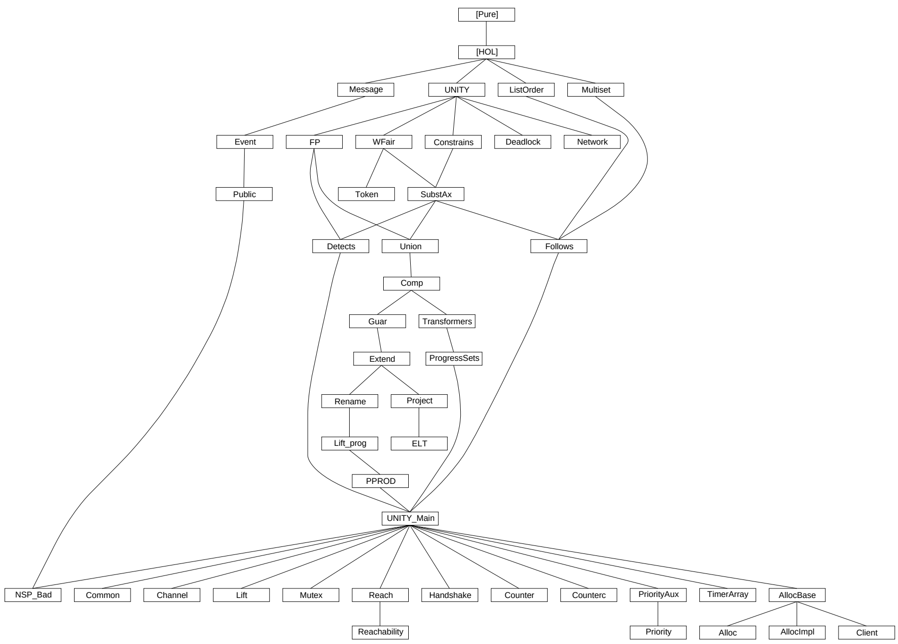
The UNITY Formalism

Sidi Ehmety and Lawrence C. Paulson

November 22, 2007

Contents



[Pure]
[HOL]
Message
UNITY
ListOrder
Multiset
Event
FP
WFair
Constrains
Deadlock
Network
Public
Token
SubstAx
Detects
Union
Follows
Comp
Guar
Transformers
Extend
ProgressSets
Rename
Project
Lift_prog
ELT
PPROD
UNITY_Main
NSP_Bad
Common
Channel
Lift
Mutex
Reach
Handshake
Counter
Counterc
PriorityAux
TimerArray
AllocBase
Reachability
Priority
Alloc
AllocImpl
Client

1 The Basic UNITY Theory

theory *UNITY* **imports** *Main* **begin**

```
typedef (Program)
  'a program = "{(init:: 'a set, acts :: ('a * 'a)set set,
                 allowed :: ('a * 'a)set set). Id ∈ acts & Id: allowed}"
```

⟨*proof*⟩

```
constdefs
  Acts :: "'a program => ('a * 'a)set set"
    "Acts F == (%(init, acts, allowed). acts) (Rep_Program F)"

  "constrains" :: "['a set, 'a set] => 'a program set"  (infixl "co"     60)
    "A co B == {F. ∀act ∈ Acts F. act``A ⊆ B}"

  unless  :: "['a set, 'a set] => 'a program set"  (infixl "unless" 60)
    "A unless B == (A-B) co (A ∪ B)"

  mk_program :: "('a set * ('a * 'a)set set * ('a * 'a)set set)
                 => 'a program"
    "mk_program == %(init, acts, allowed).
                      Abs_Program (init, insert Id acts, insert Id allowed)"

  Init :: "'a program => 'a set"
    "Init F == (%(init, acts, allowed). init) (Rep_Program F)"

  AllowedActs :: "'a program => ('a * 'a)set set"
    "AllowedActs F == (%(init, acts, allowed). allowed) (Rep_Program F)"

  Allowed :: "'a program => 'a program set"
    "Allowed F == {G. Acts G ⊆ AllowedActs F}"

  stable     :: "'a set => 'a program set"
    "stable A == A co A"

  strongest_rhs :: "['a program, 'a set] => 'a set"
    "strongest_rhs F A == Inter {B. F ∈ A co B}"

  invariant :: "'a set => 'a program set"
    "invariant A == {F. Init F ⊆ A} ∩ stable A"

  increasing :: "['a => 'b::{order}] => 'a program set"
    — Polymorphic in both states and the meaning of ≤
    "increasing f == ⋂z. stable {s. z ≤ f s}"
```

Perhaps HOL shouldn't add this in the first place!

declare *image_Collect [simp del]*

1.0.1 The abstract type of programs

```
lemmas program_typedef =
     Rep_Program Rep_Program_inverse Abs_Program_inverse
```

Program_def Init_def Acts_def AllowedActs_def mk_program_def

lemma *Id_in_Acts [iff]: "Id ∈ Acts F"*
⟨*proof*⟩

lemma *insert_Id_Acts [iff]: "insert Id (Acts F) = Acts F"*
⟨*proof*⟩

lemma *Acts_nonempty [simp]: "Acts F ≠ {}"*
⟨*proof*⟩

lemma *Id_in_AllowedActs [iff]: "Id ∈ AllowedActs F"*
⟨*proof*⟩

lemma *insert_Id_AllowedActs [iff]: "insert Id (AllowedActs F) = AllowedActs F"*
⟨*proof*⟩

1.0.2 Inspectors for type "program"

lemma *Init_eq [simp]: "Init (mk_program (init,acts,allowed)) = init"*
⟨*proof*⟩

lemma *Acts_eq [simp]: "Acts (mk_program (init,acts,allowed)) = insert Id acts"*
⟨*proof*⟩

lemma *AllowedActs_eq [simp]:*
"AllowedActs (mk_program (init,acts,allowed)) = insert Id allowed"
⟨*proof*⟩

1.0.3 Equality for UNITY programs

lemma *surjective_mk_program [simp]:*
"mk_program (Init F, Acts F, AllowedActs F) = F"
⟨*proof*⟩

lemma *program_equalityI:*
```
"[| Init F = Init G; Acts F = Acts G; AllowedActs F = AllowedActs G |]

 ==> F = G"
```
⟨*proof*⟩

lemma *program_equalityE:*
```
"[| F = G;
    [| Init F = Init G; Acts F = Acts G; AllowedActs F = AllowedActs G
|]
    ==> P |] ==> P"
```
⟨*proof*⟩

lemma *program_equality_iff:*
```
"(F=G) =
 (Init F = Init G & Acts F = Acts G &AllowedActs F = AllowedActs G)"
```
⟨*proof*⟩

1.0.4 co

lemma *constrainsI:*
```
"(!!act s s'. [| act: Acts F;  (s,s') ∈ act;  s ∈ A |] ==> s': A')
 ==> F ∈ A co A'"
```
⟨*proof*⟩

lemma *constrainsD:*
```
"[| F ∈ A co A'; act: Acts F;  (s,s'): act;  s ∈ A |] ==> s': A'"
```
⟨*proof*⟩

lemma *constrains_empty [iff]: "F ∈ {} co B"*
⟨*proof*⟩

lemma *constrains_empty2 [iff]: "(F ∈ A co {}) = (A={})"*
⟨*proof*⟩

lemma *constrains_UNIV [iff]: "(F ∈ UNIV co B) = (B = UNIV)"*
⟨*proof*⟩

lemma *constrains_UNIV2 [iff]: "F ∈ A co UNIV"*
⟨*proof*⟩

monotonic in 2nd argument

lemma *constrains_weaken_R:*
```
"[| F ∈ A co A'; A'<=B' |] ==> F ∈ A co B'"
```
⟨*proof*⟩

anti-monotonic in 1st argument

lemma *constrains_weaken_L:*
```
"[| F ∈ A co A'; B ⊆ A |] ==> F ∈ B co A'"
```
⟨*proof*⟩

lemma *constrains_weaken:*
```
"[| F ∈ A co A'; B ⊆ A; A'<=B' |] ==> F ∈ B co B'"
```
⟨*proof*⟩

1.0.5 Union

lemma *constrains_Un:*
```
"[| F ∈ A co A'; F ∈ B co B' |] ==> F ∈ (A ∪ B) co (A' ∪ B')"
```
⟨*proof*⟩

lemma *constrains_UN:*
```
"(!!i. i ∈ I ==> F ∈ (A i) co (A' i))
 ==> F ∈ (⋃i ∈ I. A i) co (⋃i ∈ I. A' i)"
```
⟨*proof*⟩

lemma *constrains_Un_distrib: "(A ∪ B) co C = (A co C) ∩ (B co C)"*
⟨*proof*⟩

lemma *constrains_UN_distrib: "(⋃i ∈ I. A i) co B = (⋂i ∈ I. A i co B)"*
⟨*proof*⟩

lemma *constrains_Int_distrib:* "C co (A ∩ B) = (C co A) ∩ (C co B)"
⟨*proof*⟩

lemma *constrains_INT_distrib:* "A co (⋂i ∈ I. B i) = (⋂i ∈ I. A co B i)"
⟨*proof*⟩

1.0.6 Intersection

lemma *constrains_Int:*
"[| F ∈ A co A'; F ∈ B co B' |] ==> F ∈ (A ∩ B) co (A' ∩ B')"
⟨*proof*⟩

lemma *constrains_INT:*
"(!!i. i ∈ I ==> F ∈ (A i) co (A' i))
==> F ∈ (⋂i ∈ I. A i) co (⋂i ∈ I. A' i)"
⟨*proof*⟩

lemma *constrains_imp_subset:* "F ∈ A co A' ==> A ⊆ A'"
⟨*proof*⟩

The reasoning is by subsets since "co" refers to single actions only. So this rule isn't that useful.

lemma *constrains_trans:*
"[| F ∈ A co B; F ∈ B co C |] ==> F ∈ A co C"
⟨*proof*⟩

lemma *constrains_cancel:*
"[| F ∈ A co (A' ∪ B); F ∈ B co B' |] ==> F ∈ A co (A' ∪ B')"
⟨*proof*⟩

1.0.7 unless

lemma *unlessI:* "F ∈ (A-B) co (A ∪ B) ==> F ∈ A unless B"
⟨*proof*⟩

lemma *unlessD:* "F ∈ A unless B ==> F ∈ (A-B) co (A ∪ B)"
⟨*proof*⟩

1.0.8 stable

lemma *stableI:* "F ∈ A co A ==> F ∈ stable A"
⟨*proof*⟩

lemma *stableD:* "F ∈ stable A ==> F ∈ A co A"
⟨*proof*⟩

lemma *stable_UNIV [simp]:* "stable UNIV = UNIV"
⟨*proof*⟩

1.0.9 Union

lemma *stable_Un:*
"[| F ∈ stable A; F ∈ stable A' |] ==> F ∈ stable (A ∪ A')"

⟨*proof*⟩

lemma *stable_UN:*
```
"(!!i. i ∈ I ==> F ∈ stable (A i)) ==> F ∈ stable (⋃i ∈ I. A i)"
```
⟨*proof*⟩

lemma *stable_Union:*
```
"(!!A. A ∈ X ==> F ∈ stable A) ==> F ∈ stable (⋃X)"
```
⟨*proof*⟩

1.0.10 Intersection

lemma *stable_Int:*
```
"[| F ∈ stable A;  F ∈ stable A' |] ==> F ∈ stable (A ∩ A')"
```
⟨*proof*⟩

lemma *stable_INT:*
```
"(!!i. i ∈ I ==> F ∈ stable (A i)) ==> F ∈ stable (⋂i ∈ I. A i)"
```
⟨*proof*⟩

lemma *stable_Inter:*
```
"(!!A. A ∈ X ==> F ∈ stable A) ==> F ∈ stable (⋂X)"
```
⟨*proof*⟩

lemma *stable_constrains_Un:*
```
"[| F ∈ stable C; F ∈ A co (C ∪ A') |] ==> F ∈ (C ∪ A) co (C ∪ A')"
```
⟨*proof*⟩

lemma *stable_constrains_Int:*
```
"[| F ∈ stable C; F ∈  (C ∩ A) co A' |] ==> F ∈ (C ∩ A) co (C ∩ A')"
```
⟨*proof*⟩

lemmas *stable_constrains_stable = stable_constrains_Int[THEN stableI, standard]*

1.0.11 invariant

lemma *invariantI: "[| Init F ⊆ A; F ∈ stable A |] ==> F ∈ invariant A"*
⟨*proof*⟩

Could also say *invariant A ∩ invariant B ⊆ invariant (A ∩ B)*

lemma *invariant_Int:*
```
"[| F ∈ invariant A;  F ∈ invariant B |] ==> F ∈ invariant (A ∩ B)"
```
⟨*proof*⟩

1.0.12 increasing

lemma *increasingD:*
```
"F ∈ increasing f ==> F ∈ stable {s. z ⊆ f s}"
```
⟨*proof*⟩

lemma *increasing_constant [iff]: "F ∈ increasing (%s. c)"*
⟨*proof*⟩

lemma *mono_increasing_o:*
```
"mono g ==> increasing f ⊆ increasing (g o f)"
```

⟨*proof*⟩

lemma *strict_increasingD:*
```
    "!!z::nat. F ∈ increasing f ==> F ∈ stable {s. z < f s}"
```
⟨*proof*⟩

lemma *elimination:*
```
   "[| ∀m ∈ M. F ∈ {s. s x = m} co (B m) |]
    ==> F ∈ {s. s x ∈ M} co (⋃m ∈ M. B m)"
```
⟨*proof*⟩

As above, but for the trivial case of a one-variable state, in which the state is identified with its one variable.

lemma *elimination_sing:*
```
   "(∀m ∈ M. F ∈ {m} co (B m)) ==> F ∈ M co (⋃m ∈ M. B m)"
```
⟨*proof*⟩

1.0.13 Theoretical Results from Section 6

lemma *constrains_strongest_rhs:*
```
   "F ∈ A co (strongest_rhs F A )"
```
⟨*proof*⟩

lemma *strongest_rhs_is_strongest:*
```
   "F ∈ A co B ==> strongest_rhs F A ⊆ B"
```
⟨*proof*⟩

1.0.14 Ad-hoc set-theory rules

lemma *Un_Diff_Diff [simp]: "A ∪ B - (A - B) = B"*
⟨*proof*⟩

lemma *Int_Union_Union: "Union(B) ∩ A = Union((%C. C ∩ A)‘B)"*
⟨*proof*⟩

Needed for WF reasoning in WFair.thy

lemma *Image_less_than [simp]: "less_than ‘‘ {k} = greaterThan k"*
⟨*proof*⟩

lemma *Image_inverse_less_than [simp]: "less_than^-1 ‘‘ {k} = lessThan k"*
⟨*proof*⟩

1.1 Partial versus Total Transitions

constdefs
```
  totalize_act :: "('a * 'a)set => ('a * 'a)set"
    "totalize_act act == act ∪ diag (-(Domain act))"

  totalize :: "'a program => 'a program"
    "totalize F == mk_program (Init F,
```

```
                                totalize_act ` Acts F,
                                AllowedActs F)"

  mk_total_program :: "('a set * ('a * 'a)set set * ('a * 'a)set set)
                   => 'a program"
    "mk_total_program args == totalize (mk_program args)"

  all_total :: "'a program => bool"
    "all_total F == ∀ act ∈ Acts F. Domain act = UNIV"
```

lemma *insert_Id_image_Acts: "f Id = Id ==> insert Id (f`Acts F) = f ` Acts F"*
⟨*proof*⟩

1.1.1 Basic properties

lemma *totalize_act_Id [simp]: "totalize_act Id = Id"*
⟨*proof*⟩

lemma *Domain_totalize_act [simp]: "Domain (totalize_act act) = UNIV"*
⟨*proof*⟩

lemma *Init_totalize [simp]: "Init (totalize F) = Init F"*
⟨*proof*⟩

lemma *Acts_totalize [simp]: "Acts (totalize F) = (totalize_act ` Acts F)"*
⟨*proof*⟩

lemma *AllowedActs_totalize [simp]: "AllowedActs (totalize F) = AllowedActs F"*
⟨*proof*⟩

lemma *totalize_constrains_iff [simp]: "(totalize F ∈ A co B) = (F ∈ A co B)"*
⟨*proof*⟩

lemma *totalize_stable_iff [simp]: "(totalize F ∈ stable A) = (F ∈ stable A)"*
⟨*proof*⟩

lemma *totalize_invariant_iff [simp]:*
"(totalize F ∈ invariant A) = (F ∈ invariant A)"
⟨*proof*⟩

lemma *all_total_totalize: "all_total (totalize F)"*
⟨*proof*⟩

lemma *Domain_iff_totalize_act: "(Domain act = UNIV) = (totalize_act act = act)"*
⟨*proof*⟩

lemma *all_total_imp_totalize: "all_total F ==> (totalize F = F)"*
⟨*proof*⟩

lemma *all_total_iff_totalize: "all_total F = (totalize F = F)"*
⟨*proof*⟩

lemma *mk_total_program_constrains_iff [simp]:*
```
"(mk_total_program args ∈ A co B) = (mk_program args ∈ A co B)"
```
⟨*proof*⟩

1.2 Rules for Lazy Definition Expansion

They avoid expanding the full program, which is a large expression

lemma *def_prg_Init:*
```
"F == mk_total_program (init,acts,allowed) ==> Init F = init"
```
⟨*proof*⟩

lemma *def_prg_Acts:*
```
"F == mk_total_program (init,acts,allowed)
 ==> Acts F = insert Id (totalize_act ‘ acts)"
```
⟨*proof*⟩

lemma *def_prg_AllowedActs:*
```
"F == mk_total_program (init,acts,allowed)
 ==> AllowedActs F = insert Id allowed"
```
⟨*proof*⟩

An action is expanded if a pair of states is being tested against it

lemma *def_act_simp:*
```
"act == {(s,s'). P s s'} ==> ((s,s') ∈ act) = P s s'"
```
⟨*proof*⟩

A set is expanded only if an element is being tested against it

lemma *def_set_simp: "A == B ==> (x ∈ A) = (x ∈ B)"*
⟨*proof*⟩

1.2.1 Inspectors for type "program"

lemma *Init_total_eq [simp]:*
```
"Init (mk_total_program (init,acts,allowed)) = init"
```
⟨*proof*⟩

lemma *Acts_total_eq [simp]:*
```
"Acts(mk_total_program(init,acts,allowed)) = insert Id (totalize_act‘acts)"
```
⟨*proof*⟩

lemma *AllowedActs_total_eq [simp]:*
```
"AllowedActs (mk_total_program (init,acts,allowed)) = insert Id allowed"
```
⟨*proof*⟩

end

2 Fixed Point of a Program

theory *FP* **imports** *UNITY* **begin**

constdefs

```
  FP_Orig :: "'a program => 'a set"
    "FP_Orig F == Union{A. ALL B. F : stable (A Int B)}"

  FP :: "'a program => 'a set"
    "FP F == {s. F : stable {s}}"
```

lemma `stable_FP_Orig_Int: "F : stable (FP_Orig F Int B)"`
⟨*proof*⟩

lemma `FP_Orig_weakest:`
```
    "(!!B. F : stable (A Int B)) ==> A <= FP_Orig F"
```
⟨*proof*⟩

lemma `stable_FP_Int: "F : stable (FP F Int B)"`
⟨*proof*⟩

lemma `FP_equivalence: "FP F = FP_Orig F"`
⟨*proof*⟩

lemma `FP_weakest:`
```
    "(!!B. F : stable (A Int B)) ==> A <= FP F"
```
⟨*proof*⟩

lemma `Compl_FP:`
```
    "-(FP F) = (UN act: Acts F. -{s. act``{s} <= {s}})"
```
⟨*proof*⟩

lemma `Diff_FP: "A - (FP F) = (UN act: Acts F. A - {s. act``{s} <= {s}})"`
⟨*proof*⟩

lemma `totalize_FP [simp]: "FP (totalize F) = FP F"`
⟨*proof*⟩

end

3 Progress

theory `WFair` **imports** `UNITY` **begin**

The original version of this theory was based on weak fairness. (Thus, the entire UNITY development embodied this assumption, until February 2003.) Weak fairness states that if a command is enabled continuously, then it is eventually executed. Ernie Cohen suggested that I instead adopt unconditional fairness: every command is executed infinitely often.

In fact, Misra's paper on "Progress" seems to be ambiguous about the correct interpretation, and says that the two forms of fairness are equivalent. They differ only on their treatment of partial transitions, which under unconditional fairness behave magically. That is because if there are partial transitions then there may be no fair executions, making all leads-to properties hold vacuously.

Unconditional fairness has some great advantages. By distinguishing partial transitions from total ones that are the identity on part of their domain, it is more expressive. Also, by simplifying the definition of the transient property, it simplifies many proofs. A drawback is that some laws only hold under the assumption that all transitions are total. The best-known of these is the impossibility law for leads-to.

constdefs

— This definition specifies conditional fairness. The rest of the theory is generic to all forms of fairness. To get weak fairness, conjoin the inclusion below with `A ⊆ Domain act`, which specifies that the action is enabled over all of `A`.

```
  transient :: "'a set => 'a program set"
    "transient A == {F. ∃act∈Acts F. act``A ⊆ -A}"

  ensures :: "['a set, 'a set] => 'a program set"       (infixl "ensures" 60)
    "A ensures B == (A-B co A ∪ B) ∩ transient (A-B)"
```

inductive_set

```
  leads :: "'a program => ('a set * 'a set) set"
```
— LEADS-TO constant for the inductive definition

```
  for F :: "'a program"
  where

    Basis:  "F ∈ A ensures B ==> (A,B) ∈ leads F"

  | Trans:  "[| (A,B) ∈ leads F;  (B,C) ∈ leads F |] ==> (A,C) ∈ leads F"

  | Union:  "∀A ∈ S. (A,B) ∈ leads F ==> (Union S, B) ∈ leads F"
```

constdefs

```
  leadsTo :: "['a set, 'a set] => 'a program set"    (infixl "leadsTo" 60)
```
— visible version of the LEADS-TO relation
```
    "A leadsTo B == {F. (A,B) ∈ leads F}"

  wlt :: "['a program, 'a set] => 'a set"
```
— predicate transformer: the largest set that leads to `B`
```
    "wlt F B == Union {A. F ∈ A leadsTo B}"
```

syntax `(xsymbols)`
```
  "op leadsTo" :: "['a set, 'a set] => 'a program set" (infixl "⟼" 60)
```

3.1 transient

lemma `stable_transient:`
```
    "[| F ∈ stable A; F ∈ transient A |] ==> ∃act∈Acts F. A ⊆ - (Domain
act)"
```
⟨*proof*⟩

lemma `stable_transient_empty:`
```
    "[| F ∈ stable A; F ∈ transient A; all_total F |] ==> A = {}"
```

⟨*proof*⟩

lemma *transient_strengthen:*
```
    "[| F ∈ transient A; B ⊆ A |] ==> F ∈ transient B"
```
⟨*proof*⟩

lemma *transientI:*
```
    "[| act: Acts F;  act``A ⊆ -A |] ==> F ∈ transient A"
```
⟨*proof*⟩

lemma *transientE:*
```
    "[| F ∈ transient A;
        !!act. [| act: Acts F;  act``A ⊆ -A |] ==> P |]
     ==> P"
```
⟨*proof*⟩

lemma *transient_empty [simp]: "transient {} = UNIV"*
⟨*proof*⟩

This equation recovers the notion of weak fairness. A totalized program satisfies a transient assertion just if the original program contains a suitable action that is also enabled.

lemma *totalize_transient_iff:*
```
   "(totalize F ∈ transient A) = (∃act∈Acts F. A ⊆ Domain act & act``A ⊆
-A)"
```
⟨*proof*⟩

lemma *totalize_transientI:*
```
    "[| act: Acts F;  A ⊆ Domain act;  act``A ⊆ -A |]
     ==> totalize F ∈ transient A"
```
⟨*proof*⟩

3.2 ensures

lemma *ensuresI:*
```
    "[| F ∈ (A-B) co (A ∪ B); F ∈ transient (A-B) |] ==> F ∈ A ensures B"
```
⟨*proof*⟩

lemma *ensuresD:*
```
    "F ∈ A ensures B ==> F ∈ (A-B) co (A ∪ B) & F ∈ transient (A-B)"
```
⟨*proof*⟩

lemma *ensures_weaken_R:*
```
    "[| F ∈ A ensures A'; A'<=B' |] ==> F ∈ A ensures B'"
```
⟨*proof*⟩

The L-version (precondition strengthening) fails, but we have this

lemma *stable_ensures_Int:*
```
    "[| F ∈ stable C;  F ∈ A ensures B |]
    ==> F ∈ (C ∩ A) ensures (C ∩ B)"
```
⟨*proof*⟩

lemma *stable_transient_ensures:*

"[| F ∈ stable A; F ∈ transient C; A ⊆ B ∪ C |] ==> F ∈ A ensures B"
⟨proof⟩

lemma *ensures_eq*: "(A ensures B) = (A unless B) ∩ transient (A-B)"
⟨proof⟩

3.3 leadsTo

lemma *leadsTo_Basis [intro]*: "F ∈ A ensures B ==> F ∈ A leadsTo B"
⟨proof⟩

lemma *leadsTo_Trans*:
"[| F ∈ A leadsTo B; F ∈ B leadsTo C |] ==> F ∈ A leadsTo C"
⟨proof⟩

lemma *leadsTo_Basis'*:
"[| F ∈ A co A ∪ B; F ∈ transient A |] ==> F ∈ A leadsTo B"
⟨proof⟩

lemma *transient_imp_leadsTo*: "F ∈ transient A ==> F ∈ A leadsTo (-A)"
⟨proof⟩

Useful with cancellation, disjunction

lemma *leadsTo_Un_duplicate*: "F ∈ A leadsTo (A' ∪ A') ==> F ∈ A leadsTo A'"
⟨proof⟩

lemma *leadsTo_Un_duplicate2*:
"F ∈ A leadsTo (A' ∪ C ∪ C) ==> F ∈ A leadsTo (A' ∪ C)"
⟨proof⟩

The Union introduction rule as we should have liked to state it

lemma *leadsTo_Union*:
"(!!A. A ∈ S ==> F ∈ A leadsTo B) ==> F ∈ (Union S) leadsTo B"
⟨proof⟩

lemma *leadsTo_Union_Int*:
"(!!A. A ∈ S ==> F ∈ (A ∩ C) leadsTo B) ==> F ∈ (Union S ∩ C) leadsTo B"
⟨proof⟩

lemma *leadsTo_UN*:
"(!!i. i ∈ I ==> F ∈ (A i) leadsTo B) ==> F ∈ ($\bigcup$i ∈ I. A i) leadsTo B"
⟨proof⟩

Binary union introduction rule

lemma *leadsTo_Un*:
"[| F ∈ A leadsTo C; F ∈ B leadsTo C |] ==> F ∈ (A ∪ B) leadsTo C"
⟨proof⟩

lemma *single_leadsTo_I*:
"(!!x. x ∈ A ==> F ∈ {x} leadsTo B) ==> F ∈ A leadsTo B"

⟨*proof*⟩

The INDUCTION rule as we should have liked to state it

```
lemma leadsTo_induct:
  "[| F ∈ za leadsTo zb;
      !!A B. F ∈ A ensures B ==> P A B;
      !!A B C. [| F ∈ A leadsTo B; P A B; F ∈ B leadsTo C; P B C |]
              ==> P A C;
      !!B S. ∀A ∈ S. F ∈ A leadsTo B & P A B ==> P (Union S) B
   |] ==> P za zb"
```
⟨*proof*⟩

```
lemma subset_imp_ensures: "A ⊆ B ==> F ∈ A ensures B"
```
⟨*proof*⟩

```
lemmas subset_imp_leadsTo = subset_imp_ensures [THEN leadsTo_Basis, standard]
```

```
lemmas leadsTo_refl = subset_refl [THEN subset_imp_leadsTo, standard]
```

```
lemmas empty_leadsTo = empty_subsetI [THEN subset_imp_leadsTo, standard,
simp]
```

```
lemmas leadsTo_UNIV = subset_UNIV [THEN subset_imp_leadsTo, standard, simp]
```

Lemma is the weak version: can't see how to do it in one step

```
lemma leadsTo_induct_pre_lemma:
  "[| F ∈ za leadsTo zb;
      P zb;
      !!A B. [| F ∈ A ensures B;  P B |] ==> P A;
      !!S. ∀A ∈ S. P A ==> P (Union S)
   |] ==> P za"
```
⟨*proof*⟩

```
lemma leadsTo_induct_pre:
  "[| F ∈ za leadsTo zb;
      P zb;
      !!A B. [| F ∈ A ensures B;  F ∈ B leadsTo zb;  P B |] ==> P A;
      !!S. ∀A ∈ S. F ∈ A leadsTo zb & P A ==> P (Union S)
   |] ==> P za"
```
⟨*proof*⟩

```
lemma leadsTo_weaken_R: "[| F ∈ A leadsTo A'; A'<=B' |] ==> F ∈ A leadsTo
B'"
```
⟨*proof*⟩

```
lemma leadsTo_weaken_L [rule_format]:
     "[| F ∈ A leadsTo A'; B ⊆ A |] ==> F ∈ B leadsTo A'"
```
⟨*proof*⟩

Distributes over binary unions

```
lemma leadsTo_Un_distrib:
     "F ∈ (A ∪ B) leadsTo C  =  (F ∈ A leadsTo C & F ∈ B leadsTo C)"
```

⟨proof⟩

lemma *leadsTo_UN_distrib:*
"F ∈ (⋃i ∈ I. A i) leadsTo B = (∀i ∈ I. F ∈ (A i) leadsTo B)"
⟨proof⟩

lemma *leadsTo_Union_distrib:*
"F ∈ (Union S) leadsTo B = (∀A ∈ S. F ∈ A leadsTo B)"
⟨proof⟩

lemma *leadsTo_weaken:*
"[| F ∈ A leadsTo A'; B ⊆ A; A'<=B' |] ==> F ∈ B leadsTo B'"
⟨proof⟩

Set difference: maybe combine with `leadsTo_weaken_L`??

lemma *leadsTo_Diff:*
"[| F ∈ (A-B) leadsTo C; F ∈ B leadsTo C |] ==> F ∈ A leadsTo C"
⟨proof⟩

lemma *leadsTo_UN_UN:*
"(!! i. i ∈ I ==> F ∈ (A i) leadsTo (A' i))
==> F ∈ (⋃i ∈ I. A i) leadsTo (⋃i ∈ I. A' i)"
⟨proof⟩

Binary union version

lemma *leadsTo_Un_Un:*
"[| F ∈ A leadsTo A'; F ∈ B leadsTo B' |]
==> F ∈ (A ∪ B) leadsTo (A' ∪ B')"
⟨proof⟩

lemma *leadsTo_cancel2:*
"[| F ∈ A leadsTo (A' ∪ B); F ∈ B leadsTo B' |]
==> F ∈ A leadsTo (A' ∪ B')"
⟨proof⟩

lemma *leadsTo_cancel_Diff2:*
"[| F ∈ A leadsTo (A' ∪ B); F ∈ (B-A') leadsTo B' |]
==> F ∈ A leadsTo (A' ∪ B')"
⟨proof⟩

lemma *leadsTo_cancel1:*
"[| F ∈ A leadsTo (B ∪ A'); F ∈ B leadsTo B' |]
==> F ∈ A leadsTo (B' ∪ A')"
⟨proof⟩

lemma *leadsTo_cancel_Diff1:*
"[| F ∈ A leadsTo (B ∪ A'); F ∈ (B-A') leadsTo B' |]
==> F ∈ A leadsTo (B' ∪ A')"
⟨proof⟩

The impossibility law

lemma *leadsTo_empty: "[|F ∈ A leadsTo {}; all_total F|] ==> A={}"*
⟨*proof*⟩

3.4 PSP: Progress-Safety-Progress

Special case of PSP: Misra's "stable conjunction"

```
lemma psp_stable:
   "[| F ∈ A leadsTo A'; F ∈ stable B |]
    ==> F ∈ (A ∩ B) leadsTo (A' ∩ B)"
```
⟨*proof*⟩

```
lemma psp_stable2:
   "[| F ∈ A leadsTo A'; F ∈ stable B |] ==> F ∈ (B ∩ A) leadsTo (B ∩ A')"
```
⟨*proof*⟩

```
lemma psp_ensures:
   "[| F ∈ A ensures A'; F ∈ B co B' |]
    ==> F ∈ (A ∩ B') ensures ((A' ∩ B) ∪ (B' - B))"
```
⟨*proof*⟩

```
lemma psp:
     "[| F ∈ A leadsTo A'; F ∈ B co B' |]
      ==> F ∈ (A ∩ B') leadsTo ((A' ∩ B) ∪ (B' - B))"
```
⟨*proof*⟩

```
lemma psp2:
     "[| F ∈ A leadsTo A'; F ∈ B co B' |]
    ==> F ∈ (B' ∩ A) leadsTo ((B ∩ A') ∪ (B' - B))"
```
⟨*proof*⟩

```
lemma psp_unless:
   "[| F ∈ A leadsTo A';  F ∈ B unless B' |]
    ==> F ∈ (A ∩ B) leadsTo ((A' ∩ B) ∪ B')"
```

⟨*proof*⟩

3.5 Proving the induction rules

```
lemma leadsTo_wf_induct_lemma:
     "[| wf r;
         ∀m. F ∈ (A ∩ f-`{m}) leadsTo
                    ((A ∩ f-`(r^-1 `` {m})) ∪ B) |]
      ==> F ∈ (A ∩ f-`{m}) leadsTo B"
```
⟨*proof*⟩

```
lemma leadsTo_wf_induct:
     "[| wf r;
         ∀m. F ∈ (A ∩ f-`{m}) leadsTo
                    ((A ∩ f-`(r^-1 `` {m})) ∪ B) |]
      ==> F ∈ A leadsTo B"
```

⟨proof⟩

```
lemma bounded_induct:
     "[| wf r;
         ∀m ∈ I. F ∈ (A ∩ f-`{m}) leadsTo
                      ((A ∩ f-`(r^-1 `` {m})) ∪ B) |]
      ==> F ∈ A leadsTo ((A - (f-`I)) ∪ B)"
```
⟨proof⟩

```
lemma lessThan_induct:
     "[| !!m::nat. F ∈ (A ∩ f-`{m}) leadsTo ((A ∩ f-`{..<m}) ∪ B) |]
      ==> F ∈ A leadsTo B"
```
⟨proof⟩

```
lemma lessThan_bounded_induct:
     "!!l::nat. [| ∀m ∈ greaterThan l.
            F ∈ (A ∩ f-`{m}) leadsTo ((A ∩ f-`(lessThan m)) ∪ B) |]
      ==> F ∈ A leadsTo ((A ∩ (f-`(atMost l))) ∪ B)"
```
⟨proof⟩

```
lemma greaterThan_bounded_induct:
     "(!!l::nat. ∀m ∈ lessThan l.
                 F ∈ (A ∩ f-`{m}) leadsTo ((A ∩ f-`(greaterThan m)) ∪ B))
      ==> F ∈ A leadsTo ((A ∩ (f-`(atLeast l))) ∪ B)"
```
⟨proof⟩

3.6 wlt

Misra's property W3

```
lemma wlt_leadsTo: "F ∈ (wlt F B) leadsTo B"
```
⟨proof⟩

```
lemma leadsTo_subset: "F ∈ A leadsTo B ==> A ⊆ wlt F B"
```
⟨proof⟩

Misra's property W2

```
lemma leadsTo_eq_subset_wlt: "F ∈ A leadsTo B = (A ⊆ wlt F B)"
```
⟨proof⟩

Misra's property W4

```
lemma wlt_increasing: "B ⊆ wlt F B"
```
⟨proof⟩

Used in the Trans case below

```
lemma lemma1:
   "[| B ⊆ A2;
       F ∈ (A1 - B) co (A1 ∪ B);
       F ∈ (A2 - C) co (A2 ∪ C) |]
    ==> F ∈ (A1 ∪ A2 - C) co (A1 ∪ A2 ∪ C)"
```

⟨proof⟩

Lemma (1,2,3) of Misra's draft book, Chapter 4, "Progress"

```
lemma leadsTo_123:
     "F ∈ A leadsTo A'
      ==> ∃B. A ⊆ B & F ∈ B leadsTo A' & F ∈ (B-A') co (B ∪ A')"
⟨proof⟩
```

Misra's property W5

```
lemma wlt_constrains_wlt: "F ∈ (wlt F B - B) co (wlt F B)"
⟨proof⟩
```

3.7 Completion: Binary and General Finite versions

```
lemma completion_lemma :
     "[| W = wlt F (B' ∪ C);
       F ∈ A leadsTo (A' ∪ C);  F ∈ A' co (A' ∪ C);
       F ∈ B leadsTo (B' ∪ C);  F ∈ B' co (B' ∪ C) |]
    ==> F ∈ (A ∩ B) leadsTo ((A' ∩ B') ∪ C)"
⟨proof⟩

lemmas completion = completion_lemma [OF refl]

lemma finite_completion_lemma:
     "finite I ==> (∀i ∈ I. F ∈ (A i) leadsTo (A' i ∪ C)) -->
                   (∀i ∈ I. F ∈ (A' i) co (A' i ∪ C)) -->
                   F ∈ (⋂i ∈ I. A i) leadsTo ((⋂i ∈ I. A' i) ∪ C)"
⟨proof⟩

lemma finite_completion:
     "[| finite I;
         !!i. i ∈ I ==> F ∈ (A i) leadsTo (A' i ∪ C);
         !!i. i ∈ I ==> F ∈ (A' i) co (A' i ∪ C) |]
      ==> F ∈ (⋂i ∈ I. A i) leadsTo ((⋂i ∈ I. A' i) ∪ C)"
⟨proof⟩

lemma stable_completion:
     "[| F ∈ A leadsTo A';  F ∈ stable A';
         F ∈ B leadsTo B';  F ∈ stable B' |]
    ==> F ∈ (A ∩ B) leadsTo (A' ∩ B')"
⟨proof⟩

lemma finite_stable_completion:
     "[| finite I;
         !!i. i ∈ I ==> F ∈ (A i) leadsTo (A' i);
         !!i. i ∈ I ==> F ∈ stable (A' i) |]
      ==> F ∈ (⋂i ∈ I. A i) leadsTo (⋂i ∈ I. A' i)"
⟨proof⟩

end
```

4 Weak Safety

theory *Constrains* **imports** *UNITY* **begin**

inductive_set
  *traces :: "['a set, ('a * 'a)set set] => ('a * 'a list) set"*
  for *init :: "'a set"* **and** *acts :: "('a * 'a)set set"*
  where

    Init: "s ∈ init ==> (s,[]) ∈ traces init acts"

  *| Acts: "[| act: acts; (s,evs) ∈ traces init acts; (s,s'): act |]
          ==> (s', s#evs) ∈ traces init acts"*

inductive_set
  reachable :: "'a program => 'a set"
  for *F :: "'a program"*
  where
    Init: "s ∈ Init F ==> s ∈ reachable F"

  *| Acts: "[| act: Acts F; s ∈ reachable F; (s,s'): act |]
          ==> s' ∈ reachable F"*

constdefs
  Constrains :: "['a set, 'a set] => 'a program set" (**infixl** *"Co" 60)*
    "A Co B == {F. F ∈ (reachable F ∩ A) co B}"

  Unless :: "['a set, 'a set] => 'a program set" (**infixl** *"Unless" 60)*
    "A Unless B == (A-B) Co (A ∪ B)"

  Stable :: "'a set => 'a program set"
    "Stable A == A Co A"

  Always :: "'a set => 'a program set"
    "Always A == {F. Init F ⊆ A} ∩ Stable A"

  Increasing :: "['a => 'b::{order}] => 'a program set"
    "Increasing f == ⋂z. Stable {s. z ≤ f s}"

4.1 traces and reachable

lemma *reachable_equiv_traces:*
    "reachable F = {s. ∃evs. (s,evs) ∈ traces (Init F) (Acts F)}"
⟨*proof*⟩

lemma *Init_subset_reachable: "Init F ⊆ reachable F"*
⟨*proof*⟩

lemma *stable_reachable [intro!,simp]:*

```
    "Acts G ⊆ Acts F ==> G ∈ stable (reachable F)"
```
⟨proof⟩

lemma *invariant_reachable: "F ∈ invariant (reachable F)"*
⟨proof⟩

lemma *invariant_includes_reachable: "F ∈ invariant A ==> reachable F ⊆ A"*
⟨proof⟩

4.2 Co

lemmas *constrains_reachable_Int =*
```
    subset_refl [THEN stable_reachable [unfolded stable_def],
                 THEN constrains_Int, standard]
```

lemma *Constrains_eq_constrains:*
```
    "A Co B = {F. F ∈ (reachable F  ∩  A) co (reachable F  ∩  B)}"
```
⟨proof⟩

lemma *constrains_imp_Constrains: "F ∈ A co A' ==> F ∈ A Co A'"*
⟨proof⟩

lemma *stable_imp_Stable: "F ∈ stable A ==> F ∈ Stable A"*
⟨proof⟩

lemma *ConstrainsI:*
```
    "(!!act s s'. [| act: Acts F;  (s,s') ∈ act;  s ∈ A |] ==> s': A')
     ==> F ∈ A Co A'"
```
⟨proof⟩

lemma *Constrains_empty [iff]: "F ∈ {} Co B"*
⟨proof⟩

lemma *Constrains_UNIV [iff]: "F ∈ A Co UNIV"*
⟨proof⟩

lemma *Constrains_weaken_R:*
```
    "[| F ∈ A Co A'; A'<=B' |] ==> F ∈ A Co B'"
```
⟨proof⟩

lemma *Constrains_weaken_L:*
```
    "[| F ∈ A Co A'; B ⊆ A |] ==> F ∈ B Co A'"
```
⟨proof⟩

lemma *Constrains_weaken:*
```
   "[| F ∈ A Co A'; B ⊆ A; A'<=B' |] ==> F ∈ B Co B'"
```
⟨proof⟩

lemma *Constrains_Un:*

"[| F ∈ A Co A'; F ∈ B Co B' |] ==> F ∈ (A ∪ B) Co (A' ∪ B')"
⟨proof⟩

lemma *Constrains_UN:*
assumes *Co: "!!i. i ∈ I ==> F ∈ (A i) Co (A' i)"*
shows *"F ∈ (⋃i ∈ I. A i) Co (⋃i ∈ I. A' i)"*
⟨proof⟩

lemma *Constrains_Int:*
"[| F ∈ A Co A'; F ∈ B Co B' |] ==> F ∈ (A ∩ B) Co (A' ∩ B')"
⟨proof⟩

lemma *Constrains_INT:*
assumes *Co: "!!i. i ∈ I ==> F ∈ (A i) Co (A' i)"*
shows *"F ∈ (⋂i ∈ I. A i) Co (⋂i ∈ I. A' i)"*
⟨proof⟩

lemma *Constrains_imp_subset: "F ∈ A Co A' ==> reachable F ∩ A ⊆ A'"*
⟨proof⟩

lemma *Constrains_trans: "[| F ∈ A Co B; F ∈ B Co C |] ==> F ∈ A Co C"*
⟨proof⟩

lemma *Constrains_cancel:*
"[| F ∈ A Co (A' ∪ B); F ∈ B Co B' |] ==> F ∈ A Co (A' ∪ B')"
⟨proof⟩

4.3 Stable

lemma *Stable_eq: "[| F ∈ Stable A; A = B |] ==> F ∈ Stable B"*
⟨proof⟩

lemma *Stable_eq_stable: "(F ∈ Stable A) = (F ∈ stable (reachable F ∩ A))"*
⟨proof⟩

lemma *StableI: "F ∈ A Co A ==> F ∈ Stable A"*
⟨proof⟩

lemma *StableD: "F ∈ Stable A ==> F ∈ A Co A"*
⟨proof⟩

lemma *Stable_Un:*
"[| F ∈ Stable A; F ∈ Stable A' |] ==> F ∈ Stable (A ∪ A')"
⟨proof⟩

lemma *Stable_Int:*
"[| F ∈ Stable A; F ∈ Stable A' |] ==> F ∈ Stable (A ∩ A')"
⟨proof⟩

lemma *Stable_Constrains_Un:*
"[| F ∈ Stable C; F ∈ A Co (C ∪ A') |]
==> F ∈ (C ∪ A) Co (C ∪ A')"

⟨*proof*⟩

lemma *Stable_Constrains_Int:*
```
"[| F ∈ Stable C; F ∈ (C ∩ A) Co A' |]
 ==> F ∈ (C ∩ A) Co (C ∩ A')"
```
⟨*proof*⟩

lemma *Stable_UN:*
```
"(!!i. i ∈ I ==> F ∈ Stable (A i)) ==> F ∈ Stable (⋃i ∈ I. A i)"
```
⟨*proof*⟩

lemma *Stable_INT:*
```
"(!!i. i ∈ I ==> F ∈ Stable (A i)) ==> F ∈ Stable (⋂i ∈ I. A i)"
```
⟨*proof*⟩

lemma *Stable_reachable: "F ∈ Stable (reachable F)"*
⟨*proof*⟩

4.4 Increasing

lemma *IncreasingD:*
```
"F ∈ Increasing f ==> F ∈ Stable {s. x ≤ f s}"
```
⟨*proof*⟩

lemma *mono_Increasing_o:*
```
"mono g ==> Increasing f ⊆ Increasing (g o f)"
```
⟨*proof*⟩

lemma *strict_IncreasingD:*
```
"!!z::nat. F ∈ Increasing f ==> F ∈ Stable {s. z < f s}"
```
⟨*proof*⟩

lemma *increasing_imp_Increasing:*
```
"F ∈ increasing f ==> F ∈ Increasing f"
```
⟨*proof*⟩

lemmas *Increasing_constant =*
```
increasing_constant [THEN increasing_imp_Increasing, standard, iff]
```

4.5 The Elimination Theorem

lemma *Elimination:*
```
"[| ∀m. F ∈ {s. s x = m} Co (B m) |]
 ==> F ∈ {s. s x ∈ M} Co (⋃m ∈ M. B m)"
```
⟨*proof*⟩

lemma *Elimination_sing:*
```
"(∀m. F ∈ {m} Co (B m)) ==> F ∈ M Co (⋃m ∈ M. B m)"
```
⟨*proof*⟩

4.6 Specialized laws for handling Always

lemma *AlwaysI: "[| Init F ⊆ A; F ∈ Stable A |] ==> F ∈ Always A"*

⟨proof⟩

lemma *AlwaysD: "F ∈ Always A ==> Init F ⊆ A & F ∈ Stable A"*
⟨proof⟩

lemmas *AlwaysE = AlwaysD [THEN conjE, standard]*
lemmas *Always_imp_Stable = AlwaysD [THEN conjunct2, standard]*

lemma *Always_includes_reachable: "F ∈ Always A ==> reachable F ⊆ A"*
⟨proof⟩

lemma *invariant_imp_Always:*
"F ∈ invariant A ==> F ∈ Always A"
⟨proof⟩

lemmas *Always_reachable =*
invariant_reachable [THEN invariant_imp_Always, standard]

lemma *Always_eq_invariant_reachable:*
"Always A = {F. F ∈ invariant (reachable F ∩ A)}"
⟨proof⟩

lemma *Always_eq_includes_reachable: "Always A = {F. reachable F ⊆ A}"*
⟨proof⟩

lemma *Always_UNIV_eq [simp]: "Always UNIV = UNIV"*
⟨proof⟩

lemma *UNIV_AlwaysI: "UNIV ⊆ A ==> F ∈ Always A"*
⟨proof⟩

lemma *Always_eq_UN_invariant: "Always A = (⋃ I ∈ Pow A. invariant I)"*
⟨proof⟩

lemma *Always_weaken: "[| F ∈ Always A; A ⊆ B |] ==> F ∈ Always B"*
⟨proof⟩

4.7 "Co" rules involving Always

lemma *Always_Constrains_pre:*
"F ∈ Always INV ==> (F ∈ (INV ∩ A) Co A') = (F ∈ A Co A')"
⟨proof⟩

lemma *Always_Constrains_post:*
"F ∈ Always INV ==> (F ∈ A Co (INV ∩ A')) = (F ∈ A Co A')"
⟨proof⟩

lemmas *Always_ConstrainsI = Always_Constrains_pre [THEN iffD1, standard]*

lemmas *Always_ConstrainsD = Always_Constrains_post [THEN iffD2, standard]*

lemma *Always_Constrains_weaken:*
```
     "[| F ∈ Always C;  F ∈ A Co A';
         C ∩ B ⊆ A;   C ∩ A' ⊆ B' |]
      ==> F ∈ B Co B'"
```
⟨*proof*⟩

lemma *Always_Int_distrib: "Always (A ∩ B) = Always A ∩ Always B"*
⟨*proof*⟩

lemma *Always_INT_distrib: "Always (INTER I A) = (⋂i ∈ I. Always (A i))"*
⟨*proof*⟩

lemma *Always_Int_I:*
```
     "[| F ∈ Always A;  F ∈ Always B |] ==> F ∈ Always (A ∩ B)"
```
⟨*proof*⟩

lemma *Always_Compl_Un_eq:*
```
     "F ∈ Always A ==> (F ∈ Always (-A ∪ B)) = (F ∈ Always B)"
```
⟨*proof*⟩

lemmas *Always_thin = thin_rl [of "F ∈ Always A", standard]*

4.8 Totalize

lemma *reachable_imp_reachable_tot:*
```
      "s ∈ reachable F ==> s ∈ reachable (totalize F)"
```
⟨*proof*⟩

lemma *reachable_tot_imp_reachable:*
```
      "s ∈ reachable (totalize F) ==> s ∈ reachable F"
```
⟨*proof*⟩

lemma *reachable_tot_eq [simp]: "reachable (totalize F) = reachable F"*
⟨*proof*⟩

lemma *totalize_Constrains_iff [simp]: "(totalize F ∈ A Co B) = (F ∈ A Co B)"*
⟨*proof*⟩

lemma *totalize_Stable_iff [simp]: "(totalize F ∈ Stable A) = (F ∈ Stable A)"*
⟨*proof*⟩

lemma *totalize_Always_iff [simp]: "(totalize F ∈ Always A) = (F ∈ Always A)"*
⟨*proof*⟩

end

5 Weak Progress

theory *SubstAx* **imports** *WFair Constrains* **begin**

constdefs
```
  Ensures :: "['a set, 'a set] => 'a program set"    (infixl "Ensures" 60)
   "A Ensures B == {F. F ∈ (reachable F ∩ A) ensures B}"

  LeadsTo :: "['a set, 'a set] => 'a program set"    (infixl "LeadsTo" 60)
   "A LeadsTo B == {F. F ∈ (reachable F ∩ A) leadsTo B}"
```

syntax *(xsymbols)*
```
 "op LeadsTo" :: "['a set, 'a set] => 'a program set" (infixl " ⟼w " 60)
```

Resembles the previous definition of LeadsTo

lemma *LeadsTo_eq_leadsTo:*
```
     "A LeadsTo B = {F. F ∈ (reachable F ∩ A) leadsTo (reachable F ∩ B)}"
```
⟨*proof*⟩

5.1 Specialized laws for handling invariants

lemma *Always_LeadsTo_pre:*
```
     "F ∈ Always INV ==> (F ∈ (INV ∩ A) LeadsTo A') = (F ∈ A LeadsTo A')"
```
⟨*proof*⟩

lemma *Always_LeadsTo_post:*
```
     "F ∈ Always INV ==> (F ∈ A LeadsTo (INV ∩ A')) = (F ∈ A LeadsTo A')"
```
⟨*proof*⟩

lemmas *Always_LeadsToI = Always_LeadsTo_pre [THEN iffD1, standard]*

lemmas *Always_LeadsToD = Always_LeadsTo_post [THEN iffD2, standard]*

5.2 Introduction rules: Basis, Trans, Union

lemma *leadsTo_imp_LeadsTo: "F ∈ A leadsTo B ==> F ∈ A LeadsTo B"*
⟨*proof*⟩

lemma *LeadsTo_Trans:*
```
     "[| F ∈ A LeadsTo B;  F ∈ B LeadsTo C |] ==> F ∈ A LeadsTo C"
```
⟨*proof*⟩

lemma *LeadsTo_Union:*
```
     "(!!A. A ∈ S ==> F ∈ A LeadsTo B) ==> F ∈ (Union S) LeadsTo B"
```
⟨*proof*⟩

5.3 Derived rules

lemma *LeadsTo_UNIV [simp]: "F ∈ A LeadsTo UNIV"*
⟨*proof*⟩

Useful with cancellation, disjunction

lemma *LeadsTo_Un_duplicate:*
```
"F ∈ A LeadsTo (A' ∪ A') ==> F ∈ A LeadsTo A'"
```
⟨*proof*⟩

lemma *LeadsTo_Un_duplicate2:*
```
"F ∈ A LeadsTo (A' ∪ C ∪ C) ==> F ∈ A LeadsTo (A' ∪ C)"
```
⟨*proof*⟩

lemma *LeadsTo_UN:*
```
"(!!i. i ∈ I ==> F ∈ (A i) LeadsTo B) ==> F ∈ (⋃i ∈ I. A i) LeadsTo
B"
```
⟨*proof*⟩

Binary union introduction rule

lemma *LeadsTo_Un:*
```
"[| F ∈ A LeadsTo C; F ∈ B LeadsTo C |] ==> F ∈ (A ∪ B) LeadsTo C"
```
⟨*proof*⟩

Lets us look at the starting state

lemma *single_LeadsTo_I:*
```
"(!!s. s ∈ A ==> F ∈ {s} LeadsTo B) ==> F ∈ A LeadsTo B"
```
⟨*proof*⟩

lemma *subset_imp_LeadsTo: "A ⊆ B ==> F ∈ A LeadsTo B"*
⟨*proof*⟩

lemmas *empty_LeadsTo = empty_subsetI [THEN subset_imp_LeadsTo, standard, simp]*

lemma *LeadsTo_weaken_R [rule_format]:*
```
"[| F ∈ A LeadsTo A';  A' ⊆ B' |] ==> F ∈ A LeadsTo B'"
```
⟨*proof*⟩

lemma *LeadsTo_weaken_L [rule_format]:*
```
"[| F ∈ A LeadsTo A';  B ⊆ A |]
 ==> F ∈ B LeadsTo A'"
```
⟨*proof*⟩

lemma *LeadsTo_weaken:*
```
"[| F ∈ A LeadsTo A';
    B  ⊆ A;   A' ⊆ B' |]
 ==> F ∈ B LeadsTo B'"
```
⟨*proof*⟩

lemma *Always_LeadsTo_weaken:*
```
"[| F ∈ Always C;  F ∈ A LeadsTo A';
    C ∩ B ⊆ A;   C ∩ A' ⊆ B' |]
 ==> F ∈ B LeadsTo B'"
```

⟨proof⟩

lemma *LeadsTo_Un_post:* "F ∈ A LeadsTo B ==> F ∈ (A ∪ B) LeadsTo B"
⟨proof⟩

lemma *LeadsTo_Trans_Un:*
```
"[| F ∈ A LeadsTo B;  F ∈ B LeadsTo C |]
 ==> F ∈ (A ∪ B) LeadsTo C"
```
⟨proof⟩

lemma *LeadsTo_Un_distrib:*
```
"(F ∈ (A ∪ B) LeadsTo C)  = (F ∈ A LeadsTo C & F ∈ B LeadsTo C)"
```
⟨proof⟩

lemma *LeadsTo_UN_distrib:*
```
"(F ∈ (⋃i ∈ I. A i) LeadsTo B)  =  (∀i ∈ I. F ∈ (A i) LeadsTo B)"
```
⟨proof⟩

lemma *LeadsTo_Union_distrib:*
```
"(F ∈ (Union S) LeadsTo B)  =  (∀A ∈ S. F ∈ A LeadsTo B)"
```
⟨proof⟩

lemma *LeadsTo_Basis:* "F ∈ A Ensures B ==> F ∈ A LeadsTo B"
⟨proof⟩

lemma *EnsuresI:*
```
"[| F ∈ (A-B) Co (A ∪ B);  F ∈ transient (A-B) |]
 ==> F ∈ A Ensures B"
```
⟨proof⟩

lemma *Always_LeadsTo_Basis:*
```
"[| F ∈ Always INV;
    F ∈ (INV ∩ (A-A')) Co (A ∪ A');
    F ∈ transient (INV ∩ (A-A')) |]
==> F ∈ A LeadsTo A'"
```
⟨proof⟩

Set difference: maybe combine with `leadsTo_weaken_L`?? This is the most useful form of the "disjunction" rule

lemma *LeadsTo_Diff:*
```
"[| F ∈ (A-B) LeadsTo C;  F ∈ (A ∩ B) LeadsTo C |]
 ==> F ∈ A LeadsTo C"
```
⟨proof⟩

lemma *LeadsTo_UN_UN:*

```
"(!! i. i ∈ I ==> F ∈ (A i) LeadsTo (A' i))
 ==> F ∈ (⋃i ∈ I. A i) LeadsTo (⋃i ∈ I. A' i)"
```
⟨proof⟩

Version with no index set

lemma *LeadsTo_UN_UN_noindex:*
```
"(!!i. F ∈ (A i) LeadsTo (A' i)) ==> F ∈ (⋃i. A i) LeadsTo (⋃i. A'
i)"
```
⟨proof⟩

Version with no index set

lemma *all_LeadsTo_UN_UN:*
```
"∀i. F ∈ (A i) LeadsTo (A' i)
 ==> F ∈ (⋃i. A i) LeadsTo (⋃i. A' i)"
```
⟨proof⟩

Binary union version

lemma *LeadsTo_Un_Un:*
```
"[| F ∈ A LeadsTo A'; F ∈ B LeadsTo B' |]
        ==> F ∈ (A ∪ B) LeadsTo (A' ∪ B')"
```
⟨proof⟩

lemma *LeadsTo_cancel2:*
```
"[| F ∈ A LeadsTo (A' ∪ B); F ∈ B LeadsTo B' |]
 ==> F ∈ A LeadsTo (A' ∪ B')"
```
⟨proof⟩

lemma *LeadsTo_cancel_Diff2:*
```
"[| F ∈ A LeadsTo (A' ∪ B); F ∈ (B-A') LeadsTo B' |]
 ==> F ∈ A LeadsTo (A' ∪ B')"
```
⟨proof⟩

lemma *LeadsTo_cancel1:*
```
"[| F ∈ A LeadsTo (B ∪ A'); F ∈ B LeadsTo B' |]
 ==> F ∈ A LeadsTo (B' ∪ A')"
```
⟨proof⟩

lemma *LeadsTo_cancel_Diff1:*
```
"[| F ∈ A LeadsTo (B ∪ A'); F ∈ (B-A') LeadsTo B' |]
 ==> F ∈ A LeadsTo (B' ∪ A')"
```
⟨proof⟩

The impossibility law

The set "A" may be non-empty, but it contains no reachable states

lemma *LeadsTo_empty:* `"[|F ∈ A LeadsTo {}; all_total F|] ==> F ∈ Always (-A)"`
⟨proof⟩

5.4 PSP: Progress-Safety-Progress

Special case of PSP: Misra's "stable conjunction"

lemma *PSP_Stable:*
```
"[| F ∈ A LeadsTo A';  F ∈ Stable B |]
 ==> F ∈ (A ∩ B) LeadsTo (A' ∩ B)"
```
⟨*proof*⟩

lemma *PSP_Stable2:*
```
"[| F ∈ A LeadsTo A'; F ∈ Stable B |]
 ==> F ∈ (B ∩ A) LeadsTo (B ∩ A')"
```
⟨*proof*⟩

lemma *PSP:*
```
"[| F ∈ A LeadsTo A'; F ∈ B Co B' |]
 ==> F ∈ (A ∩ B') LeadsTo ((A' ∩ B) ∪ (B' - B))"
```
⟨*proof*⟩

lemma *PSP2:*
```
"[| F ∈ A LeadsTo A'; F ∈ B Co B' |]
 ==> F ∈ (B' ∩ A) LeadsTo ((B ∩ A') ∪ (B' - B))"
```
⟨*proof*⟩

lemma *PSP_Unless:*
```
"[| F ∈ A LeadsTo A'; F ∈ B Unless B' |]
 ==> F ∈ (A ∩ B) LeadsTo ((A' ∩ B) ∪ B')"
```
⟨*proof*⟩

lemma *Stable_transient_Always_LeadsTo:*
```
"[| F ∈ Stable A;  F ∈ transient C;
    F ∈ Always (-A ∪ B ∪ C) |] ==> F ∈ A LeadsTo B"
```
⟨*proof*⟩

5.5 Induction rules

lemma *LeadsTo_wf_induct:*
```
"[| wf r;
    ∀m. F ∈ (A ∩ f-`{m}) LeadsTo
               ((A ∩ f-`(r^-1 `` {m})) ∪ B) |]
 ==> F ∈ A LeadsTo B"
```
⟨*proof*⟩

lemma *Bounded_induct:*
```
"[| wf r;
    ∀m ∈ I. F ∈ (A ∩ f-`{m}) LeadsTo
                 ((A ∩ f-`(r^-1 `` {m})) ∪ B) |]
 ==> F ∈ A LeadsTo ((A - (f-`I)) ∪ B)"
```
⟨*proof*⟩

lemma *LessThan_induct:*
```
"(!!m::nat. F ∈ (A ∩ f-`{m}) LeadsTo ((A ∩ f-`(lessThan m)) ∪ B))
 ==> F ∈ A LeadsTo B"
```
⟨*proof*⟩

Integer version. Could generalize from 0 to any lower bound

lemma *integ_0_le_induct:*
```
"[| F ∈ Always {s. (0::int) ≤ f s};
    !! z. F ∈ (A ∩ {s. f s = z}) LeadsTo
              ((A ∩ {s. f s < z}) ∪ B) |]
 ==> F ∈ A LeadsTo B"
```
⟨*proof*⟩

lemma *LessThan_bounded_induct:*
```
"!!l::nat. ∀m ∈ greaterThan l.
             F ∈ (A ∩ f-‘{m}) LeadsTo ((A ∩ f-‘(lessThan m)) ∪ B)
       ==> F ∈ A LeadsTo ((A ∩ (f-‘(atMost l))) ∪ B)"
```
⟨*proof*⟩

lemma *GreaterThan_bounded_induct:*
```
"!!l::nat. ∀m ∈ lessThan l.
           F ∈ (A ∩ f-‘{m}) LeadsTo ((A ∩ f-‘(greaterThan m)) ∪ B)
 ==> F ∈ A LeadsTo ((A ∩ (f-‘(atLeast l))) ∪ B)"
```
⟨*proof*⟩

5.6 Completion: Binary and General Finite versions

lemma *Completion:*
```
"[| F ∈ A LeadsTo (A' ∪ C);  F ∈ A' Co (A' ∪ C);
    F ∈ B LeadsTo (B' ∪ C);  F ∈ B' Co (B' ∪ C) |]
 ==> F ∈ (A ∩ B) LeadsTo ((A' ∩ B') ∪ C)"
```
⟨*proof*⟩

lemma *Finite_completion_lemma:*
```
"finite I
 ==> (∀i ∈ I. F ∈ (A i) LeadsTo (A' i ∪ C)) -->
     (∀i ∈ I. F ∈ (A' i) Co (A' i ∪ C)) -->
     F ∈ (⋂i ∈ I. A i) LeadsTo ((⋂i ∈ I. A' i) ∪ C)"
```
⟨*proof*⟩

lemma *Finite_completion:*
```
"[| finite I;
    !!i. i ∈ I ==> F ∈ (A i) LeadsTo (A' i ∪ C);
    !!i. i ∈ I ==> F ∈ (A' i) Co (A' i ∪ C) |]
 ==> F ∈ (⋂i ∈ I. A i) LeadsTo ((⋂i ∈ I. A' i) ∪ C)"
```
⟨*proof*⟩

lemma *Stable_completion:*
```
"[| F ∈ A LeadsTo A';  F ∈ Stable A';
    F ∈ B LeadsTo B';  F ∈ Stable B' |]
 ==> F ∈ (A ∩ B) LeadsTo (A' ∩ B')"
```
⟨*proof*⟩

lemma *Finite_stable_completion:*
```
"[| finite I;
    !!i. i ∈ I ==> F ∈ (A i) LeadsTo (A' i);
    !!i. i ∈ I ==> F ∈ Stable (A' i) |]
 ==> F ∈ (⋂i ∈ I. A i) LeadsTo (⋂i ∈ I. A' i)"
```
⟨*proof*⟩

end

6 The Detects Relation

```
theory Detects imports FP SubstAx begin

consts
   op_Detects  :: "['a set, 'a set] => 'a program set"  (infixl "Detects" 60)
   op_Equality :: "['a set, 'a set] => 'a set"          (infixl "<==>" 60)

defs
  Detects_def:  "A Detects B == (Always (-A ∪ B)) ∩ (B LeadsTo A)"
  Equality_def: "A <==> B == (-A ∪ B) ∩ (A ∪ -B)"


lemma Always_at_FP:
     "[|F ∈ A LeadsTo B; all_total F|] ==> F ∈ Always (-((FP F) ∩ A ∩ -B))"
⟨proof⟩


lemma Detects_Trans:
     "[| F ∈ A Detects B; F ∈ B Detects C |] ==> F ∈ A Detects C"
⟨proof⟩

lemma Detects_refl: "F ∈ A Detects A"
⟨proof⟩

lemma Detects_eq_Un: "(A<==>B) = (A ∩ B) ∪ (-A ∩ -B)"
⟨proof⟩


lemma Detects_antisym:
     "[| F ∈ A Detects B;  F ∈ B Detects A|] ==> F ∈ Always (A <==> B)"
⟨proof⟩


lemma Detects_Always:
     "[|F ∈ A Detects B; all_total F|] ==> F ∈ Always (-(FP F) ∪ (A <==>
B))"
⟨proof⟩


lemma Detects_Imp_LeadstoEQ:
     "F ∈ A Detects B ==> F ∈ UNIV LeadsTo (A <==> B)"
⟨proof⟩
```

end

7 Unions of Programs

theory *Union* **imports** *SubstAx FP* **begin**

constdefs

```
  ok :: "['a program, 'a program] => bool"      (infixl "ok" 65)
    "F ok G == Acts F ⊆ AllowedActs G &
               Acts G ⊆ AllowedActs F"


  OK  :: "['a set, 'a => 'b program] => bool"
    "OK I F == (∀ i ∈ I. ∀ j ∈ I-{i}. Acts (F i) ⊆ AllowedActs (F j))"

  JOIN  :: "['a set, 'a => 'b program] => 'b program"
    "JOIN I F == mk_program (⋂ i ∈ I. Init (F i), ⋃ i ∈ I. Acts (F i),
                             ⋂ i ∈ I. AllowedActs (F i))"

  Join :: "['a program, 'a program] => 'a program"      (infixl "Join" 65)
    "F Join G == mk_program (Init F ∩ Init G, Acts F ∪ Acts G,
                             AllowedActs F ∩ AllowedActs G)"

  SKIP :: "'a program"
    "SKIP == mk_program (UNIV, {}, UNIV)"


  safety_prop :: "'a program set => bool"
    "safety_prop X == SKIP: X & (∀ G. Acts G ⊆ UNION X Acts --> G ∈ X)"
```

syntax

```
  "@JOIN1"     :: "[pttrns, 'b set] => 'b set"         ("(3JN _./ _)" 10)
  "@JOIN"      :: "[pttrn, 'a set, 'b set] => 'b set"  ("(3JN _:_./ _)" 10)
```

translations

```
  "JN x : A. B"   == "JOIN A (%x. B)"
  "JN x y. B"   == "JN x. JN y. B"
  "JN x. B"     == "JOIN UNIV (%x. B)"
```

syntax *(xsymbols)*

```
  SKIP     :: "'a program"                              ("⊥")
  Join     :: "['a program, 'a program] => 'a program"  (infixl "⊔" 65)
  "@JOIN1" :: "[pttrns, 'b set] => 'b set"              ("(3⨆ _./ _)" 10)
  "@JOIN"  :: "[pttrn, 'a set, 'b set] => 'b set"       ("(3⨆ _∈_./ _)" 10)
```

7.1 SKIP

lemma *Init_SKIP [simp]: "Init SKIP = UNIV"*

⟨*proof*⟩

lemma *Acts_SKIP [simp]: "Acts SKIP = {Id}"*
⟨*proof*⟩

lemma *AllowedActs_SKIP [simp]: "AllowedActs SKIP = UNIV"*
⟨*proof*⟩

lemma *reachable_SKIP [simp]: "reachable SKIP = UNIV"*
⟨*proof*⟩

7.2 SKIP and safety properties

lemma *SKIP_in_constrains_iff [iff]: "(SKIP ∈ A co B) = (A ⊆ B)"*
⟨*proof*⟩

lemma *SKIP_in_Constrains_iff [iff]: "(SKIP ∈ A Co B) = (A ⊆ B)"*
⟨*proof*⟩

lemma *SKIP_in_stable [iff]: "SKIP ∈ stable A"*
⟨*proof*⟩

declare *SKIP_in_stable [THEN stable_imp_Stable, iff]*

7.3 Join

lemma *Init_Join [simp]: "Init (F⊔G) = Init F ∩ Init G"*
⟨*proof*⟩

lemma *Acts_Join [simp]: "Acts (F⊔G) = Acts F ∪ Acts G"*
⟨*proof*⟩

lemma *AllowedActs_Join [simp]:*
"AllowedActs (F⊔G) = AllowedActs F ∩ AllowedActs G"
⟨*proof*⟩

7.4 JN

lemma *JN_empty [simp]: "(⨆i∈{}. F i) = SKIP"*
⟨*proof*⟩

lemma *JN_insert [simp]: "(⨆i ∈ insert a I. F i) = (F a)⊔(⨆i ∈ I. F i)"*
⟨*proof*⟩

lemma *Init_JN [simp]: "Init (⨆i ∈ I. F i) = (⋂i ∈ I. Init (F i))"*
⟨*proof*⟩

lemma *Acts_JN [simp]: "Acts (⨆i ∈ I. F i) = insert Id (⋃i ∈ I. Acts (F i))"*
⟨*proof*⟩

lemma *AllowedActs_JN [simp]:*
"AllowedActs (⨆i ∈ I. F i) = (⋂i ∈ I. AllowedActs (F i))"
⟨*proof*⟩

lemma *JN_cong [cong]:*
"[| I=J; !!i. i ∈ J ==> F i = G i |] ==> (⨆i ∈ I. F i) = (⨆i ∈ J. G i)"
⟨*proof*⟩

7.5 Algebraic laws

lemma *Join_commute: "F⊔G = G⊔F"*
⟨*proof*⟩

lemma *Join_assoc: "(F⊔G)⊔H = F⊔(G⊔H)"*
⟨*proof*⟩

lemma *Join_left_commute: "A⊔(B⊔C) = B⊔(A⊔C)"*
⟨*proof*⟩

lemma *Join_SKIP_left [simp]: "SKIP⊔F = F"*
⟨*proof*⟩

lemma *Join_SKIP_right [simp]: "F⊔SKIP = F"*
⟨*proof*⟩

lemma *Join_absorb [simp]: "F⊔F = F"*
⟨*proof*⟩

lemma *Join_left_absorb: "F⊔(F⊔G) = F⊔G"*
⟨*proof*⟩

lemmas *Join_ac = Join_assoc Join_left_absorb Join_commute Join_left_commute*

7.6 Laws Governing ⨆

lemma *JN_absorb: "k ∈ I ==> F k⊔(⨆i ∈ I. F i) = (⨆i ∈ I. F i)"*
⟨*proof*⟩

lemma *JN_Un: "(⨆i ∈ I ∪ J. F i) = ((⨆i ∈ I. F i)⊔(⨆i ∈ J. F i))"*
⟨*proof*⟩

lemma *JN_constant: "(⨆i ∈ I. c) = (if I={} then SKIP else c)"*
⟨*proof*⟩

lemma *JN_Join_distrib:*
"(⨆i ∈ I. F i⊔G i) = (⨆i ∈ I. F i) ⊔ (⨆i ∈ I. G i)"
⟨*proof*⟩

lemma *JN_Join_miniscope:*
"i ∈ I ==> (⨆i ∈ I. F i⊔G) = ((⨆i ∈ I. F i)⊔G)"
⟨*proof*⟩

lemma *JN_Join_diff: "i ∈ I ==> F i⊔JOIN (I - {i}) F = JOIN I F"*

⟨*proof*⟩

7.7 Safety: co, stable, FP

lemma *JN_constrains:*
    "i ∈ I ==> (⨆i ∈ I. F i) ∈ A co B = (∀i ∈ I. F i ∈ A co B)"
⟨*proof*⟩

lemma *Join_constrains [simp]:*
    "(F⊔G ∈ A co B) = (F ∈ A co B & G ∈ A co B)"
⟨*proof*⟩

lemma *Join_unless [simp]:*
    "(F⊔G ∈ A unless B) = (F ∈ A unless B & G ∈ A unless B)"
⟨*proof*⟩

lemma *Join_constrains_weaken:*
    "[| F ∈ A co A'; G ∈ B co B' |]
    ==> F⊔G ∈ (A ∩ B) co (A' ∪ B')"
⟨*proof*⟩

lemma *JN_constrains_weaken:*
    "[| ∀i ∈ I. F i ∈ A i co A' i; i ∈ I |]
    ==> (⨆i ∈ I. F i) ∈ (⋂i ∈ I. A i) co (⋃i ∈ I. A' i)"
⟨*proof*⟩

lemma *JN_stable: "(⨆i ∈ I. F i) ∈ stable A = (∀i ∈ I. F i ∈ stable A)"*
⟨*proof*⟩

lemma *invariant_JN_I:*
    "[| !!i. i ∈ I ==> F i ∈ invariant A; i ∈ I |]
    ==> (⨆i ∈ I. F i) ∈ invariant A"
⟨*proof*⟩

lemma *Join_stable [simp]:*
    "(F⊔G ∈ stable A) =
    (F ∈ stable A & G ∈ stable A)"
⟨*proof*⟩

lemma *Join_increasing [simp]:*
    "(F⊔G ∈ increasing f) =
    (F ∈ increasing f & G ∈ increasing f)"
⟨*proof*⟩

lemma *invariant_JoinI:*
    "[| F ∈ invariant A; G ∈ invariant A |]
    ==> F⊔G ∈ invariant A"
⟨*proof*⟩

lemma *FP_JN: "FP (⨆i ∈ I. F i) = (⋂i ∈ I. FP (F i))"*

⟨proof⟩

7.8 Progress: transient, ensures

lemma *JN_transient:*
```
     "i ∈ I ==>
    (⨆ i ∈ I. F i) ∈ transient A = (∃ i ∈ I. F i ∈ transient A)"
```
⟨proof⟩

lemma *Join_transient [simp]:*
```
     "F⊔G ∈ transient A =
      (F ∈ transient A | G ∈ transient A)"
```
⟨proof⟩

lemma *Join_transient_I1:* `"F ∈ transient A ==> F⊔G ∈ transient A"`
⟨proof⟩

lemma *Join_transient_I2:* `"G ∈ transient A ==> F⊔G ∈ transient A"`
⟨proof⟩

lemma *JN_ensures:*
```
     "i ∈ I ==>
      (⨆ i ∈ I. F i) ∈ A ensures B =
      ((∀ i ∈ I. F i ∈ (A-B) co (A ∪ B)) & (∃ i ∈ I. F i ∈ A ensures B))"
```
⟨proof⟩

lemma *Join_ensures:*
```
     "F⊔G ∈ A ensures B =
      (F ∈ (A-B) co (A ∪ B) & G ∈ (A-B) co (A ∪ B) &
       (F ∈ transient (A-B) | G ∈ transient (A-B)))"
```
⟨proof⟩

lemma *stable_Join_constrains:*
```
    "[| F ∈ stable A;  G ∈ A co A' |]
     ==> F⊔G ∈ A co A'"
```
⟨proof⟩

lemma *stable_Join_Always1:*
```
     "[| F ∈ stable A;  G ∈ invariant A |] ==> F⊔G ∈ Always A"
```
⟨proof⟩

lemma *stable_Join_Always2:*
```
     "[| F ∈ invariant A;  G ∈ stable A |] ==> F⊔G ∈ Always A"
```
⟨proof⟩

lemma *stable_Join_ensures1:*
```
     "[| F ∈ stable A;  G ∈ A ensures B |] ==> F⊔G ∈ A ensures B"
```
⟨proof⟩

lemma *stable_Join_ensures2:*

```
  "[| F ∈ A ensures B;  G ∈ stable A |] ==> F⊔G ∈ A ensures B"
```
⟨proof⟩

7.9 the ok and OK relations

lemma *ok_SKIP1 [iff]: "SKIP ok F"*
⟨proof⟩

lemma *ok_SKIP2 [iff]: "F ok SKIP"*
⟨proof⟩

lemma *ok_Join_commute:*
```
  "(F ok G & (F⊔G) ok H) = (G ok H & F ok (G⊔H))"
```
⟨proof⟩

lemma *ok_commute: "(F ok G) = (G ok F)"*
⟨proof⟩

lemmas *ok_sym = ok_commute [THEN iffD1, standard]*

lemma *ok_iff_OK:*
```
  "OK {(0::int,F),(1,G),(2,H)} snd = (F ok G & (F⊔G) ok H)"
```
⟨proof⟩

lemma *ok_Join_iff1 [iff]: "F ok (G⊔H) = (F ok G & F ok H)"*
⟨proof⟩

lemma *ok_Join_iff2 [iff]: "(G⊔H) ok F = (G ok F & H ok F)"*
⟨proof⟩

lemma *ok_Join_commute_I: "[| F ok G; (F⊔G) ok H |] ==> F ok (G⊔H)"*
⟨proof⟩

lemma *ok_JN_iff1 [iff]: "F ok (JOIN I G) = (∀i ∈ I. F ok G i)"*
⟨proof⟩

lemma *ok_JN_iff2 [iff]: "(JOIN I G) ok F = (∀i ∈ I. G i ok F)"*
⟨proof⟩

lemma *OK_iff_ok: "OK I F = (∀i ∈ I. ∀j ∈ I-{i}. (F i) ok (F j))"*
⟨proof⟩

lemma *OK_imp_ok: "[| OK I F; i ∈ I; j ∈ I; i ≠ j|] ==> (F i) ok (F j)"*
⟨proof⟩

7.10 Allowed

lemma *Allowed_SKIP [simp]: "Allowed SKIP = UNIV"*
⟨proof⟩

lemma *Allowed_Join [simp]: "Allowed (F⊔G) = Allowed F ∩ Allowed G"*
⟨proof⟩

lemma *Allowed_JN [simp]: "Allowed (JOIN I F) = (⋂ i ∈ I. Allowed (F i))"*
⟨*proof*⟩

lemma *ok_iff_Allowed: "F ok G = (F ∈ Allowed G & G ∈ Allowed F)"*
⟨*proof*⟩

lemma *OK_iff_Allowed: "OK I F = (∀ i ∈ I. ∀ j ∈ I-{i}. F i ∈ Allowed(F j))"*
⟨*proof*⟩

7.11 *safety_prop*, for reasoning about given instances of "ok"

lemma *safety_prop_Acts_iff:*
"safety_prop X ==> (Acts G ⊆ insert Id (UNION X Acts)) = (G ∈ X)"
⟨*proof*⟩

lemma *safety_prop_AllowedActs_iff_Allowed:*
"safety_prop X ==> (UNION X Acts ⊆ AllowedActs F) = (X ⊆ Allowed F)"
⟨*proof*⟩

lemma *Allowed_eq:*
"safety_prop X ==> Allowed (mk_program (init, acts, UNION X Acts)) = X"
⟨*proof*⟩

lemma *safety_prop_constrains [iff]: "safety_prop (A co B) = (A ⊆ B)"*
⟨*proof*⟩

lemma *safety_prop_stable [iff]: "safety_prop (stable A)"*
⟨*proof*⟩

lemma *safety_prop_Int [simp]:*
"[| safety_prop X; safety_prop Y |] ==> safety_prop (X ∩ Y)"
⟨*proof*⟩

lemma *safety_prop_INTER1 [simp]:*
"(!!i. safety_prop (X i)) ==> safety_prop (⋂ i. X i)"
⟨*proof*⟩

lemma *safety_prop_INTER [simp]:*
"(!!i. i ∈ I ==> safety_prop (X i)) ==> safety_prop (⋂ i ∈ I. X i)"
⟨*proof*⟩

lemma *def_prg_Allowed:*
"[| F == mk_program (init, acts, UNION X Acts) ; safety_prop X |]
==> Allowed F = X"
⟨*proof*⟩

lemma *Allowed_totalize [simp]: "Allowed (totalize F) = Allowed F"*
⟨*proof*⟩

lemma *def_total_prg_Allowed:*
"[| F == mk_total_program (init, acts, UNION X Acts) ; safety_prop X |]

==> Allowed F = X"

⟨*proof*⟩

lemma *def_UNION_ok_iff:*
```
     "[| F == mk_program(init,acts,UNION X Acts); safety_prop X |]
      ==> F ok G = (G ∈ X & acts ⊆ AllowedActs G)"
```
⟨*proof*⟩

The union of two total programs is total.

lemma *totalize_Join: "totalize F⊔totalize G = totalize (F⊔G)"*
⟨*proof*⟩

lemma *all_total_Join: "[|all_total F; all_total G|] ==> all_total (F⊔G)"*
⟨*proof*⟩

lemma *totalize_JN: "(⨆i ∈ I. totalize (F i)) = totalize(⨆i ∈ I. F i)"*
⟨*proof*⟩

lemma *all_total_JN: "(!!i. i∈I ==> all_total (F i)) ==> all_total(⨆i∈I. F i)"*
⟨*proof*⟩

end

8 Composition: Basic Primitives

theory *Comp* **imports** *Union* **begin**

instance *program :: (type) ord* ⟨*proof*⟩

defs
```
  component_def:          "F ≤ H == ∃G. F⊔G = H"
  strict_component_def:   "(F < (H::'a program)) == (F ≤ H & F ≠ H)"
```

constdefs
```
  component_of :: "'a program =>'a program=> bool"
                                    (infixl "component'_of" 50)
  "F component_of H == ∃G. F ok G & F⊔G = H"

  strict_component_of :: "'a program⇒'a program=> bool"
                                    (infixl "strict'_component'_of" 50)
  "F strict_component_of H == F component_of H & F≠H"

  preserves :: "('a=>'b) => 'a program set"
    "preserves v == ⋂z. stable {s. v s = z}"

  localize  :: "('a=>'b) => 'a program => 'a program"
  "localize v F == mk_program(Init F, Acts F,
                          AllowedActs F ∩ (⋃G ∈ preserves v. Acts G))"

  funPair      :: "['a => 'b, 'a => 'c, 'a] => 'b * 'c"
  "funPair f g == %x. (f x, g x)"
```

8.1 The component relation

lemma *componentI: "H ≤ F | H ≤ G ==> H ≤ (F⊔G)"*
⟨*proof*⟩

lemma *component_eq_subset:*
"(F ≤ G) =
(Init G ⊆ Init F & Acts F ⊆ Acts G & AllowedActs G ⊆ AllowedActs F)"
⟨*proof*⟩

lemma *component_SKIP [iff]: "SKIP ≤ F"*
⟨*proof*⟩

lemma *component_refl [iff]: "F ≤ (F :: 'a program)"*
⟨*proof*⟩

lemma *SKIP_minimal: "F ≤ SKIP ==> F = SKIP"*
⟨*proof*⟩

lemma *component_Join1: "F ≤ (F⊔G)"*
⟨*proof*⟩

lemma *component_Join2: "G ≤ (F⊔G)"*
⟨*proof*⟩

lemma *Join_absorb1: "F ≤ G ==> F⊔G = G"*
⟨*proof*⟩

lemma *Join_absorb2: "G ≤ F ==> F⊔G = F"*
⟨*proof*⟩

lemma *JN_component_iff: "((JOIN I F) ≤ H) = (∀ i ∈ I. F i ≤ H)"*
⟨*proof*⟩

lemma *component_JN: "i ∈ I ==> (F i) ≤ (⨆ i ∈ I. (F i))"*
⟨*proof*⟩

lemma *component_trans: "[| F ≤ G; G ≤ H |] ==> F ≤ (H :: 'a program)"*
⟨*proof*⟩

lemma *component_antisym: "[| F ≤ G; G ≤ F |] ==> F = (G :: 'a program)"*
⟨*proof*⟩

lemma *Join_component_iff: "((F⊔G) ≤ H) = (F ≤ H & G ≤ H)"*
⟨*proof*⟩

lemma *component_constrains: "[| F ≤ G; G ∈ A co B |] ==> F ∈ A co B"*
⟨*proof*⟩

lemma *component_stable: "[| F ≤ G; G ∈ stable A |] ==> F ∈ stable A"*
⟨*proof*⟩

lemmas *program_less_le = strict_component_def [THEN meta_eq_to_obj_eq]*

8.2 The preserves property

lemma *preservesI: "(!!z. F ∈ stable {s. v s = z}) ==> F ∈ preserves v"*
⟨*proof*⟩

lemma *preserves_imp_eq:*
"[| F ∈ preserves v; act ∈ Acts F; (s,s') ∈ act |] ==> v s = v s'"
⟨*proof*⟩

lemma *Join_preserves [iff]:*
"(F⊔G ∈ preserves v) = (F ∈ preserves v & G ∈ preserves v)"
⟨*proof*⟩

lemma *JN_preserves [iff]:*
"(JOIN I F ∈ preserves v) = (∀i ∈ I. F i ∈ preserves v)"
⟨*proof*⟩

lemma *SKIP_preserves [iff]: "SKIP ∈ preserves v"*
⟨*proof*⟩

lemma *funPair_apply [simp]: "(funPair f g) x = (f x, g x)"*
⟨*proof*⟩

lemma *preserves_funPair: "preserves (funPair v w) = preserves v ∩ preserves w"*
⟨*proof*⟩

declare *preserves_funPair [THEN eqset_imp_iff, iff]*

lemma *funPair_o_distrib: "(funPair f g) o h = funPair (f o h) (g o h)"*
⟨*proof*⟩

lemma *fst_o_funPair [simp]: "fst o (funPair f g) = f"*
⟨*proof*⟩

lemma *snd_o_funPair [simp]: "snd o (funPair f g) = g"*
⟨*proof*⟩

lemma *subset_preserves_o: "preserves v ⊆ preserves (w o v)"*
⟨*proof*⟩

lemma *preserves_subset_stable: "preserves v ⊆ stable {s. P (v s)}"*
⟨*proof*⟩

lemma *preserves_subset_increasing: "preserves v ⊆ increasing v"*
⟨*proof*⟩

lemma *preserves_id_subset_stable: "preserves id ⊆ stable A"*
⟨*proof*⟩

lemma *safety_prop_preserves [iff]: "safety_prop (preserves v)"*
⟨*proof*⟩

lemma *stable_localTo_stable2:*
```
"[| F ∈ stable {s. P (v s) (w s)};
    G ∈ preserves v;  G ∈ preserves w |]
 ==> F⊔G ∈ stable {s. P (v s) (w s)}"
```
⟨*proof*⟩

lemma *Increasing_preserves_Stable:*
```
"[| F ∈ stable {s. v s ≤ w s};  G ∈ preserves v; F⊔G ∈ Increasing w
|]
 ==> F⊔G ∈ Stable {s. v s ≤ w s}"
```
⟨*proof*⟩

lemma *component_of_imp_component: "F component_of H ==> F ≤ H"*
⟨*proof*⟩

lemma *component_of_refl [simp]: "F component_of F"*
⟨*proof*⟩

lemma *component_of_SKIP [simp]: "SKIP component_of F"*
⟨*proof*⟩

lemma *component_of_trans:*
```
"[| F component_of G; G component_of H |] ==> F component_of H"
```
⟨*proof*⟩

lemmas *strict_component_of_eq =*
```
strict_component_of_def [THEN meta_eq_to_obj_eq, standard]
```

lemma *localize_Init_eq [simp]: "Init (localize v F) = Init F"*
⟨*proof*⟩

lemma *localize_Acts_eq [simp]: "Acts (localize v F) = Acts F"*
⟨*proof*⟩

lemma *localize_AllowedActs_eq [simp]:*
```
"AllowedActs (localize v F) = AllowedActs F ∩ (⋃G ∈ preserves v. Acts
G)"
```
⟨*proof*⟩

end

9 Guarantees Specifications

theory *Guar* **imports** *Comp* **begin**

instance *program :: (type) order*
⟨*proof*⟩

Existential and Universal properties. I formalize the two-program case, proving equivalence with Chandy and Sanders's n-ary definitions

constdefs

```
ex_prop  :: "'a program set => bool"
 "ex_prop X == ∀F G. F ok G -->F ∈ X | G ∈ X --> (F⊔G) ∈ X"

strict_ex_prop  :: "'a program set => bool"
 "strict_ex_prop X == ∀F G.  F ok G --> (F ∈ X | G ∈ X) = (F⊔G ∈ X)"

uv_prop  :: "'a program set => bool"
 "uv_prop X == SKIP ∈ X & (∀F G. F ok G --> F ∈ X & G ∈ X --> (F⊔G) ∈
X)"

strict_uv_prop  :: "'a program set => bool"
 "strict_uv_prop X ==
    SKIP ∈ X & (∀F G. F ok G --> (F ∈ X & G ∈ X) = (F⊔G ∈ X))"
```

Guarantees properties

constdefs

```
guar :: "['a program set, 'a program set] => 'a program set"
       (infixl "guarantees" 55)
 "X guarantees Y == {F. ∀G. F ok G --> F⊔G ∈ X --> F⊔G ∈ Y}"



 wg :: "['a program, 'a program set] =>  'a program set"
"wg F Y == Union({X. F ∈ X guarantees Y})"


 wx :: "('a program) set => ('a program)set"
 "wx X == Union({Y. Y ⊆ X & ex_prop Y})"


welldef :: "'a program set"
"welldef == {F. Init F ≠ {}}"

refines :: "['a program, 'a program, 'a program set] => bool"
                      ("(3_ refines _ wrt _)" [10,10,10] 10)
"G refines F wrt X ==
   ∀H. (F ok H & G ok H & F⊔H ∈ welldef ∩ X) -->
       (G⊔H ∈ welldef ∩ X)"

iso_refines :: "['a program, 'a program, 'a program set] => bool"
                            ("(3_ iso'_refines _ wrt _)" [10,10,10] 10)
"G iso_refines F wrt X ==
```

```
  F ∈ welldef ∩ X --> G ∈ welldef ∩ X"
```

lemma *OK_insert_iff:*
```
     "(OK (insert i I) F) =
      (if i ∈ I then OK I F else OK I F & (F i ok JOIN I F))"
```
⟨*proof*⟩

9.1 Existential Properties

lemma *ex1 [rule_format]:*
```
 "[| ex_prop X; finite GG |] ==>
     GG ∩ X ≠ {}--> OK GG (%G. G) --> (⊔G ∈ GG. G) ∈ X"
```
⟨*proof*⟩

lemma *ex2:*
```
     "∀GG. finite GG & GG ∩ X ≠ {} --> OK GG (%G. G) -->(⊔G ∈ GG. G):X

      ==> ex_prop X"
```
⟨*proof*⟩

lemma *ex_prop_finite:*
```
     "ex_prop X =
      (∀GG. finite GG & GG ∩ X ≠ {} & OK GG (%G. G)--> (⊔G ∈ GG. G) ∈
X)"
```
⟨*proof*⟩

lemma *ex_prop_equiv:*
```
     "ex_prop X = (∀G. G ∈ X = (∀H. (G component_of H) --> H ∈ X))"
```
⟨*proof*⟩

9.2 Universal Properties

lemma *uv1 [rule_format]:*
```
     "[| uv_prop X; finite GG |]
      ==> GG ⊆ X & OK GG (%G. G) --> (⊔G ∈ GG. G) ∈ X"
```
⟨*proof*⟩

lemma *uv2:*
```
     "∀GG. finite GG & GG ⊆ X & OK GG (%G. G) --> (⊔G ∈ GG. G) ∈ X
      ==> uv_prop X"
```
⟨*proof*⟩

lemma *uv_prop_finite:*
```
     "uv_prop X =
      (∀GG. finite GG & GG ⊆ X & OK GG (%G. G) --> (⊔G ∈ GG. G): X)"
```
⟨*proof*⟩

9.3 Guarantees

lemma *guaranteesI:*
```
"(!!G. [| F ok G; F⊔G ∈ X |] ==> F⊔G ∈ Y) ==> F ∈ X guarantees Y"
```
⟨*proof*⟩

lemma *guaranteesD:*
```
"[| F ∈ X guarantees Y;  F ok G;  F⊔G ∈ X |] ==> F⊔G ∈ Y"
```
⟨*proof*⟩

lemma *component_guaranteesD:*
```
"[| F ∈ X guarantees Y;  F⊔G = H;  H ∈ X;  F ok G |] ==> H ∈ Y"
```
⟨*proof*⟩

lemma *guarantees_weaken:*
```
"[| F ∈ X guarantees X'; Y ⊆ X; X' ⊆ Y' |] ==> F ∈ Y guarantees Y'"
```
⟨*proof*⟩

lemma *subset_imp_guarantees_UNIV:* `"X ⊆ Y ==> X guarantees Y = UNIV"`
⟨*proof*⟩

lemma *subset_imp_guarantees:* `"X ⊆ Y ==> F ∈ X guarantees Y"`
⟨*proof*⟩

lemma *ex_prop_imp:* `"ex_prop Y ==> (Y = UNIV guarantees Y)"`
⟨*proof*⟩

lemma *guarantees_imp:* `"(Y = UNIV guarantees Y) ==> ex_prop(Y)"`
⟨*proof*⟩

lemma *ex_prop_equiv2:* `"(ex_prop Y) = (Y = UNIV guarantees Y)"`
⟨*proof*⟩

9.4 Distributive Laws. Re-Orient to Perform Miniscoping

lemma *guarantees_UN_left:*
```
"(⋃i ∈ I. X i) guarantees Y = (⋂i ∈ I. X i guarantees Y)"
```
⟨*proof*⟩

lemma *guarantees_Un_left:*
```
"(X ∪ Y) guarantees Z = (X guarantees Z) ∩ (Y guarantees Z)"
```
⟨*proof*⟩

lemma *guarantees_INT_right:*
```
"X guarantees (⋂i ∈ I. Y i) = (⋂i ∈ I. X guarantees Y i)"
```
⟨*proof*⟩

lemma *guarantees_Int_right:*
```
"Z guarantees (X ∩ Y) = (Z guarantees X) ∩ (Z guarantees Y)"
```
⟨*proof*⟩

lemma *guarantees_Int_right_I:*
```
"[| F ∈ Z guarantees X;  F ∈ Z guarantees Y |]
==> F ∈ Z guarantees (X ∩ Y)"
```
⟨*proof*⟩

lemma *guarantees_INT_right_iff:*
```
"(F ∈ X guarantees (INTER I Y)) = (∀i∈I. F ∈ X guarantees (Y i))"
```
⟨*proof*⟩

lemma *shunting:* `"(X guarantees Y) = (UNIV guarantees (-X ∪ Y))"`
⟨*proof*⟩

lemma *contrapositive:* `"(X guarantees Y) = -Y guarantees -X"`
⟨*proof*⟩

lemma *combining1:*
```
"[| F ∈ V guarantees X;  F ∈ (X ∩ Y) guarantees Z |]
 ==> F ∈ (V ∩ Y) guarantees Z"
```
⟨*proof*⟩

lemma *combining2:*
```
"[| F ∈ V guarantees (X ∪ Y);  F ∈ Y guarantees Z |]
 ==> F ∈ V guarantees (X ∪ Z)"
```
⟨*proof*⟩

lemma *all_guarantees:*
```
"∀i∈I. F ∈ X guarantees (Y i) ==> F ∈ X guarantees (⋂i ∈ I. Y i)"
```
⟨*proof*⟩

lemma *ex_guarantees:*
```
"∃i∈I. F ∈ X guarantees (Y i) ==> F ∈ X guarantees (⋃i ∈ I. Y i)"
```
⟨*proof*⟩

9.5 Guarantees: Additional Laws (by lcp)

lemma *guarantees_Join_Int:*
```
"[| F ∈ U guarantees V;  G ∈ X guarantees Y; F ok G |]
 ==> F⊔G ∈ (U ∩ X) guarantees (V ∩ Y)"
```
⟨*proof*⟩

lemma *guarantees_Join_Un:*
```
"[| F ∈ U guarantees V;  G ∈ X guarantees Y; F ok G |]
 ==> F⊔G ∈ (U ∪ X) guarantees (V ∪ Y)"
```
⟨*proof*⟩

lemma *guarantees_JN_INT:*
```
"[| ∀i∈I. F i ∈ X i guarantees Y i;  OK I F |]
 ==> (JOIN I F) ∈ (INTER I X) guarantees (INTER I Y)"
```

⟨*proof*⟩

lemma *guarantees_JN_UN:*
```
"[| ∀i∈I. F i ∈ X i guarantees Y i;  OK I F |]
 ==> (JOIN I F) ∈ (UNION I X) guarantees (UNION I Y)"
```
⟨*proof*⟩

9.6 Guarantees Laws for Breaking Down the Program (by lcp)

lemma *guarantees_Join_I1:*
```
"[| F ∈ X guarantees Y;  F ok G |] ==> F⊔G ∈ X guarantees Y"
```
⟨*proof*⟩

lemma *guarantees_Join_I2:*
```
"[| G ∈ X guarantees Y;  F ok G |] ==> F⊔G ∈ X guarantees Y"
```
⟨*proof*⟩

lemma *guarantees_JN_I:*
```
"[| i ∈ I;  F i ∈ X guarantees Y;  OK I F |]
 ==> (⨆i ∈ I. (F i)) ∈ X guarantees Y"
```
⟨*proof*⟩

lemma *Join_welldef_D1:* "F⊔G ∈ welldef ==> F ∈ welldef"
⟨*proof*⟩

lemma *Join_welldef_D2:* "F⊔G ∈ welldef ==> G ∈ welldef"
⟨*proof*⟩

lemma *refines_refl:* "F refines F wrt X"
⟨*proof*⟩

lemma *refines_trans:*
```
"[| H refines G wrt X;  G refines F wrt X |] ==> H refines F wrt X"
```
⟨*proof*⟩

lemma *strict_ex_refine_lemma:*
```
"strict_ex_prop X
 ==> (∀H. F ok H & G ok H & F⊔H ∈ X --> G⊔H ∈ X)
        = (F ∈ X --> G ∈ X)"
```
⟨*proof*⟩

lemma *strict_ex_refine_lemma_v:*
```
"strict_ex_prop X
 ==> (∀H. F ok H & G ok H & F⊔H ∈ welldef & F⊔H ∈ X --> G⊔H ∈ X) =

     (F ∈ welldef ∩ X --> G ∈ X)"
```

⟨*proof*⟩

lemma *ex_refinement_thm:*
```
"[| strict_ex_prop X;
    ∀H. F ok H & G ok H & F⊔H ∈ welldef ∩ X --> G⊔H ∈ welldef |]
 ==> (G refines F wrt X) = (G iso_refines F wrt X)"
```
⟨*proof*⟩

lemma *strict_uv_refine_lemma:*
```
"strict_uv_prop X ==>
 (∀H. F ok H & G ok H & F⊔H ∈ X --> G⊔H ∈ X) = (F ∈ X --> G ∈ X)"
```
⟨*proof*⟩

lemma *strict_uv_refine_lemma_v:*
```
"strict_uv_prop X
 ==> (∀H. F ok H & G ok H & F⊔H ∈ welldef & F⊔H ∈ X --> G⊔H ∈ X) =

     (F ∈ welldef ∩ X --> G ∈ X)"
```
⟨*proof*⟩

lemma *uv_refinement_thm:*
```
"[| strict_uv_prop X;
    ∀H. F ok H & G ok H & F⊔H ∈ welldef ∩ X -->
        G⊔H ∈ welldef |]
 ==> (G refines F wrt X) = (G iso_refines F wrt X)"
```
⟨*proof*⟩

lemma *guarantees_equiv:*
```
"(F ∈ X guarantees Y) = (∀H. H ∈ X ⟶ (F component_of H ⟶ H ∈ Y))"
```
⟨*proof*⟩

lemma *wg_weakest:* `"!!X. F∈ (X guarantees Y) ==> X ⊆ (wg F Y)"`
⟨*proof*⟩

lemma *wg_guarantees:* `"F∈ ((wg F Y) guarantees Y)"`
⟨*proof*⟩

lemma *wg_equiv:* `"(H ∈ wg F X) = (F component_of H --> H ∈ X)"`
⟨*proof*⟩

lemma *component_of_wg:* `"F component_of H ==> (H ∈ wg F X) = (H ∈ X)"`
⟨*proof*⟩

lemma *wg_finite:*
```
"∀FF. finite FF & FF ∩ X ≠ {} --> OK FF (%F. F)
     --> (∀F∈FF. ((⨆F ∈ FF. F): wg F X) = ((⨆F ∈ FF. F):X))"
```
⟨*proof*⟩

lemma *wg_ex_prop:* `"ex_prop X ==> (F ∈ X) = (∀H. H ∈ wg F X)"`
⟨*proof*⟩

lemma *wx_subset: "(wx X)<=X"*
⟨*proof*⟩

lemma *wx_ex_prop: "ex_prop (wx X)"*
⟨*proof*⟩

lemma *wx_weakest: "∀ Z. Z<= X --> ex_prop Z --> Z ⊆ wx X"*
⟨*proof*⟩

lemma *wx'_ex_prop: "ex_prop({F. ∀ G. F ok G --> F⊔G ∈ X})"*
⟨*proof*⟩

Equivalence with the other definition of wx

lemma *wx_equiv: "wx X = {F. ∀ G. F ok G --> (F⊔G) ∈ X}"*
⟨*proof*⟩

Propositions 7 to 11 are about this second definition of wx. They are the same as the ones proved for the first definition of wx, by equivalence

lemma *guarantees_wx_eq: "(X guarantees Y) = wx(-X ∪ Y)"*
⟨*proof*⟩

lemma *stable_guarantees_Always:*
    "Init F ⊆ A ==> F ∈ (stable A) guarantees (Always A)"
⟨*proof*⟩

lemma *constrains_guarantees_leadsTo:*
    "F ∈ transient A ==> F ∈ (A co A ∪ B) guarantees (A leadsTo (B-A))"
⟨*proof*⟩

end

10 Extending State Sets

theory *Extend* **imports** *Guar* **begin**

constdefs

```
Restrict :: "[ 'a set, ('a*'b) set] => ('a*'b) set"
  "Restrict A r == r ∩ (A <*> UNIV)"

good_map :: "['a*'b => 'c] => bool"
  "good_map h == surj h & (∀ x y. fst (inv h (h (x,y))) = x)"


extend_set :: "['a*'b => 'c, 'a set] => 'c set"
  "extend_set h A == h ` (A <*> UNIV)"
```

```
  project_set :: "['a*'b => 'c, 'c set] => 'a set"
    "project_set h C == {x. ∃y. h(x,y) ∈ C}"

  extend_act :: "['a*'b => 'c, ('a*'a) set] => ('c*'c) set"
    "extend_act h == %act. ⋃(s,s') ∈ act. ⋃y. {(h(s,y), h(s',y))}"

  project_act :: "['a*'b => 'c, ('c*'c) set] => ('a*'a) set"
    "project_act h act == {(x,x'). ∃y y'. (h(x,y), h(x',y')) ∈ act}"

  extend :: "['a*'b => 'c, 'a program] => 'c program"
    "extend h F == mk_program (extend_set h (Init F),
                               extend_act h ` Acts F,
                               project_act h -` AllowedActs F)"


  project :: "['a*'b => 'c, 'c set, 'c program] => 'a program"
    "project h C F ==
       mk_program (project_set h (Init F),
                   project_act h ` Restrict C ` Acts F,
                   {act. Restrict (project_set h C) act :
                         project_act h ` Restrict C ` AllowedActs F})"
```

locale *Extend* =
  fixes *f* :: "'c => 'a"
    and *g* :: "'c => 'b"
    and *h* :: "'a*'b => 'c"
    and *slice* :: "['c set, 'b] => 'a set"
  assumes
    good_h: "good_map h"
  defines *f_def*: "f z == fst (inv h z)"
    and *g_def*: "g z == snd (inv h z)"
    and *slice_def*: "slice Z y == {x. h(x,y) ∈ Z}"

10.1 Restrict

lemma *Restrict_iff [iff]*: "((x,y): Restrict A r) = ((x,y): r & x ∈ A)"
⟨*proof*⟩

lemma *Restrict_UNIV [simp]*: "Restrict UNIV = id"
⟨*proof*⟩

lemma *Restrict_empty [simp]*: "Restrict {} r = {}"
⟨*proof*⟩

lemma *Restrict_Int [simp]*: "Restrict A (Restrict B r) = Restrict (A ∩ B) r"
⟨*proof*⟩

lemma *Restrict_triv*: "Domain r ⊆ A ==> Restrict A r = r"
⟨*proof*⟩

lemma *Restrict_subset*: "Restrict A r ⊆ r"
⟨*proof*⟩

lemma *Restrict_eq_mono:*
```
"[| A ⊆ B;  Restrict B r = Restrict B s |]
 ==> Restrict A r = Restrict A s"
```
⟨*proof*⟩

lemma *Restrict_imageI:*
```
"[| s ∈ RR;  Restrict A r = Restrict A s |]
 ==> Restrict A r ∈ Restrict A ‘ RR"
```
⟨*proof*⟩

lemma *Domain_Restrict [simp]: "Domain (Restrict A r) = A ∩ Domain r"*
⟨*proof*⟩

lemma *Image_Restrict [simp]: "(Restrict A r) ‘‘ B = r ‘‘ (A ∩ B)"*
⟨*proof*⟩

lemma *good_mapI:*
assumes *surj_h: "surj h"*
and *prem: "!! x x' y y'. h(x,y) = h(x',y') ==> x=x'"*
shows *"good_map h"*
⟨*proof*⟩

lemma *good_map_is_surj: "good_map h ==> surj h"*
⟨*proof*⟩

lemma *fst_inv_equalityI:*
assumes *surj_h: "surj h"*
and *prem: "!! x y. g (h(x,y)) = x"*
shows *"fst (inv h z) = g z"*
⟨*proof*⟩

10.2 Trivial properties of f, g, h

lemma *(***in** *Extend) f_h_eq [simp]: "f(h(x,y)) = x"*
⟨*proof*⟩

lemma *(***in** *Extend) h_inject1 [dest]: "h(x,y) = h(x',y') ==> x=x'"*
⟨*proof*⟩

lemma *(***in** *Extend) h_f_g_equiv: "h(f z, g z) == z"*
⟨*proof*⟩

lemma *(***in** *Extend) h_f_g_eq: "h(f z, g z) = z"*
⟨*proof*⟩

lemma *(***in** *Extend) split_extended_all:*
```
"(!!z. PROP P z) == (!!u y. PROP P (h (u, y)))"
```
⟨*proof*⟩

10.3 *extend_set*: basic properties

lemma *project_set_iff [iff]:*
"(x ∈ project_set h C) = (∃y. h(x,y) ∈ C)"
⟨*proof*⟩

lemma *extend_set_mono: "A ⊆ B ==> extend_set h A ⊆ extend_set h B"*
⟨*proof*⟩

lemma *(***in** *Extend) mem_extend_set_iff [iff]: "z ∈ extend_set h A = (f z ∈ A)"*
⟨*proof*⟩

lemma *(***in** *Extend) extend_set_strict_mono [iff]:*
"(extend_set h A ⊆ extend_set h B) = (A ⊆ B)"
⟨*proof*⟩

lemma *extend_set_empty [simp]: "extend_set h {} = {}"*
⟨*proof*⟩

lemma *(***in** *Extend) extend_set_eq_Collect: "extend_set h {s. P s} = {s. P(f s)}"*
⟨*proof*⟩

lemma *(***in** *Extend) extend_set_sing: "extend_set h {x} = {s. f s = x}"*
⟨*proof*⟩

lemma *(***in** *Extend) extend_set_inverse [simp]:*
"project_set h (extend_set h C) = C"
⟨*proof*⟩

lemma *(***in** *Extend) extend_set_project_set:*
"C ⊆ extend_set h (project_set h C)"
⟨*proof*⟩

lemma *(***in** *Extend) inj_extend_set: "inj (extend_set h)"*
⟨*proof*⟩

lemma *(***in** *Extend) extend_set_UNIV_eq [simp]: "extend_set h UNIV = UNIV"*
⟨*proof*⟩

10.4 *project_set*: basic properties

lemma *(***in** *Extend) project_set_eq: "project_set h C = f ‘ C"*
⟨*proof*⟩

lemma *(***in** *Extend) project_set_I: "!!z. z ∈ C ==> f z ∈ project_set h C"*
⟨*proof*⟩

10.5 More laws

lemma *(***in** *Extend) project_set_extend_set_Int:*
"project_set h ((extend_set h A) ∩ B) = A ∩ (project_set h B)"
⟨*proof*⟩

lemma *(in Extend) project_set_extend_set_Un:*
"project_set h ((extend_set h A) $\cup$ B) = A $\cup$ (project_set h B)"
⟨*proof*⟩

lemma *project_set_Int_subset:*
"project_set h (A $\cap$ B) $\subseteq$ (project_set h A) $\cap$ (project_set h B)"
⟨*proof*⟩

lemma *(in Extend) extend_set_Un_distrib:*
"extend_set h (A $\cup$ B) = extend_set h A $\cup$ extend_set h B"
⟨*proof*⟩

lemma *(in Extend) extend_set_Int_distrib:*
"extend_set h (A $\cap$ B) = extend_set h A $\cap$ extend_set h B"
⟨*proof*⟩

lemma *(in Extend) extend_set_INT_distrib:*
"extend_set h (INTER A B) = ($\bigcap$x $\in$ A. extend_set h (B x))"
⟨*proof*⟩

lemma *(in Extend) extend_set_Diff_distrib:*
"extend_set h (A - B) = extend_set h A - extend_set h B"
⟨*proof*⟩

lemma *(in Extend) extend_set_Union:*
"extend_set h (Union A) = ($\bigcup$X $\in$ A. extend_set h X)"
⟨*proof*⟩

lemma *(in Extend) extend_set_subset_Compl_eq:*
"(extend_set h A $\subseteq$ - extend_set h B) = (A $\subseteq$ - B)"
⟨*proof*⟩

10.6 extend_act

lemma *(in Extend) mem_extend_act_iff [iff]:*
"((h(s,y), h(s',y)) $\in$ extend_act h act) = ((s, s') $\in$ act)"
⟨*proof*⟩

lemma *(in Extend) extend_act_D:*
"(z, z') $\in$ extend_act h act ==> (f z, f z') $\in$ act"
⟨*proof*⟩

lemma *(in Extend) extend_act_inverse [simp]:*
"project_act h (extend_act h act) = act"
⟨*proof*⟩

lemma *(in Extend) project_act_extend_act_restrict [simp]:*
"project_act h (Restrict C (extend_act h act)) =
Restrict (project_set h C) act"
⟨*proof*⟩

lemma *(***in** *Extend) subset_extend_act_D:*
    "act' ⊆ extend_act h act ==> project_act h act' ⊆ act"
⟨*proof*⟩

lemma *(***in** *Extend) inj_extend_act: "inj (extend_act h)"*
⟨*proof*⟩

lemma *(***in** *Extend) extend_act_Image [simp]:*
    "extend_act h act ‘‘ (extend_set h A) = extend_set h (act ‘‘ A)"
⟨*proof*⟩

lemma *(***in** *Extend) extend_act_strict_mono [iff]:*
    "(extend_act h act' ⊆ extend_act h act) = (act'<=act)"
⟨*proof*⟩

declare *(***in** *Extend) inj_extend_act [THEN inj_eq, iff]*

lemma *Domain_extend_act:*
    "Domain (extend_act h act) = extend_set h (Domain act)"
⟨*proof*⟩

lemma *(***in** *Extend) extend_act_Id [simp]:*
    "extend_act h Id = Id"
⟨*proof*⟩

lemma *(***in** *Extend) project_act_I:*
    "!!z z'. (z, z') ∈ act ==> (f z, f z') ∈ project_act h act"
⟨*proof*⟩

lemma *(***in** *Extend) project_act_Id [simp]: "project_act h Id = Id"*
⟨*proof*⟩

lemma *(***in** *Extend) Domain_project_act:*
  "Domain (project_act h act) = project_set h (Domain act)"
⟨*proof*⟩

10.7 extend

Basic properties

lemma *Init_extend [simp]:*
    "Init (extend h F) = extend_set h (Init F)"
⟨*proof*⟩

lemma *Init_project [simp]:*
    "Init (project h C F) = project_set h (Init F)"
⟨*proof*⟩

lemma *(***in** *Extend) Acts_extend [simp]:*
    "Acts (extend h F) = (extend_act h ‘ Acts F)"
⟨*proof*⟩

lemma *(***in** *Extend) AllowedActs_extend [simp]:*

```
    "AllowedActs (extend h F) = project_act h -` AllowedActs F"
```
⟨proof⟩

lemma *Acts_project [simp]:*
```
    "Acts(project h C F) = insert Id (project_act h ` Restrict C ` Acts F)"
```
⟨proof⟩

lemma *(**in** Extend) AllowedActs_project [simp]:*
```
    "AllowedActs(project h C F) =
       {act. Restrict (project_set h C) act
              ∈ project_act h ` Restrict C ` AllowedActs F}"
```
⟨proof⟩

lemma *(**in** Extend) Allowed_extend:*
```
    "Allowed (extend h F) = project h UNIV -` Allowed F"
```
⟨proof⟩

lemma *(**in** Extend) extend_SKIP [simp]: "extend h SKIP = SKIP"*
⟨proof⟩

lemma *project_set_UNIV [simp]: "project_set h UNIV = UNIV"*
⟨proof⟩

lemma *project_set_Union:*
```
    "project_set h (Union A) = (⋃X ∈ A. project_set h X)"
```
⟨proof⟩

lemma *(**in** Extend) project_act_Restrict_subset:*
```
    "project_act h (Restrict C act) ⊆
     Restrict (project_set h C) (project_act h act)"
```
⟨proof⟩

lemma *(**in** Extend) project_act_Restrict_Id_eq:*
```
    "project_act h (Restrict C Id) = Restrict (project_set h C) Id"
```
⟨proof⟩

lemma *(**in** Extend) project_extend_eq:*
```
    "project h C (extend h F) =
     mk_program (Init F, Restrict (project_set h C) ` Acts F,
                 {act. Restrict (project_set h C) act
                        ∈ project_act h ` Restrict C `
                              (project_act h -` AllowedActs F)})"
```
⟨proof⟩

lemma *(**in** Extend) extend_inverse [simp]:*
```
    "project h UNIV (extend h F) = F"
```
⟨proof⟩

lemma *(**in** Extend) inj_extend: "inj (extend h)"*
⟨proof⟩

lemma *(**in** Extend) extend_Join [simp]:*

```
"extend h (F⊔G) = extend h F⊔extend h G"
```
⟨*proof*⟩

lemma *(***in** *Extend) extend_JN [simp]:*
```
"extend h (JOIN I F) = (⨆ i ∈ I. extend h (F i))"
```
⟨*proof*⟩

lemma *(***in** *Extend) extend_mono: "F ≤ G ==> extend h F ≤ extend h G"*
⟨*proof*⟩

lemma *(***in** *Extend) project_mono: "F ≤ G ==> project h C F ≤ project h C G"*
⟨*proof*⟩

lemma *(***in** *Extend) all_total_extend: "all_total F ==> all_total (extend h F)"*
⟨*proof*⟩

10.8 Safety: co, stable

lemma *(***in** *Extend) extend_constrains:*
```
"(extend h F ∈ (extend_set h A) co (extend_set h B)) =
 (F ∈ A co B)"
```
⟨*proof*⟩

lemma *(***in** *Extend) extend_stable:*
```
"(extend h F ∈ stable (extend_set h A)) = (F ∈ stable A)"
```
⟨*proof*⟩

lemma *(***in** *Extend) extend_invariant:*
```
"(extend h F ∈ invariant (extend_set h A)) = (F ∈ invariant A)"
```
⟨*proof*⟩

lemma *(***in** *Extend) extend_constrains_project_set:*
```
"extend h F ∈ A co B ==> F ∈ (project_set h A) co (project_set h B)"
```
⟨*proof*⟩

lemma *(***in** *Extend) extend_stable_project_set:*
```
"extend h F ∈ stable A ==> F ∈ stable (project_set h A)"
```
⟨*proof*⟩

10.9 Weak safety primitives: Co, Stable

lemma *(***in** *Extend) reachable_extend_f:*
```
"p ∈ reachable (extend h F) ==> f p ∈ reachable F"
```
⟨*proof*⟩

lemma *(***in** *Extend) h_reachable_extend:*
```
"h(s,y) ∈ reachable (extend h F) ==> s ∈ reachable F"
```
⟨*proof*⟩

lemma *(in Extend) reachable_extend_eq:*
```
"reachable (extend h F) = extend_set h (reachable F)"
```
⟨*proof*⟩

lemma *(in Extend) extend_Constrains:*
```
"(extend h F ∈ (extend_set h A) Co (extend_set h B)) =
 (F ∈ A Co B)"
```
⟨*proof*⟩

lemma *(in Extend) extend_Stable:*
```
"(extend h F ∈ Stable (extend_set h A)) = (F ∈ Stable A)"
```
⟨*proof*⟩

lemma *(in Extend) extend_Always:*
```
"(extend h F ∈ Always (extend_set h A)) = (F ∈ Always A)"
```
⟨*proof*⟩

lemma *project_act_mono:*
```
"D ⊆ C ==>
 project_act h (Restrict D act) ⊆ project_act h (Restrict C act)"
```
⟨*proof*⟩

lemma *(in Extend) project_constrains_mono:*
```
"[| D ⊆ C; project h C F ∈ A co B |] ==> project h D F ∈ A co B"
```
⟨*proof*⟩

lemma *(in Extend) project_stable_mono:*
```
"[| D ⊆ C;  project h C F ∈ stable A |] ==> project h D F ∈ stable A"
```
⟨*proof*⟩

lemma *(in Extend) project_constrains:*
```
"(project h C F ∈ A co B)  =
 (F ∈ (C ∩ extend_set h A) co (extend_set h B) & A ⊆ B)"
```
⟨*proof*⟩

lemma *(in Extend) project_stable:*
```
"(project h UNIV F ∈ stable A) = (F ∈ stable (extend_set h A))"
```
⟨*proof*⟩

lemma *(in Extend) project_stable_I:*
```
"F ∈ stable (extend_set h A) ==> project h C F ∈ stable A"
```
⟨*proof*⟩

lemma *(in Extend) Int_extend_set_lemma:*
```
"A ∩ extend_set h ((project_set h A) ∩ B) = A ∩ extend_set h B"
```
⟨*proof*⟩

lemma *project_constrains_project_set:*
    "G ∈ C co B ==> project h C G ∈ project_set h C co project_set h B"
⟨*proof*⟩

lemma *project_stable_project_set:*
    "G ∈ stable C ==> project h C G ∈ stable (project_set h C)"
⟨*proof*⟩

10.10 Progress: transient, ensures

lemma *(***in** *Extend) extend_transient:*
    "(extend h F ∈ transient (extend_set h A)) = (F ∈ transient A)"
⟨*proof*⟩

lemma *(***in** *Extend) extend_ensures:*
    "(extend h F ∈ (extend_set h A) ensures (extend_set h B)) =
     (F ∈ A ensures B)"
⟨*proof*⟩

lemma *(***in** *Extend) leadsTo_imp_extend_leadsTo:*
    "F ∈ A leadsTo B
     ==> extend h F ∈ (extend_set h A) leadsTo (extend_set h B)"
⟨*proof*⟩

10.11 Proving the converse takes some doing!

lemma *(***in** *Extend) slice_iff [iff]: "(x ∈ slice C y) = (h(x,y) ∈ C)"*
⟨*proof*⟩

lemma *(***in** *Extend) slice_Union: "slice (Union S) y = (⋃x ∈ S. slice x y)"*
⟨*proof*⟩

lemma *(***in** *Extend) slice_extend_set: "slice (extend_set h A) y = A"*
⟨*proof*⟩

lemma *(***in** *Extend) project_set_is_UN_slice:*
    "project_set h A = (⋃y. slice A y)"
⟨*proof*⟩

lemma *(***in** *Extend) extend_transient_slice:*
    "extend h F ∈ transient A ==> F ∈ transient (slice A y)"
⟨*proof*⟩

lemma *(***in** *Extend) extend_constrains_slice:*
    "extend h F ∈ A co B ==> F ∈ (slice A y) co (slice B y)"
⟨*proof*⟩

lemma *(***in** *Extend) extend_ensures_slice:*
    "extend h F ∈ A ensures B ==> F ∈ (slice A y) ensures (project_set h
B)"
⟨*proof*⟩

lemma *(***in** *Extend) leadsTo_slice_project_set:*

"∀ y. F ∈ (slice B y) leadsTo CU ==> F ∈ (project_set h B) leadsTo CU"
⟨proof⟩

lemma *(in Extend) extend_leadsTo_slice [rule_format]:*
"extend h F ∈ AU leadsTo BU
==> ∀ y. F ∈ (slice AU y) leadsTo (project_set h BU)"
⟨proof⟩

lemma *(in Extend) extend_leadsTo:*
"(extend h F ∈ (extend_set h A) leadsTo (extend_set h B)) =
(F ∈ A leadsTo B)"
⟨proof⟩

lemma *(in Extend) extend_LeadsTo:*
"(extend h F ∈ (extend_set h A) LeadsTo (extend_set h B)) =
(F ∈ A LeadsTo B)"
⟨proof⟩

10.12 preserves

lemma *(in Extend) project_preserves_I:*
"G ∈ preserves (v o f) ==> project h C G ∈ preserves v"
⟨proof⟩

lemma *(in Extend) project_preserves_id_I:*
"G ∈ preserves f ==> project h C G ∈ preserves id"
⟨proof⟩

lemma *(in Extend) extend_preserves:*
"(extend h G ∈ preserves (v o f)) = (G ∈ preserves v)"
⟨proof⟩

lemma *(in Extend) inj_extend_preserves: "inj h ==> (extend h G ∈ preserves g)"*
⟨proof⟩

10.13 Guarantees

lemma *(in Extend) project_extend_Join:*
"project h UNIV ((extend h F)⊔G) = F⊔(project h UNIV G)"
⟨proof⟩

lemma *(in Extend) extend_Join_eq_extend_D:*
"(extend h F)⊔G = extend h H ==> H = F⊔(project h UNIV G)"
⟨proof⟩

lemma *(in Extend) ok_extend_imp_ok_project:*
"extend h F ok G ==> F ok project h UNIV G"
⟨proof⟩

lemma *(***in** *Extend) ok_extend_iff: "(extend h F ok extend h G) = (F ok G)"*
⟨*proof*⟩

lemma *(***in** *Extend) OK_extend_iff: "OK I (%i. extend h (F i)) = (OK I F)"*
⟨*proof*⟩

lemma *(***in** *Extend) guarantees_imp_extend_guarantees:*
```
    "F ∈ X guarantees Y ==>
     extend h F ∈ (extend h ‘ X) guarantees (extend h ‘ Y)"
```
⟨*proof*⟩

lemma *(***in** *Extend) extend_guarantees_imp_guarantees:*
```
    "extend h F ∈ (extend h ‘ X) guarantees (extend h ‘ Y)
     ==> F ∈ X guarantees Y"
```
⟨*proof*⟩

lemma *(***in** *Extend) extend_guarantees_eq:*
```
    "(extend h F ∈ (extend h ‘ X) guarantees (extend h ‘ Y)) =
     (F ∈ X guarantees Y)"
```
⟨*proof*⟩

end

11 Renaming of State Sets

theory *Rename* **imports** *Extend* **begin**

constdefs

```
  rename :: "['a => 'b, 'a program] => 'b program"
    "rename h == extend (%(x,u::unit). h x)"
```

declare *image_inv_f_f [simp] image_surj_f_inv_f [simp]*

declare *Extend.intro [simp,intro]*

lemma *good_map_bij [simp,intro]: "bij h ==> good_map (%(x,u). h x)"*
⟨*proof*⟩

lemma *fst_o_inv_eq_inv: "bij h ==> fst (inv (%(x,u). h x) s) = inv h s"*
⟨*proof*⟩

lemma *mem_rename_set_iff: "bij h ==> z ∈ h‘A = (inv h z ∈ A)"*
⟨*proof*⟩

lemma *extend_set_eq_image [simp]: "extend_set (%(x,u). h x) A = h‘A"*
⟨*proof*⟩

lemma *Init_rename [simp]: "Init (rename h F) = h‘(Init F)"*
⟨*proof*⟩

11.1 inverse properties

lemma *extend_set_inv:*
```
"bij h
 ==> extend_set (%(x,u::'c). inv h x) = project_set (%(x,u::'c). h x)"
```
⟨*proof*⟩

lemma *bij_extend_act_eq_project_act:* `"bij h`
```
 ==> extend_act (%(x,u::'c). h x) = project_act (%(x,u::'c). inv h x)"
```
⟨*proof*⟩

lemma *bij_extend_act:* `"bij h ==> bij (extend_act (%(x,u::'c). h x))"`
⟨*proof*⟩

lemma *bij_project_act:* `"bij h ==> bij (project_act (%(x,u::'c). h x))"`
⟨*proof*⟩

lemma *bij_inv_project_act_eq:* `"bij h ==> inv (project_act (%(x,u::'c). inv h x)) =`
```
                 project_act (%(x,u::'c). h x)"
```
⟨*proof*⟩

lemma *extend_inv:* `"bij h`
```
 ==> extend (%(x,u::'c). inv h x) = project (%(x,u::'c). h x) UNIV"
```
⟨*proof*⟩

lemma *rename_inv_rename [simp]:* `"bij h ==> rename (inv h) (rename h F) = F"`
⟨*proof*⟩

lemma *rename_rename_inv [simp]:* `"bij h ==> rename h (rename (inv h) F) = F"`
⟨*proof*⟩

lemma *rename_inv_eq:* `"bij h ==> rename (inv h) = inv (rename h)"`
⟨*proof*⟩

lemma *bij_extend:* `"bij h ==> bij (extend (%(x,u::'c). h x))"`
⟨*proof*⟩

lemma *bij_project:* `"bij h ==> bij (project (%(x,u::'c). h x) UNIV)"`
⟨*proof*⟩

lemma *inv_project_eq:*
```
"bij h
 ==> inv (project (%(x,u::'c). h x) UNIV) = extend (%(x,u::'c). h x)"
```
⟨*proof*⟩

lemma *Allowed_rename [simp]:*
```
"bij h ==> Allowed (rename h F) = rename h ` Allowed F"
```

⟨*proof*⟩

lemma *bij_rename: "bij h ==> bij (rename h)"*
⟨*proof*⟩
lemmas *surj_rename = bij_rename [THEN bij_is_surj, standard]*

lemma *inj_rename_imp_inj: "inj (rename h) ==> inj h"*
⟨*proof*⟩

lemma *surj_rename_imp_surj: "surj (rename h) ==> surj h"*
⟨*proof*⟩

lemma *bij_rename_imp_bij: "bij (rename h) ==> bij h"*
⟨*proof*⟩

lemma *bij_rename_iff [simp]: "bij (rename h) = bij h"*
⟨*proof*⟩

11.2 the lattice operations

lemma *rename_SKIP [simp]: "bij h ==> rename h SKIP = SKIP"*
⟨*proof*⟩

lemma *rename_Join [simp]:*
"bij h ==> rename h (F Join G) = rename h F Join rename h G"
⟨*proof*⟩

lemma *rename_JN [simp]:*
"bij h ==> rename h (JOIN I F) = ($\bigsqcup$i $\in$ I. rename h (F i))"
⟨*proof*⟩

11.3 Strong Safety: co, stable

lemma *rename_constrains:*
"bij h ==> (rename h F $\in$ (h‘A) co (h‘B)) = (F $\in$ A co B)"
⟨*proof*⟩

lemma *rename_stable:*
"bij h ==> (rename h F $\in$ stable (h‘A)) = (F $\in$ stable A)"
⟨*proof*⟩

lemma *rename_invariant:*
"bij h ==> (rename h F $\in$ invariant (h‘A)) = (F $\in$ invariant A)"
⟨*proof*⟩

lemma *rename_increasing:*
"bij h ==> (rename h F $\in$ increasing func) = (F $\in$ increasing (func o h))"
⟨*proof*⟩

11.4 Weak Safety: Co, Stable

lemma *reachable_rename_eq:*
"bij h ==> reachable (rename h F) = h ‘ (reachable F)"

⟨proof⟩

lemma *rename_Constrains:*
    "bij h ==> (rename h F ∈ (h‘A) Co (h‘B)) = (F ∈ A Co B)"
⟨proof⟩

lemma *rename_Stable:*
    "bij h ==> (rename h F ∈ Stable (h‘A)) = (F ∈ Stable A)"
⟨proof⟩

lemma *rename_Always: "bij h ==> (rename h F ∈ Always (h‘A)) = (F ∈ Always A)"*
⟨proof⟩

lemma *rename_Increasing:*
    "bij h ==> (rename h F ∈ Increasing func) = (F ∈ Increasing (func o h))"
⟨proof⟩

11.5 Progress: transient, ensures

lemma *rename_transient:*
    "bij h ==> (rename h F ∈ transient (h‘A)) = (F ∈ transient A)"
⟨proof⟩

lemma *rename_ensures:*
    "bij h ==> (rename h F ∈ (h‘A) ensures (h‘B)) = (F ∈ A ensures B)"
⟨proof⟩

lemma *rename_leadsTo:*
    "bij h ==> (rename h F ∈ (h‘A) leadsTo (h‘B)) = (F ∈ A leadsTo B)"
⟨proof⟩

lemma *rename_LeadsTo:*
    "bij h ==> (rename h F ∈ (h‘A) LeadsTo (h‘B)) = (F ∈ A LeadsTo B)"
⟨proof⟩

lemma *rename_rename_guarantees_eq:*
    "bij h ==> (rename h F ∈ (rename h ‘ X) guarantees
                (rename h ‘ Y)) =
        (F ∈ X guarantees Y)"
⟨proof⟩

lemma *rename_guarantees_eq_rename_inv:*
    "bij h ==> (rename h F ∈ X guarantees Y) =
        (F ∈ (rename (inv h) ‘ X) guarantees
            (rename (inv h) ‘ Y))"
⟨proof⟩

lemma *rename_preserves:*
    "bij h ==> (rename h G ∈ preserves v) = (G ∈ preserves (v o h))"
⟨proof⟩

lemma *ok_rename_iff [simp]: "bij h ==> (rename h F ok rename h G) = (F ok*

G)"
⟨*proof*⟩

lemma *OK_rename_iff [simp]: "bij h ==> OK I (%i. rename h (F i)) = (OK I F)"*
⟨*proof*⟩

11.6 "image" versions of the rules, for lifting "guarantees" properties

lemmas *bij_eq_rename = surj_rename [THEN surj_f_inv_f, symmetric]*

lemma *rename_image_constrains:*
"bij h ==> rename h ` (A co B) = (h ` A) co (h`B)"
⟨*proof*⟩

lemma *rename_image_stable: "bij h ==> rename h ` stable A = stable (h ` A)"*
⟨*proof*⟩

lemma *rename_image_increasing:*
"bij h ==> rename h ` increasing func = increasing (func o inv h)"
⟨*proof*⟩

lemma *rename_image_invariant:*
"bij h ==> rename h ` invariant A = invariant (h ` A)"
⟨*proof*⟩

lemma *rename_image_Constrains:*
"bij h ==> rename h ` (A Co B) = (h ` A) Co (h`B)"
⟨*proof*⟩

lemma *rename_image_preserves:*
"bij h ==> rename h ` preserves v = preserves (v o inv h)"
⟨*proof*⟩

lemma *rename_image_Stable:*
"bij h ==> rename h ` Stable A = Stable (h ` A)"
⟨*proof*⟩

lemma *rename_image_Increasing:*
"bij h ==> rename h ` Increasing func = Increasing (func o inv h)"
⟨*proof*⟩

lemma *rename_image_Always: "bij h ==> rename h ` Always A = Always (h ` A)"*
⟨*proof*⟩

lemma *rename_image_leadsTo:*
"bij h ==> rename h ` (A leadsTo B) = (h ` A) leadsTo (h`B)"
⟨*proof*⟩

lemma *rename_image_LeadsTo:*
"bij h ==> rename h ` (A LeadsTo B) = (h ` A) LeadsTo (h`B)"
⟨*proof*⟩

end

12 Replication of Components

theory *Lift_prog* **imports** *Rename* **begin**

constdefs

```
  insert_map :: "[nat, 'b, nat=>'b] => (nat=>'b)"
    "insert_map i z f k == if k<i then f k
                           else if k=i then z
                           else f(k - 1)"

  delete_map :: "[nat, nat=>'b] => (nat=>'b)"
    "delete_map i g k == if k<i then g k else g (Suc k)"

  lift_map :: "[nat, 'b * ((nat=>'b) * 'c)] => (nat=>'b) * 'c"
    "lift_map i == %(s,(f,uu)). (insert_map i s f, uu)"

  drop_map :: "[nat, (nat=>'b) * 'c] => 'b * ((nat=>'b) * 'c)"
    "drop_map i == %(g, uu). (g i, (delete_map i g, uu))"

  lift_set :: "[nat, ('b * ((nat=>'b) * 'c)) set] => ((nat=>'b) * 'c) set"
    "lift_set i A == lift_map i ` A"

  lift :: "[nat, ('b * ((nat=>'b) * 'c)) program] => ((nat=>'b) * 'c) program"
    "lift i == rename (lift_map i)"


  sub :: "['a, 'a=>'b] => 'b"
    "sub == %i f. f i"
```

declare *insert_map_def [simp] delete_map_def [simp]*

lemma *insert_map_inverse: "delete_map i (insert_map i x f) = f"*
⟨*proof*⟩

lemma *insert_map_delete_map_eq: "(insert_map i x (delete_map i g)) = g(i:=x)"*
⟨*proof*⟩

12.1 Injectiveness proof

lemma *insert_map_inject1: "(insert_map i x f) = (insert_map i y g) ==> x=y"*
⟨*proof*⟩

lemma *insert_map_inject2: "(insert_map i x f) = (insert_map i y g) ==> f=g"*
⟨*proof*⟩

lemma *insert_map_inject':*
    "(insert_map i x f) = (insert_map i y g) ==> x=y & f=g"
⟨*proof*⟩

```
lemmas insert_map_inject = insert_map_inject' [THEN conjE, elim!]
```

```
lemma lift_map_eq_iff [iff]:
     "(lift_map i (s,(f,uu)) = lift_map i' (s',(f',uu')))
      = (uu = uu' & insert_map i s f = insert_map i' s' f')"
```
⟨proof⟩

```
lemma drop_map_lift_map_eq [simp]: "!!s. drop_map i (lift_map i s) = s"
```
⟨proof⟩

```
lemma inj_lift_map: "inj (lift_map i)"
```
⟨proof⟩

12.2 Surjectiveness proof

```
lemma lift_map_drop_map_eq [simp]: "!!s. lift_map i (drop_map i s) = s"
```
⟨proof⟩

```
lemma drop_map_inject [dest!]: "(drop_map i s) = (drop_map i s') ==> s=s'"
```
⟨proof⟩

```
lemma surj_lift_map: "surj (lift_map i)"
```
⟨proof⟩

```
lemma bij_lift_map [iff]: "bij (lift_map i)"
```
⟨proof⟩

```
lemma inv_lift_map_eq [simp]: "inv (lift_map i) = drop_map i"
```
⟨proof⟩

```
lemma inv_drop_map_eq [simp]: "inv (drop_map i) = lift_map i"
```
⟨proof⟩

```
lemma bij_drop_map [iff]: "bij (drop_map i)"
```
⟨proof⟩

```
lemma sub_apply [simp]: "sub i f = f i"
```
⟨proof⟩

```
lemma all_total_lift: "all_total F ==> all_total (lift i F)"
```
⟨proof⟩

```
lemma insert_map_upd_same: "(insert_map i t f)(i := s) = insert_map i s f"
```
⟨proof⟩

```
lemma insert_map_upd:
     "(insert_map j t f)(i := s) =
      (if i=j then insert_map i s f
       else if i<j then insert_map j t (f(i:=s))
       else insert_map j t (f(i - Suc 0 := s)))"
```

⟨proof⟩

lemma *insert_map_eq_diff:*
```
"[| insert_map i s f = insert_map j t g;  i≠j |]
 ==> ∃g'. insert_map i s' f = insert_map j t g'"
```
⟨proof⟩

lemma *lift_map_eq_diff:*
```
"[| lift_map i (s,(f,uu)) = lift_map j (t,(g,vv));  i≠j |]
 ==> ∃g'. lift_map i (s',(f,uu)) = lift_map j (t,(g',vv))"
```
⟨proof⟩

12.3 The Operator lift_set

lemma *lift_set_empty [simp]: "lift_set i {} = {}"*
⟨proof⟩

lemma *lift_set_iff: "(lift_map i x ∈ lift_set i A) = (x ∈ A)"*
⟨proof⟩

lemma *lift_set_iff2 [iff]:*
```
"((f,uu) ∈ lift_set i A) = ((f i, (delete_map i f, uu)) ∈ A)"
```
⟨proof⟩

lemma *lift_set_mono: "A ⊆ B ==> lift_set i A ⊆ lift_set i B"*
⟨proof⟩

lemma *lift_set_Un_distrib: "lift_set i (A ∪ B) = lift_set i A ∪ lift_set i B"*
⟨proof⟩

lemma *lift_set_Diff_distrib: "lift_set i (A-B) = lift_set i A - lift_set i B"*
⟨proof⟩

12.4 The Lattice Operations

lemma *bij_lift [iff]: "bij (lift i)"*
⟨proof⟩

lemma *lift_SKIP [simp]: "lift i SKIP = SKIP"*
⟨proof⟩

lemma *lift_Join [simp]: "lift i (F Join G) = lift i F Join lift i G"*
⟨proof⟩

lemma *lift_JN [simp]: "lift j (JOIN I F) = (⨆i ∈ I. lift j (F i))"*
⟨proof⟩

12.5 Safety: constrains, stable, invariant

lemma *lift_constrains:*

```
    "(lift i F ∈ (lift_set i A) co (lift_set i B)) = (F ∈ A co B)"
```
⟨proof⟩

lemma *lift_stable:*
```
    "(lift i F ∈ stable (lift_set i A)) = (F ∈ stable A)"
```
⟨proof⟩

lemma *lift_invariant:*
```
    "(lift i F ∈ invariant (lift_set i A)) = (F ∈ invariant A)"
```
⟨proof⟩

lemma *lift_Constrains:*
```
    "(lift i F ∈ (lift_set i A) Co (lift_set i B)) = (F ∈ A Co B)"
```
⟨proof⟩

lemma *lift_Stable:*
```
    "(lift i F ∈ Stable (lift_set i A)) = (F ∈ Stable A)"
```
⟨proof⟩

lemma *lift_Always:*
```
    "(lift i F ∈ Always (lift_set i A)) = (F ∈ Always A)"
```
⟨proof⟩

12.6 Progress: transient, ensures

lemma *lift_transient:*
```
    "(lift i F ∈ transient (lift_set i A)) = (F ∈ transient A)"
```
⟨proof⟩

lemma *lift_ensures:*
```
    "(lift i F ∈ (lift_set i A) ensures (lift_set i B)) =
     (F ∈ A ensures B)"
```
⟨proof⟩

lemma *lift_leadsTo:*
```
    "(lift i F ∈ (lift_set i A) leadsTo (lift_set i B)) =
     (F ∈ A leadsTo B)"
```
⟨proof⟩

lemma *lift_LeadsTo:*
```
    "(lift i F ∈ (lift_set i A) LeadsTo (lift_set i B)) =
     (F ∈ A LeadsTo B)"
```
⟨proof⟩

lemma *lift_lift_guarantees_eq:*
```
    "(lift i F ∈ (lift i ` X) guarantees (lift i ` Y)) =
     (F ∈ X guarantees Y)"
```
⟨proof⟩

lemma *lift_guarantees_eq_lift_inv:*
```
    "(lift i F ∈ X guarantees Y) =
```

```
      (F ∈ (rename (drop_map i) ` X) guarantees (rename (drop_map i) ` Y))"
```
⟨proof⟩

lemma *lift_preserves_snd_I: "F ∈ preserves snd ==> lift i F ∈ preserves snd"*
⟨proof⟩

lemma *delete_map_eqE':*
```
     "(delete_map i g) = (delete_map i g') ==> ∃x. g = g'(i:=x)"
```
⟨proof⟩

lemmas *delete_map_eqE = delete_map_eqE' [THEN exE, elim!]*

lemma *delete_map_neq_apply:*
```
     "[| delete_map j g = delete_map j g';  i≠j |] ==> g i = g' i"
```
⟨proof⟩

lemma *vimage_o_fst_eq [simp]: "(f o fst) -` A = (f-`A) <*> UNIV"*
⟨proof⟩

lemma *vimage_sub_eq_lift_set [simp]:*
```
     "(sub i -`A) <*> UNIV = lift_set i (A <*> UNIV)"
```
⟨proof⟩

lemma *mem_lift_act_iff [iff]:*
```
     "((s,s') ∈ extend_act (%(x,u::unit). lift_map i x) act) =
      ((drop_map i s, drop_map i s') ∈ act)"
```
⟨proof⟩

lemma *preserves_snd_lift_stable:*
```
     "[| F ∈ preserves snd;  i≠j |]
      ==> lift j F ∈ stable (lift_set i (A <*> UNIV))"
```
⟨proof⟩

lemma *constrains_imp_lift_constrains:*
```
    "[| F i ∈ (A <*> UNIV) co (B <*> UNIV);
        F j ∈ preserves snd |]
     ==> lift j (F j) ∈ (lift_set i (A <*> UNIV)) co (lift_set i (B <*> UNIV))"
```
⟨proof⟩

lemma *lift_map_image_Times:*
```
     "lift_map i ` (A <*> UNIV) =
      (⋃s ∈ A. ⋃f. {insert_map i s f}) <*> UNIV"
```
⟨proof⟩

lemma *lift_preserves_eq:*
```
     "(lift i F ∈ preserves v) = (F ∈ preserves (v o lift_map i))"
```
⟨proof⟩

lemma *lift_preserves_sub:*
```
"F ∈ preserves snd
 ==> lift i F ∈ preserves (v o sub j o fst) =
      (if i=j then F ∈ preserves (v o fst) else True)"
```
⟨*proof*⟩

12.7 Lemmas to Handle Function Composition (o) More Consistently

lemma *o_equiv_assoc:* `"f o g = h ==> f' o f o g = f' o h"`
⟨*proof*⟩

lemma *o_equiv_apply:* `"f o g = h ==> ∀x. f(g x) = h x"`
⟨*proof*⟩

lemma *fst_o_lift_map:* `"sub i o fst o lift_map i = fst"`
⟨*proof*⟩

lemma *snd_o_lift_map:* `"snd o lift_map i = snd o snd"`
⟨*proof*⟩

12.8 More lemmas about extend and project

They could be moved to theory Extend or Project

lemma *extend_act_extend_act:*
```
"extend_act h' (extend_act h act) =
 extend_act (%(x,(y,y')). h'(h(x,y),y')) act"
```
⟨*proof*⟩

lemma *project_act_project_act:*
```
"project_act h (project_act h' act) =
 project_act (%(x,(y,y')). h'(h(x,y),y')) act"
```
⟨*proof*⟩

lemma *project_act_extend_act:*
```
"project_act h (extend_act h' act) =
   {(x,x'). ∃s s' y y' z. (s,s') ∈ act &
            h(x,y) = h'(s,z) & h(x',y') = h'(s',z)}"
```
⟨*proof*⟩

12.9 OK and "lift"

lemma *act_in_UNION_preserves_fst:*
```
"act ⊆ {(x,x'). fst x = fst x'} ==> act ∈ UNION (preserves fst) Acts"
```
⟨*proof*⟩

lemma *UNION_OK_lift_I:*
```
"[| ∀i ∈ I. F i ∈ preserves snd;
    ∀i ∈ I. UNION (preserves fst) Acts ⊆ AllowedActs (F i) |]
 ==> OK I (%i. lift i (F i))"
```
⟨*proof*⟩

lemma *OK_lift_I:*
```
     "[| ∀i ∈ I. F i ∈ preserves snd;
         ∀i ∈ I. preserves fst ⊆ Allowed (F i) |]
      ==> OK I (%i. lift i (F i))"
```
⟨*proof*⟩

lemma *Allowed_lift [simp]: "Allowed (lift i F) = lift i ` (Allowed F)"*
⟨*proof*⟩

lemma *lift_image_preserves:*
```
     "lift i ` preserves v = preserves (v o drop_map i)"
```
⟨*proof*⟩

end

theory *PPROD* **imports** *Lift_prog* **begin**

constdefs

```
  PLam  :: "[nat set, nat => ('b * ((nat=>'b) * 'c)) program]
            => ((nat=>'b) * 'c) program"
    "PLam I F == ⨆i ∈ I. lift i (F i)"
```

syntax
```
  "@PLam" :: "[pttrn, nat set, 'b set] => (nat => 'b) set"
              ("(3plam _:_./ _)" 10)
```

translations
```
  "plam x : A. B"   == "PLam A (%x. B)"
```

lemma *Init_PLam [simp]: "Init (PLam I F) = (⋂i ∈ I. lift_set i (Init (F i)))"*
⟨*proof*⟩

lemma *PLam_empty [simp]: "PLam {} F = SKIP"*
⟨*proof*⟩

lemma *PLam_SKIP [simp]: "(plam i : I. SKIP) = SKIP"*
⟨*proof*⟩

lemma *PLam_insert: "PLam (insert i I) F = (lift i (F i)) Join (PLam I F)"*
⟨*proof*⟩

lemma *PLam_component_iff: "((PLam I F) ≤ H) = (∀i ∈ I. lift i (F i) ≤ H)"*
⟨*proof*⟩

lemma *component_PLam: "i ∈ I ==> lift i (F i) ≤ (PLam I F)"*

⟨*proof*⟩

lemma *PLam_constrains:*
```
"[| i ∈ I;  ∀j. F j ∈ preserves snd |]
 ==> (PLam I F ∈ (lift_set i (A <*> UNIV)) co
                 (lift_set i (B <*> UNIV)))  =
     (F i ∈ (A <*> UNIV) co (B <*> UNIV))"
```
⟨*proof*⟩

lemma *PLam_stable:*
```
"[| i ∈ I;  ∀j. F j ∈ preserves snd |]
 ==> (PLam I F ∈ stable (lift_set i (A <*> UNIV))) =
     (F i ∈ stable (A <*> UNIV))"
```
⟨*proof*⟩

lemma *PLam_transient:*
```
"i ∈ I ==>
PLam I F ∈ transient A = (∃i ∈ I. lift i (F i) ∈ transient A)"
```
⟨*proof*⟩

This holds because the `F j` cannot change `lift_set i`

lemma *PLam_ensures:*
```
"[| i ∈ I;  F i ∈ (A <*> UNIV) ensures (B <*> UNIV);
    ∀j. F j ∈ preserves snd |]
 ==> PLam I F ∈ lift_set i (A <*> UNIV) ensures lift_set i (B <*> UNIV)"
```
⟨*proof*⟩

lemma *PLam_leadsTo_Basis:*
```
"[| i ∈ I;
    F i ∈ ((A <*> UNIV) - (B <*> UNIV)) co
          ((A <*> UNIV) ∪ (B <*> UNIV));
    F i ∈ transient ((A <*> UNIV) - (B <*> UNIV));
    ∀j. F j ∈ preserves snd |]
 ==> PLam I F ∈ lift_set i (A <*> UNIV) leadsTo lift_set i (B <*> UNIV)"
```
⟨*proof*⟩

lemma *invariant_imp_PLam_invariant:*
```
"[| F i ∈ invariant (A <*> UNIV);  i ∈ I;
    ∀j. F j ∈ preserves snd |]
 ==> PLam I F ∈ invariant (lift_set i (A <*> UNIV))"
```
⟨*proof*⟩

lemma *PLam_preserves_fst [simp]:*
```
"∀j. F j ∈ preserves snd
 ==> (PLam I F ∈ preserves (v o sub j o fst)) =
     (if j ∈ I then F j ∈ preserves (v o fst) else True)"
```

⟨*proof*⟩

lemma *PLam_preserves_snd [simp,intro]:*
```
     "∀j. F j ∈ preserves snd ==> PLam I F ∈ preserves snd"
```
⟨*proof*⟩

This rule looks unsatisfactory because it refers to *lift*. One must use *lift_guarantees_eq_lift_inv* to rewrite the first subgoal and something like *lift_preserves_sub* to rewrite the third. However there's no obvious alternative for the third premise.

lemma *guarantees_PLam_I:*
```
    "[| lift i (F i): X guarantees Y;  i ∈ I;
        OK I (%i. lift i (F i)) |]
     ==> (PLam I F) ∈ X guarantees Y"
```
⟨*proof*⟩

lemma *Allowed_PLam [simp]:*
```
     "Allowed (PLam I F) = (⋂i ∈ I. lift i ` Allowed(F i))"
```
⟨*proof*⟩

lemma *PLam_preserves [simp]:*
```
     "(PLam I F) ∈ preserves v = (∀i ∈ I. F i ∈ preserves (v o lift_map
i))"
```
⟨*proof*⟩

end

13 The Prefix Ordering on Lists

theory *ListOrder* **imports** *Main* **begin**

inductive_set
```
  genPrefix :: "('a * 'a)set => ('a list * 'a list)set"
  for r :: "('a * 'a)set"
 where
   Nil:     "([],[]) : genPrefix(r)"

 | prepend: "[| (xs,ys) : genPrefix(r);  (x,y) : r |] ==>
             (x#xs, y#ys) : genPrefix(r)"

 | append:  "(xs,ys) : genPrefix(r) ==> (xs, ys@zs) : genPrefix(r)"
```

instance *list :: (type)ord* ⟨*proof*⟩

defs

```
  prefix_def:        "xs <= zs  ==  (xs,zs) : genPrefix Id"

  strict_prefix_def: "xs < zs  ==  xs <= zs & xs ~= (zs::'a list)"
```

constdefs

```
  Le :: "(nat*nat) set"
    "Le == {(x,y). x <= y}"

  Ge :: "(nat*nat) set"
    "Ge == {(x,y). y <= x}"
```

abbreviation
pfixLe :: "[nat list, nat list] => bool" (**infixl** *"pfixLe" 50)* **where**
"xs pfixLe ys == (xs,ys) : genPrefix Le"

abbreviation
pfixGe :: "[nat list, nat list] => bool" (**infixl** *"pfixGe" 50)* **where**
"xs pfixGe ys == (xs,ys) : genPrefix Ge"

13.1 preliminary lemmas

lemma *Nil_genPrefix [iff]: "([], xs) : genPrefix r"*
⟨*proof*⟩

lemma *genPrefix_length_le: "(xs,ys) : genPrefix r ==> length xs <= length ys"*
⟨*proof*⟩

lemma *cdlemma:*

```
     "[| (xs', ys'): genPrefix r |]
      ==> (ALL x xs. xs' = x#xs --> (EX y ys. ys' = y#ys & (x,y) : r & (xs,
ys) : genPrefix r))"
```

⟨*proof*⟩

lemma *cons_genPrefixE [elim!]:*

```
     "[| (x#xs, zs): genPrefix r;
         !!y ys. [| zs = y#ys;  (x,y) : r;  (xs, ys) : genPrefix r |] ==> P

      |] ==> P"
```

⟨*proof*⟩

lemma *Cons_genPrefix_Cons [iff]:*

```
     "((x#xs,y#ys) : genPrefix r) = ((x,y) : r & (xs,ys) : genPrefix r)"
```

⟨*proof*⟩

13.2 genPrefix is a partial order

lemma *refl_genPrefix: "reflexive r ==> reflexive (genPrefix r)"*

⟨*proof*⟩

lemma *genPrefix_refl [simp]: "reflexive r ==> (l,l) : genPrefix r"*
⟨*proof*⟩

lemma *genPrefix_mono: "r<=s ==> genPrefix r <= genPrefix s"*
⟨*proof*⟩

lemma *append_genPrefix [rule_format]:*
```
"ALL zs. (xs @ ys, zs) : genPrefix r --> (xs, zs) : genPrefix r"
```
⟨*proof*⟩

lemma *genPrefix_trans_O [rule_format]:*
```
"(x, y) : genPrefix r
 ==> ALL z. (y,z) : genPrefix s --> (x, z) : genPrefix (s O r)"
```
⟨*proof*⟩

lemma *genPrefix_trans [rule_format]:*
```
"[| (x,y) : genPrefix r;  (y,z) : genPrefix r;  trans r |]
 ==> (x,z) : genPrefix r"
```
⟨*proof*⟩

lemma *prefix_genPrefix_trans [rule_format]:*
```
"[| x<=y;  (y,z) : genPrefix r |] ==> (x, z) : genPrefix r"
```
⟨*proof*⟩

lemma *genPrefix_prefix_trans [rule_format]:*
```
"[| (x,y) : genPrefix r;  y<=z |] ==> (x,z) : genPrefix r"
```
⟨*proof*⟩

lemma *trans_genPrefix: "trans r ==> trans (genPrefix r)"*
⟨*proof*⟩

lemma *genPrefix_antisym [rule_format]:*
```
"[| (xs,ys) : genPrefix r;  antisym r |]
 ==> (ys,xs) : genPrefix r --> xs = ys"
```
⟨*proof*⟩

lemma *antisym_genPrefix: "antisym r ==> antisym (genPrefix r)"*
⟨*proof*⟩

13.3 recursion equations

lemma *genPrefix_Nil [simp]: "((xs, []) : genPrefix r) = (xs = [])"*
⟨*proof*⟩

lemma *same_genPrefix_genPrefix [simp]:*
```
"reflexive r ==> ((xs@ys, xs@zs) : genPrefix r) = ((ys,zs) : genPrefix
```

r)"
⟨*proof*⟩

lemma *genPrefix_Cons:*
```
"((xs, y#ys) : genPrefix r) =
 (xs=[] | (EX z zs. xs=z#zs & (z,y) : r & (zs,ys) : genPrefix r))"
```
⟨*proof*⟩

lemma *genPrefix_take_append:*
```
"[| reflexive r;  (xs,ys) : genPrefix r |]
 ==>  (xs@zs, take (length xs) ys @ zs) : genPrefix r"
```
⟨*proof*⟩

lemma *genPrefix_append_both:*
```
"[| reflexive r;  (xs,ys) : genPrefix r;  length xs = length ys |]
 ==>  (xs@zs, ys @ zs) : genPrefix r"
```
⟨*proof*⟩

lemma *append_cons_eq: "xs @ y # ys = (xs @ [y]) @ ys"*
⟨*proof*⟩

lemma *aolemma:*
```
"[| (xs,ys) : genPrefix r;  reflexive r |]
 ==> length xs < length ys --> (xs @ [ys ! length xs], ys) : genPrefix
r"
```
⟨*proof*⟩

lemma *append_one_genPrefix:*
```
"[| (xs,ys) : genPrefix r;  length xs < length ys;  reflexive r |]
 ==> (xs @ [ys ! length xs], ys) : genPrefix r"
```
⟨*proof*⟩

lemma *genPrefix_imp_nth [rule_format]:*
```
"ALL i ys. i < length xs
           --> (xs, ys) : genPrefix r --> (xs ! i, ys ! i) : r"
```
⟨*proof*⟩

lemma *nth_imp_genPrefix [rule_format]:*
```
"ALL ys. length xs <= length ys
 --> (ALL i. i < length xs --> (xs ! i, ys ! i) : r)
 --> (xs, ys) : genPrefix r"
```
⟨*proof*⟩

lemma *genPrefix_iff_nth:*
```
"((xs,ys) : genPrefix r) =
 (length xs <= length ys & (ALL i. i < length xs --> (xs!i, ys!i) : r))"
```
⟨*proof*⟩

13.4 The type of lists is partially ordered

declare *reflexive_Id [iff]*
antisym_Id [iff]
trans_Id [iff]

lemma *prefix_refl [iff]: "xs <= (xs::'a list)"*
⟨*proof*⟩

lemma *prefix_trans: "!!xs::'a list. [| xs <= ys; ys <= zs |] ==> xs <= zs"*
⟨*proof*⟩

lemma *prefix_antisym: "!!xs::'a list. [| xs <= ys; ys <= xs |] ==> xs = ys"*
⟨*proof*⟩

lemma *prefix_less_le: "!!xs::'a list. (xs < zs) = (xs <= zs & xs ~= zs)"*
⟨*proof*⟩

instance *list :: (type) order*
⟨*proof*⟩

lemma *set_mono: "xs <= ys ==> set xs <= set ys"*
⟨*proof*⟩

lemma *Nil_prefix [iff]: "[] <= xs"*
⟨*proof*⟩

lemma *prefix_Nil [simp]: "(xs <= []) = (xs = [])"*
⟨*proof*⟩

lemma *Cons_prefix_Cons [simp]: "(x#xs <= y#ys) = (x=y & xs<=ys)"*
⟨*proof*⟩

lemma *same_prefix_prefix [simp]: "(xs@ys <= xs@zs) = (ys <= zs)"*
⟨*proof*⟩

lemma *append_prefix [iff]: "(xs@ys <= xs) = (ys <= [])"*
⟨*proof*⟩

lemma *prefix_appendI [simp]: "xs <= ys ==> xs <= ys@zs"*
⟨*proof*⟩

lemma *prefix_Cons:*
"(xs <= y#ys) = (xs=[] | (? zs. xs=y#zs & zs <= ys))"
⟨*proof*⟩

lemma *append_one_prefix:*
"[| xs <= ys; length xs < length ys |] ==> xs @ [ys ! length xs] <= ys"
⟨*proof*⟩

lemma *prefix_length_le: "xs <= ys ==> length xs <= length ys"*
⟨*proof*⟩

lemma *splemma: "xs<=ys ==> xs~=ys --> length xs < length ys"*
⟨*proof*⟩

lemma *strict_prefix_length_less: "xs < ys ==> length xs < length ys"*
⟨*proof*⟩

lemma *mono_length: "mono length"*
⟨*proof*⟩

lemma *prefix_iff: "(xs <= zs) = (EX ys. zs = xs@ys)"*
⟨*proof*⟩

lemma *prefix_snoc [simp]: "(xs <= ys@[y]) = (xs = ys@[y] | xs <= ys)"*
⟨*proof*⟩

lemma *prefix_append_iff:*
"(xs <= ys@zs) = (xs <= ys | (? us. xs = ys@us & us <= zs))"
⟨*proof*⟩

lemma *common_prefix_linear [rule_format]:*
"!!zs::'a list. xs <= zs --> ys <= zs --> xs <= ys | ys <= xs"
⟨*proof*⟩

13.5 pfixLe, pfixGe: properties inherited from the translations

lemma *reflexive_Le [iff]: "reflexive Le"*
⟨*proof*⟩

lemma *antisym_Le [iff]: "antisym Le"*
⟨*proof*⟩

lemma *trans_Le [iff]: "trans Le"*
⟨*proof*⟩

lemma *pfixLe_refl [iff]: "x pfixLe x"*
⟨*proof*⟩

lemma *pfixLe_trans: "[| x pfixLe y; y pfixLe z |] ==> x pfixLe z"*
⟨*proof*⟩

lemma *pfixLe_antisym: "[| x pfixLe y; y pfixLe x |] ==> x = y"*
⟨*proof*⟩

lemma *prefix_imp_pfixLe: "xs<=ys ==> xs pfixLe ys"*
⟨*proof*⟩

lemma *reflexive_Ge [iff]: "reflexive Ge"*
⟨*proof*⟩

lemma *antisym_Ge [iff]: "antisym Ge"*
⟨*proof*⟩

lemma *trans_Ge [iff]: "trans Ge"*
⟨*proof*⟩

lemma *pfixGe_refl [iff]: "x pfixGe x"*
⟨*proof*⟩

lemma *pfixGe_trans: "[| x pfixGe y; y pfixGe z |] ==> x pfixGe z"*
⟨*proof*⟩

lemma *pfixGe_antisym: "[| x pfixGe y; y pfixGe x |] ==> x = y"*
⟨*proof*⟩

lemma *prefix_imp_pfixGe: "xs<=ys ==> xs pfixGe ys"*
⟨*proof*⟩

end

14 Multisets

theory *Multiset*
imports *Main*
begin

14.1 The type of multisets

typedef *'a multiset = "{f::'a => nat. finite {x . f x > 0}}"*
⟨*proof*⟩

lemmas *multiset_typedef [simp] =*
    Abs_multiset_inverse Rep_multiset_inverse Rep_multiset
  and *[simp] = Rep_multiset_inject [symmetric]*

definition
  Mempty :: "'a multiset" ("{#}") **where**
  "{#} = Abs_multiset (λa. 0)"

definition
  single :: "'a => 'a multiset" ("{#_#}") **where**
  "{#a#} = Abs_multiset (λb. if b = a then 1 else 0)"

definition
  count :: "'a multiset => 'a => nat" **where**
  "count = Rep_multiset"

definition
  MCollect :: "'a multiset => ('a => bool) => 'a multiset" **where**
  "MCollect M P = Abs_multiset (λx. if P x then Rep_multiset M x else 0)"

abbreviation
```
  Melem :: "'a => 'a multiset => bool"  ("(_/ :# _)" [50, 51] 50) where
  "a :# M == count M a > 0"
```

syntax
```
  "_MCollect" :: "pttrn => 'a multiset => bool => 'a multiset"    ("(1{# _
: _./ _#})")
```
translations
```
  "{#x:M. P#}" == "CONST MCollect M (λx. P)"
```

definition
```
  set_of :: "'a multiset => 'a set" where
  "set_of M = {x. x :# M}"
```

instance `multiset :: (type) "{plus, minus, zero, size}"`
```
  union_def: "M + N == Abs_multiset (λa. Rep_multiset M a + Rep_multiset N
a)"
  diff_def: "M - N == Abs_multiset (λa. Rep_multiset M a - Rep_multiset N
a)"
  Zero_multiset_def [simp]: "0 == {#}"
  size_def: "size M == setsum (count M) (set_of M)" ⟨proof⟩
```

definition
```
  multiset_inter :: "'a multiset ⇒ 'a multiset ⇒ 'a multiset"  (infixl "#∩"
70) where
  "multiset_inter A B = A - (A - B)"
```

Preservation of the representing set `multiset`.

lemma `const0_in_multiset [simp]: "(λa. 0) ∈ multiset"`
  ⟨*proof*⟩

lemma `only1_in_multiset [simp]: "(λb. if b = a then 1 else 0) ∈ multiset"`
  ⟨*proof*⟩

lemma `union_preserves_multiset [simp]:`
```
    "M ∈ multiset ==> N ∈ multiset ==> (λa. M a + N a) ∈ multiset"
```
  ⟨*proof*⟩

lemma `diff_preserves_multiset [simp]:`
```
    "M ∈ multiset ==> (λa. M a - N a) ∈ multiset"
```
  ⟨*proof*⟩

14.2 Algebraic properties of multisets

14.2.1 Union

lemma `union_empty [simp]: "M + {#} = M ∧ {#} + M = M"`
  ⟨*proof*⟩

lemma `union_commute: "M + N = N + (M::'a multiset)"`
  ⟨*proof*⟩

lemma `union_assoc: "(M + N) + K = M + (N + (K::'a multiset))"`
  ⟨*proof*⟩

lemma *union_lcomm: "M + (N + K) = N + (M + (K::'a multiset))"*
⟨*proof*⟩

lemmas *union_ac = union_assoc union_commute union_lcomm*

instance *multiset :: (type) comm_monoid_add*
⟨*proof*⟩

14.2.2 Difference

lemma *diff_empty [simp]: "M - {#} = M ∧ {#} - M = {#}"*
⟨*proof*⟩

lemma *diff_union_inverse2 [simp]: "M + {#a#} - {#a#} = M"*
⟨*proof*⟩

14.2.3 Count of elements

lemma *count_empty [simp]: "count {#} a = 0"*
⟨*proof*⟩

lemma *count_single [simp]: "count {#b#} a = (if b = a then 1 else 0)"*
⟨*proof*⟩

lemma *count_union [simp]: "count (M + N) a = count M a + count N a"*
⟨*proof*⟩

lemma *count_diff [simp]: "count (M - N) a = count M a - count N a"*
⟨*proof*⟩

14.2.4 Set of elements

lemma *set_of_empty [simp]: "set_of {#} = {}"*
⟨*proof*⟩

lemma *set_of_single [simp]: "set_of {#b#} = {b}"*
⟨*proof*⟩

lemma *set_of_union [simp]: "set_of (M + N) = set_of M ∪ set_of N"*
⟨*proof*⟩

lemma *set_of_eq_empty_iff [simp]: "(set_of M = {}) = (M = {#})"*
⟨*proof*⟩

lemma *mem_set_of_iff [simp]: "(x ∈ set_of M) = (x :# M)"*
⟨*proof*⟩

14.2.5 Size

lemma *size_empty [simp]: "size {#} = 0"*
⟨*proof*⟩

lemma *size_single [simp]: "size {#b#} = 1"*
⟨*proof*⟩

lemma *finite_set_of [iff]: "finite (set_of M)"*
⟨*proof*⟩

lemma *setsum_count_Int:*
"finite A ==> setsum (count N) (A ∩ set_of N) = setsum (count N) A"
⟨*proof*⟩

lemma *size_union [simp]: "size (M + N::'a multiset) = size M + size N"*
⟨*proof*⟩

lemma *size_eq_0_iff_empty [iff]: "(size M = 0) = (M = {#})"*
⟨*proof*⟩

lemma *size_eq_Suc_imp_elem: "size M = Suc n ==> ∃a. a :# M"*
⟨*proof*⟩

14.2.6 Equality of multisets

lemma *multiset_eq_conv_count_eq: "(M = N) = (∀a. count M a = count N a)"*
⟨*proof*⟩

lemma *single_not_empty [simp]: "{#a#} ≠ {#} ∧ {#} ≠ {#a#}"*
⟨*proof*⟩

lemma *single_eq_single [simp]: "({#a#} = {#b#}) = (a = b)"*
⟨*proof*⟩

lemma *union_eq_empty [iff]: "(M + N = {#}) = (M = {#} ∧ N = {#})"*
⟨*proof*⟩

lemma *empty_eq_union [iff]: "({#} = M + N) = (M = {#} ∧ N = {#})"*
⟨*proof*⟩

lemma *union_right_cancel [simp]: "(M + K = N + K) = (M = (N::'a multiset))"*
⟨*proof*⟩

lemma *union_left_cancel [simp]: "(K + M = K + N) = (M = (N::'a multiset))"*
⟨*proof*⟩

lemma *union_is_single:*
"(M + N = {#a#}) = (M = {#a#} ∧ N={#} ∨ M = {#} ∧ N = {#a#})"
⟨*proof*⟩

lemma *single_is_union:*
"({#a#} = M + N) = ({#a#} = M ∧ N = {#} ∨ M = {#} ∧ {#a#} = N)"
⟨*proof*⟩

lemma *add_eq_conv_diff:*
"(M + {#a#} = N + {#b#}) =
(M = N ∧ a = b ∨ M = N - {#a#} + {#b#} ∧ N = M - {#b#} + {#a#})"
⟨*proof*⟩

declare *Rep_multiset_inject [symmetric, simp del]*

instance *multiset :: (type) cancel_ab_semigroup_add*
⟨*proof*⟩

14.2.7 Intersection

lemma *multiset_inter_count:*
    "count (A #∩ B) x = min (count A x) (count B x)"
  ⟨*proof*⟩

lemma *multiset_inter_commute: "A #∩ B = B #∩ A"*
  ⟨*proof*⟩

lemma *multiset_inter_assoc: "A #∩ (B #∩ C) = A #∩ B #∩ C"*
  ⟨*proof*⟩

lemma *multiset_inter_left_commute: "A #∩ (B #∩ C) = B #∩ (A #∩ C)"*
  ⟨*proof*⟩

lemmas *multiset_inter_ac =*
  multiset_inter_commute
  multiset_inter_assoc
  multiset_inter_left_commute

lemma *multiset_union_diff_commute: "B #∩ C = {#} ⟹ A + B - C = A - C + B"*
  ⟨*proof*⟩

14.3 Induction over multisets

lemma *setsum_decr:*
  "finite F ==> (0::nat) < f a ==>
    setsum (f (a := f a - 1)) F = (if a∈F then setsum f F - 1 else setsum f F)"
  ⟨*proof*⟩

lemma *rep_multiset_induct_aux:*
  assumes *1: "P (λa. (0::nat))"*
    and *2: "!!f b. f ∈ multiset ==> P f ==> P (f (b := f b + 1))"*
  shows *"∀f. f ∈ multiset --> setsum f {x. f x ≠ 0} = n --> P f"*
  ⟨*proof*⟩

theorem *rep_multiset_induct:*
  "f ∈ multiset ==> P (λa. 0) ==>
    (!!f b. f ∈ multiset ==> P f ==> P (f (b := f b + 1))) ==> P f"
  ⟨*proof*⟩

theorem *multiset_induct [case_names empty add, induct type: multiset]:*
  assumes *empty: "P {#}"*
    and *add: "!!M x. P M ==> P (M + {#x#})"*
  shows *"P M"*
⟨*proof*⟩

lemma *MCollect_preserves_multiset:*

"M ∈ multiset ==> (λx. if P x then M x else 0) ∈ multiset"
⟨*proof*⟩

lemma *count_MCollect [simp]:*
"count {# x:M. P x #} a = (if P a then count M a else 0)"
⟨*proof*⟩

lemma *set_of_MCollect [simp]: "set_of {# x:M. P x #} = set_of M ∩ {x. P x}"*
⟨*proof*⟩

lemma *multiset_partition: "M = {# x:M. P x #} + {# x:M. ¬ P x #}"*
⟨*proof*⟩

lemma *add_eq_conv_ex:*
"(M + {#a#} = N + {#b#}) =
(M = N ∧ a = b ∨ (∃K. M = K + {#b#} ∧ N = K + {#a#}))"
⟨*proof*⟩

declare *multiset_typedef [simp del]*

14.4 Multiset orderings

14.4.1 Well-foundedness

definition
mult1 :: "('a × 'a) set => ('a multiset × 'a multiset) set" **where**
"mult1 r =
{(N, M). ∃a M0 K. M = M0 + {#a#} ∧ N = M0 + K ∧
(∀b. b :# K --> (b, a) ∈ r)}"

definition
mult :: "('a × 'a) set => ('a multiset × 'a multiset) set" **where**
"mult r = (mult1 r)$^+$"

lemma *not_less_empty [iff]: "(M, {#}) ∉ mult1 r"*
⟨*proof*⟩

lemma *less_add: "(N, M0 + {#a#}) ∈ mult1 r ==>*
(∃M. (M, M0) ∈ mult1 r ∧ N = M + {#a#}) ∨
(∃K. (∀b. b :# K --> (b, a) ∈ r) ∧ N = M0 + K)"
(**is** "_ ⟹ ?case1 (mult1 r) ∨ ?case2")
⟨*proof*⟩

lemma *all_accessible: "wf r ==> ∀M. M ∈ acc (mult1 r)"*
⟨*proof*⟩

theorem *wf_mult1: "wf r ==> wf (mult1 r)"*
⟨*proof*⟩

theorem *wf_mult: "wf r ==> wf (mult r)"*
⟨*proof*⟩

14.4.2 Closure-free presentation

lemma *diff_union_single_conv: "a :# J ==> I + J - {#a#} = I + (J - {#a#})"*
⟨*proof*⟩

One direction.

lemma *mult_implies_one_step:*
"trans r ==> (M, N) ∈ mult r ==>
∃ I J K. N = I + J ∧ M = I + K ∧ J ≠ {#} ∧
(∀ k ∈ set_of K. ∃ j ∈ set_of J. (k, j) ∈ r)"
⟨*proof*⟩

lemma *elem_imp_eq_diff_union: "a :# M ==> M = M - {#a#} + {#a#}"*
⟨*proof*⟩

lemma *size_eq_Suc_imp_eq_union: "size M = Suc n ==> ∃ a N. M = N + {#a#}"*
⟨*proof*⟩

lemma *one_step_implies_mult_aux:*
"trans r ==>
∀ I J K. (size J = n ∧ J ≠ {#} ∧ (∀ k ∈ set_of K. ∃ j ∈ set_of J. (k, j) ∈ r))
--> (I + K, I + J) ∈ mult r"
⟨*proof*⟩

lemma *one_step_implies_mult:*
"trans r ==> J ≠ {#} ==> ∀ k ∈ set_of K. ∃ j ∈ set_of J. (k, j) ∈ r
==> (I + K, I + J) ∈ mult r"
⟨*proof*⟩

14.4.3 Partial-order properties

instance *multiset :: (type) ord* ⟨*proof*⟩

defs *(***overloaded***)*
less_multiset_def: "M' < M == (M', M) ∈ mult {(x', x). x' < x}"
le_multiset_def: "M' <= M == M' = M ∨ M' < (M::'a multiset)"

lemma *trans_base_order: "trans {(x', x). x' < (x::'a::order)}"*
⟨*proof*⟩

Irreflexivity.

lemma *mult_irrefl_aux:*
"finite A ==> (∀ x ∈ A. ∃ y ∈ A. x < (y::'a::order)) ⟹ A = {}"
⟨*proof*⟩

lemma *mult_less_not_refl: "¬ M < (M::'a::order multiset)"*
⟨*proof*⟩

lemma *mult_less_irrefl [elim!]: "M < (M::'a::order multiset) ==> R"*
⟨*proof*⟩

Transitivity.

theorem *mult_less_trans: "K < M ==> M < N ==> K < (N::'a::order multiset)"*

⟨*proof*⟩

Asymmetry.

theorem *mult_less_not_sym: "M < N ==> ¬ N < (M::'a::order multiset)"*
⟨*proof*⟩

theorem *mult_less_asym:*
"M < N ==> (¬ P ==> N < (M::'a::order multiset)) ==> P"
⟨*proof*⟩

theorem *mult_le_refl [iff]: "M <= (M::'a::order multiset)"*
⟨*proof*⟩

Anti-symmetry.

theorem *mult_le_antisym:*
"M <= N ==> N <= M ==> M = (N::'a::order multiset)"
⟨*proof*⟩

Transitivity.

theorem *mult_le_trans:*
"K <= M ==> M <= N ==> K <= (N::'a::order multiset)"
⟨*proof*⟩

theorem *mult_less_le: "(M < N) = (M <= N ∧ M ≠ (N::'a::order multiset))"*
⟨*proof*⟩

Partial order.

instance *multiset :: (order) order*
⟨*proof*⟩

14.4.4 Monotonicity of multiset union

lemma *mult1_union:*
"(B, D) ∈ mult1 r ==> trans r ==> (C + B, C + D) ∈ mult1 r"
⟨*proof*⟩

lemma *union_less_mono2: "B < D ==> C + B < C + (D::'a::order multiset)"*
⟨*proof*⟩

lemma *union_less_mono1: "B < D ==> B + C < D + (C::'a::order multiset)"*
⟨*proof*⟩

lemma *union_less_mono:*
"A < C ==> B < D ==> A + B < C + (D::'a::order multiset)"
⟨*proof*⟩

lemma *union_le_mono:*
"A <= C ==> B <= D ==> A + B <= C + (D::'a::order multiset)"
⟨*proof*⟩

lemma *empty_leI [iff]: "{#} <= (M::'a::order multiset)"*
⟨*proof*⟩

lemma *union_upper1: "A <= A + (B::'a::order multiset)"*
⟨*proof*⟩

lemma *union_upper2: "B <= A + (B::'a::order multiset)"*
⟨*proof*⟩

instance *multiset :: (order) pordered_ab_semigroup_add*
⟨*proof*⟩

14.5 Link with lists

consts
multiset_of :: "'a list ⇒ 'a multiset"
primrec
"multiset_of [] = {#}"
"multiset_of (a # x) = multiset_of x + {# a #}"

lemma *multiset_of_zero_iff[simp]: "(multiset_of x = {#}) = (x = [])"*
⟨*proof*⟩

lemma *multiset_of_zero_iff_right[simp]: "({#} = multiset_of x) = (x = [])"*
⟨*proof*⟩

lemma *set_of_multiset_of[simp]: "set_of(multiset_of x) = set x"*
⟨*proof*⟩

lemma *mem_set_multiset_eq: "x ∈ set xs = (x :# multiset_of xs)"*
⟨*proof*⟩

lemma *multiset_of_append [simp]:*
"multiset_of (xs @ ys) = multiset_of xs + multiset_of ys"
⟨*proof*⟩

lemma *surj_multiset_of: "surj multiset_of"*
⟨*proof*⟩

lemma *set_count_greater_0: "set x = {a. count (multiset_of x) a > 0}"*
⟨*proof*⟩

lemma *distinct_count_atmost_1:*
"distinct x = (! a. count (multiset_of x) a = (if a ∈ set x then 1 else 0))"
⟨*proof*⟩

lemma *multiset_of_eq_setD:*
"multiset_of xs = multiset_of ys ⟹ set xs = set ys"
⟨*proof*⟩

lemma *set_eq_iff_multiset_of_eq_distinct:*
"⟦distinct x; distinct y⟧
⟹ (set x = set y) = (multiset_of x = multiset_of y)"
⟨*proof*⟩

lemma *set_eq_iff_multiset_of_remdups_eq:*

```
  "(set x = set y) = (multiset_of (remdups x) = multiset_of (remdups y))"
  ⟨proof⟩
```

lemma *multiset_of_compl_union [simp]:*

```
    "multiset_of [x←xs. P x] + multiset_of [x←xs. ¬P x] = multiset_of xs"
  ⟨proof⟩
```

lemma *count_filter:*

```
    "count (multiset_of xs) x = length [y ← xs. y = x]"
  ⟨proof⟩
```

14.6 Pointwise ordering induced by count

definition
mset_le :: "'a multiset ⇒ 'a multiset ⇒ bool" (**infix** *"≤#" 50)* **where**
"(A ≤# B) = (∀a. count A a ≤ count B a)"
definition
mset_less :: "'a multiset ⇒ 'a multiset ⇒ bool" (**infix** *"<#" 50)* **where**
"(A <# B) = (A ≤# B ∧ A ≠ B)"

lemma *mset_le_refl[simp]: "A ≤# A"*
⟨*proof*⟩

lemma *mset_le_trans: "⟦ A ≤# B; B ≤# C ⟧ ⟹ A ≤# C"*
⟨*proof*⟩

lemma *mset_le_antisym: "⟦ A ≤# B; B ≤# A ⟧ ⟹ A = B"*
⟨*proof*⟩

lemma *mset_le_exists_conv:*
"(A ≤# B) = (∃C. B = A + C)"
⟨*proof*⟩

lemma *mset_le_mono_add_right_cancel[simp]: "(A + C ≤# B + C) = (A ≤# B)"*
⟨*proof*⟩

lemma *mset_le_mono_add_left_cancel[simp]: "(C + A ≤# C + B) = (A ≤# B)"*
⟨*proof*⟩

lemma *mset_le_mono_add: "⟦ A ≤# B; C ≤# D ⟧ ⟹ A + C ≤# B + D"*
⟨*proof*⟩

lemma *mset_le_add_left[simp]: "A ≤# A + B"*
⟨*proof*⟩

lemma *mset_le_add_right[simp]: "B ≤# A + B"*
⟨*proof*⟩

lemma *multiset_of_remdups_le: "multiset_of (remdups xs) ≤# multiset_of xs"*
⟨*proof*⟩

interpretation *mset_order:*
order ["op ≤#" "op <#"]
⟨*proof*⟩

interpretation *mset_order_cancel_semigroup:*

```
  pordered_cancel_ab_semigroup_add ["op ≤#" "op <#" "op +"]
  ⟨proof⟩
```

interpretation *mset_order_semigroup_cancel:*

```
  pordered_ab_semigroup_add_imp_le ["op ≤#" "op <#" "op +"]
  ⟨proof⟩
```

end

15 The Follows Relation of Charpentier and Sivilotte

theory *Follows* **imports** *SubstAx ListOrder Multiset* **begin**

constdefs

```
  Follows :: "['a => 'b::{order}, 'a => 'b::{order}] => 'a program set"
                 (infixl "Fols" 65)
   "f Fols g == Increasing g ∩ Increasing f Int
                Always {s. f s ≤ g s} Int
                (⋂k. {s. k ≤ g s} LeadsTo {s. k ≤ f s})"
```

lemma *mono_Always_o:*

```
     "mono h ==> Always {s. f s ≤ g s} ⊆ Always {s. h (f s) ≤ h (g s)}"
⟨proof⟩
```

lemma *mono_LeadsTo_o:*

```
     "mono (h::'a::order => 'b::order)
      ==> (⋂j. {s. j ≤ g s} LeadsTo {s. j ≤ f s}) ⊆
          (⋂k. {s. k ≤ h (g s)} LeadsTo {s. k ≤ h (f s)})"
⟨proof⟩
```

lemma *Follows_constant [iff]: "F ∈ (%s. c) Fols (%s. c)"*
⟨proof⟩

lemma *mono_Follows_o: "mono h ==> f Fols g ⊆ (h o f) Fols (h o g)"*
⟨proof⟩

lemma *mono_Follows_apply:*

```
     "mono h ==> f Fols g ⊆ (%x. h (f x)) Fols (%x. h (g x))"
⟨proof⟩
```

lemma *Follows_trans:*

```
     "[| F ∈ f Fols g;  F ∈ g Fols h |] ==> F ∈ f Fols h"
⟨proof⟩
```

15.1 Destruction rules

lemma *Follows_Increasing1: "F ∈ f Fols g ==> F ∈ Increasing f"*

⟨proof⟩

lemma *Follows_Increasing2: "F ∈ f Fols g ==> F ∈ Increasing g"*
⟨proof⟩

lemma *Follows_Bounded: "F ∈ f Fols g ==> F ∈ Always {s. f s ≤ g s}"*
⟨proof⟩

lemma *Follows_LeadsTo:*
```
"F ∈ f Fols g ==> F ∈ {s. k ≤ g s} LeadsTo {s. k ≤ f s}"
```
⟨proof⟩

lemma *Follows_LeadsTo_pfixLe:*
```
"F ∈ f Fols g ==> F ∈ {s. k pfixLe g s} LeadsTo {s. k pfixLe f s}"
```
⟨proof⟩

lemma *Follows_LeadsTo_pfixGe:*
```
"F ∈ f Fols g ==> F ∈ {s. k pfixGe g s} LeadsTo {s. k pfixGe f s}"
```
⟨proof⟩

lemma *Always_Follows1:*
```
"[| F ∈ Always {s. f s = f' s}; F ∈ f Fols g |] ==> F ∈ f' Fols g"
```

⟨proof⟩

lemma *Always_Follows2:*
```
"[| F ∈ Always {s. g s = g' s}; F ∈ f Fols g |] ==> F ∈ f Fols g'"
```
⟨proof⟩

15.2 Union properties (with the subset ordering)

lemma *increasing_Un:*
```
"[| F ∈ increasing f;  F ∈ increasing g |]
 ==> F ∈ increasing (%s. (f s) ∪ (g s))"
```
⟨proof⟩

lemma *Increasing_Un:*
```
"[| F ∈ Increasing f;  F ∈ Increasing g |]
 ==> F ∈ Increasing (%s. (f s) ∪ (g s))"
```
⟨proof⟩

lemma *Always_Un:*
```
"[| F ∈ Always {s. f' s ≤ f s}; F ∈ Always {s. g' s ≤ g s} |]
 ==> F ∈ Always {s. f' s ∪ g' s ≤ f s ∪ g s}"
```
⟨proof⟩

lemma *Follows_Un_lemma:*
```
"[| F ∈ Increasing f; F ∈ Increasing g;
    F ∈ Increasing g'; F ∈ Always {s. f' s ≤ f s};
    ∀k. F ∈ {s. k ≤ f s} LeadsTo {s. k ≤ f' s} |]
 ==> F ∈ {s. k ≤ f s ∪ g s} LeadsTo {s. k ≤ f' s ∪ g s}"
```

⟨*proof*⟩

lemma *Follows_Un:*
```
"[| F ∈ f' Fols f;  F ∈ g' Fols g |]
 ==> F ∈ (%s. (f' s) ∪ (g' s)) Fols (%s. (f s) ∪ (g s))"
```
⟨*proof*⟩

15.3 Multiset union properties (with the multiset ordering)

lemma *increasing_union:*
```
"[| F ∈ increasing f;  F ∈ increasing g |]
 ==> F ∈ increasing (%s. (f s) + (g s :: ('a::order) multiset))"
```
⟨*proof*⟩

lemma *Increasing_union:*
```
"[| F ∈ Increasing f;  F ∈ Increasing g |]
 ==> F ∈ Increasing (%s. (f s) + (g s :: ('a::order) multiset))"
```
⟨*proof*⟩

lemma *Always_union:*
```
"[| F ∈ Always {s. f' s ≤ f s}; F ∈ Always {s. g' s ≤ g s} |]
 ==> F ∈ Always {s. f' s + g' s ≤ f s + (g s :: ('a::order) multiset)}"
```
⟨*proof*⟩

lemma *Follows_union_lemma:*
```
"[| F ∈ Increasing f; F ∈ Increasing g;
    F ∈ Increasing g'; F ∈ Always {s. f' s ≤ f s};
    ∀k::('a::order) multiset.
      F ∈ {s. k ≤ f s} LeadsTo {s. k ≤ f' s} |]
 ==> F ∈ {s. k ≤ f s + g s} LeadsTo {s. k ≤ f' s + g s}"
```
⟨*proof*⟩

lemma *Follows_union:*
```
"!!g g' ::'b => ('a::order) multiset.
   [| F ∈ f' Fols f;  F ∈ g' Fols g |]
   ==> F ∈ (%s. (f' s) + (g' s)) Fols (%s. (f s) + (g s))"
```
⟨*proof*⟩

lemma *Follows_setsum:*
```
"!!f ::['c,'b] => ('a::order) multiset.
   [| ∀i ∈ I. F ∈ f' i Fols f i;  finite I |]
   ==> F ∈ (%s. ∑i ∈ I. f' i s) Fols (%s. ∑i ∈ I. f i s)"
```
⟨*proof*⟩

lemma *Increasing_imp_Stable_pfixGe:*
```
"F ∈ Increasing func ==> F ∈ Stable {s. h pfixGe (func s)}"
```
⟨*proof*⟩

lemma *LeadsTo_le_imp_pfixGe:*
```
"∀z. F ∈ {s. z ≤ f s} LeadsTo {s. z ≤ g s}
 ==> F ∈ {s. z pfixGe f s} LeadsTo {s. z pfixGe g s}"
```
⟨*proof*⟩

end

16 Predicate Transformers

theory *Transformers* **imports** *Comp* **begin**

16.1 Defining the Predicate Transformers *wp*, *awp* and *wens*

constdefs
```
wp :: "[('a*'a) set, 'a set] => 'a set"
```
— Dijkstra's weakest-precondition operator (for an individual command)
```
"wp act B == - (act^-1 `` (-B))"
```

```
awp :: "['a program, 'a set] => 'a set"
```
— Dijkstra's weakest-precondition operator (for a program)
```
"awp F B == (⋂act ∈ Acts F. wp act B)"
```

```
wens :: "['a program, ('a*'a) set, 'a set] => 'a set"
```
— The weakest-ensures transformer
```
"wens F act B == gfp(λX. (wp act B ∩ awp F (B ∪ X)) ∪ B)"
```

The fundamental theorem for wp

theorem *wp_iff:* `"(A <= wp act B) = (act `` A <= B)"`
⟨*proof*⟩

This lemma is a good deal more intuitive than the definition!

lemma *in_wp_iff:* `"(a ∈ wp act B) = (∀x. (a,x) ∈ act --> x ∈ B)"`
⟨*proof*⟩

lemma *Compl_Domain_subset_wp:* `"- (Domain act) ⊆ wp act B"`
⟨*proof*⟩

lemma *wp_empty [simp]:* `"wp act {} = - (Domain act)"`
⟨*proof*⟩

The identity relation is the skip action

lemma *wp_Id [simp]:* `"wp Id B = B"`
⟨*proof*⟩

lemma *wp_totalize_act:*
```
"wp (totalize_act act) B = (wp act B ∩ Domain act) ∪ (B - Domain act)"
```
⟨*proof*⟩

lemma *awp_subset:* `"(awp F A ⊆ A)"`
⟨*proof*⟩

lemma *awp_Int_eq:* `"awp F (A∩B) = awp F A ∩ awp F B"`

⟨proof⟩

The fundamental theorem for awp

theorem *awp_iff_constrains: "(A <= awp F B) = (F ∈ A co B)"*
⟨proof⟩

lemma *awp_iff_stable: "(A ⊆ awp F A) = (F ∈ stable A)"*
⟨proof⟩

lemma *stable_imp_awp_ident: "F ∈ stable A ==> awp F A = A"*
⟨proof⟩

lemma *wp_mono: "(A ⊆ B) ==> wp act A ⊆ wp act B"*
⟨proof⟩

lemma *awp_mono: "(A ⊆ B) ==> awp F A ⊆ awp F B"*
⟨proof⟩

lemma *wens_unfold:*
"wens F act B = (wp act B ∩ awp F (B ∪ wens F act B)) ∪ B"
⟨proof⟩

lemma *wens_Id [simp]: "wens F Id B = B"*
⟨proof⟩

These two theorems justify the claim that *wens* returns the weakest assertion satisfying the ensures property

lemma *ensures_imp_wens: "F ∈ A ensures B ==> ∃act ∈ Acts F. A ⊆ wens F act B"*
⟨proof⟩

lemma *wens_ensures: "act ∈ Acts F ==> F ∈ (wens F act B) ensures B"*
⟨proof⟩

These two results constitute assertion (4.13) of the thesis

lemma *wens_mono: "(A ⊆ B) ==> wens F act A ⊆ wens F act B"*
⟨proof⟩

lemma *wens_weakening: "B ⊆ wens F act B"*
⟨proof⟩

Assertion (6), or 4.16 in the thesis

lemma *subset_wens: "A-B ⊆ wp act B ∩ awp F (B ∪ A) ==> A ⊆ wens F act B"*
⟨proof⟩

Assertion 4.17 in the thesis

lemma *Diff_wens_constrains: "F ∈ (wens F act A - A) co wens F act A"*
⟨proof⟩

Assertion (7): 4.18 in the thesis. NOTE that many of these results hold for an arbitrary action. We often do not require *act ∈ Acts F*

lemma *stable_wens: "F ∈ stable A ==> F ∈ stable (wens F act A)"*
⟨*proof*⟩

Assertion 4.20 in the thesis.

lemma *wens_Int_eq_lemma:*
```
      "[|T-B ⊆ awp F T; act ∈ Acts F|]
       ==> T ∩ wens F act B ⊆ wens F act (T∩B)"
```
⟨*proof*⟩

Assertion (8): 4.21 in the thesis. Here we indeed require `act ∈ Acts F`

lemma *wens_Int_eq:*
```
      "[|T-B ⊆ awp F T; act ∈ Acts F|]
       ==> T ∩ wens F act B = T ∩ wens F act (T∩B)"
```
⟨*proof*⟩

16.2 Defining the Weakest Ensures Set

inductive_set
  wens_set :: "['a program, 'a set] => 'a set set"
  for *F :: "'a program"* **and** *B :: "'a set"*
where

  Basis: "B ∈ wens_set F B"

| Wens: "[|X ∈ wens_set F B; act ∈ Acts F|] ==> wens F act X ∈ wens_set F B"

| Union: "W ≠ {} ==> ∀U ∈ W. U ∈ wens_set F B ==> ⋃W ∈ wens_set F B"

lemma *wens_set_imp_co: "A ∈ wens_set F B ==> F ∈ (A-B) co A"*
⟨*proof*⟩

lemma *wens_set_imp_leadsTo: "A ∈ wens_set F B ==> F ∈ A leadsTo B"*
⟨*proof*⟩

lemma *leadsTo_imp_wens_set: "F ∈ A leadsTo B ==> ∃C ∈ wens_set F B. A ⊆ C"*
⟨*proof*⟩

Assertion (9): 4.27 in the thesis.

lemma *leadsTo_iff_wens_set: "(F ∈ A leadsTo B) = (∃C ∈ wens_set F B. A ⊆ C)"*
⟨*proof*⟩

This is the result that requires the definition of `wens_set` to require `W` to be non-empty in the Unio case, for otherwise we should always have `{} ∈ wens_set F B`.

lemma *wens_set_imp_subset: "A ∈ wens_set F B ==> B ⊆ A"*
⟨*proof*⟩

16.3 Properties Involving Program Union

Assertion (4.30) of thesis, reoriented

```
lemma awp_Join_eq: "awp (F⊔G) B = awp F B ∩ awp G B"
⟨proof⟩
```

```
lemma wens_subset: "wens F act B - B ⊆ wp act B ∩ awp F (B ∪ wens F act
B)"
⟨proof⟩
```

Assertion (4.31)

```
lemma subset_wens_Join:
      "[|A = T ∩ wens F act B;  T-B ⊆ awp F T; A-B ⊆ awp G (A ∪ B)|]
       ==> A ⊆ wens (F⊔G) act B"
⟨proof⟩
```

Assertion (4.32)

```
lemma wens_Join_subset: "wens (F⊔G) act B ⊆ wens F act B"
⟨proof⟩
```

Lemma, because the inductive step is just too messy.

```
lemma wens_Union_inductive_step:
  assumes awpF: "T-B ⊆ awp F T"
      and awpG: "!!X. X ∈ wens_set F B ==> (T∩X) - B ⊆ awp G (T∩X)"
  shows "[|X ∈ wens_set F B; act ∈ Acts F; Y ⊆ X; T∩X = T∩Y|]
         ==> wens (F⊔G) act Y ⊆ wens F act X ∧
             T ∩ wens F act X = T ∩ wens (F⊔G) act Y"
⟨proof⟩
```

```
theorem wens_Union:
  assumes awpF: "T-B ⊆ awp F T"
      and awpG: "!!X. X ∈ wens_set F B ==> (T∩X) - B ⊆ awp G (T∩X)"
      and major: "X ∈ wens_set F B"
  shows "∃Y ∈ wens_set (F⊔G) B. Y ⊆ X & T∩X = T∩Y"
⟨proof⟩
```

```
theorem leadsTo_Join:
  assumes leadsTo: "F ∈ A leadsTo B"
      and awpF: "T-B ⊆ awp F T"
      and awpG: "!!X. X ∈ wens_set F B ==> (T∩X) - B ⊆ awp G (T∩X)"
  shows "F⊔G ∈ T∩A leadsTo B"
⟨proof⟩
```

16.4 The Set wens_set F B for a Single-Assignment Program

Thesis Section 4.3.3

We start by proving laws about single-assignment programs

```
lemma awp_single_eq [simp]:
     "awp (mk_program (init, {act}, allowed)) B = B ∩ wp act B"
⟨proof⟩
```

```
lemma wp_Un_subset: "wp act A ∪ wp act B ⊆ wp act (A ∪ B)"
⟨proof⟩
```

lemma *wp_Un_eq: "single_valued act ==> wp act (A ∪ B) = wp act A ∪ wp act B"*
⟨*proof*⟩

lemma *wp_UN_subset: "(⋃i∈I. wp act (A i)) ⊆ wp act (⋃i∈I. A i)"*
⟨*proof*⟩

lemma *wp_UN_eq:*
```
"[|single_valued act; I≠{}|]
 ==> wp act (⋃i∈I. A i) = (⋃i∈I. wp act (A i))"
```
⟨*proof*⟩

lemma *wens_single_eq:*
```
"wens (mk_program (init, {act}, allowed)) act B = B ∪ wp act B"
```
⟨*proof*⟩

Next, we express the `wens_set` for single-assignment programs

constdefs
```
wens_single_finite :: "[('a*'a) set, 'a set, nat] => 'a set"
  "wens_single_finite act B k == ⋃i ∈ atMost k. ((wp act)^i) B"

wens_single :: "[('a*'a) set, 'a set] => 'a set"
  "wens_single act B == ⋃i. ((wp act)^i) B"
```

lemma *wens_single_Un_eq:*
```
"single_valued act
 ==> wens_single act B ∪ wp act (wens_single act B) = wens_single act B"
```
⟨*proof*⟩

lemma *atMost_nat_nonempty: "atMost (k::nat) ≠ {}"*
⟨*proof*⟩

lemma *wens_single_finite_0 [simp]: "wens_single_finite act B 0 = B"*
⟨*proof*⟩

lemma *wens_single_finite_Suc:*
```
"single_valued act
 ==> wens_single_finite act B (Suc k) =
     wens_single_finite act B k ∪ wp act (wens_single_finite act B k)"
```
⟨*proof*⟩

lemma *wens_single_finite_Suc_eq_wens:*
```
"single_valued act
  ==> wens_single_finite act B (Suc k) =
      wens (mk_program (init, {act}, allowed)) act
           (wens_single_finite act B k)"
```
⟨*proof*⟩

lemma *def_wens_single_finite_Suc_eq_wens:*
```
"[|F = mk_program (init, {act}, allowed); single_valued act|]
  ==> wens_single_finite act B (Suc k) =
      wens F act (wens_single_finite act B k)"
```
⟨*proof*⟩

```
lemma wens_single_finite_Un_eq:
      "single_valued act
       ==> wens_single_finite act B k ∪ wp act (wens_single_finite act B k)
           ∈ range (wens_single_finite act B)"
⟨proof⟩

lemma wens_single_eq_Union:
      "wens_single act B = ⋃range (wens_single_finite act B)"
⟨proof⟩

lemma wens_single_finite_eq_Union:
     "wens_single_finite act B n = (⋃k∈atMost n. wens_single_finite act B
k)"
⟨proof⟩

lemma wens_single_finite_mono:
     "m ≤ n ==> wens_single_finite act B m ⊆ wens_single_finite act B n"
⟨proof⟩

lemma wens_single_finite_subset_wens_single:
      "wens_single_finite act B k ⊆ wens_single act B"
⟨proof⟩

lemma subset_wens_single_finite:
      "[|W ⊆ wens_single_finite act B ` (atMost k); single_valued act; W≠{}|]
       ==> ∃m. ⋃W = wens_single_finite act B m"
⟨proof⟩
```

lemma for Union case

```
lemma Union_eq_wens_single:
      "⟦∀k. ¬ W ⊆ wens_single_finite act B ` {..k};
        W ⊆ insert (wens_single act B)
            (range (wens_single_finite act B))⟧
       ⟹ ⋃W = wens_single act B"
⟨proof⟩

lemma wens_set_subset_single:
      "single_valued act
       ==> wens_set (mk_program (init, {act}, allowed)) B ⊆
           insert (wens_single act B) (range (wens_single_finite act B))"
⟨proof⟩

lemma wens_single_finite_in_wens_set:
      "single_valued act ⟹
         wens_single_finite act B k
         ∈ wens_set (mk_program (init, {act}, allowed)) B"
⟨proof⟩

lemma single_subset_wens_set:
      "single_valued act
       ==> insert (wens_single act B) (range (wens_single_finite act B)) ⊆

           wens_set (mk_program (init, {act}, allowed)) B"
```

⟨*proof*⟩

Theorem (4.29)

theorem *wens_set_single_eq:*
```
     "[|F = mk_program (init, {act}, allowed); single_valued act|]
      ==> wens_set F B =
          insert (wens_single act B) (range (wens_single_finite act B))"
```
⟨*proof*⟩

Generalizing Misra's Fixed Point Union Theorem (4.41)

lemma *fp_leadsTo_Join:*
```
    "[|T-B ⊆ awp F T; T-B ⊆ FP G; F ∈ A leadsTo B|] ==> F⊔G ∈ T∩A leadsTo
B"
```
⟨*proof*⟩

end

17 Progress Sets

theory *ProgressSets* **imports** *Transformers* **begin**

17.1 Complete Lattices and the Operator *cl*

constdefs
```
  lattice :: "'a set set => bool"
```
— Meier calls them closure sets, but they are just complete lattices
```
   "lattice L ==
        (∀M. M ⊆ L --> ⋂M ∈ L) & (∀M. M ⊆ L --> ⋃M ∈ L)"

  cl :: "['a set set, 'a set] => 'a set"
```
— short for "closure"
```
   "cl L r == ⋂{x. x∈L & r ⊆ x}"
```

lemma *UNIV_in_lattice: "lattice L ==> UNIV ∈ L"*
⟨*proof*⟩

lemma *empty_in_lattice: "lattice L ==> {} ∈ L"*
⟨*proof*⟩

lemma *Union_in_lattice: "[|M ⊆ L; lattice L|] ==> ⋃M ∈ L"*
⟨*proof*⟩

lemma *Inter_in_lattice: "[|M ⊆ L; lattice L|] ==> ⋂M ∈ L"*
⟨*proof*⟩

lemma *UN_in_lattice:*
```
     "[|lattice L; !!i. i∈I ==> r i ∈ L|] ==> (⋃i∈I. r i) ∈ L"
```
⟨*proof*⟩

lemma *INT_in_lattice:*
```
     "[|lattice L; !!i. i∈I ==> r i ∈ L|] ==> (⋂i∈I. r i)  ∈ L"
```
⟨*proof*⟩

lemma *Un_in_lattice: "[|x∈L; y∈L; lattice L|] ==> x∪y ∈ L"*
⟨*proof*⟩

lemma *Int_in_lattice: "[|x∈L; y∈L; lattice L|] ==> x∩y ∈ L"*
⟨*proof*⟩

lemma *lattice_stable: "lattice {X. F ∈ stable X}"*
⟨*proof*⟩

The next three results state that *cl L r* is the minimal element of *L* that includes *r*.

lemma *cl_in_lattice: "lattice L ==> cl L r ∈ L"*
⟨*proof*⟩

lemma *cl_least: "[|c∈L; r⊆c|] ==> cl L r ⊆ c"*
⟨*proof*⟩

The next three lemmas constitute assertion (4.61)

lemma *cl_mono: "r ⊆ r' ==> cl L r ⊆ cl L r'"*
⟨*proof*⟩

lemma *subset_cl: "r ⊆ cl L r"*
⟨*proof*⟩

A reformulation of *?r ⊆ cl ?L ?r*

lemma *clI: "x ∈ r ==> x ∈ cl L r"*
⟨*proof*⟩

A reformulation of $[\![$*?c ∈ ?L; ?r ⊆ ?c*$]\!] \Longrightarrow$ *cl ?L ?r ⊆ ?c*

lemma *clD: "[|c ∈ cl L r; B ∈ L; r ⊆ B|] ==> c ∈ B"*
⟨*proof*⟩

lemma *cl_UN_subset: "(⋃i∈I. cl L (r i)) ⊆ cl L (⋃i∈I. r i)"*
⟨*proof*⟩

lemma *cl_Un: "lattice L ==> cl L (r∪s) = cl L r ∪ cl L s"*
⟨*proof*⟩

lemma *cl_UN: "lattice L ==> cl L (⋃i∈I. r i) = (⋃i∈I. cl L (r i))"*
⟨*proof*⟩

lemma *cl_Int_subset: "cl L (r∩s) ⊆ cl L r ∩ cl L s"*
⟨*proof*⟩

lemma *cl_idem [simp]: "cl L (cl L r) = cl L r"*
⟨*proof*⟩

lemma *cl_ident: "r∈L ==> cl L r = r"*
⟨*proof*⟩

lemma *cl_empty [simp]: "lattice L ==> cl L {} = {}"*
⟨*proof*⟩

lemma *cl_UNIV [simp]: "lattice L ==> cl L UNIV = UNIV"*
⟨*proof*⟩

Assertion (4.62)

lemma *cl_ident_iff: "lattice L ==> (cl L r = r) = (r∈L)"*
⟨*proof*⟩

lemma *cl_subset_in_lattice: "[|cl L r ⊆ r; lattice L|] ==> r∈L"*
⟨*proof*⟩

17.2 Progress Sets and the Main Lemma

A progress set satisfies certain closure conditions and is a simple way of including the set *wens_set F B*.

```
constdefs
  closed :: "['a program, 'a set, 'a set,  'a set set] => bool"
   "closed F T B L == ∀M. ∀act ∈ Acts F. B⊆M & T∩M ∈ L -->
                              T ∩ (B ∪ wp act M) ∈ L"

  progress_set :: "['a program, 'a set, 'a set] => 'a set set set"
   "progress_set F T B ==
      {L. lattice L & B ∈ L & T ∈ L & closed F T B L}"
```

```
lemma closedD:
   "[|closed F T B L; act ∈ Acts F; B⊆M; T∩M ∈ L|]
    ==> T ∩ (B ∪ wp act M) ∈ L"
⟨proof⟩
```

Note: the formalization below replaces Meier's *q* by *B* and *m* by *X*.

Part of the proof of the claim at the bottom of page 97. It's proved separately because the argument requires a generalization over all *act ∈ Acts F*.

```
lemma lattice_awp_lemma:
  assumes TXC:  "T∩X ∈ C" — induction hypothesis in theorem below
      and BsubX:  "B ⊆ X"    — holds in inductive step
      and latt: "lattice C"
      and TC:   "T ∈ C"
      and BC:   "B ∈ C"
      and clos: "closed F T B C"
    shows "T ∩ (B ∪ awp F (X ∪ cl C (T∩r))) ∈ C"
⟨proof⟩
```

Remainder of the proof of the claim at the bottom of page 97.

```
lemma lattice_lemma:
  assumes TXC:  "T∩X ∈ C" — induction hypothesis in theorem below
      and BsubX:  "B ⊆ X"    — holds in inductive step
      and act:  "act ∈ Acts F"
      and latt: "lattice C"
      and TC:   "T ∈ C"
      and BC:   "B ∈ C"
      and clos: "closed F T B C"
```

```
  shows "T ∩ (wp act X ∩ awp F (X ∪ cl C (T∩r)) ∪ X) ∈ C"
⟨proof⟩
```

Induction step for the main lemma

```
lemma progress_induction_step:
 assumes TXC:  "T∩X ∈ C" — induction hypothesis in theorem below
     and act:  "act ∈ Acts F"
     and Xwens: "X ∈ wens_set F B"
     and latt: "lattice C"
     and  TC:  "T ∈ C"
     and  BC:  "B ∈ C"
     and clos: "closed F T B C"
     and Fstable: "F ∈ stable T"
 shows "T ∩ wens F act X ∈ C"
⟨proof⟩
```

Proved on page 96 of Meier's thesis. The special case when `T = UNIV` states that every progress set for the program `F` and set `B` includes the set `wens_set F B`.

```
lemma progress_set_lemma:
    "[|C ∈ progress_set F T B; r ∈ wens_set F B; F ∈ stable T|] ==> T∩r
∈ C"
⟨proof⟩
```

17.3 The Progress Set Union Theorem

```
lemma closed_mono:
 assumes BB':  "B ⊆ B'"
     and TBwp: "T ∩ (B ∪ wp act M) ∈ C"
     and B'C:  "B' ∈ C"
     and TC:   "T ∈ C"
     and latt: "lattice C"
 shows "T ∩ (B' ∪ wp act M) ∈ C"
⟨proof⟩


lemma progress_set_mono:
   assumes BB':  "B ⊆ B'"
   shows
    "[| B' ∈ C;  C ∈ progress_set F T B|]
     ==> C ∈ progress_set F T B'"
⟨proof⟩

theorem progress_set_Union:
 assumes leadsTo: "F ∈ A leadsTo B'"
     and prog: "C ∈ progress_set F T B"
     and Fstable: "F ∈ stable T"
     and BB':  "B ⊆ B'"
     and B'C:  "B' ∈ C"
     and Gco: "!!X. X∈C ==> G ∈ X-B co X"
 shows "F⊔G ∈ T∩A leadsTo B'"
⟨proof⟩
```

17.4 Some Progress Sets

lemma *UNIV_in_progress_set: "UNIV ∈ progress_set F T B"*
⟨*proof*⟩

17.4.1 Lattices and Relations

From Meier's thesis, section 4.5.3

constdefs
```
  relcl :: "'a set set => ('a * 'a) set"
    — Derived relation from a lattice
    "relcl L == {(x,y). y ∈ cl L {x}}"

  latticeof :: "('a * 'a) set => 'a set set"
    — Derived lattice from a relation: the set of upwards-closed sets
    "latticeof r == {X. ∀s t. s ∈ X & (s,t) ∈ r --> t ∈ X}"
```

lemma *relcl_refl: "(a,a) ∈ relcl L"*
⟨*proof*⟩

lemma *relcl_trans:*
"[| (a,b) ∈ relcl L; (b,c) ∈ relcl L; lattice L |] ==> (a,c) ∈ relcl L"
⟨*proof*⟩

lemma *refl_relcl: "lattice L ==> refl UNIV (relcl L)"*
⟨*proof*⟩

lemma *trans_relcl: "lattice L ==> trans (relcl L)"*
⟨*proof*⟩

lemma *lattice_latticeof: "lattice (latticeof r)"*
⟨*proof*⟩

lemma *lattice_singletonI:*
"[|lattice L; !!s. s ∈ X ==> {s} ∈ L|] ==> X ∈ L"
⟨*proof*⟩

Equation (4.71) of Meier's thesis. He gives no proof.

lemma *cl_latticeof:*
```
     "[|refl UNIV r; trans r|]
      ==> cl (latticeof r) X = {t. ∃s. s∈X & (s,t) ∈ r}"
```
⟨*proof*⟩

Related to (4.71).

lemma *cl_eq_Collect_relcl:*
"lattice L ==> cl L X = {t. ∃s. s∈X & (s,t) ∈ relcl L}"
⟨*proof*⟩

Meier's theorem of section 4.5.3

theorem *latticeof_relcl_eq: "lattice L ==> latticeof (relcl L) = L"*
⟨*proof*⟩

theorem *relcl_latticeof_eq:*
```
    "[|refl UNIV r; trans r|] ==> relcl (latticeof r) = r"
```
⟨*proof*⟩

17.4.2 Decoupling Theorems

constdefs
```
  decoupled :: "['a program, 'a program] => bool"
   "decoupled F G ==
        ∀act ∈ Acts F. ∀B. G ∈ stable B --> G ∈ stable (wp act B)"
```

Rao's Decoupling Theorem

lemma *stableco:* `"F ∈ stable A ==> F ∈ A-B co A"`
⟨*proof*⟩

theorem *decoupling:*
assumes *leadsTo:* `"F ∈ A leadsTo B"`
and *Gstable:* `"G ∈ stable B"`
and *dec:* `"decoupled F G"`
shows `"F⊔G ∈ A leadsTo B"`
⟨*proof*⟩

Rao's Weak Decoupling Theorem

theorem *weak_decoupling:*
assumes *leadsTo:* `"F ∈ A leadsTo B"`
and *stable:* `"F⊔G ∈ stable B"`
and *dec:* `"decoupled F (F⊔G)"`
shows `"F⊔G ∈ A leadsTo B"`
⟨*proof*⟩

The "Decoupling via `G'` Union Theorem"

theorem *decoupling_via_aux:*
assumes *leadsTo:* `"F ∈ A leadsTo B"`
and *prog:* `"{X. G' ∈ stable X} ∈ progress_set F UNIV B"`
and *GG':* `"G ≤ G'"`
— Beware! This is the converse of the refinement relation!
shows `"F⊔G ∈ A leadsTo B"`
⟨*proof*⟩

17.5 Composition Theorems Based on Monotonicity and Commutativity

17.5.1 Commutativity of `cl L` and assignment.

constdefs
```
  commutes :: "['a program, 'a set, 'a set,  'a set set] => bool"
   "commutes F T B L ==
       ∀M. ∀act ∈ Acts F. B ⊆ M -->
           cl L (T ∩ wp act M) ⊆ T ∩ (B ∪ wp act (cl L (T∩M)))"
```

From Meier's thesis, section 4.5.6

lemma *commutativity1_lemma:*

```
  assumes commutes: "commutes F T B L"
      and lattice:  "lattice L"
      and BL: "B ∈ L"
      and TL: "T ∈ L"
  shows "closed F T B L"
⟨proof⟩
```

Version packaged with ⟦?F ∈ ?A leadsTo ?B'; ?C ∈ progress_set ?F ?T ?B; ?F ∈ stable ?T; ?B ⊆ ?B'; ?B' ∈ ?C; ⋀X. X ∈ ?C ⟹ ?G ∈ X - ?B co X⟧ ⟹ ?F ⊔ ?G ∈ ?T ∩ ?A leadsTo ?B'

```
lemma commutativity1:
  assumes leadsTo: "F ∈ A leadsTo B"
      and lattice:  "lattice L"
      and BL: "B ∈ L"
      and TL: "T ∈ L"
      and Fstable: "F ∈ stable T"
      and Gco: "!!X. X∈L ==> G ∈ X-B co X"
      and commutes: "commutes F T B L"
  shows "F⊔G ∈ T∩A leadsTo B"
⟨proof⟩
```

Possibly move to Relation.thy, after single_valued

```
constdefs
  funof :: "[('a*'b)set, 'a] => 'b"
   "funof r == (λx. THE y. (x,y) ∈ r)"

lemma funof_eq: "[|single_valued r; (x,y) ∈ r|] ==> funof r x = y"
⟨proof⟩

lemma funof_Pair_in:
     "[|single_valued r; x ∈ Domain r|] ==> (x, funof r x) ∈ r"
⟨proof⟩

lemma funof_in:
     "[|r``{x} ⊆ A; single_valued r; x ∈ Domain r|] ==> funof r x ∈ A"
⟨proof⟩

lemma funof_imp_wp: "[|funof act t ∈ A; single_valued act|] ==> t ∈ wp act
A"
⟨proof⟩
```

17.5.2 Commutativity of Functions and Relation

Thesis, page 109

From Meier's thesis, section 4.5.6

```
lemma commutativity2_lemma:
  assumes dcommutes:
        "∀act ∈ Acts F.
         ∀s ∈ T. ∀t. (s,t) ∈ relcl L -->
                      s ∈ B | t ∈ B | (funof act s, funof act t) ∈ relcl
L"
      and determ: "!!act. act ∈ Acts F ==> single_valued act"
```

```
      and total: "!!act. act ∈ Acts F ==> Domain act = UNIV"
      and lattice:  "lattice L"
      and BL: "B ∈ L"
      and TL: "T ∈ L"
      and Fstable: "F ∈ stable T"
  shows  "commutes F T B L"
```
⟨proof⟩

Version packaged with ⟦?F ∈ ?A leadsTo ?B'; ?C ∈ progress_set ?F ?T ?B; ?F ∈ stable ?T; ?B ⊆ ?B'; ?B' ∈ ?C; ⋀X. X ∈ ?C ⟹ ?G ∈ X - ?B co X⟧ ⟹ ?F ⊔ ?G ∈ ?T ∩ ?A leadsTo ?B'

```
lemma commutativity2:
  assumes leadsTo: "F ∈ A leadsTo B"
      and dcommutes:
        "∀act ∈ Acts F.
         ∀s ∈ T. ∀t. (s,t) ∈ relcl L -->
                      s ∈ B | t ∈ B | (funof act s, funof act t) ∈ relcl
L"
      and determ: "!!act. act ∈ Acts F ==> single_valued act"
      and total: "!!act. act ∈ Acts F ==> Domain act = UNIV"
      and lattice:  "lattice L"
      and BL: "B ∈ L"
      and TL: "T ∈ L"
      and Fstable: "F ∈ stable T"
      and Gco: "!!X. X∈L ==> G ∈ X-B co X"
  shows "F⊔G ∈ T∩A leadsTo B"
```
⟨proof⟩

17.6 Monotonicity

From Meier's thesis, section 4.5.7, page 110

end

18 Comprehensive UNITY Theory

```
theory UNITY_Main imports Detects PPROD Follows ProgressSets
uses "UNITY_tactics.ML" begin
```

⟨ML⟩

end

```
theory Deadlock imports UNITY begin
```

```
lemma "[| F ∈ (A ∩ B) co A;  F ∈ (B ∩ A) co B |] ==> F ∈ stable (A ∩ B)"
```
⟨proof⟩

lemma *Collect_le_Int_equals:*
"($\bigcap$i ∈ atMost n. A(Suc i) ∩ A i) = ($\bigcap$i ∈ atMost (Suc n). A i)"
⟨*proof*⟩

lemma *UN_Int_Compl_subset:*
"($\bigcup$i ∈ lessThan n. A i) ∩ (- A n) ⊆
($\bigcup$i ∈ lessThan n. (A i) ∩ (- A (Suc i)))"
⟨*proof*⟩

lemma *INT_Un_Compl_subset:*
"($\bigcap$i ∈ lessThan n. -A i ∪ A (Suc i)) ⊆
($\bigcap$i ∈ lessThan n. -A i) ∪ A n"
⟨*proof*⟩

lemma *INT_le_equals_Int_lemma:*
"A 0 ∩ (-(A n) ∩ ($\bigcap$i ∈ lessThan n. -A i ∪ A (Suc i))) = {}"
⟨*proof*⟩

lemma *INT_le_equals_Int:*
"($\bigcap$i ∈ atMost n. A i) =
A 0 ∩ ($\bigcap$i ∈ lessThan n. -A i ∪ A(Suc i))"
⟨*proof*⟩

lemma *INT_le_Suc_equals_Int:*
"($\bigcap$i ∈ atMost (Suc n). A i) =
A 0 ∩ ($\bigcap$i ∈ atMost n. -A i ∪ A(Suc i))"
⟨*proof*⟩

lemma
assumes *zeroprem:* "F ∈ (A 0 ∩ A (Suc n)) co (A 0)"
and *allprem:*
"!!i. i ∈ atMost n ==> F ∈ (A(Suc i) ∩ A i) co (-A i ∪ A(Suc
i))"
shows "F ∈ stable ($\bigcap$i ∈ atMost (Suc n). A i)"
⟨*proof*⟩

end

theory *Common* **imports** "../UNITY_Main" **begin**

consts
ftime :: "nat=>nat"
gtime :: "nat=>nat"

axioms

```
  fmono: "m ≤ n ==> ftime m ≤ ftime n"
  gmono: "m ≤ n ==> gtime m ≤ gtime n"

  fasc:  "m ≤ ftime n"
  gasc:  "m ≤ gtime n"
```

constdefs

```
  common :: "nat set"
    "common == {n. ftime n = n & gtime n = n}"

  maxfg :: "nat => nat set"
    "maxfg m == {t. t ≤ max (ftime m) (gtime m)}"
```

lemma *common_stable:*

```
     "[| ∀m. F ∈ {m} Co (maxfg m); n ∈ common |]
      ==> F ∈ Stable (atMost n)"
```

⟨*proof*⟩

lemma *common_safety:*

```
     "[| Init F ⊆ atMost n;
         ∀m. F ∈ {m} Co (maxfg m); n ∈ common |]
      ==> F ∈ Always (atMost n)"
```

⟨*proof*⟩

lemma *"SKIP ∈ {m} co (maxfg m)"*

⟨*proof*⟩

lemma

```
"mk_total_program
          (UNIV, {range(%t.(t,ftime t)), range(%t.(t,gtime t))}, UNIV)
       ∈ {m} co (maxfg m)"
```

⟨*proof*⟩

lemma

```
"mk_total_program (UNIV, {range(%t.(t, max (ftime t) (gtime t)))},
UNIV)
       ∈ {m} co (maxfg m)"
```

⟨*proof*⟩

lemma

```
 "mk_total_program
           (UNIV, { {(t, Suc t) | t. t < max (ftime t) (gtime t)} }, UNIV)
       ∈ {m} co (maxfg m)"
```

⟨*proof*⟩

```
declare atMost_Int_atLeast [simp]

lemma leadsTo_common_lemma:
     "[| ∀m. F ∈ {m} Co (maxfg m);
         ∀m ∈ lessThan n. F ∈ {m} LeadsTo (greaterThan m);
         n ∈ common |]
      ==> F ∈ (atMost n) LeadsTo common"
⟨proof⟩


lemma leadsTo_common:
     "[| ∀m. F ∈ {m} Co (maxfg m);
         ∀m ∈ -common. F ∈ {m} LeadsTo (greaterThan m);
         n ∈ common |]
      ==> F ∈ (atMost (LEAST n. n ∈ common)) LeadsTo common"
⟨proof⟩

end


theory Network imports UNITY begin


datatype pvar = Sent | Rcvd | Idle

datatype pname = Aproc | Bproc

types state = "pname * pvar => nat"

locale F_props =
  fixes F
  assumes rsA: "F ∈ stable {s. s(Bproc,Rcvd) ≤ s(Aproc,Sent)}"
      and rsB: "F ∈ stable {s. s(Aproc,Rcvd) ≤ s(Bproc,Sent)}"
    and sent_nondec: "F ∈ stable {s. m ≤ s(proc,Sent)}"
    and rcvd_nondec: "F ∈ stable {s. n ≤ s(proc,Rcvd)}"
    and rcvd_idle: "F ∈ {s. s(proc,Idle) = Suc 0 & s(proc,Rcvd) = m}
                      co {s. s(proc,Rcvd) = m --> s(proc,Idle) = Suc 0}"
    and sent_idle: "F ∈ {s. s(proc,Idle) = Suc 0 & s(proc,Sent) = n}
                      co {s. s(proc,Sent) = n}"


lemmas (in F_props)
        sent_nondec_A = sent_nondec [of _ Aproc]
    and sent_nondec_B = sent_nondec [of _ Bproc]
    and rcvd_nondec_A = rcvd_nondec [of _ Aproc]
    and rcvd_nondec_B = rcvd_nondec [of _ Bproc]
    and rcvd_idle_A = rcvd_idle [of Aproc]
    and rcvd_idle_B = rcvd_idle [of Bproc]
```

```
  and sent_idle_A = sent_idle [of Aproc]
  and sent_idle_B = sent_idle [of Bproc]

  and rs_AB = stable_Int [OF rsA rsB]
  and sent_nondec_AB = stable_Int [OF sent_nondec_A sent_nondec_B]
  and rcvd_nondec_AB = stable_Int [OF rcvd_nondec_A rcvd_nondec_B]
  and rcvd_idle_AB = constrains_Int [OF rcvd_idle_A rcvd_idle_B]
  and sent_idle_AB = constrains_Int [OF sent_idle_A sent_idle_B]
  and nondec_AB = stable_Int [OF sent_nondec_AB rcvd_nondec_AB]
  and idle_AB = constrains_Int [OF rcvd_idle_AB sent_idle_AB]
  and nondec_idle = constrains_Int [OF nondec_AB [unfolded stable_def]
                                     idle_AB]

lemma (in F_props)
  shows "F ∈ stable {s. s(Aproc,Idle) = Suc 0 & s(Bproc,Idle) = Suc 0 &
                        s(Aproc,Sent) = s(Bproc,Rcvd) &
                        s(Bproc,Sent) = s(Aproc,Rcvd) &
                        s(Aproc,Rcvd) = m & s(Bproc,Rcvd) = n}"
⟨proof⟩

end
```

19 The Token Ring

```
theory Token
imports "../WFair"

begin
```

From Misra, "A Logic for Concurrent Programming" (1994), sections 5.2 and 13.2.

19.1 Definitions

```
datatype pstate = Hungry | Eating | Thinking
    — process states

record state =
  token :: "nat"
  proc  :: "nat => pstate"


constdefs
  HasTok :: "nat => state set"
    "HasTok i == {s. token s = i}"

  H :: "nat => state set"
    "H i == {s. proc s i = Hungry}"

  E :: "nat => state set"
    "E i == {s. proc s i = Eating}"
```

```
  T :: "nat => state set"
    "T i == {s. proc s i = Thinking}"
```

locale *Token* =
  fixes *N* **and** *F* **and** *nodeOrder* **and** *"next"*
  defines *nodeOrder_def:*

```
      "nodeOrder j == measure(%i. ((j+N)-i) mod N) ∩ {..<N} × {..<N}"
```

  and *next_def:*

```
      "next i == (Suc i) mod N"
```

  assumes *N_positive [iff]: "0<N"*

```
    and TR2:  "F ∈ (T i) co (T i ∪ H i)"
    and TR3:  "F ∈ (H i) co (H i ∪ E i)"
    and TR4:  "F ∈ (H i - HasTok i) co (H i)"
    and TR5:  "F ∈ (HasTok i) co (HasTok i ∪ -(E i))"
    and TR6:  "F ∈ (H i ∩ HasTok i) leadsTo (E i)"
    and TR7:  "F ∈ (HasTok i) leadsTo (HasTok (next i))"
```

lemma *HasToK_partition: "[| s ∈ HasTok i; s ∈ HasTok j |] ==> i=j"*
⟨*proof*⟩

lemma *not_E_eq: "(s ∉ E i) = (s ∈ H i | s ∈ T i)"*
⟨*proof*⟩

lemma *(***in** *Token) token_stable: "F ∈ stable (-(E i) ∪ (HasTok i))"*
⟨*proof*⟩

19.2 Progress under Weak Fairness

lemma *(***in** *Token) wf_nodeOrder: "wf(nodeOrder j)"*
⟨*proof*⟩

lemma *(***in** *Token) nodeOrder_eq:*

```
     "[| i<N; j<N |] ==> ((next i, i) ∈ nodeOrder j) = (i ≠ j)"
```

⟨*proof*⟩

From "A Logic for Concurrent Programming", but not used in Chapter 4. Note the use of `case_tac`. Reasoning about leadsTo takes practice!

lemma *(***in** *Token) TR7_nodeOrder:*

```
     "[| i<N; j<N |] ==>
      F ∈ (HasTok i) leadsTo ({s. (token s, i) ∈ nodeOrder j} ∪ HasTok j)"
```

⟨*proof*⟩

Chapter 4 variant, the one actually used below.

lemma *(***in** *Token) TR7_aux: "[| i<N; j<N; i≠j |]*

```
      ==> F ∈ (HasTok i) leadsTo {s. (token s, i) ∈ nodeOrder j}"
```

⟨*proof*⟩

lemma *(***in** *Token) token_lemma:*

```
     "({s. token s < N} ∩ token -‘ {m}) = (if m<N then token -‘ {m} else {})"
```

⟨*proof*⟩

Misra's TR9: the token reaches an arbitrary node

```
lemma (in Token) leadsTo_j: "j<N ==> F ∈ {s. token s < N} leadsTo (HasTok
j)"
⟨proof⟩
```

Misra's TR8: a hungry process eventually eats

```
lemma (in Token) token_progress:
     "j<N ==> F ∈ ({s. token s < N} ∩ H j) leadsTo (E j)"
⟨proof⟩


end



theory Channel imports "../UNITY_Main" begin

types state = "nat set"

consts
  F :: "state program"

constdefs
  minSet :: "nat set => nat option"
    "minSet A == if A={} then None else Some (LEAST x. x ∈ A)"

axioms

  UC1:  "F ∈ (minSet -` {Some x}) co (minSet -` (Some`atLeast x))"



  UC2:  "F ∈ (minSet -` {Some x}) leadsTo {s. x ∉ s}"



lemma minSet_eq_SomeD: "minSet A = Some x ==> x ∈ A"
⟨proof⟩

lemma minSet_empty [simp]: " minSet{} = None"
⟨proof⟩

lemma minSet_nonempty: "x ∈ A ==> minSet A = Some (LEAST x. x ∈ A)"
⟨proof⟩

lemma minSet_greaterThan:
     "F ∈ (minSet -` {Some x}) leadsTo (minSet -` (Some`greaterThan x))"
⟨proof⟩


lemma Channel_progress_lemma:
     "F ∈ (UNIV-{{}}) leadsTo (minSet -` (Some`atLeast y))"
⟨proof⟩
```

```
lemma Channel_progress: "!!y::nat. F ∈ (UNIV-{{}}) leadsTo {s. y ∉ s}"
⟨proof⟩

end



theory Lift
imports "../UNITY_Main"

begin

record state =
  floor :: "int"                — current position of the lift
  "open" :: "bool"              — whether the door is opened at floor
  stop  :: "bool"               — whether the lift is stopped at floor
  req   :: "int set"            — for each floor, whether the lift is requested
  up    :: "bool"               — current direction of movement
  move  :: "bool"               — whether moving takes precedence over opening

consts
  Min :: "int"          — least and greatest floors
  Max :: "int"          — least and greatest floors

axioms
  Min_le_Max [iff]: "Min ≤ Max"

constdefs

  — Abbreviations: the "always" part

  above :: "state set"
    "above == {s. ∃i. floor s < i & i ≤ Max & i ∈ req s}"

  below :: "state set"
    "below == {s. ∃i. Min ≤ i & i < floor s & i ∈ req s}"

  queueing :: "state set"
    "queueing == above ∪ below"

  goingup :: "state set"
    "goingup   == above ∩ ({s. up s}  ∪ -below)"

  goingdown :: "state set"
    "goingdown == below ∩ ({s. ~ up s} ∪ -above)"

  ready :: "state set"
    "ready == {s. stop s & ~ open s & move s}"

  — Further abbreviations

  moving :: "state set"
    "moving ==  {s. ~ stop s & ~ open s}"
```

```
stopped :: "state set"
  "stopped == {s. stop s  & ~ open s & ~ move s}"

opened :: "state set"
  "opened ==  {s. stop s  &  open s  &  move s}"
```

```
closed :: "state set"
```
— but this is the same as ready!!

```
  "closed ==  {s. stop s  & ~ open s &  move s}"

atFloor :: "int => state set"
  "atFloor n ==  {s. floor s = n}"

Req :: "int => state set"
  "Req n ==  {s. n ∈ req s}"
```

— The program

```
request_act :: "(state*state) set"
  "request_act == {(s,s'). s' = s (|stop:=True, move:=False|)
                               & ~ stop s & floor s ∈ req s}"

open_act :: "(state*state) set"
  "open_act ==
       {(s,s'). s' = s (|open :=True,
                         req  := req s - {floor s},
                         move := True|)
                     & stop s & ~ open s & floor s ∈ req s
                     & ~(move s & s ∈ queueing)}"

close_act :: "(state*state) set"
  "close_act == {(s,s'). s' = s (|open := False|) & open s}"

req_up :: "(state*state) set"
  "req_up ==
       {(s,s'). s' = s (|stop  :=False,
                         floor := floor s + 1,
                         up    := True|)
                     & s ∈ (ready ∩ goingup)}"

req_down :: "(state*state) set"
  "req_down ==
       {(s,s'). s' = s (|stop  :=False,
                         floor := floor s - 1,
                         up    := False|)
                     & s ∈ (ready ∩ goingdown)}"

move_up :: "(state*state) set"
  "move_up ==
       {(s,s'). s' = s (|floor := floor s + 1|)
                     & ~ stop s & up s & floor s ∉ req s}"

move_down :: "(state*state) set"
```

```
  "move_down ==
       {(s,s'). s' = s (|floor := floor s - 1|)
                      & ~ stop s & ~ up s & floor s ∉ req s}"

button_press  :: "(state*state) set"
```

— This action is omitted from prior treatments, which therefore are unrealistic: nobody asks the lift to do anything! But adding this action invalidates many of the existing progress arguments: various "ensures" properties fail. Maybe it should be constrained to only allow button presses in the current direction of travel, like in a real lift.

```
  "button_press ==
       {(s,s'). ∃n. s' = s (|req := insert n (req s)|)
                      & Min ≤ n & n ≤ Max}"


Lift :: "state program"
```

— for the moment, we OMIT `button_press`

```
  "Lift == mk_total_program
              ({s. floor s = Min & ~ up s & move s & stop s &
                         ~ open s & req s = {}},
               {request_act, open_act, close_act,
                req_up, req_down, move_up, move_down},
               UNIV)"
```

— Invariants

```
bounded :: "state set"
  "bounded == {s. Min ≤ floor s & floor s ≤ Max}"

open_stop :: "state set"
  "open_stop == {s. open s --> stop s}"

open_move :: "state set"
  "open_move == {s. open s --> move s}"

stop_floor :: "state set"
  "stop_floor == {s. stop s & ~ move s --> floor s ∈ req s}"

moving_up :: "state set"
  "moving_up == {s. ~ stop s & up s -->
                 (∃f. floor s ≤ f & f ≤ Max & f ∈ req s)}"

moving_down :: "state set"
  "moving_down == {s. ~ stop s & ~ up s -->
                   (∃f. Min ≤ f & f ≤ floor s & f ∈ req s)}"

metric :: "[int,state] => int"
  "metric ==
     %n s. if floor s < n then (if up s then n - floor s
                                else (floor s - Min) + (n-Min))
           else
           if n < floor s then (if up s then (Max - floor s) + (Max-n)
                                else floor s - n)
```

```
            else 0"
```

locale *Floor* =
  fixes *n*
  assumes *Min_le_n [iff]: "Min ≤ n"*
      and *n_le_Max [iff]: "n ≤ Max"*

lemma *not_mem_distinct: "[| x ∉ A; y ∈ A |] ==> x ≠ y"*
⟨*proof*⟩

declare *Lift_def [THEN def_prg_Init, simp]*

declare *request_act_def [THEN def_act_simp, simp]*
declare *open_act_def [THEN def_act_simp, simp]*
declare *close_act_def [THEN def_act_simp, simp]*
declare *req_up_def [THEN def_act_simp, simp]*
declare *req_down_def [THEN def_act_simp, simp]*
declare *move_up_def [THEN def_act_simp, simp]*
declare *move_down_def [THEN def_act_simp, simp]*
declare *button_press_def [THEN def_act_simp, simp]*

declare *above_def [THEN def_set_simp, simp]*
declare *below_def [THEN def_set_simp, simp]*
declare *queueing_def [THEN def_set_simp, simp]*
declare *goingup_def [THEN def_set_simp, simp]*
declare *goingdown_def [THEN def_set_simp, simp]*
declare *ready_def [THEN def_set_simp, simp]*

declare *bounded_def [simp]*
        open_stop_def [simp]
        open_move_def [simp]
        stop_floor_def [simp]
        moving_up_def [simp]
        moving_down_def [simp]

lemma *open_stop: "Lift ∈ Always open_stop"*
⟨*proof*⟩

lemma *stop_floor: "Lift ∈ Always stop_floor"*
⟨*proof*⟩

lemma *open_move: "Lift ∈ Always open_move"*
⟨*proof*⟩

lemma *moving_up: "Lift ∈ Always moving_up"*
⟨*proof*⟩

lemma *moving_down: "Lift ∈ Always moving_down"*
⟨*proof*⟩

```
lemma bounded: "Lift ∈ Always bounded"
⟨proof⟩
```

19.3 Progress

```
declare moving_def [THEN def_set_simp, simp]
declare stopped_def [THEN def_set_simp, simp]
declare opened_def [THEN def_set_simp, simp]
declare closed_def [THEN def_set_simp, simp]
declare atFloor_def [THEN def_set_simp, simp]
declare Req_def [THEN def_set_simp, simp]
```

The HUG'93 paper mistakenly omits the Req n from these!

```
lemma E_thm01: "Lift ∈ (stopped ∩ atFloor n) LeadsTo (opened ∩ atFloor
n)"
⟨proof⟩
```

```
lemma E_thm02: "Lift ∈ (Req n ∩ stopped - atFloor n) LeadsTo
                       (Req n ∩ opened - atFloor n)"
⟨proof⟩

lemma E_thm03: "Lift ∈ (Req n ∩ opened - atFloor n) LeadsTo
                       (Req n ∩ closed - (atFloor n - queueing))"
⟨proof⟩

lemma E_thm04: "Lift ∈ (Req n ∩ closed ∩ (atFloor n - queueing))
                       LeadsTo (opened ∩ atFloor n)"
⟨proof⟩
```

```
lemmas linorder_leI = linorder_not_less [THEN iffD1]

lemmas (in Floor) le_MinD = Min_le_n [THEN order_antisym]
              and Max_leD = n_le_Max [THEN [2] order_antisym]

declare (in Floor) le_MinD [dest!]
               and linorder_leI [THEN le_MinD, dest!]
               and Max_leD [dest!]
               and linorder_leI [THEN Max_leD, dest!]
```

```
lemma (in Floor) E_thm05c:
    "Lift ∈ (Req n ∩ closed - (atFloor n - queueing))
             LeadsTo ((closed ∩ goingup ∩ Req n)  ∪
                      (closed ∩ goingdown ∩ Req n))"
⟨proof⟩
```

```
lemma (in Floor) lift_2: "Lift ∈ (Req n ∩ closed - (atFloor n - queueing))

             LeadsTo (moving ∩ Req n)"
```
⟨proof⟩

```
declare split_if_asm [split]
```

```
lemma (in Floor) E_thm12a:
     "0 < N ==>
      Lift ∈ (moving ∩ Req n ∩ {s. metric n s = N} ∩
              {s. floor s ∉ req s} ∩ {s. up s})
             LeadsTo
               (moving ∩ Req n ∩ {s. metric n s < N})"
```
⟨proof⟩

```
lemma (in Floor) E_thm12b: "0 < N ==>
      Lift ∈ (moving ∩ Req n ∩ {s. metric n s = N} ∩
              {s. floor s ∉ req s} - {s. up s})
             LeadsTo (moving ∩ Req n ∩ {s. metric n s < N})"
```
⟨proof⟩

```
lemma (in Floor) lift_4:
     "0<N ==> Lift ∈ (moving ∩ Req n ∩ {s. metric n s = N} ∩
                            {s. floor s ∉ req s}) LeadsTo
                           (moving ∩ Req n ∩ {s. metric n s < N})"
```
⟨proof⟩

```
lemma (in Floor) E_thm16a: "0<N
  ==> Lift ∈ (closed ∩ Req n ∩ {s. metric n s = N} ∩ goingup) LeadsTo
             (moving ∩ Req n ∩ {s. metric n s < N})"
```
⟨proof⟩

```
lemma (in Floor) E_thm16b: "0<N ==>
      Lift ∈ (closed ∩ Req n ∩ {s. metric n s = N} ∩ goingdown) LeadsTo

                   (moving ∩ Req n ∩ {s. metric n s < N})"
```
⟨proof⟩

lemma *(***in** *Floor) E_thm16c:*
"0<N ==> Req n ∩ {s. metric n s = N} ⊆ goingup ∪ goingdown"
⟨*proof*⟩

lemma *(***in** *Floor) lift_5:*
"0<N ==> Lift ∈ (closed ∩ Req n ∩ {s. metric n s = N}) LeadsTo
(moving ∩ Req n ∩ {s. metric n s < N})"
⟨*proof*⟩

lemma *(***in** *Floor) metric_eq_0D [dest]:*
"[| metric n s = 0; Min ≤ floor s; floor s ≤ Max |] ==> floor s =
n"
⟨*proof*⟩

lemma *(***in** *Floor) E_thm11: "Lift ∈ (moving ∩ Req n ∩ {s. metric n s = 0})*
LeadsTo
(stopped ∩ atFloor n)"
⟨*proof*⟩

lemma *(***in** *Floor) E_thm13:*
"Lift ∈ (moving ∩ Req n ∩ {s. metric n s = N} ∩ {s. floor s ∈ req s})

LeadsTo (stopped ∩ Req n ∩ {s. metric n s = N} ∩ {s. floor s ∈ req s})"
⟨*proof*⟩

lemma *(***in** *Floor) E_thm14: "0 < N ==>*
Lift ∈
(stopped ∩ Req n ∩ {s. metric n s = N} ∩ {s. floor s ∈ req s})
LeadsTo (opened ∩ Req n ∩ {s. metric n s = N})"
⟨*proof*⟩

lemma *(***in** *Floor) E_thm15: "Lift ∈ (opened ∩ Req n ∩ {s. metric n s = N})*

LeadsTo (closed ∩ Req n ∩ {s. metric n s = N})"
⟨*proof*⟩

lemma *(***in** *Floor) lift_3_Req: "0 < N ==>*
Lift ∈
(moving ∩ Req n ∩ {s. metric n s = N} ∩ {s. floor s ∈ req s})
LeadsTo (moving ∩ Req n ∩ {s. metric n s < N})"

⟨proof⟩

lemma *(in Floor) Always_nonneg: "Lift ∈ Always {s. 0 ≤ metric n s}"*
⟨proof⟩

lemmas *(in Floor) R_thm11 = Always_LeadsTo_weaken [OF Always_nonneg E_thm11]*

lemma *(in Floor) lift_3:*
"Lift ∈ (moving ∩ Req n) LeadsTo (stopped ∩ atFloor n)"
⟨proof⟩

lemma *(in Floor) lift_1: "Lift ∈ (Req n) LeadsTo (opened ∩ atFloor n)"*
⟨proof⟩

end

theory *Mutex* **imports** *"../UNITY_Main"* **begin**

record *state =*

```
p :: bool
m :: int
n :: int
u :: bool
v :: bool
```

types *command = "(state*state) set"*

constdefs

```
U0 :: command
  "U0 == {(s,s'). s' = s (|u:=True, m:=1|) & m s = 0}"

U1 :: command
  "U1 == {(s,s'). s' = s (|p:= v s, m:=2|) & m s = 1}"

U2 :: command
  "U2 == {(s,s'). s' = s (|m:=3|) & ~ p s & m s = 2}"

U3 :: command
  "U3 == {(s,s'). s' = s (|u:=False, m:=4|) & m s = 3}"

U4 :: command
  "U4 == {(s,s'). s' = s (|p:=True, m:=0|) & m s = 4}"
```

```
V0 :: command
  "V0 == {(s,s'). s' = s (|v:=True, n:=1|) & n s = 0}"

V1 :: command
  "V1 == {(s,s'). s' = s (|p:= ~ u s, n:=2|) & n s = 1}"

V2 :: command
  "V2 == {(s,s'). s' = s (|n:=3|) & p s & n s = 2}"

V3 :: command
  "V3 == {(s,s'). s' = s (|v:=False, n:=4|) & n s = 3}"

V4 :: command
  "V4 == {(s,s'). s' = s (|p:=False, n:=0|) & n s = 4}"

Mutex :: "state program"
  "Mutex == mk_total_program
             ({s. ~ u s & ~ v s & m s = 0 & n s = 0},
              {U0, U1, U2, U3, U4, V0, V1, V2, V3, V4},
              UNIV)"
```

```
IU :: "state set"
  "IU == {s. (u s = (1 ≤ m s & m s ≤ 3)) & (m s = 3 --> ~ p s)}"

IV :: "state set"
  "IV == {s. (v s = (1 ≤ n s & n s ≤ 3)) & (n s = 3 --> p s)}"
```

```
bad_IU :: "state set"
  "bad_IU == {s. (u s = (1 ≤ m s & m s ≤ 3)) &
                 (3 ≤ m s & m s ≤ 4 --> ~ p s)}"
```

declare Mutex_def [THEN def_prg_Init, simp]

declare U0_def [THEN def_act_simp, simp]
declare U1_def [THEN def_act_simp, simp]
declare U2_def [THEN def_act_simp, simp]
declare U3_def [THEN def_act_simp, simp]
declare U4_def [THEN def_act_simp, simp]
declare V0_def [THEN def_act_simp, simp]
declare V1_def [THEN def_act_simp, simp]
declare V2_def [THEN def_act_simp, simp]
declare V3_def [THEN def_act_simp, simp]
declare V4_def [THEN def_act_simp, simp]

declare IU_def [THEN def_set_simp, simp]
declare IV_def [THEN def_set_simp, simp]
declare bad_IU_def [THEN def_set_simp, simp]

lemma *IU: "Mutex ∈ Always IU"*
⟨*proof*⟩

lemma *IV: "Mutex ∈ Always IV"*
⟨*proof*⟩

lemma *mutual_exclusion: "Mutex ∈ Always {s. ~ (m s = 3 & n s = 3)}"*
⟨*proof*⟩

lemma *"Mutex ∈ Always bad_IU"*
⟨*proof*⟩

lemma *eq_123: "((1::int) ≤ i & i ≤ 3) = (i = 1 | i = 2 | i = 3)"*
⟨*proof*⟩

lemma *U_F0: "Mutex ∈ {s. m s=2} Unless {s. m s=3}"*
⟨*proof*⟩

lemma *U_F1: "Mutex ∈ {s. m s=1} LeadsTo {s. p s = v s & m s = 2}"*
⟨*proof*⟩

lemma *U_F2: "Mutex ∈ {s. ~ p s & m s = 2} LeadsTo {s. m s = 3}"*
⟨*proof*⟩

lemma *U_F3: "Mutex ∈ {s. m s = 3} LeadsTo {s. p s}"*
⟨*proof*⟩

lemma *U_lemma2: "Mutex ∈ {s. m s = 2} LeadsTo {s. p s}"*
⟨*proof*⟩

lemma *U_lemma1: "Mutex ∈ {s. m s = 1} LeadsTo {s. p s}"*
⟨*proof*⟩

lemma *U_lemma123: "Mutex ∈ {s. 1 ≤ m s & m s ≤ 3} LeadsTo {s. p s}"*
⟨*proof*⟩

lemma *u_Leadsto_p: "Mutex ∈ {s. u s} LeadsTo {s. p s}"*
⟨*proof*⟩

lemma *V_F0: "Mutex ∈ {s. n s=2} Unless {s. n s=3}"*
⟨*proof*⟩

lemma *V_F1:* "Mutex ∈ {s. n s=1} LeadsTo {s. p s = (~ u s) & n s = 2}"
⟨*proof*⟩

lemma *V_F2:* "Mutex ∈ {s. p s & n s = 2} LeadsTo {s. n s = 3}"
⟨*proof*⟩

lemma *V_F3:* "Mutex ∈ {s. n s = 3} LeadsTo {s. ~ p s}"
⟨*proof*⟩

lemma *V_lemma2:* "Mutex ∈ {s. n s = 2} LeadsTo {s. ~ p s}"
⟨*proof*⟩

lemma *V_lemma1:* "Mutex ∈ {s. n s = 1} LeadsTo {s. ~ p s}"
⟨*proof*⟩

lemma *V_lemma123:* "Mutex ∈ {s. 1 ≤ n s & n s ≤ 3} LeadsTo {s. ~ p s}"
⟨*proof*⟩

lemma *v_Leadsto_not_p:* "Mutex ∈ {s. v s} LeadsTo {s. ~ p s}"
⟨*proof*⟩

lemma *m1_Leadsto_3:* "Mutex ∈ {s. m s = 1} LeadsTo {s. m s = 3}"
⟨*proof*⟩

lemma *n1_Leadsto_3:* "Mutex ∈ {s. n s = 1} LeadsTo {s. n s = 3}"
⟨*proof*⟩

end

theory *Reach* **imports** "../UNITY_Main" **begin**

typedecl *vertex*

types *state = "vertex=>bool"*

consts

```
  init :: "vertex"

  edges :: "(vertex*vertex) set"
```

constdefs

```
  asgt :: "[vertex,vertex] => (state*state) set"
    "asgt u v == {(s,s'). s' = s(v:= s u | s v)}"
```

```
  Rprg :: "state program"
    "Rprg == mk_total_program ({%v. v=init}, ⋃(u,v)∈edges. {asgt u v}, UNIV)"

  reach_invariant :: "state set"
    "reach_invariant == {s. (∀v. s v --> (init, v) ∈ edges^*) & s init}"

  fixedpoint :: "state set"
    "fixedpoint == {s. ∀(u,v)∈edges. s u --> s v}"

  metric :: "state => nat"
    "metric s == card {v. ~ s v}"
```

*We assume that the set of vertices is finite

axioms
```
  finite_graph:  "finite (UNIV :: vertex set)"
```

lemma *ifE [elim!]:*
```
    "[| if P then Q else R;
        [| P;   Q |] ==> S;
        [| ~ P; R |] ==> S |] ==> S"
```
⟨*proof*⟩

declare *Rprg_def [THEN def_prg_Init, simp]*

declare *asgt_def [THEN def_act_simp,simp]*

All vertex sets are finite

declare *finite_subset [OF subset_UNIV finite_graph, iff]*

declare *reach_invariant_def [THEN def_set_simp, simp]*

lemma *reach_invariant: "Rprg ∈ Always reach_invariant"*
⟨*proof*⟩

lemma *fixedpoint_invariant_correct:*
```
     "fixedpoint ∩ reach_invariant = { %v. (init, v) ∈ edges^* }"
```
⟨*proof*⟩

lemma *lemma1:*
```
     "FP Rprg ⊆ fixedpoint"
```
⟨*proof*⟩

lemma *lemma2:*
```
     "fixedpoint ⊆ FP Rprg"
```

⟨*proof*⟩

lemma *FP_fixedpoint: "FP Rprg = fixedpoint"*
⟨*proof*⟩

lemma *Compl_fixedpoint: "- fixedpoint = (⋃ (u,v)∈edges. {s. s u & ~ s v})"*
⟨*proof*⟩

lemma *Diff_fixedpoint:*
"A - fixedpoint = (⋃ (u,v)∈edges. A ∩ {s. s u & ~ s v})"
⟨*proof*⟩

lemma *Suc_metric: "~ s x ==> Suc (metric (s(x:=True))) = metric s"*
⟨*proof*⟩

lemma *metric_less [intro!]: "~ s x ==> metric (s(x:=True)) < metric s"*
⟨*proof*⟩

lemma *metric_le: "metric (s(y:=s x | s y)) ≤ metric s"*
⟨*proof*⟩

lemma *LeadsTo_Diff_fixedpoint:*
"Rprg ∈ ((metric-‘{m}) - fixedpoint) LeadsTo (metric-‘(lessThan m))"
⟨*proof*⟩

lemma *LeadsTo_Un_fixedpoint:*
"Rprg ∈ (metric-‘{m}) LeadsTo (metric-‘(lessThan m) ∪ fixedpoint)"
⟨*proof*⟩

lemma *LeadsTo_fixedpoint: "Rprg ∈ UNIV LeadsTo fixedpoint"*
⟨*proof*⟩

lemma *LeadsTo_correct: "Rprg ∈ UNIV LeadsTo { %v. (init, v) ∈ edges^* }"*
⟨*proof*⟩

end

theory *Reachability* **imports** *Detects Reach* **begin**

types *edge = "(vertex*vertex)"*

record *state =*
reach :: "vertex => bool"
nmsg :: "edge => nat"

```
consts root :: "vertex"
       E :: "edge set"
       V :: "vertex set"

inductive_set REACHABLE :: "edge set"
  where
    base: "v ∈ V ==> ((v,v) ∈ REACHABLE)"
  | step: "((u,v) ∈ REACHABLE) & (v,w) ∈ E ==> ((u,w) ∈ REACHABLE)"

constdefs
  reachable :: "vertex => state set"
  "reachable p == {s. reach s p}"

  nmsg_eq :: "nat => edge  => state set"
  "nmsg_eq k == %e. {s. nmsg s e = k}"

  nmsg_gt :: "nat => edge  => state set"
  "nmsg_gt k  == %e. {s. k < nmsg s e}"

  nmsg_gte :: "nat => edge => state set"
  "nmsg_gte k == %e. {s. k ≤ nmsg s e}"

  nmsg_lte  :: "nat => edge => state set"
  "nmsg_lte k  == %e. {s. nmsg s e ≤ k}"

  final :: "state set"
  "final == (⋂v∈V. reachable v <==> {s. (root, v) ∈ REACHABLE}) ∩
            (INTER E (nmsg_eq 0))"

axioms

    Graph1: "root ∈ V"

    Graph2: "(v,w) ∈ E ==> (v ∈ V) & (w ∈ V)"

    MA1:  "F ∈ Always (reachable root)"

    MA2:  "v ∈ V ==> F ∈ Always (- reachable v ∪ {s. ((root,v) ∈ REACHABLE)})"

    MA3:  "[|v ∈ V;w ∈ V|] ==> F ∈ Always (-(nmsg_gt 0 (v,w)) ∪ (reachable
v))"

    MA4:  "(v,w) ∈ E ==>
           F ∈ Always (-(reachable v) ∪ (nmsg_gt 0 (v,w)) ∪ (reachable w))"

    MA5:  "[|v ∈ V; w ∈ V|]
           ==> F ∈ Always (nmsg_gte 0 (v,w) ∩ nmsg_lte (Suc 0) (v,w))"

    MA6:  "[|v ∈ V|] ==> F ∈ Stable (reachable v)"

    MA6b: "[|v ∈ V;w ∈ W|] ==> F ∈ Stable (reachable v ∩ nmsg_lte k (v,w))"

    MA7:  "[|v ∈ V;w ∈ V|] ==> F ∈ UNIV LeadsTo nmsg_eq 0 (v,w)"
```

lemmas *E_imp_in_V_L = Graph2 [THEN conjunct1, standard]*
lemmas *E_imp_in_V_R = Graph2 [THEN conjunct2, standard]*

lemma *lemma2:*
"(v,w) ∈ E ==> F ∈ reachable v LeadsTo nmsg_eq 0 (v,w) ∩ reachable v"
⟨*proof*⟩

lemma *Induction_base: "(v,w) ∈ E ==> F ∈ reachable v LeadsTo reachable w"*
⟨*proof*⟩

lemma *REACHABLE_LeadsTo_reachable:*
"(v,w) ∈ REACHABLE ==> F ∈ reachable v LeadsTo reachable w"
⟨*proof*⟩

lemma *Detects_part1: "F ∈ {s. (root,v) ∈ REACHABLE} LeadsTo reachable v"*
⟨*proof*⟩

lemma *Reachability_Detected:*
"v ∈ V ==> F ∈ (reachable v) Detects {s. (root,v) ∈ REACHABLE}"
⟨*proof*⟩

lemma *LeadsTo_Reachability:*
"v ∈ V ==> F ∈ UNIV LeadsTo (reachable v <==> {s. (root,v) ∈ REACHABLE})"
⟨*proof*⟩

lemma *Eq_lemma1:*
"(reachable v <==> {s. (root,v) ∈ REACHABLE}) =
{s. ((s ∈ reachable v) = ((root,v) ∈ REACHABLE))}"
⟨*proof*⟩

lemma *Eq_lemma2:*
"(reachable v <==> (if (root,v) ∈ REACHABLE then UNIV else {})) =
{s. ((s ∈ reachable v) = ((root,v) ∈ REACHABLE))}"
⟨*proof*⟩

lemma *final_lemma1:*
"(⋂v ∈ V. ⋂w ∈ V. {s. ((s ∈ reachable v) = ((root,v) ∈ REACHABLE))
&
s ∈ nmsg_eq 0 (v,w)})

```
      ⊆ final"
```
⟨proof⟩

lemma *final_lemma2:*
```
 "E≠{}
  ==> (⋂v ∈ V. ⋂e ∈ E. {s. ((s ∈ reachable v) = ((root,v) ∈ REACHABLE))}

                              ∩ nmsg_eq 0 e)    ⊆  final"
```
⟨proof⟩

lemma *final_lemma3:*
```
      "E≠{}
       ==> (⋂v ∈ V. ⋂e ∈ E.
             (reachable v <==> {s. (root,v) ∈ REACHABLE}) ∩ nmsg_eq 0 e)
           ⊆ final"
```
⟨proof⟩

lemma *final_lemma4:*
```
      "E≠{}
       ==> (⋂v ∈ V. ⋂e ∈ E.
             {s. ((s ∈ reachable v) = ((root,v) ∈ REACHABLE))} ∩ nmsg_eq 0
e)
           = final"
```
⟨proof⟩

lemma *final_lemma5:*
```
      "E≠{}
       ==> (⋂v ∈ V. ⋂e ∈ E.
             ((reachable v) <==> {s. (root,v) ∈ REACHABLE}) ∩ nmsg_eq 0 e)

           = final"
```
⟨proof⟩

lemma *final_lemma6:*
```
      "(⋂v ∈ V. ⋂w ∈ V.
        (reachable v <==> {s. (root,v) ∈ REACHABLE}) ∩ nmsg_eq 0 (v,w))
       ⊆ final"
```
⟨proof⟩

lemma *final_lemma7:*
```
      "final =
       (⋂v ∈ V. ⋂w ∈ V.
        ((reachable v) <==> {s. (root,v) ∈ REACHABLE}) ∩
        (-{s. (v,w) ∈ E} ∪ (nmsg_eq 0 (v,w))))"
```
⟨proof⟩

lemma *not_REACHABLE_imp_Stable_not_reachable:*
"[| v ∈ V; (root,v) ∉ REACHABLE |] ==> F ∈ Stable (- reachable v)"
⟨*proof*⟩

lemma *Stable_reachable_EQ_R:*
"v ∈ V ==> F ∈ Stable (reachable v <==> {s. (root,v) ∈ REACHABLE})"
⟨*proof*⟩

lemma *lemma4:*
"((nmsg_gte 0 (v,w) ∩ nmsg_lte (Suc 0) (v,w)) ∩
(- nmsg_gt 0 (v,w) ∪ A))
⊆ A ∪ nmsg_eq 0 (v,w)"
⟨*proof*⟩

lemma *lemma5:*
"reachable v ∩ nmsg_eq 0 (v,w) =
((nmsg_gte 0 (v,w) ∩ nmsg_lte (Suc 0) (v,w)) ∩
(reachable v ∩ nmsg_lte 0 (v,w)))"
⟨*proof*⟩

lemma *lemma6:*
"- nmsg_gt 0 (v,w) ∪ reachable v ⊆ nmsg_eq 0 (v,w) ∪ reachable v"
⟨*proof*⟩

lemma *Always_reachable_OR_nmsg_0:*
"[|v ∈ V; w ∈ V|] ==> F ∈ Always (reachable v ∪ nmsg_eq 0 (v,w))"
⟨*proof*⟩

lemma *Stable_reachable_AND_nmsg_0:*
"[|v ∈ V; w ∈ V|] ==> F ∈ Stable (reachable v ∩ nmsg_eq 0 (v,w))"
⟨*proof*⟩

lemma *Stable_nmsg_0_OR_reachable:*
"[|v ∈ V; w ∈ V|] ==> F ∈ Stable (nmsg_eq 0 (v,w) ∪ reachable v)"
⟨*proof*⟩

lemma *not_REACHABLE_imp_Stable_not_reachable_AND_nmsg_0:*
"[| v ∈ V; w ∈ V; (root,v) ∉ REACHABLE |]
==> F ∈ Stable (- reachable v ∩ nmsg_eq 0 (v,w))"
⟨*proof*⟩

lemma *Stable_reachable_EQ_R_AND_nmsg_0:*
"[| v ∈ V; w ∈ V |]
==> F ∈ Stable ((reachable v <==> {s. (root,v) ∈ REACHABLE}) ∩
nmsg_eq 0 (v,w))"
⟨*proof*⟩

```
lemma UNIV_lemma: "UNIV ⊆ (⋂ v ∈ V. UNIV)"
⟨proof⟩

lemmas UNIV_LeadsTo_completion =
    LeadsTo_weaken_L [OF Finite_stable_completion UNIV_lemma]

lemma LeadsTo_final_E_empty: "E={} ==> F ∈ UNIV LeadsTo final"
⟨proof⟩


lemma Leadsto_reachability_AND_nmsg_0:
     "[| v ∈ V; w ∈ V |]
      ==> F ∈ UNIV LeadsTo
             ((reachable v <==> {s. (root,v): REACHABLE}) ∩ nmsg_eq 0 (v,w))"
⟨proof⟩

lemma LeadsTo_final_E_NOT_empty: "E≠{} ==> F ∈ UNIV LeadsTo final"
⟨proof⟩

lemma LeadsTo_final: "F ∈ UNIV LeadsTo final"
⟨proof⟩




lemma Stable_final_E_empty: "E={} ==> F ∈ Stable final"
⟨proof⟩


lemma Stable_final_E_NOT_empty: "E≠{} ==> F ∈ Stable final"
⟨proof⟩

lemma Stable_final: "F ∈ Stable final"
⟨proof⟩

end
```

20 Analyzing the Needham-Schroeder Public-Key Protocol in UNITY

```
theory NSP_Bad imports "../../Auth/Public" "../UNITY_Main" begin
```

This is the flawed version, vulnerable to Lowe's attack. From page 260 of Burrows, Abadi and Needham. A Logic of Authentication. Proc. Royal Soc. 426 (1989).

```
types state = "event list"
```

constdefs

```
  Fake :: "(state*state) set"
    "Fake == {(s,s').
             ∃B X. s' = Says Spy B X # s
                   & X ∈ synth (analz (spies s))}"




  NS1 :: "(state*state) set"
    "NS1 == {(s1,s').
            ∃A1 B NA.
                s' = Says A1 B (Crypt (pubK B) {|Nonce NA, Agent A1|}) # s1
              & Nonce NA ∉ used s1}"


  NS2 :: "(state*state) set"
    "NS2 == {(s2,s').
            ∃A' A2 B NA NB.
                s' = Says B A2 (Crypt (pubK A2) {|Nonce NA, Nonce NB|}) #
s2
              & Says A' B (Crypt (pubK B) {|Nonce NA, Agent A2|}) ∈ set s2
              & Nonce NB ∉ used s2}"


  NS3 :: "(state*state) set"
    "NS3 == {(s3,s').
            ∃A3 B' B NA NB.
                s' = Says A3 B (Crypt (pubK B) (Nonce NB)) # s3
              & Says A3  B (Crypt (pubK B) {|Nonce NA, Agent A3|}) ∈ set
s3
              & Says B' A3 (Crypt (pubK A3) {|Nonce NA, Nonce NB|}) ∈ set
s3}"
```

constdefs

```
  Nprg :: "state program"

    "Nprg == mk_total_program({[]}, {Fake, NS1, NS2, NS3}, UNIV)"
```

declare *spies_partsEs [elim]*
declare *analz_into_parts [dest]*
declare *Fake_parts_insert_in_Un [dest]*

For other theories, e.g. Mutex and Lift, using [iff] slows proofs down. Here, it facilitates re-use of the Auth proofs.

declare *Fake_def [THEN def_act_simp, iff]*
declare *NS1_def [THEN def_act_simp, iff]*
declare *NS2_def [THEN def_act_simp, iff]*
declare *NS3_def [THEN def_act_simp, iff]*

declare *Nprg_def [THEN def_prg_Init, simp]*

A "possibility property": there are traces that reach the end. Replace by LEAD-STO proof!

```
lemma "A ≠ B ==>
       ∃NB. ∃s ∈ reachable Nprg. Says A B (Crypt (pubK B) (Nonce NB)) ∈
set s"
⟨proof⟩
```

20.1 Inductive Proofs about ns_public

```
lemma ns_constrainsI:
     "(!!act s s'. [| act ∈ {Id, Fake, NS1, NS2, NS3};
                      (s,s') ∈ act;  s ∈ A |] ==> s' ∈ A')
      ==> Nprg ∈ A co A'"
⟨proof⟩
```

This ML code does the inductions directly.

⟨*ML*⟩

Converts invariants into statements about reachable states

```
lemmas Always_Collect_reachableD =
     Always_includes_reachable [THEN subsetD, THEN CollectD]
```

Spy never sees another agent's private key! (unless it's bad at start)

```
lemma Spy_see_priK:
     "Nprg ∈ Always {s. (Key (priK A) ∈ parts (spies s)) = (A ∈ bad)}"
⟨proof⟩
declare Spy_see_priK [THEN Always_Collect_reachableD, simp]

lemma Spy_analz_priK:
     "Nprg ∈ Always {s. (Key (priK A) ∈ analz (spies s)) = (A ∈ bad)}"
⟨proof⟩
declare Spy_analz_priK [THEN Always_Collect_reachableD, simp]
```

20.2 Authenticity properties obtained from NS2

It is impossible to re-use a nonce in both NS1 and NS2 provided the nonce is secret. (Honest users generate fresh nonces.)

```
lemma no_nonce_NS1_NS2:
 "Nprg
  ∈ Always {s. Crypt (pubK C) {|NA', Nonce NA|} ∈ parts (spies s) -->
                Crypt (pubK B) {|Nonce NA, Agent A|} ∈ parts (spies s) -->
                Nonce NA ∈ analz (spies s)}"
⟨proof⟩
```

Adding it to the claset slows down proofs...

```
lemmas no_nonce_NS1_NS2_reachable =
       no_nonce_NS1_NS2 [THEN Always_Collect_reachableD, rule_format]
```

Unicity for NS1: nonce NA identifies agents A and B

```
lemma unique_NA_lemma:
```

```
    "Nprg
  ∈ Always {s. Nonce NA ∉ analz (spies s) -->
                Crypt(pubK B) {|Nonce NA, Agent A|} ∈ parts(spies s) -->
                Crypt(pubK B') {|Nonce NA, Agent A'|} ∈ parts(spies s) -->
                A=A' & B=B'}"
```
⟨*proof*⟩

Unicity for NS1: nonce NA identifies agents A and B

lemma *unique_NA:*
```
    "[| Crypt(pubK B)  {|Nonce NA, Agent A|} ∈ parts(spies s);
        Crypt(pubK B') {|Nonce NA, Agent A'|} ∈ parts(spies s);
        Nonce NA ∉ analz (spies s);
        s ∈ reachable Nprg |]
     ==> A=A' & B=B'"
```
⟨*proof*⟩

Secrecy: Spy does not see the nonce sent in msg NS1 if A and B are secure

lemma *Spy_not_see_NA:*
```
    "[| A ∉ bad;  B ∉ bad |]
  ==> Nprg ∈ Always
             {s. Says A B (Crypt(pubK B) {|Nonce NA, Agent A|}) ∈ set s
                 --> Nonce NA ∉ analz (spies s)}"
```
⟨*proof*⟩

Authentication for A: if she receives message 2 and has used NA to start a run, then B has sent message 2.

lemma *A_trusts_NS2:*
```
 "[| A ∉ bad;  B ∉ bad |]
  ==> Nprg ∈ Always
             {s. Says A B (Crypt(pubK B) {|Nonce NA, Agent A|}) ∈ set s &
                 Crypt(pubK A) {|Nonce NA, Nonce NB|} ∈ parts (knows Spy
s)
         --> Says B A (Crypt(pubK A) {|Nonce NA, Nonce NB|}) ∈ set s}"
```

⟨*proof*⟩

If the encrypted message appears then it originated with Alice in NS1

lemma *B_trusts_NS1:*
```
    "Nprg ∈ Always
             {s. Nonce NA ∉ analz (spies s) -->
                 Crypt (pubK B) {|Nonce NA, Agent A|} ∈ parts (spies s)
         --> Says A B (Crypt (pubK B) {|Nonce NA, Agent A|}) ∈ set s}"
```
⟨*proof*⟩

20.3 Authenticity properties obtained from NS2

Unicity for NS2: nonce NB identifies nonce NA and agent A. Proof closely follows that of `unique_NA`.

lemma *unique_NB_lemma:*
```
 "Nprg
  ∈ Always {s. Nonce NB ∉ analz (spies s)  -->
                Crypt (pubK A) {|Nonce NA, Nonce NB|} ∈ parts (spies s) -->
```

```
                Crypt(pubK A'){|Nonce NA', Nonce NB|} ∈ parts(spies s) -->
                A=A' & NA=NA'}"
```
⟨proof⟩

lemma *unique_NB:*
```
     "[| Crypt(pubK A) {|Nonce NA, Nonce NB|} ∈ parts(spies s);
         Crypt(pubK A'){|Nonce NA', Nonce NB|} ∈ parts(spies s);
         Nonce NB ∉ analz (spies s);
         s ∈ reachable Nprg |]
      ==> A=A' & NA=NA'"
```
⟨proof⟩

NB remains secret PROVIDED Alice never responds with round 3

lemma *Spy_not_see_NB:*
```
     "[| A ∉ bad;  B ∉ bad |]
  ==> Nprg ∈ Always
            {s. Says B A (Crypt (pubK A) {|Nonce NA, Nonce NB|}) ∈ set s
&
                (ALL C. Says A C (Crypt (pubK C) (Nonce NB)) ∉ set s)
                --> Nonce NB ∉ analz (spies s)}"
```
⟨proof⟩

Authentication for B: if he receives message 3 and has used NB in message 2, then A has sent message 3–to somebody....

lemma *B_trusts_NS3:*
```
     "[| A ∉ bad;  B ∉ bad |]
  ==> Nprg ∈ Always
            {s. Crypt (pubK B) (Nonce NB) ∈ parts (spies s) &
                Says B A  (Crypt (pubK A) {|Nonce NA, Nonce NB|}) ∈ set
s
                --> (∃C. Says A C (Crypt (pubK C) (Nonce NB)) ∈ set s)}"
```

⟨proof⟩

Can we strengthen the secrecy theorem? NO

lemma *"[| A ∉ bad; B ∉ bad |]*
```
  ==> Nprg ∈ Always
            {s. Says B A (Crypt (pubK A) {|Nonce NA, Nonce NB|}) ∈ set s
                --> Nonce NB ∉ analz (spies s)}"
```
⟨proof⟩

end

theory *Handshake* **imports** *"../UNITY_Main"* **begin**

record *state =*
```
  BB :: bool
  NF :: nat
```

```
  NG :: nat
```

constdefs

```
  cmdF :: "(state*state) set"
    "cmdF == {(s,s'). s' = s (|NF:= Suc(NF s), BB:=False|) & BB s}"

  F :: "state program"
    "F == mk_total_program ({s. NF s = 0 & BB s}, {cmdF}, UNIV)"


  cmdG :: "(state*state) set"
    "cmdG == {(s,s'). s' = s (|NG:= Suc(NG s), BB:=True|) & ~ BB s}"

  G :: "state program"
    "G == mk_total_program ({s. NG s = 0 & BB s}, {cmdG}, UNIV)"


  invFG :: "state set"
    "invFG == {s. NG s <= NF s & NF s <= Suc (NG s) & (BB s = (NF s = NG s))}"
```

declare *F_def [THEN def_prg_Init, simp]*
G_def [THEN def_prg_Init, simp]

cmdF_def [THEN def_act_simp, simp]
cmdG_def [THEN def_act_simp, simp]

invFG_def [THEN def_set_simp, simp]

lemma *invFG: "(F Join G) : Always invFG"*
⟨*proof*⟩

lemma *lemma2_1: "(F Join G) : ({s. NF s = k} - {s. BB s}) LeadsTo*
({s. NF s = k} Int {s. BB s})"
⟨*proof*⟩

lemma *lemma2_2: "(F Join G) : ({s. NF s = k} Int {s. BB s}) LeadsTo*
{s. k < NF s}"
⟨*proof*⟩

lemma *progress: "(F Join G) : UNIV LeadsTo {s. m < NF s}"*
⟨*proof*⟩

end

21 A Family of Similar Counters: Original Version

theory *Counter* **imports** *"../UNITY_Main"* **begin**

datatype *name = C | c nat*
types *state = "name=>int"*

consts
  sum :: "[nat,state]=>int"
  sumj :: "[nat, nat, state]=>int"

primrec
  "sum 0 s = 0"
  "sum (Suc i) s = s (c i) + sum i s"

primrec
  "sumj 0 i s = 0"
  "sumj (Suc n) i s = (if n=i then sum n s else s (c n) + sumj n i s)"

types *command = "(state*state)set"*

constdefs
  a :: "nat=>command"
 "a i == {(s, s'). s'=s(c i:= s (c i) + 1, C:= s C + 1)}"

  Component :: "nat => state program"
  "Component i ==
    mk_total_program({s. s C = 0 & s (c i) = 0}, {a i},
                    $\bigcup$G ∈ preserves (%s. s (c i)). Acts G)"

declare *Component_def [THEN def_prg_Init, simp]*
declare *a_def [THEN def_act_simp, simp]*

lemma *sum_upd_gt [rule_format]: "∀n. I<n --> sum I (s(c n := x)) = sum I s"*
⟨*proof*⟩

lemma *sum_upd_eq: "sum I (s(c I := x)) = sum I s"*
⟨*proof*⟩

lemma *sum_upd_C: "sum I (s(C := x)) = sum I s"*
⟨*proof*⟩

lemma *sumj_upd_ci: "sumj I i (s(c i := x)) = sumj I i s"*
⟨*proof*⟩

lemma *sumj_upd_C: "sumj I i (s(C := x)) = sumj I i s"*
⟨*proof*⟩

lemma *sumj_sum_gt [rule_format]: "∀i. I<i--> (sumj I i s = sum I s)"*
⟨*proof*⟩

lemma *sumj_sum_eq: "(sumj I I s = sum I s)"*

⟨*proof*⟩

```
lemma sum_sumj [rule_format]: "∀i. i<I-->(sum I s = s (c i) +  sumj I i
s)"
```
⟨*proof*⟩

```
lemma p2: "Component i ∈ stable {s. s C = s (c i) + k}"
```
⟨*proof*⟩

```
lemma p3: "Component i ∈ stable {s. ∀v. v≠c i & v≠C --> s v = k v}"
```
⟨*proof*⟩

```
lemma p2_p3_lemma1:
"(∀k. Component i ∈ stable ({s. s C = s (c i) + sumj I i k}
                   ∩ {s. ∀v. v≠c i & v≠C --> s v = k v}))
   = (Component i ∈ stable {s. s C = s (c i) + sumj I i s})"
```
⟨*proof*⟩

```
lemma p2_p3_lemma2:
"∀k. Component i ∈ stable ({s. s C = s (c i) + sumj I i k} Int
                            {s. ∀v. v≠c i & v≠C --> s v = k v})"
```
⟨*proof*⟩

```
lemma p2_p3: "Component i ∈ stable {s.  s C = s (c i) + sumj I i s}"
```
⟨*proof*⟩

```
lemma sum_0' [rule_format]: "(∀i. i < I --> s (c i) = 0) --> sum I s = 0"
```
⟨*proof*⟩

```
lemma safety:
     "0<I ==> (⨆i ∈ {i. i<I}. Component i) ∈ invariant {s. s C = sum I s}"
```
⟨*proof*⟩

end

22 A Family of Similar Counters: Version with Compatibility

theory *Counterc* **imports** *"../UNITY_Main"* **begin**

typedecl *state*

```
consts
  C :: "state=>int"
  c :: "state=>nat=>int"
```

```
consts
  sum  :: "[nat,state]=>int"
  sumj :: "[nat, nat, state]=>int"

primrec
  "sum 0 s = 0"
  "sum (Suc i) s = (c s) i + sum i s"

primrec
  "sumj 0 i s = 0"
  "sumj (Suc n) i s = (if n=i then sum n s else (c s) n + sumj n i s)"

types command = "(state*state)set"

constdefs
  a :: "nat=>command"
 "a i == {(s, s'). (c s') i = (c s) i + 1 & (C s') = (C s) + 1}"

  Component :: "nat => state program"
  "Component i == mk_total_program({s. C s = 0 & (c s) i = 0},
                                   {a i},
                                   ⋃G ∈ preserves (%s. (c s) i). Acts G)"


declare Component_def [THEN def_prg_Init, simp]
declare Component_def [THEN def_prg_AllowedActs, simp]
declare a_def [THEN def_act_simp, simp]


lemma sum_sumj_eq1 [rule_format]: "∀i. I<i--> (sum I s = sumj I i s)"
⟨proof⟩

lemma sum_sumj_eq2 [rule_format]: "i<I --> sum I s  = c s i + sumj I i s"
⟨proof⟩

lemma sum_ext [rule_format]:
     "(∀i. i<I --> c s' i = c s i) --> (sum I s' = sum I s)"
⟨proof⟩

lemma sumj_ext [rule_format]:
     "(∀j. j<I & j≠i --> c s' j =  c s j) --> (sumj I i s' = sumj I i s)"
⟨proof⟩


lemma sum0 [rule_format]: "(∀i. i<I --> c s i = 0) -->  sum I s = 0"
⟨proof⟩




lemma Component_ok_iff:
     "(Component i ok G) =
      (G ∈ preserves (%s. c s i) & Component i ∈ Allowed G)"
⟨proof⟩
```

declare *Component_ok_iff [iff]*
declare *OK_iff_ok [iff]*
declare *preserves_def [simp]*

lemma *p2: "Component i ∈ stable {s. C s = (c s) i + k}"*
⟨*proof*⟩

lemma *p3:*
```
    "[| OK I Component; i∈I |]
     ==> Component i ∈ stable {s. ∀j∈I. j≠i --> c s j = c k j}"
```
⟨*proof*⟩

lemma *p2_p3_lemma1:*
```
    "[| OK {i. i<I} Component; i<I |] ==>
     ∀k. Component i ∈ stable ({s. C s = c s i + sumj I i k} Int
                               {s. ∀j∈{i. i<I}. j≠i --> c s j = c k j})"
```
⟨*proof*⟩

lemma *p2_p3_lemma2:*
```
    "(∀k. F ∈ stable ({s. C s = (c s) i + sumj I i k} Int
                      {s. ∀j∈{i. i<I}. j≠i --> c s j = c k j}))
     ==> (F ∈ stable {s. C s = c s i + sumj I i s})"
```
⟨*proof*⟩

lemma *p2_p3:*
```
    "[| OK {i. i<I} Component; i<I |]
     ==> Component i ∈ stable {s. C s = c s i + sumj I i s}"
```
⟨*proof*⟩

lemma *safety:*
```
    "[| 0<I; OK {i. i<I} Component |]
     ==> (⨆i∈{i. i<I}. (Component i)) ∈ invariant {s. C s = sum I s}"
```
⟨*proof*⟩

end

theory *PriorityAux*
imports *"../UNITY_Main"*
begin

typedecl *vertex*

constdefs
```
  symcl :: "(vertex*vertex)set=>(vertex*vertex)set"
  "symcl r == r ∪ (r^-1)"
```
— symmetric closure: removes the orientation of a relation

```
neighbors :: "[vertex, (vertex*vertex)set]=>vertex set"
"neighbors i r == ((r ∪ r^-1)``{i}) - {i}"
```
— Neighbors of a vertex i

```
R :: "[vertex, (vertex*vertex)set]=>vertex set"
"R i r == r``{i}"

A :: "[vertex, (vertex*vertex)set]=>vertex set"
"A i r == (r^-1)``{i}"

reach :: "[vertex, (vertex*vertex)set]=> vertex set"
"reach i r == (r^+)``{i}"
```
— reachable and above vertices: the original notation was R* and A*

```
above :: "[vertex, (vertex*vertex)set]=> vertex set"
"above i r == ((r^-1)^+)``{i}"

reverse :: "[vertex, (vertex*vertex) set]=>(vertex*vertex)set"
"reverse i r == (r - {(x,y). x=i | y=i} ∩ r) ∪ ({(x,y). x=i|y=i} ∩ r)^-1"

derive1 :: "[vertex, (vertex*vertex)set, (vertex*vertex)set]=>bool"
```
— The original definition
```
"derive1 i r q == symcl r = symcl q &
                  (∀k k'. k≠i & k'≠i -->((k,k'):r) = ((k,k'):q)) &
                  A i r = {} & R i q = {}"

derive :: "[vertex, (vertex*vertex)set, (vertex*vertex)set]=>bool"
```
— Our alternative definition
```
"derive i r q == A i r = {} & (q = reverse i r)"
```

axioms
```
finite_vertex_univ:  "finite (UNIV :: vertex set)"
```
— we assume that the universe of vertices is finite

declare `derive_def [simp] derive1_def [simp] symcl_def [simp]`
`A_def [simp] R_def [simp]`
`above_def [simp] reach_def [simp]`
`reverse_def [simp] neighbors_def [simp]`

All vertex sets are finite

declare `finite_subset [OF subset_UNIV finite_vertex_univ, iff]`

and relatons over vertex are finite too

lemmas `finite_UNIV_Prod =`
`finite_Prod_UNIV [OF finite_vertex_univ finite_vertex_univ]`

declare `finite_subset [OF subset_UNIV finite_UNIV_Prod, iff]`

lemma `image0_trancl_iff_image0_r: "((r^+)``{i} = {}) = (r``{i} = {})"`
⟨*proof*⟩

lemma *image0_r_iff_image0_trancl:* "(r``{i}={}) = (ALL x. ((i,x):r^+) = False)"
⟨*proof*⟩

lemma *acyclic_eq_wf:* "!!r::(vertex*vertex)set. acyclic r = wf r"
⟨*proof*⟩

lemma *derive_derive1_eq:* "derive i r q = derive1 i r q"
⟨*proof*⟩

lemma *lemma1_a:*
"[| x ∈ reach i q; derive1 k r q |] ==> x≠k --> x ∈ reach i r"
⟨*proof*⟩

lemma *reach_lemma:* "derive k r q ==> reach i q ⊆ (reach i r ∪ {k})"
⟨*proof*⟩

lemma *reach_above_lemma:*
"(∀i. reach i q ⊆ (reach i r ∪ {k})) =
(∀x. x≠k --> (∀i. i ∉ above x r --> i ∉ above x q))"
⟨*proof*⟩

lemma *maximal_converse_image0:*
"(z, i):r^+ ==> (∀y. (y, z):r --> (y,i) ∉ r^+) = ((r^-1)``{z}={})"
⟨*proof*⟩

lemma *above_lemma_a:*
"acyclic r ==> A i r≠{}-->(∃j ∈ above i r. A j r = {})"
⟨*proof*⟩

lemma *above_lemma_b:*
"acyclic r ==> above i r≠{}-->(∃j ∈ above i r. above j r = {})"
⟨*proof*⟩

end

23 The priority system

theory *Priority* **imports** *PriorityAux* **begin**

From Charpentier and Chandy, Examples of Program Composition Illustrating the Use of Universal Properties In J. Rolim (editor), Parallel and Distributed Processing, Spriner LNCS 1586 (1999), pages 1215-1227.

types *state* = "(vertex*vertex)set"
types *command* = "vertex=>(state*state)set"

consts

```
init :: "(vertex*vertex)set"
```
— the initial state

Following the definitions given in section 4.4

constdefs

```
highest :: "[vertex, (vertex*vertex)set]=>bool"
"highest i r == A i r = {}"
```
— i has highest priority in r

```
lowest :: "[vertex, (vertex*vertex)set]=>bool"
"lowest i r == R i r = {}"
```
— i has lowest priority in r

```
act :: command
"act i == {(s, s'). s'=reverse i s & highest i s}"

Component :: "vertex=>state program"
"Component i == mk_total_program({init}, {act i}, UNIV)"
```
— All components start with the same initial state

Some Abbreviations

constdefs

```
Highest :: "vertex=>state set"
"Highest i == {s. highest i s}"

Lowest :: "vertex=>state set"
"Lowest i == {s. lowest i s}"

Acyclic :: "state set"
"Acyclic == {s. acyclic s}"


Maximal :: "state set"
```
— Every "above" set has a maximal vertex

```
"Maximal == ⋂i. {s. ~highest i s-->(∃j ∈ above i  s. highest j s)}"

Maximal' :: "state set"
```
— Maximal vertex: equivalent definition

```
"Maximal' == ⋂i. Highest i Un (⋃j. {s. j ∈ above i s} Int Highest j)"


Safety :: "state set"
"Safety == ⋂i. {s. highest i s --> (∀j ∈ neighbors i s. ~highest j s)}"




system :: "state program"
"system == JN i. Component i"
```

declare *highest_def [simp] lowest_def [simp]*
declare *Highest_def [THEN def_set_simp, simp]*

and *Lowest_def [THEN def_set_simp, simp]*

declare *Component_def [THEN def_prg_Init, simp]*
declare *act_def [THEN def_act_simp, simp]*

23.1 Component correctness proofs

neighbors is stable

lemma *Component_neighbors_stable: "Component i ∈ stable {s. neighbors k s = n}"*
⟨*proof*⟩

property 4

lemma *Component_waits_priority: "Component i: {s. ((i,j):s) = b} Int (- Highest i) co {s. ((i,j):s)=b}"*
⟨*proof*⟩

property 5: charpentier and Chandy mistakenly express it as 'transient Highest i'. Consider the case where i has neighbors

lemma *Component_yields_priority:*
"Component i: {s. neighbors i s ≠ {}} Int Highest i
ensures - Highest i"
⟨*proof*⟩

or better

lemma *Component_yields_priority': "Component i ∈ Highest i ensures Lowest i"*
⟨*proof*⟩

property 6: Component doesn't introduce cycle

lemma *Component_well_behaves: "Component i ∈ Highest i co Highest i Un Lowest i"*
⟨*proof*⟩

property 7: local axiom

lemma *locality: "Component i ∈ stable {s. ∀ j k. j≠i & k≠i--> ((j,k):s) = b j k}"*
⟨*proof*⟩

23.2 System properties

property 8: strictly universal

lemma *Safety: "system ∈ stable Safety"*
⟨*proof*⟩

property 13: universal

lemma *p13: "system ∈ {s. s = q} co {s. s=q} Un {s. ∃ i. derive i q s}"*
⟨*proof*⟩

property 14: the 'above set' of a Component that hasn't got priority doesn't increase

lemma *above_not_increase:*
```
"system ∈ -Highest i Int {s. j∉above i s} co {s. j∉above i s}"
```
⟨*proof*⟩

lemma *above_not_increase':*
```
"system ∈ -Highest i Int {s. above i s = x} co {s. above i s <= x}"
```
⟨*proof*⟩

p15: universal property: all Components well behave

lemma *system_well_behaves [rule_format]:*
```
"∀ i. system ∈ Highest i co Highest i Un Lowest i"
```
⟨*proof*⟩

lemma *Acyclic_eq: "Acyclic = (⋂ i. {s. i∉above i s})"*
⟨*proof*⟩

lemmas *system_co =*
```
constrains_Un [OF above_not_increase [rule_format] system_well_behaves]
```

lemma *Acyclic_stable: "system ∈ stable Acyclic"*
⟨*proof*⟩

lemma *Acyclic_subset_Maximal: "Acyclic <= Maximal"*
⟨*proof*⟩

property 17: original one is an invariant

lemma *Acyclic_Maximal_stable: "system ∈ stable (Acyclic Int Maximal)"*
⟨*proof*⟩

property 5: existential property

lemma *Highest_leadsTo_Lowest: "system ∈ Highest i leadsTo Lowest i"*
⟨*proof*⟩

a lowest i can never be in any abover set

lemma *Lowest_above_subset: "Lowest i <= (⋂ k. {s. i∉above k s})"*
⟨*proof*⟩

property 18: a simpler proof than the original, one which uses psp

lemma *Highest_escapes_above: "system ∈ Highest i leadsTo (⋂ k. {s. i∉above k s})"*
⟨*proof*⟩

lemma *Highest_escapes_above':*
```
"system ∈ Highest j Int {s. j ∈ above i s} leadsTo {s. j∉above i s}"
```
⟨*proof*⟩

23.3 The main result: above set decreases

The original proof of the following formula was wrong

lemma *Highest_iff_above0: "Highest i = {s. above i s ={}}"*
⟨*proof*⟩

lemmas *above_decreases_lemma =*
    psp [THEN leadsTo_weaken, OF Highest_escapes_above' above_not_increase']

lemma *above_decreases:*
    "system ∈ (⋃j. {s. above i s = x} Int {s. j ∈ above i s} Int Highest j)
        leadsTo {s. above i s < x}"
⟨*proof*⟩

lemma *Maximal_eq_Maximal': "Maximal = Maximal'"*
⟨*proof*⟩

lemma *Acyclic_subset:*
  "x≠{} ==>
  Acyclic Int {s. above i s = x} <=
  (⋃j. {s. above i s = x} Int {s. j ∈ above i s} Int Highest j)"
⟨*proof*⟩

lemmas *above_decreases' = leadsTo_weaken_L [OF above_decreases Acyclic_subset]*
lemmas *above_decreases_psp = psp_stable [OF above_decreases' Acyclic_stable]*

lemma *above_decreases_psp':*
"x≠{}==> system ∈ Acyclic Int {s. above i s = x} leadsTo
        Acyclic Int {s. above i s < x}"
⟨*proof*⟩

lemmas *finite_psubset_induct = wf_finite_psubset [THEN leadsTo_wf_induct]*

lemma *Progress: "system ∈ Acyclic leadsTo Highest i"*
⟨*proof*⟩

We have proved all (relevant) theorems given in the paper. We didn't assume any thing about the relation *r*. It is not necessary that *r* be a priority relation as assumed in the original proof. It suffices that we start from a state which is finite and acyclic.

end

theory *TimerArray* **imports** *"../UNITY_Main"* **begin**

types *'a state = "nat * 'a"*

constdefs
  count :: "'a state => nat"
    "count s == fst s"

```
  decr  :: "('a state * 'a state) set"
    "decr == UN n uu. {((Suc n, uu), (n,uu))}"

  Timer :: "'a state program"
    "Timer == mk_total_program (UNIV, {decr}, UNIV)"
```

declare *Timer_def [THEN def_prg_Init, simp]*

declare *count_def [simp] decr_def [simp]*

lemma *Timer_leadsTo_zero: "Timer : UNIV leadsTo {s. count s = 0}"*
⟨*proof*⟩

lemma *Timer_preserves_snd [iff]: "Timer : preserves snd"*
⟨*proof*⟩

declare *PLam_stable [simp]*

lemma *TimerArray_leadsTo_zero:*
```
     "finite I
      ==> (plam i: I. Timer) : UNIV leadsTo {(s,uu). ALL i:I. s i = 0}"
```
⟨*proof*⟩

end

24 Projections of State Sets

theory *Project* **imports** *Extend* **begin**

constdefs
```
  projecting :: "['c program => 'c set, 'a*'b => 'c,
                  'a program, 'c program set, 'a program set] => bool"
    "projecting C h F X' X ==
       ∀G. extend h F⊔G ∈ X' --> F⊔project h (C G) G ∈ X"

  extending :: "['c program => 'c set, 'a*'b => 'c, 'a program,
                 'c program set, 'a program set] => bool"
    "extending C h F Y' Y ==
       ∀G. extend h F  ok G --> F⊔project h (C G) G ∈ Y
             --> extend h F⊔G ∈ Y'"

  subset_closed :: "'a set set => bool"
    "subset_closed U == ∀A ∈ U. Pow A ⊆ U"
```

lemma *(in Extend) project_extend_constrains_I:*
```
     "F ∈ A co B ==> project h C (extend h F) ∈ A co B"
```
⟨*proof*⟩

24.1 Safety

lemma *(in Extend) project_unless [rule_format]:*
```
"[| G ∈ stable C;  project h C G ∈ A unless B |]
 ==> G ∈ (C ∩ extend_set h A) unless (extend_set h B)"
```
⟨*proof*⟩

lemma *(in Extend) Join_project_constrains:*
```
"(F⊔project h C G ∈ A co B)  =
    (extend h F⊔G ∈ (C ∩ extend_set h A) co (extend_set h B) &
     F ∈ A co B)"
```
⟨*proof*⟩

lemma *(in Extend) Join_project_stable:*
```
"extend h F⊔G ∈ stable C
 ==> (F⊔project h C G ∈ stable A)  =
      (extend h F⊔G ∈ stable (C ∩ extend_set h A) &
       F ∈ stable A)"
```
⟨*proof*⟩

lemma *(in Extend) project_constrains_I:*
```
"extend h F⊔G ∈ extend_set h A co extend_set h B
 ==> F⊔project h C G ∈ A co B"
```
⟨*proof*⟩

lemma *(in Extend) project_increasing_I:*
```
"extend h F⊔G ∈ increasing (func o f)
 ==> F⊔project h C G ∈ increasing func"
```
⟨*proof*⟩

lemma *(in Extend) Join_project_increasing:*
```
"(F⊔project h UNIV G ∈ increasing func)  =
 (extend h F⊔G ∈ increasing (func o f))"
```
⟨*proof*⟩

lemma *(in Extend) project_constrains_D:*
```
"F⊔project h UNIV G ∈ A co B
 ==> extend h F⊔G ∈ extend_set h A co extend_set h B"
```
⟨*proof*⟩

24.2 "projecting" and union/intersection (no converses)

lemma *projecting_Int:*
```
"[| projecting C h F XA' XA;  projecting C h F XB' XB |]
 ==> projecting C h F (XA' ∩ XB') (XA ∩ XB)"
```
⟨*proof*⟩

lemma *projecting_Un:*
```
"[| projecting C h F XA' XA;  projecting C h F XB' XB |]
 ==> projecting C h F (XA' ∪ XB') (XA ∪ XB)"
```
⟨*proof*⟩

lemma *projecting_INT:*
```
"[| !!i. i ∈ I ==> projecting C h F (X' i) (X i) |]
 ==> projecting C h F (⋂i ∈ I. X' i) (⋂i ∈ I. X i)"
```
⟨*proof*⟩

lemma *projecting_UN:*
```
"[| !!i. i ∈ I ==> projecting C h F (X' i) (X i) |]
 ==> projecting C h F (⋃i ∈ I. X' i) (⋃i ∈ I. X i)"
```
⟨*proof*⟩

lemma *projecting_weaken:*
```
"[| projecting C h F X' X;  U'<=X';  X ⊆ U |] ==> projecting C h F U'
U"
```
⟨*proof*⟩

lemma *projecting_weaken_L:*
```
"[| projecting C h F X' X;  U'<=X' |] ==> projecting C h F U' X"
```
⟨*proof*⟩

lemma *extending_Int:*
```
"[| extending C h F YA' YA;  extending C h F YB' YB |]
 ==> extending C h F (YA' ∩ YB') (YA ∩ YB)"
```
⟨*proof*⟩

lemma *extending_Un:*
```
"[| extending C h F YA' YA;  extending C h F YB' YB |]
 ==> extending C h F (YA' ∪ YB') (YA ∪ YB)"
```
⟨*proof*⟩

lemma *extending_INT:*
```
"[| !!i. i ∈ I ==> extending C h F (Y' i) (Y i) |]
 ==> extending C h F (⋂i ∈ I. Y' i) (⋂i ∈ I. Y i)"
```
⟨*proof*⟩

lemma *extending_UN:*
```
"[| !!i. i ∈ I ==> extending C h F (Y' i) (Y i) |]
 ==> extending C h F (⋃i ∈ I. Y' i) (⋃i ∈ I. Y i)"
```
⟨*proof*⟩

lemma *extending_weaken:*
```
"[| extending C h F Y' Y;  Y'<=V';  V ⊆ Y |] ==> extending C h F V' V"
```
⟨*proof*⟩

lemma *extending_weaken_L:*
```
"[| extending C h F Y' Y;  Y'<=V' |] ==> extending C h F V' Y"
```
⟨*proof*⟩

lemma *projecting_UNIV: "projecting C h F X' UNIV"*
⟨*proof*⟩

lemma *(***in** *Extend) projecting_constrains:*
```
"projecting C h F (extend_set h A co extend_set h B) (A co B)"
```
⟨*proof*⟩

lemma *(in Extend) projecting_stable:*
```
"projecting C h F (stable (extend_set h A)) (stable A)"
```
⟨*proof*⟩

lemma *(in Extend) projecting_increasing:*
```
"projecting C h F (increasing (func o f)) (increasing func)"
```
⟨*proof*⟩

lemma *(in Extend) extending_UNIV: "extending C h F UNIV Y"*
⟨*proof*⟩

lemma *(in Extend) extending_constrains:*
```
"extending (%G. UNIV) h F (extend_set h A co extend_set h B) (A co B)"
```
⟨*proof*⟩

lemma *(in Extend) extending_stable:*
```
"extending (%G. UNIV) h F (stable (extend_set h A)) (stable A)"
```
⟨*proof*⟩

lemma *(in Extend) extending_increasing:*
```
"extending (%G. UNIV) h F (increasing (func o f)) (increasing func)"
```
⟨*proof*⟩

24.3 Reachability and project

lemma *(in Extend) reachable_imp_reachable_project:*
```
"[| reachable (extend h F⊔G) ⊆ C;
    z ∈ reachable (extend h F⊔G) |]
 ==> f z ∈ reachable (F⊔project h C G)"
```
⟨*proof*⟩

lemma *(in Extend) project_Constrains_D:*
```
"F⊔project h (reachable (extend h F⊔G)) G ∈ A Co B
 ==> extend h F⊔G ∈ (extend_set h A) Co (extend_set h B)"
```
⟨*proof*⟩

lemma *(in Extend) project_Stable_D:*
```
"F⊔project h (reachable (extend h F⊔G)) G ∈ Stable A
 ==> extend h F⊔G ∈ Stable (extend_set h A)"
```
⟨*proof*⟩

lemma *(in Extend) project_Always_D:*
```
"F⊔project h (reachable (extend h F⊔G)) G ∈ Always A
 ==> extend h F⊔G ∈ Always (extend_set h A)"
```
⟨*proof*⟩

lemma *(in Extend) project_Increasing_D:*
```
"F⊔project h (reachable (extend h F⊔G)) G ∈ Increasing func
 ==> extend h F⊔G ∈ Increasing (func o f)"
```
⟨*proof*⟩

24.4 Converse results for weak safety: benefits of the argument C

lemma (**in** *Extend*) *reachable_project_imp_reachable*:
```
"[| C ⊆ reachable(extend h F⊔G);
    x ∈ reachable (F⊔project h C G) |]
 ==> ∃y. h(x,y) ∈ reachable (extend h F⊔G)"
```
⟨*proof*⟩

lemma (**in** *Extend*) *project_set_reachable_extend_eq*:
```
"project_set h (reachable (extend h F⊔G)) =
 reachable (F⊔project h (reachable (extend h F⊔G)) G)"
```
⟨*proof*⟩

lemma (**in** *Extend*) *reachable_extend_Join_subset*:
```
"reachable (extend h F⊔G) ⊆ C
 ==> reachable (extend h F⊔G) ⊆
     extend_set h (reachable (F⊔project h C G))"
```
⟨*proof*⟩

lemma (**in** *Extend*) *project_Constrains_I*:
```
"extend h F⊔G ∈ (extend_set h A) Co (extend_set h B)
 ==> F⊔project h (reachable (extend h F⊔G)) G ∈ A Co B"
```
⟨*proof*⟩

lemma (**in** *Extend*) *project_Stable_I*:
```
"extend h F⊔G ∈ Stable (extend_set h A)
 ==> F⊔project h (reachable (extend h F⊔G)) G ∈ Stable A"
```
⟨*proof*⟩

lemma (**in** *Extend*) *project_Always_I*:
```
"extend h F⊔G ∈ Always (extend_set h A)
 ==> F⊔project h (reachable (extend h F⊔G)) G ∈ Always A"
```
⟨*proof*⟩

lemma (**in** *Extend*) *project_Increasing_I*:
```
"extend h F⊔G ∈ Increasing (func o f)
 ==> F⊔project h (reachable (extend h F⊔G)) G ∈ Increasing func"
```
⟨*proof*⟩

lemma (**in** *Extend*) *project_Constrains*:
```
"(F⊔project h (reachable (extend h F⊔G)) G ∈ A Co B)  =
 (extend h F⊔G ∈ (extend_set h A) Co (extend_set h B))"
```
⟨*proof*⟩

lemma (**in** *Extend*) *project_Stable*:
```
"(F⊔project h (reachable (extend h F⊔G)) G ∈ Stable A)  =
 (extend h F⊔G ∈ Stable (extend_set h A))"
```
⟨*proof*⟩

lemma (**in** *Extend*) *project_Increasing*:
```
"(F⊔project h (reachable (extend h F⊔G)) G ∈ Increasing func)  =
 (extend h F⊔G ∈ Increasing (func o f))"
```

⟨*proof*⟩

24.5 A lot of redundant theorems: all are proved to facilitate reasoning about guarantees.

lemma *(in Extend) projecting_Constrains:*
```
"projecting (%G. reachable (extend h F⊔G)) h F
            (extend_set h A Co extend_set h B) (A Co B)"
```

⟨*proof*⟩

lemma *(in Extend) projecting_Stable:*
```
"projecting (%G. reachable (extend h F⊔G)) h F
            (Stable (extend_set h A)) (Stable A)"
```
⟨*proof*⟩

lemma *(in Extend) projecting_Always:*
```
"projecting (%G. reachable (extend h F⊔G)) h F
            (Always (extend_set h A)) (Always A)"
```
⟨*proof*⟩

lemma *(in Extend) projecting_Increasing:*
```
"projecting (%G. reachable (extend h F⊔G)) h F
            (Increasing (func o f)) (Increasing func)"
```
⟨*proof*⟩

lemma *(in Extend) extending_Constrains:*
```
"extending (%G. reachable (extend h F⊔G)) h F
            (extend_set h A Co extend_set h B) (A Co B)"
```
⟨*proof*⟩

lemma *(in Extend) extending_Stable:*
```
"extending (%G. reachable (extend h F⊔G)) h F
            (Stable (extend_set h A)) (Stable A)"
```
⟨*proof*⟩

lemma *(in Extend) extending_Always:*
```
"extending (%G. reachable (extend h F⊔G)) h F
            (Always (extend_set h A)) (Always A)"
```
⟨*proof*⟩

lemma *(in Extend) extending_Increasing:*
```
"extending (%G. reachable (extend h F⊔G)) h F
            (Increasing (func o f)) (Increasing func)"
```
⟨*proof*⟩

24.6 leadsETo in the precondition (??)

24.6.1 transient

lemma *(in Extend) transient_extend_set_imp_project_transient:*
```
"[| G ∈ transient (C ∩ extend_set h A);  G ∈ stable C |]
 ==> project h C G ∈ transient (project_set h C ∩ A)"
```
⟨*proof*⟩

lemma *(in Extend) project_extend_transient_D:*
```
"project h C (extend h F) ∈ transient (project_set h C ∩ D)
 ==> F ∈ transient (project_set h C ∩ D)"
```
⟨*proof*⟩

24.6.2 ensures – a primitive combining progress with safety

lemma *(in Extend) ensures_extend_set_imp_project_ensures:*
```
"[| extend h F ∈ stable C;  G ∈ stable C;
    extend h F⊔G ∈ A ensures B;  A-B = C ∩ extend_set h D |]
 ==> F⊔project h C G
       ∈ (project_set h C ∩ project_set h A) ensures (project_set h B)"
```
⟨*proof*⟩

Transferring a transient property upwards

lemma *(in Extend) project_transient_extend_set:*
```
"project h C G ∈ transient (project_set h C ∩ A - B)
 ==> G ∈ transient (C ∩ extend_set h A - extend_set h B)"
```
⟨*proof*⟩

lemma *(in Extend) project_unless2 [rule_format]:*
```
"[| G ∈ stable C;  project h C G ∈ (project_set h C ∩ A) unless B |]

 ==> G ∈ (C ∩ extend_set h A) unless (extend_set h B)"
```
⟨*proof*⟩

lemma *(in Extend) extend_unless:*
```
"[|extend h F ∈ stable C; F ∈ A unless B|]
 ==> extend h F ∈ C ∩ extend_set h A unless extend_set h B"
```
⟨*proof*⟩

lemma *(in Extend) Join_project_ensures [rule_format]:*
```
"[| extend h F⊔G ∈ stable C;
    F⊔project h C G ∈ A ensures B |]
 ==> extend h F⊔G ∈ (C ∩ extend_set h A) ensures (extend_set h B)"
```
⟨*proof*⟩

Lemma useful for both STRONG and WEAK progress, but the transient condition's very strong

lemma *(in Extend) PLD_lemma:*
```
"[| extend h F⊔G ∈ stable C;
    F⊔project h C G ∈ (project_set h C ∩ A) leadsTo B |]
 ==> extend h F⊔G ∈
      C ∩ extend_set h (project_set h C ∩ A) leadsTo (extend_set h B)"
```
⟨*proof*⟩

lemma *(in Extend) project_leadsTo_D_lemma:*
```
"[| extend h F⊔G ∈ stable C;
    F⊔project h C G ∈ (project_set h C ∩ A) leadsTo B |]
 ==> extend h F⊔G ∈ (C ∩ extend_set h A) leadsTo (extend_set h B)"
```

⟨proof⟩

lemma (**in** Extend) Join_project_LeadsTo:
```
     "[| C = (reachable (extend h F⊔G));
         F⊔project h C G ∈ A LeadsTo B |]
      ==> extend h F⊔G ∈ (extend_set h A) LeadsTo (extend_set h B)"
```
⟨proof⟩

24.7 Towards the theorem project_Ensures_D

lemma (**in** Extend) project_ensures_D_lemma:
```
     "[| G ∈ stable ((C ∩ extend_set h A) - (extend_set h B));
         F⊔project h C G ∈ (project_set h C ∩ A) ensures B;
         extend h F⊔G ∈ stable C |]
      ==> extend h F⊔G ∈ (C ∩ extend_set h A) ensures (extend_set h B)"
```

⟨proof⟩

lemma (**in** Extend) project_ensures_D:
```
     "[| F⊔project h UNIV G ∈ A ensures B;
         G ∈ stable (extend_set h A - extend_set h B) |]
      ==> extend h F⊔G ∈ (extend_set h A) ensures (extend_set h B)"
```
⟨proof⟩

lemma (**in** Extend) project_Ensures_D:
```
     "[| F⊔project h (reachable (extend h F⊔G)) G ∈ A Ensures B;
         G ∈ stable (reachable (extend h F⊔G) ∩ extend_set h A -
                     extend_set h B) |]
      ==> extend h F⊔G ∈ (extend_set h A) Ensures (extend_set h B)"
```
⟨proof⟩

24.8 Guarantees

lemma (**in** Extend) project_act_Restrict_subset_project_act:
```
     "project_act h (Restrict C act) ⊆ project_act h act"
```
⟨proof⟩

lemma (**in** Extend) subset_closed_ok_extend_imp_ok_project:
```
     "[| extend h F ok G; subset_closed (AllowedActs F) |]
      ==> F ok project h C G"
```
⟨proof⟩

lemma (**in** Extend) project_guarantees_raw:
assumes xguary: "F ∈ X guarantees Y"
    and closed: "subset_closed (AllowedActs F)"
    and project: "!!G. extend h F⊔G ∈ X'
                     ==> F⊔project h (C G) G ∈ X"
    and extend: "!!G. [| F⊔project h (C G) G ∈ Y |]
                     ==> extend h F⊔G ∈ Y'"

```
 shows "extend h F ∈ X' guarantees Y'"
```
⟨proof⟩

```
lemma (in Extend) project_guarantees:
     "[| F ∈ X guarantees Y;  subset_closed (AllowedActs F);
         projecting C h F X' X;  extending C h F Y' Y |]
      ==> extend h F ∈ X' guarantees Y'"
```
⟨proof⟩

24.9 guarantees corollaries

24.9.1 Some could be deleted: the required versions are easy to prove

```
lemma (in Extend) extend_guar_increasing:
     "[| F ∈ UNIV guarantees increasing func;
         subset_closed (AllowedActs F) |]
      ==> extend h F ∈ X' guarantees increasing (func o f)"
```
⟨proof⟩

```
lemma (in Extend) extend_guar_Increasing:
     "[| F ∈ UNIV guarantees Increasing func;
         subset_closed (AllowedActs F) |]
      ==> extend h F ∈ X' guarantees Increasing (func o f)"
```
⟨proof⟩

```
lemma (in Extend) extend_guar_Always:
     "[| F ∈ Always A guarantees Always B;
         subset_closed (AllowedActs F) |]
      ==> extend h F
             ∈ Always(extend_set h A) guarantees Always(extend_set h B)"
```
⟨proof⟩

24.9.2 Guarantees with a leadsTo postcondition

```
lemma (in Extend) project_leadsTo_D:
     "F⊔project h UNIV G ∈ A leadsTo B
      ==> extend h F⊔G ∈ (extend_set h A) leadsTo (extend_set h B)"
```
⟨proof⟩

```
lemma (in Extend) project_LeadsTo_D:
     "F⊔project h (reachable (extend h F⊔G)) G ∈ A LeadsTo B
       ==> extend h F⊔G ∈ (extend_set h A) LeadsTo (extend_set h B)"
```
⟨proof⟩

```
lemma (in Extend) extending_leadsTo:
     "extending (%G. UNIV) h F
                (extend_set h A leadsTo extend_set h B) (A leadsTo B)"
```
⟨proof⟩

```
lemma (in Extend) extending_LeadsTo:
     "extending (%G. reachable (extend h F⊔G)) h F
                (extend_set h A LeadsTo extend_set h B) (A LeadsTo B)"
```
⟨proof⟩

end

25 Progress Under Allowable Sets

theory *ELT* **imports** *Project* **begin**

inductive_set

```
  elt :: "['a set set, 'a program] => ('a set * 'a set) set"
  for CC :: "'a set set" and F :: "'a program"
 where

   Basis:  "[| F : A ensures B;  A-B : (insert {} CC) |] ==> (A,B) : elt CC
F"

 | Trans:  "[| (A,B) : elt CC F;  (B,C) : elt CC F |] ==> (A,C) : elt CC F"

 | Union:  "ALL A: S. (A,B) : elt CC F ==> (Union S, B) : elt CC F"
```

constdefs

```
  givenBy :: "['a => 'b] => 'a set set"
    "givenBy f == range (%B. f-` B)"


  leadsETo :: "['a set, 'a set set, 'a set] => 'a program set"
                                    ("(3_/ leadsTo[_]/ _)" [80,0,80] 80)
    "leadsETo A CC B == {F. (A,B) : elt CC F}"

  LeadsETo :: "['a set, 'a set set, 'a set] => 'a program set"
                                    ("(3_/ LeadsTo[_]/ _)" [80,0,80] 80)
    "LeadsETo A CC B ==
      {F. F : (reachable F Int A) leadsTo[(%C. reachable F Int C) ` CC] B}"
```

lemma *givenBy_id [simp]: "givenBy id = UNIV"*
⟨*proof*⟩

lemma *givenBy_eq_all: "(givenBy v) = {A. ALL x:A. ALL y. v x = v y --> y: A}"*
⟨*proof*⟩

lemma *givenByI: "(!!x y. [| x:A; v x = v y |] ==> y: A) ==> A: givenBy v"*
⟨*proof*⟩

lemma *givenByD: "[| A: givenBy v; x:A; v x = v y |] ==> y: A"*
⟨*proof*⟩

lemma *empty_mem_givenBy [iff]: "{} : givenBy v"*
⟨*proof*⟩

lemma *givenBy_imp_eq_Collect: "A: givenBy v ==> EX P. A = {s. P(v s)}"*
⟨*proof*⟩

lemma *Collect_mem_givenBy: "{s. P(v s)} : givenBy v"*
⟨*proof*⟩

lemma *givenBy_eq_Collect: "givenBy v = {A. EX P. A = {s. P(v s)}}"*
⟨*proof*⟩

lemma *preserves_givenBy_imp_stable:*
"[| F : preserves v; D : givenBy v |] ==> F : stable D"
⟨*proof*⟩

lemma *givenBy_o_subset: "givenBy (w o v) <= givenBy v"*
⟨*proof*⟩

lemma *givenBy_DiffI:*
"[| A : givenBy v; B : givenBy v |] ==> A-B : givenBy v"
⟨*proof*⟩

lemma *leadsETo_Basis [intro]:*
"[| F: A ensures B; A-B: insert {} CC |] ==> F : A leadsTo[CC] B"
⟨*proof*⟩

lemma *leadsETo_Trans:*
"[| F : A leadsTo[CC] B; F : B leadsTo[CC] C |] ==> F : A leadsTo[CC]
C"
⟨*proof*⟩

lemma *leadsETo_Un_duplicate:*
"F : A leadsTo[CC] (A' Un A') ==> F : A leadsTo[CC] A'"
⟨*proof*⟩

lemma *leadsETo_Un_duplicate2:*
"F : A leadsTo[CC] (A' Un C Un C) ==> F : A leadsTo[CC] (A' Un C)"
⟨*proof*⟩

lemma *leadsETo_Union:*
"(!!A. A : S ==> F : A leadsTo[CC] B) ==> F : (Union S) leadsTo[CC] B"
⟨*proof*⟩

lemma *leadsETo_UN:*
"(!!i. i : I ==> F : (A i) leadsTo[CC] B)
==> F : (UN i:I. A i) leadsTo[CC] B"

⟨*proof*⟩

lemma *leadsETo_induct:*
```
  "[| F : za leadsTo[CC] zb;
      !!A B. [| F : A ensures B;  A-B : insert {} CC |] ==> P A B;
      !!A B C. [| F : A leadsTo[CC] B; P A B; F : B leadsTo[CC] C; P B C |]

               ==> P A C;
      !!B S. ALL A:S. F : A leadsTo[CC] B & P A B ==> P (Union S) B
   |] ==> P za zb"
```
⟨*proof*⟩

lemma *leadsETo_mono: "CC' <= CC ==> (A leadsTo[CC'] B) <= (A leadsTo[CC] B)"*
⟨*proof*⟩

lemma *leadsETo_Trans_Un:*
```
     "[| F : A leadsTo[CC] B;  F : B leadsTo[DD] C |]
      ==> F : A leadsTo[CC Un DD] C"
```
⟨*proof*⟩

lemma *leadsETo_Union_Int:*
```
 "(!!A. A : S ==> F : (A Int C) leadsTo[CC] B)
  ==> F : (Union S Int C) leadsTo[CC] B"
```
⟨*proof*⟩

lemma *leadsETo_Un:*
```
     "[| F : A leadsTo[CC] C; F : B leadsTo[CC] C |]
      ==> F : (A Un B) leadsTo[CC] C"
```
⟨*proof*⟩

lemma *single_leadsETo_I:*
```
     "(!!x. x : A ==> F : {x} leadsTo[CC] B) ==> F : A leadsTo[CC] B"
```
⟨*proof*⟩

lemma *subset_imp_leadsETo: "A<=B ==> F : A leadsTo[CC] B"*
⟨*proof*⟩

lemmas *empty_leadsETo = empty_subsetI [THEN subset_imp_leadsETo, simp]*

lemma *leadsETo_weaken_R:*
```
     "[| F : A leadsTo[CC] A';  A'<=B' |] ==> F : A leadsTo[CC] B'"
```
⟨*proof*⟩

lemma *leadsETo_weaken_L [rule_format]:*
```
"[| F : A leadsTo[CC] A'; B<=A |] ==> F : B leadsTo[CC] A'"
```
⟨*proof*⟩

lemma *leadsETo_Un_distrib:*
```
"F : (A Un B) leadsTo[CC] C  =
 (F : A leadsTo[CC] C & F : B leadsTo[CC] C)"
```
⟨*proof*⟩

lemma *leadsETo_UN_distrib:*
```
"F : (UN i:I. A i) leadsTo[CC] B  =
 (ALL i : I. F : (A i) leadsTo[CC] B)"
```
⟨*proof*⟩

lemma *leadsETo_Union_distrib:*
```
"F : (Union S) leadsTo[CC] B  =  (ALL A : S. F : A leadsTo[CC] B)"
```
⟨*proof*⟩

lemma *leadsETo_weaken:*
```
"[| F : A leadsTo[CC'] A'; B<=A; A'<=B';  CC' <= CC |]
 ==> F : B leadsTo[CC] B'"
```
⟨*proof*⟩

lemma *leadsETo_givenBy:*
```
"[| F : A leadsTo[CC] A';  CC <= givenBy v |]
 ==> F : A leadsTo[givenBy v] A'"
```
⟨*proof*⟩

lemma *leadsETo_Diff:*
```
"[| F : (A-B) leadsTo[CC] C; F : B leadsTo[CC] C |]
 ==> F : A leadsTo[CC] C"
```
⟨*proof*⟩

lemma *leadsETo_Un_Un:*
```
"[| F : A leadsTo[CC] A';  F : B leadsTo[CC] B' |]
 ==> F : (A Un B) leadsTo[CC] (A' Un B')"
```
⟨*proof*⟩

lemma *leadsETo_cancel2:*
```
"[| F : A leadsTo[CC] (A' Un B); F : B leadsTo[CC] B' |]
 ==> F : A leadsTo[CC] (A' Un B')"
```
⟨*proof*⟩

lemma *leadsETo_cancel1:*
```
"[| F : A leadsTo[CC] (B Un A'); F : B leadsTo[CC] B' |]
==> F : A leadsTo[CC] (B' Un A')"
```

⟨proof⟩

lemma *leadsETo_cancel_Diff1:*
```
     "[| F : A leadsTo[CC] (B Un A'); F : (B-A') leadsTo[CC] B' |]
    ==> F : A leadsTo[CC] (B' Un A')"
```
⟨proof⟩

lemma *e_psp_stable:*
```
   "[| F : A leadsTo[CC] A';  F : stable B;  ALL C:CC. C Int B : CC |]
    ==> F : (A Int B) leadsTo[CC] (A' Int B)"
```
⟨proof⟩

lemma *e_psp_stable2:*
```
     "[| F : A leadsTo[CC] A'; F : stable B;  ALL C:CC. C Int B : CC |]
      ==> F : (B Int A) leadsTo[CC] (B Int A')"
```
⟨proof⟩

lemma *e_psp:*
```
     "[| F : A leadsTo[CC] A'; F : B co B';
         ALL C:CC. C Int B Int B' : CC |]
      ==> F : (A Int B') leadsTo[CC] ((A' Int B) Un (B' - B))"
```
⟨proof⟩

lemma *e_psp2:*
```
     "[| F : A leadsTo[CC] A'; F : B co B';
         ALL C:CC. C Int B Int B' : CC |]
      ==> F : (B' Int A) leadsTo[CC] ((B Int A') Un (B' - B))"
```
⟨proof⟩

lemma *gen_leadsETo_imp_Join_leadsETo:*
```
     "[| F: (A leadsTo[givenBy v] B);  G : preserves v;
         F⊔G : stable C |]
      ==> F⊔G : ((C Int A) leadsTo[(%D. C Int D) ` givenBy v] B)"
```
⟨proof⟩

lemma *leadsETo_subset_leadsTo:* `"(A leadsTo[CC] B) <= (A leadsTo B)"`
⟨proof⟩

lemma *leadsETo_UNIV_eq_leadsTo:* `"(A leadsTo[UNIV] B) = (A leadsTo B)"`
⟨proof⟩

lemma *LeadsETo_eq_leadsETo:*
```
    "A LeadsTo[CC] B =
       {F. F : (reachable F Int A) leadsTo[(%C. reachable F Int C) ` CC]
       (reachable F Int B)}"
```
⟨*proof*⟩

lemma *LeadsETo_Trans:*
```
    "[| F : A LeadsTo[CC] B;  F : B LeadsTo[CC] C |]
     ==> F : A LeadsTo[CC] C"
```
⟨*proof*⟩

lemma *LeadsETo_Union:*
```
    "(!!A. A : S ==> F : A LeadsTo[CC] B) ==> F : (Union S) LeadsTo[CC] B"
```
⟨*proof*⟩

lemma *LeadsETo_UN:*
```
    "(!!i. i : I ==> F : (A i) LeadsTo[CC] B)
     ==> F : (UN i:I. A i) LeadsTo[CC] B"
```
⟨*proof*⟩

lemma *LeadsETo_Un:*
```
    "[| F : A LeadsTo[CC] C; F : B LeadsTo[CC] C |]
     ==> F : (A Un B) LeadsTo[CC] C"
```
⟨*proof*⟩

lemma *single_LeadsETo_I:*
```
    "(!!s. s : A ==> F : {s} LeadsTo[CC] B) ==> F : A LeadsTo[CC] B"
```
⟨*proof*⟩

lemma *subset_imp_LeadsETo:*
```
    "A <= B ==> F : A LeadsTo[CC] B"
```
⟨*proof*⟩

lemmas *empty_LeadsETo = empty_subsetI [THEN subset_imp_LeadsETo, standard]*

lemma *LeadsETo_weaken_R [rule_format]:*
```
    "[| F : A LeadsTo[CC] A';  A' <= B' |] ==> F : A LeadsTo[CC] B'"
```
⟨*proof*⟩

lemma *LeadsETo_weaken_L [rule_format]:*
```
    "[| F : A LeadsTo[CC] A';  B <= A |] ==> F : B LeadsTo[CC] A'"
```
⟨*proof*⟩

lemma *LeadsETo_weaken:*
```
    "[| F : A LeadsTo[CC'] A';
        B <= A;  A' <= B';  CC' <= CC |]
     ==> F : B LeadsTo[CC] B'"
```
⟨*proof*⟩

lemma *LeadsETo_subset_LeadsTo: "(A LeadsTo[CC] B) <= (A LeadsTo B)"*
⟨*proof*⟩

lemma *reachable_ensures:*
"F : A ensures B ==> F : (reachable F Int A) ensures B"
⟨*proof*⟩

lemma *lel_lemma:*
"F : A leadsTo B ==> F : (reachable F Int A) leadsTo[Pow(reachable F)] B"
⟨*proof*⟩

lemma *LeadsETo_UNIV_eq_LeadsTo: "(A LeadsTo[UNIV] B) = (A LeadsTo B)"*
⟨*proof*⟩

lemma *(***in** *Extend) givenBy_o_eq_extend_set:*
"givenBy (v o f) = extend_set h ‘ (givenBy v)"
⟨*proof*⟩

lemma *(***in** *Extend) givenBy_eq_extend_set: "givenBy f = range (extend_set h)"*
⟨*proof*⟩

lemma *(***in** *Extend) extend_set_givenBy_I:*
"D : givenBy v ==> extend_set h D : givenBy (v o f)"
⟨*proof*⟩

lemma *(***in** *Extend) leadsETo_imp_extend_leadsETo:*
```
"F : A leadsTo[CC] B
 ==> extend h F : (extend_set h A) leadsTo[extend_set h ‘ CC]
                  (extend_set h B)"
```
⟨*proof*⟩

lemma *(***in** *Extend) Join_project_ensures_strong:*
```
"[| project h C G ~: transient (project_set h C Int (A-B)) |
      project_set h C Int (A - B) = {};
    extend h F⊔G : stable C;
    F⊔project h C G : (project_set h C Int A) ensures B |]
 ==> extend h F⊔G : (C Int extend_set h A) ensures (extend_set h B)"
```
⟨*proof*⟩

lemma *(***in** *Extend) pli_lemma:*
```
"[| extend h F⊔G : stable C;
```

```
         F⊔project h C G
           : project_set h C Int project_set h A leadsTo project_set h B |]

      ==> F⊔project h C G
            : project_set h C Int project_set h A leadsTo
              project_set h C Int project_set h B"
```
⟨*proof*⟩

lemma *(***in** *Extend) project_leadsETo_I_lemma:*

```
     "[| extend h F⊔G : stable C;
         extend h F⊔G :
           (C Int A) leadsTo[(%D. C Int D)`givenBy f]  B |]
  ==> F⊔project h C G
    : (project_set h C Int project_set h (C Int A)) leadsTo (project_set h
B)"
```
⟨*proof*⟩

lemma *(***in** *Extend) project_leadsETo_I:*

```
     "extend h F⊔G : (extend_set h A) leadsTo[givenBy f] (extend_set h B)
      ==> F⊔project h UNIV G : A leadsTo B"
```
⟨*proof*⟩

lemma *(***in** *Extend) project_LeadsETo_I:*

```
     "extend h F⊔G : (extend_set h A) LeadsTo[givenBy f] (extend_set h B)

      ==> F⊔project h (reachable (extend h F⊔G)) G
           : A LeadsTo B"
```
⟨*proof*⟩

lemma *(***in** *Extend) projecting_leadsTo:*

```
     "projecting (%G. UNIV) h F
                 (extend_set h A leadsTo[givenBy f] extend_set h B)
                 (A leadsTo B)"
```
⟨*proof*⟩

lemma *(***in** *Extend) projecting_LeadsTo:*

```
     "projecting (%G. reachable (extend h F⊔G)) h F
                 (extend_set h A LeadsTo[givenBy f] extend_set h B)
                 (A LeadsTo B)"
```
⟨*proof*⟩

end

26 Common Declarations for Chandy and Charpentier's Allocator

theory *AllocBase* **imports** *"../UNITY_Main"* **begin**

consts

```
  NbT      :: nat
  Nclients :: nat
```

axioms
```
  NbT_pos:  "0 < NbT"
```

consts `tokens     :: "nat list => nat"`
primrec
```
  "tokens [] = 0"
  "tokens (x#xs) = x + tokens xs"
```

consts
```
  bag_of :: "'a list => 'a multiset"
```

primrec
```
  "bag_of []     = {#}"
  "bag_of (x#xs) = {#x#} + bag_of xs"
```

lemma `setsum_fun_mono [rule_format]:`
```
     "!!f :: nat=>nat.
      (ALL i. i<n --> f i <= g i) -->
      setsum f (lessThan n) <= setsum g (lessThan n)"
```
⟨proof⟩

lemma `tokens_mono_prefix [rule_format]:`
```
     "ALL xs. xs <= ys --> tokens xs <= tokens ys"
```
⟨proof⟩

lemma `mono_tokens: "mono tokens"`
⟨proof⟩

lemma `bag_of_append [simp]: "bag_of (l@l') = bag_of l + bag_of l'"`
⟨proof⟩

lemma `mono_bag_of: "mono (bag_of :: 'a list => ('a::order) multiset)"`
⟨proof⟩

declare `setsum_cong [cong]`

lemma `bag_of_sublist_lemma:`
```
     "(∑ i∈ A Int lessThan k. {#if i<k then f i else g i#}) =
      (∑ i∈ A Int lessThan k. {#f i#})"
```
⟨proof⟩

lemma `bag_of_sublist:`
```
     "bag_of (sublist l A) =
      (∑ i∈ A Int lessThan (length l). {# l!i #})"
```
⟨proof⟩

lemma `bag_of_sublist_Un_Int:`

```
    "bag_of (sublist l (A Un B)) + bag_of (sublist l (A Int B)) =
     bag_of (sublist l A) + bag_of (sublist l B)"
⟨proof⟩

lemma bag_of_sublist_Un_disjoint:
    "A Int B = {}
     ==> bag_of (sublist l (A Un B)) =
         bag_of (sublist l A) + bag_of (sublist l B)"
⟨proof⟩

lemma bag_of_sublist_UN_disjoint [rule_format]:
    "[| finite I; ALL i:I. ALL j:I. i~=j --> A i Int A j = {} |]
     ==> bag_of (sublist l (UNION I A)) =
         (∑i∈I. bag_of (sublist l (A i)))"
⟨proof⟩

end



theory Alloc
imports AllocBase "../PPROD"
begin
```

26.1 State definitions. OUTPUT variables are locals

```
record clientState =
  giv :: "nat list"    — client's INPUT history: tokens GRANTED
  ask :: "nat list"    — client's OUTPUT history: tokens REQUESTED
  rel :: "nat list"    — client's OUTPUT history: tokens RELEASED

record 'a clientState_d =
  clientState +
  dummy :: 'a          — dummy field for new variables

constdefs
  — DUPLICATED FROM Client.thy, but with "tok" removed
  — Maybe want a special theory section to declare such maps
  non_dummy :: "'a clientState_d => clientState"
    "non_dummy s == (|giv = giv s, ask = ask s, rel = rel s|)"

  — Renaming map to put a Client into the standard form
  client_map :: "'a clientState_d => clientState*'a"
    "client_map == funPair non_dummy dummy"


record allocState =
  allocGiv :: "nat => nat list"    — OUTPUT history: source of "giv" for i
  allocAsk :: "nat => nat list"    — INPUT: allocator's copy of "ask" for i
  allocRel :: "nat => nat list"    — INPUT: allocator's copy of "rel" for i

record 'a allocState_d =
  allocState +
  dummy    :: 'a                   — dummy field for new variables
```

```
record 'a systemState =
  allocState +
  client :: "nat => clientState"  — states of all clients
  dummy  :: 'a                    — dummy field for new variables

constdefs

— * Resource allocation system specification *

  — spec (1)
  system_safety :: "'a systemState program set"
    "system_safety ==
     Always {s. (SUM i: lessThan Nclients. (tokens o giv o sub i o client)s)
     ≤ NbT + (SUM i: lessThan Nclients. (tokens o rel o sub i o client)s)}"

  — spec (2)
  system_progress :: "'a systemState program set"
    "system_progress == INT i : lessThan Nclients.
                        INT h.
                          {s. h ≤ (ask o sub i o client)s} LeadsTo
                          {s. h pfixLe (giv o sub i o client) s}"

  system_spec :: "'a systemState program set"
    "system_spec == system_safety Int system_progress"

— * Client specification (required) **

  — spec (3)
  client_increasing :: "'a clientState_d program set"
    "client_increasing ==
         UNIV guarantees  Increasing ask Int Increasing rel"

  — spec (4)
  client_bounded :: "'a clientState_d program set"
    "client_bounded ==
         UNIV guarantees  Always {s. ALL elt : set (ask s). elt ≤ NbT}"

  — spec (5)
  client_progress :: "'a clientState_d program set"
    "client_progress ==
         Increasing giv  guarantees
         (INT h. {s. h ≤ giv s & h pfixGe ask s}
                 LeadsTo {s. tokens h ≤ (tokens o rel) s})"

  — spec: preserves part
  client_preserves :: "'a clientState_d program set"
    "client_preserves == preserves giv Int preserves clientState_d.dummy"

  — environmental constraints
  client_allowed_acts :: "'a clientState_d program set"
    "client_allowed_acts ==
       {F. AllowedActs F =
```

```
          insert Id (UNION (preserves (funPair rel ask)) Acts)}"

client_spec :: "'a clientState_d program set"
  "client_spec == client_increasing Int client_bounded Int client_progress
                Int client_allowed_acts Int client_preserves"
```

— * Allocator specification (required) *

— spec (6)

```
alloc_increasing :: "'a allocState_d program set"
  "alloc_increasing ==
       UNIV  guarantees
       (INT i : lessThan Nclients. Increasing (sub i o allocGiv))"
```

— spec (7)

```
alloc_safety :: "'a allocState_d program set"
  "alloc_safety ==
       (INT i : lessThan Nclients. Increasing (sub i o allocRel))
       guarantees
       Always {s. (SUM i: lessThan Nclients. (tokens o sub i o allocGiv)s)
       ≤ NbT + (SUM i: lessThan Nclients. (tokens o sub i o allocRel)s)}"
```

— spec (8)

```
alloc_progress :: "'a allocState_d program set"
  "alloc_progress ==
       (INT i : lessThan Nclients. Increasing (sub i o allocAsk) Int
                                   Increasing (sub i o allocRel))
       Int
       Always {s. ALL i<Nclients.
                   ALL elt : set ((sub i o allocAsk) s). elt ≤ NbT}
       Int
       (INT i : lessThan Nclients.
        INT h. {s. h ≤ (sub i o allocGiv)s & h pfixGe (sub i o allocAsk)s}
               LeadsTo
               {s. tokens h ≤ (tokens o sub i o allocRel)s})
       guarantees
           (INT i : lessThan Nclients.
            INT h. {s. h ≤ (sub i o allocAsk) s}
                   LeadsTo
                   {s. h pfixLe (sub i o allocGiv) s})"
```

— spec: preserves part

```
alloc_preserves :: "'a allocState_d program set"
  "alloc_preserves == preserves allocRel Int preserves allocAsk Int
                      preserves allocState_d.dummy"
```

— environmental constraints

```
alloc_allowed_acts :: "'a allocState_d program set"
  "alloc_allowed_acts ==
     {F. AllowedActs F =
          insert Id (UNION (preserves allocGiv) Acts)}"
```

```
alloc_spec :: "'a allocState_d program set"
  "alloc_spec == alloc_increasing Int alloc_safety Int alloc_progress Int
                 alloc_allowed_acts Int alloc_preserves"
```

— * Network specification *

— spec (9.1)

```
network_ask :: "'a systemState program set"
  "network_ask == INT i : lessThan Nclients.
                      Increasing (ask o sub i o client)  guarantees
                      ((sub i o allocAsk) Fols (ask o sub i o client))"
```

— spec (9.2)

```
network_giv :: "'a systemState program set"
  "network_giv == INT i : lessThan Nclients.
                      Increasing (sub i o allocGiv)
                      guarantees
                      ((giv o sub i o client) Fols (sub i o allocGiv))"
```

— spec (9.3)

```
network_rel :: "'a systemState program set"
  "network_rel == INT i : lessThan Nclients.
                      Increasing (rel o sub i o client)
                      guarantees
                      ((sub i o allocRel) Fols (rel o sub i o client))"
```

— spec: preserves part

```
network_preserves :: "'a systemState program set"
  "network_preserves ==
     preserves allocGiv  Int
     (INT i : lessThan Nclients. preserves (rel o sub i o client)  Int
                                 preserves (ask o sub i o client))"
```

— environmental constraints

```
network_allowed_acts :: "'a systemState program set"
  "network_allowed_acts ==
     {F. AllowedActs F =
         insert Id
          (UNION (preserves allocRel Int
                  (INT i: lessThan Nclients. preserves(giv o sub i o client)))
                Acts)}"

network_spec :: "'a systemState program set"
  "network_spec == network_ask Int network_giv Int
                   network_rel Int network_allowed_acts Int
                   network_preserves"
```

— * State mappings *

```
sysOfAlloc :: "((nat => clientState) * 'a) allocState_d => 'a systemState"
  "sysOfAlloc == %s. let (cl,xtr) = allocState_d.dummy s
                     in (| allocGiv = allocGiv s,
                           allocAsk = allocAsk s,
                           allocRel = allocRel s,
```

```
                          client   = cl,
                          dummy    = xtr|)"


  sysOfClient :: "(nat => clientState) * 'a allocState_d => 'a systemState"
    "sysOfClient == %(cl,al). (| allocGiv = allocGiv al,
                              allocAsk = allocAsk al,
                              allocRel = allocRel al,
                              client   = cl,
                              systemState.dummy = allocState_d.dummy al|)"
```

consts

```
    Alloc   :: "'a allocState_d program"
    Client  :: "'a clientState_d program"
    Network :: "'a systemState program"
    System  :: "'a systemState program"
```

axioms

```
    Alloc:   "Alloc   : alloc_spec"
    Client:  "Client  : client_spec"
    Network: "Network : network_spec"
```

defs

```
    System_def:
      "System == rename sysOfAlloc Alloc Join Network Join
                 (rename sysOfClient
                  (plam x: lessThan Nclients. rename client_map Client))"
```

declare *image_Collect [simp del]*

declare *subset_preserves_o [THEN [2] rev_subsetD, intro]*
declare *subset_preserves_o [THEN [2] rev_subsetD, simp]*
declare *funPair_o_distrib [simp]*
declare *Always_INT_distrib [simp]*
declare *o_apply [simp del]*

lemmas *[simp] =*

```
  rename_image_constrains
  rename_image_stable
  rename_image_increasing
  rename_image_invariant
  rename_image_Constrains
  rename_image_Stable
  rename_image_Increasing
  rename_image_Always
  rename_image_leadsTo
  rename_image_LeadsTo
  rename_preserves
  rename_image_preserves
```

```
  lift_image_preserves
  bij_image_INT
  bij_is_inj [THEN image_Int]
  bij_image_Collect_eq
```

⟨ML⟩

lemmas `lessThanBspec = lessThan_iff [THEN iffD2, THEN [2] bspec]`

⟨ML⟩

lemma `inj_sysOfAlloc [iff]: "inj sysOfAlloc"`
  ⟨proof⟩

We need the inverse; also having it simplifies the proof of surjectivity

lemma `inv_sysOfAlloc_eq [simp]: "!!s. inv sysOfAlloc s =`
```
             (| allocGiv = allocGiv s,
                allocAsk = allocAsk s,
                allocRel = allocRel s,
                allocState_d.dummy = (client s, dummy s) |)"
```
  ⟨proof⟩

lemma `surj_sysOfAlloc [iff]: "surj sysOfAlloc"`
  ⟨proof⟩

lemma `bij_sysOfAlloc [iff]: "bij sysOfAlloc"`
  ⟨proof⟩

26.1.1 bijectivity of `sysOfClient`

lemma `inj_sysOfClient [iff]: "inj sysOfClient"`
  ⟨proof⟩

lemma `inv_sysOfClient_eq [simp]: "!!s. inv sysOfClient s =`
```
             (client s,
              (| allocGiv = allocGiv s,
                 allocAsk = allocAsk s,
                 allocRel = allocRel s,
                 allocState_d.dummy = systemState.dummy s|) )"
```
  ⟨proof⟩

lemma `surj_sysOfClient [iff]: "surj sysOfClient"`
  ⟨proof⟩

lemma `bij_sysOfClient [iff]: "bij sysOfClient"`
  ⟨proof⟩

26.1.2 bijectivity of `client_map`

lemma `inj_client_map [iff]: "inj client_map"`
  ⟨proof⟩

lemma `inv_client_map_eq [simp]: "!!s. inv client_map s =`
```
             (%(x,y).(|giv = giv x, ask = ask x, rel = rel x,
```

clientState_d.dummy = y|)) s"
⟨proof⟩

lemma *surj_client_map [iff]: "surj client_map"*
⟨proof⟩

lemma *bij_client_map [iff]: "bij client_map"*
⟨proof⟩

o-simprules for *client_map*

lemma *fst_o_client_map: "fst o client_map = non_dummy"*
⟨proof⟩

⟨ML⟩
declare *fst_o_client_map' [simp]*

lemma *snd_o_client_map: "snd o client_map = clientState_d.dummy"*
⟨proof⟩

⟨ML⟩
declare *snd_o_client_map' [simp]*

26.2 o-simprules for *sysOfAlloc* [MUST BE AUTOMATED]

lemma *client_o_sysOfAlloc: "client o sysOfAlloc = fst o allocState_d.dummy
"*
⟨proof⟩

⟨ML⟩
declare *client_o_sysOfAlloc' [simp]*

lemma *allocGiv_o_sysOfAlloc_eq: "allocGiv o sysOfAlloc = allocGiv"*
⟨proof⟩

⟨ML⟩
declare *allocGiv_o_sysOfAlloc_eq' [simp]*

lemma *allocAsk_o_sysOfAlloc_eq: "allocAsk o sysOfAlloc = allocAsk"*
⟨proof⟩

⟨ML⟩
declare *allocAsk_o_sysOfAlloc_eq' [simp]*

lemma *allocRel_o_sysOfAlloc_eq: "allocRel o sysOfAlloc = allocRel"*
⟨proof⟩

⟨ML⟩
declare *allocRel_o_sysOfAlloc_eq' [simp]*

26.3 o-simprules for *sysOfClient* [MUST BE AUTOMATED]

lemma *client_o_sysOfClient: "client o sysOfClient = fst"*
⟨proof⟩

⟨*ML*⟩
declare *client_o_sysOfClient' [simp]*

lemma *allocGiv_o_sysOfClient_eq: "allocGiv o sysOfClient = allocGiv o snd "*
⟨*proof*⟩

⟨*ML*⟩
declare *allocGiv_o_sysOfClient_eq' [simp]*

lemma *allocAsk_o_sysOfClient_eq: "allocAsk o sysOfClient = allocAsk o snd "*
⟨*proof*⟩

⟨*ML*⟩
declare *allocAsk_o_sysOfClient_eq' [simp]*

lemma *allocRel_o_sysOfClient_eq: "allocRel o sysOfClient = allocRel o snd "*
⟨*proof*⟩

⟨*ML*⟩
declare *allocRel_o_sysOfClient_eq' [simp]*

lemma *allocGiv_o_inv_sysOfAlloc_eq: "allocGiv o inv sysOfAlloc = allocGiv"*
⟨*proof*⟩

⟨*ML*⟩
declare *allocGiv_o_inv_sysOfAlloc_eq' [simp]*

lemma *allocAsk_o_inv_sysOfAlloc_eq: "allocAsk o inv sysOfAlloc = allocAsk"*
⟨*proof*⟩

⟨*ML*⟩
declare *allocAsk_o_inv_sysOfAlloc_eq' [simp]*

lemma *allocRel_o_inv_sysOfAlloc_eq: "allocRel o inv sysOfAlloc = allocRel"*
⟨*proof*⟩

⟨*ML*⟩
declare *allocRel_o_inv_sysOfAlloc_eq' [simp]*

lemma *rel_inv_client_map_drop_map: "(rel o inv client_map o drop_map i o inv sysOfClient) =
rel o sub i o client"*
⟨*proof*⟩

⟨*ML*⟩
declare *rel_inv_client_map_drop_map [simp]*

lemma *ask_inv_client_map_drop_map: "(ask o inv client_map o drop_map i o inv sysOfClient) =
ask o sub i o client"*
⟨*proof*⟩

⟨ML⟩

```
declare ask_inv_client_map_drop_map [simp]
```

```
declare finite_lessThan [iff]
```

Client : ¡unfolded specification¿

```
lemmas client_spec_simps =
  client_spec_def client_increasing_def client_bounded_def
  client_progress_def client_allowed_acts_def client_preserves_def
  guarantees_Int_right
```

⟨ML⟩

```
declare
  Client_Increasing_ask [iff]
  Client_Increasing_rel [iff]
  Client_Bounded [iff]
  Client_preserves_giv [iff]
  Client_preserves_dummy [iff]
```

Network : ¡unfolded specification¿

```
lemmas network_spec_simps =
  network_spec_def network_ask_def network_giv_def
  network_rel_def network_allowed_acts_def network_preserves_def
  ball_conj_distrib
```

⟨ML⟩

```
declare Network_preserves_allocGiv [iff]
```

```
declare
  Network_preserves_rel [simp]
  Network_preserves_ask [simp]
```

```
declare
  Network_preserves_rel [simplified o_def, simp]
  Network_preserves_ask [simplified o_def, simp]
```

Alloc : ¡unfolded specification¿

```
lemmas alloc_spec_simps =
  alloc_spec_def alloc_increasing_def alloc_safety_def
  alloc_progress_def alloc_allowed_acts_def alloc_preserves_def
```

⟨ML⟩

Strip off the INT in the guarantees postcondition

```
lemmas Alloc_Increasing = Alloc_Increasing_0 [normalized]
```

```
declare
  Alloc_preserves_allocRel [iff]
  Alloc_preserves_allocAsk [iff]
  Alloc_preserves_dummy [iff]
```

26.4 Components Lemmas [MUST BE AUTOMATED]

lemma *Network_component_System: "Network Join*
      ((rename sysOfClient
        (plam x: (lessThan Nclients). rename client_map Client)) Join
       rename sysOfAlloc Alloc)
      = System"
  ⟨*proof*⟩

lemma *Client_component_System: "(rename sysOfClient*
       (plam x: (lessThan Nclients). rename client_map Client)) Join
      (Network Join rename sysOfAlloc Alloc) = System"
  ⟨*proof*⟩

lemma *Alloc_component_System: "rename sysOfAlloc Alloc Join*
       ((rename sysOfClient (plam x: (lessThan Nclients). rename client_map Client)) Join
        Network) = System"
  ⟨*proof*⟩

declare
  Client_component_System [iff]
  Network_component_System [iff]
  Alloc_component_System [iff]

* These preservation laws should be generated automatically *

lemma *Client_Allowed [simp]: "Allowed Client = preserves rel Int preserves ask"*
  ⟨*proof*⟩

lemma *Network_Allowed [simp]: "Allowed Network =*
        preserves allocRel Int
        (INT i: lessThan Nclients. preserves(giv o sub i o client))"
  ⟨*proof*⟩

lemma *Alloc_Allowed [simp]: "Allowed Alloc = preserves allocGiv"*
  ⟨*proof*⟩

needed in `rename_client_map_tac`

lemma *OK_lift_rename_Client [simp]: "OK I (%i. lift i (rename client_map Client))"*
  ⟨*proof*⟩

lemma *fst_lift_map_eq_fst [simp]: "fst (lift_map i x) i = fst x"*
⟨*proof*⟩

lemma *fst_o_lift_map' [simp]:*
     "(f ∘ sub i ∘ fst ∘ lift_map i ∘ g) = f o fst o g"
⟨*proof*⟩

⟨*ML*⟩

Lifting `Client_Increasing` to *systemState*

lemma *rename_Client_Increasing: "i : I*
```
      ==> rename sysOfClient (plam x: I. rename client_map Client) :
            UNIV  guarantees
            Increasing (ask o sub i o client) Int
            Increasing (rel o sub i o client)"
```
⟨*proof*⟩

lemma *preserves_sub_fst_lift_map: "[| F : preserves w; i ~= j |]*
```
      ==> F : preserves (sub i o fst o lift_map j o funPair v w)"
```
⟨*proof*⟩

lemma *client_preserves_giv_oo_client_map: "[| i < Nclients; j < Nclients |]*
```
      ==> Client : preserves (giv o sub i o fst o lift_map j o client_map)"
```
⟨*proof*⟩

lemma *rename_sysOfClient_ok_Network:*
```
  "rename sysOfClient (plam x: lessThan Nclients. rename client_map Client)
    ok Network"
```
⟨*proof*⟩

lemma *rename_sysOfClient_ok_Alloc:*
```
  "rename sysOfClient (plam x: lessThan Nclients. rename client_map Client)
    ok rename sysOfAlloc Alloc"
```
⟨*proof*⟩

lemma *rename_sysOfAlloc_ok_Network: "rename sysOfAlloc Alloc ok Network"*
⟨*proof*⟩

declare
```
  rename_sysOfClient_ok_Network [iff]
  rename_sysOfClient_ok_Alloc [iff]
  rename_sysOfAlloc_ok_Network [iff]
```

The "ok" laws, re-oriented. But not sure this works: theorem *ok_commute* is needed below

declare
```
  rename_sysOfClient_ok_Network [THEN ok_sym, iff]
  rename_sysOfClient_ok_Alloc [THEN ok_sym, iff]
  rename_sysOfAlloc_ok_Network [THEN ok_sym]
```

lemma *System_Increasing: "i < Nclients*
```
      ==> System : Increasing (ask o sub i o client) Int
                   Increasing (rel o sub i o client)"
```
⟨*proof*⟩

lemmas *rename_guarantees_sysOfAlloc_I =*
```
  bij_sysOfAlloc [THEN rename_rename_guarantees_eq, THEN iffD2, standard]
```

lemmas *rename_Alloc_Increasing =*
```
  Alloc_Increasing
    [THEN rename_guarantees_sysOfAlloc_I,
```

```
    simplified surj_rename [THEN surj_range] o_def sub_apply
               rename_image_Increasing bij_sysOfAlloc
               allocGiv_o_inv_sysOfAlloc_eq']
```

lemma *System_Increasing_allocGiv:*
```
    "i < Nclients ==> System : Increasing (sub i o allocGiv)"
```
⟨*proof*⟩

⟨*ML*⟩

declare *System_Increasing' [intro!]*

Follows consequences. The "Always (INT ...) formulation expresses the general safety property and allows it to be combined using *Always_Int_rule* below.

lemma *System_Follows_rel:*
```
  "i < Nclients ==> System : ((sub i o allocRel) Fols (rel o sub i o client))"
```
⟨*proof*⟩

lemma *System_Follows_ask:*
```
  "i < Nclients ==> System : ((sub i o allocAsk) Fols (ask o sub i o client))"
```
⟨*proof*⟩

lemma *System_Follows_allocGiv:*
```
  "i < Nclients ==> System : (giv o sub i o client) Fols (sub i o allocGiv)"
```
⟨*proof*⟩

lemma *Always_giv_le_allocGiv:*
```
"System : Always (INT i: lessThan Nclients.
                       {s. (giv o sub i o client) s ≤ (sub i o allocGiv) s})"
```
⟨*proof*⟩

lemma *Always_allocAsk_le_ask:*
```
"System : Always (INT i: lessThan Nclients.
                       {s. (sub i o allocAsk) s ≤ (ask o sub i o client) s})"
```
⟨*proof*⟩

lemma *Always_allocRel_le_rel:*
```
"System : Always (INT i: lessThan Nclients.
                       {s. (sub i o allocRel) s ≤ (rel o sub i o client) s})"
```
⟨*proof*⟩

26.5 Proof of the safety property (1)

safety (1), step 1 is *System_Follows_rel*

safety (1), step 2

lemmas *System_Increasing_allocRel = System_Follows_rel [THEN Follows_Increasing1, standard]*

safety (1), step 3

lemma *System_sum_bounded:*

```
    "System : Always {s. (∑i ∈ lessThan Nclients. (tokens o sub i o allocGiv)
s)
            ≤ NbT + (∑i ∈ lessThan Nclients. (tokens o sub i o allocRel)
s)}"
  ⟨proof⟩
```

Follows reasoning

```
lemma Always_tokens_giv_le_allocGiv: "System : Always (INT i: lessThan Nclients.
                          {s. (tokens o giv o sub i o client) s
                           ≤ (tokens o sub i o allocGiv) s})"
  ⟨proof⟩
```

```
lemma Always_tokens_allocRel_le_rel: "System : Always (INT i: lessThan Nclients.
                          {s. (tokens o sub i o allocRel) s
                           ≤ (tokens o rel o sub i o client) s})"
  ⟨proof⟩
```

safety (1), step 4 (final result!)

```
theorem System_safety: "System : system_safety"
  ⟨proof⟩
```

26.6 Proof of the progress property (2)

progress (2), step 1 is `System_Follows_ask` and `System_Follows_rel`

progress (2), step 2; see also `System_Increasing_allocRel`

```
lemmas System_Increasing_allocAsk =  System_Follows_ask [THEN Follows_Increasing1,
standard]
```

progress (2), step 3: lifting `Client_Bounded` to systemState

```
lemma rename_Client_Bounded: "i : I
    ==> rename sysOfClient (plam x: I. rename client_map Client) :
          UNIV  guarantees
          Always {s. ALL elt : set ((ask o sub i o client) s). elt ≤ NbT}"
  ⟨proof⟩
```

```
lemma System_Bounded_ask: "i < Nclients
      ==> System : Always
                     {s. ALL elt : set ((ask o sub i o client) s). elt ≤ NbT}"
  ⟨proof⟩
```

```
lemma Collect_all_imp_eq: "{x. ALL y. P y --> Q x y} = (INT y: {y. P y}.
{x. Q x y})"
  ⟨proof⟩
```

progress (2), step 4

```
lemma System_Bounded_allocAsk: "System : Always {s. ALL i<Nclients.
                          ALL elt : set ((sub i o allocAsk) s). elt ≤ NbT}"
  ⟨proof⟩
```

progress (2), step 5 is `System_Increasing_allocGiv` progress (2), step 6

```
lemmas System_Increasing_giv =  System_Follows_allocGiv [THEN Follows_Increasing1,
standard]
```

```
lemma rename_Client_Progress: "i: I
   ==> rename sysOfClient (plam x: I. rename client_map Client)
        : Increasing (giv o sub i o client)
          guarantees
          (INT h. {s. h ≤ (giv o sub i o client) s &
                            h pfixGe (ask o sub i o client) s}
                  LeadsTo {s. tokens h ≤ (tokens o rel o sub i o client) s})"
  ⟨proof⟩
```

progress (2), step 7

```
lemma System_Client_Progress:
  "System : (INT i : (lessThan Nclients).
            INT h. {s. h ≤ (giv o sub i o client) s &
                       h pfixGe (ask o sub i o client) s}
                LeadsTo {s. tokens h ≤ (tokens o rel o sub i o client) s})"
  ⟨proof⟩
```

```
lemmas System_lemma1 =
  Always_LeadsToD [OF System_Follows_ask [THEN Follows_Bounded]
                      System_Follows_allocGiv [THEN Follows_LeadsTo]]
```

```
lemmas System_lemma2 =
  PSP_Stable [OF System_lemma1
              System_Follows_ask [THEN Follows_Increasing1, THEN IncreasingD]]
```

```
lemma System_lemma3: "i < Nclients
      ==> System : {s. h ≤ (sub i o allocGiv) s &
                       h pfixGe (sub i o allocAsk) s}
                   LeadsTo
                   {s. h ≤ (giv o sub i o client) s &
                       h pfixGe (ask o sub i o client) s}"
  ⟨proof⟩
```

progress (2), step 8: Client i's "release" action is visible system-wide

```
lemma System_Alloc_Client_Progress: "i < Nclients
      ==> System : {s. h ≤ (sub i o allocGiv) s &
                       h pfixGe (sub i o allocAsk) s}
                   LeadsTo {s. tokens h ≤ (tokens o sub i o allocRel) s}"
  ⟨proof⟩
```

Lifting `Alloc_Progress` up to the level of systemState

progress (2), step 9

```
lemma System_Alloc_Progress:
 "System : (INT i : (lessThan Nclients).
```

```
          INT h. {s. h ≤ (sub i o allocAsk) s}
                LeadsTo {s. h pfixLe (sub i o allocGiv) s})"
```
⟨*proof*⟩

progress (2), step 10 (final result!)

lemma `System_Progress: "System : system_progress"`
⟨*proof*⟩

theorem `System_correct: "System : system_spec"`
⟨*proof*⟩

Some obsolete lemmas

lemma `non_dummy_eq_o_funPair: "non_dummy = (% (g,a,r). (| giv = g, ask = a, rel = r |)) o`
```
                              (funPair giv (funPair ask rel))"
```
⟨*proof*⟩

lemma `preserves_non_dummy_eq: "(preserves non_dummy) =`
```
      (preserves rel Int preserves ask Int preserves giv)"
```
⟨*proof*⟩

Could go to Extend.ML

lemma `bij_fst_inv_inv_eq: "bij f ==> fst (inv (%(x, u). inv f x) z) = f z"`
⟨*proof*⟩

end

27 Implementation of a multiple-client allocator from a single-client allocator

theory `AllocImpl` **imports** `AllocBase Follows PPROD` **begin**

record `'b merge =`
```
  In   :: "nat => 'b list"
  Out  :: "'b list"
  iOut :: "nat list"
```

record `('a,'b) merge_d =`
```
  "'b merge" +
  dummy :: 'a
```

constdefs
```
  non_dummy :: "('a,'b) merge_d => 'b merge"
    "non_dummy s == (|In = In s, Out = Out s, iOut = iOut s|)"
```

record `'b distr =`

```
  In  :: "'b list"
  iIn :: "nat list"
  Out :: "nat => 'b list"
```

record ('a,'b) distr_d =

```
  "'b distr" +
  dummy :: 'a
```

record allocState =

```
  giv :: "nat list"
  ask :: "nat list"
  rel :: "nat list"
```

record 'a allocState_d =

```
  allocState +
  dummy    :: 'a
```

record 'a systemState =

```
  allocState +
  mergeRel :: "nat merge"
  mergeAsk :: "nat merge"
  distr    :: "nat distr"
  dummy    :: 'a
```

constdefs

```
  merge_increasing :: "('a,'b) merge_d program set"
    "merge_increasing ==
         UNIV guarantees (Increasing merge.Out) Int (Increasing merge.iOut)"


  merge_eqOut :: "('a,'b) merge_d program set"
    "merge_eqOut ==
         UNIV guarantees
         Always {s. length (merge.Out s) = length (merge.iOut s)}"


  merge_bounded :: "('a,'b) merge_d program set"
    "merge_bounded ==
         UNIV guarantees
         Always {s. ∀elt ∈ set (merge.iOut s). elt < Nclients}"


  merge_follows :: "('a,'b) merge_d program set"
    "merge_follows ==
         (⋂i ∈ lessThan Nclients. Increasing (sub i o merge.In))
         guarantees
         (⋂i ∈ lessThan Nclients.
          (%s. sublist (merge.Out s)
                      {k. k < size(merge.iOut s) & merge.iOut s! k = i})
```

```
        Fols (sub i o merge.In))"


merge_preserves :: "('a,'b) merge_d program set"
  "merge_preserves == preserves merge.In Int preserves merge_d.dummy"


merge_allowed_acts :: "('a,'b) merge_d program set"
  "merge_allowed_acts ==
     {F. AllowedActs F =
          insert Id (UNION (preserves (funPair merge.Out merge.iOut)) Acts)}"

merge_spec :: "('a,'b) merge_d program set"
  "merge_spec == merge_increasing Int merge_eqOut Int merge_bounded Int
                 merge_follows Int merge_allowed_acts Int merge_preserves"




distr_follows :: "('a,'b) distr_d program set"
  "distr_follows ==
       Increasing distr.In Int Increasing distr.iIn Int
       Always {s. ∀elt ∈ set (distr.iIn s). elt < Nclients}
       guarantees
       (⋂i ∈ lessThan Nclients.
        (sub i o distr.Out) Fols
        (%s. sublist (distr.In s)
                     {k. k < size(distr.iIn s) & distr.iIn s ! k = i}))"

distr_allowed_acts :: "('a,'b) distr_d program set"
  "distr_allowed_acts ==
     {D. AllowedActs D = insert Id (UNION (preserves distr.Out) Acts)}"

distr_spec :: "('a,'b) distr_d program set"
  "distr_spec == distr_follows Int distr_allowed_acts"




alloc_increasing :: "'a allocState_d program set"
  "alloc_increasing == UNIV  guarantees  Increasing giv"


alloc_safety :: "'a allocState_d program set"
  "alloc_safety ==
       Increasing rel
       guarantees  Always {s. tokens (giv s) ≤ NbT + tokens (rel s)}"


alloc_progress :: "'a allocState_d program set"
  "alloc_progress ==
       Increasing ask Int Increasing rel Int
       Always {s. ∀elt ∈ set (ask s). elt ≤ NbT}
       Int
```

```
    (⋂h. {s. h ≤ giv s & h pfixGe (ask s)}
            LeadsTo
            {s. tokens h ≤ tokens (rel s)})
    guarantees  (⋂h. {s. h ≤ ask s} LeadsTo {s. h pfixLe giv s})"


alloc_preserves :: "'a allocState_d program set"
  "alloc_preserves == preserves rel Int
                      preserves ask Int
                      preserves allocState_d.dummy"



alloc_allowed_acts :: "'a allocState_d program set"
  "alloc_allowed_acts ==
     {F. AllowedActs F = insert Id (UNION (preserves giv) Acts)}"

alloc_spec :: "'a allocState_d program set"
  "alloc_spec == alloc_increasing Int alloc_safety Int alloc_progress Int
                 alloc_allowed_acts Int alloc_preserves"
```

locale *Merge* =
  fixes *M :: "('a,'b::order) merge_d program"*
  assumes
    Merge_spec: "M ∈ merge_spec"

locale *Distrib* =
  fixes *D :: "('a,'b::order) distr_d program"*
  assumes
    Distrib_spec: "D ∈ distr_spec"

declare *subset_preserves_o [THEN subsetD, intro]*
declare *funPair_o_distrib [simp]*
declare *Always_INT_distrib [simp]*
declare *o_apply [simp del]*

27.1 Theorems for Merge

lemma *(***in** *Merge) Merge_Allowed:*
    "Allowed M = (preserves merge.Out) Int (preserves merge.iOut)"
⟨*proof*⟩

lemma *(***in** *Merge) M_ok_iff [iff]:*
    "M ok G = (G ∈ preserves merge.Out & G ∈ preserves merge.iOut &
              M ∈ Allowed G)"
⟨*proof*⟩

lemma *(***in** *Merge) Merge_Always_Out_eq_iOut:*
    "[| G ∈ preserves merge.Out; G ∈ preserves merge.iOut; M ∈ Allowed G

```
|]
      ==> M Join G ∈ Always {s. length (merge.Out s) = length (merge.iOut
s)}"
```
⟨proof⟩

lemma (**in** Merge) Merge_Bounded:
```
     "[| G ∈ preserves merge.iOut; G ∈ preserves merge.Out; M ∈ Allowed G
|]
      ==> M Join G ∈ Always {s. ∀elt ∈ set (merge.iOut s). elt < Nclients}"
```
⟨proof⟩

lemma (**in** Merge) Merge_Bag_Follows_lemma:
```
     "[| G ∈ preserves merge.iOut; G ∈ preserves merge.Out; M ∈ Allowed G
|]
  ==> M Join G ∈ Always
          {s. (∑i ∈ lessThan Nclients. bag_of (sublist (merge.Out s)
                                  {k. k < length (iOut s) & iOut s ! k = i}))
=
              (bag_of o merge.Out) s}"
```
⟨proof⟩

lemma (**in** Merge) Merge_Bag_Follows:
```
     "M ∈ (⋂i ∈ lessThan Nclients. Increasing (sub i o merge.In))
          guarantees
             (bag_of o merge.Out) Fols
             (%s. ∑i ∈ lessThan Nclients. (bag_of o sub i o merge.In) s)"
```
⟨proof⟩

27.2 Theorems for Distributor

lemma (**in** Distrib) Distr_Increasing_Out:
```
     "D ∈ Increasing distr.In Int Increasing distr.iIn Int
          Always {s. ∀elt ∈ set (distr.iIn s). elt < Nclients}
          guarantees
          (⋂i ∈ lessThan Nclients. Increasing (sub i o distr.Out))"
```
⟨proof⟩

lemma (**in** Distrib) Distr_Bag_Follows_lemma:
```
     "[| G ∈ preserves distr.Out;
         D Join G ∈ Always {s. ∀elt ∈ set (distr.iIn s). elt < Nclients}
|]
  ==> D Join G ∈ Always
          {s. (∑i ∈ lessThan Nclients. bag_of (sublist (distr.In s)
                                  {k. k < length (iIn s) & iIn s ! k = i}))
=
              bag_of (sublist (distr.In s) (lessThan (length (iIn s))))}"
```
⟨proof⟩

lemma (**in** Distrib) D_ok_iff [iff]:
```
     "D ok G = (G ∈ preserves distr.Out & D ∈ Allowed G)"
```
⟨proof⟩

lemma (**in** Distrib) Distr_Bag_Follows:
```
 "D ∈ Increasing distr.In Int Increasing distr.iIn Int
```

```
    Always {s. ∀elt ∈ set (distr.iIn s). elt < Nclients}
    guarantees
     (⋂i ∈ lessThan Nclients.
      (%s. ∑i ∈ lessThan Nclients. (bag_of o sub i o distr.Out) s)
      Fols
      (%s. bag_of (sublist (distr.In s) (lessThan (length(distr.iIn s))))))"
```

⟨proof⟩

27.3 Theorems for Allocator

lemma *alloc_refinement_lemma:*

```
     "!!f::nat=>nat. (⋂i ∈ lessThan n. {s. f i ≤ g i s})
      ⊆ {s. (SUM x: lessThan n. f x) ≤ (SUM x: lessThan n. g x s)}"
```

⟨proof⟩

lemma *alloc_refinement:*

```
"(⋂i ∈ lessThan Nclients. Increasing (sub i o allocAsk) Int
                          Increasing (sub i o allocRel))
  Int
  Always {s. ∀i. i<Nclients -->
             (∀elt ∈ set ((sub i o allocAsk) s). elt ≤ NbT)}
  Int
  (⋂i ∈ lessThan Nclients.
   ⋂h. {s. h ≤ (sub i o allocGiv)s & h pfixGe (sub i o allocAsk)s}
       LeadsTo {s. tokens h ≤ (tokens o sub i o allocRel)s})
  ⊆
 (⋂i ∈ lessThan Nclients. Increasing (sub i o allocAsk) Int
                          Increasing (sub i o allocRel))
  Int
  Always {s. ∀i. i<Nclients -->
             (∀elt ∈ set ((sub i o allocAsk) s). elt ≤ NbT)}
  Int
  (⋂hf. (⋂i ∈ lessThan Nclients.
        {s. hf i ≤ (sub i o allocGiv)s & hf i pfixGe (sub i o allocAsk)s})
  LeadsTo {s. (∑i ∈ lessThan Nclients. tokens (hf i)) ≤
             (∑i ∈ lessThan Nclients. (tokens o sub i o allocRel)s)})"
```

⟨proof⟩

end

28 Distributed Resource Management System: the Client

theory *Client* **imports** *Rename AllocBase* **begin**

types

tokbag = nat — tokbags could be multisets...or any ordered type?

record *state =*

giv :: "tokbag list" — input history: tokens granted
ask :: "tokbag list" — output history: tokens requested
rel :: "tokbag list" — output history: tokens released

```
  tok :: tokbag           — current token request

record 'a state_d =
  state +
  dummy :: 'a             — new variables
```

constdefs

```
  rel_act :: "('a state_d * 'a state_d) set"
    "rel_act == {(s,s').
                 ∃nrel. nrel = size (rel s) &
                        s' = s (| rel := rel s @ [giv s!nrel] |) &
                        nrel < size (giv s) &
                        ask s!nrel ≤ giv s!nrel}"

  tok_act :: "('a state_d * 'a state_d) set"
     "tok_act == {(s,s'). s'=s | s' = s (|tok := Suc (tok s mod NbT) |)}"

  ask_act :: "('a state_d * 'a state_d) set"
    "ask_act == {(s,s'). s'=s |
                        (s' = s (|ask := ask s @ [tok s]|))}"

  Client :: "'a state_d program"
    "Client ==
       mk_total_program
            ({s. tok s ∈ atMost NbT &
                 giv s = [] & ask s = [] & rel s = []},
             {rel_act, tok_act, ask_act},
             ⋃G ∈ preserves rel Int preserves ask Int preserves tok.
                   Acts G)"

  non_dummy :: "'a state_d => state"
    "non_dummy s == (|giv = giv s, ask = ask s, rel = rel s, tok = tok s|)"

  client_map :: "'a state_d => state*'a"
    "client_map == funPair non_dummy dummy"
```

declare *Client_def [THEN def_prg_Init, simp]*
declare *Client_def [THEN def_prg_AllowedActs, simp]*
declare *rel_act_def [THEN def_act_simp, simp]*
declare *tok_act_def [THEN def_act_simp, simp]*
declare *ask_act_def [THEN def_act_simp, simp]*

```
lemma Client_ok_iff [iff]:
     "(Client ok G) =
      (G ∈ preserves rel & G ∈ preserves ask & G ∈ preserves tok &
       Client ∈ Allowed G)"
⟨proof⟩
```

Safety property 1: ask, rel are increasing

```
lemma increasing_ask_rel:
     "Client ∈ UNIV guarantees Increasing ask Int Increasing rel"
⟨proof⟩

declare nth_append [simp] append_one_prefix [simp]
```

Safety property 2: the client never requests too many tokens. With no Substitution Axiom, we must prove the two invariants simultaneously.

```
lemma ask_bounded_lemma:
     "Client ok G
      ==> Client Join G ∈
              Always ({s. tok s ≤ NbT}  Int
                      {s. ∀elt ∈ set (ask s). elt ≤ NbT})"
⟨proof⟩
```

export version, with no mention of tok in the postcondition, but unfortunately tok must be declared local.

```
lemma ask_bounded:
     "Client ∈ UNIV guarantees Always {s. ∀elt ∈ set (ask s). elt ≤ NbT}"
⟨proof⟩
```

** Towards proving the liveness property **

```
lemma stable_rel_le_giv: "Client ∈ stable {s. rel s ≤ giv s}"
⟨proof⟩

lemma Join_Stable_rel_le_giv:
     "[| Client Join G ∈ Increasing giv;  G ∈ preserves rel |]
      ==> Client Join G ∈ Stable {s. rel s ≤ giv s}"
⟨proof⟩

lemma Join_Always_rel_le_giv:
     "[| Client Join G ∈ Increasing giv;  G ∈ preserves rel |]
      ==> Client Join G ∈ Always {s. rel s ≤ giv s}"
⟨proof⟩

lemma transient_lemma:
     "Client ∈ transient {s. rel s = k & k<h & h ≤ giv s & h pfixGe ask s}"
⟨proof⟩


lemma induct_lemma:
     "[| Client Join G ∈ Increasing giv;  Client ok G |]
  ==> Client Join G ∈ {s. rel s = k & k<h & h ≤ giv s & h pfixGe ask s}
                  LeadsTo {s. k < rel s & rel s ≤ giv s &
                               h ≤ giv s & h pfixGe ask s}"
```

⟨*proof*⟩

lemma *rel_progress_lemma:*
```
     "[| Client Join G ∈ Increasing giv;  Client ok G |]
  ==> Client Join G ∈ {s. rel s < h & h ≤ giv s & h pfixGe ask s}
                      LeadsTo {s. h ≤ rel s}"
```
⟨*proof*⟩

lemma *client_progress_lemma:*
```
     "[| Client Join G ∈ Increasing giv;  Client ok G |]
      ==> Client Join G ∈ {s. h ≤ giv s & h pfixGe ask s}
                          LeadsTo {s. h ≤ rel s}"
```
⟨*proof*⟩

Progress property: all tokens that are given will be released

lemma *client_progress:*
```
     "Client ∈
       Increasing giv  guarantees
       (INT h. {s. h ≤ giv s & h pfixGe ask s} LeadsTo {s. h ≤ rel s})"
```
⟨*proof*⟩

This shows that the Client won't alter other variables in any state that it is combined with

lemma *client_preserves_dummy: "Client ∈ preserves dummy"*
⟨*proof*⟩

* Obsolete lemmas from first version of the Client *

lemma *stable_size_rel_le_giv:*
```
     "Client ∈ stable {s. size (rel s) ≤ size (giv s)}"
```
⟨*proof*⟩

clients return the right number of tokens

lemma *ok_guar_rel_prefix_giv:*
```
     "Client ∈ Increasing giv  guarantees  Always {s. rel s ≤ giv s}"
```
⟨*proof*⟩

end