in r end;

fun get-state state =
let
  val {context,goal} = Proof.simple-goal state;
in (context,goal) end
```

```
fun maybe-trace (SOME (tr, pos)) (ctxt, st) =
let
  val deps = Apply-Trace.used-facts ctxt st;
  val query = if tr =  then NONE else SOME (tr, pos);
  val pr = Apply-Trace.pretty-deps false query ctxt st deps;
in Pretty.writeln pr end
  | maybe-trace NONE (ctxt, st) = ()

val active-debug-threads = Synchronized.var active-debug-threads ([] : unit future
list);

fun update-max-threads extra =
let
  val n-active = Synchronized.change-result active-debug-threads (fn ts =>
    let
      val ts' = List.filter (not o Future.is-finished) ts;
    in (length ts',ts') end)
  val - = Multithreading.max-threads-update (start-max-threads + ((n-active + ex-
tra) * 3));
in () end


fun continue i-opt m-opt =
(map-state (fn (ctxt,thm) =>
      let

        val ctxt = set-can-break true ctxt

        val thm = Apply-Trace.clear-deps thm;

         val - = if is-none (get-debug-ident ctxt) then error Cannot continue in a
non−debug state else ();

        val id = get-the-debug-ident ctxt;

        val start-cont = get-continuation ctxt; (* how many breakpoints so far *)

        val trans-id = maybe-restart id start-cont (ctxt,thm);
          (* possibly restart if the thread has made too much progress.
             trans-id is the current number of restarts, used to avoid manipulating
             stale states *)

         val - = nth-pre-result id start-cont; (* block until we've hit the start of this
continuation *)

        fun get-final n (st as (ctxt,-))  =
         case (i-opt,m-opt) of
           (SOME i,NONE) => if i < 1 then error Can only continue a positive
number of breakpoints else
```

```
    if n = start-cont + i then SOME st else NONE
  | (NONE, SOME m) => (m (apfst init-interactive st))
  | (-, -) => error Invalid continue arguments

  val ex-results = peek-all-results id |> rev;

  fun tick-up n (-,thm) =
    if n < length ex-results then error Unexpected number of existing results
      (*case get-final n (#pre-state (nth ex-results n)) of SOME st' => (st',
false, n)
      | NONE => tick-up (n + 1) st *)
    else
    let
     val - = if n > length ex-results then set-next-state id trans-id thm else ();
      val (n-r, b) = wait-break-state id trans-id;
      val st' = poke-error n-r;
    in if b then (st',b, n) else
      case get-final n st' of SOME st'' => (st'', false, n)
      | NONE => tick-up (n + 1) st' end

  val - = if length ex-results < start-cont then
 (debug-print id; @{print} (start-cont,start-cont); @{print} (trans-id,trans-id);
      error Unexpected number of existing results)
    else ()


  val (st',b, cont) = tick-up (start-cont + 1) (ctxt, thm)

  val st'' = if b then (Output.writeln Final Result.; st' |> apfst clear-debug)
           else st' |> apfst (set-continuation cont) |> apfst (init-interactive);

  (* markup for matching breakpoints to continues *)

  val sr = serial ();

  fun markup-def rng =
    (Output.report
       [Markup.markup (Markup.entity breakpoint
        |> Markup.properties (Position.entity-properties-of true sr
             (Position.range-position rng))) ]);

  val - = Option.map markup-def (get-latest-range (fst st''));
  val - = Option.map markup-def (get-breakpoint-range (fst st''));

  val - =
    (Context-Position.report ctxt (Position.thread-data ())
       (Markup.entity breakpoint
            |> Markup.properties (Position.entity-properties-of false sr Posi-
tion.none)))
```

```
      val _ = maybe_trace (#trace (get_break_opts ctxt)) st'';

    in st'' end))

fun do_apply pos rng opts m =
let
  val {tags, trace, show_running} = opts;
  val batch_mode = is_some (Position.line_of (fst rng));
  val show_running = if batch_mode then false else show_running;

  val _ = if batch_mode then () else update_max_threads 1;

in
 (fn st => map_state (fn (ctxt,thm) =>
  let
     val ident = Synchronized.var debug_state init_state;
     val markup_id = if show_running then SOME (Synchronized.var markup_state
init_markup_state)
       else NONE;
     fun maybe_markup m = if show_running then do_markup rng m else ();

     val _ = if is_debug_ctxt ctxt then
      error Cannot use apply_debug while debugging else ();

     val m = apfst (fn f => f ctxt) m;

     val st = Proof.map_context
      (set_can_break true
       #> set_break_opts opts
       #> set_markup_state markup_id
       #> set_debug_ident ident
       #> set_continuation ~1) st
       |> map_state (apsnd Apply_Trace.clear_deps);

     fun do_cancel thread = (Future.cancel thread; Future.join_result thread; ());

     fun do_fork trans_id = Future.fork (fn () =>
       let
        val (ctxt,thm) = get_state st;

        val r = case Exn.interruptible_capture (fn st =>
        let val _ = Seq.pull (break ctxt NONE thm) in
        (case (Seq.pull o Proof.apply m) st
          of (SOME (Seq.Result st', _)) => RESULT (get_state st')
           | (SOME (Seq.Error e, _)) => ERR e
           | _ => ERR (fn _ => No results)) end) st
          of Exn.Res (RESULT r) => RESULT r
            | Exn.Res (ERR e) => ERR e
```

```
      | Exn.Exn e => ERR (fn - => Runtime.exn-message e)
    val - = set-finished-result ident trans-id r;

    val - = clear-running markup-id;

   in () end)


  val thread = do-fork 0;
  val - = Synchronized.change ident (map-restart (fn - => (fn () => do-cancel
thread, ~1)));

  val - = maybe-markup Markup.finished;

  val - = Future.fork (fn () => markup-worker markup-id ());

 val st' = get-state (continue (SOME 1) NONE (Proof.map-context (set-continuation
0) st))

  val - = maybe-markup Markup.joined;


 val main-thread = if batch-mode then Future.fork (fn () => ()) else Future.fork
(fn () =>
   let

    fun restart-state gls e = e
      |> map-prev-results (fn - => map #post-state (take gls (rev (#results
e))))
     |> map-results (fn - => [])
     |> map-final (fn - => NONE)
     |> map-ignore-breaks (fn - => false)
     |> map-restart (fn - => (K (), gls))
     |> map-break-state (fn - => NONE)
     |> map-next-state (fn - => NONE)
     |> map-trans-id (fn i => i + 1);

    fun main-loop () =
     let
      val r = Synchronized.timed-access ident (fn - => SOME (seconds 0.1))
(fn e as {restart,next-state,...} =>
       if is-restarting e andalso is-none next-state then
        SOME ((fst restart, #trans-id e), restart-state (snd restart) e) else
NONE);
      val - = OS.Process.sleep (seconds 0.1);
      in case r of NONE => main-loop ()
      | SOME (f,trans-id) =>
       let
        val - = f ();
```

```
                val - = clear-running markup-id;
                val thread = do-fork (trans-id + 1);
                 val - = Synchronized.change ident (map-restart (fn - => (fn () =>
do-cancel thread, ~1)))
              in main-loop () end
            end;
        in main-loop () end);

        val - = maybe-markup Markup.running;
        val - = maybe-markup Markup.forked;

        val - = Synchronized.change active-debug-threads (cons main-thread);

   in st' end) st)
end

fun apply-debug opts (m', rng)  =
  let
      val - = Method.report (m', rng);

      val m'' = (fn ctxt => add-debug ctxt m')
      val m = (m'',rng)
      val pos = Position.thread-data ();

     in do-apply pos rng opts m end;

fun quasi-keyword x = Scan.trace (Args.$$$ x) >>
  (fn (s,[tok]) => (Position.reports [(Token.pos-of tok, Markup.quasi-keyword)];
s))

val parse-tags = (Args.parens (quasi-keyword tags |-- Parse.enum1 , Parse.string));
val parse-trace = Scan.option (Args.parens (quasi-keyword trace |-- Scan.option
(Parse.position Parse.cartouche))) >>
  (fn SOME NONE => SOME (, Position.none) | SOME (SOME x) => SOME
x | - => NONE);

val parse-opts1 = (parse-tags -- parse-trace) >>
  (fn (tags,trace) => {tags = tags, trace = trace});

val parse-opts2 = (parse-trace -- (Scan.optional parse-tags [])) >>
  (fn (trace,tags) => {tags = tags, trace = trace});

fun mode s = Scan.optional (Args.parens (quasi-keyword s) >> (K true)) false

val parse-opts = ((parse-opts1 || parse-opts2) -- mode show-running) >>
  (fn ({tags, trace}, show-running) => {tags = tags, trace = trace, show-running
= show-running} : break-opts) ;

val - =
```

```
  Outer-Syntax.command @{command-keyword apply-debug} initial goal refinement
step (unstructured)
    (Scan.trace
      (parse-opts --  Method.parse) >>
    (fn ((opts, (m,-)),toks) => Toplevel.proof (apply-debug opts (m,Token.range-of
toks))));

val finish = map-state (fn (ctxt,-) =>
      let
           val - = if is-none (get-debug-ident ctxt) then error Cannot finish in a
non−debug state else ();
         val f = get-finish (get-the-debug-ident ctxt);
      in f |> poke-error |> apfst clear-debug end)




fun continue-cmd i-opt m-opt state =
let
  val {context,...} = Proof.simple-goal state;
  val check = Method.map-source (Method.method-closure (init-interactive contex-
t))

  val m-opt' = Option.map (check o Method.check-text context o fst) m-opt;

  fun eval-method txt =
    (fn (ctxt,thm) => try (fst o Seq.first-result method) (Method.evaluate txt ctxt
[] (ctxt,thm)))

  val i-opt' = case (i-opt,m-opt) of (NONE,NONE) => SOME 1 | - => i-opt;

in continue i-opt' (Option.map eval-method m-opt') state end

val - =
  Outer-Syntax.command @{command-keyword continue} step to next breakpoint
  (Scan.option Parse.int -- Scan.option Method.parse >> (fn (i-opt,m-opt) =>
    (Toplevel.proof (continue-cmd i-opt m-opt))))

val - =
  Outer-Syntax.command @{command-keyword finish} finish debugging
  (Scan.succeed (Toplevel.proof (continue NONE (SOME (fn - => NONE)))))

fun pretty-hidden-goals ctxt0 thm =
  let
    val ctxt = ctxt0
      |> Config.put show-types (Config.get ctxt0 show-types orelse Config.get ctxt0
show-sorts)
      |> Config.put show-sorts false;
```

```
    val prt-term =
      singleton (Syntax.uncheck-terms ctxt) #>
      Type-Annotation.ignore-free-types #>
      Syntax.unparse-term ctxt;
    val prt-subgoal = prt-term

    fun pretty-subgoal s A =
      Pretty.markup (Markup.subgoal s) [Pretty.str (  ^ s ^ . ), prt-subgoal A];
    fun pretty-subgoals n = map-index (fn (i, A) => pretty-subgoal (string-of-int
(i + n)) A);

    fun collect-extras prop =
      case try Logic.unprotect prop of
      SOME prop' =>
      (if Logic.count-prems prop' > 0 then
        (case try Logic.strip-horn prop'
            of SOME (As, B) =>  As :: collect-extras B
            | NONE => [])
      else [])
      | NONE => []

     val (As,B) = Logic.strip-horn (Thm.prop-of thm);
     val extras' = collect-extras B;
      val extra-goals-limit = Int.max (Config.get ctxt0 Goal-Display.goals-limit −
length As, 0);
     val all-extras = flat (take (length extras' − 1) extras');
     val extras = take extra-goals-limit all-extras;

     val pretty = pretty-subgoals (length As + 1) extras @
       (if extra-goals-limit < length all-extras then
           [Pretty.str (A total of  ^ (string-of-int (length all-extras)) ^  hidden
subgoals...)]
        else [])
  in pretty end

fun pretty-state state =
  if Toplevel.is-proof state
    then
  let
    val st = Toplevel.proof-of state;
    val {goal, context, ...} = Proof.raw-goal st;
    val pretty = Toplevel.pretty-state state;
    val hidden = pretty-hidden-goals context goal;
    val out = pretty @
      (if length hidden > 0 then [Pretty.keyword1 hidden goals] @ hidden else []);
  in SOME (Pretty.chunks out) end
  else NONE

end
```

⟩

ML ⟨*val - =*
```
  Query-Operation.register {name = print-state, pri = Task-Queue.urgent-pri}
    (fn {state = st, output-result, ...} =>
      case Apply-Debug.pretty-state st of
        SOME prt => output-result (Markup.markup Markup.state (Pretty.string-of
prt))
      | NONE => ());⟩
```

end

theory *Find-Names*
imports *Pure*
keywords *find-names* :: *diag*
begin

The **find-names** command, when given a theorem, finds other names the theorem appears under, via matching on the whole proposition. It will not identify unnamed theorems.

ML ⟨

```
local
(* all-facts-of and pretty-ref taken verbatim from non−exposed version
   in Find-Theorems.ML of official Isabelle/HOL distribution *)
fun all-facts-of ctxt =
  let
    val thy = Proof-Context.theory-of ctxt;
    val transfer = Global-Theory.transfer-theories thy;
    val local-facts = Proof-Context.facts-of ctxt;
    val global-facts = Global-Theory.facts-of thy;
  in
   (Facts.dest-all (Context.Proof ctxt) false [global-facts] local-facts
    @ Facts.dest-all (Context.Proof ctxt) false [] global-facts)
   |> maps Facts.selections
   |> map (apsnd transfer)
  end;

fun pretty-ref ctxt thmref =
  let
    val (name, sel) =
      (case thmref of
        Facts.Named ((name, -), sel) => (name, sel)
      | Facts.Fact - => raise Fail Illegal literal fact);
  in
    [Pretty.marks-str (#1 (Proof-Context.markup-extern-fact ctxt name), name),
      Pretty.str (Facts.string-of-selection sel)]
  end;
```

```
in

fun find-names ctxt thm =
  let
    fun eq-filter body thmref = (body = Thm.full-prop-of (snd thmref));
  in
    (filter (eq-filter (Thm.full-prop-of thm))) (all-facts-of ctxt)
    |> map #1
  end;

fun pretty-find-names ctxt thm =
  let
    val results = find-names ctxt thm;
   val position-markup = Position.markup (Position.thread-data ()) Markup.position;
  in
    ((Pretty.mark position-markup (Pretty.keyword1 find-names)) ::
      Par-List.map (Pretty.item o (pretty-ref ctxt)) results)
    |> Pretty.fbreaks |> Pretty.block |> Pretty.writeln
  end

end

val - =
  Outer-Syntax.command @{command-keyword find-names}
    find other names of a named theorem
    (Parse.thms1 >> (fn srcs => Toplevel.keep (fn st =>
      pretty-find-names (Toplevel.context-of st)
        (hd (Attrib.eval-thms (Toplevel.context-of st) srcs)))));
⟩
```

end

theory *TSubst*
imports
  Main
begin

```
method-setup tsubst = ⟨
  Scan.lift (Args.mode asm --
            Scan.optional (Args.parens (Scan.repeat Parse.nat)) [0] --
            Parse.term)
   >> (fn ((asm,occs),t) => (fn ctxt =>
  Method.SIMPLE-METHOD (Subgoal.FOCUS-PARAMS (fn focus => (fn thm
=>
  let
```

```
(* This code used to use Thm.certify-inst in 2014, which was removed.
   The following is just a best guess for what it did. *)
fun certify-inst ctxt (typ-insts, term-insts) =
     (typ-insts
      |> map (fn (tvar, inst) =>
            (Thm.ctyp-of ctxt (TVar tvar),
             Thm.ctyp-of ctxt inst)),
      term-insts
      |> map (fn (var, inst) =>
            (Thm.cterm-of ctxt (Var var),
             Thm.cterm-of ctxt inst)))

val ctxt' = #context focus

val ((-, schematic-terms), ctxt2) =
  Variable.import-inst true [(#concl focus) |> Thm.term-of] ctxt'
  |>> certify-inst ctxt'

 val ctxt3 = fold (fn (t,t') => Variable.bind-term (Thm.term-of t |> Term.dest-Var |> fst, (t' |> Thm.term-of))) schematic-terms ctxt2


val athm = Syntax.read-term ctxt3 t
     |> Object-Logic.ensure-propT ctxt'
     |> Thm.cterm-of ctxt'
     |> Thm.trivial

 val thm' = Thm.instantiate ([], map (apfst (Thm.term-of #> dest-Var)) schematic-terms) thm

in
  (if asm then EqSubst.eqsubst-asm-tac else EqSubst.eqsubst-tac)
    ctxt3 occs [athm] 1 thm'
    |> Seq.map (singleton (Variable.export ctxt3 ctxt'))
  end)) ctxt 1)))
› subst, with term instead of theorem as equation
```

schematic-goal
  assumes a: $\bigwedge x\ y.\ P\ x \Longrightarrow P\ y$
  fixes x :: $'b$
  shows $\bigwedge x\ ::'a\ ::\ type.\ ?Q\ x \Longrightarrow P\ x \wedge ?Q\ x$
  apply (*tsubst* (*asm*) $?Q\ x = (P\ x \wedge P\ x)$)
   apply (*rule refl*)
  apply (*tsubst* $P\ x = P\ y$,*simp add*:a)+
  apply (*tsubst* (*2*) $P\ y = P\ x$, *simp add*:a)
  apply (*clarsimp simp*: a)
  done

end

theory *Time-Methods-Cmd* **imports**
Main
begin

ML ‹
```
structure Time-Methods = struct
  (* Work around Isabelle running every apply method on a dummy proof state *)
  fun skip-dummy-state (method: Method.method) : Method.method =
    fn facts => fn (ctxt, st) =>
      case Thm.prop-of st of
        Const (Pure.prop, -) $ (Const (Pure.term, -) $ Const (Pure.dummy-pattern,
-)) =>
          Seq.succeed (Seq.Result (ctxt, st))
        | - => method facts (ctxt, st);

  (* ML interface. Takes a list of (possibly−named) methods, then calls the supplied
   * callback with the method index (starting from 1), supplied name and timing.
   * Also returns the list of timings at the end. *)
  fun time-methods
        (no-check: bool)
        (skip-fail: bool)
        (callback: (int * string option −> Timing.timing −> unit))
        (maybe-named-methods: (string option * Method.method) list)
        (* like Method.method but also returns timing list *)
        : thm list −> context-state −> (Timing.timing list * context-state Seq.result
Seq.seq)
    = fn facts => fn (ctxt, st) => let
        fun run method =
              Timing.timing (fn () =>
                case method facts (ctxt, st) |> Seq.pull of
                  (* Peek at first result, then put it back *)
                    NONE => (NONE, Seq.empty)
                  | SOME (r as Seq.Result (-, st′), rs) => (SOME st′, Seq.cons r rs)
                  | SOME (r as Seq.Error -, rs) => (NONE, Seq.cons r rs)
              ) ()

        val results = tag-list 1 maybe-named-methods
              |> map (fn (idx1, (maybe-name, method)) =>
                  let val (time, (st′, results)) = run method
                      val - =
                        if Option.isSome st′ orelse not skip-fail
                        then callback (idx1, maybe-name) time
                        else ()
                      val name = Option.getOpt (maybe-name, [method ^ string-of-int
idx1 ^ ])
```

```
            in {name = name, state = st', results = results, time = time} end)

      val canonical-result = hd results
      val other-results = tl results
      val return-val = (map #time results, #results canonical-result)
      fun show-state NONE = @{thm FalseE[where P=METHOD-FAILED]}
        | show-state (SOME st) = st
    in
      if no-check then return-val else
      (* Compare the proof states that we peeked at *)
      case other-results
           |> filter (fn result =>
                 (* It's tempting to use aconv, etc., here instead of (<>), but
                  * minute differences such as bound names in Pure.all can
                  * break a proof script later on. *)
                 Option.map Thm.full-prop-of (#state result) <>
                 Option.map Thm.full-prop-of (#state canonical-result)) of
          [] => return-val
        | (bad-result::-) =>
            raise THM (methods \ ^ #name canonical-result ^
                       \ and \ ^ #name bad-result ^ \ have different results,
                       1, map (show-state o #state) [canonical-result, bad-result])
    end
end
⟩
```

method-setup *time-methods* = ⟨

```
let
 fun scan-flag name = Scan.lift (Scan.optional (Args.parens (Parse.reserved name)
>> K true) false)
  val parse-no-check = scan-flag no-check
  val parse-skip-fail = scan-flag skip-fail
 val parse-maybe-name = Scan.option (Scan.lift (Parse.liberal-name --| Parse.$$$
:))
  fun auto-name (idx1, maybe-name) =
      Option.getOpt (maybe-name, [method  ^ string-of-int idx1 ^ ])
in
  parse-no-check -- parse-skip-fail --
  Scan.repeat1 (parse-maybe-name -- Method.text-closure) >>
  (fn ((no-check, skip-fail), maybe-named-methods-text) => fn ctxt =>
    let
      val max-length = tag-list 1 (map fst maybe-named-methods-text)
                       |> map (String.size o auto-name)
                       |> (fn ls => fold (curry Int.max) ls 0)
      fun pad-name s =
          let val pad-length = max-length + String.size : - String.size s
          in s ^ replicate-string pad-length  end
        fun timing-callback id time = warning (pad-name (auto-name id ^ : ) ^
Timing.message time)
```

```
    val maybe-named-methods = maybe-named-methods-text
        |> map (apsnd (fn method-text => Method.evaluate method-text ctxt))
  val timed-method = Time-Methods.time-methods no-check skip-fail timing-callback
maybe-named-methods
    fun method-discard-times facts st = snd (timed-method facts st)
  in
    method-discard-times
    |> Time-Methods.skip-dummy-state
  end)
end
› Compare running time of several methods on the current proof state
```

end

theory *Try-Attribute*
imports *Main*
begin

ML ‹
```
local

val parse-warn = Scan.lift (Scan.optional (Args.parens (Parse.reserved warn) >>
K true) false)

val attribute-generic = Context.cases Attrib.attribute-global Attrib.attribute

fun try-attribute-cmd (warn, attr-srcs) (ctxt, thm) =
  let
    val attrs = map (attribute-generic ctxt) attr-srcs
    val (th', context') =
      fold (uncurry o Thm.apply-attribute) attrs (thm, ctxt)
      handle e =>
        (if Exn.is-interrupt e then Exn.reraise e
         else if warn then warning (TRY: ignoring exception:  ^ (@{make-string}
e))
         else ();
        (thm, ctxt))
  in (SOME context', SOME th') end

in

val - = Theory.setup
  (Attrib.setup @{binding TRY}
    (parse-warn -- Attrib.attribs >> try-attribute-cmd)
    higher order attribute combinator to try other attributes, ignoring failure)

end
›
```

The *TRY* attribute is an attribute combinator that applies other attributes, ignoring any failures by returning the original state. Note that since attributes are applied separately to each theorem in a theorem list, *TRY* will leave failing theorems unchanged while modifying the rest.

Accepts a "warn" flag to print any errors encountered.

Usage: thm foo[TRY [¡attributes¿]]

thm foo[TRY (warn) [¡attributes¿]]

14 Examples

experiment begin
  lemma *eq1*: $(1 :: nat) = 1 + 0$ **by** *simp*
  lemma *eq2*: $(2 :: nat) = 1 + 1$ **by** *simp*

  lemmas *eqs* = *eq1 TrueI eq2*

'eqs[symmetric]' would fail because there are no unifiers with *True*, but *TRY* ignores that.

  lemma
    $1 + 0 = (1 :: nat)$
    True
    $1 + 1 = (2 :: nat)$
    by (*rule eqs*[*TRY* [*symmetric*]])+

You can chain calls to *TRY* at the top level, to apply different attributes to different theorems.

  lemma *ineq*: $(1 :: nat) < 2$ **by** *simp*
  lemmas *ineqs* = *eq1 ineq*
  lemma
    $1 + 0 = (1 :: nat)$
    $(1 :: nat) \leq 2$
    by (*rule ineqs*[*TRY* [*symmetric*], *TRY* [*THEN order.strict-implies-order*]])+

You can chain calls to *TRY* within each other, to chain more attributes onto particular theorems.

  lemmas *more-eqs* = *eq1 eq2*
  lemma
    $1 = (1 :: nat)$
    $1 + 1 = (2 :: nat)$
    by (*rule more-eqs*[*TRY* [*symmetric*, *TRY* [*simplified add-0-right*]]])+

The 'warn' flag will print out any exceptions encountered. Since *symmetric* doesn't apply to *True* or $1 < 2$, this will log two errors.

  lemmas *yet-another-group* = *eq1 TrueI eq2 ineq*
  thm *yet-another-group*[*TRY* (*warn*) [*symmetric*]]

TRY should handle pretty much anything it might encounter.

```
  thm eq1[TRY (warn) [where x=5]]
  thm eq1[TRY (warn) [OF refl]]
end

end
```

term$_pat : MLantiquotationforpatternmatchingonterms.$

See TermPatternAntiquote$_Testsforexamplesandtests.$

```
theory TermPatternAntiquote imports
  Pure
begin

ML ‹
structure Term-Pattern-Antiquote = struct

val quote-string = quote

(∗ typ matching; doesn't support matching on named TVars.
 ∗ This is because each TVar is likely to appear many times in the pattern. ∗)
fun gen-typ-pattern (TVar -) = -
  | gen-typ-pattern (TFree (v, sort)) =
      Term.TFree ( ^ quote-string v ^ , [ ^ commas (map quote-string sort) ^ ])
  | gen-typ-pattern (Type (typ-head, args)) =
       Term.Type ( ^ quote-string typ-head ^ , [ ^ commas (map gen-typ-pattern
args) ^ ])

(∗ term matching; does support matching on named (non−dummy) Vars.
 ∗ The ML var generated will be identical to the Var name except in
 ∗ indexed names like ?v1.2, which creates the var v12. ∗)
fun gen-term-pattern (Var ((-dummy-, -), -)) = -
  | gen-term-pattern (Var ((v, 0), -)) = v
  | gen-term-pattern (Var ((v, n), -)) = v ^ string-of-int n
  | gen-term-pattern (Const (n, typ)) =
      Term.Const ( ^ quote-string n ^ ,  ^ gen-typ-pattern typ ^ )
  | gen-term-pattern (Free (n, typ)) =
      Term.Free ( ^ quote-string n ^ ,  ^ gen-typ-pattern typ ^ )
  | gen-term-pattern (t as f $ x) =
      (∗ (read-term-pattern -) helpfully generates a dummy var that is
       ∗ applied to all bound vars in scope. We go back and remove them. ∗)
      let fun default () = ( ^ gen-term-pattern f ^ $  ^ gen-term-pattern x ^ );
      in case strip-comb t of
             (h as Var ((-dummy-, -), -), bs) =>
               if forall is-Bound bs then gen-term-pattern h else default ()
           | - => default () end
  | gen-term-pattern (Abs (-, typ, t)) =
      Term.Abs (-,  ^ gen-typ-pattern typ ^ ,  ^ gen-term-pattern t ^ )
  | gen-term-pattern (Bound n) = Bound  ^ string-of-int n
```

(* *Create term pattern. All Var names must be distinct in order to generate ML variables.* *)
fun term-pattern-antiquote ctxt s =
let val pat = Proof-Context.read-term-pattern ctxt s
val add-var-names′ = fold-aterms (fn Var (v, -) => curry (::) v | - => I);
val vars = add-var-names′ pat [] |> filter (fn (n, -) => n <> -dummy-)
val - = if vars = distinct (=) vars then () else
raise TERM (Pattern contains duplicate vars, [pat])
in (ˆ gen-term-pattern pat ˆ) end

end;
val - = Context.>> (Context.map-theory (
ML-Antiquotation.inline @{binding term-pat}
((Args.context -- Scan.lift Args.embedded-inner-syntax)
>> uncurry Term-Pattern-Antiquote.term-pattern-antiquote)))
⟩

end

theory *Trace-Schematic-Insts*
imports
Main
ml−helpers/MLUtils
ml−helpers/TermPatternAntiquote
begin

See Trace$_Schematic_Insts_Test for tests and examples.$

locale *data-stash*
begin

We use this to stash a list of the schematics in the conclusion of the proof state. After running a method, we can read off the schematic instantiations (if any) from this list, then restore the original conclusion. Schematic types are added as "undefined :: ?'a" (for now, we don't worry about types that don't have sort "type").

TODO: there ought to be some standard way of stashing things into the proof state. Find out what that is and refactor

definition *container* :: *′a ⇒ bool ⇒ bool*
where
container a b ≡ True

lemma *proof-state-add*:
Pure.prop PROP P ≡ PROP Pure.prop (*container True xs ⟹ PROP P*)
by (*simp add*: *container-def*)

lemma *proof-state-remove*:
PROP Pure.prop (*container True xs ⟹ PROP P*) ≡ *Pure.prop* (*PROP P*)

by (*simp add*: *container-def*)

lemma *rule-add*:
PROP P $\equiv$ (*container True xs* $\Longrightarrow$ *PROP P*)
by (*simp add*: *container-def*)

lemma *rule-remove*:
(*container True xs* $\Longrightarrow$ *PROP P*) $\equiv$ *PROP P*
by (*simp add*: *container-def*)

lemma *elim*:
container a b
by (*simp add*: *container-def*)

ML ‹

```
signature TRACE-SCHEMATIC-INSTS = sig
  type instantiations = (term * (int * term)) list * (typ * typ) list

  val trace-schematic-insts:
        Method.method -> (instantiations -> unit) -> Method.method
  val default-report:
        Proof.context -> string -> instantiations -> unit

  val trace-schematic-insts-tac:
        Proof.context ->
        (instantiations -> instantiations -> unit) ->
        (thm -> int -> tactic) ->
        thm -> int -> tactic
  val default-rule-report:
        Proof.context -> string -> instantiations -> instantiations -> unit

  val skip-dummy-state: Method.method -> Method.method
  val make-term-container: term list -> term
  val dest-term-container: term -> term list

  val attach-proof-annotations: Proof.context -> term list -> thm -> thm
  val detach-proof-annotations: Proof.context -> thm -> (int * term) list * thm

  val attach-rule-annotations: Proof.context -> term list -> thm -> thm
  val detach-rule-result-annotations: Proof.context -> thm -> (int * term) list *
thm
end

structure Trace-Schematic-Insts: TRACE-SCHEMATIC-INSTS = struct
```

— Each pair is a (schematic, instantiation) pair.
The int in the term instantiations is the number of binders which are due to subgoal bounds.
An explanation: if we instantiate some schematic '?P' within a subgoal like $\bigwedge x\ y.$

Q, it might be instantiated to $\lambda a.\ R\ a\ x$. We need to capture 'x' when reporting the instantiation, so we report that '?P' has been instantiated to $\lambda x\ y\ a.\ R\ a\ x$. In order to distinguish between the bound 'x', 'y', and 'a', we record that the two outermost binders are actually due to the subgoal bounds.

```
type instantiations = (term * (int * term)) list * (typ * typ) list

— Work around Isabelle running every apply method on a dummy proof state
fun skip-dummy-state method =
  fn facts => fn (ctxt, st) =>
    case Thm.prop-of st of
        Const (@{const-name Pure.prop}, -) $
       (Const (@{const-name Pure.term}, -) $ Const (@{const-name Pure.dummy-pattern},
-)) =>
          Seq.succeed (Seq.Result (ctxt, st))
      | - => method facts (ctxt, st);

— Utils
fun rewrite-state-concl eqn st =
  Conv.fconv-rule (Conv.concl-conv (Thm.nprems-of st) (K eqn)) st

— Strip the Pure.prop that wraps proof state conclusions
fun strip-prop ct =
      case Thm.term-of ct of
      Const (@{const-name Pure.prop}, @{typ prop ⇒ prop}) $ - => Thm.dest-arg
ct
      | - => raise CTERM (strip-prop: head is not Pure.prop, [ct])

fun cconcl-of st =
  funpow (Thm.nprems-of st) Thm.dest-arg (Thm.cprop-of st)
  |> strip-prop

fun vars-of-term t =
  Term.add-vars t []
  |> sort-distinct Term-Ord.var-ord

fun type-vars-of-term t =
  Term.add-tvars t []
  |> sort-distinct Term-Ord.tvar-ord

— Create annotation list
fun make-term-container ts =
      fold (fn t => fn container =>
              Const (@{const-name container},
                   fastype-of t --> @{typ bool ⇒ bool}) $
                 t $ container)
        (rev ts) @{term True}

— Retrieve annotation list
fun dest-term-container
```

```
    (Const (@{const-name container}, -) $ x $ list) =
        x :: dest-term-container list
  | dest-term-container - = []
```

— Attach some terms to a proof state, by "hiding" them in the protected goal.

```
fun attach-proof-annotations ctxt terms st =
  let
    val container = make-term-container terms
    (* FIXME: this might affect st's maxidx *)
    val add-eqn =
          Thm.instantiate
            ([],
             [(((P, 0), @{typ prop}), cconcl-of st),
              (((xs, 0), @{typ bool}), Thm.cterm-of ctxt container)])
            @{thm proof-state-add}
  in
    rewrite-state-concl add-eqn st
  end
```

— Retrieve attached terms from a proof state

```
fun detach-proof-annotations ctxt st =
  let
    val st-concl = cconcl-of st
    val (ccontainer', real-concl) = Thm.dest-implies st-concl
    val ccontainer =
          ccontainer'
          |> Thm.dest-arg (* strip Trueprop *)
          |> Thm.dest-arg — strip outer container True
    val terms =
          ccontainer
          |> Thm.term-of
          |> dest-term-container
    val remove-eqn =
          Thm.instantiate
            ([],
             [(((P, 0), @{typ prop}), real-concl),
              (((xs, 0), @{typ bool}), ccontainer)])
            @{thm proof-state-remove}
  in
    (map (pair 0) terms, rewrite-state-concl remove-eqn st)
  end
```

— Attaches the given terms to the given thm by stashing them as a new *container* premise, *after* all the existing premises (this minimises disruption when the rule is used with things like ‘erule‘).

```
fun attach-rule-annotations ctxt terms thm =
  let
    val container = make-term-container terms
    (* FIXME: this might affect thm's maxidx *)
```

```
    val add-eqn =
        Thm.instantiate
          ([],
           [(((P, 0), @{typ prop}), Thm.cconcl-of thm),
            (((xs, 0), @{typ bool}), Thm.cterm-of ctxt container)])
          @{thm rule-add}
  in
    rewrite-state-concl add-eqn thm
  end
```

— Finds all the variables and type variables in the given thm, then uses ‘attach‘ to stash them in a *container* within the thm.
Returns a tuple containing the variables and type variables which were attached this way.

```
fun annotate-with-vars-using (attach: Proof.context -> term list -> thm ->
thm) ctxt thm =
  let
    val tvars = type-vars-of-term (Thm.prop-of thm) |> map TVar
    val tvar-carriers = map (fn tvar => Const (@{const-name undefined}, tvar))
tvars
    val vars = vars-of-term (Thm.prop-of thm) |> map Var
    val annotated-rule = attach ctxt (vars @ tvar-carriers) thm
  in ((vars, tvars), annotated-rule) end

val annotate-rule = annotate-with-vars-using attach-rule-annotations
val annotate-proof-state = annotate-with-vars-using attach-proof-annotations

fun split-and-zip-instantiations (vars, tvars) insts =
    let val (var-insts, tvar-insts) = chop (length vars) insts
    in (vars ~~ var-insts, tvars ~~ map (snd #> fastype-of) tvar-insts) end
```

— Term version of `Thm.dest_arg`.

```
val dest-arg = Term.dest-comb #> snd
```

— Cousin of `Term.strip_abs`.

```
fun strip-all t = (Term.strip-all-vars t, Term.strip-all-body t)
```

— Matches subgoals of the form:
$\bigwedge A\ B\ C.\ [\![X;\ Y;\ Z]\!] \Longrightarrow$ *container True data*
Extracts the instantiation variables from ‘?data‘, and re-applies the surrounding meta abstractions (in this case ‘And¿A B C‘).

```
fun dest-instantiation-container-subgoal t =
    let
      val (vars, goal) = t |> strip-all
      val goal = goal |> Logic.strip-imp-concl
    in
      case goal of
        @{term-pat Trueprop (container True ?data)} =>
            dest-term-container data
```

```
          |> map (fn t => (length vars, Logic.rlist-abs (rev vars, t))) (* reapply
variables *)
          |> SOME
      | - => NONE
    end
```

— Finds the first subgoal with a *container* conclusion. Extracts the data from the container and removes the subgoal.

```
fun detach-rule-result-annotations ctxt st =
  let
    val (idx, data) =
        st
        |> Thm.prems-of
        |> Library.get-index dest-instantiation-container-subgoal
        |> OptionExtras.get-or-else (fn () => error No container subgoal!)
    val st' =
        st
        |> resolve-tac ctxt @{thms elim} (idx + 1)
        |> Seq.hd
  in
    (data, st')
  end
```

— 'abs$_a$llnt'wrapsthefirst'n'lambdaabstractionsin't'withinterleavedPure.allconstructors.Forexample,'abs$_a$
ab.lambda > c.P".Theresultingtermisusuallynotwell − typed.
Used to disambiguate schematic instantiations where the instantiation is a lambda.

```
fun abs-all 0 t = t
  | abs-all n (t as (Abs (v, typ, body))) =
      if n < 0 then error Number of lambdas to wrap should be positive. else
      Const (@{const-name Pure.all}, dummyT)
        $ Abs (v, typ, abs-all (n − 1) body)
  | abs-all n - = error (Expected at least ˆ Int.toString n ˆ more lambdas.)

fun filtered-instantiation-lines ctxt (var-insts, tvar-insts) =
  let
    val vars-lines =
        map (fn (var, (abs, inst)) =>
          if var = inst then (* don't show unchanged *) else
                ˆ Syntax.string-of-term ctxt var ˆ  =>  ˆ
              Syntax.string-of-term ctxt (abs-all abs inst) ˆ \n)
        var-insts
    val tvars-lines =
        map (fn (tvar, inst) =>
          if tvar = inst then (* don't show unchanged *) else
                ˆ Syntax.string-of-typ ctxt tvar ˆ  =>  ˆ
              Syntax.string-of-typ ctxt inst ˆ \n)
        tvar-insts
  in
    vars-lines @ tvars-lines
```

```
end
```

— Default callback for black-box method tracing. Prints nontrivial instantiations to tracing output with the given title line.

```
fun default-report ctxt title insts =
  let
    val all-insts = String.concat (filtered-instantiation-lines ctxt insts)
  (* TODO: add a quiet flag, to suppress output when nothing was instantiated *)
  in title ˆ \n ˆ (if all-insts =  then   (no instantiations)\n else all-insts)
     |> tracing
  end
```

— Default callback for tracing rule applications. Prints nontrivial instantiations to tracing output with the given title line. Separates instantiations of rule variables and goal variables.

```
fun default-rule-report ctxt title rule-insts proof-insts =
  let
    val rule-lines = String.concat (filtered-instantiation-lines ctxt rule-insts)
    val rule-lines =
        if rule-lines =
        then (no rule instantiations)\n
        else rule instantiations:\n  ˆ rule-lines;
    val proof-lines = String.concat (filtered-instantiation-lines ctxt proof-insts)
    val proof-lines =
        if proof-lines =
        then (no goal instantiations)\n
        else goal instantiations:\n  ˆ proof-lines;
  in title ˆ \n ˆ rule-lines ˆ \n ˆ proof-lines |> tracing  end
```

— ‘trace$_s chematic_i nsts_t acctxtcallbacktacticthmidx$‘$doesthefollowing:$
- Produce a *container*-annotated version of ‘thm‘. - Runs ‘tactic‘ on subgoal ‘idx‘, using the annotated version of ‘thm‘. - If the tactic succeeds, call ‘callback‘ with the rule instantiations and the goal instantiations, in that order.

```
fun trace-schematic-insts-tac
    ctxt
    (callback: instantiations −> instantiations −> unit)
    (tactic: thm −> int −> tactic)
    thm idx st =
  let
    val (rule-vars, annotated-rule) = annotate-rule ctxt thm
    val (proof-vars, annotated-proof-state) = annotate-proof-state ctxt st
    val st = tactic annotated-rule idx annotated-proof-state
  in
    st |> Seq.map (fn st =>
      let
        val (rule-terms, st) = detach-rule-result-annotations ctxt st
        val (proof-terms, st) = detach-proof-annotations ctxt st
        val rule-insts = split-and-zip-instantiations rule-vars rule-terms
        val proof-insts = split-and-zip-instantiations proof-vars proof-terms
```

```
      val () = callback rule-insts proof-insts
    in
      st
    end
  )
end
```

— ML interface, calls the supplied function with schematic unifications (will be given all variables, including those that haven't been instantiated).

```
fun trace-schematic-insts (method: Method.method) callback
  = fn facts => fn (ctxt, st) =>
    let
      val (vars, annotated-st) = annotate-proof-state ctxt st
    in (* Run the method *)
      method facts (ctxt, annotated-st)
      |> Seq.map-result (fn (ctxt', annotated-st') => let
            (* Retrieve the stashed list, now with unifications *)
            val (annotations, st') = detach-proof-annotations ctxt' annotated-st'
            val insts = split-and-zip-instantiations vars annotations
            (* Report the list *)
            val - = callback insts
        in (ctxt', st') end)
    end

end
›
```

end

method-setup *trace-schematic-insts* = ‹

```
  let
    open Trace-Schematic-Insts
  in
    (Scan.option (Scan.lift Parse.liberal-name) -- Method.text-closure) >>
    (fn (maybe-title, method-text) => fn ctxt =>
      trace-schematic-insts
          (Method.evaluate method-text ctxt)
          (default-report ctxt
              (Option.getOpt (maybe-title, trace-schematic-insts:)))
      |> skip-dummy-state
    )
  end
```

› *Method combinator to trace schematic variable and type instantiations*

end

theory *Insulin*

imports
Pure
keywords
desugar-term desugar-thm desugar-goal :: *diag*
begin

ML ‹
```
structure Insulin = struct

val desugar-random-tag = dsfjdssdfsd
fun fresh-substring s = let
  fun next [] = [#a]
    | next (#z :: n) = #a :: next n
    | next (c :: n) = Char.succ c :: n
  fun fresh n = let
    val ns = String.implode n
    in if String.isSubstring ns s then fresh (next n) else ns end
  in fresh [#a] end

(* Encode a (possibly qualified) constant name as an (expected−to−be−)unused
name.
 * The encoded name will be treated as a free variable. *)
fun escape-const c = let
  val delim = fresh-substring c
  in desugar-random-tag ^ delim ^ - ^
       String.concat (case Long-Name.explode c of
                        (a :: b :: xs) => a :: map (fn x => delim ^ x) (b :: xs)
                      | xs => xs)
  end
(* Decode; if it fails, return input string *)
fun unescape-const s =
  if not (String.isPrefix desugar-random-tag s) then s else
 let val cs = String.extract (s, String.size desugar-random-tag, NONE) |> String.explode
      fun readDelim d (#- :: cs) = (d, cs)
        | readDelim d (c :: cs) = readDelim (d @ [c]) cs
      val (delim, cs) = readDelim [] cs
      val delimlen = length delim
      fun splitDelim name cs =
            if take delimlen cs = delim then name :: splitDelim [] (drop delimlen cs)
              else case cs of [] => if null name then [] else [name]
                            | (c::cs) => splitDelim (name @ [c]) cs
      val names = splitDelim [] cs
  in Long-Name.implode (map String.implode names) end
  handle Match => s

fun dropQuotes s = if String.isPrefix \ s andalso String.isSuffix \ s
                      then String.substring (s, 1, String.size s − 2) else s
```

```
(* Translate markup from consts−encoded−as−free−variables to actual consts *)
fun desugar-reconst ctxt (tr as XML.Elem ((tag, attrs), children))
  = if tag = fixed orelse tag = intensify then
      let val s = XML.content-of [tr]
          val name = unescape-const s
          fun get-entity-attrs (XML.Elem ((entity, attrs), -)) = SOME attrs
            | get-entity-attrs (XML.Elem (-, body)) =
                find-first (K true) (List.mapPartial get-entity-attrs body)
            | get-entity-attrs (XML.Text -) = NONE
      in
        if name = s then tr else
          (* try to look up the const's info *)
          case Syntax.read-term ctxt name
               |> Thm.cterm-of ctxt
               |> Proof-Display.pp-cterm (fn - => Proof-Context.theory-of ctxt)
               |> Pretty.string-of
               |> dropQuotes
               |> YXML.parse
               |> get-entity-attrs of
              SOME attrs =>
                XML.Elem ((entity, attrs), [XML.Text name])
            | - =>
                XML.Elem ((entity, [(name, name), (kind, constant)]),
                          [XML.Text name]) end
    else XML.Elem ((tag, attrs), map (desugar-reconst ctxt) children)
  | desugar-reconst - (t as XML.Text -) = t

fun term-to-string ctxt no-markup =
  Syntax.pretty-term ctxt
  #> Pretty.string-of
  #> YXML.parse-body
  #> map (desugar-reconst ctxt)
  #> (if no-markup then XML.content-of else YXML.string-of-body)
  #> dropQuotes

(* Strip constant names from a term.
 * A term is split to a term-unconst and a string list of the
 * const names in tree preorder. *)
datatype term-unconst =
    UCConst of typ |
    UCAbs of string * typ * term-unconst |
    UCApp of term-unconst * term-unconst |
    UCVar of term

fun is-ident-char c = Char.isAlphaNum c orelse c = #- orelse c = #. orelse c =
#'

fun term-to-unconst (Const (name, typ)) =
```

```
  (* some magical constants have strange names, such as ==>; ignore them *)
    if forall is-ident-char (String.explode name) then (UCConst typ, [name])
      else (UCVar (Const (name, typ)), [])
  | term-to-unconst (Abs (var, typ, body)) = let
      val (body', consts) = term-to-unconst body
      in (UCAbs (var, typ, body'), consts) end
  | term-to-unconst (f $ x) = let
      val (f', consts1) = term-to-unconst f
      val (x', consts2) = term-to-unconst x
      in (UCApp (f', x'), consts1 @ consts2) end
  | term-to-unconst t = (UCVar t, [])

fun term-from-unconst (UCConst typ) (name :: consts) =
      ((if unescape-const name = name then Const else Free) (name, typ), consts)
  | term-from-unconst (UCAbs (var, typ, body)) consts = let
      val (body', consts) = term-from-unconst body consts
      in (Abs (var, typ, body'), consts) end
  | term-from-unconst (UCApp (f, x)) consts = let
      val (f', consts) = term-from-unconst f consts
      val (x', consts) = term-from-unconst x consts
      in (f' $ x', consts) end
  | term-from-unconst (UCVar v) consts = (v, consts)

(* Count occurrences of bad strings.
 * Bad strings are allowed to overlap, but for each string, non−overlapping occur-
rences are counted.
 * Note that we search on string lists, to deal with symbols correctly. *)
fun count-matches (haystack: ''a list) (needles: ''a list list): int list =
  let (* Naive algorithm. Probably ok, given that we're calling the term printer a
lot elsewhere. *)
      fun try-match xs [] = SOME xs
        | try-match (x::xs) (y::ys) = if x = y then try-match xs ys else NONE
        | try-match - - = NONE
      fun count [] = 0
        | count needle = let
            fun f [] occs = occs
              | f haystack' occs = case try-match haystack' needle of
                                      NONE => f (tl haystack') occs
                                    | SOME tail => f tail (occs + 1)
            in f haystack 0 end
  in map count needles end

fun focus-list (xs: 'a list): ('a list * 'a * 'a list) list =
  let fun f head x [] = [(head, x, [])]
        | f head x (tail as x'::tail') = (head, x, tail) :: f (head @ [x]) x' tail'
  in case xs of [] => []
              | (x::xs) => f [] x xs end

(* Do one rewrite pass: try every constant in sequence, then collect the ones which
```

```
 * reduced the occurrences of bad strings *)
fun rewrite-pass ctxt (t: term) (improved: term -> bool) (escape-const: string ->
string): term =
  let val (ucterm, consts) = term-to-unconst t
      fun rewrite-one (prev, const, rest) =
            let val (t', []) = term-from-unconst ucterm (prev @ [escape-const const]
@ rest)
            in improved t' end
      val consts-to-rewrite = focus-list consts |> map rewrite-one
      val consts' = map2 (fn rewr => fn const => if rewr then escape-const const
else const) consts-to-rewrite consts
      val (t', []) = term-from-unconst ucterm consts'
  in t' end

(* Do rewrite passes until bad strings are gone or no more rewrites are possible *)
fun desugar ctxt (t0: term) (bads: string list): term =
   let fun count t = count-matches (Symbol.explode (term-to-string ctxt true t))
(map Symbol.explode bads)
      val - = if null bads then error Nothing to desugar else ()
      fun rewrite t = let
        val counts0 = count t
        fun improved t' = exists (<) (count t' ~~ counts0)
        val t' = rewrite-pass ctxt t improved escape-const
        in if forall (fn c => c = 0) (count t') (* bad strings gone *)
           then t'
           else if t = t' (* no more rewrites *)
             then let
               val bads' = filter (fn (c, -) => c > 0) (counts0 ~~ bads) |> map snd
               val - = warning (Sorry, failed to desugar  ^ commas-quote bads')
               in t end
             else rewrite t'
         end
  in rewrite t0 end

fun span - [] = ([],[])
  | span p (a::s) =
      if p a then let val (y, n) = span p s in (a::y, n) end else ([], a::s)

fun check-desugar s = let
  fun replace [] = []
    | replace xs =
      if take (String.size desugar-random-tag) xs = String.explode desugar-random-tag
          then case span is-ident-char xs of
                   (v, xs) => String.explode (unescape-const (String.implode v)) @
replace xs
          else hd xs :: replace (tl xs)
  val desugar-string = String.implode o replace o String.explode
  in if not (String.isSubstring desugar-random-tag s) then s
       else desugar-string s end
```

```
fun desugar-term ctxt t s =
  desugar ctxt t s |> term-to-string ctxt false |> check-desugar

fun desugar-thm ctxt thm s = desugar-term ctxt (Thm.prop-of thm) s

fun desugar-goal ctxt goal n s = let
  val subgoals = goal |> Thm.prems-of
  val subgoals = if n = 0 then subgoals else
                 if n < 1 orelse n > length subgoals then
                      (* trigger error *) [Logic.get-goal (Thm.term-of (Thm.cprop-of
goal)) n]
                 else [nth subgoals (n − 1)]
  val results = map (fn t => (NONE, desugar-term ctxt t s)
                             handle ex as TERM - => (SOME ex, term-to-string ctxt
false t))
                    subgoals
  in if null results
       then error No subgoals to desugar
     else if forall (Option.isSome o fst) results
       then raise the (fst (hd results))
     else map snd results
  end

end
⟩

ML ⟨
Outer-Syntax.command @{command-keyword desugar-term}
  term str str2... −> desugar str in term
  (Parse.term −− Scan.repeat1 Parse.string >> (fn (t, s) =>
    Toplevel.keep (fn state => let val ctxt = Toplevel.context-of state in
      Insulin.desugar-term ctxt (Syntax.read-term ctxt t) s
      |> writeln end)))
⟩

ML ⟨
Outer-Syntax.command @{command-keyword desugar-thm}
  thm str str2... −> desugar str in thm
  (Parse.thm −− Scan.repeat1 Parse.string >> (fn (t, s) =>
    Toplevel.keep (fn state => let val ctxt = Toplevel.context-of state in
      Insulin.desugar-thm ctxt (Attrib.eval-thms ctxt [t] |> hd) s |> writeln end)))
⟩

ML ⟨
fun print-subgoals (x::xs) n = (writeln (Int.toString n ^ .  ^ x); print-subgoals xs
(n+1))
  | print-subgoals [] - = ();
Outer-Syntax.command @{command-keyword desugar-goal}
```

```
  goal-num str str2... -> desugar str in goal
  (Scan.option Parse.int -- Scan.repeat1 Parse.string >> (fn (n, s) =>
    Toplevel.keep (fn state => let val ctxt = Toplevel.context-of state in
      Insulin.desugar-goal ctxt (Toplevel.proof-of state |> Proof.raw-goal |> #goal)
(Option.getOpt (n, 0)) s
      |> (fn xs => case xs of
            [x] => writeln x
            | - => print-subgoals xs 1) end)))
›
```

end

theory *ShowTypes* **imports**
  Main
keywords *term-show-types thm-show-types goal-show-types* :: *diag*
begin

ML ‹
```
structure Show-Types = struct

fun pretty-markup-to-string no-markup =
  Pretty.string-of
  #> YXML.parse-body
  #> (if no-markup then XML.content-of else YXML.string-of-body)

fun term-show-types no-markup ctxt term =
  let val keywords = Thy-Header.get-keywords' ctxt
      val ctxt' = ctxt
      |> Config.put show-markup false
      |> Config.put Printer.show-type-emphasis false

      (* FIXME: the sledgehammer code also sets these,
       *        but do we always want to force them on the user? *)
      (*
      |> Config.put show-types false
      |> Config.put show-sorts false
      |> Config.put show-consts false
      *)
      |> Variable.auto-fixes term
  in
    singleton (Syntax.uncheck-terms ctxt') term
    |> Sledgehammer-Isar-Annotate.annotate-types-in-term ctxt'
    |> Syntax.unparse-term ctxt'
    |> pretty-markup-to-string no-markup
  end
```

```
fun goal-show-types no-markup ctxt goal n = let
  val subgoals = goal |> Thm.prems-of
  val subgoals = if n = 0 then subgoals else
                 if n < 1 orelse n > length subgoals then
                     (* trigger error *) [Logic.get-goal (Thm.term-of (Thm.cprop-of
goal)) n]
                 else [nth subgoals (n - 1)]
  val results = map (fn t => (NONE, term-show-types no-markup ctxt t)
                              handle ex as TERM - => (SOME ex, term-show-types
no-markup ctxt t))
                   subgoals
  in if null results
       then error No subgoals to show
     else if forall (Option.isSome o fst) results
       then raise the (fst (hd results))
     else map snd results
  end

end;

Outer-Syntax.command @{command-keyword term-show-types}
  term-show-types TERM -> show TERM with type annotations
  (Parse.term >> (fn t =>
    Toplevel.keep (fn state =>
      let val ctxt = Toplevel.context-of state in
        Show-Types.term-show-types false ctxt (Syntax.read-term ctxt t)
        |> writeln end)));

Outer-Syntax.command @{command-keyword thm-show-types}
  thm-show-types THM1 THM2 ... -> show theorems with type annotations
  (Parse.thms1 >> (fn ts =>
    Toplevel.keep (fn state =>
      let val ctxt = Toplevel.context-of state in
        Attrib.eval-thms ctxt ts
      |> app (Thm.prop-of #> Show-Types.term-show-types false ctxt #> writeln)
end)));

let
  fun print-subgoals (x::xs) n = (writeln (Int.toString n ^ . ^ x); print-subgoals xs
(n+1))
    | print-subgoals [] - = ();
in
Outer-Syntax.command @{command-keyword goal-show-types}
  goal-show-types [N] -> show subgoals (or Nth goal) with type annotations
  (Scan.option Parse.int >> (fn n =>
    Toplevel.keep (fn state =>
      let val ctxt = Toplevel.context-of state
          val goal = Toplevel.proof-of state |> Proof.raw-goal |> #goal
      in Show-Types.goal-show-types false ctxt goal (Option.getOpt (n, 0))
```

```
    |> (fn xs => case xs of
                    [x] => writeln x
                  | _ => print_subgoals xs 1) end)))
end;
⟩
```

end

theory *AutoLevity-Base*
imports *Main Apply-Trace*
keywords *levity-tag* :: *thy-decl*
begin

ML ⟨
```
fun is_simp (_: Proof.context) (_: thm) = true
⟩
```

ML ⟨
```
val is_simp_installed = is_some (
 try (ML_Context.eval ML_Compiler.flags @{here})
  (ML_Lex.read_text (val is_simp = Raw_Simplifier.is_simp, @{here} )));
⟩
```

ML⟨
```
(* Describing a ordering on Position.T. Optionally we compare absolute document position, or
   just line numbers. Somewhat complicated by the fact that jEdit positions don't have line or
   file identifiers. *)

fun pos_ord use_offset (pos1, pos2) =
  let
    fun get_offset pos = if use_offset then Position.offset_of pos else SOME 0;

    fun get_props pos =
      (SOME (Position.file_of pos |> the,
             (Position.line_of pos |> the,
              get_offset pos |> the)), NONE)
      handle Option => (NONE, Position.parse_id pos)

    val props1 = get_props pos1;
    val props2 = get_props pos2;

  in prod_ord
      (option_ord (prod_ord string_ord (prod_ord int_ord int_ord)))
      (option_ord (int_ord))
      (props1, props2) end
```

```
structure Postab = Table(type key = Position.T val ord = (pos-ord false));
structure Postab-strict = Table(type key = Position.T val ord = (pos-ord true));


signature AUTOLEVITY-BASE =
sig
type extras = {levity-tag : string option, subgoals : int}



val get-transactions : unit -> ((string * extras) Postab-strict.table * string list
Postab-strict.table) Symtab.table;

val get-applys : unit -> ((string * string list) list) Postab-strict.table Symtab.table;

val add-attribute-test: string -> (Proof.context -> thm -> bool) -> theory ->
theory;

val attribs-of: Proof.context -> thm -> string list;

val used-facts: Proof.context option -> thm -> (string * thm) list;
val used-facts-attribs: Proof.context -> thm -> (string * string list) list;

(*
  Returns the proof body form of the prop proved by a theorem.

  Unfortunately, proof bodies don't contain terms in the same form as what you'd
get
  from things like `Thm.full-prop-of`: the proof body terms have sort constraints
  pulled out as separate assumptions, rather than as annotations on the types of
  terms.

  It's easier for our dependency-tracking purposes to treat this transformed
  term as the 'canonical' form of a theorem, since it's always available as the
  top-level prop of a theorem's proof body.
*)
val proof-body-prop-of: thm -> term;

(*
  Get every (named) term that was proved in the proof body of the given thm.

  The returned terms are in proof body form.
*)
val used-named-props-of: thm -> (string * term) list;

(*
  Distinguish whether the thm name foo-3 refers to foo(3) or foo-3 by comparing
  against the given term. Assumes the term is in proof body form.
```

```
  The provided context should match the context used to extract the (name, prop)
pair
  (that is, it should match the context used to extract the thm passed into
  ‘proof-body-prop-of‘ or ‘used-named-props-of‘).

  Returns SOME (foo, SOME 3) if the answer is 'it refers to foo(3)'.
  Returns SOME (foo-3, NONE) if the answer is 'it refers to foo-3'.
  Returns NONE if the answer is 'it doesn't seem to refer to anything.'
*)
val disambiguate-indices: Proof.context -> string * term -> (string * int option)
option;

(* Install toplevel hook for tracking command positions. *)

val setup-command-hook: {trace-apply : bool} -> theory -> theory;

(* Used to trace the dependencies of all apply statements.
   They are set up by setup-command-hook if the appropriate hooks in the Proof
   module exist. *)

val pre-apply-hook: Proof.context -> Method.text -> thm -> thm;
val post-apply-hook: Proof.context -> Method.text -> thm -> thm -> thm;


end;




structure AutoLevity-Base : AUTOLEVITY-BASE =
struct

val applys = Synchronized.var applys
  (Symtab.empty : (((string * string list) list) Postab-strict.table) Symtab.table)

fun get-applys () = Synchronized.value applys;

type extras = {levity-tag : string option, subgoals : int}


val transactions = Synchronized.var hook
 (Symtab.empty : ((string * extras) Postab-strict.table * ((string list) Postab-strict.table))
Symtab.table);

fun get-transactions () =
  Synchronized.value transactions;
```

```
structure Data = Theory-Data
  (
    type T = (bool *
        string option *
       (Proof.context -> thm -> bool) Symtab.table); (* command-hook * levity
tag * attribute tests *)
    val empty = (false, NONE, Symtab.empty);
    val extend = I;
     fun merge (((b1, -, tab), (b2, -, tab')) : T * T) = (b1 orelse b2, NONE,
Symtab.merge (fn - => true) (tab, tab'));
  );

val set-command-hook-flag = Data.map (@{apply 3(1)} (fn - => true));
val get-command-hook-flag = #1 o Data.get

fun set-levity-tag tag = Data.map (@{apply 3(2)} (fn - => tag));
val get-levity-tag = #2 o Data.get

fun update-attrib-tab f = Data.map (@{apply 3(3)} f);


fun add-attribute-test nm f =
let
  val f' = (fn ctxt => fn thm => the-default false (try (f ctxt) thm))
in update-attrib-tab (Symtab.update-new (nm,f')) end;

val get-attribute-tests = Symtab.dest o #3 o Data.get;

(* Internal fact names get the naming scheme foo-3 to indicate the third
   member of the multi-thm foo. We need to do some work to guess if
   such a fact refers to an indexed multi-thm or a real fact named foo-3 *)

fun base-and-index nm =
let
  val exploded = space-explode - nm;
  val base =
    (exploded, (length exploded) - 1)
      |> try (List.take #> space-implode -)
      |> Option.mapPartial (Option.filter (fn nm => nm <> ))
  val idx = exploded |> try (List.last #> Int.fromString) |> Option.join;
in
  case (base, idx) of
    (SOME base, SOME idx) => SOME (base, idx)
  | - => NONE
end

fun maybe-nth idx xs = idx |> try (curry List.nth xs)

fun fact-from-derivation ctxt prop xnm =
```

```
let
  val facts = Proof-Context.facts-of ctxt;
  (* TODO: Check that exported local fact is equivalent to external one *)
  fun check-prop thm = Thm.full-prop-of thm = prop

  fun entry (name, idx) =
    (name, Position.none)
      |> try (Facts.retrieve (Context.Proof ctxt) facts)
      |> Option.mapPartial (#thms #> maybe-nth (idx - 1))
      |> Option.mapPartial (Option.filter check-prop)
      |> Option.map (pair name)

  val idx-result = (base-and-index xnm) |> Option.mapPartial entry
  val non-idx-result = (xnm, 1) |> entry

  val - =
    if is-some idx-result andalso is-some non-idx-result
    then warning (
       Levity: found two possible results for name  ^ quote xnm ^  with the same
prop:\n ^
      (@{make-string} (the idx-result)) ^ ,\nand\n ^
      (@{make-string} (the non-idx-result)) ^ .\nUsing the first one.)
    else ()
in
  merge-options (idx-result, non-idx-result)
end


(* Local facts (from locales) aren't marked in proof bodies, we only
   see their external variants. We guess the local name from the external one
   (i.e. Theory-Name.Locale-Name.foo -> foo)

   This is needed to perform localized attribute tests (e.g.. is this locale assumption
marked as simp?) *)

(* TODO: extend-locale breaks this naming scheme by adding the chunk qualifier.
This can
   probably just be handled as a special case *)

fun most-local-fact-of ctxt xnm prop =
let
 val local-name = xnm |> try (Long-Name.explode #> tl #> tl #> Long-Name.implode)
  val local-result = local-name |> Option.mapPartial (fact-from-derivation ctxt
prop)
  fun global-result () = fact-from-derivation ctxt prop xnm
in
  if is-some local-result then local-result else global-result ()
end
```

```
fun thms-of (PBody {thms,...}) = thms

(* We recursively descend into the proof body to find dependent facts.
   We skip over empty derivations or facts that we fail to find, but recurse
  into their dependents. This ensures that an attempt to re−build the proof dependencies
   graph will result in a connected graph. *)

fun proof-body-deps
  (filter-name: string −> bool)
  (get-fact: string −> term −> (string ∗ thm) option)
  (thm-ident, thm-node)
  (tab: (string ∗ thm) option Inttab.table) =
let
  val name = Proofterm.thm-node-name thm-node
  val body = Proofterm.thm-node-body thm-node
  val prop = Proofterm.thm-node-prop thm-node
  val result = if filter-name name then NONE else get-fact name prop
  val is-new-result = not (Inttab.defined tab thm-ident)
  val insert = if is-new-result then Inttab.update (thm-ident, result) else I
  val descend =
    if is-new-result andalso is-none result
    then fold (proof-body-deps filter-name get-fact) (thms-of (Future.join body))
    else I
in
  tab |> insert |> descend
end

fun used-facts opt-ctxt thm =
let
  val nm = Thm.get-name-hint thm;
  val get-fact =
    case opt-ctxt of
      SOME ctxt => most-local-fact-of ctxt
    | NONE => fn name => fn - => (SOME (name, Drule.dummy-thm));
  val body = thms-of (Thm.proof-body-of thm);
  fun filter-name nm′ = nm′ =  orelse nm′ = nm;
in
  fold (proof-body-deps filter-name get-fact) body Inttab.empty
    |> Inttab.dest |> map-filter snd
end

fun attribs-of ctxt =
let
  val tests = get-attribute-tests (Proof-Context.theory-of ctxt)
  |> map (apsnd (fn test => test ctxt));

in (fn t => map-filter (fn (testnm, test) => if test t then SOME testnm else
NONE) tests) end;
```

```
fun used-facts-attribs ctxt thm =
let
  val fact-nms = used-facts (SOME ctxt) thm;

  val attribs-of = attribs-of ctxt;

in map (apsnd attribs-of) fact-nms end

local
  fun app3 f g h x = (f x, g x, h x);

  datatype ('a, 'b) Either =
      Left of 'a
    | Right of 'b;

  local
    fun partition-map-foldr f (x, (ls, rs)) =
      case f x of
        Left l => (l :: ls, rs)
      | Right r => (ls, r :: rs);
  in
    fun partition-map f = List.foldr (partition-map-foldr f) ([], []);
  end

  (*
    Extracts the bits we care about from a thm-node: the name, the prop,
    and (the next steps of) the proof.
  *)
  val thm-node-dest =
    app3
      Proofterm.thm-node-name
      Proofterm.thm-node-prop
      (Proofterm.thm-node-body #> Future.join);

  (*
    Partitioning function for thm-node data. We want to insert any named props,
   then recursively find the named props used by any unnamed intermediate/anonymous
props.
  *)
  fun insert-or-descend (name, prop, proof) =
    if name =  then Right proof else Left (name, prop);

  (*
    Extracts the next layer of proof data from a proof step.
  *)
  val next-level = thms-of #> List.map (snd #> thm-node-dest);

  (*
    Secretly used as a set, using '()' as the values.
```

```
  *)
  structure NamePropTab = Table(
    type key = string * term;
    val ord = prod-ord fast-string-ord Term-Ord.fast-term-ord);

  val insert-all = List.foldr (fn (k, tab) => NamePropTab.update (k, ()) tab)

  (*
     Proofterm.fold-body-thms unconditionally recursively descends into the proof
body,
    so instead of only getting the topmost named props we'd get -all- of them. Here
    we do a more controlled recursion.
  *)
  fun used-props-foldr (proof, named-props) =
    let
      val (to-insert, child-proofs) =
        proof |> next-level |> partition-map insert-or-descend;
      val thms = insert-all named-props to-insert;
    in
      List.foldr used-props-foldr thms child-proofs
    end;

  (*
    Extracts the outermost proof step of a thm (which is just the proof of the prop
of the thm).
  *)
  val initial-proof =
    Thm.proof-body-of
      #> thms-of
      #> List.hd
      #> snd
      #> Proofterm.thm-node-body
      #> Future.join;

in
  fun used-named-props-of thm =
    let val used-props = used-props-foldr (initial-proof thm, NamePropTab.empty);
    in used-props |> NamePropTab.keys
    end;
end

val proof-body-prop-of =
  Thm.proof-body-of
    #> thms-of
    #> List.hd
    #> snd
    #> Proofterm.thm-node-prop

local
```

```
  fun thm-matches prop thm = proof-body-prop-of thm = prop

  fun entry ctxt prop (name, idx) =
    name
      |> try (Proof-Context.get-thms ctxt)
      |> Option.mapPartial (maybe-nth (idx − 1))
      |> Option.mapPartial (Option.filter (thm-matches prop))
      |> Option.map (K (name, SOME idx))

  fun warn-if-ambiguous
      name
      (idx-result: (string ∗ int option) option)
      (non-idx-result: (string ∗ int option) option) =
    if is-some idx-result andalso is-some non-idx-result
    then warning (
       Levity: found two possible results for name  ^ quote name ^  with the same
prop:\n ^
      (@{make-string} (the idx-result)) ^ ,\nand\n ^
      (@{make-string} (the non-idx-result)) ^ .\nUsing the first one.)
    else ()

in
  fun disambiguate-indices ctxt (name, prop) =
    let
      val entry = entry ctxt prop
      val idx-result = (base-and-index name) |> Option.mapPartial entry
      val non-idx-result = (name, 1) |> entry |> Option.map (apsnd (K NONE))
      val - = warn-if-ambiguous name idx-result non-idx-result
    in
      merge-options (idx-result, non-idx-result)
    end
end

(∗ We identify apply applications by the document position of their corresponding
method.
   We can only get a document position out of real methods, so internal methods
   (i.e. Method.Basic) won't have a position.∗)

fun get-pos-of-text' (Method.Source src) = SOME (snd (Token.name-of-src src))
   | get-pos-of-text' (Method.Combinator (-, -, texts)) = get-first get-pos-of-text'
texts
  | get-pos-of-text' - = NONE

(∗ We only want to apply our hooks in batch mode, so we test if our position has a
line number
   (in jEdit it will only have an id number) ∗)

fun get-pos-of-text text = case get-pos-of-text' text of
  SOME pos => if is-some (Position.line-of pos) then SOME pos else NONE
```

```
  | NONE => NONE

(* Clear the theorem dependencies using the apply-trace oracle, then
   pick up the new ones after the apply step is finished. *)

fun pre-apply-hook ctxt text thm =
  case get-pos-of-text text of NONE => thm
  | SOME - =>
      if Apply-Trace.can-clear (Proof-Context.theory-of ctxt)
      then Apply-Trace.clear-deps thm
      else thm;


val post-apply-hook = (fn ctxt => fn text => fn pre-thm => fn post-thm =>
  case get-pos-of-text text of NONE => post-thm
  | SOME pos => if Apply-Trace.can-clear (Proof-Context.theory-of ctxt) then
    (let
      val thy-nm = Context.theory-name (Thm.theory-of-thm post-thm);

      val used-facts = the-default [] (try (used-facts-attribs ctxt) post-thm);
      val - =
        Synchronized.change applys
         (Symtab.map-default
           (thy-nm, Postab-strict.empty) (Postab-strict.update (pos, used-facts)))

     (* We want to keep our old theorem dependencies around, so we put them back
into
         the goal thm when we are done *)

      val post-thm' = post-thm
        |> Apply-Trace.join-deps pre-thm

    in post-thm' end)
    else post-thm)

(* The Proof hooks need to be patched in to track apply dependencies, but the rest
of levity
   can work without them. Here we graciously fail if the hook interface is missing
*)

fun setup-pre-apply-hook () =
 try (ML-Context.eval ML-Compiler.flags @{here})
 (ML-Lex.read-text (Proof.set-pre-apply-hook AutoLevity-Base.pre-apply-hook, @{here}));

fun setup-post-apply-hook () =
 try (ML-Context.eval ML-Compiler.flags @{here})
   (ML-Lex.read-text (Proof.set-post-apply-hook AutoLevity-Base.post-apply-hook,
@{here}));
```

```
(* This command is treated specially by AutoLevity-Theory-Report. The command
executed directly
   after this one will be tagged with the given tag *)

val - =
  Outer-Syntax.command @{command-keyword levity-tag} tag for levity
    (Parse.string >> (fn str =>
      Toplevel.local-theory NONE NONE
        (Local-Theory.raw-theory (set-levity-tag (SOME str)))))

fun get-subgoals' state =
let
  val proof-state = Toplevel.proof-of state;
  val {goal, ...} = Proof.raw-goal proof-state;
in Thm.nprems-of goal end

fun get-subgoals state = the-default ~1 (try get-subgoals' state);

fun setup-toplevel-command-hook () =
Toplevel.add-hook (fn transition => fn start-state => fn end-state =>
  let val name = Toplevel.name-of transition
      val pos = Toplevel.pos-of transition;
      val thy = Toplevel.theory-of start-state;
      val thynm = Context.theory-name thy;
      val end-thy = Toplevel.theory-of end-state;
 in
  if name = clear-deps orelse name = dummy-apply orelse Position.line-of pos =
NONE then () else
  (let

    val levity-input = if name = levity-tag then get-levity-tag end-thy else NONE;

    val subgoals = get-subgoals start-state;

    val entry = {levity-tag = levity-input, subgoals = subgoals}

    val - =
      Synchronized.change transactions
          (Symtab.map-default (thynm, (Postab-strict.empty, Postab-strict.empty))
              (apfst (Postab-strict.update (pos, (name, entry)))))
  in () end) handle e => if Exn.is-interrupt e then Exn.reraise e else
    Synchronized.change transactions
          (Symtab.map-default (thynm, (Postab-strict.empty, Postab-strict.empty))
                (apsnd (Postab-strict.map-default (pos, []) (cons (@{make-string}
e)))))
  end)

fun setup-attrib-tests theory = if not (is-simp-installed) then
error Missing interface into Raw-Simplifier. Can't trace apply statements with un-
```

```
patched isabelle.
else
let
  fun is-first-cong ctxt thm =
    let
      val simpset = Raw-Simplifier.internal-ss (Raw-Simplifier.simpset-of ctxt);
      val (congs, -) = #congs simpset;
      val cong-thm = #mk-cong (#mk-rews simpset) ctxt thm;
    in
      case (find-first (fn (-, thm') => Thm.eq-thm-prop (cong-thm, thm')) congs)
of
        SOME (nm, -) =>
          Thm.eq-thm-prop (find-first (fn (nm', -) => nm' = nm) congs |> the |>
snd, cong-thm)
      | NONE => false
    end

  fun is-classical proj ctxt thm =
    let
      val intros = proj (Classical.claset-of ctxt |> Classical.rep-cs);
      val results = Item-Net.retrieve intros (Thm.full-prop-of thm);
    in exists (fn (thm', -, -) => Thm.eq-thm-prop (thm',thm)) results end
in
 theory
|> add-attribute-test simp is-simp
|> add-attribute-test cong is-first-cong
|> add-attribute-test intro (is-classical #unsafeIs)
|> add-attribute-test intro! (is-classical #safeIs)
|> add-attribute-test elim (is-classical #unsafeEs)
|> add-attribute-test elim! (is-classical #safeEs)
|> add-attribute-test dest (fn ctxt => fn thm => is-classical #unsafeEs ctxt
(Tactic.make-elim thm))
|> add-attribute-test dest! (fn ctxt => fn thm => is-classical #safeEs ctxt (Tactic.make-elim
thm))
end


fun setup-command-hook {trace-apply, ...} theory =
if get-command-hook-flag theory then theory else
let
  val - = if trace-apply then
    (the (setup-pre-apply-hook ());
     the (setup-post-apply-hook ()))
        handle Option => error Missing interface into Proof module. Can't trace
apply statements with unpatched isabelle
    else ()

  val - = setup-toplevel-command-hook ();
```

```
  val theory' = theory
    |> trace-apply ? setup-attrib-tests
    |> set-command-hook-flag

in theory' end;


end
⟩
```

end

theory *AutoLevity-Theory-Report*
imports *AutoLevity-Base*
begin

```
ML ⟨
(* An antiquotation for creating json−like serializers for
   simple records. Serializers for primitive types are automatically used,
   while serializers for complex types are given as parameters. *)
val JSON-string-encode: string −> string =
  String.translate (
      fn #\\ => \\\\
       | #\n => \\n
       | x => if Char.isPrint x then String.str x else
                 \\u ^ align-right 0 4 (Int.fmt StringCvt.HEX (Char.ord x)))
  #> quote;

fun JSON-int-encode (i: int): string =
  if i < 0 then − ^ Int.toString (~i) else Int.toString i

val - = Theory.setup(
ML-Antiquotation.inline @{binding string-record}
  (Scan.lift
    (Parse.name −−|
      Parse.$$$ = −−
      Parse.position Parse.string) >>
    (fn (name,(source,pos)) =>

    let

      val entries =
      let
        val chars = String.explode source
          |> filter-out (fn #\n => true | - => false)

        val trim =
        String.explode
```

```
  #> chop-prefix (fn #  => true | - => false)
  #> snd
  #> chop-suffix (fn #  => true | - => false)
  #> fst
  #> String.implode

  val str = String.implode chars
    |> String.fields (fn #, => true | #: => true | - => false)
    |> map trim


  fun pairify [] = []
    | pairify (a::b::l) = ((a,b) :: pairify l)
    | pairify -  = error (Record syntax error ^ Position.here pos)

in
  pairify str
end

val typedecl =
type  ^ name ^ = {
^ (map (fn (nm,typ) => nm ^ : ^ typ) entries |> String.concatWith ,)
^ };

val base-typs = [string,int,bool, string list]


val encodes = map snd entries |> distinct (op =)
  |> filter-out (member (op =) base-typs)

val sanitize = String.explode
#> map (fn #  => #-
        | #. => #-
        | #* => #P
        | #( => #B
        | #) => #R
        | x => x)
#> String.implode

fun mk-encode typ =
if typ = string
then JSON-string-encode
else if typ = int
then JSON-int-encode
else if typ = bool
then Bool.toString
else if typ = string list
then (fn xs => (enclose \[\ \]\ (String.concatWith \, \ (map JSON-string-encode
xs))))
```

else (*sanitize typ*) ˆ -*encode*

fun mk-elem nm - value =
(*ML-Syntax.print-string* (*JSON-string-encode nm*) ˆ ˆ \ : \ ) ˆ ˆ (ˆ *value* ˆ)

fun mk-head body =
(\ ˆ {\ ˆ *String.concatWith* \, \ (ˆ *body* ˆ) ˆ \}\)

val global-head = *if* (*null encodes*) *then* *else*
fn (ˆ (*map mk-encode encodes* |> *String.concatWith* ,) ˆ) =>

val encode-body =
fn { ˆ (*map fst entries* |> *String.concatWith* ,) ˆ } : ˆ *name* ˆ => ˆ
mk-head
(*ML-Syntax.print-list* (*fn* (*field*,*typ*) => *mk-elem field typ* (*mk-encode typ* ˆ ˆ *field*)) *entries*)

val val-expr =
val (ˆ *name* ˆ -*encode*) = (
ˆ *global-head* ˆ (ˆ *encode-body* ˆ))

val - = @{*print*} *val-expr*

in
typedecl ˆ *val-expr*
end)))
⟩

ML ⟨

@{*string-record deps* = *consts* : *string list*, *types*: *string list*}
@{*string-record lemma-deps* = *consts*: *string list*, *types*: *string list*, *lemmas*: *string list*}
@{*string-record location* = *file* : *string*, *start-line* : *int*, *end-line* : *int*}
@{*string-record levity-tag* = *tag* : *string*, *location* : *location*}
@{*string-record apply-dep* = *name* : *string*, *attribs* : *string list*}

@{*string-record proof-command* =
command-name : *string*, *location* : *location*, *subgoals* : *int*, *depth* : *int*,
apply-deps : *apply-dep list* }

@{*string-record lemma-entry* =
name : *string*, *command-name* : *string*, *levity-tag* : *levity-tag option*, *location* : *location*,

```
  proof-commands : proof-command list,
  deps : lemma-deps}

@{string-record dep-entry =
  name : string, command-name : string, levity-tag : levity-tag option, location:
location,
  deps : deps}

@{string-record theory-entry =
  name : string, file : string}

@{string-record log-entry =
  errors : string list, location : location}

fun encode-list enc x = [ ^ (String.concatWith , (map enc x)) ^ ]

fun encode-option enc (SOME x) = enc x
  | encode-option - NONE = {}

val opt-levity-tag-encode = encode-option (levity-tag-encode location-encode);

val proof-command-encode = proof-command-encode (location-encode, encode-list
apply-dep-encode);

val lemma-entry-encode = lemma-entry-encode
 (opt-levity-tag-encode, location-encode, encode-list proof-command-encode, lemma-deps-encode)

val dep-entry-encode = dep-entry-encode
 (opt-levity-tag-encode, location-encode, deps-encode)

val log-entry-encode = log-entry-encode (location-encode)

›
```

ML ‹

```
signature AUTOLEVITY-THEORY-REPORT =
sig
val get-reports-for-thy: theory ->
  string * log-entry list * theory-entry list * lemma-entry list * dep-entry list *
dep-entry list

val string-reports-of:
  string * log-entry list * theory-entry list * lemma-entry list * dep-entry list *
dep-entry list
  -> string list

end;
```

```
structure AutoLevity-Theory-Report : AUTOLEVITY-THEORY-REPORT =
struct

fun map-pos-line f pos =
let
  val line = Position.line-of pos |> the;
  val file = Position.file-of pos |> the;

  val line' = f line;

  val - = if line' < 1 then raise Option else ();

in SOME (Position.line-file-only line' file) end handle Option => NONE


(* A Position.T table based on offsets (Postab-strict) can be collapsed into a line−based
one
   with lists of entries on for each line. This function searches such a table
   for the closest entry, either backwards (LESS) or forwards (GREATER) from
   the given position. *)

(* TODO: If everything is sane then the search depth shouldn't be necessary. In
practice
    entries won't be more than one or two lines apart, but if something has gone
wrong in the
   collection phase we might end up wasting a lot of time looking for an entry that
doesn't exist. *)

fun search-by-lines depth ord-kind f h pos = if depth = 0 then NONE else
  let
     val line-change = case ord-kind of LESS => ~1 | GREATER => 1 | - =>
raise Fail Bad relation
    val idx-change = case ord-kind of GREATER => 1 | - => 0;
  in
  case f pos of
   SOME x =>
    let
      val i = find-index (fn e => h (pos, e) = ord-kind) x;
    in if i > ~1 then SOME (List.nth(x, i + idx-change)) else SOME (hd x) end

  | NONE =>
    (case (map-pos-line (fn i => i + line-change) pos) of
      SOME pos' => search-by-lines (depth − 1) ord-kind f h pos'
     | NONE => NONE)
   end

fun location-from-range (start-pos, end-pos) =
  let
    val start-file = Position.file-of start-pos |> the;
```

```
    val end-file = Position.file-of end-pos |> the;
    val - = if start-file = end-file then () else raise Option;
    val start-line = Position.line-of start-pos |> the;
    val end-line = Position.line-of end-pos |> the;
  in
  SOME ({file = start-file, start-line = start-line, end-line = end-line} : location)
end
  handle Option => NONE

(* Here we collapse our proofs (lemma foo .. done) into single entries with start/end
positions. *)

fun get-command-ranges-of keywords thy-nm =
let
  fun is-ignored nm' = nm' = <ignored>
  fun is-levity-tag nm' = nm' = levity-tag

  fun is-proof-cmd nm' = nm' = apply orelse nm' = by orelse nm' = proof

  (* All top−level transactions for the given theory *)

  val (transactions, log) =
          Symtab.lookup (AutoLevity-Base.get-transactions ()) thy-nm
          |> the-default (Postab-strict.empty, Postab-strict.empty)
          ||> Postab-strict.dest
          |>> Postab-strict.dest

  (* Line−based position table of all apply statements for the given theory *)

  val applytab =
    Symtab.lookup (AutoLevity-Base.get-applys ()) thy-nm
    |> the-default Postab-strict.empty
    |> Postab-strict.dest
    |> map (fn (pos,e) => (pos, (pos,e)))
    |> Postab.make-list
   |> Postab.map (fn - => sort (fn ((pos,-),(pos', -)) => pos-ord true (pos, pos')))


  (* A special ignored command lets us find the real end of commands which span
     multiple lines. After finding a real command, we assume the last ignored one
     was part of the syntax for that command *)

  fun find-cmd-end last-pos ((pos', (nm', ext)) :: rest) =
    if is-ignored nm' then
       find-cmd-end pos' rest
    else (last-pos, ((pos', (nm', ext)) :: rest))
    | find-cmd-end last-pos [] = (last-pos, [])

  fun change-level nm level =
```

```
    if Keyword.is-proof-open keywords nm then level + 1
    else if Keyword.is-proof-close keywords nm then level − 1
    else if Keyword.is-qed-global keywords nm then ~1
    else level


  fun make-apply-deps lemma-deps =
    map (fn (nm, atts) => {name = nm, attribs = atts} : apply-dep) lemma-deps

  (* For a given apply statement, search forward in the document for the closest
method to retrieve
     its lemma dependencies *)

  fun find-apply pos = if Postab.is-empty applytab then [] else
    search-by-lines 5 GREATER (Postab.lookup applytab) (fn (pos, (pos', -)) =>
pos-ord true (pos, pos')) pos
   |> Option.map snd |> the-default [] |> make-apply-deps

  fun find-proof-end level ((pos', (nm', ext)) :: rest) =
    let val level' = change-level nm' level in
     if level' > ~1 then
       let
         val (cmd-end, rest') = find-cmd-end pos' rest;
         val ((prf-cmds, prf-end), rest'') = find-proof-end level' rest'
       in (({command-name = nm', location = location-from-range (pos', cmd-end)
|> the,
            depth = level, apply-deps = if is-proof-cmd nm' then find-apply pos' else
[],
            subgoals = #subgoals ext} :: prf-cmds, prf-end), rest'') end
     else
       let
         val (cmd-end, rest') = find-cmd-end pos' rest;
       in (([{command-name = nm', location = location-from-range (pos', cmd-end)
|> the,
            apply-deps = if is-proof-cmd nm' then find-apply pos' else [],
            depth = level, subgoals = #subgoals ext}], cmd-end), rest') end
     end
     | find-proof-end - - = (([], Position.none), [])


  fun find-ends tab tag ((pos,(nm, ext)) :: rest) =
   let
     val (cmd-end, rest') = find-cmd-end pos rest;

     val ((prf-cmds, pos'), rest'') =
       if Keyword.is-theory-goal keywords nm
       then find-proof-end 0 rest'
       else (([],cmd-end),rest');
```

```
    val tab' = Postab.cons-list (pos, (pos, (nm, pos', tag, prf-cmds))) tab;

    val tag' =
      if is-levity-tag nm then Option.map (rpair (pos,pos')) (#levity-tag ext) else
NONE;

  in find-ends tab' tag' rest'' end
    | find-ends tab - [] = tab

  val command-ranges = find-ends Postab.empty NONE transactions
  |> Postab.map (fn - => sort (fn ((pos,-),(pos',-)) => pos-ord true (pos, pos')))

in (command-ranges, log) end



fun make-deps (const-deps, type-deps): deps =
  {consts = distinct (op =) const-deps, types = distinct (op =) type-deps}

fun make-lemma-deps (const-deps, type-deps, lemma-deps): lemma-deps =
  {
    consts = distinct (op =) const-deps,
    types = distinct (op =) type-deps,
    lemmas = distinct (op =) lemma-deps
  }

fun make-tag (SOME (tag, range)) = (case location-from-range range
  of SOME rng => SOME ({tag = tag, location = rng} : levity-tag)
  | NONE => NONE)
  | make-tag NONE = NONE



fun add-deps (((Defs.Const, nm), -) :: rest) =
  let val (consts, types) = add-deps rest in
    (nm :: consts, types) end
  | add-deps (((Defs.Type, nm), -) :: rest) =
  let val (consts, types) = add-deps rest in
    (consts, nm :: types) end
  | add-deps - = ([], [])

fun get-deps ({rhs, ...} : Defs.spec) = add-deps rhs

fun typs-of-typ (Type (nm, Ts)) = nm :: (map typs-of-typ Ts |> flat)
  | typs-of-typ - = []

fun typs-of-term t = Term.fold-types (append o typs-of-typ) t []

fun deps-of-thm thm =
```

```
let
  val consts = Term.add-const-names (Thm.prop-of thm) [];
  val types = typs-of-term (Thm.prop-of thm);
in (consts, types) end

fun file-of-thy thy =
  let
    val path = Resources.master-directory thy;
    val name = Context.theory-name thy;
    val path' = Path.append path (Path.basic (name ^ .thy))
  in Path.smart-implode path' end;

fun entry-of-thy thy = ({name = Context.theory-name thy, file = file-of-thy thy}
: theory-entry)

fun used-facts thy thm =
  AutoLevity-Base.used-named-props-of thm
    |> map-filter (AutoLevity-Base.disambiguate-indices (Proof-Context.init-global
thy))
    |> List.map fst;

fun get-reports-for-thy thy =
  let
    val thy-nm = Context.theory-name thy;
    val all-facts = Global-Theory.facts-of thy;
    val fact-space = Facts.space-of all-facts;

   val (tab, log) = get-command-ranges-of (Thy-Header.get-keywords thy) thy-nm;

    val parent-facts = map Global-Theory.facts-of (Theory.parents-of thy);

    val search-backwards = search-by-lines 5 LESS (Postab.lookup tab)
      (fn (pos, (pos', -)) => pos-ord true (pos, pos'))
      #> the

    val lemmas =  Facts.dest-static false parent-facts (Global-Theory.facts-of thy)
    |> map-filter (fn (xnm, thms) =>
       let
          val {pos, theory-name, ...} = Name-Space.the-entry fact-space xnm;
          in
            if theory-name = thy-nm then
            let
             val thms' = map (Thm.transfer thy) thms;

            val (real-start, (cmd-name, end-pos, tag, prf-cmds)) = search-backwards
pos

             val lemma-deps =
                  if cmd-name = datatype
```

```
              then []
              else map (used-facts thy) thms' |> flat |> distinct (op =);

          val (consts, types) = map deps-of-thm thms' |> ListPair.unzip |> apply2
flat
           val deps = make-lemma-deps (consts, types, lemma-deps)

           val location = location-from-range (real-start, end-pos) |> the;

           val (lemma-entry : lemma-entry) =
             {name  = xnm, command-name = cmd-name, levity-tag = make-tag
tag,
             location = location, proof-commands = prf-cmds, deps = deps}

          in SOME (pos, lemma-entry) end
          else NONE end handle Option => NONE)
     |> Postab-strict.make-list
     |> Postab-strict.dest |> map snd |> flat

   val defs = Theory.defs-of thy;

   fun get-deps-of kind space xnms = xnms
   |> map-filter (fn xnm =>
     let
         val {pos, theory-name, ...} = Name-Space.the-entry space xnm;
         in
           if theory-name = thy-nm then
           let
             val specs = Defs.specifications-of defs (kind, xnm);

             val deps =
               map get-deps specs
              |> ListPair.unzip
              |> (apply2 flat #> make-deps);

             val (real-start, (cmd-name, end-pos, tag, -)) = search-backwards pos

             val loc = location-from-range (real-start, end-pos) |> the;

             val entry =
               ({name = xnm, command-name = cmd-name, levity-tag = make-tag
tag,
                 location = loc, deps = deps} : dep-entry)

           in SOME (pos, entry) end
           else NONE end handle Option => NONE)
     |> Postab-strict.make-list
     |> Postab-strict.dest |> map snd |> flat
```

```
  val {const-space, constants, ...} = Consts.dest (Sign.consts-of thy);

  val consts = get-deps-of Defs.Const const-space (map fst constants);

  val {types, ...} = Type.rep-tsig (Sign.tsig-of thy);

  val type-space = Name-Space.space-of-table types;
  val type-names = Name-Space.fold-table (fn (xnm, -) => cons xnm) types [];

  val types = get-deps-of Defs.Type type-space type-names;

  val thy-parents = map entry-of-thy (Theory.parents-of thy);

  val logs = log |>
   map (fn (pos, errs) => {errors = errs, location = location-from-range (pos,
pos) |> the} : log-entry)

 in (thy-nm, logs, thy-parents, lemmas, consts, types) end

fun add-commas (s :: s' :: ss) = s ^ , :: (add-commas (s' :: ss))
 | add-commas [s] = [s]
 | add-commas - = []


fun string-reports-of (thy-nm, logs, thy-parents, lemmas, consts, types) =
    [{\theory-name\ :  ^ JSON-string-encode thy-nm ^ ,] @
    [\logs\ : [] @
    add-commas (map (log-entry-encode) logs) @
    [],,\theory-imports\ : [] @
    add-commas (map (theory-entry-encode) thy-parents) @
    [],,\lemmas\ : [] @
    add-commas (map (lemma-entry-encode) lemmas) @
    [],,\consts\ : [] @
    add-commas (map (dep-entry-encode) consts) @
    [],,\types\ : [] @
    add-commas (map (dep-entry-encode) types) @
    []}]
    |> map (fn s => s ^ \n)

end
›
```

end

theory *AutoLevity-Hooks*
imports
 AutoLevity-Base
 AutoLevity-Theory-Report

begin

end

theory *Locale-Abbrev*
  imports *Main*
  keywords *revert-abbrev* :: *thy-decl* **and** *locale-abbrev* :: *thy-decl*
begin

ML ‹
```
local

fun revert-abbrev (mode,name) lthy =
  let
    val the-const = (fst o dest-Const) oo Proof-Context.read-const {proper = true,
strict = false};
  in
    Local-Theory.raw-theory (Sign.revert-abbrev (fst mode) (the-const lthy name))
lthy
  end

fun name-of spec lthy = Local-Defs.abs-def (Syntax.read-term lthy spec) |> #1 |>
#1

in

val - =
  Outer-Syntax.local-theory @{command-keyword revert-abbrev}
    make an abbreviation available for output
    (Parse.syntax-mode -- Parse.const >> revert-abbrev)

val - =
  Outer-Syntax.local-theory' @{command-keyword locale-abbrev}
    constant abbreviation that provides also provides printing in locales
    (Parse.syntax-mode -- Scan.option Parse-Spec.constdecl -- Parse.prop --
Parse.for-fixes
      >> (fn (((mode, decl), spec), params) => fn restricted => fn lthy =>
          lthy
          |> Local-Theory.open-target |> snd
          |> Specification.abbreviation-cmd mode decl params spec restricted
          |> Local-Theory.close-target (* commit new abbrev. name *)
          |> revert-abbrev (mode, name-of spec lthy)));

end
```
›

end

theory *NICTATools*
imports
  Apply-Trace-Cmd
  Apply-Debug
  Find-Names

  Rule-By-Method
  Eisbach-Methods
  TSubst
  Time-Methods-Cmd
  Try-Attribute
  Trace-Schematic-Insts
  Insulin
  ShowTypes
  AutoLevity-Hooks
  Locale-Abbrev
begin

15 Detect unused meta-forall

ML ‹

```
(* Return a list of meta−forall variable names that appear
 * to be unused in the input term. *)
fun find-unused-metaall (Const (@{const-name Pure.all}, -) $ Abs (n, -, t)) =
      (if not (Term.is-dependent t) then [n] else []) @ find-unused-metaall t
  | find-unused-metaall (Abs (-, -, t)) =
      find-unused-metaall t
  | find-unused-metaall (a $ b) =
      find-unused-metaall a @ find-unused-metaall b
  | find-unused-metaall - = []

(* Given a proof state, analyse its assumptions for unused
 * meta−foralls. *)
fun detect-unused-meta-forall - (state : Proof.state) =
let
  (* Fetch all assumptions and the main goal, and analyse them. *)
  val {context = lthy, goal = goal, ...} = Proof.goal state
  val checked-terms =
      [Thm.concl-of goal] @ map Thm.term-of (Assumption.all-assms-of lthy)
  val results = List.concat (map find-unused-metaall checked-terms)

  (* Produce a message. *)
  fun message results =
    Pretty.paragraph [
```

```
      Pretty.str Unused meta−forall(s): ,
      Pretty.commas
        (map (fn b => Pretty.mark-str (Markup.bound, b)) results)
      |> Pretty.paragraph,
      Pretty.str .
    ]

  (* We use a warning instead of the standard mechanisms so that
   * we can produce a warning icon in Isabelle/jEdit. *)
  val - =
    if length results > 0 then
      warning (message results |> Pretty.string-of)
    else ()
in
  (false, (, []))
end

(* Setup the tool, stealing the auto-solve-direct option. *)
val - = Try.tool-setup (unused-meta-forall,
    (1, @{system-option auto-solve-direct}, detect-unused-meta-forall))
›
```

lemma *test-unused-meta-forall*: $\bigwedge x.\ y \vee \neg\ y$
  oops

end
Library **theory** *Lib*
imports
  Value-Abbreviation
  Match-Abbreviation
  Try-Methods
  Extract-Conjunct
  Eval-Bool
  NICTATools
  HOL−Library.Prefix-Order
  HOL−Word.Word
begin

abbreviation (*input*)
  split :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c$
where
  split == *case-prod*

lemma *hd-map-simp*:
  $b \neq [] \Longrightarrow hd\ (map\ a\ b) = a\ (hd\ b)$
  by (*rule hd-map*)

lemma *tl-map-simp*:
  $tl\ (map\ a\ b) = map\ a\ (tl\ b)$
  by (*induct b*,*auto*)

lemma *Collect-eq*:
  $\{x.\ P\ x\} = \{x.\ Q\ x\} \longleftrightarrow (\forall\, x.\ P\ x = Q\ x)$
  by (*rule iffI*) *auto*

lemma *iff-impI*: $[\![P \Longrightarrow Q = R]\!] \Longrightarrow (P \longrightarrow Q) = (P \longrightarrow R)$ **by** *blast*

definition
  fun-app :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**infixr** $ *10*) **where**
  $f\ \$\ x \equiv f\ x$

declare *fun-app-def* [*iff*]

lemma *fun-app-cong*[*fundef-cong*]:
  $[\![\ f\ x = f'\ x'\ ]\!] \Longrightarrow (f\ \$\ x) = (f'\ \$\ x')$
  by *simp*

lemma *fun-app-apply-cong*[*fundef-cong*]:
  $f\ x\ y = f'\ x'\ y' \Longrightarrow (f\ \$\ x)\ y = (f'\ \$\ x')\ y'$
  by *simp*

lemma *if-apply-cong*[*fundef-cong*]:
  $[\![\ P = P';\ x = x';\ P' \Longrightarrow f\ x' = f'\ x';\ \neg\ P' \Longrightarrow g\ x' = g'\ x'\ ]\!]$
    $\Longrightarrow (\mathit{if}\ P\ \mathit{then}\ f\ \mathit{else}\ g)\ x = (\mathit{if}\ P'\ \mathit{then}\ f'\ \mathit{else}\ g')\ x'$
  by *simp*

lemma *case-prod-apply-cong*[*fundef-cong*]:
  $[\![\ f\ (fst\ p)\ (snd\ p)\ s = f'\ (fst\ p')\ (snd\ p')\ s'\ ]\!] \Longrightarrow case\text{-}prod\ f\ p\ s = case\text{-}prod\ f'$
$p'\ s'$
  by (*simp add*: *split-def*)

lemma *prod-injects*:
  $(x,y) = p \Longrightarrow x = fst\ p \wedge y = snd\ p$
  $p = (x,y) \Longrightarrow x = fst\ p \wedge y = snd\ p$
  by *auto*

definition
  pred-conj :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool)$ (**infixl** *and 35*)
where
  $pred\text{-}conj\ P\ Q \equiv \lambda x.\ P\ x \wedge Q\ x$

definition

pred-disj :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool)$ (**infixl** *or 30*)
where
pred-disj $P\ Q \equiv \lambda x.\ P\ x \lor Q\ x$

definition
pred-neg :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool)$ (*not* - [*40*] *40*)
where
pred-neg $P \equiv \lambda x.\ \neg\ P\ x$

definition $K \equiv \lambda x\ y.\ x$

definition
zipWith :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow 'c\ list$ **where**
zipWith f xs ys $\equiv$ *map* (*case-prod f*) (*zip xs ys*)

primrec
delete :: $'a \Rightarrow 'a\ list \Rightarrow 'a\ list$
where
delete y [] = []
| *delete y* (*x#xs*) = (*if y=x then xs else x # delete y xs*)

definition
swp f $\equiv \lambda x\ y.\ f\ y\ x$

lemma *swp-apply*[*simp*]: *swp f y x* = *f x y*
by (*simp add*: *swp-def*)

primrec (*nonexhaustive*)
theRight :: $'a + 'b \Rightarrow 'b$ **where**
theRight (*Inr x*) = *x*

primrec (*nonexhaustive*)
theLeft :: $'a + 'b \Rightarrow 'a$ **where**
theLeft (*Inl x*) = *x*

definition
isLeft x $\equiv (\exists\, y.\ x = Inl\ y)$

definition
isRight x $\equiv (\exists\, y.\ x = Inr\ y)$

definition
const x $\equiv \lambda y.\ x$

primrec
opt-rel :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ option \Rightarrow 'b\ option \Rightarrow bool$
where
opt-rel f None $y = (y = None)$
| *opt-rel f* (*Some x*) $y = (\exists\, y'.\ y = Some\ y' \land f\ x\ y')$

lemma *opt-rel-None-rhs*[*simp*]:
  opt-rel f x None = (*x* = *None*)
  by (*cases x*, *simp-all*)

lemma *opt-rel-Some-rhs*[*simp*]:
  opt-rel f x (*Some y*) = ($\exists x'$. *x* = *Some x'* $\wedge$ *f x' y*)
  by (*cases x*, *simp-all*)

lemma *tranclD2*:
  $(x, y) \in R^+ \Longrightarrow \exists z. (x, z) \in R^* \wedge (z, y) \in R$
  by (*erule tranclE*) *auto*

lemma *linorder-min-same1* [*simp*]:
  (*min y x* = *y*) = ($y \leq$ (*x*::'*a*::*linorder*))
  by (*auto simp*: *min-def linorder-not-less*)

lemma *linorder-min-same2* [*simp*]:
  (*min x y* = *y*) = ($y \leq$ (*x*::'*a*::*linorder*))
  by (*auto simp*: *min-def linorder-not-le*)

A combinator for pairing up well-formed relations. The divisor function splits the population in halves, with the True half greater than the False half, and the supplied relations control the order within the halves.

definition
  wf-sum :: ('*a* $\Rightarrow$ *bool*) $\Rightarrow$ ('*a* $\times$ '*a*) *set* $\Rightarrow$ ('*a* $\times$ '*a*) *set* $\Rightarrow$ ('*a* $\times$ '*a*) *set*
where
  wf-sum divisor r r' $\equiv$
    ($\{(x, y). \neg$ *divisor x* $\wedge \neg$ *divisor y*$\} \cap r'$)
  $\cup$ $\{(x, y). \neg$ *divisor x* $\wedge$ *divisor y*$\}$
  $\cup$ ($\{(x, y).$ *divisor x* $\wedge$ *divisor y*$\} \cap r$)

lemma *wf-sum-wf*:
  $[\![$ *wf r*; *wf r'* $]\!] \Longrightarrow$ *wf* (*wf-sum divisor r r'*)
  apply (*simp add*: *wf-sum-def*)
  apply (*rule wf-Un*)+
      apply (*erule wf-Int2*)
     apply (*rule wf-subset*
             [**where** *r*=*measure* (λx. *If* (*divisor x*) *1 0*)])
      apply *simp*
     apply *clarsimp*
    apply *blast*
   apply (*erule wf-Int2*)
  apply *blast*
  done

abbreviation(*input*)
 option-map == *map-option*

lemmas *option-map-def* = *map-option-case*

lemma *False-implies-equals* [*simp*]:
  $((False \Longrightarrow P) \Longrightarrow PROP\ Q) \equiv PROP\ Q$
  apply (*rule equal-intr-rule*)
   apply (*erule meta-mp*)
   apply *simp*
  apply *simp*
  done

lemma *split-paired-Ball*:
  $(\forall x \in A.\ P\ x) = (\forall x\ y.\ (x,y) \in A \longrightarrow P\ (x,y))$
  by *auto*

lemma *split-paired-Bex*:
  $(\exists x \in A.\ P\ x) = (\exists x\ y.\ (x,y) \in A \wedge P\ (x,y))$
  by *auto*

lemma *delete-remove1*:
  delete x xs = *remove1 x xs*
  by (*induct xs*, *auto*)

lemma *ignore-if*:
  $(y\ and\ z)\ s \Longrightarrow (if\ x\ then\ y\ else\ z)\ s$
  by (*clarsimp simp*: *pred-conj-def*)

lemma *zipWith-Nil2* :
  zipWith f xs [] = []
  unfolding *zipWith-def* **by** *simp*

lemma *isRight-right-map*:
  isRight (*case-sum Inl* (*Inr o f*) *v*) = *isRight v*
  by (*simp add*: *isRight-def split*: *sum.split*)

lemma *zipWith-nth*:
  $[\![\ n < min\ (length\ xs)\ (length\ ys)\ ]\!] \Longrightarrow zipWith\ f\ xs\ ys\ !\ n = f\ (xs\ !\ n)\ (ys\ !\ n)$
  unfolding *zipWith-def* **by** *simp*

lemma *length-zipWith* [*simp*]:
  length (*zipWith f xs ys*) = *min* (*length xs*) (*length ys*)
  unfolding *zipWith-def* **by** *simp*

lemma *first-in-uptoD*:
  $a \leq b \Longrightarrow (a::'a::order) \in \{a..b\}$
  by *simp*

lemma *construct-singleton*:

$\llbracket\ S \neq \{\};\ \forall\, s{\in}S.\ \forall\, s'.\ s \neq s' \longrightarrow s' \notin S\ \rrbracket \Longrightarrow \exists\, x.\ S = \{x\}$
by *blast*

lemmas *insort-com* = *insort-left-comm*

lemma *bleeding-obvious*:
$(P \Longrightarrow \mathit{True}) \equiv (\mathit{Trueprop\ True})$
by (*rule*, *simp-all*)

lemma *Some-helper*:
$x = \mathit{Some}\ y \Longrightarrow x \neq \mathit{None}$
by *simp*

lemma *in-empty-interE*:
$\llbracket\ A \cap B = \{\};\ x \in A;\ x \in B\ \rrbracket \Longrightarrow \mathit{False}$
by *blast*

lemma *None-upd-eq*:
$g\ x = \mathit{None} \Longrightarrow g(x := \mathit{None}) = g$
by (*rule ext*) *simp*

lemma *exx* [*iff*]: $\exists\, x.\ x$ **by** *blast*
lemma *ExNot* [*iff*]: *Ex Not* **by** *blast*

lemma *cases-simp2* [*simp*]:
$((\neg\ P \longrightarrow Q) \wedge (P \longrightarrow Q)) = Q$
by *blast*

lemma *a-imp-b-imp-b*:
$((a \longrightarrow b) \longrightarrow b) = (a \vee b)$
by *blast*

lemma *length-neq*:
$\mathit{length}\ \mathit{as} \neq \mathit{length}\ \mathit{bs} \Longrightarrow \mathit{as} \neq \mathit{bs}$ **by** *auto*

lemma *take-neq-length*:
$\llbracket\ x \neq y;\ x \leq \mathit{length}\ \mathit{as};\ y \leq \mathit{length}\ \mathit{bs}\rrbracket \Longrightarrow \mathit{take}\ x\ \mathit{as} \neq \mathit{take}\ y\ \mathit{bs}$
by (*rule length-neq*, *simp*)

lemma *eq-concat-lenD*:
$\mathit{xs} = \mathit{ys}\ @\ \mathit{zs} \Longrightarrow \mathit{length}\ \mathit{xs} = \mathit{length}\ \mathit{ys} + \mathit{length}\ \mathit{zs}$
by *simp*

lemma *map-upt-reindex'*: $\mathit{map}\ f\ [a\ ..{<}\ b] = \mathit{map}\ (\lambda n.\ f\ (n + a - x))\ [x\ ..{<}\ x + b - a]$
by (*rule nth-equalityI*; *clarsimp simp*: *add.commute*)

lemma *map-upt-reindex*: $\mathit{map}\ f\ [a\ ..{<}\ b] = \mathit{map}\ (\lambda n.\ f\ (n + a))\ [0\ ..{<}\ b - a]$
by (*subst map-upt-reindex'* [**where** $x{=}0$]) *clarsimp*

lemma *notemptyI*:
  $x \in S \Longrightarrow S \neq \{\}$
  by *clarsimp*

lemma *setcomp-Max-has-prop*:
  assumes *a*: $P\ x$
  shows $P\ (Max\ \{(x::'a::\{finite,linorder\}).\ P\ x\})$
proof −
  from *a* **have** $Max\ \{x.\ P\ x\} \in \{x.\ P\ x\}$
    by − (*rule Max-in*, *auto intro*: *notemptyI*)
  thus *?thesis* **by** *auto*
qed

lemma *cons-set-intro*:
  $lst = x\ \#\ xs \Longrightarrow x \in set\ lst$
  by *fastforce*

lemma *list-all2-conj-nth*:
  assumes *lall*: *list-all2 P as cs*
  and *rl*: $\bigwedge n.\ [\![P\ (as\ !\ n)\ (cs\ !\ n);\ n < length\ as]\!] \Longrightarrow Q\ (as\ !\ n)\ (cs\ !\ n)$
  shows *list-all2* $(\lambda a\ b.\ P\ a\ b \wedge Q\ a\ b)$ *as cs*
proof (*rule list-all2-all-nthI*)
  from *lall* **show** $length\ as = length\ cs$ **..**
next
  fix *n*
  assume $n < length\ as$

  show $P\ (as\ !\ n)\ (cs\ !\ n) \wedge Q\ (as\ !\ n)\ (cs\ !\ n)$
  proof
    from *lall* **show** $P\ (as\ !\ n)\ (cs\ !\ n)$ **by** (*rule list-all2-nthD*) *fact*
    thus $Q\ (as\ !\ n)\ (cs\ !\ n)$ **by** (*rule rl*) *fact*
  qed
qed

lemma *list-all2-conj*:
  assumes *lall1*: *list-all2 P as cs*
  and *lall2*: *list-all2 Q as cs*
  shows *list-all2* $(\lambda a\ b.\ P\ a\ b \wedge Q\ a\ b)$ *as cs*
proof (*rule list-all2-all-nthI*)
  from *lall1* **show** $length\ as = length\ cs$ **..**
next
  fix *n*
  assume $n < length\ as$

  show $P\ (as\ !\ n)\ (cs\ !\ n) \wedge Q\ (as\ !\ n)\ (cs\ !\ n)$
  proof
    from *lall1* **show** $P\ (as\ !\ n)\ (cs\ !\ n)$ **by** (*rule list-all2-nthD*) *fact*
    from *lall2* **show** $Q\ (as\ !\ n)\ (cs\ !\ n)$ **by** (*rule list-all2-nthD*) *fact*

qed
qed

lemma *all-set-into-list-all2*:
assumes *lall*: $\forall x \in set\ ls.\ P\ x$
and $length\ ls = length\ ls'$
shows *list-all2* $(\lambda a\ b.\ P\ a)\ ls\ ls'$
proof (*rule list-all2-all-nthI*)
fix n
assume $n < length\ ls$
from *lall* **show** $P\ (ls\ !\ n)$
by (*rule bspec* [*OF - nth-mem*]) *fact*
qed *fact*

lemma *GREATEST-lessE*:
fixes $x :: 'a :: order$
assumes *gts*: $(GREATEST\ x.\ P\ x) < X$
and *px*: $P\ x$
and *gtst*: $\exists\, max.\ P\ max \wedge (\forall\, z.\ P\ z \longrightarrow (z \leq max))$
shows $x < X$
proof −
from *gtst* **obtain** *max* **where** *pm*: $P\ max$ **and** g': $\bigwedge z.\ P\ z \Longrightarrow z \leq max$
by *auto*

hence $(GREATEST\ x.\ P\ x) = max$
by (*auto intro*: *Greatest-equality*)

moreover have $x \leq max$ **using** *px* **by** (*rule* g')

ultimately show *?thesis* **using** *gts* **by** *simp*
qed

lemma *set-has-max*:
fixes $ls :: ('a :: linorder)\ list$
assumes *ls*: $ls \neq []$
shows $\exists\, max \in set\ ls.\ \forall\, z \in set\ ls.\ z \leq max$
using *ls*
proof (*induct ls*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons l ls*)

show *?case*
proof (*cases* $ls = []$)
case *True*
thus *?thesis* **by** *simp*
next
case *False*
then obtain *max* **where** *mv*: $max \in set\ ls$ **and** *mm*: $\forall\, z \in set\ ls.\ z \leq max$

using *Cons.hyps*
  by *auto*
 show *?thesis*
 proof (*cases max* $\leq$ *l*)
  case *True*
  have $l \in$ *set* (*l* # *ls*) **by** *simp*
  thus *?thesis*
  proof
   from *mm* **show** $\forall z \in$ *set* (*l* # *ls*). $z \leq l$ **using** *True* **by** *auto*
  qed
 next
  case *False*
  from *mv* **have** *max* $\in$ *set* (*l* # *ls*) **by** *simp*
  thus *?thesis*
  proof
   from *mm* **show** $\forall z \in$ *set* (*l* # *ls*). $z \leq$ *max* **using** *False* **by** *auto*
  qed
 qed
qed
qed

lemma *True-notin-set-replicate-conv*:
 True $\notin$ *set ls* = (*ls* = *replicate* (*length ls*) *False*)
 by (*induct ls*) *simp+*

lemma *Collect-singleton-eqI*:
 $(\bigwedge x.\ P\ x = (x = v)) \Longrightarrow \{x.\ P\ x\} = \{v\}$
 by *auto*

lemma *exEI*:
 $[\![\ \exists y.\ P\ y;\ \bigwedge x.\ P\ x \Longrightarrow Q\ x\ ]\!] \Longrightarrow \exists z.\ Q\ z$
 by (*rule ex-forward*)

lemma *allEI*:
 assumes $\forall x.\ P\ x$
 assumes $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
 shows $\forall x.\ Q\ x$
 using *assms* **by** (*rule all-forward*)

General lemmas that should be in the library

lemma *dom-ran*:
 $x \in$ *dom f* $\Longrightarrow$ *the* (*f x*) $\in$ *ran f*
 by (*simp add*: *dom-def ran-def*, *erule exE*, *simp*, *rule exI*, *simp*)

lemma *orthD1*:
 $[\![\ S \cap S' = \{\};\ x \in S]\!] \Longrightarrow x \notin S'$ **by** *auto*

lemma *orthD2*:
 $[\![\ S \cap S' = \{\};\ x \in S']\!] \Longrightarrow x \notin S$ **by** *auto*

lemma *distinct-element*:
$[\![\ b \cap d = \{\}; \ a \in b; \ c \in d]\!] \Longrightarrow a \neq c$
by *auto*

lemma *ball-reorder*:
$(\forall x \in A. \forall y \in B. \ P \ x \ y) = (\forall y \in B. \forall x \in A. \ \ P \ x \ y)$
by *auto*

lemma *hd-map*: $ls \neq [] \Longrightarrow hd \ (map \ f \ ls) = f \ (hd \ ls)$
by $(cases \ ls)$ *auto*

lemma *tl-map*: $tl \ (map \ f \ ls) = map \ f \ (tl \ ls)$
by $(cases \ ls)$ *auto*

lemma *not-NilE*:
$[\![\ xs \neq []; \ \bigwedge x \ xs'. \ xs = x \ \# \ xs' \Longrightarrow R \ ]\!] \Longrightarrow R$
by $(cases \ xs)$ *auto*

lemma *length-SucE*:
$[\![\ length \ xs = Suc \ n; \ \bigwedge x \ xs'. \ xs = x \ \# \ xs' \Longrightarrow R \ ]\!] \Longrightarrow R$
by $(cases \ xs)$ *auto*

lemma *map-upt-unfold*:
assumes *ab*: $a < b$
shows $map \ f \ [a \ ..< \ b] = f \ a \ \# \ map \ f \ [Suc \ a \ ..< \ b]$
using *assms upt-conv-Cons* **by** *auto*

lemma *tl-nat-list-simp*:
$tl \ [a..<b] = [a + 1 \ ..<b]$
by $(induct \ b, \ auto)$

lemma *image-Collect2*:
$case\text{-}prod \ f \ ` \ \{x. \ P \ (fst \ x) \ (snd \ x)\} = \{f \ x \ y \ |x \ y. \ P \ x \ y\}$
by $(subst \ image\text{-}Collect)$ *simp*

lemma *image-id'*:
$id \ ` \ Y = Y$
by *clarsimp*

lemma *image-invert*:
assumes r: $f \circ g = id$
and g: $B = g \ ` \ A$
shows $A = f \ ` \ B$
by $(simp \ add\colon g \ image\text{-}comp \ r)$

lemma *Collect-image-fun-cong*:
assumes *rl*: $\bigwedge a. \ P \ a \Longrightarrow f \ a = g \ a$
shows $\{f \ x \ | \ x. \ P \ x\} = \{g \ x \ | \ x. \ P \ x\}$

using *rl* **by** *force*

lemma *inj-on-take*:
  shows *inj-on* (*take* *n*) {*x*. *drop* *n* *x* = *k*}
proof (*rule* *inj-onI*)
  fix *x* *y*
  assume *xv*: *x* ∈ {*x*. *drop* *n* *x* = *k*}
    and *yv*: *y* ∈ {*x*. *drop* *n* *x* = *k*}
    and *tk*: *take* *n* *x* = *take* *n* *y*

  from *xv* **have** *take* *n* *x* @ *k* = *x*
    using *append-take-drop-id* *mem-Collect-eq* **by** *auto*
  moreover from *yv* *tk*
  have *take* *n* *x* @ *k* = *y*
    using *append-take-drop-id* *mem-Collect-eq* **by** *auto*
  ultimately show *x* = *y* **by** *simp*
qed

lemma *foldr-upd-dom*:
  dom (*foldr* (λ*p* *ps*. *ps* (*p* ↦ *f* *p*)) *as* *g*) = *dom* *g* ∪ *set* *as*
proof (*induct* *as*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons* *a* *as*)
  show *?case*
  proof (*cases* *a* ∈ *set* *as* ∨ *a* ∈ *dom* *g*)
    case *True*
    hence *ain*: *a* ∈ *dom* *g* ∪ *set* *as* **by** *auto*
    hence *dom* *g* ∪ *set* (*a* # *as*) = *dom* *g* ∪ *set* *as* **by** *auto*
    thus *?thesis* **using** *Cons* **by** *fastforce*
  next
    case *False*
    hence *a* ∉ (*dom* *g* ∪ *set* *as*) **by** *simp*
    hence *dom* *g* ∪ *set* (*a* # *as*) = *insert* *a* (*dom* *g* ∪ *set* *as*) **by** *simp*
    thus *?thesis* **using** *Cons* **by** *fastforce*
  qed
qed

lemma *foldr-upd-app*:
  assumes *xin*: *x* ∈ *set* *as*
  shows (*foldr* (λ*p* *ps*. *ps* (*p* ↦ *f* *p*)) *as* *g*) *x* = *Some* (*f* *x*)
  (**is** (*?f* *as* *g*) *x* = *Some* (*f* *x*))
  using *xin*
proof (*induct* *as* *arbitrary*: *x*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons* *a* *as*)
  from *Cons.prems* **show** *?case* **by** (*subst* *foldr.simps*) (*auto* *intro*: *Cons.hyps*)
qed

lemma *foldr-upd-app-other*:
  assumes *xin*: $x \notin$ *set as*
  shows (*foldr* (λp *ps*. *ps* ($p \mapsto f\ p$)) *as g*) $x = g\ x$
  (**is** (*?f as g*) $x = g\ x$)
  using *xin*
proof (*induct as arbitrary*: *x*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons a as*)
  from *Cons.prems* **show** *?case*
    by (*subst foldr.simps*) (*auto intro*: *Cons.hyps*)
qed

lemma *foldr-upd-app-if*:
  foldr (λp *ps*. *ps*($p \mapsto f\ p$)) *as* g = (λx. *if* $x \in$ *set as then Some* ($f\ x$) *else* $g\ x$)
  by (*auto simp*: *foldr-upd-app foldr-upd-app-other*)

lemma *foldl-fun-upd-value*:
  $\bigwedge Y$. *foldl* ($\lambda f\ p$. $f(p := X\ p)$) Y e p = (*if* $p \in$*set* e *then* $X\ p$ *else* $Y\ p$)
  by (*induct e*) *simp-all*

lemma *foldr-fun-upd-value*:
  $\bigwedge Y$. *foldr* ($\lambda p\ f$. $f(p := X\ p)$) e Y p = (*if* $p \in$*set* e *then* $X\ p$ *else* $Y\ p$)
  by (*induct e*) *simp-all*

lemma *foldl-fun-upd-eq-foldr*:
  !!*m*. *foldl* ($\lambda f\ p$. $f(p := g\ p)$) *m xs* = *foldr* ($\lambda p\ f$. $f(p := g\ p)$) *xs m*
  by (*rule ext*) (*simp add*: *foldl-fun-upd-value foldr-fun-upd-value*)

lemma *Cons-eq-neq*:
  $[\![\ y = x;\ x\ \#\ xs \neq y\ \#\ ys\ ]\!] \Longrightarrow xs \neq ys$
  by *simp*

lemma *map-upt-append*:
  assumes *lt*: $x \leq y$
  and *lt2*: $a \leq x$
  shows *map f* [*a* ..< *y*] = *map f* [*a* ..< *x*] @ *map f* [*x* ..< *y*]
proof (*subst map-append* [*symmetric*], *rule arg-cong* [**where** f = *map f*])
  from *lt* **obtain** *k* **where** *ky*: $x + k = y$
    by (*auto simp*: *le-iff-add*)

  thus [*a* ..< *y*] = [*a* ..< *x*] @ [*x* ..< *y*]
    using *lt2*
    by (*auto intro*: *upt-add-eq-append*)
qed

lemma *Min-image-distrib*:
  assumes *minf*: $\bigwedge x\ y$. $[\![\ x \in A;\ y \in A\ ]\!] \Longrightarrow$ *min* ($f\ x$) ($f\ y$) = f (*min x y*)

```
  and      fa: finite A
  and      ane: A ≠ {}
  shows Min (f ‘ A) = f (Min A)
proof −
  have rl: ⋀F. ⟦ F ⊆ A; F ≠ {} ⟧ ⟹ Min (f ‘ F) = f (Min F)
  proof −
    fix F
    assume fa: F ⊆ A and fne: F ≠ {}
    have finite F by (rule finite-subset) fact+

    thus ?thesis F
      unfolding min-def using fa fne fa
    proof (induct rule: finite-subset-induct)
      case empty
      thus ?case by simp
    next
      case (insert x F)
      thus ?case
        by (cases F = {}) (auto dest: Min-in intro: minf)
    qed
  qed

  show ?thesis by (rule rl [OF order-refl]) fact+
qed


lemma min-of-mono':
  assumes (f a ≤ f c) = (a ≤ c)
  shows min (f a) (f c) = f (min a c)
  unfolding min-def
  by (subst if-distrib [where f = f, symmetric], rule arg-cong [where f = f], rule
if-cong [OF - refl refl]) fact+

lemma nat-diff-less:
  fixes x :: nat
  shows ⟦ x < y + z; z ≤ x⟧ ⟹ x − z < y
  using less-diff-conv2 by blast

lemma take-map-Not:
  (take n (map Not xs) = take n xs) = (n = 0 ∨ xs = [])
  by (cases n; simp) (cases xs; simp)

lemma union-trans:
  assumes SR: ⋀x y z. ⟦ (x,y) ∈ S; (y,z) ∈ R ⟧ ⟹ (x,z) ∈ S^*
  shows (R ∪ S)^* = R^* ∪ R^* O S^*
  apply (rule set-eqI)
  apply clarsimp
  apply (rule iffI)
   apply (erule rtrancl-induct; simp)
```

apply (*erule disjE*)
apply (*erule disjE*)
apply (*drule* (*1*) *rtrancl-into-rtrancl*)
apply *blast*
apply *clarsimp*
apply (*drule rtranclD* [**where** $R=S$])
apply (*erule disjE*)
apply *simp*
apply (*erule conjE*)
apply (*drule tranclD2*)
apply (*elim exE conjE*)
apply (*drule* (*1*) *SR*)
apply (*drule* (*1*) *rtrancl-trans*)
apply *blast*
apply (*rule disjI2*)
apply (*erule disjE*)
apply (*blast intro*: *in-rtrancl-UnI*)
apply *clarsimp*
apply (*drule* (*1*) *rtrancl-into-rtrancl*)
apply (*erule* (*1*) *relcompI*)
apply (*erule disjE*)
apply (*blast intro*: *in-rtrancl-UnI*)
apply *clarsimp*
apply (*blast intro*: *in-rtrancl-UnI rtrancl-trans*)
done

lemma *trancl-trancl*:
$(R^+)^+ = R^+$
by *auto*

Some rules for showing that the reflexive transitive closure of a relation/predicate doesn't add much if it was already transitively closed.

lemma *rtrancl-eq-reflc-trans*:
assumes *trans*: *trans* X
shows *rtrancl* $X = X \cup Id$
by (*simp only*: *rtrancl-trancl-reflcl trancl-id*[*OF trans*])

lemma *rtrancl-id*:
assumes *refl*: $Id \subseteq X$
assumes *trans*: *trans* X
shows *rtrancl* $X = X$
using *refl rtrancl-eq-reflc-trans*[*OF trans*]
by *blast*

lemma *rtranclp-eq-reflcp-transp*:
assumes *trans*: *transp* X
shows *rtranclp* $X = (\lambda x\ y.\ X\ x\ y \vee x = y)$
by (*simp add*: *Enum.rtranclp-rtrancl-eq fun-eq-iff*
rtrancl-eq-reflc-trans trans[*unfolded transp-trans*])

lemma *rtranclp-id*:
  shows $reflp\ X \Longrightarrow transp\ X \Longrightarrow rtranclp\ X = X$
  apply (*simp add*: *rtranclp-eq-reflcp-transp*)
  apply (*auto simp*: *fun-eq-iff elim*: *reflpD*)
  done

lemmas *rtranclp-id2* = *rtranclp-id*[*unfolded reflp-def transp-relcompp le-fun-def*]

lemma *if-1-0-0*:
  $((if\ P\ then\ 1\ else\ 0) = (0 :: ('a :: zero\text{-}neq\text{-}one))) = (\neg\ P)$
  by (*simp split*: *if-split*)

lemma *neq-Nil-lengthI*:
  $Suc\ 0 \leq length\ xs \Longrightarrow xs \neq []$
  by (*cases xs*, *auto*)

lemmas *ex-with-length* = *Ex-list-of-length*

lemma *in-singleton*:
  $S = \{x\} \Longrightarrow x \in S$
  by *simp*

lemma *singleton-set*:
 $x \in set\ [a] \Longrightarrow x = a$
  by *auto*

lemma *take-drop-eqI*:
  assumes *t*: $take\ n\ xs = take\ n\ ys$
  assumes *d*: $drop\ n\ xs = drop\ n\ ys$
  shows $xs = ys$
proof −
  have $xs = take\ n\ xs\ @\ drop\ n\ xs$ **by** *simp*
  with *t d*
  have $xs = take\ n\ ys\ @\ drop\ n\ ys$ **by** *simp*
  moreover
  have $ys = take\ n\ ys\ @\ drop\ n\ ys$ **by** *simp*
  ultimately
  show *?thesis* **by** *simp*
qed

lemma *append-len2*:
  $zs = xs\ @\ ys \Longrightarrow length\ xs = length\ zs - length\ ys$
  by *auto*

lemma *if-flip*:
  $(if\ \neg P\ then\ T\ else\ F) = (if\ P\ then\ F\ else\ T)$
  by *simp*

lemma *not-in-domIff*: $f\ x = None = (x \notin dom\ f)$
by *blast*

lemma *not-in-domD*:
$x \notin dom\ f \Longrightarrow f\ x = None$
by (*simp add*:*not-in-domIff*)

definition
$graph\text{-}of\ f \equiv \{(x,y).\ f\ x = Some\ y\}$

lemma *graph-of-None-update*:
$graph\text{-}of\ (f\ (p := None)) = graph\text{-}of\ f - \{p\} \times UNIV$
by (*auto simp*: *graph-of-def split*: *if-split-asm*)

lemma *graph-of-Some-update*:
$graph\text{-}of\ (f\ (p \mapsto v)) = (graph\text{-}of\ f - \{p\} \times UNIV) \cup \{(p,v)\}$
by (*auto simp*: *graph-of-def split*: *if-split-asm*)

lemma *graph-of-restrict-map*:
$graph\text{-}of\ (m\ |\text{`}\ S) \subseteq graph\text{-}of\ m$
by (*simp add*: *graph-of-def restrict-map-def subset-iff*)

lemma *graph-ofD*:
$(x,y) \in graph\text{-}of\ f \Longrightarrow f\ x = Some\ y$
by (*simp add*: *graph-of-def*)

lemma *graph-ofI*:
$m\ x = Some\ y \Longrightarrow (x,\ y) \in graph\text{-}of\ m$
by (*simp add*: *graph-of-def*)

lemma *graph-of-empty* :
$graph\text{-}of\ Map.empty = \{\}$
by (*simp add*: *graph-of-def*)

lemma *graph-of-in-ranD*: $\forall\, y \in ran\ f.\ P\ y \Longrightarrow (x,y) \in graph\text{-}of\ f \Longrightarrow P\ y$
by (*auto simp*: *graph-of-def ran-def*)

lemma *graph-of-SomeD*:
$[\![\ graph\text{-}of\ f \subseteq graph\text{-}of\ g;\ f\ x = Some\ y\ ]\!] \Longrightarrow g\ x = Some\ y$
unfolding *graph-of-def*
by *auto*

lemma *in-set-zip-refl* :
$(x,y) \in set\ (zip\ xs\ xs) = (y = x \wedge x \in set\ xs)$
by (*induct xs*) *auto*

lemma *map-conv-upd*:
$m\ v = None \Longrightarrow m\ o\ (f\ (x := v)) = (m\ o\ f)\ (x := None)$
by (*rule ext*) (*clarsimp simp*: *o-def*)

lemma *sum-all-ex* [*simp*]:
$(\forall\, a.\ x \neq \mathit{Inl}\ a) = (\exists\, a.\ x = \mathit{Inr}\ a)$
$(\forall\, a.\ x \neq \mathit{Inr}\ a) = (\exists\, a.\ x = \mathit{Inl}\ a)$
by (*metis Inr-not-Inl sum.exhaust*)+

lemma *split-distrib*: $\mathit{case\text{-}prod}\ (\lambda a\ b.\ T\ (f\ a\ b)) = (\lambda x.\ T\ (\mathit{case\text{-}prod}\ (\lambda a\ b.\ f\ a\ b)\ x))$
by (*clarsimp simp*: *split-def*)

lemma *case-sum-triv* [*simp*]:
$(\mathit{case}\ x\ \mathit{of}\ \mathit{Inl}\ x \Rightarrow \mathit{Inl}\ x \mid \mathit{Inr}\ x \Rightarrow \mathit{Inr}\ x) = x$
by (*clarsimp split*: *sum.splits*)

lemma *set-eq-UNIV*: $(\{a.\ P\ a\} = \mathit{UNIV}) = (\forall\, a.\ P\ a)$
by *force*

lemma *allE2*:
$[\![\forall\, x\ y.\ P\ x\ y;\ P\ x\ y \Longrightarrow R]\!] \Longrightarrow R$
by *blast*

lemma *allE3*: $[\![\ \forall\, x\ y\ z.\ P\ x\ y\ z;\ P\ x\ y\ z \Longrightarrow R\ ]\!] \Longrightarrow R$
by *auto*

lemma *my-BallE*: $[\![\ \forall\, x \in A.\ P\ x;\ y \in A;\ P\ y \Longrightarrow Q\ ]\!] \Longrightarrow Q$
by (*simp add*: *Ball-def*)

lemma *unit-Inl-or-Inr* [*simp*]:
$\bigwedge a.\ (a \neq \mathit{Inl}\ ()) = (a = \mathit{Inr}\ ())$
$\bigwedge a.\ (a \neq \mathit{Inr}\ ()) = (a = \mathit{Inl}\ ())$
by (*case-tac a*; *clarsimp*)+

lemma *disjE-L*: $[\![\ a \vee b;\ a \Longrightarrow R;\ [\![\ \neg\ a;\ b\ ]\!] \Longrightarrow R\ ]\!] \Longrightarrow R$
by *blast*

lemma *disjE-R*: $[\![\ a \vee b;\ [\![\ \neg\ b;\ a\ ]\!] \Longrightarrow R;\ [\![\ b\ ]\!] \Longrightarrow R\ ]\!] \Longrightarrow R$
by *blast*

lemma *int-max-thms*:
$(a :: \mathit{int}) \leq \mathit{max}\ a\ b$
$(b :: \mathit{int}) \leq \mathit{max}\ a\ b$
by (*auto simp*: *max-def*)

lemma *sgn-negation* [*simp*]:
$\mathit{sgn}\ (-(x{::}\mathit{int})) = -\ \mathit{sgn}\ x$
by (*clarsimp simp*: *sgn-if*)

lemma *sgn-sgn-nonneg* [*simp*]:
$\mathit{sgn}\ (a :: \mathit{int}) * \mathit{sgn}\ a \neq -\mathit{1}$

by (*clarsimp simp*: *sgn-if*)

lemma *inj-inj-on*:
$inj\ f \Longrightarrow inj\text{-}on\ f\ A$
by (*metis injD inj-onI*)

lemma *ex-eqI*:
$[\![\bigwedge x.\ f\ x = g\ x]\!] \Longrightarrow (\exists\, x.\ f\ x) = (\exists\, x.\ g\ x)$
by *simp*

lemma *pre-post-ex*:
$[\![\exists\, x.\ P\ x;\ \bigwedge x.\ P\ x \Longrightarrow Q\ x]\!] \Longrightarrow \exists\, x.\ Q\ x$
by *auto*

lemma *ex-conj-increase*:
$((\exists\, x.\ P\ x) \wedge Q) = (\exists\, x.\ P\ x \wedge Q)$
$(R \wedge (\exists\, x.\ S\ x)) = (\exists\, x.\ R \wedge S\ x)$
by *simp+*

lemma *all-conj-increase*:
$((\ \forall\, x.\ P\ x) \wedge Q) = (\forall\, x.\ P\ x \wedge Q)$
$(R \wedge (\forall\, x.\ S\ x)) = (\forall\, x.\ R \wedge S\ x)$
by *simp+*

lemma *Ball-conj-increase*:
$xs \neq \{\} \Longrightarrow ((\ \forall\, x \in xs.\ P\ x) \wedge Q) = (\forall\, x \in xs.\ P\ x \wedge Q)$
$xs \neq \{\} \Longrightarrow (R \wedge (\forall\, x \in xs.\ S\ x)) = (\forall\, x \in xs.\ R \wedge S\ x)$
by *auto*

lemma *disjoint-subset*:
assumes $A' \subseteq A$ **and** $A \cap B = \{\}$
shows $A' \cap B = \{\}$
using *assms* **by** *auto*

lemma *disjoint-subset2*:
assumes $B' \subseteq B$ **and** $A \cap B = \{\}$
shows $A \cap B' = \{\}$
using *assms* **by** *auto*

lemma *UN-nth-mem*:
$i < length\ xs \Longrightarrow f\ (xs\ !\ i) \subseteq (\bigcup x{\in}set\ xs.\ f\ x)$
by (*metis UN-upper nth-mem*)

lemma *Union-equal*:
$f\ `\ A = f\ `\ B \Longrightarrow (\bigcup x \in A.\ f\ x) = (\bigcup x \in B.\ f\ x)$
by *blast*

lemma *UN-Diff-disjoint*:
$i < length\ xs \Longrightarrow (A - (\bigcup x{\in}set\ xs.\ f\ x)) \cap f\ (xs\ !\ i) = \{\}$
by (*metis Diff-disjoint Int-commute UN-nth-mem disjoint-subset*)

lemma *image-list-update*:
$f\ a = f\ (xs\ !\ i)$
$\Longrightarrow f\ `\ set\ (xs\ [i := a]) = f\ `\ set\ xs$
by (*metis list-update-id map-update set-map*)

lemma *Union-list-update-id*:
$f\ a = f\ (xs\ !\ i) \Longrightarrow (\bigcup x{\in}set\ (xs\ [i := a]).\ f\ x) = (\bigcup x{\in}set\ xs.\ f\ x)$
by (*rule Union-equal*) (*erule image-list-update*)

lemma *Union-list-update-id'*:
$[\![i < length\ xs;\ \bigwedge x.\ g\ (f\ x) = g\ x]\!]$
$\Longrightarrow (\bigcup x{\in}set\ (xs\ [i := f\ (xs\ !\ i)]).\ g\ x) = (\bigcup x{\in}set\ xs.\ g\ x)$
by (*metis Union-list-update-id*)

lemma *Union-subset*:
$[\![\bigwedge x.\ x \in A \Longrightarrow (f\ x) \subseteq (g\ x)]\!] \Longrightarrow (\bigcup x \in A.\ f\ x) \subseteq (\bigcup x \in A.\ g\ x)$
by (*metis UN-mono order-refl*)

lemma *UN-sub-empty*:
$[\![list\text{-}all\ P\ xs;\ \bigwedge x.\ P\ x \Longrightarrow f\ x = g\ x]\!] \Longrightarrow (\bigcup x{\in}set\ xs.\ f\ x) - (\bigcup x{\in}set\ xs.\ g\ x) = \{\}$
by (*simp add*: *Ball-set-list-all*[*symmetric*] *Union-subset*)

lemma *bij-betw-fun-updI*:
$[\![x \notin A;\ y \notin B;\ bij\text{-}betw\ f\ A\ B]\!] \Longrightarrow bij\text{-}betw\ (f(x := y))\ (insert\ x\ A)\ (insert\ y\ B)$
by (*clarsimp simp*: *bij-betw-def fun-upd-image inj-on-fun-updI split*: *if-split-asm*; *blast*)

definition
$bij\text{-}betw\text{-}map\ f\ A\ B \equiv bij\text{-}betw\ f\ A\ (Some\ `\ B)$

lemma *bij-betw-map-fun-updI*:
$[\![x \notin A;\ y \notin B;\ bij\text{-}betw\text{-}map\ f\ A\ B]\!]$
$\Longrightarrow bij\text{-}betw\text{-}map\ (f(x \mapsto y))\ (insert\ x\ A)\ (insert\ y\ B)$
unfolding *bij-betw-map-def* **by** *clarsimp* (*erule bij-betw-fun-updI*; *clarsimp*)

lemma *bij-betw-map-imp-inj-on*:
$bij\text{-}betw\text{-}map\ f\ A\ B \Longrightarrow inj\text{-}on\ f\ A$
by (*simp add*: *bij-betw-map-def bij-betw-imp-inj-on*)

lemma *bij-betw-empty-dom-exists*:
$r = \{\} \Longrightarrow \exists\, t.\ bij\text{-}betw\ t\ \{\}\ r$

by (*clarsimp simp*: *bij-betw-def*)

lemma *bij-betw-map-empty-dom-exists*:
$r = \{\} \Longrightarrow \exists\, t.\ bij\text{-}betw\text{-}map\ t\ \{\}\ r$
by (*clarsimp simp*: *bij-betw-map-def bij-betw-empty-dom-exists*)

lemma *funpow-add* [*simp*]:
fixes $f :: 'a \Rightarrow 'a$
shows $(f\ \hat{}\hat{}\ a)\ ((f\ \hat{}\hat{}\ b)\ s) = (f\ \hat{}\hat{}\ (a + b))\ s$
by (*metis comp-apply funpow-add*)

lemma *funpow-unfold*:
fixes $f :: 'a \Rightarrow 'a$
assumes $n > 0$
shows $f\ \hat{}\hat{}\ n = (f\ \hat{}\hat{}\ (n - 1)) \circ f$
by (*metis Suc-diff-1 assms funpow-Suc-right*)

lemma *relpow-unfold*: $n > 0 \Longrightarrow S\ \hat{}\hat{}\ n = (S\ \hat{}\hat{}\ (n - 1))\ O\ S$
by (*cases n*, *auto*)

definition
equiv-of :: $('s \Rightarrow 't) \Rightarrow ('s \times 's)$ *set*
where
equiv-of proj $\equiv \{(a, b).\ proj\ a = proj\ b\}$

lemma *equiv-of-is-equiv-relation* [*simp*]:
equiv UNIV (*equiv-of proj*)
by (*auto simp*: *equiv-of-def intro*!: *equivI refl-onI symI transI*)

lemma *in-equiv-of* [*simp*]:
$((a, b) \in equiv\text{-}of\ f) \longleftrightarrow (f\ a = f\ b)$
by (*clarsimp simp*: *equiv-of-def*)

lemma *equiv-relation-to-projection*:
fixes $R :: ('a \times 'a)$ *set*
assumes *equiv*: *equiv UNIV R*
shows $\exists f :: 'a \Rightarrow 'a\ set.\ \forall\, x\ y.\ f\ x = f\ y \longleftrightarrow (x, y) \in R$
apply (*rule exI* [*of* - $\lambda x.\ \{y.\ (x, y) \in R\}$])
apply *clarsimp*
apply (*case-tac* $(x, y) \in R$)
apply *clarsimp*
apply (*rule set-eqI*)

```
   apply clarsimp
   apply (metis equivE sym-def trans-def equiv)
  apply (clarsimp)
  apply (metis UNIV-I equiv equivE mem-Collect-eq refl-on-def)
  done

lemma range-constant [simp]:
  range (λ-. k) = {k}
  by (clarsimp simp: image-def)

lemma dom-unpack:
  dom (map-of (map (λx. (f x, g x)) xs)) = set (map (λx. f x) xs)
  by (simp add: dom-map-of-conv-image-fst image-image)

lemma fold-to-disj:
fold (++) ms a x = Some y ⟹ (∃ b ∈ set ms. b x = Some y) ∨ a x = Some y
  by (induct ms arbitrary:a x y; clarsimp) blast

lemma fold-ignore1:
  a x = Some y ⟹ fold (++) ms a x = Some y
  by (induct ms arbitrary:a x y; clarsimp)

lemma fold-ignore2:
  fold (++) ms a x = None ⟹ a x = None
  by (metis fold-ignore1 option.collapse)

lemma fold-ignore3:
  fold (++) ms a x = None ⟹ (∀ b ∈ set ms. b x = None)
  by (induct ms arbitrary:a x; clarsimp) (meson fold-ignore2 map-add-None)

lemma fold-ignore4:
  b ∈ set ms ⟹ b x = Some y ⟹ ∃ y. fold (++) ms a x = Some y
  using fold-ignore3 by fastforce

lemma dom-unpack2:
  dom (fold (++) ms Map.empty) = ⋃(set (map dom ms))
  apply (induct ms; clarsimp simp:dom-def)
  apply (rule equalityI; clarsimp)
   apply (drule fold-to-disj)
   apply (erule disjE)
    apply clarsimp
    apply (rename-tac b)
    apply (erule-tac x=b in ballE; clarsimp)
   apply clarsimp
  apply (rule conjI)
   apply clarsimp
   apply (rule-tac x=y in exI)
   apply (erule fold-ignore1)
  apply clarsimp
```

apply (*rename-tac y*)
apply (*erule-tac y=y* **in** *fold-ignore4*; *clarsimp*)
done

lemma *fold-ignore5*:*fold* (++) *ms a x* = *Some y* $\Longrightarrow$ *a x* = *Some y* $\vee$ ($\exists$ *b* $\in$ *set ms*. *b x* = *Some y*)
by (*induct ms arbitrary*:*a x y*; *clarsimp*) *blast*

lemma *dom-inter-nothing*:*dom f* $\cap$ *dom g* = {} $\Longrightarrow$ $\forall$ *x*. *f x* = *None* $\vee$ *g x* = *None*
by *auto*

lemma *fold-ignore6*:
f x = *None* $\Longrightarrow$ *fold* (++) *ms f x* = *fold* (++) *ms Map.empty x*
apply (*induct ms arbitrary*:*f x*; *clarsimp simp*:*map-add-def*)
by (*metis* (*no-types*, *lifting*) *fold-ignore1 option.collapse option.simps*(*4*))

lemma *fold-ignore7*:
m x = *m′ x* $\Longrightarrow$ *fold* (++) *ms m x* = *fold* (++) *ms m′ x*
apply (*case-tac m x*)
apply (*frule-tac ms=ms* **in** *fold-ignore6*)
apply (*cut-tac f=m′* **and** *ms=ms* **and** *x=x* **in** *fold-ignore6*)
apply *clarsimp+*
apply (*rename-tac a*)
apply (*cut-tac ms=ms* **and** *a=m* **and** *x=x* **and** *y=a* **in** *fold-ignore1*, *clarsimp*)
apply (*cut-tac ms=ms* **and** *a=m′* **and** *x=x* **and** *y=a* **in** *fold-ignore1*; *clarsimp*)
done

lemma *fold-ignore8*:
fold (++) *ms* [*x* $\mapsto$ *y*] = (*fold* (++) *ms Map.empty*)(*x* $\mapsto$ *y*)
apply (*rule ext*)
apply (*rename-tac xa*)
apply (*case-tac xa* = *x*)
apply *clarsimp*
apply (*rule fold-ignore1*)
apply *clarsimp*
apply (*subst fold-ignore6*; *clarsimp*)
done

lemma *fold-ignore9*:
$[\![$*fold* (++) *ms* [*x* $\mapsto$ *y*] *x′* = *Some z*; *x* = *x′*$]\!]$ $\Longrightarrow$ *y* = *z*
by (*subst* (*asm*) *fold-ignore8*) *clarsimp*

lemma *fold-to-map-of*:
fold (++) (*map* (λ*x*. [*f x* $\mapsto$ *g x*]) *xs*) *Map.empty* = *map-of* (*map* (λ*x*. (*f x*, *g x*)) *xs*)
apply (*rule ext*)
apply (*rename-tac x*)
apply (*case-tac fold* (++) (*map* (λ*x*. [*f x* $\mapsto$ *g x*]) *xs*) *Map.empty x*)

apply *clarsimp*
apply (*drule fold-ignore3*)
apply (*clarsimp split*:*if-split-asm*)
apply (*rule sym*)
apply (*subst map-of-eq-None-iff*)
apply *clarsimp*
apply (*rename-tac xa*)
apply (*erule-tac x*=*xa* **in** *ballE*; *clarsimp*)
apply *clarsimp*
apply (*frule fold-ignore5*; *clarsimp split*:*if-split-asm*)
apply (*subst map-add-map-of-foldr*[**where** *m*=*Map.empty*, *simplified*])
apply (*induct xs arbitrary*:*f g*; *clarsimp split*:*if-split*)
apply (*rule conjI*; *clarsimp*)
apply (*drule fold-ignore9*; *clarsimp*)
apply (*cut-tac ms*=*map* ($\lambda x.$ [*f x* $\mapsto$ *g x*]) *xs* **and** *f*=[*f a* $\mapsto$ *g a*] **and** *x*=*f b* **in** *fold-ignore6*, *clarsimp*)
apply *auto*
done

lemma *if-n-0-0*:
((*if P then n else 0*) $\neq$ *0*) = (*P* $\wedge$ *n* $\neq$ *0*)
by (*simp split*: *if-split*)

lemma *insert-dom*:
assumes *fx*: *f x* = *Some y*
shows *insert x* (*dom f*) = *dom f*
unfolding *dom-def* **using** *fx* **by** *auto*

lemma *map-comp-subset-dom*:
dom (*prj* $\circ_m$ *f*) $\subseteq$ *dom f*
unfolding *dom-def*
by (*auto simp*: *map-comp-Some-iff*)

lemmas *map-comp-subset-domD* = *subsetD* [*OF map-comp-subset-dom*]

lemma *dom-map-comp*:
x $\in$ *dom* (*prj* $\circ_m$ *f*) = ($\exists$ *y z*. *f x* = *Some y* $\wedge$ *prj y* = *Some z*)
by (*fastforce simp*: *dom-def map-comp-Some-iff*)

lemma *map-option-Some-eq2*:
(*Some y* = *map-option f x*) = ($\exists$ *z*. *x* = *Some z* $\wedge$ *f z* = *y*)
by (*metis map-option-eq-Some*)

lemma *map-option-eq-dom-eq*:
assumes *ome*: *map-option f* $\circ$ *g* = *map-option f* $\circ$ *g'*
shows *dom g* = *dom g'*
proof (*rule set-eqI*)
fix *x*
{

assume $x \in dom\ g$
hence $Some\ (f\ (the\ (g\ x))) = (map\text{-}option\ f \circ g)\ x$
by $(auto\ simp\colon map\text{-}option\text{-}case\ split\colon option.splits)$
also have $\ldots = (map\text{-}option\ f \circ g')\ x$ **by** $(simp\ add\colon ome)$
finally have $x \in dom\ g'$
by $(auto\ simp\colon map\text{-}option\text{-}case\ split\colon option.splits)$
} **moreover**
{
assume $x \in dom\ g'$
hence $Some\ (f\ (the\ (g'\ x))) = (map\text{-}option\ f \circ g')\ x$
by $(auto\ simp\colon map\text{-}option\text{-}case\ split\colon option.splits)$
also have $\ldots = (map\text{-}option\ f \circ g)\ x$ **by** $(simp\ add\colon ome)$
finally have $x \in dom\ g$
by $(auto\ simp\colon map\text{-}option\text{-}case\ split\colon option.splits)$
} **ultimately show** $(x \in dom\ g) = (x \in dom\ g')$ **by** *auto*
qed

lemma *cart-singleton-image*:
$S \times \{s\} = (\lambda v.\ (v,\ s))\ `\ S$
by *auto*

lemma *singleton-eq-o2s*:
$(\{x\} = set\text{-}option\ v) = (v = Some\ x)$
by $(cases\ v,\ auto)$

lemma *option-set-singleton-eq*:
$(set\text{-}option\ opt = \{v\}) = (opt = Some\ v)$
by $(cases\ opt,\ simp\text{-}all)$

lemmas *option-set-singleton-eqs*
$=$ *option-set-singleton-eq*
trans[*OF eq-commute option-set-singleton-eq*]

lemma *map-option-comp2*:
$map\text{-}option\ (f\ o\ g) = map\text{-}option\ f\ o\ map\text{-}option\ g$
by $(simp\ add\colon option.map\text{-}comp\ fun\text{-}eq\text{-}iff)$

lemma *compD*:
$[\![f \circ g = f \circ g';\ g\ x = v\ ]\!] \Longrightarrow f\ (g'\ x) = f\ v$
by $(metis\ comp\text{-}apply)$

lemma *map-option-comp-eqE*:
assumes $om\colon map\text{-}option\ f \circ mp = map\text{-}option\ f \circ mp'$
and $p1\colon [\![\ mp\ x = None;\ mp'\ x = None\ ]\!] \Longrightarrow P$
and $p2\colon \bigwedge v\ v'.\ [\![\ mp\ x = Some\ v;\ mp'\ x = Some\ v';\ f\ v = f\ v'\ ]\!] \Longrightarrow P$
shows P
proof $(cases\ mp\ x)$
case *None*

hence $x \notin dom\ mp$ **by** $(simp\ add: domIff)$
hence $mp'\ x = None$ **by** $(simp\ add: map\text{-}option\text{-}eq\text{-}dom\text{-}eq\ [OF\ om]\ domIff)$
with $None$ **show** $?thesis$ **by** $(rule\ p1)$
next
case $(Some\ v)$
hence $x \in dom\ mp$ **by** $clarsimp$
then obtain v' **where** $Some'$: $mp'\ x = Some\ v'$ **by** $(clarsimp\ simp\ add: map\text{-}option\text{-}eq\text{-}dom\text{-}eq\ [OF\ om])$
with $Some$ **show** $?thesis$
proof $(rule\ p2)$
show $f\ v = f\ v'$ **using** $Some'\ compD\ [OF\ om,\ OF\ Some]$ **by** $simp$
qed
qed

lemma $Some\text{-}the$:
$x \in dom\ f \Longrightarrow f\ x = Some\ (the\ (f\ x))$
by $clarsimp$

lemma $map\text{-}comp\text{-}update$:
$f \circ_m (g(x \mapsto v)) = (f \circ_m g)(x := f\ v)$
by $(rule\ ext,\ rename\text{-}tac\ y)\ (case\text{-}tac\ g\ y;\ simp)$

lemma $restrict\text{-}map\text{-}eqI$:
assumes req: $A \restriction' S = B \restriction' S$
and mem: $x \in S$
shows $A\ x = B\ x$
proof −
from mem **have** $A\ x = (A \restriction' S)\ x$ **by** $simp$
also have $\ldots = (B \restriction' S)\ x$ **using** req **by** $simp$
also have $\ldots = B\ x$ **using** mem **by** $simp$
finally show $?thesis$ **.**
qed

lemma $map\text{-}comp\text{-}eqI$:
assumes dm: $dom\ g = dom\ g'$
and fg: $\bigwedge x.\ x \in dom\ g' \Longrightarrow f\ (the\ (g'\ x)) = f\ (the\ (g\ x))$
shows $f \circ_m g = f \circ_m g'$
apply $(rule\ ext)$
apply $(case\text{-}tac\ x \in dom\ g)$
apply $(frule\ subst\ [OF\ dm])$
apply $(clarsimp\ split: option.splits)$
apply $(frule\ domI$ [**where** $m = g'$])
apply $(drule\ fg)$
apply $simp$
apply $(frule\ subst\ [OF\ dm])$
apply $clarsimp$
apply $(drule\ not\text{-}sym)$
apply $(clarsimp\ simp: map\text{-}comp\text{-}Some\text{-}iff)$
done

definition
  modify-map m p f $\equiv$ *m* (*p* := *map-option f* (*m p*))

lemma *modify-map-id*:
  modify-map m p id = *m*
  by (*auto simp add*: *modify-map-def map-option-case split*: *option.splits*)

lemma *modify-map-addr-com*:
  assumes *com*: $x \neq y$
  shows *modify-map* (*modify-map m x g*) *y f* = *modify-map* (*modify-map m y f*) *x g*
  by (*rule ext*) (*simp add*: *modify-map-def map-option-case com split*: *option.splits*)

lemma *modify-map-dom* :
  dom (*modify-map m p f*) = *dom m*
  unfolding *modify-map-def* **by** (*auto simp*: *dom-def*)

lemma *modify-map-None*:
  m x = *None* $\Longrightarrow$ *modify-map m x f* = *m*
  by (*rule ext*) (*simp add*: *modify-map-def*)

lemma *modify-map-ndom* :
  $x \notin$ *dom m* $\Longrightarrow$ *modify-map m x f* = *m*
  by (*rule modify-map-None*) *clarsimp*

lemma *modify-map-app*:
  (*modify-map m p f*) *q* = (*if p* = *q then map-option f* (*m p*) *else m q*)
  unfolding *modify-map-def* **by** *simp*

lemma *modify-map-apply*:
  m p = *Some x* $\Longrightarrow$ *modify-map m p f* = *m* ($p \mapsto f\ x$)
  by (*simp add*: *modify-map-def*)

lemma *modify-map-com*:
  assumes *com*: $\bigwedge x.\ f\ (g\ x) = g\ (f\ x)$
  shows *modify-map* (*modify-map m x g*) *y f* = *modify-map* (*modify-map m y f*) *x g*
  using *assms* **by** (*auto simp*: *modify-map-def map-option-case split*: *option.splits*)

lemma *modify-map-comp*:
  modify-map m x (*f o g*) = *modify-map* (*modify-map m x g*) *x f*
  by (*rule ext*) (*simp add*: *modify-map-def option.map-comp*)

lemma *modify-map-exists-eq*:
  ($\exists$ *cte*. *modify-map m p' f p*= *Some cte*) = ($\exists$ *cte*. *m p* = *Some cte*)
  by (*auto simp*: *modify-map-def split*: *if-splits*)

lemma *modify-map-other*:
$p \neq q \Longrightarrow (\textit{modify-map}\ m\ p\ f)\ q = (m\ q)$
by (*simp add*: *modify-map-app*)

lemma *modify-map-same*:
$\textit{modify-map}\ m\ p\ f\ p = \textit{map-option}\ f\ (m\ p)$
by (*simp add*: *modify-map-app*)

lemma *next-update-is-modify*:
$[\![\ m\ p = \textit{Some}\ \textit{cte}';\ \textit{cte} = f\ \textit{cte}'\ ]\!] \Longrightarrow (m(p \mapsto \textit{cte})) = \textit{modify-map}\ m\ p\ f$
unfolding *modify-map-def* **by** *simp*

lemma *nat-power-minus-less*:
$a < 2\ \hat{}\ (x - n) \Longrightarrow (a :: \textit{nat}) < 2\ \hat{}\ x$
by (*erule order-less-le-trans*) *simp*

lemma *neg-rtranclI*:
$[\![\ x \neq y;\ (x,\ y) \notin R^+\ ]\!] \Longrightarrow (x,\ y) \notin R^*$
by (*meson rtranclD*)

lemma *neg-rtrancl-into-trancl*:
$\neg\ (x,\ y) \in R^* \Longrightarrow \neg\ (x,\ y) \in R^+$
by (*erule contrapos-nn*, *erule trancl-into-rtrancl*)

lemma *set-neqI*:
$[\![\ x \in S;\ x \notin S'\ ]\!] \Longrightarrow S \neq S'$
by *clarsimp*

lemma *set-pair-UN*:
$\{x.\ P\ x\} = \textit{UNION}\ \{xa.\ \exists\, xb.\ P\ (xa,\ xb)\}\ (\lambda xa.\ \{xa\} \times \{xb.\ P\ (xa,\ xb)\})$
by *fastforce*

lemma *singleton-elemD*: $S = \{x\} \Longrightarrow x \in S$
by *simp*

lemma *singleton-eqD*: $A = \{x\} \Longrightarrow x \in A$
by *blast*

lemma *ball-ran-fun-updI*:
$[\![\ \forall\, v \in \textit{ran}\ m.\ P\ v;\ \forall\, v.\ y = \textit{Some}\ v \longrightarrow P\ v\ ]\!] \Longrightarrow \forall\, v \in \textit{ran}\ (m\ (x := y)).\ P\ v$
by (*auto simp add*: *ran-def*)

lemma *ball-ran-eq*:
$(\forall\, y \in \textit{ran}\ m.\ P\ y) = (\forall\, x\ y.\ m\ x = \textit{Some}\ y \longrightarrow P\ y)$
by (*auto simp add*: *ran-def*)

lemma *cart-helper*:
$(\{\} = \{x\} \times S) = (S = \{\})$
by *blast*

lemmas *converse-trancl-induct′* = *converse-trancl-induct* [*consumes 1*, *case-names base step*]

lemma *disjCI2*: $(\neg\ P \Longrightarrow Q) \Longrightarrow P \vee Q$ **by** *blast*

lemma *insert-UNIV* :
  $insert\ x\ UNIV = UNIV$
  by *blast*

lemma *not-singletonE*:
  $[\![\forall p.\ S \neq \{p\};\ S \neq \{\};\ \bigwedge p\ p'.\ [\![p \neq p';\ p \in S;\ p' \in S]\!] \Longrightarrow R]\!] \Longrightarrow R$
  by *blast*

lemma *not-singleton-oneE*:
  $[\![\forall p.\ S \neq \{p\};\ p \in S;\ \bigwedge p'.\ [\![p \neq p';\ p' \in S]\!] \Longrightarrow R]\!] \Longrightarrow R$
  using *not-singletonE* **by** *fastforce*

lemma *ball-ran-modify-map-eq*:
  $[\![\forall v.\ m\ x = Some\ v \longrightarrow P\ (f\ v) = P\ v]\!]$
  $\Longrightarrow (\forall v \in ran\ (modify\text{-}map\ m\ x\ f).\ P\ v) = (\forall v \in ran\ m.\ P\ v)$
  by (*auto simp*: *modify-map-def ball-ran-eq*)

lemma *disj-imp*: $(P \vee Q) = (\neg P \longrightarrow Q)$ **by** *blast*

lemma *eq-singleton-redux*:
  $[\![S = \{x\}]\!] \Longrightarrow x \in S$
  by *simp*

lemma *if-eq-elem-helperE*:
  $[\![x \in (if\ P\ then\ S\ else\ S');\ \ [\![P;\ \ \ x \in S\ ]\!] \Longrightarrow a = b;\ \ [\![\neg\ P;\ x \in S']\!] \Longrightarrow a = c]\!]$
  $\Longrightarrow a = (if\ P\ then\ b\ else\ c)$
  by *fastforce*

lemma *if-option-Some*:
  $((if\ P\ then\ None\ else\ Some\ x) = Some\ y) = (\neg P \wedge x = y)$
  by *simp*

lemma *insert-minus-eq*:
  $x \notin A \Longrightarrow A - S = (A - (S - \{x\}))$
  by *auto*

lemma *modify-map-K-D*:
  $modify\text{-}map\ m\ p\ (\lambda x.\ y)\ p' = Some\ v \Longrightarrow (m\ (p \mapsto y))\ p' = Some\ v$
  by (*simp add*: *modify-map-def split*: *if-split-asm*)

lemma *tranclE2*:
  assumes *trancl*: $(a,\ b) \in r^+$

and *base*: $(a, b) \in r \Longrightarrow P$
and *step*: $\bigwedge c.\ [\![(a, c) \in r; (c, b) \in r^+]\!] \Longrightarrow P$
shows P
using *trancl base step*
proof −
note rl = *converse-trancl-induct* [**where** $P = \lambda x.\ x = a \longrightarrow P$]
from *trancl* **have** $a = a \longrightarrow P$
by (*rule rl*, (*iprover intro*: *base step*)+)
thus *?thesis* **by** *simp*
qed

lemmas *tranclE2'* = *tranclE2* [*consumes 1*, *case-names base trancl*]

lemma *weak-imp-cong*:
$[\![\ P = R;\ Q = S\ ]\!] \Longrightarrow (P \longrightarrow Q) = (R \longrightarrow S)$
by *simp*

lemma *Collect-Diff-restrict-simp*:
$T - \{x \in T.\ Q\ x\} = T - \{x.\ Q\ x\}$
by (*auto intro*: *Collect-cong*)

lemma *Collect-Int-pred-eq*:
$\{x \in S.\ P\ x\} \cap \{x \in T.\ P\ x\} = \{x \in (S \cap T).\ P\ x\}$
by (*simp add*: *Collect-conj-eq* [*symmetric*] *conj-comms*)

lemma *Collect-restrict-predR*:
$\{x.\ P\ x\} \cap T = \{\} \Longrightarrow \{x.\ P\ x\} \cap \{x \in T.\ Q\ x\} = \{\}$
by (*fastforce simp*: *disjoint-iff-not-equal*)

lemma *Diff-Un2*:
assumes *emptyad*: $A \cap D = \{\}$
and *emptybc*: $B \cap C = \{\}$
shows $(A \cup B) - (C \cup D) = (A - C) \cup (B - D)$
proof −
have $(A \cup B) - (C \cup D) = (A \cup B - C) \cap (A \cup B - D)$
by (*rule Diff-Un*)
also have $\ldots = ((A - C) \cup B) \cap (A \cup (B - D))$ **using** *emptyad emptybc*
by (*simp add*: *Un-Diff Diff-triv*)
also have $\ldots = (A - C) \cup (B - D)$
proof −
have $(A - C) \cap (A \cup (B - D)) = A - C$ **using** *emptyad emptybc*
by (*metis Diff-Int2 Diff-Int-distrib2 inf-sup-absorb*)
moreover
have $B \cap (A \cup (B - D)) = B - D$ **using** *emptyad emptybc*
by (*metis Int-Diff Un-Diff Un-Diff-Int Un-commute Un-empty-left inf-sup-absorb*)
ultimately show *?thesis*
by (*simp add*: *Int-Un-distrib2*)
qed
finally show *?thesis* **.**

qed

lemma *ballEI*:
$[\![\forall x \in S.\ Q\ x;\ \bigwedge x.\ [\![x \in S;\ Q\ x]\!] \Longrightarrow P\ x]\!] \Longrightarrow \forall x \in S.\ P\ x$
by *auto*

lemma *dom-if-None*:
$dom\ (\lambda x.\ if\ P\ x\ then\ None\ else\ f\ x) = dom\ f - \{x.\ P\ x\}$
by (*simp add*: *dom-def*) *fastforce*

lemma *restrict-map-Some-iff*:
$((m\ |`\ S)\ x = Some\ y) = (m\ x = Some\ y \wedge x \in S)$
by (*cases* $x \in S$, *simp-all*)

lemma *context-case-bools*:
$[\![\bigwedge v.\ P\ v \Longrightarrow R\ v;\ [\![\neg\ P\ v;\ \bigwedge v.\ P\ v \Longrightarrow R\ v]\!] \Longrightarrow R\ v]\!] \Longrightarrow R\ v$
by (*cases* $P\ v$, *simp-all*)

lemma *inj-on-fun-upd-strongerI*:
$[\![inj\text{-}on\ f\ A;\ y \notin f\ `\ (A - \{x\})]\!] \Longrightarrow inj\text{-}on\ (f(x := y))\ A$
by (*fastforce simp*: *inj-on-def*)

lemma *less-handy-casesE*:
$[\![m < n;\ m = 0 \Longrightarrow R;\ \bigwedge m'\ n'.\ [\![n = Suc\ n';\ m = Suc\ m';\ m < n]\!] \Longrightarrow R]\!]$
$\Longrightarrow R$
by (*case-tac* n; *simp*) (*case-tac* m; *simp*)

lemma *subset-drop-Diff-strg*:
$(A \subseteq C) \longrightarrow (A - B \subseteq C)$
by *blast*

lemma *inj-case-bool*:
$inj\ (case\text{-}bool\ a\ b) = (a \neq b)$
by (*auto dest*: *inj-onD*[**where** x=*True* **and** y=*False*] *intro*: *inj-onI split*: *bool.split-asm*)

lemma *foldl-fun-upd*:
$foldl\ (\lambda s\ r.\ s\ (r := g\ r))\ f\ rs = (\lambda x.\ if\ x \in set\ rs\ then\ g\ x\ else\ f\ x)$
by (*induct rs arbitrary*: f) (*auto simp*: *fun-eq-iff*)

lemma *all-rv-choice-fn-eq-pred*:
$[\![\bigwedge rv.\ P\ rv \Longrightarrow \exists fn.\ f\ rv = g\ fn]\!] \Longrightarrow \exists fn.\ \forall rv.\ P\ rv \longrightarrow f\ rv = g\ (fn\ rv)$
apply (*rule-tac* x=$\lambda rv.\ SOME\ h.\ f\ rv = g\ h$ **in** *exI*)
apply (*clarsimp split*: *if-split*)
by (*meson someI-ex*)

lemma *ex-const-function*:
$\exists f.\ \forall s.\ f\ (f'\ s) = v$
by *force*

lemma *if-Const-helper*:
$If\ P\ (Con\ x)\ (Con\ y) = Con\ (If\ P\ x\ y)$
by (*simp split*: *if-split*)

lemmas *if-Some-helper* = *if-Const-helper*[**where** *Con=Some*]

lemma *expand-restrict-map-eq*:
$(m\ |`\ S = m'\ |`\ S) = (\forall x.\ x \in S \longrightarrow m\ x = m'\ x)$
by (*simp add*: *fun-eq-iff restrict-map-def split*: *if-split*)

lemma *disj-imp-rhs*:
$(P \Longrightarrow Q) \Longrightarrow (P \vee Q) = Q$
by *blast*

lemma *remove1-filter*:
$distinct\ xs \Longrightarrow remove1\ x\ xs = filter\ (\lambda y.\ x \neq y)\ xs$
by (*induct xs*) (*auto intro*!: *filter-True* [*symmetric*])

lemma *Int-Union-empty*:
$(\bigwedge x.\ x \in S \Longrightarrow A \cap P\ x = \{\}) \Longrightarrow A \cap (\bigcup x \in S.\ P\ x) = \{\}$
by *auto*

lemma *UN-Int-empty*:
$(\bigwedge x.\ x \in S \Longrightarrow P\ x \cap T = \{\}) \Longrightarrow (\bigcup x \in S.\ P\ x) \cap T = \{\}$
by *auto*

lemma *disjointI*:
$[\![\bigwedge x\ y.\ [\![\ x \in A;\ y \in B\ ]\!] \Longrightarrow x \neq y\ ]\!] \Longrightarrow A \cap B = \{\}$
by *auto*

lemma *UN-disjointI*:
assumes *rl*: $\bigwedge x\ y.\ [\![\ x \in A;\ y \in B\ ]\!] \Longrightarrow P\ x \cap Q\ y = \{\}$
shows $(\bigcup x \in A.\ P\ x) \cap (\bigcup x \in B.\ Q\ x) = \{\}$
by (*auto dest*: *rl*)

lemma *UN-set-member*:
assumes *sub*: $A \subseteq (\bigcup x \in S.\ P\ x)$
and *nz*: $A \neq \{\}$
shows $\exists x \in S.\ P\ x \cap A \neq \{\}$
proof −
from *nz* **obtain** *z* **where** *zA*: $z \in A$ **by** *fastforce*
with *sub* **obtain** *x* **where** $x \in S$ **and** $z \in P\ x$ **by** *auto*
hence $P\ x \cap A \neq \{\}$ **using** *zA* **by** *auto*
thus *?thesis* **using** *sub nz* **by** *auto*
qed

lemma *append-Cons-cases* [*consumes 1*, *case-names pre mid post*]:
$[\![(x, y) \in set\ (as\ @\ b\ \#\ bs);$
$(x, y) \in set\ as \Longrightarrow R;$

$[\![(x,\ y) \notin set\ as;\ (x,\ y) \notin set\ bs;\ (x,\ y) = b]\!] \Longrightarrow R;$
$(x,\ y) \in set\ bs \Longrightarrow R]\!] \Longrightarrow R$
by *auto*

lemma *cart-singletons*:
$\{a\} \times \{b\} = \{(a,\ b)\}$
by *blast*

lemma *disjoint-subset-neg1*:
$[\![\ B \cap C = \{\};\ A \subseteq B;\ A \neq \{\}\ ]\!] \Longrightarrow \neg\ A \subseteq C$
by *auto*

lemma *disjoint-subset-neg2*:
$[\![\ B \cap C = \{\};\ A \subseteq C;\ A \neq \{\}\ ]\!] \Longrightarrow \neg\ A \subseteq B$
by *auto*

lemma *iffE2*:
$[\![\ P = Q;\ [\![\ P;\ Q\ ]\!] \Longrightarrow R;\ [\![\ \neg\ P;\ \neg\ Q\ ]\!] \Longrightarrow R\ ]\!] \Longrightarrow R$
by *blast*

lemma *list-case-If*:
$(case\ xs\ of\ []\ \Rightarrow P\ |\ \text{-} \Rightarrow Q) = (if\ xs = []\ then\ P\ else\ Q)$
by (*rule list.case-eq-if*)

lemma *remove1-Nil-in-set*:
$[\![\ remove1\ x\ xs = [];\ xs \neq []\ ]\!] \Longrightarrow x \in set\ xs$
by (*induct xs*) (*auto split*: *if-split-asm*)

lemma *remove1-empty*:
$(remove1\ v\ xs = []) = (xs = [v] \lor xs = [])$
by (*cases xs*; *simp*)

lemma *set-remove1*:
$x \in set\ (remove1\ y\ xs) \Longrightarrow x \in set\ xs$
by (*induct xs*) (*auto split*: *if-split-asm*)

lemma *If-rearrage*:
$(if\ P\ then\ if\ Q\ then\ x\ else\ y\ else\ z) = (if\ P \land Q\ then\ x\ else\ if\ P\ then\ y\ else\ z)$
by *simp*

lemma *disjI2-strg*:
$Q \longrightarrow (P \lor Q)$
by *simp*

lemma *eq-imp-strg*:
$P\ t \longrightarrow (t = s \longrightarrow P\ s)$
by *clarsimp*

lemma *if-both-strengthen*:

$P \wedge Q \longrightarrow (\mathit{if}\ G\ \mathit{then}\ P\ \mathit{else}\ Q)$
by *simp*

lemma *if-both-strengthen2*:
$P\ s \wedge Q\ s \longrightarrow (\mathit{if}\ G\ \mathit{then}\ P\ \mathit{else}\ Q)\ s$
by *simp*

lemma *if-swap*:
$(\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R) = (\mathit{if}\ \neg P\ \mathit{then}\ R\ \mathit{else}\ Q)$ **by** *simp*

lemma *imp-consequent*:
$P \longrightarrow Q \longrightarrow P$ **by** *simp*

lemma *list-case-helper*:
$xs \neq [] \Longrightarrow \mathit{case\text{-}list}\ f\ g\ xs = g\ (\mathit{hd}\ xs)\ (\mathit{tl}\ xs)$
by (*cases xs*, *simp-all*)

lemma *list-cons-rewrite*:
$(\forall x\ xs.\ L = x \# xs \longrightarrow P\ x\ xs) = (L \neq [] \longrightarrow P\ (\mathit{hd}\ L)\ (\mathit{tl}\ L))$
by (*auto simp*: *neq-Nil-conv*)

lemma *list-not-Nil-manip*:
$[\![\ xs = y \# ys;\ \mathit{case}\ xs\ \mathit{of}\ [] \Rightarrow \mathit{False} \mid (y \# ys) \Rightarrow P\ y\ ys\ ]\!] \Longrightarrow P\ y\ ys$
by *simp*

lemma *ran-ball-triv*:
$\bigwedge P\ m\ S.\ [\![\ \forall x \in (\mathit{ran}\ S).\ P\ x\ ;\ m \in (\mathit{ran}\ S)\ ]\!] \Longrightarrow P\ m$
by *blast*

lemma *singleton-tuple-cartesian*:
$(\{(a,\ b)\} = S \times T) = (\{a\} = S \wedge \{b\} = T)$
$(S \times T = \{(a,\ b)\}) = (\{a\} = S \wedge \{b\} = T)$
by *blast+*

lemma *strengthen-ignore-if*:
$A\ s \wedge B\ s \longrightarrow (\mathit{if}\ P\ \mathit{then}\ A\ \mathit{else}\ B)\ s$
by *clarsimp*

lemma *case-sum-True* :
$(\mathit{case}\ r\ \mathit{of}\ \mathit{Inl}\ a \Rightarrow \mathit{True} \mid \mathit{Inr}\ b \Rightarrow f\ b) = (\forall b.\ r = \mathit{Inr}\ b \longrightarrow f\ b)$
by (*cases r*) *auto*

lemma *sym-ex-elim*:
$F\ x = y \Longrightarrow \exists x.\ y = F\ x$
by *auto*

lemma *tl-drop-1* :
$\mathit{tl}\ xs = \mathit{drop}\ 1\ xs$
by (*simp add*: *drop-Suc*)

lemma *upt-lhs-sub-map*:
  $[x\ ..< y] = map\ ((+)\ x)\ [0\ ..< y - x]$
  by (*induct y*) (*auto simp*: *Suc-diff-le*)

lemma *upto-0-to-4*:
  $[0..<4] = 0\ \#\ [1..<4]$
  by (*subst upt-rec*) *simp*

lemma *disjEI*:
  $[\![\ P \vee Q;\ P \Longrightarrow R;\ Q \Longrightarrow S\ ]\!]$
    $\Longrightarrow R \vee S$
  by *fastforce*

lemma *dom-fun-upd2*:
  $s\ x = Some\ z \Longrightarrow dom\ (s\ (x \mapsto y)) = dom\ s$
  by (*simp add*: *insert-absorb domI*)

lemma *foldl-True* :
  $foldl\ (\vee)\ True\ bs$
  by (*induct bs*) *auto*

lemma *image-set-comp*:
  $f\ `\ \{g\ x \mid x.\ Q\ x\} = (f \circ g)\ `\ \{x.\ Q\ x\}$
  by *fastforce*

lemma *mutual-exE*:
  $[\![\ \exists x.\ P\ x;\ \bigwedge x.\ P\ x \Longrightarrow Q\ x\ ]\!] \Longrightarrow \exists x.\ Q\ x$
  by *blast*

lemma *nat-diff-eq*:
  fixes $x :: nat$
  shows $[\![\ x - y = x - z;\ y < x]\!] \Longrightarrow y = z$
  by *arith*

lemma *comp-upd-simp*:
  $(f \circ (g\ (x := y))) = ((f \circ g)\ (x := f\ y))$
  by (*rule fun-upd-comp*)

lemma *dom-option-map*:
  $dom\ (map\text{-}option\ f\ o\ m) = dom\ m$
  by (*rule dom-map-option-comp*)

lemma *drop-imp*:
  $P \Longrightarrow (A \longrightarrow P) \wedge (B \longrightarrow P)$ **by** *blast*

lemma *inj-on-fun-updI2*:
  $[\![\ inj\text{-}on\ f\ A;\ y \notin f\ `\ (A - \{x\})\ ]\!] \Longrightarrow inj\text{-}on\ (f(x := y))\ A$
  by (*rule inj-on-fun-upd-strongerI*)

lemma *inj-on-fun-upd-elsewhere*:
$x \notin S \Longrightarrow$ *inj-on* $(f\ (x := y))\ S =$ *inj-on* $f\ S$
by (*simp add*: *inj-on-def*) *blast*

lemma *not-Some-eq-tuple*:
$(\forall\, y\ z.\ x \neq \mathit{Some}\ (y,\ z)) = (x = \mathit{None})$
by (*cases* x, *simp-all*)

lemma *ran-option-map*:
$\mathit{ran}\ (\mathit{map\text{-}option}\ f\ o\ m) = f\ `\ \mathit{ran}\ m$
by (*auto simp add*: *ran-def*)

lemma *All-less-Ball*:
$(\forall\, x < n.\ P\ x) = (\forall\, x {\in} \{..< n\}.\ P\ x)$
by *fastforce*

lemma *Int-image-empty*:
$[\![\bigwedge x\ y.\ f\ x \neq g\ y]\!]$
$\Longrightarrow f\ `\ S \cap g\ `\ T = \{\}$
by *auto*

lemma *Max-prop*:
$[\![\mathit{Max}\ S \in S \Longrightarrow P\ (\mathit{Max}\ S);\ (S :: ('a :: \{\mathit{finite},\ \mathit{linorder}\})\ \mathit{set}) \neq \{\}]\!] \Longrightarrow P$
$(\mathit{Max}\ S)$
by *auto*

lemma *Min-prop*:
$[\![\mathit{Min}\ S \in S \Longrightarrow P\ (\mathit{Min}\ S);\ (S :: ('a :: \{\mathit{finite},\ \mathit{linorder}\})\ \mathit{set}) \neq \{\}]\!] \Longrightarrow P$
$(\mathit{Min}\ S)$
by *auto*

lemma *findSomeD*:
$\mathit{find}\ P\ \mathit{xs} = \mathit{Some}\ x \Longrightarrow P\ x \wedge x \in \mathit{set}\ \mathit{xs}$
by (*induct xs*) (*auto split*: *if-split-asm*)

lemma *findNoneD*:
$\mathit{find}\ P\ \mathit{xs} = \mathit{None} \Longrightarrow \forall\, x \in \mathit{set}\ \mathit{xs}.\ \neg P\ x$
by (*induct xs*) (*auto split*: *if-split-asm*)

lemma *dom-upd*:
$\mathit{dom}\ (\lambda x.\ \mathit{if}\ x = y\ \mathit{then}\ \mathit{None}\ \mathit{else}\ f\ x) = \mathit{dom}\ f - \{y\}$
by (*rule set-eqI*) (*auto split*: *if-split-asm*)

definition
is-inv $:: ('a \rightharpoonup 'b) \Rightarrow ('b \rightharpoonup 'a) \Rightarrow \mathit{bool}$ **where**
is-inv $f\ g \equiv \mathit{ran}\ f = \mathit{dom}\ g \wedge (\forall\, x\ y.\ f\ x = \mathit{Some}\ y \longrightarrow g\ y = \mathit{Some}\ x)$

lemma *is-inv-NoneD*:
  assumes *g x = None*
  assumes *is-inv f g*
  shows $x \notin$ *ran f*
proof −
  from *assms*
  have $x \notin$ *dom g* **by** (*auto simp*: *ran-def*)
  moreover
  from *assms*
  have *ran f = dom g*
    by (*simp add*: *is-inv-def*)
  ultimately
  show *?thesis* **by** *simp*
qed

lemma *is-inv-SomeD*:
  ⟦ *f x = Some y*; *is-inv f g* ⟧ ⟹ *g y = Some x*
  by (*simp add*: *is-inv-def*)

lemma *is-inv-com*:
  is-inv f g ⟹ *is-inv g f*
  apply (*unfold is-inv-def*)
  apply *safe*
    apply (*clarsimp simp*: *ran-def dom-def set-eq-iff*)
    apply (*erule-tac x=a* **in** *allE*)
    apply *clarsimp*
   apply (*clarsimp simp*: *ran-def dom-def set-eq-iff*)
   apply *blast*
  apply (*clarsimp simp*: *ran-def dom-def set-eq-iff*)
  apply (*erule-tac x=x* **in** *allE*)
  apply *clarsimp*
  done

lemma *is-inv-inj*:
  is-inv f g ⟹ *inj-on f* (*dom f*)
  apply (*frule is-inv-com*)
  apply (*clarsimp simp*: *inj-on-def*)
  apply (*drule* (*1*) *is-inv-SomeD*)
  apply (*auto dest*: *is-inv-SomeD*)
  done

lemma *ran-upd'*:
  ⟦*inj-on f* (*dom f*); *f y = Some z*⟧ ⟹ *ran* (*f* (*y* := *None*)) = *ran f* − {*z*}
  by (*force simp*: *ran-def inj-on-def dom-def intro*!: *set-eqI*)

lemma *is-inv-None-upd*:
  ⟦ *is-inv f g*; *g x = Some y*⟧ ⟹ *is-inv* (*f*(*y* := *None*)) (*g*(*x* := *None*))
  apply (*subst is-inv-def*)
  apply (*clarsimp simp*: *dom-upd*)

apply (*drule is-inv-SomeD*, *erule is-inv-com*)
apply (*frule is-inv-inj*)
apply (*auto simp*: *ran-upd'* *is-inv-def dest*: *is-inv-SomeD is-inv-inj*)
done

lemma *is-inv-inj2*:
is-inv f g $\Longrightarrow$ *inj-on g* (*dom g*)
using *is-inv-com is-inv-inj* **by** *blast*

lemma *range-convergence1*:
$\llbracket \forall z.\ x < z \wedge z \leq y \longrightarrow P\ z; \forall z > y.\ P\ (z :: 'a :: linorder) \rrbracket \Longrightarrow \forall z > x.\ P\ z$
using *not-le* **by** *blast*

lemma *range-convergence2*:
$\llbracket \forall z.\ x < z \wedge z \leq y \longrightarrow P\ z; \forall z.\ z > y \wedge z < w \longrightarrow P\ (z :: 'a :: linorder) \rrbracket$
$\Longrightarrow \forall z.\ z > x \wedge z < w \longrightarrow P\ z$
using *range-convergence1*[**where** $P{=}\lambda z.\ z < w \longrightarrow P\ z$ **and** $x{=}x$ **and** $y{=}y$]
by *auto*

lemma *zip-upt-Cons*:
$a < b \Longrightarrow zip\ [a\ ..{<}\ b]\ (x\ \#\ xs) = (a,\ x)\ \#\ zip\ [Suc\ a\ ..{<}\ b]\ xs$
by (*simp add*: *upt-conv-Cons*)

lemma *map-comp-eq*:
$f \circ_m g = case\text{-}option\ None\ f \circ g$
apply (*rule ext*)
apply (*case-tac g x*)
by *auto*

lemma *dom-If-Some*:
$dom\ (\lambda x.\ if\ x \in S\ then\ Some\ v\ else\ f\ x) = (S \cup dom\ f)$
by (*auto split*: *if-split*)

lemma *foldl-fun-upd-const*:
$foldl\ (\lambda s\ x.\ s(f\ x := v))\ s\ xs$
$= (\lambda x.\ if\ x \in f\ `\ set\ xs\ then\ v\ else\ s\ x)$
by (*induct xs arbitrary*: *s*) *auto*

lemma *foldl-id*:
$foldl\ (\lambda s\ x.\ s)\ s\ xs = s$
by (*induct xs*) *auto*

lemma *SucSucMinus*: $2 \leq n \Longrightarrow Suc\ (Suc\ (n - 2)) = n$ **by** *arith*

lemma *ball-to-all*:
$(\bigwedge x.\ (x \in A) = (P\ x)) \Longrightarrow (\forall x \in A.\ B\ x) = (\forall x.\ P\ x \longrightarrow B\ x)$
by *blast*

lemma *case-option-If*:

case-option P $(\lambda x.\ Q)$ v = (*if* v = *None then* P *else* Q)
by *clarsimp*

lemma *case-option-If2*:
case-option P Q v = *If* ($v \neq$ *None*) (Q (*the* v)) P
by (*simp split*: *option.split*)

lemma *if3-fold*:
(*if* P *then* x *else if* Q *then* y *else* x) = (*if* $P \vee \neg\ Q$ *then* x *else* y)
by *simp*

lemma *rtrancl-insert*:
assumes *x-new*: $\bigwedge y.\ (x,y) \notin R$
shows $R\hat{}*$ `` *insert* $x$ $S$ = *insert* $x$ ($R\hat{}*$ `` S)
proof –
have $R\hat{}*$ `` *insert* $x$ $S$ = $R\hat{}*$ `` ($\{x\} \cup S$) **by** *simp*
also
have $R\hat{}*$ `` ($\{x\} \cup S$) = $R\hat{}*$ `` $\{x\} \cup R\hat{}*$ `` S
by (*subst Image-Un*) *simp*
also
have $R\hat{}*$ `` $\{x\} = \{x\}$
by (*meson Image-closed-trancl Image-singleton-iff subsetI x-new*)
finally
show *?thesis* **by** *simp*
qed

lemma *ran-del-subset*:
$y \in$ *ran* (f (x := *None*)) $\Longrightarrow y \in$ *ran* f
by (*auto simp*: *ran-def split*: *if-split-asm*)

lemma *trancl-sub-lift*:
assumes *sub*: $\bigwedge p\ p'.\ (p,p') \in r \Longrightarrow (p,p') \in r'$
shows $(p,p') \in r\hat{}+ \Longrightarrow (p,p') \in r'\hat{}+$
by (*fastforce intro*: *trancl-mono sub*)

lemma *trancl-step-lift*:
assumes *x-step*: $\bigwedge p\ p'.\ (p,p') \in r' \Longrightarrow (p,p') \in r \vee (p = x \wedge p' = y)$
assumes *y-new*: $\bigwedge p'.\ \neg(y,p') \in r$
shows $(p,p') \in r'\hat{}+ \Longrightarrow (p,p') \in r\hat{}+ \vee ((p,x) \in r\hat{}+ \wedge p' = y) \vee (p = x \wedge p'$
$= y)$
apply (*erule trancl-induct*)
apply (*drule x-step*)
apply *fastforce*
apply (*erule disjE*)
apply (*drule x-step*)
apply (*erule disjE*)
apply (*drule trancl-trans*, *drule r-into-trancl*, *assumption*)
apply *blast*
apply *fastforce*

apply (*fastforce simp*: *y-new dest*: *x-step*)
done

lemma *rtrancl-simulate-weak*:
assumes *r*: $(x,z) \in R^*$
assumes *s*: $\bigwedge y.\ (x,y) \in R \Longrightarrow (y,z) \in R^* \Longrightarrow (x,y) \in R' \wedge (y,z) \in R'^*$
shows $(x,z) \in R'^*$
apply (*rule converse-rtranclE*[*OF r*])
apply *simp*
apply (*frule* (*1*) *s*)
apply *clarsimp*
by (*rule converse-rtrancl-into-rtrancl*)

lemma *list-case-If2*:
case-list f g xs = *If* (*xs* = []) *f* (*g* (*hd xs*) (*tl xs*))
by (*simp split*: *list.split*)

lemma *length-ineq-not-Nil*:
$\mathit{length}\ xs > n \Longrightarrow xs \neq []$
$\mathit{length}\ xs \geq n \Longrightarrow n \neq 0 \longrightarrow xs \neq []$
$\neg\ \mathit{length}\ xs < n \Longrightarrow n \neq 0 \longrightarrow xs \neq []$
$\neg\ \mathit{length}\ xs \leq n \Longrightarrow xs \neq []$
by *auto*

lemma *numeral-eqs*:
2 = *Suc* (*Suc 0*)
3 = *Suc* (*Suc* (*Suc 0*))
4 = *Suc* (*Suc* (*Suc* (*Suc 0*)))
5 = *Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*))))
6 = *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*)))))
by *simp+*

lemma *psubset-singleton*:
$(S \subset \{x\}) = (S = \{\})$
by *blast*

lemma *length-takeWhile-ge*:
length (*takeWhile f xs*) = *n* $\Longrightarrow$ *length xs* = *n* $\vee$ (*length xs* > *n* $\wedge$ $\neg$ *f* (*xs* ! *n*))
by (*induct xs arbitrary*: *n*) (*auto split*: *if-split-asm*)

lemma *length-takeWhile-le*:
$\neg$ *f* (*xs* ! *n*) $\Longrightarrow$ *length* (*takeWhile f xs*) $\leq$ *n*
by (*induct xs arbitrary*: *n*; *simp*) (*case-tac n*; *simp*)

lemma *length-takeWhile-gt*:
n < *length* (*takeWhile f xs*)
$\Longrightarrow$ ($\exists$ *ys zs*. *length ys* = *Suc n* $\wedge$ *xs* = *ys* @ *zs* $\wedge$ *takeWhile f xs* = *ys* @ *takeWhile f zs*)
apply (*induct xs arbitrary*: *n*; *simp split*: *if-split-asm*)

apply (*case-tac n*; *simp*)
apply (*rule-tac x*=[*a*] **in** *exI*)
apply *simp*
apply (*erule meta-allE*, *drule*(*1*) *meta-mp*)
apply *clarsimp*
apply (*rule-tac x*=*a* # *ys* **in** *exI*)
apply *simp*
done

lemma *hd-drop-conv-nth2*:
$n < \mathit{length}\ xs \Longrightarrow \mathit{hd}\ (\mathit{drop}\ n\ xs) = xs\ !\ n$
by (*rule hd-drop-conv-nth*) *clarsimp*

lemma *map-upt-eq-vals-D*:
$[\![\ \mathit{map}\ f\ [0\ ..<\ n] = ys;\ m < \mathit{length}\ ys\ ]\!] \Longrightarrow f\ m = ys\ !\ m$
by *clarsimp*

lemma *length-le-helper*:
$[\![\ n \leq \mathit{length}\ xs;\ n \neq 0\ ]\!] \Longrightarrow xs \neq [] \wedge n - 1 \leq \mathit{length}\ (\mathit{tl}\ xs)$
by (*cases xs*, *simp-all*)

lemma *all-ex-eq-helper*:
$(\forall\, v.\ (\exists\, v'.\ v = f\ v' \wedge P\ v\ v') \longrightarrow Q\ v)$
$= (\forall\, v'.\ P\ (f\ v')\ v' \longrightarrow Q\ (f\ v'))$
by *auto*

lemma *nat-less-cases'*:
$(x{::}\mathit{nat}) < y \Longrightarrow x = y - 1 \vee x < y - 1$
by *auto*

lemma *filter-to-shorter-upto*:
$n \leq m \Longrightarrow \mathit{filter}\ (\lambda x.\ x < n)\ [0\ ..<\ m] = [0\ ..<\ n]$
by (*induct m*) (*auto elim*: *le-SucE*)

lemma *in-emptyE*: $[\![\ A = \{\};\ x \in A\ ]\!] \Longrightarrow P$ **by** *blast*

lemma *Ball-emptyI*:
$S = \{\} \Longrightarrow (\forall\, x \in S.\ P\ x)$
by *simp*

lemma *allfEI*:
$[\![\ \forall\, x.\ P\ x;\ \bigwedge x.\ P\ (f\ x) \Longrightarrow Q\ x\ ]\!] \Longrightarrow \forall\, x.\ Q\ x$
by *fastforce*

lemma *cart-singleton-empty2*:
$(\{x\} \times S = \{\}) = (S = \{\})$
$(\{\} = S \times \{e\}) = (S = \{\})$
by *auto*

lemma *cases-simp-conj*:
$((P \longrightarrow Q) \wedge (\neg\ P \longrightarrow Q) \wedge R) = (Q \wedge R)$
by *fastforce*

lemma *domE* :
$[\![\ x \in \mathit{dom}\ m;\ \bigwedge r.\ [\![m\ x = \mathit{Some}\ r]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
by *clarsimp*

lemma *dom-eqD*:
$[\![\ f\ x = \mathit{Some}\ v;\ \mathit{dom}\ f = S\ ]\!] \Longrightarrow x \in S$
by *clarsimp*

lemma *exception-set-finite-1*:
$\mathit{finite}\ \{x.\ P\ x\} \Longrightarrow \mathit{finite}\ \{x.\ (x = y \longrightarrow Q\ x) \wedge P\ x\}$
by (*simp add*: *Collect-conj-eq*)

lemma *exception-set-finite-2*:
$\mathit{finite}\ \{x.\ P\ x\} \Longrightarrow \mathit{finite}\ \{x.\ x \neq y \longrightarrow P\ x\}$
by (*simp add*: *imp-conv-disj*)

lemmas *exception-set-finite* = *exception-set-finite-1 exception-set-finite-2*

lemma *exfEI*:
$[\![\ \exists x.\ P\ x;\ \bigwedge x.\ P\ x \Longrightarrow Q\ (f\ x)\ ]\!] \Longrightarrow \exists x.\ Q\ x$
by *fastforce*

lemma *Collect-int-vars*:
$\{s.\ P\ \mathit{rv}\ s\} \cap \{s.\ \mathit{rv} = \mathit{xf}\ s\} = \{s.\ P\ (\mathit{xf}\ s)\ s\} \cap \{s.\ \mathit{rv} = \mathit{xf}\ s\}$
by *auto*

lemma *if-0-1-eq*:
$((\mathit{if}\ P\ \mathit{then}\ 1\ \mathit{else}\ 0) = (\mathit{case}\ Q\ \mathit{of}\ \mathit{True} \Rightarrow \mathit{of\text{-}nat}\ 1 \mid \mathit{False} \Rightarrow \mathit{of\text{-}nat}\ 0)) = (P = Q)$
by (*simp split*: *if-split bool.split*)

lemma *modify-map-exists-cte* :
$(\exists\,\mathit{cte}.\ \mathit{modify\text{-}map}\ m\ p\ f\ p' = \mathit{Some}\ \mathit{cte}) = (\exists\,\mathit{cte}.\ m\ p' = \mathit{Some}\ \mathit{cte})$
by (*simp add*: *modify-map-def*)

lemma *dom-eqI*:
assumes *c1*: $\bigwedge x\ y.\ P\ x = \mathit{Some}\ y \Longrightarrow \exists y.\ Q\ x = \mathit{Some}\ y$
and *c2*: $\bigwedge x\ y.\ Q\ x = \mathit{Some}\ y \Longrightarrow \exists y.\ P\ x = \mathit{Some}\ y$
shows $\mathit{dom}\ P = \mathit{dom}\ Q$
unfolding *dom-def* **by** (*auto simp*: *c1 c2*)

lemma *dvd-reduce-multiple*:
fixes $k :: \mathit{nat}$
shows $(k\ \mathit{dvd}\ k * m + n) = (k\ \mathit{dvd}\ n)$
by (*induct m*) (*auto simp*: *add-ac*)

lemma *image-iff2*:
  $inj\ f \Longrightarrow f\ x \in f\ `\ S = (x \in S)$
  by (*rule inj-image-mem-iff*)

lemma *map-comp-restrict-map-Some-iff*:
  $((g \circ_m (m\ |`\ S))\ x = Some\ y) = ((g \circ_m m)\ x = Some\ y \wedge x \in S)$
  by (*auto simp add*: *map-comp-Some-iff restrict-map-Some-iff*)

lemma *range-subsetD*:
  fixes $a :: 'a :: order$
  shows $[\![\ \{a..b\} \subseteq \{c..d\};\ a \leq b\ ]\!] \Longrightarrow c \leq a \wedge b \leq d$
  by *simp*

lemma *case-option-dom*:
  $(case\ f\ x\ of\ None \Rightarrow a \mid Some\ v \Rightarrow b\ v) = (if\ x \in dom\ f\ then\ b\ (the\ (f\ x))\ else\ a)$
  by (*auto split*: *option.split*)

lemma *contrapos-imp*:
  $P \longrightarrow Q \Longrightarrow \neg\ Q \longrightarrow \neg\ P$
  by *clarsimp*

lemma *filter-eq-If*:
  $distinct\ xs \Longrightarrow filter\ (\lambda v.\ v = x)\ xs = (if\ x \in set\ xs\ then\ [x]\ else\ [])$
  by (*induct xs*) *auto*

lemma (**in** *semigroup-add*) *foldl-assoc*:
shows $foldl\ (+)\ (x+y)\ zs = x + (foldl\ (+)\ y\ zs)$
  by (*induct zs arbitrary*: *y*) (*simp-all add*:*add.assoc*)

lemma (**in** *monoid-add*) *foldl-absorb0*:
shows $x + (foldl\ (+)\ 0\ zs) = foldl\ (+)\ x\ zs$
  by (*induct zs*) (*simp-all add*:*foldl-assoc*)

lemma *foldl-conv-concat*:
  $foldl\ (@)\ xs\ xss = xs\ @\ concat\ xss$
proof (*induct xss arbitrary*: *xs*)
  case *Nil* **show** *?case* **by** *simp*
next
  interpret *monoid-add* (@) [] **proof qed** *simp-all*
  case *Cons* **then show** *?case* **by** (*simp add*: *foldl-absorb0*)
qed

lemma *foldl-concat-concat*:
  $foldl\ (@)\ []\ (xs\ @\ ys) = foldl\ (@)\ []\ xs\ @\ foldl\ (@)\ []\ ys$
  by (*simp add*: *foldl-conv-concat*)

lemma *foldl-does-nothing*:
  $[\![\ \bigwedge x.\ x \in set\ xs \Longrightarrow f\ s\ x = s\ ]\!] \Longrightarrow foldl\ f\ s\ xs = s$

by (*induct xs*) *auto*

lemma *foldl-use-filter*:
$[\![\bigwedge v\ x.\ [\![\neg\ g\ x;\ x \in set\ xs\ ]\!] \Longrightarrow f\ v\ x = v\ ]\!] \Longrightarrow foldl\ f\ v\ xs = foldl\ f\ v\ (filter\ g\ xs)$
by (*induct xs arbitrary*: *v*) *auto*

lemma *map-comp-update-lift*:
assumes *fv*: $f\ v = Some\ v'$
shows $(f \circ_m (g(ptr \mapsto v))) = ((f \circ_m g)(ptr \mapsto v'))$
by (*simp add*: *fv map-comp-update*)

lemma *restrict-map-cong*:
assumes *sv*: $S = S'$
and *rl*: $\bigwedge p.\ p \in S' \Longrightarrow mp\ p = mp'\ p$
shows $mp\ |`\ S = mp'\ |`\ S'$
using *expand-restrict-map-eq rl sv* **by** *auto*

lemma *case-option-over-if*:
case-option P Q (*if G then None else Some v*)
= (*if G then P else Q v*)
case-option P Q (*if G then Some v else None*)
= (*if G then Q v else P*)
by (*simp split*: *if-split*)+

lemma *map-length-cong*:
$[\![length\ xs = length\ ys;\ \bigwedge x\ y.\ (x,\ y) \in set\ (zip\ xs\ ys) \Longrightarrow f\ x = g\ y\ ]\!]$
$\Longrightarrow map\ f\ xs = map\ g\ ys$
apply *atomize*
apply (*erule rev-mp*, *erule list-induct2*)
apply *auto*
done

lemma *take-min-len*:
take (*min* (*length xs*) *n*) *xs* = *take n xs*
by (*simp add*: *min-def*)

lemmas *interval-empty* = *atLeastatMost-empty-iff*

lemma *fold-and-false*[*simp*]:
$\neg$(*fold* ($\wedge$) *xs False*)
apply *clarsimp*
apply (*induct xs*)
apply *simp*
apply *simp*
done

lemma *fold-and-true*:
$fold\ (\wedge)\ xs\ True \Longrightarrow \forall\, i < length\ xs.\ xs\ !\ i$

apply *clarsimp*
apply (*induct xs*)
apply *simp*
apply (*case-tac* $i = 0$; *simp*)
apply (*case-tac a*; *simp*)
apply (*case-tac a*; *simp*)
done

lemma *fold-or-true*[*simp*]:
fold (∨) *xs True*
by (*induct xs*, *simp*+)

lemma *fold-or-false*:
¬(*fold* (∨) *xs False*) ⟹ ∀ *i* < *length xs*. ¬(*xs* ! *i*)
apply (*induct xs*, *simp*+)
apply (*case-tac a*, *simp*+)
apply (*rule allI*, *case-tac* $i = 0$, *simp*+)
done

16 Take, drop, zip, list$_{a}lletcrules$

method *two-induct* **for** *xs ys* =
((*induct xs arbitrary*: *ys*; *simp?*), (*case-tac ys*; *simp*)*?*)

lemma *map-fst-zip-prefix*:
map fst (*zip xs ys*) ≤ *xs*
by (*two-induct xs ys*)

lemma *map-snd-zip-prefix*:
map snd (*zip xs ys*) ≤ *ys*
by (*two-induct xs ys*)

lemma *nth-upt-0* [*simp*]:
i < *length xs* ⟹ [*0*..<*length xs*] ! *i* = *i*
by *simp*

lemma *take-insert-nth*:
i < *length xs*⟹ *insert* (*xs* ! *i*) (*set* (*take i xs*)) = *set* (*take* (*Suc i*) *xs*)
by (*subst take-Suc-conv-app-nth*, *assumption*, *fastforce*)

lemma *zip-take-drop*:
⟦*n* < *length xs*; *length ys* = *length xs*⟧ ⟹
zip xs (*take n ys* @ *a* # *drop* (*Suc n*) *ys*) =
zip (*take n xs*) (*take n ys*) @ (*xs* ! *n*, *a*) # *zip* (*drop* (*Suc n*) *xs*) (*drop* (*Suc n*) *ys*)
by (*subst id-take-nth-drop*, *assumption*, *simp*)

lemma *take-nth-distinct*:
⟦*distinct xs*; *n* < *length xs*; *xs* ! *n* ∈ *set* (*take n xs*)⟧ ⟹ *False*

by (*fastforce simp*: *distinct-conv-nth in-set-conv-nth*)

lemma *take-drop-append*:
drop a xs = *take b* (*drop a xs*) @ *drop* (*a* + *b*) *xs*
by (*metis append-take-drop-id drop-drop add.commute*)

lemma *drop-take-drop*:
drop a (*take* (*b* + *a*) *xs*) @ *drop* (*b* + *a*) *xs* = *drop a xs*
by (*metis add.commute take-drop take-drop-append*)

lemma *not-prefixI*:
$\llbracket$ *xs* $\neq$ *ys*; *length xs* = *length ys*$\rrbracket$ $\Longrightarrow$ $\neg$ *xs* $\leq$ *ys*
by (*auto elim*: *prefixE*)

lemma *map-fst-zip'*:
length xs $\leq$ *length ys* $\Longrightarrow$ *map fst* (*zip xs ys*) = *xs*
by (*metis length-map length-zip map-fst-zip-prefix min-absorb1 not-prefixI*)

lemma *zip-take-triv*:
n $\geq$ *length bs* $\Longrightarrow$ *zip* (*take n as*) *bs* = *zip as bs*
apply (*induct bs arbitrary*: *n as*; *simp*)
apply (*case-tac n*; *simp*)
apply (*case-tac as*; *simp*)
done

lemma *zip-take-triv2*:
length as $\leq$ *n* $\Longrightarrow$ *zip as* (*take n bs*) = *zip as bs*
apply (*induct as arbitrary*: *n bs*; *simp*)
apply (*case-tac n*; *simp*)
apply (*case-tac bs*; *simp*)
done

lemma *zip-take-length*:
zip xs (*take* (*length xs*) *ys*) = *zip xs ys*
by (*metis order-refl zip-take-triv2*)

lemma *zip-singleton*:
ys $\neq$ [] $\Longrightarrow$ *zip* [*a*] *ys* = [(*a*, *ys* ! *0*)]
by (*case-tac ys*, *simp-all*)

lemma *zip-append-singleton*:
$\llbracket$*i* = *length xs*; *length xs* < *length ys*$\rrbracket$ $\Longrightarrow$ *zip* (*xs* @ [*a*]) *ys* = (*zip xs ys*) @ [(*a*,*ys* ! *i*)]
by (*induct xs*; *case-tac ys*; *simp*)
(*clarsimp simp*: *zip-append1 zip-take-length zip-singleton*)

lemma *ran-map-of-zip*:
$\llbracket$*length xs* = *length ys*; *distinct xs*$\rrbracket$ $\Longrightarrow$ *ran* (*map-of* (*zip xs ys*)) = *set ys*
by (*induct rule*: *list-induct2*) *auto*

lemma *ranE*:
$[\![v \in ran\ f;\ \bigwedge x.\ f\ x = Some\ v \Longrightarrow R]\!] \Longrightarrow R$
by (*auto simp*: *ran-def*)

lemma *ran-map-option-restrict-eq*:
$[\![x \in ran\ (map\text{-}option\ f\ o\ g);\ x \notin ran\ (map\text{-}option\ f\ o\ (g\ |`\ (-\ \{y\})))\ ]\!]$
$\Longrightarrow \exists v.\ g\ y = Some\ v \wedge f\ v = x$
apply (*clarsimp simp*: *elim*!: *ranE*)
apply (*rename-tac w z*)
apply (*case-tac w = y*)
 apply *clarsimp*
apply (*erule notE*, *rule-tac a=w* **in** *ranI*)
apply (*simp add*: *restrict-map-def*)
done

lemma *map-of-zip-range*:
$[\![length\ xs = length\ ys;\ distinct\ xs]\!] \Longrightarrow (\lambda x.\ (the\ (map\text{-}of\ (zip\ xs\ ys)\ x)))\ `\ set\ xs = set\ ys$
apply (*clarsimp simp*: *image-def*)
apply (*subst ran-map-of-zip* [*symmetric*, **where** *xs=xs* **and** *ys=ys*]; *simp?*)
apply (*clarsimp simp*: *ran-def*)
apply (*rule equalityI*)
 apply *clarsimp*
 apply (*rename-tac x*)
 apply (*frule-tac x=x* **in** *map-of-zip-is-Some*; *fastforce*)
apply (*clarsimp simp*: *set-zip*)
by (*metis domI dom-map-of-zip nth-mem ranE ran-map-of-zip option.sel*)

lemma *map-zip-fst*:
$length\ xs = length\ ys \Longrightarrow map\ (\lambda(x,\ y).\ f\ x)\ (zip\ xs\ ys) = map\ f\ xs$
by (*two-induct xs ys*)

lemma *map-zip-fst'*:
$length\ xs \le length\ ys \Longrightarrow map\ (\lambda(x,\ y).\ f\ x)\ (zip\ xs\ ys) = map\ f\ xs$
by (*metis length-map map-fst-zip' map-zip-fst zip-map-fst-snd*)

lemma *map-zip-snd*:
$length\ xs = length\ ys \Longrightarrow map\ (\lambda(x,\ y).\ f\ y)\ (zip\ xs\ ys) = map\ f\ ys$
by (*two-induct xs ys*)

lemma *map-zip-snd'*:
$length\ ys \le length\ xs \Longrightarrow map\ (\lambda(x,\ y).\ f\ y)\ (zip\ xs\ ys) = map\ f\ ys$
by (*two-induct xs ys*)

lemma *map-of-zip-tuple-in*:
$[\![(x,\ y) \in set\ (zip\ xs\ ys);\ distinct\ xs]\!] \Longrightarrow map\text{-}of\ (zip\ xs\ ys)\ x = Some\ y$
by (*two-induct xs ys*) (*auto intro*: *in-set-zipE*)

lemma *in-set-zip1*:
$(x, y) \in set\ (zip\ xs\ ys) \Longrightarrow x \in set\ xs$
by (*erule in-set-zipE*)

lemma *in-set-zip2*:
$(x, y) \in set\ (zip\ xs\ ys) \Longrightarrow y \in set\ ys$
by (*erule in-set-zipE*)

lemma *map-zip-snd-take*:
$map\ (\lambda(x, y).\ f\ y)\ (zip\ xs\ ys) = map\ f\ (take\ (length\ xs)\ ys)$
apply (*subst map-zip-snd′* [*symmetric*, **where** *xs=xs* **and** *ys=take* (*length xs*) *ys*], *simp*)
apply (*subst zip-take-length* [*symmetric*], *simp*)
done

lemma *map-of-zip-is-index*:
$[\![length\ xs = length\ ys; x \in set\ xs]\!] \Longrightarrow \exists\, i.\ (map\text{-}of\ (zip\ xs\ ys))\ x = Some\ (ys\ !\ i)$
apply (*induct rule*: *list-induct2*; *simp*)
apply (*rule conjI*; *clarsimp*)
apply (*metis nth-Cons-0*)
apply (*metis nth-Cons-Suc*)
done

lemma *map-of-zip-take-update*:
$[\![i < length\ xs; length\ xs \le length\ ys; distinct\ xs]\!]$
$\Longrightarrow map\text{-}of\ (zip\ (take\ i\ xs)\ ys)(xs\ !\ i \mapsto (ys\ !\ i)) = map\text{-}of\ (zip\ (take\ (Suc\ i)\ xs)\ ys)$
apply (*rule ext*, *rename-tac x*)
apply (*case-tac x=xs ! i*; *clarsimp*)
apply (*rule map-of-is-SomeI*[*symmetric*])
apply (*simp add*: *map-fst-zip′*)
apply (*force simp add*: *set-zip*)
apply (*clarsimp simp*: *take-Suc-conv-app-nth zip-append-singleton map-add-def split*: *option.splits*)
done

lemma *map-of-zip-is-Some′*:
$length\ xs \le length\ ys \Longrightarrow (x \in set\ xs) = (\exists\, y.\ map\text{-}of\ (zip\ xs\ ys)\ x = Some\ y)$
apply (*subst zip-take-length*[*symmetric*])
apply (*rule map-of-zip-is-Some*)
by (*metis length-take min-absorb2*)

lemma *map-of-zip-inj*:
$[\![distinct\ xs; distinct\ ys; length\ xs = length\ ys]\!]$
$\Longrightarrow inj\text{-}on\ (\lambda x.\ (the\ (map\text{-}of\ (zip\ xs\ ys)\ x)))\ (set\ xs)$
apply (*clarsimp simp*: *inj-on-def*)
apply (*subst* (*asm*) *map-of-zip-is-Some*, *assumption*)+

apply *clarsimp*
apply (*clarsimp simp*: *set-zip*)
by (*metis nth-eq-iff-index-eq*)

lemma *map-of-zip-inj'*:
$[\![$*distinct xs*; *distinct ys*; *length xs* $\leq$ *length ys*$]\!]$
$\Longrightarrow$ *inj-on* (λx. (*the* (*map-of* (*zip xs ys*) *x*))) (*set xs*)
apply (*subst zip-take-length*[*symmetric*])
apply (*erule map-of-zip-inj*, *simp*)
by (*metis length-take min-absorb2*)

lemma *list-all-nth*:
$[\![$*list-all P xs*; $i <$ *length xs*$]\!] \Longrightarrow P$ (*xs* ! *i*)
by (*metis list-all-length*)

lemma *list-all-update*:
$[\![$*list-all P xs*; $i <$ *length xs*; $\bigwedge x.\ P\ x \Longrightarrow P\ (f\ x)]\!]$
$\Longrightarrow$ *list-all P* (*xs* [$i := f$ (*xs* ! *i*)])
by (*metis length-list-update list-all-length nth-list-update*)

lemma *list-allI*:
$[\![$*list-all P xs*; $\bigwedge x.\ P\ x \Longrightarrow P'\ x]\!] \Longrightarrow$ *list-all* P' *xs*
by (*metis list-all-length*)

lemma *list-all-imp-filter*:
list-all ($\lambda x.\ f\ x \longrightarrow g\ x$) *xs* = *list-all* ($\lambda x.\ g\ x$) [$x \leftarrow$*xs* . $f\ x$]
by (*fastforce simp*: *Ball-set-list-all*[*symmetric*])

lemma *list-all-imp-filter2*:
list-all ($\lambda x.\ f\ x \longrightarrow g\ x$) *xs* = *list-all* ($\lambda x.\ \neg f\ x$) [$x \leftarrow$*xs* . ($\lambda x.\ \neg g\ x$) x]
by (*fastforce simp*: *Ball-set-list-all*[*symmetric*])

lemma *list-all-imp-chain*:
$[\![$*list-all* ($\lambda x.\ f\ x \longrightarrow g\ x$) *xs*; *list-all* ($\lambda x.\ f'\ x \longrightarrow f\ x$) *xs*$]\!]$
$\Longrightarrow$ *list-all* ($\lambda x.\ f'\ x \longrightarrow g\ x$) *xs*
by (*clarsimp simp*: *Ball-set-list-all* [*symmetric*])

lemma *inj-Pair*:
inj-on (*Pair x*) *S*
by (*rule inj-onI*, *simp*)

lemma *inj-on-split*:
inj-on f S $\Longrightarrow$ *inj-on* (λx. (z, $f\ x$)) *S*
by (*auto simp*: *inj-on-def*)

lemma *split-state-strg*:
$(\exists x.\ f\ s = x \wedge P\ x\ s) \longrightarrow P\ (f\ s)\ s$ **by** *clarsimp*

lemma *theD*:
$[\![the\ (f\ x) = y;\ \ x \in dom\ f\ ]\!] \Longrightarrow f\ x = Some\ y$
by (*auto simp add*: *dom-def*)

lemma *bspec-split*:
$[\![\ \forall (a,\ b) \in S.\ P\ a\ b;\ (a,\ b) \in S\ ]\!] \Longrightarrow P\ a\ b$
by *fastforce*

lemma *set-zip-same*:
$set\ (zip\ xs\ xs) = Id \cap (set\ xs \times set\ xs)$
by (*induct xs*) *auto*

lemma *ball-ran-updI*:
$(\forall x \in ran\ m.\ P\ x) \Longrightarrow P\ v \Longrightarrow (\forall x \in ran\ (m\ (y \mapsto v)).\ P\ x)$
by (*auto simp add*: *ran-def*)

lemma *not-psubset-eq*:
$[\![\ \neg\ A \subset B;\ A \subseteq B\ ]\!] \Longrightarrow A = B$
by *blast*

lemma *in-image-op-plus*:
$(x + y \in (+)\ x\ `\ S) = ((y :: 'a :: ring) \in S)$
by (*simp add*: *image-def*)

lemma *insert-subtract-new*:
$x \notin S \Longrightarrow (insert\ x\ S - S) = \{x\}$
by *auto*

lemma *zip-is-empty*:
$(zip\ xs\ ys = []) = (xs = [] \vee ys = [])$
by (*cases xs*; *simp*) (*cases ys*; *simp*)

lemma *minus-Suc-0-lt*:
$a \neq 0 \Longrightarrow a - Suc\ 0 < a$
by *simp*

lemma *fst-last-zip-upt*:
$zip\ [0\ ..{<}\ m]\ xs \neq [] \Longrightarrow$
$fst\ (last\ (zip\ [0\ ..{<}\ m]\ xs)) = (if\ length\ xs < m\ then\ length\ xs - 1\ else\ m - 1)$
apply (*subst last-conv-nth*, *assumption*)
apply (*simp only*: *One-nat-def*)
apply (*subst nth-zip*)
apply (*rule order-less-le-trans*[*OF minus-Suc-0-lt*])
apply (*simp add*: *zip-is-empty*)
apply *simp*

```
   apply (rule order-less-le-trans[OF minus-Suc-0-lt])
    apply (simp add: zip-is-empty)
   apply simp
  apply (simp add: min-def zip-is-empty)
  done

lemma neq-into-nprefix:
  ⟦ x ≠ take (length x) y ⟧ ⟹ ¬ x ≤ y
  by (clarsimp simp: prefix-def less-eq-list-def)

lemma suffix-eqI:
  ⟦ suffix xs as; suffix xs bs; length as = length bs;
    take (length as − length xs) as ≤ take (length bs − length xs) bs⟧ ⟹ as = bs
  by (clarsimp elim!: prefixE suffixE)

lemma suffix-Cons-mem:
  suffix (x # xs) as ⟹ x ∈ set as
  by (metis in-set-conv-decomp suffix-def)

lemma distinct-imply-not-in-tail:
  ⟦ distinct list; suffix (y # ys) list⟧ ⟹ y ∉ set ys
  by (clarsimp simp:suffix-def)

lemma list-induct-suffix [case-names Nil Cons]:
  assumes nilr: P []
  and     consr: ⋀x xs. ⟦P xs; suffix (x # xs) as ⟧ ⟹ P (x # xs)
  shows  P as
proof −
  define as' where as' == as

  have suffix as as' unfolding as'-def by simp
  then show ?thesis
  proof (induct as)
    case Nil show ?case by fact
  next
    case (Cons x xs)

    show ?case
    proof (rule consr)
      from Cons.prems show suffix (x # xs) as unfolding as'-def .
      then have suffix xs as' by (auto dest: suffix-ConsD simp: as'-def)
      then show P xs using Cons.hyps by simp
    qed
  qed
qed
```

Parallel etc. and lemmas for list prefix

```
lemma prefix-induct [consumes 1, case-names Nil Cons]:
  fixes prefix
```

```
  assumes np: prefix ≤ lst
  and base:   ⋀xs. P [] xs
  and rl:     ⋀x xs y ys. ⟦ x = y; xs ≤ ys; P xs ys ⟧ ⟹ P (x#xs) (y#ys)
  shows P prefix lst
  using np
proof (induct prefix arbitrary: lst)
  case Nil show ?case by fact
next
  case (Cons x xs)

  have prem: (x # xs) ≤ lst by fact
  then obtain y ys where lv: lst = y # ys
    by (rule prefixE, auto)

  have ih: ⋀lst. xs ≤ lst ⟹ P xs lst by fact

  show ?case using prem
    by (auto simp: lv intro!: rl ih)
qed

lemma not-prefix-cases:
  fixes prefix
  assumes pfx: ¬ prefix ≤ lst
  and c1: ⟦ prefix ≠ []; lst = [] ⟧ ⟹ R
  and c2: ⋀a as x xs. ⟦ prefix = a#as; lst = x#xs; x = a; ¬ as ≤ xs⟧ ⟹ R
  and c3: ⋀a as x xs. ⟦ prefix = a#as; lst = x#xs; x ≠ a⟧ ⟹ R
  shows R
proof (cases prefix)
  case Nil then show ?thesis using pfx by simp
next
  case (Cons a as)

  have c: prefix = a#as by fact

  show ?thesis
  proof (cases lst)
    case Nil then show ?thesis
      by (intro c1, simp add: Cons)
  next
    case (Cons x xs)
    show ?thesis
    proof (cases x = a)
      case True
      show ?thesis
      proof (intro c2)
        show ¬ as ≤ xs using pfx c Cons True
          by simp
      qed fact+
    next
```

```
    case False
    show ?thesis by (rule c3) fact+
  qed
 qed
qed

lemma not-prefix-induct [consumes 1, case-names Nil Neq Eq]:
 fixes prefix
 assumes np: ¬ prefix ≤ lst
 and base:  ⋀x xs. P (x#xs) []
 and r1:    ⋀x xs y ys. x ≠ y ⟹ P (x#xs) (y#ys)
 and r2:    ⋀x xs y ys. ⟦ x = y; ¬ xs ≤ ys; P xs ys ⟧ ⟹ P (x#xs) (y#ys)
 shows P prefix lst
 using np
proof (induct lst arbitrary: prefix)
 case Nil then show ?case
  by (auto simp: neq-Nil-conv elim!: not-prefix-cases intro!: base)
next
 case (Cons y ys)

 have npfx: ¬ prefix ≤ (y # ys) by fact
 then obtain x xs where pv: prefix = x # xs
  by (rule not-prefix-cases) auto

 have ih: ⋀prefix. ¬ prefix ≤ ys ⟹ P prefix ys by fact

 show ?case using npfx
  by (simp only: pv) (erule not-prefix-cases, auto intro: r1 r2 ih)
qed

lemma rsubst:
 ⟦ P s; s = t ⟧ ⟹ P t
 by simp

lemma ex-impE: ((∃ x. P x) ⟶ Q) ⟹ P x ⟹ Q
 by blast

lemma option-Some-value-independent:
 ⟦ f x = Some v; ⋀v'. f x = Some v' ⟹ f y = Some v' ⟧ ⟹ f y = Some v
 by blast
```

Some int bitwise lemmas. Helpers for proofs about `NatBitwise.thy`

```
lemma int-2p-eq-shiftl:
 (2::int)^x = 1 << x
 by (simp add: shiftl-int-def)

lemma nat-int-mul:
 nat (int a * b) = a * nat b
 by (simp add: nat-mult-distrib)
```

lemma *int-shiftl-less-cancel*:
  $n \leq m \Longrightarrow ((x :: int) << n < y << m) = (x < y << (m - n))$
  apply (*drule le-Suc-ex*)
  apply (*clarsimp simp*: *shiftl-int-def power-add*)
  done

lemma *int-shiftl-lt-2p-bits*:
  $0 \leq (x::int) \Longrightarrow x < 1 << n \Longrightarrow \forall i \geq n.\ \neg\ x\ !!\ i$
  apply (*clarsimp simp*: *shiftl-int-def*)
  apply (*clarsimp simp*: *bin-nth-eq-mod even-iff-mod-2-eq-zero*)
  apply (*drule-tac z=2^i* **in** *less-le-trans*)
   apply *simp*
  apply *simp*
  done
— TODO: The converse should be true as well, but seems hard to prove.

lemma *int-eq-test-bit*:
  $((x :: int) = y) = (\forall i.\ test\text{-}bit\ x\ i = test\text{-}bit\ y\ i)$
  apply *simp*
  apply (*metis bin-eqI*)
  done
lemmas *int-eq-test-bitI* = *int-eq-test-bit*[*THEN iffD2*, *rule-format*]

lemma *le-nat-shrink-left*:
  $y \leq z \Longrightarrow y = Suc\ x \Longrightarrow x < z$
  by *simp*

lemma *length-ge-split*:
  $n < length\ xs \Longrightarrow \exists x\ xs'.\ xs = x \# xs' \wedge n \leq length\ xs'$
  by (*cases xs*) *auto*

end
Nondeterministic State Monad with Failure **theory** *NonDetMonad*
imports ../*Lib*
begin

State monads are used extensively in the seL4 specification. They are defined below.

17 The Monad

The basic type of the nondeterministic state monad with failure is very similar to the normal state monad. Instead of a pair consisting of result and new state, we return a set of these pairs coupled with a failure flag. Each element in the set is a potential result of the computation. The flag is *True* if there is an execution path in the computation that may have failed.

Conversely, if the flag is *False*, none of the computations resulting in the returned set can have failed.

type-synonym $('s,'a)$ *nondet-monad* $= 's \Rightarrow ('a \times 's)$ *set* $\times$ *bool*

Print the type $('s, 'a)$ *nondet-monad* instead of its unwieldy expansion. Needs an AST translation in code, because it needs to check that the state variable $'s$ occurs twice. This comparison is not guaranteed to always work as expected (AST instances might have different decoration), but it does seem to work here.

print-ast-translation ‹
```
  let
    fun monad-tr - [t1, Ast.Appl [Ast.Constant @{type-syntax prod},
                        Ast.Appl [Ast.Constant @{type-syntax set},
                          Ast.Appl [Ast.Constant @{type-syntax prod}, t2, t3]],
                        Ast.Constant @{type-syntax bool}]] =
      if t3 = t1
      then Ast.Appl [Ast.Constant @{type-syntax nondet-monad}, t1, t2]
      else raise Match
  in [(@{type-syntax fun}, monad-tr)] end
```
›

The definition of fundamental monad functions *return* and *bind*. The monad function *return x* does not change the state, does not fail, and returns *x*.

definition
return :: $'a \Rightarrow ('s,'a)$ *nondet-monad* **where**
return a $\equiv \lambda s.\ (\{(a,s)\},False)$

The monad function *bind f g*, also written $f >>= g$, is the execution of *f* followed by the execution of *g*. The function *g* takes the result value *and* the result state of *f* as parameter. The definition says that the result of the combined operation is the union of the set of sets that is created by *g* applied to the result sets of *f*. The combined operation may have failed, if *f* may have failed or *g* may have failed on any of the results of *f*.

definition
bind :: $('s, 'a)$ *nondet-monad* $\Rightarrow ('a \Rightarrow ('s, 'b)$ *nondet-monad*$) \Rightarrow$
$('s, 'b)$ *nondet-monad* (**infixl** $>>=$ *60*)
where
bind f g $\equiv \lambda s.\ (\bigcup(fst\ `\ case\text{-}prod\ g\ `\ fst\ (f\ s)),$
$True \in snd\ `\ case\text{-}prod\ g\ `\ fst\ (f\ s) \vee snd\ (f\ s))$

Sometimes it is convenient to write *bind* in reverse order.

abbreviation(*input*)
bind-rev :: $('c \Rightarrow ('a, 'b)$ *nondet-monad*$) \Rightarrow ('a, 'c)$ *nondet-monad* $\Rightarrow$
$('a, 'b)$ *nondet-monad* (**infixl** $=<<$ *60*) **where**
$g =<< f \equiv f >>= g$

The basic accessor functions of the state monad. *get* returns the current state as result, does not fail, and does not change the state. *put s* returns nothing (*unit*), changes the current state to *s* and does not fail.

definition
$get :: ('s,'s)\ nondet\text{-}monad$ **where**
$get \equiv \lambda s.\ (\{(s,s)\},\ False)$

definition
$put :: 's \Rightarrow ('s,\ unit)\ nondet\text{-}monad$ **where**
$put\ s \equiv \lambda\text{-}.\ (\{((),s)\},\ False)$

17.1 Nondeterminism

Basic nondeterministic functions. *select A* chooses an element of the set *A*, does not change the state, and does not fail (even if the set is empty). *f OR g* executes *f* or executes *g*. It retuns the union of results of *f* and *g*, and may have failed if either may have failed.

definition
$select :: 'a\ set \Rightarrow ('s,'a)\ nondet\text{-}monad$ **where**
$select\ A \equiv \lambda s.\ (A \times \{s\},\ False)$

definition
$alternative :: ('s,'a)\ nondet\text{-}monad \Rightarrow ('s,'a)\ nondet\text{-}monad \Rightarrow ('s,'a)\ nondet\text{-}monad$
(**infixl** *OR 20*)
where
$f\ OR\ g \equiv \lambda s.\ (fst\ (f\ s) \cup fst\ (g\ s),\ snd\ (f\ s) \vee snd\ (g\ s))$

Alternative notation for *OR*

notation (*xsymbols*) *alternative* (**infixl** $\sqcap$ *20*)

A variant of *select* that takes a pair. The first component is a set as in normal *select*, the second component indicates whether the execution failed. This is useful to lift monads between different state spaces.

definition
$select\text{-}f :: 'a\ set \times bool \Rightarrow ('s,'a)\ nondet\text{-}monad$ **where**
$select\text{-}f\ S \equiv \lambda s.\ (fst\ S \times \{s\},\ snd\ S)$

select-state takes a relationship between states, and outputs nondeterministically a state related to the input state.

definition
$state\text{-}select :: ('s \times 's)\ set \Rightarrow ('s,\ unit)\ nondet\text{-}monad$
where
$state\text{-}select\ r \equiv \lambda s.\ ((\lambda x.\ ((),\ x))\ `\ \{s'.\ (s,\ s') \in r\},\ \neg\ (\exists s'.\ (s,\ s') \in r))$

17.2 Failure

The monad function that always fails. Returns an empty set of results and sets the failure flag.

definition
 fail :: ($'s$, $'a$) *nondet-monad* **where**
fail $\equiv$ $\lambda s.$ ($\{\}$, *True*)

Assertions: fail if the property P is not true

definition
 assert :: *bool* $\Rightarrow$ ($'a$, *unit*) *nondet-monad* **where**
assert P $\equiv$ *if P then return* () *else fail*

Fail if the value is *None*, return result v for *Some* v

definition
 assert-opt :: $'a$ *option* $\Rightarrow$ ($'b$, $'a$) *nondet-monad* **where**
assert-opt v $\equiv$ *case v of None* $\Rightarrow$ *fail* | *Some v* $\Rightarrow$ *return v*

An assertion that also can introspect the current state.

definition
 state-assert :: ($'s$ $\Rightarrow$ *bool*) $\Rightarrow$ ($'s$, *unit*) *nondet-monad*
where
 state-assert P $\equiv$ *get* >>= ($\lambda s.$ *assert* (*P s*))

17.3 Generic functions on top of the state monad

Apply a function to the current state and return the result without changing the state.

definition
 gets :: ($'s$ $\Rightarrow$ $'a$) $\Rightarrow$ ($'s$, $'a$) *nondet-monad* **where**
gets f $\equiv$ *get* >>= ($\lambda s.$ *return* (*f s*))

Modify the current state using the function passed in.

definition
 modify :: ($'s$ $\Rightarrow$ $'s$) $\Rightarrow$ ($'s$, *unit*) *nondet-monad* **where**
modify f $\equiv$ *get* >>= ($\lambda s.$ *put* (*f s*))

lemma *simpler-gets-def*: *gets f* = ($\lambda s.$ ($\{$(*f s*, *s*)$\}$, *False*))
 apply (*simp add*: *gets-def return-def bind-def get-def*)
 done

lemma *simpler-modify-def*:
 modify f = ($\lambda s.$ ($\{$((), *f s*)$\}$, *False*))
 by (*simp add*: *modify-def bind-def get-def put-def*)

Execute the given monad when the condition is true, return () otherwise.

definition

when :: *bool* ⇒ (*'s*, *unit*) *nondet-monad* ⇒
(*'s*, *unit*) *nondet-monad* **where**
when P m ≡ *if P then m else return* ()

Execute the given monad unless the condition is true, return () otherwise.

definition
unless :: *bool* ⇒ (*'s*, *unit*) *nondet-monad* ⇒
(*'s*, *unit*) *nondet-monad* **where**
unless P m ≡ *when* (¬*P*) *m*

Perform a test on the current state, performing the left monad if the result is true or the right monad if the result is false.

definition
condition :: (*'s* ⇒ *bool*) ⇒ (*'s*, *'r*) *nondet-monad* ⇒ (*'s*, *'r*) *nondet-monad* ⇒ (*'s*, *'r*) *nondet-monad*
where
condition P L R ≡ λ*s*. *if* (*P s*) *then* (*L s*) *else* (*R s*)

notation (**output**)
condition ((*condition* (-)// (-)// (-)) [*1000*,*1000*,*1000*] *1000*)

Apply an option valued function to the current state, fail if it returns *None*, return *v* if it returns *Some v*.

definition
gets-the :: (*'s* ⇒ *'a option*) ⇒ (*'s*, *'a*) *nondet-monad* **where**
gets-the f ≡ *gets f* >>= *assert-opt*

Get a map (such as a heap) from the current state and apply an argument to the map. Fail if the map returns *None*, otherwise return the value.

definition
gets-map :: (*'s* ⇒ *'a* ⇒ *'b option*) ⇒ *'a* ⇒ (*'s*, *'b*) *nondet-monad* **where**
gets-map f p ≡ *gets f* >>= (λ*m*. *assert-opt* (*m p*))

17.4 The Monad Laws

A more expanded definition of *bind*

lemma *bind-def'*:
(*f* >>= *g*) ≡
λ*s*. ({(*r''*, *s''*). ∃(*r'*, *s'*) ∈ *fst* (*f s*). (*r''*, *s''*) ∈ *fst* (*g r' s'*) },
snd (*f s*) ∨ (∃(*r'*, *s'*) ∈ *fst* (*f s*). *snd* (*g r' s'*)))
apply (*rule eq-reflection*)
apply (*auto simp add*: *bind-def split-def Let-def*)
done

Each monad satisfies at least the following three laws.

return is absorbed at the left of a (>>=), applying the return value directly:

lemma *return-bind* [*simp*]: (*return* x >>= f) = f x
  by (*simp add*: *return-def bind-def*)

return is absorbed on the right of a (>>=)

lemma *bind-return* [*simp*]: (m >>= *return*) = m
  apply (*rule ext*)
  apply (*simp add*: *bind-def return-def split-def*)
  done

(>>=) is associative

lemma *bind-assoc*:
  fixes m :: ($'a$,$'b$) *nondet-monad*
  fixes f :: $'b \Rightarrow$ ($'a$,$'c$) *nondet-monad*
  fixes g :: $'c \Rightarrow$ ($'a$,$'d$) *nondet-monad*
  shows (m >>= f) >>= g = m >>= (λx. f x >>= g)
  apply (*unfold bind-def Let-def split-def*)
  apply (*rule ext*)
  apply *clarsimp*
  apply (*auto intro*: *rev-image-eqI*)
  done

18 Adding Exceptions

The type ($'s$, $'a$) *nondet-monad* gives us nondeterminism and failure. We now extend this monad with exceptional return values that abort normal execution, but can be handled explicitly. We use the sum type to indicate exceptions.

In ($'s$, $'e$ + $'a$) *nondet-monad*, $'s$ is the state, $'e$ is an exception, and $'a$ is a normal return value.

This new type itself forms a monad again. Since type classes in Isabelle are not powerful enough to express the class of monads, we provide new names for the *return* and (>>=) functions in this monad. We call them *returnOk* (for normal return values) and *bindE* (for composition). We also define *throwError* to return an exceptional value.

definition
  returnOk :: $'a \Rightarrow$ ($'s$, $'e$ + $'a$) *nondet-monad* **where**
  returnOk $\equiv$ *return o Inr*

definition
  throwError :: $'e \Rightarrow$ ($'s$, $'e$ + $'a$) *nondet-monad* **where**
  throwError $\equiv$ *return o Inl*

Lifting a function over the exception type: if the input is an exception, return that exception; otherwise continue execution.

definition

$lift :: ('a \Rightarrow ('s, 'e + 'b)\ nondet\text{-}monad) \Rightarrow$
$\quad 'e + 'a \Rightarrow ('s, 'e + 'b)\ nondet\text{-}monad$

where

$lift\ f\ v \equiv case\ v\ of\ Inl\ e \Rightarrow throwError\ e$
$\quad\quad\quad\quad \mid Inr\ v' \Rightarrow f\ v'$

The definition of (>>=) in the exception monad (new name *bindE*): the same as normal (>>=), but the right-hand side is skipped if the left-hand side produced an exception.

definition

$bindE :: ('s, 'e + 'a)\ nondet\text{-}monad \Rightarrow$
$\quad ('a \Rightarrow ('s, 'e + 'b)\ nondet\text{-}monad) \Rightarrow$
$\quad ('s, 'e + 'b)\ nondet\text{-}monad$ (**infixl** $>>=E\ 60$)

where

$bindE\ f\ g \equiv bind\ f\ (lift\ g)$

Lifting a normal nondeterministic monad into the exception monad is achieved by always returning its result as normal result and never throwing an exception.

definition

$liftE :: ('s,'a)\ nondet\text{-}monad \Rightarrow ('s, 'e+'a)\ nondet\text{-}monad$

where

$liftE\ f \equiv f >>= (\lambda r.\ return\ (Inr\ r))$

Since the underlying type and *return* function changed, we need new definitions for when and unless:

definition

$whenE :: bool \Rightarrow ('s, 'e + unit)\ nondet\text{-}monad \Rightarrow$
$\quad ('s, 'e + unit)\ nondet\text{-}monad$

where

$whenE\ P\ f \equiv if\ P\ then\ f\ else\ returnOk\ ()$

definition

$unlessE :: bool \Rightarrow ('s, 'e + unit)\ nondet\text{-}monad \Rightarrow$
$\quad ('s, 'e + unit)\ nondet\text{-}monad$

where

$unlessE\ P\ f \equiv if\ P\ then\ returnOk\ ()\ else\ f$

Throwing an exception when the parameter is *None*, otherwise returning v for *Some* v.

definition

$throw\text{-}opt :: 'e \Rightarrow 'a\ option \Rightarrow ('s, 'e + 'a)\ nondet\text{-}monad$ **where**
$throw\text{-}opt\ ex\ x \equiv$
$case\ x\ of\ None \Rightarrow throwError\ ex \mid Some\ v \Rightarrow returnOk\ v$

Failure in the exception monad is redefined in the same way as *whenE* and *unlessE*, with *returnOk* instead of *return*.

definition
assertE :: *bool* ⇒ ('*a*, '*e* + *unit*) *nondet-monad* **where**
assertE P ≡ *if P then returnOk* () *else fail*

18.1 Monad Laws for the Exception Monad

More direct definition of *liftE*:

lemma *liftE-def2*:
liftE f = ($\lambda s.$ (($\lambda(v,s')$. (*Inr v*, *s'*)) ‘ *fst* (*f s*), *snd* (*f s*)))
by (*auto simp*: *liftE-def return-def split-def bind-def*)

Left *returnOk* absorbtion over (>>=*E*):

lemma *returnOk-bindE* [*simp*]: (*returnOk x* >>=*E f*) = *f x*
apply (*unfold bindE-def returnOk-def*)
apply (*clarsimp simp*: *lift-def*)
done

lemma *lift-return* [*simp*]:
lift (*return* ∘ *Inr*) = *return*
by (*rule ext*)
(*simp add*: *lift-def throwError-def split*: *sum.splits*)

Right *returnOk* absorbtion over (>>=*E*):

lemma *bindE-returnOk* [*simp*]: (*m* >>=*E returnOk*) = *m*
by (*simp add*: *bindE-def returnOk-def*)

Associativity of (>>=*E*):

lemma *bindE-assoc*:
(*m* >>=*E f*) >>=*E g* = *m* >>=*E* ($\lambda x.$ *f x* >>=*E g*)
apply (*simp add*: *bindE-def bind-assoc*)
apply (*rule arg-cong* [**where** *f*=$\lambda x.$ *m* >>= *x*])
apply (*rule ext*)
apply (*case-tac x*, *simp-all add*: *lift-def throwError-def*)
done

returnOk could also be defined via *liftE*:

lemma *returnOk-liftE*:
returnOk x = *liftE* (*return x*)
by (*simp add*: *liftE-def returnOk-def*)

Execution after throwing an exception is skipped:

lemma *throwError-bindE* [*simp*]:
(*throwError E* >>=*E f*) = *throwError E*
by (*simp add*: *bindE-def bind-def throwError-def lift-def return-def*)

19 Syntax

This section defines traditional Haskell-like do-syntax for the state monad in Isabelle.

19.1 Syntax for the Nondeterministic State Monad

We use *K-bind* to syntactically indicate the case where the return argument of the left side of a (>>=) is ignored

definition
K-bind-def [*iff*]: *K-bind* ≡ λ*x y*. *x*

nonterminal
dobinds **and** *dobind* **and** *nobind*

syntax
```
-dobind    :: [pttrn, 'a] => dobind              ((- <−/ -) 10)
           :: dobind => dobinds                  (-)
-nobind    :: 'a => dobind                       (-)
-dobinds   :: [dobind, dobinds] => dobinds       ((-);//(-))

-do        :: [dobinds, 'a] => 'a                ((do ((-);//(-))//od) 100)
```
syntax (*xsymbols*)
```
-dobind    :: [pttrn, 'a] => dobind              ((- ←/ -) 10)
```

translations
```
-do (-dobinds b bs) e  == -do b (-do bs e)
-do (-nobind b) e      == b >>= (CONST K-bind e)
do x <− a; e od        == a >>= (λx. e)
```

Syntax examples:

lemma *do x* ← *return 1*;
return (*2*::*nat*);
return x
od =
return 1 >>=
(λ*x*. *return* (*2*::*nat*) >>=
K-bind (*return x*))
by (*rule refl*)

lemma *do x* ← *return 1*;
return 2;
return x
od = *return 1*
by *simp*

19.2 Syntax for the Exception Monad

Since the exception monad is a different type, we need to syntactically distinguish it in the syntax. We use *doE*/*odE* for this, but can re-use most of the productions from *do*/*od* above.

syntax
-doE :: [*dobinds*, *'a*] => *'a* ((*doE* ((-);//(-))//*odE*) *100*)

translations
-doE (*-dobinds b bs*) *e* == *-doE b* (*-doE bs e*)
-doE (*-nobind b*) *e* == *b* >>=*E* (*CONST K-bind e*)
doE x <− *a*; *e odE* == *a* >>=*E* (λx. *e*)

Syntax examples:

lemma *doE x* ← *returnOk 1*;
returnOk (*2*::*nat*);
returnOk x
odE =
returnOk 1 >>=*E*
(λx. *returnOk* (*2*::*nat*) >>=*E*
K-bind (*returnOk x*))
by (*rule refl*)

lemma *doE x* ← *returnOk 1*;
returnOk 2;
returnOk x
odE = *returnOk 1*
by *simp*

20 Library of Monadic Functions and Combinators

Lifting a normal function into the monad type:

definition
liftM :: (*'a* ⇒ *'b*) ⇒ (*'s*,*'a*) *nondet-monad* ⇒ (*'s*, *'b*) *nondet-monad*
where
liftM f m ≡ *do x* ← *m*; *return* (*f x*) *od*

The same for the exception monad:

definition
liftME :: (*'a* ⇒ *'b*) ⇒ (*'s*,*'e*+*'a*) *nondet-monad* ⇒ (*'s*,*'e*+*'b*) *nondet-monad*
where
liftME f m ≡ *doE x* ← *m*; *returnOk* (*f x*) *odE*

Run a sequence of monads from left to right, ignoring return values.

definition

sequence-x :: ('s, 'a) *nondet-monad list* ⇒ ('s, *unit*) *nondet-monad*
where
sequence-x xs ≡ *foldr* (λ*x y*. *x* >>= (λ-. *y*)) *xs* (*return* ())

Map a monadic function over a list by applying it to each element of the list from left to right, ignoring return values.

definition
mapM-x :: ('a ⇒ ('s,'b) *nondet-monad*) ⇒ 'a *list* ⇒ ('s, *unit*) *nondet-monad*
where
mapM-x f xs ≡ *sequence-x* (*map f xs*)

Map a monadic function with two parameters over two lists, going through both lists simultaneously, left to right, ignoring return values.

definition
zipWithM-x :: ('a ⇒ 'b ⇒ ('s,'c) *nondet-monad*) ⇒
'a *list* ⇒ 'b *list* ⇒ ('s, *unit*) *nondet-monad*
where
zipWithM-x f xs ys ≡ *sequence-x* (*zipWith f xs ys*)

The same three functions as above, but returning a list of return values instead of *unit*

definition
sequence :: ('s, 'a) *nondet-monad list* ⇒ ('s, 'a *list*) *nondet-monad*
where
sequence xs ≡ *let mcons* = (λ*p q*. *p* >>= (λ*x*. *q* >>= (λ*y*. *return* (*x*#*y*))))
in foldr mcons xs (*return* [])

definition
mapM :: ('a ⇒ ('s,'b) *nondet-monad*) ⇒ 'a *list* ⇒ ('s, 'b *list*) *nondet-monad*
where
mapM f xs ≡ *sequence* (*map f xs*)

definition
zipWithM :: ('a ⇒ 'b ⇒ ('s,'c) *nondet-monad*) ⇒
'a *list* ⇒ 'b *list* ⇒ ('s, 'c *list*) *nondet-monad*
where
zipWithM f xs ys ≡ *sequence* (*zipWith f xs ys*)

definition
foldM :: ('b ⇒ 'a ⇒ ('s, 'a) *nondet-monad*) ⇒ 'b *list* ⇒ 'a ⇒ ('s, 'a) *nondet-monad*
where
foldM m xs a ≡ *foldr* (λ*p q*. *q* >>= *m p*) *xs* (*return a*)

definition
foldME ::('b ⇒ 'a ⇒ ('s,('e + 'b)) *nondet-monad*) ⇒ 'b ⇒ 'a *list* ⇒ ('s, ('e + 'b)) *nondet-monad*
where *foldME m a xs* ≡ *foldr* (λ*p q*. *q* >>=*E swp m p*) *xs* (*returnOk a*)

The sequence and map functions above for the exception monad, with and without lists of return value

definition
sequenceE-x :: ('s, 'e+'a) *nondet-monad list* ⇒ ('s, 'e+*unit*) *nondet-monad*
where
sequenceE-x xs ≡ *foldr* (λ*x y*. *doE* - <− *x*; *y odE*) *xs* (*returnOk* ())

definition
mapME-x :: ('a ⇒ ('s,'e+'b) *nondet-monad*) ⇒ 'a *list* ⇒
('s,'e+*unit*) *nondet-monad*
where
mapME-x f xs ≡ *sequenceE-x* (*map f xs*)

definition
sequenceE :: ('s, 'e+'a) *nondet-monad list* ⇒ ('s, 'e+'a *list*) *nondet-monad*
where
sequenceE xs ≡ *let mcons* = (λ*p q*. *p* >>=*E* (λ*x*. *q* >>=*E* (λ*y*. *returnOk* (*x*#*y*))))
in foldr mcons xs (*returnOk* [])

definition
mapME :: ('a ⇒ ('s,'e+'b) *nondet-monad*) ⇒ 'a *list* ⇒
('s,'e+'b *list*) *nondet-monad*
where
mapME f xs ≡ *sequenceE* (*map f xs*)

Filtering a list using a monadic function as predicate:

primrec
filterM :: ('a ⇒ ('s, *bool*) *nondet-monad*) ⇒ 'a *list* ⇒ ('s, 'a *list*) *nondet-monad*
where
filterM P [] = *return* []
| *filterM P* (*x* # *xs*) = *do*
b <− *P x*;
ys <− *filterM P xs*;
return (*if b then* (*x* # *ys*) *else ys*)
od

21 Catching and Handling Exceptions

Turning an exception monad into a normal state monad by catching and handling any potential exceptions:

definition
catch :: ('s, 'e + 'a) *nondet-monad* ⇒
('e ⇒ ('s, 'a) *nondet-monad*) ⇒
('s, 'a) *nondet-monad* (**infix** <*catch*> *10*)
where
f <*catch*> *handler* ≡

```
do x ← f;
   case x of
     Inr b ⇒ return b
   | Inl e ⇒ handler e
od
```

Handling exceptions, but staying in the exception monad. The handler may throw a type of exceptions different from the left side.

definition
```
  handleE' :: ('s, 'e1 + 'a) nondet-monad ⇒
              ('e1 ⇒ ('s, 'e2 + 'a) nondet-monad) ⇒
              ('s, 'e2 + 'a) nondet-monad (infix <handle2> 10)
```
where
```
  f <handle2> handler ≡
   do
      v ← f;
      case v of
        Inl e ⇒ handler e
      | Inr v' ⇒ return (Inr v')
   od
```

A type restriction of the above that is used more commonly in practice: the exception handle (potentially) throws exception of the same type as the left-hand side.

definition
```
  handleE :: ('s, 'x + 'a) nondet-monad ⇒
             ('x ⇒ ('s, 'x + 'a) nondet-monad) ⇒
             ('s, 'x + 'a) nondet-monad (infix <handle> 10)
```
where
```
  handleE ≡ handleE'
```

Handling exceptions, and additionally providing a continuation if the left-hand side throws no exception:

definition
```
  handle-elseE :: ('s, 'e + 'a) nondet-monad ⇒
                  ('e ⇒ ('s, 'ee + 'b) nondet-monad) ⇒
                  ('a ⇒ ('s, 'ee + 'b) nondet-monad) ⇒
                  ('s, 'ee + 'b) nondet-monad
  (- <handle> - <else> - 10)
```
where
```
  f <handle> handler <else> continue ≡
   do v ← f;
   case v of Inl e  ⇒ handler e
           | Inr v' ⇒ continue v'
   od
```

21.1 Loops

Loops are handled using the following inductive predicate; non-termination is represented using the failure flag of the monad.

inductive-set
  whileLoop-results :: $('r \Rightarrow 's \Rightarrow bool) \Rightarrow ('r \Rightarrow ('s, 'r)\ nondet\text{-}monad) \Rightarrow ((('r \times 's)\ option) \times (('r \times 's)\ option))\ set$
  for $C\ B$
where
  $[\![\neg\ C\ r\ s]\!] \Longrightarrow (Some\ (r,\ s),\ Some\ (r,\ s)) \in whileLoop\text{-}results\ C\ B$
  $|\ [\![C\ r\ s;\ snd\ (B\ r\ s)]\!] \Longrightarrow (Some\ (r,\ s),\ None) \in whileLoop\text{-}results\ C\ B$
  $|\ [\![C\ r\ s;\ (r',\ s') \in fst\ (B\ r\ s);\ (Some\ (r',\ s'),\ z) \in whileLoop\text{-}results\ C\ B\ ]\!]$
    $\Longrightarrow (Some\ (r,\ s),\ z) \in whileLoop\text{-}results\ C\ B$

inductive-cases *whileLoop-results-cases-valid*: $(Some\ x,\ Some\ y) \in whileLoop\text{-}results\ C\ B$
inductive-cases *whileLoop-results-cases-fail*: $(Some\ x,\ None) \in whileLoop\text{-}results\ C\ B$
inductive-simps *whileLoop-results-simps*: $(Some\ x,\ y) \in whileLoop\text{-}results\ C\ B$
inductive-simps *whileLoop-results-simps-valid*: $(Some\ x,\ Some\ y) \in whileLoop\text{-}results\ C\ B$
inductive-simps *whileLoop-results-simps-start-fail* [*simp*]: $(None,\ x) \in whileLoop\text{-}results\ C\ B$

inductive
  whileLoop-terminates :: $('r \Rightarrow 's \Rightarrow bool) \Rightarrow ('r \Rightarrow ('s, 'r)\ nondet\text{-}monad) \Rightarrow 'r \Rightarrow 's \Rightarrow bool$
  for $C\ B$
where
  $\neg\ C\ r\ s \Longrightarrow whileLoop\text{-}terminates\ C\ B\ r\ s$
  $|\ [\![C\ r\ s;\ \forall (r',\ s') \in fst\ (B\ r\ s).\ whileLoop\text{-}terminates\ C\ B\ r'\ s']\!]$
    $\Longrightarrow whileLoop\text{-}terminates\ C\ B\ r\ s$

inductive-cases *whileLoop-terminates-cases*: $whileLoop\text{-}terminates\ C\ B\ r\ s$
inductive-simps *whileLoop-terminates-simps*: $whileLoop\text{-}terminates\ C\ B\ r\ s$

definition
  $whileLoop\ C\ B \equiv (\lambda r\ s.$
    $(\{(r',s').\ (Some\ (r,\ s),\ Some\ (r',\ s')) \in whileLoop\text{-}results\ C\ B\},$
      $(Some\ (r,\ s),\ None) \in whileLoop\text{-}results\ C\ B \vee (\neg\ whileLoop\text{-}terminates\ C\ B\ r\ s)))$

notation (**output**)
  whileLoop ((*whileLoop* (-)// (-)) [*1000*, *1000*] *1000*)

definition
  whileLoopE :: $('r \Rightarrow 's \Rightarrow bool) \Rightarrow ('r \Rightarrow ('s, 'e + 'r)\ nondet\text{-}monad)$
    $\Rightarrow 'r \Rightarrow 's \Rightarrow (('e + 'r) \times 's)\ set \times bool$
where

$whileLoopE\ C\ body \equiv$
$\quad \lambda r.\ whileLoop\ (\lambda r\ s.\ (case\ r\ of\ Inr\ v \Rightarrow C\ v\ s \mid \text{-} \Rightarrow False))\ (lift\ body)\ (Inr\ r)$

notation (**output**)
$whileLoopE\ \ ((whileLoopE\ (\text{-})//\ \ (\text{-}))\ [1000,\ 1000]\ 1000)$

22 Hoare Logic

22.1 Validity

This section defines a Hoare logic for partial correctness for the nondeterministic state monad as well as the exception monad. The logic talks only about the behaviour part of the monad and ignores the failure flag.

The logic is defined semantically. Rules work directly on the validity predicate.

In the nondeterministic state monad, validity is a triple of precondition, monad, and postcondition. The precondition is a function from state to bool (a state predicate), the postcondition is a function from return value to state to bool. A triple is valid if for all states that satisfy the precondition, all result values and result states that are returned by the monad satisfy the postcondition. Note that if the computation returns the empty set, the triple is trivially valid. This means *assert P* does not require us to prove that P holds, but rather allows us to assume P! Proving non-failure is done via separate predicate and calculus (see below).

definition
$valid :: ('s \Rightarrow bool) \Rightarrow ('s,'a)\ nondet\text{-}monad \Rightarrow ('a \Rightarrow 's \Rightarrow bool) \Rightarrow bool$
$(\{\!|\text{-}|\!\}/\ \text{-}\ /\{\!|\text{-}|\!\})$
where
$\{\!|P|\!\}\ f\ \{\!|Q|\!\} \equiv \forall s.\ P\ s \longrightarrow (\forall (r,s') \in fst\ (f\ s).\ Q\ r\ s')$

We often reason about invariant predicates. The following provides shorthand syntax that avoids repeating potentially long predicates.

abbreviation (*input*)
$invariant :: ('s,'a)\ nondet\text{-}monad \Rightarrow ('s \Rightarrow bool) \Rightarrow bool\ (\text{-}\ \{\!|\text{-}|\!\}\ [59,0]\ 60)$
where
$invariant\ f\ P \equiv \{\!|P|\!\}\ f\ \{\!|\lambda\text{-}.\ P|\!\}$

Validity for the exception monad is similar and build on the standard validity above. Instead of one postcondition, we have two: one for normal and one for exceptional results.

definition
$validE :: ('s \Rightarrow bool) \Rightarrow ('s,\ 'a + 'b)\ nondet\text{-}monad \Rightarrow$
$\qquad ('b \Rightarrow 's \Rightarrow bool) \Rightarrow$
$\qquad ('a \Rightarrow 's \Rightarrow bool) \Rightarrow bool$
$(\{\!|\text{-}|\!\}/\ \text{-}\ /(\{\!|\text{-}|\!\},/\ \{\!|\text{-}|\!\}))$

where
$\{\!|P|\!\}\ f\ \{\!|Q|\!\},\{\!|E|\!\} \equiv \{\!|P|\!\}\ f\ \{\!|\ \lambda v\ s.\ \mathit{case}\ v\ \mathit{of}\ \mathit{Inr}\ r \Rightarrow Q\ r\ s \mid \mathit{Inl}\ e \Rightarrow E\ e\ s\ |\!\}$

The following two instantiations are convenient to separate reasoning for exceptional and normal case.

definition
$\mathit{validE\text{-}R} :: ('s \Rightarrow \mathit{bool}) \Rightarrow ('s,\ 'e + 'a)\ \mathit{nondet\text{-}monad} \Rightarrow$
$('a \Rightarrow 's \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$
$(\{\!|\text{-}|\!\}/\ \text{-}\ /\{\!|\text{-}|\!\},\ -)$
where
$\{\!|P|\!\}\ f\ \{\!|Q|\!\},- \equiv \mathit{validE}\ P\ f\ Q\ (\lambda x\ y.\ \mathit{True})$

definition
$\mathit{validE\text{-}E} :: ('s \Rightarrow \mathit{bool}) \Rightarrow ('s,\ 'e + 'a)\ \mathit{nondet\text{-}monad} \Rightarrow$
$('e \Rightarrow 's \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}$
$(\{\!|\text{-}|\!\}/\ \text{-}\ /-,\ \{\!|\text{-}|\!\})$
where
$\{\!|P|\!\}\ f\ -,\{\!|Q|\!\} \equiv \mathit{validE}\ P\ f\ (\lambda x\ y.\ \mathit{True})\ Q$

Abbreviations for trivial preconditions:

abbreviation(*input*)
$\mathit{top} :: 'a \Rightarrow \mathit{bool}\ (\top)$
where
$\top \equiv \lambda\text{-}.\ \mathit{True}$

abbreviation(*input*)
$\mathit{bottom} :: 'a \Rightarrow \mathit{bool}\ (\bot)$
where
$\bot \equiv \lambda\text{-}.\ \mathit{False}$

Abbreviations for trivial postconditions (taking two arguments):

abbreviation(*input*)
$\mathit{toptop} :: 'a \Rightarrow 'b \Rightarrow \mathit{bool}\ (\top\top)$
where
$\top\top \equiv \lambda\text{-}\ \text{-}.\ \mathit{True}$

abbreviation(*input*)
$\mathit{botbot} :: 'a \Rightarrow 'b \Rightarrow \mathit{bool}\ (\bot\bot)$
where
$\bot\bot \equiv \lambda\text{-}\ \text{-}.\ \mathit{False}$

Lifting $\wedge$ and $\vee$ over two arguments. Lifting $\wedge$ and $\vee$ over one argument is already defined (written *and* and *or*).

definition
$\mathit{bipred\text{-}conj} :: ('a \Rightarrow 'b \Rightarrow \mathit{bool}) \Rightarrow ('a \Rightarrow 'b \Rightarrow \mathit{bool}) \Rightarrow ('a \Rightarrow 'b \Rightarrow \mathit{bool})$
(**infixl** *And 96*)
where
$\mathit{bipred\text{-}conj}\ P\ Q \equiv \lambda x\ y.\ P\ x\ y \wedge Q\ x\ y$

definition
  $bipred\text{-}disj :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool)$
  (**infixl** *Or 91*)
where
  $bipred\text{-}disj\ P\ Q \equiv \lambda x\ y.\ P\ x\ y \lor Q\ x\ y$

22.2 Determinism

A monad of type *nondet-monad* is deterministic iff it returns exactly one state and result and does not fail

definition
  $det :: ('a,'s)\ nondet\text{-}monad \Rightarrow bool$
where
  $det\ f \equiv \forall s.\ \exists r.\ f\ s = (\{r\}, False)$

A deterministic *nondet-monad* can be turned into a normal state monad:

definition
  $the\text{-}run\text{-}state :: ('s,'a)\ nondet\text{-}monad \Rightarrow 's \Rightarrow 'a \times 's$
where
  $the\text{-}run\text{-}state\ M \equiv \lambda s.\ THE\ s'.\ fst\ (M\ s) = \{s'\}$

22.3 Non-Failure

With the failure flag, we can formulate non-failure separately from validity. A monad m does not fail under precondition P, if for no start state in that precondition it sets the failure flag.

definition
  $no\text{-}fail :: ('s \Rightarrow bool) \Rightarrow ('s,'a)\ nondet\text{-}monad \Rightarrow bool$
where
  $no\text{-}fail\ P\ m \equiv \forall s.\ P\ s \longrightarrow \neg\ (snd\ (m\ s))$

It is often desired to prove non-failure and a Hoare triple simultaneously, as the reasoning is often similar. The following definitions allow such reasoning to take place.

definition
  $validNF :: ('s \Rightarrow bool) \Rightarrow ('s,'a)\ nondet\text{-}monad \Rightarrow ('a \Rightarrow 's \Rightarrow bool) \Rightarrow bool$
    (⦃-⦄/ - /⦃-⦄!)
where
  $validNF\ P\ f\ Q \equiv valid\ P\ f\ Q \land no\text{-}fail\ P\ f$

definition
  $validE\text{-}NF :: ('s \Rightarrow bool) \Rightarrow ('s, 'a + 'b)\ nondet\text{-}monad \Rightarrow$
        $('b \Rightarrow 's \Rightarrow bool) \Rightarrow$
        $('a \Rightarrow 's \Rightarrow bool) \Rightarrow bool$
  (⦃-⦄/ - /(⦃-⦄,/ ⦃-⦄!))
where

validE-NF P f Q E $\equiv$ *validE P f Q E* $\wedge$ *no-fail P f*

lemma *validE-NF-alt-def*:
$\{\!\!|\ P\ |\!\!\}\ B\ \{\!\!|\ Q\ |\!\!\},\{\!\!|\ E\ |\!\!\}! = \{\!\!|\ P\ |\!\!\}\ B\ \{\!\!|\ \lambda v\ s.\ case\ v\ of\ Inl\ e \Rightarrow E\ e\ s \mid Inr\ r \Rightarrow Q\ r\ s\ |\!\!\}!$
by (*clarsimp simp*: *validE-NF-def validE-def validNF-def*)

Usually, well-formed monads constructed from the primitives above will have the following property: if they return an empty set of results, they will have the failure flag set.

definition
empty-fail :: $('s,'a)$ *nondet-monad* $\Rightarrow$ *bool*
where
empty-fail m $\equiv \forall s.\ fst\ (m\ s) = \{\} \longrightarrow snd\ (m\ s)$

Useful in forcing otherwise unknown executions to have the *empty-fail* property.

definition
mk-ef :: $'a\ set \times bool \Rightarrow 'a\ set \times bool$
where
mk-ef S $\equiv (fst\ S,\ fst\ S = \{\} \vee snd\ S)$

23 Basic exception reasoning

The following predicates *no-throw* and *no-return* allow reasoning that functions in the exception monad either do no throw an exception or never return normally.

definition *no-throw P A* $\equiv \{\!\!|\ P\ |\!\!\}\ A\ \{\!\!|\ \lambda\text{-}\ \text{-}.\ True\ |\!\!\},\{\!\!|\ \lambda\text{-}\ \text{-}.\ False\ |\!\!\}$

definition *no-return P A* $\equiv \{\!\!|\ P\ |\!\!\}\ A\ \{\!\!|\lambda\text{-}\ \text{-}.\ False|\!\!\},\{\!\!|\lambda\text{-}\ \text{-}.\ True\ |\!\!\}$

end

theory *NonDetMonadLemmas*
imports *NonDetMonad*
begin

24 General Lemmas Regarding the Nondeterministic State Monad

24.1 Congruence Rules for the Function Package

lemma *bind-cong*[*fundef-cong*]:
$[\![\ f = f';\ \bigwedge v\ s\ s'.\ (v,\ s') \in fst\ (f'\ s) \Longrightarrow g\ v\ s' = g'\ v\ s'\ ]\!] \Longrightarrow f >>= g = f' >>= g'$

apply (*rule ext*)
apply (*auto simp*: *bind-def Let-def split-def intro*: *rev-image-eqI*)
done

lemma *bind-apply-cong* [*fundef-cong*]:
$[\![f\ s = f'\ s'; \bigwedge rv\ st.\ (rv,\ st) \in fst\ (f'\ s') \Longrightarrow g\ rv\ st = g'\ rv\ st\ ]\!]$
$\Longrightarrow (f >>= g)\ s = (f' >>= g')\ s'$
apply (*simp add*: *bind-def*)
apply (*auto simp*: *split-def intro*: *SUP-cong* [*OF refl*] *intro*: *rev-image-eqI*)
done

lemma *bindE-cong*[*fundef-cong*]:
$[\![M = M'\ ; \bigwedge v\ s\ s'.\ (Inr\ v,\ s') \in fst\ (M'\ s) \Longrightarrow N\ v\ s' = N'\ v\ s'\ ]\!] \Longrightarrow bindE$
$M\ N = bindE\ M'\ N'$
apply (*simp add*: *bindE-def*)
apply (*rule bind-cong*)
apply (*rule refl*)
apply (*unfold lift-def*)
apply (*case-tac v*, *simp-all*)
done

lemma *bindE-apply-cong*[*fundef-cong*]:
$[\![f\ s = f'\ s'; \bigwedge rv\ st.\ (Inr\ rv,\ st) \in fst\ (f'\ s') \Longrightarrow g\ rv\ st = g'\ rv\ st\ ]\!]$
$\Longrightarrow (f >>=E\ g)\ s = (f' >>=E\ g')\ s'$
apply (*simp add*: *bindE-def*)
apply (*rule bind-apply-cong*)
apply *assumption*
apply (*case-tac rv*, *simp-all add*: *lift-def*)
done

lemma *K-bind-apply-cong*[*fundef-cong*]:
$[\![f\ st = f'\ st'\ ]\!] \Longrightarrow K\text{-}bind\ f\ arg\ st = K\text{-}bind\ f'\ arg'\ st'$
by *simp*

lemma *when-apply-cong*[*fundef-cong*]:
$[\![C = C'; s = s'; C' \Longrightarrow m\ s' = m'\ s'\ ]\!] \Longrightarrow whenE\ C\ m\ s = whenE\ C'\ m'\ s'$
by (*simp add*: *whenE-def*)

lemma *unless-apply-cong*[*fundef-cong*]:
$[\![C = C'; s = s'; \neg\ C' \Longrightarrow m\ s' = m'\ s'\ ]\!] \Longrightarrow unlessE\ C\ m\ s = unlessE\ C'\ m'$
s'
by (*simp add*: *unlessE-def*)

lemma *whenE-apply-cong*[*fundef-cong*]:
$[\![C = C'; s = s'; C' \Longrightarrow m\ s' = m'\ s'\ ]\!] \Longrightarrow whenE\ C\ m\ s = whenE\ C'\ m'\ s'$
by (*simp add*: *whenE-def*)

lemma *unlessE-apply-cong*[*fundef-cong*]:
$[\![C = C'; s = s'; \neg\ C' \Longrightarrow m\ s' = m'\ s'\ ]\!] \Longrightarrow unlessE\ C\ m\ s = unlessE\ C'\ m'$

s'
by (*simp add*: *unlessE-def*)

24.2 Simplifying Monads

lemma *nested-bind* [*simp*]:
do x <− do y <− f; *return* (*g y*) *od*; *h x od* =
do y <− f; *h* (*g y*) *od*
apply (*clarsimp simp add*: *bind-def*)
apply (*rule ext*)
apply (*clarsimp simp add*: *Let-def split-def return-def*)
done

lemma *fail-bind* [*simp*]:
fail >>= f = fail
by (*simp add*: *bind-def fail-def*)

lemma *fail-bindE* [*simp*]:
fail >>=E f = fail
by (*simp add*: *bindE-def bind-def fail-def*)

lemma *assert-False* [*simp*]:
assert False >>= f = fail
by (*simp add*: *assert-def*)

lemma *assert-True* [*simp*]:
assert True >>= f = f ()
by (*simp add*: *assert-def*)

lemma *assertE-False* [*simp*]:
assertE False >>=E f = fail
by (*simp add*: *assertE-def*)

lemma *assertE-True* [*simp*]:
assertE True >>=E f = f ()
by (*simp add*: *assertE-def*)

lemma *when-False-bind* [*simp*]:
when False g >>= f = f ()
by (*rule ext*) (*simp add*: *when-def bind-def return-def*)

lemma *when-True-bind* [*simp*]:
when True g >>= f = g >>= f
by (*simp add*: *when-def bind-def return-def*)

lemma *whenE-False-bind* [*simp*]:
whenE False g >>=E f = f ()
by (*simp add*: *whenE-def bindE-def returnOk-def lift-def*)

lemma *whenE-True-bind* [*simp*]:
whenE True g >>=*E f* = *g* >>=*E f*
by (*simp add*: *whenE-def bindE-def returnOk-def lift-def*)

lemma *when-True* [*simp*]: *when True X* = *X*
by (*clarsimp simp*: *when-def*)

lemma *when-False* [*simp*]: *when False X* = *return* ()
by (*clarsimp simp*: *when-def*)

lemma *unless-False* [*simp*]: *unless False X* = *X*
by (*clarsimp simp*: *unless-def*)

lemma *unlessE-False* [*simp*]: *unlessE False f* = *f*
unfolding *unlessE-def* **by** *fastforce*

lemma *unless-True* [*simp*]: *unless True X* = *return* ()
by (*clarsimp simp*: *unless-def*)

lemma *unlessE-True* [*simp*]: *unlessE True f* = *returnOk* ()
unfolding *unlessE-def* **by** *fastforce*

lemma *unlessE-whenE*:
unlessE P = *whenE* (~*P*)
by (*rule ext*)+ (*simp add*: *unlessE-def whenE-def*)

lemma *unless-when*:
unless P = *when* (~*P*)
by (*rule ext*)+ (*simp add*: *unless-def when-def*)

lemma *gets-to-return* [*simp*]: *gets* (λ*s*. *v*) = *return v*
by (*clarsimp simp*: *gets-def put-def get-def bind-def return-def*)

lemma *assert-opt-Some*:
assert-opt (*Some x*) = *return x*
by (*simp add*: *assert-opt-def*)

lemma *assertE-liftE*:
assertE P = *liftE* (*assert P*)
by (*simp add*: *assertE-def assert-def liftE-def returnOk-def*)

lemma *liftE-handleE'* [*simp*]: ((*liftE a*) <*handle2*> *b*) = *liftE a*
apply (*clarsimp simp*: *liftE-def handleE'-def*)
done

lemma *liftE-handleE* [*simp*]: ((*liftE a*) <*handle*> *b*) = *liftE a*
apply (*unfold handleE-def*)
apply *simp*
done

lemma *condition-split*:
P (*condition* C a b s) = ((((C s) $\longrightarrow$ P (a s)) $\wedge$ ($\neg$ (C s) $\longrightarrow$ P (b s))))
apply (*clarsimp simp*: *condition-def*)
done

lemma *condition-split-asm*:
P (*condition* C a b s) = ($\neg$ (C s $\wedge$ $\neg$ P (a s) $\vee$ $\neg$ C s $\wedge$ $\neg$ P (b s)))
apply (*clarsimp simp*: *condition-def*)
done

lemmas *condition-splits* = *condition-split condition-split-asm*

lemma *condition-true-triv* [*simp*]:
condition (λ-. *True*) A B = A
apply (*rule ext*)
apply (*clarsimp split*: *condition-splits*)
done

lemma *condition-false-triv* [*simp*]:
condition (λ-. *False*) A B = B
apply (*rule ext*)
apply (*clarsimp split*: *condition-splits*)
done

lemma *condition-true*: $[\![$ P s $]\!]$ $\Longrightarrow$ *condition* P A B s = A s
apply (*clarsimp simp*: *condition-def*)
done

lemma *condition-false*: $[\![$ $\neg$ P s $]\!]$ $\Longrightarrow$ *condition* P A B s = B s
apply (*clarsimp simp*: *condition-def*)
done

lemmas *arg-cong-bind* = *arg-cong2*[**where** *f*=*bind*]
lemmas *arg-cong-bind1* = *arg-cong-bind*[*OF refl ext*]

25 Low-level monadic reasoning

lemma *monad-eqI* [*intro*]:
$[\![$ $\bigwedge r\ t\ s.$ (r, t) $\in$ *fst* (A s) $\Longrightarrow$ (r, t) $\in$ *fst* (B s);
$\bigwedge r\ t\ s.$ (r, t) $\in$ *fst* (B s) $\Longrightarrow$ (r, t) $\in$ *fst* (A s);
$\bigwedge x.$ *snd* (A x) = *snd* (B x) $]\!]$
$\Longrightarrow$ (A :: ($'s$, $'a$) *nondet-monad*) = B
apply (*fastforce intro*!: *set-eqI prod-eqI*)
done

lemma *monad-state-eqI* [*intro*]:
$[\![$ $\bigwedge r\ t.$ (r, t) $\in$ *fst* (A s) $\Longrightarrow$ (r, t) $\in$ *fst* (B s');
$\bigwedge r\ t.$ (r, t) $\in$ *fst* (B s') $\Longrightarrow$ (r, t) $\in$ *fst* (A s);

snd (*A s*) = *snd* (*B s′*) ⟧
⟹ (*A* :: (′*s*, ′*a*) *nondet-monad*) *s* = *B s′*
apply (*fastforce intro*!: *set-eqI prod-eqI*)
done

25.1 General whileLoop reasoning

definition
whileLoop-terminatesE C B ≡ (λ*r*.
whileLoop-terminates (λ*r s*. *case r of Inr v* ⇒ *C v s* | - ⇒ *False*) (*lift B*) (*Inr r*))

lemma *whileLoop-cond-fail*:
⟦ ¬ *C x s* ⟧ ⟹ (*whileLoop C B x s*) = (*return x s*)
apply (*auto simp*: *return-def whileLoop-def*
intro: *whileLoop-results.intros*
whileLoop-terminates.intros
elim!: *whileLoop-results.cases*)
done

lemma *whileLoopE-cond-fail*:
⟦ ¬ *C x s* ⟧ ⟹ (*whileLoopE C B x s*) = (*returnOk x s*)
apply (*clarsimp simp*: *whileLoopE-def returnOk-def*)
apply (*auto intro*: *whileLoop-cond-fail*)
done

lemma *whileLoop-results-simps-no-move* [*simp*]:
shows ((*Some x*, *Some x*) ∈ *whileLoop-results C B*) = (¬ *C* (*fst x*) (*snd x*))
(**is** *?LHS x* = *?RHS x*)
proof (*rule iffI*)
assume *?LHS x*
then have (∃ *a*. *Some x* = *Some a*) ⟶ *?RHS* (*the* (*Some x*))
by (*induct rule*: *whileLoop-results.induct*, *auto*)
thus *?RHS x*
by *clarsimp*
next
assume *?RHS x*
thus *?LHS x*
by (*metis surjective-pairing whileLoop-results.intros*(*1*))
qed

lemma *whileLoop-unroll*:
(*whileLoop C B r*) = ((*condition* (*C r*) (*B r* >>= (*whileLoop C B*)) (*return r*)))
(**is** *?LHS r* = *?RHS r*)
proof −
have *cond-fail*: ⋀*r s*. ¬ *C r s* ⟹ *?LHS r s* = *?RHS r s*
apply (*subst whileLoop-cond-fail*, *simp*)
apply (*clarsimp simp*: *condition-def bind-def return-def*)
done

have *cond-pass*: $\bigwedge r\ s.\ C\ r\ s \Longrightarrow whileLoop\ C\ B\ r\ s = (B\ r >>= (whileLoop\ C\ B))\ s$
  apply (*rule monad-state-eqI*)
    apply (*clarsimp simp*: *whileLoop-def bind-def split-def*)
    apply (*subst* (*asm*) *whileLoop-results-simps-valid*)
    apply *fastforce*
   apply (*clarsimp simp*: *whileLoop-def bind-def split-def*)
   apply (*subst whileLoop-results.simps*)
   apply *fastforce*
  apply (*clarsimp simp*: *whileLoop-def bind-def split-def*)
  apply (*subst whileLoop-results.simps*)
  apply (*subst whileLoop-terminates.simps*)
  apply *fastforce*
  done

show *?thesis*
  apply (*rule ext*)
  apply (*metis cond-fail cond-pass condition-def*)
  done
qed

lemma *whileLoop-unroll'*:
  $(whileLoop\ C\ B\ r) = ((condition\ (C\ r)\ (B\ r)\ (return\ r)) >>= (whileLoop\ C\ B))$
 apply (*rule ext*)
 apply (*subst whileLoop-unroll*)
 apply (*clarsimp simp*: *condition-def bind-def return-def split-def*)
 apply (*subst whileLoop-cond-fail*, *simp*)
 apply (*clarsimp simp*: *return-def*)
 done

lemma *whileLoopE-unroll*:
  $(whileLoopE\ C\ B\ r) = ((condition\ (C\ r)\ (B\ r >>=E\ (whileLoopE\ C\ B))\ (returnOk\ r)))$
 apply (*rule ext*)
 apply (*unfold whileLoopE-def*)
 apply (*subst whileLoop-unroll*)
 apply (*clarsimp simp*: *whileLoopE-def bindE-def returnOk-def split*: *condition-splits*)
 apply (*clarsimp simp*: *lift-def*)
 apply (*rule-tac* $f=\lambda a.\ (B\ r >>= a)\ x$ **in** *arg-cong*)
 apply (*rule ext*)+
 apply (*clarsimp simp*: *lift-def split*: *sum.splits*)
 apply (*subst whileLoop-unroll*)
 apply (*subst condition-false*)
  apply *fastforce*
 apply (*clarsimp simp*: *throwError-def*)
 done

lemma *whileLoopE-unroll′*:
($whileLoopE\ C\ B\ r$) = ((*condition* ($C\ r$) ($B\ r$) (*returnOk* r)) >>=E (*whileLoopE* $C\ B$))
apply (*rule ext*)
apply (*subst whileLoopE-unroll*)
apply (*clarsimp simp*: *condition-def bindE-def bind-def returnOk-def return-def lift-def split-def*)
apply (*subst whileLoopE-cond-fail*, *simp*)
apply (*clarsimp simp*: *returnOk-def return-def*)
done

lemma *valid-make-schematic-post*:
$(\forall s0.\ \{\!|\ \lambda s.\ P\ s0\ s\ |\!\}\ f\ \{\!|\ \lambda rv\ s.\ Q\ s0\ rv\ s\ |\!\}) \Longrightarrow$
$\{\!|\ \lambda s.\ \exists s0.\ P\ s0\ s \wedge (\forall rv\ s'.\ Q\ s0\ rv\ s' \longrightarrow Q'\ rv\ s')\ |\!\}\ f\ \{\!|\ Q'\ |\!\}$
by (*auto simp add*: *valid-def no-fail-def split*: *prod.splits*)

lemma *validNF-make-schematic-post*:
$(\forall s0.\ \{\!|\ \lambda s.\ P\ s0\ s\ |\!\}\ f\ \{\!|\ \lambda rv\ s.\ Q\ s0\ rv\ s\ |\!\}!) \Longrightarrow$
$\{\!|\ \lambda s.\ \exists s0.\ P\ s0\ s \wedge (\forall rv\ s'.\ Q\ s0\ rv\ s' \longrightarrow Q'\ rv\ s')\ |\!\}\ f\ \{\!|\ Q'\ |\!\}!$
by (*auto simp add*: *valid-def validNF-def no-fail-def split*: *prod.splits*)

lemma *validE-make-schematic-post*:
$(\forall s0.\ \{\!|\ \lambda s.\ P\ s0\ s\ |\!\}\ f\ \{\!|\ \lambda rv\ s.\ Q\ s0\ rv\ s\ |\!\}, \{\!|\ \lambda rv\ s.\ E\ s0\ rv\ s\ |\!\}) \Longrightarrow$
$\{\!|\ \lambda s.\ \exists s0.\ P\ s0\ s \wedge (\forall rv\ s'.\ Q\ s0\ rv\ s' \longrightarrow Q'\ rv\ s')$
$\wedge (\forall rv\ s'.\ E\ s0\ rv\ s' \longrightarrow E'\ rv\ s')\ |\!\}\ f\ \{\!|\ Q'\ |\!\}, \{\!|\ E'\ |\!\}$
by (*auto simp add*: *validE-def valid-def no-fail-def split*: *prod.splits sum.splits*)

lemma *validE-NF-make-schematic-post*:
$(\forall s0.\ \{\!|\ \lambda s.\ P\ s0\ s\ |\!\}\ f\ \{\!|\ \lambda rv\ s.\ Q\ s0\ rv\ s\ |\!\}, \{\!|\ \lambda rv\ s.\ E\ s0\ rv\ s\ |\!\}!) \Longrightarrow$
$\{\!|\ \lambda s.\ \exists s0.\ P\ s0\ s \wedge (\forall rv\ s'.\ Q\ s0\ rv\ s' \longrightarrow Q'\ rv\ s')$
$\wedge (\forall rv\ s'.\ E\ s0\ rv\ s' \longrightarrow E'\ rv\ s')\ |\!\}\ f\ \{\!|\ Q'\ |\!\}, \{\!|\ E'\ |\!\}!$
by (*auto simp add*: *validE-NF-def validE-def valid-def no-fail-def split*: *prod.splits sum.splits*)

lemma *validNF-conjD1*: $\{\!|\ P\ |\!\}\ f\ \{\!|\ \lambda rv\ s.\ Q\ rv\ s \wedge Q'\ rv\ s\ |\!\}! \Longrightarrow \{\!|\ P\ |\!\}\ f\ \{\!|\ Q\ |\!\}!$
by (*fastforce simp*: *validNF-def valid-def no-fail-def*)

lemma *validNF-conjD2*: $\{\!|\ P\ |\!\}\ f\ \{\!|\ \lambda rv\ s.\ Q\ rv\ s \wedge Q'\ rv\ s\ |\!\}! \Longrightarrow \{\!|\ P\ |\!\}\ f\ \{\!|\ Q'\ |\!\}!$
by (*fastforce simp*: *validNF-def valid-def no-fail-def*)

end

theory *WP-Pre*
imports
Main
HOL−Eisbach.Eisbach-Tools

begin

named-theorems *wp-pre*

ML ‹
structure WP-Pre = struct

fun append-used-thm thm used-thms = used-thms := !used-thms @ [thm]

fun pre-tac ctxt pre-rules used-ref-option i t = let
fun append-thm used-thm thm =
if Option.isSome used-ref-option
then Seq.map (fn thm => (append-used-thm used-thm (Option.valOf used-ref-option); thm)) thm
else thm;
fun apply-rule t thm = append-thm t (resolve-tac ctxt [t] i thm)
val t2 = FIRST (map apply-rule pre-rules) t |> Seq.hd
val etac = TRY o eresolve-tac ctxt [@{thm FalseE}]
fun dummy-t2 - - = Seq.single t2
val t3 = (dummy-t2 THEN-ALL-NEW etac) i t |> Seq.hd
in if Thm.nprems-of t3 <> Thm.nprems-of t2
then Seq.empty else Seq.single t2 end
handle Option => Seq.empty

fun tac used-ref-option ctxt = let
val pres = Named-Theorems.get ctxt @{named-theorems wp-pre}
in pre-tac ctxt pres used-ref-option end

val method
= Args.context >> (fn - => fn ctxt => Method.SIMPLE-METHOD′ (tac NONE ctxt));
end
›

method-setup *wp-pre0* = ‹*WP-Pre.method*›
method *wp-pre* = *wp-pre0?*

definition
test-wp-pre :: *bool* ⇒ *bool* ⇒ *bool*
where
test-wp-pre P Q = $(P \longrightarrow Q)$

lemma *test-wp-pre-pre*[*wp-pre*]:
test-wp-pre P′ Q $\Longrightarrow (P \Longrightarrow P')$
$\Longrightarrow$ *test-wp-pre P Q*
by (*simp add*: *test-wp-pre-def*)

lemma *demo*:
test-wp-pre P P

apply *wp-pre0+*
 apply (*simp add*: *test-wp-pre-def*, *rule imp-refl*)
apply *simp*
done

end

theory *Datatype-Schematic*

imports
../ml−helpers/MLUtils
../ml−helpers/TermPatternAntiquote
begin

Introduces a method for improving unification outcomes for schematics with datatype expressions as parameters.

There are two variants: 1. In cases where a schematic is applied to a constant like *True*, we wrap the constant to avoid some undesirable unification candidates.

2. In cases where a schematic is applied to a constructor expression like *Some x* or (x, y), we supply selector expressions like *the* or *fst* to provide more unification candidates. This is only done if parameter that would be selected (e.g. x in *Some x*) contains bound variables which the schematic does not have as parameters.

In the "constructor expression" case, we let users supply additional constructor handlers via the 'datatype$_schematic$'$attribute.Themethodusesrulesofthefollowingform:$

$\bigwedge x1\ x2\ x3.\ getter\ (constructor\ x1\ x2\ x3) = x2$

These are essentially simp rules for simple "accessor" primrec functions, which are used to turn schematics like

$?P\ (constructor\ x1\ x2\ x3)$

into

$?P'\ x2\ (constructor\ x1\ x2\ x3)$.

ML ‹
— Anchor used to link error messages back to the documentation above.
val usage-pos = @{*here*};
›

definition
ds-id :: $'a \Rightarrow 'a$
where
ds-id = ($\lambda x.\ x$)

lemma *wrap-ds-id*:
x = *ds-id* x

by (*simp add*: *ds-id-def*)

ML ‹
structure Datatype-Schematic = struct

fun eq ((idx1, name1, thm1), (idx2, name2, thm2)) =
idx1 = idx2 andalso
name1 = name2 andalso
(Thm.full-prop-of thm1) aconv (Thm.full-prop-of thm2);

structure Datatype-Schematic-Data = Generic-Data
(
— Keys are names of datatype constructors (like (#)), values are ‘(index, function$_n$*ame*, *thm*)‘.
- ‘function$_n$*ame‘isthenameofan”accessor”functionthataccessespartoftheconstructorspecifiedbythekey(soth*
- ‘thm‘ is a theorem showing that the function accesses one of the arguments to the constructor (like *hd* (*?x21.0 # ?x22.0*) = *?x21.0*).
- ‘idx‘ is the index of the constructor argument that the accessor accesses. (eg. since ‘hd‘ accesses the first argument, ‘idx = 0‘; since ‘tl‘ accesses the second argument, ‘idx = 1‘).
*type T = ((int * string * thm) list) Symtab.table;*
val empty = Symtab.empty;
val extend = I;
val merge = Symtab.merge-list eq;
);

fun gen-att m =
Thm.declaration-attribute (fn thm => fn context =>
Datatype-Schematic-Data.map (m (Context.proof-of context) thm) context);

(gathers schematic applications from the goal. no effort is made*
to normalise bound variables here, since we'll always be comparing
elements within a compound application which will be at the same
*level as regards lambdas. *)*
fun gather-schem-apps (f $ x) insts = let
val (f, xs) = strip-comb (f $ x)
val insts = fold (gather-schem-apps) (f :: xs) insts
in if is-Var f then (f, xs) :: insts else insts end
| gather-schem-apps (Abs (-, -, t)) insts
= gather-schem-apps t insts
| gather-schem-apps - insts = insts

fun sfirst xs f = get-first f xs

fun get-action ctxt prop = let
val schem-insts = gather-schem-apps prop [];
val actions = Datatype-Schematic-Data.get (Context.Proof ctxt);
fun mk-sel selname T i = let
val (argTs, resT) = strip-type T
in Const (selname, resT --> nth argTs i) end

```
  in
    sfirst schem-insts
    (fn (var, xs) => sfirst (Library.tag-list 0 xs)
        (try (fn (idx, x) => let
            val (c, ys) = strip-comb x
            val (fname, T) = dest-Const c
            val acts = Symtab.lookup-list actions fname
            fun interesting arg = not (member Term.aconv-untyped xs arg)
                andalso exists (fn i => not (member (=) xs (Bound i)))
                    (Term.loose-bnos arg)
          in the (sfirst acts (fn (i, selname, thms) => if interesting (nth ys i)
            then SOME (var, idx, mk-sel selname T i, thms) else NONE))
          end)))
  end

fun get-bound-tac ctxt = SUBGOAL (fn (t, i) => case get-action ctxt t of
  SOME (Var ((nm, ix), T), idx, sel, thm) => (fn t => let
    val (argTs, -) = strip-type T
    val ix2 = Thm.maxidx-of t + 1
    val xs = map (fn (i, T) => Free (x ^ string-of-int i, T))
        (Library.tag-list 1 argTs)
    val nx = sel $ nth xs idx
    val v' = Var ((nm, ix2), fastype-of nx --> T)
    val inst-v = fold lambda (rev xs) (betapplys (v' $ nx, xs))
    val t' = Drule.infer-instantiate ctxt
        [((nm, ix), Thm.cterm-of ctxt inst-v)] t
    val t'' = Conv.fconv-rule (Thm.beta-conversion true) t'
  in safe-full-simp-tac (clear-simpset ctxt addsimps [thm]) i t'' end)
  | - => no-tac)

fun id-applicable (f $ x) = let
    val (f, xs) = strip-comb (f $ x)
    val here = is-Var f andalso exists is-Const xs
  in here orelse exists id-applicable (f :: xs) end
  | id-applicable (Abs (-, -, t)) = id-applicable t
  | id-applicable - = false

fun combination-conv cv1 cv2 ct =
  let
    val (ct1, ct2) = Thm.dest-comb ct
    val r1 = SOME (cv1 ct1) handle Option => NONE
    val r2 = SOME (cv2 ct2) handle Option => NONE
    fun mk - (SOME res) = res
      | mk ct NONE = Thm.reflexive ct
  in case (r1, r2) of
      (NONE, NONE) => raise Option
    | - => Thm.combination (mk ct1 r1) (mk ct2 r2)
  end
```

```
val wrap = mk-meta-eq @{thm wrap-ds-id}

fun wrap-const-conv - ct = if is-Const (Thm.term-of ct)
      andalso fastype-of (Thm.term-of ct) <> @{typ unit}
   then Conv.rewr-conv wrap ct
   else raise Option

fun combs-conv conv ctxt ct = case Thm.term-of ct of
   - $ - => combination-conv (combs-conv conv ctxt) (conv ctxt) ct
 | - => conv ctxt ct

fun wrap-conv ctxt ct = case Thm.term-of ct of
   Abs - => Conv.sub-conv wrap-conv ctxt ct
 | f $ x => if is-Var (head-of f) then combs-conv wrap-const-conv ctxt ct
   else if not (id-applicable (f $ x)) then raise Option
   else combs-conv wrap-conv ctxt ct
 | - => raise Option

fun CONVERSION-opt conv i t = CONVERSION conv i t
   handle Option => no-tac t

exception Datatype-Schematic-Error of Pretty.T;

fun apply-pos-markup pos text =
 let
   val props = Position.def-properties-of pos;
   val markup = Markup.properties props (Markup.entity  );
 in Pretty.mark-str (markup, text) end;

fun invalid-accessor ctxt thm : exn =
 Datatype-Schematic-Error ([
   Pretty.str Bad input theorem ',
   Syntax.pretty-term ctxt (Thm.full-prop-of thm),
   Pretty.str '. Click ,
   apply-pos-markup usage-pos *here*,
   Pretty.str  for info on the required rule format. ] |> Pretty.paragraph);

local
 fun dest-accessor' thm =
   case (thm |> Thm.full-prop-of |> HOLogic.dest-Trueprop) of
     @{term-pat ?fun-name ?data-pat = ?rhs} =>
       let
         val fun-name = Term.dest-Const fun-name |> fst;
         val (data-const, data-args) = Term.strip-comb data-pat;
         val data-vars = data-args |> map (Term.dest-Var #> fst);
         val rhs-var = rhs |> Term.dest-Var |> fst;
         val data-name = Term.dest-Const data-const |> fst;
        val rhs-idx = ListExtras.find-index (curry op = rhs-var) data-vars |> the;
       in (fun-name, data-name, rhs-idx) end;
```

```
in
  fun dest-accessor ctxt thm =
    case try dest-accessor' thm of
      SOME x => x
    | NONE => raise invalid-accessor ctxt thm;
end

fun add-rule ctxt thm data =
  let
    val (fun-name, data-name, idx) = dest-accessor ctxt thm;
    val entry = (data-name, (idx, fun-name, thm));
  in Symtab.insert-list eq entry data end;

fun del-rule ctxt thm data =
  let
    val (fun-name, data-name, idx) = dest-accessor ctxt thm;
    val entry = (data-name, (idx, fun-name, thm));
  in Symtab.remove-list eq entry data end;

val add = gen-att add-rule;
val del = gen-att del-rule;

fun wrap-tac ctxt = CONVERSION-opt (wrap-conv ctxt)

fun tac1 ctxt = REPEAT-ALL-NEW (get-bound-tac ctxt) THEN' (TRY o wrap-tac
ctxt)

fun tac ctxt = tac1 ctxt ORELSE' wrap-tac ctxt

val add-section =
  Args.add -- Args.colon >> K (Method.modifier add @{here});

val method =
 Method.sections [add-section] >> (fn - => fn ctxt => Method.SIMPLE-METHOD'
(tac ctxt));

end
›
```

setup ‹
```
  Attrib.setup
    @{binding datatype-schematic}
    (Attrib.add-del Datatype-Schematic.add Datatype-Schematic.del)
    Accessor rules to fix datatypes in schematics
›
```

method-setup *datatype-schem* = ‹
```
  Datatype-Schematic.method
›
```

declare *prod.sel*[*datatype-schematic*]
declare *option.sel*[*datatype-schematic*]
declare *list.sel*(*1*,*3*)[*datatype-schematic*]

locale *datatype-schem-demo* **begin**

lemma *handles-nested-constructors*:
  $\exists f.\ \forall y.\ f\ True\ (Some\ [x,\ (y,\ z)]) = y$
  apply (*rule exI*, *rule allI*)
  apply *datatype-schem*
  apply (*rule refl*)
  done

datatype *foo* =
    basic nat int
  | *another nat*

primrec *get-basic-0* **where**
  get-basic-0 (*basic x0 x1*) = *x0*

primrec *get-nat* **where**
    get-nat (*basic x -*) = *x*
  | *get-nat* (*another z*) = *z*

lemma *selectively-exposing-datatype-arugments*:
  notes *get-basic-0*.*simps*[*datatype-schematic*]
  shows $\exists x.\ \forall a\ b.\ x\ (basic\ a\ b) = a$
  apply (*rule exI*, (*rule allI*)+)
  apply *datatype-schem* — Only exposes ‘a‘ to the schematic.
  by (*rule refl*)

lemma *method-handles-primrecs-with-two-constructors*:
  shows $\exists x.\ \forall a\ b.\ x\ (basic\ a\ b) = a$
  apply (*rule exI*, (*rule allI*)+)
  apply (*datatype-schem add*: *get-nat*.*simps*)
  by (*rule refl*)

end

end

theory *Strengthen*
imports *Main*
begin

Implementation of the *strengthen* tool and the *mk-strg* attribute. See the theory *Strengthen-Demo* for a demonstration.

locale *strengthen-implementation* **begin**

definition *st P rel x y* = (*x* = *y* ∨ (*P* ∧ *rel x y*) ∨ (¬ *P* ∧ *rel y x*))

definition
st-prop1 :: *prop* ⇒ *prop* ⇒ *prop*
where
st-prop1 (*PROP P*) (*PROP Q*) ≡ (*PROP Q* ⟹ *PROP P*)

definition
st-prop2 :: *prop* ⇒ *prop* ⇒ *prop*
where
st-prop2 (*PROP P*) (*PROP Q*) ≡ (*PROP P* ⟹ *PROP Q*)

definition *failed* == *True*

definition *elim* :: *prop* ⇒ *prop*
where
elim (*P* :: *prop*) == *P*

definition *oblig* (*P* :: *prop*) == *P*

end

notation *strengthen-implementation.elim* ({*elim*| - |})
notation *strengthen-implementation.oblig* ({*oblig*| - |})
notation *strengthen-implementation.failed* (<*strg−failed*>)

syntax
-ap-strg-bool :: ['*a*, '*a*] => '*a* (- =*strg*<−−|=> -)
-ap-wkn-bool :: ['*a*, '*a*] => '*a* (- =*strg*−−>|=> -)
-ap-ge-bool :: ['*a*, '*a*] => '*a* (- =*strg*<=|=> -)
-ap-le-bool :: ['*a*, '*a*] => '*a* (- =*strg*>=|=> -)

syntax(*xsymbols*)
-ap-strg-bool :: ['*a*, '*a*] => '*a* (- =*strg*⟵|=> -)
-ap-wkn-bool :: ['*a*, '*a*] => '*a* (- =*strg*⟶|=> -)
-ap-ge-bool :: ['*a*, '*a*] => '*a* (- =*strg*≤|=> -)
-ap-le-bool :: ['*a*, '*a*] => '*a* (- =*strg*≥|=> -)

translations
P =*strg*⟵|=> *Q* == *CONST strengthen-implementation.st* (*CONST False*) (*CONST HOL.implies*) *P Q*
P =*strg*⟶|=> *Q* == *CONST strengthen-implementation.st* (*CONST True*) (*CONST HOL.implies*) *P Q*
P =*strg*≤|=> *Q* == *CONST strengthen-implementation.st* (*CONST False*) (*CONST Orderings.less-eq*) *P Q*
P =*strg*≥|=> *Q* == *CONST strengthen-implementation.st* (*CONST True*) (*CONST Orderings.less-eq*) *P Q*

context *strengthen-implementation* **begin**

lemma *failedI*:
  $<$*strg*$-$*failed*$>$
  by (*simp add*: *failed-def*)

lemma *strengthen-refl*:
  st P rel x x
  by (*simp add*: *st-def*)

lemma *st-prop-refl*:
  PROP (*st-prop1* (*PROP P*) (*PROP P*))
  PROP (*st-prop2* (*PROP P*) (*PROP P*))
  unfolding *st-prop1-def st-prop2-def*
  by *safe*

lemma *strengthenI*:
  rel x y $\Longrightarrow$ *st True rel x y*
  rel y x $\Longrightarrow$ *st False rel x y*
  by (*simp-all add*: *st-def*)

lemmas *imp-to-strengthen* = *strengthenI*(*2*)[**where** *rel*=($\longrightarrow$)]
lemmas *rev-imp-to-strengthen* = *strengthenI*(*1*)[**where** *rel*=($\longrightarrow$)]
lemmas *ord-to-strengthen* = *strengthenI*[**where** *rel*=($\leq$)]

lemma *use-strengthen-imp*:
  st False ($\longrightarrow$) *Q P* $\Longrightarrow$ *P* $\Longrightarrow$ *Q*
  by (*simp add*: *st-def*)

lemma *use-strengthen-prop-elim*:
  PROP P $\Longrightarrow$ *PROP* (*st-prop2* (*PROP P*) (*PROP Q*))
    $\Longrightarrow$ (*PROP Q* $\Longrightarrow$ *PROP R*) $\Longrightarrow$ *PROP R*
  unfolding *st-prop2-def*
  apply (*drule*(*1*) *meta-mp*)+
  apply *assumption*
  done

lemma *strengthen-Not*:
  st False rel x y $\Longrightarrow$ *st* ($\neg$ *True*) *rel x y*
  st True rel x y $\Longrightarrow$ *st* ($\neg$ *False*) *rel x y*
  by *auto*

lemmas *gather* =
    swap-prems-eq[**where** *A*=*PROP* (*Trueprop P*) **and** *B*=*PROP* (*elim Q*) **for** *P Q*]
    swap-prems-eq[**where** *A*=*PROP* (*Trueprop P*) **and** *B*=*PROP* (*oblig Q*) **for** *P Q*]

lemma *mk-True-imp*:
  $P \equiv True \longrightarrow P$
  by *simp*

lemma *narrow-quant*:
  $(\bigwedge x.\ PROP\ P \Longrightarrow PROP\ (Q\ x)) \equiv (PROP\ P \Longrightarrow (\bigwedge x.\ PROP\ (Q\ x)))$
  $(\bigwedge x.\ (R \longrightarrow S\ x)) \equiv PROP\ (Trueprop\ (R \longrightarrow (\forall x.\ S\ x)))$
  $(\bigwedge x.\ (S\ x \longrightarrow R)) \equiv PROP\ (Trueprop\ ((\exists x.\ S\ x) \longrightarrow R))$
  apply (*simp-all add*: *atomize-all*)
  apply *rule*
   apply *assumption*
  apply *assumption*
  done

ML ‹
```
structure Make-Strengthen-Rule = struct

fun binop-conv' cv1 cv2 = Conv.combination-conv (Conv.arg-conv cv1) cv2;

val mk-elim = Conv.rewr-conv @{thm elim-def[symmetric]}
val mk-oblig = Conv.rewr-conv @{thm oblig-def[symmetric]}

fun count-vars t = Term.fold-aterms
    (fn (Var v) => Termtab.map-default (Var v, 0) (fn x => x + 1)
        | - => I) t Termtab.empty

fun gather-to-imp ctxt drule pattern = let
    val pattern = (if drule then D :: pattern else pattern)
    fun inner pat ct = case (head-of (Thm.term-of ct), pat) of
        (@{term Pure.imp}, (E :: pat)) => binop-conv' mk-elim (inner pat) ct
      | (@{term Pure.imp}, (A :: pat)) => binop-conv' mk-elim (inner pat) ct
      | (@{term Pure.imp}, (O :: pat)) => binop-conv' mk-oblig (inner pat) ct
      | (@{term Pure.imp}, -) => binop-conv' (Object-Logic.atomize ctxt) (inner
(drop 1 pat)) ct
      | (-, []) => Object-Logic.atomize ctxt ct
      | (-, pat) => raise THM (gather-to-imp: leftover pattern: ^ commas pat, 1,
[])
    fun simp thms = Raw-Simplifier.rewrite ctxt false thms
    fun ensure-imp ct = case strip-comb (Thm.term-of ct) |> apsnd (map head-of)
     of
        (@{term Pure.imp}, -) => Conv.arg-conv ensure-imp ct
      | (@{term HOL.Trueprop}, [@{term HOL.implies}]) => Conv.all-conv ct
      | (@{term HOL.Trueprop}, -) => Conv.arg-conv (Conv.rewr-conv @{thm
mk-True-imp}) ct
      | - => raise CTERM (gather-to-imp, [ct])
    val gather = simp @{thms gather}
        then-conv (if drule then Conv.all-conv else simp @{thms atomize-conjL})
        then-conv simp @{thms atomize-imp}
        then-conv ensure-imp
```

```
  in Conv.fconv-rule (inner pattern then-conv gather) end

fun imp-list t = let
    val (x, y) = Logic.dest-implies t
  in x :: imp-list y end handle TERM - => [t]

fun mk-ex (xnm, T) t = HOLogic.exists-const T $ Term.lambda (Var (xnm, T))
t
fun mk-all (xnm, T) t = HOLogic.all-const T $ Term.lambda (Var (xnm, T)) t

fun quantify-vars ctxt drule thm = let
    val (lhs, rhs) = Thm.concl-of thm |> HOLogic.dest-Trueprop
      |> HOLogic.dest-imp
    val all-vars = count-vars (Thm.prop-of thm)
    val new-vars = count-vars (if drule then rhs else lhs)
     val quant = filter (fn v => Termtab.lookup new-vars v = Termtab.lookup
all-vars v)
            (Termtab.keys new-vars)
        |> map (Thm.cterm-of ctxt)
  in fold Thm.forall-intr quant thm
    |> Conv.fconv-rule (Raw-Simplifier.rewrite ctxt false @{thms narrow-quant})
  end

fun mk-strg (typ, pat) ctxt thm = let
    val drule = typ = D orelse typ = D'
    val imp = gather-to-imp ctxt drule pat thm
      |> (if typ = I' orelse typ = D'
          then quantify-vars ctxt drule else I)
  in if typ = I orelse typ = I'
    then imp RS @{thm imp-to-strengthen}
    else if drule then imp RS @{thm rev-imp-to-strengthen}
    else if typ = lhs then imp RS @{thm ord-to-strengthen(1)}
    else if typ = rhs then imp RS @{thm ord-to-strengthen(2)}
    else raise THM (mk-strg: unknown type:  ^ typ, 1, [thm])
 end

fun auto-mk ctxt thm = let
    val concl-C = try (fst o dest-Const o head-of
        o HOLogic.dest-Trueprop) (Thm.concl-of thm)
  in case (Thm.nprems-of thm, concl-C) of
    (-, SOME @{const-name failed}) => thm
  | (-, SOME @{const-name st}) => thm
  | (0, SOME @{const-name HOL.implies}) => (thm RS @{thm imp-to-strengthen}
      handle THM - => @{thm failedI})
  | - => mk-strg (I', []) ctxt thm
  end

fun mk-strg-args (SOME (typ, pat)) ctxt thm = mk-strg (typ, pat) ctxt thm
  | mk-strg-args NONE ctxt thm = auto-mk ctxt thm
```

val arg-pars = *Scan.option* (*Scan.first* (*map Args.$$$* [*I*, *I'*, *D*, *D'*, *lhs*, *rhs*])
  −− *Scan.repeat* (*Args.$$$ A* || *Args.$$$ E* || *Args.$$$ O* || *Args.$$$* -))

val attr-pars : *attribute context-parser*
    = (*Scan.lift arg-pars* −− *Args.context*)
        >> (*fn* (*args*, *ctxt*) => *Thm.rule-attribute* [] (*K* (*mk-strg-args args ctxt*)))

end
›

end

attribute-setup *mk-strg* = ‹*Make-Strengthen-Rule.attr-pars*›
    put rule in 'strengthen' form (*see theory Strengthen-Demo*)

Quick test.

lemmas *foo* = *nat.induct*[*mk-strg I O O*]
    nat.induct[*mk-strg D O*]
    nat.induct[*mk-strg I' E*]
    exI[*mk-strg I'*] *exI*[*mk-strg I*]

context *strengthen-implementation* **begin**

lemma *do-elim*:
  PROP P ⟹ *PROP elim* (*PROP P*)
  by (*simp add*: *elim-def*)

lemma *intro-oblig*:
  PROP P ⟹ *PROP oblig* (*PROP P*)
  by (*simp add*: *oblig-def*)

ML ‹

structure Strengthen = *struct*

structure Congs = *Theory-Data*
(*struct*
    type T = *thm list*
    val empty = []
    val extend = *I*
    val merge = *Thm.merge-thms*;
end);

val tracing = *Attrib.config-bool* @{*binding strengthen-trace*} (*K false*)

fun map-context-total f (*Context.Theory t*) = (*Context.Theory* (*f t*))
  | *map-context-total f* (*Context.Proof p*)

```
  = (Context.Proof (Context.raw-transfer (f (Proof-Context.theory-of p)) p))

val strg-add = Thm.declaration-attribute
      (fn thm => map-context-total (Congs.map (Thm.add-thm thm)));

val strg-del = Thm.declaration-attribute
      (fn thm => map-context-total (Congs.map (Thm.del-thm thm)));

val setup =
  Attrib.setup @{binding strg} (Attrib.add-del strg-add strg-del)
    strengthening congruence rules
    #> snd tracing;

fun goal-predicate t = let
    val gl = Logic.strip-assums-concl t
    val cn = head-of #> dest-Const #> fst
  in if cn gl = @{const-name oblig} then oblig
    else if cn gl = @{const-name elim} then elim
    else if cn gl = @{const-name st-prop1} then st-prop1
    else if cn gl = @{const-name st-prop2} then st-prop2
    else if cn (HOLogic.dest-Trueprop gl) = @{const-name st} then st
    else
  end handle TERM - =>

fun do-elim ctxt = SUBGOAL (fn (t, i) => if goal-predicate t = elim
    then eresolve-tac ctxt @{thms do-elim} i else all-tac)

fun final-oblig-strengthen ctxt = SUBGOAL (fn (t, i) => case goal-predicate t of
    oblig => resolve-tac ctxt @{thms intro-oblig} i
  | st => resolve-tac ctxt @{thms strengthen-refl} i
  | st-prop1 => resolve-tac ctxt @{thms st-prop-refl} i
  | st-prop2 => resolve-tac ctxt @{thms st-prop-refl} i
  | - => all-tac)

infix 1 THEN-TRY-ALL-NEW;

(* Like THEN-ALL-NEW but allows failure, although at least one subsequent
   method must succeed. *)
fun (tac1 THEN-TRY-ALL-NEW tac2) i st = let
    fun inner b j st = if i > j then (if b then all-tac else no-tac) st
      else ((tac2 j THEN inner true (j − 1)) ORELSE inner b (j − 1)) st
  in st |> (tac1 i THEN (fn st' =>
    inner false (i + Thm.nprems-of st' − Thm.nprems-of st) st')) end

fun maybe-trace-tac false - - = K all-tac
  | maybe-trace-tac true ctxt msg = SUBGOAL (fn (t, -) => let
    val tr = Pretty.big-list msg [Syntax.pretty-term ctxt t]
  in
    Pretty.writeln tr;
```

```
    all-tac
  end)

fun maybe-trace-rule false - - rl = rl
  | maybe-trace-rule true ctxt msg rl = let
    val tr = Pretty.big-list msg [Syntax.pretty-term ctxt (Thm.prop-of rl)]
  in
    Pretty.writeln tr;
    rl
  end

type params = {trace : bool, once : bool}

fun params once ctxt = {trace = Config.get ctxt (fst tracing), once = once}

fun apply-tac-as-strg ctxt (params : params) (tac : tactic)
  = SUBGOAL (fn (t, i) => case Logic.strip-assums-concl t of
      @{term Trueprop} $ (@{term st False (⟶)} $ x $ -)
      => let
    val triv = Thm.trivial (Thm.cterm-of ctxt (HOLogic.mk-Trueprop x))
    val trace = #trace params
  in
    fn thm => tac triv
        |> Seq.map (maybe-trace-rule trace ctxt apply-tac-as-strg: making strg)
        |> Seq.maps (Seq.try (Make-Strengthen-Rule.auto-mk ctxt))
        |> Seq.maps (fn str-rl => resolve-tac ctxt [str-rl] i thm)
  end | - => no-tac)

fun opt-tac f (SOME v) = f v
  | opt-tac - NONE = K no-tac

fun apply-strg ctxt (params : params) congs rules tac = EVERY' [
    maybe-trace-tac (#trace params) ctxt apply-strg,
    DETERM o TRY o resolve-tac ctxt @{thms strengthen-Not},
    DETERM o ((resolve-tac ctxt rules THEN-ALL-NEW do-elim ctxt)
        ORELSE' (opt-tac (apply-tac-as-strg ctxt params) tac)
        ORELSE' (resolve-tac ctxt congs THEN-TRY-ALL-NEW
            (fn i => apply-strg ctxt params congs rules tac i)))
]

fun setup-strg ctxt params thms meths = let
    val congs = Congs.get (Proof-Context.theory-of ctxt)
    val rules = map (Make-Strengthen-Rule.auto-mk ctxt) thms
    val tac = case meths of [] => NONE
      | - => SOME (FIRST (map (fn meth => Method.NO-CONTEXT-TACTIC
ctxt
        (Method.evaluate meth ctxt [])) meths))
  in apply-strg ctxt params congs rules tac
        THEN-ALL-NEW final-oblig-strengthen ctxt end
```

```
fun strengthen ctxt asm concl thms meths = let
    val strg = setup-strg ctxt (params false ctxt) thms meths
  in
    (if not concl then K no-tac
        else resolve-tac ctxt @{thms use-strengthen-imp} THEN' strg)
    ORELSE' (if not asm then K no-tac
        else eresolve-tac ctxt @{thms use-strengthen-prop-elim} THEN' strg)
  end

fun default-strengthen ctxt thms = strengthen ctxt false true thms []

val strengthen-args =
  Attrib.thms >> curry (fn (rules, ctxt) =>
    Method.CONTEXT-METHOD (fn - =>
      Method.RUNTIME (Method.CONTEXT-TACTIC
        (strengthen ctxt false true rules [] 1))
    )
  );

val strengthen-asm-args =
  Attrib.thms >> curry (fn (rules, ctxt) =>
    Method.CONTEXT-METHOD (fn - =>
      Method.RUNTIME (Method.CONTEXT-TACTIC
        (strengthen ctxt true false rules [] 1))
    )
  );

val strengthen-method-args =
  Method.text-closure >> curry (fn (meth, ctxt) =>
    Method.CONTEXT-METHOD (fn - =>
      Method.RUNTIME (Method.CONTEXT-TACTIC
        (strengthen ctxt true true [] [meth] 1))
    )
  );

end

›
```

end

setup *Strengthen.setup*

method-setup *strengthen* = ‹*Strengthen.strengthen-args*›
  strengthen the goal (see theory Strengthen-Demo)

method-setup *strengthen-asm* = ‹*Strengthen.strengthen-asm-args*›
  apply ''strengthen'' to weaken an assumption

method-setup *strengthen-method* = ‹*Strengthen.strengthen-method-args*›
use an argument method in ''strengthen'' sites

Important strengthen congruence rules.

context *strengthen-implementation* **begin**

lemma *strengthen-imp-imp*[*simp*]:
$st\ True\ (\longrightarrow)\ A\ B = (A \longrightarrow B)$
$st\ False\ (\longrightarrow)\ A\ B = (B \longrightarrow A)$
by (*simp-all add*: *st-def*)

abbreviation(*input*)
$st\text{-}ord\ t \equiv st\ t\ ((\leq) :: ('a :: preorder) \Rightarrow \text{-})$

lemma *strengthen-imp-ord*[*simp*]:
$st\text{-}ord\ True\ A\ B = (A \leq B)$
$st\text{-}ord\ False\ A\ B = (B \leq A)$
by (*auto simp add*: *st-def*)

lemma *strengthen-imp-conj* [*strg*]:
$[\![\ A' \Longrightarrow st\ F\ (\longrightarrow)\ B\ B';\ B \Longrightarrow st\ F\ (\longrightarrow)\ A\ A'\ ]\!]$
$\Longrightarrow st\ F\ (\longrightarrow)\ (A \wedge B)\ (A' \wedge B')$
by (*cases F*, *auto*)

lemma *strengthen-imp-disj* [*strg*]:
$[\![\ \neg\ A' \Longrightarrow st\ F\ (\longrightarrow)\ B\ B';\ \neg\ B \Longrightarrow st\ F\ (\longrightarrow)\ A\ A'\ ]\!]$
$\Longrightarrow st\ F\ (\longrightarrow)\ (A \vee B)\ (A' \vee B')$
by (*cases F*, *auto*)

lemma *strengthen-imp-implies* [*strg*]:
$[\![\ st\ (\neg\ F)\ (\longrightarrow)\ X\ X';\ X \Longrightarrow st\ F\ (\longrightarrow)\ Y\ Y'\ ]\!]$
$\Longrightarrow st\ F\ (\longrightarrow)\ (X \longrightarrow Y)\ (X' \longrightarrow Y')$
by (*cases F*, *auto*)

lemma *strengthen-all*[*strg*]:
$[\![\ \bigwedge x.\ st\ F\ (\longrightarrow)\ (P\ x)\ (Q\ x)\ ]\!]$
$\Longrightarrow st\ F\ (\longrightarrow)\ (\forall x.\ P\ x)\ (\forall x.\ Q\ x)$
by (*cases F*, *auto*)

lemma *strengthen-ex*[*strg*]:
$[\![\ \bigwedge x.\ st\ F\ (\longrightarrow)\ (P\ x)\ (Q\ x)\ ]\!]$
$\Longrightarrow st\ F\ (\longrightarrow)\ (\exists x.\ P\ x)\ (\exists x.\ Q\ x)$
by (*cases F*, *auto*)

lemma *strengthen-Ball*[*strg*]:
$[\![\ st\text{-}ord\ (Not\ F)\ S\ S';$
$\bigwedge x.\ x \in S \Longrightarrow st\ F\ (\longrightarrow)\ (P\ x)\ (Q\ x)\ ]\!]$
$\Longrightarrow st\ F\ (\longrightarrow)\ (\forall x \in S.\ P\ x)\ (\forall x \in S'.\ Q\ x)$

by (*cases F*, *auto*)

lemma *strengthen-Bex*[*strg*]:
$[\![$ *st-ord F S S'*;
$\bigwedge x.\ x \in S \Longrightarrow st\ F\ (\longrightarrow)\ (P\ x)\ (Q\ x)$ $]\!]$
$\Longrightarrow st\ F\ (\longrightarrow)\ (\exists x \in S.\ P\ x)\ (\exists x \in S'.\ Q\ x)$
by (*cases F*, *auto*)

lemma *strengthen-Collect*[*strg*]:
$[\![\ \bigwedge x.\ st\ F\ (\longrightarrow)\ (P\ x)\ (P'\ x)\ ]\!]$
$\Longrightarrow$ *st-ord F* $\{x.\ P\ x\}\ \{x.\ P'\ x\}$
by (*cases F*, *auto*)

lemma *strengthen-mem*[*strg*]:
$[\![$ *st-ord F S S'* $]\!]$
$\Longrightarrow st\ F\ (\longrightarrow)\ (x \in S)\ (x \in S')$
by (*cases F*, *auto*)

lemma *strengthen-ord*[*strg*]:
st-ord $(\neg\ F)\ x\ x' \Longrightarrow$ *st-ord* $F\ y\ y'$
$\Longrightarrow st\ F\ (\longrightarrow)\ (x \le y)\ (x' \le y')$
by (*cases F*, *simp-all*, (*metis order-trans*)+)

lemma *strengthen-strict-ord*[*strg*]:
st-ord $(\neg\ F)\ x\ x' \Longrightarrow$ *st-ord* $F\ y\ y'$
$\Longrightarrow st\ F\ (\longrightarrow)\ (x < y)\ (x' < y')$
by (*cases F*, *simp-all*, (*metis order-le-less-trans order-less-le-trans*)+)

lemma *strengthen-image*[*strg*]:
st-ord $F\ S\ S' \Longrightarrow$ *st-ord* $F\ (f\ \text{`}\ S)\ (f\ \text{`}\ S')$
by (*cases F*, *auto*)

lemma *strengthen-vimage*[*strg*]:
st-ord $F\ S\ S' \Longrightarrow$ *st-ord* $F\ (f\ -\text{`}\ S)\ (f\ -\text{`}\ S')$
by (*cases F*, *auto*)

lemma *strengthen-Int*[*strg*]:
st-ord $F\ A\ A' \Longrightarrow$ *st-ord* $F\ B\ B' \Longrightarrow$ *st-ord* $F\ (A \cap B)\ (A' \cap B')$
by (*cases F*, *auto*)

lemma *strengthen-Un*[*strg*]:
st-ord $F\ A\ A' \Longrightarrow$ *st-ord* $F\ B\ B' \Longrightarrow$ *st-ord* $F\ (A \cup B)\ (A' \cup B')$
by (*cases F*, *auto*)

lemma *strengthen-UN*[*strg*]:
st-ord $F\ A\ A' \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow$ *st-ord* $F\ (B\ x)\ (B'\ x))$
$\Longrightarrow$ *st-ord* $F\ (\bigcup x \in A.\ B\ x)\ (\bigcup x \in A'.\ B'\ x)$
by (*cases F*, *auto*)

lemma *strengthen-INT*[*strg*]:
$st\text{-}ord\ (\neg\ F)\ A\ A' \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow st\text{-}ord\ F\ (B\ x)\ (B'\ x))$
$\Longrightarrow st\text{-}ord\ F\ (\bigcap x \in A.\ B\ x)\ (\bigcap x \in A'.\ B'\ x)$
by (*cases F*, *auto*)

lemma *strengthen-imp-strengthen-prop*[*strg*]:
$st\ False\ (\longrightarrow)\ P\ Q \Longrightarrow PROP\ (st\text{-}prop1\ (Trueprop\ P)\ (Trueprop\ Q))$
$st\ True\ (\longrightarrow)\ P\ Q \Longrightarrow PROP\ (st\text{-}prop2\ (Trueprop\ P)\ (Trueprop\ Q))$
unfolding *st-prop1-def st-prop2-def*
by *auto*

lemma *st-prop-meta-imp*[*strg*]:
$PROP\ (st\text{-}prop2\ (PROP\ X)\ (PROP\ X'))$
$\Longrightarrow PROP\ (st\text{-}prop1\ (PROP\ Y)\ (PROP\ Y'))$
$\Longrightarrow PROP\ (st\text{-}prop1\ (PROP\ X \Longrightarrow PROP\ Y)\ (PROP\ X' \Longrightarrow PROP\ Y'))$
$PROP\ (st\text{-}prop1\ (PROP\ X)\ (PROP\ X'))$
$\Longrightarrow PROP\ (st\text{-}prop2\ (PROP\ Y)\ (PROP\ Y'))$
$\Longrightarrow PROP\ (st\text{-}prop2\ (PROP\ X \Longrightarrow PROP\ Y)\ (PROP\ X' \Longrightarrow PROP\ Y'))$
unfolding *st-prop1-def st-prop2-def*
by (*erule meta-mp* | *assumption*)+

lemma *st-prop-meta-all*[*strg*]:
$(\bigwedge x.\ PROP\ (st\text{-}prop1\ (PROP\ (X\ x))\ (PROP\ (X'\ x))))$
$\Longrightarrow PROP\ (st\text{-}prop1\ (\bigwedge x.\ PROP\ (X\ x))\ (\bigwedge x.\ PROP\ (X'\ x)))$
$(\bigwedge x.\ PROP\ (st\text{-}prop2\ (PROP\ (X\ x))\ (PROP\ (X'\ x))))$
$\Longrightarrow PROP\ (st\text{-}prop2\ (\bigwedge x.\ PROP\ (X\ x))\ (\bigwedge x.\ PROP\ (X'\ x)))$
unfolding *st-prop1-def st-prop2-def*
apply (*rule Pure.asm-rl*)
apply (*erule meta-allE*, *erule meta-mp*)
apply *assumption*
apply (*rule Pure.asm-rl*)
apply (*erule meta-allE*, *erule meta-mp*)
apply *assumption*
done

end

lemma *imp-consequent*:
$P \longrightarrow Q \longrightarrow P$ **by** *simp*

Test cases.

lemma
assumes x: $\bigwedge x.\ P\ x \longrightarrow Q\ x$
shows $\{x.\ x \neq None \wedge P\ (the\ x)\} \subseteq \{y.\ \forall x.\ y = Some\ x \longrightarrow Q\ x\}$
apply (*strengthen x*)
apply *clarsimp*
done

locale *strengthen-silly-test* **begin**

definition
  silly :: *nat* ⇒ *nat* ⇒ *bool*
where
  silly x y = (*x* ≤ *y*)

lemma *silly-trans*:
  silly x y ⟹ *silly y z* ⟹ *silly x z*
  by (*simp add*: *silly-def*)

lemma *silly-refl*:
  silly x x
  by (*simp add*: *silly-def*)

lemma *foo*:
  silly x y ⟹ *silly a b* ⟹ *silly b c*
    ⟹ *silly x y* ∧ (∀ *x* :: *nat*. *silly a c*)
  using [[*strengthen-trace* = *true*]]
  apply (*strengthen silly-trans*[*mk-strg I E*])+
  apply (*strengthen silly-refl*)
  apply *simp*
  done

lemma *foo-asm*:
  silly x y ⟹ *silly y z*
    ⟹ (*silly x z* ⟹ *silly a b*) ⟹ *silly z z* ⟹ *silly a b*
  apply (*strengthen-asm silly-trans*[*mk-strg I A*])
  apply (*strengthen-asm silly-trans*[*mk-strg I A*])
  apply *simp*
  done

lemma *foo-method*:
  silly x y ⟹ *silly a b* ⟹ *silly b c*
    ⟹ *silly x y* ∧ (∀ *x* :: *nat*. *z* ⟶ *silly a c*)
  using [[*strengthen-trace* = *true*]]
  apply *simp*
  apply (*strengthen-method* ‹*rule silly-trans*›)
  apply (*strengthen-method* ‹*rule exI*[*where x*=*b*]›)
  apply *simp*
  done

end
end

theory *WPFix*

imports

../*Datatype-Schematic*
../*Strengthen*

begin

WPFix handles four issues which are annoying with precondition schematics: 1. Schematics in obligation (postcondition) positions which remain unset after goals are solved. They should be instantiated to True. 2. Schematics which appear in multiple precondition positions. They should be instantiated to a conjunction and then separated. 3/4. Schematics applied to datatype expressions such as *True* or *Some x*. for details.

lemma *use-strengthen-prop-intro*:
PROP P $\Longrightarrow$ *PROP* (*strengthen-implementation.st-prop1* (*PROP Q*) (*PROP P*))
$\Longrightarrow$ *PROP Q*
unfolding *strengthen-implementation.st-prop1-def*
apply (*drule*(*1*) *meta-mp*)+
apply *assumption*
done

definition
target-var :: *int* $\Rightarrow$ *'a* $\Rightarrow$ *'a*
where
target-var n x = *x*

lemma *strengthen-to-conjunct1-target*:
strengthen-implementation.st True ($\longrightarrow$)
(*target-var n* ($P \wedge Q$)) (*target-var n P*)
by (*simp add*: *strengthen-implementation.st-def target-var-def*)

lemma *strengthen-to-conjunct2-target-trans*:
strengthen-implementation.st True ($\longrightarrow$)
(*target-var n Q*) *R*
$\Longrightarrow$ *strengthen-implementation.st True* ($\longrightarrow$)
(*target-var n* ($P \wedge Q$)) *R*
by (*simp add*: *strengthen-implementation.st-def target-var-def*)

lemma *target-var-drop-func*:
target-var n f = (λx. *target-var n* (*f x*))
by (*simp add*: *target-var-def*)

named-theorems *wp-fix-strgs*

lemma *strg-target-to-true*:
strengthen-implementation.st F ($\longrightarrow$) (*target-var n True*) *True*
by (*simp add*: *target-var-def strengthen-implementation.strengthen-refl*)

ML ‹
structure WPFix = *struct*

```
val st-refl = @{thm strengthen-implementation.strengthen-refl}
val st-refl-True = @{thm strengthen-implementation.strengthen-refl[where x=True]}
val st-refl-target-True = @{thm strg-target-to-true}
val st-refl-non-target
  = @{thm strengthen-implementation.strengthen-refl[where x=target-var (−1) v
for v]}

val conv-to-target = mk-meta-eq @{thm target-var-def[symmetric]}

val tord = Term-Ord.fast-term-ord
fun has-var vars t = not (null (Ord-List.inter tord vars
      (Ord-List.make tord (map Var (Term.add-vars t [])))))

fun get-vars prop = map Var (Term.add-vars prop [])
  |> Ord-List.make tord
  |> filter (fn v => snd (strip-type (fastype-of v)) = HOLogic.boolT)

val st-intro = @{thm use-strengthen-prop-intro}
val st-not = @{thms strengthen-implementation.strengthen-Not}
val st-conj2-trans = @{thm strengthen-to-conjunct2-target-trans}
val st-conj1 = @{thm strengthen-to-conjunct1-target}

(∗ assumes Strengthen.goal-predicate g is st ∗)
fun dest-strg g = case Strengthen.goal-predicate g of
  st => (case HOLogic.dest-Trueprop (Logic.strip-assums-concl g) of
      (Const - $ mode $ rel $ lhs $ rhs) => (st, SOME (mode, rel, lhs, rhs))
    | - => error (dest-strg  ^ @{make-string} g)
  )
 | nm => (nm, NONE)

fun get-target (Const (@{const-name target-var}, -) $ n $ -)
 = (try (HOLogic.dest-number #> snd) n)
 | get-target - = NONE

fun is-target P t = case get-target t of NONE => false
 | SOME v => P v

fun is-target-head P (f $ v) = is-target P (f $ v) orelse is-target-head P f
 | is-target-head - - = false

fun has-target P (f $ v) = is-target P (f $ v)
  orelse has-target P f orelse has-target P v
 | has-target P (Abs (-, -, t)) = has-target P t
 | has-target - - = false

fun apply-strgs congs ctxt = SUBGOAL (fn (t, i) => case
      dest-strg t of
  (st-prop1, -) => resolve-tac ctxt congs i
```

```
  | (st-prop2, -) => resolve-tac ctxt congs i
  | (st, SOME (-, -, lhs, -)) => resolve-tac ctxt st-not i
    ORELSE eresolve-tac ctxt [thin-rl] i
    ORELSE resolve-tac ctxt [st-refl-non-target] i
    ORELSE (if is-target-head (fn v => v >= 0) lhs
        then no-tac
        else if not (has-target (fn v => v >= 0) lhs)
        then resolve-tac ctxt [st-refl] i
        else if is-Const (head-of lhs)
        then (resolve-tac ctxt congs i ORELSE resolve-tac ctxt [st-refl] i)
        else resolve-tac ctxt [st-refl] i
    )
  | - => no-tac
  )

fun strg-proc ctxt = let
    val congs1 = Named-Theorems.get ctxt @{named-theorems wp-fix-strgs}
    val thy = Proof-Context.theory-of ctxt
    val congs2 = Strengthen.Congs.get thy
    val strg = apply-strgs (congs1 @ congs2) ctxt
  in REPEAT-ALL-NEW strg end

fun target-var-conv vars ctxt ct = case Thm.term-of ct of
    Abs - => Conv.sub-conv (target-var-conv vars) ctxt ct
  | Var v => Conv.rewr-conv (Drule.infer-instantiate ctxt
        [((n, 1), Thm.cterm-of ctxt (HOLogic.mk-number @{typ int}
            (find-index (fn v2 => v2 = Var v) vars)))] conv-to-target) ct
  | - $ - => Datatype-Schematic.combs-conv (target-var-conv vars) ctxt ct
  | - => raise Option

fun st-intro-tac ctxt = CSUBGOAL (fn (ct, i) => fn thm => let
        val intro = Drule.infer-instantiate ctxt [((Q, 0), ct)]
          (Thm.incr-indexes (Thm.maxidx-of thm + 1) st-intro)
      in compose-tac ctxt (false, intro, 2) i
      end thm)

fun intro-tac ctxt vs = SUBGOAL (fn (t, i) => if has-var vs t
    then CONVERSION (target-var-conv vs ctxt) i
        THEN CONVERSION (Simplifier.full-rewrite (clear-simpset ctxt
            addsimps @{thms target-var-drop-func}
        )) i
        THEN st-intro-tac ctxt i
    else all-tac)

fun classify v thm = let
    val has-t = has-target (fn v' => v' = v)
    val relevant = filter (has-t o fst)
        (Thm.prems-of thm ~~ (1 upto Thm.nprems-of thm))
        |> map (apfst (Logic.strip-assums-concl #> Envir.beta-eta-contract))
```

```
    fun class t = case dest-strg t of
        (st, SOME (@{term True}, @{term (-->)}, lhs, -))
          => if has-t lhs then SOME true else NONE
      | (st, SOME (@{term False}, @{term (-->)}, lhs, -))
          => if has-t lhs then SOME false else NONE
      | - => NONE
    val classn = map (apfst class) relevant
    fun get k = map snd (filter (fn (k', -) => k' = k) classn)
  in if (null relevant) then NONE
    else if not (null (get NONE))
    then NONE
    else if null (get (SOME true))
    then SOME (to-true, map snd relevant)
    else if length (get (SOME true)) > 1
    then SOME (to-conj, get (SOME true))
    else NONE
  end

fun ONGOALS tac is = let
    val is = rev (sort int-ord is)
  in EVERY (map tac is) end

fun act-on ctxt (to-true, is)
    = ONGOALS (resolve-tac ctxt [st-refl-target-True]) is
  | act-on ctxt (to-conj, is)
    = ONGOALS (resolve-tac ctxt [st-conj2-trans]) (drop 1 is)
      THEN (if length is > 2 then act-on ctxt (to-conj, drop 1 is)
        else ONGOALS (resolve-tac ctxt [st-refl]) (drop 1 is))
      THEN ONGOALS (resolve-tac ctxt [st-conj1]) (take 1 is)
  | act-on - (s, -) = error (act-on: ^ s)

fun act ctxt check vs thm = let
    val acts = map-filter (fn v => classify v thm) vs
  in if null acts
    then (if check then no-tac else all-tac) thm
    else (act-on ctxt (hd acts) THEN act ctxt false vs) thm end

fun cleanup ctxt = SUBGOAL (fn (t, i) => case Strengthen.goal-predicate t of
    st => resolve-tac ctxt [st-refl] i
  | - => all-tac)

fun tac ctxt = SUBGOAL (fn (t, -) => let
    val vs = get-vars t
  in if null vs then no-tac else ALLGOALS (intro-tac ctxt vs)
    THEN ALLGOALS (TRY o strg-proc ctxt)
    THEN act ctxt true (0 upto (length vs - 1))
    THEN ALLGOALS (cleanup ctxt)
    THEN Local-Defs.unfold-tac ctxt @{thms target-var-def}
  end)
```

fun both-tac ctxt = (*Datatype-Schematic.tac ctxt THEN′* (*TRY o tac ctxt*))
ORELSE′ tac ctxt

val method =
Method.sections [*Datatype-Schematic.add-section*] >>
(*fn* - => *fn ctxt* => *Method.SIMPLE-METHOD′* (*both-tac ctxt*));

end
⟩

method-setup *wpfix* = ⟨*WPFix.method*⟩

lemma *demo1*:
$(\exists\, Ia\ Ib\ Ic\ Id\ Ra.$
$(Ia\ (Suc\ 0) \longrightarrow Qa)$
$\wedge\ (Ib \longrightarrow Qb)$
$\wedge\ (Ic \longrightarrow Ra)$
$\wedge\ (Id \longrightarrow Qc)$
$\wedge\ (Id \longrightarrow Qd)$
$\wedge\ (Qa \wedge Qb \wedge Qc \wedge Qd \longrightarrow Ia\ v \wedge Ib \wedge Ic \wedge Id))$
apply (*intro exI conjI impI*)

apply (*wpfix* | *assumption*)+
apply *auto*
done

lemma *demo2*:
assumes *P*: $\bigwedge x.\ P\ (x + Suc\ x) \longrightarrow R\ (Inl\ x)$
$\bigwedge x.\ P\ ((x * 2) - 1) \longrightarrow R\ (Inr\ x)$
assumes *P17*: *P 17*
shows $\exists\, I.\ I\ (Some\ 9)$
$\wedge\ (\forall\, x.\ I\ x \longrightarrow (case\ x\ of\ None \Rightarrow R\ (Inl\ 8) \mid Some\ y \Rightarrow R\ (Inr\ y)))$
$\wedge\ (\forall\, x.\ I\ x \longrightarrow (case\ x\ of\ None \Rightarrow R\ (Inr\ 9) \mid Some\ y \Rightarrow R\ (Inl\ (y - 1))))$
apply (*intro exI conjI*[*rotated*] *allI*)
apply (*case-tac x*; *simp*)
apply *wpfix*
apply (*rule P*)
apply *wpfix*
apply (*rule P*)
apply (*case-tac x*; *simp*)
apply *wpfix*
apply (*rule P*)
apply *wpfix*
apply (*rule P*)
apply (*simp add*: *P17*)
done

— Shows how to use *datatype-schematic* rules as "accessors".

lemma (**in** *datatype-schem-demo*) *demo3*:
  $\exists x. \forall a\ b.\ x\ (basic\ a\ b) = a$
  apply (*rule exI*, (*rule allI*)+)
  apply (*wpfix add*: *get-basic-0.simps*) — Only exposes 'a' to the schematic.
  by (*rule refl*)

end

theory *WP*
imports
  WP-Pre
  WPFix
  ../../Apply-Debug
  ../../ml−helpers/MLUtils
begin

definition
  $triple\text{-}judgement :: ('a \Rightarrow bool) \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$
where
 $triple\text{-}judgement\ pre\ body\ property = (\forall s.\ pre\ s \longrightarrow property\ s\ body)$

definition
  $postcondition :: ('r \Rightarrow 's \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow ('r \times 's)\ set)$
  $\Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
where
 $postcondition\ P\ f = (\lambda a\ b.\ \forall (rv, s) \in f\ a\ b.\ P\ rv\ s)$

definition
  $postconditions :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool)$
where
 $postconditions\ P\ Q = (\lambda a\ b.\ P\ a\ b \wedge Q\ a\ b)$

lemma *conj-TrueI*: $P \Longrightarrow True \wedge P$ **by** *simp*
lemma *conj-TrueI2*: $P \Longrightarrow P \wedge True$ **by** *simp*

ML-file *WP−method.ML*

declare [[*wp-trace* = *false*]]

setup *WeakestPre.setup*

method-setup *wp* = ‹*WeakestPre.apply-wp-args*›
  applies weakest precondition rules

end

theory *WPC*

imports *WP-Pre*
keywords *wpc-setup* :: *thy-decl*

begin

definition
wpc-helper :: $(('a \Rightarrow bool) \times 'b\ set)$
$\Rightarrow (('a \Rightarrow bool) \times 'b\ set) \Rightarrow bool \Rightarrow bool$ **where**
wpc-helper $\equiv \lambda(P,\ P')\ (Q,\ Q')\ R.\ ((\forall s.\ P\ s \longrightarrow Q\ s) \wedge P' \subseteq Q') \longrightarrow R$

lemma *wpc-conj-process*:
$[\![$ *wpc-helper* $(P,\ P')\ (A,\ A')\ C$; *wpc-helper* $(P,\ P')\ (B,\ B')\ D\ ]\!]$
$\Longrightarrow$ *wpc-helper* $(P,\ P')\ (\lambda s.\ A\ s \wedge B\ s,\ A' \cap B')\ (C \wedge D)$
by (*clarsimp simp add*: *wpc-helper-def*)

lemma *wpc-all-process*:
$[\![\ \bigwedge x.$ *wpc-helper* $(P,\ P')\ (Q\ x,\ Q'\ x)\ (R\ x)\ ]\!]$
$\Longrightarrow$ *wpc-helper* $(P,\ P')\ (\lambda s.\ \forall x.\ Q\ x\ s,\ \{s.\ \forall x.\ s \in Q'\ x\})\ (\forall x.\ R\ x)$
by (*clarsimp simp*: *wpc-helper-def subset-iff*)

lemma *wpc-all-process-very-weak*:
$[\![\ \bigwedge x.$ *wpc-helper* $(P,\ P')\ (Q,\ Q')\ (R\ x)\ ]\!] \Longrightarrow$ *wpc-helper* $(P,\ P')\ (Q,\ Q')\ (\forall x.$
$R\ x)$
by (*clarsimp simp*: *wpc-helper-def*)

lemma *wpc-imp-process*:
$[\![\ Q \Longrightarrow$ *wpc-helper* $(P,\ P')\ (R,\ R')\ S\ ]\!]$
$\Longrightarrow$ *wpc-helper* $(P,\ P')\ (\lambda s.\ Q \longrightarrow R\ s,\ \{s.\ Q \longrightarrow s \in R'\})\ (Q \longrightarrow S)$
by (*clarsimp simp add*: *wpc-helper-def subset-iff*)

lemma *wpc-imp-process-weak*:
$[\![$ *wpc-helper* $(P,\ P')\ (R,\ R')\ S\ ]\!] \Longrightarrow$ *wpc-helper* $(P,\ P')\ (R,\ R')\ (Q \longrightarrow S)$
by (*clarsimp simp add*: *wpc-helper-def*)

lemmas *wpc-processors*
= *wpc-conj-process wpc-all-process wpc-imp-process*
lemmas *wpc-weak-processors*
= *wpc-conj-process wpc-all-process wpc-imp-process-weak*
lemmas *wpc-vweak-processors*
= *wpc-conj-process wpc-all-process-very-weak wpc-imp-process-weak*

lemma *wpc-helperI*:
wpc-helper $(P,\ P')\ (P,\ P')\ Q \Longrightarrow Q$
by (*simp add*: *wpc-helper-def*)

lemma *wpc-foo*: $[\![$ *undefined* x; *False* $]\!] \Longrightarrow P\ x$
by *simp*

lemma *foo*:

assumes *foo-elim*: $\bigwedge P\ Q\ h.\ [\![\ foo\ Q\ h;\ \bigwedge s.\ P\ s \Longrightarrow Q\ s\ ]\!] \Longrightarrow foo\ P\ h$
shows
$[\![\ \bigwedge x.\ foo\ (Q\ x)\ (f\ x);\ foo\ R\ g\ ]\!] \Longrightarrow$
$\quad foo\ (\lambda s.\ (\forall x.\ Q\ x\ s) \wedge (y = None \longrightarrow R\ s))$
$\quad\quad (case\ y\ of\ Some\ x \Rightarrow f\ x \mid None \Rightarrow g)$
by (*auto split*: *option.split intro*: *foo-elim*)

ML ‹

```
signature WPC = sig
  exception WPCFailed of string * term list * thm list;

  val foo-thm: thm;
  val iffd2-thm: thm;
  val wpc-helperI: thm;

  val instantiate-concl-pred: Proof.context -> cterm -> thm -> thm;

  val detect-term: Proof.context -> int -> thm -> cterm -> (cterm * term)
list;
  val detect-terms: Proof.context -> (term -> cterm -> thm -> int -> tactic)
-> int -> tactic;

  val split-term: thm list -> Proof.context -> term -> cterm -> thm -> int
-> tactic;

  val wp-cases-tac: thm list -> Proof.context -> int -> tactic;
  val wp-debug-tac: thm list -> Proof.context -> int -> tactic;
  val wp-cases-method: thm list -> (Proof.context -> Method.method) context-parser;

end;

structure WPCPredicateAndFinals = Theory-Data
(struct
    type T = (cterm * thm) list
    val empty = []
    val extend = I
    fun merge (xs, ys) =
        (* Order of predicates is important, so we can't reorder *)
        let val tms = map (Thm.term-of o fst) xs
            fun inxs x = exists (fn y => x aconv y) tms
            val ys' = filter (not o inxs o Thm.term-of o fst) ys
        in
            xs @ ys'
        end
end);

structure WeakestPreCases : WPC =
struct
```

```
exception WPCFailed of string * term list * thm list;

val iffd2-thm = @{thm iffD2};
val wpc-helperI = @{thm wpc-helperI};
val foo-thm = @{thm wpc-foo};

(* it looks like cterm-instantiate would do the job better,
   but this handles the case where ?'a must be instantiated
   to ?'a × ?'b *)
fun instantiate-concl-pred ctxt pred thm =
let
  val get-concl-pred  = (fst o strip-comb o HOLogic.dest-Trueprop o Thm.concl-of);
  val get-concl-predC = (Thm.cterm-of ctxt o get-concl-pred);

  val get-pred-tvar   = domain-type o Thm.typ-of o Thm.ctyp-of-cterm;
  val thm-pred        = get-concl-predC thm;
  val thm-pred-tvar   = Term.dest-TVar (get-pred-tvar thm-pred);
  val pred-tvar       = Thm.ctyp-of ctxt (get-pred-tvar pred);

  val thm2            = Thm.instantiate ([(thm-pred-tvar, pred-tvar)], []) thm;

  val thm2-pred       = Term.dest-Var (get-concl-pred thm2);
in
  Thm.instantiate ([], [(thm2-pred, pred)]) thm2
end;

fun detect-term ctxt n thm tm =
let
  val foo-thm-tm   = instantiate-concl-pred ctxt tm foo-thm;
  val matches      = resolve-tac ctxt [foo-thm-tm] n thm;
  val outcomes     = Seq.list-of matches;
  val get-goalterm = (HOLogic.dest-Trueprop o Logic.strip-assums-concl
                      o Envir.beta-eta-contract o hd o Thm.prems-of);
  val get-argument = hd o snd o strip-comb;
in
  map (pair tm o get-argument o get-goalterm) outcomes
end;

fun detect-terms ctxt tactic2 n thm =
let
  val pfs          = WPCPredicateAndFinals.get (Proof-Context.theory-of ctxt);
  val detects      = map (fn (tm, rl) => (detect-term ctxt n thm tm, rl)) pfs;
  val detects2     = filter (not o null o fst) detects;
  val ((pred, arg), fin)  = case detects2 of
                            [] => raise WPCFailed (detect-terms: no match, [], [thm])
                             | ((d3, fin) :: -) => (hd d3, fin)
in
  tactic2 arg pred fin n thm
```

```
end;

(* give each rule in the list one possible resolution outcome *)
fun resolve-each-once-tac ctxt thms i
    = fold (curry (APPEND'))
        (map (DETERM oo resolve-tac ctxt o single) thms)
        (K no-tac) i

fun resolve-single-tac ctxt rules n thm =
  case Seq.chop 2 (resolve-each-once-tac ctxt rules n thm)
  of ([], -) => raise WPCFailed
                        (resolve-single-tac: no rules could apply,
                         [], thm :: rules)
   | (- :: - :: -, -) => raise WPCFailed
                        (resolve-single-tac: multiple rules applied,
                         [], thm :: rules)
   | ([x], -) => Seq.single x;

fun split-term processors ctxt target pred fin =
let
  val hdTarget       = head-of target;
  val (constNm, -)  = dest-Const hdTarget handle TERM (-, tms)
                      => raise WPCFailed (split-term: couldn't dest-Const, tms, []);
  val split = case (Ctr-Sugar.ctr-sugar-of-case ctxt constNm) of
      SOME sugar => #split sugar
    | - => raise WPCFailed (split-term: not a case, [hdTarget], []);
  val subst          = split RS iffd2-thm;
  val subst2         = instantiate-concl-pred ctxt pred subst;
in
 (resolve-tac ctxt [subst2])
   THEN'
 (resolve-tac ctxt [wpc-helperI])
   THEN'
 (REPEAT-ALL-NEW (resolve-tac ctxt processors)
    THEN-ALL-NEW
  resolve-single-tac ctxt [fin])
end;

(* n.b. need to concretise the lazy sequence via a list to ensure exceptions
  have been raised already and catch them *)
fun wp-cases-tac processors ctxt n thm =
  detect-terms ctxt (split-term processors ctxt) n thm
      |> Seq.list-of |> Seq.of-list
    handle WPCFailed - => no-tac thm;

fun wp-debug-tac processors ctxt n thm =
  detect-terms ctxt (split-term processors ctxt) n thm
      |> Seq.list-of |> Seq.of-list
     handle WPCFailed e => (warning (@{make-string} (WPCFailed e)); no-tac
```

thm);

```
fun wp-cases-method processors = Scan.succeed (fn ctxt =>
  Method.SIMPLE-METHOD' (wp-cases-tac processors ctxt));

local structure P = Parse and K = Keyword in

fun add-wpc tm thm lthy = let
  val ctxt = Local-Theory.target-of lthy
  val tm' = (Syntax.read-term ctxt tm) |> Thm.cterm-of ctxt o Logic.varify-global
  val thm' = Proof-Context.get-thm ctxt thm
in
  Local-Theory.background-theory (WPCPredicateAndFinals.map (fn xs => (tm',
thm') :: xs)) lthy
end;

val - =
    Outer-Syntax.command
        @{command-keyword wpc-setup}
        Add wpc stuff
          (P.term -- P.name >> (fn (tm, thm) => Toplevel.local-theory NONE
NONE (add-wpc tm thm)))

end;
end;
```

⟩

ML ⟨

```
val wp-cases-tactic-weak = WeakestPreCases.wp-cases-tac @{thms wpc-weak-processors};
val wp-cases-method-strong = WeakestPreCases.wp-cases-method @{thms wpc-processors};
val wp-cases-method-weak  = WeakestPreCases.wp-cases-method @{thms wpc-weak-processors};
val wp-cases-method-vweak  = WeakestPreCases.wp-cases-method @{thms wpc-vweak-processors};
```

⟩

method-setup *wpc0* = ⟨*wp-cases-method-strong*⟩
  case splitter for weakest−precondition proofs

method-setup *wpcw0* = ⟨*wp-cases-method-weak*⟩
  weak−form case splitter for weakest−precondition proofs

method *wpc* = (*wp-pre*, *wpc0*)
method *wpcw* = (*wp-pre*, *wpcw0*)

definition
  wpc-test :: $'a$ *set* ⇒ ($'a$ × $'b$) *set* ⇒ $'b$ *set* ⇒ *bool*
  where

wpc-test P R S $\equiv$ (*R* `` *P*) $\subseteq$ *S*

lemma *wpc-test-weaken*:
  $[\![$ *wpc-test Q R S*; *P* $\subseteq$ *Q* $]\!]$ $\Longrightarrow$ *wpc-test P R S*
  by (*simp add*: *wpc-test-def*, *blast*)

lemma *wpc-helper-validF*:
  wpc-test Q′ R S $\Longrightarrow$ *wpc-helper* (*P*, *P′*) (*Q*, *Q′*) (*wpc-test P′ R S*)
  by (*simp add*: *wpc-test-def wpc-helper-def*, *blast*)

setup ‹
let
  val tm = *Thm.cterm-of* @{*context*} (*Logic.varify-global* @{*term* λ*R*. *wpc-test P R S*});
  val thm = @{*thm wpc-helper-validF*};
in
  WPCPredicateAndFinals.map (*fn xs* => (*tm*, *thm*) :: *xs*)
end
›

lemma *set-conj-Int-simp*:
  {*s* ∈ *S*. *P s*} = *S* ∩ {*s*. *P s*}
  by *auto*

lemma *case-options-weak-wp*:
  $[\![$ *wpc-test P R S*; $\bigwedge$*x*. *wpc-test P′* (*R′ x*) *S* $]\!]$
    $\Longrightarrow$ *wpc-test* (*P* ∩ *P′*) (*case opt of None* ⇒ *R* | *Some x* ⇒ *R′ x*) *S*
  apply (*rule wpc-test-weaken*)
   apply *wpcw*
    apply *assumption*
   apply *assumption*
  apply *simp*
  done

end

theory *Simp-No-Conditional*

imports *Main*

begin

Simplification without conditional rewriting. Setting the simplifier depth limit to zero prevents attempts at conditional rewriting. This should make the simplifier faster and more predictable on average. It may be particularly useful in derived tactics and methods to avoid situations where the simplifier repeatedly attempts and fails a conditional rewrite.

As always, there are caveats. Failing to perform a simple conditional rewrite may open the door to expensive alternatives. Various simprocs which are conditional in nature will not be deactivated.

ML ‹

```
structure Simp-No-Conditional = struct

val set-no-cond = Config.put Raw-Simplifier.simp-depth-limit 0

val simp-tac = Simplifier.simp-tac o set-no-cond
val asm-simp-tac = Simplifier.asm-simp-tac o set-no-cond
val full-simp-tac = Simplifier.full-simp-tac o set-no-cond
val asm-full-simp-tac = Simplifier.asm-full-simp-tac o set-no-cond

val clarsimp-tac = Clasimp.clarsimp-tac o set-no-cond
val auto-tac = Clasimp.auto-tac o set-no-cond

fun mk-method secs tac
  = Method.sections secs >> K (SIMPLE-METHOD' o tac)
val mk-clasimp-method = mk-method Clasimp.clasimp-modifiers

fun mk-clasimp-all-method tac =
  Method.sections Clasimp.clasimp-modifiers >> K (SIMPLE-METHOD o tac)

val simp-method = mk-method Simplifier.simp-modifiers
  (CHANGED-PROP oo asm-full-simp-tac)
val clarsimp-method = mk-clasimp-method (CHANGED-PROP oo clarsimp-tac)
val auto-method = mk-clasimp-all-method (CHANGED-PROP o auto-tac)

end
```

›

method-setup *simp-no-cond* = ‹*Simp-No-Conditional.simp-method*›
Simplification with no conditional simplification.

method-setup *clarsimp-no-cond* = ‹*Simp-No-Conditional.clarsimp-method*›
Clarsimp with no conditional simplification.

method-setup *auto-no-cond* = ‹*Simp-No-Conditional.auto-method*›
Auto with no conditional simplification.

end

theory *WPSimp*
imports
WP
WPC

WPFix
../../Simp-No-Conditional
begin

method *wpsimp* **uses** *wp wp-del simp simp-del split split-del cong comb comb-del* =
((*determ* ‹*wpfix* | *wp add*: *wp del*: *wp-del comb*: *comb comb del*: *comb-del* | *wpc* |
clarsimp-no-cond simp: *simp simp del*: *simp-del split*: *split split del*: *split-del cong*: *cong* |
clarsimp simp: *simp simp del*: *simp-del split*: *split split del*: *split-del cong*: *cong*›)+)[*1*]

end

theory *NonDetMonadVCG*
imports
NonDetMonadLemmas
wp/WPSimp
Strengthen
begin

declare *K-def* [*simp*]

26 Satisfiability

The dual to validity: an existential instead of a universal quantifier for the post condition. In refinement, it is often sufficient to know that there is one state that satisfies a condition.

definition
exs-valid :: $('a \Rightarrow bool) \Rightarrow ('a, 'b)$ *nondet-monad* $\Rightarrow$
$('b \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
(⦃-⦄ - ∃⦃-⦄)
where
exs-valid $P\ f\ Q \equiv (\forall s.\ P\ s \longrightarrow (\exists (rv, s') \in fst\ (f\ s).\ Q\ rv\ s'))$

The above for the exception monad

definition
ex-exs-validE :: $('a \Rightarrow bool) \Rightarrow ('a, 'e + 'b)$ *nondet-monad* $\Rightarrow$
$('b \Rightarrow 'a \Rightarrow bool) \Rightarrow ('e \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
(⦃-⦄ - ∃⦃-⦄, ⦃-⦄)
where
ex-exs-validE $P\ f\ Q\ E \equiv$
exs-valid $P\ f\ (\lambda rv.\ case\ rv\ of\ Inl\ e \Rightarrow E\ e \mid Inr\ v \Rightarrow Q\ v)$

27 Lemmas

27.1 Determinism

lemma *det-set-iff*:
  $det\ f \Longrightarrow (r \in fst\ (f\ s)) = (fst\ (f\ s) = \{r\})$
  apply (*simp add*: *det-def*)
  apply (*rule iffI*)
  apply (*erule-tac x=s* **in** *allE*)
  apply *auto*
  done

lemma *return-det* [*iff*]:
  $det\ (return\ x)$
  by (*simp add*: *det-def return-def*)

lemma *put-det* [*iff*]:
  $det\ (put\ s)$
  by (*simp add*: *det-def put-def*)

lemma *get-det* [*iff*]:
  $det\ get$
  by (*simp add*: *det-def get-def*)

lemma *det-gets* [*iff*]:
  $det\ (gets\ f)$
  by (*auto simp add*: *gets-def det-def get-def return-def bind-def*)

lemma *det-UN*:
  $det\ f \Longrightarrow (\bigcup x \in fst\ (f\ s).\ g\ x) = (g\ (THE\ x.\ x \in fst\ (f\ s)))$
  unfolding *det-def*
  apply *simp*
  apply (*drule spec* [*of - s*])
  apply *clarsimp*
  done

lemma *bind-detI* [*simp*, *intro*!]:
  $[\![\ det\ f;\ \forall x.\ det\ (g\ x)\ ]\!] \Longrightarrow det\ (f >>= g)$
  apply (*simp add*: *bind-def det-def split-def*)
  apply *clarsimp*
  apply (*erule-tac x=s* **in** *allE*)
  apply *clarsimp*
  apply (*erule-tac x=a* **in** *allE*)
  apply (*erule-tac x=b* **in** *allE*)
  apply *clarsimp*
  done

lemma *the-run-stateI*:
  $fst\ (M\ s) = \{s'\} \Longrightarrow the\text{-}run\text{-}state\ M\ s = s'$
  by (*simp add*: *the-run-state-def*)

lemma *the-run-state-det*:
  $[\![\ s' \in fst\ (M\ s);\ det\ M\ ]\!] \Longrightarrow the\text{-}run\text{-}state\ M\ s = s'$
  by (*simp add*: *the-run-stateI det-set-iff*)

27.2 Lifting and Alternative Basic Definitions

lemma *liftE-liftM*: *liftE* = *liftM Inr*
  apply (*rule ext*)
  apply (*simp add*: *liftE-def liftM-def*)
  done

lemma *liftME-liftM*: *liftME f* = *liftM* (*case-sum Inl* (*Inr* ∘ *f*))
  apply (*rule ext*)
  apply (*simp add*: *liftME-def liftM-def bindE-def returnOk-def lift-def*)
  apply (*rule-tac f=bind x* **in** *arg-cong*)
  apply (*rule ext*)
  apply (*case-tac xa*)
   apply (*simp-all add*: *lift-def throwError-def*)
  done

lemma *liftE-bindE*:
  (*liftE a*) >>=E *b* = *a* >>= *b*
  apply (*simp add*: *liftE-def bindE-def lift-def bind-assoc*)
  done

lemma *liftM-id*[*simp*]: *liftM id* = *id*
  apply (*rule ext*)
  apply (*simp add*: *liftM-def*)
  done

lemma *liftM-bind*:
  (*liftM t f* >>= *g*) = (*f* >>= ($\lambda x.\ g\ (t\ x)$))
  by (*simp add*: *liftM-def bind-assoc*)

lemma *gets-bind-ign*: *gets f* >>= ($\lambda x.\ m$) = *m*
  apply (*rule ext*)
  apply (*simp add*: *bind-def simpler-gets-def*)
  done

lemma *get-bind-apply*: (*get* >>= *f*) *x* = *f x x*
  by (*simp add*: *get-def bind-def*)

lemma *exec-gets*:
  (*gets f* >>= *m*) *s* = *m* (*f s*) *s*
  by (*simp add*: *simpler-gets-def bind-def*)

lemma *exec-get*:
  (*get* >>= *m*) *s* = *m s s*

by (*simp add*: *get-def bind-def*)

lemma *bind-eqI*:
$[\![f = f'; \bigwedge x.\ g\ x = g'\ x]\!] \Longrightarrow f >>= g = f' >>= g'$
apply (*rule ext*)
apply (*simp add*: *bind-def*)
apply (*auto simp*: *split-def*)
done

27.3 Simplification Rules for Lifted And/Or

lemma *pred-andE*[*elim*!]: $[\![(A\ and\ B)\ x; [\![A\ x; B\ x]\!] \Longrightarrow R]\!] \Longrightarrow R$
by(*simp add*:*pred-conj-def*)

lemma *pred-andI*[*intro*!]: $[\![A\ x; B\ x]\!] \Longrightarrow (A\ and\ B)\ x$
by(*simp add*:*pred-conj-def*)

lemma *pred-conj-app*[*simp*]: $(P\ and\ Q)\ x = (P\ x \wedge Q\ x)$
by(*simp add*:*pred-conj-def*)

lemma *bipred-andE*[*elim*!]: $[\![(A\ And\ B)\ x\ y; [\![A\ x\ y; B\ x\ y]\!] \Longrightarrow R]\!] \Longrightarrow R$
by(*simp add*:*bipred-conj-def*)

lemma *bipred-andI*[*intro*!]: $[\![A\ x\ y; B\ x\ y]\!] \Longrightarrow (A\ And\ B)\ x\ y$
by (*simp add*:*bipred-conj-def*)

lemma *bipred-conj-app*[*simp*]: $(P\ And\ Q)\ x = (P\ x\ and\ Q\ x)$
by(*simp add*:*pred-conj-def bipred-conj-def*)

lemma *pred-disjE*[*elim*!]: $[\![(P\ or\ Q)\ x; P\ x \Longrightarrow R; Q\ x \Longrightarrow R]\!] \Longrightarrow R$
by (*fastforce simp*: *pred-disj-def*)

lemma *pred-disjI1*[*intro*]: $P\ x \Longrightarrow (P\ or\ Q)\ x$
by (*simp add*: *pred-disj-def*)

lemma *pred-disjI2*[*intro*]: $Q\ x \Longrightarrow (P\ or\ Q)\ x$
by (*simp add*: *pred-disj-def*)

lemma *pred-disj-app*[*simp*]: $(P\ or\ Q)\ x = (P\ x \vee Q\ x)$
by *auto*

lemma *bipred-disjI1*[*intro*]: $P\ x\ y \Longrightarrow (P\ Or\ Q)\ x\ y$
by (*simp add*: *bipred-disj-def*)

lemma *bipred-disjI2*[*intro*]: $Q\ x\ y \Longrightarrow (P\ Or\ Q)\ x\ y$
by (*simp add*: *bipred-disj-def*)

lemma *bipred-disj-app*[*simp*]: $(P\ Or\ Q)\ x = (P\ x\ or\ Q\ x)$
by(*simp add*:*pred-disj-def bipred-disj-def*)

lemma *pred-notnotD*[*simp*]: $(not\ not\ P) = P$
  by(*simp add*:*pred-neg-def*)

lemma *pred-and-true*[*simp*]: $(P\ and\ \top) = P$
  by(*simp add*:*pred-conj-def*)

lemma *pred-and-true-var*[*simp*]: $(\top\ and\ P) = P$
  by(*simp add*:*pred-conj-def*)

lemma *pred-and-false*[*simp*]: $(P\ and\ \bot) = \bot$
  by(*simp add*:*pred-conj-def*)

lemma *pred-and-false-var*[*simp*]: $(\bot\ and\ P) = \bot$
  by(*simp add*:*pred-conj-def*)

lemma *pred-conj-assoc*:
  $(P\ and\ Q\ and\ R) = (P\ and\ (Q\ and\ R))$
  unfolding *pred-conj-def* **by** *simp*

27.4 Hoare Logic Rules

lemma *validE-def2*:
  $\{\!|P|\!\}\ f\ \{\!|Q|\!\},\{\!|R|\!\} \equiv \forall s.\ P\ s \longrightarrow (\forall (r,s') \in fst\ (f\ s).\ case\ r\ of\ Inr\ b \Rightarrow Q\ b\ s'$
  $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad | \ Inl\ a \Rightarrow R\ a\ s')$
  by (*unfold valid-def validE-def*)

lemma *seq'*:
  $[\![\ \{\!|A|\!\}\ f\ \{\!|B|\!\};$
  $\quad \forall x.\ P\ x \longrightarrow \{\!|C|\!\}\ g\ x\ \{\!|D|\!\};$
  $\quad \forall x\ s.\ B\ x\ s \longrightarrow P\ x \wedge C\ s\ ]\!] \Longrightarrow$
  $\{\!|A|\!\}\ do\ x \leftarrow f;\ g\ x\ od\ \{\!|D|\!\}$
  apply (*clarsimp simp*: *valid-def bind-def*)
  apply *fastforce*
  done

lemma *seq*:
  assumes *f-valid*: $\{\!|A|\!\}\ f\ \{\!|B|\!\}$
  assumes *g-valid*: $\bigwedge x.\ P\ x \Longrightarrow \{\!|C|\!\}\ g\ x\ \{\!|D|\!\}$
  assumes *bind*: $\bigwedge x\ s.\ B\ x\ s \Longrightarrow P\ x \wedge C\ s$
  shows $\{\!|A|\!\}\ do\ x \leftarrow f;\ g\ x\ od\ \{\!|D|\!\}$
apply (*insert f-valid g-valid bind*)
apply (*blast intro*: *seq'*)
done

lemma *seq-ext'*:
  $[\![\ \{\!|A|\!\}\ f\ \{\!|B|\!\};$
  $\quad \forall x.\ \{\!|B\ x|\!\}\ g\ x\ \{\!|C|\!\}\ ]\!] \Longrightarrow$
  $\{\!|A|\!\}\ do\ x \leftarrow f;\ g\ x\ od\ \{\!|C|\!\}$

by (*fastforce simp*: *valid-def bind-def Let-def split-def*)

lemma *seq-ext*:
assumes *f-valid*: $\{\!|A|\!\}\ f\ \{\!|B|\!\}$
assumes *g-valid*: $\bigwedge x.\ \{\!|B\ x|\!\}\ g\ x\ \{\!|C|\!\}$
shows $\{\!|A|\!\}\ do\ x \leftarrow f;\ g\ x\ od\ \{\!|C|\!\}$
apply(*insert f-valid g-valid*)
apply(*blast intro*: *seq-ext'*)
done

lemma *seqE'*:
$[\![\ \{\!|A|\!\}\ f\ \{\!|B|\!\},\{\!|E|\!\};$
$\forall x.\ \{\!|B\ x|\!\}\ g\ x\ \{\!|C|\!\},\{\!|E|\!\}\ ]\!] \Longrightarrow$
$\{\!|A|\!\}\ doE\ x \leftarrow f;\ g\ x\ odE\ \{\!|C|\!\},\{\!|E|\!\}$
apply(*simp add*:*bindE-def lift-def bind-def Let-def split-def*)
apply(*clarsimp simp*:*validE-def2*)
apply (*fastforce simp add*: *throwError-def return-def lift-def*
split: *sum.splits*)
done

lemma *seqE*:
assumes *f-valid*: $\{\!|A|\!\}\ f\ \{\!|B|\!\},\{\!|E|\!\}$
assumes *g-valid*: $\bigwedge x.\ \{\!|B\ x|\!\}\ g\ x\ \{\!|C|\!\},\{\!|E|\!\}$
shows $\{\!|A|\!\}\ doE\ x \leftarrow f;\ g\ x\ odE\ \{\!|C|\!\},\{\!|E|\!\}$
apply(*insert f-valid g-valid*)
apply(*blast intro*: *seqE'*)
done

lemma *hoare-TrueI*: $\{\!|P|\!\}\ f\ \{\!|\lambda\text{-}.\ \top|\!\}$
by (*simp add*: *valid-def*)

lemma *hoareE-TrueI*: $\{\!|P|\!\}\ f\ \{\!|\lambda\text{-}.\ \top|\!\},\ \{\!|\lambda r.\ \top|\!\}$
by (*simp add*: *validE-def valid-def*)

lemma *hoare-True-E-R* [*simp*]:
$\{\!|P|\!\}\ f\ \{\!|\lambda r\ s.\ True|\!\},\ -$
by (*auto simp add*: *validE-R-def validE-def valid-def split*: *sum.splits*)

lemma *hoare-post-conj* [*intro*]:
$[\![\ \{\!|\ P\ |\!\}\ a\ \{\!|\ Q\ |\!\};\ \{\!|\ P\ |\!\}\ a\ \{\!|\ R\ |\!\}\ ]\!] \Longrightarrow \{\!|\ P\ |\!\}\ a\ \{\!|\ Q\ And\ R\ |\!\}$
by (*fastforce simp*: *valid-def split-def bipred-conj-def*)

lemma *hoare-pre-disj* [*intro*]:
$[\![\ \{\!|\ P\ |\!\}\ a\ \{\!|\ R\ |\!\};\ \{\!|\ Q\ |\!\}\ a\ \{\!|\ R\ |\!\}\ ]\!] \Longrightarrow \{\!|\ P\ or\ Q\ |\!\}\ a\ \{\!|\ R\ |\!\}$
by (*simp add*:*valid-def pred-disj-def*)

lemma *hoare-conj*:
$[\![\ \{\!|P|\!\}\ f\ \{\!|Q|\!\};\ \{\!|P'|\!\}\ f\ \{\!|Q'|\!\}\ ]\!] \Longrightarrow \{\!|P\ and\ P'|\!\}\ f\ \{\!|Q\ And\ Q'|\!\}$
unfolding *valid-def* **by** *auto*

lemma *hoare-post-taut*: $\{\!|\ P\ |\!\}$ a $\{\!|\ \top\top\ |\!\}$
by (*simp add*:*valid-def*)

lemma *wp-post-taut*: $\{\!|\lambda r.\ True|\!\}$ f $\{\!|\lambda r\ s.\ True|\!\}$
by (*rule hoare-post-taut*)

lemma *wp-post-tautE*: $\{\!|\lambda r.\ True|\!\}$ f $\{\!|\lambda r\ s.\ True|\!\},\{\!|\lambda f\ s.\ True|\!\}$
proof −
have P: $\bigwedge r.$ (*case* r *of* *Inl* $a \Rightarrow$ *True* | - $\Rightarrow$ *True*) = *True*
by (*case-tac* r, *simp-all*)
show *?thesis*
by (*simp add*: *validE-def* P *wp-post-taut*)
qed

lemma *hoare-pre-cont* [*simp*]: $\{\!|\ \bot\ |\!\}$ a $\{\!|\ P\ |\!\}$
by (*simp add*:*valid-def*)

27.5 Strongest Postcondition Rules

lemma *get-sp*:
$\{\!|P|\!\}$ *get* $\{\!|\lambda a\ s.\ s = a \wedge P\ s|\!\}$
by(*simp add*:*get-def valid-def*)

lemma *put-sp*:
$\{\!|\top|\!\}$ *put* a $\{\!|\lambda\text{-}\ s.\ s = a|\!\}$
by(*simp add*:*put-def valid-def*)

lemma *return-sp*:
$\{\!|P|\!\}$ *return* a $\{\!|\lambda b\ s.\ b = a \wedge P\ s|\!\}$
by(*simp add*:*return-def valid-def*)

lemma *assert-sp*:
$\{\!|\ P\ |\!\}$ *assert* Q $\{\!|\ \lambda r\ s.\ P\ s \wedge Q\ |\!\}$
by (*simp add*: *assert-def fail-def return-def valid-def*)

lemma *hoare-gets-sp*:
$\{\!|P|\!\}$ *gets* f $\{\!|\lambda rv\ s.\ rv = f\ s \wedge P\ s|\!\}$
by (*simp add*: *valid-def simpler-gets-def*)

lemma *hoare-return-drop-var* [*iff*]: $\{\!|\ Q\ |\!\}$ *return* x $\{\!|\ \lambda r.\ Q\ |\!\}$
by (*simp add*:*valid-def return-def*)

lemma *hoare-gets* [*intro*]: $[\![\ \bigwedge s.\ P\ s \Longrightarrow Q\ (f\ s)\ s\ ]\!] \Longrightarrow \{\!|\ P\ |\!\}$ *gets* f $\{\!|\ Q\ |\!\}$
by (*simp add*:*valid-def gets-def get-def bind-def return-def*)

lemma *hoare-modifyE-var*:
$[\![\ \bigwedge s.\ P\ s \Longrightarrow Q\ (f\ s)\ ]\!] \Longrightarrow \{\!|\ P\ |\!\}$ *modify* f $\{\!|\ \lambda r\ s.\ Q\ s\ |\!\}$
by(*simp add*: *valid-def modify-def put-def get-def bind-def*)

lemma *hoare-if*:
$\llbracket$ $P \Longrightarrow \{\!|\ Q\ |\!\}\ a\ \{\!|\ R\ |\!\}$; $\neg\ P \Longrightarrow \{\!|\ Q\ |\!\}\ b\ \{\!|\ R\ |\!\}$ $\rrbracket \Longrightarrow$
$\{\!|\ Q\ |\!\}$ *if P then a else b* $\{\!|\ R\ |\!\}$
by (*simp add*:*valid-def*)

lemma *hoare-pre-subst*: $\llbracket$ $A = B$; $\{\!|A|\!\}\ a\ \{\!|C|\!\}$ $\rrbracket \Longrightarrow \{\!|B|\!\}\ a\ \{\!|C|\!\}$
by(*clarsimp simp*:*valid-def split-def*)

lemma *hoare-post-subst*: $\llbracket$ $B = C$; $\{\!|A|\!\}\ a\ \{\!|B|\!\}$ $\rrbracket \Longrightarrow \{\!|A|\!\}\ a\ \{\!|C|\!\}$
by(*clarsimp simp*:*valid-def split-def*)

lemma *hoare-pre-tautI*: $\llbracket$ $\{\!|A$ *and* $P|\!\}\ a\ \{\!|B|\!\}$; $\{\!|A$ *and not* $P|\!\}\ a\ \{\!|B|\!\}$ $\rrbracket \Longrightarrow \{\!|A|\!\}\ a$
$\{\!|B|\!\}$
by(*fastforce simp*:*valid-def split-def pred-conj-def pred-neg-def*)

lemma *hoare-pre-imp*: $\llbracket$ $\bigwedge s.\ P\ s \Longrightarrow Q\ s$; $\{\!|Q|\!\}\ a\ \{\!|R|\!\}$ $\rrbracket \Longrightarrow \{\!|P|\!\}\ a\ \{\!|R|\!\}$
by (*fastforce simp add*:*valid-def*)

lemma *hoare-post-imp*: $\llbracket$ $\bigwedge r\ s.\ Q\ r\ s \Longrightarrow R\ r\ s$; $\{\!|P|\!\}\ a\ \{\!|Q|\!\}$ $\rrbracket \Longrightarrow \{\!|P|\!\}\ a\ \{\!|R|\!\}$
by(*fastforce simp*:*valid-def split-def*)

lemma *hoare-post-impErr'*: $\llbracket$ $\{\!|P|\!\}\ a\ \{\!|Q|\!\},\{\!|E|\!\}$;
$\forall\, r\ s.\ Q\ r\ s \longrightarrow R\ r\ s$;
$\forall\, e\ s.\ E\ e\ s \longrightarrow F\ e\ s$ $\rrbracket \Longrightarrow$
$\{\!|P|\!\}\ a\ \{\!|R|\!\},\{\!|F|\!\}$
apply (*simp add*: *validE-def*)
apply (*rule-tac Q*=$\lambda r\ s.$ *case r of Inl a* $\Rightarrow$ *E a s* | *Inr b* $\Rightarrow$ *Q b s* **in** *hoare-post-imp*)
apply (*case-tac r*)
apply *simp-all*
done

lemma *hoare-post-impErr*: $\llbracket$ $\{\!|P|\!\}\ a\ \{\!|Q|\!\},\{\!|E|\!\}$;
$\bigwedge r\ s.\ Q\ r\ s \Longrightarrow R\ r\ s$;
$\bigwedge e\ s.\ E\ e\ s \Longrightarrow F\ e\ s$ $\rrbracket \Longrightarrow$
$\{\!|P|\!\}\ a\ \{\!|R|\!\},\{\!|F|\!\}$
apply (*blast intro*: *hoare-post-impErr'*)
done

lemma *hoare-validE-cases*:
$\llbracket$ $\{\!|\ P\ |\!\}\ f\ \{\!|\ Q\ |\!\}$, $\{\!|\ \lambda$- -. *True* $|\!\}$; $\{\!|\ P\ |\!\}\ f\ \{\!|\ \lambda$- -. *True* $|\!\}$, $\{\!|\ R\ |\!\}$ $\rrbracket$
$\Longrightarrow \{\!|\ P\ |\!\}\ f\ \{\!|\ Q\ |\!\}$, $\{\!|\ R\ |\!\}$
by (*simp add*: *validE-def valid-def split*: *sum.splits*) *blast*

lemma *hoare-post-imp-dc*:
$\llbracket\{\!|P|\!\}\ a\ \{\!|\lambda r.\ Q|\!\}$; $\bigwedge s.\ Q\ s \Longrightarrow R\ s\rrbracket \Longrightarrow \{\!|P|\!\}\ a\ \{\!|\lambda r.\ R|\!\},\{\!|\lambda r.\ R|\!\}$
by (*simp add*: *validE-def valid-def split*: *sum.splits*) *blast*

lemma *hoare-post-imp-dc2*:

$[\![\{\!|P|\!\}\ a\ \{\!|\lambda r.\ Q|\!\};\ \bigwedge s.\ Q\ s \Longrightarrow R\ s]\!] \Longrightarrow \{\!|P|\!\}\ a\ \{\!|\lambda r.\ R|\!\},\{\!|\lambda r\ s.\ True|\!\}$
by (*simp add*: *validE-def valid-def split*: *sum.splits*) *blast*

lemma *hoare-post-imp-dc2E*:
$[\![\{\!|P|\!\}\ a\ \{\!|\lambda r.\ Q|\!\};\ \bigwedge s.\ Q\ s \Longrightarrow R\ s]\!] \Longrightarrow \{\!|P|\!\}\ a\ \{\!|\lambda r\ s.\ True|\!\},\ \{\!|\lambda r.\ R|\!\}$
by (*simp add*: *validE-def valid-def split*: *sum.splits*) *fast*

lemma *hoare-post-imp-dc2E-actual*:
$[\![\{\!|P|\!\}\ a\ \{\!|\lambda r.\ R|\!\}]\!] \Longrightarrow \{\!|P|\!\}\ a\ \{\!|\lambda r\ s.\ True|\!\},\ \{\!|\lambda r.\ R|\!\}$
by (*simp add*: *validE-def valid-def split*: *sum.splits*) *fast*

lemma *hoare-post-imp-dc2-actual*:
$[\![\{\!|P|\!\}\ a\ \{\!|\lambda r.\ R|\!\}]\!] \Longrightarrow \{\!|P|\!\}\ a\ \{\!|\lambda r.\ R|\!\},\ \{\!|\lambda r\ s.\ True|\!\}$
by (*simp add*: *validE-def valid-def split*: *sum.splits*) *fast*

lemma *hoare-post-impE*: $[\![\ \bigwedge r\ s.\ Q\ r\ s \Longrightarrow R\ r\ s;\ \{\!|P|\!\}\ a\ \{\!|Q|\!\}\ ]\!] \Longrightarrow \{\!|P|\!\}\ a\ \{\!|R|\!\}$
by (*fastforce simp*:*valid-def split-def*)

lemma *hoare-conjD1*:
$\{\!|P|\!\}\ f\ \{\!|\lambda rv.\ Q\ rv\ and\ R\ rv|\!\} \Longrightarrow \{\!|P|\!\}\ f\ \{\!|\lambda rv.\ Q\ rv|\!\}$
unfolding *valid-def* **by** *auto*

lemma *hoare-conjD2*:
$\{\!|P|\!\}\ f\ \{\!|\lambda rv.\ Q\ rv\ and\ R\ rv|\!\} \Longrightarrow \{\!|P|\!\}\ f\ \{\!|\lambda rv.\ R\ rv|\!\}$
unfolding *valid-def* **by** *auto*

lemma *hoare-post-disjI1*:
$\{\!|P|\!\}\ f\ \{\!|\lambda rv.\ Q\ rv|\!\} \Longrightarrow \{\!|P|\!\}\ f\ \{\!|\lambda rv.\ Q\ rv\ or\ R\ rv|\!\}$
unfolding *valid-def* **by** *auto*

lemma *hoare-post-disjI2*:
$\{\!|P|\!\}\ f\ \{\!|\lambda rv.\ R\ rv|\!\} \Longrightarrow \{\!|P|\!\}\ f\ \{\!|\lambda rv.\ Q\ rv\ or\ R\ rv|\!\}$
unfolding *valid-def* **by** *auto*

lemma *hoare-weaken-pre*:
$[\![\{\!|Q|\!\}\ a\ \{\!|R|\!\};\ \bigwedge s.\ P\ s \Longrightarrow Q\ s]\!] \Longrightarrow \{\!|P|\!\}\ a\ \{\!|R|\!\}$
apply (*rule hoare-pre-imp*)
 prefer *2*
 apply *assumption*
apply *blast*
done

lemma *hoare-strengthen-post*:
$[\![\{\!|P|\!\}\ a\ \{\!|Q|\!\};\ \bigwedge r\ s.\ Q\ r\ s \Longrightarrow R\ r\ s]\!] \Longrightarrow \{\!|P|\!\}\ a\ \{\!|R|\!\}$
apply (*rule hoare-post-imp*)
 prefer *2*
 apply *assumption*
apply *blast*
done

lemma *use-valid*: $[\![(r,\ s') \in \mathit{fst}\ (f\ s);\ \{\!|P|\!\}\ f\ \{\!|Q|\!\};\ P\ s\ ]\!] \Longrightarrow Q\ r\ s'$
  apply (*simp add*: *valid-def*)
  apply *blast*
  done

lemma *use-validE-norm*: $[\![\ (\mathit{Inr}\ r',\ s') \in \mathit{fst}\ (B\ s);\ \{\!|\ P\ |\!\}\ B\ \{\!|\ Q\ |\!\},\{\!|\ E\ |\!\};\ P\ s\ ]\!] \Longrightarrow Q\ r'\ s'$
  apply (*clarsimp simp*: *validE-def valid-def*)
  apply *force*
  done

lemma *use-validE-except*: $[\![\ (\mathit{Inl}\ r',\ s') \in \mathit{fst}\ (B\ s);\ \{\!|\ P\ |\!\}\ B\ \{\!|\ Q\ |\!\},\{\!|\ E\ |\!\};\ P\ s\ ]\!] \Longrightarrow E\ r'\ s'$
  apply (*clarsimp simp*: *validE-def valid-def*)
  apply *force*
  done

lemma *in-inv-by-hoareD*:
  $[\![\ \bigwedge P.\ \{\!|P|\!\}\ f\ \{\!|\lambda\text{-}.\ P|\!\};\ (x,s') \in \mathit{fst}\ (f\ s)\ ]\!] \Longrightarrow s' = s$
  by (*auto simp add*: *valid-def*) *blast*

27.6 Satisfiability

lemma *exs-hoare-post-imp*: $[\![\bigwedge r\ s.\ Q\ r\ s \Longrightarrow R\ r\ s;\ \{\!|P|\!\}\ a\ \exists\{\!|Q|\!\}]\!] \Longrightarrow \{\!|P|\!\}\ a\ \exists\{\!|R|\!\}$
  apply (*simp add*: *exs-valid-def*)
  apply *safe*
  apply (*erule-tac x=s* **in** *allE*, *simp*)
  apply *blast*
  done

lemma *use-exs-valid*: $[\![\{\!|P|\!\}\ f\ \exists\{\!|Q|\!\};\ P\ s\ ]\!] \Longrightarrow \exists(r,\ s') \in \mathit{fst}\ (f\ s).\ Q\ r\ s'$
  by (*simp add*: *exs-valid-def*)

definition *exs-postcondition* $P\ f \equiv (\lambda a\ b.\ \exists(rv,\ s) \in f\ a\ b.\ P\ rv\ s)$

lemma *exs-valid-is-triple*:
  exs-valid $P\ f\ Q$ = *triple-judgement* $P\ f$ (*exs-postcondition* Q ($\lambda s\ f.\ \mathit{fst}\ (f\ s)$))
  by (*simp add*: *triple-judgement-def exs-postcondition-def exs-valid-def*)

lemmas [*wp-trip*] = *exs-valid-is-triple*

lemma *exs-valid-weaken-pre*[*wp-pre*]:
  $[\![\ \{\!|\ P'\ |\!\}\ f\ \exists\{\!|\ Q\ |\!\};\ \bigwedge s.\ P\ s \Longrightarrow P'\ s\ ]\!] \Longrightarrow \{\!|\ P\ |\!\}\ f\ \exists\{\!|\ Q\ |\!\}$
  apply *atomize*
  apply (*clarsimp simp*: *exs-valid-def*)
  done

lemma *exs-valid-chain*:
$\llbracket$ $\{\!|$ P $|\!\}$ f $\exists\{\!|$ Q $|\!\}$; $\bigwedge s.\ R\ s \Longrightarrow P\ s$; $\bigwedge r\ s.\ Q\ r\ s \Longrightarrow S\ r\ s$ $\rrbracket \Longrightarrow$ $\{\!|$ R $|\!\}$ f $\exists\{\!|$ S $|\!\}$
apply *atomize*
apply (*fastforce simp*: *exs-valid-def Bex-def*)
done

lemma *exs-valid-assume-pre*:
$\llbracket$ $\bigwedge s.\ P\ s \Longrightarrow$ $\{\!|$ P $|\!\}$ f $\exists\{\!|$ Q $|\!\}$ $\rrbracket \Longrightarrow$ $\{\!|$ P $|\!\}$ f $\exists\{\!|$ Q $|\!\}$
apply (*fastforce simp*: *exs-valid-def*)
done

lemma *exs-valid-bind* [*wp-split*]:
$\llbracket$ $\bigwedge x.\ \{\!|B\ x|\!\}\ g\ x\ \exists\{\!|C|\!\}$; $\{\!|A|\!\}\ f\ \exists\{\!|B|\!\}$ $\rrbracket \Longrightarrow$ $\{\!|$ A $|\!\}$ f $>>=$ $(\lambda x.\ g\ x)$ $\exists\{\!|$ C $|\!\}$
apply *atomize*
apply (*clarsimp simp*: *exs-valid-def bind-def'*)
apply *blast*
done

lemma *exs-valid-return* [*wp*]:
$\{\!|$ $Q\ v$ $|\!\}$ *return* v $\exists\{\!|$ Q $|\!\}$
by (*clarsimp simp*: *exs-valid-def return-def*)

lemma *exs-valid-select* [*wp*]:
$\{\!|$ $\lambda s.\ \exists\, r \in S.\ Q\ r\ s$ $|\!\}$ *select* S $\exists\{\!|$ Q $|\!\}$
by (*clarsimp simp*: *exs-valid-def select-def*)

lemma *exs-valid-get* [*wp*]:
$\{\!|$ $\lambda s.\ Q\ s\ s$ $|\!\}$ *get* $\exists\{\!|$ Q $|\!\}$
by (*clarsimp simp*: *exs-valid-def get-def*)

lemma *exs-valid-gets* [*wp*]:
$\{\!|$ $\lambda s.\ Q\ (f\ s)\ s$ $|\!\}$ *gets* f $\exists\{\!|$ Q $|\!\}$
by (*clarsimp simp*: *gets-def*) *wp*

lemma *exs-valid-put* [*wp*]:
$\{\!|$ $Q\ v$ $|\!\}$ *put* v $\exists\{\!|$ Q $|\!\}$
by (*clarsimp simp*: *put-def exs-valid-def*)

lemma *exs-valid-state-assert* [*wp*]:
$\{\!|$ $\lambda s.\ Q\ ()\ s \wedge G\ s$ $|\!\}$ *state-assert* G $\exists\{\!|$ Q $|\!\}$
by (*clarsimp simp*: *state-assert-def exs-valid-def get-def*
assert-def bind-def' return-def)

lemmas *exs-valid-guard* = *exs-valid-state-assert*

lemma *exs-valid-fail* [*wp*]:
$\{\!|$ λ-. *False* $|\!\}$ *fail* $\exists\{\!|$ Q $|\!\}$
by (*clarsimp simp*: *fail-def exs-valid-def*)

lemma *exs-valid-condition* [*wp*]:
$[\![\ \{\!|\ P\ |\!\}\ L\ \exists\{\!|\ Q\ |\!\};\ \{\!|\ P'\ |\!\}\ R\ \exists\{\!|\ Q\ |\!\}\ ]\!] \Longrightarrow$
$\{\!|\ \lambda s.\ (C\ s \wedge P\ s) \vee (\neg\ C\ s \wedge P'\ s)\ |\!\}\ \textit{condition}\ C\ L\ R\ \exists\{\!|\ Q\ |\!\}$
by (*clarsimp simp*: *condition-def exs-valid-def split*: *sum.splits*)

27.7 MISC

lemma *hoare-return-simp*:
$\{\!|P|\!\}\ \textit{return}\ x\ \{\!|Q|\!\} = (\forall s.\ P\ s \longrightarrow Q\ x\ s)$
by (*simp add*: *valid-def return-def*)

lemma *hoare-gen-asm*:
$(P \Longrightarrow \{\!|P'|\!\}\ f\ \{\!|Q|\!\}) \Longrightarrow \{\!|P'\ \textit{and}\ K\ P|\!\}\ f\ \{\!|Q|\!\}$
by (*fastforce simp add*: *valid-def*)

lemma *hoare-gen-asm-lk*:
$(P \Longrightarrow \{\!|P'|\!\}\ f\ \{\!|Q|\!\}) \Longrightarrow \{\!|K\ P\ \textit{and}\ P'|\!\}\ f\ \{\!|Q|\!\}$
by (*fastforce simp add*: *valid-def*)

— Useful for forward reasoning, when P is known. The first version allows weakening the precondition.
lemma *hoare-gen-asm-spec'*:
$(\bigwedge s.\ P\ s \Longrightarrow S \wedge R\ s)$
$\Longrightarrow (S \Longrightarrow \{\!|R|\!\}\ f\ \{\!|Q|\!\})$
$\Longrightarrow \{\!|P|\!\}\ f\ \{\!|Q|\!\}$
by (*fastforce simp*: *valid-def*)

lemma *hoare-gen-asm-spec*:
$(\bigwedge s.\ P\ s \Longrightarrow S)$
$\Longrightarrow (S \Longrightarrow \{\!|P|\!\}\ f\ \{\!|Q|\!\})$
$\Longrightarrow \{\!|P|\!\}\ f\ \{\!|Q|\!\}$
by (*rule hoare-gen-asm-spec'*[**where** *S*=*S* **and** *R*=*P*]) *simp*

lemma *hoare-conjI*:
$[\![\ \{\!|P|\!\}\ f\ \{\!|Q|\!\};\ \{\!|P|\!\}\ f\ \{\!|R|\!\}\ ]\!] \Longrightarrow \{\!|P|\!\}\ f\ \{\!|\lambda r\ s.\ Q\ r\ s \wedge R\ r\ s|\!\}$
unfolding *valid-def* **by** *blast*

lemma *hoare-disjI1*:
$[\![\ \{\!|P|\!\}\ f\ \{\!|Q|\!\}\ ]\!] \Longrightarrow \{\!|P|\!\}\ f\ \{\!|\lambda r\ s.\ Q\ r\ s \vee\ R\ r\ s\ |\!\}$
unfolding *valid-def* **by** *blast*

lemma *hoare-disjI2*:
$[\![\ \{\!|P|\!\}\ f\ \{\!|R|\!\}\ ]\!] \Longrightarrow \{\!|P|\!\}\ f\ \{\!|\lambda r\ s.\ Q\ r\ s \vee\ R\ r\ s\ |\!\}$
unfolding *valid-def* **by** *blast*

lemma *hoare-assume-pre*:
$(\bigwedge s.\ P\ s \Longrightarrow \{\!|P|\!\}\ f\ \{\!|Q|\!\}) \Longrightarrow \{\!|P|\!\}\ f\ \{\!|Q|\!\}$
by (*auto simp*: *valid-def*)

lemma *hoare-returnOk-sp*:
$\{\!|P|\!\}$ *returnOk x* $\{\!|\lambda r\ s.\ r = x \wedge P\ s|\!\}, \{\!|Q|\!\}$
by (*simp add*: *valid-def validE-def returnOk-def return-def*)

lemma *hoare-assume-preE*:
$(\bigwedge s.\ P\ s \Longrightarrow \{\!|P|\!\}\ f\ \{\!|Q|\!\},\{\!|R|\!\}) \Longrightarrow \{\!|P|\!\}\ f\ \{\!|Q|\!\},\{\!|R|\!\}$
by (*auto simp*: *valid-def validE-def*)

lemma *hoare-allI*:
$(\bigwedge x.\ \{\!|P|\!\}f\{\!|Q\ x|\!\}) \Longrightarrow \{\!|P|\!\}f\{\!|\lambda r\ s.\ \forall x.\ Q\ x\ r\ s|\!\}$
by (*simp add*: *valid-def*) *blast*

lemma *validE-allI*:
$(\bigwedge x.\ \{\!|P|\!\}\ f\ \{\!|\lambda r\ s.\ Q\ x\ r\ s|\!\},\{\!|E|\!\}) \Longrightarrow \{\!|P|\!\}\ f\ \{\!|\lambda r\ s.\ \forall x.\ Q\ x\ r\ s|\!\},\{\!|E|\!\}$
by (*fastforce simp*: *valid-def validE-def split*: *sum.splits*)

lemma *hoare-exI*:
$\{\!|P|\!\}\ f\ \{\!|Q\ x|\!\} \Longrightarrow \{\!|P|\!\}\ f\ \{\!|\lambda r\ s.\ \exists x.\ Q\ x\ r\ s|\!\}$
by (*simp add*: *valid-def*) *blast*

lemma *hoare-impI*:
$(R \Longrightarrow \{\!|P|\!\}f\{\!|Q|\!\}) \Longrightarrow \{\!|P|\!\}f\{\!|\lambda r\ s.\ R \longrightarrow Q\ r\ s|\!\}$
by (*simp add*: *valid-def*) *blast*

lemma *validE-impI*:
$[\![\bigwedge E.\ \{\!|P|\!\}\ f\ \{\!|\lambda\text{- -}.\ True|\!\},\{\!|E|\!\};\ (P' \Longrightarrow \{\!|P|\!\}\ f\ \{\!|Q|\!\},\{\!|E|\!\})]\!] \Longrightarrow$
$\{\!|P|\!\}\ f\ \{\!|\lambda r\ s.\ P' \longrightarrow Q\ r\ s|\!\},\ \{\!|E|\!\}$
by (*fastforce simp*: *validE-def valid-def split*: *sum.splits*)

lemma *hoare-case-option-wp*:
$[\![\ \{\!|P|\!\}\ f\ None\ \{\!|Q|\!\};$
$\bigwedge x.\ \ \{\!|P'\ x|\!\}\ f\ (Some\ x)\ \{\!|Q'\ x|\!\}\ ]\!]$
$\Longrightarrow \{\!|case\text{-}option\ P\ P'\ v|\!\}\ f\ v\ \{\!|\lambda rv.\ case\ v\ of\ None \Rightarrow Q\ rv \mid Some\ x \Rightarrow Q'\ x\ rv|\!\}$
by (*cases v*) *auto*

27.8 Reasoning directly about states

lemma *in-throwError*:
$((v,\ s') \in fst\ (throwError\ e\ s)) = (v = Inl\ e \wedge s' = s)$
by (*simp add*: *throwError-def return-def*)

lemma *in-returnOk*:
$((v',\ s') \in fst\ (returnOk\ v\ s)) = (v' = Inr\ v \wedge s' = s)$
by (*simp add*: *returnOk-def return-def*)

lemma *in-bind*:
$((r,s') \in fst\ ((do\ x \leftarrow f;\ g\ x\ od)\ s)) =$
$(\exists s''\ x.\ (x,\ s'') \in fst\ (f\ s) \wedge (r,\ s') \in fst\ (g\ x\ s''))$
apply (*simp add*: *bind-def split-def*)

apply *force*
done

lemma *in-bindE-R*:
$((Inr\ r,s') \in fst\ ((doE\ x \leftarrow f;\ g\ x\ odE)\ s)) =$
$(\exists s''\ x.\ (Inr\ x,\ s'') \in fst\ (f\ s) \wedge (Inr\ r,\ s') \in fst\ (g\ x\ s''))$
apply (*simp add*: *bindE-def lift-def split-def bind-def*)
apply (*clarsimp simp*: *throwError-def return-def lift-def split*: *sum.splits*)
apply *safe*
apply (*case-tac a*)
apply *fastforce*
apply *fastforce*
apply *force*
done

lemma *in-bindE-L*:
$((Inl\ r,\ s') \in fst\ ((doE\ x \leftarrow f;\ g\ x\ odE)\ s)) \Longrightarrow$
$(\exists s''\ x.\ (Inr\ x,\ s'') \in fst\ (f\ s) \wedge (Inl\ r,\ s') \in fst\ (g\ x\ s'')) \vee ((Inl\ r,\ s') \in fst\ (f\ s))$
apply (*simp add*: *bindE-def lift-def bind-def*)
apply *safe*
apply (*simp add*: *return-def throwError-def lift-def split-def split*: *sum.splits if-split-asm*)
apply *force*
done

lemma *in-liftE*:
$((v,\ s') \in fst\ (liftE\ f\ s)) = (\exists v'.\ v = Inr\ v' \wedge (v',\ s') \in fst\ (f\ s))$
by (*force simp add*: *liftE-def bind-def return-def split-def*)

lemma *in-whenE*: $((v,\ s') \in fst\ (whenE\ P\ f\ s)) = ((P \longrightarrow (v,\ s') \in fst\ (f\ s)) \wedge$
$(\neg P \longrightarrow v = Inr\ () \wedge s' = s))$
by (*simp add*: *whenE-def in-returnOk*)

lemma *inl-whenE*:
$((Inl\ x,\ s') \in fst\ (whenE\ P\ f\ s)) = (P \wedge (Inl\ x,\ s') \in fst\ (f\ s))$
by (*auto simp add*: *in-whenE*)

lemma *inr-in-unlessE-throwError*[*termination-simp*]:
$(Inr\ (),\ s') \in fst\ (unlessE\ P\ (throwError\ E)\ s) = (P \wedge s'=s)$
by (*simp add*: *unlessE-def returnOk-def throwError-def return-def*)

lemma *in-fail*:
$r \in fst\ (fail\ s) = False$
by (*simp add*: *fail-def*)

lemma *in-return*:
$(r,\ s') \in fst\ (return\ v\ s) = (r = v \wedge s' = s)$
by (*simp add*: *return-def*)

lemma *in-assert*:
  $(r,\ s') \in$ *fst* (*assert P s*) $= (P \wedge s' = s)$
  by (*simp add*: *assert-def return-def fail-def*)

lemma *in-assertE*:
  $(r,\ s') \in$ *fst* (*assertE P s*) $= (P \wedge r =$ *Inr* () $\wedge\ s' = s)$
  by (*simp add*: *assertE-def returnOk-def return-def fail-def*)

lemma *in-assert-opt*:
  $(r,\ s') \in$ *fst* (*assert-opt v s*) $= (v =$ *Some r* $\wedge\ s' = s)$
  by (*auto simp*: *assert-opt-def in-fail in-return split*: *option.splits*)

lemma *in-get*:
  $(r,\ s') \in$ *fst* (*get s*) $= (r = s \wedge s' = s)$
  by (*simp add*: *get-def*)

lemma *in-gets*:
  $(r,\ s') \in$ *fst* (*gets f s*) $= (r = f\ s \wedge s' = s)$
  by (*simp add*: *simpler-gets-def*)

lemma *in-put*:
  $(r,\ s') \in$ *fst* (*put x s*) $= (s' = x \wedge r = ())$
  by (*simp add*: *put-def*)

lemma *in-when*:
  $(v,\ s') \in$ *fst* (*when P f s*) $= ((P \longrightarrow (v,\ s') \in$ *fst* $(f\ s)) \wedge (\neg P \longrightarrow v = () \wedge s'$
$= s))$
  by (*simp add*: *when-def in-return*)

lemma *in-modify*:
  $(v,\ s') \in$ *fst* (*modify f s*) $= (s'{=}f\ s \wedge v = ())$
  by (*simp add*: *modify-def bind-def get-def put-def*)

lemma *gets-the-in-monad*:
  $((v,\ s') \in$ *fst* (*gets-the f s*)) $= (s' = s \wedge f\ s =$ *Some v*)
  by (*auto simp*: *gets-the-def in-bind in-gets in-assert-opt split*: *option.split*)

lemma *in-alternative*:
  $(r{,}s') \in$ *fst* $((f \sqcap g)\ s) = ((r{,}s') \in$ *fst* $(f\ s) \vee (r{,}s') \in$ *fst* $(g\ s))$
  by (*simp add*: *alternative-def*)

lemmas *in-monad* = *inl-whenE in-whenE in-liftE in-bind in-bindE-L*
    in-bindE-R in-returnOk in-throwError in-fail
    in-assertE in-assert in-return in-assert-opt
    in-get in-gets in-put in-when unlessE-whenE
    unless-when in-modify gets-the-in-monad
    in-alternative

27.9 Non-Failure

lemma *no-failD*:
  $[\![$ *no-fail* P m; P s $]\!] \Longrightarrow \neg$(*snd* ($m$ s))
  by (*simp add*: *no-fail-def*)

lemma *non-fail-modify* [*wp*,*simp*]:
  no-fail $\top$ (*modify* f)
  by (*simp add*: *no-fail-def modify-def get-def put-def bind-def*)

lemma *non-fail-gets-simp*[*simp*]:
  no-fail P (*gets* f)
  unfolding *no-fail-def gets-def get-def return-def bind-def*
  by *simp*

lemma *non-fail-gets*:
  no-fail $\top$ (*gets* f)
  by *simp*

lemma *non-fail-select* [*simp*]:
  no-fail $\top$ (*select* S)
  by (*simp add*: *no-fail-def select-def*)

lemma *no-fail-pre*:
  $[\![$ *no-fail* P f; $\bigwedge s.$ Q $s \Longrightarrow P$ $s]\!] \Longrightarrow$ *no-fail* Q f
  by (*simp add*: *no-fail-def*)

lemma *no-fail-alt* [*wp*]:
  $[\![$ *no-fail* P f; *no-fail* Q g $]\!] \Longrightarrow$ *no-fail* (P *and* Q) (f *OR* g)
  by (*simp add*: *no-fail-def alternative-def*)

lemma *no-fail-return* [*simp*, *wp*]:
  no-fail $\top$ (*return* x)
  by (*simp add*: *return-def no-fail-def*)

lemma *no-fail-get* [*simp*, *wp*]:
  no-fail $\top$ *get*
  by (*simp add*: *get-def no-fail-def*)

lemma *no-fail-put* [*simp*, *wp*]:
  no-fail $\top$ (*put* s)
  by (*simp add*: *put-def no-fail-def*)

lemma *no-fail-when* [*wp*]:
  ($P \Longrightarrow$ *no-fail* Q f) $\Longrightarrow$ *no-fail* (*if* P *then* Q *else* $\top$) (*when* P f)
  by (*simp add*: *when-def*)

lemma *no-fail-unless* [*wp*]:
  ($\neg P \Longrightarrow$ *no-fail* Q f) $\Longrightarrow$ *no-fail* (*if* P *then* $\top$ *else* Q) (*unless* P f)
  by (*simp add*: *unless-def when-def*)

lemma *no-fail-fail* [*simp*, *wp*]:
  no-fail ⊥ *fail*
  by (*simp add*: *fail-def no-fail-def*)

lemmas [*wp*] = *non-fail-gets*

lemma *no-fail-assert* [*simp*, *wp*]:
  no-fail (λ-. *P*) (*assert P*)
  by (*simp add*: *assert-def*)

lemma *no-fail-assert-opt* [*simp*, *wp*]:
  no-fail (λ-. *P* ≠ *None*) (*assert-opt P*)
  by (*simp add*: *assert-opt-def split*: *option.splits*)

lemma *no-fail-case-option* [*wp*]:
  assumes *f*: *no-fail P f*
  assumes *g*: ⋀*x*. *no-fail* (*Q x*) (*g x*)
  shows *no-fail* (*if x* = *None then P else Q* (*the x*)) (*case-option f g x*)
  by (*clarsimp simp add*: *f g*)

lemma *no-fail-if* [*wp*]:
  ⟦ *P* ⟹ *no-fail Q f*; ¬*P* ⟹ *no-fail R g* ⟧ ⟹
  no-fail (*if P then Q else R*) (*if P then f else g*)
  by *simp*

lemma *no-fail-apply* [*wp*]:
  no-fail P (*f* (*g x*)) ⟹ *no-fail P* (*f* $ *g x*)
  by *simp*

lemma *no-fail-undefined* [*simp*, *wp*]:
  no-fail ⊥ *undefined*
  by (*simp add*: *no-fail-def*)

lemma *no-fail-returnOK* [*simp*, *wp*]:
  no-fail ⊤ (*returnOk x*)
  by (*simp add*: *returnOk-def*)

lemma *no-fail-bind* [*wp*]:
  assumes *f*: *no-fail P f*
  assumes *g*: ⋀*rv*. *no-fail* (*R rv*) (*g rv*)
  assumes *v*: ⦃*Q*⦄ *f* ⦃*R*⦄
  shows *no-fail* (*P and Q*) (*f* >>= (λ*rv*. *g rv*))
  apply (*clarsimp simp*: *no-fail-def bind-def*)
  apply (*rule conjI*)
   prefer *2*
   apply (*erule no-failD* [*OF f*])
  apply *clarsimp*
  apply (*drule* (*1*) *use-valid* [*OF* - *v*])

apply (*drule no-failD* [*OF g*])
apply *simp*
done

Empty results implies non-failure

lemma *empty-fail-modify* [*simp*, *wp*]:
empty-fail (*modify f*)
by (*simp add*: *empty-fail-def simpler-modify-def*)

lemma *empty-fail-gets* [*simp*, *wp*]:
empty-fail (*gets f*)
by (*simp add*: *empty-fail-def simpler-gets-def*)

lemma *empty-failD*:
$\llbracket$ *empty-fail m*; *fst* (*m s*) = {} $\rrbracket \Longrightarrow$ *snd* (*m s*)
by (*simp add*: *empty-fail-def*)

lemma *empty-fail-select-f* [*simp*]:
assumes *ef*: *fst S* = {} $\Longrightarrow$ *snd S*
shows *empty-fail* (*select-f S*)
by (*fastforce simp add*: *empty-fail-def select-f-def intro*: *ef*)

lemma *empty-fail-bind* [*simp*]:
$\llbracket$ *empty-fail a*; $\bigwedge$*x*. *empty-fail* (*b x*) $\rrbracket \Longrightarrow$ *empty-fail* (*a* >>= *b*)
apply (*simp add*: *bind-def empty-fail-def split-def*)
apply *clarsimp*
apply (*case-tac fst* (*a s*) = {})
apply *blast*
apply (*clarsimp simp*: *ex-in-conv* [*symmetric*])
done

lemma *empty-fail-return* [*simp*, *wp*]:
empty-fail (*return x*)
by (*simp add*: *empty-fail-def return-def*)

lemma *empty-fail-mapM* [*simp*]:
assumes *m*: $\bigwedge$*x*. *empty-fail* (*m x*)
shows *empty-fail* (*mapM m xs*)
proof (*induct xs*)
case *Nil*
thus *?case* **by** (*simp add*: *mapM-def sequence-def*)
next
case *Cons*
have *P*: $\bigwedge$*m x xs*. *mapM m* (*x* # *xs*) = (*do y* ← *m x*; *ys* ← (*mapM m xs*); *return* (*y* # *ys*) *od*)
by (*simp add*: *mapM-def sequence-def Let-def*)
from *Cons*
show *?case* **by** (*simp add*: *P m*)
qed

lemma *empty-fail* [*simp*]:
empty-fail fail
by (*simp add*: *fail-def empty-fail-def*)

lemma *empty-fail-assert-opt* [*simp*]:
empty-fail (*assert-opt x*)
by (*simp add*: *assert-opt-def split*: *option.splits*)

lemma *empty-fail-mk-ef*:
empty-fail (*mk-ef o m*)
by (*simp add*: *empty-fail-def mk-ef-def*)

lemma *empty-fail-gets-map*[*simp*]:
empty-fail (*gets-map f p*)
unfolding *gets-map-def* **by** *simp*

27.10 Failure

lemma *fail-wp*: $\{\!|\lambda x.\ \mathit{True}|\!\}\ \mathit{fail}\ \{\!|Q|\!\}$
by (*simp add*: *valid-def fail-def*)

lemma *failE-wp*: $\{\!|\lambda x.\ \mathit{True}|\!\}\ \mathit{fail}\ \{\!|Q|\!\},\{\!|E|\!\}$
by (*simp add*: *validE-def fail-wp*)

lemma *fail-update* [*iff*]:
$\mathit{fail}\ (f\ s) = \mathit{fail}\ s$
by (*simp add*: *fail-def*)

We can prove postconditions using hoare triples

lemma *post-by-hoare*: $[\![\ \{\!|P|\!\}\ f\ \{\!|Q|\!\};\ P\ s;\ (r,\ s') \in \mathit{fst}\ (f\ s)\ ]\!] \Longrightarrow Q\ r\ s'$
apply (*simp add*: *valid-def*)
apply *blast*
done

Weakest Precondition Rules

lemma *hoare-vcg-prop*:
$\{\!|\lambda s.\ P|\!\}\ f\ \{\!|\lambda rv\ s.\ P|\!\}$
by (*simp add*: *valid-def*)

lemma *return-wp*:
$\{\!|P\ x|\!\}\ \mathit{return}\ x\ \{\!|P|\!\}$
by(*simp add*:*valid-def return-def*)

lemma *get-wp*:
$\{\!|\lambda s.\ P\ s\ s|\!\}\ \mathit{get}\ \{\!|P|\!\}$
by(*simp add*:*valid-def split-def get-def*)

lemma *gets-wp*:

$\{\!|\lambda s.\ P\ (f\ s)\ s|\!\}$ *gets f* $\{\!|P|\!\}$
by(*simp add*:*valid-def split-def gets-def return-def get-def bind-def*)

lemma *modify-wp*:
$\{\!|\lambda s.\ P\ ()\ (f\ s)|\!\}$ *modify f* $\{\!|P|\!\}$
by(*simp add*:*valid-def split-def modify-def get-def put-def bind-def*)

lemma *put-wp*:
$\{\!|\lambda s.\ P\ ()\ x|\!\}$ *put x* $\{\!|P|\!\}$
by(*simp add*:*valid-def put-def*)

lemma *returnOk-wp*:
$\{\!|P\ x|\!\}$ *returnOk x* $\{\!|P|\!\},\{\!|E|\!\}$
by(*simp add*:*validE-def2 returnOk-def return-def*)

lemma *throwError-wp*:
$\{\!|E\ e|\!\}$ *throwError e* $\{\!|P|\!\},\{\!|E|\!\}$
by(*simp add*:*validE-def2 throwError-def return-def*)

lemma *returnOKE-R-wp* : $\{\!|P\ x|\!\}$ *returnOk x* $\{\!|P|\!\}, -$
by (*simp add*: *validE-R-def validE-def valid-def returnOk-def return-def*)

lemma *liftE-wp*:
$\{\!|P|\!\}\ f\ \{\!|Q|\!\} \Longrightarrow \{\!|P|\!\}$ *liftE f* $\{\!|Q|\!\},\{\!|E|\!\}$
by(*clarsimp simp*:*valid-def validE-def2 liftE-def split-def Let-def bind-def return-def*)

lemma *catch-wp*:
$[\![\ \bigwedge x.\ \{\!|E\ x|\!\}$ *handler x* $\{\!|Q|\!\};\ \{\!|P|\!\}\ f\ \{\!|Q|\!\},\{\!|E|\!\}\ ]\!] \Longrightarrow$
$\{\!|P|\!\}$ *catch f handler* $\{\!|Q|\!\}$
apply (*unfold catch-def valid-def validE-def return-def*)
apply (*fastforce simp*: *bind-def split*: *sum.splits*)
done

lemma *handleE′-wp*:
$[\![\ \bigwedge x.\ \{\!|F\ x|\!\}$ *handler x* $\{\!|Q|\!\},\{\!|E|\!\};\ \{\!|P|\!\}\ f\ \{\!|Q|\!\},\{\!|F|\!\}\ ]\!] \Longrightarrow$
$\{\!|P|\!\}$ *f <handle2> handler* $\{\!|Q|\!\},\{\!|E|\!\}$
apply (*unfold handleE′-def valid-def validE-def return-def*)
apply (*fastforce simp*: *bind-def split*: *sum.splits*)
done

lemma *handleE-wp*:
assumes *x*: $\bigwedge x.\ \{\!|F\ x|\!\}$ *handler x* $\{\!|Q|\!\},\{\!|E|\!\}$
assumes *y*: $\{\!|P|\!\}\ f\ \{\!|Q|\!\},\{\!|F|\!\}$
shows $\{\!|P|\!\}$ *f <handle> handler* $\{\!|Q|\!\},\{\!|E|\!\}$
by (*simp add*: *handleE-def handleE′-wp* [*OF x y*])

lemma *hoare-vcg-if-split*:
$[\![\ P \Longrightarrow \{\!|Q|\!\}\ f\ \{\!|S|\!\};\ \neg P \Longrightarrow \{\!|R|\!\}\ g\ \{\!|S|\!\}\ ]\!] \Longrightarrow$
$\{\!|\lambda s.\ (P \longrightarrow Q\ s) \wedge (\neg P \longrightarrow R\ s)|\!\}$ *if P then f else g* $\{\!|S|\!\}$

by *simp*

lemma *hoare-vcg-if-splitE*:
$[\![\ P \Longrightarrow \{\!|Q|\!\}\ f\ \{\!|S|\!\},\{\!|E|\!\};\ \neg P \Longrightarrow \{\!|R|\!\}\ g\ \{\!|S|\!\},\{\!|E|\!\}\ ]\!] \Longrightarrow$
$\{\!|\lambda s.\ (P \longrightarrow Q\ s) \wedge (\neg P \longrightarrow R\ s)|\!\}\ \mathit{if}\ P\ \mathit{then}\ f\ \mathit{else}\ g\ \{\!|S|\!\},\{\!|E|\!\}$
by *simp*

lemma *hoare-liftM-subst*: $\{\!|P|\!\}\ \mathit{liftM}\ f\ m\ \{\!|Q|\!\} = \{\!|P|\!\}\ m\ \{\!|Q \circ f|\!\}$
apply (*simp add*: *liftM-def bind-def return-def split-def*)
apply (*simp add*: *valid-def Ball-def*)
apply (*rule-tac f=All* **in** *arg-cong*)
apply (*rule ext*)
apply *fastforce*
done

lemma *liftE-validE*[*simp*]: $\{\!|P|\!\}\ \mathit{liftE}\ f\ \{\!|Q|\!\},\{\!|E|\!\} = \{\!|P|\!\}\ f\ \{\!|Q|\!\}$
apply (*simp add*: *liftE-liftM validE-def hoare-liftM-subst o-def*)
done

lemma *liftM-wp*: $\{\!|P|\!\}\ m\ \{\!|Q \circ f|\!\} \Longrightarrow \{\!|P|\!\}\ \mathit{liftM}\ f\ m\ \{\!|Q|\!\}$
by (*simp add*: *hoare-liftM-subst*)

lemma *hoare-liftME-subst*: $\{\!|P|\!\}\ \mathit{liftME}\ f\ m\ \{\!|Q|\!\},\{\!|E|\!\} = \{\!|P|\!\}\ m\ \{\!|Q \circ f|\!\},\{\!|E|\!\}$
apply (*simp add*: *validE-def liftME-liftM hoare-liftM-subst o-def*)
apply (*rule-tac f=valid P m* **in** *arg-cong*)
apply (*rule ext*)+
apply (*case-tac x*, *simp-all*)
done

lemma *liftME-wp*: $\{\!|P|\!\}\ m\ \{\!|Q \circ f|\!\},\{\!|E|\!\} \Longrightarrow \{\!|P|\!\}\ \mathit{liftME}\ f\ m\ \{\!|Q|\!\},\{\!|E|\!\}$
by (*simp add*: *hoare-liftME-subst*)

lemma *o-const-simp*[*simp*]: $(\lambda x.\ C) \circ f = (\lambda x.\ C)$
by (*simp add*: *o-def*)

lemma *hoare-vcg-split-case-option*:
$[\![\ \bigwedge x.\ x = \mathit{None} \Longrightarrow \{\!|P\ x|\!\}\ f\ x\ \{\!|R\ x|\!\};$
$\bigwedge x\ y.\ x = \mathit{Some}\ y \Longrightarrow \{\!|Q\ x\ y|\!\}\ g\ x\ y\ \{\!|R\ x|\!\}\ ]\!] \Longrightarrow$
$\{\!|\lambda s.\ (x = \mathit{None} \longrightarrow P\ x\ s) \wedge$
$(\forall y.\ x = \mathit{Some}\ y \longrightarrow Q\ x\ y\ s)|\!\}$
case x of None $\Rightarrow$ *f x*
| *Some y* $\Rightarrow$ *g x y*
$\{\!|R\ x|\!\}$
apply(*simp add*:*valid-def split-def*)
apply(*case-tac x*, *simp-all*)
done

lemma *hoare-vcg-split-case-optionE*:

assumes *none-case*: $\bigwedge x.\ x = None \Longrightarrow \{\!|P\ x|\!\}\ f\ x\ \{\!|R\ x|\!\},\{\!|E\ x|\!\}$
assumes *some-case*: $\bigwedge x\ y.\ x = Some\ y \Longrightarrow \{\!|Q\ x\ y|\!\}\ g\ x\ y\ \{\!|R\ x|\!\},\{\!|E\ x|\!\}$
shows $\{\!|\lambda s.\ (x = None \longrightarrow P\ x\ s) \wedge$
$(\forall y.\ x = Some\ y \longrightarrow Q\ x\ y\ s)|\!\}$
case x of None $\Rightarrow$ *f x*
$|$ *Some y* $\Rightarrow$ *g x y*
$\{\!|R\ x|\!\},\{\!|E\ x|\!\}$
apply(*case-tac x*, *simp-all*)
apply(*rule none-case*, *simp*)
apply(*rule some-case*, *simp*)
done

lemma *hoare-vcg-split-case-sum*:
$[\![\ \bigwedge x\ a.\ x = Inl\ a \Longrightarrow \{\!|P\ x\ a|\!\}\ f\ x\ a\ \{\!|R\ x|\!\};$
$\bigwedge x\ b.\ x = Inr\ b \Longrightarrow \{\!|Q\ x\ b|\!\}\ g\ x\ b\ \{\!|R\ x|\!\}\ ]\!] \Longrightarrow$
$\{\!|\lambda s.\ (\forall a.\ x = Inl\ a \longrightarrow P\ x\ a\ s) \wedge$
$(\forall b.\ x = Inr\ b \longrightarrow Q\ x\ b\ s)\ |\!\}$
case x of Inl a $\Rightarrow$ *f x a*
$|$ *Inr b* $\Rightarrow$ *g x b*
$\{\!|R\ x|\!\}$
apply(*simp add*:*valid-def split-def*)
apply(*case-tac x*, *simp-all*)
done

lemma *hoare-vcg-split-case-sumE*:
assumes *left-case*: $\bigwedge x\ a.\ x = Inl\ a \Longrightarrow \{\!|P\ x\ a|\!\}\ f\ x\ a\ \{\!|R\ x|\!\}$
assumes *right-case*: $\bigwedge x\ b.\ x = Inr\ b \Longrightarrow \{\!|Q\ x\ b|\!\}\ g\ x\ b\ \{\!|R\ x|\!\}$
shows $\{\!|\lambda s.\ (\forall a.\ x = Inl\ a \longrightarrow P\ x\ a\ s) \wedge$
$(\forall b.\ x = Inr\ b \longrightarrow Q\ x\ b\ s)\ |\!\}$
case x of Inl a $\Rightarrow$ *f x a*
$|$ *Inr b* $\Rightarrow$ *g x b*
$\{\!|R\ x|\!\}$
apply(*case-tac x*, *simp-all*)
apply(*rule left-case*, *simp*)
apply(*rule right-case*, *simp*)
done

lemma *hoare-vcg-precond-imp*:
$[\![\ \{\!|Q|\!\}\ f\ \{\!|R|\!\};\ \bigwedge s.\ P\ s \Longrightarrow Q\ s\ ]\!] \Longrightarrow \{\!|P|\!\}\ f\ \{\!|R|\!\}$
by (*fastforce simp add*:*valid-def*)

lemma *hoare-vcg-precond-impE*:
$[\![\ \{\!|Q|\!\}\ f\ \{\!|R|\!\},\{\!|E|\!\};\ \bigwedge s.\ P\ s \Longrightarrow Q\ s\ ]\!] \Longrightarrow \{\!|P|\!\}\ f\ \{\!|R|\!\},\{\!|E|\!\}$
by (*fastforce simp add*:*validE-def2*)

lemma *hoare-seq-ext*:
assumes *g-valid*: $\bigwedge x.\ \{\!|B\ x|\!\}\ g\ x\ \{\!|C|\!\}$
assumes *f-valid*: $\{\!|A|\!\}\ f\ \{\!|B|\!\}$
shows $\{\!|A|\!\}$ *do* $x \leftarrow f;\ g\ x$ *od* $\{\!|C|\!\}$

apply(*insert f-valid g-valid*)
apply(*blast intro*: *seq-ext'*)
done

lemma *hoare-vcg-seqE*:
assumes *g-valid*: $\bigwedge x.\ \{\!|B\ x|\!\}\ g\ x\ \{\!|C|\!\},\{\!|E|\!\}$
assumes *f-valid*: $\{\!|A|\!\}\ f\ \{\!|B|\!\},\{\!|E|\!\}$
shows $\{\!|A|\!\}\ doE\ x \leftarrow f;\ g\ x\ odE\ \{\!|C|\!\},\{\!|E|\!\}$
apply(*insert f-valid g-valid*)
apply(*blast intro*: *seqE'*)
done

lemma *hoare-seq-ext-nobind*:
$[\![\ \{\!|B|\!\}\ g\ \{\!|C|\!\};$
$\{\!|A|\!\}\ f\ \{\!|\lambda r\ s.\ B\ s|\!\}\ ]\!] \Longrightarrow$
$\{\!|A|\!\}\ do\ f;\ g\ od\ \{\!|C|\!\}$
apply (*clarsimp simp*: *valid-def bind-def Let-def split-def*)
apply *fastforce*
done

lemma *hoare-seq-ext-nobindE*:
$[\![\ \{\!|B|\!\}\ g\ \{\!|C|\!\},\{\!|E|\!\};$
$\{\!|A|\!\}\ f\ \{\!|\lambda r\ s.\ B\ s|\!\},\{\!|E|\!\}\ ]\!] \Longrightarrow$
$\{\!|A|\!\}\ doE\ f;\ g\ odE\ \{\!|C|\!\},\{\!|E|\!\}$
apply (*clarsimp simp*:*validE-def*)
apply (*simp add*:*bindE-def Let-def split-def bind-def lift-def*)
apply (*fastforce simp add*: *valid-def throwError-def return-def lift-def*
split: *sum.splits*)
done

lemma *hoare-chain*:
$[\![\ \{\!|P|\!\}\ f\ \{\!|Q|\!\};$
$\bigwedge s.\ R\ s \Longrightarrow P\ s;$
$\bigwedge r\ s.\ Q\ r\ s \Longrightarrow S\ r\ s\ ]\!] \Longrightarrow$
$\{\!|R|\!\}\ f\ \{\!|S|\!\}$
by(*fastforce simp add*:*valid-def split-def*)

lemma *validE-weaken*:
$[\![\ \{\!|P'|\!\}\ A\ \{\!|Q'|\!\},\{\!|E'|\!\};\ \bigwedge s.\ P\ s \Longrightarrow P'\ s;\ \bigwedge r\ s.\ Q'\ r\ s \Longrightarrow Q\ r\ s;\ \bigwedge r\ s.\ E'\ r\ s$
$\Longrightarrow E\ r\ s\ ]\!] \Longrightarrow \{\!|P|\!\}\ A\ \{\!|Q|\!\},\{\!|E|\!\}$
by (*fastforce simp*: *validE-def2 split*: *sum.splits*)

lemmas *hoare-chainE* = *validE-weaken*

lemma *hoare-vcg-handle-elseE*:
$[\![\ \{\!|P|\!\}\ f\ \{\!|Q|\!\},\{\!|E|\!\};$
$\bigwedge e.\ \{\!|E\ e|\!\}\ g\ e\ \{\!|R|\!\},\{\!|F|\!\};$
$\bigwedge x.\ \{\!|Q\ x|\!\}\ h\ x\ \{\!|R|\!\},\{\!|F|\!\}\ ]\!] \Longrightarrow$
$\{\!|P|\!\}\ f\ <handle>\ g\ <else>\ h\ \{\!|R|\!\},\{\!|F|\!\}$

apply (*simp add*: *handle-elseE-def validE-def*)
apply (*rule seq-ext*)
 apply *assumption*
apply (*case-tac x*, *simp-all*)
done

lemma *alternative-valid*:
assumes *x*: $\{\!|P|\!\}$ *f* $\{\!|Q|\!\}$
assumes *y*: $\{\!|P|\!\}$ f' $\{\!|Q|\!\}$
shows $\{\!|P|\!\}$ *f OR* f' $\{\!|Q|\!\}$
apply (*simp add*: *valid-def alternative-def*)
apply *safe*
 apply (*simp add*: *post-by-hoare* [*OF x*])
apply (*simp add*: *post-by-hoare* [*OF y*])
done

lemma *alternative-wp*:
assumes *x*: $\{\!|P|\!\}$ *f* $\{\!|Q|\!\}$
assumes *y*: $\{\!|P'|\!\}$ f' $\{\!|Q|\!\}$
shows $\{\!|P$ *and* $P'|\!\}$ *f OR* f' $\{\!|Q|\!\}$
apply (*rule alternative-valid*)
 apply (*rule hoare-pre-imp* [*OF - x*], *simp*)
apply (*rule hoare-pre-imp* [*OF - y*], *simp*)
done

lemma *alternativeE-wp*:
assumes *x*: $\{\!|P|\!\}$ *f* $\{\!|Q|\!\},\{\!|E|\!\}$ **and** *y*: $\{\!|P'|\!\}$ f' $\{\!|Q|\!\},\{\!|E|\!\}$
shows $\{\!|P$ *and* $P'|\!\}$ *f OR* f' $\{\!|Q|\!\},\{\!|E|\!\}$
apply (*unfold validE-def*)
apply (*wp add*: *x y alternative-wp* | *simp* | *fold validE-def*)+
done

lemma *alternativeE-R-wp*:
$[\![$ $\{\!|P|\!\}$ *f* $\{\!|Q|\!\},-$; $\{\!|P'|\!\}$ f' $\{\!|Q|\!\},-$ $]\!] \Longrightarrow \{\!|P$ *and* $P'|\!\}$ *f OR* f' $\{\!|Q|\!\},-$
apply (*simp add*: *validE-R-def*)
apply (*rule alternativeE-wp*)
 apply *assumption*+
done

lemma *alternative-R-wp*:
$[\![$ $\{\!|P|\!\}$ *f* $-,\{\!|Q|\!\}$; $\{\!|P'|\!\}$ *g* $-,\{\!|Q|\!\}$ $]\!] \Longrightarrow \{\!|P$ *and* $P'|\!\}$ $f \sqcap g$ $-,$ $\{\!|Q|\!\}$
by (*fastforce simp*: *alternative-def validE-E-def validE-def valid-def*)

lemma *select-wp*: $\{\!|\lambda s.\ \forall\, x \in S.\ Q\ x\ s|\!\}$ *select S* $\{\!|Q|\!\}$
by (*simp add*: *select-def valid-def*)

lemma *select-f-wp*:
$\{\!|\lambda s.\ \forall\, x{\in}fst\ S.\ Q\ x\ s|\!\}$ *select-f S* $\{\!|Q|\!\}$
by (*simp add*: *select-f-def valid-def*)

lemma *state-select-wp* [*wp*]: $\{\!|\ \lambda s.\ \forall t.\ (s,\ t) \in f \longrightarrow P\ ()\ t\ |\!\}$ *state-select f* $\{\!|\ P\ |\!\}$
  apply (*clarsimp simp*: *state-select-def*)
  apply (*clarsimp simp*: *valid-def*)
  done

lemma *condition-wp* [*wp*]:
  $[\![\ \{\!|\ Q\ |\!\}\ A\ \{\!|\ P\ |\!\};\ \{\!|\ R\ |\!\}\ B\ \{\!|\ P\ |\!\}\ ]\!] \Longrightarrow \{\!|\ \lambda s.\ \mathit{if}\ C\ s\ \mathit{then}\ Q\ s\ \mathit{else}\ R\ s\ |\!\}\ \mathit{condition}\ C\ A\ B\ \{\!|\ P\ |\!\}$
  apply (*clarsimp simp*: *condition-def*)
  apply (*clarsimp simp*: *valid-def pred-conj-def pred-neg-def split-def*)
  done

lemma *conditionE-wp* [*wp*]:
  $[\![\ \{\!|\ P\ |\!\}\ A\ \{\!|\ Q\ |\!\},\{\!|\ R\ |\!\};\ \{\!|\ P'\ |\!\}\ B\ \{\!|\ Q\ |\!\},\{\!|\ R\ |\!\}\ ]\!] \Longrightarrow\ \{\!|\ \lambda s.\ \mathit{if}\ C\ s\ \mathit{then}\ P\ s\ \mathit{else}\ P'\ s\ |\!\}\ \mathit{condition}\ C\ A\ B\ \{\!|Q|\!\},\{\!|R|\!\}$
  apply (*clarsimp simp*: *condition-def*)
  apply (*clarsimp simp*: *validE-def valid-def*)
  done

lemma *state-assert-wp* [*wp*]: $\{\!|\ \lambda s.\ f\ s \longrightarrow P\ ()\ s\ |\!\}$ *state-assert f* $\{\!|\ P\ |\!\}$
  apply (*clarsimp simp*: *state-assert-def get-def*
    assert-def bind-def valid-def return-def fail-def)
  done

The weakest precondition handler which works on conjunction

lemma *hoare-vcg-conj-lift*:
  assumes *x*: $\{\!|P|\!\}\ f\ \{\!|Q|\!\}$
  assumes *y*: $\{\!|P'|\!\}\ f\ \{\!|Q'|\!\}$
  shows $\{\!|\lambda s.\ P\ s \wedge P'\ s|\!\}\ f\ \{\!|\lambda rv\ s.\ Q\ rv\ s \wedge Q'\ rv\ s|\!\}$
  apply (*subst bipred-conj-def* [*symmetric*], *rule hoare-post-conj*)
   apply (*rule hoare-pre-imp* [*OF - x*], *simp*)
  apply (*rule hoare-pre-imp* [*OF - y*], *simp*)
  done

lemma *hoare-vcg-conj-liftE1*:
  $[\![\ \{\!|P|\!\}\ f\ \{\!|Q|\!\},-;\ \{\!|P'|\!\}\ f\ \{\!|Q'|\!\},\{\!|E|\!\}\ ]\!] \Longrightarrow$
  $\{\!|P\ \mathit{and}\ P'|\!\}\ f\ \{\!|\lambda r\ s.\ Q\ r\ s \wedge Q'\ r\ s|\!\},\{\!|E|\!\}$
  unfolding *valid-def validE-R-def validE-def*
  apply (*clarsimp simp*: *split-def split*: *sum.splits*)
  apply (*erule allE*, *erule* (*1*) *impE*)
  apply (*erule allE*, *erule* (*1*) *impE*)
  apply (*drule* (*1*) *bspec*)
  apply (*drule* (*1*) *bspec*)
  apply *clarsimp*
  done

lemma *hoare-vcg-disj-lift*:
  assumes *x*: $\{\!|P|\!\}\ \ f\ \{\!|Q|\!\}$

assumes y: $\{\!|P'|\!\}$ f $\{\!|Q'|\!\}$
shows $\{\!|\lambda s.\ P\ s \lor P'\ s|\!\}$ f $\{\!|\lambda rv\ s.\ Q\ rv\ s \lor Q'\ rv\ s|\!\}$
apply (*simp add*: *valid-def*)
apply *safe*
apply (*erule*(*1*) *post-by-hoare* [*OF x*])
apply (*erule notE*)
apply (*erule*(*1*) *post-by-hoare* [*OF y*])
done

lemma *hoare-vcg-const-Ball-lift*:
$[\![\ \bigwedge x.\ x \in S \Longrightarrow \{\!|P\ x|\!\}\ f\ \{\!|Q\ x|\!\}\ ]\!] \Longrightarrow \{\!|\lambda s.\ \forall x{\in}S.\ P\ x\ s|\!\}\ f\ \{\!|\lambda rv\ s.\ \forall x{\in}S.\ Q\ x\ rv\ s|\!\}$
by (*fastforce simp*: *valid-def*)

lemma *hoare-vcg-const-Ball-lift-R*:
$[\![\ \bigwedge x.\ x \in S \Longrightarrow \{\!|P\ x|\!\}\ f\ \{\!|Q\ x|\!\},-\ ]\!] \Longrightarrow$
$\{\!|\lambda s.\ \forall x \in S.\ P\ x\ s|\!\}\ f\ \{\!|\lambda rv\ s.\ \forall x \in S.\ Q\ x\ rv\ s|\!\},-$
apply (*simp add*: *validE-R-def validE-def*)
apply (*rule hoare-strengthen-post*)
apply (*erule hoare-vcg-const-Ball-lift*)
apply (*simp split*: *sum.splits*)
done

lemma *hoare-vcg-all-lift*:
$[\![\ \bigwedge x.\ \{\!|P\ x|\!\}\ f\ \{\!|Q\ x|\!\}\ ]\!] \Longrightarrow \{\!|\lambda s.\ \forall x.\ P\ x\ s|\!\}\ f\ \{\!|\lambda rv\ s.\ \forall x.\ Q\ x\ rv\ s|\!\}$
by (*fastforce simp*: *valid-def*)

lemma *hoare-vcg-all-lift-R*:
$(\bigwedge x.\ \{\!|P\ x|\!\}\ f\ \{\!|Q\ x|\!\},\ -) \Longrightarrow \{\!|\lambda s.\ \forall x.\ P\ x\ s|\!\}\ f\ \{\!|\lambda rv\ s.\ \forall x.\ Q\ x\ rv\ s|\!\},\ -$
by (*rule hoare-vcg-const-Ball-lift-R*[**where** *S*=*UNIV*, *simplified*])

lemma *hoare-vcg-imp-lift*:
$[\![\ \{\!|P'|\!\}\ f\ \{\!|\lambda rv\ s.\ \neg\ P\ rv\ s|\!\};\ \{\!|Q'|\!\}\ f\ \{\!|Q|\!\}\ ]\!] \Longrightarrow \{\!|\lambda s.\ P'\ s \lor Q'\ s|\!\}\ f\ \{\!|\lambda rv\ s.\ P\ rv\ s \longrightarrow Q\ rv\ s|\!\}$
apply (*simp only*: *imp-conv-disj*)
apply (*erule*(*1*) *hoare-vcg-disj-lift*)
done

lemma *hoare-vcg-imp-lift'*:
$[\![\ \{\!|P'|\!\}\ f\ \{\!|\lambda rv\ s.\ \neg\ P\ rv\ s|\!\};\ \{\!|Q'|\!\}\ f\ \{\!|Q|\!\}\ ]\!] \Longrightarrow \{\!|\lambda s.\ \neg\ P'\ s \longrightarrow Q'\ s|\!\}\ f\ \{\!|\lambda rv\ s.\ P\ rv\ s \longrightarrow Q\ rv\ s|\!\}$
apply (*simp only*: *imp-conv-disj*)
apply *simp*
apply (*erule* (*1*) *hoare-vcg-imp-lift*)
done

lemma *hoare-vcg-imp-conj-lift*[*wp-comb*]:
$\{\!|P|\!\}\ f\ \{\!|\lambda rv\ s.\ Q\ rv\ s \longrightarrow Q'\ rv\ s|\!\} \Longrightarrow \{\!|P'|\!\}\ f\ \{\!|\lambda rv\ s.\ (Q\ rv\ s \longrightarrow Q''\ rv\ s) \land$

$Q''' \ rv \ s\}$
  $\Longrightarrow \{P \ and \ P'\} \ f \ \{\lambda rv \ s. \ (Q \ rv \ s \longrightarrow Q' \ rv \ s \wedge Q'' \ rv \ s) \wedge Q''' \ rv \ s\}$
  by (*auto simp*: *valid-def*)

lemmas *hoare-vcg-imp-conj-lift'*[*wp-unsafe*] = *hoare-vcg-imp-conj-lift*[**where** $Q'''=\top\top$, *simplified*]

lemma *hoare-absorb-imp*:
  $\{ \ P \ \} \ f \ \{\lambda rv \ s. \ Q \ rv \ s \wedge R \ rv \ s \ \} \Longrightarrow \{ \ P \ \} \ f \ \{\lambda rv \ s. \ Q \ rv \ s \longrightarrow R \ rv \ s \ \}$
  by (*erule hoare-post-imp*[*rotated*], *blast*)

lemma *hoare-weaken-imp*:
  $[\![\ \bigwedge rv \ s. \ Q \ rv \ s \Longrightarrow Q' \ rv \ s \ ; \ \{P\} \ f \ \{\lambda rv \ s. \ Q' \ rv \ s \longrightarrow R \ rv \ s\} \ ]\!]$
   $\Longrightarrow \{P\} \ f \ \{\lambda rv \ s. \ Q \ rv \ s \longrightarrow R \ rv \ s\}$
  by (*clarsimp simp*: *NonDetMonad.valid-def split-def*)

lemma *hoare-vcg-const-imp-lift*:
  $[\![\ P \Longrightarrow \{Q\} \ m \ \{R\} \ ]\!] \Longrightarrow$
   $\{\lambda s. \ P \longrightarrow Q \ s\} \ m \ \{\lambda rv \ s. \ P \longrightarrow R \ rv \ s\}$
  by (*cases P*, *simp-all add*: *hoare-vcg-prop*)

lemma *hoare-vcg-const-imp-lift-R*:
  $(P \Longrightarrow \{Q\} \ m \ \{R\},-) \Longrightarrow \{\lambda s. \ P \longrightarrow Q \ s\} \ m \ \{\lambda rv \ s. \ P \longrightarrow R \ rv \ s\},-$
  by (*fastforce simp*: *validE-R-def validE-def valid-def split-def split*: *sum.splits*)

lemma *hoare-weak-lift-imp*:
  $\{P'\} \ f \ \{Q\} \Longrightarrow \{\lambda s. \ P \longrightarrow P' \ s\} \ f \ \{\lambda rv \ s. \ P \longrightarrow Q \ rv \ s\}$
  by (*auto simp add*: *valid-def split-def*)

lemma *hoare-vcg-weaken-imp*:
  $[\![\ \bigwedge rv \ s. \ Q \ rv \ s \Longrightarrow Q' \ rv \ s \ ; \ \{ \ P \ \} \ f \ \{\lambda rv \ s. \ Q' \ rv \ s \longrightarrow R \ rv \ s\} \ ]\!]$
  $\Longrightarrow \{ \ P \ \} \ f \ \{\lambda rv \ s. \ Q \ rv \ s \longrightarrow R \ rv \ s\}$
  by (*clarsimp simp*: *valid-def split-def*)

lemma *hoare-vcg-ex-lift*:
  $[\![\ \bigwedge x. \ \{P \ x\} \ f \ \{Q \ x\} \ ]\!] \Longrightarrow \{\lambda s. \ \exists x. \ P \ x \ s\} \ f \ \{\lambda rv \ s. \ \exists x. \ Q \ x \ rv \ s\}$
  by (*clarsimp simp*: *valid-def*, *blast*)

lemma *hoare-vcg-ex-lift-R1*:
  $(\bigwedge x. \ \{P \ x\} \ f \ \{Q\}, \ -) \Longrightarrow \{\lambda s. \ \exists x. \ P \ x \ s\} \ f \ \{Q\}, \ -$
  by (*fastforce simp*: *valid-def validE-R-def validE-def split*: *sum.splits*)

lemma *hoare-liftP-ext*:
  assumes $\bigwedge P \ x. \ m \ \{\lambda s. \ P \ (f \ s \ x)\}$
  shows $m \ \{\lambda s. \ P \ (f \ s)\}$
  unfolding *valid-def*
  apply *clarsimp*
  apply (*erule rsubst*[**where** $P=P$])
  apply (*rule ext*)

apply (*drule use-valid*, *rule assms*, *rule refl*)
apply *simp*
done

lemma *hoare-triv*: $\{\!|P|\!\}f\{\!|Q|\!\} \Longrightarrow \{\!|P|\!\}f\{\!|Q|\!\}$ **.**
lemma *hoare-trivE*: $\{\!|P|\!\}\ f\ \{\!|Q|\!\},\{\!|E|\!\} \Longrightarrow \{\!|P|\!\}\ f\ \{\!|Q|\!\},\{\!|E|\!\}$ **.**
lemma *hoare-trivE-R*: $\{\!|P|\!\}\ f\ \{\!|Q|\!\},- \Longrightarrow \{\!|P|\!\}\ f\ \{\!|Q|\!\},-$ **.**
lemma *hoare-trivR-R*: $\{\!|P|\!\}\ f\ -,\{\!|E|\!\} \Longrightarrow \{\!|P|\!\}\ f\ -,\{\!|E|\!\}$ **.**

lemma *hoare-weaken-preE-E*:
$[\![\ \{\!|P'|\!\}\ f\ -,\{\!|Q|\!\};\ \bigwedge s.\ P\ s \Longrightarrow P'\ s\ ]\!] \Longrightarrow \{\!|P|\!\}\ f\ -,\{\!|Q|\!\}$
by (*fastforce simp add*: *validE-E-def validE-def valid-def*)

lemma *hoare-vcg-E-conj*:
$[\![\ \{\!|P|\!\}\ f\ -,\{\!|E|\!\};\ \{\!|P'|\!\}\ f\ \{\!|Q'|\!\},\{\!|E'|\!\}\ ]\!]$
$\Longrightarrow \{\!|\lambda s.\ P\ s \wedge P'\ s|\!\}\ f\ \{\!|Q'|\!\},\ \{\!|\lambda rv\ s.\ E\ rv\ s \wedge E'\ rv\ s|\!\}$
apply (*unfold validE-def validE-E-def*)
apply (*rule hoare-post-imp* [*OF - hoare-vcg-conj-lift*], *simp-all*)
apply (*case-tac r*, *simp-all*)
done

lemma *hoare-vcg-E-elim*:
$[\![\ \{\!|P|\!\}\ f\ -,\{\!|E|\!\};\ \{\!|P'|\!\}\ f\ \{\!|Q|\!\},-\ ]\!]$
$\Longrightarrow \{\!|\lambda s.\ P\ s \wedge P'\ s|\!\}\ f\ \{\!|Q|\!\},\{\!|E|\!\}$
by (*rule hoare-post-impErr* [*OF hoare-vcg-E-conj*],
(*simp add*: *validE-R-def*)+)

lemma *hoare-vcg-R-conj*:
$[\![\ \{\!|P|\!\}\ f\ \{\!|Q|\!\},-;\ \{\!|P'|\!\}\ f\ \{\!|Q'|\!\},-\ ]\!]$
$\Longrightarrow \{\!|\lambda s.\ P\ s \wedge P'\ s|\!\}\ f\ \{\!|\lambda rv\ s.\ Q\ rv\ s \wedge Q'\ rv\ s|\!\},-$
apply (*unfold validE-R-def validE-def*)
apply (*rule hoare-post-imp* [*OF - hoare-vcg-conj-lift*], *simp-all*)
apply (*case-tac r*, *simp-all*)
done

lemma *valid-validE*:
$\{\!|P|\!\}\ f\ \{\!|\lambda rv.\ Q|\!\} \Longrightarrow \{\!|P|\!\}\ f\ \{\!|\lambda rv.\ Q|\!\},\{\!|\lambda rv.\ Q|\!\}$
apply (*simp add*: *validE-def*)
done

lemma *valid-validE2*:
$[\![\ \{\!|P|\!\}\ f\ \{\!|\lambda\text{-}.\ Q'|\!\};\ \bigwedge s.\ Q'\ s \Longrightarrow Q\ s;\ \bigwedge s.\ Q'\ s \Longrightarrow E\ s\ ]\!] \Longrightarrow \{\!|P|\!\}\ f\ \{\!|\lambda\text{-}.\ Q|\!\},\{\!|\lambda\text{-}.\ E|\!\}$
unfolding *valid-def validE-def*
by (*clarsimp split*: *sum.splits*) *blast*

lemma *validE-valid*: $\{\!|P|\!\}\ f\ \{\!|\lambda rv.\ Q|\!\},\{\!|\lambda rv.\ Q|\!\} \Longrightarrow \{\!|P|\!\}\ f\ \{\!|\lambda rv.\ Q|\!\}$
apply (*unfold validE-def*)

apply (*rule hoare-post-imp*)
defer
apply *assumption*
apply (*case-tac r*, *simp-all*)
done

lemma *valid-validE-R*:
$\{\!|P|\!\}$ f $\{\!|\lambda rv.\ Q|\!\} \Longrightarrow \{\!|P|\!\}$ f $\{\!|\lambda rv.\ Q|\!\},-$
by (*simp add*: *validE-R-def hoare-post-impErr* [*OF valid-validE*])

lemma *valid-validE-E*:
$\{\!|P|\!\}$ f $\{\!|\lambda rv.\ Q|\!\} \Longrightarrow \{\!|P|\!\}$ f $-,\{\!|\lambda rv.\ Q|\!\}$
by (*simp add*: *validE-E-def hoare-post-impErr* [*OF valid-validE*])

lemma *validE-validE-R*: $\{\!|P|\!\}$ f $\{\!|Q|\!\},\{\!|\top\top|\!\} \Longrightarrow \{\!|P|\!\}$ f $\{\!|Q|\!\},-$
by (*simp add*: *validE-R-def*)

lemma *validE-R-validE*: $\{\!|P|\!\}$ f $\{\!|Q|\!\},- \Longrightarrow \{\!|P|\!\}$ f $\{\!|Q|\!\},\{\!|\top\top|\!\}$
by (*simp add*: *validE-R-def*)

lemma *validE-validE-E*: $\{\!|P|\!\}$ f $\{\!|\top\top|\!\},\{\!|E|\!\} \Longrightarrow \{\!|P|\!\}$ f $-,\{\!|E|\!\}$
by (*simp add*: *validE-E-def*)

lemma *validE-E-validE*: $\{\!|P|\!\}$ f $-,\{\!|E|\!\} \Longrightarrow \{\!|P|\!\}$ f $\{\!|\top\top|\!\},\{\!|E|\!\}$
by (*simp add*: *validE-E-def*)

lemma *hoare-post-imp-R*: $[\![\ \{\!|P|\!\}$ f $\{\!|Q'|\!\},-;\ \bigwedge r\ s.\ Q'\ r\ s \Longrightarrow Q\ r\ s\ ]\!] \Longrightarrow \{\!|P|\!\}$ f
$\{\!|Q|\!\},-$
apply (*unfold validE-R-def*)
apply (*erule hoare-post-impErr*, *simp+*)
done

lemma *hoare-post-imp-E*: $[\![\ \{\!|P|\!\}$ f $-,\{\!|Q'|\!\};\ \bigwedge r\ s.\ Q'\ r\ s \Longrightarrow Q\ r\ s\ ]\!] \Longrightarrow \{\!|P|\!\}$ f
$-,\{\!|Q|\!\}$
apply (*unfold validE-E-def*)
apply (*erule hoare-post-impErr*, *simp+*)
done

lemma *hoare-post-comb-imp-conj*:
$[\![\ \{\!|P'|\!\}$ f $\{\!|Q|\!\};\ \{\!|P|\!\}$ f $\{\!|Q'|\!\};\ \bigwedge s.\ P\ s \Longrightarrow P'\ s\ ]\!] \Longrightarrow \{\!|P|\!\}$ f $\{\!|\lambda rv\ s.\ Q\ rv\ s \wedge Q'$
$rv\ s|\!\}$
apply (*rule hoare-pre-imp*)
defer
apply (*rule hoare-vcg-conj-lift*)
apply *assumption+*
apply *simp*
done

lemma *hoare-vcg-precond-impE-R*: $[\![\ \{\!|P'|\!\}$ f $\{\!|Q|\!\},-;\ \bigwedge s.\ P\ s \Longrightarrow P'\ s\ ]\!] \Longrightarrow \{\!|P|\!\}$

f $\{\!|Q|\!\}$,−
by (*unfold validE-R-def*, *rule hoare-vcg-precond-impE*, *simp*+)

lemma *valid-is-triple*:
valid P f Q = *triple-judgement P f* (*postcondition Q* ($\lambda s\ f$. *fst* ($f\ s$)))
by (*simp add*: *triple-judgement-def valid-def postcondition-def*)

lemma *validE-is-triple*:
validE P f Q E = *triple-judgement P f*
(*postconditions* (*postcondition Q* ($\lambda s\ f$. $\{(rv, s')$. (*Inr rv*, s') $\in$ *fst* ($f\ s$)$\}$))
(*postcondition E* ($\lambda s\ f$. $\{(rv, s')$. (*Inl rv*, s') $\in$ *fst* ($f\ s$)$\}$)))
apply (*simp add*: *validE-def triple-judgement-def valid-def postcondition-def postconditions-def split-def split*: *sum.split*)
apply *fastforce*
done

lemma *validE-R-is-triple*:
validE-R P f Q = *triple-judgement P f*
(*postcondition Q* ($\lambda s\ f$. $\{(rv, s')$. (*Inr rv*, s') $\in$ *fst* ($f\ s$)$\}$))
by (*simp add*: *validE-R-def validE-is-triple postconditions-def postcondition-def*)

lemma *validE-E-is-triple*:
validE-E P f E = *triple-judgement P f*
(*postcondition E* ($\lambda s\ f$. $\{(rv, s')$. (*Inl rv*, s') $\in$ *fst* ($f\ s$)$\}$))
by (*simp add*: *validE-E-def validE-is-triple postconditions-def postcondition-def*)

lemmas *hoare-wp-combs* = *hoare-vcg-conj-lift*

lemmas *hoare-wp-combsE* =
validE-validE-R
hoare-vcg-R-conj
hoare-vcg-E-elim
hoare-vcg-E-conj

lemmas *hoare-wp-state-combsE* =
valid-validE-R
hoare-vcg-R-conj[*OF valid-validE-R*]
hoare-vcg-E-elim[*OF valid-validE-E*]
hoare-vcg-E-conj[*OF valid-validE-E*]

lemmas *hoare-classic-wp-combs*
= *hoare-post-comb-imp-conj hoare-vcg-precond-imp hoare-wp-combs*
lemmas *hoare-classic-wp-combsE*
= *hoare-vcg-precond-impE hoare-vcg-precond-impE-R hoare-wp-combsE*
lemmas *hoare-classic-wp-state-combsE*
= *hoare-vcg-precond-impE*[*OF valid-validE*]
hoare-vcg-precond-impE-R[*OF valid-validE-R*] *hoare-wp-state-combsE*
lemmas *all-classic-wp-combs* =
hoare-classic-wp-state-combsE hoare-classic-wp-combsE hoare-classic-wp-combs

lemmas *hoare-wp-splits* [*wp-split*] =
hoare-seq-ext hoare-vcg-seqE handleE′-wp handleE-wp
validE-validE-R [*OF hoare-vcg-seqE* [*OF validE-R-validE*]]
validE-validE-R [*OF handleE′-wp* [*OF validE-R-validE*]]
validE-validE-R [*OF handleE-wp* [*OF validE-R-validE*]]
catch-wp hoare-vcg-if-split hoare-vcg-if-splitE
validE-validE-R [*OF hoare-vcg-if-splitE* [*OF validE-R-validE validE-R-validE*]]
liftM-wp liftME-wp
validE-validE-R [*OF liftME-wp* [*OF validE-R-validE*]]
validE-valid

lemmas [*wp-comb*] = *hoare-wp-state-combsE hoare-wp-combsE hoare-wp-combs*

lemmas [*wp*] = *hoare-vcg-prop*
wp-post-taut
return-wp
put-wp
get-wp
gets-wp
modify-wp
returnOk-wp
throwError-wp
fail-wp
failE-wp
liftE-wp
select-f-wp

lemmas [*wp-trip*] = *valid-is-triple validE-is-triple validE-E-is-triple validE-R-is-triple*

lemmas *validE-E-combs*[*wp-comb*] =
hoare-vcg-E-conj[**where** Q'=⊤⊤, *folded validE-E-def*]
valid-validE-E
hoare-vcg-E-conj[**where** Q'=⊤⊤, *folded validE-E-def*, *OF valid-validE-E*]

Simplifications on conjunction

lemma *hoare-post-eq*: $\llbracket\ Q = Q';\ \{\!|P|\!\}\ f\ \{\!|Q'|\!\}\ \rrbracket \Longrightarrow \{\!|P|\!\}\ f\ \{\!|Q|\!\}$
by *simp*
lemma *hoare-post-eqE1*: $\llbracket\ Q = Q';\ \{\!|P|\!\}\ f\ \{\!|Q'|\!\},\{\!|E|\!\}\ \rrbracket \Longrightarrow \{\!|P|\!\}\ f\ \{\!|Q|\!\},\{\!|E|\!\}$
by *simp*
lemma *hoare-post-eqE2*: $\llbracket\ E = E';\ \{\!|P|\!\}\ f\ \{\!|Q|\!\},\{\!|E'|\!\}\ \rrbracket \Longrightarrow \{\!|P|\!\}\ f\ \{\!|Q|\!\},\{\!|E|\!\}$
by *simp*
lemma *hoare-post-eqE-R*: $\llbracket\ Q = Q';\ \{\!|P|\!\}\ f\ \{\!|Q'|\!\},-\ \rrbracket \Longrightarrow \{\!|P|\!\}\ f\ \{\!|Q|\!\},-$
by *simp*

lemma *pred-conj-apply-elim*: $(\lambda r.\ Q\ r\ \mathit{and}\ Q'\ r) = (\lambda r\ s.\ Q\ r\ s \wedge Q'\ r\ s)$
by (*simp add*: *pred-conj-def*)
lemma *pred-conj-conj-elim*: $(\lambda r\ s.\ (Q\ r\ \mathit{and}\ Q'\ r)\ s \wedge Q''\ r\ s) = (\lambda r\ s.\ Q\ r\ s \wedge Q'\ r\ s \wedge Q''\ r\ s)$

by *simp*
lemma *conj-assoc-apply*: $(\lambda r\ s.\ (Q\ r\ s \wedge Q'\ r\ s) \wedge Q''\ r\ s) = (\lambda r\ s.\ Q\ r\ s \wedge Q'\ r\ s \wedge Q''\ r\ s)$
by *simp*
lemma *all-elim*: $(\lambda rv\ s.\ \forall x.\ P\ rv\ s) = P$
by *simp*
lemma *all-conj-elim*: $(\lambda rv\ s.\ (\forall x.\ P\ rv\ s) \wedge Q\ rv\ s) = (\lambda rv\ s.\ P\ rv\ s \wedge Q\ rv\ s)$
by *simp*

lemmas *vcg-rhs-simps* = *pred-conj-apply-elim pred-conj-conj-elim conj-assoc-apply all-elim all-conj-elim*

lemma *if-apply-reduct*: $\{\!|P|\!\}$ *If* $P'\ (f\ x)\ (g\ x)\ \{\!|Q|\!\} \Longrightarrow \{\!|P|\!\}$ *If* $P'\ f\ g\ x\ \{\!|Q|\!\}$
by (*cases* P', *simp-all*)
lemma *if-apply-reductE*: $\{\!|P|\!\}$ *If* $P'\ (f\ x)\ (g\ x)\ \{\!|Q|\!\},\{\!|E|\!\} \Longrightarrow \{\!|P|\!\}$ *If* $P'\ f\ g\ x\ \{\!|Q|\!\},\{\!|E|\!\}$
by (*cases* P', *simp-all*)
lemma *if-apply-reductE-R*: $\{\!|P|\!\}$ *If* $P'\ (f\ x)\ (g\ x)\ \{\!|Q|\!\},- \Longrightarrow \{\!|P|\!\}$ *If* $P'\ f\ g\ x\ \{\!|Q|\!\},-$
by (*cases* P', *simp-all*)

lemmas *hoare-wp-simps* [*wp-split*] =
vcg-rhs-simps [*THEN hoare-post-eq*] *vcg-rhs-simps* [*THEN hoare-post-eqE1*]
vcg-rhs-simps [*THEN hoare-post-eqE2*] *vcg-rhs-simps* [*THEN hoare-post-eqE-R*]
if-apply-reduct if-apply-reductE if-apply-reductE-R TrueI

schematic-goal *if-apply-test*: $\{\!|?Q|\!\}$ (*if A then returnOk else K fail*) $x\ \{\!|P|\!\},\{\!|E|\!\}$
by *wpsimp*

lemma *hoare-elim-pred-conj*:
$\{\!|P|\!\}\ f\ \{\!|\lambda r\ s.\ Q\ r\ s \wedge Q'\ r\ s|\!\} \Longrightarrow \{\!|P|\!\}\ f\ \{\!|\lambda r.\ Q\ r$ *and* $Q'\ r|\!\}$
by (*unfold pred-conj-def*)

lemma *hoare-elim-pred-conjE1*:
$\{\!|P|\!\}\ f\ \{\!|\lambda r\ s.\ Q\ r\ s \wedge Q'\ r\ s|\!\},\{\!|E|\!\} \Longrightarrow \{\!|P|\!\}\ f\ \{\!|\lambda r.\ Q\ r$ *and* $Q'\ r|\!\},\{\!|E|\!\}$
by (*unfold pred-conj-def*)

lemma *hoare-elim-pred-conjE2*:
$\{\!|P|\!\}\ f\ \{\!|Q|\!\},\ \{\!|\lambda x\ s.\ E\ x\ s \wedge E'\ x\ s|\!\} \Longrightarrow \{\!|P|\!\}\ f\ \{\!|Q|\!\},\{\!|\lambda x.\ E\ x$ *and* $E'\ x|\!\}$
by (*unfold pred-conj-def*)

lemma *hoare-elim-pred-conjE-R*:
$\{\!|P|\!\}\ f\ \{\!|\lambda r\ s.\ Q\ r\ s \wedge Q'\ r\ s|\!\},- \Longrightarrow \{\!|P|\!\}\ f\ \{\!|\lambda r.\ Q\ r$ *and* $Q'\ r|\!\},-$
by (*unfold pred-conj-def*)

lemmas *hoare-wp-pred-conj-elims* =
hoare-elim-pred-conj hoare-elim-pred-conjE1
hoare-elim-pred-conjE2 hoare-elim-pred-conjE-R

lemmas *hoare-weaken-preE* = *hoare-vcg-precond-impE*

lemmas *hoare-pre* [*wp-pre*] =
hoare-weaken-pre
hoare-weaken-preE
hoare-vcg-precond-impE-R
hoare-weaken-preE-E

declare *no-fail-pre* [*wp-pre*]

bundle *no-pre* = *hoare-pre* [*wp-pre del*] *no-fail-pre* [*wp-pre del*]

bundle *classic-wp-pre* = *hoare-pre* [*wp-pre del*] *no-fail-pre* [*wp-pre del*]
all-classic-wp-combs[*wp-comb del*] *all-classic-wp-combs*[*wp-comb*]

Miscellaneous lemmas on hoare triples

lemma *hoare-vcg-mp*:
assumes *a*: $\{\!|P|\!\}\ f\ \{\!|Q|\!\}$
assumes *b*: $\{\!|P|\!\}\ f\ \{\!|\lambda r\ s.\ Q\ r\ s \longrightarrow Q'\ r\ s|\!\}$
shows $\{\!|P|\!\}\ f\ \{\!|Q'|\!\}$
using *assms*
by (*auto simp*: *valid-def split-def*)

lemma *hoare-add-post*:
assumes *r*: $\{\!|P'|\!\}\ f\ \{\!|Q'|\!\}$
assumes *impP*: $\bigwedge s.\ P\ s \Longrightarrow P'\ s$
assumes *impQ*: $\{\!|P|\!\}\ f\ \{\!|\lambda rv\ s.\ Q'\ rv\ s \longrightarrow Q\ rv\ s|\!\}$
shows $\{\!|P|\!\}\ f\ \{\!|Q|\!\}$
apply (*rule hoare-chain*)
apply (*rule hoare-vcg-conj-lift*)
apply (*rule r*)
apply (*rule impQ*)
apply *simp*
apply (*erule impP*)
apply *simp*
done

lemma *hoare-gen-asmE*:
$(P \Longrightarrow \{\!|P'|\!\}\ f\ \{\!|Q|\!\},-) \Longrightarrow \{\!|P'\ and\ K\ P|\!\}\ f\ \{\!|Q|\!\}, -$
by (*simp add*: *validE-R-def validE-def valid-def*) *blast*

lemma *hoare-list-case*:
assumes *P1*: $\{\!|P1|\!\}\ f\ f1\ \{\!|Q|\!\}$
assumes *P2*: $\bigwedge y\ ys.\ xs = y\#ys \Longrightarrow \{\!|P2\ y\ ys|\!\}\ f\ (f2\ y\ ys)\ \{\!|Q|\!\}$
shows $\{\!|case\ xs\ of\ [] \Rightarrow P1 \mid y\#ys \Rightarrow P2\ y\ ys|\!\}$
$f\ (case\ xs\ of\ [] \Rightarrow f1 \mid y\#ys \Rightarrow f2\ y\ ys)$
$\{\!|Q|\!\}$
apply (*cases xs*; *simp*)

apply (*rule P1*)
apply (*rule P2*)
apply *simp*
done

lemma *hoare-when-wp* [*wp-split*]:
$[\![\ P \Longrightarrow \{\!|Q|\!\}\ f\ \{\!|R|\!\}\ ]\!] \Longrightarrow \{\!|\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R\ ()|\!\}\ \mathit{when}\ P\ f\ \{\!|R|\!\}$
by (*clarsimp simp*: *when-def valid-def return-def*)

lemma *hoare-unless-wp*[*wp-split*]:
$(\neg P \Longrightarrow \{\!|Q|\!\}\ f\ \{\!|R|\!\}) \Longrightarrow \{\!|\mathit{if}\ P\ \mathit{then}\ R\ ()\ \mathit{else}\ Q|\!\}\ \mathit{unless}\ P\ f\ \{\!|R|\!\}$
unfolding *unless-def* **by** *wp auto*

lemma *hoare-whenE-wp*:
$(P \Longrightarrow \{\!|Q|\!\}\ f\ \{\!|R|\!\},\ \{\!|E|\!\}) \Longrightarrow \{\!|\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R\ ()|\!\}\ \mathit{whenE}\ P\ f\ \{\!|R|\!\},\ \{\!|E|\!\}$
unfolding *whenE-def* **by** *clarsimp wp*

lemmas *hoare-whenE-wps*[*wp-split*]
= *hoare-whenE-wp hoare-whenE-wp*[*THEN validE-validE-R*] *hoare-whenE-wp*[*THEN validE-validE-E*]

lemma *hoare-unlessE-wp*:
$(\neg\ P \Longrightarrow \{\!|Q|\!\}\ f\ \{\!|R|\!\},\ \{\!|E|\!\}) \Longrightarrow \{\!|\mathit{if}\ P\ \mathit{then}\ R\ ()\ \mathit{else}\ Q|\!\}\ \mathit{unlessE}\ P\ f\ \{\!|R|\!\},\ \{\!|E|\!\}$
unfolding *unlessE-def* **by** *wp auto*

lemmas *hoare-unlessE-wps*[*wp-split*]
= *hoare-unlessE-wp hoare-unlessE-wp*[*THEN validE-validE-R*] *hoare-unlessE-wp*[*THEN validE-validE-E*]

lemma *hoare-use-eq*:
assumes *x*: $\bigwedge P.\ \{\!|\lambda s.\ P\ (f\ s)|\!\}\ m\ \{\!|\lambda rv\ s.\ P\ (f\ s)|\!\}$
assumes *y*: $\bigwedge f.\ \{\!|\lambda s.\ P\ f\ s|\!\}\ m\ \{\!|\lambda rv\ s.\ Q\ f\ s|\!\}$
shows $\{\!|\lambda s.\ P\ (f\ s)\ s|\!\}\ m\ \{\!|\lambda rv\ s.\ Q\ (f\ s :: {'c} :: type)\ s\ |\!\}$
apply (*rule-tac* $Q{=}\lambda rv\ s.\ \exists f'.\ f' = f\ s \wedge Q\ f'\ s$ **in** *hoare-post-imp*)
apply *simp*
apply (*wpsimp wp*: *hoare-vcg-ex-lift x y*)
done

lemma *hoare-return-sp*:
$\{\!|P|\!\}\ \mathit{return}\ x\ \{\!|\lambda r.\ P\ \mathit{and}\ K\ (r = x)|\!\}$
by (*simp add*: *valid-def return-def*)

lemma *hoare-fail-any* [*simp*]:
$\{\!|P|\!\}\ \mathit{fail}\ \{\!|Q|\!\}$ **by** *wp*

lemma *hoare-failE* [*simp*]: $\{\!|P|\!\}\ \mathit{fail}\ \{\!|Q|\!\},\{\!|E|\!\}$ **by** *wp*

lemma *hoare-FalseE* [*simp*]:
$\{\!|\lambda s.\ \mathit{False}|\!\}\ f\ \{\!|Q|\!\},\{\!|E|\!\}$

by (*simp add*: *valid-def validE-def*)

lemma *hoare-K-bind* [*wp-split*]:
$\{\!|P|\!\}$ *f* $\{\!|Q|\!\}$ $\Longrightarrow$ $\{\!|P|\!\}$ *K-bind f x* $\{\!|Q|\!\}$
by *simp*

lemma *validE-K-bind* [*wp-split*]:
$\{\!|$ *P* $|\!\}$ *x* $\{\!|$ *Q* $|\!\}$, $\{\!|$ *E* $|\!\}$ $\Longrightarrow$ $\{\!|$ *P* $|\!\}$ *K-bind x f* $\{\!|$ *Q* $|\!\}$, $\{\!|$ *E* $|\!\}$
by *simp*

Setting up the precondition case splitter.

lemma *wpc-helper-valid*:
$\{\!|Q|\!\}$ *g* $\{\!|S|\!\}$ $\Longrightarrow$ *wpc-helper* (P, P') (Q, Q') $\{\!|P|\!\}$ *g* $\{\!|S|\!\}$
by (*clarsimp simp*: *wpc-helper-def elim*!: *hoare-pre*)

lemma *wpc-helper-validE*:
$\{\!|Q|\!\}$ *f* $\{\!|R|\!\}$,$\{\!|E|\!\}$ $\Longrightarrow$ *wpc-helper* (P, P') (Q, Q') $\{\!|P|\!\}$ *f* $\{\!|R|\!\}$,$\{\!|E|\!\}$
by (*clarsimp simp*: *wpc-helper-def elim*!: *hoare-pre*)

lemma *wpc-helper-validE-R*:
$\{\!|Q|\!\}$ *f* $\{\!|R|\!\}$,$-$ $\Longrightarrow$ *wpc-helper* (P, P') (Q, Q') $\{\!|P|\!\}$ *f* $\{\!|R|\!\}$,$-$
by (*clarsimp simp*: *wpc-helper-def elim*!: *hoare-pre*)

lemma *wpc-helper-validR-R*:
$\{\!|Q|\!\}$ *f* $-$,$\{\!|E|\!\}$ $\Longrightarrow$ *wpc-helper* (P, P') (Q, Q') $\{\!|P|\!\}$ *f* $-$,$\{\!|E|\!\}$
by (*clarsimp simp*: *wpc-helper-def elim*!: *hoare-pre*)

lemma *wpc-helper-no-fail-final*:
no-fail Q f $\Longrightarrow$ *wpc-helper* (P, P') (Q, Q') (*no-fail P f*)
by (*clarsimp simp*: *wpc-helper-def elim*!: *no-fail-pre*)

lemma *wpc-helper-empty-fail-final*:
empty-fail f $\Longrightarrow$ *wpc-helper* (P, P') (Q, Q') (*empty-fail f*)
by (*clarsimp simp*: *wpc-helper-def*)

lemma *wpc-helper-validNF*:
$\{\!|Q|\!\}$ *g* $\{\!|S|\!\}!$ $\Longrightarrow$ *wpc-helper* (P, P') (Q, Q') $\{\!|P|\!\}$ *g* $\{\!|S|\!\}!$
apply (*clarsimp simp*: *wpc-helper-def*)
by (*metis hoare-vcg-precond-imp no-fail-pre validNF-def*)

wpc-setup $\lambda m.$ $\{\!|P|\!\}$ *m* $\{\!|Q|\!\}$ *wpc-helper-valid*
wpc-setup $\lambda m.$ $\{\!|P|\!\}$ *m* $\{\!|Q|\!\}$,$\{\!|E|\!\}$ *wpc-helper-validE*
wpc-setup $\lambda m.$ $\{\!|P|\!\}$ *m* $\{\!|Q|\!\}$,$-$ *wpc-helper-validE-R*
wpc-setup $\lambda m.$ $\{\!|P|\!\}$ *m* $-$,$\{\!|E|\!\}$ *wpc-helper-validR-R*
wpc-setup $\lambda m.$ *no-fail P m wpc-helper-no-fail-final*
wpc-setup $\lambda m.$ *empty-fail m wpc-helper-empty-fail-final*
wpc-setup $\lambda m.$ $\{\!|P|\!\}$ *m* $\{\!|Q|\!\}!$ *wpc-helper-validNF*

lemma *in-liftM*:

$((r,\ s') \in fst\ (liftM\ t\ f\ s)) = (\exists\, r'.\ (r',\ s') \in fst\ (f\ s) \wedge r = t\ r')$
apply (*simp add*: *liftM-def return-def bind-def*)
apply (*simp add*: *Bex-def*)
done

lemmas *handy-liftM-lemma* = *in-liftM*

lemma *hoare-fun-app-wp*[*wp*]:
$\{\!|P|\!\}\ f'\ x\ \{\!|Q'|\!\} \Longrightarrow \{\!|P|\!\}\ f'\ \$\ x\ \{\!|Q'|\!\}$
$\{\!|P|\!\}\ f\ x\ \{\!|Q|\!\},\{\!|E|\!\} \Longrightarrow \{\!|P|\!\}\ f\ \$\ x\ \{\!|Q|\!\},\{\!|E|\!\}$
$\{\!|P|\!\}\ f\ x\ \{\!|Q|\!\},- \Longrightarrow \{\!|P|\!\}\ f\ \$\ x\ \{\!|Q|\!\},-$
$\{\!|P|\!\}\ f\ x\ -,\{\!|E|\!\} \Longrightarrow \{\!|P|\!\}\ f\ \$\ x\ -,\{\!|E|\!\}$
by *simp+*

lemma *hoare-validE-pred-conj*:
$[\![\ \{\!|P|\!\}f\{\!|Q|\!\},\{\!|E|\!\};\ \{\!|P|\!\}f\{\!|R|\!\},\{\!|E|\!\}\ ]\!] \Longrightarrow \{\!|P|\!\}f\{\!|Q\ And\ R|\!\},\{\!|E|\!\}$
unfolding *valid-def validE-def* **by** (*simp add*: *split-def split*: *sum.splits*)

lemma *hoare-validE-conj*:
$[\![\ \{\!|P|\!\}f\{\!|Q|\!\},\{\!|E|\!\};\ \{\!|P|\!\}f\{\!|R|\!\},\{\!|E|\!\}\ ]\!] \Longrightarrow \{\!|P|\!\}\ f\ \{\!|\lambda r\ s.\ Q\ r\ s \wedge R\ r\ s|\!\},\{\!|E|\!\}$
unfolding *valid-def validE-def* **by** (*simp add*: *split-def split*: *sum.splits*)

lemmas *hoare-valid-validE* = *valid-validE*

lemma *liftE-validE-E* [*wp*]:
$\{\!|\top|\!\}\ liftE\ f\ -,\ \{\!|Q|\!\}$
by (*clarsimp simp*: *validE-E-def valid-def*)

declare *validE-validE-E*[*wp-comb*]

lemmas *if-validE-E* [*wp-split*] =
validE-validE-E [*OF hoare-vcg-if-splitE* [*OF validE-E-validE validE-E-validE*]]

lemma *returnOk-E* [*wp*]:
$\{\!|\top|\!\}\ returnOk\ r\ -,\ \{\!|Q|\!\}$
by (*simp add*: *validE-E-def*) *wp*

lemma *hoare-drop-imp*:
$\{\!|P|\!\}\ f\ \{\!|Q|\!\} \Longrightarrow \{\!|P|\!\}\ f\ \{\!|\lambda r\ s.\ R\ r\ s \longrightarrow Q\ r\ s|\!\}$
by (*auto simp*: *valid-def*)

lemma *hoare-drop-impE*:
$[\![\{\!|P|\!\}\ f\ \{\!|\lambda r.\ Q|\!\},\ \{\!|E|\!\}]\!] \Longrightarrow \{\!|P|\!\}\ f\ \{\!|\lambda r\ s.\ R\ r\ s \longrightarrow Q\ s|\!\},\ \{\!|E|\!\}$
by (*simp add*: *validE-weaken*)

lemma *hoare-drop-impE-R*:
$\{\!|P|\!\}\ f\ \{\!|Q|\!\},- \Longrightarrow \{\!|P|\!\}\ f\ \{\!|\lambda r\ s.\ R\ r\ s \longrightarrow Q\ r\ s|\!\},\ -$

by (*auto simp*: *validE-R-def validE-def valid-def split-def split*: *sum.splits*)

lemma *hoare-drop-impE-E*:
$\{\!|P|\!\}\ f\ -,\{\!|Q|\!\} \Longrightarrow \{\!|P|\!\}\ f\ -,\{\!|\lambda r\ s.\ R\ r\ s \longrightarrow Q\ r\ s|\!\}$
by (*auto simp*: *validE-E-def validE-def valid-def split-def split*: *sum.splits*)

lemmas *hoare-drop-imps* = *hoare-drop-imp hoare-drop-impE-R hoare-drop-impE-E*

lemma *hoare-drop-imp-conj*[*wp-unsafe*]:
$\{\!|P|\!\}\ f\ \{\!|Q'|\!\} \Longrightarrow \{\!|P'|\!\}\ f\ \{\!|\lambda rv\ s.\ (Q\ rv\ s \longrightarrow Q''\ rv\ s) \wedge Q'''\ rv\ s|\!\}$
$\Longrightarrow \{\!|P\ and\ P'|\!\}\ f\ \{\!|\lambda rv\ s.\ (Q\ rv\ s \longrightarrow Q'\ rv\ s \wedge Q''\ rv\ s) \wedge Q'''\ rv\ s|\!\}$
by (*auto simp*: *valid-def*)

lemmas *hoare-drop-imp-conj'*[*wp-unsafe*] = *hoare-drop-imp-conj*[**where** Q'''=⊤⊤, *simplified*]

lemma *bind-det-exec*:
$fst\ (a\ s) = \{(r,s')\} \Longrightarrow fst\ ((a >>= b)\ s) = fst\ (b\ r\ s')$
by (*simp add*: *bind-def*)

lemma *in-bind-det-exec*:
$fst\ (a\ s) = \{(r,s')\} \Longrightarrow (s'' \in fst\ ((a >>= b)\ s)) = (s'' \in fst\ (b\ r\ s'))$
by (*simp add*: *bind-def*)

lemma *exec-put*:
$(put\ s' >>= m)\ s = m\ ()\ s'$
by (*simp add*: *bind-def put-def*)

lemma *bind-execI*:
$[\![\ (r'',s'') \in fst\ (f\ s);\ \exists x \in fst\ (g\ r''\ s'').\ P\ x\ ]\!] \Longrightarrow$
$\exists x \in fst\ ((f >>= g)\ s).\ P\ x$
by (*force simp*: *in-bind split-def bind-def*)

lemma *True-E-E* [*wp*]: $\{\!|\top|\!\}\ f\ -,\{\!|\top\top|\!\}$
by (*auto simp*: *validE-E-def validE-def valid-def split*: *sum.splits*)

lemmas [*wp-split*] =
validE-validE-E [*OF hoare-vcg-seqE* [*OF validE-E-validE*]]

lemma *case-option-wp*:
assumes *x*: $\bigwedge x.\ \{\!|P\ x|\!\}\ m\ x\ \{\!|Q|\!\}$
assumes *y*: $\{\!|P'|\!\}\ m'\ \{\!|Q|\!\}$
shows $\{\!|\lambda s.\ (x = None \longrightarrow P'\ s) \wedge (x \neq None \longrightarrow P\ (the\ x)\ s)|\!\}$
$case\text{-}option\ m'\ m\ x\ \{\!|Q|\!\}$
apply (*cases x*; *simp*)
apply (*rule y*)
apply (*rule x*)

done

lemma *case-option-wpE*:
assumes x: $\bigwedge x.\ \{\!|P\ x|\!\}\ m\ x\ \{\!|Q|\!\},\{\!|E|\!\}$
assumes y: $\{\!|P'|\!\}\ m'\ \{\!|Q|\!\},\{\!|E|\!\}$
shows $\{\!|\lambda s.\ (x = None \longrightarrow P'\ s) \wedge (x \neq None \longrightarrow P\ (the\ x)\ s)|\!\}$
$case\text{-}option\ m'\ m\ x\ \{\!|Q|\!\},\{\!|E|\!\}$
apply (*cases x*; *simp*)
apply (*rule y*)
apply (*rule x*)
done

lemma *in-bindE*:
$(rv, s') \in fst\ ((f >>=E (\lambda rv'.\ g\ rv'))\ s) =$
$((\exists\, ex.\ rv = Inl\ ex \wedge (Inl\ ex, s') \in fst\ (f\ s)) \vee$
$(\exists\, rv'\ s''.\ (rv, s') \in fst\ (g\ rv'\ s'') \wedge (Inr\ rv', s'') \in fst\ (f\ s)))$
apply (*rule iffI*)
apply (*clarsimp simp*: *bindE-def bind-def*)
apply (*case-tac a*)
apply (*clarsimp simp*: *lift-def throwError-def return-def*)
apply (*clarsimp simp*: *lift-def*)
apply *safe*
apply (*clarsimp simp*: *bindE-def bind-def*)
apply (*erule rev-bexI*)
apply (*simp add*: *lift-def throwError-def return-def*)
apply (*clarsimp simp*: *bindE-def bind-def*)
apply (*erule rev-bexI*)
apply (*simp add*: *lift-def*)
done

lemmas [*wp-split*] = *validE-validE-E* [*OF liftME-wp*, *simplified*, *OF validE-E-validE*]

lemma *assert-A-True*[*simp*]: *assert True* = *return* ()
by (*simp add*: *assert-def*)

lemma *assert-wp* [*wp*]: $\{\!|\lambda s.\ P \longrightarrow Q\ ()\ s|\!\}\ assert\ P\ \{\!|Q|\!\}$
by (*cases P*, (*simp add*: *assert-def* | *wp*)+)

lemma *list-cases-wp*:
assumes a: $\{\!|P\text{-}A|\!\}\ a\ \{\!|Q|\!\}$
assumes b: $\bigwedge x\ xs.\ ts = x\#xs \Longrightarrow \{\!|P\text{-}B\ x\ xs|\!\}\ b\ x\ xs\ \{\!|Q|\!\}$
shows $\{\!|case\text{-}list\ P\text{-}A\ P\text{-}B\ ts|\!\}\ case\ ts\ of\ [] \Rightarrow a \mid x\ \#\ xs \Rightarrow b\ x\ xs\ \{\!|Q|\!\}$
by (*cases ts*, *auto simp*: *a b*)

lemma *whenE-throwError-wp*:
$\{\!|\lambda s.\ \neg Q \longrightarrow P\ s|\!\}\ whenE\ Q\ (throwError\ e)\ \{\!|\lambda rv.\ P|\!\}, -$
unfolding *whenE-def* **by** *wpsimp*

lemma *select-throwError-wp*:
$\{\!|\lambda s.\ \forall x{\in}S.\ Q\ x\ s|\!\}$ *select* S >>= *throwError* $-$, $\{\!|Q|\!\}$
by (*simp add*: *bind-def throwError-def return-def select-def validE-E-def validE-def valid-def*)

lemma *assert-opt-wp*[*wp*]:
$\{\!|\lambda s.\ x \neq None \longrightarrow Q\ (the\ x)\ s|\!\}$ *assert-opt* x $\{\!|Q|\!\}$
by (*case-tac* x, (*simp add*: *assert-opt-def* | *wp*)+)

lemma *gets-the-wp*[*wp*]:
$\{\!|\lambda s.\ (f\ s \neq None) \longrightarrow Q\ (the\ (f\ s))\ s|\!\}$ *gets-the* f $\{\!|Q|\!\}$
by (*unfold gets-the-def*, *wp*)

lemma *gets-the-wp'*:
$\{\!|\lambda s.\ \forall rv.\ f\ s = Some\ rv \longrightarrow Q\ rv\ s|\!\}$ *gets-the* f $\{\!|Q|\!\}$
unfolding *gets-the-def* **by** *wpsimp*

lemma *gets-map-wp*:
$\{\!|\lambda s.\ f\ s\ p \neq None \longrightarrow Q\ (the\ (f\ s\ p))\ s|\!\}$ *gets-map* f p $\{\!|Q|\!\}$
unfolding *gets-map-def* **by** *wpsimp*

lemma *gets-map-wp'*[*wp*]:
$\{\!|\lambda s.\ \forall rv.\ f\ s\ p = Some\ rv \longrightarrow Q\ rv\ s|\!\}$ *gets-map* f p $\{\!|Q|\!\}$
unfolding *gets-map-def* **by** *wpsimp*

lemma *no-fail-gets-map*[*wp*]:
no-fail $(\lambda s.\ f\ s\ p \neq None)$ (*gets-map* f p)
unfolding *gets-map-def* **by** *wpsimp*

lemma *hoare-vcg-set-pred-lift*:
assumes $\bigwedge P\ x.\ m\ \{\!|\ \lambda s.\ P\ (f\ x\ s)\ |\!\}$
shows $m\ \{\!|\ \lambda s.\ P\ \{x.\ f\ x\ s\}\ |\!\}$
using *assms*[**where** $P{=}\lambda x\ .\ x$] *assms*[**where** $P{=}Not$] *use-valid*
by (*fastforce simp*: *valid-def elim*!: *rsubst*[**where** $P{=}P$])

lemma *hoare-vcg-set-pred-lift-mono*:
assumes f: $\bigwedge x.\ m\ \{\!|\ f\ x\ |\!\}$
assumes *mono*: $\bigwedge A\ B.\ A \subseteq B \Longrightarrow P\ A \Longrightarrow P\ B$
shows $m\ \{\!|\ \lambda s.\ P\ \{x.\ f\ x\ s\}\ |\!\}$
by (*fastforce simp*: *valid-def elim*!: *mono*[*rotated*] *dest*: *use-valid*[*OF - f*])

28 validNF Rules

28.1 Basic validNF theorems

lemma *validNF* [*intro?*]:
$[\![\ \{\!|\ P\ |\!\}\ f\ \{\!|\ Q\ |\!\};\ no\text{-}fail\ P\ f\ ]\!] \Longrightarrow \{\!|\ P\ |\!\}\ f\ \{\!|\ Q\ |\!\}!$
by (*clarsimp simp*: *validNF-def*)

lemma *validNF-valid*: $[\![\ \{\!| \ P \ |\!\} \ f \ \{\!| \ Q \ |\!\}! \ ]\!] \Longrightarrow \{\!| \ P \ |\!\} \ f \ \{\!| \ Q \ |\!\}$
by (*clarsimp simp*: *validNF-def*)

lemma *validNF-no-fail*: $[\![\ \{\!| \ P \ |\!\} \ f \ \{\!| \ Q \ |\!\}! \ ]\!] \Longrightarrow$ *no-fail P f*
by (*clarsimp simp*: *validNF-def*)

lemma *snd-validNF*:
$[\![\ \{\!| \ P \ |\!\} \ f \ \{\!| \ Q \ |\!\}!; \ P \ s \ ]\!] \Longrightarrow \neg \ snd \ (f \ s)$
by (*clarsimp simp*: *validNF-def no-fail-def*)

lemma *use-validNF*:
$[\![\ (r', s') \in fst \ (f \ s); \ \{\!| \ P \ |\!\} \ f \ \{\!| \ Q \ |\!\}!; \ P \ s \ ]\!] \Longrightarrow Q \ r' \ s'$
by (*fastforce simp*: *validNF-def valid-def*)

28.2 validNF weakest pre-condition rules

lemma *validNF-return* [*wp*]:
$\{\!| \ P \ x \ |\!\} \ return \ x \ \{\!| \ P \ |\!\}!$
by (*wp validNF*)+

lemma *validNF-get* [*wp*]:
$\{\!| \ \lambda s. \ P \ s \ s \ |\!\} \ get \ \{\!| \ P \ |\!\}!$
by (*wp validNF*)+

lemma *validNF-put* [*wp*]:
$\{\!| \ \lambda s. \ P \ () \ x \ |\!\} \ put \ x \ \{\!| \ P \ |\!\}!$
by (*wp validNF*)+

lemma *validNF-K-bind* [*wp*]:
$\{\!| \ P \ |\!\} \ x \ \{\!| \ Q \ |\!\}! \Longrightarrow \{\!| \ P \ |\!\} \ K\text{-}bind \ x \ f \ \{\!| \ Q \ |\!\}!$
by *simp*

lemma *validNF-fail* [*wp*]:
$\{\!| \ \lambda s. \ False \ |\!\} \ fail \ \{\!| \ Q \ |\!\}!$
by (*clarsimp simp*: *validNF-def fail-def no-fail-def*)

lemma *validNF-prop* [*wp-unsafe*]:
$[\![\ no\text{-}fail \ (\lambda s. \ P) \ f \ ]\!] \Longrightarrow \{\!| \ \lambda s. \ P \ |\!\} \ f \ \{\!| \ \lambda rv \ s. \ P \ |\!\}!$
by (*wp validNF*)+

lemma *validNF-post-conj* [*intro*!]:
$[\![\ \{\!| \ P \ |\!\} \ a \ \{\!| \ Q \ |\!\}!; \ \{\!| \ P \ |\!\} \ a \ \{\!| \ R \ |\!\}! \ ]\!] \Longrightarrow \{\!| \ P \ |\!\} \ a \ \{\!| \ Q \ And \ R \ |\!\}!$
by (*auto simp*: *validNF-def*)

lemma *no-fail-or*:
$[\![no\text{-}fail \ P \ a; \ no\text{-}fail \ Q \ a]\!] \Longrightarrow no\text{-}fail \ (P \ or \ Q) \ a$
by (*clarsimp simp*: *no-fail-def*)

lemma *validNF-pre-disj* [*intro*!]:
$\llbracket$ $\{\!|$ P $|\!\}$ a $\{\!|$ R $|\!\}$!; $\{\!|$ Q $|\!\}$ a $\{\!|$ R $|\!\}$! $\rrbracket \Longrightarrow$ $\{\!|$ P *or* Q $|\!\}$ a $\{\!|$ R $|\!\}$!
by (*rule validNF*) (*auto dest*: *validNF-valid validNF-no-fail intro*: *no-fail-or*)

definition *validNF-property* $Q\ s\ b \equiv \neg\ snd\ (b\ s) \wedge (\forall (r', s') \in fst\ (b\ s).\ Q\ r'\ s')$

lemma *validNF-is-triple* [*wp-trip*]:
validNF $P\ f\ Q$ = *triple-judgement* $P\ f$ (*validNF-property* Q)
apply (*clarsimp simp*: *validNF-def triple-judgement-def validNF-property-def*)
apply (*auto simp*: *no-fail-def valid-def*)
done

lemma *validNF-weaken-pre*[*wp-pre*]:
$\llbracket\{\!|Q|\!\}$ a $\{\!|R|\!\}$!; $\bigwedge s.\ P\ s \Longrightarrow Q\ s\rrbracket \Longrightarrow \{\!|P|\!\}$ a $\{\!|R|\!\}$!
by (*metis hoare-pre-imp no-fail-pre validNF-def*)

lemma *validNF-post-comb-imp-conj*:
$\llbracket$ $\{\!|P'|\!\}$ f $\{\!|Q|\!\}$!; $\{\!|P|\!\}$ f $\{\!|Q'|\!\}$!; $\bigwedge s.\ P\ s \Longrightarrow P'\ s$ $\rrbracket \Longrightarrow \{\!|P|\!\}$ f $\{\!|\lambda rv\ s.\ Q\ rv\ s \wedge Q'$
$rv\ s|\!\}$!
by (*fastforce simp*: *validNF-def valid-def*)

lemma *validNF-post-comb-conj-L*:
$\llbracket$ $\{\!|P'|\!\}$ f $\{\!|Q|\!\}$!; $\{\!|P|\!\}$ f $\{\!|Q'|\!\}$ $\rrbracket \Longrightarrow \{\!|\lambda s.\ P\ s \wedge P'\ s$ $|\!\}$ f $\{\!|\lambda rv\ s.\ Q\ rv\ s \wedge Q'\ rv\ s|\!\}$!
apply (*clarsimp simp*: *validNF-def valid-def no-fail-def*)
apply *force*
done

lemma *validNF-post-comb-conj-R*:
$\llbracket$ $\{\!|P'|\!\}$ f $\{\!|Q|\!\}$; $\{\!|P|\!\}$ f $\{\!|Q'|\!\}$! $\rrbracket \Longrightarrow \{\!|\lambda s.\ P\ s \wedge P'\ s$ $|\!\}$ f $\{\!|\lambda rv\ s.\ Q\ rv\ s \wedge Q'\ rv\ s|\!\}$!
apply (*clarsimp simp*: *validNF-def valid-def no-fail-def*)
apply *force*
done

lemma *validNF-post-comb-conj*:
$\llbracket$ $\{\!|P'|\!\}$ f $\{\!|Q|\!\}$!; $\{\!|P|\!\}$ f $\{\!|Q'|\!\}$! $\rrbracket \Longrightarrow \{\!|\lambda s.\ P\ s \wedge P'\ s$ $|\!\}$ f $\{\!|\lambda rv\ s.\ Q\ rv\ s \wedge Q'\ rv$
$s|\!\}$!
apply (*clarsimp simp*: *validNF-def valid-def no-fail-def*)
apply *force*
done

lemma *validNF-if-split* [*wp-split*]:
$\llbracket P \Longrightarrow \{\!|Q|\!\}$ f $\{\!|S|\!\}$!; $\neg\ P \Longrightarrow \{\!|R|\!\}$ g $\{\!|S|\!\}$!$\rrbracket \Longrightarrow \{\!|\lambda s.\ (P \longrightarrow Q\ s) \wedge (\neg\ P \longrightarrow R$
$s)|\!\}$ *if P then f else g* $\{\!|S|\!\}$!
by *simp*

lemma *validNF-vcg-conj-lift*:
$\llbracket$ $\{\!|P|\!\}$ f $\{\!|Q|\!\}$!; $\{\!|P'|\!\}$ f $\{\!|Q'|\!\}$! $\rrbracket \Longrightarrow$

$\{\!|\lambda s.\ P\ s \wedge P'\ s|\!\}\ f\ \{\!|\lambda rv\ s.\ Q\ rv\ s \wedge Q'\ rv\ s|\!\}!$
apply (*subst bipred-conj-def*[*symmetric*], *rule validNF-post-conj*)
apply (*erule validNF-weaken-pre*, *fastforce*)
apply (*erule validNF-weaken-pre*, *fastforce*)
done

lemma *validNF-vcg-disj-lift*:
$[\![\ \{\!|P|\!\}\ f\ \{\!|Q|\!\}!;\ \{\!|P'|\!\}\ f\ \{\!|Q'|\!\}!\ ]\!] \Longrightarrow$
$\{\!|\lambda s.\ P\ s \vee P'\ s|\!\}\ f\ \{\!|\lambda rv\ s.\ Q\ rv\ s \vee Q'\ rv\ s|\!\}!$
apply (*clarsimp simp*: *validNF-def*)
apply *safe*
apply (*auto intro*!: *hoare-vcg-disj-lift*)[*1*]
apply (*clarsimp simp*: *no-fail-def*)
done

lemma *validNF-vcg-all-lift* [*wp*]:
$[\![\ \bigwedge x.\ \{\!|P\ x|\!\}\ f\ \{\!|Q\ x|\!\}!\ ]\!] \Longrightarrow \{\!|\lambda s.\ \forall x.\ P\ x\ s|\!\}\ f\ \{\!|\lambda rv\ s.\ \forall x.\ Q\ x\ rv\ s|\!\}!$
apply *atomize*
apply (*rule validNF*)
apply (*clarsimp simp*: *validNF-def*)
apply (*rule hoare-vcg-all-lift*)
apply *force*
apply (*clarsimp simp*: *no-fail-def validNF-def*)
done

lemma *validNF-bind* [*wp-split*]:
$[\![\ \bigwedge x.\ \{\!|B\ x|\!\}\ g\ x\ \{\!|C|\!\}!;\ \{\!|A|\!\}\ f\ \{\!|B|\!\}!\ ]\!] \Longrightarrow$
$\{\!|A|\!\}\ do\ x \leftarrow f;\ g\ x\ od\ \{\!|C|\!\}!$
apply (*rule validNF*)
apply (*metis validNF-valid hoare-seq-ext*)
apply (*clarsimp simp*: *no-fail-def validNF-def bind-def′ valid-def*)
apply *blast*
done

lemmas *validNF-seq-ext* = *validNF-bind*

28.3 validNF compound rules

lemma *validNF-state-assert* [*wp*]:
$\{\!|\ \lambda s.\ P\ ()\ s \wedge G\ s\ \ |\!\}$ *state-assert G* $\{\!|\ P\ |\!\}!$
apply (*rule validNF*)
apply *wpsimp*
apply (*clarsimp simp*: *no-fail-def state-assert-def*
bind-def′ assert-def return-def get-def)
done

lemma *validNF-modify* [*wp*]:
$\{\!|\ \lambda s.\ P\ ()\ (f\ s)\ |\!\}$ *modify f* $\{\!|\ P\ |\!\}!$
apply (*clarsimp simp*: *modify-def*)

apply *wp*
done

lemma *validNF-gets* [*wp*]:
$\{\!|\lambda s.\ P\ (f\ s)\ s|\!\}\ gets\ f\ \{\!|P|\!\}!$
apply (*clarsimp simp*: *gets-def*)
apply *wp*
done

lemma *validNF-condition* [*wp*]:
$[\![\ \{\!|\ Q\ |\!\}\ A\ \{\!|P|\!\}!;\ \{\!|\ R\ |\!\}\ B\ \{\!|P|\!\}!]\!] \Longrightarrow \{\!|\lambda s.\ if\ C\ s\ then\ Q\ s\ else\ R\ s|\!\}\ condition\ C\ A\ B\ \{\!|P|\!\}!$
apply *rule*
apply (*drule validNF-valid*)+
apply (*erule* (*1*) *condition-wp*)
apply (*drule validNF-no-fail*)+
apply (*clarsimp simp*: *no-fail-def condition-def*)
done

lemma *validNF-alt-def*:
$validNF\ P\ m\ Q = (\forall s.\ P\ s \longrightarrow ((\forall (r', s') \in fst\ (m\ s).\ Q\ r'\ s') \wedge \neg\ snd\ (m\ s)))$
by (*fastforce simp*: *validNF-def valid-def no-fail-def*)

lemma *validNF-assert* [*wp*]:
$\{\!|\ (\lambda s.\ P)\ and\ (R\ ())\ |\!\}\ assert\ P\ \{\!|\ R\ |\!\}!$
apply (*rule validNF*)
apply (*clarsimp simp*: *valid-def in-return*)
apply (*clarsimp simp*: *no-fail-def return-def*)
done

lemma *validNF-false-pre*:
$\{\!|\ \lambda\text{-}.\ False\ |\!\}\ P\ \{\!|\ Q\ |\!\}!$
by (*clarsimp simp*: *validNF-def no-fail-def*)

lemma *validNF-chain*:
$[\![\{\!|P'|\!\}\ a\ \{\!|R'|\!\}!;\ \bigwedge s.\ P\ s \Longrightarrow P'\ s;\ \bigwedge r\ s.\ R'\ r\ s \Longrightarrow R\ r\ s]\!] \Longrightarrow \{\!|P|\!\}\ a\ \{\!|R|\!\}!$
by (*fastforce simp*: *validNF-def valid-def no-fail-def Ball-def*)

lemma *validNF-case-prod* [*wp*]:
$[\![\ \bigwedge x\ y.\ validNF\ (P\ x\ y)\ (B\ x\ y)\ Q\ ]\!] \Longrightarrow validNF\ (case\text{-}prod\ P\ v)\ (case\text{-}prod\ (\lambda x\ y.\ B\ x\ y)\ v)\ Q$
by (*metis prod.exhaust split-conv*)

lemma *validE-NF-case-prod* [*wp*]:
$[\![\ \bigwedge a\ b.\ \{\!|P\ a\ b|\!\}\ f\ a\ b\ \{\!|Q|\!\},\ \{\!|E|\!\}!\ ]\!] \Longrightarrow$
$\{\!|case\ x\ of\ (a,\ b) \Rightarrow P\ a\ b|\!\}\ case\ x\ of\ (a,\ b) \Rightarrow f\ a\ b\ \{\!|Q|\!\},\ \{\!|E|\!\}!$
apply (*clarsimp simp*: *validE-NF-alt-def*)
apply (*erule validNF-case-prod*)
done

lemma *no-fail-is-validNF-True*: *no-fail P s* = ($\{\!|$ *P* $|\!\}$ *s* $\{\!|$ λ- -. *True* $|\!\}$!)
by (*clarsimp simp*: *no-fail-def validNF-def valid-def*)

28.4 validNF reasoning in the exception monad

lemma *validE-NF* [*intro?*]:
$[\![$ $\{\!|$ *P* $|\!\}$ *f* $\{\!|$ *Q* $|\!\}$,$\{\!|$ *E* $|\!\}$; *no-fail P f* $]\!] \Longrightarrow$ $\{\!|$ *P* $|\!\}$ *f* $\{\!|$ *Q* $|\!\}$,$\{\!|$ *E* $|\!\}$!
apply (*clarsimp simp*: *validE-NF-def*)
done

lemma *validE-NF-valid*:
$[\![$ $\{\!|$ *P* $|\!\}$ *f* $\{\!|$ *Q* $|\!\}$,$\{\!|$ *E* $|\!\}$! $]\!] \Longrightarrow$ $\{\!|$ *P* $|\!\}$ *f* $\{\!|$ *Q* $|\!\}$,$\{\!|$ *E* $|\!\}$
apply (*clarsimp simp*: *validE-NF-def*)
done

lemma *validE-NF-no-fail*:
$[\![$ $\{\!|$ *P* $|\!\}$ *f* $\{\!|$ *Q* $|\!\}$,$\{\!|$ *E* $|\!\}$! $]\!] \Longrightarrow$ *no-fail P f*
apply (*clarsimp simp*: *validE-NF-def*)
done

lemma *validE-NF-weaken-pre*[*wp-pre*]:
$[\![\{\!|Q|\!\}$ *a* $\{\!|R|\!\}$,$\{\!|E|\!\}$!; $\bigwedge s.$ *P s* $\Longrightarrow$ *Q s*$]\!] \Longrightarrow \{\!|P|\!\}$ *a* $\{\!|R|\!\}$,$\{\!|E|\!\}$!
apply (*clarsimp simp*: *validE-NF-alt-def*)
apply (*erule validNF-weaken-pre*)
apply *simp*
done

lemma *validE-NF-post-comb-conj-L*:
$[\![$ $\{\!|P|\!\}$ *f* $\{\!|Q|\!\}$, $\{\!|$ *E* $|\!\}$!; $\{\!|P'|\!\}$ *f* $\{\!|Q'|\!\}$, $\{\!|$ λ- -. *True* $|\!\}$ $]\!] \Longrightarrow$ $\{\!|\lambda s.$ *P s* $\wedge$ *P' s* $|\!\}$ *f* $\{\!|\lambda rv\ s.$ *Q rv s* $\wedge$ *Q' rv s*$|\!\}$, $\{\!|$ *E* $|\!\}$!
apply (*clarsimp simp*: *validE-NF-alt-def validE-def validNF-def valid-def no-fail-def split*: *sum.splits*)
apply *force*
done

lemma *validE-NF-post-comb-conj-R*:
$[\![$ $\{\!|P|\!\}$ *f* $\{\!|Q|\!\}$, $\{\!|$ λ- -. *True* $|\!\}$; $\{\!|P'|\!\}$ *f* $\{\!|Q'|\!\}$, $\{\!|$ *E* $|\!\}$! $]\!] \Longrightarrow$ $\{\!|\lambda s.$ *P s* $\wedge$ *P' s* $|\!\}$ *f* $\{\!|\lambda rv\ s.$ *Q rv s* $\wedge$ *Q' rv s*$|\!\}$, $\{\!|$ *E* $|\!\}$!
apply (*clarsimp simp*: *validE-NF-alt-def validE-def validNF-def valid-def no-fail-def split*: *sum.splits*)
apply *force*
done

lemma *validE-NF-post-comb-conj*:
$[\![$ $\{\!|P|\!\}$ *f* $\{\!|Q|\!\}$, $\{\!|$ *E* $|\!\}$!; $\{\!|P'|\!\}$ *f* $\{\!|Q'|\!\}$, $\{\!|$ *E* $|\!\}$! $]\!] \Longrightarrow$ $\{\!|\lambda s.$ *P s* $\wedge$ *P' s* $|\!\}$ *f* $\{\!|\lambda rv\ s.$ *Q rv s* $\wedge$ *Q' rv s*$|\!\}$, $\{\!|$ *E* $|\!\}$!
apply (*clarsimp simp*: *validE-NF-alt-def validE-def validNF-def valid-def no-fail-def split*: *sum.splits*)

apply *force*
done

lemma *validE-NF-chain*:
$[\![\{\!|P'|\!\}\ a\ \{\!|R'|\!\},\{\!|E'|\!\}!;$
$\bigwedge s.\ P\ s \Longrightarrow P'\ s;$
$\bigwedge r'\ s'.\ R'\ r'\ s' \Longrightarrow R\ r'\ s';$
$\bigwedge r''\ s''.\ E'\ r''\ s'' \Longrightarrow E\ r''\ s'']\!] \Longrightarrow$
$\{\!|\lambda s.\ P\ s\ |\!\}\ a\ \{\!|\lambda r'\ s'.\ R\ r'\ s'|\!\},\{\!|\lambda r''\ s''.\ E\ r''\ s''|\!\}!$
by (*fastforce simp*: *validE-NF-def validE-def2 no-fail-def Ball-def split*: *sum.splits*)

lemma *validE-NF-bind-wp* [*wp*]:
$[\![\bigwedge x.\ \{\!|B\ x|\!\}\ g\ x\ \{\!|C|\!\},\ \{\!|E|\!\}!;\ \{\!|A|\!\}\ f\ \{\!|B|\!\},\ \{\!|E|\!\}!]\!] \Longrightarrow \{\!|A|\!\}\ f\ >>=E\ (\lambda x.\ g\ x)\ \{\!|C|\!\},$
$\{\!|E|\!\}!$
apply (*unfold validE-NF-alt-def bindE-def*)
apply (*rule validNF-bind* [*rotated*])
apply *assumption*
apply (*clarsimp simp*: *lift-def throwError-def split*: *sum.splits*)
apply *wpsimp*
done

lemma *validNF-catch* [*wp*]:
$[\![\bigwedge x.\ \{\!|E\ x|\!\}\ handler\ x\ \{\!|Q|\!\}!;\ \{\!|P|\!\}\ f\ \{\!|Q|\!\},\ \{\!|E|\!\}!]\!] \Longrightarrow \{\!|P|\!\}\ f\ <catch>\ (\lambda x.\ handler$
$x)\ \{\!|Q|\!\}!$
apply (*unfold validE-NF-alt-def catch-def*)
apply (*rule validNF-bind* [*rotated*])
apply *assumption*
apply (*clarsimp simp*: *lift-def throwError-def split*: *sum.splits*)
apply *wp*
done

lemma *validNF-throwError* [*wp*]:
$\{\!|E\ e|\!\}\ throwError\ e\ \{\!|P|\!\},\ \{\!|E|\!\}!$
by (*unfold validE-NF-alt-def throwError-def o-def*) *wpsimp*

lemma *validNF-returnOk* [*wp*]:
$\{\!|P\ e|\!\}\ returnOk\ e\ \{\!|P|\!\},\ \{\!|E|\!\}!$
by (*clarsimp simp*: *validE-NF-alt-def returnOk-def*) *wpsimp*

lemma *validNF-whenE* [*wp*]:
$(P \Longrightarrow \{\!|Q|\!\}\ f\ \{\!|R|\!\},\ \{\!|E|\!\}!) \Longrightarrow \{\!|if\ P\ then\ Q\ else\ R\ ()|\!\}\ whenE\ P\ f\ \{\!|R|\!\},\ \{\!|E|\!\}!$
unfolding *whenE-def* **by** *clarsimp wp*

lemma *validNF-nobindE* [*wp*]:
$[\![\ \{\!|B|\!\}\ g\ \{\!|C|\!\},\{\!|E|\!\}!;$
$\{\!|A|\!\}\ f\ \{\!|\lambda r\ s.\ B\ s|\!\},\{\!|E|\!\}!\ ]\!] \Longrightarrow$
$\{\!|A|\!\}\ doE\ f;\ g\ odE\ \{\!|C|\!\},\{\!|E|\!\}!$
by *clarsimp wp*

Setup triple rules for *validE-NF* so that we can use wp combinator rules.

definition *validE-NF-property* $Q\ E\ s\ b \equiv \neg\ snd\ (b\ s)$
$\quad \wedge (\forall (r', s') \in fst\ (b\ s).\ case\ r'\ of\ Inl\ x \Rightarrow E\ x\ s' \mid Inr\ x \Rightarrow Q\ x\ s')$

lemma *validE-NF-is-triple* [*wp-trip*]:
validE-NF P f Q E = triple-judgement P f (validE-NF-property Q E)
apply (*clarsimp simp*: *validE-NF-def validE-def2 no-fail-def triple-judgement-def validE-NF-property-def split*: *sum.splits*)
apply *blast*
done

lemma *validNF-cong*:
$[\![\bigwedge s.\ P\ s = P'\ s;\ \bigwedge s.\ P\ s \Longrightarrow m\ s = m'\ s;$
$\quad \bigwedge r'\ s'\ s.\ [\![P\ s;\ (r', s') \in fst\ (m\ s)]\!] \Longrightarrow Q\ r'\ s' = Q'\ r'\ s']\!] \Longrightarrow$
$(\{\!|\ P\ |\!\}\ m\ \{\!|\ Q\ |\!\}!) = (\{\!|\ P'\ |\!\}\ m'\ \{\!|\ Q'\ |\!\}!)$
by (*fastforce simp*: *validNF-alt-def*)

lemma *validE-NF-liftE* [*wp*]:
$\{\!|P|\!\}\ f\ \{\!|Q|\!\}! \Longrightarrow \{\!|P|\!\}\ liftE\ f\ \{\!|Q|\!\},\{\!|E|\!\}!$
by (*wpsimp simp*: *validE-NF-alt-def liftE-def*)

lemma *validE-NF-handleE'* [*wp*]:
$[\![\bigwedge x.\ \{\!|F\ x|\!\}\ handler\ x\ \{\!|Q|\!\},\{\!|E|\!\}!;\ \{\!|P|\!\}\ f\ \{\!|Q|\!\},\{\!|F|\!\}!\ ]\!] \Longrightarrow$
$\{\!|P|\!\}\ f\ <handle2>\ (\lambda x.\ handler\ x)\ \{\!|Q|\!\},\{\!|E|\!\}!$
apply (*unfold validE-NF-alt-def handleE'-def*)
apply (*rule validNF-bind* [*rotated*])
apply *assumption*
apply (*clarsimp split*: *sum.splits*)
apply *wpsimp*
done

lemma *validE-NF-handleE* [*wp*]:
$[\![\bigwedge x.\ \{\!|F\ x|\!\}\ handler\ x\ \{\!|Q|\!\},\{\!|E|\!\}!;\ \{\!|P|\!\}\ f\ \{\!|Q|\!\},\{\!|F|\!\}!\ ]\!] \Longrightarrow$
$\{\!|P|\!\}\ f\ <handle>\ handler\ \{\!|Q|\!\},\{\!|E|\!\}!$
apply (*unfold handleE-def*)
apply (*metis validE-NF-handleE'*)
done

lemma *validE-NF-condition* [*wp*]:
$[\![\{\!|\ Q\ |\!\}\ A\ \{\!|P|\!\},\{\!|\ E\ |\!\}!;\ \{\!|\ R\ |\!\}\ B\ \{\!|P|\!\},\{\!|\ E\ |\!\}!]\!]$
$\quad \Longrightarrow \{\!|\lambda s.\ if\ C\ s\ then\ Q\ s\ else\ R\ s|\!\}\ condition\ C\ A\ B\ \{\!|P|\!\},\{\!|\ E\ |\!\}!$
apply *rule*
apply (*drule validE-NF-valid*)+
apply *wp*
apply (*drule validE-NF-no-fail*)+
apply (*clarsimp simp*: *no-fail-def condition-def*)
done

Strengthen setup.

context *strengthen-implementation* **begin**

lemma *strengthen-hoare* [*strg*]:
$(\bigwedge r\ s.\ st\ F\ (\longrightarrow)\ (Q\ r\ s)\ (R\ r\ s))$
$\Longrightarrow st\ F\ (\longrightarrow)\ (\{\!|P|\!\}\ f\ \{\!|Q|\!\})\ (\{\!|P|\!\}\ f\ \{\!|R|\!\})$
by (*cases F*, *auto elim*: *hoare-strengthen-post*)

lemma *strengthen-validE-R-cong*[*strg*]:
$(\bigwedge r\ s.\ st\ F\ (\longrightarrow)\ (Q\ r\ s)\ (R\ r\ s))$
$\Longrightarrow st\ F\ (\longrightarrow)\ (\{\!|P|\!\}\ f\ \{\!|Q|\!\},\ -)\ (\{\!|P|\!\}\ f\ \{\!|R|\!\},\ -)$
by (*cases F*, *auto intro*: *hoare-post-imp-R*)

lemma *strengthen-validE-cong*[*strg*]:
$(\bigwedge r\ s.\ st\ F\ (\longrightarrow)\ (Q\ r\ s)\ (R\ r\ s))$
$\Longrightarrow (\bigwedge r\ s.\ st\ F\ (\longrightarrow)\ (S\ r\ s)\ (T\ r\ s))$
$\Longrightarrow st\ F\ (\longrightarrow)\ (\{\!|P|\!\}\ f\ \{\!|Q|\!\},\ \{\!|S|\!\})\ (\{\!|P|\!\}\ f\ \{\!|R|\!\},\ \{\!|T|\!\})$
by (*cases F*, *auto elim*: *hoare-post-impErr*)

lemma *strengthen-validE-E-cong*[*strg*]:
$(\bigwedge r\ s.\ st\ F\ (\longrightarrow)\ (S\ r\ s)\ (T\ r\ s))$
$\Longrightarrow st\ F\ (\longrightarrow)\ (\{\!|P|\!\}\ f\ -,\ \{\!|S|\!\})\ (\{\!|P|\!\}\ f\ -,\ \{\!|T|\!\})$
by (*cases F*, *auto elim*: *hoare-post-impErr simp*: *validE-E-def*)

lemma *wpfix-strengthen-hoare*:
$(\bigwedge s.\ st\ (\neg\ F)\ (\longrightarrow)\ (P\ s)\ (P'\ s))$
$\Longrightarrow (\bigwedge r\ s.\ st\ F\ (\longrightarrow)\ (Q\ r\ s)\ (Q'\ r\ s))$
$\Longrightarrow st\ F\ (\longrightarrow)\ (\{\!|P|\!\}\ f\ \{\!|Q|\!\})\ (\{\!|P'|\!\}\ f\ \{\!|Q'|\!\})$
by (*cases F*, *auto elim*: *hoare-chain*)

lemma *wpfix-strengthen-validE-R-cong*:
$(\bigwedge s.\ st\ (\neg\ F)\ (\longrightarrow)\ (P\ s)\ (P'\ s))$
$\Longrightarrow (\bigwedge r\ s.\ st\ F\ (\longrightarrow)\ (Q\ r\ s)\ (Q'\ r\ s))$
$\Longrightarrow st\ F\ (\longrightarrow)\ (\{\!|P|\!\}\ f\ \{\!|Q|\!\},\ -)\ (\{\!|P'|\!\}\ f\ \{\!|Q'|\!\},\ -)$
by (*cases F*, *auto elim*: *hoare-chainE simp*: *validE-R-def*)

lemma *wpfix-strengthen-validE-cong*:
$(\bigwedge s.\ st\ (\neg\ F)\ (\longrightarrow)\ (P\ s)\ (P'\ s))$
$\Longrightarrow (\bigwedge r\ s.\ st\ F\ (\longrightarrow)\ (Q\ r\ s)\ (R\ r\ s))$
$\Longrightarrow (\bigwedge r\ s.\ st\ F\ (\longrightarrow)\ (S\ r\ s)\ (T\ r\ s))$
$\Longrightarrow st\ F\ (\longrightarrow)\ (\{\!|P|\!\}\ f\ \{\!|Q|\!\},\ \{\!|S|\!\})\ (\{\!|P'|\!\}\ f\ \{\!|R|\!\},\ \{\!|T|\!\})$
by (*cases F*, *auto elim*: *hoare-chainE*)

lemma *wpfix-strengthen-validE-E-cong*:
$(\bigwedge s.\ st\ (\neg\ F)\ (\longrightarrow)\ (P\ s)\ (P'\ s))$
$\Longrightarrow (\bigwedge r\ s.\ st\ F\ (\longrightarrow)\ (S\ r\ s)\ (T\ r\ s))$
$\Longrightarrow st\ F\ (\longrightarrow)\ (\{\!|P|\!\}\ f\ -,\ \{\!|S|\!\})\ (\{\!|P'|\!\}\ f\ -,\ \{\!|T|\!\})$
by (*cases F*, *auto elim*: *hoare-chainE simp*: *validE-E-def*)

lemma *wpfix-no-fail-cong*:
$(\bigwedge s.\ st\ (\neg\ F)\ (\longrightarrow)\ (P\ s)\ (P'\ s))$

$\Longrightarrow$ *st F* $(\longrightarrow)$ (*no-fail P f*) (*no-fail P' f*)
by (*cases F*, *auto elim*: *no-fail-pre*)

lemmas *nondet-wpfix-strgs* =
wpfix-strengthen-validE-R-cong
wpfix-strengthen-validE-E-cong
wpfix-strengthen-validE-cong
wpfix-strengthen-hoare
wpfix-no-fail-cong

end

lemmas *nondet-wpfix-strgs*[*wp-fix-strgs*]
= *strengthen-implementation.nondet-wpfix-strgs*

end
theory *NonDetMonad-call*
imports *NonDetMonadVCG*
begin

definition *call* :: $('s, 'a)$ *nondet-monad* $\Rightarrow ('t \Rightarrow 's) \Rightarrow ('t \Rightarrow 's \Rightarrow 't) \Rightarrow ('t, 'a)$ *nondet-monad*
where *call md r u* $\equiv$ ($\lambda t.$ ($\{(x,y).\ \exists a \in (fst\ (md\ (r\ t))).\ x = fst\ a \wedge y = u\ t\ (snd\ a)\}$, $snd\ (md\ (r\ t))$))

definition
call-spec c r u A B $\equiv$ *call c r u*

lemma *call-add-spec*: *call c r u* = *call-spec c r u A B*
by (*clarsimp simp*: *call-spec-def*)

lemma *hoare-wp-call-spec*[*wp*]:
$\{\!|A|\!\}\ c\ \{\!|B|\!\} \Longrightarrow \{\!|\lambda s.\ A\ (r\ s) \wedge (\forall a\ x.\ (A\ (r\ s) \longrightarrow B\ a\ x) \longrightarrow Q\ a\ (u\ s\ x))|\!\}$ *call-spec c r u A B* $\{\!|Q|\!\}$
apply(*simp add*:*valid-def call-def split-def call-spec-def*) **by** *fast*

record *state* = *x* :: *nat y* :: *int*

definition *updx* :: *nat* $\Rightarrow$ (*nat*,*nat*) *nondet-monad*
where *updx i* $\equiv$ *do x* $\leftarrow$ *get*; *put* $(x + i)$; *return* $(x+i)$ *od*

definition *updy* :: *int* $\Rightarrow$ (*int*,*int*) *nondet-monad*
where *updy i* $\equiv$ *do x* $\leftarrow$ *get*; *put* $(x + i)$; *return* $(x+i)$ *od*

abbreviation *rx* $\equiv$ *x*
abbreviation *ux* $\equiv (\lambda s\ i.\ s(\!|x := i|\!))$
abbreviation *ry* $\equiv$ *y*
abbreviation *uy* $\equiv (\lambda s\ i.\ s(\!|y := i|\!))$

definition *upd1* :: *nat* ⇒ *int* ⇒ (*state*,*unit*) *nondet-monad*
  where *upd1 i j* ≡
    do
      x ← *call* (*updx i*) *rx ux*;
      y ← *call* (*updy j*) *ry uy*;
      return ()
    od

lemma $\{\!|\lambda s.\ s = t|\!\}$ *upd1 i j* $\{\!|\lambda r\ s.\ s = t(\!|x := x\ t + i,\ y := y\ t + j|\!)|\!\}$
  by (*simp add*:*upd1-def updx-def updy-def valid-def return-def bind-def put-def get-def call-def*)

end
theory *type-Decls*

imports *lib/Monad-WP/NonDetMonad lib/Monad-WP/NonDetMonad-call ../common/commfunc*

begin

29 data types definitions and states

29.1 System Data types

type-synonym *u16* = *nat*
type-synonym *u32* = *nat*
type-synonym *u64* = *nat*
type-synonym *i64* = *nat*
type-synonym *usize* = *nat*
type-synonym *isize* = *nat*
type-synonym *u8* = *char*
type-synonym *vm-id* = *u64*
type-synonym *cpu-id* = *u64*

type-synonym *vcpu-num* = *nat*

type-synonym *region-num* = *nat*
type-synonym *page-num* = *nat*
type-synonym *region-size* = *nat*
type-synonym *page-free* = *nat*
type-synonym *page-last* = *nat*

type-synonym *mem-region-index* = *nat*
type-synonym *page-index* = *nat*
type-synonym *bitmap-region* = *page-num* ⇒ *mem-region-index option*
type-synonym *bitmap-heap* = *page-num* ⇒ *page-index option*

definition *GIC-SGIS-NUM* = *nat 16*
definition *GIC-PPIS-NUM* = *nat 16*
definition *BITMAP-UNIT-LEN* = *nat 32*
definition *INTERRUPT-NUM-MAX* = *nat 1024*
definition *GIC-INTS-MAX* ≡ *INTERRUPT-NUM-MAX*
definition *GIC-PRIVINT-NUM* = *GIC-SGIS-NUM* + *GIC-PPIS-NUM*
definition *CPU-MASTER* ≡ *nat 0*
definition *INTERRUPT-IRQ-IPI* ≡ *nat 1*

type-synonym *bitmap* = *bool list*

record *context'* = *gpr*::*u64 list*
spsr::*u64*
elr::*u64*
dummy::*u64*

record *vm-context* = *esr*::*u32*
far::*u64*
hpfar::*u64*

type-synonym *spinlock-t* = *u32*

datatype *vm-type* = *VM-T-OS* | *VM-T-BMA*

record *cpu-pt* = *lvl1*::*u64 list*
lvl2::*u64 list*

lvl3::*u64 list*

definition *spin-lock* :: *spinlock-t* ⇒ (*'s*, *unit*)*nondet-monad* **where** *spin-lock sp* ≡ *return* ()
definition *spin-unlock* :: *spinlock-t* ⇒ (*'s*, *unit*)*nondet-monad* **where** *spin-unlock sp* ≡ *return* ()

29.2 Bitmap Operations

definition *extract-list* :: *bitmap* ⇒ *u64* ⇒ *u64* ⇒ *bitmap* **where**
extract-list mp b e ≡ (*drop b* (*take e mp*))

definition *bitmap-set* :: *bitmap* ⇒ *u64* ⇒ *bitmap* **where**
bitmap-set mp b ≡ (*mp*[*b* := *True*])

definition *bitmap-clear* :: *bitmap* ⇒ *u64* ⇒ *bitmap* **where**
bitmap-clear mp b ≡ *mp* [(*b div BITMAP-UNIT-LEN*) := *False*]

definition *bitmap-get* :: (*bitmap*) ⇒ *u64* ⇒ *bool* **where**
bitmap-get mp b ≡ *mp* ! *b*

definition *bitmap-set-consecutive* :: *bitmap* ⇒ *u64* ⇒ *u64* ⇒ *bitmap* **where**
bitmap-set-consecutive mp b e ≡ (*take b mp*)@(*list.map* (*λf*. *True*) (*extract-list mp b e*))@(*drop e mp*)

definition *bitmap-clear-consecutive* :: *bitmap* ⇒ *u64* ⇒ *u64* ⇒ *bitmap* **where**
bitmap-clear-consecutive mp b e ≡
(*take b mp*)@(*list.map* (*λf*. *False*) (*extract-list mp b e*))@(*drop e mp*)

thm *bitmap-clear-consecutive-def*

definition *bitmap-clear'* :: *bitmap* ⇒ *u64* ⇒ (*bitmap*, *unit*)*nondet-monad* **where**
bitmap-clear' mp b ≡
modify (*λs*. *mp* [(*b div BITMAP-UNIT-LEN*) := *False*])

definition *bitmap-set'* :: *bitmap* ⇒ *u64* ⇒ (*bitmap*, *unit*)*nondet-monad* **where**
bitmap-set' mp b ≡ *modify* (*λs*. (*mp*[(*b div BITMAP-UNIT-LEN*) := *True*]))

lemma *bmp-set-proof* : ⦃*λs*. *s*≠*Nil*⦄*bitmap-set' mp b*⦃*λr s*. *s*=(*s*[(*b div BITMAP-UNIT-LEN*)

:= (

$(s\ !\ (b\ div\ BITMAP\text{-}UNIT\text{-}LEN)) \lor$

$True)])\}$

apply (*unfold bitmap-set′-def*)

oops

definition *bitmap--clear* :: *bitmap* ⇒ *u64* ⇒ (*bitmap*, *unit*)*nondet-monad* **where**

bitmap--clear mp b ≡ *do cur* ← *return* (*mp* [(*b div BITMAP-UNIT-LEN*) :=
False]);
return ()
od

definition *bitmap-set-consecutive′* :: *bitmap* ⇒ *u64* ⇒ *u64* ⇒ (*′s*, *unit*)*nondet-monad*
where

```
bitmap-set-consecutive′ mp st n′ ≡ do
    i ← return 0;
    i ← (whileLoop (λi s. nat i < n′)
    (λi. do
      return (bitmap-set mp (st + nat i));
      return (i+1)
    od)
    (i));
    return ()
od
```

definition *bitmap-clear-consecutive′* :: *bitmap* ⇒ *u64* ⇒ *u64* ⇒ (*bitmap*, *unit*)*nondet-monad*
where

```
bitmap-clear-consecutive′ mp st n′ ≡ do
    i ← return 0;
    i ← (whileLoop (λi s. nat i < n′)
    (λi. do
      bitmap--clear mp (st + nat i);
      return (i+1)
    od)
    i);
    return ()
od
```

lemma *bmp-clear-proof* : $\{\lambda s.\ s \neq Nil\}$ *bitmap-clear′ mp b* $\{\lambda r\ s.\ s=(s\ [(b\ div\ BITMAP\text{-}UNIT\text{-}LEN)$
:= (

$(s\ !\ (b\ div\ BITMAP\text{-}UNIT\text{-}LEN)) \land$

$\neg(True))])\}$

apply (*unfold bitmap-clear′-def*)

oops

lemma *bitmap-clear-proof* : $\{\!|\lambda s.\ True|\!\}$ *bitmap--clear mp b* $\{\!|\lambda r\ s.\ r{=}unit|\!\}$
 apply (*simp add*: *bitmap-clear-def return-def*)
 oops

definition *bitmap-count* :: *bitmap* ⇒ *u64* ⇒ *u64* ⇒ *bool* ⇒ (*bitmap*, *u64*)*nondet-monad*
where
 bitmap-count mp st n′ f ≡ *return* (*length* (*removeAll* (¬*f*) (*drop st* (*take* (*n′*+*1*) *mp*))))

definition *bitmap-count′* :: *bitmap* ⇒ *u64* ⇒ *u64* ⇒ *bool* ⇒ *u64* **where**
 bitmap-count′ mp st n′ f ≡ (*length* (*removeAll* (¬*f*) (*drop st* (*take* (*n′*+*1*) *mp*))))

term *bitmap-clear-consecutive* [*False*, *True*, *False*, *False*, *True*, *True*, *True*, *False*, *True*] *2 6*
value *bitmap-set-consecutive* [*False*, *True*, *False*, *False*, *True*, *True*, *True*, *False*, *True*] *0 6*

value *char-of* (*nat 5*)

value *length* (*enumerate 2* [*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*])

value *removeAll False* [*False*, *True*, *False*, *False*, *True*, *True*, *True*, *False*, *True*]

value *dropWhile* (λ*f*. *True*) [*False*, *True*, *False*, *False*, *True*, *True*, *True*, *False*, *True*]
value *list.map* (λ*f*. *False*) [*False*, *True*, *False*, *False*, *True*, *True*, *True*, *False*, *True*]

value *let A* = [*False*, *True*, *False*, *False*, *True*, *True*, *True*, *False*, *True*]
 in
 append (*append* (*take 2 A*) (*drop 2* (*take 6 A*))) (*drop 6 A*)

definition *bmp-mgmt* :: *bitmap* ⇒ *u64* ⇒ *u64* ⇒ *bitmap* **where**
 bmp-mgmt mp b e ≡ (*take b mp*)@(*extract-list mp b e*)@(*drop e mp*)

end
theory *Interrupts-Decls*

imports *type-Decls*

begin
type-synonym *irq-handler-t* = *u64* ⇒ *u64* ⇒ *unit*

type-synonym *Interrupts-State* = *Interrupts*

definition *write-Interrupts-State h g hdl* ≡ (|*Interrupt-hyper-bitmap*=*h*,
Interrupt-glb-bitmap=*g*,
Interrupt-handlers=*hdl*|)
definition *read-Interrupts-State-ihyper S* ≡ *Interrupt-hyper-bitmap S*
definition *read-Interrupts-State-iglb S* ≡ *Interrupt-glb-bitmap S*
definition *read-Interrupts-int-hdl S* ≡ *Interrupt-handlers S*
definition *uptdate-Interrupts-State-ihyper S h* ≡ *S*(|*Interrupt-hyper-bitmap*:=*h*|)
definition *uptdate-Interrupts-State-iglb S g* ≡ *S*(|*Interrupt-glb-bitmap*:=*g*|)
definition *uptdate-Interrupts-State-int-hdl S l* ≡ *S*(|*Interrupt-handlers*:=*l*|)

definition *updtate-Interrupts-spec-hdl S hdl i* ≡ *S*(|*Interrupt-handlers*:=(*Interrupt-handlers S*)[*i*:=*hdl*]|)
definition *update-Interrupts-spec-state S h g l* ≡ *S*(|*Interrupt-hyper-bitmap*=*h*,
Interrupt-glb-bitmap=*g*,
Interrupt-handlers=*l*|)
definition *update-Interrupts-state-iglb-hyp S i* ≡ *S*(|*Interrupt-hyper-bitmap*:=*i*,
Interrupt-glb-bitmap:=*i*|)
end
theory *VM-Decls*

imports *type-Decls Interrupts-Decls*

begin

record *VM-State0* = *vm-info* :: *VM list*
work-state :: *VM-WORK-STATE list*

end

theory *VCPU-Decls*

imports *type-Decls VM-Decls*

begin

record *VCPU-State* = *vcpu-info* :: *VCPU list*
work-state :: *VCPU-WORK-STATE list*

end
theory *CPU-Decls*

imports *type-Decls VCPU-Decls*

begin

record *CPU-State0* = *cpu-info* :: *CPU list*
work-state :: *CPU-WORK-STATE list*

end
theory *Audit-Decls*

imports *type-Decls*

begin

datatype *audit-event-type* = *AUDIT-EVENT-T-SYNC* |
AUDIT-EVENT-T-HVC |
AUDIT-EVENT-T-SMC |
AUDIT-EVENT-T-IRQ

datatype *audit-event-hvc-oper* =

AUDIT-EVENT-HVC-O-SYS-RESET |
AUDIT-EVENT-HVC-O-SYS-OFF |

AUDIT-EVENT-HVC-O-VMM-LIST-VM |
AUDIT-EVENT-HVC-O-VMM-GET-VM-STATE |
AUDIT-EVENT-HVC-O-VMM-SHUTDOWN-VM |
AUDIT-EVENT-HVC-O-VMM-REBOOT-VM |
AUDIT-EVENT-HVC-O-VMM-GET-VM-DEF-CFG |
AUDIT-EVENT-HVC-O-VMM-SET-VM-CFG |
AUDIT-EVENT-HVC-O-VMM-GET-VM-ID |

AUDIT-EVENT-HVC-O-IVC-UPDATE-MQ |
AUDIT-EVENT-HVC-O-IVC-SEND-MSG |
AUDIT-EVENT-HVC-O-IVC-BROADCAST-MSG |

AUDIT-EVENT-HVC-O-DEV-LIST-DEVS |
AUDIT-EVENT-HVC-O-DEV-STATE |

AUDIT-EVENT-HVC-O-SECURITY-SET-CFG |
AUDIT-EVENT-HVC-O-SECURITY-GET-DEF-CFG |
AUDIT-EVENT-HVC-O-SECURITY-GET-CFG |
AUDIT-EVENT-HVC-O-SECURITY-GET-LOG |

AUDIT-EVENT-HVC-O-UNKNOWN |
AUDIT-EVENT-HVC-O-INVALIED

datatype *audit-resource-type* =
AUDIT-RESOURCE-T-CPU |
AUDIT-RESOURCE-T-MEM |
AUDIT-RESOURCE-T-INTC |
AUDIT-RESOURCE-T-INTID |
AUDIT-RESOURCE-T-SERIAL |
AUDIT-RESOURCE-T-BLK |
AUDIT-RESOURCE-T-NET |
AUDIT-RESOURCE-T-SYS |
AUDIT-RESOURCE-T-VM |
AUDIT-RESOURCE-T-NONE

datatype *audit-result* = *AUDIT-RESULT-T-SUCCEED* | *AUDIT-RESULT-T-FAILED* | *AUDIT-RESULT-T-UNKNOWN*

record *Audit* = *event* :: *audit-event-type*
oper :: *audit-event-hvc-oper*
res-type :: *audit-resource-type*
res-id :: *u32*
result :: *audit-result*

type-synonym *audit* = *Audit list*

definition *write-Audit a b c d e*≡ (|*event*=*a*, *oper*=*b*, *res-type*=*c*, *res-id*=*d*, *result*=*e*|)
definition *audit-append-event* :: *Audit* ⇒ (*audit*,*unit*)*nondet-monad* **where**
audit-append-event a0 ≡
(*do*
al ← *gets* (λσ. σ);
modify (λσ. *a0*#*al*);
return ()
od)

record *Audit-State* = *a*::*nat*
end
theory *Security-Decls*

imports *type-Decls*

begin

record *vm-port-config* = *num*::*u64*
contact-vm::*u64 list*
type-bitmap::*u32*

record *vm-contact-entry* = *port*::*vm-port-config*
record *SEC-State* = *a*::*nat*

definition *write-vm-port-config n c t* ≡ (|*num*=*n*, *contact-vm*=*c*, *type-bitmap*=*t*|)
definition *read-vm-port-num vmpt-cfg* ≡ (*num vmpt-cfg*)
definition *read-vm-port-ctvm vmpt-cfg* ≡ (*contact-vm vmpt-cfg*)
definition *read-vm-port-tbmp vmpt-cfg* ≡ (*type-bitmap vmpt-cfg*)

definition *write-vm-contact-entry p* ≡ (|*port*=*p*|)
definition *read-vm-contact-entry-port cfg* ≡ (*vm-contact-entry.port cfg*)

end
theory *MEM-Decls*

imports *type-Decls ../util/List-Index ../common/commdata*

begin

record *mem-regionD* = *baseD* :: *u64*
lastD :: *page-last*

record *Mem-VM* = *vm-region-num-des* :: *region-num*
vm-region-des :: *mem-region list*
vm-region-detail :: *mem-regionD list*
map-des :: *bitmap-region*

record *Mem-Heap* = *sizeD* :: *region-size*
freeD :: *page-free*
last :: *page-last*
base :: *u64*
map-size :: *u64*
mapD :: *bitmap-heap*
free-countD :: *count-heap*

value (*7* ∗ *2*)::*int*

end
theory *HVC-Decls*

imports *type-Decls*

begin

datatype *hvc-fid* = *HVC-SYS nat 0* | *HVC-VMM* | *HVC-IVC* | *HVC-DEVICE* | *HVC-SECURITY*

datatype *hvc-ivc-event* = *HVC-IVC-UPDATE-MQ nat 0* |
HVC-IVC-SEND-MSG |
HVC-IVC-BROADCAST-MSG |
HVC-IVC-INIT-KEEP-ALIVE |
HVC-IVC-KEEP-ALIVE |
HVC-IVC-ACK |
HVC-IVC-GET-TIME

end
theory *IPI-Decls*

imports *type-Decls*

begin

datatype *ipi-type* = *IPI-T-INTC 0* | *IPI-T-POWER* | *IPI-T-ETHERNET-MSG*

| *IPI-T-ETHERNET-ACK* | *IPI-T-HVC*
record *ipi-intc-msg* = *event*::*u32*
vm-id::*u64*
int-id::*u16*
val::*u8*

record *ipi-power-msg* = *event*::*u64*
entry::*u64*
context'::*u64*

record *ipi-ethernet-msg* = *src*::*u64*
len::*u64*
frame::*u64*

record *ipi-ethernet-ack-msg* = *succeed*::*bool*
len::*u64*

record *ipi-hvc-msg* = *src*::*u16*
fid::*u16*
event::*u16*
data::*u8 list*

consts *IPI-HANDLER-MAX* :: *16*
consts *ipi-handler-num* :: *0*

end

theory *IVC-Decls*

imports *type-Decls VM-Decls ../common/commfunc*

begin

end
theory *GLOBAL-STATE*

imports *CPU-Decls VM-Decls Interrupts-Decls Audit-Decls Security-Decls MEM-Decls HVC-Decls IPI-Decls IVC-Decls VCPU-Decls MEM-Decls ../util/List-Index*

begin
record *GLB-STATE* = *cpu-state0* :: *CPU-State0*

vm-state0 :: *VM-State0*
vcpu-state :: *VCPU-State*
interrupts-state::*Interrupts-State*
ivc-state::*IVC-State*
mem-vm :: *Mem-VM*
mem-heap :: *Mem-Heap*
audit-state :: *Audit-State*
sec-state::*SEC-State*

definition *update-GLB-cpu-state glb cs* ≡ *glb*(|*cpu-state0*:=*cs*|)
definition *update-GLB-vm-state glb vs* ≡ *glb*(|*vm-state0*:=*vs*|)
definition *update-GLB-ivc-state glb ivs* ≡ *glb*(|*ivc-state*:=*ivs*|)
definition *update-GLB-int-state glb is* ≡ *glb*(|*interrupts-state*:=*is*|)

definition *read-GLB-vm-state glb* ≡ (*vm-state0 glb*)
definition *read-GLB-ivc-state glb* ≡ (*ivc-state glb*)
definition *read-GLB-int-state glb* ≡ (*interrupts-state glb*)

end
theory *IVC-Mgmt*

imports *GLOBAL-STATE*

begin

definition *get-vmid-cid-des* :: *cpu-id* ⇒ *GLB-STATE* ⇒ *vm-id*
  where *get-vmid-cid-des cidx g* ≡
    let
      cpu = (*cpu-info* (*cpu-state0 g*))!*cidx*;
      vidx = *active-vcpu cpu*;
      vcpu = (*vcpu-info* (*vcpu-state g*))!*vidx*
    in
      vmID vcpu

definition *channel-is-legal* :: *vm-id* ⇒ *vm-id* ⇒ *channel* ⇒ *bool*
  where *channel-is-legal curID tarID ch* ≡
      let
        src = *the* (*portSrc ch*);
        des = *the* (*portDes ch*)
      in
        idV src = *curID* ∧ *idV des* = *tarID*

definition *ivc-find-channel-des* :: *vm-id* ⇒ *vm-id* ⇒ *GLB-STATE* ⇒ *channel-id*

```
option
  where  ivc-find-channel-des cur-vmid targ-vmid g ≡
        let
           chs = channels (ivc-state g);
           limit = length chs;
           idx = find-index (λx. channel-is-legal cur-vmid targ-vmid x ) chs
        in
           if (idx < limit) then
            Some idx
           else
            None

definition ivc-send-msg-des ::  cpu-id ⇒ vm-id ⇒ MSG ⇒ (GLB-STATE,bool)
nondet-monad
  where ivc-send-msg-des cid vm-tgrt mesg ≡
  (do
    idTarget ← gets (λσ. get-vmid-cid-des cid σ);
    if( idTarget = vm-tgrt ) then
      return False
    else
      condition (λσ. ivc-find-channel-des idTarget vm-tgrt σ = None )
        (return False)
        (do
          channelID ← gets (λσ. the(ivc-find-channel-des idTarget vm-tgrt σ));
          channel0 ← gets (λσ. (channels (ivc-state σ)!channelID)⦇msg:=mesg⦈);
          channelsN ← gets (λσ. (channels(ivc-state σ))[channelID:=channel0]);
          ivcN ← gets (λσ. (ivc-state σ)⦇channels:=channelsN⦈);
          modify(λσ. σ⦇ivc-state:=ivcN⦈);
          return True
        od)
  od)


definition ivc-channel-init-des ::   (GLB-STATE,unit)nondet-monad
  where ivc-channel-init-des ≡
    (do
      ivc ← gets (λσ.(ivc-state σ));
      chans ← return (channels ivc);
         chansN ← return ( List.map (λx. x⦇flag:=False, portSrc:=None, port-
Des:=None ⦈) chans);
      ivcN ← return(ivc⦇channel-num:=0,channels:=chansN⦈);
      modify (λσ. σ⦇ivc-state:=ivcN⦈);
      return ()
    od)


definition channel-is-available ::  vm-port => vm-id ⇒ channel ⇒ bool
  where channel-is-available pt idx chan ≡
    if(flag chan) then
```

```
    if((vm-port.type pt = RECEIVE) ∧ ¬ (portSrc chan = None)) then
      idx = idV (the (portSrc chan))
    else
      if((vm-port.type pt = SEND) ∧ ¬ (portDes chan = None)) then
        idx = idV (the (portDes chan))
      else
        False

  else
    False
```

definition *get-channel-des* :: *vm-port* ⇒ *vm-id* ⇒ *GLB-STATE* ⇒ *channel-id option*

```
 where get-channel-des pt idx g ≡
    let
      channels = channels (ivc-state g);
      nums = channel-num (ivc-state g);
      channelsTemp = take nums channels;
      vid = find-index (λx. (channel-is-available pt idx x)) channelsTemp
    in
      if(vid=nums) then
        None
      else
        Some vid
```

definition *vm-init-channel-des* :: *vm-port* ⇒ *vm-id* ⇒ (*GLB-STATE*,*bool*)*nondet-monad*

```
 where vm-init-channel-des pt idx ≡
  (do
    channels ← gets (λσ. (channels (ivc-state σ)));
    nums ← gets (λσ. (channel-num (ivc-state σ)));
    ret ← gets (λσ. get-channel-des pt idx σ);
    condition (λσ. ¬ ret = None)
      (condition (λσ. vm-port.type pt = RECEIVE)
          (do
            idx ← return (the ret);
            channel0 ← return ( (channels!idx)(|portDes:= Some pt |) );
            channelsN ← return ( channels[idx:=channel0] );
            ivcN ← gets (λσ. (ivc-state σ)(|channels:=channelsN|));
            modify(λσ. σ(|ivc-state:=ivcN|));
            return True
          od)
          (do
            idx ← return (the ret);
            channel0 ← return ( (channels ! idx)(|portSrc:= Some pt|) );
            channelsN ← return (channels[idx:=channel0]);
```

```
              ivcN ← gets (λσ. (ivc-state σ)(|channels:=channelsN|));
              modify(λσ. σ(|ivc-state:=ivcN|));
              return True
            od))
        (condition (λσ. nums≥ MAX-CHANNEL-NUM)
          (return False)
          (do
            idx← return (nums);
            condition (λσ. vm-port.type pt = RECEIVE)
            (do
                 channel0 ← return ( (channels!idx)(|flag:=True,portDes:= Some pt
|) );
                channelsN ← return ( channels[idx:=channel0] );
            ivcN ← gets (λσ. (ivc-state σ)(|channels:=channelsN,channel-num:=nums+1|));

                modify(λσ. σ(|ivc-state:=ivcN|));
                return True
            od)
            (do
                channel0 ← return ( (channels!idx)(|flag:=True,portSrc:= Some pt |)
);
                channelsN ← return ( channels[idx:=channel0] );
            ivcN ← gets (λσ. (ivc-state σ)(|channels:=channelsN,channel-num:=nums+1|));

                modify(λσ. σ(|ivc-state:=ivcN|));
                return True
            od)
          od))
    od)
```

value *List.filter* (λx. ¬x=2) [1..5]

definition *init-port-in-channel-des* :: *vm-id* ⇒ (*GLB-STATE*,*nat*)*nondet-monad*

```
  where init-port-in-channel-des idx ≡
    do
      channels ← gets (λσ. channels (ivc-state σ));
      limit ← return (length channels);
      ret ← (whileLoopE (λi t. i<limit)
              (λ i.
                 condition (λt. flag (channels ! i)) (throwError i)
                 (returnOk (i+1))
              )
```

```
            (0)
           )<catch> (λe. return e) ;
    return ret
  od
```

```
definition init-port-in-channel-des2 ::  vm-id ⇒ (GLB-STATE,nat)nondet-monad

  where init-port-in-channel-des2 idx ≡
  do
    channels ← gets (λσ. channels (ivc-state σ));
    limit ← return (length channels);
    ret ← (whileLoopE (λi t. i<limit)
            (λ i.
               condition (λt. flag (channels ! i)) (throwError i)
               (returnOk (i+1))
            )
            (0)
           )<catch> (λe. return e) ;
    return ret
  od
```

```
end
theory IVC
  imports ../Req/hvc ../Design/IVC-Mgmt
begin
```

```
definition channels-corrs ::  GLB-STATE ⇒ HV ⇒ bool
  where channels-corrs g h ≡
      let
        channels1 = channels (ivc-state g);
        nums1 = channel-num (ivc-state g);
        channels2 = channels (ivc (commu h));
        nums2 = channel-num (ivc (commu h))
      in
        channels1 = channels2 ∧ nums1 = nums2
```

```
definition getChannel-corrs :: GLB-STATE ⇒ HV ⇒ cpu-id ⇒ vm-id ⇒ bool
  where getChannel-corrs g h cid id2 ≡
  let
    id11 = get-vmid-cid-des cid g;
    channel1 = ivc-find-channel-des id11 id2 g;
    id12 = get-vmid-cid-req cid h;
```

channel2 = (*getChannel* (*commu h*)) *id12 id2*
in
channel1 = *channel2*

definition *channelAvail-corrs* :: *GLB-STATE* ⇒ *HV* ⇒ *vm-port* ⇒ *vm-id* ⇒ *bool*
where *channelAvail-corrs g h vmp id0* ≡
let
ret1 = *get-channel-des vmp id0 g*;
ret2 = (*availChannel* (*commu h*)) *vmp id0*
in
ret1 = *ret2*

definition *cpuInfo-corrs* :: *GLB-STATE* => *HV* => *bool*
where *cpuInfo-corrs g h* ≡
let
cpusD = *cpu-info* (*cpu-state0 g*);
cpusR = *cpu h*
in
cpusD = *cpusR*

definition *vcpuInfo-corrs* :: *GLB-STATE* ⇒ *HV* ⇒ *bool*
where *vcpuInfo-corrs g h* ≡
let
vcpus1 = (*vcpu-info* (*vcpu-state g*));
vcpus2 = (*vcpu h*)
in
vcpus1 = *vcpus2*

definition *vmInfo-corrs* :: *GLB-STATE* ⇒ *HV* ⇒ *bool*
where *vmInfo-corrs g h* ≡
let
vms1 = (*vm-info* (*vm-state0 g*));
vms2 = (*HV.vm h*)
in
vms1 = *vms2*

definition *sysInfo-corrs* :: *GLB-STATE* ⇒ *HV* ⇒ *bool*
where *sysInfo-corrs g h* ≡
cpuInfo-corrs g h ∧ *vmInfo-corrs g h* ∧ *vcpuInfo-corrs g h*

lemma *aux1* : *cpuInfo-corrs g h* $\wedge$ *vmInfo-corrs g h* $\wedge$ *vcpuInfo-corrs g h* $\Longrightarrow$*get-vmid-cid-des cid g* = *get-vmid-cid-req cid h*
unfolding *cpuInfo-corrs-def Let-def*
unfolding *vmInfo-corrs-def Let-def*
unfolding *vcpuInfo-corrs-def Let-def*
unfolding *get-vmid-cid-des-def get-vmid-cid-req-def Let-def*
apply *auto*
done

lemma *ivc-send-msg* :
$\{\!|\lambda s.$ *sysInfo-corrs s hv* $\wedge$
channels-corrs s hv $\wedge$
getChannel-corrs s hv cid vmid2 $|\!\}$
ivc-send-msg-des cid vmid2 m0
$\{\!|\lambda r\ s.$ *channels-corrs s* (*fst*(*ivc-send-msg-req hv cid vmid2 m0*))
$\wedge$ *r* = *snd*(*ivc-send-msg-req hv cid vmid2 m0*) $|\!\}$

unfolding *channels-corrs-def Let-def*
unfolding *getChannel-corrs-def Let-def*
apply (*simp add*:*ivc-send-msg-des-def*)
apply *wpsimp*
apply *auto*
unfolding *ivc-send-msg-req-def Let-def*
apply *auto*
unfolding *insert-channel-msg-req-def Let-def*
apply *auto*
apply (*smt snd-conv*)
unfolding *sysInfo-corrs-def*
using *aux1*
apply *simp*
apply (*simp add*: *aux1*)+
done

lemma *vm-init-channel* :
$\{\!|\lambda s.$ *channels-corrs s h* $\wedge$
channelAvail-corrs s h vmp vmid$|\!\}$
vm-init-channel-des vmp vmid
$\{\!|\lambda r\ s.$ *channels-corrs s* (*fst*(*vm-init-channel-req h vmp vmid*)) $\wedge$
r = *snd* (*vm-init-channel-req h vmp vmid*)$|\!\}$
unfolding *channels-corrs-def Let-def*
unfolding *channelAvail-corrs-def Let-def*
apply (*simp add*:*vm-init-channel-des-def*)
apply *wpsimp*
apply *auto*
apply (*simp add*:*vm-init-channel-req-def Let-def*)+
done

end
theory *MEM-Mgmt*

imports *GLOBAL-STATE*

begin

definition *mem-vm-region-alloc-des* :: *u64* ⇒ (*GLB-STATE*,*u64 option*) *nondet-monad*
  where *mem-vm-region-alloc-des pageNum* ≡
  (*do*
      mem0 ← *gets* (λσ. *mem-vm* σ);
      pageIDX ← *return* ((*Mem-VM.map-des mem0*) *pageNum*);
      if (¬ *pageIDX* = *None*) *then*
        (*do*
          idx ← *return* (*the pageIDX*);
          free0 ← *return* (*freeB* ((*vm-region-des mem0*)!*idx*) − *pageNum*);
          base0 ← *return* (*baseD* ((*vm-region-detail mem0*)!*idx*));
          last0 ← *return* (*lastD* ((*vm-region-detail mem0*)!*idx*) + *pageNum*);
          region0 ← *return*
                (((*vm-region-des mem0*)!*idx*)(|*mem-region.freeB*:=*free0*|));
          region1 ← *return*
                (((*vm-region-detail mem0*)!*idx*)(|*mem-regionD.lastD*:=*last0*|));
          regionN0 ← *return* ((*vm-region-des mem0*)[*idx*:=*region0*]);
          regionN1 ← *return* ((*vm-region-detail mem0*)[*idx*:=*region1*]);
        mem-vmN ← *return* (*mem0*(|*vm-region-des* :=*regionN0*,*vm-region-detail*:=*regionN1*|));
          modify (λσ. σ(| *mem-vm*:= *mem-vmN* |));
          return (*Some* (*base0*+*last0*∗*PAGE-SIZE*))
        od)
      else
        return None
  od)

definition *range-in-range* :: *u64* ⇒ *u64* ⇒ *u64* ⇒ *u64* ⇒ *bool* **where**
  range-in-range base1 size1 base2 size2 ≡
      if base1≥*base2* ∧ (*base1*+*size1*)≤(*base2*+*size2*) *then*
         True
      else
         False

definition *range-in-range-heap* :: *u64* ⇒ *u64* ⇒ *Mem-Heap* ⇒ *bool* **where**
  range-in-range-heap base1 size1 x1 ≡
     let
      base2 = *Mem-Heap.base x1*;
      size2 = (*Mem-Heap.sizeD x1*)∗*PAGE-SIZE*

```
  in
   if base1≥base2 ∧ (base1+size1)≤(base2+size2) then
     True
   else
     False
```

definition *mem-vm-region-clear-des* :: *u64* ⇒ *page-num* ⇒ (*GLB-STATE*,*bool*) *nondet-monad*

```
 where mem-vm-region-clear-des start0 pageNum ≡
 (do
    limit ← gets (λσ. vm-region-num-des (mem-vm σ));
     idx ← gets (λσ. (find-index (λx. range-in-range start0 pageNum (sizeB x) (freeB x))
            (vm-region-des (mem-vm σ))));
    if(idx<limit) then
    (do
      free0 ← gets (λσ. freeB((vm-region-des (mem-vm σ))!idx));
      region0 ← gets
          (λσ. ((vm-region-des (mem-vm σ))!idx)(| freeB:=free0+pageNum|) );
      regionN ← gets (λσ. (vm-region-des (mem-vm σ))[idx:=region0]);
      mem-vmN ← gets (λσ. (mem-vm σ)(|vm-region-des:=regionN|));
      modify (λσ. σ(| mem-vm:= mem-vmN |));
      return True
     od)
     else
    (do
      modify (λσ. σ);
      return False
     od)
 od)
```

definition *mem-heap-alloc-des* :: *nat* ⇒ (*GLB-STATE*, *u64 option*)*nondet-monad*

```
 where  mem-heap-alloc-des pageNum ≡
   condition (λσ. pageNum> (freeD (mem-heap σ)) ∨ pageNum = 0 )
     (do
       modify(λσ. σ);
       return None
     od)
     (do
       idx ← gets (λσ. (mapD (mem-heap σ)) pageNum);
       case idx of
         None ⇒ return None |
         Some page-index ⇒
           (do
```

```
          base0 ← gets (λσ. Mem-Heap.base (mem-heap σ));
          free0 ← gets (λσ. Mem-Heap.freeD (mem-heap σ) − pageNum);
          size0 ← gets (λσ. Mem-Heap.sizeD (mem-heap σ));
          if (page-index + pageNum) < size0 then
          (do
              mem-heapN ← gets (λσ. (mem-heap σ)(|Mem-Heap.freeD:=free0,
Mem-Heap.last:=page-index + pageNum|) );
            modify (λσ. σ(|mem-heap:=mem-heapN|));
            return (Some (base0+page-index∗PAGE-SIZE))
          od)
          else
          (do
              mem-heapN ← gets(λσ. (mem-heap σ)(|Mem-Heap.freeD:=free0,
Mem-Heap.last:= map-size (mem-heap σ) |) );
            modify (λσ. σ(|mem-heap:=mem-heapN|));
            return (Some (base0+page-index∗PAGE-SIZE))
          od)
      od)
    od)
```

definition *mem-pages-free-des* :: *u64* ⇒ *nat* ⇒ (*GLB-STATE*, *bool*)*nondet-monad*

```
  where mem-pages-free-des addr pageScale ≡
    condition (λσ. range-in-range-heap addr (pageScale∗PAGE-SIZE) (mem-heap
σ) )
    (do
      base0 ← gets (λσ. ( base (mem-heap σ)) div PAGE-SIZE);
      last0 ← gets (λσ. (addr − base (mem-heap σ)) div PAGE-SIZE);
      free0 ← gets (λσ.
       freeD (mem-heap σ) + (free-countD (mem-heap σ) ((addr − base (mem-heap
σ)) div PAGE-SIZE) pageScale));
      mem-heapN ← gets (λσ. (mem-heap σ)(|last:=last0,freeD:=free0|));
      modify(λσ. σ(|mem-heap:=mem-heapN|));
      return True
    od)
    (do
      modify(λσ. σ);
      return False
    od)
```

end
theory *MEM*
  imports ../*Design*/*MEM-Mgmt* ../*Req*/*hvc*

begin

definition *test-mem-vm-region* :: *GLB-STATE* ⇒ *HV* ⇒ *nat* ⇒ *bool*
  where *test-mem-vm-region g h n* ≡
  let
    md = *GLB-STATE.mem-vm g*;
    mr = *HV.mem-vm h*
  in
   if vm-region-des md = *vm-region mr* ∧ *map-des md* = *map mr then*
      True
    else
      False

definition *test-mem-vm-region-result* :: *u64 option* ⇒ *bool* ⇒ *bool*
  where *test-mem-vm-region-result addr b* ≡
    case (*addr*, *b*) *of*
        (*Some i*, *True*) ⇒ *True* |
        (*None*, *False*) ⇒ *True* |
        - ⇒ *False*

lemma *mem-vm-alloc* :
   ⋀*num0*. *num0*>*0* ⟹⦃λ*s*. *test-mem-vm-region s hv num0*⦄
      mem-vm-region-alloc-des num0
   ⦃λ*r s*. *test-mem-vm-region s* (*fst* (*vm-region-alloc-req hv num0*)) *num0*
      ∧ *test-mem-vm-region-result r* (*snd* (*vm-region-alloc-req hv num0*)) ⦄
  apply (*simp add*: *test-mem-vm-region-def*)
  unfolding *Let-def*
  apply (*simp add*: *mem-vm-region-alloc-des-def*)
  apply *wpsimp*
  apply *auto*
  unfolding *vm-region-alloc-req-def*
  unfolding *Let-def*
     apply *auto*
     defer
    apply (*simp add*: *PAGE-SIZE-def*)
  apply (*simp add*: *test-mem-vm-region-result-def*)
  unfolding *test-mem-vm-region-result-def*
   apply *auto*
  by (*simp add*: *get-memRegion-def*)

definition *test-mem-heap-alloc* :: *GLB-STATE* ⇒ *HV* ⇒ *bool*
  where *test-mem-heap-alloc g h* ≡
  let
    md = *GLB-STATE.mem-heap g*;
    mr = *HV.mem-heap h*
  in

if mapD md = *mapR mr* ∧ *freeD md* = *Mem-Heap.freeR mr* ∧ *sizeD md* = *size mr* ∧ *map-size md* < *sizeD md then*
True
else
False

definition *test-mem-heap-alloc-result* :: *u64 option* ⇒ *GLB-STATE* ⇒ *bool*
where *test-mem-heap-alloc-result addr g* ≡
case addr of None ⇒ *True* |
Some u64 ⇒ *let*
last0 = *last* (*GLB-STATE.mem-heap g*);
size0 = *sizeD* (*GLB-STATE.mem-heap g*)
in
last0 < *size0*

lemma *mem-heap-alloc* :
⦃*λs. test-mem-heap-alloc s hv*⦄
mem-heap-alloc-des num0
⦃*λr s. test-mem-heap-alloc s* (*fst*(*heap-alloc-req hv num0*)) ∧ *test-mem-heap-alloc-result r s* ⦄
apply(*simp add*: *test-mem-heap-alloc-def*)
apply(*simp add*: *Let-def*)
apply(*simp add*: *mem-heap-alloc-des-def*)
apply *wpsimp*
apply(*simp add*: *test-mem-heap-alloc-result-def*)
apply(*simp add*: *heap-alloc-req-def*)
unfolding *PAGE-SIZE-def*
try
by *auto*

definition *test-mem-heap-free-arg* :: *GLB-STATE* ⇒ *u64* ⇒ *nat* ⇒ *page-index* ⇒ *page-num* ⇒ *HV* ⇒ *bool*
where *test-mem-heap-free-arg g a0 s0 idx num0 h* ≡
let
idx2 = (*a0* − *base* (*GLB-STATE.mem-heap g*)) *div PAGE-SIZE*
in
idx = *idx2* ∧ *s0* = *num0* ∧
(*idx* + *num0* < *Mem-Heap.size* (*HV.mem-heap h*)) = *range-in-range-heap a0 s0* (*GLB-STATE.mem-heap g*)

definition *test-mem-heap-free* :: *GLB-STATE* ⇒ *HV* ⇒ *bool*
where *test-mem-heap-free g h* ≡
let

md = *GLB-STATE.mem-heap g*;
mr = *HV.mem-heap h*
in
if free-countD md = *free-countR mr* ∧ *freeD md* = *Mem-Heap.freeR mr* ∧ *sizeD md* = *size mr then*
True
else
False

lemma *mem-heap-free* :
{|λ*s. test-mem-heap-free s hv* ∧ *test-mem-heap-free-arg s a0 s0 idx num0 hv*|}
mem-pages-free-des a0 s0
{|λ*r s. test-mem-heap-free s* (*fst* (*heap-free-req hv idx num0*)) |}

apply (*simp add*: *test-mem-heap-free-def*)
unfolding *Let-def*
apply (*simp add*: *mem-pages-free-des-def*)
apply (*simp add*: *range-in-range-heap-def*)
apply *wpsimp*
unfolding *Let-def*
apply (*simp add*: *test-mem-heap-free-arg-def*)
apply (*simp add*: *range-in-range-heap-def*)
unfolding *Let-def*
unfolding *PAGE-SIZE-def*
apply *auto*
apply (*simp add*: *heap-free-req-def*) +
done

end
theory *CPU-Mgmt*

imports *GLOBAL-STATE*

begin

This script will describe the CPU event

```
definition cpu-idle-des :: cpu-id ⇒(GLB-STATE,unit) nondet-monad where
  cpu-idle-des idx ≡
    (do
      cpuS ← gets (λσ. work-state (cpu-state0 σ) );
      cpuSN ← return (cpuS[idx:=CPU-S-IDLE]);
      cpuN ← gets (λσ. (cpu-state0 σ)(|work-state:=cpuSN|));
      modify (λσ. σ(|cpu-state0:=cpuN|));
      return ()
    od)
```

end
theory *VCPU-Mgmt*

imports *GLOBAL-STATE CPU-Mgmt*

begin

```
definition get-vmid-cid-des :: cpu-id ⇒ GLB-STATE ⇒ vm-id
  where get-vmid-cid-des cidx g ≡
    let
      cpu = (cpu-info (cpu-state0 g))!cidx;
      vidx = active-vcpu cpu;
      vcpu = (vcpu-info (vcpu-state g))!vidx
    in
      vmID vcpu
```

```
definition vcpu-run-des :: cpu-id ⇒ (GLB-STATE ,unit)nondet-monad
  where vcpu-run-des idx ≡
    (do
      cpuInfo ← gets (λσ. (cpu-info (cpu-state0 σ))!idx);
      vm-id ← gets (λσ. (get-vmid-cid-des idx σ));
      if( running-num cpuInfo > 1 ) then
        (do
          vmWorkStas ← gets(λσ. VM-State0.work-state(vm-state0 σ));
          vmWorkStasN ← return (vmWorkStas[vm-id:= VM-S-ACT] );
          vm ← gets (λσ. (vm-state0 σ)(| VM-State0.work-state:=vmWorkStasN|));
          cpuWorkStas ← gets(λσ. CPU-State0.work-state(cpu-state0 σ));
          cpuWorkStasN ← return (cpuWorkStas[idx:= CPU-S-RUN] );
         cpu ← gets (λσ. (cpu-state0 σ)(| CPU-State0.work-state:=cpuWorkStasN|));
          modify(λσ. σ(|vm-state0:=vm,cpu-state0:=cpu|));
          return()
        od)
```

else
return()
od)

definition *vcpu-pool-pop-pending-des0* :: *cpu-id* ⇒ (*GLB-STATE*, *vcpu-id option*)*nondet-monad*
where *vcpu-pool-pop-pending-des0 idx* ≡
(*do*
cpuInfo ← *gets*(λσ. (*cpu-info* (*cpu-state0* σ))!*idx*);
if(*length* (*CPU.vcpus cpuInfo*)>*0*) *then*
return None
else
(*do*
vcpus ← *gets* (λσ. *VCPU-State.work-state* (*vcpu-state* σ));
limit ← *return* (*length vcpus*);
idx ← *return* (*find-index* (λ*x*. *x* = *VCPU-S-PEND*) (*vcpus*));
if (*idx*<*limit*) *then*
return (*Some idx*)
else
return None
od)
od)

definition *vcpu-pool-pop-pending-des* :: *cpu-id* ⇒ *GLB-STATE* ⇒ *vcpu-id option*

where *vcpu-pool-pop-pending-des idx g* ≡
let
cpuInfo =(*cpu-info* (*cpu-state0 g*))!*idx*;
vcids = *CPU.vcpus cpuInfo*;
vcpus = *VCPU-State.work-state* (*vcpu-state g*);
vcid = *find-index* (λ*x*. *vcpus*!*x* = *VCPU-S-PEND*) *vcids*
in
if(*vcid*<*length vcids*) *then*
Some (*vcid*)
else
None

definition *get-activeNew-des* :: *vcpu-id* ⇒ *cpu-id* ⇒ *GLB-STATE* ⇒ *vcpu-id option*
where *get-activeNew-des target cid g* ≡
let

```
      cpu0 = (cpu-info (cpu-state0 g))!cid;
      vcids = CPU.vcpus cpu0;
      limit = length (vcpu-info (vcpu-state g))
    in
      if(target = limit ) then
        vcpu-pool-pop-pending-des cid g
      else
        if(target ∈ set vcids) then
          Some (target)
        else
          None
```

```
definition vcpu-pool-switch-des :: cpu-id ⇒ vcpu-id ⇒ (GLB-STATE,unit)nondet-monad
  where vcpu-pool-switch-des idx target≡
 (do
    cpu0 ← gets (λσ. (cpu-info (cpu-state0 σ))!idx );
    if(target = active-vcpu cpu0 ∨ running-num cpu0 = 0 ) then
      return ()
    else
      (do
         temp ←  gets (λσ. get-activeNew-des target idx σ);
         if(temp = None) then
            return ()
         else
         (do
            targetN ← return (the temp);
            if( targetN=active-vcpu cpu0) then
            (do
              workState ← gets(λσ. (VCPU-State.work-state (vcpu-state σ))
              [active-vcpu cpu0:=VCPU-WORK-STATE.VCPU-S-PEND,targetN:=VCPU-WORK-STATE.VC
                     vcpuStateN ← gets(λσ. (vcpu-state σ)(|VCPU-State.work-state
:=workState|));
              cpu1 ← return (cpu0(|active-vcpu:=targetN|));
              cpus ← gets (λσ. (cpu-info (cpu-state0 σ))[idx:=cpu1]);
              cpuN ← gets (λσ. (cpu-state0 σ)(|cpu-info:=cpus|));
              modify (λσ .σ(|cpu-state0:= cpuN, vcpu-state:=vcpuStateN|));
              return ()
            od)
            else
            (return ())
          od)
      od)
  od)
```

value $1 \in set([1..5])$

definition *get-targetVCPU-des* :: *vcpu-id list* ⇒ *vm-id* ⇒ *GLB-STATE* ⇒ *vcpu-id option*

```
  where get-targetVCPU-des vl idx g ≡
    let
      vcpuInfos = vcpu-info (vcpu-state g);
      limit = length vl;
      idx = find-index (λx. vmID (vcpuInfos ! x)  = idx ) vl
    in
      if (idx<limit) then
        Some idx
      else
        None
```

definition *vcpu-pool-pop-through-vmid-des* :: *cpu-id* ⇒ *vm-id* => (*GLB-STATE*,*vcpu-id option*)*nondet-monad*

```
  where vcpu-pool-pop-through-vmid-des cid vmid ≡
  (do
    cpu0 ← gets (λσ. (cpu-info (cpu-state0 σ))!cid );
    if(length (CPU.vcpus cpu0) = 0) then
      return None
    else
      (do
        vids ← return (CPU.vcpus cpu0);
        ret ← gets (λσ. get-targetVCPU-des vids vmid σ);
        return ret
      od)
  od)
```

definition *is-vcpu-inPool* :: *cpu-id* => *vcpu-id* ⇒ *GLB-STATE* ⇒ *nat option*

```
  where is-vcpu-inPool cid vcid g ≡
    let
      cpu0 = (cpu-info (cpu-state0 g)!cid);
      vcpuPool = CPU.vcpus cpu0;
      limit = length vcpuPool;
      idx = find-index (λx. x=vcid) vcpuPool
    in
      if idx < limit then
        Some idx
      else
        None
```

definition *vcpu-pool-suspend-des* :: *cpu-id* => *vcpu-id* ⇒ (*GLB-STATE*,*bool*)*nondet-monad*

```
  where vcpu-pool-suspend-des cid vcid ≡
    condition (λσ. CPU.vcpus ((cpu-info (cpu-state0 σ))!cid) = Nil )
      (return False)
      (do
        ans ← gets (λσ. is-vcpu-inPool cid vcid σ);
        if ¬ans = None then
        condition (λσ.(VCPU-State.work-state (vcpu-state σ))!vcid = VCPU-S-INV
)
          (return True)
          (do
        vwsN ← gets (λσ. (VCPU-State.work-state (vcpu-state σ))[vcid:=VCPU-S-INV]);
            vsN ← gets (λσ. (vcpu-state σ)(|VCPU-State.work-state:=vwsN|));
            cpu0 ← gets (λσ. (cpu-info (cpu-state0 σ))!cid);
            cpu0 ← return(cpu0(|running-num:=running-num cpu0 −1|));
            cpus ← gets (λσ. (cpu-info (cpu-state0 σ))[cid:=cpu0]);
            csN ←  gets (λσ. (cpu-state0 σ)(|cpu-info:=cpus|));
            modify(λσ. σ(|vcpu-state:=vsN,cpu-state0:=csN|));
            return True
          od)
        else
          return False
      od)


definition vcpu-pool-wakeup-des :: cpu-id => vcpu-id ⇒ (GLB-STATE,bool)nondet-monad
  where vcpu-pool-wakeup-des cid vcid ≡
    condition (λσ. CPU.vcpus ((cpu-info (cpu-state0 σ))!cid) = Nil )
      (return False)
      (do
        ans ← gets (λσ. is-vcpu-inPool cid vcid σ);
        if ¬ans=None then
       condition (λσ. ¬(VCPU-State.work-state (vcpu-state σ))!vcid = VCPU-S-INV
)
          (return True)
          (do
        vwsN ← gets (λσ. (VCPU-State.work-state (vcpu-state σ))[vcid:=VCPU-S-PEND]);
            vsN ← gets (λσ. (vcpu-state σ)(|VCPU-State.work-state:=vwsN|));
            cpu0 ← gets (λσ. (cpu-info (cpu-state0 σ))!cid);
            cpu0 ← return(cpu0(|running-num:=running-num cpu0 + 1|));
            cpus ← gets (λσ. (cpu-info (cpu-state0 σ))[cid:=cpu0]);
            csN ←  gets (λσ. (cpu-state0 σ)(|cpu-info:=cpus|));
            modify(λσ. σ(|vcpu-state:=vsN,cpu-state0:=csN|));
            return True
          od)
        else
          return False
      od)
```

```
definition vcpu-pool-remove-des :: cpu-id => vcpu-id ⇒ (GLB-STATE,bool)nondet-monad
  where vcpu-pool-remove-des cid vcid ≡
    condition (λσ. CPU.vcpus ((cpu-info (cpu-state0 σ))!cid) = Nil )
      (return False)
      (do
        ans ← gets (λσ. is-vcpu-inPool cid vcid σ);
        if ¬ ans = None then
        (do
          target ←return (the ans);
          limit ← gets (λσ. length(vcpu-info (vcpu-state σ)));
          num0 ← gets (λσ. running-num ((cpu-info (cpu-state0 σ))!cid));
          numN ← condition (λσ. ¬(VCPU-State.work-state (vcpu-state σ))!vcid =
VCPU-S-INV )
                    ( return (num0−1))
                    ( return  num0);
       vwsN ← gets (λσ. (VCPU-State.work-state (vcpu-state σ))[vcid:=VCPU-S-INV]);
          vsN ← gets (λσ. (vcpu-state σ)(|VCPU-State.work-state:=vwsN|));
       vcpusN ← gets (λσ. (CPU.vcpus((cpu-info (cpu-state0 σ))!cid))[target:=limit]
);
          cpu0 ← gets (λσ. (cpu-info (cpu-state0 σ))!cid);
          cpu0 ← return( cpu0(|running-num:=numN,CPU.vcpus:=vcpusN|));
          cpus ← gets (λσ. (cpu-info (cpu-state0 σ))[cid:=cpu0]);
          csN ←  gets (λσ. (cpu-state0 σ)(|cpu-info:=cpus|));
          modify(λσ. σ(|vcpu-state:=vsN,cpu-state0:=csN|));
          return True
        od)
        else
          return False
      od)
```

```
definition vcpu-shutdown-des :: cpu-id ⇒ vcpu-id ⇒ (GLB-STATE,unit)nondet-monad
  where vcpu-shutdown-des cid vcid ≡
    (do
      ans ← gets (λσ. is-vcpu-inPool cid vcid σ);
      if(ans = None) then
        cpu-idle-des cid
      else
        (do
          limit ← gets (λσ. length (vcpu-info (vcpu-state σ)));
          aN ← gets (λσ. get-activeNew-des limit cid σ );
       vwsN ← gets (λσ. (VCPU-State.work-state (vcpu-state σ))[vcid:=VCPU-S-INV]);
          vsN ← gets (λσ. (vcpu-state σ)(|VCPU-State.work-state:=vwsN|));
          cpu0 ← gets (λσ. (cpu-info (cpu-state0 σ))!cid);
```

```
        cpu0 ← return(cpu0(|running-num:=running-num cpu0 −1|));
        cpus ← gets (λσ. (cpu-info (cpu-state0 σ))[cid:=cpu0]);
        csN ←  gets (λσ. (cpu-state0 σ)(|cpu-info:=cpus|));
        if(running-num cpu0 = 0) then
        (do
          modify(λσ. σ(|cpu-state0:=csN, vcpu-state:=vsN |));
          cpu-idle-des cid
        od)
         else
          if vcid = active-vcpu cpu0 then
            if(aN = None ∨ the aN = active-vcpu cpu0 ) then
            (do
              modify(λσ. σ(|cpu-state0:=csN, vcpu-state:=vsN |));
              return ()
            od)
            else
              (do
                vwsN ← return ( vwsN[the aN:= VCPU-S-ACT]);
                vsN ← gets (λσ. (vcpu-state σ)(|VCPU-State.work-state:=vwsN|));
                cpu0 ← return( cpu0(|active-vcpu:= the aN|) );
                cpus ← gets (λσ. (cpu-info (cpu-state0 σ))[cid:=cpu0]);
                csN ←  gets (λσ. (cpu-state0 σ)(|cpu-info:=cpus|));
                modify(λσ. σ(|cpu-state0:=csN, vcpu-state:=vsN |));
                return ()
              od)
          else
            modify(λσ. σ(|cpu-state0:=csN, vcpu-state:=vsN |));
            return ()
      od)
    od)
```

end
theory *Schedule-Mgmt*

imports *CPU-Decls VCPU-Decls VCPU-Mgmt*

begin

```
definition cpu-schedule-des :: cpu-id ⇒ (GLB-STATE,unit)nondet-monad where
  cpu-schedule-des cid ≡
    condition (λσ. running-num((cpu-info (cpu-state0 σ)) ! cid) > 1)
      (do
        cpu0 ← gets (λσ. (cpu-info (cpu-state0 σ))!cid );
        limit ← gets (λσ. length (vcpu-info (vcpu-state σ)));
        temp ←  gets (λσ. get-activeNew-des limit cid σ);
```

```
          workState ← gets(λσ. (VCPU-State.work-state (vcpu-state σ))
                    [active-vcpu cpu0:=VCPU-WORK-STATE.VCPU-S-PEND,the
temp:=VCPU-WORK-STATE.VCPU-S-ACT]);
        vcpuStateN ← gets(λσ. (vcpu-state σ)(|VCPU-State.work-state :=workState|));
          cpu1 ← return (cpu0(|active-vcpu:=the temp|));
          cpus ← gets (λσ. (cpu-info (cpu-state0 σ))[cid:=cpu1]);
          cpuN ← gets (λσ. (cpu-state0 σ)(|cpu-info:=cpus|));
          modify (λσ. σ(|cpu-state0:= cpuN, vcpu-state:=vcpuStateN|));
          return ()
      od)
      (return ())



end
theory Schedule
  imports ../Req/hvc ../Design/Schedule-Mgmt

begin

lemma cpu-schedule :
    cid < length (cpu hv) ∧ idx = length (vcpu hv) ⟹ {|λs. vcpuPool-corrs cid s
hv ∧ vcpuState-corrs s hv
      ∧ length (cpu hv) = length (cpu-info (cpu-state0 s))
      ∧ vcpuLimit-corrs s hv
      ∧ cpuBasicInfo-corrs cid s hv
      ∧ cpuState-corrs s hv
      ∧ activeNew-corrs cid idx s hv ∧ vcpuIsContain-corrs cid vcid s hv |}
      cpu-schedule-des cid
   {|λr s. True |}

  by (simp add: hoare-post-taut)

end
theory VCPU
  imports ../Design/VCPU-Mgmt ../Req/hvc ../Design/Schedule-Mgmt
begin

definition cpuInfo-corrs :: GLB-STATE ⇒ HV ⇒ bool
  where cpuInfo-corrs g h ≡
    let
      cpus1 = (cpu-info (cpu-state0 g));
      cpus2 = (cpu h)
    in
      cpus1 = cpus2



definition vmID-corrs :: cpu-id ⇒ GLB-STATE ⇒ HV ⇒ bool
```

where *vmID-corrs idx g h* ≡
get-vmid-cid-des idx g = *get-vmid-cid-req idx h*

definition *vmState-corrs* :: *GLB-STATE* ⇒ *HV* ⇒ *bool*
where *vmState-corrs g h* ≡
let
vmsD = *VM-State0.work-state* (*vm-state0 g*);
vmsR = *vm-wk-st h*
in
vmsD = *vmsR*

definition *vcpuState-corrs* :: *GLB-STATE* ⇒ *HV* ⇒ *bool*
where *vcpuState-corrs g h* ≡
let
vcpusD = *VCPU-State.work-state* (*vcpu-state g*);
vcpusR = *vcpu-wk-st h*
in
vcpusD = *vcpusR*

definition *cpuState-corrs* :: *GLB-STATE* ⇒ *HV* ⇒ *bool*
where *cpuState-corrs g h* ≡
let
cpusD = *CPU-State0.work-state* (*cpu-state0 g*);
cpusR = *cpu-wk-st h*
in
cpusD = *cpusR*

definition *runningNum-corrs* :: *cpu-id* ⇒ *GLB-STATE* ⇒ *HV* ⇒ *bool*
where *runningNum-corrs idx g h* ≡
let
num1 = *running-num* ((*cpu-info* (*cpu-state0 g*))!*idx*);
num2 = *running-num* ((*cpu h*)!*idx*)
in
num1 = *num2*

definition *cpuBasicInfo-corrs* :: *cpu-id* ⇒ *GLB-STATE* ⇒ *HV* ⇒ *bool*
where *cpuBasicInfo-corrs idx g h* ≡
let
cpu1 = (*cpu-info* (*cpu-state0 g*))!*idx*;
cpu2 = (*cpu h*)!*idx*
in

cpu1 = *cpu2*

definition *vcpuLimit-corrs* :: *GLB-STATE* ⇒ *HV* ⇒ *bool*
where *vcpuLimit-corrs g h* ≡
let
limit1 = *length* (*vcpu-info* (*vcpu-state g*));
limit2 = *length* (*vcpu h*)
in
limit1 = *limit2*

definition *vcpuPool-corrs* :: *cpu-id* ⇒ *GLB-STATE* ⇒ *HV* ⇒ *bool*
where *vcpuPool-corrs idx g h* ≡
let
pool1 = *CPU.vcpus* ((*cpu-info* (*cpu-state0 g*))!*idx*);
pool2 = *CPU.vcpus* ((*cpu h*)!*idx*)
in
pool1 = *pool2*

definition *activeVCPU-corrs* :: *cpu-id* ⇒ *GLB-STATE* ⇒ *HV* ⇒ *bool*
where *activeVCPU-corrs idx g h* ≡
let
vid1 = *running-num* ((*cpu-info* (*cpu-state0 g*))!*idx*);
vid2 = *running-num* ((*cpu h*)!*idx*)
in
vid1 = *vid2*

definition *activeNew-corrs* :: *cpu-id* ⇒ *vcpu-id* ⇒ *GLB-STATE* ⇒ *HV* ⇒ *bool*
where *activeNew-corrs cidx vcidx g h* ≡
let
vidN1= *get-activeNew-des vcidx cidx g*;
vidN2= ((*CPU-MAPS.cpu-map-active* (*cpu-maps h*))!*cidx*) *vcidx*
in
vidN1 = *vidN2*

definition *vcpuForVM-corrs* :: *cpu-id* ⇒ *vm-id* ⇒ *GLB-STATE* ⇒ *HV* ⇒ *bool*
where *vcpuForVM-corrs cidx vmidx g h* ≡
let
vl1 =*CPU.vcpus* ((*cpu-info* (*cpu-state0 g*))!*cidx*);
vido1 = *get-targetVCPU-des vl1 vmidx g*;
map2 = (*cpu-map-idByVM* (*cpu-maps h*)) ! *cidx*;
vido2 = *map2 vmidx*
in
vido1 = *vido2*

definition *vcpuIsContain-corrs* :: *cpu-id* ⇒ *vcpu-id* ⇒ *GLB-STATE* ⇒ *HV* ⇒ *bool*
  where *vcpuIsContain-corrs cidx vcidx g h* ≡
    let
      ret1 = *is-vcpu-inPool cidx vcidx g*;
      map2 = (*cpu-is-haveVCPU* (*cpu-maps h*)) ! *cidx*;
      ret2 = *map2 vcidx*
    in
      ret1 = *ret2*

definition *vcpuID-result-corrs* :: *vcpu-id option* ⇒ *bool* ⇒ *bool*
  where *vcpuID-result-corrs ido b* ≡
    case (*ido*, *b*) *of*
        (*Some i*, *True*) ⇒ *True* |
        (*None*, *False*) ⇒ *True* |
        - ⇒ *False*

lemma *vcpu-run* :
  ⦃λ*s*. *cpuInfo-corrs s hv* ∧ *vmState-corrs s hv* ∧ *cpuState-corrs s hv*
        ∧ *vmID-corrs idx s hv* ⦄
    vcpu-run-des idx
   ⦃λ*r s*. *vmState-corrs s* (*vcpu-run-req hv idx*) ∧
     cpuState-corrs s (*vcpu-run-req hv idx*) ⦄
  unfolding *cpuInfo-corrs-def vmState-corrs-def cpuState-corrs-def*
  apply (*simp add*: *Let-def*)
  unfolding *vcpu-run-des-def*
  apply *wpsimp*
  apply (*simp add*: *vcpu-run-req-def*)
  unfolding *get-vmid-cid-des-def get-vmid-cid-req-def*
  apply (*simp add*: *Let-def*)
  unfolding *get-vmid-cid-req-def*
  apply *auto*
  unfolding *vmID-corrs-def*
  apply (*simp add*: *get-vmid-cid-des-def Let-def*)
  apply (*simp add*: *get-vmid-cid-req-def Let-def*)
  done

lemma *vcpu-pool-switch* :
  ⦃λ*s*. *vcpuState-corrs s hv* ∧ *cpuInfo-corrs s hv* ∧ *activeNew-corrs cid vcid s hv*
⦄
    vcpu-pool-switch-des cid vcid
    ⦃λ*r s*. *vcpuState-corrs s* (*vcpu-pool-switch-req hv cid vcid*) ∧
       cpuInfo-corrs s (*vcpu-pool-switch-req hv cid vcid*) ⦄

unfolding *vcpuState-corrs-def cpuInfo-corrs-def activeNew-corrs-def*
unfolding *Let-def*
unfolding *vcpu-pool-switch-des-def*
apply *wpsimp*
apply (*simp add*: *vcpu-pool-switch-req-def Let-def*)
apply *auto*
unfolding *set-active-vcpu-req-def Let-def*
unfolding *runningNum-corrs-def Let-def*
apply *simp+*
done

lemma *vcpu-pool-pop-through-vmid* :
$\{\!|\lambda s.$ *cpuState-corrs s hv* $\wedge$ *vmState-corrs s hv* $\wedge$ *vcpuPool-corrs cid s hv*
$\wedge$ *vcpuForVM-corrs cid vmid s hv* $|\!\}$
vcpu-pool-pop-through-vmid-des cid vmid
$\{\!|\lambda r\ s.$ *cpuState-corrs s* (*fst* (*vcpu-pool-pop-through-vmid-req hv cid vmid*))
$\wedge$ *vmState-corrs s* (*fst* (*vcpu-pool-pop-through-vmid-req hv cid vmid*))
$\wedge$ *vcpuID-result-corrs r* (*snd* (*vcpu-pool-pop-through-vmid-req hv cid vmid*))$|\!\}$

unfolding *cpuState-corrs-def Let-def*
unfolding *vmState-corrs-def Let-def*
unfolding *vcpuPool-corrs-def Let-def*
unfolding *vcpuForVM-corrs-def Let-def*
apply (*simp add*: *vcpu-pool-pop-through-vmid-des-def*)
apply *wpsimp*
apply *auto*
unfolding *vcpuID-result-corrs-def*
apply *auto*
apply (*simp add*: *vcpu-pool-pop-through-vmid-req-def*)+
unfolding *vcpu-pool-pop-through-vmid-req-def Let-def*
apply *simp*
try
by *auto*

lemma *mathSub*: $p1 = (q1 :: nat) \Longrightarrow p1 - 1 = q1 - 1$
by *simp*

lemma *vcpu-pool-suspend* :
$cid < length\ (cpu\ hv) \Longrightarrow$ $\{\!|\lambda s.$ *vcpuPool-corrs cid s hv* $\wedge$ *vcpuState-corrs s hv* $\wedge$ *runningNum-corrs cid s hv*
$\wedge$ *vcpuIsContain-corrs cid vcid s hv*
$\wedge$ *length* (*cpu hv*) = *length* (*cpu-info* (*cpu-state0 s*)) $|\!\}$
vcpu-pool-suspend-des cid vcid

$\{\!|\lambda r\ s.$ *vcpuState-corrs s* $($*fst*$($*vcpu-pool-suspend-req hv cid vcid*$))$ $\wedge$
runningNum-corrs cid s $($*fst*$($*vcpu-pool-suspend-req hv cid vcid*$))$ $\wedge$
$r =$ *snd*$($*vcpu-pool-suspend-req hv cid vcid*$)$ $|\!\}$

unfolding *vcpuPool-corrs-def Let-def*
unfolding *vcpuState-corrs-def Let-def*
unfolding *runningNum-corrs-def Let-def*
unfolding *vcpuIsContain-corrs-def Let-def*
apply (*simp add*: *vcpu-pool-suspend-des-def*)
apply *wpsimp*
apply *auto*
apply (*simp add*: *vcpu-pool-suspend-req-def*)+
defer
unfolding *vcpu-pool-suspend-req-def Let-def*
apply *auto*
done

lemma *vcpu-pool-wakeup* :
cid $<$ *length* (*cpu hv*) $\Longrightarrow$ $\{\!|\lambda s.$ *vcpuPool-corrs cid s hv* $\wedge$ *vcpuState-corrs s hv* $\wedge$ *runningNum-corrs cid s hv*
$\wedge$ *vcpuIsContain-corrs cid vcid s hv*
$\wedge$ *length* (*cpu hv*) $=$ *length* (*cpu-info* (*cpu-state0 s*)) $|\!\}$
vcpu-pool-wakeup-des cid vcid
$\{\!|\lambda r\ s.$ *vcpuState-corrs s* $($*fst*$($*vcpu-pool-wakeup-req hv cid vcid*$))$ $\wedge$
runningNum-corrs cid s $($*fst*$($*vcpu-pool-wakeup-req hv cid vcid*$))$ $\wedge$
$r =$ *snd*$($*vcpu-pool-wakeup-req hv cid vcid*$)$ $|\!\}$
unfolding *vcpuPool-corrs-def Let-def*
unfolding *vcpuState-corrs-def Let-def*
unfolding *runningNum-corrs-def Let-def*
unfolding *vcpuIsContain-corrs-def Let-def*
apply (*simp add*: *vcpu-pool-wakeup-des-def*)
apply *wpsimp*
apply *auto*
apply (*simp add*: *vcpu-pool-wakeup-req-def*)+
unfolding *Let-def*
defer
apply (*simp add*: *vcpu-pool-wakeup-req-def Let-def*)+
done

lemma *vcpu-pool-remove* :
cid $<$ *length* (*cpu hv*) $\Longrightarrow$ $\{\!|\lambda s.$ *vcpuPool-corrs cid s hv* $\wedge$ *vcpuState-corrs s hv* $\wedge$ *cpuBasicInfo-corrs cid s hv*
$\wedge$ *vcpuIsContain-corrs cid vcid s hv*
$\wedge$ *length* (*cpu hv*) $=$ *length* (*cpu-info* (*cpu-state0 s*))
$\wedge$ *vcpuLimit-corrs s hv*$|\!\}$
vcpu-pool-remove-des cid vcid

$\{\!|\lambda r\ s.\ vcpuState\text{-}corrs\ s\ (fst(vcpu\text{-}pool\text{-}remove\text{-}req\ hv\ cid\ vcid))\ \wedge$
$cpuBasicInfo\text{-}corrs\ cid\ s\ (fst(vcpu\text{-}pool\text{-}remove\text{-}req\ hv\ cid\ vcid)) \wedge$
$r = snd(vcpu\text{-}pool\text{-}remove\text{-}req\ hv\ cid\ vcid)\ |\!\}$

unfolding *vcpuPool-corrs-def Let-def*
unfolding *vcpuState-corrs-def Let-def*
unfolding *cpuBasicInfo-corrs-def Let-def*
unfolding *vcpuIsContain-corrs-def Let-def*
apply (*simp add*: *vcpu-pool-remove-des-def*)
apply *wpsimp*
apply *auto*
apply (*simp add*: *vcpu-pool-remove-req-def*)+
unfolding *Let-def*
apply *auto*
unfolding *vcpu-pool-remove-req-def Let-def*
apply *auto*
apply (*simp add*: *vcpuLimit-corrs-def*)+
done

lemma *vcpu-shutdown* :
$cid < length\ (cpu\ hv) \wedge idx = length\ (vcpu\ hv) \Longrightarrow \{\!|\lambda s.\ vcpuPool\text{-}corrs\ cid\ s\ hv \wedge vcpuState\text{-}corrs\ s\ hv$
$\wedge\ length\ (cpu\ hv) = length\ (cpu\text{-}info\ (cpu\text{-}state0\ s))$
$\wedge\ vcpuLimit\text{-}corrs\ s\ hv$
$\wedge\ cpuBasicInfo\text{-}corrs\ cid\ s\ hv$
$\wedge\ cpuState\text{-}corrs\ s\ hv$
$\wedge\ activeNew\text{-}corrs\ cid\ idx\ s\ hv\ \wedge\ vcpuIsContain\text{-}corrs\ cid\ vcid\ s\ hv\ |\!\}$
vcpu-shutdown-des cid vcid
$\{\!|\lambda r\ s.\ vcpuState\text{-}corrs\ s\ (vcpu\text{-}shutdown\text{-}req\ hv\ cid\ vcid)$
$\wedge\ cpuState\text{-}corrs\ s\ (vcpu\text{-}shutdown\text{-}req\ hv\ cid\ vcid)$
$\wedge\ cpuBasicInfo\text{-}corrs\ cid\ s\ (vcpu\text{-}shutdown\text{-}req\ hv\ cid\ vcid)\ |\!\}$

unfolding *vcpuPool-corrs-def Let-def*
unfolding *vcpuState-corrs-def Let-def*
unfolding *cpuBasicInfo-corrs-def Let-def*
unfolding *activeNew-corrs-def Let-def*
unfolding *vcpuIsContain-corrs-def Let-def*
apply (*simp add*: *vcpu-shutdown-des-def*)
unfolding *cpu-idle-des-def*
apply *wpsimp*
apply *auto*
unfolding *vcpu-shutdown-req-def cpu-idle-req-def*
apply (*simp add*: *Let-def*)+
unfolding *cpuState-corrs-def*
apply (*simp add*: *Let-def*)+
unfolding *vcpuLimit-corrs-def*
apply *auto*
unfolding *Let-def*

try
 apply *simp+*
done

lemma *cpu-schedule* :
$cid < length\ (cpu\ hv) \wedge idx = length\ (vcpu\ hv) \Longrightarrow \{\!|\lambda s.\ vcpuPool\text{-}corrs\ cid\ s\ hv \wedge vcpuState\text{-}corrs\ s\ hv$
$\wedge\ length\ (cpu\ hv) = length\ (cpu\text{-}info\ (cpu\text{-}state0\ s))$
$\wedge\ vcpuLimit\text{-}corrs\ s\ hv$
$\wedge\ cpuBasicInfo\text{-}corrs\ cid\ s\ hv$
$\wedge\ activeNew\text{-}corrs\ cid\ idx\ s\ hv\ |\!\}$
$cpu\text{-}schedule\text{-}des\ cid$
$\{\!|\lambda r\ s.\ cpuBasicInfo\text{-}corrs\ cid\ s\ (cpu\text{-}schedule\text{-}req\ hv\ cid)$
$\wedge\ vcpuState\text{-}corrs\ s\ (cpu\text{-}schedule\text{-}req\ hv\ cid)\ |\!\}$

unfolding *vcpuPool-corrs-def Let-def*
unfolding *vcpuState-corrs-def Let-def*
unfolding *cpuBasicInfo-corrs-def Let-def*
unfolding *activeNew-corrs-def Let-def*
unfolding *vcpuIsContain-corrs-def Let-def*
apply (*simp add*: *cpu-schedule-des-def*)
apply *wpsimp*
apply *auto*
unfolding *cpu-schedule-req-def vcpuLimit-corrs-def Let-def*
 apply *auto*
done

lemma *aux1*: $cid < length\ A \Longrightarrow running\text{-}num\ (A[cid := C]\ !\ cid) = running\text{-}num\ C$
apply(*subgoal-tac* $(A[cid := C]\ !\ cid) = C$)
by *auto*

lemma
$cid < length\ A \wedge cid < length\ B \Longrightarrow running\text{-}num\ (A\ !\ cid) = running\text{-}num\ (B\ !\ cid) \Longrightarrow$
$running\text{-}num$
$(A[cid := (A!\ cid)(\!|running\text{-}num := running\text{-}num\ (B\ !\ cid) - Suc\ 0|\!)]\ !\ cid) =$
$running\text{-}num\ (B[cid := (B!\ cid)(\!|running\text{-}num := running\text{-}num\ (B\ !\ cid) - Suc\ 0|\!)]\ !\ cid)$

by *simp*

end
theory *VM-Mgmt*

imports *GLOBAL-STATE VCPU-Mgmt*

begin

this section Describes VM events

value *filter* ($\lambda x.$ $\neg x=2$) [*1..5*]

definition *range-in-range* :: *u64* $\Rightarrow$ *u64* $\Rightarrow$ *u64* $\Rightarrow$ *u64* $\Rightarrow$ *bool* **where**
range-in-range base1 size1 base2 size2 $\equiv$
(*base1* >= *base2*) $\wedge$ ((*base1* + *size1*) <= (*base2* + *size2*))

definition *getRegionIDX-byIPA* :: *u64* $\Rightarrow$ *vm-id* $\Rightarrow$ *GLB-STATE* $\Rightarrow$ *region-idx option* **where**
getRegionIDX-byIPA ipa idx g $\equiv$
let
regions = *VM.address* ((*vm-info*(*vm-state0 g*))!*idx*);
ret = *find-index* ($\lambda x.$ *range-in-range* (*ipa*−*offset x*) *0* (*pa-start x*) (*pa-length x*)) *regions*
in
if(*ret*<*length regions*) *then*
Some ret
else
None

definition *getRegionIDX-byPA* :: *u64* $\Rightarrow$ *vm-id* $\Rightarrow$ *GLB-STATE* $\Rightarrow$ *region-idx option* **where**
getRegionIDX-byPA pa idx g $\equiv$
let
regions = *VM.address* ((*vm-info*(*vm-state0 g*))!*idx*);
ret = *find-index* ($\lambda x.$ *range-in-range pa 0* (*pa-start x*) (*pa-length x*)) *regions*
in
if(*ret*<*length regions*) *then*
Some ret

else
None

definition *vm-ipa2pa-des* :: *vm-id* ⇒ *u64* ⇒ (*GLB-STATE*,*u64*) *nondet-monad*
where
vm-ipa2pa-des vmid ipa ≡
condition (λσ. *ipa* = *0*)
(*return 0*)
(*do*
ret ← *gets* (λσ. *getRegionIDX-byIPA ipa vmid* σ);
if(¬(*ret* = *None*)) *then*
(*do*
idx ← *return* (*the ret*);
regions ← *gets* (λσ. *VM.address* ((*vm-info*(*vm-state0* σ))!*vmid*));
return (*ipa* − *offset* (*regions*!*idx*))
od)
else
return 0
od)

definition *vm-pa2ipa-des* :: *vm-id* ⇒ *u64* ⇒ (*GLB-STATE*,*u64*) *nondet-monad*
where
vm-pa2ipa-des vmid pa ≡
condition (λσ. *pa* = *0*)
(*return 0*)
(*do*
ret ← *gets* (λσ. *getRegionIDX-byPA pa vmid* σ);
if(¬(*ret* = *None*)) *then*
(*do*
idx ← *return* (*the ret*);
regions ← *gets* (λσ. *VM.address* ((*vm-info*(*vm-state0* σ))!*vmid*));
return (*pa* + *offset* (*regions*!*idx*))
od)
else
return 0
od)

definition *vmm-shutdown-vm-des* :: *cpu-id* ⇒ (*GLB-STATE*,*unit*)*nondet-monad*
where

$vmm\text{-}shutdown\text{-}vm\text{-}des\ cid \equiv$
(*do*
  $cpu0 \leftarrow gets\ (\lambda\sigma.\ (cpu\text{-}info\ (cpu\text{-}state0\ \sigma))!cid);$
  $actIDX \leftarrow return\ (active\text{-}vcpu\ cpu0);$
  $vcpu \leftarrow gets\ (\lambda\sigma.\ (vcpu\text{-}info(vcpu\text{-}state\ \sigma))!actIDX);$
  $idTarget \leftarrow return\ (vmID\ vcpu);$
  $ws \leftarrow gets\ (\lambda\sigma.\ VM\text{-}State0.work\text{-}state\ (vm\text{-}state0\ \sigma));$
  $wsN \leftarrow return\ (ws[idTarget{:=}VM\text{-}S\text{-}INV]);$
  $vm\text{-}stateN \leftarrow gets\ (\lambda\sigma.\ (vm\text{-}state0\ \sigma)(\!|VM\text{-}State0.work\text{-}state{:=}wsN|\!));$
  (*do*
    $limit \leftarrow gets\ (\lambda\sigma.\ length\ (vcpu\text{-}info\ (vcpu\text{-}state\ \sigma)));$
    $aN \leftarrow gets\ (\lambda\sigma.\ get\text{-}activeNew\text{-}des\ limit\ cid\ \sigma\ );$
    $vwsN \leftarrow gets\ (\lambda\sigma.\ (VCPU\text{-}State.work\text{-}state\ (vcpu\text{-}state\ \sigma))[actIDX{:=}VCPU\text{-}S\text{-}INV]);$
    $vsN \leftarrow gets\ (\lambda\sigma.\ (vcpu\text{-}state\ \sigma)(\!|VCPU\text{-}State.work\text{-}state{:=}vwsN|\!));$
    $cpu0 \leftarrow gets\ (\lambda\sigma.\ (cpu\text{-}info\ (cpu\text{-}state0\ \sigma))!cid);$
    $cpu0 \leftarrow return(cpu0(\!|running\text{-}num{:=}running\text{-}num\ cpu0\ -1|\!));$
    $cpus \leftarrow gets\ (\lambda\sigma.\ (cpu\text{-}info\ (cpu\text{-}state0\ \sigma))[cid{:=}cpu0]);$
    $csN \leftarrow\ gets\ (\lambda\sigma.\ (cpu\text{-}state0\ \sigma)(\!|cpu\text{-}info{:=}cpus|\!));$
    if$(running\text{-}num\ cpu0\ =\ 0)$ *then*
    (*do*
    $modify(\lambda\sigma.\ \sigma(\!|cpu\text{-}state0{:=}csN,\ vcpu\text{-}state{:=}vsN,vm\text{-}state0{:=}vm\text{-}stateN|\!));$
      cpu-idle-des cid
    od)
     else
       if$(aN\ =\ None\ )$ *then*
       (*do*
       $modify(\lambda\sigma.\ \sigma(\!|cpu\text{-}state0{:=}csN,\ vcpu\text{-}state{:=}vsN,vm\text{-}state0{:=}vm\text{-}stateN$
$|\!));$
         return ()
       od)
       else
         (*do*
           $vwsN \leftarrow return\ (\ vwsN[the\ aN{:=}\ VCPU\text{-}S\text{-}ACT]);$
           $vsN \leftarrow gets\ (\lambda\sigma.\ (vcpu\text{-}state\ \sigma)(\!|VCPU\text{-}State.work\text{-}state{:=}vwsN|\!));$
           $cpu0 \leftarrow return(\ cpu0(\!|active\text{-}vcpu{:=}\ the\ aN|\!)\ );$
           $cpus \leftarrow gets\ (\lambda\sigma.\ (cpu\text{-}info\ (cpu\text{-}state0\ \sigma))[cid{:=}cpu0]);$
           $csN \leftarrow\ gets\ (\lambda\sigma.\ (cpu\text{-}state0\ \sigma)(\!|cpu\text{-}info{:=}cpus|\!));$
       $modify(\lambda\sigma.\ \sigma(\!|cpu\text{-}state0{:=}csN,\ vcpu\text{-}state{:=}vsN,vm\text{-}state0{:=}vm\text{-}stateN$
$|\!));$
           return ()
         od)
    od)
od)

definition $vmm\text{-}reset\text{-}vm\text{-}des :: cpu\text{-}id \Rightarrow (GLB\text{-}STATE,unit)nondet\text{-}monad$ **where**
  $vmm\text{-}reset\text{-}vm\text{-}des\ cid \equiv$
(*do*

```
    cpu0 ← gets (λσ. (cpu-info (cpu-state0 σ))!cid);
    actIDX ← return (active-vcpu cpu0);
    vcpu ← gets (λσ. (vcpu-info(vcpu-state σ))!actIDX);
    idTarget ← return (vmID vcpu);
    cws ← gets (λσ. CPU-State0.work-state (cpu-state0 σ));
    cwsN ← return (cws[cid:=CPU-S-RUN]);
    cpu-stateN ← gets (λσ. (cpu-state0 σ)(|CPU-State0.work-state:=cwsN|));
    vmws ← gets (λσ. VM-State0.work-state (vm-state0 σ));
    vmwsN ← return (vmws[idTarget:=VM-S-ACT]);
    vm-stateN ← gets (λσ. (vm-state0 σ)(|VM-State0.work-state:=vmwsN|));
    modify(λσ. σ(|cpu-state0:=cpu-stateN, vm-state0:=vm-stateN|) );
    return ()
  od)
```

```
definition vm-vcpuid-to-pcpuid-des :: vm-id ⇒ vcpu-idx ⇒ (GLB-STATE,cpu-id
option)nondet-monad where
  vm-vcpuid-to-pcpuid-des vmid vidx ≡
    condition (λσ . vidx < length (VM.vcpus((vm-info (vm-state0 σ))!vmid)) )
      (do
          vm0 ← gets (λσ. ( vm-info (vm-state0 σ) )!vmid );
          vid ← gets (λσ. (VM.vcpus vm0)!vidx );
          vcpu ← gets (λσ. (vcpu-info (vcpu-state σ)) ! vid );
          cid ← return (physID vcpu);
          return (Some cid)
      od)
      (return None)
```

```
definition getVCPU-byVM :: vcpu-id list ⇒ cpu-id ⇒ GLB-STATE ⇒ vcpu-idx
option where
  getVCPU-byVM vcls cid g ≡
      let
        limit = length vcls;
        vcpuGLB = vcpu-info (vcpu-state g);
        idx = find-index (λx. (physID (vcpuGLB!x)) = cid ) vcls
      in
        if(idx < limit) then
          Some idx
        else
          None
```

```
definition vm-pcpuid-to-vcpuid-des :: vm-id ⇒ cpu-id ⇒ (GLB-STATE,vcpu-idx
option)nondet-monad where
  vm-pcpuid-to-vcpuid-des vmid cid ≡
```

```
  (do
     vm0 ← gets (λσ. ( vm-info (vm-state0 σ) )!vmid );
     vcpus ← gets (λσ. (VM.vcpus vm0));
     ret ← gets (λσ. getVCPU-byVM vcpus cid σ);
     return ret
  od)
```

value *fst* (*List.zip* [*True*,*False*,*True*] [*3..5*] ! *1*)

```
definition vmm-list-vm-info-des :: (GLB-STATE, VM-INFO list)nondet-monad
where
 vmm-list-vm-info-des ≡
 (do
   vms ← gets (λσ. vm-info (vm-state0 σ)) ;
   vmws ← gets (λσ. VM-State0.work-state (vm-state0 σ)) ;
   vmZ ← return (List.zip vms vmws);
   vmInfoS ← return ( List.map
     (λx. (|vmId=VM.id (fst x), vmName=VM.name (fst x), vmType=VM.type
(fst x),
          vmState=snd x|))
    vmZ);
   return vmInfoS
  od)
```

```
definition vmm-get-vm-id-des :: vm-name ⇒ (GLB-STATE, vm-id option)nondet-monad
where
  vmm-get-vm-id-des name0 ≡
   (do
     vms ← gets(λσ. vm-info (vm-state0 σ));
     limit ← return (length vms);
     ret ← (whileLoopE (λi t. i<limit)
             (λ i.
                condition (λt. (name (vms ! i)) =name0 )
                (throwError i)
                (returnOk (i+1))
             )
             (0)
           )<catch> (λe. return e);
     condition (λσ. ret=limit)
       (return None)
       (return (Some ret))
   od)
```

end