

The UNITY Formalism

Sidi Ehmety and Lawrence C. Paulson

June 21, 2010

Contents

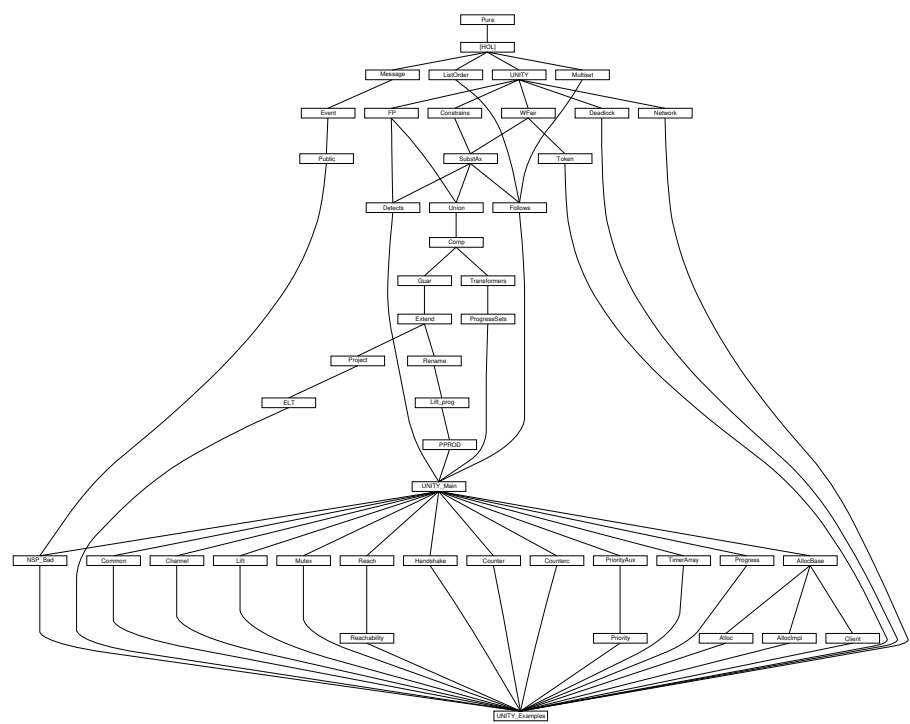
1	The Basic UNITY Theory	7
1.0.1	The abstract type of programs	7
1.0.2	Inspectors for type "program"	8
1.0.3	Equality for UNITY programs	8
1.0.4	co	9
1.0.5	Union	9
1.0.6	Intersection	10
1.0.7	unless	10
1.0.8	stable	10
1.0.9	Union	10
1.0.10	Intersection	11
1.0.11	invariant	11
1.0.12	increasing	11
1.0.13	Theoretical Results from Section 6	12
1.0.14	Ad-hoc set-theory rules	12
1.1	Partial versus Total Transitions	12
1.1.1	Basic properties	13
1.2	Rules for Lazy Definition Expansion	14
1.2.1	Inspectors for type "program"	14
2	Fixed Point of a Program	14
3	Progress	15
3.1	transient	16
3.2	ensures	17
3.3	leadsTo	18
3.4	PSP: Progress-Safety-Progress	21
3.5	Proving the induction rules	21
3.6	wlt	22
3.7	Completion: Binary and General Finite versions	23
4	Weak Safety	24
4.1	traces and reachable	24
4.2	Co	25
4.3	Stable	26
4.4	Increasing	27
4.5	The Elimination Theorem	27

4.6	Specialized laws for handling Always	28
4.7	"Co" rules involving Always	28
4.8	Totalize	29
5	Weak Progress	30
5.1	Specialized laws for handling invariants	30
5.2	Introduction rules: Basis, Trans, Union	30
5.3	Derived rules	31
5.4	PSP: Progress-Safety-Progress	33
5.5	Induction rules	34
5.6	Completion: Binary and General Finite versions	35
6	The Detects Relation	36
7	Unions of Programs	37
7.1	SKIP	38
7.2	SKIP and safety properties	38
7.3	Join	38
7.4	JN	38
7.5	Algebraic laws	39
7.6	Laws Governing \sqcup	39
7.7	Safety: co, stable, FP	40
7.8	Progress: transient, ensures	41
7.9	the ok and OK relations	42
7.10	Allowed	43
7.11	<i>safety_prop</i> , for reasoning about given instances of "ok"	43
8	Composition: Basic Primitives	44
8.1	The component relation	45
8.2	The preserves property	46
9	Guarantees Specifications	48
9.1	Existential Properties	49
9.2	Universal Properties	50
9.3	Guarantees	50
9.4	Distributive Laws. Re-Orient to Perform Miniscoping	51
9.5	Guarantees: Additional Laws (by lcp)	52
9.6	Guarantees Laws for Breaking Down the Program (by lcp)	52
10	Extending State Sets	55
10.1	Restrict	56
10.2	Trivial properties of f, g, h	57
10.3	<i>extend_set</i> : basic properties	57
10.4	<i>project_set</i> : basic properties	58
10.5	More laws	58
10.6	<i>extend_act</i>	59
10.7	extend	60
10.8	Safety: co, stable	61
10.9	Weak safety primitives: Co, Stable	62
10.10	Progress: transient, ensures	63

10.11 Proving the converse takes some doing!	63
10.12 preserves	64
10.13 Guarantees	65
11 Renaming of State Sets	65
11.1 inverse properties	66
11.2 the lattice operations	67
11.3 Strong Safety: co, stable	67
11.4 Weak Safety: Co, Stable	68
11.5 Progress: transient, ensures	68
11.6 "image" versions of the rules, for lifting "guarantees" properties	69
12 Replication of Components	70
12.1 Injectiveness proof	71
12.2 Surjectiveness proof	71
12.3 The Operator <i>lift_set</i>	72
12.4 The Lattice Operations	73
12.5 Safety: constrains, stable, invariant	73
12.6 Progress: transient, ensures	73
12.7 Lemmas to Handle Function Composition (o) More Consistently	75
12.8 More lemmas about extend and project	75
12.9 OK and "lift"	76
13 The Prefix Ordering on Lists	79
13.1 preliminary lemmas	79
13.2 genPrefix is a partial order	80
13.3 recursion equations	81
13.4 The type of lists is partially ordered	82
13.5 pfixLe, pfixGe: properties inherited from the translations	84
14 (Finite) multisets	84
14.1 The type of multisets	85
14.2 Representing multisets	85
14.3 Basic operations	86
14.3.1 Union	86
14.3.2 Difference	86
14.3.3 Equality of multisets	87
14.3.4 Pointwise ordering induced by count	88
14.3.5 Intersection	90
14.3.6 Comprehension (filter)	91
14.3.7 Set of elements	91
14.3.8 Size	91
14.4 Induction and case splits	92
14.4.1 Strong induction and subset induction for multisets	93
14.5 Alternative representations	94
14.5.1 Lists	94
14.5.2 Association lists – including rudimentary code generation	95
14.6 The multiset order	97
14.6.1 Well-foundedness	97
14.6.2 Closure-free presentation	97

14.6.3	Partial-order properties	98
14.6.4	Monotonicity of multiset union	98
14.7	The fold combinator	99
14.8	Image	100
14.9	Termination proofs with multiset orders	101
14.10	Legacy theorem bindings	102
15	The Follows Relation of Charpentier and Sivilotte	103
15.1	Destruction rules	104
15.2	Union properties (with the subset ordering)	104
15.3	Multiset union properties (with the multiset ordering)	105
16	Predicate Transformers	106
16.1	Defining the Predicate Transformers <i>wp</i> , <i>awp</i> and <i>wens</i>	106
16.2	Defining the Weakest Ensures Set	108
16.3	Properties Involving Program Union	109
16.4	The Set <i>wens_set</i> $F B$ for a Single-Assignment Program	110
17	Progress Sets	112
17.1	Complete Lattices and the Operator <i>c1</i>	112
17.2	Progress Sets and the Main Lemma	114
17.3	The Progress Set Union Theorem	115
17.4	Some Progress Sets	116
17.4.1	Lattices and Relations	116
17.4.2	Decoupling Theorems	117
17.5	Composition Theorems Based on Monotonicity and Commutativity	117
17.5.1	Commutativity of <i>c1</i> L and assignment.	117
17.5.2	Commutativity of Functions and Relation	118
17.6	Monotonicity	119
18	Comprehensive UNITY Theory	119
19	The Token Ring	123
19.1	Definitions	123
19.2	Progress under Weak Fairness	124
19.3	Progress	130
20	Analyzing the Needham-Schroeder Public-Key Protocol in UNITY	144
20.1	Inductive Proofs about <i>ns_public</i>	145
20.2	Authenticity properties obtained from NS2	146
20.3	Authenticity properties obtained from NS2	147
21	A Family of Similar Counters: Original Version	149
22	A Family of Similar Counters: Version with Compatibility	151
23	The priority system	155
23.1	Component correctness proofs	156
23.2	System properties	157
23.3	The main result: above set decreases	158

24 Progress Set Examples	160
24.1 The Composition of Two Single-Assignment Programs	160
24.1.1 Calculating $wens_set\ FF\ \{k..\}$	160
24.1.2 Proving $FF \in UNIV \mapsto \{k..\}$	160
25 Common Declarations for Chandy and Charpentier's Allocator	161
25.1 State definitions. OUTPUT variables are locals	163
25.1.1 bijectivity of $sysOfClient$	168
25.1.2 bijectivity of $client_map$	168
25.2 o-simprules for $sysOfAlloc$ [MUST BE AUTOMATED]	169
25.3 o-simprules for $sysOfClient$ [MUST BE AUTOMATED]	169
25.4 Components Lemmas [MUST BE AUTOMATED]	172
25.5 Proof of the safety property (1)	174
25.6 Proof of the progress property (2)	175
26 Implementation of a multiple-client allocator from a single-client allocator	177
26.1 Theorems for Merge	181
26.2 Theorems for Distributor	181
26.3 Theorems for Allocator	182
27 Distributed Resource Management System: the Client	183
28 Projections of State Sets	186
28.1 Safety	186
28.2 "projecting" and union/intersection (no converses)	187
28.3 Reachability and project	189
28.4 Converse results for weak safety: benefits of the argument C . . .	189
28.5 A lot of redundant theorems: all are proved to facilitate reasoning about guarantees.	190
28.6 leadsETo in the precondition (??)	191
28.6.1 transient	191
28.6.2 ensures – a primitive combining progress with safety . . .	191
28.7 Towards the theorem $project_Ensures_D$	192
28.8 Guarantees	193
28.9 guarantees corollaries	193
28.9.1 Some could be deleted: the required versions are easy to prove	193
28.9.2 Guarantees with a leadsTo postcondition	194
29 Progress Under Allowable Sets	194
30 Various examples for UNITY	202



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⟩

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

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

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

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

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

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

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

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

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

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

definition increasing :: "[ 'a => 'b::order] => 'a program set" where
  — 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*: " $C \text{ co } (A \cap B) = (C \text{ co } A) \cap (C \text{ co } B)$ "
 $\langle \text{proof} \rangle$

lemma *constrains_INT_distrib*: " $A \text{ co } (\bigcap i \in I. B \ i) = (\bigcap i \in I. A \text{ co } B \ i)$ "
 $\langle \text{proof} \rangle$

1.0.6 Intersection

lemma *constrains_Int*:
 $"[| F \in A \text{ co } A'; F \in B \text{ co } B' |] \implies F \in (A \cap B) \text{ co } (A' \cap B')"$
 $\langle \text{proof} \rangle$

lemma *constrains_INT*:
 $"(!i. i \in I \implies F \in (A \ i) \text{ co } (A' \ i)) \implies F \in (\bigcap i \in I. A \ i) \text{ co } (\bigcap i \in I. A' \ i)"$
 $\langle \text{proof} \rangle$

lemma *constrains_imp_subset*: " $F \in A \text{ co } A' \implies A \subseteq A'$ "
 $\langle \text{proof} \rangle$

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

lemma *constrains_trans*:
 $"[| F \in A \text{ co } B; F \in B \text{ co } C |] \implies F \in A \text{ co } C"$
 $\langle \text{proof} \rangle$

lemma *constrains_cancel*:
 $"[| F \in A \text{ co } (A' \cup B); F \in B \text{ co } B' |] \implies F \in A \text{ co } (A' \cup B')"$
 $\langle \text{proof} \rangle$

1.0.7 unless

lemma *unlessI*: " $F \in (A-B) \text{ co } (A \cup B) \implies F \in A \text{ unless } B$ "
 $\langle \text{proof} \rangle$

lemma *unlessD*: " $F \in A \text{ unless } B \implies F \in (A-B) \text{ co } (A \cup B)$ "
 $\langle \text{proof} \rangle$

1.0.8 stable

lemma *stableI*: " $F \in A \text{ co } A \implies F \in \text{stable } A$ "
 $\langle \text{proof} \rangle$

lemma *stableD*: " $F \in \text{stable } A \implies F \in A \text{ co } A$ "
 $\langle \text{proof} \rangle$

lemma *stable_UNIV [simp]*: " $\text{stable UNIV} = \text{UNIV}$ "
 $\langle \text{proof} \rangle$

1.0.9 Union

lemma *stable_Un*:
 $"[| F \in \text{stable } A; F \in \text{stable } A' |] \implies F \in \text{stable } (A \cup A')"$
 $\langle \text{proof} \rangle$

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 $\text{invariant } A \cap \text{invariant } B \subseteq \text{invariant } (A \cap B)$

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)"

AllowedActs F)"

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

```
definition all_total :: "'a program => bool" where
  "all_total F ==  $\forall \text{act} \in \text{Acts } F. \text{Domain act} = \text{UNIV}$ "
```

```
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  $\in$  A co B) = (F  $\in$  A co
  B)"
<proof>
```

```
lemma totalize_stable_iff [simp]: "(totalize F  $\in$  stable A) = (F  $\in$  stable
  A)"
<proof>
```

```
lemma totalize_invariant_iff [simp]:
  "(totalize F  $\in$  invariant A) = (F  $\in$  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

```

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

definition FP :: "'a program => 'a set" where
  "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.

definition

— 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 $A \subseteq \text{Domain act}$, which specifies that the action is enabled over all of A .

```
transient :: "'a set => 'a program set" where
  "transient A == {F.  $\exists \text{act} \in \text{Acts F. act} \subseteq A \subseteq \text{act}^{-1} A$ }"
```

definition

```
ensures :: "[ 'a set, 'a set ] => 'a program set" (infixl "ensures" 60)
where
  "A ensures B == (A-B co A  $\cup$  B)  $\cap$  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  $\in$  A ensures B ==> (A,B)  $\in$  leads F"
  | Trans: "[ (A,B)  $\in$  leads F; (B,C)  $\in$  leads F ] ==> (A,C)  $\in$  leads F"
  | Union: " $\forall A \in S. (A,B) \in \text{leads F} ==> (\text{Union } S, B) \in \text{leads F}"$ 
```

definition leadsTo :: "['a set, 'a set] => 'a program set" (infixl "leadsTo" 60) where

— visible version of the LEADS-TO relation

```
"A leadsTo B == {F. (A,B)  $\in$  leads F}"
```

definition wlt :: "['a program, 'a set] => 'a set" where

— predicate transformer: the largest set that leads to B

```
"wlt F B == Union {A. F  $\in$  A leadsTo B}"
```

notation (xsymbols)

```
leadsTo (infixl " $\longrightarrow$ " 60)
```

3.1 transient

lemma stable_transient:

```
"[ F  $\in$  stable A; F  $\in$  transient A ] ==>  $\exists \text{act} \in \text{Acts F. } A \subseteq \text{act}^{-1} A$ "
  <proof>
```

lemma stable_transient_empty:

```
"[ F  $\in$  stable A; F  $\in$  transient A; all_total F ] ==> A = {}"
```


<proof>

lemma *transient_strengthen:*

"[| $F \in \text{transient } A$; $B \subseteq A$ |] ==> $F \in \text{transient } B$ "

<proof>

lemma *transientI:*

"[| $\text{act}: \text{Acts } F$; $\text{act}''A \subseteq -A$ |] ==> $F \in \text{transient } A$ "

<proof>

lemma *transientE:*

"[| $F \in \text{transient } A$;

!! $\text{act}. [\text{act}: \text{Acts } F$; $\text{act}''A \subseteq -A$ |] ==> P |]

==> P "

<proof>

lemma *transient_empty [simp]:* " $\text{transient } \{\} = \text{UNIV}$ "

<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:*

"($\text{totalize } F \in \text{transient } A$) = ($\exists \text{act} \in \text{Acts } F. A \subseteq \text{Domain act} \ \& \ \text{act}''A \subseteq -A$)"

<proof>

lemma *totalize_transientI:*

"[| $\text{act}: \text{Acts } F$; $A \subseteq \text{Domain act}$; $\text{act}''A \subseteq -A$ |]

==> $\text{totalize } F \in \text{transient } A$ "

<proof>

3.2 ensures

lemma *ensuresI:*

"[| $F \in (A-B) \text{ co } (A \cup B)$; $F \in \text{transient } (A-B)$ |] ==> $F \in A \text{ ensures } B$ "

<proof>

lemma *ensuresD:*

" $F \in A \text{ ensures } B$ ==> $F \in (A-B) \text{ co } (A \cup B) \ \& \ F \in \text{transient } (A-B)$ "

<proof>

lemma *ensures_weaken_R:*

"[| $F \in A \text{ ensures } A'$; $A' \leq B'$ |] ==> $F \in A \text{ ensures } B'$ "

<proof>

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

lemma *stable_ensures_Int:*

"[| $F \in \text{stable } C$; $F \in A \text{ ensures } B$ |]

==> $F \in (C \cap A) \text{ ensures } (C \cap B)$ "

<proof>

lemma *stable_transient_ensures:*

"[| $F \in \text{stable } A$; $F \in \text{transient } C$; $A \subseteq B \cup C$ |] $\implies F \in A$ ensures B "
 $\langle \text{proof} \rangle$

lemma ensures_eq: " $(A \text{ ensures } B) = (A \text{ unless } B) \cap \text{transient } (A-B)$ "
 $\langle \text{proof} \rangle$

3.3 leadsTo

lemma leadsTo_Basis [intro]: " $F \in A \text{ ensures } B \implies F \in A \text{ leadsTo } B$ "
 $\langle \text{proof} \rangle$

lemma leadsTo_Trans:
 "[| $F \in A \text{ leadsTo } B$; $F \in B \text{ leadsTo } C$ |] $\implies F \in A \text{ leadsTo } C$ "
 $\langle \text{proof} \rangle$

lemma leadsTo_Basis':
 "[| $F \in A \text{ co } A \cup B$; $F \in \text{transient } A$ |] $\implies F \in A \text{ leadsTo } B$ "
 $\langle \text{proof} \rangle$

lemma transient_imp_leadsTo: " $F \in \text{transient } A \implies F \in A \text{ leadsTo } (\neg A)$ "
 $\langle \text{proof} \rangle$

Useful with cancellation, disjunction

lemma leadsTo_Un_duplicate: " $F \in A \text{ leadsTo } (A' \cup A') \implies F \in A \text{ leadsTo } A'$ "
 $\langle \text{proof} \rangle$

lemma leadsTo_Un_duplicate2:
 " $F \in A \text{ leadsTo } (A' \cup C \cup C) \implies F \in A \text{ leadsTo } (A' \cup C)$ "
 $\langle \text{proof} \rangle$

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

lemma leadsTo_Union:
 " $(\forall A. A \in S \implies F \in A \text{ leadsTo } B) \implies F \in (\text{Union } S) \text{ leadsTo } B$ "
 $\langle \text{proof} \rangle$

lemma leadsTo_Union_Int:
 " $(\forall A. A \in S \implies F \in (A \cap C) \text{ leadsTo } B) \implies F \in (\text{Union } S \cap C) \text{ leadsTo } B$ "
 $\langle \text{proof} \rangle$

lemma leadsTo_UN:
 " $(\forall i. i \in I \implies F \in (A \ i) \text{ leadsTo } B) \implies F \in (\bigcup i \in I. A \ i) \text{ leadsTo } B$ "
 $\langle \text{proof} \rangle$

Binary union introduction rule

lemma leadsTo_Un:
 "[| $F \in A \text{ leadsTo } C$; $F \in B \text{ leadsTo } C$ |] $\implies F \in (A \cup B) \text{ leadsTo } C$ "
 $\langle \text{proof} \rangle$

lemma single_leadsTo_I:
 " $(\forall x. x \in A \implies F \in \{x\} \text{ leadsTo } B) \implies F \in A \text{ leadsTo } B$ "

<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_sures: "A ⊆ B ==> F ∈ A ensures B"
<proof>
```

```
lemmas subset_imp_leadsTo = subset_imp_sures [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)"
```

$\langle proof \rangle$

lemma *leadsTo_UN_distrib*:

" $F \in (\bigcup i \in I. A \ i) \text{ leadsTo } B = (\forall i \in I. F \in (A \ i) \text{ leadsTo } B)$ "

$\langle proof \rangle$

lemma *leadsTo_Union_distrib*:

" $F \in (\text{Union } S) \text{ leadsTo } B = (\forall A \in S. F \in A \text{ leadsTo } B)$ "

$\langle proof \rangle$

lemma *leadsTo_weaken*:

" $[| F \in A \text{ leadsTo } A'; B \subseteq A; A' \leq B' |] \implies F \in B \text{ leadsTo } B'$ "

$\langle proof \rangle$

Set difference: maybe combine with *leadsTo_weaken_L*??

lemma *leadsTo_Diff*:

" $[| F \in (A-B) \text{ leadsTo } C; F \in B \text{ leadsTo } C |] \implies F \in A \text{ leadsTo } C$ "

$\langle proof \rangle$

lemma *leadsTo_UN_UN*:

" $(\forall i. i \in I \implies F \in (A \ i) \text{ leadsTo } (A' \ i))$

$\implies F \in (\bigcup i \in I. A \ i) \text{ leadsTo } (\bigcup i \in I. A' \ i)$ "

$\langle proof \rangle$

Binary union version

lemma *leadsTo_Un_Un*:

" $[| F \in A \text{ leadsTo } A'; F \in B \text{ leadsTo } B' |]$

$\implies F \in (A \cup B) \text{ leadsTo } (A' \cup B')$ "

$\langle proof \rangle$

lemma *leadsTo_cancel2*:

" $[| F \in A \text{ leadsTo } (A' \cup B); F \in B \text{ leadsTo } B' |]$

$\implies F \in A \text{ leadsTo } (A' \cup B')$ "

$\langle proof \rangle$

lemma *leadsTo_cancel_Diff2*:

" $[| F \in A \text{ leadsTo } (A' \cup B); F \in (B-A') \text{ leadsTo } B' |]$

$\implies F \in A \text{ leadsTo } (A' \cup B')$ "

$\langle proof \rangle$

lemma *leadsTo_cancel1*:

" $[| F \in A \text{ leadsTo } (B \cup A'); F \in B \text{ leadsTo } B' |]$

$\implies F \in A \text{ leadsTo } (B' \cup A')$ "

$\langle proof \rangle$

lemma *leadsTo_cancel_Diff1*:

" $[| F \in A \text{ leadsTo } (B \cup A'); F \in (B-A') \text{ leadsTo } B' |]$

$\implies F \in A \text{ leadsTo } (B' \cup A')$ "

$\langle proof \rangle$

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"

$\langle proof \rangle$

```

lemma bounded_induct:
  "[| wf r;
     $\forall m \in I. F \in (A \cap f^{-'\{m\}}) \text{ leadsTo } ((A \cap f^{-'(r^{-1} \text{ '' } \{m\})}) \cup B) \text{ |}]$ 
    ==>  $F \in A \text{ leadsTo } ((A - (f^{-'I})) \cup B)$ "
 $\langle proof \rangle$ 

```

```

lemma lessThan_induct:
  "[| !!m::nat.  $F \in (A \cap f^{-'\{m\}}) \text{ leadsTo } ((A \cap f^{-'\{..
    ==>  $F \in A \text{ leadsTo } B$ "
 $\langle proof \rangle$$ 
```

```

lemma lessThan_bounded_induct:
  "!!l::nat. [|  $\forall m \in \text{greaterThan } l. F \in (A \cap f^{-'\{m\}}) \text{ leadsTo } ((A \cap f^{-'(\text{lessThan } m)}) \cup B) \text{ |}]$ 
    ==>  $F \in A \text{ leadsTo } ((A \cap (f^{-'(\text{atMost } l)))) \cup B$ "
 $\langle proof \rangle$ 

```

```

lemma greaterThan_bounded_induct:
  "(!!l::nat.  $\forall m \in \text{lessThan } l. F \in (A \cap f^{-'\{m\}}) \text{ leadsTo } ((A \cap f^{-'(\text{greaterThan } m)}) \cup B) \text{ |}]$ 
    ==>  $F \in A \text{ leadsTo } ((A \cap (f^{-'(\text{atLeast } l)))) \cup B$ "
 $\langle proof \rangle$ 

```

3.6 wlt

Misra's property W3

```

lemma wlt_leadsTo: " $F \in (wlt F B) \text{ leadsTo } B$ "
 $\langle proof \rangle$ 

```

```

lemma leadsTo_subset: " $F \in A \text{ leadsTo } B \implies A \subseteq wlt F B$ "
 $\langle proof \rangle$ 

```

Misra's property W2

```

lemma leadsTo_eq_subset_wlt: " $F \in A \text{ leadsTo } B = (A \subseteq wlt F B)$ "
 $\langle proof \rangle$ 

```

Misra's property W4

```

lemma wlt_increasing: " $B \subseteq wlt F B$ "
 $\langle proof \rangle$ 

```

Used in the Trans case below

```

lemma lemma1:
  "[|  $B \subseteq A2$ ;
     $F \in (A1 - B) \text{ co } (A1 \cup B)$ ;
     $F \in (A2 - C) \text{ co } (A2 \cup C) \text{ |}]$ 
    ==>  $F \in (A1 \cup A2 - C) \text{ co } (A1 \cup A2 \cup C)$ "

```

<proof>

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

lemma *leadsTo_123*:

" $F \in A \text{ leadsTo } A'$ "

$\implies \exists B. A \subseteq B \ \& \ F \in B \text{ leadsTo } A' \ \& \ F \in (B - A') \text{ co } (B \cup A')$ "

<proof>

Misra's property W5

lemma *wlt_constrains_wlt*: " $F \in (\text{wlt } F \ B - B) \text{ co } (\text{wlt } F \ B)$ "

<proof>

3.7 Completion: Binary and General Finite versions

lemma *completion_lemma* :

" $[\ W = \text{wlt } F \ (B' \cup C);$

$F \in A \text{ leadsTo } (A' \cup C); \ F \in A' \text{ co } (A' \cup C);$

$F \in B \text{ leadsTo } (B' \cup C); \ F \in B' \text{ co } (B' \cup C) \]$

$\implies F \in (A \cap B) \text{ leadsTo } ((A' \cap B') \cup C)$ "

<proof>

lemmas *completion* = *completion_lemma* [*OF refl*]

lemma *finite_completion_lemma*:

"*finite* $I \implies (\forall i \in I. F \in (A \ i) \text{ leadsTo } (A' \ i \cup C)) \dashrightarrow$

$(\forall i \in I. F \in (A' \ i) \text{ co } (A' \ i \cup C)) \dashrightarrow$

$F \in (\bigcap i \in I. A \ i) \text{ leadsTo } ((\bigcap i \in I. A' \ i) \cup C)$ "

<proof>

lemma *finite_completion*:

" $[\ \text{finite } I;$

$!!i. i \in I \implies F \in (A \ i) \text{ leadsTo } (A' \ i \cup C);$

$!!i. i \in I \implies F \in (A' \ i) \text{ co } (A' \ i \cup C) \]$

$\implies F \in (\bigcap i \in I. A \ i) \text{ leadsTo } ((\bigcap i \in I. A' \ i) \cup C)$ "

<proof>

lemma *stable_completion*:

" $[\ F \in A \text{ leadsTo } A'; \ F \in \text{stable } A';$

$F \in B \text{ leadsTo } B'; \ F \in \text{stable } B' \]$

$\implies F \in (A \cap B) \text{ leadsTo } (A' \cap B')$ "

<proof>

lemma *finite_stable_completion*:

" $[\ \text{finite } I;$

$!!i. i \in I \implies F \in (A \ i) \text{ leadsTo } (A' \ i);$

$!!i. i \in I \implies F \in \text{stable } (A' \ i) \]$

$\implies F \in (\bigcap i \in I. A \ i) \text{ leadsTo } (\bigcap i \in I. A' \ i)$ "

<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"
```

definition Constrains :: "['a set, 'a set] => 'a program set" (infixl "Co" 60)
where

"A Co B == {F. F ∈ (reachable F ∩ A) co B}"

definition Unless :: "['a set, 'a set] => 'a program set" (infixl "Unless" 60) where

"A Unless B == (A-B) Co (A ∪ B)"

definition Stable :: "'a set => 'a program set" where

"Stable A == A Co A"

definition Always :: "'a set => 'a program set" where

"Always A == {F. Init F ⊆ A} ∩ Stable A"

definition Increasing :: "['a => 'b::order] => 'a program set" where

"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  $\subseteq$  Acts F ==> G  $\in$  stable (reachable F)"
<proof>

```

```

lemma invariant_reachable: "F  $\in$  invariant (reachable F)"
<proof>

```

```

lemma invariant_includes_reachable: "F  $\in$  invariant A ==> reachable F  $\subseteq$  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  $\in$  (reachable F  $\cap$  A) co (reachable F  $\cap$  B)}"
<proof>

```

```

lemma constrains_imp_Constrains: "F  $\in$  A co A' ==> F  $\in$  A Co A'"
<proof>

```

```

lemma stable_imp_Stable: "F  $\in$  stable A ==> F  $\in$  Stable A"
<proof>

```

```

lemma ConstrainsI:
  "(!!act s s'. [| act: Acts F; (s,s')  $\in$  act; s  $\in$  A |] ==> s': A')
  ==> F  $\in$  A Co A'"
<proof>

```

```

lemma Constrains_empty [iff]: "F  $\in$  {} Co B"
<proof>

```

```

lemma Constrains_UNIV [iff]: "F  $\in$  A Co UNIV"
<proof>

```

```

lemma Constrains_weaken_R:
  "[| F  $\in$  A Co A'; A'  $\leq$  B' |] ==> F  $\in$  A Co B'"
<proof>

```

```

lemma Constrains_weaken_L:
  "[| F  $\in$  A Co A'; B  $\subseteq$  A |] ==> F  $\in$  B Co A'"
<proof>

```

```

lemma Constrains_weaken:
  "[| F  $\in$  A Co A'; B  $\subseteq$  A; A'  $\leq$  B' |] ==> F  $\in$  B Co B'"
<proof>

```

lemma *Constrains_Un*:

"[| $F \in A \text{ Co } A'$; $F \in B \text{ Co } B'$ |] $\implies F \in (A \cup B) \text{ Co } (A' \cup B')$ "
 $\langle \text{proof} \rangle$

lemma *Constrains_UN*:

assumes *Co*: " $\forall i. i \in I \implies F \in (A \ i) \text{ Co } (A' \ i)$ "
 shows " $F \in (\bigcup i \in I. A \ i) \text{ Co } (\bigcup i \in I. A' \ i)$ "
 $\langle \text{proof} \rangle$

lemma *Constrains_Int*:

"[| $F \in A \text{ Co } A'$; $F \in B \text{ Co } B'$ |] $\implies F \in (A \cap B) \text{ Co } (A' \cap B')$ "
 $\langle \text{proof} \rangle$

lemma *Constrains_INT*:

assumes *Co*: " $\forall i. i \in I \implies F \in (A \ i) \text{ Co } (A' \ i)$ "
 shows " $F \in (\bigcap i \in I. A \ i) \text{ Co } (\bigcap i \in I. A' \ i)$ "
 $\langle \text{proof} \rangle$

lemma *Constrains_imp_subset*: " $F \in A \text{ Co } A' \implies \text{reachable } F \cap A \subseteq A'$ "

$\langle \text{proof} \rangle$

lemma *Constrains_trans*: "[| $F \in A \text{ Co } B$; $F \in B \text{ Co } C$ |] $\implies F \in A \text{ Co } C$ "

$\langle \text{proof} \rangle$

lemma *Constrains_cancel*:

"[| $F \in A \text{ Co } (A' \cup B)$; $F \in B \text{ Co } B'$ |] $\implies F \in A \text{ Co } (A' \cup B')$ "
 $\langle \text{proof} \rangle$

4.3 Stable

lemma *Stable_eq*: "[| $F \in \text{Stable } A$; $A = B$ |] $\implies F \in \text{Stable } B$ "

$\langle \text{proof} \rangle$

lemma *Stable_eq_stable*: " $(F \in \text{Stable } A) = (F \in \text{stable } (\text{reachable } F \cap A))$ "

$\langle \text{proof} \rangle$

lemma *StableI*: " $F \in A \text{ Co } A \implies F \in \text{Stable } A$ "

$\langle \text{proof} \rangle$

lemma *StableD*: " $F \in \text{Stable } A \implies F \in A \text{ Co } A$ "

$\langle \text{proof} \rangle$

lemma *Stable_Un*:

"[| $F \in \text{Stable } A$; $F \in \text{Stable } A'$ |] $\implies F \in \text{Stable } (A \cup A')$ "
 $\langle \text{proof} \rangle$

lemma *Stable_Int*:

"[| $F \in \text{Stable } A$; $F \in \text{Stable } A'$ |] $\implies F \in \text{Stable } (A \cap A')$ "
 $\langle \text{proof} \rangle$

lemma *Stable_Constrains_Un*:

"[| $F \in \text{Stable } C$; $F \in A \text{ Co } (C \cup A')$ |]

```

    ==> 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 $F \subseteq A$; $F \in \text{Stable } A$ |] $\implies F \in \text{Always } A$ "
 $\langle \text{proof} \rangle$

lemma *AlwaysD*: " $F \in \text{Always } A \implies \text{Init } F \subseteq A \ \& \ F \in \text{Stable } A$ "
 $\langle \text{proof} \rangle$

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

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

lemma *Always_includes_reachable*: " $F \in \text{Always } A \implies \text{reachable } F \subseteq A$ "
 $\langle \text{proof} \rangle$

lemma *invariant_imp_Always*:
 $\quad "F \in \text{invariant } A \implies F \in \text{Always } A"$
 $\langle \text{proof} \rangle$

lemmas *Always_reachable* =
 $\quad \text{invariant_reachable [THEN invariant_imp_Always, standard]}$

lemma *Always_eq_invariant_reachable*:
 $\quad "Always \ A = \{F. F \in \text{invariant } (\text{reachable } F \cap A)\}"$
 $\langle \text{proof} \rangle$

lemma *Always_eq_includes_reachable*: " $Always \ A = \{F. \text{reachable } F \subseteq A\}"$
 $\langle \text{proof} \rangle$

lemma *Always_UNIV_eq* [simp]: " $Always \ UNIV = UNIV$ "
 $\langle \text{proof} \rangle$

lemma *UNIV_AlwaysI*: " $UNIV \subseteq A \implies F \in \text{Always } A$ "
 $\langle \text{proof} \rangle$

lemma *Always_eq_UN_invariant*: " $Always \ A = (\bigcup I \in \text{Pow } A. \text{invariant } I)"$
 $\langle \text{proof} \rangle$

lemma *Always_weaken*: "[| $F \in \text{Always } A$; $A \subseteq B$ |] $\implies F \in \text{Always } B$ "
 $\langle \text{proof} \rangle$

4.7 "Co" rules involving Always

lemma *Always_Constrains_pre*:
 $\quad "F \in \text{Always } INV \implies (F \in (INV \cap A) \text{ Co } A') = (F \in A \text{ Co } A')"$
 $\langle \text{proof} \rangle$

lemma *Always_Constrains_post*:
 $\quad "F \in \text{Always } INV \implies (F \in A \text{ Co } (INV \cap A')) = (F \in A \text{ Co } A')"$
 $\langle \text{proof} \rangle$

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)"

```

A)"
⟨proof⟩

end

```

5 Weak Progress

```
theory SubstAx imports WFair Constrains begin
```

```
definition Ensures :: "[ 'a set, 'a set ] => 'a program set" (infixl "Ensures"
60) where
```

```
"A Ensures B == {F. F ∈ (reachable F ∩ A) ensures B}"
```

```
definition LeadsTo :: "[ 'a set, 'a set ] => 'a program set" (infixl "LeadsTo"
60) where
```

```
"A LeadsTo B == {F. F ∈ (reachable F ∩ A) leadsTo B}"
```

```
notation (xsymbols)
```

```
LeadsTo (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]: " $F \in A$ LeadsTo UNIV"
 <proof>

Useful with cancellation, disjunction

lemma *LeadsTo_Un_duplicate*:
 " $F \in A$ LeadsTo $(A' \cup A')$ $\implies F \in A$ LeadsTo A' "
 <proof>

lemma *LeadsTo_Un_duplicate2*:
 " $F \in A$ LeadsTo $(A' \cup C \cup C)$ $\implies F \in A$ LeadsTo $(A' \cup C)$ "
 <proof>

lemma *LeadsTo_UN*:
 " $(\forall i. i \in I \implies F \in (A \ i) \text{ LeadsTo } B) \implies F \in (\bigcup i \in I. A \ i) \text{ LeadsTo } B$ "
 <proof>

Binary union introduction rule

lemma *LeadsTo_Un*:
 " $[F \in A \text{ LeadsTo } C; F \in B \text{ LeadsTo } C] \implies F \in (A \cup B) \text{ LeadsTo } C$ "
 <proof>

Lets us look at the starting state

lemma *single_LeadsTo_I*:
 " $(\forall s. s \in A \implies F \in \{s\} \text{ LeadsTo } B) \implies F \in A \text{ LeadsTo } B$ "
 <proof>

lemma *subset_imp_LeadsTo*: " $A \subseteq B \implies F \in A \text{ LeadsTo } B$ "
 <proof>

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

lemma *LeadsTo_weaken_R* [rule_format]:
 " $[F \in A \text{ LeadsTo } A'; A' \subseteq B'] \implies F \in A \text{ LeadsTo } B'$ "
 <proof>

lemma *LeadsTo_weaken_L* [rule_format]:
 " $[F \in A \text{ LeadsTo } A'; B \subseteq A] \implies F \in B \text{ LeadsTo } A'$ "
 <proof>

lemma *LeadsTo_weaken*:
 " $[F \in A \text{ LeadsTo } A'; B \subseteq A; A' \subseteq B'] \implies F \in B \text{ LeadsTo } B'$ "
 <proof>

lemma *Always_LeadsTo_weaken*:
 " $[F \in \text{Always } C; F \in A \text{ LeadsTo } A'; C \cap B \subseteq A; C \cap A' \subseteq B'] \implies F \in B \text{ LeadsTo } B'$ "

$\langle proof \rangle$

lemma *LeadsTo_Un_post*: " $F \in A \text{ LeadsTo } B \implies F \in (A \cup B) \text{ LeadsTo } B$ "
 $\langle proof \rangle$

lemma *LeadsTo_Trans_Un*:
 " $[| F \in A \text{ LeadsTo } B; F \in B \text{ LeadsTo } C |]$
 $\implies F \in (A \cup B) \text{ LeadsTo } C$ "
 $\langle proof \rangle$

lemma *LeadsTo_Un_distrib*:
 " $(F \in (A \cup B) \text{ LeadsTo } C) = (F \in A \text{ LeadsTo } C \ \& \ F \in B \text{ LeadsTo } C)$ "
 $\langle proof \rangle$

lemma *LeadsTo_UN_distrib*:
 " $(F \in (\bigcup i \in I. A \ i) \text{ LeadsTo } B) = (\forall i \in I. F \in (A \ i) \text{ LeadsTo } B)$ "
 $\langle proof \rangle$

lemma *LeadsTo_Union_distrib*:
 " $(F \in (\text{Union } S) \text{ LeadsTo } B) = (\forall A \in S. F \in A \text{ LeadsTo } B)$ "
 $\langle proof \rangle$

lemma *LeadsTo_Basis*: " $F \in A \text{ Ensures } B \implies F \in A \text{ LeadsTo } B$ "
 $\langle proof \rangle$

lemma *EnsuresI*:
 " $[| F \in (A-B) \text{ Co } (A \cup B); F \in \text{transient } (A-B) |]$
 $\implies F \in A \text{ Ensures } B$ "
 $\langle proof \rangle$

lemma *Always_LeadsTo_Basis*:
 " $[| F \in \text{Always } INV;$
 $F \in (INV \cap (A-A')) \text{ Co } (A \cup A');$
 $F \in \text{transient } (INV \cap (A-A')) |]$
 $\implies F \in A \text{ LeadsTo } A'$ "
 $\langle proof \rangle$

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

lemma *LeadsTo_Diff*:
 " $[| F \in (A-B) \text{ LeadsTo } C; F \in (A \cap B) \text{ LeadsTo } C |]$
 $\implies F \in A \text{ LeadsTo } C$ "
 $\langle proof \rangle$

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* ($\neg A \cup B$)) \cap (*B LeadsTo A*)"
Equality_def: "*A <==> B* == ($\neg A \cup B$) \cap ($A \cup \neg B$)"

lemma *Always_at_FP*:

"[*F* \in *A LeadsTo B*; all_total *F*] ==> *F* \in *Always* ($\neg((FP\ F) \cap A \cap \neg B)$)"
 <proof>

lemma *Detects_Trans*:

"[*F* \in *A Detects B*; *F* \in *B Detects C*] ==> *F* \in *A Detects C*"
 <proof>

lemma *Detects_refl*: "*F* \in *A Detects A*"

<proof>

lemma *Detects_eq_Un*: " $(A <==> B) = (A \cap B) \cup (\neg A \cap \neg B)$ "

<proof>

lemma *Detects_antisym*:

"[*F* \in *A Detects B*; *F* \in *B Detects A*] ==> *F* \in *Always* (*A <==> B*)"
 <proof>

lemma *Detects_Always*:

"[*F* \in *A Detects B*; all_total *F*] ==> *F* \in *Always* ($\neg(FP\ F) \cup (A <==> B)$)"
 <proof>

lemma *Detects_Imp_LeadstoEQ*:

"*F* \in *A Detects B* ==> *F* \in *UNIV LeadsTo* (*A <==> B*)"
 <proof>

end

7 Unions of Programs

theory *Union* imports *SubstAx FP* begin

definition

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

definition

```
OK :: "[ 'a set, 'a => 'b program ] => bool"
where "OK I F = ( $\forall i \in I. \forall j \in I - \{i\}. Acts (F i) \subseteq AllowedActs (F j)$ )"
```

definition

```
JOIN :: "[ 'a set, 'a => 'b program ] => 'b program"
where "JOIN I F = mk_program ( $\bigcap i \in I. Init (F i), \bigcup i \in I. Acts (F i),$ 
                                $\bigcap i \in I. AllowedActs (F i)$ )"
```

definition

```
Join :: "[ 'a program, 'a program ] => 'a program"      (infixl "Join" 65)
where "F Join G = mk_program (Init F  $\cap$  Init G, Acts F  $\cup$  Acts G,
                               AllowedActs F  $\cap$  AllowedActs G)"
```

definition

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

definition

```
safety_prop :: "'a program set => bool"
where "safety_prop X <-> SKIP: X & ( $\forall G. Acts G \subseteq UNION X Acts \rightarrow G \in X$ )"
```

notation (xsymbols)

```
SKIP ("⊥") and
Join (infixl "⊔" 65)
```

syntax

```
"_JOIN1"      :: "[pttrns, 'b set] => 'b set"          ("(3JN _./ _)" 10)
"_JOIN"       :: "[pttrn, 'a set, 'b set] => 'b set"    ("(3JN _:./ _)" 10)
```

syntax (xsymbols)

```
"_JOIN1" :: "[pttrns, 'b set] => 'b set"          ("(3⊔ _./ _)" 10)
"_JOIN"  :: "[pttrn, 'a set, 'b set] => 'b set"    ("(3⊔ _∈./ _)" 10)
```

translations

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

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

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 ($\sqcup i \in I. F\ i$) = insert Id ($\bigcup i \in I. Acts\ (F\ i)$)"
 <proof>

lemma *AllowedActs_JN [simp]*:
 "AllowedActs ($\sqcup i \in I. F\ i$) = ($\bigcap i \in I. AllowedActs\ (F\ i)$)"
 <proof>

lemma *JN_cong [cong]*:
 "[| I=J; !!i. i \in J ==> F i = G i |] ==> ($\sqcup i \in I. F\ i$) = ($\sqcup i \in J. G\ i$)"
 <proof>

7.5 Algebraic laws

lemma *Join_commute*: "F \sqcup G = G \sqcup F"
 <proof>

lemma *Join_assoc*: "(F \sqcup G) \sqcup H = F \sqcup (G \sqcup H)"
 <proof>

lemma *Join_left_commute*: "A \sqcup (B \sqcup C) = B \sqcup (A \sqcup C)"
 <proof>

lemma *Join_SKIP_left [simp]*: "SKIP \sqcup F = F"
 <proof>

lemma *Join_SKIP_right [simp]*: "F \sqcup SKIP = F"
 <proof>

lemma *Join_absorb [simp]*: "F \sqcup F = F"
 <proof>

lemma *Join_left_absorb*: "F \sqcup (F \sqcup G) = F \sqcup G"
 <proof>

lemmas *Join_ac = Join_assoc Join_left_absorb Join_commute Join_left_commute*

7.6 Laws Governing \sqcup

lemma *JN_absorb*: "k \in I ==> F k \sqcup ($\sqcup i \in I. F\ i$) = ($\sqcup i \in I. F\ i$)"
 <proof>

lemma *JN_Un*: "($\sqcup i \in I \cup J. F\ i$) = (($\sqcup i \in I. F\ i$) \sqcup ($\sqcup i \in J. F\ i$))"
 <proof>

lemma *JN_constant*: "($\sqcup i \in I. c$) = (if I={} then SKIP else c)"
 <proof>

lemma *JN_Join_distrib*:
 "($\sqcup i \in I. F\ i \sqcup G\ i$) = ($\sqcup i \in I. F\ i$) \sqcup ($\sqcup i \in I. G\ i$)"
 <proof>

lemma *JN_Join_miniscope*:
 "i ∈ I ==> ($\bigsqcup_{i \in I} F \sqcup G$) = ($(\bigsqcup_{i \in I} F \sqcup G)$)"
 <proof>

lemma *JN_Join_diff*: "i ∈ I ==> F i \sqcup JOIN (I - {i}) F = JOIN I F"
 <proof>

7.7 Safety: co, stable, FP

lemma *JN_constrains*:
 "i ∈ I ==> ($\bigsqcup_{i \in I} F \sqcup G$) ∈ A co B = (∀ i ∈ I. F i ∈ A co B)"
 <proof>

lemma *Join_constrains [simp]*:
 "(F \sqcup G ∈ A co B) = (F ∈ A co B & G ∈ A co B)"
 <proof>

lemma *Join_unless [simp]*:
 "(F \sqcup 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 \sqcup G ∈ (A ∩ B) co (A' ∪ B')"
 <proof>

lemma *JN_constrains_weaken*:
 "[| ∀ i ∈ I. F i ∈ A i co A' i; i ∈ I |]
 ==> ($\bigsqcup_{i \in I} F \sqcup G$) ∈ ($\bigcap_{i \in I} A \sqcup G$) co ($\bigcup_{i \in I} A' \sqcup G$)"
 <proof>

lemma *JN_stable*: "($\bigsqcup_{i \in I} F \sqcup G$) ∈ stable A = (∀ i ∈ I. F i ∈ stable A)"
 <proof>

lemma *invariant_JN_I*:
 "[| !!i. i ∈ I ==> F i ∈ invariant A; i ∈ I |]
 ==> ($\bigsqcup_{i \in I} F \sqcup G$) ∈ invariant A"
 <proof>

lemma *Join_stable [simp]*:
 "(F \sqcup G ∈ stable A) =
 (F ∈ stable A & G ∈ stable A)"
 <proof>

lemma *Join_increasing [simp]*:
 "(F \sqcup 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 ( $\bigcup i \in I. F i$ ) = ( $\bigcap i \in I. FP (F i)$ )"
<proof>

```

7.8 Progress: transient, ensures

```

lemma JN_transient:
  "i ∈ I ==>
   ( $\bigcup i \in I. F i$ ) ∈ transient A = ( $\exists i \in I. F i \in transient A$ )"
<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 ==>
   ( $\bigcup i \in I. F i$ ) ∈ A ensures B =
   (( $\forall i \in I. F i \in (A-B) \text{ co } (A \cup B)$ ) & ( $\exists i \in I. F i \in A \text{ ensures } B$ ))"
<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 \sqcup G) = Allowed F \cap Allowed G"*
<proof>

lemma *Allowed_JN [simp]: "Allowed (JOIN I F) = ($\bigcap_{i \in I} \text{Allowed } (F \ i)$)"*
<proof>

lemma *ok_iff_Allowed: "F ok G = (F \in Allowed G & G \in Allowed F)"*
<proof>

lemma *OK_iff_Allowed: "OK I F = ($\forall i \in I. \forall j \in I - \{i\}. F \ i \in \text{Allowed}(F \ j)$)"*
<proof>

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

lemma *safety_prop_Acts_iff:*
"safety_prop X ==> (Acts G \subseteq insert Id (UNION X Acts)) = (G \in X)"
<proof>

lemma *safety_prop_AllowedActs_iff_Allowed:*
"safety_prop X ==> (UNION X Acts \subseteq AllowedActs F) = (X \subseteq 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 \subseteq 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 \cap Y)"
<proof>

lemma *safety_prop_INTER1 [simp]:*
"(!i. safety_prop (X i)) ==> safety_prop ($\bigcap_{i \in I} X \ i$)"
<proof>

lemma *safety_prop_INTER [simp]:*
"(!i. i \in I ==> safety_prop (X i)) ==> safety_prop ($\bigcap_{i \in I} X \ i$)"
<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

instantiation program :: (type) ord
begin

definition
  component_def: "F ≤ H <-> (∃ G. F ⊔ G = H)"

definition
  strict_component_def: "F < (H :: 'a program) <-> (F ≤ H & F ≠ H)"

instance <proof>

end

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

```

```

definition strict_component_of :: "'a program  $\Rightarrow$  'a program  $\Rightarrow$  bool" (infixl "strict'_component'_of"
50) where
  "F strict_component_of H == F component_of H & F  $\neq$  H"

definition preserves :: "('a  $\Rightarrow$  'b)  $\Rightarrow$  'a program set" where
  "preserves v ==  $\bigcap z$ . stable {s. v s = z}"

definition localize :: "('a  $\Rightarrow$  'b)  $\Rightarrow$  'a program  $\Rightarrow$  'a program" where
  "localize v F == mk_program(Init F, Acts F,
    AllowedActs F  $\cap$  ( $\bigcup G \in$  preserves v. Acts G))"

definition funPair :: "[ 'a  $\Rightarrow$  'b, 'a  $\Rightarrow$  'c, 'a ]  $\Rightarrow$  'b * 'c" where
  "funPair f g == %x. (f x, g x)"

```

8.1 The component relation

```

lemma componentI: "H  $\leq$  F | H  $\leq$  G  $\Rightarrow$  H  $\leq$  (F  $\sqcup$  G)"
<proof>

lemma component_eq_subset:
  "(F  $\leq$  G) =
  (Init G  $\subseteq$  Init F & Acts F  $\subseteq$  Acts G & AllowedActs G  $\subseteq$  AllowedActs F)"
<proof>

lemma component_SKIP [iff]: "SKIP  $\leq$  F"
<proof>

lemma component_refl [iff]: "F  $\leq$  (F :: 'a program)"
<proof>

lemma SKIP_minimal: "F  $\leq$  SKIP  $\Rightarrow$  F = SKIP"
<proof>

lemma component_Join1: "F  $\leq$  (F  $\sqcup$  G)"
<proof>

lemma component_Join2: "G  $\leq$  (F  $\sqcup$  G)"
<proof>

lemma Join_absorb1: "F  $\leq$  G  $\Rightarrow$  F  $\sqcup$  G = G"
<proof>

lemma Join_absorb2: "G  $\leq$  F  $\Rightarrow$  F  $\sqcup$  G = F"
<proof>

lemma JN_component_iff: "((JOIN I F)  $\leq$  H) = ( $\forall i \in I$ . F i  $\leq$  H)"
<proof>

lemma component_JN: "i  $\in$  I  $\Rightarrow$  (F i)  $\leq$  ( $\bigsqcup i \in I$ . (F i))"
<proof>

lemma component_trans: "[| F  $\leq$  G; G  $\leq$  H |]  $\Rightarrow$  F  $\leq$  (H :: 'a program)"
<proof>

```

lemma *component_antisym*: " $[| F \leq G; G \leq F |] \implies F = (G :: \text{'a program})$ "
 $\langle \text{proof} \rangle$

lemma *Join_component_iff*: " $((F \sqcup G) \leq H) = (F \leq H \ \& \ G \leq H)$ "
 $\langle \text{proof} \rangle$

lemma *component_constrains*: " $[| F \leq G; G \in A \text{ co } B |] \implies F \in A \text{ co } B$ "
 $\langle \text{proof} \rangle$

lemma *component_stable*: " $[| F \leq G; G \in \text{stable } A |] \implies F \in \text{stable } A$ "
 $\langle \text{proof} \rangle$

lemmas *program_less_le* = *strict_component_def*

8.2 The preserves property

lemma *preservesI*: " $(!!z. F \in \text{stable } \{s. v \ s = z\}) \implies F \in \text{preserves } v$ "
 $\langle \text{proof} \rangle$

lemma *preserves_imp_eq*:
 $[| F \in \text{preserves } v; \text{ act} \in \text{Acts } F; (s, s') \in \text{act} |] \implies v \ s = v \ s'$
 $\langle \text{proof} \rangle$

lemma *Join_preserves [iff]*:
 $(F \sqcup G \in \text{preserves } v) = (F \in \text{preserves } v \ \& \ G \in \text{preserves } v)$
 $\langle \text{proof} \rangle$

lemma *JN_preserves [iff]*:
 $(\text{JOIN } I \ F \in \text{preserves } v) = (\forall i \in I. F \ i \in \text{preserves } v)$
 $\langle \text{proof} \rangle$

lemma *SKIP_preserves [iff]*: " $\text{SKIP} \in \text{preserves } v$ "
 $\langle \text{proof} \rangle$

lemma *funPair_apply [simp]*: " $(\text{funPair } f \ g) \ x = (f \ x, \ g \ x)$ "
 $\langle \text{proof} \rangle$

lemma *preserves_funPair*: " $\text{preserves } (\text{funPair } v \ w) = \text{preserves } v \cap \text{preserves } w$ "
 $\langle \text{proof} \rangle$

declare *preserves_funPair* [THEN *eqset_imp_iff*, *iff*]

lemma *funPair_o_distrib*: " $(\text{funPair } f \ g) \ o \ h = \text{funPair } (f \ o \ h) \ (g \ o \ h)$ "
 $\langle \text{proof} \rangle$

lemma *fst_o_funPair [simp]*: " $\text{fst} \ o \ (\text{funPair } f \ g) = f$ "
 $\langle \text{proof} \rangle$

lemma *snd_o_funPair [simp]*: " $\text{snd} \ o \ (\text{funPair } f \ g) = g$ "

<proof>

lemma subset_preserves_o: "preserves $v \subseteq$ preserves ($w \circ v$)"

<proof>

lemma preserves_subset_stable: "preserves $v \subseteq$ stable $\{s. P (v s)\}$ "

<proof>

lemma preserves_subset_increasing: "preserves $v \subseteq$ increasing v "

<proof>

lemma preserves_id_subset_stable: "preserves $id \subseteq$ stable A "

<proof>

lemma safety_prop_preserves [iff]: "safety_prop (preserves v)"

<proof>

lemma stable_localTo_stable2:

"[| $F \in$ stable $\{s. P (v s) (w s)\};$
 $G \in$ preserves v ; $G \in$ preserves w |]
 $\implies F \sqcup G \in$ stable $\{s. P (v s) (w s)\}$ "

<proof>

lemma Increasing_preserves_Stable:

"[| $F \in$ stable $\{s. v s \leq w s\};$ $G \in$ preserves v ; $F \sqcup G \in$ Increasing w |]
 $\implies F \sqcup G \in$ Stable $\{s. v s \leq w s\}$ "

<proof>

lemma component_of_imp_component: " F component_of $H \implies F \leq H$ "

<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 |] $\implies F$ component_of H "

<proof>

lemmas strict_component_of_eq = strict_component_of_def

```

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  $\cap$  ( $\bigcup G \in \text{preserves } v. \text{Acts } G$ )"
  <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

```

definition ex_prop :: "'a program set => bool" where
  "ex_prop X ==  $\forall F G. F \text{ ok } G \rightarrow F \in X \mid G \in X \rightarrow (F \sqcup G) \in X$ "

```

```

definition strict_ex_prop :: "'a program set => bool" where
  "strict_ex_prop X ==  $\forall F G. F \text{ ok } G \rightarrow (F \in X \mid G \in X) = (F \sqcup G \in X)$ "

```

```

definition uv_prop :: "'a program set => bool" where
  "uv_prop X == SKIP  $\in$  X & ( $\forall F G. F \text{ ok } G \rightarrow F \in X \ \& \ G \in X \rightarrow (F \sqcup G) \in X$ )"

```

```

definition strict_uv_prop :: "'a program set => bool" where
  "strict_uv_prop X ==
    SKIP  $\in$  X & ( $\forall F G. F \text{ ok } G \rightarrow (F \in X \ \& \ G \in X) = (F \sqcup G \in X)$ )"

```

Guarantees properties

```

definition guar :: "[ 'a program set, 'a program set ] => 'a program set" (infixl
  "guarantees" 55) where

```

```

  "X guarantees Y == {F.  $\forall G. F \text{ ok } G \rightarrow F \sqcup G \in X \rightarrow F \sqcup G \in Y$ }"

```

```

definition wg :: "[ 'a program, 'a program set ] => 'a program set" where
  "wg F Y == Union({X. F  $\in$  X guarantees Y})"

```



```

definition wx :: "('a program) set => ('a program)set" where
  "wx X == Union({Y. Y  $\subseteq$  X & ex_prop Y})"

```

```

definition welldef :: "'a program set" where
  "welldef == {F. Init F  $\neq$  {}}"

```

```

definition refines :: "[ 'a program, 'a program, 'a program set] => bool"
  ("(3_ refines _ wrt _)" [10,10,10] 10) where
  "G refines F wrt X ==
     $\forall H. (F \text{ ok } H \ \& \ G \text{ ok } H \ \& \ F \sqcup H \in \text{welldef} \cap X) \rightarrow$ 
     $(G \sqcup H \in \text{welldef} \cap X)"$ 

```

```

definition iso_refines :: "[ 'a program, 'a program, 'a program set] => bool"
  ("(3_ iso' refines _ wrt _)" [10,10,10] 10) where
  "G iso_refines F wrt X ==
    F  $\in \text{welldef} \cap X \rightarrow G \in \text{welldef} \cap X"$ 

```

```

lemma OK_insert_iff:
  "(OK (insert i I) F) =
    (if i  $\in$  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  $\cap$  X  $\neq$  {}  $\rightarrow$  OK GG (%G. G)  $\rightarrow$  ( $\bigcup$  G  $\in$  GG. G)  $\in$  X"
<proof>

```

```

lemma ex2:
  " $\forall$  GG. finite GG & GG  $\cap$  X  $\neq$  {}  $\rightarrow$  OK GG (%G. G)  $\rightarrow$  ( $\bigcup$  G  $\in$  GG. G):X
    ==> ex_prop X"
<proof>

```

```

lemma ex_prop_finite:
  "ex_prop X =
    ( $\forall$  GG. finite GG & GG  $\cap$  X  $\neq$  {} & OK GG (%G. G)  $\rightarrow$  ( $\bigcup$  G  $\in$  GG. G)  $\in$ 
    X)"
<proof>

```

```

lemma ex_prop_equiv:
  "ex_prop X = ( $\forall$  G. G  $\in$  X = ( $\forall$  H. (G component_of H)  $\rightarrow$  H  $\in$  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*:

" $(\bigcup i \in I. X\ i)\ \text{guarantees}\ Y = (\bigcap i \in I. X\ i\ \text{guarantees}\ Y)$ "
 <proof>

lemma *guarantees_Un_left*:

" $(X \cup Y)\ \text{guarantees}\ Z = (X\ \text{guarantees}\ Z) \cap (Y\ \text{guarantees}\ Z)$ "
 <proof>

lemma *guarantees_INT_right*:

" $X\ \text{guarantees}\ (\bigcap i \in I. Y\ i) = (\bigcap i \in I. X\ \text{guarantees}\ Y\ i)$ "
 <proof>

lemma *guarantees_Int_right*:

" $Z\ \text{guarantees}\ (X \cap Y) = (Z\ \text{guarantees}\ X) \cap (Z\ \text{guarantees}\ Y)$ "
 <proof>

lemma *guarantees_Int_right_I*:

" $[| F \in Z\ \text{guarantees}\ X; F \in Z\ \text{guarantees}\ Y |]$
 $\implies F \in Z\ \text{guarantees}\ (X \cap Y)$ "
 <proof>

lemma *guarantees_INT_right_iff*:

" $(F \in X\ \text{guarantees}\ (\text{INTER } I\ Y)) = (\forall i \in I. F \in X\ \text{guarantees}\ (Y\ i))$ "
 <proof>

lemma *shunting*: " $(X\ \text{guarantees}\ Y) = (\text{UNIV}\ \text{guarantees}\ (\neg X \cup Y))$ "

<proof>

lemma *contrapositive*: " $(X\ \text{guarantees}\ Y) = \neg Y\ \text{guarantees}\ \neg X$ "

<proof>

lemma *combining1*:

" $[| F \in V\ \text{guarantees}\ X; F \in (X \cap Y)\ \text{guarantees}\ Z |]$
 $\implies F \in (V \cap Y)\ \text{guarantees}\ Z$ "
 <proof>

lemma *combining2*:

" $[| F \in V\ \text{guarantees}\ (X \cup Y); F \in Y\ \text{guarantees}\ Z |]$
 $\implies F \in V\ \text{guarantees}\ (X \cup Z)$ "
 <proof>

lemma *all_guarantees*:

" $\forall i \in I. F \in X\ \text{guarantees}\ (Y\ i) \implies F \in X\ \text{guarantees}\ (\bigcap i \in I. Y\ i)$ "
 <proof>

lemma *ex_guarantees*:

" $\exists i \in I. F \in X \text{ guarantees } (Y \ i) \implies F \in X \text{ guarantees } (\bigcup i \in I. Y \ i)$ "
 <proof>

9.5 Guarantees: Additional Laws (by lcp)

lemma *guarantees_Join_Int*:
 "[| $F \in U \text{ guarantees } V$; $G \in X \text{ guarantees } Y$; $F \text{ ok } G$ |]
 $\implies F \sqcup G \in (U \cap X) \text{ guarantees } (V \cap Y)$ "
 <proof>

lemma *guarantees_Join_Un*:
 "[| $F \in U \text{ guarantees } V$; $G \in X \text{ guarantees } Y$; $F \text{ ok } G$ |]
 $\implies F \sqcup G \in (U \cup X) \text{ guarantees } (V \cup Y)$ "
 <proof>

lemma *guarantees_JN_INT*:
 "[| $\forall i \in I. F \ i \in X \ i \text{ guarantees } Y \ i$; $OK \ I \ F$ |]
 $\implies (JOIN \ I \ F) \in (INTER \ I \ X) \text{ guarantees } (INTER \ I \ Y)$ "
 <proof>

lemma *guarantees_JN_UN*:
 "[| $\forall i \in I. F \ i \in X \ i \text{ guarantees } Y \ i$; $OK \ I \ F$ |]
 $\implies (JOIN \ I \ F) \in (UNION \ I \ X) \text{ guarantees } (UNION \ I \ Y)$ "
 <proof>

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

lemma *guarantees_Join_I1*:
 "[| $F \in X \text{ guarantees } Y$; $F \text{ ok } G$ |] $\implies F \sqcup G \in X \text{ guarantees } Y$ "
 <proof>

lemma *guarantees_Join_I2*:
 "[| $G \in X \text{ guarantees } Y$; $F \text{ ok } G$ |] $\implies F \sqcup G \in X \text{ guarantees } Y$ "
 <proof>

lemma *guarantees_JN_I*:
 "[| $i \in I$; $F \ i \in X \ i \text{ guarantees } Y$; $OK \ I \ F$ |]
 $\implies (\bigcup i \in I. (F \ i)) \in X \text{ guarantees } Y$ "
 <proof>

lemma *Join_welldef_D1*: " $F \sqcup G \in \text{welldef} \implies F \in \text{welldef}$ "
 <proof>

lemma *Join_welldef_D2*: " $F \sqcup G \in \text{welldef} \implies G \in \text{welldef}$ "
 <proof>

lemma *refines_refl*: " $F \text{ refines } F \text{ wrt } X$ "
 <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
   ==> ( $\forall H. F \text{ ok } H \ \& \ G \text{ ok } H \ \& \ F \sqcup H \in X \rightarrow G \sqcup H \in X$ )
       = ( $F \in X \rightarrow G \in X$ )"
  <proof>

lemma strict_ex_refine_lemma_v:
  "strict_ex_prop X
   ==> ( $\forall H. F \text{ ok } H \ \& \ G \text{ ok } H \ \& \ F \sqcup H \in \text{welldef} \ \& \ F \sqcup H \in X \rightarrow G \sqcup H \in X$ ) =
       ( $F \in \text{welldef} \cap X \rightarrow G \in X$ )"
  <proof>

lemma ex_refinement_thm:
  "[| strict_ex_prop X;
      $\forall H. F \text{ ok } H \ \& \ G \text{ ok } H \ \& \ F \sqcup H \in \text{welldef} \cap X \rightarrow G \sqcup H \in \text{welldef}$  |]
   ==> ( $G \text{ refines } F \text{ wrt } X$ ) = ( $G \text{ iso\_refines } F \text{ wrt } X$ )"
  <proof>

lemma strict_uv_refine_lemma:
  "strict_uv_prop X ==>
   ( $\forall H. F \text{ ok } H \ \& \ G \text{ ok } H \ \& \ F \sqcup H \in X \rightarrow G \sqcup H \in X$ ) = ( $F \in X \rightarrow G \in X$ )"
  <proof>

lemma strict_uv_refine_lemma_v:
  "strict_uv_prop X
   ==> ( $\forall H. F \text{ ok } H \ \& \ G \text{ ok } H \ \& \ F \sqcup H \in \text{welldef} \ \& \ F \sqcup H \in X \rightarrow G \sqcup H \in X$ ) =
       ( $F \in \text{welldef} \cap X \rightarrow G \in X$ )"
  <proof>

lemma uv_refinement_thm:
  "[| strict_uv_prop X;
      $\forall H. F \text{ ok } H \ \& \ G \text{ ok } H \ \& \ F \sqcup H \in \text{welldef} \cap X \rightarrow$ 
        $G \sqcup H \in \text{welldef}$  |]
   ==> ( $G \text{ refines } F \text{ wrt } X$ ) = ( $G \text{ iso\_refines } F \text{ wrt } X$ )"
  <proof>

lemma guarantees_equiv:
  "( $F \in X \text{ guarantees } Y$ ) = ( $\forall H. H \in X \rightarrow (F \text{ component\_of } H \rightarrow H \in Y)$ )"
  <proof>

lemma wg_weakest: "!!X.  $F \in (X \text{ guarantees } Y) \Rightarrow X \subseteq (\text{wg } F \ Y)$ "
  <proof>

```

lemma *wg_guarantees*: " $F \in ((\text{wg } F \ Y) \text{ guarantees } Y)$ "
 $\langle \text{proof} \rangle$

lemma *wg_equiv*: " $(H \in \text{wg } F \ X) = (F \text{ component_of } H \rightarrow H \in X)$ "
 $\langle \text{proof} \rangle$

lemma *component_of_wg*: " $F \text{ component_of } H \Rightarrow (H \in \text{wg } F \ X) = (H \in X)$ "
 $\langle \text{proof} \rangle$

lemma *wg_finite*:
 $\text{"}\forall FF. \text{ finite } FF \ \& \ FF \cap X \neq \{\} \rightarrow OK \ FF \ (\%F. \ F)$
 $\rightarrow (\forall F \in FF. ((\bigsqcup F \in FF. \ F): \text{wg } F \ X) = ((\bigsqcup F \in FF. \ F):X))\text{"}$
 $\langle \text{proof} \rangle$

lemma *wg_ex_prop*: " $\text{ex_prop } X \Rightarrow (F \in X) = (\forall H. H \in \text{wg } F \ X)$ "
 $\langle \text{proof} \rangle$

lemma *wx_subset*: " $(\text{wx } X) \leq X$ "
 $\langle \text{proof} \rangle$

lemma *wx_ex_prop*: " $\text{ex_prop } (\text{wx } X)$ "
 $\langle \text{proof} \rangle$

lemma *wx_weakest*: " $\forall Z. Z \leq X \rightarrow \text{ex_prop } Z \rightarrow Z \subseteq \text{wx } X$ "
 $\langle \text{proof} \rangle$

lemma *wx'_ex_prop*: " $\text{ex_prop}(\{F. \ \forall G. \ F \text{ ok } G \rightarrow F \sqcup G \in X\})$ "
 $\langle \text{proof} \rangle$

Equivalence with the other definition of wx

lemma *wx_equiv*: " $\text{wx } X = \{F. \ \forall G. \ F \text{ ok } G \rightarrow (F \sqcup G) \in X\}$ "
 $\langle \text{proof} \rangle$

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*: " $(X \text{ guarantees } Y) = \text{wx}(-X \cup Y)$ "
 $\langle \text{proof} \rangle$

lemma *stable_guarantees_Always*:
 $\text{"Init } F \subseteq A \Rightarrow F \in (\text{stable } A) \text{ guarantees } (\text{Always } A)\text{"}$
 $\langle \text{proof} \rangle$

lemma *constrains_guarantees_leadsTo*:
 $\text{"}F \in \text{transient } A \Rightarrow F \in (A \text{ co } A \cup B) \text{ guarantees } (A \text{ leadsTo } (B-A))\text{"}$
 $\langle \text{proof} \rangle$

end

10 Extending State Sets

theory Extend imports Guar begin

definition

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

definition

```
good_map :: "[ 'a*'b => 'c] => bool"
where "good_map h <-> surj h & ( $\forall$  x y. fst (inv h (h (x,y))) = x)"
```

definition

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

definition

```
project_set :: "[ 'a*'b => 'c, 'c set] => 'a set"
where "project_set h C = {x.  $\exists$  y. h(x,y)  $\in$  C}"
```

definition

```
extend_act :: "[ 'a*'b => 'c, ('a*'a) set] => ('c*'c) set"
where "extend_act h = (%act.  $\bigcup$  (s,s')  $\in$  act.  $\bigcup$  y. {h(s,y), h(s',y)}))"
```

definition

```
project_act :: "[ 'a*'b => 'c, ('c*'c) set] => ('a*'a) set"
where "project_act h act = {(x,x').  $\exists$  y y'. (h(x,y), h(x',y'))  $\in$  act}"
```

definition

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

definition

```
project :: "[ 'a*'b => 'c, 'c set, 'c program] => 'a program"
where "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]: " $((x,y): \text{Restrict } A \ r) = ((x,y): r \ \& \ x \in A)$ "
 <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]: " $(\text{Restrict } A \ r) \text{ `` } B = r \text{ `` } (A \cap B)$ "
 <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"

$\langle proof \rangle$

10.2 Trivial properties of f, g, h

lemma (in Extend) f_h_eq [simp]: " $f(h(x,y)) = x$ "
 $\langle proof \rangle$

lemma (in Extend) $h_inject1$ [dest]: " $h(x,y) = h(x',y') \implies x=x'$ "
 $\langle proof \rangle$

lemma (in Extend) $h_f_g_equiv$: " $h(f\ z, g\ z) == z$ "
 $\langle proof \rangle$

lemma (in Extend) $h_f_g_eq$: " $h(f\ z, g\ z) = z$ "
 $\langle proof \rangle$

lemma (in Extend) $split_extended_all$:
 " $(\forall z. PROP\ P\ z) == (\forall u\ y. PROP\ P\ (h\ (u, y)))$ "
 $\langle proof \rangle$

10.3 $extend_set$: basic properties

lemma $project_set_iff$ [iff]:
 " $(x \in project_set\ h\ C) = (\exists y. h(x,y) \in C)$ "
 $\langle proof \rangle$

lemma $extend_set_mono$: " $A \subseteq B \implies extend_set\ h\ A \subseteq extend_set\ h\ B$ "
 $\langle proof \rangle$

lemma (in Extend) $mem_extend_set_iff$ [iff]: " $z \in extend_set\ h\ A = (f\ z \in A)$ "
 $\langle proof \rangle$

lemma (in Extend) $extend_set_strict_mono$ [iff]:
 " $(extend_set\ h\ A \subseteq extend_set\ h\ B) = (A \subseteq B)$ "
 $\langle proof \rangle$

lemma $extend_set_empty$ [simp]: " $extend_set\ h\ \{\} = \{\}$ "
 $\langle proof \rangle$

lemma (in Extend) $extend_set_eq_Collect$: " $extend_set\ h\ \{s. P\ s\} = \{s. P(f\ s)\}$ "
 $\langle proof \rangle$

lemma (in Extend) $extend_set_sing$: " $extend_set\ h\ \{x\} = \{s. f\ s = x\}$ "
 $\langle proof \rangle$

lemma (in Extend) $extend_set_inverse$ [simp]:
 " $project_set\ h\ (extend_set\ h\ C) = C$ "
 $\langle proof \rangle$

lemma (in Extend) $extend_set_project_set$:
 " $C \subseteq extend_set\ h\ (project_set\ h\ C)$ "

$\langle proof \rangle$

lemma (in Extend) inj_extend_set: "inj (extend_set h)"
 $\langle proof \rangle$

lemma (in Extend) extend_set_UNIV_eq [simp]: "extend_set h UNIV = UNIV"
 $\langle proof \rangle$

10.4 project_set: basic properties

lemma (in Extend) project_set_eq: "project_set h C = f ` C"
 $\langle proof \rangle$

lemma (in Extend) project_set_I: "!!z. z ∈ C ==> f z ∈ project_set h C"
 $\langle proof \rangle$

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)"
 $\langle proof \rangle$

lemma (in Extend) project_set_extend_set_Un:
 "project_set h ((extend_set h A) ∪ B) = A ∪ (project_set h B)"
 $\langle proof \rangle$

lemma project_set_Int_subset:
 "project_set h (A ∩ B) ⊆ (project_set h A) ∩ (project_set h B)"
 $\langle proof \rangle$

lemma (in Extend) extend_set_Un_distrib:
 "extend_set h (A ∪ B) = extend_set h A ∪ extend_set h B"
 $\langle proof \rangle$

lemma (in Extend) extend_set_Int_distrib:
 "extend_set h (A ∩ B) = extend_set h A ∩ extend_set h B"
 $\langle proof \rangle$

lemma (in Extend) extend_set_INT_distrib:
 "extend_set h (INTER A B) = (⋂ x ∈ A. extend_set h (B x))"
 $\langle proof \rangle$

lemma (in Extend) extend_set_Diff_distrib:
 "extend_set h (A - B) = extend_set h A - extend_set h B"
 $\langle proof \rangle$

lemma (in Extend) extend_set_Union:
 "extend_set h (Union A) = (⋃ X ∈ A. extend_set h X)"
 $\langle proof \rangle$

lemma (in Extend) extend_set_subset_Compl_eq:
 "(extend_set h A ⊆ - extend_set h B) = (A ⊆ - B)"

<proof>

10.6 extend_act

lemma (in Extend) mem_extend_act_iff [iff]:
 "((h(s,y), h(s',y)) ∈ extend_act h act) = ((s, s') ∈ act)"
<proof>

lemma (in Extend) extend_act_D:
 "(z, z') ∈ extend_act h act ==> (f z, f z') ∈ 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:
```

"[$D \subseteq C$; $\text{project } h \ C \ F \in \text{stable } A$] $\implies \text{project } h \ D \ F \in \text{stable } A$ "
 <proof>

lemma (in Extend) project_constrains:
 "(project $h \ C \ F \in A$ co B) =
 ($F \in (C \cap \text{extend_set } h \ A)$ co ($\text{extend_set } h \ B$) & $A \subseteq B$)"
 <proof>

lemma (in Extend) project_stable:
 "(project $h \ \text{UNIV} \ F \in \text{stable } A$) = ($F \in \text{stable } (\text{extend_set } h \ A)$)"
 <proof>

lemma (in Extend) project_stable_I:
 " $F \in \text{stable } (\text{extend_set } h \ A) \implies \text{project } h \ C \ F \in \text{stable } A$ "
 <proof>

lemma (in Extend) Int_extend_set_lemma:
 " $A \cap \text{extend_set } h \ ((\text{project_set } h \ A) \cap B) = A \cap \text{extend_set } h \ B$ "
 <proof>

lemma project_constrains_project_set:
 " $G \in C$ co $B \implies \text{project } h \ C \ G \in \text{project_set } h \ C$ co $\text{project_set } h \ B$ "
 <proof>

lemma project_stable_project_set:
 " $G \in \text{stable } C \implies \text{project } h \ C \ G \in \text{stable } (\text{project_set } h \ C)$ "
 <proof>

10.10 Progress: transient, ensures

lemma (in Extend) extend_transient:
 "(extend $h \ F \in \text{transient } (\text{extend_set } h \ A)$) = ($F \in \text{transient } A$)"
 <proof>

lemma (in Extend) extend Ensures:
 "(extend $h \ F \in (\text{extend_set } h \ A)$ ensures ($\text{extend_set } h \ B$)) =
 ($F \in A$ ensures B)"
 <proof>

lemma (in Extend) leadsTo_imp_extend_leadsTo:
 " $F \in A$ leadsTo B
 $\implies \text{extend } h \ F \in (\text{extend_set } h \ A)$ leadsTo ($\text{extend_set } h \ B$)"
 <proof>

10.11 Proving the converse takes some doing!

lemma (in Extend) slice_iff [iff]: " $(x \in \text{slice } C \ y) = (h(x,y) \in C)$ "
 <proof>

lemma (in Extend) slice_Union: " $\text{slice } (\text{Union } S) \ y = (\bigcup x \in S. \text{slice } x \ y)$ "
 <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 = ($\bigcup y$. slice A y)"
 <proof>

lemma (in Extend) extend_transient_slice:
 "extend h F \in transient A \implies F \in transient (slice A y)"
 <proof>

lemma (in Extend) extend_constrains_slice:
 "extend h F \in A co B \implies F \in (slice A y) co (slice B y)"
 <proof>

lemma (in Extend) extend_sures_slice:
 "extend h F \in A ensures B \implies F \in (slice A y) ensures (project_set h B)"
 <proof>

lemma (in Extend) leadsTo_slice_project_set:
 " $\forall y$. F \in (slice B y) leadsTo CU \implies F \in (project_set h B) leadsTo CU"
 <proof>

lemma (in Extend) extend_leadsTo_slice [rule_format]:
 "extend h F \in AU leadsTo BU
 $\implies \forall y$. F \in (slice AU y) leadsTo (project_set h BU)"
 <proof>

lemma (in Extend) extend_leadsTo:
 "(extend h F \in (extend_set h A) leadsTo (extend_set h B)) =
 (F \in A leadsTo B)"
 <proof>

lemma (in Extend) extend_LeadsTo:
 "(extend h F \in (extend_set h A) LeadsTo (extend_set h B)) =
 (F \in A LeadsTo B)"
 <proof>

10.12 preserves

lemma (in Extend) project_preserves_I:
 "G \in preserves (v o f) \implies project h C G \in preserves v"
 <proof>

lemma (in Extend) project_preserves_id_I:
 "G \in preserves f \implies project h C G \in preserves id"
 <proof>

lemma (in Extend) extend_preserves:
 "(extend h G \in preserves (v o f)) = (G \in 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
```

```
definition rename :: "[ 'a => 'b, 'a program] => 'b program" where
  "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 (λ(x,u::'c). h x))"
<proof>

lemma bij_project: "bij h ==> bij (project (λ(x,u::'c). h x) UNIV)"
<proof>

lemma inv_project_eq:
 "bij h
 ==> inv (project (λ(x,u::'c). h x) UNIV) = extend (λ(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) = (λi ∈ I. rename h (F i))"
<proof>

11.3 Strong Safety: co, stable

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_invariant:
 "bij h ==> (rename h F ∈ invariant (h'A)) = (F ∈ invariant A)"
 <proof>

lemma rename_increasing:
 "bij h ==> (rename h F ∈ increasing func) = (F ∈ 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

```

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

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

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

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

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

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

definition sub :: "[ 'a, 'a=>'b] => 'b" where
  "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"*

i B"
 $\langle \text{proof} \rangle$

12.4 The Lattice Operations

lemma *bij_lift [iff]: "bij (lift i)"*
 $\langle \text{proof} \rangle$

lemma *lift_SKIP [simp]: "lift i SKIP = SKIP"*
 $\langle \text{proof} \rangle$

lemma *lift_Join [simp]: "lift i (F Join G) = lift i F Join lift i G"*
 $\langle \text{proof} \rangle$

lemma *lift_JN [simp]: "lift j (JOIN I F) = ($\bigsqcup_{i \in I}$ lift j (F i))"*
 $\langle \text{proof} \rangle$

12.5 Safety: constrains, stable, invariant

lemma *lift_constrains:*
 $"(\text{lift } i \text{ } F \in (\text{lift_set } i \text{ } A) \text{ co } (\text{lift_set } i \text{ } B)) = (F \in A \text{ co } B)"$
 $\langle \text{proof} \rangle$

lemma *lift_stable:*
 $"(\text{lift } i \text{ } F \in \text{stable } (\text{lift_set } i \text{ } A)) = (F \in \text{stable } A)"$
 $\langle \text{proof} \rangle$

lemma *lift_invariant:*
 $"(\text{lift } i \text{ } F \in \text{invariant } (\text{lift_set } i \text{ } A)) = (F \in \text{invariant } A)"$
 $\langle \text{proof} \rangle$

lemma *lift_Constrains:*
 $"(\text{lift } i \text{ } F \in (\text{lift_set } i \text{ } A) \text{ Co } (\text{lift_set } i \text{ } B)) = (F \in A \text{ Co } B)"$
 $\langle \text{proof} \rangle$

lemma *lift_Stable:*
 $"(\text{lift } i \text{ } F \in \text{Stable } (\text{lift_set } i \text{ } A)) = (F \in \text{Stable } A)"$
 $\langle \text{proof} \rangle$

lemma *lift_Always:*
 $"(\text{lift } i \text{ } F \in \text{Always } (\text{lift_set } i \text{ } A)) = (F \in \text{Always } A)"$
 $\langle \text{proof} \rangle$

12.6 Progress: transient, ensures

lemma *lift_transient:*
 $"(\text{lift } i \text{ } F \in \text{transient } (\text{lift_set } i \text{ } A)) = (F \in \text{transient } A)"$
 $\langle \text{proof} \rangle$

lemma *lift Ensures:*
 $"(\text{lift } i \text{ } F \in (\text{lift_set } i \text{ } A) \text{ ensures } (\text{lift_set } i \text{ } B)) =$
 $(F \in A \text{ ensures } B)"$
 $\langle \text{proof} \rangle$

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

```

definition PLam :: "[nat set, nat => ('b * ((nat=>'b) * 'c)) program]
  => ((nat=>'b) * 'c) program" where
  "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" == "CONST 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: " $((\text{PLam } I \text{ } F) \leq H) = (\forall i \in I. \text{lift } i \text{ } (F \text{ } i) \leq H)$ "
 <proof>

lemma component_PLam: " $i \in I \implies \text{lift } i \text{ } (F \text{ } i) \leq (\text{PLam } I \text{ } F)$ "
 <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 = ($\exists i \in I. \text{lift } i \text{ } (F \text{ } i) \in \text{transient } A$)"
 <proof>

This holds because the $F \text{ } j$ cannot change $\text{lift_set } i$

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)"

instantiation list :: (type) ord
begin

definition
  prefix_def:      "xs <= zs ⟷ (xs, zs) : genPrefix Id"

definition
  strict_prefix_def: "xs < zs ⟷ xs ≤ zs ∧ ¬ zs ≤ (xs :: 'a list)"

instance ⟨proof⟩

end

definition Le :: "(nat*nat) set" where
  "Le == {(x,y). x <= y}"

definition Ge :: "(nat*nat) set" where
  "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: "refl r ==> refl (genPrefix r)"
  <proof>

```

```

lemma genPrefix_refl [simp]: "refl 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_0 [rule_format]:
  "(x, y) : genPrefix r
   ==> ALL z. (y,z) : genPrefix s --> (x, z) : genPrefix (r 0 s)"
  <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]:
  "refl 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:
  "[| refl r; (xs,ys) : genPrefix r |]
   ==> (xs@zs, take (length xs) ys @ zs) : genPrefix r"
⟨proof⟩
```

```
lemma genPrefix_append_both:
  "[| refl 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; refl 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; refl 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 refl_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_not_le: "!!xs::'a list. (xs < zs) = (xs <= zs & ¬ zs
  ≤ xs)"
⟨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 refl_Le [iff]: "refl 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 refl_Ge [iff]: "refl 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 (Finite) multisets

```

theory Multiset
imports Main
begin

```

14.1 The type of multisets

```
typedef 'a multiset = "{f :: 'a => nat. finite {x. f x > 0}}"
  morphisms count Abs_multiset
⟨proof⟩
```

```
lemmas multiset_typedef = Abs_multiset_inverse count_inverse count
```

```
abbreviation Melem :: "'a => 'a multiset => bool"  ("(_/ :# _)" [50, 51] 50)
where
  "a :# M == 0 < count M a"
```

```
notation (xsymbols)
  Melem (infix "∈#" 50)
```

```
lemma multiset_ext_iff:
  "M = N ⟷ (∀ a. count M a = count N a)"
⟨proof⟩
```

```
lemma multiset_ext:
  "(⋀ x. count A x = count B x) ⟹ A = B"
⟨proof⟩
```

Preservation of the representing set *multiset*.

```
lemma const0_in_multiset:
  "(λ a. 0) ∈ multiset"
⟨proof⟩
```

```
lemma only1_in_multiset:
  "(λ b. if b = a then n else 0) ∈ multiset"
⟨proof⟩
```

```
lemma union_preserves_multiset:
  "M ∈ multiset ⟹ N ∈ multiset ⟹ (λ a. M a + N a) ∈ multiset"
⟨proof⟩
```

```
lemma diff_preserves_multiset:
  assumes "M ∈ multiset"
  shows "(λ a. M a - N a) ∈ multiset"
⟨proof⟩
```

```
lemma MCollect_preserves_multiset:
  assumes "M ∈ multiset"
  shows "(λ x. if P x then M x else 0) ∈ multiset"
⟨proof⟩
```

```
lemmas in_multiset = const0_in_multiset only1_in_multiset
  union_preserves_multiset diff_preserves_multiset MCollect_preserves_multiset
```

14.2 Representing multisets

Multiset comprehension

```
definition MCollect :: "'a multiset => ('a => bool) => 'a multiset" where
```

```

    "MCollect M P = Abs_multiset (λx. if P x then count M x else 0)"

syntax
  "_MCollect" :: "pttrn => 'a multiset => bool => 'a multiset"      ("(1{# _
: # _./ _#})")
translations
  "{#x :# M. P#}" == "CONST MCollect M (λx. P)"

Multiset enumeration

instantiation multiset :: (type) "{zero, plus}"
begin

definition Mempty_def:
  "0 = Abs_multiset (λa. 0)"

abbreviation Mempty :: "'a multiset" ("{#}") where
  "Mempty ≡ 0"

definition union_def:
  "M + N = Abs_multiset (λa. count M a + count N a)"

instance <proof>

end

definition single :: "'a => 'a multiset" where
  "single a = Abs_multiset (λb. if b = a then 1 else 0)"

syntax
  "_multiset" :: "args => 'a multiset"      ("{#(_)#}")
translations
  "{#x, xs#}" == "{#x#} + {#xs#}"
  "{#x#}" == "CONST single x"

lemma count_empty [simp]: "count {#} a = 0"
  <proof>

lemma count_single [simp]: "count {#b#} a = (if b = a then 1 else 0)"
  <proof>

```

14.3 Basic operations

14.3.1 Union

```

lemma count_union [simp]: "count (M + N) a = count M a + count N a"
  <proof>

```

```

instance multiset :: (type) cancel_comm_monoid_add <proof>

```

14.3.2 Difference

```

instantiation multiset :: (type) minus
begin

```

```

definition diff_def:

```

```

" M - N = Abs_multiset (λa. count M a - count N a)"

instance ⟨proof⟩

end

lemma count_diff [simp]: "count (M - N) a = count M a - count N a"
  ⟨proof⟩

lemma diff_empty [simp]: "M - {#} = M ∧ {#} - M = {#}"
  ⟨proof⟩

lemma diff_cancel [simp]: "A - A = {#}"
  ⟨proof⟩

lemma diff_union_cancelR [simp]: "M + N - N = (M::'a multiset)"
  ⟨proof⟩

lemma diff_union_cancelL [simp]: "N + M - N = (M::'a multiset)"
  ⟨proof⟩

lemma insert_DiffM:
  "x ∈# M ⇒ {#x#} + (M - {#x#}) = M"
  ⟨proof⟩

lemma insert_DiffM2 [simp]:
  "x ∈# M ⇒ M - {#x#} + {#x#} = M"
  ⟨proof⟩

lemma diff_right_commute:
  "(M::'a multiset) - N - Q = M - Q - N"
  ⟨proof⟩

lemma diff_add:
  "(M::'a multiset) - (N + Q) = M - N - Q"
  ⟨proof⟩

lemma diff_union_swap:
  "a ≠ b ⇒ M - {#a#} + {#b#} = M + {#b#} - {#a#}"
  ⟨proof⟩

lemma diff_union_single_conv:
  "a ∈# J ⇒ I + J - {#a#} = I + (J - {#a#})"
  ⟨proof⟩

```

14.3.3 Equality of multisets

```

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 = {#}"

```

$\langle proof \rangle$

lemma `empty_eq_union [iff]: "{#} = M + N \longleftrightarrow M = {#} \wedge N = {#}"`
 $\langle proof \rangle$

lemma `multi_self_add_other_not_self [simp]: "M = M + {#x#} \longleftrightarrow False"`
 $\langle proof \rangle$

lemma `diff_single_trivial:`
`" \neg x \in # M \implies M - {#x#} = M"`
 $\langle proof \rangle$

lemma `diff_single_eq_union:`
`"x \in # M \implies M - {#x#} = N \longleftrightarrow M = N + {#x#}"`
 $\langle proof \rangle$

lemma `union_single_eq_diff:`
`"M + {#x#} = N \implies M = N - {#x#}"`
 $\langle proof \rangle$

lemma `union_single_eq_member:`
`"M + {#x#} = N \implies x \in # N"`
 $\langle proof \rangle$

lemma `union_is_single:`
`"M + N = {#a#} \longleftrightarrow M = {#a#} \wedge N = {#} \vee M = {#} \wedge N = {#a#}" (is "?lhs`
`= ?rhs") $\langle proof \rangle$`

lemma `single_is_union:`
`"{#a#} = M + N \longleftrightarrow {#a#} = M \wedge N = {#} \vee M = {#} \wedge {#a#} = N"`
 $\langle proof \rangle$

lemma `add_eq_conv_diff:`
`"M + {#a#} = N + {#b#} \longleftrightarrow M = N \wedge a = b \vee M = N - {#a#} + {#b#} \wedge N =`
`M - {#b#} + {#a#}" (is "?lhs = ?rhs")`
 $\langle proof \rangle$

lemma `insert_noteq_member:`
`assumes BC: "B + {#b#} = C + {#c#}"`
`and bnotc: "b \neq c"`
`shows "c \in # B"`
 $\langle proof \rangle$

lemma `add_eq_conv_ex:`
`"(M + {#a#} = N + {#b#}) =`
`(M = N \wedge a = b \vee (\exists K. M = K + {#b#} \wedge N = K + {#a#}))"`
 $\langle proof \rangle$

14.3.4 Pointwise ordering induced by count

`instantiation multiset :: (type) ordered_ab_semigroup_add_imp_le`
`begin`

definition `less_eq_multiset` :: "'a multiset \Rightarrow 'a multiset \Rightarrow bool" where
`mset_le_def`: " $A \leq B \iff (\forall a. \text{count } A \ a \leq \text{count } B \ a)$ "

definition `less_multiset` :: "'a multiset \Rightarrow 'a multiset \Rightarrow bool" where
`mset_less_def`: " $(A::'a \text{ multiset}) < B \iff A \leq B \wedge A \neq B$ "

instance `<proof>`

end

lemma `mset_less_eqI`:
 $(\bigwedge x. \text{count } A \ x \leq \text{count } B \ x) \implies A \leq B$
`<proof>`

lemma `mset_le_exists_conv`:
 $(A::'a \text{ multiset}) \leq B \iff (\exists C. B = A + C)$
`<proof>`

lemma `mset_le_mono_add_right_cancel` [simp]:
 $(A::'a \text{ multiset}) + C \leq B + C \iff A \leq B$
`<proof>`

lemma `mset_le_mono_add_left_cancel` [simp]:
 $C + (A::'a \text{ multiset}) \leq C + B \iff A \leq B$
`<proof>`

lemma `mset_le_mono_add`:
 $(A::'a \text{ multiset}) \leq B \implies C \leq D \implies A + C \leq B + D$
`<proof>`

lemma `mset_le_add_left` [simp]:
 $(A::'a \text{ multiset}) \leq A + B$
`<proof>`

lemma `mset_le_add_right` [simp]:
 $B \leq (A::'a \text{ multiset}) + B$
`<proof>`

lemma `mset_le_single`:
 $a : \# B \implies \{ \#a \# \} \leq B$
`<proof>`

lemma `multiset_diff_union_assoc`:
 $C \leq B \implies (A::'a \text{ multiset}) + B - C = A + (B - C)$
`<proof>`

lemma `mset_le_multiset_union_diff_commute`:
 $B \leq A \implies (A::'a \text{ multiset}) - B + C = A + C - B$
`<proof>`

lemma `mset_lessD`: $A < B \implies x \in \# A \implies x \in \# B$
`<proof>`

lemma `mset_leD`: $A \leq B \implies x \in \# A \implies x \in \# B$

<proof>

lemma *mset_less_insertD*: " $(A + \{x\} < B) \implies (x \in\# B \wedge A < B)$ "
<proof>

lemma *mset_le_insertD*: " $(A + \{x\} \leq B) \implies (x \in\# B \wedge A \leq B)$ "
<proof>

lemma *mset_less_of_empty[simp]*: " $A < \{\#\} \longleftrightarrow \text{False}$ "
<proof>

lemma *multi_psub_of_add_self[simp]*: " $A < A + \{x\}$ "
<proof>

lemma *multi_psub_self[simp]*: " $(A::'a \text{ multiset}) < A = \text{False}$ "
<proof>

lemma *mset_less_add_bothersides*:
 $"T + \{x\} < S + \{x\} \implies T < S"$
<proof>

lemma *mset_less_empty_nonempty*:
 $"\{\#\} < S \longleftrightarrow S \neq \{\#}"$
<proof>

lemma *mset_less_diff_self*:
 $"c \in\# B \implies B - \{c\} < B"$
<proof>

14.3.5 Intersection

instantiation *multiset* :: (type) *semilattice_inf*
begin

definition *inf_multiset* :: "'a multiset \Rightarrow 'a multiset \Rightarrow 'a multiset" **where**
 $\text{multiset_inter_def: } "inf_multiset\ A\ B = A - (A - B)"$

instance *<proof>*

end

abbreviation *multiset_inter* :: "'a multiset \Rightarrow 'a multiset \Rightarrow 'a multiset"
 $(\text{infixl } "\#\cap" 70)$ **where**
 $"multiset_inter \equiv inf"$

lemma *multiset_inter_count*:
 $"count\ (A\ \#\cap\ B)\ x = \min\ (count\ A\ x)\ (count\ B\ x)"$
<proof>

lemma *multiset_inter_single*: " $a \neq b \implies \{a\} \#\cap \{b\} = \{\#\}"$
<proof>

lemma *multiset_union_diff_commute*:
 $assumes\ "B\ \#\cap\ C = \{\#}"$

```

  shows "A + B - C = A - C + B"
  <proof>

```

14.3.6 Comprehension (filter)

```

lemma count_MCollect [simp]:
  "count {# x:#M. P x #} a = (if P a then count M a else 0)"
  <proof>

```

```

lemma MCollect_empty [simp]: "MCollect {#} P = {#}"
  <proof>

```

```

lemma MCollect_single [simp]:
  "MCollect {#x#} P = (if P x then {#x#} else {#})"
  <proof>

```

```

lemma MCollect_union [simp]:
  "MCollect (M + N) f = MCollect M f + MCollect N f"
  <proof>

```

14.3.7 Set of elements

```

definition set_of :: "'a multiset => 'a set" where
  "set_of M = {x. x :# M}"

```

```

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>

```

```

lemma set_of_MCollect [simp]: "set_of {# x:#M. P x #} = set_of M ∩ {x. P
x}"
  <proof>

```

```

lemma finite_set_of [iff]: "finite (set_of M)"
  <proof>

```

14.3.8 Size

```

instantiation multiset :: (type) size
begin

```

```

definition size_def:
  "size M = setsum (count M) (set_of M)"

```

```

instance ⟨proof⟩

end

lemma size_empty [simp]: "size {#} = 0"
⟨proof⟩

lemma size_single [simp]: "size {#b#} = 1"
⟨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 nonempty_has_size: "(S ≠ {#}) = (0 < size S)"
⟨proof⟩

lemma size_eq_Suc_imp_elem: "size M = Suc n ==> ∃ a. a :# M"
⟨proof⟩

lemma size_eq_Suc_imp_eq_union:
  assumes "size M = Suc n"
  shows "∃ a N. M = N + {#a#}"
⟨proof⟩

```

14.4 Induction and case splits

```

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 *multi_nonempty_split*: " $M \neq \{\#\} \implies \exists A \ a. \ M = A + \{\#a\# \}$ "
<proof>

lemma *multiset_cases* [*cases type, case_names empty add*]:
assumes *em*: " $M = \{\#\} \implies P$ "
assumes *add*: " $\bigwedge N \ x. \ M = N + \{\#x\# \} \implies P$ "
shows " P "
<proof>

lemma *multi_member_split*: " $x \in\# M \implies \exists A. \ M = A + \{\#x\# \}$ "
<proof>

lemma *multi_drop_mem_not_eq*: " $c \in\# B \implies B - \{\#c\# \} \neq B$ "
<proof>

lemma *multiset_partition*: " $M = \{\# \ x:\#M. \ P \ x \ \#\} + \{\# \ x:\#M. \ \neg P \ x \ \#\}$ "
<proof>

lemma *mset_less_size*: " $(A::'a \ multiset) < B \implies \text{size } A < \text{size } B$ "
<proof>

14.4.1 Strong induction and subset induction for multisets

Well-foundedness of proper subset operator:

proper multiset subset

definition

mset_less_rel :: " $('a \ multiset * 'a \ multiset) \ set$ " **where**
"mset_less_rel = {(A,B). A < B}"

lemma *multiset_add_sub_el_shuffle*:
assumes " $c \in\# B$ " **and** " $b \neq c$ "
shows " $B - \{\#c\# \} + \{\#b\# \} = B + \{\#b\# \} - \{\#c\# \}$ "
<proof>

lemma *wf_mset_less_rel*: "*wf mset_less_rel*"
<proof>

The induction rules:

lemma *full_multiset_induct* [*case_names less*]:
assumes *ih*: " $\bigwedge B. \ \forall (A::'a \ multiset). \ A < B \longrightarrow P \ A \implies P \ B$ "
shows " $P \ B$ "
<proof>

lemma *multi_subset_induct* [*consumes 2, case_names empty add*]:
assumes " $F \leq A$ "
and *empty*: " $P \ \{\#\}$ "
and *insert*: " $\bigwedge a \ F. \ a \in\# A \implies P \ F \implies P \ (F + \{\#a\# \})$ "
shows " $P \ F$ "
<proof>

14.5 Alternative representations

14.5.1 Lists

```
primrec multiset_of :: "'a list  $\Rightarrow$  'a multiset" where
  "multiset_of [] = {#}" |
  "multiset_of (a # x) = multiset_of x + {# a #}"
```

```
lemma in_multiset_in_set:
  "x  $\in$  # multiset_of xs  $\longleftrightarrow$  x  $\in$  set xs"
  <proof>
```

```
lemma count_multiset_of:
  "count (multiset_of xs) x = length (filter ( $\lambda$ y. x = y) xs)"
  <proof>
```

```
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  $\in$  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  $\in$  set x then 1 else 0))"
  <proof>
```

```
lemma multiset_of_eq_setD:
  "multiset_of xs = multiset_of ys  $\implies$  set xs = set ys"
  <proof>
```

```
lemma set_eq_iff_multiset_of_eq_distinct:
  "distinct x  $\implies$  distinct y  $\implies$ 
   (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⟩

lemma nth_mem_multiset_of: "i < length ls ⇒ (ls ! i) :# multiset_of ls"
  ⟨proof⟩

lemma multiset_of_remove1 [simp]:
  "multiset_of (remove1 a xs) = multiset_of xs - {a#}"
  ⟨proof⟩

lemma multiset_of_eq_length:
  assumes "multiset_of xs = multiset_of ys"
  shows "length xs = length ys"
  ⟨proof⟩

lemma (in linorder) multiset_of_insort [simp]:
  "multiset_of (insort x xs) = {x#} + multiset_of xs"
  ⟨proof⟩

lemma (in linorder) multiset_of_sort [simp]:
  "multiset_of (sort xs) = multiset_of xs"
  ⟨proof⟩

This lemma shows which properties suffice to show that a function  $f$  with  $f\ xs = ys$  behaves like sort.

lemma (in linorder) properties_for_sort:
  "multiset_of ys = multiset_of xs ⇒ sorted ys ⇒ sort xs = ys"
  ⟨proof⟩

lemma multiset_of_remdups_le: "multiset_of (remdups xs) ≤ multiset_of xs"
  ⟨proof⟩

lemma multiset_of_update:
  "i < length ls ⇒ multiset_of (ls[i := v]) = multiset_of ls - {ls ! i#}
  + {v#}"
  ⟨proof⟩

lemma multiset_of_swap:
  "i < length ls ⇒ j < length ls ⇒
  multiset_of (ls[j := ls ! i, i := ls ! j]) = multiset_of ls"
  ⟨proof⟩

```

14.5.2 Association lists – including rudimentary code generation

```

definition count_of :: "('a × nat) list ⇒ 'a ⇒ nat" where
  "count_of xs x = (case map_of xs x of None ⇒ 0 | Some n ⇒ n)"

```

```

lemma count_of_multiset:

```

```

"count_of xs ∈ multiset"
⟨proof⟩

lemma count_simps [simp]:
  "count_of [] = (λ_. 0)"
  "count_of ((x, n) # xs) = (λy. if x = y then n else count_of xs y)"
  ⟨proof⟩

lemma count_of_empty:
  "x ∉ fst ` set xs ⇒ count_of xs x = 0"
  ⟨proof⟩

lemma count_of_filter:
  "count_of (filter (P ∘ fst) xs) x = (if P x then count_of xs x else 0)"
  ⟨proof⟩

definition Bag :: "('a × nat) list ⇒ 'a multiset" where
  "Bag xs = Abs_multiset (count_of xs)"

code_datatype Bag

lemma count_Bag [simp, code]:
  "count (Bag xs) = count_of xs"
  ⟨proof⟩

lemma Mempty_Bag [code]:
  "{#} = Bag []"
  ⟨proof⟩

lemma single_Bag [code]:
  "{#x#} = Bag [(x, 1)]"
  ⟨proof⟩

lemma MCollect_Bag [code]:
  "MCollect (Bag xs) P = Bag (filter (P ∘ fst) xs)"
  ⟨proof⟩

lemma mset_less_eq_Bag [code]:
  "Bag xs ≤ A ⟷ (∀ (x, n) ∈ set xs. count_of xs x ≤ count A x)"
  (is "?lhs ⟷ ?rhs")
  ⟨proof⟩

instantiation multiset :: (eq) eq
begin

definition
  "HOL.eq A B ⟷ (A :: 'a multiset) ≤ B ∧ B ≤ A"

instance ⟨proof⟩

end

definition (in term_syntax)
  bagify :: "('a::typerep × nat) list × (unit ⇒ Code_Evaluation.term)"

```



```

    ⇒ 'a multiset × (unit ⇒ Code_Evaluation.term)" where
  [code_unfold]: "bagify xs = Code_Evaluation.valtermify Bag {·} xs"

notation fcomp (infixl "o>" 60)
notation scomp (infixl "o→" 60)

instantiation multiset :: (random) random
begin

definition
  "Quickcheck.random i = Quickcheck.random i o→ (λxs. Pair (bagify xs))"

instance ⟨proof⟩

end

no_notation fcomp (infixl "o>" 60)
no_notation scomp (infixl "o→" 60)

hide_const (open) bagify

```

14.6 The multiset order

14.6.1 Well-foundedness

```

definition mult1 :: "('a × 'a) set => ('a multiset × 'a multiset) set" where
  [code del]: "mult1 r = {(N, M). ∃ a MO K. M = MO + {#a#} ∧ N = MO + K ∧
    (∀ b. b :# K --> (b, a) ∈ r)}"

definition mult :: "('a × 'a) set => ('a multiset × 'a multiset) set" where
  [code del]: "mult r = (mult1 r)⁺"

lemma not_less_empty [iff]: "(M, {#}) ∉ mult1 r"
  ⟨proof⟩

lemma less_add: "(N, MO + {#a#}) ∈ mult1 r ==>
  (∃ M. (M, MO) ∈ mult1 r ∧ N = M + {#a#}) ∨
  (∃ K. (∀ b. b :# K --> (b, a) ∈ r) ∧ N = MO + 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⟩

```

14.6.2 Closure-free presentation

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 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⟩

```

14.6.3 Partial-order properties

```

definition less_multiset :: "'a::order multiset ⇒ 'a multiset ⇒ bool" (infix
"<#" 50) where
  "M' <# M ⟷ (M', M) ∈ mult {(x', x). x' < x}"

```

```

definition le_multiset :: "'a::order multiset ⇒ 'a multiset ⇒ bool" (infix
"<=#" 50) where
  "M' <=# M ⟷ M' <# M ∨ M' = M"

```

```

notation (xsymbols) less_multiset (infix "<#" 50)
notation (xsymbols) le_multiset (infix "<=#" 50)

```

```

interpretation multiset_order: order le_multiset less_multiset
⟨proof⟩

```

```

lemma mult_less_irrefl [elim!]:
  "M <# (M::'a::order multiset) ==> R"
⟨proof⟩

```

14.6.4 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⟩

```

```

interpretation multiset_order: ordered_ab_semigroup_add plus le_multiset less_multiset
⟨proof⟩

```

14.7 The fold combinator

The intended behaviour is $\text{fold_mset } f \ z \ \{\#x_1, \dots, x_n\} = f \ x_1 \ (\dots (f \ x_n \ z) \dots)$ if f is associative-commutative.

The graph of fold_mset , z : the start element, f : folding function, A : the multi-set, y : the result.

inductive

```
fold_msetG :: "('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a multiset ⇒ 'b ⇒ bool"
for f :: "'a ⇒ 'b ⇒ 'b"
and z :: 'b
```

where

```
emptyI [intro]: "fold_msetG f z {#} z"
| insertI [intro]: "fold_msetG f z A y ⇒ fold_msetG f z (A + {#x#}) (f x y)"
```

inductive_cases empty_fold_msetGE [elim!]: "fold_msetG f z {#} x"

inductive_cases insert_fold_msetGE: "fold_msetG f z (A + {#}) y"

definition

```
fold_mset :: "('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a multiset ⇒ 'b" where
"fold_mset f z A = (THE x. fold_msetG f z A x)"
```

lemma Diff1_fold_msetG:

```
"fold_msetG f z (A - {#x#}) y ⇒ x ∈# A ⇒ fold_msetG f z A (f x y)"
⟨proof⟩
```

lemma fold_msetG_nonempty: "∃ x. fold_msetG f z A x"

⟨proof⟩

lemma fold_mset_empty[simp]: "fold_mset f z {#} = z"

⟨proof⟩

context fun_left_comm

begin

lemma fold_msetG_determ:

```
"fold_msetG f z A x ⇒ fold_msetG f z A y ⇒ y = x"
⟨proof⟩
```

lemma fold_mset_insert_aux:

```
"(fold_msetG f z (A + {#x#}) v) =
  (∃ y. fold_msetG f z A y ∧ v = f x y)"
⟨proof⟩
```

lemma fold_mset_equality: "fold_msetG f z A y ⇒ fold_mset f z A = y"

⟨proof⟩

lemma fold_mset_insert:

```
"fold_mset f z (A + {#x#}) = f x (fold_mset f z A)"
⟨proof⟩
```

lemma fold_mset_insert_idem:

```
"fold_mset f z (A + {#a#}) = f a (fold_mset f z A)"
```

<proof>

lemma *fold_mset_commute*: "f x (fold_mset f z A) = fold_mset f (f x z) A"
<proof>

lemma *fold_mset_single [simp]*: "fold_mset f z {#x#} = f x z"
<proof>

lemma *fold_mset_union [simp]*:
 "fold_mset f z (A+B) = fold_mset f (fold_mset f z A) B"
<proof>

lemma *fold_mset_fusion*:
 assumes "fun_left_comm g"
 shows "($\bigwedge x y. h (g x y) = f x (h y)$) $\implies h (fold_mset g w A) = fold_mset f (h w) A$ " (is "PROP ?P")
<proof>

lemma *fold_mset_rec*:
 assumes "a \in # A"
 shows "fold_mset f z A = f a (fold_mset f z (A - {#a#}))"
<proof>

end

A note on code generation: When defining some function containing a sub-term *fold_mset F*, code generation is not automatic. When interpreting locale *left_commutative* with *F*, the would be code thms for *fold_mset* become thms like *fold_mset F z {#} = z* where *F* is not a pattern but contains defined symbols, i.e. is not a code thm. Hence a separate constant with its own code thms needs to be introduced for *F*. See the image operator below.

14.8 Image

definition *image_mset* :: "('a \Rightarrow 'b) \Rightarrow 'a multiset \Rightarrow 'b multiset" where
 "image_mset f = fold_mset (op + o single o f) {#}"

interpretation *image_left_comm*: fun_left_comm "op + o single o f"
<proof>

lemma *image_mset_empty [simp]*: "image_mset f {#} = {#}"
<proof>

lemma *image_mset_single [simp]*: "image_mset f {#x#} = {#f x#}"
<proof>

lemma *image_mset_insert*:
 "image_mset f (M + {#a#}) = image_mset f M + {#f a#}"
<proof>

lemma *image_mset_union [simp]*:
 "image_mset f (M+N) = image_mset f M + image_mset f N"
<proof>

```
lemma size_image_mset [simp]: "size (image_mset f M) = size M"
<proof>
```

```
lemma image_mset_is_empty_iff [simp]: "image_mset f M = {}  $\longleftrightarrow$  M = {}"
<proof>
```

```
syntax
```

```
"_comprehension1_mset" :: "'a  $\Rightarrow$  'b  $\Rightarrow$  'b multiset  $\Rightarrow$  'a multiset"
  ("({#_/. _ :# _#})")
```

```
translations
```

```
"{#e. x:#M#}" == "CONST image_mset (%x. e) M"
```

```
syntax
```

```
"_comprehension2_mset" :: "'a  $\Rightarrow$  'b  $\Rightarrow$  'b multiset  $\Rightarrow$  bool  $\Rightarrow$  'a multiset"
  ("({#_/. | _ :# _./ _#})")
```

```
translations
```

```
"{#e | x:#M. P#}"  $\Rightarrow$  "{#e. x :# {# x:#M. P#}#}"
```

This allows to write not just filters like $\{x : M. x < c\}$ but also images like $\{x + x. x : M\}$ and $\{x+x/x:#M. x<c\}$, where the latter is currently displayed as $\{x + x. x : \{x : M. x < c\}\}$.

14.9 Termination proofs with multiset orders

```
lemma multi_member_skip: "x  $\in$  XS  $\implies$  x  $\in$  {# y #} + XS"
  and multi_member_this: "x  $\in$  {# x #} + XS"
  and multi_member_last: "x  $\in$  {# x #}"
<proof>
```

```
definition "ms_strict = mult pair_less"
```

```
definition [code del]: "ms_weak = ms_strict  $\cup$  Id"
```

```
lemma ms_reduction_pair: "reduction_pair (ms_strict, ms_weak)"
<proof>
```

```
lemma smsI:
```

```
"(set_of A, set_of B)  $\in$  max_strict  $\implies$  (Z + A, Z + B)  $\in$  ms_strict"
<proof>
```

```
lemma wmsI:
```

```
"(set_of A, set_of B)  $\in$  max_strict  $\vee$  A = {#}  $\wedge$  B = {#}
 $\implies$  (Z + A, Z + B)  $\in$  ms_weak"
```

```
<proof>
```

```
inductive pw_leq
```

```
where
```

```
  pw_leq_empty: "pw_leq {#} {#}"
```

```
| pw_leq_step: "[ (x,y)  $\in$  pair_leq; pw_leq X Y ]  $\implies$  pw_leq ({#x#} + X) ({#y#} + Y)"
```

```
lemma pw_leq_1step:
```

```
"(x, y)  $\in$  pair_leq  $\implies$  pw_leq {#x#} {#y#}"
```

```
<proof>
```

```

lemma pw_leq_split:
  assumes "pw_leq X Y"
  shows " $\exists A B Z. X = A + Z \wedge Y = B + Z \wedge ((\text{set\_of } A, \text{set\_of } B) \in \text{max\_strict} \vee (B = \{\#\} \wedge A = \{\#\}))$ "
  <proof>

```

```

lemma
  assumes pwleq: "pw_leq Z Z'"
  shows ms_strictI: " $(\text{set\_of } A, \text{set\_of } B) \in \text{max\_strict} \implies (Z + A, Z' + B) \in \text{ms\_strict}$ "
  and ms_weakI1: " $(\text{set\_of } A, \text{set\_of } B) \in \text{max\_strict} \implies (Z + A, Z' + B) \in \text{ms\_weak}$ "
  and ms_weakI2: " $(Z + \{\#\}, Z' + \{\#\}) \in \text{ms\_weak}$ "
  <proof>

```

```

lemma empty_idemp: " $\{\#\} + x = x$ " " $x + \{\#\} = x$ "
and nonempty_plus: " $\{\# x \#\} + rs \neq \{\#\}$ "
and nonempty_single: " $\{\# x \#\} \neq \{\#\}$ "
<proof>

```

<ML>

14.10 Legacy theorem bindings

```

lemmas multi_count_eq = multiset_ext_iff [symmetric]

```

```

lemma union_commute: " $M + N = N + (M::'a \text{ multiset})$ "
  <proof>

```

```

lemma union_assoc: " $(M + N) + K = M + (N + (K::'a \text{ multiset}))$ "
  <proof>

```

```

lemma union_lcomm: " $M + (N + K) = N + (M + (K::'a \text{ multiset}))$ "
  <proof>

```

```

lemmas union_ac = union_assoc union_commute union_lcomm

```

```

lemma union_right_cancel: " $M + K = N + K \longleftrightarrow M = (N::'a \text{ multiset})$ "
  <proof>

```

```

lemma union_left_cancel: " $K + M = K + N \longleftrightarrow M = (N::'a \text{ multiset})$ "
  <proof>

```

```

lemma multi_union_self_other_eq: " $(A::'a \text{ multiset}) + X = A + Y \implies X = Y$ "
  <proof>

```

```

lemma mset_less_trans: " $(M::'a \text{ multiset}) < K \implies K < N \implies M < N$ "
  <proof>

```

```

lemma multiset_inter_commute: " $A \# \cap B = B \# \cap A$ "
  <proof>

```

```

lemma multiset_inter_assoc: " $A \# \cap (B \# \cap C) = A \# \cap B \# \cap C$ "
  <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 mult_less_not_refl:
  "¬ M ⊂# (M::'a::order multiset)"
  ⟨proof⟩

lemma mult_less_trans:
  "K ⊂# M ==> M ⊂# N ==> K ⊂# (N::'a::order multiset)"
  ⟨proof⟩

lemma mult_less_not_sym:
  "M ⊂# N ==> ¬ N ⊂# (M::'a::order multiset)"
  ⟨proof⟩

lemma mult_less_asym:
  "M ⊂# N ==> (¬ P ==> N ⊂# (M::'a::order multiset)) ==> P"
  ⟨proof⟩

⟨ML⟩

end

```

15 The Follows Relation of Charpentier and Sivilotte

```

theory Follows imports SubstAx ListOrder Multiset begin

definition Follows :: "[ 'a => 'b::{order}, 'a => 'b::{order} ] => 'a program
  set" (infixl "Fols" 65) where
  "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 \subseteq (h o f) Fols (h o g)"
<proof>

lemma *mono_Follows_apply*:
 "mono h ==> f Fols g \subseteq (%x. h (f x)) Fols (%x. h (g x))"
<proof>

lemma *Follows_trans*:
 "[| F \in f Fols g; F \in g Fols h |] ==> F \in f Fols h"
<proof>

15.1 Destruction rules

lemma *Follows_Increasing1*: "F \in f Fols g ==> F \in Increasing f"
<proof>

lemma *Follows_Increasing2*: "F \in f Fols g ==> F \in Increasing g"
<proof>

lemma *Follows_Bounded*: "F \in f Fols g ==> F \in Always {s. f s \leq g s}"
<proof>

lemma *Follows_LeadsTo*:
 "F \in f Fols g ==> F \in {s. k \leq g s} LeadsTo {s. k \leq f s}"
<proof>

lemma *Follows_LeadsTo_prefixLe*:
 "F \in f Fols g ==> F \in {s. k prefixLe g s} LeadsTo {s. k prefixLe f s}"
<proof>

lemma *Follows_LeadsTo_prefixGe*:
 "F \in f Fols g ==> F \in {s. k prefixGe g s} LeadsTo {s. k prefixGe f s}"
<proof>

lemma *Always_Follows1*:
 "[| F \in Always {s. f s = f' s}; F \in f Fols g |] ==> F \in f' Fols g"
<proof>

lemma *Always_Follows2*:
 "[| F \in Always {s. g s = g' s}; F \in f Fols g |] ==> F \in f Fols g'"
<proof>

15.2 Union properties (with the subset ordering)

lemma *increasing_Un*:
 "[| F \in increasing f; F \in increasing g |]
 ==> F \in increasing (%s. (f s) \cup (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_prefixGe:
  "F ∈ Increasing func ==> F ∈ Stable {s. h prefixGe (func s)}"
  <proof>

lemma LeadsTo_le_imp_prefixGe:
  "∀ z. F ∈ {s. z ≤ f s} LeadsTo {s. z ≤ g s}
   ==> F ∈ {s. z prefixGe f s} LeadsTo {s. z prefixGe g s}"
  <proof>

end

```

16 Predicate Transformers

theory Transformers imports Comp begin

16.1 Defining the Predicate Transformers *wp*, *awp* and *wens*

```

definition wp :: "['a*'a] set, 'a set] => 'a set" where
  — Dijkstra's weakest-precondition operator (for an individual command)
  "wp act B == - (act^-1 `` (-B))"

definition awp :: "['a program, 'a set] => 'a set" where
  — Dijkstra's weakest-precondition operator (for a program)
  "awp F B == (⋂ act ∈ Acts F. wp act B)"

definition wens :: "['a program, ('a*'a) set, 'a set] => 'a set" where
  — 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*]: " $\text{wp act \{ \} = - (Domain act)}$ "
 <proof>

The identity relation is the skip action

lemma *wp_Id* [*simp*]: " wp Id B = B "
 <proof>

lemma *wp_totalize_act*:
 " $\text{wp (totalize_act act) B = (wp act B \cap Domain act) \cup (B - Domain act)}$ "
 <proof>

lemma *awp_subset*: " $\text{awp F A} \subseteq \text{A}$ "
 <proof>

lemma *awp_Int_eq*: " $\text{awp F (A \cap B) = awp F A} \cap \text{awp F B}$ "
 <proof>

The fundamental theorem for *awp*

theorem *awp_iff_constrains*: " $\text{A} \leq \text{awp F B} = (\text{F} \in \text{A co B})$ "
 <proof>

lemma *awp_iff_stable*: " $\text{A} \subseteq \text{awp F A} = (\text{F} \in \text{stable A})$ "
 <proof>

lemma *stable_imp_awp_ident*: " $\text{F} \in \text{stable A} \implies \text{awp F A} = \text{A}$ "
 <proof>

lemma *wp_mono*: " $\text{A} \subseteq \text{B} \implies \text{wp act A} \subseteq \text{wp act B}$ "
 <proof>

lemma *awp_mono*: " $\text{A} \subseteq \text{B} \implies \text{awp F A} \subseteq \text{awp F B}$ "
 <proof>

lemma *wens_unfold*:
 " $\text{wens F act B} = (\text{wp act B} \cap \text{awp F (B} \cup \text{wens F act B)}) \cup \text{B}$ "
 <proof>

lemma *wens_Id* [*simp*]: " $\text{wens F Id B} = \text{B}$ "
 <proof>

These two theorems justify the claim that *wens* returns the weakest assertion satisfying the ensures property

lemma *ensures_imp_wens*: " $\text{F} \in \text{A ensures B} \implies \exists \text{act} \in \text{Acts F. A} \subseteq \text{wens F act B}$ "
 <proof>

lemma *wens_ensures*: " $\text{act} \in \text{Acts F} \implies \text{F} \in (\text{wens F act B}) \text{ ensures B}$ "
 <proof>

These two results constitute assertion (4.13) of the thesis

lemma *wens_mono*: " $\text{A} \subseteq \text{B} \implies \text{wens F act A} \subseteq \text{wens F act B}$ "
 <proof>

lemma *wens_weakening*: " $B \subseteq \text{wens } F \text{ act } B$ "
 $\langle \text{proof} \rangle$

Assertion (6), or 4.16 in the thesis

lemma *subset_wens*: " $A-B \subseteq \text{wp act } B \cap \text{awp } F (B \cup A) \implies A \subseteq \text{wens } F \text{ act } B$ "
 $\langle \text{proof} \rangle$

Assertion 4.17 in the thesis

lemma *Diff_wens_constrains*: " $F \in (\text{wens } F \text{ act } A - A) \text{ co wens } F \text{ act } A$ "
 $\langle \text{proof} \rangle$

Assertion (7): 4.18 in the thesis. NOTE that many of these results hold for an arbitrary action. We often do not require $\text{act} \in \text{Acts } F$

lemma *stable_wens*: " $F \in \text{stable } A \implies F \in \text{stable } (\text{wens } F \text{ act } A)$ "
 $\langle \text{proof} \rangle$

Assertion 4.20 in the thesis.

lemma *wens_Int_eq_lemma*:
 $\text{"[|T-B} \subseteq \text{awp } F \text{ T; act} \in \text{Acts } F\text{|]} \implies T \cap \text{wens } F \text{ act } B \subseteq \text{wens } F \text{ act } (T \cap B)\text{"}$
 $\langle \text{proof} \rangle$

Assertion (8): 4.21 in the thesis. Here we indeed require $\text{act} \in \text{Acts } F$

lemma *wens_Int_eq*:
 $\text{"[|T-B} \subseteq \text{awp } F \text{ T; act} \in \text{Acts } F\text{|]} \implies T \cap \text{wens } F \text{ act } B = T \cap \text{wens } F \text{ act } (T \cap B)\text{"}$
 $\langle \text{proof} \rangle$

16.2 Defining the Weakest Ensures Set

inductive_set

wens_set :: " $'a \text{ program}, 'a \text{ set} \Rightarrow 'a \text{ set set}$ "
 for $F :: "'a \text{ program}"$ and $B :: "'a \text{ set}"$
where

Basis: " $B \in \text{wens_set } F \text{ B}$ "

| Wens: " $\text{"[|X} \in \text{wens_set } F \text{ B; act} \in \text{Acts } F\text{|]} \implies \text{wens } F \text{ act } X \in \text{wens_set } F \text{ B}"$

| Union: " $W \neq \{\} \implies \forall U \in W. U \in \text{wens_set } F \text{ B} \implies \bigcup W \in \text{wens_set } F \text{ B}$ "

lemma *wens_set_imp_co*: " $A \in \text{wens_set } F \text{ B} \implies F \in (A-B) \text{ co } A$ "
 $\langle \text{proof} \rangle$

lemma *wens_set_imp_leadsTo*: " $A \in \text{wens_set } F \text{ B} \implies F \in A \text{ leadsTo } B$ "
 $\langle \text{proof} \rangle$

lemma *leadsTo_imp_wens_set*: " $F \in A \text{ leadsTo } B \implies \exists C \in \text{wens_set } F \text{ B}. A \subseteq C$ "
 $\langle \text{proof} \rangle$

Assertion (9): 4.27 in the thesis.

lemma *leadsTo_iff_wens_set*: " $(F \in A \text{ leadsTo } B) = (\exists C \in \text{wens_set } F \ B. A \subseteq C)$ "
 <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 $\{\}$ $\in \text{wens_set } F \ B$.

lemma *wens_set_imp_subset*: " $A \in \text{wens_set } F \ B \implies B \subseteq A$ "
 <proof>

16.3 Properties Involving Program Union

Assertion (4.30) of thesis, reoriented

lemma *awp_Join_eq*: " $\text{awp } (F \sqcup G) \ B = \text{awp } F \ B \cap \text{awp } G \ B$ "
 <proof>

lemma *wens_subset*: " $\text{wens } F \ \text{act } B - B \subseteq \text{wp } \text{act } B \cap \text{awp } F \ (B \cup \text{wens } F \ \text{act } B)$ "
 <proof>

Assertion (4.31)

lemma *subset_wens_Join*:
 " $[A = T \cap \text{wens } F \ \text{act } B; \ T - B \subseteq \text{awp } F \ T; \ A - B \subseteq \text{awp } G \ (A \cup B)] \implies A \subseteq \text{wens } (F \sqcup G) \ \text{act } B$ "
 <proof>

Assertion (4.32)

lemma *wens_Join_subset*: " $\text{wens } (F \sqcup G) \ \text{act } B \subseteq \text{wens } F \ \text{act } B$ "
 <proof>

Lemma, because the inductive step is just too messy.

lemma *wens_Union_inductive_step*:
assumes *awpF*: " $T - B \subseteq \text{awp } F \ T$ "
and *awpG*: " $\forall X. X \in \text{wens_set } F \ B \implies (T \cap X) - B \subseteq \text{awp } G \ (T \cap X)$ "
shows " $[X \in \text{wens_set } F \ B; \ \text{act} \in \text{Acts } F; \ Y \subseteq X; \ T \cap X = T \cap Y] \implies \text{wens } (F \sqcup G) \ \text{act } Y \subseteq \text{wens } F \ \text{act } X \wedge T \cap \text{wens } F \ \text{act } X = T \cap \text{wens } (F \sqcup G) \ \text{act } Y$ "
 <proof>

theorem *wens_Union*:
assumes *awpF*: " $T - B \subseteq \text{awp } F \ T$ "
and *awpG*: " $\forall X. X \in \text{wens_set } F \ B \implies (T \cap X) - B \subseteq \text{awp } G \ (T \cap X)$ "
and *major*: " $X \in \text{wens_set } F \ B$ "
shows " $\exists Y \in \text{wens_set } (F \sqcup G) \ B. Y \subseteq X \ \& \ T \cap X = T \cap Y$ "
 <proof>

theorem *leadsTo_Join*:
assumes *leadsTo*: " $F \in A \text{ leadsTo } B$ "
and *awpF*: " $T - B \subseteq \text{awp } F \ T$ "
and *awpG*: " $\forall X. X \in \text{wens_set } F \ B \implies (T \cap X) - B \subseteq \text{awp } G \ (T \cap X)$ "
shows " $F \sqcup G \in T \cap A \text{ leadsTo } B$ "
 <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 \cap wp act B"`
`<proof>`

lemma `wp_Un_subset: "wp act A \cup wp act B \subseteq wp act (A \cup B)"`

`<proof>`

lemma `wp_Un_eq: "single_valued act ==> wp act (A \cup B) = wp act A \cup wp act B"`

`<proof>`

lemma `wp_UN_subset: "($\bigcup_{i \in I} \text{wp act } (A\ i)$) \subseteq wp act ($\bigcup_{i \in I} A\ i$)"`

`<proof>`

lemma `wp_UN_eq:`

`"[/single_valued act; I \neq {}]/`
`==> wp act ($\bigcup_{i \in I} A\ i$) = ($\bigcup_{i \in I} \text{wp act } (A\ i)$)"`
`<proof>`

lemma `wens_single_eq:`

`"wens (mk_program (init, {act}, allowed)) act B = B \cup wp act B"`
`<proof>`

Next, we express the `wens_set` for single-assignment programs

definition `wens_single_finite :: "(['a*'a) set, 'a set, nat] => 'a set" where`

`"wens_single_finite act B k == $\bigcup_{i \in \text{atMost } k} (\text{wp act } ^i) B$ "`

definition `wens_single :: "(['a*'a) set, 'a set] => 'a set" where`

`"wens_single act B == $\bigcup_{i} (\text{wp act } ^i) B$ "`

lemma `wens_single_Un_eq:`

`"single_valued act`
`==> wens_single act B \cup wp act (wens_single act B) = wens_single act B"`
`<proof>`

lemma `atMost_nat_nonempty: "atMost (k::nat) \neq {}"`

`<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 \cup 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  $\cup$  wp act (wens_single_finite act B k)
       $\in$  range (wens_single_finite act B)"
```

<proof>

lemma *wens_single_eq_Union:*

```
"wens_single act B =  $\bigcup$  range (wens_single_finite act B)"
```

<proof>

lemma *wens_single_finite_eq_Union:*

```
"wens_single_finite act B n = ( $\bigcup$  k $\in$ atMost n. wens_single_finite act B
k)"
```

<proof>

lemma *wens_single_finite_mono:*

```
"m  $\leq$  n ==> wens_single_finite act B m  $\subseteq$  wens_single_finite act B n"
```

<proof>

lemma *wens_single_finite_subset_wens_single:*

```
"wens_single_finite act B k  $\subseteq$  wens_single act B"
```

<proof>

lemma *subset_wens_single_finite:*

```
"[|W  $\subseteq$  wens_single_finite act B ' (atMost k); single_valued act; W $\neq$ {}|]
==>  $\exists$  m.  $\bigcup$  W = wens_single_finite act B m"
```

<proof>

lemma for Union case

lemma *Union_eq_wens_single:*

```
"[| $\forall$  k.  $\neg$  W  $\subseteq$  wens_single_finite act B ' {...k};
  W  $\subseteq$  insert (wens_single act B)
  (range (wens_single_finite act B))|]
==>  $\bigcup$  W = wens_single act B"
```

<proof>

lemma *wens_set_subset_single:*

```
"single_valued act
==> wens_set (mk_program (init, {act}, allowed)) B  $\subseteq$ 
      insert (wens_single act B) (range (wens_single_finite act B))"
```

<proof>

```

lemma wens_single_finite_in_wens_set:
  "single_valued act  $\implies$ 
    wens_single_finite act B k
     $\in$  wens_set (mk_program (init, {act}, allowed)) B"
  <proof>

lemma single_subset_wens_set:
  "single_valued act
     $\implies$  insert (wens_single act B) (range (wens_single_finite act B))  $\subseteq$ 
      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/]
     $\implies$  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  $\subseteq$  awp F T; T-B  $\subseteq$  FP G; F  $\in$  A leadsTo B/]  $\implies$  F  $\sqcup$  G  $\in$  T  $\cap$  A leadsTo B"
  <proof>

end

```

17 Progress Sets

theory ProgressSets **imports** Transformers **begin**

17.1 Complete Lattices and the Operator c_l

```

definition lattice :: "'a set set => bool" where
  — Meier calls them closure sets, but they are just complete lattices
  "lattice L ==
    ( $\forall M. M \subseteq L \rightarrow \bigcap M \in L$ ) & ( $\forall M. M \subseteq L \rightarrow \bigcup M \in L$ )"

definition cl :: "[ 'a set set, 'a set ] => 'a set" where
  — short for "closure"
  "cl L r ==  $\bigcap \{x. x \in L \ \& \ r \subseteq x\}$ "

lemma UNIV_in_lattice: "lattice L  $\implies$  UNIV  $\in$  L"
  <proof>

lemma empty_in_lattice: "lattice L  $\implies$  {}  $\in$  L"
  <proof>

lemma Union_in_lattice: "[M  $\subseteq$  L; lattice L/]  $\implies$   $\bigcup M \in L$ "
  <proof>

```


lemma *Inter_in_lattice*: " $[M \subseteq L; \text{lattice } L] \Rightarrow \bigcap M \in L$ "
 $\langle \text{proof} \rangle$

lemma *UN_in_lattice*:
" $[/\text{lattice } L; !!i. i \in I \Rightarrow r\ i \in L] \Rightarrow (\bigcup_{i \in I} r\ i) \in L$ "
 $\langle \text{proof} \rangle$

lemma *INT_in_lattice*:
" $[/\text{lattice } L; !!i. i \in I \Rightarrow r\ i \in L] \Rightarrow (\bigcap_{i \in I} r\ i) \in L$ "
 $\langle \text{proof} \rangle$

lemma *Un_in_lattice*: " $[x \in L; y \in L; \text{lattice } L] \Rightarrow x \cup y \in L$ "
 $\langle \text{proof} \rangle$

lemma *Int_in_lattice*: " $[x \in L; y \in L; \text{lattice } L] \Rightarrow x \cap y \in L$ "
 $\langle \text{proof} \rangle$

lemma *lattice_stable*: " $\text{lattice } \{X. F \in \text{stable } X\}$ "
 $\langle \text{proof} \rangle$

The next three results state that $cl\ L\ r$ is the minimal element of L that includes r .

lemma *cl_in_lattice*: " $\text{lattice } L \Rightarrow cl\ L\ r \in L$ "
 $\langle \text{proof} \rangle$

lemma *cl_least*: " $[c \in L; r \subseteq c] \Rightarrow cl\ L\ r \subseteq c$ "
 $\langle \text{proof} \rangle$

The next three lemmas constitute assertion (4.61)

lemma *cl_mono*: " $r \subseteq r' \Rightarrow cl\ L\ r \subseteq cl\ L\ r'$ "
 $\langle \text{proof} \rangle$

lemma *subset_cl*: " $r \subseteq cl\ L\ r$ "
 $\langle \text{proof} \rangle$

A reformulation of $?r \subseteq cl\ ?L\ ?r$

lemma *clI*: " $x \in r \Rightarrow x \in cl\ L\ r$ "
 $\langle \text{proof} \rangle$

A reformulation of $[?c \in ?L; ?r \subseteq ?c] \Rightarrow cl\ ?L\ ?r \subseteq ?c$

lemma *clD*: " $[c \in cl\ L\ r; B \in L; r \subseteq B] \Rightarrow c \in B$ "
 $\langle \text{proof} \rangle$

lemma *cl_UN_subset*: " $(\bigcup_{i \in I} cl\ L\ (r\ i)) \subseteq cl\ L\ (\bigcup_{i \in I} r\ i)$ "
 $\langle \text{proof} \rangle$

lemma *cl_Un*: " $\text{lattice } L \Rightarrow cl\ L\ (r \cup s) = cl\ L\ r \cup cl\ L\ s$ "
 $\langle \text{proof} \rangle$

lemma *cl_UN*: " $\text{lattice } L \Rightarrow cl\ L\ (\bigcup_{i \in I} r\ i) = (\bigcup_{i \in I} cl\ L\ (r\ i))$ "
 $\langle \text{proof} \rangle$

lemma *cl_Int_subset*: " $\text{cl } L (r \cap s) \subseteq \text{cl } L r \cap \text{cl } L s$ "
 <proof>

lemma *cl_idem [simp]*: " $\text{cl } L (\text{cl } L r) = \text{cl } L r$ "
 <proof>

lemma *cl_ident*: " $r \in L \implies \text{cl } L r = r$ "
 <proof>

lemma *cl_empty [simp]*: " $\text{lattice } L \implies \text{cl } L \{\} = \{\}$ "
 <proof>

lemma *cl_UNIV [simp]*: " $\text{lattice } L \implies \text{cl } L \text{ UNIV} = \text{UNIV}$ "
 <proof>

Assertion (4.62)

lemma *cl_ident_iff*: " $\text{lattice } L \implies (\text{cl } L r = r) = (r \in L)$ "
 <proof>

lemma *cl_subset_in_lattice*: " $[\text{cl } L r \subseteq r; \text{lattice } L] \implies r \in L$ "
 <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*.

definition *closed* :: "[*'a program, 'a set, 'a set, 'a set set*] => bool" where
 "*closed F T B L* == $\forall M. \forall \text{act} \in \text{Acts } F. B \subseteq M \ \& \ T \cap M \in L \implies$
 $T \cap (B \cup \text{wp act } M) \in L$ "

definition *progress_set* :: "[*'a program, 'a set, 'a set*] => *'a set set set*"
 where
 "*progress_set F T B* ==
 $\{L. \text{lattice } L \ \& \ B \in L \ \& \ T \in L \ \& \ \text{closed } F T B L\}$ "

lemma *closedD*:
 " $[\text{closed } F T B L; \text{act} \in \text{Acts } F; B \subseteq M; T \cap M \in L]$
 $\implies T \cap (B \cup \text{wp act } M) \in L$ "
 <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* $\in \text{Acts } F$.

lemma *lattice_awp_lemma*:
assumes *TXC*: " $T \cap X \in C$ " — induction hypothesis in theorem below
and *BsubX*: " $B \subseteq X$ " — holds in inductive step
and *latt*: " $\text{lattice } C$ "
and *TC*: " $T \in C$ "
and *BC*: " $B \in C$ "
and *clos*: " $\text{closed } F T B C$ "
shows " $T \cap (B \cup \text{awp } F (X \cup \text{cl } C (T \cap r))) \in C$ "
 <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 $T = \text{UNIV}$ states that every progress set for the program F and set B includes the set $\text{wens_set } F B$.

```

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

```

definition relcl :: "'a set set => ('a * 'a) set" where
  — Derived relation from a lattice
  "relcl L == {(x,y). y ∈ cl L {x}}"

```

```

definition latticeof :: "('a * 'a) set => 'a set set" where
  — 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 (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 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.  $\exists s. s \in X \ \& \ (s, t) \in \text{relcl } L$ }"
<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 r; trans r|] ==> relcl (latticeof r) = r"
<proof>

```

17.4.2 Decoupling Theorems

```

definition decoupled :: "[ 'a program, 'a program ] => bool" where
  "decoupled F G ==
     $\forall \text{act} \in \text{Acts}. \forall B. G \in \text{stable } B \rightarrow G \in \text{stable } (\text{wp act } B) "$ 

```

Rao's Decoupling Theorem

```

lemma stableco: "F  $\in$  stable A ==> F  $\in$  A-B co A"
<proof>

```

```

theorem decoupling:
  assumes leadsTo: "F  $\in$  A leadsTo B"
  and Gstable: "G  $\in$  stable B"
  and dec:      "decoupled F G"
  shows "F  $\sqcup$  G  $\in$  A leadsTo B"
<proof>

```

Rao's Weak Decoupling Theorem

```

theorem weak_decoupling:
  assumes leadsTo: "F  $\in$  A leadsTo B"
  and stable: "F  $\sqcup$  G  $\in$  stable B"
  and dec:      "decoupled F (F  $\sqcup$  G)"
  shows "F  $\sqcup$  G  $\in$  A leadsTo B"
<proof>

```

The “Decoupling via G' Union Theorem”

```

theorem decoupling_via_aux:
  assumes leadsTo: "F  $\in$  A leadsTo B"
  and prog: "{X. G'  $\in$  stable X}  $\in$  progress_set F UNIV B"
  and GG':  "G  $\leq$  G'"
  — Beware! This is the converse of the refinement relation!
  shows "F  $\sqcup$  G  $\in$  A leadsTo B"
<proof>

```

17.5 Composition Theorems Based on Monotonicity and Commutativity

17.5.1 Commutativity of $\text{cl } L$ and assignment.

```

definition commutes :: "[ 'a program, 'a set, 'a set, 'a set set ] => bool"
where

```

```

"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 $\llbracket ?F \in ?A \mapsto ?B'; ?C \in \text{progress_set } ?F ?T ?B; ?F \in \text{stable } ?T; ?B \subseteq ?B'; ?B' \in ?C; \bigwedge X. X \in ?C \implies ?G \in X - ?B \text{ co } X \rrbracket \implies ?F \sqcup ?G \in ?T \cap ?A \mapsto ?B'$

```

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*

```

definition funof :: "('a*'b)set, 'a] => 'b" where
  "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 $\llbracket ?F \in ?A \mapsto ?B'; ?C \in \text{progress_set } ?F ?T ?B; ?F \in \text{stable } ?T; ?B \subseteq ?B'; ?B' \in ?C; \bigwedge X. X \in ?C \implies ?G \in X - ?B \text{ co } X \rrbracket \implies ?F \sqcup ?G \in ?T \cap ?A \mapsto ?B'$

```

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 "[| $F \in (A \cap B) \text{ co } A$; $F \in (B \cap A) \text{ co } B$ |] ==> $F \in \text{stable } (A \cap B)$ "
 $\langle \text{proof} \rangle$

lemma Collect_le_Int_equals:
 $"(\bigcap i \in \text{atMost } n. A(\text{Suc } i) \cap A i) = (\bigcap i \in \text{atMost } (\text{Suc } n). A i)"$
 $\langle \text{proof} \rangle$

lemma UN_Int_Compl_subset:
 $"(\bigcup i \in \text{lessThan } n. A i) \cap (\neg A n) \subseteq$
 $(\bigcup i \in \text{lessThan } n. (A i) \cap (\neg A (\text{Suc } i)))"$
 $\langle \text{proof} \rangle$

lemma INT_Un_Compl_subset:
 $"(\bigcap i \in \text{lessThan } n. \neg A i \cup A (\text{Suc } i)) \subseteq$
 $(\bigcap i \in \text{lessThan } n. \neg A i) \cup A n"$
 $\langle \text{proof} \rangle$

lemma INT_le_equals_Int_lemma:
 $"A 0 \cap (\neg(A n) \cap (\bigcap i \in \text{lessThan } n. \neg A i \cup A (\text{Suc } i))) = \{\}"$
 $\langle \text{proof} \rangle$

lemma INT_le_equals_Int:
 $"(\bigcap i \in \text{atMost } n. A i) =$
 $A 0 \cap (\bigcap i \in \text{lessThan } n. \neg A i \cup A(\text{Suc } i))"$
 $\langle \text{proof} \rangle$

lemma INT_le_Suc_equals_Int:
 $"(\bigcap i \in \text{atMost } (\text{Suc } n). A i) =$
 $A 0 \cap (\bigcap i \in \text{atMost } n. \neg A i \cup A(\text{Suc } i))"$
 $\langle \text{proof} \rangle$

lemma
assumes zeroprem: " $F \in (A 0 \cap A (\text{Suc } n)) \text{ co } (A 0)$ "
and allprem:
 $"!!i. i \in \text{atMost } n ==> F \in (A(\text{Suc } i) \cap A i) \text{ co } (\neg A i \cup A(\text{Suc } i))"$
shows " $F \in \text{stable } (\bigcap i \in \text{atMost } (\text{Suc } n). A i)$ "
 $\langle \text{proof} \rangle$

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"

definition common :: "nat set" where
  "common == {n. ftime n = n & gtime n = n}"

definition maxfg :: "nat => nat set" where
  "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⟩

```

```

lemma leadsTo_common_lemma:
  assumes "∀m. F ∈ {m} Co (maxfg m)"
  and "∀m ∈ lessThan n. F ∈ {m} LeadsTo (greaterThan m)"
  and "n ∈ common"
  shows "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"

```

```

definition HasTok :: "nat => state set" where
  "HasTok i == {s. token s = i}"

```

```

definition H :: "nat => state set" where

```

```

    "H i == {s. proc s i = Hungry}"

definition E :: "nat => state set" where
    "E i == {s. proc s i = Eating}"

definition T :: "nat => state set" where
    "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)  $\cap$  {.. $N$ }  $\times$  {.. $N$ }"
  and next_def:
    "next i == (Suc i) mod N"
  assumes N_positive [iff]: "0<N"
  and TR2: "F  $\in$  (T i) co (T i  $\cup$  H i)"
  and TR3: "F  $\in$  (H i) co (H i  $\cup$  E i)"
  and TR4: "F  $\in$  (H i - HasTok i) co (H i)"
  and TR5: "F  $\in$  (HasTok i) co (HasTok i  $\cup$  -(E i))"
  and TR6: "F  $\in$  (H i  $\cap$  HasTok i) leadsTo (E i)"
  and TR7: "F  $\in$  (HasTok i) leadsTo (HasTok (next i))"

```

```

lemma HasTok_partition: "[| s  $\in$  HasTok i; s  $\in$  HasTok j |] ==> i=j"
  <proof>

```

```

lemma not_E_eq: "(s  $\notin$  E i) = (s  $\in$  H i | s  $\in$  T i)"
  <proof>

```

```

lemma (in Token) token_stable: "F  $\in$  stable (-(E i)  $\cup$  (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)  $\in$  nodeOrder j) = (i  $\neq$  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  $\in$  (HasTok i) leadsTo {s. (token s, i)  $\in$  nodeOrder j}  $\cup$  HasTok j"
  <proof>

```

Chapter 4 variant, the one actually used below.

```

lemma (in Token) TR7_aux: "[| i<N; j<N; i $\neq$ j |]
  ==> F  $\in$  (HasTok i) leadsTo {s. (token s, i)  $\in$  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"

```

```

definition minSet :: "nat set => nat option" where

```

```

  "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
```

```
axiomatization
```

```
  Min :: "int" and — least and greatest floors
  Max :: "int" — least and greatest floors
```

```
where
```

```
  Min_le_Max [iff]: "Min ≤ Max"
```

— Abbreviations: the "always" part

```
definition
```

```
  above :: "state set"
  where "above = {s. ∃ i. floor s < i & i ≤ Max & i ∈ req s}"
```

```
definition
```

```
  below :: "state set"
  where "below = {s. ∃ i. Min ≤ i & i < floor s & i ∈ req s}"
```

```
definition
```

```
  queueing :: "state set"
  where "queueing = above ∪ below"
```

```
definition
```

```
  goingup :: "state set"
  where "goingup = above ∩ ({s. up s} ∪ -below)"
```

```
definition
```

```
  goingdown :: "state set"
  where "goingdown = below ∩ ({s. ~ up s} ∪ -above)"
```

```
definition
```

```
ready :: "state set"
where "ready = {s. stop s & ~ open s & move s}"
```

— Further abbreviations

definition

```
moving :: "state set"
where "moving = {s. ~ stop s & ~ open s}"
```

definition

```
stopped :: "state set"
where "stopped = {s. stop s & ~ open s & ~ move s}"
```

definition

```
opened :: "state set"
where "opened = {s. stop s & open s & move s}"
```

definition

```
closed :: "state set" — but this is the same as ready!!
where "closed = {s. stop s & ~ open s & move s}"
```

definition

```
atFloor :: "int => state set"
where "atFloor n = {s. floor s = n}"
```

definition

```
Req :: "int => state set"
where "Req n = {s. n ∈ req s}"
```

— The program

definition

```
request_act :: "(state*state) set"
where "request_act = {(s,s'). s' = s (/stop:=True, move:=False/)
& ~ stop s & floor s ∈ req s}"
```

definition

```
open_act :: "(state*state) set"
where "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)}"
```

definition

```
close_act :: "(state*state) set"
where "close_act = {(s,s'). s' = s (/open := False/) & open s}"
```

definition

```
req_up :: "(state*state) set"
where "req_up =
```

```

{(s,s'). s' = s (/stop :=False,
                  floor := floor s + 1,
                  up    := True|)
 & s ∈ (ready ∩ goingup))}"

```

definition

```

req_down :: "(state*state) set"
where "req_down =
      {(s,s'). s' = s (/stop :=False,
                      floor := floor s - 1,
                      up    := False|)
      & s ∈ (ready ∩ goingdown))}"

```

definition

```

move_up :: "(state*state) set"
where "move_up =
      {(s,s'). s' = s (/floor := floor s + 1|)
      & ~ stop s & up s & floor s ∉ req s}"

```

definition

```

move_down :: "(state*state) set"
where "move_down =
      {(s,s'). s' = s (/floor := floor s - 1|)
      & ~ stop s & ~ up s & floor s ∉ req s}"

```

definition

```

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.

```

where "button_press =
      {(s,s'). ∃ n. s' = s (/req := insert n (req s)|)
      & Min ≤ n & n ≤ Max}"

```

definition

```

Lift :: "state program"
— for the moment, we OMIT button_press
where "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

definition

```

bounded :: "state set"
where "bounded = {s. Min ≤ floor s & floor s ≤ Max}"

```


definition

```
open_stop :: "state set"
where "open_stop = {s. open s --> stop s}"
```

definition

```
open_move :: "state set"
where "open_move = {s. open s --> move s}"
```

definition

```
stop_floor :: "state set"
where "stop_floor = {s. stop s & ~ move s --> floor s ∈ req s}"
```

definition

```
moving_up :: "state set"
where "moving_up = {s. ~ stop s & up s -->
      (∃f. floor s ≤ f & f ≤ Max & f ∈ req s)}"
```

definition

```
moving_down :: "state set"
where "moving_down = {s. ~ stop s & ~ up s -->
      (∃f. Min ≤ f & f ≤ floor s & f ∈ req s)}"
```

definition

```
metric :: "[int,state] => int"
where "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  $\in$  Always open_stop"
<proof>

```

```

lemma stop_floor: "Lift  $\in$  Always stop_floor"
<proof>

```

```

lemma open_move: "Lift  $\in$  Always open_move"
<proof>

```

```

lemma moving_up: "Lift  $\in$  Always moving_up"
<proof>

```

```

lemma moving_down: "Lift  $\in$  Always moving_down"
<proof>

```

```

lemma bounded: "Lift  $\in$  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  $\in$  (stopped  $\cap$  atFloor  $n$ ) LeadsTo (opened  $\cap$  atFloor
n)"
<proof>

```

```

lemma E_thm02: "Lift  $\in$  (Req  $n \cap$  stopped - atFloor  $n$ ) LeadsTo
(Req  $n \cap$  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_OD [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  $\cap$  atFloor n)"
  <proof>

lemma (in Floor) E_thm13:
  "Lift  $\in$  (moving  $\cap$  Req n  $\cap$  {s. metric n s = N}  $\cap$  {s. floor s  $\in$  req s})

  LeadsTo (stopped  $\cap$  Req n  $\cap$  {s. metric n s = N}  $\cap$  {s. floor s  $\in$  req s})"
  <proof>

lemma (in Floor) E_thm14: "0 < N ==>
  Lift  $\in$ 
    (stopped  $\cap$  Req n  $\cap$  {s. metric n s = N}  $\cap$  {s. floor s  $\in$  req s})
    LeadsTo (opened  $\cap$  Req n  $\cap$  {s. metric n s = N})"
  <proof>

lemma (in Floor) E_thm15: "Lift  $\in$  (opened  $\cap$  Req n  $\cap$  {s. metric n s = N})

  LeadsTo (closed  $\cap$  Req n  $\cap$  {s. metric n s = N})"
  <proof>

lemma (in Floor) lift_3_Req: "0 < N ==>
  Lift  $\in$ 
    (moving  $\cap$  Req n  $\cap$  {s. metric n s = N}  $\cap$  {s. floor s  $\in$  req s})
    LeadsTo (moving  $\cap$  Req n  $\cap$  {s. metric n s < N})"
  <proof>

lemma (in Floor) Always_nonneg: "Lift  $\in$  Always {s. 0  $\leq$  metric n s}"
  <proof>

lemmas (in Floor) R_thm11 = Always_LeadsTo_weaken [OF Always_nonneg E_thm11]

lemma (in Floor) lift_3:
  "Lift  $\in$  (moving  $\cap$  Req n) LeadsTo (stopped  $\cap$  atFloor n)"
  <proof>

lemma (in Floor) lift_1: "Lift  $\in$  (Req n) LeadsTo (opened  $\cap$  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"

definition U0 :: command
  where "U0 = {(s,s'). s' = s (|u:=True, m:=1|) & m s = 0}"

definition U1 :: command
  where "U1 = {(s,s'). s' = s (|p:= v s, m:=2|) & m s = 1}"

definition U2 :: command
  where "U2 = {(s,s'). s' = s (|m:=3|) & ~ p s & m s = 2}"

definition U3 :: command
  where "U3 = {(s,s'). s' = s (|u:=False, m:=4|) & m s = 3}"

definition U4 :: command
  where "U4 = {(s,s'). s' = s (|p:=True, m:=0|) & m s = 4}"

definition V0 :: command
  where "V0 = {(s,s'). s' = s (|v:=True, n:=1|) & n s = 0}"

definition V1 :: command
  where "V1 = {(s,s'). s' = s (|p:= ~ u s, n:=2|) & n s = 1}"

definition V2 :: command
  where "V2 = {(s,s'). s' = s (|n:=3|) & p s & n s = 2}"

definition V3 :: command
  where "V3 = {(s,s'). s' = s (|v:=False, n:=4|) & n s = 3}"

definition V4 :: command
  where "V4 = {(s,s'). s' = s (|p:=False, n:=0|) & n s = 4}"

definition Mutex :: "state program"
  where "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)"

definition IU :: "state set"

```

```

where "IU = {s. (u s = (1 ≤ m s & m s ≤ 3)) & (m s = 3 --> ~ p s)}"

definition IV :: "state set"
where "IV = {s. (v s = (1 ≤ n s & n s ≤ 3)) & (n s = 3 --> p s)}"

definition bad_IU :: "state set"
where "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 $\in \{s. m\ s=1\}$ LeadsTo $\{s. p\ s = v\ s \ \& \ m\ s = 2\}$ "
 $\langle proof \rangle$

lemma U_F2: "Mutex $\in \{s. \sim p\ s \ \& \ m\ s = 2\}$ LeadsTo $\{s. m\ s = 3\}$ "
 $\langle proof \rangle$

lemma U_F3: "Mutex $\in \{s. m\ s = 3\}$ LeadsTo $\{s. p\ s\}$ "
 $\langle proof \rangle$

lemma U_lemma2: "Mutex $\in \{s. m\ s = 2\}$ LeadsTo $\{s. p\ s\}$ "
 $\langle proof \rangle$

lemma U_lemma1: "Mutex $\in \{s. m\ s = 1\}$ LeadsTo $\{s. p\ s\}$ "
 $\langle proof \rangle$

lemma U_lemma123: "Mutex $\in \{s. 1 \leq m\ s \ \& \ m\ s \leq 3\}$ LeadsTo $\{s. p\ s\}$ "
 $\langle proof \rangle$

lemma u_Leadsto_p: "Mutex $\in \{s. u\ s\}$ LeadsTo $\{s. p\ s\}$ "
 $\langle proof \rangle$

lemma V_F0: "Mutex $\in \{s. n\ s=2\}$ Unless $\{s. n\ s=3\}$ "
 $\langle proof \rangle$

lemma V_F1: "Mutex $\in \{s. n\ s=1\}$ LeadsTo $\{s. p\ s = (\sim u\ s) \ \& \ n\ s = 2\}$ "
 $\langle proof \rangle$

lemma V_F2: "Mutex $\in \{s. p\ s \ \& \ n\ s = 2\}$ LeadsTo $\{s. n\ s = 3\}$ "
 $\langle proof \rangle$

lemma V_F3: "Mutex $\in \{s. n\ s = 3\}$ LeadsTo $\{s. \sim p\ s\}$ "
 $\langle proof \rangle$

lemma V_lemma2: "Mutex $\in \{s. n\ s = 2\}$ LeadsTo $\{s. \sim p\ s\}$ "
 $\langle proof \rangle$

lemma V_lemma1: "Mutex $\in \{s. n\ s = 1\}$ LeadsTo $\{s. \sim p\ s\}$ "
 $\langle proof \rangle$

lemma V_lemma123: "Mutex $\in \{s. 1 \leq n\ s \ \& \ n\ s \leq 3\}$ LeadsTo $\{s. \sim p\ s\}$ "
 $\langle proof \rangle$

lemma v_Leadsto_not_p: "Mutex $\in \{s. v\ s\}$ LeadsTo $\{s. \sim p\ s\}$ "
 $\langle proof \rangle$


```
lemma m1_Leadsto_3: "Mutex  $\in$  {s. m s = 1} LeadsTo {s. m s = 3}"
<proof>
```

```
lemma n1_Leadsto_3: "Mutex  $\in$  {s. n s = 1} LeadsTo {s. n s = 3}"
<proof>
```

```
end
```

```
theory Reach imports "../UNITY_Main" begin
```

```
typeddecl vertex
```

```
types state = "vertex=>bool"
```

```
consts
```

```
init :: "vertex"
```

```
edges :: "(vertex*vertex) set"
```

```
definition asgt :: "[vertex,vertex] => (state*state) set"
```

```
where "asgt u v = {(s,s'). s' = s(v:= s u | s v)}"
```

```
definition Rprg :: "state program"
```

```
where "Rprg = mk_total_program ({%v. v=init},  $\bigcup_{(u,v) \in \text{edges}} \{\text{asgt } u \ v\}$ , UNIV)"
```

```
definition reach_invariant :: "state set"
```

```
where "reach_invariant = {s. ( $\forall v. s \ v \ \rightarrow (init, v) \in \text{edges}^*$ ) & s init}"
```

```
definition fixedpoint :: "state set"
```

```
where "fixedpoint = {s.  $\forall (u,v) \in \text{edges}. s \ u \ \rightarrow s \ v$ }"
```

```
definition metric :: "state => nat"
```

```
where "metric s = card {v.  $\sim s \ v$ }"
```

*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;
```

```
 [|  $\sim P$ ; 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)"

definition reachable :: "vertex => state set" where
  "reachable p == {s. reach s p}"

definition nmsg_eq :: "nat => edge => state set" where
  "nmsg_eq k == %e. {s. nmsg s e = k}"

definition nmsg_gt :: "nat => edge => state set" where
  "nmsg_gt k == %e. {s. k < nmsg s e}"

definition nmsg_gte :: "nat => edge => state set" where
  "nmsg_gte k == %e. {s. k ≤ nmsg s e}"

definition nmsg_lte :: "nat => edge => state set" where
  "nmsg_lte k == %e. {s. nmsg s e ≤ k}"

definition final :: "state set" where

```

```
"final == ( $\bigcap_{v \in V} \text{reachable } v \iff \{s. (\text{root}, v) \in \text{REACHABLE}\} \cap$ 
  (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})"
```

$\langle proof \rangle$

lemma *Eq_lemma1*:
 "(reachable v <==> {s. (root,v) ∈ REACHABLE}) =
 {s. ((s ∈ reachable v) = ((root,v) ∈ REACHABLE)))}"
 $\langle proof \rangle$

lemma *Eq_lemma2*:
 "(reachable v <==> (if (root,v) ∈ REACHABLE then UNIV else {})) =
 {s. ((s ∈ reachable v) = ((root,v) ∈ REACHABLE)))}"
 $\langle proof \rangle$

lemma *final_lemma1*:
 "(($\bigcap v \in V. \bigcap w \in V. \{s. ((s \in \text{reachable } v) = ((\text{root},v) \in \text{REACHABLE}))$)
 &
 $s \in \text{nmsg_eq } 0 \ (v,w)\}$)
 $\subseteq \text{final}$)"
 $\langle proof \rangle$

lemma *final_lemma2*:
 "E ≠ {}
 ==> (($\bigcap v \in V. \bigcap e \in E. \{s. ((s \in \text{reachable } v) = ((\text{root},v) \in \text{REACHABLE}))\}$)
 $\cap \text{nmsg_eq } 0 \ e)$ $\subseteq \text{final}$)"
 $\langle proof \rangle$

lemma *final_lemma3*:
 "E ≠ {}
 ==> (($\bigcap v \in V. \bigcap e \in E.$
 $(\text{reachable } v \text{ <==> } \{s. (\text{root},v) \in \text{REACHABLE}\}) \cap \text{nmsg_eq } 0 \ e)$
 $\subseteq \text{final}$)"
 $\langle proof \rangle$

lemma *final_lemma4*:
 "E ≠ {}
 ==> (($\bigcap v \in V. \bigcap e \in E.$
 $\{s. ((s \in \text{reachable } v) = ((\text{root},v) \in \text{REACHABLE}))\} \cap \text{nmsg_eq } 0 \ e)$
 $= \text{final}$)"
 $\langle proof \rangle$

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"
```

```
definition
```

```
Fake :: "(state*state) set"
where "Fake = {(s,s').
        ∃ B X. s' = Says Spy B X # s
        & X ∈ synth (analz (spies s))}"
```

```
definition
```

```
NS1 :: "(state*state) set"
where "NS1 = {(s1,s').
        ∃ A1 B NA.
        s' = Says A1 B (Crypt (pubK B) {|Nonce NA, Agent A1|}) # s1
        & Nonce NA ∉ used s1}"
```

```
definition
```

```
NS2 :: "(state*state) set"
where "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}"
```


definition

```

NS3 :: "(state*state) set"
where "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}"

```

definition *Nprg* :: "state program" **where**

```
"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

```

definition

```

cmdF :: "(state*state) set"
where "cmdF = {(s,s'). s' = s (|NF:= Suc(NF s), BB:=False|) & BB s}"

```

definition

```

F :: "state program"
where "F = mk_total_program ({s. NF s = 0 & BB s}, {cmdF}, UNIV)"

```

definition

```

cmdG :: "(state*state) set"
where "cmdG = {(s,s'). s' = s (|NG:= Suc(NG s), BB:=True|) & ~ BB s}"

```

definition

```

G :: "state program"
where "G = mk_total_program ({s. NG s = 0 & BB s}, {cmdG}, UNIV)"

```

definition

```

invFG :: "state set"
where "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"

primrec sum  :: "[nat,state]=>int" where

  "sum 0 s = 0"
| "sum (Suc i) s = s (c i) + sum i s"

primrec sumj :: "[nat, nat, state]=>int" where
  "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"

definition a :: "nat=>command" where
  "a i = {(s, s'). s'=s(c i:= s (c i) + 1, C:= s C + 1)}"

definition Component :: "nat => state program" where
  "Component i =
    mk_total_program({s. s C = 0 & s (c i) = 0}, {a i},
       $\bigcup G \in \text{preserves } (\%s. s (c i)). \text{Acts } G$ )"

```

```

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

typedcl state

consts
  C :: "state=>int"
  c :: "state=>nat=>int"

primrec sum :: "[nat,state]=>int" where

  "sum 0 s = 0"
| "sum (Suc i) s = (c s) i + sum i s"

primrec sumj :: "[nat, nat, state]=>int" where
  "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"

definition a :: "nat=>command" where
  "a i = {(s, s'). (c s') i = (c s) i + 1 & (C s') = (C s) + 1}"

definition Component :: "nat => state program" where
  "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]: " $i < I \rightarrow \text{sum } I \text{ } s = c \text{ } s \text{ } i + \text{sumj } I \text{ } i \text{ } s$ "
<proof>

lemma *sum_ext* [rule_format]:
 " $(\forall i. i < I \rightarrow c \text{ } s' \text{ } i = c \text{ } s \text{ } i) \rightarrow (\text{sum } I \text{ } s' = \text{sum } I \text{ } s)$ "
<proof>

lemma *sumj_ext* [rule_format]:
 " $(\forall j. j < I \ \& \ j \neq i \rightarrow c \text{ } s' \text{ } j = c \text{ } s \text{ } j) \rightarrow (\text{sumj } I \text{ } i \text{ } s' = \text{sumj } I \text{ } i \text{ } s)$ "
<proof>

lemma *sum0* [rule_format]: " $(\forall i. i < I \rightarrow c \text{ } s \text{ } i = 0) \rightarrow \text{sum } I \text{ } s = 0$ "
<proof>

lemma *Component_ok_iff*:
 " $(\text{Component } i \text{ ok } G) =$
 $(G \in \text{preserves } (\%s. c \text{ } s \text{ } i) \ \& \ \text{Component } i \in \text{Allowed } G)$ "
<proof>
declare *Component_ok_iff* [iff]
declare *OK_iff_ok* [iff]
declare *preserves_def* [simp]

lemma *p2*: " $\text{Component } i \in \text{stable } \{s. C \text{ } s = (c \text{ } s) \text{ } i + k\}$ "
<proof>

lemma *p3*:
 " $[| \text{OK } I \text{ Component}; i \in I |]$
 $\Rightarrow \text{Component } i \in \text{stable } \{s. \forall j \in I. j \neq i \rightarrow c \text{ } s \text{ } j = c \text{ } k \text{ } j\}$ "
<proof>

lemma *p2_p3_lemma1*:
 " $[| \text{OK } \{i. i < I\} \text{ Component}; i < I |] \Rightarrow$
 $\forall k. \text{Component } i \in \text{stable } (\{s. C \text{ } s = c \text{ } s \text{ } i + \text{sumj } I \text{ } i \text{ } k\} \text{ Int}$
 $\{s. \forall j \in \{i. i < I\}. j \neq i \rightarrow c \text{ } s \text{ } j = c \text{ } k \text{ } j\})$ "
<proof>

lemma *p2_p3_lemma2*:
 " $(\forall k. F \in \text{stable } (\{s. C \text{ } s = (c \text{ } s) \text{ } i + \text{sumj } I \text{ } i \text{ } k\} \text{ Int}$
 $\{s. \forall j \in \{i. i < I\}. j \neq i \rightarrow c \text{ } s \text{ } j = c \text{ } k \text{ } j\}))$
 $\Rightarrow (F \in \text{stable } \{s. C \text{ } s = c \text{ } s \text{ } i + \text{sumj } I \text{ } i \text{ } s\})$ "
<proof>

lemma *p2_p3*:
 " $[| \text{OK } \{i. i < I\} \text{ Component}; i < I |]$
 $\Rightarrow \text{Component } i \in \text{stable } \{s. C \text{ } s = c \text{ } s \text{ } i + \text{sumj } I \text{ } i \text{ } s\}$ "

<proof>

```
lemma safety:
  "[| 0<I; OK {i. i<I} Component |]
   ==> ( $\bigcup$  i $\in$ {i. i<I}. (Component i))  $\in$  invariant {s. C s = sum I s}"
<proof>

end
```

```
theory PriorityAux
imports "../UNITY_Main"
begin
```

```
typedcl vertex
```

```
definition symcl :: "(vertex*vertex)set=>(vertex*vertex)set" where
  "symcl r == r  $\cup$  (r-1)"
  — symmetric closure: removes the orientation of a relation
```

```
definition neighbors :: "[vertex, (vertex*vertex)set]=>vertex set" where
  "neighbors i r == ((r  $\cup$  r-1)''{i}) - {i}"
  — Neighbors of a vertex i
```

```
definition R :: "[vertex, (vertex*vertex)set]=>vertex set" where
  "R i r == r''{i}"
```

```
definition A :: "[vertex, (vertex*vertex)set]=>vertex set" where
  "A i r == (r-1)''{i}"
```

```
definition reach :: "[vertex, (vertex*vertex)set]=> vertex set" where
  "reach i r == (r+)''{i}"
  — reachable and above vertices: the original notation was R* and A*
```

```
definition above :: "[vertex, (vertex*vertex)set]=> vertex set" where
  "above i r == ((r-1)+)''{i}"
```

```
definition reverse :: "[vertex, (vertex*vertex) set]=>(vertex*vertex)set" where
  "reverse i r == (r - {(x,y). x=i | y=i}  $\cap$  r)  $\cup$  ({(x,y). x=i|y=i}  $\cap$  r)-1"
```

```
definition derive1 :: "[vertex, (vertex*vertex)set, (vertex*vertex)set]=>bool"
where
```

```
  — The original definition
  "derive1 i r q == symcl r = symcl q &
    ( $\forall$  k k'. k $\neq$ i & k' $\neq$ i -->((k,k'):r) = ((k,k'):q)) &
    A i r = {} & R i q = {}"
```

```
definition derive :: "[vertex, (vertex*vertex)set, (vertex*vertex)set]=>bool"
where
```

```
  — 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*: " $((r^+)''\{i\} = \{j\}) = (r''\{i\} = \{j\})$ "
 <proof>

lemma *image0_r_iff_image0_trancl*: " $(r''\{i\} = \{j\}) = (\text{ALL } x. ((i, x):r^+) = \text{False})$ "
 <proof>

lemma *acyclic_eq_wf*: " $!!r::(\text{vertex}*\text{vertex})\text{set}. \text{acyclic } r = \text{wf } r$ "
 <proof>

lemma *derive_derive1_eq*: " $\text{derive } i \ r \ q = \text{derive1 } i \ r \ q$ "
 <proof>

lemma *lemma1_a*:
 " $[| x \in \text{reach } i \ q; \text{derive1 } k \ r \ q |] \implies x \neq k \implies x \in \text{reach } i \ r$ "
 <proof>

lemma *reach_lemma*: " $\text{derive } k \ r \ q \implies \text{reach } i \ q \subseteq (\text{reach } i \ r \cup \{k\})$ "
 <proof>

lemma *reach_above_lemma*:
 " $(\forall i. \text{reach } i \ q \subseteq (\text{reach } i \ r \cup \{k\})) =$
 $(\forall x. x \neq k \implies (\forall i. i \notin \text{above } x \ r \implies i \notin \text{above } x \ q))$ "
 <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

```

definition highest :: "[vertex, (vertex*vertex)set]=>bool"
  where "highest i r <-> A i r = {}"
  — i has highest priority in r

```

```

definition lowest :: "[vertex, (vertex*vertex)set]=>bool"
  where "lowest i r <-> R i r = {}"
  — i has lowest priority in r

```

```

definition act :: command
  where "act i = {(s, s'). s'=reverse i s & highest i s}"

```

```

definition Component :: "vertex=>state program"
  where "Component i = mk_total_program({init}, {act i}, UNIV)"
  — All components start with the same initial state

```

Some Abbreviations

```

definition Highest :: "vertex=>state set"
  where "Highest i = {s. highest i s}"

```

```

definition Lowest :: "vertex=>state set"
  where "Lowest i = {s. lowest i s}"

```

```

definition Acyclic :: "state set"

```

where "Acyclic = {s. acyclic s}"

definition Maximal :: "state set"

— Every "above" set has a maximal vertex

where "Maximal = ($\bigcap i. \{s. \sim \text{highest } i \ s \rightarrow (\exists j \in \text{above } i \ s. \text{highest } j \ s)\}$)"

definition Maximal' :: "state set"

— Maximal vertex: equivalent definition

where "Maximal' = ($\bigcap i. \text{Highest } i \ \text{Un } (\bigcup j. \{s. j \in \text{above } i \ s\} \ \text{Int } \text{Highest } j)\}$)"

definition Safety :: "state set"

where "Safety = ($\bigcap i. \{s. \text{highest } i \ s \rightarrow (\forall j \in \text{neighbors } i \ s. \sim \text{highest } j \ s)\}$)"

definition system :: "state program"

where "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 \in 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 \neq {}} Int Highest i ensures - Highest i"

<proof>

or better

lemma *Component_yields_priority'*: "Component $i \in \text{Highest } i$ ensures $\text{Lowest } i$ "
 $\langle \text{proof} \rangle$

property 6: Component doesn't introduce cycle

lemma *Component_well_behaves*: "Component $i \in \text{Highest } i$ co $\text{Highest } i$ Un $\text{Lowest } i$ "
 $\langle \text{proof} \rangle$

property 7: local axiom

lemma *locality*: "Component $i \in \text{stable } \{s. \forall j k. j \neq i \ \& \ k \neq i \rightarrow ((j,k):s) = b \ j \ k\}$ "
 $\langle \text{proof} \rangle$

23.2 System properties

property 8: strictly universal

lemma *Safety*: " $\text{system} \in \text{stable } \text{Safety}$ "
 $\langle \text{proof} \rangle$

property 13: universal

lemma *p13*: " $\text{system} \in \{s. s = q\}$ co $\{s. s=q\}$ Un $\{s. \exists i. \text{derive } i \ q \ s\}$ "
 $\langle \text{proof} \rangle$

property 14: the 'above set' of a Component that hasn't got priority doesn't increase

lemma *above_not_increase*:
" $\text{system} \in \neg \text{Highest } i \text{ Int } \{s. j \notin \text{above } i \ s\}$ co $\{s. j \notin \text{above } i \ s\}$ "
 $\langle \text{proof} \rangle$

lemma *above_not_increase'*:
" $\text{system} \in \neg \text{Highest } i \text{ Int } \{s. \text{above } i \ s = x\}$ co $\{s. \text{above } i \ s \leq x\}$ "
 $\langle \text{proof} \rangle$

p15: universal property: all Components well behave

lemma *system_well_behaves* [rule_format]:
" $\forall i. \text{system} \in \text{Highest } i$ co $\text{Highest } i$ Un $\text{Lowest } i$ "
 $\langle \text{proof} \rangle$

lemma *Acyclic_eq*: " $\text{Acyclic} = (\bigcap i. \{s. i \notin \text{above } i \ s\})$ "
 $\langle \text{proof} \rangle$

lemmas *system_co* =
 $\text{constrains_Un } [OF \ \text{above_not_increase} \ [\text{rule_format}] \ \text{system_well_behaves}]$

lemma *Acyclic_stable*: " $\text{system} \in \text{stable } \text{Acyclic}$ "
 $\langle \text{proof} \rangle$

lemma *Acyclic_subset_Maximal*: "Acyclic <= Maximal"
 <proof>

property 17: original one is an invariant

lemma *Acyclic_Maximal_stable*: "system \in stable (Acyclic Int Maximal)"
 <proof>

property 5: existential property

lemma *Highest_leadsTo_Lowest*: "system \in Highest i leadsTo Lowest i"
 <proof>

a lowest i can never be in any abover set

lemma *Lowest_above_subset*: "Lowest i <= ($\bigcap k. \{s. i \notin \text{above } k \ s\}$)"
 <proof>

property 18: a simpler proof than the original, one which uses psp

lemma *Highest_escapes_above*: "system \in Highest i leadsTo ($\bigcap k. \{s. i \notin \text{above } k \ s\}$)"
 <proof>

lemma *Highest_escapes_above'*:
 "system \in Highest j Int {s. j \in above i s} leadsTo {s. j \notin 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 \in ($\bigcup j. \{s. \text{above } i \ s = x\}$ Int {s. j \in above i s} Int Highest
 j)
 leadsTo {s. above i s < x}"
 <proof>

lemma *Maximal_eq_Maximal'*: "Maximal = Maximal'"
 <proof>

lemma *Acyclic_subset*:
 "x \neq {} ==>
 Acyclic Int {s. above i s = x} <=
 ($\bigcup j. \{s. \text{above } i \ s = x\}$ Int {s. j \in 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"
```

```
definition count :: "'a state => nat"
  where "count s = fst s"
```

```
definition decr :: "('a state * 'a state) set"
  where "decr = (UN n uu. {(Suc n, uu), (n, uu)})"
```

```
definition Timer :: "'a state program"
  where "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 Progress Set Examples

theory Progress imports "../UNITY_Main" begin

24.1 The Composition of Two Single-Assignment Programs

Thesis Section 4.4.2

definition FF :: "int program" where
 "FF == mk_total_program (UNIV, {range ($\lambda x. (x, x+1)$)}, UNIV)"

definition GG :: "int program" where
 "GG == mk_total_program (UNIV, {range ($\lambda x. (x, 2*x)$)}, UNIV)"

24.1.1 Calculating wens_set FF {k..}

lemma Domain_actFF: "Domain (range ($\lambda x::\text{int}. (x, x + 1)$)) = UNIV"
 <proof>

lemma FF_eq:
 "FF = mk_program (UNIV, {range ($\lambda x. (x, x+1)$)}, UNIV)"
 <proof>

lemma wp_actFF:
 "wp (range ($\lambda x::\text{int}. (x, x + 1)$)) (atLeast k) = atLeast (k - 1)"
 <proof>

lemma wens_FF: "wens FF (range ($\lambda x. (x, x+1)$)) (atLeast k) = atLeast (k - 1)"
 <proof>

lemma single_valued_actFF: "single_valued (range ($\lambda x::\text{int}. (x, x + 1)$))"
 <proof>

lemma wens_single_finite_FF:
 "wens_single_finite (range ($\lambda x. (x, x+1)$)) (atLeast k) n =
 atLeast (k - int n)"
 <proof>

lemma wens_single_FF_eq_UNIV:
 "wens_single (range ($\lambda x::\text{int}. (x, x + 1)$)) (atLeast k) = UNIV"
 <proof>

lemma wens_set_FF:
 "wens_set FF (atLeast k) = insert UNIV (atLeast ' atMost k)"
 <proof>

24.1.2 Proving FF \in UNIV \mapsto {k..}

lemma atLeast_ensures: "FF \in atLeast (k - 1) ensures atLeast (k::int)"

<proof>

lemma *atLeast_leadsTo*: " $FF \in \text{atLeast } (k - \text{int } n) \text{ leadsTo } \text{atLeast } (k::\text{int})$ "

<proof>

lemma *UN_atLeast_UNIV*: " $(\bigcup n. \text{atLeast } (k - \text{int } n)) = \text{UNIV}$ "

<proof>

lemma *FF_leadsTo*: " $FF \in \text{UNIV leadsTo } \text{atLeast } (k::\text{int})$ "

<proof>

Result (4.39): Applying the leadsTo-Join Theorem

theorem " $FF \sqcup GG \in \text{atLeast } 0 \text{ leadsTo } \text{atLeast } (k::\text{int})$ "

<proof>

end

25 Common Declarations for Chandy and Charpentier's Allocator

theory *AllocBase* imports "*../UNITY_Main*" begin

consts

NbT :: nat

Nclients :: nat

axioms

NbT_pos: " $0 < \text{NbT}$ "

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:
  "( $\sum i \in A \text{ Int lessThan } k. \{\# \text{if } i < k \text{ then } f \text{ } i \text{ else } g \text{ } i \#\}$ ) =
   ( $\sum i \in A \text{ Int lessThan } k. \{\# f \text{ } i \#\}$ )"
<proof>
```

```
lemma bag_of_sublist:
  "bag_of (sublist l A) =
   ( $\sum i \in A \text{ Int lessThan } (\text{length } l). \{\# l ! i \#\}$ )"
<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)) =
        ( $\sum i \in I. \text{bag\_of (sublist } l \text{ (A } i))$ )"
<proof>
```

```
end
```

```
theory Alloc
imports AllocBase "../PPROD"
begin
```

25.1 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

definition
  — DUPLICATED FROM Client.thy, but with "tok" removed
  — Maybe want a special theory section to declare such maps
  non_dummy :: "'a clientState_d => clientState"
  where "non_dummy s = (|giv = giv s, ask = ask s, rel = rel s|)"

definition
  — Renaming map to put a Client into the standard form
  client_map :: "'a clientState_d => clientState*'a"
  where "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

```

— * Resource allocation system specification *

```

definition
  — spec (1)
  system_safety :: "'a systemState program set"
  where "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)}"

```

```

definition
  — spec (2)
  system_progress :: "'a systemState program set"
  where "system_progress = (INT i : lessThan Nclients.
    INT h.
      {s. h ≤ (ask o sub i o client)s} LeadsTo
      {s. h prefixLe (giv o sub i o client) s})"

```

definition

```
system_spec :: "'a systemState program set"
where "system_spec = system_safety Int system_progress"
```

— * Client specification (required) **

definition

```
— spec (3)
client_increasing :: "'a clientState_d program set"
where "client_increasing = UNIV guarantees Increasing ask Int Increasing
rel"
```

definition

```
— spec (4)
client_bounded :: "'a clientState_d program set"
where "client_bounded = UNIV guarantees Always {s. ALL elt : set (ask s).
elt ≤ NbT}"
```

definition

```
— spec (5)
client_progress :: "'a clientState_d program set"
where "client_progress =
Increasing giv guarantees
(INT h. {s. h ≤ giv s & h pfixGe ask s}
LeadsTo {s. tokens h ≤ (tokens o rel) s})"
```

definition

```
— spec: preserves part
client_preserves :: "'a clientState_d program set"
where "client_preserves = preserves giv Int preserves clientState_d.dummy"
```

definition

```
— environmental constraints
client_allowed_acts :: "'a clientState_d program set"
where "client_allowed_acts =
{F. AllowedActs F =
insert Id (UNION (preserves (funPair rel ask)) Acts)}"
```

definition

```
client_spec :: "'a clientState_d program set"
where "client_spec = client_increasing Int client_bounded Int client_progress
Int client_allowed_acts Int client_preserves"
```

— * Allocator specification (required) *

definition

```
— spec (6)
alloc_increasing :: "'a allocState_d program set"
where "alloc_increasing =
UNIV guarantees
(INT i : lessThan Nclients. Increasing (sub i o allocGiv))"
```

definition

```

— spec (7)
alloc_safety :: "'a allocState_d program set"
where "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)}"

definition
— spec (8)
alloc_progress :: "'a allocState_d program set"
where "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})"

definition
— spec: preserves part
alloc_preserves :: "'a allocState_d program set"
where "alloc_preserves = preserves allocRel Int preserves allocAsk Int
      preserves allocState_d.dummy"

definition
— environmental constraints
alloc_allowed_acts :: "'a allocState_d program set"
where "alloc_allowed_acts =
      {F. AllowedActs F =
        insert Id (UNION (preserves allocGiv) Acts)}"

definition
alloc_spec :: "'a allocState_d program set"
where "alloc_spec = alloc_increasing Int alloc_safety Int alloc_progress
      Int
        alloc_allowed_acts Int alloc_preserves"

— * Network specification *

definition
— spec (9.1)
network_ask :: "'a systemState program set"

```

```

where "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)))"

```

definition

```

— spec (9.2)
network_giv :: "'a systemState program set"
where "network_giv = (INT i : lessThan Nclients.
  Increasing (sub i o allocGiv)
  guarantees
  ((giv o sub i o client) Fols (sub i o allocGiv)))"

```

definition

```

— spec (9.3)
network_rel :: "'a systemState program set"
where "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)))"

```

definition

```

— spec: preserves part
network_preserves :: "'a systemState program set"
where "network_preserves =
  preserves allocGiv Int
  (INT i : lessThan Nclients. preserves (rel o sub i o client) Int
    preserves (ask o sub i o client))"

```

definition

```

— environmental constraints
network_allowed_acts :: "'a systemState program set"
where "network_allowed_acts =
  {F. AllowedActs F =
    insert Id
    (UNION (preserves allocRel Int
      (INT i: lessThan Nclients. preserves(giv o sub i o client)))
    Acts)}"

```

definition

```

network_spec :: "'a systemState program set"
where "network_spec = network_ask Int network_giv Int
  network_rel Int network_allowed_acts Int
  network_preserves"

```

— * State mappings *

definition

```

sysOfAlloc :: "((nat => clientState) * 'a) allocState_d => 'a systemState"
where "sysOfAlloc = (%s. let (cl,xtr) = allocState_d.dummy s
  in (| allocGiv = allocGiv s,
    allocAsk = allocAsk s,
    allocRel = allocRel s,
    client = cl,
    dummy = xtr|))"

```

definition

```

sysOfClient :: "(nat => clientState) * 'a allocState_d => 'a systemState"
where "sysOfClient = (%(cl,al). (| allocGiv = allocGiv al,
                                allocAsk = allocAsk al,
                                allocRel = allocRel al,
                                client   = cl,
                                systemState.dummy = allocState_d.dummy al|))"
```

```

axiomatization Alloc :: "'a allocState_d program"
```

```

  where Alloc: "Alloc : alloc_spec"
```

```

axiomatization Client :: "'a clientState_d program"
```

```

  where Client: "Client : client_spec"
```

```

axiomatization Network :: "'a systemState program"
```

```

  where Network: "Network : network_spec"
```

```

definition System :: "'a systemState program"
```

```

  where "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
```

$\langle ML \rangle$

```
lemmas lessThanBspec = lessThan_iff [THEN iffd2, THEN [2] bspec]
```

$\langle ML \rangle$

```
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>
```

25.1.1 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>
```

25.1.2 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"
```


$\langle proof \rangle$
lemma *bij_client_map* [iff]: "bij client_map"
 $\langle proof \rangle$
 o-simprules for *client_map*
lemma *fst_o_client_map*: "fst o client_map = non_dummy"
 $\langle proof \rangle$
 $\langle ML \rangle$
declare *fst_o_client_map'* [simp]
lemma *snd_o_client_map*: "snd o client_map = clientState_d.dummy"
 $\langle proof \rangle$
 $\langle ML \rangle$
declare *snd_o_client_map'* [simp]

25.2 o-simprules for *sysOfAlloc* [MUST BE AUTOMATED]

lemma *client_o_sysOfAlloc*: "client o sysOfAlloc = fst o allocState_d.dummy"
 $\langle proof \rangle$
 $\langle ML \rangle$
declare *client_o_sysOfAlloc'* [simp]
lemma *allocGiv_o_sysOfAlloc_eq*: "allocGiv o sysOfAlloc = allocGiv"
 $\langle proof \rangle$
 $\langle ML \rangle$
declare *allocGiv_o_sysOfAlloc_eq'* [simp]
lemma *allocAsk_o_sysOfAlloc_eq*: "allocAsk o sysOfAlloc = allocAsk"
 $\langle proof \rangle$
 $\langle ML \rangle$
declare *allocAsk_o_sysOfAlloc_eq'* [simp]
lemma *allocRel_o_sysOfAlloc_eq*: "allocRel o sysOfAlloc = allocRel"
 $\langle proof \rangle$
 $\langle ML \rangle$
declare *allocRel_o_sysOfAlloc_eq'* [simp]

25.3 o-simprules for *sysOfClient* [MUST BE AUTOMATED]

lemma *client_o_sysOfClient*: "client o sysOfClient = fst"
 $\langle proof \rangle$
 $\langle ML \rangle$
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 :  $\downarrow$ unfolded specification $\downarrow$ 

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

 $\langle ML \rangle$ 

declare
  Client_Increasing_ask [iff]
  Client_Increasing_rel [iff]
  Client_Bounded [iff]
  Client_preserves_giv [iff]
  Client_preserves_dummy [iff]

Network :  $\downarrow$ unfolded specification $\downarrow$ 

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

 $\langle ML \rangle$ 

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 :  $\downarrow$ unfolded specification $\downarrow$ 

lemmas alloc_spec_simps =
  alloc_spec_def alloc_increasing_def alloc_safety_def
  alloc_progress_def alloc_allowed_acts_def alloc_preserves_def

 $\langle ML \rangle$ 

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]

```

25.4 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 o sub i o fst o lift_map i o 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_subfst_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>

```

25.5 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. ( $\sum i \in \text{lessThan } Nclients. (\text{tokens o sub } i \text{ o allocGiv})$ 
s)
 $\leq NbT + (\sum i \in \text{lessThan } Nclients. (\text{tokens o sub } i \text{ o allocRel})$ 
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
 $\leq (\text{tokens o sub } i \text{ o allocGiv}) s$  })"
<proof>

```

```

lemma Always_tokens_allocRel_le_rel: "System : Always (INT i: lessThan Nclients.
{ s. (tokens o sub i o allocRel) s
 $\leq (\text{tokens o rel o sub } i \text{ o client}) s$  })"
<proof>

```

safety (1), step 4 (final result!)

```

theorem System_safety: "System : system_safety"
<proof>

```

25.6 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  $\leq NbT$ }"
<proof>

```

```

lemma System_Bounded_ask: "i < Nclients
==> System : Always
{ s. ALL elt : set ((ask o sub i o client) s). elt  $\leq NbT$  }"
<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  $\leq NbT$ }"
<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 prefixGe (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 prefixGe (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 prefixGe (sub i o allocAsk) s}
      LeadsTo
        {s. h ≤ (giv o sub i o client) s &
          h prefixGe (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 prefixGe (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). (l giv = g, ask =
a, rel = r l)) 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

```

26 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

definition non_dummy :: "('a, 'b) merge_d => 'b merge" where
  "non_dummy s = (lIn = 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

```

definition

```

merge_increasing :: "('a,'b) merge_d program set"
where "merge_increasing =
  UNIV guarantees (Increasing merge.Out) Int (Increasing merge.iOut)"

```

definition

```

merge_eqOut :: "('a,'b) merge_d program set"
where "merge_eqOut =
  UNIV guarantees
  Always {s. length (merge.Out s) = length (merge.iOut s)}"

```

definition

```

merge_bounded :: "('a,'b) merge_d program set"
where "merge_bounded =
  UNIV guarantees
  Always {s.  $\forall \text{elt} \in \text{set } (\text{merge.iOut } s). \text{elt} < \text{Nclients}$ }"

```

definition

```

merge_follows :: "('a,'b) merge_d program set"
where "merge_follows =
  ( $\bigcap i \in \text{lessThan } \text{Nclients}. \text{Increasing } (\text{sub } i \text{ o merge.In})$ )
  guarantees
  ( $\bigcap i \in \text{lessThan } \text{Nclients}.
    (\%s. \text{sublist } (\text{merge.Out } s)$ 

```

```

      {k. k < size(merge.iOut s) & merge.iOut s! k = i})
    Fols (sub i o merge.In))"

```

definition

```

merge_preserves :: "('a,'b) merge_d program set"
where "merge_preserves = preserves merge.In Int preserves merge_d.dummy"

```

definition

```

merge_allowed_acts :: "('a,'b) merge_d program set"
where "merge_allowed_acts =
  {F. AllowedActs F =
    insert Id (UNION (preserves (funPair merge.Out merge.iOut)) Acts)}"

```

definition

```

merge_spec :: "('a,'b) merge_d program set"
where "merge_spec = merge_increasing Int merge_eqOut Int merge_bounded Int
      merge_follows Int merge_allowed_acts Int merge_preserves"

```

definition

```

distr_follows :: "('a,'b) distr_d program set"
where "distr_follows =
  Increasing distr.In Int Increasing distr.iIn Int
  Always {s.  $\forall \text{elt} \in \text{set } (\text{distr.iIn } s). \text{elt} < \text{Nclients}$ }
  guarantees
  ( $\bigcap i \in \text{lessThan } \text{Nclients}.$ 
    (sub i o distr.Out) Fols
    (%s. sublist (distr.In s)
      {k. k < size(distr.iIn s) & distr.iIn s ! k = i}))"

```

definition

```

distr_allowed_acts :: "('a,'b) distr_d program set"
where "distr_allowed_acts =
  {D. AllowedActs D = insert Id (UNION (preserves distr.Out) Acts)}"

```

definition

```

distr_spec :: "('a,'b) distr_d program set"
where "distr_spec = distr_follows Int distr_allowed_acts"

```

definition

```

alloc_increasing :: "'a allocState_d program set"
where "alloc_increasing = UNIV guarantees Increasing giv"

```

definition

```

alloc_safety :: "'a allocState_d program set"
where "alloc_safety =

```

```

Increasing rel
guarantees Always {s. tokens (giv s) ≤ NbT + tokens (rel s)}"

```

definition

```

alloc_progress :: "'a allocState_d program set"
where "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})"

```

definition

```

alloc_preserves :: "'a allocState_d program set"
where "alloc_preserves = preserves rel Int
  preserves ask Int
  preserves allocState_d.dummy"

```

definition

```

alloc_allowed_acts :: "'a allocState_d program set"
where "alloc_allowed_acts =
  {F. AllowedActs F = insert Id (UNION (preserves giv) Acts)}"

```

definition

```

alloc_spec :: "'a allocState_d program set"
where "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]

```

26.1 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>

26.2 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}
 |]
 <proof>

```

==> 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⟩

```

26.3 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

```

27 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

definition
  rel_act :: "('a state_d * 'a state_d) set"
  where "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}"

definition
  tok_act :: "('a state_d * 'a state_d) set"
  where "tok_act = {(s,s'). s'=s | s' = s (| tok := Suc (tok s mod NbT) |)}"

definition
  ask_act :: "('a state_d * 'a state_d) set"
  where "ask_act = {(s,s'). s'=s |
    (s' = s (| ask := ask s @ [tok s] |))}"

definition
  Client :: "'a state_d program"
  where "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)"

```

definition

```

non_dummy :: "'a state_d => state"
where "non_dummy s = (/giv = giv s, ask = ask s, rel = rel s, tok = tok
s/)"

```

definition

```

client_map :: "'a state_d => state*'a"
where "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  $\in$  Increasing giv guarantees Always  $\{s. \text{rel } s \leq \text{giv } s\}$ "
    <proof>

```

```

end

```

28 Projections of State Sets

```

theory Project imports Extend begin

```

```

definition projecting :: "'c program => 'c set, 'a*'b => 'c,
      'a program, 'c program set, 'a program set] => bool" where
  "projecting C h F X' X ==
     $\forall G. \text{extend } h \ F \sqcup G \in X' \rightarrow F \sqcup \text{project } h \ (C \ G) \ G \in X$ "

```

```

definition extending :: "'c program => 'c set, 'a*'b => 'c, 'a program,
      'c program set, 'a program set] => bool" where
  "extending C h F Y' Y ==
     $\forall G. \text{extend } h \ F \text{ ok } G \rightarrow F \sqcup \text{project } h \ (C \ G) \ G \in Y$ 
     $\rightarrow \text{extend } h \ F \sqcup G \in Y'$ "

```

```

definition subset_closed :: "'a set set => bool" where
  "subset_closed U ==  $\forall A \in U. \text{Pow } A \subseteq U$ "

```

```

lemma (in Extend) project_extend_constrains_I:
  "F  $\in$  A co B ==> project h C (extend h F)  $\in$  A co B"
<proof>

```

28.1 Safety

```

lemma (in Extend) project_unless [rule_format]:
  "[| G  $\in$  stable C; project h C G  $\in$  A unless B |]
  ==> G  $\in$  (C  $\cap$  extend_set h A) unless (extend_set h B)"
<proof>

```

```

lemma (in Extend) Join_project_constrains:
  "(F  $\sqcup$  project h C G  $\in$  A co B) =
    (extend h F  $\sqcup$  G  $\in$  (C  $\cap$  extend_set h A) co (extend_set h B) &
     F  $\in$  A co B)"
<proof>

```

```

lemma (in Extend) Join_project_stable:
  "extend h F  $\sqcup$  G  $\in$  stable C
  ==> (F  $\sqcup$  project h C G  $\in$  stable A) =
    (extend h F  $\sqcup$  G  $\in$  stable (C  $\cap$  extend_set h A) &
     F  $\in$  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⟩

```

28.2 "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'  $\cap$  YB') (YA  $\cap$  YB)"
  <proof>

```

```

lemma extending_Un:
  "[| extending C h F YA' YA; extending C h F YB' YB |]
   ==> extending C h F (YA'  $\cup$  YB') (YA  $\cup$  YB)"
  <proof>

```

```

lemma extending_INT:
  "[| !!i. i  $\in$  I ==> extending C h F (Y' i) (Y i) |]
   ==> extending C h F ( $\bigcap$  i  $\in$  I. Y' i) ( $\bigcap$  i  $\in$  I. Y i)"
  <proof>

```

```

lemma extending_UN:
  "[| !!i. i  $\in$  I ==> extending C h F (Y' i) (Y i) |]
   ==> extending C h F ( $\bigcup$  i  $\in$  I. Y' i) ( $\bigcup$  i  $\in$  I. Y i)"
  <proof>

```

```

lemma extending_weaken:
  "[| extending C h F Y' Y; Y'  $\leq$  V'; V  $\subseteq$  Y |] ==> extending C h F V' V"
  <proof>

```

```

lemma extending_weaken_L:
  "[| extending C h F Y' Y; Y'  $\leq$  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>

28.3 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>

28.4 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>
```

28.5 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⟩

28.6 leadsETo in the precondition (??)

28.6.1 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⟩

28.6.2 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⟩

```

28.7 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>

```

28.8 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 xguar: "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>

```

28.9 guarantees corollaries

28.9.1 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⟩

```

28.9.2 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

29 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"

```

definition

```

  givenBy :: "[ 'a => 'b] => 'a set set"

```

where "givenBy f = range (%B. f-‘ B)"

definition

leadsETo :: "[’a set, ’a set set, ’a set] => ’a program set"
 ("(3_/ leadsTo[_]/ _)" [80,0,80] 80)
where "leadsETo A CC B = {F. (A,B) : elt CC F}"

definition

LeadsETo :: "[’a set, ’a set set, ’a set] => ’a program set"
 ("(3_/ LeadsTo[_]/ _)" [80,0,80] 80)
where "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

```

30 Various examples for UNITY

```

theory UNITY_Examples
imports
  UNITY_Main

```

```

"Simple/Deadlock"
"Simple/Common"
"Simple/Network"
"Simple/Token"
"Simple/Channel"
"Simple/Lift"
"Simple/Mutex"
"Simple/Reach"
"Simple/Reachability"

```

```

"Simple/NSP_Bad"

```

```

"Comp/Handshake"

```

```

"Comp/Counter"
"Comp/CounterC"
"Comp/Priority"

```

```

"Comp/TimerArray"
"Comp/Progress"

```

```

"Comp/Alloc"
"Comp/AllocImpl"

```

"Comp/Client"

"ELT"

begin

end