

# The UNITY Formalism

Sidi Ehmety and Lawrence C. Paulson

December 17, 2025

## Contents

<b>1</b>	<b>The Basic UNITY Theory</b>	<b>2</b>
1.0.1	The abstract type of programs . . . . .	3
1.0.2	Inspectors for type "program" . . . . .	3
1.0.3	Equality for UNITY programs . . . . .	3
1.0.4	co . . . . .	4
1.0.5	Union . . . . .	4
1.0.6	Intersection . . . . .	5
1.0.7	unless . . . . .	5
1.0.8	stable . . . . .	5
1.0.9	Union . . . . .	6
1.0.10	Intersection . . . . .	6
1.0.11	invariant . . . . .	7
1.0.12	increasing . . . . .	7
1.0.13	Theoretical Results from Section 6 . . . . .	7
1.0.14	Ad-hoc set-theory rules . . . . .	8
1.1	Partial versus Total Transitions . . . . .	8
1.1.1	Basic properties . . . . .	8
1.2	Rules for Lazy Definition Expansion . . . . .	9
1.2.1	Inspectors for type "program" . . . . .	10
<b>2</b>	<b>Fixed Point of a Program</b>	<b>10</b>
<b>3</b>	<b>Progress</b>	<b>11</b>
3.1	transient . . . . .	12
3.2	ensures . . . . .	13
3.3	leadsTo . . . . .	14
3.4	PSP: Progress-Safety-Progress . . . . .	18
3.5	Proving the induction rules . . . . .	19
3.6	wlt . . . . .	20
3.7	Completion: Binary and General Finite versions . . . . .	21
<b>4</b>	<b>Weak Safety</b>	<b>23</b>
4.1	traces and reachable . . . . .	24
4.2	Co . . . . .	24
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	29
4.8	Totalize	30
<b>5</b>	<b>Weak Progress</b>	<b>30</b>
5.1	Specialized laws for handling invariants	31
5.2	Introduction rules: Basis, Trans, Union	31
5.3	Derived rules	31
5.4	PSP: Progress-Safety-Progress	35
5.5	Induction rules	36
5.6	Completion: Binary and General Finite versions	37
<b>6</b>	<b>The Detects Relation</b>	<b>38</b>
<b>7</b>	<b>Unions of Programs</b>	<b>39</b>
7.1	SKIP	40
7.2	SKIP and safety properties	40
7.3	Join	41
7.4	JN	41
7.5	Algebraic laws	41
7.6	Laws Governing $\sqcup$	42
7.7	Safety: co, stable, FP	42
7.8	Progress: transient, ensures	43
7.9	the ok and OK relations	45
7.10	Allowed	46
7.11	<i>safety_prop</i> , for reasoning about given instances of "ok"	46
<b>8</b>	<b>Composition: Basic Primitives</b>	<b>47</b>
8.1	The component relation	48
8.2	The preserves property	49
<b>9</b>	<b>Guarantees Specifications</b>	<b>52</b>
9.1	Existential Properties	53
9.2	Universal Properties	54
9.3	Guarantees	54
9.4	Distributive Laws. Re-Orient to Perform Miniscoping	55
9.5	Guarantees: Additional Laws (by lcp)	56
9.6	Guarantees Laws for Breaking Down the Program (by lcp)	57
<b>10</b>	<b>Extending State Sets</b>	<b>61</b>
10.1	Restrict	62
10.2	Trivial properties of f, g, h	63
10.3	<i>extend_set</i> : basic properties	63
10.4	<i>project_set</i> : basic properties	64
10.5	More laws	64
10.6	<i>extend_act</i>	65
10.7	extend	66
10.8	Safety: co, stable	68
10.9	Weak safety primitives: Co, Stable	69
10.10	Progress: transient, ensures	70
10.11	Proving the converse takes some doing!	71

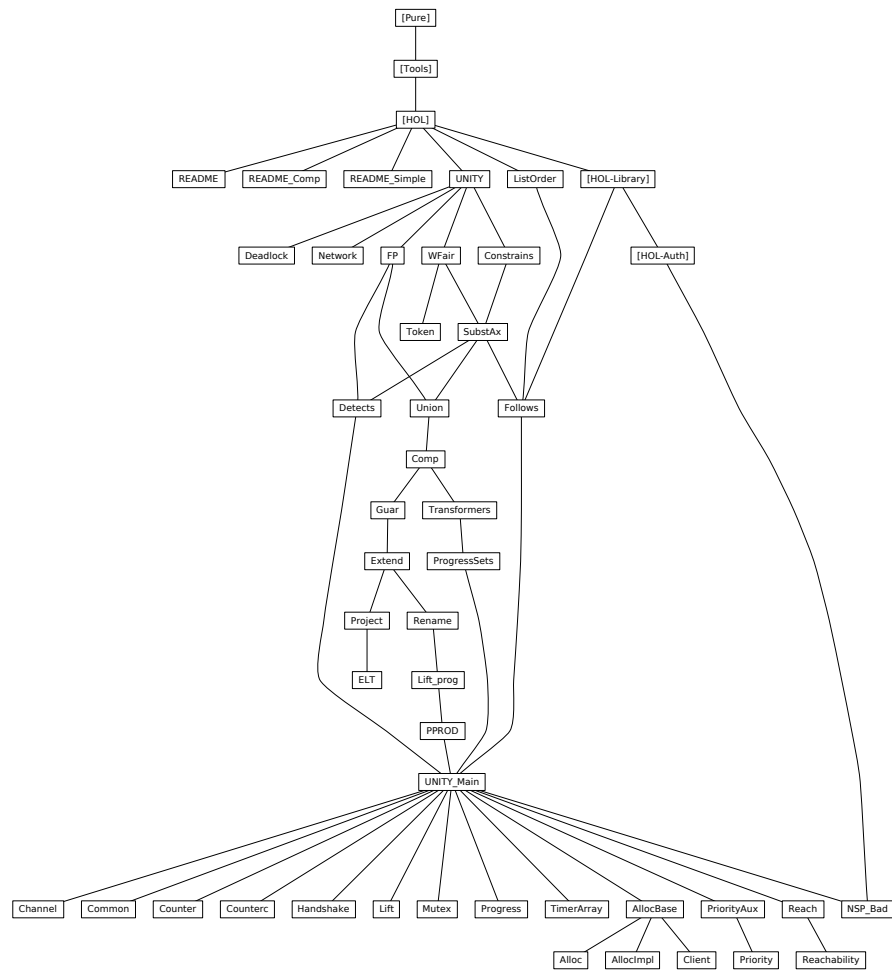
10.12	preserves . . . . .	72
10.13	Guarantees . . . . .	72
<b>11</b>	<b>Renaming of State Sets</b>	<b>74</b>
11.1	inverse properties . . . . .	74
11.2	the lattice operations . . . . .	77
11.3	Strong Safety: co, stable . . . . .	77
11.4	Weak Safety: Co, Stable . . . . .	77
11.5	Progress: transient, ensures . . . . .	78
11.6	"image" versions of the rules, for lifting "guarantees" properties . . . . .	79
<b>12</b>	<b>Replication of Components</b>	<b>81</b>
12.1	Injectiveness proof . . . . .	82
12.2	Surjectiveness proof . . . . .	82
12.3	The Operator <i>lift_set</i> . . . . .	83
12.4	The Lattice Operations . . . . .	84
12.5	Safety: constrains, stable, invariant . . . . .	84
12.6	Progress: transient, ensures . . . . .	85
12.7	Lemmas to Handle Function Composition (o) More Consistently . . . . .	87
12.8	More lemmas about extend and project . . . . .	87
12.9	OK and "lift" . . . . .	88
<b>13</b>	<b>The Prefix Ordering on Lists</b>	<b>91</b>
13.1	preliminary lemmas . . . . .	92
13.2	genPrefix is a partial order . . . . .	92
13.3	recursion equations . . . . .	94
13.4	The type of lists is partially ordered . . . . .	95
13.5	pfixLe, pfixGe: properties inherited from the translations . . . . .	98
<b>14</b>	<b>The Follows Relation of Charpentier and Sivilotte</b>	<b>99</b>
14.1	Destruction rules . . . . .	100
14.2	Union properties (with the subset ordering) . . . . .	101
14.3	Multiset union properties (with the multiset ordering) . . . . .	102
<b>15</b>	<b>Predicate Transformers</b>	<b>103</b>
15.1	Defining the Predicate Transformers <i>wp</i> , <i>awp</i> and <i>wens</i> . . . . .	104
15.2	Defining the Weakest Ensures Set . . . . .	106
15.3	Properties Involving Program Union . . . . .	107
15.4	The Set <i>wens_set F B</i> for a Single-Assignment Program . . . . .	109
<b>16</b>	<b>Progress Sets</b>	<b>113</b>
16.1	Complete Lattices and the Operator <i>c1</i> . . . . .	113
16.2	Progress Sets and the Main Lemma . . . . .	115
16.3	The Progress Set Union Theorem . . . . .	117
16.4	Some Progress Sets . . . . .	118
16.4.1	Lattices and Relations . . . . .	118
16.4.2	Decoupling Theorems . . . . .	120
16.5	Composition Theorems Based on Monotonicity and Commutativity . . . . .	121
16.5.1	Commutativity of <i>c1 L</i> and assignment. . . . .	121
16.5.2	Commutativity of Functions and Relation . . . . .	122
16.6	Monotonicity . . . . .	123

<b>17 Comprehensive UNITY Theory</b>	<b>123</b>
<b>18 The Token Ring</b>	<b>128</b>
18.1 Definitions . . . . .	129
18.2 Progress under Weak Fairness . . . . .	130
18.3 Progress . . . . .	137
<b>19 Analyzing the Needham-Schroeder Public-Key Protocol in UNITY</b>	<b>155</b>
19.1 Inductive Proofs about <i>ns_public</i> . . . . .	156
19.2 Authenticity properties obtained from NS2 . . . . .	157
19.3 Authenticity properties obtained from NS2 . . . . .	159
<b>20 A Family of Similar Counters: Original Version</b>	<b>162</b>
<b>21 A Family of Similar Counters: Version with Compatibility</b>	<b>164</b>
<b>22 The priority system</b>	<b>169</b>
22.1 Component correctness proofs . . . . .	170
22.2 System properties . . . . .	171
22.3 The main result: above set decreases . . . . .	172
<b>23 Progress Set Examples</b>	<b>175</b>
23.1 The Composition of Two Single-Assignment Programs . . . . .	175
23.1.1 Calculating <i>wens_set FF {k..}</i> . . . . .	175
23.1.2 Proving $FF \in UNIV \mapsto \{k..\}$ . . . . .	176
<b>24 Common Declarations for Chandy and Charpentier's Allocator</b>	<b>177</b>
24.1 State definitions. OUTPUT variables are locals . . . . .	178
24.1.1 Resource allocation system specification . . . . .	179
24.1.2 Client specification (required) . . . . .	179
24.1.3 Allocator specification (required) . . . . .	180
24.1.4 Network specification . . . . .	181
24.1.5 State mappings . . . . .	182
24.1.6 bijectivity of <i>sysOfClient</i> . . . . .	184
24.1.7 bijectivity of <i>client_map</i> . . . . .	185
24.2 o-simprules for <i>sysOfAlloc</i> [MUST BE AUTOMATED] . . . . .	186
24.3 o-simprules for <i>sysOfClient</i> [MUST BE AUTOMATED] . . . . .	186
24.4 Components Lemmas [MUST BE AUTOMATED] . . . . .	189
24.5 Proof of the safety property (1) . . . . .	193
24.6 Proof of the progress property (2) . . . . .	194
<b>25 Implementation of a multiple-client allocator from a single-client allocator</b>	<b>197</b>
25.1 Theorems for Merge . . . . .	201
25.2 Theorems for Distributor . . . . .	202
25.3 Theorems for Allocator . . . . .	203
<b>26 Distributed Resource Management System: the Client</b>	<b>204</b>

## 27 Projections of State Sets 208

27.1 Safety . . . . .	209
27.2 "projecting" and union/intersection (no converses) . . . . .	210
27.3 Reachability and project . . . . .	212
27.4 Converse results for weak safety: benefits of the argument C . . .	213
27.5 A lot of redundant theorems: all are proved to facilitate reasoning about guarantees. . . . .	214
27.6 leadsETo in the precondition (??) . . . . .	215
27.6.1 transient . . . . .	215
27.6.2 ensures – a primitive combining progress with safety . . .	216
27.7 Towards the theorem <i>project_Ensures_D</i> . . . . .	218
27.8 Guarantees . . . . .	218
27.9 guarantees corollaries . . . . .	219
27.9.1 Some could be deleted: the required versions are easy to prove . . . . .	219
27.9.2 Guarantees with a leadsTo postcondition . . . . .	220

## 28 Progress Under Allowable Sets 220



# 1 The Basic UNITY Theory

theory UNITY imports Main begin

definition

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

```
typedef 'a program = "Program :: ('a set * ('a * 'a) set set * ('a * 'a) set
set) set"
```

```
morphisms Rep_Program Abs_Program
unfolding Program_def by blast
```

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 == ⋂ {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  $\leq$

```
"increasing f == ⋂ z. stable {s. z ≤ f s}"
```

### 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"
apply (cut_tac x = F in Rep_Program)
apply (auto simp add: program_typedef)
done

lemma insert_Id_Acts [iff]: "insert Id (Acts F) = Acts F"
by (simp add: insert_absorb)

lemma Acts_nonempty [simp]: "Acts F ≠ {}"
by auto

lemma Id_in_AllowedActs [iff]: "Id ∈ AllowedActs F"
apply (cut_tac x = F in Rep_Program)
apply (auto simp add: program_typedef)
done

lemma insert_Id_AllowedActs [iff]: "insert Id (AllowedActs F) = AllowedActs F"
by (simp add: insert_absorb)

```

### 1.0.2 Inspectors for type "program"

```

lemma Init_eq [simp]: "Init (mk_program (init,acts,allowed)) = init"
by (simp add: program_typedef)

lemma Acts_eq [simp]: "Acts (mk_program (init,acts,allowed)) = insert Id acts"
by (simp add: program_typedef)

lemma AllowedActs_eq [simp]:
  "AllowedActs (mk_program (init,acts,allowed)) = insert Id allowed"
by (simp add: program_typedef)

```

### 1.0.3 Equality for UNITY programs

```

lemma surjective_mk_program [simp]:
  "mk_program (Init F, Acts F, AllowedActs F) = F"
apply (cut_tac x = F in Rep_Program)
apply (auto simp add: program_typedef)
apply (drule_tac f = Abs_Program in arg_cong)+
apply (simp add: program_typedef insert_absorb)
done

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

  ==> F = G"
apply (rule_tac t = F in surjective_mk_program [THEN subst])
apply (rule_tac t = G in surjective_mk_program [THEN subst], simp)

```



done

```
lemma program_equalityE:
  "[| F = G;
    [| Init F = Init G; Acts F = Acts G; AllowedActs F = AllowedActs G
    |]
    ==> P |] ==> P"
by simp
```

```
lemma program_equality_iff:
  "(F=G) =
    (Init F = Init G & Acts F = Acts G & AllowedActs F = AllowedActs G)"
by (blast intro: program_equalityI program_equalityE)
```

#### 1.0.4 co

```
lemma constrainsI:
  "(!act s s'. [| act ∈ Acts F; (s,s') ∈ act; s ∈ A |] ==> s' ∈ A')
  ==> F ∈ A co A'"
by (simp add: constrains_def, blast)
```

```
lemma constrainsD:
  "[| F ∈ A co A'; act ∈ Acts F; (s,s') ∈ act; s ∈ A |] ==> s' ∈ A'"
by (unfold constrains_def, blast)
```

```
lemma constrains_empty [iff]: "F ∈ {} co B"
by (unfold constrains_def, blast)
```

```
lemma constrains_empty2 [iff]: "(F ∈ A co {}) = (A={})"
by (unfold constrains_def, blast)
```

```
lemma constrains_UNIV [iff]: "(F ∈ UNIV co B) = (B = UNIV)"
by (unfold constrains_def, blast)
```

```
lemma constrains_UNIV2 [iff]: "F ∈ A co UNIV"
by (unfold constrains_def, blast)
```

monotonic in 2nd argument

```
lemma constrains_weaken_R:
  "[| F ∈ A co A'; A' ≤ B' |] ==> F ∈ A co B'"
by (unfold constrains_def, blast)
```

anti-monotonic in 1st argument

```
lemma constrains_weaken_L:
  "[| F ∈ A co A'; B ⊆ A |] ==> F ∈ B co A'"
by (unfold constrains_def, blast)
```

```
lemma constrains_weaken:
  "[| F ∈ A co A'; B ⊆ A; A' ≤ B' |] ==> F ∈ B co B'"
by (unfold constrains_def, blast)
```

#### 1.0.5 Union

```
lemma constrains_Un:
```

"[|  $F \in A \text{ co } A'$ ;  $F \in B \text{ co } B'$  |]  $\implies F \in (A \cup B) \text{ co } (A' \cup B')$ "  
 by (unfold constrains\_def, blast)

**lemma** constrains\_UN:  
 "(!!i.  $i \in I \implies F \in (A \ i) \text{ co } (A' \ i)$ )  
 $\implies F \in (\bigcup i \in I. A \ i) \text{ co } (\bigcup i \in I. A' \ i)$ "  
 by (unfold constrains\_def, blast)

**lemma** constrains\_Un\_distrib: " $(A \cup B) \text{ co } C = (A \text{ co } C) \cap (B \text{ co } C)$ "  
 by (unfold constrains\_def, blast)

**lemma** constrains\_UN\_distrib: " $(\bigcup i \in I. A \ i) \text{ co } B = (\bigcap i \in I. A \ i \text{ co } B)$ "  
 by (unfold constrains\_def, blast)

**lemma** constrains\_Int\_distrib: " $C \text{ co } (A \cap B) = (C \text{ co } A) \cap (C \text{ co } B)$ "  
 by (unfold constrains\_def, blast)

**lemma** constrains\_INT\_distrib: " $A \text{ co } (\bigcap i \in I. B \ i) = (\bigcap i \in I. A \text{ co } B \ i)$ "  
 by (unfold constrains\_def, blast)

### 1.0.6 Intersection

**lemma** constrains\_Int:  
 "[|  $F \in A \text{ co } A'$ ;  $F \in B \text{ co } B'$  |]  $\implies F \in (A \cap B) \text{ co } (A' \cap B')$ "  
 by (unfold constrains\_def, blast)

**lemma** constrains\_INT:  
 "(!!i.  $i \in I \implies F \in (A \ i) \text{ co } (A' \ i)$ )  
 $\implies F \in (\bigcap i \in I. A \ i) \text{ co } (\bigcap i \in I. A' \ i)$ "  
 by (unfold constrains\_def, blast)

**lemma** constrains\_imp\_subset: " $F \in A \text{ co } A' \implies A \subseteq A'$ "  
 by (unfold constrains\_def, auto)

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

**lemma** constrains\_trans:  
 "[|  $F \in A \text{ co } B$ ;  $F \in B \text{ co } C$  |]  $\implies F \in A \text{ co } C$ "  
 by (unfold constrains\_def, blast)

**lemma** constrains\_cancel:  
 "[|  $F \in A \text{ co } (A' \cup B)$ ;  $F \in B \text{ co } B'$  |]  $\implies F \in A \text{ co } (A' \cup B')$ "  
 by (unfold constrains\_def, clarify, blast)

### 1.0.7 unless

**lemma** unlessI: " $F \in (A-B) \text{ co } (A \cup B) \implies F \in A \text{ unless } B$ "  
 by (unfold unless\_def, assumption)

**lemma** unlessD: " $F \in A \text{ unless } B \implies F \in (A-B) \text{ co } (A \cup B)$ "  
 by (unfold unless\_def, assumption)

### 1.0.8 stable

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

```
by (unfold stable_def, assumption)
```

```
lemma stableD: "F ∈ stable A ==> F ∈ A co A"
by (unfold stable_def, assumption)
```

```
lemma stable_UNIV [simp]: "stable UNIV = UNIV"
by (unfold stable_def constrains_def, auto)
```

### 1.0.9 Union

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

```
apply (unfold stable_def)
apply (blast intro: constrains_Un)
done
```

```
lemma stable_UN:
  "(!!i. i ∈ I ==> F ∈ stable (A i)) ==> F ∈ stable (⋃ i ∈ I. A i)"
apply (unfold stable_def)
apply (blast intro: constrains_UN)
done
```

```
lemma stable_Union:
  "(!!A. A ∈ X ==> F ∈ stable A) ==> F ∈ stable (⋃ X)"
by (unfold stable_def constrains_def, blast)
```

### 1.0.10 Intersection

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

```
apply (unfold stable_def)
apply (blast intro: constrains_Int)
done
```

```
lemma stable_INT:
  "(!!i. i ∈ I ==> F ∈ stable (A i)) ==> F ∈ stable (⋂ i ∈ I. A i)"
apply (unfold stable_def)
apply (blast intro: constrains_INT)
done
```

```
lemma stable_Inter:
  "(!!A. A ∈ X ==> F ∈ stable A) ==> F ∈ stable (⋂ X)"
by (unfold stable_def constrains_def, blast)
```

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

```
by (unfold stable_def constrains_def, blast)
```

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

```
by (unfold stable_def constrains_def, blast)
```

```
lemmas stable_constrains_stable = stable_constrains_Int[THEN stableI]
```

### 1.0.11 invariant

```
lemma invariantI: "[| Init F  $\subseteq$  A; F  $\in$  stable A |] ==> F  $\in$  invariant A"
by (simp add: invariant_def)
```

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

```
lemma invariant_Int:
  "[| F  $\in$  invariant A; F  $\in$  invariant B |] ==> F  $\in$  invariant (A  $\cap$  B)"
by (auto simp add: invariant_def stable_Int)
```

### 1.0.12 increasing

```
lemma increasingD:
  "F  $\in$  increasing f ==> F  $\in$  stable {s. z  $\subseteq$  f s}"
by (unfold increasing_def, blast)
```

```
lemma increasing_constant [iff]: "F  $\in$  increasing (%s. c)"
by (unfold increasing_def stable_def, auto)
```

```
lemma mono_increasing_o:
  "mono g ==> increasing f  $\subseteq$  increasing (g o f)"
apply (unfold increasing_def stable_def constrains_def, auto)
apply (blast intro: monoD order_trans)
done
```

```
lemma strict_increasingD:
  "!!z::nat. F  $\in$  increasing f ==> F  $\in$  stable {s. z < f s}"
by (simp add: increasing_def Suc_le_eq [symmetric])
```

```
lemma elimination:
  "[|  $\forall m \in M. F \in \{s. s \ x = m\}$  co (B m) |]
  ==> F  $\in \{s. s \ x \in M\}$  co ( $\bigcup_{m \in M. B \ m}$ )"
by (unfold constrains_def, blast)
```

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

```
lemma elimination_sing:
  "( $\forall m \in M. F \in \{m\}$  co (B m)) ==> F  $\in M$  co ( $\bigcup_{m \in M. B \ m}$ )"
by (unfold constrains_def, blast)
```

### 1.0.13 Theoretical Results from Section 6

```
lemma constrains_strongest_rhs:
  "F  $\in A$  co (strongest_rhs F A)"
by (unfold constrains_def strongest_rhs_def, blast)
```

```
lemma strongest_rhs_is_strongest:
  "F  $\in A$  co B ==> strongest_rhs F A  $\subseteq B$ "
by (unfold constrains_def strongest_rhs_def, blast)
```

### 1.0.14 Ad-hoc set-theory rules

**lemma** *Un\_Diff\_Diff* [simp]: " $A \cup B - (A - B) = B$ "  
**by** blast

**lemma** *Int\_Union\_Union*: " $\bigcup B \cap A = \bigcup ((\%C. C \cap A) 'B)$ "  
**by** blast

Needed for WF reasoning in WFair.thy

**lemma** *Image\_less\_than* [simp]: " $\text{less\_than } \{k\} = \text{greaterThan } k$ "  
**by** blast

**lemma** *Image\_inverse\_less\_than* [simp]: " $\text{less\_than}^{-1} \{k\} = \text{lessThan } k$ "  
**by** blast

## 1.1 Partial versus Total Transitions

**definition** *totalize\_act* ::  $(\text{'a} * \text{'a})\text{set} \Rightarrow (\text{'a} * \text{'a})\text{set}$  **where**  
 $\text{"totalize\_act act == act } \cup \text{ Id\_on } (\neg(\text{Domain act}))\text{"}$

**definition** *totalize* ::  $\text{'a program} \Rightarrow \text{'a program}$  **where**  
 $\text{"totalize F == mk\_program (Init F,}$   
 $\text{totalize\_act ' Acts F,}$   
 $\text{AllowedActs F})\text{"}$

**definition** *mk\_total\_program* ::  $(\text{'a set} * (\text{'a} * \text{'a})\text{set set} * (\text{'a} * \text{'a})\text{set set})$   
 $\Rightarrow \text{'a program}$  **where**  
 $\text{"mk\_total\_program args == totalize (mk\_program args)}\text{"}$

**definition** *all\_total* ::  $\text{'a program} \Rightarrow \text{bool}$  **where**  
 $\text{"all\_total F == } \forall \text{ act } \in \text{Acts F. Domain act = UNIV}"$

**lemma** *insert\_Id\_image\_Acts*: " $f \text{ Id} = \text{Id} \Rightarrow \text{insert Id } (f ' \text{Acts F}) = f ' \text{Acts F}$ "  
**by** (blast intro: sym [THEN image\_eqI])

### 1.1.1 Basic properties

**lemma** *totalize\_act\_Id* [simp]: " $\text{totalize\_act Id} = \text{Id}$ "  
**by** (simp add: totalize\_act\_def)

**lemma** *Domain\_totalize\_act* [simp]: " $\text{Domain } (\text{totalize\_act act}) = \text{UNIV}$ "  
**by** (auto simp add: totalize\_act\_def)

**lemma** *Init\_totalize* [simp]: " $\text{Init } (\text{totalize F}) = \text{Init F}$ "  
**by** (unfold totalize\_def, auto)

**lemma** *Acts\_totalize* [simp]: " $\text{Acts } (\text{totalize F}) = (\text{totalize\_act ' Acts F})$ "  
**by** (simp add: totalize\_def insert\_Id\_image\_Acts)

**lemma** *AllowedActs\_totalize* [simp]: " $\text{AllowedActs } (\text{totalize F}) = \text{AllowedActs F}$ "  
**by** (simp add: totalize\_def)

```

lemma totalize_constrains_iff [simp]: "(totalize F ∈ A co B) = (F ∈ A co B)"
by (simp add: totalize_def totalize_act_def constrains_def, blast)

lemma totalize_stable_iff [simp]: "(totalize F ∈ stable A) = (F ∈ stable A)"
by (simp add: stable_def)

lemma totalize_invariant_iff [simp]:
  "(totalize F ∈ invariant A) = (F ∈ invariant A)"
by (simp add: invariant_def)

lemma all_total_totalize: "all_total (totalize F)"
by (simp add: totalize_def all_total_def)

lemma Domain_iff_totalize_act: "(Domain act = UNIV) = (totalize_act act = act)"
by (force simp add: totalize_act_def)

lemma all_total_imp_totalize: "all_total F ==> (totalize F = F)"
apply (simp add: all_total_def totalize_def)
apply (rule program_equalityI)
  apply (simp_all add: Domain_iff_totalize_act image_def)
done

lemma all_total_iff_totalize: "all_total F = (totalize F = F)"
apply (rule iffI)
  apply (erule all_total_imp_totalize)
  apply (erule subst)
  apply (rule all_total_totalize)
done

lemma mk_total_program_constrains_iff [simp]:
  "(mk_total_program args ∈ A co B) = (mk_program args ∈ A co B)"
by (simp add: mk_total_program_def)

```

## 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"
by (simp add: mk_total_program_def)

lemma def_prg_Acts:
  "F = mk_total_program (init,acts,allowed)
  ==> Acts F = insert Id (totalize_act ' acts)"
by (simp add: mk_total_program_def)

lemma def_prg_AllowedActs:
  "F = mk_total_program (init,acts,allowed)
  ==> AllowedActs F = insert Id allowed"
by (simp add: mk_total_program_def)

```

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'"
by (simp add: mk_total_program_def)

```

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

```

lemma def_set_simp: "A = B ==> (x ∈ A) = (x ∈ B)"
by (simp add: mk_total_program_def)

```

### 1.2.1 Inspectors for type "program"

```

lemma Init_total_eq [simp]:
  "Init (mk_total_program (init,acts,allowed)) = init"
by (simp add: mk_total_program_def)

lemma Acts_total_eq [simp]:
  "Acts(mk_total_program(init,acts,allowed)) = insert Id (totalize_act'acts)"
by (simp add: mk_total_program_def)

lemma AllowedActs_total_eq [simp]:
  "AllowedActs (mk_total_program (init,acts,allowed)) = insert Id allowed"
by (auto simp add: mk_total_program_def)

end

```

## 2 Fixed Point of a Program

theory FP imports UNITY begin

```

definition FP_Orig :: "'a program => 'a set" where
  "FP_Orig F == ⋃{A. ∀B. F ∈ stable (A ∩ 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)"
apply (simp only: FP_Orig_def stable_def Int_Union2)
apply (blast intro: constrains_UN)
done

```

```

lemma FP_Orig_weakest:
  "(⋂B. F ∈ stable (A ∩ B)) ==> A <= FP_Orig F"
by (simp add: FP_Orig_def stable_def, blast)

```

```

lemma stable_FP_Int: "F ∈ stable (FP F ∩ B)"
proof -
  have "F ∈ stable (⋃x∈B. FP F ∩ {x})"
  apply (simp only: Int_insert_right FP_def stable_def)
  apply (rule constrains_UN)
  apply simp
  done
  also have "(⋃x∈B. FP F ∩ {x}) = FP F ∩ B"
  by simp
  finally show ?thesis .

```

qed

```

lemma FP_equivalence: "FP F = FP_Orig F"
apply (rule equalityI)
  apply (rule stable_FP_Int [THEN FP_Orig_weakest])
apply (simp add: FP_Orig_def FP_def, clarify)
apply (drule_tac x = "{x}" in spec)
apply (simp add: Int_insert_right)
done

lemma FP_weakest:
  "( $\bigwedge B. F \in \text{stable } (A \text{ Int } B) \implies A \leq FP F$ )"
by (simp add: FP_equivalence FP_Orig_weakest)

lemma Compl_FP:
  "~(FP F) = (UN act: Acts F. ~{s. act '{s} <= {s}})"
by (simp add: FP_def stable_def constrains_def, blast)

lemma Diff_FP: "A - (FP F) = (UN act: Acts F. A - {s. act '{s} <= {s}})"
by (simp add: Diff_eq Compl_FP)

lemma totalize_FP [simp]: "FP (totalize F) = FP F"
by (simp add: FP_def)

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.  $\exists \text{act} \in \text{Acts } F. \text{act} \text{ `` } A \subseteq -A$ }"

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 ==  $\bigcup \{A. F \in A \text{ leadsTo } B\}$ "

notation leadsTo (infixl <⟶> 60)

```

### 3.1 transient

```

lemma stable_transient:
  "[| F  $\in$  stable A; F  $\in$  transient A |] ==>  $\exists \text{act} \in \text{Acts } F. A \subseteq - (\text{Domain } \text{act})$ "
apply (simp add: stable_def constrains_def transient_def, clarify)
apply (rule rev_bexI, auto)
done

lemma stable_transient_empty:
  "[| F  $\in$  stable A; F  $\in$  transient A; all_total F |] ==> A = {}"
apply (drule stable_transient, assumption)
apply (simp add: all_total_def)
done

lemma transient_strengthen:
  "[| F  $\in$  transient A; B  $\subseteq$  A |] ==> F  $\in$  transient B"
apply (unfold transient_def, clarify)
apply (blast intro!: rev_bexI)
done

```

```

lemma transientI:
  "[| act ∈ Acts F; act' 'A ⊆ -A |] ==> F ∈ transient A"
by (unfold transient_def, blast)

```

```

lemma transientE:
  "[| F ∈ transient A;
    ∧ act. [| act ∈ Acts F; act' 'A ⊆ -A |] ==> P |]
  ==> P"
by (unfold transient_def, blast)

```

```

lemma transient_empty [simp]: "transient {} = UNIV"
by (unfold transient_def, auto)

```

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

```

lemma totalize_transient_iff:
  "(totalize F ∈ transient A) = (∃ act ∈ Acts F. A ⊆ Domain act & act' 'A ⊆
  -A)"
apply (simp add: totalize_def totalize_act_def transient_def
  Un_Image, safe)
apply (blast intro!: rev_bexI)+
done

```

```

lemma totalize_transientI:
  "[| act ∈ Acts F; A ⊆ Domain act; act' 'A ⊆ -A |]
  ==> totalize F ∈ transient A"
by (simp add: totalize_transient_iff, blast)

```

### 3.2 ensures

```

lemma ensuresI:
  "[| F ∈ (A-B) co (A ∪ B); F ∈ transient (A-B) |] ==> F ∈ A ensures B"
by (unfold ensures_def, blast)

```

```

lemma ensuresD:
  "F ∈ A ensures B ==> F ∈ (A-B) co (A ∪ B) & F ∈ transient (A-B)"
by (unfold ensures_def, blast)

```

```

lemma ensures_weaken_R:
  "[| F ∈ A ensures A'; A' ≤ B' |] ==> F ∈ A ensures B'"
apply (unfold ensures_def)
apply (blast intro: constrains_weaken transient_strengthen)
done

```

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

```

lemma stable_ensures_Int:
  "[| F ∈ stable C; F ∈ A ensures B |]
  ==> F ∈ (C ∩ A) ensures (C ∩ B)"
apply (unfold ensures_def)
apply (auto simp add: ensures_def Int_Un_distrib [symmetric] Diff_Int_distrib
  [symmetric])
prefer 2 apply (blast intro: transient_strengthen)

```

```

apply (blast intro: stable_constrains_Int constrains_weaken)
done

lemma stable_transient_ensures:
  "[| F ∈ stable A; F ∈ transient C; A ⊆ B ∪ C |] ==> F ∈ A ensures
  B"
apply (simp add: ensures_def stable_def)
apply (blast intro: constrains_weaken transient_strengthen)
done

lemma ensures_eq: "(A ensures B) = (A unless B) ∩ transient (A-B)"
by (simp (no_asm) add: ensures_def unless_def)

```

### 3.3 leadsTo

```

lemma leadsTo_Basis [intro]: "F ∈ A ensures B ==> F ∈ A leadsTo B"
apply (unfold leadsTo_def)
apply (blast intro: leads.Basis)
done

lemma leadsTo_Trans:
  "[| F ∈ A leadsTo B; F ∈ B leadsTo C |] ==> F ∈ A leadsTo C"
apply (unfold leadsTo_def)
apply (blast intro: leads.Trans)
done

lemma leadsTo_Basis':
  "[| F ∈ A co A ∪ B; F ∈ transient A |] ==> F ∈ A leadsTo B"
apply (drule_tac B = "A-B" in constrains_weaken_L)
apply (drule_tac [2] B = "A-B" in transient_strengthen)
apply (rule_tac [3] ensuresI [THEN leadsTo_Basis])
apply (blast+)
done

lemma transient_imp_leadsTo: "F ∈ transient A ==> F ∈ A leadsTo (-A)"
by (simp (no_asm_simp) add: leadsTo_Basis ensuresI Compl_partition)

Useful with cancellation, disjunction

lemma leadsTo_Un_duplicate: "F ∈ A leadsTo (A' ∪ A') ==> F ∈ A leadsTo A'"
by (simp add: Un_ac)

lemma leadsTo_Un_duplicate2:
  "F ∈ A leadsTo (A' ∪ C ∪ C) ==> F ∈ A leadsTo (A' ∪ C)"
by (simp add: Un_ac)

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

lemma leadsTo_Union:
  "(!!A. A ∈ S ==> F ∈ A leadsTo B) ==> F ∈ (⋃ S) leadsTo B"
apply (unfold leadsTo_def)
apply (blast intro: leads.Union)
done

lemma leadsTo_Union_Int:
  "(!!A. A ∈ S ==> F ∈ (A ∩ C) leadsTo B) ==> F ∈ (⋃ S ∩ C) leadsTo B"

```

```

apply (unfold leadsTo_def)
apply (simp only: Int_Union_Union)
apply (blast intro: leads.Union)
done

lemma leadsTo_UN:
  "(!!i. i ∈ I ==> F ∈ (A i) leadsTo B) ==> F ∈ (⋃ i ∈ I. A i) leadsTo B"
apply (blast intro: leadsTo_Union)
done

```

Binary union introduction rule

```

lemma leadsTo_Un:
  "[| F ∈ A leadsTo C; F ∈ B leadsTo C |] ==> F ∈ (A ∪ B) leadsTo C"
  using leadsTo_Union [of "{A, B}" F C] by auto

lemma single_leadsTo_I:
  "(!!x. x ∈ A ==> F ∈ {x} leadsTo B) ==> F ∈ A leadsTo B"
by (subst UN_singleton [symmetric], rule leadsTo_UN, blast)

```

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 (⋃ S) B
  |] ==> P za zb"
apply (unfold leadsTo_def)
apply (drule CollectD, erule leads.induct)
apply (blast+)
done

```

```

lemma subset_imp_ensures: "A ⊆ B ==> F ∈ A ensures B"
by (unfold ensures_def constrains_def transient_def, blast)

```

```

lemmas subset_imp_leadsTo = subset_imp_ensures [THEN leadsTo_Basis]

```

```

lemmas leadsTo_refl = subset_refl [THEN subset_imp_leadsTo]

```

```

lemmas empty_leadsTo = empty_subsetI [THEN subset_imp_leadsTo, simp]

```

```

lemmas leadsTo_UNIV = subset_UNIV [THEN subset_imp_leadsTo, 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 (⋃ S)
  |] ==> P za"

```

by induction on this formula

```
apply (subgoal_tac "P zb --> P za")
```

now solve first subgoal: this formula is sufficient

```
apply (blast intro: leadsTo_refl)
```

```
apply (erule leadsTo_induct)
```

```
apply (blast+)
```

```
done
```

```
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 (⋃ S)
```

```
  |] ==> P za"
```

```
apply (subgoal_tac "F ∈ za leadsTo zb & P za")
```

```
apply (erule conjunct2)
```

```
apply (erule leadsTo_induct_pre_lemma)
```

```
prefer 3 apply (blast intro: leadsTo_Union)
```

```
prefer 2 apply (blast intro: leadsTo_Trans)
```

```
apply (blast intro: leadsTo_refl)
```

```
done
```

```
lemma leadsTo_weaken_R: "[| F ∈ A leadsTo A'; A' ≤ B' |] ==> F ∈ A leadsTo B'"
```

```
by (blast intro: subset_imp_leadsTo leadsTo_Trans)
```

```
lemma leadsTo_weaken_L:
```

```
"[| F ∈ A leadsTo A'; B ⊆ A |] ==> F ∈ B leadsTo A'"
```

```
by (blast intro: leadsTo_Trans subset_imp_leadsTo)
```

Distributes over binary unions

```
lemma leadsTo_Un_distrib:
```

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

```
by (blast intro: leadsTo_Un leadsTo_weaken_L)
```

```
lemma leadsTo_UN_distrib:
```

```
"F ∈ (⋃ i ∈ I. A i) leadsTo B = (∀ i ∈ I. F ∈ (A i) leadsTo B)"
```

```
by (blast intro: leadsTo_UN leadsTo_weaken_L)
```

```
lemma leadsTo_Union_distrib:
```

```
"F ∈ (⋃ S) leadsTo B = (∀ A ∈ S. F ∈ A leadsTo B)"
```

```
by (blast intro: leadsTo_Union leadsTo_weaken_L)
```

```
lemma leadsTo_weaken:
```

```
"[| F ∈ A leadsTo A'; B ⊆ A; A' ≤ B' |] ==> F ∈ B leadsTo B'"
```

```
by (blast intro: leadsTo_weaken_R leadsTo_weaken_L leadsTo_Trans)
```

Set difference: maybe combine with leadsTo\_weaken\_L??

```
lemma leadsTo_Diff:
```

```
"[| F ∈ (A-B) leadsTo C; F ∈ B leadsTo C |] ==> F ∈ A leadsTo C"
```

```
by (blast intro: leadsTo_Un leadsTo_weaken)
```

```

lemma leadsTo_UN_UN:
  "(!! i. i ∈ I ==> F ∈ (A i) leadsTo (A' i))
   ==> F ∈ (⋃ i ∈ I. A i) leadsTo (⋃ i ∈ I. A' i)"
apply (blast intro: leadsTo_Union leadsTo_weaken_R)
done

```

Binary union version

```

lemma leadsTo_Un_Un:
  "[| F ∈ A leadsTo A'; F ∈ B leadsTo B' |]
   ==> F ∈ (A ∪ B) leadsTo (A' ∪ B')"
by (blast intro: leadsTo_Un leadsTo_weaken_R)

```

```

lemma leadsTo_cancel2:
  "[| F ∈ A leadsTo (A' ∪ B); F ∈ B leadsTo B' |]
   ==> F ∈ A leadsTo (A' ∪ B')"
by (blast intro: leadsTo_Un_Un subset_imp_leadsTo leadsTo_Trans)

```

```

lemma leadsTo_cancel_Diff2:
  "[| F ∈ A leadsTo (A' ∪ B); F ∈ (B-A') leadsTo B' |]
   ==> F ∈ A leadsTo (A' ∪ B')"
apply (rule leadsTo_cancel2)
prefer 2 apply assumption
apply (simp_all (no_asm_simp))
done

```

```

lemma leadsTo_cancel1:
  "[| F ∈ A leadsTo (B ∪ A'); F ∈ B leadsTo B' |]
   ==> F ∈ A leadsTo (B' ∪ A')"
apply (simp add: Un_commute)
apply (blast intro!: leadsTo_cancel2)
done

```

```

lemma leadsTo_cancel_Diff1:
  "[| F ∈ A leadsTo (B ∪ A'); F ∈ (B-A') leadsTo B' |]
   ==> F ∈ A leadsTo (B' ∪ A')"
apply (rule leadsTo_cancel1)
prefer 2 apply assumption
apply (simp_all (no_asm_simp))
done

```

The impossibility law

```

lemma leadsTo_empty: "[| F ∈ A leadsTo {}; all_total F |] ==> A={} "
apply (erule leadsTo_induct_pre)
apply (simp_all add: ensures_def constrains_def transient_def all_total_def,
clarify)
apply (drule bspec, assumption)+
apply blast
done

```

### 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)"
  apply (unfold stable_def)
  apply (erule leadsTo_induct)
  prefer 3 apply (blast intro: leadsTo_Union_Int)
  prefer 2 apply (blast intro: leadsTo_Trans)
  apply (rule leadsTo_Basis)
  apply (simp add: ensures_def Diff_Int_distrib2 [symmetric] Int_Un_distrib2
    [symmetric])
  apply (blast intro: transient_strengthen constrains_Int)
  done

lemma psp_stable2:
  "[| F ∈ A leadsTo A'; F ∈ stable B |] ==> F ∈ (B ∩ A) leadsTo (B ∩ A')"
  by (simp add: psp_stable Int_ac)

lemma psp Ensures:
  "[| F ∈ A ensures A'; F ∈ B co B' |]
   ==> F ∈ (A ∩ B') ensures ((A' ∩ B) ∪ (B' - B))"
  apply (unfold ensures_def constrains_def, clarify)
  apply (blast intro: transient_strengthen)
  done

lemma psp:
  "[| F ∈ A leadsTo A'; F ∈ B co B' |]
   ==> F ∈ (A ∩ B') leadsTo ((A' ∩ B) ∪ (B' - B))"
  apply (erule leadsTo_induct)
  prefer 3 apply (blast intro: leadsTo_Union_Int)

Basis case
  apply (blast intro: psp Ensures)

Transitivity case has a delicate argument involving "cancellation"
  apply (rule leadsTo_Un_duplicate2)
  apply (erule leadsTo_cancel_Diff1)
  apply (simp add: Int_Diff Diff_triv)
  apply (blast intro: leadsTo_weaken_L dest: constrains_imp_subset)
  done

lemma psp2:
  "[| F ∈ A leadsTo A'; F ∈ B co B' |]
   ==> F ∈ (B' ∩ A) leadsTo ((B ∩ A') ∪ (B' - B))"
  by (simp (no_asm_simp) add: psp Int_ac)

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

  apply (unfold unless_def)
  apply (drule psp, assumption)

```

```

apply (blast intro: leadsTo_weaken)
done

```

### 3.5 Proving the induction rules

```

lemma leadsTo_wf_induct_lemma:
  "[| wf r;
     $\forall m. F \in (A \cap f^{-'\{m\}}) \text{ leadsTo } ((A \cap f^{-'(r^{-1} \text{ `` } \{m\})}) \cup B) \text{ |}]$ 
    ==>  $F \in (A \cap f^{-'\{m\}}) \text{ leadsTo } B$ "
  apply (erule_tac a = m in wf_induct)
  apply (subgoal_tac "F ∈ (A ∩ (f -' (r-1 `` {x}))) leadsTo B")
  apply (blast intro: leadsTo_cancel1 leadsTo_Un_duplicate)
  apply (subst vimage_eq_UN)
  apply (simp only: UN_simps [symmetric])
  apply (blast intro: leadsTo_UN)
done

```

```

lemma leadsTo_wf_induct:
  "[| wf r;
     $\forall m. F \in (A \cap f^{-'\{m\}}) \text{ leadsTo } ((A \cap f^{-'(r^{-1} \text{ `` } \{m\})}) \cup B) \text{ |}]$ 
    ==>  $F \in A \text{ leadsTo } B$ "
  apply (rule_tac t = A in subst)
  defer 1
  apply (rule leadsTo_UN)
  apply (erule leadsTo_wf_induct_lemma)
  apply assumption
  apply fast
done

```

```

lemma bounded_induct:
  "[| wf r;
     $\forall m \in I. F \in (A \cap f^{-'\{m\}}) \text{ leadsTo } ((A \cap f^{-'(r^{-1} \text{ `` } \{m\})}) \cup B) \text{ |}]$ 
    ==>  $F \in A \text{ leadsTo } ((A - (f^{-'I})) \cup B)$ "
  apply (erule leadsTo_wf_induct, safe)
  apply (case_tac "m ∈ I")
  apply (blast intro: leadsTo_weaken)
  apply (blast intro: subset_imp_leadsTo)
done

```

```

lemma lessThan_induct:
  "[| !!m::nat. F ∈ (A ∩ f-'{m}) leadsTo ((A ∩ f-'{..m}) ∪ B) |]
    ==> F ∈ A leadsTo B"
  apply (rule wf_less_than [THEN leadsTo_wf_induct])
  apply (simp (no_asm_simp))
  apply blast
done

```



```

lemma lessThan_bounded_induct:
  "(!l::nat. [|  $\forall m \in \text{greaterThan } l.$ 
     $F \in (A \cap f^{-1}\{m\}) \text{ leadsTo } ((A \cap f^{-1}(\text{lessThan } m)) \cup B)$  |]
     $\implies F \in A \text{ leadsTo } ((A \cap (f^{-1}(\text{atMost } l))) \cup B)$  )"
apply (simp only: Diff_eq [symmetric] vimage_Compl Compl_greaterThan [symmetric])
apply (rule wf_less_than [THEN bounded_induct])
apply (simp (no_asm_simp))
done

```

```

lemma greaterThan_bounded_induct:
  "(!l::nat.  $\forall m \in \text{lessThan } l.$ 
     $F \in (A \cap f^{-1}\{m\}) \text{ leadsTo } ((A \cap f^{-1}(\text{greaterThan } m)) \cup B)$ 
     $\implies F \in A \text{ leadsTo } ((A \cap (f^{-1}(\text{atLeast } l))) \cup B)$  )"
apply (rule_tac f = f and f1 = "%k. 1 - k"
  in wf_less_than [THEN wf_inv_image, THEN leadsTo_wf_induct])
apply (simp (no_asm) add:Image_singleton)
apply clarify
apply (case_tac "m<l")
  apply (blast intro: leadsTo_weaken_R diff_less_mono2)
  apply (blast intro: not_le_imp_less subset_imp_leadsTo)
done

```

### 3.6 wlt

Misra's property W3

```

lemma wlt_leadsTo: "F  $\in$  (wlt F B) leadsTo B"
apply (unfold wlt_def)
apply (blast intro!: leadsTo_Union)
done

```

```

lemma leadsTo_subset: "F  $\in$  A leadsTo B  $\implies A \subseteq$  wlt F B"
apply (unfold wlt_def)
apply (blast intro!: leadsTo_Union)
done

```

Misra's property W2

```

lemma leadsTo_eq_subset_wlt: "F  $\in$  A leadsTo B = (A  $\subseteq$  wlt F B)"
by (blast intro!: leadsTo_subset wlt_leadsTo [THEN leadsTo_weaken_L])

```

Misra's property W4

```

lemma wlt_increasing: "B  $\subseteq$  wlt F B"
apply (simp (no_asm_simp) add: leadsTo_eq_subset_wlt [symmetric] subset_imp_leadsTo)
done

```

Used in the Trans case below

```

lemma lemma1:
  "[| B  $\subseteq$  A2;
    F  $\in$  (A1 - B) co (A1  $\cup$  B);
    F  $\in$  (A2 - C) co (A2  $\cup$  C) |]
     $\implies F \in (A1 \cup A2 - C) \text{ co } (A1 \cup A2 \cup C)$ "
by (unfold constrains_def, clarify, blast)

```

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

```
lemma leadsTo_123:
  "F ∈ A leadsTo A'
   ==> ∃B. A ⊆ B & F ∈ B leadsTo A' & F ∈ (B-A') co (B ∪ A')"
apply (erule leadsTo_induct)
```

Basis

```
  apply (blast dest: ensuresD)
```

Trans

```
  apply clarify
  apply (rule_tac x = "Ba ∪ Bb" in exI)
  apply (blast intro: lemma1 leadsTo_Un_Un leadsTo_cancel1 leadsTo_Un_duplicate)
```

Union

```
  apply (clarify dest!: ball_conj_distrib [THEN iffD1] bchoice)
  apply (rule_tac x = "⋃ A ∈ S. f A" in exI)
  apply (auto intro: leadsTo_UN)

  apply (rule_tac I1=S and A1="%i. f i - B" and A'1="%i. f i ∪ B"
    in constrains_UN [THEN constrains_weaken], auto)
done
```

Misra's property W5

```
lemma wlt_constrains_wlt: "F ∈ (wlt F B - B) co (wlt F B)"
proof -
  from wlt_leadsTo [of F B, THEN leadsTo_123]
  show ?thesis
  proof (elim exE conjE)

    fix C
    assume wlt: "wlt F B ⊆ C"
      and lt: "F ∈ C leadsTo B"
      and co: "F ∈ C - B co C ∪ B"
    have eq: "C = wlt F B"
    proof -
      from lt and wlt show ?thesis
      by (blast dest: leadsTo_eq_subset_wlt [THEN iffD1])
    qed
    from co show ?thesis by (simp add: eq wlt_increasing Un_absorb2)
  qed
qed
```

### 3.7 Completion: Binary and General Finite versions

```
lemma completion_lemma :
  "[| W = wlt F (B' ∪ C);
   F ∈ A leadsTo (A' ∪ C); F ∈ A' co (A' ∪ C);
   F ∈ B leadsTo (B' ∪ C); F ∈ B' co (B' ∪ C) |]"
  ==> F ∈ (A ∩ B) leadsTo ((A' ∩ B') ∪ C)"
apply (subgoal_tac "F ∈ (W-C) co (W ∪ B' ∪ C) ")
prefer 2
apply (blast intro: wlt_constrains_wlt [THEN [2] constrains_Un,
```

```

                                THEN constrains_weaken])
apply (subgoal_tac "F ∈ (W-C) co W")
prefer 2
  apply (simp add: wlt_increasing Un_assoc Un_absorb2)
apply (subgoal_tac "F ∈ (A ∩ W - C) leadsTo (A' ∩ W ∪ C) ")
prefer 2 apply (blast intro: wlt_leadsTo psp [THEN leadsTo_weaken])

apply (subgoal_tac "F ∈ (A' ∩ W ∪ C) leadsTo (A' ∩ B' ∪ C) ")
prefer 2
  apply (rule leadsTo_Un_duplicate2)
  apply (blast intro: leadsTo_Un_Un wlt_leadsTo
    [THEN psp2, THEN leadsTo_weaken] leadsTo_refl)
apply (drule leadsTo_Diff)
apply (blast intro: subset_imp_leadsTo)
apply (subgoal_tac "A ∩ B ⊆ A ∩ W")
prefer 2
  apply (blast dest!: leadsTo_subset intro!: subset_refl [THEN Int_mono])
apply (blast intro: leadsTo_Trans subset_imp_leadsTo)
done

lemmas completion = completion_lemma [OF refl]

lemma finite_completion_lemma:
  "finite I ==> (∀ i ∈ I. F ∈ (A i) leadsTo (A' i ∪ C)) -->
    (∀ i ∈ I. F ∈ (A' i) co (A' i ∪ C)) -->
    F ∈ (⋂ i ∈ I. A i) leadsTo ((⋂ i ∈ I. A' i) ∪ C)"
apply (erule finite_induct, auto)
apply (rule completion)
prefer 4
  apply (simp only: INT_simps [symmetric])
  apply (rule constrains_INT, auto)
done

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)"
by (blast intro: finite_completion_lemma [THEN mp, THEN mp])

lemma stable_completion:
  "[| F ∈ A leadsTo A'; F ∈ stable A';
    F ∈ B leadsTo B'; F ∈ stable B' |]
  ==> F ∈ (A ∩ B) leadsTo (A' ∩ B')"
apply (unfold stable_def)
apply (rule_tac C1 = "{}" in completion [THEN leadsTo_weaken_R])
apply (force+)
done

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)"

```

```

apply (unfold stable_def)
apply (rule_tac C1 = "{}" in finite_completion [THEN leadsTo_weaken_R])
apply (simp_all (no_asm_simp))
apply blast+
done

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.  $\exists$  evs. (s, evs)  $\in$  traces (Init F) (Acts F)}"
apply safe
apply (erule_tac [2] traces.induct)
apply (erule reachable.induct)
apply (blast intro: reachable.intros traces.intros)+
done

lemma Init_subset_reachable: "Init F  $\subseteq$  reachable F"
by (blast intro: reachable.intros)

lemma stable_reachable [intro!, simp]:
  "Acts G  $\subseteq$  Acts F  $\implies$  G  $\in$  stable (reachable F)"
by (blast intro: stableI constrainsI reachable.intros)

lemma invariant_reachable: "F  $\in$  invariant (reachable F)"
apply (simp add: invariant_def)
apply (blast intro: reachable.intros)
done

lemma invariant_includes_reachable: "F  $\in$  invariant A  $\implies$  reachable F  $\subseteq$  A"
apply (simp add: stable_def constrains_def invariant_def)
apply (rule subsetI)
apply (erule reachable.induct)
apply (blast intro: reachable.intros)+
done



## 4.2 Co



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

lemma Constrains_eq_constrains:
  "A Co B = {F. F  $\in$  (reachable F  $\cap$  A) co (reachable F  $\cap$  B)}"
apply (unfold Constrains_def)
apply (blast dest: constrains_reachable_Int intro: constrains_weaken)
done

lemma constrains_imp_Constrains: "F  $\in$  A co A'  $\implies$  F  $\in$  A Co A'"
apply (unfold Constrains_def)
apply (blast intro: constrains_weaken_L)
done

lemma stable_imp_Stable: "F  $\in$  stable A  $\implies$  F  $\in$  Stable A"
apply (unfold stable_def Stable_def)
apply (erule constrains_imp_Constrains)
done

lemma ConstrainsI:
  "(!act s s'. [| act  $\in$  Acts F; (s, s')  $\in$  act; s  $\in$  A |]  $\implies$  s'  $\in$  A')

```

```

    ==> F ∈ A Co A'"
  apply (rule constrains_imp_Constrains)
  apply (blast intro: constrainsI)
done

lemma Constrains_empty [iff]: "F ∈ {} Co B"
by (unfold Constrains_def constrains_def, blast)

lemma Constrains_UNIV [iff]: "F ∈ A Co UNIV"
by (blast intro: ConstrainsI)

lemma Constrains_weaken_R:
  "[| F ∈ A Co A'; A' ≤ B' |] ==> F ∈ A Co B'"
  apply (unfold Constrains_def)
  apply (blast intro: constrains_weaken_R)
done

lemma Constrains_weaken_L:
  "[| F ∈ A Co A'; B ⊆ A |] ==> F ∈ B Co A'"
  apply (unfold Constrains_def)
  apply (blast intro: constrains_weaken_L)
done

lemma Constrains_weaken:
  "[| F ∈ A Co A'; B ⊆ A; A' ≤ B' |] ==> F ∈ B Co B'"
  apply (unfold Constrains_def)
  apply (blast intro: constrains_weaken)
done

lemma Constrains_Un:
  "[| F ∈ A Co A'; F ∈ B Co B' |] ==> F ∈ (A ∪ B) Co (A' ∪ B')"
  apply (unfold Constrains_def)
  apply (blast intro: constrains_Un [THEN constrains_weaken])
done

lemma Constrains_UN:
  assumes Co: "!!i. i ∈ I ==> F ∈ (A i) Co (A' i)"
  shows "F ∈ (⋃ i ∈ I. A i) Co (⋃ i ∈ I. A' i)"
  apply (unfold Constrains_def)
  apply (rule CollectI)
  apply (rule Co [unfolded Constrains_def, THEN CollectD, THEN constrains_UN,
    THEN constrains_weaken], auto)
done

lemma Constrains_Int:
  "[| F ∈ A Co A'; F ∈ B Co B' |] ==> F ∈ (A ∩ B) Co (A' ∩ B')"
  apply (unfold Constrains_def)
  apply (blast intro: constrains_Int [THEN constrains_weaken])

```

done

lemma Constrains\_INT:

assumes Co: " $\forall i. i \in I \implies F \in (A\ i)\ Co\ (A'\ i)$ "

shows " $F \in (\bigcap_{i \in I}. A\ i)\ Co\ (\bigcap_{i \in I}. A'\ i)$ "

apply (unfold Constrains\_def)

apply (rule CollectI)

apply (rule Co [unfolded Constrains\_def, THEN CollectD, THEN constrains\_INT,

THEN constrains\_weaken], auto)

done

lemma Constrains\_imp\_subset: " $F \in A\ Co\ A' \implies \text{reachable } F \cap A \subseteq A'$ "

by (simp add: constrains\_imp\_subset Constrains\_def)

lemma Constrains\_trans: " $[F \in A\ Co\ B; F \in B\ Co\ C] \implies F \in A\ Co\ C$ "

apply (simp add: Constrains\_eq\_constrains)

apply (blast intro: constrains\_trans constrains\_weaken)

done

lemma Constrains\_cancel:

" $[F \in A\ Co\ (A' \cup B); F \in B\ Co\ B'] \implies F \in A\ Co\ (A' \cup B')$ "

apply (simp add: Constrains\_eq\_constrains constrains\_def)

apply best

done

### 4.3 Stable

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

by blast

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

by (simp add: Stable\_def Constrains\_eq\_constrains stable\_def)

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

by (unfold Stable\_def, assumption)

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

by (unfold Stable\_def, assumption)

lemma Stable\_Un:

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

apply (unfold Stable\_def)

apply (blast intro: Constrains\_Un)

done

lemma Stable\_Int:

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

apply (unfold Stable\_def)

apply (blast intro: Constrains\_Int)

done

lemma Stable\_Constrains\_Un:

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

```

    ==> F ∈ (C ∪ A) Co (C ∪ A')"
  apply (unfold Stable_def)
  apply (blast intro: Constrains_Un [THEN Constrains_weaken])
done

lemma Stable_Constrains_Int:
  "[| F ∈ Stable C; F ∈ (C ∩ A) Co A' |]
   ==> F ∈ (C ∩ A) Co (C ∩ A')"
  apply (unfold Stable_def)
  apply (blast intro: Constrains_Int [THEN Constrains_weaken])
done

lemma Stable_UN:
  "(!!i. i ∈ I ==> F ∈ Stable (A i)) ==> F ∈ Stable (⋃ i ∈ I. A i)"
  by (simp add: Stable_def Constrains_UN)

lemma Stable_INT:
  "(!!i. i ∈ I ==> F ∈ Stable (A i)) ==> F ∈ Stable (⋂ i ∈ I. A i)"
  by (simp add: Stable_def Constrains_INT)

lemma Stable_reachable: "F ∈ Stable (reachable F)"
  by (simp add: Stable_eq_stable)

```

#### 4.4 Increasing

```

lemma IncreasingD:
  "F ∈ Increasing f ==> F ∈ Stable {s. x ≤ f s}"
  by (unfold Increasing_def, blast)

lemma mono_Increasing_o:
  "mono g ==> Increasing f ⊆ Increasing (g o f)"
  apply (simp add: Increasing_def Stable_def Constrains_def stable_def
    constrains_def)
  apply (blast intro: monoD order_trans)
done

lemma strict_IncreasingD:
  "!!z::nat. F ∈ Increasing f ==> F ∈ Stable {s. z < f s}"
  by (simp add: Increasing_def Suc_le_eq [symmetric])

lemma increasing_imp_Increasing:
  "F ∈ increasing f ==> F ∈ Increasing f"
  apply (unfold increasing_def Increasing_def)
  apply (blast intro: stable_imp_Stable)
done

lemmas Increasing_constant = increasing_constant [THEN increasing_imp_Increasing,
  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)"

```



```
by (unfold Constrains_def constrains_def, blast)
```

```
lemma Elimination_sing:
  "( $\forall m. F \in \{m\} \text{ Co } (B \ m)$ ) ==>  $F \in M \text{ Co } (\bigcup m \in M. B \ m)$ "
by (unfold Constrains_def constrains_def, blast)
```

## 4.6 Specialized laws for handling Always

```
lemma AlwaysI: "[| Init F  $\subseteq$  A; F  $\in$  Stable A |] ==> F  $\in$  Always A"
by (simp add: Always_def)
```

```
lemma AlwaysD: "F  $\in$  Always A ==> Init F  $\subseteq$  A & F  $\in$  Stable A"
by (simp add: Always_def)
```

```
lemmas AlwaysE = AlwaysD [THEN conjE]
lemmas Always_imp_Stable = AlwaysD [THEN conjunct2]
```

```
lemma Always_includes_reachable: "F  $\in$  Always A ==> reachable F  $\subseteq$  A"
apply (simp add: Stable_def Constrains_def constrains_def Always_def)
apply (rule subsetI)
apply (erule reachable.induct)
apply (blast intro: reachable.intros)+
done
```

```
lemma invariant_imp_Always:
  "F  $\in$  invariant A ==> F  $\in$  Always A"
apply (unfold Always_def invariant_def Stable_def stable_def)
apply (blast intro: constrains_imp_Constrains)
done
```

```
lemmas Always_reachable = invariant_reachable [THEN invariant_imp_Always]
```

```
lemma Always_eq_invariant_reachable:
  "Always A = {F. F  $\in$  invariant (reachable F  $\cap$  A)}"
apply (simp add: Always_def invariant_def Stable_def Constrains_eq_constrains
  stable_def)
apply (blast intro: reachable.intros)
done
```

```
lemma Always_eq_includes_reachable: "Always A = {F. reachable F  $\subseteq$  A}"
by (auto dest: invariant_includes_reachable simp add: Int_absorb2 invariant_reachable
  Always_eq_invariant_reachable)
```

```
lemma Always_UNIV_eq [simp]: "Always UNIV = UNIV"
by (auto simp add: Always_eq_includes_reachable)
```

```
lemma UNIV_AlwaysI: "UNIV  $\subseteq$  A ==> F  $\in$  Always A"
by (auto simp add: Always_eq_includes_reachable)
```

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

```

apply (simp add: Always_eq_includes_reachable)
apply (blast intro: invariantI Init_subset_reachable [THEN subsetD]
         invariant_includes_reachable [THEN subsetD])
done

lemma Always_weaken: "[| F ∈ Always A; A ⊆ B |] ==> F ∈ Always B"
by (auto simp add: Always_eq_includes_reachable)

```

#### 4.7 "Co" rules involving Always

```

lemma Always_Constrains_pre:
  "F ∈ Always INV ==> (F ∈ (INV ∩ A) Co A') = (F ∈ A Co A')"
by (simp add: Always_includes_reachable [THEN Int_absorb2] Constrains_def

      Int_assoc [symmetric])

lemma Always_Constrains_post:
  "F ∈ Always INV ==> (F ∈ A Co (INV ∩ A')) = (F ∈ A Co A')"
by (simp add: Always_includes_reachable [THEN Int_absorb2]
      Constrains_eq_constrains Int_assoc [symmetric])

lemmas Always_ConstrainsI = Always_Constrains_pre [THEN iffD1]

lemmas Always_ConstrainsD = Always_Constrains_post [THEN iffD2]

```

```

lemma Always_Constrains_weaken:
  "[| F ∈ Always C; F ∈ A Co A';
     C ∩ B ⊆ A; C ∩ A' ⊆ B' |]
  ==> F ∈ B Co B'"
apply (rule Always_ConstrainsI, assumption)
apply (drule Always_ConstrainsD, assumption)
apply (blast intro: Constrains_weaken)
done

```

```

lemma Always_Int_distrib: "Always (A ∩ B) = Always A ∩ Always B"
by (auto simp add: Always_eq_includes_reachable)

```

```

lemma Always_INT_distrib: "Always (⋂ (A ' I)) = (⋂ i ∈ I. Always (A i))"
by (auto simp add: Always_eq_includes_reachable)

```

```

lemma Always_Int_I:
  "[| F ∈ Always A; F ∈ Always B |] ==> F ∈ Always (A ∩ B)"
by (simp add: Always_Int_distrib)

```

```

lemma Always_Compl_Un_eq:
  "F ∈ Always A ==> (F ∈ Always (¬A ∪ B)) = (F ∈ Always B)"
by (auto simp add: Always_eq_includes_reachable)

```

```
lemmas Always_thin = thin_rl [of "F ∈ Always A"] for F A
```

## 4.8 Totalize

```
lemma reachable_imp_reachable_tot:
  "s ∈ reachable F ==> s ∈ reachable (totalize F)"
apply (erule reachable.induct)
  apply (rule reachable.Init)
  apply simp
apply (rule_tac act = "totalize_act act" in reachable.Acts)
apply (auto simp add: totalize_act_def)
done

lemma reachable_tot_imp_reachable:
  "s ∈ reachable (totalize F) ==> s ∈ reachable F"
apply (erule reachable.induct)
  apply (rule reachable.Init, simp)
apply (force simp add: totalize_act_def intro: reachable.Acts)
done

lemma reachable_tot_eq [simp]: "reachable (totalize F) = reachable F"
by (blast intro: reachable_imp_reachable_tot reachable_tot_imp_reachable)

lemma totalize_Constrains_iff [simp]: "(totalize F ∈ A Co B) = (F ∈ A Co B)"
by (simp add: Constrains_def)

lemma totalize_Stable_iff [simp]: "(totalize F ∈ Stable A) = (F ∈ Stable A)"
by (simp add: Stable_def)

lemma totalize_Always_iff [simp]: "(totalize F ∈ Always A) = (F ∈ Always A)"
by (simp add: Always_def)

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 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)}"
apply (unfold LeadsTo_def)
apply (blast dest: psp_stable2 intro: leadsTo_weaken)
done
```

### 5.1 Specialized laws for handling invariants

```
lemma Always_LeadsTo_pre:
  "F ∈ Always INV ==> (F ∈ (INV ∩ A) LeadsTo A') = (F ∈ A LeadsTo A')"
by (simp add: LeadsTo_def Always_eq_includes_reachable Int_absorb2
    Int_assoc [symmetric])
```

```
lemma Always_LeadsTo_post:
  "F ∈ Always INV ==> (F ∈ A LeadsTo (INV ∩ A')) = (F ∈ A LeadsTo A')"
by (simp add: LeadsTo_eq_leadsTo Always_eq_includes_reachable Int_absorb2
    Int_assoc [symmetric])
```

```
lemmas Always_LeadsToI = Always_LeadsTo_pre [THEN iffD1]
```

```
lemmas Always_LeadsToD = Always_LeadsTo_post [THEN iffD2]
```

### 5.2 Introduction rules: Basis, Trans, Union

```
lemma leadsTo_imp_LeadsTo: "F ∈ A leadsTo B ==> F ∈ A LeadsTo B"
apply (simp add: LeadsTo_def)
apply (blast intro: leadsTo_weaken_L)
done
```

```
lemma LeadsTo_Trans:
  "[| F ∈ A LeadsTo B; F ∈ B LeadsTo C |] ==> F ∈ A LeadsTo C"
apply (simp add: LeadsTo_eq_leadsTo)
apply (blast intro: leadsTo_Trans)
done
```

```
lemma LeadsTo_Union:
  "(!!A. A ∈ S ==> F ∈ A LeadsTo B) ==> F ∈ (⋃ S) LeadsTo B"
apply (simp add: LeadsTo_def)
apply (subst Int_Union)
apply (blast intro: leadsTo_UN)
done
```

### 5.3 Derived rules

```
lemma LeadsTo_UNIV [simp]: "F ∈ A LeadsTo UNIV"
by (simp add: LeadsTo_def)
```

Useful with cancellation, disjunction

```
lemma LeadsTo_Un_duplicate:
```

```
"F ∈ A LeadsTo (A' ∪ A') ==> F ∈ A LeadsTo A'"
by (simp add: Un_ac)
```

```
lemma LeadsTo_Un_duplicate2:
  "F ∈ A LeadsTo (A' ∪ C ∪ C) ==> F ∈ A LeadsTo (A' ∪ C)"
by (simp add: Un_ac)
```

```
lemma LeadsTo_UN:
  "(!!i. i ∈ I ==> F ∈ (A i) LeadsTo B) ==> F ∈ (⋃ i ∈ I. A i) LeadsTo
  B"
apply (blast intro: LeadsTo_Union)
done
```

Binary union introduction rule

```
lemma LeadsTo_Un:
  "[| F ∈ A LeadsTo C; F ∈ B LeadsTo C |] ==> F ∈ (A ∪ B) LeadsTo C"
  using LeadsTo_UN [of "{A, B}" F id C] by auto
```

Lets us look at the starting state

```
lemma single_LeadsTo_I:
  "(!!s. s ∈ A ==> F ∈ {s} LeadsTo B) ==> F ∈ A LeadsTo B"
by (subst UN_singleton [symmetric], rule LeadsTo_UN, blast)
```

```
lemma subset_imp_LeadsTo: "A ⊆ B ==> F ∈ A LeadsTo B"
apply (simp add: LeadsTo_def)
apply (blast intro: subset_imp_leadsTo)
done
```

```
lemmas empty_LeadsTo = empty_subsetI [THEN subset_imp_LeadsTo, simp]
```

```
lemma LeadsTo_weaken_R:
  "[| F ∈ A LeadsTo A'; A' ⊆ B' |] ==> F ∈ A LeadsTo B'"
apply (simp add: LeadsTo_def)
apply (blast intro: leadsTo_weaken_R)
done
```

```
lemma LeadsTo_weaken_L:
  "[| F ∈ A LeadsTo A'; B ⊆ A |]
  ==> F ∈ B LeadsTo A'"
apply (simp add: LeadsTo_def)
apply (blast intro: leadsTo_weaken_L)
done
```

```
lemma LeadsTo_weaken:
  "[| F ∈ A LeadsTo A';
  B ⊆ A; A' ⊆ B' |]
  ==> F ∈ B LeadsTo B'"
by (blast intro: LeadsTo_weaken_R LeadsTo_weaken_L LeadsTo_Trans)
```

```
lemma Always_LeadsTo_weaken:
  "[| F ∈ Always C; F ∈ A LeadsTo A';
  C ∩ B ⊆ A; C ∩ A' ⊆ B' |]
  ==> F ∈ B LeadsTo B'"
by (blast dest: Always_LeadsToI intro: LeadsTo_weaken intro: Always_LeadsToD)
```

```
lemma LeadsTo_Un_post: "F ∈ A LeadsTo B ==> F ∈ (A ∪ B) LeadsTo B"
by (blast intro: LeadsTo_Un subset_imp_LeadsTo)
```

```
lemma LeadsTo_Trans_Un:
  "[| F ∈ A LeadsTo B; F ∈ B LeadsTo C |]
   ==> F ∈ (A ∪ B) LeadsTo C"
by (blast intro: LeadsTo_Un subset_imp_LeadsTo LeadsTo_weaken_L LeadsTo_Trans)
```

```
lemma LeadsTo_Un_distrib:
  "(F ∈ (A ∪ B) LeadsTo C) = (F ∈ A LeadsTo C & F ∈ B LeadsTo C)"
by (blast intro: LeadsTo_Un LeadsTo_weaken_L)
```

```
lemma LeadsTo_UN_distrib:
  "(F ∈ (⋃ i ∈ I. A i) LeadsTo B) = (∀ i ∈ I. F ∈ (A i) LeadsTo B)"
by (blast intro: LeadsTo_UN LeadsTo_weaken_L)
```

```
lemma LeadsTo_Union_distrib:
  "(F ∈ (⋃ S) LeadsTo B) = (∀ A ∈ S. F ∈ A LeadsTo B)"
by (blast intro: LeadsTo_Union LeadsTo_weaken_L)
```

```
lemma LeadsTo_Basis: "F ∈ A Ensures B ==> F ∈ A LeadsTo B"
by (simp add: Ensures_def LeadsTo_def leadsTo_Basis)
```

```
lemma EnsuresI:
  "[| F ∈ (A-B) Co (A ∪ B); F ∈ transient (A-B) |]
   ==> F ∈ A Ensures B"
apply (simp add: Ensures_def Constrains_eq_constrains)
apply (blast intro: ensuresI constrains_weaken transient_strengthen)
done
```

```
lemma Always_LeadsTo_Basis:
  "[| F ∈ Always INV;
     F ∈ (INV ∩ (A-A')) Co (A ∪ A');
     F ∈ transient (INV ∩ (A-A')) |]
   ==> F ∈ A LeadsTo A'"
apply (rule Always_LeadsToI, assumption)
apply (blast intro: EnsuresI LeadsTo_Basis Always_ConstrainsD [THEN Constrains_weaken]
transient_strengthen)
done
```

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

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

by (blast intro: LeadsTo\_Un LeadsTo\_weaken)

```
lemma LeadsTo_UN_UN:
  "(!! i. i ∈ I ==> F ∈ (A i) LeadsTo (A' i))
   ==> F ∈ (⋃ i ∈ I. A i) LeadsTo (⋃ i ∈ I. A' i)"
apply (blast intro: LeadsTo_Union LeadsTo_weaken_R)
done
```

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)"
by (blast intro: LeadsTo_UN_UN)
```

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)"
by (blast intro: LeadsTo_UN_UN)
```

Binary union version

```
lemma LeadsTo_Un_Un:
  "[| F ∈ A LeadsTo A'; F ∈ B LeadsTo B' |]
   ==> F ∈ (A ∪ B) LeadsTo (A' ∪ B')"
by (blast intro: LeadsTo_Un LeadsTo_weaken_R)
```

```
lemma LeadsTo_cancel2:
  "[| F ∈ A LeadsTo (A' ∪ B); F ∈ B LeadsTo B' |]
   ==> F ∈ A LeadsTo (A' ∪ B')"
by (blast intro: LeadsTo_Un_Un subset_imp_LeadsTo LeadsTo_Trans)
```

```
lemma LeadsTo_cancel_Diff2:
  "[| F ∈ A LeadsTo (A' ∪ B); F ∈ (B-A') LeadsTo B' |]
   ==> F ∈ A LeadsTo (A' ∪ B')"
apply (rule LeadsTo_cancel2)
prefer 2 apply assumption
apply (simp_all (no_asm_simp))
done
```

```
lemma LeadsTo_cancel1:
  "[| F ∈ A LeadsTo (B ∪ A'); F ∈ B LeadsTo B' |]
   ==> F ∈ A LeadsTo (B' ∪ A')"
apply (simp add: Un_commute)
apply (blast intro!: LeadsTo_cancel2)
done
```

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

```

apply (rule LeadsTo_cancel1)
prefer 2 apply assumption
apply (simp_all (no_asm_simp))
done

```

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)"
apply (simp add: LeadsTo_def Always_eq_includes_reachable)
apply (drule leadsTo_empty, auto)
done

```

## 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)"
apply (simp add: LeadsTo_eq_leadsTo Stable_eq_stable)
apply (drule psp_stable, assumption)
apply (simp add: Int_ac)
done

```

```

lemma PSP_Stable2:
  "[| F ∈ A LeadsTo A'; F ∈ Stable B |]
   ==> F ∈ (B ∩ A) LeadsTo (B ∩ A')"
by (simp add: PSP_Stable Int_ac)

```

```

lemma PSP:
  "[| F ∈ A LeadsTo A'; F ∈ B Co B' |]
   ==> F ∈ (A ∩ B') LeadsTo ((A' ∩ B) ∪ (B' - B))"
apply (simp add: LeadsTo_def Constrains_eq_constrains)
apply (blast dest: psp intro: leadsTo_weaken)
done

```

```

lemma PSP2:
  "[| F ∈ A LeadsTo A'; F ∈ B Co B' |]
   ==> F ∈ (B' ∩ A) LeadsTo ((B ∩ A') ∪ (B' - B))"
by (simp add: PSP Int_ac)

```

```

lemma PSP_Unless:
  "[| F ∈ A LeadsTo A'; F ∈ B Unless B' |]
   ==> F ∈ (A ∩ B) LeadsTo ((A' ∩ B) ∪ B'"
apply (unfold Unless_def)
apply (drule PSP, assumption)
apply (blast intro: LeadsTo_Diff LeadsTo_weaken subset_imp_LeadsTo)
done

```

```

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

```



```

apply (erule Always_LeadsTo_weaken)
apply (rule LeadsTo_Diff)
  prefer 2
  apply (erule
    transient_imp_leadsTo [THEN leadsTo_imp_LeadsTo, THEN PSP_Stable2])
  apply (blast intro: subset_imp_LeadsTo)+
done

```

## 5.5 Induction rules

```

lemma LeadsTo_wf_induct:
  "[| wf r;
     $\forall m. F \in (A \cap f^{-1}\{m\}) \text{ LeadsTo } ((A \cap f^{-1}(r^{-1} \text{ `` } \{m\})) \cup B) \text{ |}]$ 
  ==> F \in A \text{ LeadsTo } B"
apply (simp add: LeadsTo_eq_leadsTo)
apply (erule leadsTo_wf_induct)
apply (blast intro: leadsTo_weaken)
done

```

```

lemma Bounded_induct:
  "[| wf r;
     $\forall m \in I. F \in (A \cap f^{-1}\{m\}) \text{ LeadsTo } ((A \cap f^{-1}(r^{-1} \text{ `` } \{m\})) \cup B) \text{ |}]$ 
  ==> F \in A \text{ LeadsTo } ((A - (f^{-1} I)) \cup B)"
apply (erule LeadsTo_wf_induct, safe)
apply (case_tac "m \in I")
apply (blast intro: LeadsTo_weaken)
apply (blast intro: subset_imp_LeadsTo)
done

```

```

lemma LessThan_induct:
  " $(!!m::nat. F \in (A \cap f^{-1}\{m\}) \text{ LeadsTo } ((A \cap f^{-1}(\text{lessThan } m)) \cup B))$ 
  ==> F \in A \text{ LeadsTo } B"
by (rule wf_less_than [THEN LeadsTo_wf_induct], auto)

```

Integer version. Could generalize from 0 to any lower bound

```

lemma integ_0_le_induct:
  "[| F \in Always {s. (0::int) \le f s};
    !! z. F \in (A \cap {s. f s = z}) \text{ LeadsTo } ((A \cap {s. f s < z}) \cup B) \text{ |}]
  ==> F \in A \text{ LeadsTo } B"
apply (rule_tac f = "nat o f" in LessThan_induct)
apply (simp add: vimage_def)
apply (rule Always_LeadsTo_weaken, assumption+)
apply (auto simp add: nat_eq_iff nat_less_iff)
done

```

```

lemma LessThan_bounded_induct:
  " $(!!l::nat. \forall m \in \text{greaterThan } l. F \in (A \cap f^{-1}\{m\}) \text{ LeadsTo } ((A \cap f^{-1}(\text{lessThan } m)) \cup B))$ 
  ==> F \in A \text{ LeadsTo } ((A \cap (f^{-1}(\text{atMost } l))) \cup B)"

```

```

apply (simp only: Diff_eq [symmetric] vimage_Compl
              Compl_greaterThan [symmetric])
apply (rule wf_less_than [THEN Bounded_induct], simp)
done

lemma GreaterThan_bounded_induct:
  "!!l::nat.  $\forall m \in \text{lessThan } l.$ 
     $F \in (A \cap f^{-1}\{m\}) \text{ LeadsTo } ((A \cap f^{-1}(\text{greaterThan } m)) \cup B)$ 
  ==>  $F \in A \text{ LeadsTo } ((A \cap (f^{-1}(\text{atLeast } l))) \cup B)$ "
apply (rule_tac f = f and f1 = "%k. l - k"
      in wf_less_than [THEN wf_inv_image, THEN LeadsTo_wf_induct])
apply (simp add: Image_singleton, clarify)
apply (case_tac "m < l")
  apply (blast intro: LeadsTo_weaken_R diff_less_mono2)
  apply (blast intro: not_le_imp_less subset_imp_LeadsTo)
done

```

## 5.6 Completion: Binary and General Finite versions

```

lemma Completion:
  "[|  $F \in A \text{ LeadsTo } (A' \cup C); F \in A' \text{ Co } (A' \cup C);$ 
     $F \in B \text{ LeadsTo } (B' \cup C); F \in B' \text{ Co } (B' \cup C)$  |]
  ==>  $F \in (A \cap B) \text{ LeadsTo } ((A' \cap B') \cup C)$ "
apply (simp add: LeadsTo_eq_leadsTo Constrains_eq_constrains Int_Un_distrib)
apply (blast intro: completion_leadsTo_weaken)
done

```

```

lemma Finite_completion_lemma:
  "finite I
  ==> ( $\forall i \in I. F \in (A \ i) \text{ LeadsTo } (A' \ i \cup C)$ ) -->
    ( $\forall i \in I. F \in (A' \ i) \text{ Co } (A' \ i \cup C)$ ) -->
     $F \in (\bigcap i \in I. A \ i) \text{ LeadsTo } ((\bigcap i \in I. A' \ i) \cup C)$ "
apply (erule finite_induct, auto)
apply (rule Completion)
  prefer 4
  apply (simp only: INT_simps [symmetric])
  apply (rule Constrains_INT, auto)
done

```

```

lemma Finite_completion:
  "[| finite I;
    !!i.  $i \in I \implies F \in (A \ i) \text{ LeadsTo } (A' \ i \cup C);$ 
    !!i.  $i \in I \implies F \in (A' \ i) \text{ Co } (A' \ i \cup C)$  |]
  ==>  $F \in (\bigcap i \in I. A \ i) \text{ LeadsTo } ((\bigcap i \in I. A' \ i) \cup C)$ "
by (blast intro: Finite_completion_lemma [THEN mp, THEN mp])

```

```

lemma Stable_completion:
  "[|  $F \in A \text{ LeadsTo } A'; F \in \text{Stable } A';$ 
     $F \in B \text{ LeadsTo } B'; F \in \text{Stable } B'$  |]
  ==>  $F \in (A \cap B) \text{ LeadsTo } (A' \cap B')$ "
apply (unfold Stable_def)
apply (rule_tac C1 = "{}" in Completion [THEN LeadsTo_weaken_R])
apply (force+)
done

```

```

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)"
apply (unfold Stable_def)
apply (rule_tac C1 = "{}" in Finite_completion [THEN LeadsTo_weaken_R])
apply (simp_all, blast+)
done

end

```

## 6 The Detects Relation

```

theory Detects imports FP SubstAx begin

```

```

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

```

```

  where "A Detects B = (Always (-A ∪ B)) ∩ (B LeadsTo A)"

```

```

definition Equality :: "[ 'a set, 'a set ] => 'a set" (infixl <==> 60)

```

```

  where "A <==> B = (-A ∪ B) ∩ (A ∪ -B)"

```

```

lemma Always_at_FP:

```

```

  "[| F ∈ A LeadsTo B; all_total F |] ==> F ∈ Always (-((FP F) ∩ A ∩ -B))"
supply [[simproc del: boolean_algebra_cancel_inf]] inf_compl_bot_right[simp
del]
apply (rule LeadsTo_empty)
apply (subgoal_tac "F ∈ (FP F ∩ A ∩ - B) LeadsTo (B ∩ (FP F ∩ -B))")
apply (subgoal_tac [2] " (FP F ∩ A ∩ - B) = (A ∩ (FP F ∩ -B))")
apply (subgoal_tac "(B ∩ (FP F ∩ -B)) = {}")
apply auto
apply (blast intro: PSP_Stable stable_imp_Stable stable_FP_Int)
done

```

```

lemma Detects_Trans:

```

```

  "[| F ∈ A Detects B; F ∈ B Detects C |] ==> F ∈ A Detects C"
apply (unfold Detects_def Int_def)
apply (simp (no_asm))
apply safe
apply (rule_tac [2] LeadsTo_Trans, auto)
apply (subgoal_tac "F ∈ Always ((-A ∪ B) ∩ (-B ∪ C))")
  apply (blast intro: Always_weaken)
apply (simp add: Always_Int_distrib)
done

```

```

lemma Detects_refl: "F ∈ A Detects A"

```

```

apply (unfold Detects_def)

```

```

apply (simp (no_asm) add: Un_commute Compl_partition subset_imp_LeadsTo)
done

```

```

lemma Detects_eq_Un: "(A<==>B) = (A ∩ B) ∪ (-A ∩ -B)"
by (unfold Equality_def, blast)

```

```

lemma Detects_antisym:
  "[| F ∈ A Detects B; F ∈ B Detects A |] ==> F ∈ Always (A <==> B)"
apply (unfold Detects_def Equality_def)
apply (simp add: Always_Int_I Un_commute)
done

```

```

lemma Detects_Always:
  "[| F ∈ A Detects B; all_total F |] ==> F ∈ Always (-(FP F) ∪ (A <==> B))"
apply (unfold Detects_def Equality_def)
apply (simp add: Un_Int_distrib Always_Int_distrib)
apply (blast dest: Always_at_FP intro: Always_weaken)
done

```

```

lemma Detects_Imp_LeadstoEQ:
  "F ∈ A Detects B ==> F ∈ UNIV LeadsTo (A <==> B)"
apply (unfold Detects_def Equality_def)
apply (rule_tac B = B in LeadsTo_Diff)
  apply (blast intro: Always_LeadsToI subset_imp_LeadsTo)
  apply (blast intro: Always_LeadsTo_weaken)
done

```

```

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 ⊆ AllowedActs G &
        Acts G ⊆ AllowedActs F"

```

```

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

```

```

definition

```

```

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

```

**definition**

```

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

```

**definition** SKIP :: "'a program" (<⊥>)

```

where "⊥ = mk_program (UNIV, {}, UNIV)"

```

**definition**

```

safety_prop :: "'a program set => bool"
where "safety_prop X  $\longleftrightarrow$  SKIP  $\in$  X  $\wedge$  ( $\forall G. \text{Acts } G \subseteq \bigcup (\text{Acts } 'X) \longrightarrow G \in X$ )"

```

**syntax**

```

"_JOIN1" :: "[pttrns, 'b set] => 'b set" (<(<indent=3 notation=<binder ⊔>>⊔_./_)> 10)

```

```

"_JOIN"  :: "[pttrn, 'a set, 'b set] => 'b set" (<(<indent=3 notation=<binder ⊔>>⊔_./_)> 10)

```

**syntax\_consts**

```

"_JOIN1" "_JOIN" == JOIN

```

**translations**

```

"⊔ x  $\in$  A. B" == "CONST JOIN A ( $\lambda x. B$ )"
"⊔ x y. B" == "⊔ x. ⊔ y. B"
"⊔ x. B" == "CONST JOIN (CONST UNIV) ( $\lambda x. B$ )"

```

## 7.1 SKIP

**lemma** Init\_SKIP [simp]: "Init SKIP = UNIV"

by (simp add: SKIP\_def)

**lemma** Acts\_SKIP [simp]: "Acts SKIP = {Id}"

by (simp add: SKIP\_def)

**lemma** AllowedActs\_SKIP [simp]: "AllowedActs SKIP = UNIV"

by (auto simp add: SKIP\_def)

**lemma** reachable\_SKIP [simp]: "reachable SKIP = UNIV"

by (force elim: reachable.induct intro: reachable.intros)

## 7.2 SKIP and safety properties

**lemma** SKIP\_in\_constrains\_iff [iff]: "(SKIP  $\in$  A co B) = (A  $\subseteq$  B)"

by (unfold constrains\_def, auto)

**lemma** SKIP\_in\_Constrains\_iff [iff]: "(SKIP  $\in$  A Co B) = (A  $\subseteq$  B)"

by (unfold Constrains\_def, auto)

**lemma** SKIP\_in\_stable [iff]: "SKIP  $\in$  stable A"

by (unfold stable\_def, auto)

```
declare SKIP_in_stable [THEN stable_imp_Stable, iff]
```

### 7.3 Join

```
lemma Init_Join [simp]: "Init (F⊔G) = Init F ∩ Init G"
by (simp add: Join_def)
```

```
lemma Acts_Join [simp]: "Acts (F⊔G) = Acts F ∪ Acts G"
by (auto simp add: Join_def)
```

```
lemma AllowedActs_Join [simp]:
  "AllowedActs (F⊔G) = AllowedActs F ∩ AllowedActs G"
by (auto simp add: Join_def)
```

### 7.4 JN

```
lemma JN_empty [simp]: "(⋂ i ∈ {}. F i) = SKIP"
by (unfold JOIN_def SKIP_def, auto)
```

```
lemma JN_insert [simp]: "(⋂ i ∈ insert a I. F i) = (F a)⊔(⋂ i ∈ I. F i)"
apply (rule program_equalityI)
apply (auto simp add: JOIN_def Join_def)
done
```

```
lemma Init_JN [simp]: "Init (⋂ i ∈ I. F i) = (⋂ i ∈ I. Init (F i))"
by (simp add: JOIN_def)
```

```
lemma Acts_JN [simp]: "Acts (⋂ i ∈ I. F i) = insert Id (⋃ i ∈ I. Acts (F i))"
by (auto simp add: JOIN_def)
```

```
lemma AllowedActs_JN [simp]:
  "AllowedActs (⋂ i ∈ I. F i) = (⋂ i ∈ I. AllowedActs (F i))"
by (auto simp add: JOIN_def)
```

```
lemma JN_cong [cong]:
  "[| I=J; !!i. i ∈ J ==> F i = G i |] ==> (⋂ i ∈ I. F i) = (⋂ i ∈ J. G i)"
by (simp add: JOIN_def)
```

### 7.5 Algebraic laws

```
lemma Join_commute: "F⊔G = G⊔F"
by (simp add: Join_def Un_commute Int_commute)
```

```
lemma Join_assoc: "(F⊔G)⊔H = F⊔(G⊔H)"
by (simp add: Un_ac Join_def Int_assoc insert_absorb)
```

```
lemma Join_left_commute: "A⊔(B⊔C) = B⊔(A⊔C)"
by (simp add: Un_ac Int_ac Join_def insert_absorb)
```

```
lemma Join_SKIP_left [simp]: "SKIP⊔F = F"
```

```

apply (unfold Join_def SKIP_def)
apply (rule program_equalityI)
apply (simp_all (no_asm) add: insert_absorb)
done

```

```

lemma Join_SKIP_right [simp]: "F  $\sqcup$  SKIP = F"
apply (unfold Join_def SKIP_def)
apply (rule program_equalityI)
apply (simp_all (no_asm) add: insert_absorb)
done

```

```

lemma Join_absorb [simp]: "F  $\sqcup$  F = F"
apply (unfold Join_def)
apply (rule program_equalityI, auto)
done

```

```

lemma Join_left_absorb: "F  $\sqcup$  (F  $\sqcup$  G) = F  $\sqcup$  G"
apply (unfold Join_def)
apply (rule program_equalityI, auto)
done

```

```

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)"
by (auto intro!: program_equalityI)

```

```

lemma JN_Un: "(( $\sqcup$  i  $\in$  I  $\cup$  J. F i) = (( $\sqcup$  i  $\in$  I. F i)  $\sqcup$  ( $\sqcup$  i  $\in$  J. F i)))"
by (auto intro!: program_equalityI)

```

```

lemma JN_constant: "(( $\sqcup$  i  $\in$  I. c) = (if I={} then SKIP else c))"
by (rule program_equalityI, auto)

```

```

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))"
by (auto intro!: program_equalityI)

```

```

lemma JN_Join_miniscope:
  "i  $\in$  I ==> ( $\sqcup$  i  $\in$  I. F i  $\sqcup$  G) = (( $\sqcup$  i  $\in$  I. F i)  $\sqcup$  G)"
by (auto simp add: JN_Join_distrib JN_constant)

```

```

lemma JN_Join_diff: "i  $\in$  I ==> F i  $\sqcup$  JOIN (I - {i}) F = JOIN I F"
apply (unfold JOIN_def Join_def)
apply (rule program_equalityI, auto)
done

```

## 7.7 Safety: co, stable, FP

```

lemma JN_constrains:
  "i  $\in$  I ==> ( $\sqcup$  i  $\in$  I. F i)  $\in$  A co B = ( $\forall$  i  $\in$  I. F i  $\in$  A co B)"
by (simp add: constrains_def JOIN_def, blast)

```

```

lemma Join_constrains [simp]:
  "(F⊔G ∈ A co B) = (F ∈ A co B & G ∈ A co B)"
by (auto simp add: constrains_def Join_def)

lemma Join_unless [simp]:
  "(F⊔G ∈ A unless B) = (F ∈ A unless B & G ∈ A unless B)"
by (simp add: unless_def)

lemma Join_constrains_weaken:
  "[| F ∈ A co A'; G ∈ B co B' |]
   ==> F⊔G ∈ (A ∩ B) co (A' ∪ B')"
by (simp, blast intro: constrains_weaken)

lemma JN_constrains_weaken:
  "[| ∀i ∈ I. F i ∈ A i co A' i; i ∈ I |]
   ==> (⊔i ∈ I. F i) ∈ (⋂i ∈ I. A i) co (⋃i ∈ I. A' i)"
apply (simp (no_asm_simp) add: JN_constrains)
apply (blast intro: constrains_weaken)
done

lemma JN_stable: "(⊔i ∈ I. F i) ∈ stable A = (∀i ∈ I. F i ∈ stable A)"
by (simp add: stable_def constrains_def JOIN_def)

lemma invariant_JN_I:
  "[| !!i. i ∈ I ==> F i ∈ invariant A; i ∈ I |]
   ==> (⊔i ∈ I. F i) ∈ invariant A"
by (simp add: invariant_def JN_stable, blast)

lemma Join_stable [simp]:
  "(F⊔G ∈ stable A) =
   (F ∈ stable A & G ∈ stable A)"
by (simp add: stable_def)

lemma Join_increasing [simp]:
  "(F⊔G ∈ increasing f) =
   (F ∈ increasing f & G ∈ increasing f)"
by (auto simp add: increasing_def)

lemma invariant_JoinI:
  "[| F ∈ invariant A; G ∈ invariant A |]
   ==> F⊔G ∈ invariant A"
by (auto simp add: invariant_def)

lemma FP_JN: "FP (⊔i ∈ I. F i) = (⋂i ∈ I. FP (F i))"
by (simp add: FP_def JN_stable INTER_eq)

```

## 7.8 Progress: transient, ensures

```

lemma JN_transient:

```



```

      "i ∈ I ==>
      (⋀ i ∈ I. F i) ∈ transient A = (∃ i ∈ I. F i ∈ transient A)"
by (auto simp add: transient_def JOIN_def)

lemma Join_transient [simp]:
  "F⊔G ∈ transient A =
  (F ∈ transient A | G ∈ transient A)"
by (auto simp add: bex_Un transient_def Join_def)

lemma Join_transient_I1: "F ∈ transient A ==> F⊔G ∈ transient A"
by simp

lemma Join_transient_I2: "G ∈ transient A ==> F⊔G ∈ transient A"
by simp

lemma JN_ensures:
  "i ∈ I ==>
  (⋀ i ∈ I. F i) ∈ A ensures B =
  ((∀ i ∈ I. F i ∈ (A-B) co (A ∪ B)) & (∃ i ∈ I. F i ∈ A ensures B))"
by (auto simp add: ensures_def JN_constrains JN_transient)

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)))"
by (auto simp add: ensures_def)

lemma stable_Join_constrains:
  "[| F ∈ stable A; G ∈ A co A' |]
  ==> F⊔G ∈ A co A'"
apply (unfold stable_def constrains_def Join_def)
apply (simp add: ball_Un, blast)
done

lemma stable_Join_Always1:
  "[| F ∈ stable A; G ∈ invariant A |] ==> F⊔G ∈ Always A"
apply (simp (no_asm_use) add: Always_def invariant_def Stable_eq_stable)
apply (force intro: stable_Int)
done

lemma stable_Join_Always2:
  "[| F ∈ invariant A; G ∈ stable A |] ==> F⊔G ∈ Always A"
apply (subst Join_commute)
apply (blast intro: stable_Join_Always1)
done

lemma stable_Join_ensures1:
  "[| F ∈ stable A; G ∈ A ensures B |] ==> F⊔G ∈ A ensures B"
apply (simp (no_asm_simp) add: Join_ensures)
apply (simp add: stable_def ensures_def)
apply (erule constrains_weaken, auto)

```

done

```
lemma stable_Join_ensures2:
  "[| F ∈ A ensures B;  G ∈ stable A |] ==> F ⊔ G ∈ A ensures B"
apply (subst Join_commute)
apply (blast intro: stable_Join_ensures1)
done
```

## 7.9 the ok and OK relations

```
lemma ok_SKIP1 [iff]: "SKIP ok F"
by (simp add: ok_def)
```

```
lemma ok_SKIP2 [iff]: "F ok SKIP"
by (simp add: ok_def)
```

```
lemma ok_Join_commute:
  "(F ok G & (F ⊔ G) ok H) = (G ok H & F ok (G ⊔ H))"
by (auto simp add: ok_def)
```

```
lemma ok_commute: "(F ok G) = (G ok F)"
by (auto simp add: ok_def)
```

```
lemmas ok_sym = ok_commute [THEN iffD1]
```

```
lemma ok_iff_OK:
  "OK {(0::int,F),(1,G),(2,H)} snd = (F ok G & (F ⊔ G) ok H)"
apply (simp add: Ball_def conj_disj_distribR ok_def Join_def OK_def insert_absorb
  all_conj_distrib)
apply blast
done
```

```
lemma ok_Join_iff1 [iff]: "F ok (G ⊔ H) = (F ok G & F ok H)"
by (auto simp add: ok_def)
```

```
lemma ok_Join_iff2 [iff]: "(G ⊔ H) ok F = (G ok F & H ok F)"
by (auto simp add: ok_def)
```

```
lemma ok_Join_commute_I: "[| F ok G; (F ⊔ G) ok H |] ==> F ok (G ⊔ H)"
by (auto simp add: ok_def)
```

```
lemma ok_JN_iff1 [iff]: "F ok (JOIN I G) = (∀ i ∈ I. F ok G i)"
by (auto simp add: ok_def)
```

```
lemma ok_JN_iff2 [iff]: "(JOIN I G) ok F = (∀ i ∈ I. G i ok F)"
by (auto simp add: ok_def)
```

```
lemma OK_iff_ok: "OK I F = (∀ i ∈ I. ∀ j ∈ I - {i}. (F i) ok (F j))"
by (auto simp add: ok_def OK_def)
```

```
lemma OK_imp_ok: "[| OK I F; i ∈ I; j ∈ I; i ≠ j |] ==> (F i) ok (F j)"
by (auto simp add: OK_iff_ok)
```

## 7.10 Allowed

**lemma** *Allowed\_SKIP [simp]: "Allowed SKIP = UNIV"*

**by** (auto simp add: Allowed\_def)

**lemma** *Allowed\_Join [simp]: "Allowed (F  $\sqcup$  G) = Allowed F  $\cap$  Allowed G"*

**by** (auto simp add: Allowed\_def)

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

**by** (auto simp add: Allowed\_def)

**lemma** *ok\_iff\_Allowed: "F ok G = (F  $\in$  Allowed G & G  $\in$  Allowed F)"*

**by** (simp add: ok\_def Allowed\_def)

**lemma** *OK\_iff\_Allowed: "OK I F = ( $\forall i \in I. \forall j \in I - \{i\}. F \ i \in \text{Allowed}(F \ j)$ )"*

**by** (auto simp add: OK\_iff\_ok ok\_iff\_Allowed)

## 7.11 *safety\_prop*, for reasoning about given instances of "ok"

**lemma** *safety\_prop\_Acts\_iff:*

*"safety\_prop X ==> (Acts G  $\subseteq$  insert Id ( $\bigcup$  (Acts ' X))) = (G  $\in$  X)"*

**by** (auto simp add: safety\_prop\_def)

**lemma** *safety\_prop\_AllowedActs\_iff\_Allowed:*

*"safety\_prop X ==> ( $\bigcup$  (Acts ' X)  $\subseteq$  AllowedActs F) = (X  $\subseteq$  Allowed F)"*

**by** (auto simp add: Allowed\_def safety\_prop\_Acts\_iff [symmetric])

**lemma** *Allowed\_eq:*

*"safety\_prop X ==> Allowed (mk\_program (init, acts,  $\bigcup$  (Acts ' X))) = X"*

**by** (simp add: Allowed\_def safety\_prop\_Acts\_iff)

**lemma** *safety\_prop\_constrains [iff]: "safety\_prop (A co B) = (A  $\subseteq$  B)"*

**by** (simp add: safety\_prop\_def constrains\_def, blast)

**lemma** *safety\_prop\_stable [iff]: "safety\_prop (stable A)"*

**by** (simp add: stable\_def)

**lemma** *safety\_prop\_Int [simp]:*

*"safety\_prop X  $\implies$  safety\_prop Y  $\implies$  safety\_prop (X  $\cap$  Y)"*

**proof** (clarsimp simp add: safety\_prop\_def)

**fix** G

**assume** " $\forall G. \text{Acts } G \subseteq (\bigcup_{x \in X. \text{Acts } x}) \implies G \in X$ "

**then have** X: " $\text{Acts } G \subseteq (\bigcup_{x \in X. \text{Acts } x}) \implies G \in X$ " **by** blast

**assume** " $\forall G. \text{Acts } G \subseteq (\bigcup_{x \in Y. \text{Acts } x}) \implies G \in Y$ "

**then have** Y: " $\text{Acts } G \subseteq (\bigcup_{x \in Y. \text{Acts } x}) \implies G \in Y$ " **by** blast

**assume** Acts: " $\text{Acts } G \subseteq (\bigcup_{x \in X \cap Y. \text{Acts } x})$ "

**with** X **and** Y **show** " $G \in X \wedge G \in Y$ " **by** auto

**qed**

**lemma** *safety\_prop\_INTER [simp]:*

*"( $\bigwedge i. i \in I \implies \text{safety\_prop } (X \ i)$ )  $\implies$  safety\_prop ( $\bigcap_{i \in I. X \ i}$ )"*

**proof** (clarsimp simp add: safety\_prop\_def)

**fix** G **and** i

```

    assume " $\bigwedge i. i \in I \implies \perp \in X\ i \wedge$ 
      ( $\forall G. \text{Acts } G \subseteq (\bigcup_{x \in X} i. \text{Acts } x) \longrightarrow G \in X\ i$ )"
    then have *: " $i \in I \implies \text{Acts } G \subseteq (\bigcup_{x \in X} i. \text{Acts } x) \implies G \in X\ i$ "
      by blast
    assume "i ∈ I"
    moreover assume " $\text{Acts } G \subseteq (\bigcup_{j \in \bigcap i \in I. X\ i. \text{Acts } j})$ "
    ultimately have " $\text{Acts } G \subseteq (\bigcup_{i \in X} i. \text{Acts } i)$ "
      by auto
    with * <i ∈ I> show "G ∈ X i" by blast
qed

lemma safety_prop_INTER1 [simp]:
  " $(\bigwedge i. \text{safety\_prop } (X\ i)) \implies \text{safety\_prop } (\bigcap i. X\ i)$ "
  by (rule safety_prop_INTER) simp

lemma def_prg_Allowed:
  "[| F == mk_program (init, acts,  $\bigcup (\text{Acts } 'X)$ ) ; safety_prop X |]
    ==> Allowed F = X"
  by (simp add: Allowed_eq)

lemma Allowed_totalize [simp]: "Allowed (totalize F) = Allowed F"
  by (simp add: Allowed_def)

lemma def_total_prg_Allowed:
  "[| F = mk_total_program (init, acts,  $\bigcup (\text{Acts } 'X)$ ) ; safety_prop X |]
    ==> Allowed F = X"
  by (simp add: mk_total_program_def def_prg_Allowed)

lemma def_UNION_ok_iff:
  "[| F = mk_program (init, acts,  $\bigcup (\text{Acts } 'X)$ ) ; safety_prop X |]
    ==> F ok G = (G ∈ X & acts ⊆ AllowedActs G)"
  by (auto simp add: ok_def safety_prop_Acts_iff)

The union of two total programs is total.

lemma totalize_Join: "totalize F ⊔ totalize G = totalize (F ⊔ G)"
  by (simp add: program_equalityI totalize_def Join_def image_UN)

lemma all_total_Join: "[| all_total F; all_total G |] ==> all_total (F ⊔ G)"
  by (simp add: all_total_def, blast)

lemma totalize_JN: " $(\bigcup i \in I. \text{totalize } (F\ i)) = \text{totalize } (\bigcup i \in I. F\ i)$ "
  by (simp add: program_equalityI totalize_def JOIN_def image_UN)

lemma all_total_JN: " $(\forall i. i \in I \implies \text{all\_total } (F\ i)) \implies \text{all\_total } (\bigcup i \in I. F\ i)$ "
  by (simp add: all_total_iff_totalize totalize_JN [symmetric])

end

```

## 8 Composition: Basic Primitives

theory Comp

```

imports Union
begin

instantiation program :: (type) ord
begin

definition component_def: " $F \leq H \longleftrightarrow (\exists G. F \sqcup G = H)$ "

definition strict_component_def: " $F < (H :: 'a \text{ program}) \longleftrightarrow (F \leq H \ \& \ F \neq H)$ "

instance ..

end

definition component_of :: "'a program => 'a program => bool" (infixl <component'_of>
50)
  where "F component_of H ==  $\exists G. F \text{ ok } G \ \& \ F \sqcup G = H$ "

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

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

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

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

```

## 8.1 The component relation

```

lemma componentI: " $H \leq F \mid H \leq G \implies H \leq (F \sqcup G)$ "
apply (unfold component_def, auto)
apply (rule_tac x = " $G \sqcup G$ " in exI)
apply (rule_tac [2] x = " $G \sqcup F$ " in exI)
apply (auto simp add: Join_ac)
done

lemma component_eq_subset:
  " $(F \leq G) =$ 
    ( $\text{Init } G \subseteq \text{Init } F \ \& \ \text{Acts } F \subseteq \text{Acts } G \ \& \ \text{AllowedActs } G \subseteq \text{AllowedActs } F$ )"
apply (unfold component_def)
apply (force intro!: exI program_equalityI)
done

lemma component_SKIP [iff]: " $\text{SKIP} \leq F$ "
apply (unfold component_def)
apply (force intro: Join_SKIP_left)
done

lemma component_refl [iff]: " $F \leq (F :: 'a \text{ program})$ "

```

```

apply (unfold component_def)
apply (blast intro: Join_SKIP_right)
done

lemma SKIP_minimal: "F ≤ SKIP ==> F = SKIP"
by (auto intro!: program_equalityI simp add: component_eq_subset)

lemma component_Join1: "F ≤ (F ⊔ G)"
by (unfold component_def, blast)

lemma component_Join2: "G ≤ (F ⊔ G)"
apply (unfold component_def)
apply (simp add: Join_commute, blast)
done

lemma Join_absorb1: "F ≤ G ==> F ⊔ G = G"
by (auto simp add: component_def Join_left_absorb)

lemma Join_absorb2: "G ≤ F ==> F ⊔ G = F"
by (auto simp add: Join_ac component_def)

lemma JN_component_iff: "((JOIN I F) ≤ H) = (∀ i ∈ I. F i ≤ H)"
by (simp add: component_eq_subset, blast)

lemma component_JN: "i ∈ I ==> (F i) ≤ (⊔ i ∈ I. (F i))"
apply (unfold component_def)
apply (blast intro: JN_absorb)
done

lemma component_trans: "[| F ≤ G; G ≤ H |] ==> F ≤ (H :: 'a program)"
apply (unfold component_def)
apply (blast intro: Join_assoc [symmetric])
done

lemma component_antisym: "[| F ≤ G; G ≤ F |] ==> F = (G :: 'a program)"
apply (simp (no_asm_use) add: component_eq_subset)
apply (blast intro!: program_equalityI)
done

lemma Join_component_iff: "((F ⊔ G) ≤ H) = (F ≤ H & G ≤ H)"
by (simp add: component_eq_subset, blast)

lemma component_constrains: "[| F ≤ G; G ∈ A co B |] ==> F ∈ A co B"
by (auto simp add: constrains_def component_eq_subset)

lemma component_stable: "[| F ≤ G; G ∈ stable A |] ==> F ∈ stable A"
by (auto simp add: stable_def component_constrains)

lemmas program_less_le = strict_component_def

```

## 8.2 The preserves property

```

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

```

```

by (unfold preserves_def, blast)

lemma preserves_imp_eq:
  "[| F ∈ preserves v; act ∈ Acts F; (s,s') ∈ act |] ==> v s = v s'"
by (unfold preserves_def stable_def constrains_def, force)

lemma Join_preserves [iff]:
  "(F ⊔ G ∈ preserves v) = (F ∈ preserves v & G ∈ preserves v)"
by (unfold preserves_def, auto)

lemma JN_preserves [iff]:
  "(JOIN I F ∈ preserves v) = (∀ i ∈ I. F i ∈ preserves v)"
by (simp add: JN_stable preserves_def, blast)

lemma SKIP_preserves [iff]: "SKIP ∈ preserves v"
by (auto simp add: preserves_def)

lemma funPair_apply [simp]: "(funPair f g) x = (f x, g x)"
by (simp add: funPair_def)

lemma preserves_funPair: "preserves (funPair v w) = preserves v ∩ preserves w"
by (auto simp add: preserves_def stable_def constrains_def, blast)

declare preserves_funPair [THEN eqset_imp_iff, iff]

lemma funPair_o_distrib: "(funPair f g) o h = funPair (f o h) (g o h)"
by (simp add: funPair_def o_def)

lemma fst_o_funPair [simp]: "fst o (funPair f g) = f"
by (simp add: funPair_def o_def)

lemma snd_o_funPair [simp]: "snd o (funPair f g) = g"
by (simp add: funPair_def o_def)

lemma subset_preserves_o: "preserves v ⊆ preserves (w o v)"
by (force simp add: preserves_def stable_def constrains_def)

lemma preserves_subset_stable: "preserves v ⊆ stable {s. P (v s)}"
apply (auto simp add: preserves_def stable_def constrains_def)
apply (rename_tac s' s)
apply (subgoal_tac "v s = v s'")
apply (force+)
done

lemma preserves_subset_increasing: "preserves v ⊆ increasing v"
by (auto simp add: preserves_subset_stable [THEN subsetD] increasing_def)

lemma preserves_id_subset_stable: "preserves id ⊆ stable A"
by (force simp add: preserves_def stable_def constrains_def)

```

```
lemma safety_prop_preserves [iff]: "safety_prop (preserves v)"
by (auto intro: safety_prop_INTER1 simp add: preserves_def)
```

```
lemma stable_localTo_stable2:
  "[| F ∈ stable {s. P (v s) (w s)};
    G ∈ preserves v; G ∈ preserves w |]
  ==> F ⊔ G ∈ stable {s. P (v s) (w s)}"
apply simp
apply (subgoal_tac "G ∈ preserves (funPair v w) ")
  prefer 2 apply simp
apply (drule_tac P1 = "case_prod Q" for Q in preserves_subset_stable [THEN
subsetD],
  auto)
done
```

```
lemma Increasing_preserves_Stable:
  "[| F ∈ stable {s. v s ≤ w s}; G ∈ preserves v; F ⊔ G ∈ Increasing w
  |]
  ==> F ⊔ G ∈ Stable {s. v s ≤ w s}"
apply (auto simp add: stable_def Stable_def Increasing_def Constrains_def
all_conj_distrib)
apply (blast intro: constrains_weaken)

apply (auto simp add: preserves_def stable_def constrains_def)

apply (erule_tac V = "∀ act ∈ Acts F. P act" for P in thin_rl)
apply (erule_tac V = "∀ z. ∀ act ∈ Acts F. P z act" for P in thin_rl)
apply (subgoal_tac "v x = v xa")
  apply auto
apply (erule order_trans, blast)
done
```

```
lemma component_of_imp_component: "F component_of H ==> F ≤ H"
by (unfold component_def component_of_def, blast)
```

```
lemma component_of_refl [simp]: "F component_of F"
apply (unfold component_of_def)
apply (rule_tac x = SKIP in exI, auto)
done
```

```
lemma component_of_SKIP [simp]: "SKIP component_of F"
by (unfold component_of_def, auto)
```

```
lemma component_of_trans:
```



```

    "[/ F component_of G; G component_of H [/ ==> F component_of H"
  apply (unfold component_of_def)
  apply (blast intro: Join_assoc [symmetric])
done

lemmas strict_component_of_eq = strict_component_of_def

lemma localize_Init_eq [simp]: "Init (localize v F) = Init F"
by (simp add: localize_def)

lemma localize_Acts_eq [simp]: "Acts (localize v F) = Acts F"
by (simp add: localize_def)

lemma localize_AllowedActs_eq [simp]:
  "AllowedActs (localize v F) = AllowedActs F  $\cap$  ( $\bigcup G \in \text{preserves } v. \text{Acts } G$ )"
by (unfold localize_def, auto)

end

```

## 9 Guarantees Specifications

```

theory Guar
imports Comp
begin

instance program :: (type) order
  by standard (auto simp add: program_less_le dest: component_antisym intro:
    component_trans)

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 ==  $\text{SKIP} \in X \ \& \ (\forall F G. F \text{ ok } G \rightarrow F \in X \ \& \ G \in X \rightarrow (F \sqcup G) \in X)$ "

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

Guarantees properties

definition guar :: "[ 'a program set, 'a program set ] => 'a program set" (infixl
  <guarantees> 55) where
  "X guarantes 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 ==  $\bigcup (\{X. F \in X \text{ guarantees } Y\})"$ "

```

```

definition wx :: "( 'a program ) set => ( 'a program ) set" where
  "wx X ==  $\bigcup (\{Y. Y \subseteq X \ \& \ \text{ex\_prop } Y\})"$ "

```

```

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 ==
     $\forall H. (F \text{ ok } H \ \& \ G \text{ ok } H \ \& \ F \sqcup H \in \text{welldef} \cap X) \longrightarrow$ 
     $(G \sqcup H \in \text{welldef} \cap X)"$ 

```

```

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 ==
     $F \in \text{welldef} \cap X \longrightarrow G \in \text{welldef} \cap X"$ 

```

```

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))"
by (auto intro: ok_sym simp add: OK_iff_ok)

```

## 9.1 Existential Properties

```

lemma ex1:
  assumes "ex_prop X" and "finite GG"
  shows "GG  $\cap$  X  $\neq$   $\{\}$   $\implies$  OK GG ( $\%G. G$ )  $\implies$  ( $\bigcup G \in GG. G$ )  $\in$  X"
  apply (atomize (full))
  using assms(2) apply induct
  using assms(1) apply (unfold ex_prop_def)
  apply (auto simp add: OK_insert_iff Int_insert_left)
  done

```

```

lemma ex2:
  " $\forall GG. \text{finite } GG \ \& \ GG \cap X \neq \{\} \longrightarrow \text{OK } GG (\lambda G. G) \longrightarrow (\bigcup G \in GG. G) \in$ 
  X
   $\implies \text{ex\_prop } X"$ 
  apply (unfold ex_prop_def, clarify)
  apply (drule_tac x = "{F,G}" in spec)
  apply (auto dest: ok_sym simp add: OK_iff_ok)
  done

```

```

lemma ex_prop_finite:
  "ex_prop X =

```

```

    (∀ GG. finite GG & GG ∩ X ≠ {} & OK GG (%G. G) --> (⋂ G ∈ GG. G) ∈ X)"
  by (blast intro: ex1 ex2)

```

```

lemma ex_prop_equiv:
  "ex_prop X = (∀ G. G ∈ X = (∀ H. (G component_of H) --> H ∈ X))"
  apply auto
  apply (unfold ex_prop_def component_of_def, safe, blast, blast)
  apply (subst Join_commute)
  apply (drule ok_sym, blast)
  done

```

## 9.2 Universal Properties

```

lemma uv1:
  assumes "uv_prop X"
    and "finite GG"
    and "GG ⊆ X"
    and "OK GG (%G. G)"
  shows "(⋂ G ∈ GG. G) ∈ X"
  using assms(2-)
  apply induct
    using assms(1)
    apply (unfold uv_prop_def)
    apply (auto simp add: Int_insert_left OK_insert_iff)
  done

```

```

lemma uv2:
  "∀ GG. finite GG & GG ⊆ X & OK GG (%G. G) --> (⋂ G ∈ GG. G) ∈ X
    ==> uv_prop X"
  apply (unfold uv_prop_def)
  apply (rule conjI)
    apply (drule_tac x = "{}" in spec)
    prefer 2
    apply clarify
    apply (drule_tac x = "{F,G}" in spec)
  apply (auto dest: ok_sym simp add: OK_iff_ok)
  done

```

```

lemma uv_prop_finite:
  "uv_prop X =
    (∀ GG. finite GG ∧ GG ⊆ X ∧ OK GG (λG. G) → (⋂ G ∈ GG. G) ∈ X)"
  by (blast intro: uv1 uv2)

```

## 9.3 Guarantees

```

lemma guaranteesI:
  "(!!G. [| F ok G; F ⊔ G ∈ X |] ==> F ⊔ G ∈ Y) ==> F ∈ X guarantees Y"
  by (simp add: guar_def component_def)

```

```

lemma guaranteesD:
  "[| F ∈ X guarantees Y; F ok G; F ⊔ G ∈ X |] ==> F ⊔ G ∈ Y"

```

by (unfold guar\_def component\_def, blast)

```
lemma component_guaranteesD:
  "[| F ∈ X guarantees Y; F ⊔ G = H; H ∈ X; F ok G |] ==> H ∈ Y"
by (unfold guar_def, blast)
```

```
lemma guarantees_weaken:
  "[| F ∈ X guarantees X'; Y ⊆ X; X' ⊆ Y' |] ==> F ∈ Y guarantees Y'"
by (unfold guar_def, blast)
```

```
lemma subset_imp_guarantees_UNIV: "X ⊆ Y ==> X guarantees Y = UNIV"
by (unfold guar_def, blast)
```

```
lemma subset_imp_guarantees: "X ⊆ Y ==> F ∈ X guarantees Y"
by (unfold guar_def, blast)
```

```
lemma ex_prop_imp: "ex_prop Y ==> (Y = UNIV guarantees Y)"
apply (simp (no_asm_use) add: guar_def ex_prop_equiv)
apply safe
  apply (drule_tac x = x in spec)
  apply (drule_tac [2] x = x in spec)
  apply (drule_tac [2] sym)
apply (auto simp add: component_of_def)
done
```

```
lemma guarantees_imp: "(Y = UNIV guarantees Y) ==> ex_prop(Y)"
by (auto simp add: guar_def ex_prop_equiv component_of_def dest: sym)
```

```
lemma ex_prop_equiv2: "(ex_prop Y) = (Y = UNIV guarantees Y)"
apply (rule iffI)
apply (rule ex_prop_imp)
apply (auto simp add: guarantees_imp)
done
```

## 9.4 Distributive Laws. Re-Orient to Perform Miniscoping

```
lemma guarantees_UN_left:
  "(⋃ i ∈ I. X i) guarantees Y = (⋂ i ∈ I. X i guarantees Y)"
by (unfold guar_def, blast)
```

```
lemma guarantees_Un_left:
  "(X ∪ Y) guarantees Z = (X guarantees Z) ∩ (Y guarantees Z)"
by (unfold guar_def, blast)
```

```
lemma guarantees_INT_right:
  "X guarantees (⋂ i ∈ I. Y i) = (⋂ i ∈ I. X guarantees Y i)"
by (unfold guar_def, blast)
```

```
lemma guarantees_Int_right:
  "Z guarantees (X ∩ Y) = (Z guarantees X) ∩ (Z guarantees Y)"
```

```
by (unfold guar_def, blast)
```

```
lemma guarantees_Int_right_I:
  "[| F ∈ Z guarantees X; F ∈ Z guarantees Y |]
   ==> F ∈ Z guarantees (X ∩ Y)"
by (simp add: guarantees_Int_right)
```

```
lemma guarantees_INT_right_iff:
  "(F ∈ X guarantees (⋂ (Y ' I))) = (∀ i ∈ I. F ∈ X guarantees (Y i))"
by (simp add: guarantees_INT_right)
```

```
lemma shunting: "(X guarantees Y) = (UNIV guarantees (-X ∪ Y))"
by (unfold guar_def, blast)
```

```
lemma contrapositive: "(X guarantees Y) = -Y guarantees -X"
by (unfold guar_def, blast)
```

```
lemma combining1:
  "[| F ∈ V guarantees X; F ∈ (X ∩ Y) guarantees Z |]
   ==> F ∈ (V ∩ Y) guarantees Z"
by (unfold guar_def, blast)
```

```
lemma combining2:
  "[| F ∈ V guarantees (X ∪ Y); F ∈ Y guarantees Z |]
   ==> F ∈ V guarantees (X ∪ Z)"
by (unfold guar_def, blast)
```

```
lemma all_guarantees:
  "∀ i ∈ I. F ∈ X guarantees (Y i) ==> F ∈ X guarantees (⋂ i ∈ I. Y i)"
by (unfold guar_def, blast)
```

```
lemma ex_guarantees:
  "∃ i ∈ I. F ∈ X guarantees (Y i) ==> F ∈ X guarantees (⋃ i ∈ I. Y i)"
by (unfold guar_def, blast)
```

## 9.5 Guarantees: Additional Laws (by lcp)

```
lemma guarantees_Join_Int:
  "[| F ∈ U guarantees V; G ∈ X guarantees Y; F ok G |]
   ==> F ⋈ G ∈ (U ∩ X) guarantees (V ∩ Y)"
apply (simp add: guar_def, safe)
  apply (simp add: Join_assoc)
  apply (subgoal_tac "F ⋈ G ⋈ Ga = G ⋈ (F ⋈ Ga) ")
  apply (simp add: ok_commute)
  apply (simp add: Join_ac)
done
```

```
lemma guarantees_Join_Un:
```

```

    "[/ F ∈ U guarantees V; G ∈ X guarantees Y; F ok G [/]
    ==> F⊔G ∈ (U ∪ X) guarantees (V ∪ Y)"
  apply (simp add: guar_def, safe)
  apply (simp add: Join_assoc)
  apply (subgoal_tac "F⊔G⊔Ga = G⊔(F⊔Ga) ")
  apply (simp add: ok_commute)
  apply (simp add: Join_ac)
done

lemma guarantees_JN_INT:
  "[/ ∀i∈I. F i ∈ X i guarantees Y i; OK I F [/]
  ==> (JOIN I F) ∈ (⋂ (X ' I)) guarantees (⋂ (Y ' I))"
  apply (unfold guar_def, auto)
  apply (drule bspec, assumption)
  apply (rename_tac "i")
  apply (drule_tac x = "JOIN (I-{i}) F⊔G" in spec)
  apply (auto intro: OK_imp_ok
    simp add: Join_assoc [symmetric] JN_Join_diff JN_absorb)
done

lemma guarantees_JN_UN:
  "[/ ∀i∈I. F i ∈ X i guarantees Y i; OK I F [/]
  ==> (JOIN I F) ∈ (⋃ (X ' I)) guarantees (⋃ (Y ' I))"
  apply (unfold guar_def, auto)
  apply (drule bspec, assumption)
  apply (rename_tac "i")
  apply (drule_tac x = "JOIN (I-{i}) F⊔G" in spec)
  apply (auto intro: OK_imp_ok
    simp add: Join_assoc [symmetric] JN_Join_diff JN_absorb)
done

```

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

```

lemma guarantees_Join_I1:
  "[/ F ∈ X guarantees Y; F ok G [/] ==> F⊔G ∈ X guarantees Y"
  by (simp add: guar_def Join_assoc)

lemma guarantees_Join_I2:
  "[/ G ∈ X guarantees Y; F ok G [/] ==> F⊔G ∈ X guarantees Y"
  apply (simp add: Join_commute [of _ G] ok_commute [of _ G])
  apply (blast intro: guarantees_Join_I1)
done

lemma guarantees_JN_I:
  "[/ i ∈ I; F i ∈ X guarantees Y; OK I F [/]
  ==> (⋂ i ∈ I. (F i)) ∈ X guarantees Y"
  apply (unfold guar_def, clarify)
  apply (drule_tac x = "JOIN (I-{i}) F⊔G" in spec)
  apply (auto intro: OK_imp_ok simp add: JN_Join_diff Join_assoc [symmetric])
done

```

```

lemma Join_welldef_D1: "F ⊔ G ∈ welldef ==> F ∈ welldef"
by (unfold welldef_def, auto)

lemma Join_welldef_D2: "F ⊔ G ∈ welldef ==> G ∈ welldef"
by (unfold welldef_def, auto)

lemma refines_refl: "F refines F wrt X"
by (unfold refines_def, blast)

lemma refines_trans:
  "[| H refines G wrt X; G refines F wrt X |] ==> H refines F wrt X"
apply (simp add: refines_def)
oops

lemma strict_ex_refine_lemma:
  "strict_ex_prop X
   ==> (∀H. F ok H & G ok H & F ⊔ H ∈ X --> G ⊔ H ∈ X)
       = (F ∈ X --> G ∈ X)"
by (unfold strict_ex_prop_def, auto)

lemma strict_ex_refine_lemma_v:
  "strict_ex_prop X
   ==> (∀H. F ok H & G ok H & F ⊔ H ∈ welldef & F ⊔ H ∈ X --> G ⊔ H ∈ X) =
       (F ∈ welldef ∩ X --> G ∈ X)"
apply (unfold strict_ex_prop_def, safe)
apply (erule_tac x = SKIP and P = "%H. PP H --> RR H" for PP RR in allE)
apply (auto dest: Join_welldef_D1 Join_welldef_D2)
done

lemma ex_refinement_thm:
  "[| strict_ex_prop X;
     ∀H. F ok H & G ok H & F ⊔ H ∈ welldef ∩ X --> G ⊔ H ∈ welldef |]
   ==> (G refines F wrt X) = (G iso_refines F wrt X)"
apply (rule_tac x = SKIP in allE, assumption)
apply (simp add: refines_def iso_refines_def strict_ex_refine_lemma_v)
done

lemma strict_uv_refine_lemma:
  "strict_uv_prop X ==>
   (∀H. F ok H & G ok H & F ⊔ H ∈ X --> G ⊔ H ∈ X) = (F ∈ X --> G ∈ X)"
by (unfold strict_uv_prop_def, blast)

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

```

```

apply (unfold strict_uv_prop_def, safe)
apply (erule_tac x = SKIP and P = "%H. PP H --> RR H" for PP RR in allE)
apply (auto dest: Join_welldef_D1 Join_welldef_D2)
done

```

```

lemma uv_refinement_thm:
  "[| strict_uv_prop X;
    ∀H. F ok H & G ok H & F ⊔ H ∈ welldef ∩ X -->
      G ⊔ H ∈ welldef |]
  ==> (G refines F wrt X) = (G iso_refines F wrt X)"
apply (rule_tac x = SKIP in allE, assumption)
apply (simp add: refines_def iso_refines_def strict_uv_refine_lemma_v)
done

```

```

lemma guarantees_equiv:
  "(F ∈ X guarantees Y) = (∀H. H ∈ X → (F component_of H → H ∈ Y))"
by (unfold guar_def component_of_def, auto)

```

```

lemma wg_weakest: "!!X. F ∈ (X guarantees Y) ==> X ⊆ (wg F Y)"
by (unfold wg_def, auto)

```

```

lemma wg_guarantees: "F ∈ ((wg F Y) guarantees Y)"
by (unfold wg_def guar_def, blast)

```

```

lemma wg_equiv: "(H ∈ wg F X) = (F component_of H --> H ∈ X)"
by (simp add: guarantees_equiv wg_def, blast)

```

```

lemma component_of_wg: "F component_of H ==> (H ∈ wg F X) = (H ∈ X)"
by (simp add: wg_equiv)

```

```

lemma wg_finite:
  "∀FF. finite FF ∧ FF ∩ X ≠ {} → OK FF (λF. F)
  → (∀F∈FF. ((⊔ F ∈ FF. F) ∈ wg F X) = ((⊔ F ∈ FF. F) ∈ X))"
apply clarify
apply (subgoal_tac "F component_of (⊔ F ∈ FF. F) ")
apply (drule_tac X = X in component_of_wg, simp)
apply (simp add: component_of_def)
apply (rule_tac x = "⊔ F ∈ (FF-{F}) . F" in exI)
apply (auto intro: JN_Join_diff dest: ok_sym simp add: OK_iff_ok)
done

```

```

lemma wg_ex_prop: "ex_prop X ==> (F ∈ X) = (∀H. H ∈ wg F X)"
apply (simp (no_asm_use) add: ex_prop_equiv wg_equiv)
apply blast
done

```

```

lemma wx_subset: "(wx X) ≤ X"
by (unfold wx_def, auto)

```

```

lemma wx_ex_prop: "ex_prop (wx X)"
apply (simp add: wx_def ex_prop_equiv cong: bex_cong, safe, blast)

```



```

apply force
done

```

```

lemma wx_weakest: "∀ Z. Z ≤ X --> ex_prop Z --> Z ⊆ wx X"
by (auto simp add: wx_def)

```

```

lemma wx'_ex_prop: "ex_prop({F. ∀ G. F ok G --> F ⊔ G ∈ X})"
apply (unfold ex_prop_def, safe)
  apply (drule_tac x = "G ⊔ Ga" in spec)
  apply (force simp add: Join_assoc)
  apply (drule_tac x = "F ⊔ Ga" in spec)
  apply (simp add: ok_commute Join_ac)
done

```

Equivalence with the other definition of wx

```

lemma wx_equiv: "wx X = {F. ∀ G. F ok G --> (F ⊔ G) ∈ X}"
apply (unfold wx_def, safe)
  apply (simp add: ex_prop_def, blast)
  apply (simp (no_asm))
  apply (rule_tac x = "{F. ∀ G. F ok G --> F ⊔ G ∈ X}" in exI, safe)
  apply (rule_tac [2] wx'_ex_prop)
  apply (drule_tac x = SKIP in spec)+
  apply auto
done

```

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

```

lemma guarantees_wx_eq: "(X guarantees Y) = wx(-X ∪ Y)"
by (simp add: guar_def wx_equiv)

```

```

lemma stable_guarantees_Always:
  "Init F ⊆ A ==> F ∈ (stable A) guarantees (Always A)"
apply (rule guaranteesI)
apply (simp add: Join_commute)
apply (rule stable_Join_Always1)
  apply (simp_all add: invariant_def)
done

```

```

lemma constrains_guarantees_leadsTo:
  "F ∈ transient A ==> F ∈ (A co A ∪ B) guarantees (A leadsTo (B-A))"
apply (rule guaranteesI)
apply (rule leadsTo_Basis')
  apply (drule constrains_weaken_R)
  prefer 2 apply assumption
  apply blast
  apply (blast intro: Join_transient_I1)
done

```

```

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  $\times$  UNIV)"
```

definition

```
good_map :: "[ 'a*'b => 'c] => bool"
where "good_map h  $\longleftrightarrow$  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  $\times$  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  $\in$ 
                    project_act h ` Restrict C ` AllowedActs F})"
```

locale *Extend* =

```
fixes f      :: "'c => 'a"
and g        :: "'c => 'b"
and h        :: "'a*'b => 'c"
and slice :: "[ 'c set, 'b] => 'a set"
assumes
  good_h: "good_map h"
defines f_def: "f z == fst (inv h z)"
and g_def: "g z == snd (inv h z)"
```

```
and slice_def: "slice Z y == {x. h(x,y) ∈ Z}"
```

## 10.1 Restrict

```
lemma Restrict_iff [iff]: "((x,y) ∈ Restrict A r) = ((x,y) ∈ r & x ∈ A)"
by (unfold Restrict_def, blast)
```

```
lemma Restrict_UNIV [simp]: "Restrict UNIV = id"
apply (rule ext)
apply (auto simp add: Restrict_def)
done
```

```
lemma Restrict_empty [simp]: "Restrict {} r = {}"
by (auto simp add: Restrict_def)
```

```
lemma Restrict_Int [simp]: "Restrict A (Restrict B r) = Restrict (A ∩ B)
r"
by (unfold Restrict_def, blast)
```

```
lemma Restrict_triv: "Domain r ⊆ A ==> Restrict A r = r"
by (unfold Restrict_def, auto)
```

```
lemma Restrict_subset: "Restrict A r ⊆ r"
by (unfold Restrict_def, auto)
```

```
lemma Restrict_eq_mono:
  "[| A ⊆ B; Restrict B r = Restrict B s |]
   ==> Restrict A r = Restrict A s"
by (unfold Restrict_def, blast)
```

```
lemma Restrict_imageI:
  "[| s ∈ RR; Restrict A r = Restrict A s |]
   ==> Restrict A r ∈ Restrict A ' RR"
by (unfold Restrict_def image_def, auto)
```

```
lemma Domain_Restrict [simp]: "Domain (Restrict A r) = A ∩ Domain r"
by blast
```

```
lemma Image_Restrict [simp]: "(Restrict A r) `` B = r `` (A ∩ B)"
by blast
```

```
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"
apply (simp add: good_map_def)
apply (safe intro!: surj_h)
apply (rule prem)
apply (subst surjective_pairing [symmetric])
apply (subst surj_h [THEN surj_f_inv_f])
apply (rule refl)
done
```

```
lemma good_map_is_surj: "good_map h ==> surj h"
by (unfold good_map_def, auto)
```

```
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"
by (metis UNIV_I f_inv_into_f prod.collapse prem surj_h)
```

## 10.2 Trivial properties of f, g, h

```
context Extend
begin
```

```
lemma f_h_eq [simp]: "f(h(x,y)) = x"
by (simp add: f_def good_h [unfolded good_map_def, THEN conjunct2])
```

```
lemma h_inject1 [dest]: "h(x,y) = h(x',y') ==> x=x'"
apply (drule_tac f = f in arg_cong)
apply (simp add: f_def good_h [unfolded good_map_def, THEN conjunct2])
done
```

```
lemma h_f_g_equiv: "h(f z, g z) == z"
by (simp add: f_def g_def
    good_h [unfolded good_map_def, THEN conjunct1, THEN surj_f_inv_f])
```

```
lemma h_f_g_eq: "h(f z, g z) = z"
by (simp add: h_f_g_equiv)
```

```
lemma split_extended_all:
  "(!!z. PROP P z) == (!!u y. PROP P (h (u, y)))"
proof
  assume allP: " $\bigwedge z. \text{PROP } P \ z$ "
  fix u y
  show "PROP P (h (u, y))" by (rule allP)
next
  assume allPh: " $\bigwedge u y. \text{PROP } P \ (h(u,y))$ "
  fix z
  have Phfgz: "PROP P (h (f z, g z))" by (rule allPh)
  show "PROP P z" by (rule Phfgz [unfolded h_f_g_equiv])
qed
end
```

## 10.3 extend\_set: basic properties

```
lemma project_set_iff [iff]:
  " $(x \in \text{project\_set } h \ C) = (\exists y. h(x,y) \in C)$ "
by (simp add: project_set_def)
```

```
lemma extend_set_mono: " $A \subseteq B \implies \text{extend\_set } h \ A \subseteq \text{extend\_set } h \ B$ "
by (unfold extend_set_def, blast)
```

```

context Extend
begin

lemma mem_extend_set_iff [iff]: "z ∈ extend_set h A = (f z ∈ A)"
apply (unfold extend_set_def)
apply (force intro: h_f_g_eq [symmetric])
done

lemma extend_set_strict_mono [iff]:
  "(extend_set h A ⊆ extend_set h B) = (A ⊆ B)"
by (unfold extend_set_def, force)

lemma (in -) extend_set_empty [simp]: "extend_set h {} = {}"
by (unfold extend_set_def, auto)

lemma extend_set_eq_Collect: "extend_set h {s. P s} = {s. P(f s)}"
by auto

lemma extend_set_sing: "extend_set h {x} = {s. f s = x}"
by auto

lemma extend_set_inverse [simp]: "project_set h (extend_set h C) = C"
by (unfold extend_set_def, auto)

lemma extend_set_project_set: "C ⊆ extend_set h (project_set h C)"
apply (unfold extend_set_def)
apply (auto simp add: split_extended_all, blast)
done

lemma inj_extend_set: "inj (extend_set h)"
apply (rule inj_on_inverseI)
apply (rule extend_set_inverse)
done

lemma extend_set_UNIV_eq [simp]: "extend_set h UNIV = UNIV"
apply (unfold extend_set_def)
apply (auto simp add: split_extended_all)
done

```

#### 10.4 *project\_set*: basic properties

```

lemma project_set_eq: "project_set h C = f ` C"
by (auto intro: f_h_eq [symmetric] simp add: split_extended_all)

```

```

lemma project_set_I: "!!z. z ∈ C ==> f z ∈ project_set h C"
by (auto simp add: split_extended_all)

```

#### 10.5 More laws

```

lemma project_set_extend_set_Int: "project_set h ((extend_set h A) ∩ B) =
  A ∩ (project_set h B)"
  by auto

```

```
lemma project_set_extend_set_Un: "project_set h ((extend_set h A)  $\cup$  B) =
A  $\cup$  (project_set h B)"
  by auto
```

```
lemma (in -) project_set_Int_subset:
  "project_set h (A  $\cap$  B)  $\subseteq$  (project_set h A)  $\cap$  (project_set h B)"
  by auto
```

```
lemma extend_set_Un_distrib: "extend_set h (A  $\cup$  B) = extend_set h A  $\cup$  extend_set
h B"
  by auto
```

```
lemma extend_set_Int_distrib: "extend_set h (A  $\cap$  B) = extend_set h A  $\cap$  extend_set
h B"
  by auto
```

```
lemma extend_set_INT_distrib: "extend_set h ( $\bigcap$  (B ' A)) = ( $\bigcap$  x  $\in$  A. extend_set
h (B x))"
  by auto
```

```
lemma extend_set_Diff_distrib: "extend_set h (A - B) = extend_set h A - extend_set
h B"
  by auto
```

```
lemma extend_set_Union: "extend_set h ( $\bigcup$  A) = ( $\bigcup$  X  $\in$  A. extend_set h X)"
  by blast
```

```
lemma extend_set_subset_Compl_eq: "(extend_set h A  $\subseteq$  - extend_set h B) =
(A  $\subseteq$  - B)"
  by (auto simp: extend_set_def)
```

## 10.6 extend\_act

```
lemma mem_extend_act_iff [iff]: "((h(s,y), h(s',y))  $\in$  extend_act h act) =
((s, s')  $\in$  act)"
  by (auto simp: extend_act_def)
```

```
lemma extend_act_D: "(z, z')  $\in$  extend_act h act ==> (f z, f z')  $\in$  act"
  by (auto simp: extend_act_def)
```

```
lemma extend_act_inverse [simp]: "project_act h (extend_act h act) = act"
  unfolding extend_act_def project_act_def by blast
```

```
lemma project_act_extend_act_restrict [simp]:
  "project_act h (Restrict C (extend_act h act)) =
  Restrict (project_set h C) act"
  unfolding extend_act_def project_act_def by blast
```

```
lemma subset_extend_act_D: "act'  $\subseteq$  extend_act h act ==> project_act h act'
 $\subseteq$  act"
  unfolding extend_act_def project_act_def by force
```

```

lemma inj_extend_act: "inj (extend_act h)"
  apply (rule inj_on_inverseI)
  apply (rule extend_act_inverse)
  done

lemma extend_act_Image [simp]:
  "extend_act h act ' ' (extend_set h A) = extend_set h (act ' ' A)"
  unfolding extend_set_def extend_act_def by force

lemma extend_act_strict_mono [iff]:
  "(extend_act h act'  $\subseteq$  extend_act h act) = (act'  $\leq$  act)"
  by (auto simp: extend_act_def)

lemma [iff]: "(extend_act h act = extend_act h act') = (act = act')"
  by (rule inj_extend_act [THEN inj_eq])

lemma (in -) Domain_extend_act:
  "Domain (extend_act h act) = extend_set h (Domain act)"
  unfolding extend_set_def extend_act_def by force

lemma extend_act_Id [simp]: "extend_act h Id = Id"
  unfolding extend_act_def by (force intro: h_f_g_eq [symmetric])

lemma project_act_I: "!!z z'. (z, z')  $\in$  act ==> (f z, f z')  $\in$  project_act
  h act"
  unfolding project_act_def by (force simp add: split_extended_all)

lemma project_act_Id [simp]: "project_act h Id = Id"
  unfolding project_act_def by force

lemma Domain_project_act: "Domain (project_act h act) = project_set h (Domain
  act)"
  unfolding project_act_def by (force simp add: split_extended_all)

```

## 10.7 extend

Basic properties

```

lemma (in -) Init_extend [simp]:
  "Init (extend h F) = extend_set h (Init F)"
  by (auto simp: extend_def)

lemma (in -) Init_project [simp]:
  "Init (project h C F) = project_set h (Init F)"
  by (auto simp: project_def)

lemma Acts_extend [simp]: "Acts (extend h F) = (extend_act h ' Acts F)"
  by (simp add: extend_def insert_Id_image_Acts)

lemma AllowedActs_extend [simp]:
  "AllowedActs (extend h F) = project_act h - ' AllowedActs F"
  by (simp add: extend_def insert_absorb)

```

```

lemma (in -) Acts_project [simp]:
  "Acts(project h C F) = insert Id (project_act h ' Restrict C ' Acts F)"
  by (auto simp add: project_def image_iff)

lemma AllowedActs_project [simp]:
  "AllowedActs(project h C F) =
    {act. Restrict (project_set h C) act
      ∈ project_act h ' Restrict C ' AllowedActs F}"
  apply (simp (no_asm) add: project_def image_iff)
  apply (subst insert_absorb)
  apply (auto intro!: bexI [of _ Id] simp add: project_act_def)
  done

lemma Allowed_extend: "Allowed (extend h F) = project h UNIV - ' Allowed F"
  by (auto simp add: Allowed_def)

lemma extend_SKIP [simp]: "extend h SKIP = SKIP"
  apply (unfold SKIP_def)
  apply (rule program_equalityI, auto)
  done

lemma (in -) project_set_UNIV [simp]: "project_set h UNIV = UNIV"
  by auto

lemma (in -) project_set_Union: "project_set h ( $\bigcup A$ ) = ( $\bigcup X \in A. project\_set h X$ )"
  by blast

lemma (in -) project_act_Restrict_subset:
  "project_act h (Restrict C act)  $\subseteq$  Restrict (project_set h C) (project_act h act)"
  by (auto simp add: project_act_def)

lemma project_act_Restrict_Id_eq: "project_act h (Restrict C Id) = Restrict (project_set h C) Id"
  by (auto simp add: project_act_def)

lemma 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}})"
  apply (rule program_equalityI)
  apply simp
  apply (simp add: image_image)
  apply (simp add: project_def)
  done

lemma extend_inverse [simp]:
  "project h UNIV (extend h F) = F"
  apply (simp (no_asm_simp) add: project_extend_eq)

```



```

      subset_UNIV [THEN subset_trans, THEN Restrict_triv])
apply (rule program_equalityI)
apply (simp_all (no_asm))
apply (subst insert_absorb)
apply (simp (no_asm) add: bexI [of _ Id])
apply auto
apply (simp add: image_def)
using project_act_Id apply blast
apply (simp add: image_def)
apply (rename_tac "act")
apply (rule_tac x = "extend_act h act" in exI)
apply simp
done

lemma inj_extend: "inj (extend h)"
apply (rule inj_on_inverseI)
apply (rule extend_inverse)
done

lemma extend_Join [simp]: "extend h (F  $\sqcup$  G) = extend h F  $\sqcup$  extend h G"
apply (rule program_equalityI)
apply (simp (no_asm) add: extend_set_Int_distrib)
apply (simp add: image_Un, auto)
done

lemma extend_JN [simp]: "extend h (JOIN I F) = ( $\bigsqcup_{i \in I}$  extend h (F i))"
apply (rule program_equalityI)
  apply (simp (no_asm) add: extend_set_INT_distrib)
  apply (simp add: image_UN, auto)
done

lemma extend_mono: "F  $\leq$  G ==> extend h F  $\leq$  extend h G"
  by (force simp add: component_eq_subset)

lemma project_mono: "F  $\leq$  G ==> project h C F  $\leq$  project h C G"
  by (simp add: component_eq_subset, blast)

lemma all_total_extend: "all_total F ==> all_total (extend h F)"
  by (simp add: all_total_def Domain_extend_act)

```

## 10.8 Safety: co, stable

```

lemma extend_constrains:
  "(extend h F  $\in$  (extend_set h A) co (extend_set h B)) =
   (F  $\in$  A co B)"
  by (simp add: constrains_def)

lemma extend_stable:
  "(extend h F  $\in$  stable (extend_set h A)) = (F  $\in$  stable A)"
  by (simp add: stable_def extend_constrains)

lemma extend_invariant:

```

```

"(extend h F ∈ invariant (extend_set h A)) = (F ∈ invariant A)"
by (simp add: invariant_def extend_stable)

```

```

lemma extend_constrains_project_set:
  "extend h F ∈ A co B ==> F ∈ (project_set h A) co (project_set h B)"
by (auto simp add: constrains_def, force)

```

```

lemma extend_stable_project_set:
  "extend h F ∈ stable A ==> F ∈ stable (project_set h A)"
by (simp add: stable_def extend_constrains_project_set)

```

## 10.9 Weak safety primitives: Co, Stable

```

lemma reachable_extend_f: "p ∈ reachable (extend h F) ==> f p ∈ reachable
F"
by (induct set: reachable) (auto intro: reachable.intros simp add: extend_act_def
image_iff)

```

```

lemma h_reachable_extend: "h(s,y) ∈ reachable (extend h F) ==> s ∈ reachable
F"
by (force dest!: reachable_extend_f)

```

```

lemma reachable_extend_eq: "reachable (extend h F) = extend_set h (reachable
F)"
apply (unfold extend_set_def)
apply (rule equalityI)
apply (force intro: h_f_g_eq [symmetric] dest!: reachable_extend_f, clarify)
apply (erule reachable.induct)
apply (force intro: reachable.intros)+
done

```

```

lemma extend_Constrains:
  "(extend h F ∈ (extend_set h A) Co (extend_set h B)) =
  (F ∈ A Co B)"
by (simp add: Constrains_def reachable_extend_eq extend_constrains
extend_set_Int_distrib [symmetric])

```

```

lemma extend_Stable: "(extend h F ∈ Stable (extend_set h A)) = (F ∈ Stable
A)"
by (simp add: Stable_def extend_Constrains)

```

```

lemma extend_Always: "(extend h F ∈ Always (extend_set h A)) = (F ∈ Always
A)"
by (simp add: Always_def extend_Stable)

```

```

lemma (in -) project_act_mono:
  "D ⊆ C ==>
  project_act h (Restrict D act) ⊆ project_act h (Restrict C act)"

```

```

by (auto simp add: project_act_def)

lemma project_constrains_mono:
  "[| D ⊆ C; project h C F ∈ A co B |] ==> project h D F ∈ A co B"
apply (auto simp add: constrains_def)
apply (drule project_act_mono, blast)
done

lemma project_stable_mono:
  "[| D ⊆ C; project h C F ∈ stable A |] ==> project h D F ∈ stable A"
by (simp add: stable_def project_constrains_mono)

lemma project_constrains:
  "(project h C F ∈ A co B) =
   (F ∈ (C ∩ extend_set h A) co (extend_set h B) & A ⊆ B)"
apply (unfold constrains_def)
apply (auto intro!: project_act_I simp add: ball_Un)
apply (force intro!: project_act_I dest!: subsetD)

apply (unfold project_act_def)
apply (force dest!: subsetD)
done

lemma project_stable: "(project h UNIV F ∈ stable A) = (F ∈ stable (extend_set
h A))"
by (simp add: stable_def project_constrains)

lemma project_stable_I: "F ∈ stable (extend_set h A) ==> project h C F ∈
stable A"
apply (drule project_stable [THEN iffD2])
apply (blast intro: project_stable_mono)
done

lemma Int_extend_set_lemma:
  "A ∩ extend_set h ((project_set h A) ∩ B) = A ∩ extend_set h B"
by (auto simp add: split_extended_all)

lemma project_constrains_project_set:
  "G ∈ C co B ==> project h C G ∈ project_set h C co project_set h B"
by (simp add: constrains_def project_def project_act_def, blast)

lemma project_stable_project_set:
  "G ∈ stable C ==> project h C G ∈ stable (project_set h C)"
by (simp add: stable_def project_constrains_project_set)

```

## 10.10 Progress: transient, ensures

```

lemma extend_transient:
  "(extend h F ∈ transient (extend_set h A)) = (F ∈ transient A)"
by (auto simp add: transient_def extend_set_subset_Compl_eq Domain_extend_act)

lemma extend Ensures:

```

```

      "(extend h F ∈ (extend_set h A) ensures (extend_set h B)) =
       (F ∈ A ensures B)"
    by (simp add: ensures_def extend_constrains extend_transient
        extend_set_Un_distrib [symmetric] extend_set_Diff_distrib [symmetric])

lemma leadsTo_imp_extend_leadsTo:
  "F ∈ A leadsTo B
   ==> extend h F ∈ (extend_set h A) leadsTo (extend_set h B)"
apply (erule leadsTo_induct)
  apply (simp add: leadsTo_Basis extend_ensures)
  apply (blast intro: leadsTo_Trans)
apply (simp add: leadsTo_UN extend_set_Union)
done

```

### 10.11 Proving the converse takes some doing!

```

lemma slice_iff [iff]: "(x ∈ slice C y) = (h(x,y) ∈ C)"
  by (simp add: slice_def)

lemma slice_Union: "slice (⋃ S) y = (⋃ x ∈ S. slice x y)"
  by auto

lemma slice_extend_set: "slice (extend_set h A) y = A"
  by auto

lemma project_set_is_UN_slice: "project_set h A = (⋃ y. slice A y)"
  by auto

lemma extend_transient_slice:
  "extend h F ∈ transient A ==> F ∈ transient (slice A y)"
  by (auto simp: transient_def)

lemma extend_constrains_slice:
  "extend h F ∈ A co B ==> F ∈ (slice A y) co (slice B y)"
  by (auto simp add: constrains_def)

lemma extend_ensures_slice:
  "extend h F ∈ A ensures B ==> F ∈ (slice A y) ensures (project_set h
  B)"
apply (auto simp add: ensures_def extend_constrains extend_transient)
apply (erule_tac [2] extend_transient_slice [THEN transient_strengthen])
apply (erule extend_constrains_slice [THEN constrains_weaken], auto)
done

lemma leadsTo_slice_project_set:
  "∀ y. F ∈ (slice B y) leadsTo CU ==> F ∈ (project_set h B) leadsTo CU"
apply (simp add: project_set_is_UN_slice)
apply (blast intro: leadsTo_UN)
done

lemma extend_leadsTo_slice [rule_format]:
  "extend h F ∈ AU leadsTo BU
   ==> ∀ y. F ∈ (slice AU y) leadsTo (project_set h BU)"

```

```

apply (erule leadsTo_induct)
  apply (blast intro: extend_ensures_slice)
  apply (blast intro: leadsTo_slice_project_set leadsTo_Trans)
apply (simp add: leadsTo_UN slice_Union)
done

lemma extend_leadsTo:
  "(extend h F ∈ (extend_set h A) leadsTo (extend_set h B)) =
   (F ∈ A leadsTo B)"
apply safe
apply (erule_tac [2] leadsTo_imp_extend_leadsTo)
apply (drule extend_leadsTo_slice)
apply (simp add: slice_extend_set)
done

lemma extend_LeadsTo:
  "(extend h F ∈ (extend_set h A) LeadsTo (extend_set h B)) =
   (F ∈ A LeadsTo B)"
by (simp add: LeadsTo_def reachable_extend_eq extend_leadsTo
  extend_set_Int_distrib [symmetric])

```

## 10.12 preserves

```

lemma project_preserves_I:
  "G ∈ preserves (v o f) ==> project h C G ∈ preserves v"
by (auto simp add: preserves_def project_stable_I extend_set_eq_Collect)

lemma project_preserves_id_I:
  "G ∈ preserves f ==> project h C G ∈ preserves id"
by (simp add: project_preserves_I)

lemma extend_preserves:
  "(extend h G ∈ preserves (v o f)) = (G ∈ preserves v)"
by (auto simp add: preserves_def extend_stable [symmetric]
  extend_set_eq_Collect)

lemma inj_extend_preserves: "inj h ==> (extend h G ∈ preserves g)"
by (auto simp add: preserves_def extend_def extend_act_def stable_def
  constrains_def g_def)

```

## 10.13 Guarantees

```

lemma project_extend_Join: "project h UNIV ((extend h F)⊔G) = F⊔(project
h UNIV G)"
  apply (rule program_equalityI)
  apply (auto simp add: project_set_extend_set_Int image_iff)
  apply (metis Un_iff extend_act_inverse image_iff)
  apply (metis Un_iff extend_act_inverse image_iff)
done

lemma extend_Join_eq_extend_D:
  "(extend h F)⊔G = extend h H ==> H = F⊔(project h UNIV G)"
apply (drule_tac f = "project h UNIV" in arg_cong)

```

```

apply (simp add: project_extend_Join)
done

```

```

lemma ok_extend_imp_ok_project: "extend h F ok G ==> F ok project h UNIV
G"
apply (auto simp add: ok_def)
apply (drule subsetD)
apply (auto intro!: rev_image_eqI)
done

```

```

lemma ok_extend_iff: "(extend h F ok extend h G) = (F ok G)"
apply (simp add: ok_def, safe)
apply force+
done

```

```

lemma OK_extend_iff: "OK I (%i. extend h (F i)) = (OK I F)"
apply (unfold OK_def, safe)
apply (drule_tac x = i in bspec)
apply (drule_tac [2] x = j in bspec)
apply force+
done

```

```

lemma guarantees_imp_extend_guarantees:
  "F ∈ X guarantees Y ==>
    extend h F ∈ (extend h ' X) guarantees (extend h ' Y)"
apply (rule guaranteesI, clarify)
apply (blast dest: ok_extend_imp_ok_project extend_Join_eq_extend_D
guaranteesD)
done

```

```

lemma extend_guarantees_imp_guarantees:
  "extend h F ∈ (extend h ' X) guarantees (extend h ' Y)
==> F ∈ X guarantees Y"
apply (auto simp add: guar_def)
apply (drule_tac x = "extend h G" in spec)
apply (simp del: extend_Join
      add: extend_Join [symmetric] ok_extend_iff
      inj_extend [THEN inj_image_mem_iff])
done

```

```

lemma extend_guarantees_eq:
  "(extend h F ∈ (extend h ' X) guarantees (extend h ' Y)) =
  (F ∈ X guarantees Y)"
by (blast intro: guarantees_imp_extend_guarantees
    extend_guarantees_imp_guarantees)

```

```

end

```

```

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_f_inv_f [simp]

declare Extend.intro [simp,intro]

lemma good_map_bij [simp,intro]: "bij h ==> good_map (%(x,u). h x)"
apply (rule good_mapI)
apply (unfold bij_def inj_on_def surj_def, auto)
done

lemma fst_o_inv_eq_inv: "bij h ==> fst (inv (%(x,u). h x) s) = inv h s"
apply (unfold bij_def split_def, clarify)
apply (subgoal_tac "surj (%p. h (fst p))")
  prefer 2 apply (simp add: surj_def)
apply (erule injD)
apply (simp (no_asm_simp) add: surj_f_inv_f)
apply (erule surj_f_inv_f)
done

lemma mem_rename_set_iff: "bij h ==> z ∈ h'A = (inv h z ∈ A)"
by (force simp add: bij_is_inj bij_is_surj [THEN surj_f_inv_f])

lemma extend_set_eq_image [simp]: "extend_set (%(x,u). h x) A = h'A"
by (force simp add: extend_set_def)

lemma Init_rename [simp]: "Init (rename h F) = h'(Init F)"
by (simp add: rename_def)

```

### 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)"
apply (unfold bij_def)
apply (rule ext)
apply (force simp add: extend_set_def project_set_def surj_f_inv_f)
done

lemma bij_extend_act_eq_project_act: "bij h
   ==> extend_act (%(x,u::'c). h x) = project_act (%(x,u::'c). inv h x)"
apply (rule ext)
apply (force simp add: extend_act_def project_act_def bij_def surj_f_inv_f)
done

lemma bij_extend_act: "bij h ==> bij (extend_act (%(x,u::'c). h x))"

```

```

apply (rule bijI)
apply (rule Extend.inj_extend_act)
apply simp
apply (simp add: bij_extend_act_eq_project_act)
apply (rule surjI)
apply (rule Extend.extend_act_inverse)
apply (blast intro: bij_imp_bij_inv)
done

lemma bij_project_act: "bij h ==> bij (project_act (%(x,u::'c). h x))"
apply (frule bij_imp_bij_inv [THEN bij_extend_act])
apply (simp add: bij_extend_act_eq_project_act bij_imp_bij_inv inv_inv_eq)
done

lemma bij_inv_project_act_eq: "bij h ==> inv (project_act (%(x,u::'c). inv
h x)) =
      project_act (%(x,u::'c). h x)"
apply (simp (no_asm_simp) add: bij_extend_act_eq_project_act [symmetric])
apply (rule surj_imp_inv_eq)
apply (blast intro!: bij_extend_act bij_is_surj)
apply (simp (no_asm_simp) add: Extend.extend_act_inverse)
done

lemma extend_inv: "bij h
==> extend (%(x,u::'c). inv h x) = project (%(x,u::'c). h x) UNIV"
apply (frule bij_imp_bij_inv)
apply (rule ext)
apply (rule program_equalityI)
  apply (simp (no_asm_simp) add: extend_set_inv)
  apply (simp add: Extend.project_act_Id Extend.Acts_extend
    insert_Id_image_Acts bij_extend_act_eq_project_act inv_inv_eq)
apply (simp add: Extend.AllowedActs_extend Extend.AllowedActs_project
  bij_project_act bij_vimage_eq_inv_image bij_inv_project_act_eq)
done

lemma rename_inv_rename [simp]: "bij h ==> rename (inv h) (rename h F) =
F"
by (simp add: rename_def extend_inv Extend.extend_inverse)

lemma rename_rename_inv [simp]: "bij h ==> rename h (rename (inv h) F) =
F"
apply (frule bij_imp_bij_inv)
apply (erule inv_inv_eq [THEN subst], erule rename_inv_rename)
done

lemma rename_inv_eq: "bij h ==> rename (inv h) = inv (rename h)"
by (rule inv_equality [symmetric], auto)

lemma bij_extend: "bij h ==> bij (extend (%(x,u::'c). h x))"
apply (rule bijI)
apply (blast intro: Extend.inj_extend)
apply (rule_tac f = "extend (% (x,u) . inv h x)" in surjI)

```



```

apply (subst (1 2) inv_inv_eq [of h, symmetric], assumption+)
apply (simp add: bij_imp_bij_inv extend_inv [of "inv h"])
apply (simp add: inv_inv_eq)
apply (rule Extend.extend_inverse)
apply (simp add: bij_imp_bij_inv)
done

lemma bij_project: "bij h ==> bij (project (%(x,u::'c). h x) UNIV)"
apply (subst extend_inv [symmetric])
apply (auto simp add: bij_imp_bij_inv bij_extend)
done

lemma inv_project_eq:
  "bij h
   ==> inv (project (%(x,u::'c). h x) UNIV) = extend (%(x,u::'c). h x)"
apply (rule inj_imp_inv_eq)
apply (erule bij_project [THEN bij_is_inj])
apply (simp (no_asm_simp) add: Extend.extend_inverse)
done

lemma Allowed_rename [simp]:
  "bij h ==> Allowed (rename h F) = rename h ' Allowed F"
apply (simp (no_asm_simp) add: rename_def Extend.Allowed_extend)
apply (subst bij_vimage_eq_inv_image)
apply (rule bij_project, blast)
apply (simp (no_asm_simp) add: inv_project_eq)
done

lemma bij_rename: "bij h ==> bij (rename h)"
apply (simp (no_asm_simp) add: rename_def bij_extend)
done
lemmas surj_rename = bij_rename [THEN bij_is_surj]

lemma inj_rename_imp_inj: "inj (rename h) ==> inj h"
apply (unfold inj_on_def, auto)
apply (drule_tac x = "mk_program ({x}, {}, {})" in spec)
apply (drule_tac x = "mk_program ({y}, {}, {})" in spec)
apply (auto simp add: program_equality_iff rename_def extend_def)
done

lemma surj_rename_imp_surj: "surj (rename h) ==> surj h"
apply (unfold surj_def, auto)
apply (drule_tac x = "mk_program ({y}, {}, {})" in spec)
apply (auto simp add: program_equality_iff rename_def extend_def)
done

lemma bij_rename_imp_bij: "bij (rename h) ==> bij h"
apply (unfold bij_def)
apply (simp (no_asm_simp) add: inj_rename_imp_inj surj_rename_imp_surj)
done

lemma bij_rename_iff [simp]: "bij (rename h) = bij h"
by (blast intro: bij_rename bij_rename_imp_bij)

```

## 11.2 the lattice operations

```
lemma rename_SKIP [simp]: "bij h ==> rename h SKIP = SKIP"
by (simp add: rename_def Extend.extend_SKIP)
```

```
lemma rename_Join [simp]:
  "bij h ==> rename h (F  $\sqcup$  G) = rename h F  $\sqcup$  rename h G"
by (simp add: rename_def Extend.extend_Join)
```

```
lemma rename_JN [simp]:
  "bij h ==> rename h (JOIN I F) = ( $\bigsqcup$  i  $\in$  I. rename h (F i))"
by (simp add: rename_def Extend.extend_JN)
```

## 11.3 Strong Safety: co, stable

```
lemma rename_constrains:
  "bij h ==> (rename h F  $\in$  (h'A) co (h'B)) = (F  $\in$  A co B)"
apply (unfold rename_def)
apply (subst extend_set_eq_image [symmetric])
apply (erule good_map_bij [THEN Extend.intro, THEN Extend.extend_constrains])
done
```

```
lemma rename_stable:
  "bij h ==> (rename h F  $\in$  stable (h'A)) = (F  $\in$  stable A)"
apply (simp add: stable_def rename_constrains)
done
```

```
lemma rename_invariant:
  "bij h ==> (rename h F  $\in$  invariant (h'A)) = (F  $\in$  invariant A)"
apply (simp add: invariant_def rename_stable bij_is_inj [THEN inj_image_subset_iff])
done
```

```
lemma rename_increasing:
  "bij h ==> (rename h F  $\in$  increasing func) = (F  $\in$  increasing (func o
h))"
apply (simp add: increasing_def rename_stable [symmetric] bij_image_Collect_eq
bij_is_surj [THEN surj_f_inv_f])
done
```

## 11.4 Weak Safety: Co, Stable

```
lemma reachable_rename_eq:
  "bij h ==> reachable (rename h F) = h ' (reachable F)"
apply (simp add: rename_def Extend.reachable_extend_eq)
done
```

```
lemma rename_Constrains:
  "bij h ==> (rename h F  $\in$  (h'A) Co (h'B)) = (F  $\in$  A Co B)"
by (simp add: Constrains_def reachable_rename_eq rename_constrains
bij_is_inj image_Int [symmetric])
```

```
lemma rename_Stable:
  "bij h ==> (rename h F  $\in$  Stable (h'A)) = (F  $\in$  Stable A)"
by (simp add: Stable_def rename_Constrains)
```

```

lemma rename_Always: "bij h ==> (rename h F ∈ Always (h'A)) = (F ∈ Always
A)"
by (simp add: Always_def rename_Stable bij_is_inj [THEN inj_image_subset_iff])

lemma rename_Increasing:
  "bij h ==> (rename h F ∈ Increasing func) = (F ∈ Increasing (func o
h))"
by (simp add: Increasing_def rename_Stable [symmetric] bij_image_Collect_eq

      bij_is_surj [THEN surj_f_inv_f])

```

## 11.5 Progress: transient, ensures

```

lemma rename_transient:
  "bij h ==> (rename h F ∈ transient (h'A)) = (F ∈ transient A)"
apply (unfold rename_def)
apply (subst extend_set_eq_image [symmetric])
apply (erule good_map_bij [THEN Extend.intro, THEN Extend.extend_transient])
done

```

```

lemma rename Ensures:
  "bij h ==> (rename h F ∈ (h'A) ensures (h'B)) = (F ∈ A ensures B)"
apply (unfold rename_def)
apply (subst extend_set_eq_image [symmetric])
apply (erule good_map_bij [THEN Extend.intro, THEN Extend.extend Ensures])
done

```

```

lemma rename_leadsTo:
  "bij h ==> (rename h F ∈ (h'A) leadsTo (h'B)) = (F ∈ A leadsTo B)"
apply (unfold rename_def)
apply (subst extend_set_eq_image [symmetric])
apply (erule good_map_bij [THEN Extend.intro, THEN Extend.extend_leadsTo])
done

```

```

lemma rename_LeadsTo:
  "bij h ==> (rename h F ∈ (h'A) LeadsTo (h'B)) = (F ∈ A LeadsTo B)"
apply (unfold rename_def)
apply (subst extend_set_eq_image [symmetric])
apply (erule good_map_bij [THEN Extend.intro, THEN Extend.extend_LeadsTo])
done

```

```

lemma rename_rename_guarantees_eq:
  "bij h ==> (rename h F ∈ (rename h ' X) guarantees
              (rename h ' Y)) =
              (F ∈ X guarantees Y)"
apply (unfold rename_def)
apply (subst good_map_bij [THEN Extend.intro, THEN Extend.extend_guarantees_eq
[symmetric]], assumption)
apply (simp (no_asm_simp) add: fst_o_inv_eq_inv o_def)
done

```

```

lemma rename_guarantees_eq_rename_inv:
  "bij h ==> (rename h F ∈ X guarantees Y) =
              (F ∈ (rename (inv h) ' X) guarantees

```

```

      (rename (inv h) ' Y))"
  apply (subst rename_rename_guarantees_eq [symmetric], assumption)
  apply (simp add: o_def bij_is_surj [THEN surj_f_inv_f] image_comp)
done

lemma rename_preserves:
  "bij h ==> (rename h G ∈ preserves v) = (G ∈ preserves (v o h))"
  apply (subst good_map_bij [THEN Extend.intro, THEN Extend.extend_preserves
    [symmetric]], assumption)
  apply (simp add: o_def fst_o_inv_eq_inv rename_def bij_is_surj [THEN surj_f_inv_f])
done

lemma ok_rename_iff [simp]: "bij h ==> (rename h F ok rename h G) = (F ok
  G)"
  by (simp add: Extend.ok_extend_iff rename_def)

lemma OK_rename_iff [simp]: "bij h ==> OK I (%i. rename h (F i)) = (OK I
  F)"
  by (simp add: Extend.OK_extend_iff rename_def)

```

## 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)"
  apply auto
  defer 1
  apply (rename_tac F)
  apply (subgoal_tac "∃ G. F = rename h G")
  apply (auto intro!: bij_eq_rename simp add: rename_constrains)
done

lemma rename_image_stable: "bij h ==> rename h ' stable A = stable (h ' A)"
  apply auto
  defer 1
  apply (rename_tac F)
  apply (subgoal_tac "∃ G. F = rename h G")
  apply (auto intro!: bij_eq_rename simp add: rename_stable)
done

lemma rename_image_increasing:
  "bij h ==> rename h ' increasing func = increasing (func o inv h)"
  apply auto
  defer 1
  apply (rename_tac F)
  apply (subgoal_tac "∃ G. F = rename h G")
  apply (auto intro!: bij_eq_rename simp add: rename_increasing o_def bij_is_inj)
done

lemma rename_image_invariant:
  "bij h ==> rename h ' invariant A = invariant (h ' A)"

```

```

apply auto
  defer 1
    apply (rename_tac F)
    apply (subgoal_tac " $\exists G. F = \text{rename } h \ G$ ")
    apply (auto intro!: bij_eq_rename simp add: rename_invariant)
  done

lemma rename_image_Constrains:
  " $\text{bij } h \implies \text{rename } h \ ' \ (A \text{ Co } B) = (h \ ' \ A) \text{ Co } (h \ ' \ B)$ "
apply auto
  defer 1
    apply (rename_tac F)
    apply (subgoal_tac " $\exists G. F = \text{rename } h \ G$ ")
    apply (auto intro!: bij_eq_rename simp add: rename_Constrains)
  done

lemma rename_image_preserves:
  " $\text{bij } h \implies \text{rename } h \ ' \ \text{preserves } v = \text{preserves } (v \circ \text{inv } h)$ "
by (simp add: o_def rename_image_stable preserves_def bij_image_INT
    bij_image_Collect_eq)

lemma rename_image_Stable:
  " $\text{bij } h \implies \text{rename } h \ ' \ \text{Stable } A = \text{Stable } (h \ ' \ A)$ "
apply auto
  defer 1
    apply (rename_tac F)
    apply (subgoal_tac " $\exists G. F = \text{rename } h \ G$ ")
    apply (auto intro!: bij_eq_rename simp add: rename_Stable)
  done

lemma rename_image_Increasing:
  " $\text{bij } h \implies \text{rename } h \ ' \ \text{Increasing } \text{func} = \text{Increasing } (\text{func} \circ \text{inv } h)$ "
apply auto
  defer 1
    apply (rename_tac F)
    apply (subgoal_tac " $\exists G. F = \text{rename } h \ G$ ")
    apply (auto intro!: bij_eq_rename simp add: rename_Increasing o_def bij_is_inj)
  done

lemma rename_image_Always: " $\text{bij } h \implies \text{rename } h \ ' \ \text{Always } A = \text{Always } (h \ ' \ A)$ "
apply auto
  defer 1
    apply (rename_tac F)
    apply (subgoal_tac " $\exists G. F = \text{rename } h \ G$ ")
    apply (auto intro!: bij_eq_rename simp add: rename_Always)
  done

lemma rename_image_leadsTo:
  " $\text{bij } h \implies \text{rename } h \ ' \ (A \text{ leadsTo } B) = (h \ ' \ A) \text{ leadsTo } (h \ ' \ B)$ "
apply auto
  defer 1
    apply (rename_tac F)
    apply (subgoal_tac " $\exists G. F = \text{rename } h \ G$ ")
    apply (auto intro!: bij_eq_rename simp add: rename_leadsTo)
  done

```

```

done

lemma rename_image_LeadsTo:
  "bij h ==> rename h ' (A LeadsTo B) = (h ' A) LeadsTo (h'B)"
apply auto
defer 1
  apply (rename_tac F)
  apply (subgoal_tac "∃ G. F = rename h G")
  apply (auto intro!: bij_eq_rename simp add: rename_LeadsTo)
done

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"
by (rule ext, simp)

```

```

lemma insert_map_delete_map_eq: "(insert_map i x (delete_map i g)) = g(i:=x)"
apply (rule ext)
apply (auto split: nat_diff_split)
done

```

## 12.1 Injectiveness proof

```
lemma insert_map_inject1: "(insert_map i x f) = (insert_map i y g) ==> x=y"
by (drule_tac x = i in fun_cong, simp)
```

```
lemma insert_map_inject2: "(insert_map i x f) = (insert_map i y g) ==> f=g"
apply (drule_tac f = "delete_map i" in arg_cong)
apply (simp add: insert_map_inverse)
done
```

```
lemma insert_map_inject':
  "(insert_map i x f) = (insert_map i y g) ==> x=y & f=g"
by (blast dest: insert_map_inject1 insert_map_inject2)
```

```
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'))"
by (unfold lift_map_def, auto)
```

```
lemma drop_map_lift_map_eq [simp]: "!!s. drop_map i (lift_map i s) = s"
apply (unfold lift_map_def drop_map_def)
apply (force intro: insert_map_inverse)
done
```

```
lemma inj_lift_map: "inj (lift_map i)"
apply (unfold lift_map_def)
apply (rule inj_onI, auto)
done
```

## 12.2 Surjectiveness proof

```
lemma lift_map_drop_map_eq [simp]: "!!s. lift_map i (drop_map i s) = s"
apply (unfold lift_map_def drop_map_def)
apply (force simp add: insert_map_delete_map_eq)
done
```

```
lemma drop_map_inject [dest!]: "(drop_map i s) = (drop_map i s') ==> s=s'"
by (drule_tac f = "lift_map i" in arg_cong, simp)
```

```
lemma surj_lift_map: "surj (lift_map i)"
apply (rule surjI)
apply (rule lift_map_drop_map_eq)
done
```

```
lemma bij_lift_map [iff]: "bij (lift_map i)"
by (simp add: bij_def inj_lift_map surj_lift_map)
```

```
lemma inv_lift_map_eq [simp]: "inv (lift_map i) = drop_map i"
by (rule inv_equality, auto)
```

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

```
by (rule inv_equality, auto)
```

```
lemma bij_drop_map [iff]: "bij (drop_map i)"
by (simp del: inv_lift_map_eq add: inv_lift_map_eq [symmetric] bij_imp_bij_inv)
```

```
lemma sub_apply [simp]: "sub i f = f i"
by (simp add: sub_def)
```

```
lemma all_total_lift: "all_total F ==> all_total (lift i F)"
by (simp add: lift_def rename_def Extend.all_total_extend)
```

```
lemma insert_map_upd_same: "(insert_map i t f)(i := s) = insert_map i s f"
by (rule ext, auto)
```

```
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)))"
apply (rule ext)
apply (simp split: nat_diff_split)
```

This simplification is VERY slow

done

```
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'"
apply (subst insert_map_upd_same [symmetric])
apply (erule ssubst)
apply (simp only: insert_map_upd if_False split: if_split, blast)
done
```

```
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))"
apply (unfold lift_map_def, auto)
apply (blast dest: insert_map_eq_diff)
done
```

### 12.3 The Operator `lift_set`

```
lemma lift_set_empty [simp]: "lift_set i {} = {}"
by (unfold lift_set_def, auto)
```

```
lemma lift_set_iff: "(lift_map i x ∈ lift_set i A) = (x ∈ A)"
apply (unfold lift_set_def)
apply (rule inj_lift_map [THEN inj_image_mem_iff])
done
```

```
lemma lift_set_iff2 [iff]:
  "((f,uu) ∈ lift_set i A) = ((f i, (delete_map i f, uu)) ∈ A)"
```



```
by (simp add: lift_set_def mem_rename_set_iff drop_map_def)
```

```
lemma lift_set_mono: "A  $\subseteq$  B ==> lift_set i A  $\subseteq$  lift_set i B"
apply (unfold lift_set_def)
apply (erule image_mono)
done
```

```
lemma lift_set_Un_distrib: "lift_set i (A  $\cup$  B) = lift_set i A  $\cup$  lift_set i B"
by (simp add: lift_set_def image_Un)
```

```
lemma lift_set_Diff_distrib: "lift_set i (A-B) = lift_set i A - lift_set i B"
apply (unfold lift_set_def)
apply (rule inj_lift_map [THEN image_set_diff])
done
```

## 12.4 The Lattice Operations

```
lemma bij_lift [iff]: "bij (lift i)"
by (simp add: lift_def)
```

```
lemma lift_SKIP [simp]: "lift i SKIP = SKIP"
by (simp add: lift_def)
```

```
lemma lift_Join [simp]: "lift i (F  $\sqcup$  G) = lift i F  $\sqcup$  lift i G"
by (simp add: lift_def)
```

```
lemma lift_JN [simp]: "lift j (JOIN I F) = ( $\bigsqcup$  i  $\in$  I. lift j (F i))"
by (simp add: lift_def)
```

## 12.5 Safety: constrains, stable, invariant

```
lemma lift_constrains:
  "(lift i F  $\in$  (lift_set i A) co (lift_set i B)) = (F  $\in$  A co B)"
by (simp add: lift_def lift_set_def rename_constrains)
```

```
lemma lift_stable:
  "(lift i F  $\in$  stable (lift_set i A)) = (F  $\in$  stable A)"
by (simp add: lift_def lift_set_def rename_stable)
```

```
lemma lift_invariant:
  "(lift i F  $\in$  invariant (lift_set i A)) = (F  $\in$  invariant A)"
by (simp add: lift_def lift_set_def rename_invariant)
```

```
lemma lift_Constrains:
  "(lift i F  $\in$  (lift_set i A) Co (lift_set i B)) = (F  $\in$  A Co B)"
by (simp add: lift_def lift_set_def rename_Constrains)
```

```
lemma lift_Stable:
  "(lift i F  $\in$  Stable (lift_set i A)) = (F  $\in$  Stable A)"
by (simp add: lift_def lift_set_def rename_Stable)
```

```

lemma lift_Always:
  "(lift i F ∈ Always (lift_set i A)) = (F ∈ Always A)"
by (simp add: lift_def lift_set_def rename_Always)

```

## 12.6 Progress: transient, ensures

```

lemma lift_transient:
  "(lift i F ∈ transient (lift_set i A)) = (F ∈ transient A)"
by (simp add: lift_def lift_set_def rename_transient)

```

```

lemma lift Ensures:
  "(lift i F ∈ (lift_set i A) ensures (lift_set i B)) =
   (F ∈ A ensures B)"
by (simp add: lift_def lift_set_def rename Ensures)

```

```

lemma lift_leadsTo:
  "(lift i F ∈ (lift_set i A) leadsTo (lift_set i B)) =
   (F ∈ A leadsTo B)"
by (simp add: lift_def lift_set_def rename_leadsTo)

```

```

lemma lift_LeadsTo:
  "(lift i F ∈ (lift_set i A) LeadsTo (lift_set i B)) =
   (F ∈ A LeadsTo B)"
by (simp add: lift_def lift_set_def rename_LeadsTo)

```

```

lemma lift_lift_guarantees_eq:
  "(lift i F ∈ (lift i ' X) guarantees (lift i ' Y)) =
   (F ∈ X guarantees Y)"
apply (unfold lift_def)
apply (subst bij_lift_map [THEN rename_rename_guarantees_eq, symmetric])
apply (simp add: o_def)
done

```

```

lemma lift_guarantees_eq_lift_inv:
  "(lift i F ∈ X guarantees Y) =
   (F ∈ (rename (drop_map i) ' X) guarantees (rename (drop_map i) ' Y))"
by (simp add: bij_lift_map [THEN rename_guarantees_eq_rename_inv] lift_def)

```

```

lemma lift_preserves_snd_I: "F ∈ preserves snd ==> lift i F ∈ preserves
snd"
apply (drule_tac w1=snd in subset_preserves_o [THEN subsetD])
apply (simp add: lift_def rename_preserves)
apply (simp add: lift_map_def o_def split_def)
done

```

```

lemma delete_map_eqE':
  "(delete_map i g) = (delete_map i g') ==> ∃x. g = g'(i:=x)"
apply (drule_tac f = "insert_map i (g i)" in arg_cong)
apply (simp add: insert_map_delete_map_eq)

```

```

apply (erule exI)
done

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"
by force

lemma vimage_o_fst_eq [simp]: "(f o fst) -' A = (f-'A) × UNIV"
by auto

lemma vimage_sub_eq_lift_set [simp]:
  "(sub i -'A) × UNIV = lift_set i (A × UNIV)"
by auto

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)"
apply (unfold extend_act_def, auto)
apply (rule bexI, auto)
done

lemma preserves_snd_lift_stable:
  "[| F ∈ preserves_snd; i ≠ j |]
   ==> lift j F ∈ stable (lift_set i (A × UNIV))"
apply (auto simp add: lift_def lift_set_def stable_def constrains_def
  rename_def extend_def mem_rename_set_iff)
apply (auto dest!: preserves_imp_eq simp add: lift_map_def drop_map_def)
apply (drule_tac x = i in fun_cong, auto)
done

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))"
apply (cases "i=j")
apply (simp add: lift_def lift_set_def rename_constrains)
apply (erule preserves_snd_lift_stable[THEN stableD, THEN constrains_weaken_R],
  assumption)
apply (erule constrains_imp_subset [THEN lift_set_mono])
done

lemma lift_map_image_Times:
  "lift_map i ' (A × UNIV) =
   (⋃ s ∈ A. ⋃ f. {insert_map i s f}) × UNIV"
apply (auto intro!: bexI image_eqI simp add: lift_map_def)
apply (rule split_conv [symmetric])
done

```

```

lemma lift_preserves_eq:
  "(lift i F ∈ preserves v) = (F ∈ preserves (v o lift_map i))"
by (simp add: lift_def rename_preserves)

```

```

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)"
apply (drule subset_preserves_o [THEN subsetD])
apply (simp add: lift_preserves_eq o_def)
apply (auto cong del: if_weak_cong
  simp add: lift_map_def eq_commute split_def o_def)
done

```

## 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"
by (simp add: fun_eq_iff o_def)

```

```

lemma o_equiv_apply: "f o g = h ==> ∀x. f(g x) = h x"
by (simp add: fun_eq_iff o_def)

```

```

lemma fst_o_lift_map: "sub i o fst o lift_map i = fst"
apply (rule ext)
apply (auto simp add: o_def lift_map_def sub_def)
done

```

```

lemma snd_o_lift_map: "snd o lift_map i = snd o snd"
apply (rule ext)
apply (auto simp add: o_def lift_map_def)
done

```

## 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 (λ(x,(y,y')). h'(h(x,y),y')) act"
apply (auto elim!: rev_bexI simp add: extend_act_def, blast)
done

```

```

lemma project_act_project_act:
  "project_act h (project_act h' act) =
    project_act (λ(x,(y,y')). h'(h(x,y),y')) act"
by (auto elim!: rev_bexI simp add: project_act_def)

```

```

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)}"
by (simp add: extend_act_def project_act_def, blast)

```

## 12.9 OK and "lift"

```

lemma act_in_UNION_preserves_fst:
  "act  $\subseteq$   $\{(x, x') . \text{fst } x = \text{fst } x'\}$  ==> act  $\in \bigcup (\text{Acts } ' (\text{preserves fst}))$ "
apply (rule_tac a = "mk_program (UNIV, {act}, UNIV) " in UN_I)
apply (auto simp add: preserves_def stable_def constrains_def)
done

lemma UNION_OK_lift_I:
  "[|  $\forall i \in I . F i \in \text{preserves snd};$ 
     $\forall i \in I . \bigcup (\text{Acts } ' (\text{preserves fst})) \subseteq \text{AllowedActs } (F i)$  |]
  ==> OK I ( $\lambda i . \text{lift } i (F i)$ )"
apply (auto simp add: OK_def lift_def rename_def Extend.Acts_extend)
apply (simp add: Extend.AllowedActs_extend project_act_extend_act)
apply (rename_tac "act")
apply (subgoal_tac
  "{(x, x') .  $\exists s f u s' f' u' .$ 
     $((s, f, u), s', f', u') \in \text{act} \ \&$ 
     $\text{lift\_map } j \ x = \text{lift\_map } i \ (s, f, u) \ \&$ 
     $\text{lift\_map } j \ x' = \text{lift\_map } i \ (s', f', u')$  }
     $\subseteq \{(x, x') . \text{fst } x = \text{fst } x'\}$ ")
apply (blast intro: act_in_UNION_preserves_fst, clarify)
apply (drule_tac x = j in fun_cong)+
apply (drule_tac x = i in bspec, assumption)
apply (frule preserves_imp_eq, auto)
done

lemma OK_lift_I:
  "[|  $\forall i \in I . F i \in \text{preserves snd};$ 
     $\forall i \in I . \text{preserves fst} \subseteq \text{Allowed } (F i)$  |]
  ==> OK I ( $\lambda i . \text{lift } i (F i)$ )"
by (simp add: safety_prop_AllowedActs_iff_Allowed UNION_OK_lift_I)

lemma Allowed_lift [simp]: "Allowed (lift i F) = lift i ' (Allowed F)"
by (simp add: lift_def)

lemma lift_image_preserves:
  "lift i ' preserves v = preserves (v o drop_map i)"
by (simp add: rename_image_preserves lift_def)

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 ==  $\bigsqcup i \in I . \text{lift } i (F i)$ "

syntax
  "_PLam" :: "[pttrn, nat set, 'b set] => (nat => 'b) set"
  (<(3plam _:_./
  _)> 10)
syntax_consts
  "_PLam" == PLam

```

translations

```
"plam x : A. B" == "CONST PLam A (%x. B)"
```

```
lemma Init_PLam [simp]: "Init (PLam I F) = ( $\bigcap$  i  $\in$  I. lift_set i (Init (F i)))"
```

```
by (simp add: PLam_def lift_def lift_set_def)
```

```
lemma PLam_empty [simp]: "PLam {} F = SKIP"
```

```
by (simp add: PLam_def)
```

```
lemma PLam_SKIP [simp]: "(plam i : I. SKIP) = SKIP"
```

```
by (simp add: PLam_def JN_constant)
```

```
lemma PLam_insert: "PLam (insert i I) F = (lift i (F i))  $\sqcup$  (PLam I F)"
```

```
by (unfold PLam_def, auto)
```

```
lemma PLam_component_iff: "((PLam I F)  $\leq$  H) = ( $\forall$  i  $\in$  I. lift i (F i)  $\leq$  H)"
```

```
by (simp add: PLam_def JN_component_iff)
```

```
lemma component_PLam: "i  $\in$  I ==> lift i (F i)  $\leq$  (PLam I F)"
```

```
apply (unfold PLam_def)
```

```
apply (fast intro: component_JN)
```

```
done
```

```
lemma PLam_constrains:
```

```
"[| i  $\in$  I;  $\forall$  j. F j  $\in$  preserves snd |]"
```

```
==> (PLam I F  $\in$  (lift_set i (A  $\times$  UNIV)) co  
      (lift_set i (B  $\times$  UNIV))) =  
      (F i  $\in$  (A  $\times$  UNIV) co (B  $\times$  UNIV))"
```

```
apply (simp add: PLam_def JN_constrains)
```

```
apply (subst insert_Diff [symmetric], assumption)
```

```
apply (simp add: lift_constrains)
```

```
apply (blast intro: constrains_imp_lift_constrains)
```

```
done
```

```
lemma PLam_stable:
```

```
"[| i  $\in$  I;  $\forall$  j. F j  $\in$  preserves snd |]"
```

```
==> (PLam I F  $\in$  stable (lift_set i (A  $\times$  UNIV))) =  
      (F i  $\in$  stable (A  $\times$  UNIV))"
```

```
by (simp add: stable_def PLam_constrains)
```

```
lemma PLam_transient:
```

```
"i  $\in$  I ==>
```

```
PLam I F  $\in$  transient A = ( $\exists$  i  $\in$  I. lift i (F i)  $\in$  transient A)"
```

```
by (simp add: JN_transient PLam_def)
```

This holds because the  $F j$  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)"
apply (simp add: ensures_def PLam_constrains PLam_transient
  lift_set_Un_distrib [symmetric] lift_set_Diff_distrib [symmetric]
  Times_Un_distrib1 [symmetric] Times_Diff_distrib1 [symmetric])
apply (rule rev_bexI, assumption)
apply (simp add: lift_transient)
done

```

```

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)"
by (rule PLam_ensures [THEN leadsTo_Basis], rule_tac [2] ensuresI)

```

```

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))"
by (auto simp add: PLam_stable invariant_def)

```

```

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)"
by (simp add: PLam_def lift_preserves_sub)

```

```

lemma PLam_preserves_snd [simp,intro]:
  "∀ j. F j ∈ preserves snd ==> PLam I F ∈ preserves snd"
by (simp add: PLam_def lift_preserves_snd_I)

```

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 (λi. lift i (F i)) |]"
  ==> (PLam I F) ∈ X guarantees Y"
apply (unfold PLam_def)
apply (simp add: guarantees_JN_I)
done

```

```

lemma Allowed_PLam [simp]:
  "Allowed (PLam I F) = (⋂ i ∈ I. lift i ' Allowed(F i))"

```

```
by (simp add: PLam_def)
```

```
lemma PLam_preserves [simp]:
  "(PLam I F) ∈ preserves v = (∀ i ∈ I. F i ∈ preserves (v o lift_map i))"
by (simp add: PLam_def lift_def rename_preserves)
```

```
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 ..
```

```
end
```

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

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



**abbreviation**

```

prefixLe :: "[nat list, nat list] => bool" (infixl <prefixLe> 50) where
  "xs prefixLe ys == (xs,ys) ∈ genPrefix Le"

```

**abbreviation**

```

prefixGe :: "[nat list, nat list] => bool" (infixl <prefixGe> 50) where
  "xs prefixGe ys == (xs,ys) ∈ genPrefix Ge"

```

**13.1 preliminary lemmas**

```

lemma Nil_genPrefix [iff]: "([], xs) ∈ genPrefix r"
by (cut_tac genPrefix.Nil [THEN genPrefix.append], auto)

```

```

lemma genPrefix_length_le: "(xs,ys) ∈ genPrefix r ⟹ length xs ≤ length ys"
by (erule genPrefix.induct, auto)

```

**lemma cdlemma:**

```

  "[| (xs', ys') ∈ genPrefix r |]
   ==> (∀x xs. xs' = x#xs ⟶ (∃y ys. ys' = y#ys & (x,y) ∈ r & (xs, ys)
    ∈ genPrefix r))"
apply (erule genPrefix.induct, blast, blast)
apply (force intro: genPrefix.append)
done

```

**lemma cons\_genPrefixE [elim!]:**

```

  "[| (x#xs, zs) ∈ genPrefix r;
    !!y ys. [| zs = y#ys; (x,y) ∈ r; (xs, ys) ∈ genPrefix r |] ==>
    P
    |] ==> P"
by (drule cdlemma, simp, blast)

```

**lemma Cons\_genPrefix\_Cons [iff]:**

```

  "((x#xs,y#ys) ∈ genPrefix r) = ((x,y) ∈ r ∧ (xs,ys) ∈ genPrefix r)"
by (blast intro: genPrefix.prepend)

```

**13.2 genPrefix is a partial order**

```

lemma refl_genPrefix: "refl r ==> refl (genPrefix r)"
apply (unfold refl_on_def, auto)
apply (induct_tac "x")
prefer 2 apply (blast intro: genPrefix.prepend)
apply (blast intro: genPrefix.Nil)
done

```

```

lemma genPrefix_refl [simp]: "refl r ⟹ (1,1) ∈ genPrefix r"
by (erule refl_onD [OF refl_genPrefix UNIV_I])

```

```

lemma genPrefix_mono: "r ≤ s ==> genPrefix r ≤ genPrefix s"
apply clarify
apply (erule genPrefix.induct)
apply (auto intro: genPrefix.append)

```

done

```
lemma append_genPrefix:
  "(xs @ ys, zs) ∈ genPrefix r ⟹ (xs, zs) ∈ genPrefix r"
  by (induct xs arbitrary: zs) auto
```

```
lemma genPrefix_trans_0:
  assumes "(x, y) ∈ genPrefix r"
  shows "⋀z. (y, z) ∈ genPrefix s ⟹ (x, z) ∈ genPrefix (r 0 s)"
  apply (atomize (full))
  using assms
  apply induct
  apply blast
  apply (blast intro: genPrefix.prepend)
  apply (blast dest: append_genPrefix)
  done
```

```
lemma genPrefix_trans:
  "(x, y) ∈ genPrefix r ⟹ (y, z) ∈ genPrefix r ⟹ trans r
   ⟹ (x, z) ∈ genPrefix r"
  apply (rule trans_0_subset [THEN genPrefix_mono, THEN subsetD])
  apply assumption
  apply (blast intro: genPrefix_trans_0)
  done
```

```
lemma prefix_genPrefix_trans:
  "[| x ≤ y; (y, z) ∈ genPrefix r |] ⟹ (x, z) ∈ genPrefix r"
  apply (unfold prefix_def)
  apply (drule genPrefix_trans_0, assumption)
  apply simp
  done
```

```
lemma genPrefix_prefix_trans:
  "[| (x, y) ∈ genPrefix r; y ≤ z |] ⟹ (x, z) ∈ genPrefix r"
  apply (unfold prefix_def)
  apply (drule genPrefix_trans_0, assumption)
  apply simp
  done
```

```
lemma trans_genPrefix: "trans r ⟹ trans (genPrefix r)"
  by (blast intro: transI genPrefix_trans)
```

```
lemma genPrefix_antisym:
  assumes 1: "(xs, ys) ∈ genPrefix r"
  and 2: "antisym r"
  and 3: "(ys, xs) ∈ genPrefix r"
```

```

    shows "xs = ys"
    using 1 3
  proof induct
    case Nil
    then show ?case by blast
  next
    case prepend
    then show ?case using 2 by (simp add: antisym_def)
  next
    case (append xs ys zs)
    then show ?case
      apply -
      apply (subgoal_tac "length zs = 0", force)
      apply (drule genPrefix_length_le)+
      apply (simp del: length_0_conv)
      done
  qed

lemma antisym_genPrefix: "antisym r ==> antisym (genPrefix r)"
  by (blast intro: antisymI genPrefix_antisym)

```

### 13.3 recursion equations

```

lemma genPrefix_Nil [simp]: "((xs, []) ∈ genPrefix r) = (xs = [])"
  by (induct xs) auto

lemma same_genPrefix_genPrefix [simp]:
  "refl r ==> ((xs@ys, xs@zs) ∈ genPrefix r) = ((ys,zs) ∈ genPrefix r)"
  by (induct xs) (simp_all add: refl_on_def)

lemma genPrefix_Cons:
  "((xs, y#ys) ∈ genPrefix r) =
   (xs=[] | (∃z zs. xs=z#zs & (z,y) ∈ r & (zs,ys) ∈ genPrefix r))"
  by (cases xs) auto

lemma genPrefix_take_append:
  "[| refl r; (xs,ys) ∈ genPrefix r |]
   ==> (xs@zs, take (length xs) ys @ zs) ∈ genPrefix r"
  apply (erule genPrefix.induct)
  apply (frule_tac [3] genPrefix_length_le)
  apply (simp_all (no_asm_simp) add: diff_is_0_eq [THEN iffD2])
  done

lemma genPrefix_append_both:
  "[| refl r; (xs,ys) ∈ genPrefix r; length xs = length ys |]
   ==> (xs@zs, ys @ zs) ∈ genPrefix r"
  apply (drule genPrefix_take_append, assumption)
  apply simp
  done

lemma append_cons_eq: "xs @ y # ys = (xs @ [y]) @ ys"
  by auto

```

```

lemma aolemma:
  "[/ (xs,ys) ∈ genPrefix r; refl r [/
    ==> length xs < length ys → (xs @ [ys ! length xs], ys) ∈ genPrefix
  r"
apply (erule genPrefix.induct)
  apply blast
  apply simp

Append case is hardest

apply simp
apply (frule genPrefix_length_le [THEN le_imp_less_or_eq])
apply (erule disjE)
apply (simp_all (no_asm_simp) add: neq_Nil_conv nth_append)
apply (blast intro: genPrefix.append, auto)
apply (subst append_cons_eq, fast intro: genPrefix_append_both genPrefix.append)
done

```

```

lemma append_one_genPrefix:
  "[/ (xs,ys) ∈ genPrefix r; length xs < length ys; refl r [/
    ==> (xs @ [ys ! length xs], ys) ∈ genPrefix r"
by (blast intro: aolemma [THEN mp])

```

```

lemma genPrefix_imp_nth:
  "i < length xs ⇒ (xs, ys) ∈ genPrefix r ⇒ (xs ! i, ys ! i) ∈ r"
  apply (induct xs arbitrary: i ys)
  apply auto
  apply (case_tac i)
  apply auto
  done

```

```

lemma nth_imp_genPrefix:
  "length xs ≤ length ys ⇒
    (∀ i. i < length xs → (xs ! i, ys ! i) ∈ r) ⇒
    (xs, ys) ∈ genPrefix r"
  apply (induct xs arbitrary: ys)
  apply (simp_all add: less_Suc_eq_0_disj all_conj_distrib)
  apply (case_tac ys)
  apply (force+)
  done

```

```

lemma genPrefix_iff_nth:
  "((xs,ys) ∈ genPrefix r) =
    (length xs ≤ length ys & (∀ i. i < length xs → (xs ! i, ys ! i) ∈ r))"
  apply (blast intro: genPrefix_length_le genPrefix_imp_nth nth_imp_genPrefix)
  done

```

### 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)"
by (simp add: prefix_def)

lemma prefix_trans: "!!xs::'a list. [| xs <= ys; ys <= zs |] ==> xs <= zs"
apply (unfold prefix_def)
apply (blast intro: genPrefix_trans)
done

lemma prefix_antisym: "!!xs::'a list. [| xs <= ys; ys <= xs |] ==> xs = ys"
apply (unfold prefix_def)
apply (blast intro: genPrefix_antisym)
done

lemma prefix_less_le_not_le: "!!xs::'a list. (xs < zs) = (xs <= zs & ¬ zs
< xs)"
by (unfold strict_prefix_def, auto)

instance list :: (type) order
by (intro_classes,
    (assumption | rule prefix_refl prefix_trans prefix_antisym
    prefix_less_le_not_le)+)

lemma set_mono: "xs <= ys ==> set xs <= set ys"
apply (unfold prefix_def)
apply (erule genPrefix.induct, auto)
done

lemma Nil_prefix [iff]: "[] <= xs"
by (simp add: prefix_def)

lemma prefix_Nil [simp]: "(xs <= []) = (xs = [])"
by (simp add: prefix_def)

lemma Cons_prefix_Cons [simp]: "(x#xs <= y#ys) = (x=y & xs<=ys)"
by (simp add: prefix_def)

lemma same_prefix_prefix [simp]: "(xs@ys <= xs@zs) = (ys <= zs)"
by (simp add: prefix_def)

lemma append_prefix [iff]: "(xs@ys <= xs) = (ys <= [])"
by (insert same_prefix_prefix [of xs ys "[]"], simp)

lemma prefix_appendI [simp]: "xs <= ys ==> xs <= ys@zs"
apply (unfold prefix_def)
apply (erule genPrefix.append)
done

lemma prefix_Cons:

```

```

"(xs <= y#ys) = (xs=[] | (∃zs. xs=y#zs ∧ zs <= ys))"
by (simp add: prefix_def genPrefix_Cons)

lemma append_one_prefix:
  "[| xs <= ys; length xs < length ys |] ==> xs @ [ys ! length xs] <= ys"
apply (unfold prefix_def)
apply (simp add: append_one_genPrefix)
done

lemma prefix_length_le: "xs <= ys ==> length xs <= length ys"
apply (unfold prefix_def)
apply (erule genPrefix_length_le)
done

lemma splemma: "xs<=ys ==> xs~=ys --> length xs < length ys"
apply (unfold prefix_def)
apply (erule genPrefix.induct, auto)
done

lemma strict_prefix_length_less: "xs < ys ==> length xs < length ys"
apply (unfold strict_prefix_def)
apply (blast intro: splemma [THEN mp])
done

lemma mono_length: "mono length"
by (blast intro: monoI prefix_length_le)

lemma prefix_iff: "(xs <= zs) = (∃ys. zs = xs@ys)"
apply (unfold prefix_def)
apply (auto simp add: genPrefix_iff_nth nth_append)
apply (rule_tac x = "drop (length xs) zs" in exI)
apply (rule nth_equalityI)
apply (simp_all (no_asm_simp) add: nth_append)
done

lemma prefix_snoc [simp]: "(xs <= ys@[y]) = (xs = ys@[y] | xs <= ys)"
apply (simp add: prefix_iff)
apply (rule iffI)
  apply (erule exE)
  apply (rename_tac "zs")
  apply (rule_tac xs = zs in rev_exhaust)
  apply simp
  apply clarify
  apply (simp del: append_assoc add: append_assoc [symmetric], force)
done

lemma prefix_append_iff:
  "(xs <= ys@zs) = (xs <= ys | (∃us. xs = ys@us & us <= zs))"
apply (rule_tac xs = zs in rev_induct)
  apply force
  apply (simp del: append_assoc add: append_assoc [symmetric], force)
done

```

```

lemma common_prefix_linear:
  fixes xs ys zs :: "'a list"
  shows "xs <= zs  $\implies$  ys <= zs  $\implies$  xs <= ys  $\mid$  ys <= xs"
  by (induct zs rule: rev_induct) auto

```

### 13.5 pfixLe, pfixGe: properties inherited from the translations

```

lemma refl_Le [iff]: "refl Le"
by (unfold refl_on_def Le_def, auto)

```

```

lemma antisym_Le [iff]: "antisym Le"
by (unfold antisym_def Le_def, auto)

```

```

lemma trans_Le [iff]: "trans Le"
by (unfold trans_def Le_def, auto)

```

```

lemma pfixLe_refl [iff]: "x pfixLe x"
by simp

```

```

lemma pfixLe_trans: "[| x pfixLe y; y pfixLe z |]  $\implies$  x pfixLe z"
by (blast intro: genPrefix_trans)

```

```

lemma pfixLe_antisym: "[| x pfixLe y; y pfixLe x |]  $\implies$  x = y"
by (blast intro: genPrefix_antisym)

```

```

lemma prefix_imp_pfixLe: "xs <= ys  $\implies$  xs pfixLe ys"
apply (unfold prefix_def Le_def)
apply (blast intro: genPrefix_mono [THEN [2] rev_subsetD])
done

```

```

lemma refl_Ge [iff]: "refl Ge"
by (unfold refl_on_def Ge_def, auto)

```

```

lemma antisym_Ge [iff]: "antisym Ge"
by (unfold antisym_def Ge_def, auto)

```

```

lemma trans_Ge [iff]: "trans Ge"
by (unfold trans_def Ge_def, auto)

```

```

lemma pfixGe_refl [iff]: "x pfixGe x"
by simp

```

```

lemma pfixGe_trans: "[| x pfixGe y; y pfixGe z |]  $\implies$  x pfixGe z"
by (blast intro: genPrefix_trans)

```

```

lemma pfixGe_antisym: "[| x pfixGe y; y pfixGe x |]  $\implies$  x = y"
by (blast intro: genPrefix_antisym)

```

```

lemma prefix_imp_pfixGe: "xs <= ys  $\implies$  xs pfixGe ys"
apply (unfold prefix_def Ge_def)
apply (blast intro: genPrefix_mono [THEN [2] rev_subsetD])
done

```

end

## 14 The Follows Relation of Charpentier and Sivilotte

```

theory Follows
imports SubstAx ListOrder "HOL-Library.Multiset"
begin

definition Follows :: "[ 'a => 'b::{order}, 'a => 'b::{order} ] => 'a program
set" (infixl <Fols> 65) where
  "f Fols g == Increasing g  $\cap$  Increasing f Int
    Always {s. f s  $\leq$  g s} Int
    ( $\bigcap$  k. {s. k  $\leq$  g s} LeadsTo {s. k  $\leq$  f s})"

lemma mono_Always_o:
  "mono h ==> Always {s. f s  $\leq$  g s}  $\subseteq$  Always {s. h (f s)  $\leq$  h (g s)}"
apply (simp add: Always_eq_includes_reachable)
apply (blast intro: monoD)
done

lemma mono_LeadsTo_o:
  "mono (h::'a::order => 'b::order)
    ==> ( $\bigcap$  j. {s. j  $\leq$  g s} LeadsTo {s. j  $\leq$  f s})  $\subseteq$ 
    ( $\bigcap$  k. {s. k  $\leq$  h (g s)} LeadsTo {s. k  $\leq$  h (f s)})"
apply auto
apply (rule single_LeadsTo_I)
apply (drule_tac x = "g s" in spec)
apply (erule LeadsTo_weaken)
apply (blast intro: monoD order_trans)+
done

lemma Follows_constant [iff]: "F  $\in$  (%s. c) Fols (%s. c)"
by (simp add: Follows_def)

lemma mono_Follows_o:
  assumes "mono h"
  shows "f Fols g  $\subseteq$  (h o f) Fols (h o g)"
proof
  fix x
  assume "x  $\in$  f Fols g"
  with assms show "x  $\in$  (h o f) Fols (h o g)"
  by (auto simp add: Follows_def mono_Increasing_o [THEN [2] rev_subsetD]
    mono_Always_o [THEN [2] rev_subsetD]
    mono_LeadsTo_o [THEN [2] rev_subsetD, THEN INT_D])
qed

lemma mono_Follows_apply:
  "mono h ==> f Fols g  $\subseteq$  (%x. h (f x)) Fols (%x. h (g x))"
apply (drule mono_Follows_o)
apply (force simp add: o_def)

```



done

```
lemma Follows_trans:
  "[| F ∈ f Fols g; F ∈ g Fols h |] ==> F ∈ f Fols h"
apply (simp add: Follows_def)
apply (simp add: Always_eq_includes_reachable)
apply (blast intro: order_trans LeadsTo_Trans)
done
```

## 14.1 Destruction rules

```
lemma Follows_Increasing1: "F ∈ f Fols g ==> F ∈ Increasing f"
by (simp add: Follows_def)
```

```
lemma Follows_Increasing2: "F ∈ f Fols g ==> F ∈ Increasing g"
by (simp add: Follows_def)
```

```
lemma Follows_Bounded: "F ∈ f Fols g ==> F ∈ Always {s. f s ≤ g s}"
by (simp add: Follows_def)
```

```
lemma Follows_LeadsTo:
  "F ∈ f Fols g ==> F ∈ {s. k ≤ g s} LeadsTo {s. k ≤ f s}"
by (simp add: Follows_def)
```

```
lemma Follows_LeadsTo_prefixLe:
  "F ∈ f Fols g ==> F ∈ {s. k prefixLe g s} LeadsTo {s. k prefixLe f s}"
apply (rule single_LeadsTo_I, clarify)
apply (drule_tac k="g s" in Follows_LeadsTo)
apply (erule LeadsTo_weaken)
  apply blast
apply (blast intro: prefixLe_trans prefix_imp_prefixLe)
done
```

```
lemma Follows_LeadsTo_prefixGe:
  "F ∈ f Fols g ==> F ∈ {s. k prefixGe g s} LeadsTo {s. k prefixGe f s}"
apply (rule single_LeadsTo_I, clarify)
apply (drule_tac k="g s" in Follows_LeadsTo)
apply (erule LeadsTo_weaken)
  apply blast
apply (blast intro: prefixGe_trans prefix_imp_prefixGe)
done
```

```
lemma Always_Follows1:
  "[| F ∈ Always {s. f s = f' s}; F ∈ f Fols g |] ==> F ∈ f' Fols g"

apply (simp add: Follows_def Increasing_def Stable_def, auto)
apply (erule_tac [3] Always_LeadsTo_weaken)
apply (erule_tac A = "{s. x ≤ f s}" and A' = "{s. x ≤ f' s}"
  in Always_Constrains_weaken, auto)
apply (drule Always_Int_I, assumption)
apply (force intro: Always_weaken)
done
```

```

lemma Always_Follows2:
  "[| F ∈ Always {s. g s = g' s}; F ∈ f Fols g |] ==> F ∈ f Fols g'"
apply (simp add: Follows_def Increasing_def Stable_def, auto)
apply (erule_tac [3] Always_LeadsTo_weaken)
apply (erule_tac A = "{s. x ≤ g s}" and A' = "{s. x ≤ g s}"
      in Always_Constrains_weaken, auto)
apply (drule Always_Int_I, assumption)
apply (force intro: Always_weaken)
done

```

## 14.2 Union properties (with the subset ordering)

```

lemma increasing_Un:
  "[| F ∈ increasing f; F ∈ increasing g |]
   ==> F ∈ increasing (%s. (f s) ∪ (g s))"
apply (simp add: increasing_def stable_def constrains_def, auto)
apply (drule_tac x = "f xb" in spec)
apply (drule_tac x = "g xb" in spec)
apply (blast dest!: bspec)
done

```

```

lemma Increasing_Un:
  "[| F ∈ Increasing f; F ∈ Increasing g |]
   ==> F ∈ Increasing (%s. (f s) ∪ (g s))"
apply (auto simp add: Increasing_def Stable_def Constrains_def
                  stable_def constrains_def)
apply (drule_tac x = "f xb" in spec)
apply (drule_tac x = "g xb" in spec)
apply (blast dest!: bspec)
done

```

```

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}"
by (simp add: Always_eq_includes_reachable, blast)

```

```

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}"
apply (rule single_LeadsTo_I)
apply (drule_tac x = "f s" in IncreasingD)
apply (drule_tac x = "g s" in IncreasingD)
apply (rule LeadsTo_weaken)
apply (rule PSP_Stable)
apply (erule_tac x = "f s" in spec)
apply (erule Stable_Int, assumption, blast+)
done

```

```

lemma Follows_Un:
  "[| F ∈ f' Fols f; F ∈ g' Fols g |]

```

```

==> F ∈ (%s. (f' s) ∪ (g' s)) Fols (%s. (f s) ∪ (g s))"
apply (simp add: Follows_def Increasing_Un Always_Un del: Un_subset_iff sup.bounded_iff,
auto)
apply (rule LeadsTo_Trans)
apply (blast intro: Follows_Un_lemma)

apply (blast intro: Follows_Un_lemma [THEN LeadsTo_weaken])
done

```

### 14.3 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))"
apply (simp add: increasing_def stable_def constrains_def, auto)
apply (drule_tac x = "f xb" in spec)
apply (drule_tac x = "g xb" in spec)
apply (drule bspec, assumption)
apply (blast intro: add_mono order_trans)
done

lemma Increasing_union:
  "[| F ∈ Increasing f; F ∈ Increasing g |]
   ==> F ∈ Increasing (%s. (f s) + (g s :: ('a::order) multiset))"
apply (auto simp add: Increasing_def Stable_def Constrains_def
  stable_def constrains_def)
apply (drule_tac x = "f xb" in spec)
apply (drule_tac x = "g xb" in spec)
apply (drule bspec, assumption)
apply (blast intro: add_mono order_trans)
done

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)}"
apply (simp add: Always_eq_includes_reachable)
apply (blast intro: add_mono)
done

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}"
apply (rule single_LeadsTo_I)
apply (drule_tac x = "f s" in IncreasingD)
apply (drule_tac x = "g s" in IncreasingD)
apply (rule LeadsTo_weaken)
apply (rule PSP_Stable)
apply (erule_tac x = "f s" in spec)
apply (erule Stable_Int, assumption, blast)

```

```

apply (blast intro: add_mono order_trans)
done

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))"
apply (simp add: Follows_def)
apply (simp add: Increasing_union Always_union, auto)
apply (rule LeadsTo_Trans)
apply (blast intro: Follows_union_lemma)

apply (simp add: union_commute)
apply (blast intro: Follows_union_lemma)
done

lemma Follows_sum:
  "!!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)"
apply (erule rev_mp)
apply (erule finite_induct, simp)
apply (simp add: Follows_union)
done

lemma Increasing_imp_Stable_prefixGe:
  "F ∈ Increasing func ==> F ∈ Stable {s. h prefixGe (func s)}"
apply (simp add: Increasing_def Stable_def Constrains_def constrains_def)
apply (blast intro: trans_Ge [THEN trans_genPrefix, THEN transD]
  prefix_imp_prefixGe)
done

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}"
apply (rule single_LeadsTo_I)
apply (drule_tac x = "f s" in spec)
apply (erule LeadsTo_weaken)
prefer 2
apply (blast intro: trans_Ge [THEN trans_genPrefix, THEN transD]
  prefix_imp_prefixGe, blast)
done

end

```

## 15 Predicate Transformers

```
theory Transformers imports Comp begin
```

## 15.1 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<sup>-1</sup> ' ' (-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)"  
**by** (force simp add: wp\_def)

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)"  
**by** (simp add: wp\_def, blast)

**lemma** *Compl\_Domain\_subset\_wp*: "- (Domain act) ⊆ wp act B"  
**by** (force simp add: wp\_def)

**lemma** *wp\_empty* [simp]: "wp act {} = - (Domain act)"  
**by** (force simp add: wp\_def)

The identity relation is the skip action

**lemma** *wp\_Id* [simp]: "wp Id B = B"  
**by** (simp add: wp\_def)

**lemma** *wp\_totalize\_act*:  
 "wp (totalize\_act act) B = (wp act B ∩ Domain act) ∪ (B - Domain act)"  
**by** (simp add: wp\_def totalize\_act\_def, blast)

**lemma** *awp\_subset*: "(awp F A ⊆ A)"  
**by** (force simp add: awp\_def wp\_def)

**lemma** *awp\_Int\_eq*: "awp F (A ∩ B) = awp F A ∩ awp F B"  
**by** (simp add: awp\_def wp\_def, blast)

The fundamental theorem for *awp*

**theorem** *awp\_iff\_constrains*: "(A ≤ awp F B) = (F ∈ A co B)"  
**by** (simp add: awp\_def constrains\_def wp\_iff INT\_subset\_iff)

**lemma** *awp\_iff\_stable*: "(A ⊆ awp F A) = (F ∈ stable A)"  
**by** (simp add: awp\_iff\_constrains stable\_def)

**lemma** *stable\_imp\_awp\_ident*: "F ∈ stable A ==> awp F A = A"  
**apply** (rule equalityI [OF awp\_subset])  
**apply** (simp add: awp\_iff\_stable)  
**done**

```
lemma wp_mono: "(A ⊆ B) ==> wp act A ⊆ wp act B"
by (simp add: wp_def, blast)
```

```
lemma awp_mono: "(A ⊆ B) ==> awp F A ⊆ awp F B"
by (simp add: awp_def wp_def, blast)
```

```
lemma wens_unfold:
  "wens F act B = (wp act B ∩ awp F (B ∪ wens F act B)) ∪ B"
apply (simp add: wens_def)
apply (rule gfp_unfold)
apply (simp add: mono_def wp_def awp_def, blast)
done
```

```
lemma wens_Id [simp]: "wens F Id B = B"
by (simp add: wens_def gfp_def wp_def awp_def, blast)
```

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

```
lemma ensures_imp_wens: "F ∈ A ensures B ==> ∃ act ∈ Acts F. A ⊆ wens F act B"
apply (simp add: wens_def ensures_def transient_def, clarify)
apply (rule rev_bexI, assumption)
apply (rule gfp_upperbound)
apply (simp add: constrains_def awp_def wp_def, blast)
done
```

```
lemma wens_ensures: "act ∈ Acts F ==> F ∈ (wens F act B) ensures B"
by (simp add: wens_def gfp_def constrains_def awp_def wp_def
  ensures_def transient_def, blast)
```

These two results constitute assertion (4.13) of the thesis

```
lemma wens_mono: "(A ⊆ B) ==> wens F act A ⊆ wens F act B"
apply (simp add: wens_def wp_def awp_def)
apply (rule gfp_mono, blast)
done
```

```
lemma wens_weakening: "B ⊆ wens F act B"
by (simp add: wens_def gfp_def, blast)
```

Assertion (6), or 4.16 in the thesis

```
lemma subset_wens: "A-B ⊆ wp act B ∩ awp F (B ∪ A) ==> A ⊆ wens F act B"
apply (simp add: wens_def wp_def awp_def)
apply (rule gfp_upperbound, blast)
done
```

Assertion 4.17 in the thesis

```
lemma Diff_wens_constrains: "F ∈ (wens F act A - A) co wens F act A"
by (simp add: wens_def gfp_def wp_def awp_def constrains_def, blast)
```

— Proved instantly, yet remarkably fragile. If `Un_subset_iff` is declared as an iff-rule, then it's almost impossible to prove. One proof is via `meson` after expanding all definitions, but it's slow!

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

```
lemma stable_wens: "F ∈ stable A ==> F ∈ stable (wens F act A)"
apply (simp add: stable_def)
apply (drule constrains_Un [OF Diff_wens_constrains [of F act A]])
apply (simp add: Un_Int_distrib2 Compl_partition2)
apply (erule constrains_weaken, blast)
apply (simp add: wens_weakening)
done
```

Assertion 4.20 in the thesis.

```
lemma wens_Int_eq_lemma:
  "[|T-B ⊆ awp F T; act ∈ Acts F|]
   ==> T ∩ wens F act B ⊆ wens F act (T∩B)"
apply (rule subset_wens)
apply (rule_tac P="λx. f x ⊆ b" for f b in ssubst [OF wens_unfold])
apply (simp add: wp_def awp_def, blast)
done
```

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

```
lemma wens_Int_eq:
  "[|T-B ⊆ awp F T; act ∈ Acts F|]
   ==> T ∩ wens F act B = T ∩ wens F act (T∩B)"
apply (rule equalityI)
  apply (simp_all add: Int_lower1)
  apply (rule wens_Int_eq_lemma, assumption+)
apply (rule subset_trans [OF _ wens_mono [of "T∩B" B]], auto)
done
```

## 15.2 Defining the Weakest Ensures Set

inductive\_set

```
wens_set :: "[’a program, ’a set] => ’a set set"
for F :: "’a program" and B :: "’a set"
```

where

```
Basis: "B ∈ wens_set F B"
```

```
| Wens: "[|X ∈ wens_set F B; act ∈ Acts F|] ==> wens F act X ∈ wens_set F B"
```

```
| Union: "W ≠ {} ==> ∀U ∈ W. U ∈ wens_set F B ==> ⋃W ∈ wens_set F B"
```

```
lemma wens_set_imp_co: "A ∈ wens_set F B ==> F ∈ (A-B) co A"
apply (erule wens_set.induct)
  apply (simp add: constrains_def)
  apply (drule_tac act1=act and A1=X
    in constrains_Un [OF Diff_wens_constrains])
  apply (erule constrains_weaken, blast)
  apply (simp add: wens_weakening)
  apply (rule constrains_weaken)
  apply (rule_tac I=W and A="λv. v-B" and A'="λv. v" in constrains_UN, blast+)
done
```

```

lemma wens_set_imp_leadsTo: "A ∈ wens_set F B ==> F ∈ A leadsTo B"
apply (erule wens_set.induct)
  apply (rule leadsTo_refl)
  apply (blast intro: wens_ensures leadsTo_Trans)
apply (blast intro: leadsTo_Union)
done

lemma leadsTo_imp_wens_set: "F ∈ A leadsTo B ==> ∃ C ∈ wens_set F B. A ⊆ C"
apply (erule leadsTo_induct_pre)
  apply (blast dest!: ensures_imp_wens intro: wens_set.Basis wens_set.Wens)

  apply (clarify, drule ensures_weaken_R, assumption)
  apply (blast dest!: ensures_imp_wens intro: wens_set.Wens)
apply (case_tac "S={}")
  apply (simp, blast intro: wens_set.Basis)
apply (clarsimp dest!: bchoice simp: ball_conj_distrib Bex_def)
apply (rule_tac x = "⋃ {Z. ∃ U ∈ S. Z = f U}" in exI)
apply (blast intro: wens_set.Union)
done

```

Assertion (9): 4.27 in the thesis.

```

lemma leadsTo_iff_wens_set: "(F ∈ A leadsTo B) = (∃ C ∈ wens_set F B. A ⊆ C)"
by (blast intro: leadsTo_imp_wens_set leadsTo_weaken_L wens_set_imp_leadsTo)

```

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 ∈ wens_set F B ==> B ⊆ A"
apply (erule wens_set.induct)
  apply (blast intro: wens_weakening [THEN subsetD])
done

```

### 15.3 Properties Involving Program Union

Assertion (4.30) of thesis, reoriented

```

lemma awp_Join_eq: "awp (F⊔G) B = awp F B ∩ awp G B"
by (simp add: awp_def wp_def, blast)

```

```

lemma wens_subset: "wens F act B - B ⊆ wp act B ∩ awp F (B ∪ wens F act B)"
by (subst wens_unfold, fast)

```

Assertion (4.31)

```

lemma subset_wens_Join:
  "[/A = T ∩ wens F act B; T-B ⊆ awp F T; A-B ⊆ awp G (A ∪ B)]
  ==> A ⊆ wens (F⊔G) act B"
apply (subgoal_tac "(T ∩ wens F act B) - B ⊆
  wp act B ∩ awp F (B ∪ wens F act B) ∩ awp F T")
  apply (rule subset_wens)

```



```

apply (simp add: awp_Join_eq awp_Int_eq Un_commute)
apply (simp add: awp_def wp_def, blast)
apply (insert wens_subset [of F act B], blast)
done

```

Assertion (4.32)

```

lemma wens_Join_subset: "wens (F⊔G) act B ⊆ wens F act B"
apply (simp add: wens_def)
apply (rule gfp_mono)
apply (auto simp add: awp_Join_eq)
done

```

Lemma, because the inductive step is just too messy.

```

lemma wens_Union_inductive_step:
  assumes awpF: "T-B ⊆ awp F T"
    and awpG: "!!X. X ∈ wens_set F B ==> (T∩X) - B ⊆ awp G (T∩X)"
  shows "[|X ∈ wens_set F B; act ∈ Acts F; Y ⊆ X; T∩X = T∩Y|]
    ==> wens (F⊔G) act Y ⊆ wens F act X ∧
      T ∩ wens F act X = T ∩ wens (F⊔G) act Y"
apply (subgoal_tac "wens (F⊔G) act Y ⊆ wens F act X")
prefer 2
  apply (blast dest: wens_mono intro: wens_Join_subset [THEN subsetD], simp)
apply (rule equalityI)
prefer 2 apply blast
apply (simp add: Int_lower1)
apply (frule wens_set_imp_subset)
apply (subgoal_tac "T-X ⊆ awp F T")
  prefer 2 apply (blast intro: awpF [THEN subsetD])
apply (rule_tac B = "wens (F⊔G) act (T∩X)" in subset_trans)
  prefer 2 apply (blast intro!: wens_mono)
apply (subst wens_Int_eq, assumption+)
apply (rule subset_wens_Join [of _ T], simp, blast)
apply (subgoal_tac "T ∩ wens F act (T∩X) ∪ T∩X = T ∩ wens F act X")
  prefer 2
  apply (subst wens_Int_eq [symmetric], assumption+)
  apply (blast intro: wens_weakening [THEN subsetD], simp)
apply (blast intro: awpG [THEN subsetD] wens_set.Wens)
done

```

```

theorem wens_Union:
  assumes awpF: "T-B ⊆ awp F T"
    and awpG: "!!X. X ∈ wens_set F B ==> (T∩X) - B ⊆ awp G (T∩X)"
    and major: "X ∈ wens_set F B"
  shows "∃Y ∈ wens_set (F⊔G) B. Y ⊆ X & T∩X = T∩Y"
apply (rule wens_set.induct [OF major])

```

Basis: trivial

```

  apply (blast intro: wens_set.Basis)

```

Inductive step

```

  apply clarify
  apply (rule_tac x = "wens (F⊔G) act Y" in rev_bexI)
  apply (force intro: wens_set.Wens)

```

```

    apply (simp add: wens_Union_inductive_step [OF awpF awpG])

Union: by Axiom of Choice

apply (simp add: ball_conj_distrib Bex_def)
apply (clarify dest!: bchoice)
apply (rule_tac x = " $\bigcup \{Z. \exists U \in W. Z = f U\}$ " in exI)
apply (blast intro: wens_set.Union)
done

theorem leadsTo_Join:
  assumes leadsTo: " $F \in A$  leadsTo  $B$ "
  and awpF: " $T \cdot B \subseteq \text{awp } F \ T$ "
  and awpG: " $\neg \exists X. X \in \text{wens\_set } F \ B \implies (T \cap X) \cdot B \subseteq \text{awp } G \ (T \cap X)$ "
  shows " $F \sqcup G \in T \cap A$  leadsTo  $B$ "
apply (rule leadsTo [THEN leadsTo_imp_wens_set, THEN bexE])
apply (rule wens_Union [THEN bexE])
  apply (rule awpF)
  apply (erule awpG, assumption)
apply (blast intro: wens_set_imp_leadsTo [THEN leadsTo_weaken_L])
done

```

## 15.4 The Set $\text{wens\_set } F \ B$ for a Single-Assignment Program

Thesis Section 4.3.3

We start by proving laws about single-assignment programs

```

lemma awp_single_eq [simp]:
  " $\text{awp } (\text{mk\_program } (\text{init}, \{\text{act}\}, \text{allowed})) \ B = B \cap \text{wp act } B$ "
by (force simp add: awp_def wp_def)

lemma wp_Un_subset: " $\text{wp act } A \cup \text{wp act } B \subseteq \text{wp act } (A \cup B)$ "
by (force simp add: wp_def)

lemma wp_Un_eq: " $\text{single\_valued act} \implies \text{wp act } (A \cup B) = \text{wp act } A \cup \text{wp act } B$ "
apply (rule equalityI)
  apply (force simp add: wp_def single_valued_def)
apply (rule wp_Un_subset)
done

lemma wp_UN_subset: " $(\bigcup_{i \in I}. \text{wp act } (A \ i)) \subseteq \text{wp act } (\bigcup_{i \in I}. A \ i)$ "
by (force simp add: wp_def)

lemma wp_UN_eq:
  "[/single_valued act;  $I \neq \{\}$ ]/
   $\implies \text{wp act } (\bigcup_{i \in I}. A \ i) = (\bigcup_{i \in I}. \text{wp act } (A \ i))$ "
apply (rule equalityI)
  prefer 2 apply (rule wp_UN_subset)
  apply (simp add: wp_def Image_INT_eq)
done

lemma wens_single_eq:
  " $\text{wens } (\text{mk\_program } (\text{init}, \{\text{act}\}, \text{allowed})) \ \text{act } B = B \cup \text{wp act } B$ "

```

by (simp add: wens\_def gfp\_def wp\_def, blast)

Next, we express the  $wens\_set$  for single-assignment programs

**definition**  $wens\_single\_finite$  ::  $[(\text{'a} \times \text{'a}) \text{ set}, \text{'a set}, \text{nat}] \Rightarrow \text{'a set}$  where

$wens\_single\_finite \text{ act } B \text{ k} == \bigcup i \in \text{atMost } k. (\text{wp act } \hat{\sim} i) B$

**definition**  $wens\_single$  ::  $[(\text{'a} \times \text{'a}) \text{ set}, \text{'a set}] \Rightarrow \text{'a set}$  where

$wens\_single \text{ act } B == \bigcup i. (\text{wp act } \hat{\sim} i) B$

**lemma**  $wens\_single\_Un\_eq$ :

"single\_valued act

$\Rightarrow wens\_single \text{ act } B \cup \text{wp act } (wens\_single \text{ act } B) = wens\_single \text{ act }$

$B$ "

**apply** (rule equalityI)

**apply** (simp\_all add: Un\_upper1)

**apply** (simp add: wens\_single\_def wp\_UN\_eq, clarify)

**apply** (rule\_tac a="Suc xa" in UN\_I, auto)

**done**

**lemma**  $\text{atMost\_nat\_nonempty}$ :  $\text{atMost } (k::\text{nat}) \neq \{\}$

**by** force

**lemma**  $wens\_single\_finite\_0$  [simp]:  $wens\_single\_finite \text{ act } B \text{ } 0 = B$

**by** (simp add: wens\_single\_finite\_def)

**lemma**  $wens\_single\_finite\_Suc$ :

"single\_valued act

$\Rightarrow wens\_single\_finite \text{ act } B (\text{Suc } k) =$

$wens\_single\_finite \text{ act } B \text{ k} \cup \text{wp act } (wens\_single\_finite \text{ act } B \text{ k})$

**apply** (simp add: wens\_single\_finite\_def wp\_UN\_eq [OF \_ atMost\_nat\_nonempty])

**apply** (force elim!: le\_SucE)

**done**

**lemma**  $wens\_single\_finite\_Suc\_eq\_wens$ :

"single\_valued act

$\Rightarrow wens\_single\_finite \text{ act } B (\text{Suc } k) =$

$wens (\text{mk\_program } (\text{init}, \{\text{act}\}, \text{allowed})) \text{ act}$

$(wens\_single\_finite \text{ act } B \text{ k})$ "

**by** (simp add: wens\_single\_finite\_Suc wens\_single\_eq)

**lemma**  $\text{def\_wens\_single\_finite\_Suc\_eq\_wens}$ :

"[ $F = \text{mk\_program } (\text{init}, \{\text{act}\}, \text{allowed})$ ; single\_valued act]

$\Rightarrow wens\_single\_finite \text{ act } B (\text{Suc } k) =$

$wens F \text{ act } (wens\_single\_finite \text{ act } B \text{ k})$ "

**by** (simp add: wens\_single\_finite\_Suc\_eq\_wens)

**lemma**  $wens\_single\_finite\_Un\_eq$ :

"single\_valued act

$\Rightarrow wens\_single\_finite \text{ act } B \text{ k} \cup \text{wp act } (wens\_single\_finite \text{ act } B \text{ k})$

$\in \text{range } (wens\_single\_finite \text{ act } B)$ "

**by** (simp add: wens\_single\_finite\_Suc [symmetric])

**lemma**  $wens\_single\_eq\_Union$ :

```

      "wens_single act B =  $\bigcup$  (range (wens_single_finite act B))"
by (simp add: wens_single_finite_def wens_single_def, blast)

lemma wens_single_finite_eq_Union:
  "wens_single_finite act B n = ( $\bigcup_{k \in \text{atMost } n} \text{wens\_single\_finite act B } k$ )"
apply (auto simp add: wens_single_finite_def)
apply (blast intro: le_trans)
done

lemma wens_single_finite_mono:
  " $m \leq n \implies \text{wens\_single\_finite act B } m \subseteq \text{wens\_single\_finite act B } n$ "
by (force simp add: wens_single_finite_eq_Union [of act B n])

lemma wens_single_finite_subset_wens_single:
  "wens_single_finite act B k  $\subseteq$  wens_single act B"
by (simp add: wens_single_eq_Union, blast)

lemma subset_wens_single_finite:
  "[ $|W \subseteq \text{wens\_single\_finite act B } k$ ; single_valued act;  $W \neq \{\}$ ]|
   $\implies \exists m. \bigcup W = \text{wens\_single\_finite act B } m$ "
apply (induct k)
  apply (rule_tac x=0 in exI, simp, blast)
  apply (auto simp add: atMost_Suc)
  apply (case_tac "wens_single_finite act B (Suc k)  $\in W$ ")
    prefer 2 apply blast
  apply (drule_tac x="Suc k" in spec)
  apply (erule notE, rule equalityI)
    prefer 2 apply blast
  apply (subst wens_single_finite_eq_Union)
  apply (simp add: atMost_Suc, blast)
done

lemma for Union case

lemma Union_eq_wens_single:
  "[ $\forall k. \neg W \subseteq \text{wens\_single\_finite act B } k$ ;
    $W \subseteq \text{insert (wens\_single act B) (range (wens\_single\_finite act B))}$ ]
   $\implies \bigcup W = \text{wens\_single act B}$ "
apply (cases "wens_single act B  $\in W$ ")
  apply (blast dest: wens_single_finite_subset_wens_single [THEN subsetD])

apply (simp add: wens_single_eq_Union)
apply (rule equalityI, blast)
apply (simp add: UN_subset_iff, clarify)
apply (subgoal_tac " $\exists y \in W. \exists n. y = \text{wens\_single\_finite act B } n \ \& \ i \leq n$ ")
  apply (blast intro: wens_single_finite_mono [THEN subsetD])
  apply (drule_tac x=i in spec)
  apply (force simp add: atMost_def)
done

lemma wens_set_subset_single:
  "single_valued act
   $\implies \text{wens\_set (mk\_program (init, \{act\}, allowed)) B} \subseteq$ "

```

```

      insert (wens_single act B) (range (wens_single_finite act B))"
apply (rule subsetI)
apply (erule wens_set.induct)

```

Basis

```

  apply (fastforce simp add: wens_single_finite_def)

```

Wens inductive step

```

  apply (case_tac "acta = Id", simp)
  apply (simp add: wens_single_eq)
  apply (elim disjE)
  apply (simp add: wens_single_Un_eq)
  apply (force simp add: wens_single_finite_Un_eq)

```

Union inductive step

```

apply (case_tac "∃k. W ⊆ wens_single_finite act B ' (atMost k)")
  apply (blast dest!: subset_wens_single_finite, simp)
  apply (rule disjI1 [OF Union_eq_wens_single], blast+)
done

```

```

lemma wens_single_finite_in_wens_set:
  "single_valued act ⇒
    wens_single_finite act B k
    ∈ wens_set (mk_program (init, {act}, allowed)) B"
apply (induct_tac k)
  apply (simp add: wens_single_finite_def wens_set.Basis)
  apply (simp add: wens_set.Wens
    wens_single_finite_Suc_eq_wens [of act B _ init allowed])

```

done

```

lemma single_subset_wens_set:
  "single_valued act
  ==> insert (wens_single act B) (range (wens_single_finite act B)) ⊆

    wens_set (mk_program (init, {act}, allowed)) B"
apply (simp add: image_def wens_single_eq_Union)
apply (blast intro: wens_set.Union wens_single_finite_in_wens_set)
done

```

Theorem (4.29)

```

theorem wens_set_single_eq:
  "[|F = mk_program (init, {act}, allowed); single_valued act|]
  ==> wens_set F B =
    insert (wens_single act B) (range (wens_single_finite act B))"
apply (rule equalityI)
  apply (simp add: wens_set_subset_single)
  apply (erule ssubst, erule single_subset_wens_set)
done

```

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/] ==> F  $\sqcup$  G  $\in$  T  $\cap$  A leadsTo B"
    apply (rule leadsTo_Join, assumption, blast)
    apply (simp add: FP_def awp_iff_constrains stable_def constrains_def, blast)

  done

end

```

## 16 Progress Sets

theory ProgressSets imports Transformers begin

### 16.1 Complete Lattices and the Operator $c1$

```

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 c1 :: "[’a set set, ’a set] => ’a set" where
  — short for “closure”
  "c1 L r ==  $\bigcap \{x. x \in L \ \& \ r \subseteq x\}$ "

```

```

lemma UNIV_in_lattice: "lattice L ==> UNIV  $\in$  L"
by (force simp add: lattice_def)

```

```

lemma empty_in_lattice: "lattice L ==> {}  $\in$  L"
by (force simp add: lattice_def)

```

```

lemma Union_in_lattice: "[/M  $\subseteq$  L; lattice L/] ==>  $\bigcup M \in L$ "
by (simp add: lattice_def)

```

```

lemma Inter_in_lattice: "[/M  $\subseteq$  L; lattice L/] ==>  $\bigcap M \in L$ "
by (simp add: lattice_def)

```

```

lemma UN_in_lattice:
  "[/lattice L; !!i. i  $\in$  I ==> r i  $\in$  L/] ==> ( $\bigcup i \in I. r i$ )  $\in$  L"
apply (blast intro: Union_in_lattice)
done

```

```

lemma INT_in_lattice:
  "[/lattice L; !!i. i  $\in$  I ==> r i  $\in$  L/] ==> ( $\bigcap i \in I. r i$ )  $\in$  L"
apply (blast intro: Inter_in_lattice)
done

```

```

lemma Un_in_lattice: "[/x  $\in$  L; y  $\in$  L; lattice L/] ==> x  $\cup$  y  $\in$  L"
using Union_in_lattice [of "{x, y}" L] by simp

```

```

lemma Int_in_lattice: "[/x  $\in$  L; y  $\in$  L; lattice L/] ==> x  $\cap$  y  $\in$  L"
using Inter_in_lattice [of "{x, y}" L] by simp

```

```

lemma lattice_stable: "lattice {X. F  $\in$  stable X}"

```

by (simp add: lattice\_def stable\_def constrains\_def, blast)

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

lemma  $cl\_in\_lattice$ : " $lattice\ L \implies cl\ L\ r \in L$ "  
by (simp add: lattice\_def  $cl\_def$ )

lemma  $cl\_least$ : " $[c \in L; r \subseteq c] \implies cl\ L\ r \subseteq c$ "  
by (force simp add:  $cl\_def$ )

The next three lemmas constitute assertion (4.61)

lemma  $cl\_mono$ : " $r \subseteq r' \implies cl\ L\ r \subseteq cl\ L\ r'$ "  
by (simp add:  $cl\_def$ , blast)

lemma  $subset\_cl$ : " $r \subseteq cl\ L\ r$ "  
by (simp add:  $cl\_def\ le\_Inf\_iff$ )

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

lemma  $clI$ : " $x \in r \implies x \in cl\ L\ r$ "  
by (simp add:  $cl\_def$ , blast)

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

lemma  $clD$ : " $[c \in cl\ L\ r; B \in L; r \subseteq B] \implies c \in B$ "  
by (force simp add:  $cl\_def$ )

lemma  $cl\_UN\_subset$ : " $(\bigcup_{i \in I}. cl\ L\ (r\ i)) \subseteq cl\ L\ (\bigcup_{i \in I}. r\ i)$ "  
by (simp add:  $cl\_def$ , blast)

lemma  $cl\_Un$ : " $lattice\ L \implies cl\ L\ (r \cup s) = cl\ L\ r \cup cl\ L\ s$ "  
apply (rule equalityI)  
prefer 2  
apply (simp add:  $cl\_def$ , blast)  
apply (rule  $cl\_least$ )  
apply (blast intro:  $Un\_in\_lattice\ cl\_in\_lattice$ )  
apply (blast intro:  $subset\_cl\ [THEN\ subsetD]$ )  
done

lemma  $cl\_UN$ : " $lattice\ L \implies cl\ L\ (\bigcup_{i \in I}. r\ i) = (\bigcup_{i \in I}. cl\ L\ (r\ i))$ "  
apply (rule equalityI)  
prefer 2 apply (simp add:  $cl\_def$ , blast)  
apply (rule  $cl\_least$ )  
apply (blast intro:  $UN\_in\_lattice\ cl\_in\_lattice$ )  
apply (blast intro:  $subset\_cl\ [THEN\ subsetD]$ )  
done

lemma  $cl\_Int\_subset$ : " $cl\ L\ (r \cap s) \subseteq cl\ L\ r \cap cl\ L\ s$ "  
by (simp add:  $cl\_def$ , blast)

lemma  $cl\_idem\ [simp]$ : " $cl\ L\ (cl\ L\ r) = cl\ L\ r$ "  
by (simp add:  $cl\_def$ , blast)

lemma  $cl\_ident$ : " $r \in L \implies cl\ L\ r = r$ "  
by (force simp add:  $cl\_def$ )

```
lemma cl_empty [simp]: "lattice L ==> cl L {} = {}"
by (simp add: cl_ident empty_in_lattice)
```

```
lemma cl_UNIV [simp]: "lattice L ==> cl L UNIV = UNIV"
by (simp add: cl_ident UNIV_in_lattice)
```

Assertion (4.62)

```
lemma cl_ident_iff: "lattice L ==> (cl L r = r) = (r ∈ L)"
apply (rule iffI)
  apply (erule subst)
  apply (erule cl_in_lattice)
apply (erule cl_ident)
done
```

```
lemma cl_subset_in_lattice: "[cl L r ⊆ r; lattice L] ==> r ∈ L"
by (simp add: cl_ident_iff [symmetric] equalityI subset_cl)
```

## 16.2 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 == ∀M. ∀act ∈ Acts F. B ⊆ M & T ∩ M ∈ L -->
    T ∩ (B ∪ wp act M) ∈ L"
```

```
definition progress_set :: "[’a program, ’a set, ’a set] => ’a set set set"
where
  "progress_set F T B ==
    {L. lattice L & B ∈ L & T ∈ L & closed F T B L}"
```

```
lemma closedD:
  "[closed F T B L; act ∈ Acts F; B ⊆ M; T ∩ M ∈ L]
  ==> T ∩ (B ∪ wp act M) ∈ L"
by (simp add: closed_def)
```

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  $act \in Acts F$ .

```
lemma lattice_await_lemma:
  assumes TXC: "T ∩ X ∈ C" — induction hypothesis in theorem below
    and BsubX: "B ⊆ X" — holds in inductive step
    and latt: "lattice C"
    and TC: "T ∈ C"
    and BC: "B ∈ C"
    and clos: "closed F T B C"
  shows "T ∩ (B ∪ await F (X ∪ cl C (T ∩ r))) ∈ C"
apply (simp del: INT_simps add: await_def INT_extend_simps)
apply (rule INT_in_lattice [OF latt])
apply (erule closedD [OF clos])
apply (simp add: subset_trans [OF BsubX Un_upper1])
apply (subgoal_tac "T ∩ (X ∪ cl C (T ∩ r)) = (T ∩ X) ∪ cl C (T ∩ r)")
```



```

prefer 2 apply (blast intro: TC clD)
apply (erule ssubst)
apply (blast intro: Un_in_lattice latt cl_in_lattice TXC)
done

```

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"
  apply (subgoal_tac "T ∩ (B ∪ wp act X) ∈ C")
  prefer 2 apply (simp add: closedD [OF clos] act BsubX TXC)
  apply (drule Int_in_lattice
    [OF _ lattice_awn_lemma [OF TXC BsubX latt TC BC clos, of r]
    latt])
  apply (subgoal_tac
    "T ∩ (B ∪ wp act X) ∩ (T ∩ (B ∪ awp F (X ∪ cl C (T ∩ r)))) =
    T ∩ (B ∪ wp act X ∩ awp F (X ∪ cl C (T ∩ r)))")
  prefer 2 apply blast
  apply simp
  apply (drule Un_in_lattice [OF _ TXC latt])
  apply (subgoal_tac
    "T ∩ (B ∪ wp act X ∩ awp F (X ∪ cl C (T ∩ r))) ∪ T ∩ X =
    T ∩ (wp act X ∩ awp F (X ∪ cl C (T ∩ r))) ∪ X")
  apply simp
  apply (blast intro: BsubX [THEN subsetD])
done

```

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 -
    from Xwens have BsubX: "B ⊆ X"
    by (rule wens_set_imp_subset)
    let ?r = "wens F act X"
    have "?r ⊆ (wp act X ∩ awp F (X ∪ ?r)) ∪ X"
    by (simp add: wens_unfold [symmetric])
    then have "T ∩ ?r ⊆ T ∩ ((wp act X ∩ awp F (X ∪ ?r)) ∪ X)"
    by blast
    then have "T ∩ ?r ⊆ T ∩ ((wp act X ∩ awp F (T ∩ (X ∪ ?r))) ∪ X)"
    by (simp add: awp_Int_eq Fstable stable_imp_awn_ident, blast)
  end

```

```

then have "T∩?r ⊆ T ∩ ((wp act X ∩ awp F (X ∪ cl C (T∩?r))) ∪ X)"
  by (blast intro: awp_mono [THEN [2] rev_subsetD] subset_cl [THEN subsetD])
then have "cl C (T∩?r) ⊆
  cl C (T ∩ ((wp act X ∩ awp F (X ∪ cl C (T∩?r))) ∪ X))"
  by (rule cl_mono)
then have "cl C (T∩?r) ⊆
  T ∩ ((wp act X ∩ awp F (X ∪ cl C (T∩?r))) ∪ X)"
  by (simp add: cl_ident lattice_lemma [OF TXC BsubX act latt TC BC clos])
then have "cl C (T∩?r) ⊆ (wp act X ∩ awp F (X ∪ cl C (T∩?r))) ∪ X"
  by blast
then have "cl C (T∩?r) ⊆ ?r"
  by (blast intro!: subset_wens)
then have cl_subset: "cl C (T∩?r) ⊆ T∩?r"
  by (simp add: cl_ident TC
    subset_trans [OF cl_mono [OF Int_lower1]])
show ?thesis
  by (rule cl_subset_in_lattice [OF cl_subset latt])
qed

```

Proved on page 96 of Meier's thesis. The special case when  $T = UNIV$  states that every progress set for the program  $F$  and set  $B$  includes the set  $wens\_set F B$ .

```

lemma progress_set_lemma:
  "[|C ∈ progress_set F T B; r ∈ wens_set F B; F ∈ stable T|] ==> T∩r
  ∈ C"
apply (simp add: progress_set_def, clarify)
apply (erule wens_set.induct)

```

Base

```

  apply (simp add: Int_in_lattice)

```

The difficult *wens* case

```

  apply (simp add: progress_induction_step)

```

Disjunctive case

```

apply (subgoal_tac "(⋃ U ∈ W. T ∩ U) ∈ C")
apply simp
apply (blast intro: UN_in_lattice)
done

```

### 16.3 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 -
  from TBwp have "(T∩B) ∪ (T ∩ wp act M) ∈ C"
    by (simp add: Int_Un_distrib)
  then have TBBC: "(T∩B') ∪ ((T∩B) ∪ (T ∩ wp act M)) ∈ C"
    by (blast intro: Int_in_lattice Un_in_lattice TC B'C latt)

```

```

show ?thesis
  by (rule eqelem_imp_iff [THEN iffD1, OF _ TBBC],
      blast intro: BB' [THEN subsetD])
qed

lemma progress_set_mono:
  assumes BB': "B  $\subseteq$  B'"
  shows
    "[| B'  $\in$  C; C  $\in$  progress_set F T B |]
     ==> C  $\in$  progress_set F T B'"
  by (simp add: progress_set_def closed_def closed_mono [OF BB']
      subset_trans [OF BB'])

theorem progress_set_Union:
  assumes leadsTo: "F  $\in$  A leadsTo B'"
  and prog: "C  $\in$  progress_set F T B"
  and Fstable: "F  $\in$  stable T"
  and BB': "B  $\subseteq$  B'"
  and B'C: "B'  $\in$  C"
  and Gco: "!!X. X  $\in$  C ==> G  $\in$  X-B co X"
  shows "F  $\sqcup$  G  $\in$  T  $\cap$  A leadsTo B'"
  apply (insert prog Fstable)
  apply (rule leadsTo_Join [OF leadsTo])
  apply (force simp add: progress_set_def awp_iff_stable [symmetric])
  apply (simp add: awp_iff_constrains)
  apply (drule progress_set_mono [OF BB' B'C])
  apply (blast intro: progress_set_lemma Gco constrains_weaken_L
      BB' [THEN subsetD])
done

```

## 16.4 Some Progress Sets

```

lemma UNIV_in_progress_set: "UNIV  $\in$  progress_set F T B"
  by (simp add: progress_set_def lattice_def closed_def)

```

### 16.4.1 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  $\in$  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.  $\forall s t. s \in X \ \& \ (s,t) \in r \rightarrow t \in X}$ "
```

```

lemma relcl_refl: "(a,a)  $\in$  relcl L"

```

```

  by (simp add: relcl_def subset_cl [THEN subsetD])

```

```

lemma relcl_trans:

```

```

  "[| (a,b)  $\in$  relcl L; (b,c)  $\in$  relcl L; lattice L |] ==> (a,c)  $\in$  relcl
  L"
```

```

apply (simp add: relcl_def)
apply (blast intro: clD cl_in_lattice)
done

lemma refl_relcl: "lattice L ==> refl (relcl L)"
by (simp add: refl_onI relcl_def subset_cl [THEN subsetD])

lemma trans_relcl: "lattice L ==> trans (relcl L)"
by (blast intro: relcl_trans transI)

lemma lattice_latticeof: "lattice (latticeof r)"
by (auto simp add: lattice_def latticeof_def)

lemma lattice_singletonI:
  "[/lattice L; !!s. s ∈ X ==> {s} ∈ L/] ==> X ∈ L"
apply (cut_tac UN_singleton [of X])
apply (erule subst)
apply (simp only: UN_in_lattice)
done

```

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}"
apply (rule equalityI)
apply (rule cl_least)
  apply (simp (no_asm_use) add: latticeof_def trans_def, blast)
  apply (simp add: latticeof_def refl_on_def, blast)
apply (simp add: latticeof_def, clarify)
apply (unfold cl_def, blast)
done

```

Related to (4.71).

```

lemma cl_eq_Collect_relcl:
  "lattice L ==> cl L X = {t. ∃s. s ∈ X & (s,t) ∈ relcl L}"
apply (cut_tac UN_singleton [of X])
apply (erule subst)
apply (force simp only: relcl_def cl_UN)
done

```

Meier's theorem of section 4.5.3

```

theorem latticeof_relcl_eq: "lattice L ==> latticeof (relcl L) = L"
apply (rule equalityI)
  prefer 2 apply (force simp add: latticeof_def relcl_def cl_def, clarify)

apply (rename_tac X)
apply (rule cl_subset_in_lattice)
  prefer 2 apply assumption
apply (drule cl_ident_iff [OF lattice_latticeof, THEN iffD2])
apply (drule equalityD1)
apply (rule subset_trans)
  prefer 2 apply assumption
apply (thin_tac "_ ⊆ X")

```

```

apply (cut_tac A=X in UN_singleton)
apply (erule subst)
apply (simp only: cl_UN lattice_latticeof
                 cl_latticeof [OF refl_relcl trans_relcl])
apply (simp add: relcl_def)
done

theorem relcl_latticeof_eq:
  "[|refl r; trans r|] ==> relcl (latticeof r) = r"
by (simp add: relcl_def cl_latticeof)

```

### 16.4.2 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 ∈ stable A ==> F ∈ A-B co A"
by (simp add: stable_def constrains_def, blast)

```

```

theorem decoupling:
  assumes leadsTo: "F ∈ A leadsTo B"
  and Gstable: "G ∈ stable B"
  and dec:      "decoupled F G"
  shows "F⊔G ∈ A leadsTo B"
proof -
  have prog: "{X. G ∈ stable X} ∈ progress_set F UNIV B"
  by (simp add: progress_set_def lattice_stable Gstable closed_def
              stable_Un [OF Gstable] dec [unfolded decoupled_def])
  have "F⊔G ∈ (UNIV∩A) leadsTo B"
  by (rule progress_set_Union [OF leadsTo prog],
      simp_all add: Gstable stableco)
  thus ?thesis by simp
qed

```

Rao’s Weak Decoupling Theorem

```

theorem weak_decoupling:
  assumes leadsTo: "F ∈ A leadsTo B"
  and stable: "F⊔G ∈ stable B"
  and dec:      "decoupled F (F⊔G)"
  shows "F⊔G ∈ A leadsTo B"
proof -
  have prog: "{X. F⊔G ∈ stable X} ∈ progress_set F UNIV B"
  by (simp del: Join_stable
      add: progress_set_def lattice_stable stable closed_def
          stable_Un [OF stable] dec [unfolded decoupled_def])
  have "F⊔G ∈ (UNIV∩A) leadsTo B"
  by (rule progress_set_Union [OF leadsTo prog],
      simp_all del: Join_stable add: stable,
      simp add: stableco)
  thus ?thesis by simp
qed

```

The “Decoupling via  $G'$  Union Theorem”

```

theorem decoupling_via_aux:
  assumes leadsTo: "F ∈ A leadsTo B"
    and prog: "{X. G' ∈ stable X} ∈ progress_set F UNIV B"
    and GG': "G ≤ G'"
    — Beware! This is the converse of the refinement relation!
  shows "F ⊔ G ∈ A leadsTo B"
proof -
  from prog have stable: "G' ∈ stable B"
  by (simp add: progress_set_def)
  have "F ⊔ G ∈ (UNIV ∩ A) leadsTo B"
  by (rule progress_set_Union [OF leadsTo prog],
      simp_all add: stable stableco component_stable [OF GG'])
  thus ?thesis by simp
qed

```

## 16.5 Composition Theorems Based on Monotonicity and Commutativity

### 16.5.1 Commutativity of $cl\ L$ 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"
apply (simp add: closed_def, clarify)
apply (rule ProgressSets.cl_subset_in_lattice [OF _ lattice])
apply (simp add: Int_Un_distrib cl_Un [OF lattice]
      cl_ident Int_in_lattice [OF TL BL lattice] Un_upper1)
apply (subgoal_tac "cl L (T ∩ wp act M) ⊆ T ∩ (B ∪ wp act (cl L (T ∩ M)))")

  prefer 2
  apply (cut_tac commutes, simp add: commutes_def)
apply (erule subset_trans)
apply (simp add: cl_ident)
apply (blast intro: rev_subsetD [OF _ wp_mono])
done

```

Version packaged with  $\llbracket ?F \in ?A \mapsto ?B'; ?C \in \text{progress\_set } ?F ?T ?B; ?F \in \text{UNITY.stable } ?T; ?B \subseteq ?B'; ?B' \in ?C; \bigwedge X. X \in ?C \implies ?G \in X - ?B \text{ co } X \rrbracket \implies ?F \sqcup ?G \in ?T \cap ?A \mapsto ?B'$

```

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"
by (rule progress_set_Union [OF leadsTo _ Fstable subset_refl BL Gco],
    simp add: progress_set_def commutativity1_lemma commutes lattice BL TL)

```

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"
by (simp add: funof_def single_valued_def, blast)

```

```

lemma funof_Pair_in:
  "[|single_valued r; x ∈ Domain r|] ==> (x, funof r x) ∈ r"
by (force simp add: funof_eq)

```

```

lemma funof_in:
  "[|r' '{x} ⊆ A; single_valued r; x ∈ Domain r|] ==> funof r x ∈ A"
by (force simp add: funof_eq)

```

```

lemma funof_imp_wp: "[|funof act t ∈ A; single_valued act|] ==> t ∈ wp act A"
by (force simp add: in_wp_iff funof_eq)

```

### 16.5.2 Commutativity of Functions and Relation

Thesis, page 109

From Meier's thesis, section 4.5.6

```

lemma commutativity2_lemma:
  assumes dcommutes:
    "⋀act s t. act ∈ Acts F ==> s ∈ 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 -
  { fix M and act and t
    assume 1: "B ⊆ M" "act ∈ Acts F" "t ∈ cl L (T ∩ wp act M)"
    then have "∃s. (s,t) ∈ relcl L ∧ s ∈ T ∩ wp act M"
      by (force simp add: cl_eq_Collect_relcl [OF lattice])
    then obtain s where 2: "(s, t) ∈ relcl L" "s ∈ T" "s ∈ wp act M"
      by blast
    then have 3: "∀u ∈ L. s ∈ u --> t ∈ u"
      apply (intro ballI impI)
      apply (subst cl_ident [symmetric], assumption)

```

```

    apply (simp add: relcl_def)
    apply (blast intro: cl_mono [THEN [2] rev_subsetD])
  done
with 1 2 Fstable have 4: "funof act s ∈ T ∩ M"
  by (force intro!: funof_in
      simp add: wp_def stable_def constrains_def determ total)
with 1 2 3 have 5: "s ∈ B | t ∈ B | (funof act s, funof act t) ∈ relcl
L"
  by (intro dcommutes) assumption+
with 1 2 3 4 have "t ∈ B | funof act t ∈ cl L (T ∩ M)"
  by (simp add: relcl_def) (blast intro: BL cl_mono [THEN [2] rev_subsetD])

with 1 2 3 4 5 have "t ∈ B | t ∈ wp act (cl L (T ∩ M))"
  by (blast intro: funof_imp_wp determ)
with 2 3 have "t ∈ T ∧ (t ∈ B ∨ t ∈ wp act (cl L (T ∩ M)))"
  by (blast intro: TL cl_mono [THEN [2] rev_subsetD])
then have "t ∈ T ∩ (B ∪ wp act (cl L (T ∩ M)))"
  by simp
}
then show "commutes F T B L" unfolding commutes_def by clarify
qed

```

Version packaged with  $\llbracket ?F \in ?A \mapsto ?B'; ?C \in \text{progress\_set } ?F ?T ?B; ?F \in \text{UNITY.stable } ?T; ?B \subseteq ?B'; ?B' \in ?C; \bigwedge X. X \in ?C \implies ?G \in X - ?B \text{ co } X \rrbracket \implies ?F \sqcup ?G \in ?T \cap ?A \mapsto ?B'$

```

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"
apply (rule commutativity1 [OF leadsTo lattice])
apply (simp_all add: Gco commutativity2_lemma dcommutes determ total
    lattice BL TL Fstable)
done

```

## 16.6 Monotonicity

From Meier's thesis, section 4.5.7, page 110

end

## 17 Comprehensive UNITY Theory

theory UNITY\_Main



```

imports Detects PPROD Follows ProgressSets
begin

ML_file <UNITY_tactics.ML>

method_setup safety = <
  Scan.succeed (SIMPLE_METHOD' o constrains_tac)>
  "for proving safety properties"

method_setup ensures_tac = <
  Args.goal_spec -- Scan.lift Parse.embedded_inner_syntax >>
  (fn (quant, s) => fn ctxt => SIMPLE_METHOD'' quant (ensures_tac ctxt s))
> "for proving progress properties"

setup <
  map_theory_simpset (fn ctxt => ctxt
    addsimps (make_o_equivs ctxt @ {thm fst_o_funPair} @ make_o_equivs ctxt
      @ {thm snd_o_funPair})
    addsimps (make_o_equivs ctxt @ {thm fst_o_lift_map} @ make_o_equivs ctxt
      @ {thm snd_o_lift_map}))
  >

end

theory Deadlock imports "../UNITY" begin

lemma "[/ F ∈ (A ∩ B) co A; F ∈ (B ∩ A) co B /] ==> F ∈ stable (A ∩ B)"
  unfolding constrains_def stable_def by blast

lemma Collect_le_Int_equals:
  "( $\bigcap i \in \text{atMost } n. A(\text{Suc } i) \cap A i$ ) = ( $\bigcap i \in \text{atMost } (\text{Suc } n). A i$ )"
  by (induct n) (auto simp add: atMost_Suc)

lemma UN_Int_Compl_subset:
  "( $\bigcup i \in \text{lessThan } n. A i$ )  $\cap$  ( $\neg A n$ )  $\subseteq$ 
  ( $\bigcup i \in \text{lessThan } n. (A i) \cap (\neg A (\text{Suc } i))$ )"
  by (induct n) (auto simp: lessThan_Suc)

lemma INT_Un_Compl_subset:
  "( $\bigcap i \in \text{lessThan } n. \neg A i \cup A (\text{Suc } i)$ )  $\subseteq$ 
  ( $\bigcap i \in \text{lessThan } n. \neg A i$ )  $\cup A n$ "
  by (induct n) (auto simp: lessThan_Suc)

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

```

```
by (blast intro: grOI dest: INT_Un_Compl_subset [THEN subsetD])
```

```
lemma INT_le_equals_Int:
  "(\big i \in atMost n. A i) =
   A 0 \cap (\big i \in lessThan n. \neg A i \cup A(Suc i))"
  by (induct n)
    (simp_all add: Int_ac Int_Un_distrib Int_Un_distrib2
      INT_le_equals_Int_lemma lessThan_Suc atMost_Suc)
```

```
lemma INT_le_Suc_equals_Int:
  "(\big i \in atMost (Suc n). A i) =
   A 0 \cap (\big i \in atMost n. \neg A i \cup A(Suc i))"
  by (simp add: lessThan_Suc_atMost INT_le_equals_Int)
```

```
lemma
  assumes zeroprem: "F \in (A 0 \cap A (Suc n)) co (A 0)"
  and allprem:
    "!!i. i \in atMost n ==> F \in (A(Suc i) \cap A i) co (\neg A i \cup A(Suc
i))"
  shows "F \in stable (\big i \in atMost (Suc n). A i)"
  apply (unfold stable_def)
  apply (rule constrains_Int [THEN constrains_weaken])
  apply (rule zeroprem)
  apply (rule constrains_INT)
  apply (erule allprem)
  apply (simp add: Collect_le_Int_equals Int_assoc INT_absorb)
  apply (simp add: INT_le_Suc_equals_Int)
  done

end
```

```
theory Common
imports "../UNITY_Main"
begin
```

```
consts
  ftime :: "nat=>nat"
  gtime :: "nat=>nat"
```

```
axiomatization where
  fmono: "m \le n ==> ftime m \le ftime n" and
  gmono: "m \le n ==> gtime m \le gtime n" and

  fasc: "m \le ftime n" and
  gasc: "m \le 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)"
apply (drule_tac M = "{t. t ≤ n}" in Elimination_sing)
apply (simp add: atMost_def Stable_def common_def maxfg_def le_max_iff_disj)
apply (erule Constrains_weaken_R)
apply (blast intro: order_eq_refl le_trans dest: fmono gmono)
done
```

```
lemma common_safety:
  "[| Init F ⊆ atMost n;
    ∀m. F ∈ {m} Co (maxfg m); n ∈ common |]
   ==> F ∈ Always (atMost n)"
by (simp add: AlwaysI common_stable)
```

```
lemma "SKIP ∈ {m} co (maxfg m)"
by (simp add: constrains_def maxfg_def le_max_iff_disj fasc)
```

```
lemma "mk_total_program
  (UNIV, {range(%t.(t,ftime t)), range(%t.(t,gtime t))}, UNIV)
  ∈ {m} co (maxfg m)"
apply (simp add: mk_total_program_def)
apply (simp add: constrains_def maxfg_def le_max_iff_disj fasc)
done
```

```
lemma "mk_total_program (UNIV, {range(%t.(t, max (ftime t) (gtime t)))},
  UNIV)
  ∈ {m} co (maxfg m)"
apply (simp add: mk_total_program_def)
apply (simp add: constrains_def maxfg_def gasc max.absorb2)
done
```

```
lemma "mk_total_program
  (UNIV, { {(t, Suc t) | t. t < max (ftime t) (gtime t)} }, UNIV)
  ∈ {m} co (maxfg m)"
apply (simp add: mk_total_program_def)
apply (simp add: constrains_def maxfg_def gasc max.absorb2)
done
```

```

lemma leadsTo_common_lemma:
  assumes "\m. F \in {m} Co (maxfg m)"
    and "\m \in lessThan n. F \in {m} LeadsTo (greaterThan m)"
    and "n \in common"
  shows "F \in (atMost n) LeadsTo common"
proof (rule LeadsTo_weaken_R)
  show "F \in {..n} LeadsTo {..n} \cap id -' {n..} \cup common"
  proof (induct rule: GreaterThan_bounded_induct [of n _ _ id])
    case 1
    from assms have "\m \in {..<n}. F \in {..n} \cap {m} LeadsTo {..n} \cap {m<..}
  \cup common"
    by (blast dest: PSP_Stable2 intro: common_stable LeadsTo_weaken_R)
    then show ?case by simp
  qed
next
  from <n \in common>
  show "{..n} \cap id -' {n..} \cup common \subseteq common"
  by (simp add: atMost_Int_atLeast)
qed

```

```

lemma leadsTo_common:
  "[| \forall m. F \in {m} Co (maxfg m);
   \forall m \in -common. F \in {m} LeadsTo (greaterThan m);
   n \in common |]
   ==> F \in (atMost (LEAST n. n \in common)) LeadsTo common"
apply (rule leadsTo_common_lemma)
apply (simp_all (no_asm_simp))
apply (erule_tac [2] LeastI)
apply (blast dest!: not_less_Least)
done

end

```

```
theory Network imports "../UNITY" begin

```

```
datatype pvar = Sent | Rcvd | Idle

```

```
datatype pname = Aproc | Bproc

```

```
type_synonym state = "pname * pvar => nat"

```

```

locale F_props =
  fixes F
  assumes rsA: "F \in stable {s. s(Bproc,Rcvd) \le s(Aproc,Sent)}"
    and rsB: "F \in stable {s. s(Aproc,Rcvd) \le s(Bproc,Sent)}"
    and sent_nondec: "F \in stable {s. m \le s(proc,Sent)}"
    and rcvd_nondec: "F \in stable {s. n \le s(proc,Rcvd)}"
    and rcvd_idle: "F \in {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}"
begin

lemmas 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]

lemmas 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]

lemmas nondec_AB = stable_Int [OF sent_nondec_AB rcvd_nondec_AB]
    and idle_AB = constrains_Int [OF rcvd_idle_AB sent_idle_AB]

lemmas nondec_idle = constrains_Int [OF nondec_AB [unfolded stable_def] idle_AB]

lemma
  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}"
apply (unfold stable_def)
apply (rule constrainsI)
apply (drule constrains_Int [OF rs_AB [unfolded stable_def] nondec_idle,
                                THEN constrainsD], assumption)

apply simp_all
apply (blast del: le0, clarify)
apply (subgoal_tac "s' (Aproc, Rcvd) = s (Aproc, Rcvd)")
apply (subgoal_tac "s' (Bproc, Rcvd) = s (Bproc, Rcvd)")
apply simp
apply (blast intro: order_antisym le_trans eq_imp_le)+
done

end

end

```

## 18 The Token Ring

```

theory Token
imports "../WFair"

```

```

begin

```

From Misra, "A Logic for Concurrent Programming" (1994), sections 5.2 and

13.2.

## 18.1 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"
by (unfold HasTok_def, auto)
```

```
lemma not_E_eq: "(s  $\notin$  E i) = (s  $\in$  H i | s  $\in$  T i)"
apply (simp add: H_def E_def T_def)
apply (cases "proc s i", auto)
done
```

```
context Token
begin
```

```
lemma token_stable: "F  $\in$  stable (-(E i)  $\cup$  (HasTok i))"
apply (unfold stable_def)
apply (rule constrains_weaken)
```

```

apply (rule constrains_Un [OF constrains_Un [OF TR2 TR4] TR5])
apply (auto simp add: not_E_eq)
apply (simp_all add: H_def E_def T_def)
done

```

## 18.2 Progress under Weak Fairness

```

lemma wf_nodeOrder: "wf(nodeOrder j)"
apply (unfold nodeOrder_def)
apply (rule wf_measure [THEN wf_subset], blast)
done

```

```

lemma nodeOrder_eq:
  "[| i<N; j<N |] ==> ((next i, i) ∈ nodeOrder j) = (i ≠ j)"
  apply (cases <i < j>)
    apply (auto simp add: nodeOrder_def next_def mod_Suc add.commute [of _
N])
    apply (simp only: diff_add_assoc mod_add_self1)
    apply simp
  done

```

From "A Logic for Concurrent Programming", but not used in Chapter 4. Note the use of *cases*. Reasoning about *leadsTo* takes practice!

```

lemma TR7_nodeOrder:
  "[| i<N; j<N |] ==>
    F ∈ (HasTok i) leadsTo ({s. (token s, i) ∈ nodeOrder j} ∪ HasTok j)"
  apply (cases "i=j")
  apply (blast intro: subset_imp_leadsTo)
  apply (rule TR7 [THEN leadsTo_weaken_R])
  apply (auto simp add: HasTok_def nodeOrder_eq)
done

```

Chapter 4 variant, the one actually used below.

```

lemma TR7_aux: "[| i<N; j<N; i≠j |]
  ==> F ∈ (HasTok i) leadsTo {s. (token s, i) ∈ nodeOrder j}"
apply (rule TR7 [THEN leadsTo_weaken_R])
apply (auto simp add: HasTok_def nodeOrder_eq)
done

```

```

lemma token_lemma:
  "({s. token s < N} ∩ token - '{m}') = (if m<N then token - '{m}' else {})"
by auto

```

Misra's TR9: the token reaches an arbitrary node

```

lemma leadsTo_j: "j<N ==> F ∈ {s. token s < N} leadsTo (HasTok j)"
apply (rule leadsTo_weaken_R)
apply (rule_tac I = "-{j}" and f = token and B = "{}"
  in wf_nodeOrder [THEN bounded_induct])
apply (simp_all (no_asm_simp) add: token_lemma vimage_Diff HasTok_def)
  prefer 2 apply blast
apply clarify
apply (rule TR7_aux [THEN leadsTo_weaken])
apply (auto simp add: HasTok_def nodeOrder_def)

```

done

Misra's TR8: a hungry process eventually eats

```
lemma token_progress:
  "j < N ==> F ∈ ({s. token s < N} ∩ H j) leadsTo (E j)"
apply (rule leadsTo_cancel1 [THEN leadsTo_Un_duplicate])
apply (rule_tac [2] TR6)
apply (rule psp [OF leadsTo_j TR3, THEN leadsTo_weaken], blast+)
done
```

end

end

theory Channel imports "../UNITY\_Main" begin

type\_synonym state = "nat set"

consts

$F :: \text{"state program"}$

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

$\text{"minSet } A == \text{if } A = \{\} \text{ then None else Some (LEAST } x. x \in A)\text{"}$

axiomatization where

$UC1: \text{"} F \in (\text{minSet } -' \{ \text{Some } x \}) \text{ co } (\text{minSet } -' (\text{Some } \text{'atLeast } x)) \text{" and}$

$UC2: \text{"} F \in (\text{minSet } -' \{ \text{Some } x \}) \text{ leadsTo } \{ s. x \notin s \} \text{"}$

lemma minSet\_eq\_SomeD: "minSet A = Some x ==> x ∈ A"

```
apply (unfold minSet_def)
apply (simp split: if_split_asm)
apply (fast intro: LeastI)
done
```

lemma minSet\_empty [simp]: "minSet {} = None"

by (unfold minSet\_def, simp)

lemma minSet\_nonempty: "x ∈ A ==> minSet A = Some (LEAST x. x ∈ A)"

by (unfold minSet\_def, auto)

lemma minSet\_greaterThan:

```
"F ∈ (minSet -' {Some x}) leadsTo (minSet -' (Some 'greaterThan x))"
apply (rule leadsTo_weaken)
apply (rule_tac x1=x in psp [OF UC2 UC1], safe)
apply (auto dest: minSet_eq_SomeD simp add: linorder_neq_iff)
done
```



```

lemma Channel_progress_lemma:
  "F ∈ (UNIV-{{}}) leadsTo (minSet -' (Some 'atLeast y))"
apply (rule leadsTo_weaken_R)
apply (rule_tac l = y and f = "the o minSet" and B = "{}"
      in greaterThan_bounded_induct, safe)
apply (simp_all (no_asm_simp))
apply (drule_tac [2] minSet_nonempty)
  prefer 2 apply simp
apply (rule minSet_greaterThan [THEN leadsTo_weaken], safe)
apply simp_all
apply (drule minSet_nonempty, simp)
done

lemma Channel_progress: "!!y::nat. F ∈ (UNIV-{{}}) leadsTo {s. y ∉ s}"
apply (rule Channel_progress_lemma [THEN leadsTo_weaken_R], clarify)
apply (frule minSet_nonempty)
apply (auto dest: Suc_le_lessD not_less_Least)
done

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  $\cup$  below"

```

**definition**

```

goingup :: "state set"
where "goingup = above  $\cap$  ( $\{s. \text{up } s\} \cup \text{-below}$ )"

```

**definition**

```

goingdown :: "state set"
where "goingdown = below  $\cap$  ( $\{s. \sim \text{up } s\} \cup \text{-above}$ )"

```

**definition**

```

ready :: "state set"
where "ready =  $\{s. \text{stop } s \ \& \ \sim \text{open } s \ \& \ \text{move } s\}$ "

```

— Further abbreviations

**definition**

```

moving :: "state set"
where "moving =  $\{s. \sim \text{stop } s \ \& \ \sim \text{open } s\}$ "

```

**definition**

```

stopped :: "state set"
where "stopped =  $\{s. \text{stop } s \ \& \ \sim \text{open } s \ \& \ \sim \text{move } s\}$ "

```

**definition**

```

opened :: "state set"
where "opened =  $\{s. \text{stop } s \ \& \ \text{open } s \ \& \ \text{move } s\}$ "

```

**definition**

```

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

```

**definition**

```

atFloor :: "int => state set"
where "atFloor n =  $\{s. \text{floor } s = n\}$ "

```

**definition**

```

Req :: "int => state set"
where "Req n =  $\{s. n \in \text{req } s\}$ "

```

— The program

**definition**

```

request_act :: "(state*state) set"
where "request_act =  $\{(s,s'). \ s' = s \ (\text{!stop}:=\text{True}, \text{move}:=\text{False})$   

 $\ \& \ \sim \text{stop } s \ \& \ \text{floor } s \in \text{req } s\}$ "

```

**definition**

```

open_act :: "(state*state) set"
where "open_act =  

 $\{(s,s'). \ s' = s \ (\text{!open}:=\text{True},$ 
```

```

      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"  
 by blast

**declare** Lift\_def [THEN def\_prg\_Init, simp]

```

declare request_act_def [THEN def_act_simp, simp]
declare open_act_def [THEN def_act_simp, simp]
declare close_act_def [THEN def_act_simp, simp]
declare req_up_def [THEN def_act_simp, simp]
declare req_down_def [THEN def_act_simp, simp]
declare move_up_def [THEN def_act_simp, simp]
declare move_down_def [THEN def_act_simp, simp]
declare button_press_def [THEN def_act_simp, simp]

declare above_def [THEN def_set_simp, simp]
declare below_def [THEN def_set_simp, simp]
declare queueing_def [THEN def_set_simp, simp]
declare goingup_def [THEN def_set_simp, simp]
declare goingdown_def [THEN def_set_simp, simp]
declare ready_def [THEN def_set_simp, simp]

declare bounded_def [simp]
  open_stop_def [simp]
  open_move_def [simp]
  stop_floor_def [simp]
  moving_up_def [simp]
  moving_down_def [simp]

lemma open_stop: "Lift ∈ Always open_stop"
  apply (rule AlwaysI, force)
  apply (unfold Lift_def, safety)
  done

lemma stop_floor: "Lift ∈ Always stop_floor"
  apply (rule AlwaysI, force)
  apply (unfold Lift_def, safety)
  done

lemma open_move: "Lift ∈ Always open_move"
  apply (cut_tac open_stop)
  apply (rule AlwaysI, force)
  apply (unfold Lift_def, safety)
  done

lemma moving_up: "Lift ∈ Always moving_up"
  apply (rule AlwaysI, force)
  apply (unfold Lift_def, safety)
  apply (auto dest: order_le_imp_less_or_eq simp add: add1_zle_eq)
  done

lemma moving_down: "Lift ∈ Always moving_down"
  apply (rule AlwaysI, force)
  apply (unfold Lift_def, safety)
  apply (blast dest: order_le_imp_less_or_eq)
  done

```

```

lemma bounded: "Lift  $\in$  Always bounded"
apply (cut_tac moving_up moving_down)
apply (rule AlwaysI, force)
apply (unfold Lift_def, safety, auto)
apply (drule not_mem_distinct, assumption, arith)+
done

```

### 18.3 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  $n$ )"
apply (cut_tac stop_floor)
apply (unfold Lift_def, ensures_tac "open_act")
done

```

```

lemma E_thm02: "Lift  $\in$  (Req  $n \cap$  stopped - atFloor  $n$ ) LeadsTo
                (Req  $n \cap$  opened - atFloor  $n$ )"
apply (cut_tac stop_floor)
apply (unfold Lift_def, ensures_tac "open_act")
done

```

```

lemma E_thm03: "Lift  $\in$  (Req  $n \cap$  opened - atFloor  $n$ ) LeadsTo
                (Req  $n \cap$  closed - (atFloor  $n$  - queueing))"
apply (unfold Lift_def, ensures_tac "close_act")
done

```

```

lemma E_thm04: "Lift  $\in$  (Req  $n \cap$  closed  $\cap$  (atFloor  $n$  - queueing))
                LeadsTo (opened  $\cap$  atFloor  $n$ )"
apply (unfold Lift_def, ensures_tac "open_act")
done

```

```

lemmas linorder_leI = linorder_not_less [THEN iffD1]

```

```

context Floor
begin

```

```

lemmas le_MinD = Min_le_n [THEN order_antisym]
      and Max_leD = n_le_Max [THEN [2] order_antisym]

```

```

declare le_MinD [dest!]
  and linorder_leI [THEN le_MinD, dest!]
  and Max_leD [dest!]
  and linorder_leI [THEN Max_leD, dest!]

lemma E_thm05c:
  "Lift ∈ (Req n ∩ closed - (atFloor n - queueing))
    LeadsTo ((closed ∩ goingup ∩ Req n) ∪
              (closed ∩ goingdown ∩ Req n))"
by (auto intro!: subset_imp_LeadsTo simp add: linorder_neq_iff)

lemma lift_2: "Lift ∈ (Req n ∩ closed - (atFloor n - queueing))
  LeadsTo (moving ∩ Req n)"
apply (rule LeadsTo_Trans [OF E_thm05c LeadsTo_Un])
apply (unfold Lift_def)
apply (ensures_tac [2] "req_down")
apply (ensures_tac "req_up", auto)
done

declare if_split_asm [split]

lemma 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})"
apply (cut_tac moving_up)
apply (unfold Lift_def, ensures_tac "move_up", safe)

apply (erule linorder_leI [THEN order_antisym, symmetric])
apply (auto simp add: metric_def)
done

lemma 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})"
apply (cut_tac moving_down)
apply (unfold Lift_def, ensures_tac "move_down", safe)

apply (erule linorder_leI [THEN order_antisym, symmetric])
apply (auto simp add: metric_def)
done

```

```

lemma 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})"
apply (rule LeadsTo_Trans [OF subset_imp_LeadsTo
  LeadsTo_Un [OF E_thm12a E_thm12b]] , auto)
done

```

```

lemma E_thm16a: "0 < N
  ==> Lift ∈ (closed ∩ Req n ∩ {s. metric n s = N} ∩ goingup) LeadsTo
    (moving ∩ Req n ∩ {s. metric n s < N})"
apply (cut_tac bounded)
apply (unfold Lift_def, ensures_tac "req_up")
apply (auto simp add: metric_def)
done

```

```

lemma E_thm16b: "0 < N ==>
  Lift ∈ (closed ∩ Req n ∩ {s. metric n s = N} ∩ goingdown) LeadsTo
    (moving ∩ Req n ∩ {s. metric n s < N})"
apply (cut_tac bounded)
apply (unfold Lift_def, ensures_tac "req_down")
apply (auto simp add: metric_def)
done

```

```

lemma E_thm16c:
  "0 < N ==> Req n ∩ {s. metric n s = N} ⊆ goingup ∪ goingdown"
by (force simp add: metric_def)

```

```

lemma lift_5:
  "0 < N ==> Lift ∈ (closed ∩ Req n ∩ {s. metric n s = N}) LeadsTo
    (moving ∩ Req n ∩ {s. metric n s < N})"
apply (rule LeadsTo_Trans [OF subset_imp_LeadsTo
  LeadsTo_Un [OF E_thm16a E_thm16b]])
apply (drule E_thm16c, auto)
done

```

```

lemma metric_eq_OD [dest]:
  "[| metric n s = 0; Min ≤ floor s; floor s ≤ Max |] ==> floor s =

```



```

n"
by (force simp add: metric_def)

```

```

lemma E_thm11: "Lift ∈ (moving ∩ Req n ∩ {s. metric n s = 0}) LeadsTo
                (stopped ∩ atFloor n)"
apply (cut_tac bounded)
apply (unfold Lift_def, ensures_tac "request_act", auto)
done

```

```

lemma E_thm13:
  "Lift ∈ (moving ∩ Req n ∩ {s. metric n s = N} ∩ {s. floor s ∈ req s})

  LeadsTo (stopped ∩ Req n ∩ {s. metric n s = N} ∩ {s. floor s ∈ req s})"
apply (unfold Lift_def, ensures_tac "request_act")
apply (auto simp add: metric_def)
done

```

```

lemma E_thm14: "0 < N ==>
  Lift ∈
    (stopped ∩ Req n ∩ {s. metric n s = N} ∩ {s. floor s ∈ req s})
    LeadsTo (opened ∩ Req n ∩ {s. metric n s = N})"
apply (unfold Lift_def, ensures_tac "open_act")
apply (auto simp add: metric_def)
done

```

```

lemma E_thm15: "Lift ∈ (opened ∩ Req n ∩ {s. metric n s = N})
                LeadsTo (closed ∩ Req n ∩ {s. metric n s = N})"
apply (unfold Lift_def, ensures_tac "close_act")
apply (auto simp add: metric_def)
done

```

```

lemma lift_3_Req: "0 < N ==>
  Lift ∈
    (moving ∩ Req n ∩ {s. metric n s = N} ∩ {s. floor s ∈ req s})
    LeadsTo (moving ∩ Req n ∩ {s. metric n s < N})"
apply (blast intro!: E_thm13 E_thm14 E_thm15 lift_5 intro: LeadsTo_Trans)
done

```

```

lemma Always_nonneg: "Lift ∈ Always {s. 0 ≤ metric n s}"
apply (rule bounded [THEN Always_weaken])
apply (auto simp add: metric_def)
done

```

```

lemmas R_thm11 = Always_LeadsTo_weaken [OF Always_nonneg E_thm11]

```

```

lemma lift_3: "Lift  $\in$  (moving  $\cap$  Req n) LeadsTo (stopped  $\cap$  atFloor n)"
apply (rule Always_nonneg [THEN integ_0_le_induct])
apply (case_tac "0 < z")

prefer 2 apply (force intro: R_thm11 order_antisym simp add: linorder_not_less)
apply (rule LeadsTo_weaken_R [OF asm_rl Un_upper1])
apply (rule LeadsTo_Trans [OF subset_imp_LeadsTo
                        LeadsTo_Un [OF lift_4 lift_3_Req]], auto)

done

lemma lift_1: "Lift  $\in$  (Req n) LeadsTo (opened  $\cap$  atFloor n)"
apply (rule LeadsTo_Trans)
prefer 2
  apply (rule LeadsTo_Un [OF E_thm04 LeadsTo_Un_post])
  apply (rule E_thm01 [THEN [2] LeadsTo_Trans_Un])
  apply (rule lift_3 [THEN [2] LeadsTo_Trans_Un])
  apply (rule lift_2 [THEN [2] LeadsTo_Trans_Un])
  apply (rule LeadsTo_Trans_Un [OF E_thm02 E_thm03])
  apply (rule open_move [THEN Always_LeadsToI])
  apply (rule Always_LeadsToI [OF open_stop subset_imp_LeadsTo], clarify)

apply (case_tac "open x", auto)
done

end

end

theory Mutex imports "../UNITY_Main" begin

record state =
  p :: bool
  m :: int
  n :: int
  u :: bool
  v :: bool

type_synonym 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  $\in$  Always IU"
apply (rule AlwaysI, force)
apply (unfold Mutex_def, safety)
done

lemma IV: "Mutex  $\in$  Always IV"
apply (rule AlwaysI, force)
apply (unfold Mutex_def, safety)
done

lemma mutual_exclusion: "Mutex  $\in$  Always {s.  $\sim$  (m s = 3 & n s = 3)}"
apply (rule Always_weaken)
apply (rule Always_Int_I [OF IU IV], auto)
done

lemma "Mutex  $\in$  Always bad_IU"
apply (rule AlwaysI, force)
apply (unfold Mutex_def, safety, auto)

oops

lemma eq_123: "((1::int)  $\leq$  i & i  $\leq$  3) = (i = 1 | i = 2 | i = 3)"
by arith

lemma U_F0: "Mutex  $\in$  {s. m s=2} Unless {s. m s=3}"
by (unfold Unless_def Mutex_def, safety)

lemma U_F1: "Mutex  $\in$  {s. m s=1} LeadsTo {s. p s = v s & m s = 2}"
by (unfold Mutex_def, ensures_tac U1)

lemma U_F2: "Mutex  $\in$  {s.  $\sim$  p s & m s = 2} LeadsTo {s. m s = 3}"
apply (cut_tac IU)
apply (unfold Mutex_def, ensures_tac U2)
done

lemma U_F3: "Mutex  $\in$  {s. m s = 3} LeadsTo {s. p s}"
apply (rule_tac B = "{s. m s = 4}" in LeadsTo_Trans)
apply (unfold Mutex_def)

```

```

  apply (ensures_tac U3)
  apply (ensures_tac U4)
done

lemma U_lemma2: "Mutex  $\in$  {s. m s = 2} LeadsTo {s. p s}"
apply (rule LeadsTo_Diff [OF LeadsTo_weaken_L
                        Int_lower2 [THEN subset_imp_LeadsTo]])
apply (rule LeadsTo_Trans [OF U_F2 U_F3], auto)
done

lemma U_lemma1: "Mutex  $\in$  {s. m s = 1} LeadsTo {s. p s}"
by (rule LeadsTo_Trans [OF U_F1 [THEN LeadsTo_weaken_R] U_lemma2], blast)

lemma U_lemma123: "Mutex  $\in$  {s. 1  $\leq$  m s & m s  $\leq$  3} LeadsTo {s. p s}"
by (simp add: eq_123 Collect_disj_eq LeadsTo_Un_distrib U_lemma1 U_lemma2
    U_F3)

lemma u_Leadsto_p: "Mutex  $\in$  {s. u s} LeadsTo {s. p s}"
by (rule Always_LeadsTo_weaken [OF IU U_lemma123], auto)

lemma V_F0: "Mutex  $\in$  {s. n s=2} Unless {s. n s=3}"
by (unfold Unless_def Mutex_def, safety)

lemma V_F1: "Mutex  $\in$  {s. n s=1} LeadsTo {s. p s = ( $\sim$  u s) & n s = 2}"
by (unfold Mutex_def, ensures_tac "V1")

lemma V_F2: "Mutex  $\in$  {s. p s & n s = 2} LeadsTo {s. n s = 3}"
apply (cut_tac IV)
apply (unfold Mutex_def, ensures_tac "V2")
done

lemma V_F3: "Mutex  $\in$  {s. n s = 3} LeadsTo {s.  $\sim$  p s}"
apply (rule_tac B = "{s. n s = 4}" in LeadsTo_Trans)
  apply (unfold Mutex_def)
  apply (ensures_tac V3)
  apply (ensures_tac V4)
done

lemma V_lemma2: "Mutex  $\in$  {s. n s = 2} LeadsTo {s.  $\sim$  p s}"
apply (rule LeadsTo_Diff [OF LeadsTo_weaken_L
                        Int_lower2 [THEN subset_imp_LeadsTo]])
apply (rule LeadsTo_Trans [OF V_F2 V_F3], auto)
done

lemma V_lemma1: "Mutex  $\in$  {s. n s = 1} LeadsTo {s.  $\sim$  p s}"
by (rule LeadsTo_Trans [OF V_F1 [THEN LeadsTo_weaken_R] V_lemma2], blast)

lemma V_lemma123: "Mutex  $\in$  {s. 1  $\leq$  n s & n s  $\leq$  3} LeadsTo {s.  $\sim$  p s}"
by (simp add: eq_123 Collect_disj_eq LeadsTo_Un_distrib V_lemma1 V_lemma2

```

V\_F3)

```
lemma v_Leadsto_not_p: "Mutex  $\in$  {s. v s} LeadsTo {s.  $\sim$  p s}"
by (rule Always_LeadsTo_weaken [OF IV V_lemma123], auto)
```

```
lemma m1_Leadsto_3: "Mutex  $\in$  {s. m s = 1} LeadsTo {s. m s = 3}"
apply (rule LeadsTo_cancel2 [THEN LeadsTo_Un_duplicate])
apply (rule_tac [2] U_F2)
apply (simp add: Collect_conj_eq)
apply (subst Un_commute)
apply (rule LeadsTo_cancel2 [THEN LeadsTo_Un_duplicate])
  apply (rule_tac [2] PSP_Unless [OF v_Leadsto_not_p U_F0])
apply (rule U_F1 [THEN LeadsTo_weaken_R], auto)
done
```

```
lemma n1_Leadsto_3: "Mutex  $\in$  {s. n s = 1} LeadsTo {s. n s = 3}"
apply (rule LeadsTo_cancel2 [THEN LeadsTo_Un_duplicate])
apply (rule_tac [2] V_F2)
apply (simp add: Collect_conj_eq)
apply (subst Un_commute)
apply (rule LeadsTo_cancel2 [THEN LeadsTo_Un_duplicate])
  apply (rule_tac [2] PSP_Unless [OF u_Leadsto_p V_F0])
apply (rule V_F1 [THEN LeadsTo_weaken_R], auto)
done
```

end

```
theory Reach imports "../UNITY_Main" begin
```

```
typedecl vertex
```

```
type_synonym 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. (∀ v. s v --> (init, v) ∈ edges*) & s init}"

definition fixedpoint :: "state set"
  where "fixedpoint = {s. ∀ (u,v) ∈ edges. s u --> s v}"

definition metric :: "state => nat"
  where "metric s = card {v. ~ s v}"

*We assume that the set of vertices is finite

axiomatization where
  finite_graph: "finite (UNIV :: vertex set)"

lemma ifE [elim!]:
  "[| if P then Q else R;
    [| P;   Q |] ==> S;
    [| ~ P; R |] ==> S |] ==> S"
by (simp split: if_split_asm)

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"
apply (rule AlwaysI, force)
apply (unfold Rprg_def, safety)
apply (blast intro: rtrancl_trans)
done

lemma fixedpoint_invariant_correct:
  "fixedpoint ∩ reach_invariant = { %v. (init, v) ∈ edges* }"
apply (unfold fixedpoint_def)
apply (rule equalityI)
apply (auto intro!: ext)
apply (erule rtrancl_induct, auto)
done

lemma lemma1:
  "FP Rprg ⊆ fixedpoint"
apply (simp add: FP_def fixedpoint_def Rprg_def mk_total_program_def)
apply (auto simp add: stable_def constrains_def)

```

```

apply (drule bspec, assumption)
apply (simp add: Image_singleton image_iff)
apply (drule fun_cong, auto)
done

lemma lemma2:
  "fixedpoint  $\subseteq$  FP Rprg"
apply (simp add: FP_def fixedpoint_def Rprg_def mk_total_program_def)
apply (auto intro!: ext simp add: stable_def constrains_def)
done

lemma FP_fixedpoint: "FP Rprg = fixedpoint"
by (rule equalityI [OF lemma1 lemma2])

lemma Compl_fixedpoint: "- fixedpoint = ( $\bigcup_{(u,v) \in \text{edges}} \{s. s \ u \ \& \ \sim s \ v\}$ )"
apply (simp add: FP_fixedpoint [symmetric] Rprg_def mk_total_program_def)
apply (rule subset_antisym)
apply (auto simp add: Compl_FP UN_UN_flatten)
  apply (rule fun_upd_idem, force)
apply (force intro!: rev_bexI simp add: fun_upd_idem_iff)
done

lemma Diff_fixedpoint:
  "A - fixedpoint = ( $\bigcup_{(u,v) \in \text{edges}} A \cap \{s. s \ u \ \& \ \sim s \ v\}$ )"
by (simp add: Diff_eq Compl_fixedpoint, blast)

lemma Suc_metric: "~ s x ==> Suc (metric (s(x:=True))) = metric s"
apply (unfold metric_def)
apply (subgoal_tac "{v. ~ (s (x:=True)) v} = {v. ~ s v} - {x}")
  prefer 2 apply force
apply (simp add: card_Suc_Diff1 del:card_Diff_insert)
done

lemma metric_less [intro!]: "~ s x ==> metric (s(x:=True)) < metric s"
by (erule Suc_metric [THEN subst], blast)

lemma metric_le: "metric (s(y:=s x | s y))  $\leq$  metric s"
by (cases "s x --> s y") (auto intro: less_imp_le simp add: fun_upd_idem)

lemma LeadsTo_Diff_fixedpoint:
  "Rprg  $\in$  ((metric-'{m}) - fixedpoint) LeadsTo (metric-'{lessThan m})"
apply (simp (no_asm) add: Diff_fixedpoint Rprg_def)
apply (rule LeadsTo_UN, auto)
apply (ensures_tac "asgt a b")
  prefer 2 apply blast
apply (simp (no_asm_use) add: linorder_not_less)
apply (drule metric_le [THEN order_antisym])
apply (auto elim: less_not_refl3 [THEN [2] rev_notE])

```



done

lemma LeadsTo\_Un\_fixedpoint:

"Rprg  $\in$  (metric- $\{m\}$ ) LeadsTo (metric- $\{(lessThan\ m) \cup fixedpoint\}$ )"

apply (rule LeadsTo\_Diff [OF LeadsTo\_Diff\_fixedpoint [THEN LeadsTo\_weaken\_R] subset\_imp\_LeadsTo], auto)

done

lemma LeadsTo\_fixedpoint: "Rprg  $\in$  UNIV LeadsTo fixedpoint"

apply (rule LessThan\_induct, auto)

apply (rule LeadsTo\_Un\_fixedpoint)

done

lemma LeadsTo\_correct: "Rprg  $\in$  UNIV LeadsTo { %v. (init, v)  $\in$  edges\* }"

apply (subst fixedpoint\_invariant\_correct [symmetric])

apply (rule Always\_LeadsTo\_weaken [OF reach\_invariant LeadsTo\_fixedpoint], auto)

done

end

theory Reachability imports "../Detects" Reach begin

type\_synonym edge = "vertex \* vertex"

record state =

reach :: "vertex  $\Rightarrow$  bool"

nmsg :: "edge  $\Rightarrow$  nat"

consts root :: "vertex"

E :: "edge set"

V :: "vertex set"

inductive\_set REACHABLE :: "edge set"

where

base: " $v \in V \Rightarrow ((v, v) \in REACHABLE)$ "

| step: " $((u, v) \in REACHABLE) \ \& \ (v, w) \in E \Rightarrow ((u, w) \in REACHABLE)$ "

definition reachable :: "vertex  $\Rightarrow$  state set" where

"reachable p == {s. reach s p}"

definition nmsg\_eq :: "nat  $\Rightarrow$  edge  $\Rightarrow$  state set" where

"nmsg\_eq k == %e. {s. nmsg s e = k}"

definition nmsg\_gt :: "nat  $\Rightarrow$  edge  $\Rightarrow$  state set" where

"nmsg\_gt k == %e. {s. k < nmsg s e}"

definition nmsg\_gte :: "nat  $\Rightarrow$  edge  $\Rightarrow$  state set" where

"nmsg\_gte k == %e. {s. k  $\leq$  nmsg s e}"

definition nmsg\_lte :: "nat  $\Rightarrow$  edge  $\Rightarrow$  state set" where

```

nmsg_lte k == %e. {s. nmsg s e ≤ k}"

definition final :: "state set" where
  "final == (⋂ v∈V. reachable v <==> {s. (root, v) ∈ REACHABLE}) ∩
    (⋂ ((nmsg_eq 0) ' E))"

axiomatization
where
  Graph1: "root ∈ V" and

  Graph2: "(v,w) ∈ E ==> (v ∈ V) & (w ∈ V)" and

  MA1: "F ∈ Always (reachable root)" and

  MA2: "v ∈ V ==> F ∈ Always (- reachable v ∪ {s. ((root,v) ∈ REACHABLE)})"
and

  MA3: "[|v ∈ V; w ∈ V|] ==> F ∈ Always (-(nmsg_gt 0 (v,w)) ∪ (reachable
v))" and

  MA4: "(v,w) ∈ E ==>
    F ∈ Always (-(reachable v) ∪ (nmsg_gt 0 (v,w)) ∪ (reachable w))"
and

  MA5: "[|v ∈ V; w ∈ V|]
    ==> F ∈ Always (nmsg_gte 0 (v,w) ∩ nmsg_lte (Suc 0) (v,w))" and

  MA6: "[|v ∈ V|] ==> F ∈ Stable (reachable v)" and

  MA6b: "[|v ∈ V; w ∈ W|] ==> F ∈ Stable (reachable v ∩ nmsg_lte k (v,w))"
and

  MA7: "[|v ∈ V; w ∈ V|] ==> F ∈ UNIV LeadsTo nmsg_eq 0 (v,w)"

lemmas E_imp_in_V_L = Graph2 [THEN conjunct1]
lemmas E_imp_in_V_R = Graph2 [THEN conjunct2]

lemma lemma2:
  "(v,w) ∈ E ==> F ∈ reachable v LeadsTo nmsg_eq 0 (v,w) ∩ reachable v"
apply (rule MA7 [THEN PSP_Stable, THEN LeadsTo_weaken_L])
apply (rule_tac [3] MA6)
apply (auto simp add: E_imp_in_V_L E_imp_in_V_R)
done

lemma Induction_base: "(v,w) ∈ E ==> F ∈ reachable v LeadsTo reachable w"
apply (rule MA4 [THEN Always_LeadsTo_weaken])
apply (rule_tac [2] lemma2)
apply (auto simp add: nmsg_eq_def nmsg_gt_def)
done

lemma REACHABLE_LeadsTo_reachable:
  "(v,w) ∈ REACHABLE ==> F ∈ reachable v LeadsTo reachable w"
apply (erule REACHABLE.induct)

```

```

apply (rule subset_imp_LeadsTo, blast)
apply (blast intro: LeadsTo_Trans Induction_base)
done

```

```

lemma Detects_part1: "F ∈ {s. (root,v) ∈ REACHABLE} LeadsTo reachable v"
apply (rule single_LeadsTo_I)
apply (simp split: if_split_asm)
apply (rule MA1 [THEN Always_LeadsToI])
apply (erule REACHABLE_LeadsTo_reachable [THEN LeadsTo_weaken_L], auto)
done

```

```

lemma Reachability_Detected:
  "v ∈ V ==> F ∈ (reachable v) Detects {s. (root,v) ∈ REACHABLE}"
apply (unfold Detects_def, auto)
  prefer 2 apply (blast intro: MA2 [THEN Always_weaken])
apply (rule Detects_part1 [THEN LeadsTo_weaken_L], blast)
done

```

```

lemma LeadsTo_Reachability:
  "v ∈ V ==> F ∈ UNIV LeadsTo (reachable v <==> {s. (root,v) ∈ REACHABLE})"
by (erule Reachability_Detected [THEN Detects_Imp_LeadstoEQ])

```

```

lemma Eq_lemma1:
  "(reachable v <==> {s. (root,v) ∈ REACHABLE}) =
   {s. ((s ∈ reachable v) = ((root,v) ∈ REACHABLE))}"
by (unfold Equality_def, blast)

```

```

lemma Eq_lemma2:
  "(reachable v <==> (if (root,v) ∈ REACHABLE then UNIV else {})) =
   {s. ((s ∈ reachable v) = ((root,v) ∈ REACHABLE))}"
by (unfold Equality_def, auto)

```

```

lemma final_lemma1:
  "(( $\bigcap v \in V. \bigcap w \in V. \{s. ((s \in \text{reachable } v) = ((\text{root}, v) \in \text{REACHABLE}))$ )
  &
    $s \in \text{nmsg\_eq } 0 (v, w)\}$ )
    $\subseteq \text{final}$ "
apply (unfold final_def Equality_def, auto)
apply (frule E_imp_in_V_R)
apply (frule E_imp_in_V_L, blast)
done

```

```

lemma final_lemma2:
  "E≠{}
  ==> ( $\bigcap v \in V. \bigcap e \in E. \{s. ((s \in \text{reachable } v) = ((\text{root}, v) \in \text{REACHABLE}))\}$ 
       $\cap \text{nmsg\_eq } 0 \ e)$   $\subseteq \text{final}$ "
  apply (unfold final_def Equality_def)
  apply (auto split!: if_split)
  apply (frule E_imp_in_V_L, blast)
done

```

```

lemma final_lemma3:
  "E≠{}
  ==> ( $\bigcap v \in V. \bigcap e \in E.
      (\text{reachable } v \iff \{s. (\text{root}, v) \in \text{REACHABLE}\}) \cap \text{nmsg\_eq } 0 \ e)$ 
       $\subseteq \text{final}$ "
  apply (frule final_lemma2)
  apply (simp (no_asm_use) add: Eq_lemma2)
done

```

```

lemma final_lemma4:
  "E≠{}
  ==> ( $\bigcap v \in V. \bigcap e \in E.
      \{s. ((s \in \text{reachable } v) = ((\text{root}, v) \in \text{REACHABLE}))\} \cap \text{nmsg\_eq } 0 \ e)$ 
      = final"
  apply (rule subset_antisym)
  apply (erule final_lemma2)
  apply (unfold final_def Equality_def, blast)
done

```

```

lemma final_lemma5:
  "E≠{}
  ==> ( $\bigcap v \in V. \bigcap e \in E.
      ((\text{reachable } v) \iff \{s. (\text{root}, v) \in \text{REACHABLE}\}) \cap \text{nmsg\_eq } 0 \ e)$ 
      = final"
  apply (frule final_lemma4)
  apply (simp (no_asm_use) add: Eq_lemma2)
done

```

```

lemma final_lemma6:
  "( $\bigcap v \in V. \bigcap w \in V.
      (\text{reachable } v \iff \{s. (\text{root}, v) \in \text{REACHABLE}\}) \cap \text{nmsg\_eq } 0 \ (v, w)$ )
   $\subseteq \text{final}$ "
  apply (simp (no_asm) add: Eq_lemma2 Int_def)
  apply (rule final_lemma1)
done

```

```

lemma final_lemma7:
  "final =
  ( $\bigcap v \in V. \bigcap w \in V.$ 

```

```

      ((reachable v) <==> {s. (root,v) ∈ REACHABLE}) ∩
      (¬{s. (v,w) ∈ E} ∪ (nmsg_eq 0 (v,w))))"
apply (unfold final_def)
apply (rule subset_antisym, blast)
apply (auto split: if_split_asm)
apply (blast dest: E_imp_in_V_L E_imp_in_V_R)+
done

lemma not_REACHABLE_imp_Stable_not_reachable:
  "[| v ∈ V; (root,v) ∉ REACHABLE |] ==> F ∈ Stable (¬ reachable v)"
apply (drule MA2 [THEN AlwaysD], auto)
done

lemma Stable_reachable_EQ_R:
  "v ∈ V ==> F ∈ Stable (reachable v <==> {s. (root,v) ∈ REACHABLE})"
apply (simp (no_asm) add: Equality_def Eq_lemma2)
apply (blast intro: MA6 not_REACHABLE_imp_Stable_not_reachable)
done

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)"
apply (unfold nmsg_gte_def nmsg_lte_def nmsg_gt_def nmsg_eq_def, auto)
done

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)))"
by (unfold nmsg_gte_def nmsg_lte_def nmsg_gt_def nmsg_eq_def, auto)

lemma lemma6:
  "¬ nmsg_gt 0 (v,w) ∪ reachable v ⊆ nmsg_eq 0 (v,w) ∪ reachable v"
apply (unfold nmsg_gte_def nmsg_lte_def nmsg_gt_def nmsg_eq_def, auto)
done

lemma Always_reachable_OR_nmsg_0:
  "[|v ∈ V; w ∈ V|] ==> F ∈ Always (reachable v ∪ nmsg_eq 0 (v,w))"
apply (rule Always_Int_I [OF MA5 MA3, THEN Always_weaken])
apply (rule_tac [5] lemma4, auto)
done

lemma Stable_reachable_AND_nmsg_0:
  "[|v ∈ V; w ∈ V|] ==> F ∈ Stable (reachable v ∩ nmsg_eq 0 (v,w))"
apply (subst lemma5)

```

```

apply (blast intro: MA5 Always_imp_Stable [THEN Stable_Int] MA6b)
done

```

```

lemma Stable_nmsg_0_OR_reachable:
  "[| v ∈ V; w ∈ V |] ==> F ∈ Stable (nmsg_eq 0 (v,w) ∪ reachable v)"
by (blast intro!: Always_weaken [THEN Always_imp_Stable] lemma6 MA3)

```

```

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))"
apply (rule Stable_Int [OF MA2 [THEN Always_imp_Stable]
  Stable_nmsg_0_OR_reachable,
  THEN Stable_eq])
  prefer 4 apply blast
apply auto
done

```

```

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))"
by (simp add: Equality_def Eq_lemma2 Stable_reachable_AND_nmsg_0
  not_REACHABLE_imp_Stable_not_reachable_AND_nmsg_0)

```

```

lemma UNIV_lemma: "UNIV ⊆ (⋂ v ∈ V. UNIV)"
by blast

```

```

lemmas UNIV_LeadsTo_completion =
  LeadsTo_weaken_L [OF Finite_stable_completion UNIV_lemma]

```

```

lemma LeadsTo_final_E_empty: "E={} ==> F ∈ UNIV LeadsTo final"
apply (unfold final_def, simp)
apply (rule UNIV_LeadsTo_completion)
  apply safe
  apply (erule LeadsTo_Reachability [simplified])
apply (drule Stable_reachable_EQ_R, simp)
done

```

```

lemma Leadsto_reachability_AND_nmsg_0:
  "[| v ∈ V; w ∈ V |]
   ==> F ∈ UNIV LeadsTo
    ((reachable v <==> {s. (root,v) ∈ REACHABLE}) ∩ nmsg_eq 0 (v,w))"
apply (rule LeadsTo_Reachability [THEN LeadsTo_Trans], blast)
apply (subgoal_tac
  "F ∈ (reachable v <==> {s. (root,v) ∈ REACHABLE}) ∩
  UNIV LeadsTo (reachable v <==> {s. (root,v) ∈ REACHABLE}) ∩

```

```

      nmsg_eq 0 (v,w) ")
apply simp
apply (rule PSP_Stable2)
apply (rule MA7)
apply (rule_tac [3] Stable_reachable_EQ_R, auto)
done

lemma LeadsTo_final_E_NOT_empty: "E≠{} ==> F ∈ UNIV LeadsTo final"
apply (rule LeadsTo_weaken_L [OF LeadsTo_weaken_R UNIV_lemma])
apply (rule_tac [2] final_lemma6)
apply (rule Finite_stable_completion)
  apply blast
  apply (rule UNIV_LeadsTo_completion)
    apply (blast intro: Stable_INT Stable_reachable_EQ_R_AND_nmsg_0
      Leadsto_reachability_AND_nmsg_0)+
done

lemma LeadsTo_final: "F ∈ UNIV LeadsTo final"
apply (cases "E={}")
  apply (rule_tac [2] LeadsTo_final_E_NOT_empty)
  apply (rule LeadsTo_final_E_empty, auto)
done

lemma Stable_final_E_empty: "E={ } ==> F ∈ Stable final"
apply (unfold final_def, simp)
apply (rule Stable_INT)
apply (drule Stable_reachable_EQ_R, simp)
done

lemma Stable_final_E_NOT_empty: "E≠{} ==> F ∈ Stable final"
apply (subst final_lemma7)
apply (rule Stable_INT)
apply (rule Stable_INT)
apply (simp (no_asm) add: Eq_lemma2)
apply safe
apply (rule Stable_eq)
apply (subgoal_tac [2]
  "{s. (s ∈ reachable v) = ((root,v) ∈ REACHABLE) } ∩ nmsg_eq 0 (v,w))"
=
  "{s. (s ∈ reachable v) = ( (root,v) ∈ REACHABLE) } ∩ (- UNIV ∪ nmsg_eq
0 (v,w))")
prefer 2 apply blast
prefer 2 apply blast
apply (rule Stable_reachable_EQ_R_AND_nmsg_0
  [simplified Eq_lemma2 Collect_const])
apply (blast, blast)
apply (rule Stable_eq)
  apply (rule Stable_reachable_EQ_R [simplified Eq_lemma2 Collect_const])

```

```

  apply simp
  apply (subgoal_tac
    "{s. (s ∈ reachable v) = ((root,v) ∈ REACHABLE) } =
     {s. (s ∈ reachable v) = ( (root,v) ∈ REACHABLE) } ∩
     (- { } ∪ nmsg_eq 0 (v,w)))")
  apply blast+
done

lemma Stable_final: "F ∈ Stable final"
  apply (cases "E={}")
  prefer 2 apply (blast intro: Stable_final_E_NOT_empty)
  apply (blast intro: Stable_final_E_empty)
done

end

```

## 19 Analyzing the Needham-Schroeder Public-Key Protocol in UNITY

theory NSP\_Bad imports "HOL-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).

type\_synonym 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"
apply (intro exI bexI)
apply (rule_tac [2] act = "totalize_act NS3" in reachable.Acts)
apply (rule_tac [3] act = "totalize_act NS2" in reachable.Acts)
apply (rule_tac [4] act = "totalize_act NS1" in reachable.Acts)
apply (rule_tac [5] reachable.Init)
apply (simp_all (no_asm_simp) add: Nprg_def totalize_act_def)
apply auto
done

```

**19.1 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'"
apply (simp add: Nprg_def mk_total_program_def)
apply (rule constrainsI, auto)
done

```

This ML code does the inductions directly.

ML<

```

fun ns_constrains_tac ctxt i =
  SELECT_GOAL
  (EVERY
    [REPEAT (eresolve_tac ctxt @{thms Always_ConstrainsI} 1),
     REPEAT (resolve_tac ctxt [@{thm StableI}, @{thm stableI}, @{thm constrains_imp_ConstrainsI} 1),
     resolve_tac ctxt @{thms ns_constrainsI} 1,
     full_simp_tac ctxt 1,
     REPEAT (FIRSTGOAL (eresolve_tac ctxt [disjE])),
     ALLGOALS (clarify_tac (ctxt delrules [impI, @{thm impCE}])),
     REPEAT (FIRSTGOAL (analz_mono_contra_tac ctxt)),
     ALLGOALS (asm_simp_tac ctxt)]) i;

(*Tactic for proving secrecy theorems*)
fun ns_induct_tac ctxt =
  (SELECT_GOAL o EVERY)
  [resolve_tac ctxt @{thms AlwaysI} 1,
   force_tac ctxt 1,
   (*"reachable" gets in here*)
   resolve_tac ctxt [@{thm Always_reachable} RS @{thm Always_ConstrainsI}
    RS @{thm StableI}] 1,
   ns_constrains_tac ctxt 1];
>

method_setup ns_induct = <
  Scan.succeed (SIMPLE_METHOD' o ns_induct_tac)>
  "for inductive reasoning about the Needham-Schroeder protocol"

```

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)}"
apply ns_induct
apply blast
done
declare Spy_see_priK [THEN Always_Collect_reachableD, simp]

```

```

lemma Spy_analz_priK:
  "Nprg ∈ Always {s. (Key (priK A) ∈ analz (spies s)) = (A ∈ bad)}"
by (rule Always_reachable [THEN Always_weaken], auto)
declare Spy_analz_priK [THEN Always_Collect_reachableD, simp]

```

## 19.2 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)}"
apply ns_induct
apply (blast intro: analz_insertI)+
done

```

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'}"
apply ns_induct
apply auto

```

Fake, NS1 are non-trivial

**done**

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'"
by (blast dest: unique_NA_lemma [THEN Always_Collect_reachableD])

```

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)}"
apply ns_induct

```

NS3

```

prefer 4 apply (blast intro: no_nonce_NS1_NS2_reachable)

```

NS2

```

prefer 3 apply (blast dest: unique_NA)

```

NS1

```

prefer 2 apply blast

```

Fake

```

apply spy_analz
done

```

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}"

apply (insert Spy_not_see_NA [of A B NA], simp, ns_induct)
apply (auto dest: unique_NA)
done

```

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}"

apply ns_induct
apply blast
done

```

### 19.3 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'}"

apply ns_induct
apply auto

```

Fake, NS2 are non-trivial

**done**

```

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'"

apply (blast dest: unique_NB_lemma [THEN Always_Collect_reachableD])
done

```

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 &
        (∀ C. Says A C (Crypt (pubK C) (Nonce NB)) ∉ set s)
        --> Nonce NB ∉ analz (spies s)}"
apply ns_induct
apply (simp_all (no_asm_simp) add: all_conj_distrib)

NS3: because NB determines A
prefer 4 apply (blast dest: unique_NB)

NS2: by freshness and unicity of NB
prefer 3 apply (blast intro: no_nonce_NS1_NS2_reachable)

NS1: by freshness
prefer 2 apply blast

```

Fake

```

apply spy_analz
done

```

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)}"

apply (insert Spy_not_see_NB [of A B NA NB], simp, ns_induct)
apply (simp_all (no_asm_simp) add: ex_disj_distrib)
apply auto

```

NS3: because NB determines A. This use of `unique_NB` is robust.

```

apply (blast intro: unique_NB [THEN conjunct1])
done

```

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)}"

apply ns_induct
apply auto

```

Fake

```

apply spy_analz

```

NS2: by freshness and unicity of NB

```

  apply (blast intro: no_nonce_NS1_NS2_reachable)

```

NS3: unicity of NB identifies A and NA, but not B

```

apply (frule_tac A'=A in Says_imp_spies [THEN parts.Inj, THEN unique_NB])

```

```

apply (erule Says_imp_spies [THEN parts.Inj], auto)
apply (rename_tac s B' C)

```

This is the attack!

```

1.  $\bigwedge s$  B' C.
    $\llbracket A \notin \text{bad}; B \notin \text{bad}; s \in \text{reachable } \text{Nprg};$ 
    $\text{Says } A \ C \ (\text{Crypt } (\text{pubEK } C) \ \{\text{Nonce } NA, \text{Agent } A\}) \in \text{set } s;$ 
    $\text{Says } B' \ A \ (\text{Crypt } (\text{pubEK } A) \ \{\text{Nonce } NA, \text{Nonce } NB\}) \in \text{set } s;$ 
    $C \in \text{bad};$ 
    $\text{Says } B \ A \ (\text{Crypt } (\text{pubEK } A) \ \{\text{Nonce } NA, \text{Nonce } NB\}) \in \text{set } s;$ 
    $\text{Nonce } NB \notin \text{analz } (\text{knows } \text{Spy } s) \rrbracket$ 
 $\implies \text{False}$ 

```

oops

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  $\sqcup$  G)  $\in$  Always invFG"
apply (rule AlwaysI)
apply force
apply (rule constrains_imp_Constrains [THEN StableI])
apply auto
  apply (unfold F_def, safety)
  apply (unfold G_def, safety)
done

lemma lemma2_1: "(F  $\sqcup$  G)  $\in$  ({s. NF s = k} - {s. BB s}) LeadsTo
  ({s. NF s = k} Int {s. BB s})"
apply (rule stable_Join_ensures1[THEN leadsTo_Basis, THEN leadsTo_imp_LeadsTo])
  apply (unfold F_def, safety)
  apply (unfold G_def, ensures_tac "cmdG")
done

lemma lemma2_2: "(F  $\sqcup$  G)  $\in$  ({s. NF s = k} Int {s. BB s}) LeadsTo
  {s. k < NF s}"
apply (rule stable_Join_ensures2[THEN leadsTo_Basis, THEN leadsTo_imp_LeadsTo])
  apply (unfold F_def, ensures_tac "cmdF")
  apply (unfold G_def, safety)
done

lemma progress: "(F  $\sqcup$  G)  $\in$  UNIV LeadsTo {s. m < NF s}"
apply (rule LeadsTo_weaken_R)
apply (rule_tac f = "NF" and l = "Suc m" and B = "{}"
  in GreaterThan_bounded_induct)

apply (auto intro!: lemma2_1 lemma2_2
  intro: LeadsTo_Trans LeadsTo_Diff simp add: vimage_def)
done

end

```

## 20 A Family of Similar Counters: Original Version

```
theory Counter imports "../UNITY_Main" begin
```

```
datatype name = C | c nat
type_synonym 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)"

type_synonym 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: "I < n ==> sum I (s(c n := x)) = sum I s"
  by (induct I) auto

lemma sum_upd_eq: "sum I (s(c I := x)) = sum I s"
  by (induct I) (auto simp add: sum_upd_gt [unfolded fun_upd_def])

lemma sum_upd_C: "sum I (s(C := x)) = sum I s"
  by (induct I) auto

lemma sumj_upd_ci: "sumj I i (s(c i := x)) = sumj I i s"
  by (induct I) (auto simp add: sum_upd_eq [unfolded fun_upd_def])

lemma sumj_upd_C: "sumj I i (s(C := x)) = sumj I i s"
  by (induct I) (auto simp add: sum_upd_C [unfolded fun_upd_def])

lemma sumj_sum_gt: "I < i ==> sumj I i s = sum I s"
  by (induct I) auto

lemma sumj_sum_eq: "(sumj I I s = sum I s)"
  by (induct I) (auto simp add: sumj_sum_gt)

lemma sum_sumj: "i < I ==> sum I s = s (c i) + sumj I i s"
  by (induct I) (auto simp add: linorder_neq_iff sumj_sum_eq)

lemma p2: "Component i ∈ stable {s. s C = s (c i) + k}"
  by (simp add: Component_def, safety)

lemma p3: "Component i ∈ stable {s. ∀ v. v ≠ c i & v ≠ C --> s v = k v}"

```



```

by (simp add: Component_def, safety)

lemma p2_p3_lemma1:
  "( $\forall k. \text{Component } i \in \text{stable } \{s. s \ C = s \ (c \ i) + \text{sumj } I \ i \ k\}$ 
     $\cap \{s. \forall v. v \neq c \ i \ \& \ v \neq C \ \rightarrow s \ v = k \ v\}$ )
   = ( $\text{Component } i \in \text{stable } \{s. s \ C = s \ (c \ i) + \text{sumj } I \ i \ s\}$ )"
apply (simp add: Component_def mk_total_program_def)
apply (auto simp add: constrains_def stable_def sumj_upd_C sumj_upd_ci)
done

lemma p2_p3_lemma2:
  " $\forall k. \text{Component } i \in \text{stable } \{s. s \ C = s \ (c \ i) + \text{sumj } I \ i \ k\} \text{ Int}$ 
     $\{s. \forall v. v \neq c \ i \ \& \ v \neq C \ \rightarrow s \ v = k \ v\}$ "
by (blast intro: stable_Int [OF p2 p3])

lemma p2_p3: "Component  $i \in \text{stable } \{s. s \ C = s \ (c \ i) + \text{sumj } I \ i \ s\}$ "
by (auto intro!: p2_p3_lemma2 simp add: p2_p3_lemma1 [symmetric])

lemma sum_0': " $(\bigwedge i. i < I \Rightarrow s \ (c \ i) = 0) \Rightarrow \text{sum } I \ s = 0$ "
by (induct I) auto

lemma safety:
  " $0 < I \Rightarrow (\bigcup i \in \{i. i < I\}. \text{Component } i) \in \text{invariant } \{s. s \ C = \text{sum } I \ s\}$ "
apply (simp (no_asm) add: invariant_def JN_stable sum_sumj)
apply (force intro: p2_p3 sum_0')
done

end

```

## 21 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)"

```

```

type_synonym 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},
     $\bigcup G \in \text{preserves } (\%s. (c s) i). \text{Acts } G$ )"

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: "I<i ==> sum I s = sumj I i s"
  by (induct I) auto

lemma sum_sumj_eq2: "i<I ==> sum I s = c s i + sumj I i s"
  by (induct I) (auto simp add: linorder_neq_iff sum_sumj_eq1)

lemma sum_ext: "( $\bigwedge i. i<I \implies c s' i = c s i$ ) ==> sum I s' = sum I s"
  by (induct I) auto

lemma sumj_ext: "( $\bigwedge j. j<I \implies j \neq i \implies c s' j = c s j$ ) ==> sumj I i s' = sumj I i s"
  by (induct I) (auto intro!: sum_ext)

lemma sum0: "( $\bigwedge i. i<I \implies c s i = 0$ ) ==> sum I s = 0"
  by (induct I) auto

lemma Component_ok_iff:
  "(Component i ok G) =
    (G  $\in$  preserves (%s. c s i) & Component i  $\in$  Allowed G)"
apply (auto simp add: ok_iff_Allowed Component_def [THEN def_total_prg_Allowed])
done
declare Component_ok_iff [iff]
declare OK_iff_ok [iff]
declare preserves_def [simp]

lemma p2: "Component i  $\in$  stable {s. C s = (c s) i + k}"
  by (simp add: Component_def, safety)

lemma p3:
  "[| OK I Component; i $\in$ I |]
   ==> Component i  $\in$  stable {s.  $\forall j \in I. j \neq i \implies c s j = c k j$ }"
apply simp
apply (unfold Component_def mk_total_program_def)
apply (simp (no_asm_use) add: stable_def constrains_def)

```

```

apply blast
done

```

```

lemma p2_p3_lemma1:
  "[| OK {i. i<I} Component; i<I |] ==>
    ∀k. Component i ∈ stable ({s. C s = c s i + sumj I i k} Int
      {s. ∀j∈{i. i<I}. j≠i --> c s j = c k j})"
by (blast intro: stable_Int [OF p2 p3])

```

```

lemma p2_p3_lemma2:
  "(∀k. F ∈ stable ({s. C s = (c s) i + sumj I i k} Int
    {s. ∀j∈{i. i<I}. j≠i --> c s j = c k j}))
  ==> (F ∈ stable {s. C s = c s i + sumj I i s})"
apply (simp add: constrains_def stable_def)
apply (force intro!: sumj_ext)
done

```

```

lemma p2_p3:
  "[| OK {i. i<I} Component; i<I |]
  ==> Component i ∈ stable {s. C s = c s i + sumj I i s}"
by (blast intro: p2_p3_lemma1 [THEN p2_p3_lemma2])

```

```

lemma safety:
  "[| 0<I; OK {i. i<I} Component |]
  ==> (⋂ i∈{i. i<I}. (Component i)) ∈ invariant {s. C s = sum I s}"
apply (simp (no_asm) add: invariant_def JN_stable sum_sumj_eq2)
apply (auto intro!: sum0 p2_p3)
done

end

```

```

theory PriorityAux
imports "../UNITY_Main"
begin

```

```

typedecl vertex

```

```

definition symcl :: "(vertex*vertex)set=>(vertex*vertex)set" where
  "symcl r == r ∪ (r-1)"
  — symmetric closure: removes the orientation of a relation

```

```

definition neighbors :: "[vertex, (vertex*vertex)set]=>vertex set" where
  "neighbors i r == ((r ∪ 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} ∩ r) ∪ ({(x,y). x=i|y=i} ∩ r)-1"

definition derive1 :: "[vertex, (vertex*vertex)set, (vertex*vertex)set]=>bool"
where
  — The original definition
  "derive1 i r q == symcl r = symcl q &
    (∀ k k'. k≠i & k'≠i -->((k,k') ∈ r) = ((k,k') ∈ q)) ∧
    A i r = {} & R i q = {}"

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)"

axiomatization where
  finite_vertex_univ: "finite (UNIV :: vertex set)"
  — we assume that the universe of vertices is finite

declare derive_def [simp] derive1_def [simp] symcl_def [simp]
  A_def [simp] R_def [simp]
  above_def [simp] reach_def [simp]
  reverse_def [simp] neighbors_def [simp]

All vertex sets are finite

declare finite_subset [OF subset_UNIV finite_vertex_univ, iff]

and relatons over vertex are finite too

lemmas finite_UNIV_Prod =
  finite_Prod_UNIV [OF finite_vertex_univ finite_vertex_univ]

declare finite_subset [OF subset_UNIV finite_UNIV_Prod, iff]

lemma image0_trancl_iff_image0_r: "((r+)' '{i} = {}) = (r' '{i} = {})"
apply auto
apply (erule trancl_induct, auto)
done

lemma image0_r_iff_image0_trancl: "(r' '{i}={}) = (∀ x. ((i,x) ∈ r+) = False)"
apply auto
apply (drule image0_trancl_iff_image0_r [THEN ssubst], auto)

```

done

```
lemma acyclic_eq_wf: "!!r::(vertex*vertex)set. acyclic r = wf r"
by (auto simp add: wf_iff_acyclic_if_finite)
```

```
lemma derive_derive1_eq: "derive i r q = derive1 i r q"
by auto
```

```
lemma lemma1_a:
  "[| x ∈ reach i q; derive1 k r q |] ==> x≠k --> x ∈ reach i r"
apply (unfold reach_def)
apply (erule ImageE)
apply (erule trancl_induct)
  apply (cases "i=k", simp_all)
  apply (blast, blast, clarify)
apply (drule_tac x = y in spec)
apply (drule_tac x = z in spec)
apply (blast dest: r_into_trancl intro: trancl_trans)
done
```

```
lemma reach_lemma: "derive k r q ==> reach i q ⊆ (reach i r ∪ {k})"
apply clarify
apply (drule lemma1_a)
apply (auto simp add: derive_derive1_eq
  simp del: reach_def derive_def derive1_def)
done
```

```
lemma reach_above_lemma:
  "(∀i. reach i q ⊆ (reach i r ∪ {k})) =
  (∀x. x≠k --> (∀i. i ∉ above x r --> i ∉ above x q))"
by (auto simp add: trancl_converse)
```

```
lemma maximal_converse_image0:
  "(z, i) ∈ r+ ==> (∀y. (y, z) ∈ r → (y, i) ∉ r+) = ((r-1)+ "{z}={}")"
apply auto
apply (frule_tac r = r in trancl_into_trancl2, auto)
done
```

```
lemma above_lemma_a:
  "acyclic r ==> A i r≠{}-->(∃j ∈ above i r. A j r = {})"
apply (simp add: acyclic_eq_wf wf_eq_minimal)
apply (drule_tac x = " ((r-1)+ "{i}" in spec)
apply auto
apply (simp add: maximal_converse_image0 trancl_converse)
done
```

```
lemma above_lemma_b:
  "acyclic r ==> above i r≠{}-->(∃j ∈ above i r. above j r = {})"
```

```

apply (drule above_lemma_a)
apply (auto simp add: image0_trancl_iff_image0_r)
done

end

```

## 22 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.

```

type_synonym state = "(vertex*vertex)set"
type_synonym 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  $\longleftrightarrow$  A i r = {}"
  — i has highest priority in r

```

```

definition lowest :: "[vertex, (vertex*vertex)set]=>bool"
  where "lowest i r  $\longleftrightarrow$  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.  $\sim$ highest i s  $\longrightarrow$  ( $\exists$  j  $\in$  above i s. highest j s)})"

```

```

definition Maximal' :: "state set"

```

— Maximal vertex: equivalent definition  
**where** "Maximal" =  $(\bigcap i. \text{Highest } i \text{ Un } (\bigcup j. \{s. j \in \text{above } i \text{ s}\} \text{ Int Highest } j))$ "

**definition** Safety :: "state set"  
**where** "Safety =  $(\bigcap i. \{s. \text{highest } i \text{ s} \rightarrow (\forall j \in \text{neighbors } i \text{ s}. \sim \text{highest } j \text{ s})\})$ "

**definition** system :: "state program"  
**where** "system =  $(\bigcup i. \text{Component } i)$ "

**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]

## 22.1 Component correctness proofs

neighbors is stable

**lemma** Component\_neighbors\_stable: "Component i  $\in$  stable {s. neighbors k s = n}"  
**by** (simp add: Component\_def, safety, auto)

property 4

**lemma** Component\_waits\_priority: "Component i  $\in$  {s.  $((i,j) \in s) = b\} \cap (\sim \text{Highest } i) \text{ co } \{s. ((i,j) \in s) = b\}$ "  
**by** (simp add: Component\_def, safety)

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  $\in$  {s. neighbors i s  $\neq$  {}} Int Highest i  
 ensures - Highest i"  
**apply** (simp add: Component\_def)  
**apply** (ensures\_tac "act i", blast+)  
**done**

or better

**lemma** Component\_yields\_priority': "Component i  $\in$  Highest i ensures Lowest i"  
**apply** (simp add: Component\_def)  
**apply** (ensures\_tac "act i", blast+)  
**done**

property 6: Component doesn't introduce cycle

```

lemma Component_well_behaves: "Component i ∈ Highest i co Highest i Un Lowest i"
by (simp add: Component_def, safety, fast)

```

property 7: local axiom

```

lemma locality: "Component i ∈ stable {s. ∀ j k. j ≠ i & k ≠ i → ((j,k) ∈ s) = b j k}"
by (simp add: Component_def, safety)

```

## 22.2 System properties

property 8: strictly universal

```

lemma Safety: "system ∈ stable Safety"
apply (unfold Safety_def)
apply (rule stable_INT)
apply (simp add: system_def, safety, fast)
done

```

property 13: universal

```

lemma p13: "system ∈ {s. s = q} co {s. s=q} Un {s. ∃ i. derive i q s}"
by (simp add: system_def Component_def mk_total_program_def totalize_JN, safety, blast)

```

property 14: the 'above set' of a Component that hasn't got priority doesn't increase

```

lemma above_not_increase:
  "system ∈ -Highest i Int {s. j ∉ above i s} co {s. j ∉ above i s}"
apply (insert reach_lemma [of concl: j])
apply (simp add: system_def Component_def mk_total_program_def totalize_JN,
  safety)
apply (simp add: trancl_converse, blast)
done

```

```

lemma above_not_increase':
  "system ∈ -Highest i Int {s. above i s = x} co {s. above i s ≤ x}"
apply (insert above_not_increase [of i])
apply (simp add: trancl_converse constrains_def, blast)
done

```

p15: universal property: all Components well behave

```

lemma system_well_behaves: "system ∈ Highest i co Highest i Un Lowest i"
by (simp add: system_def Component_def mk_total_program_def totalize_JN, safety, auto)

```

```

lemma Acyclic_eq: "Acyclic = (⋂ i. {s. i ∉ above i s})"
by (auto simp add: Acyclic_def acyclic_def trancl_converse)

```

lemmas system\_co =



```
constrains_Un [OF above_not_increase [rule_format] system_well_behaves]
```

```
lemma Acyclic_stable: "system ∈ stable Acyclic"
apply (simp add: stable_def Acyclic_eq)
apply (auto intro!: constrains_INT system_co [THEN constrains_weaken]
      simp add: image0_r_iff_image0_trancl trancl_converse)
done
```

```
lemma Acyclic_subset_Maximal: "Acyclic ≤ Maximal"
apply (unfold Acyclic_def Maximal_def, clarify)
apply (drule above_lemma_b, auto)
done
```

property 17: original one is an invariant

```
lemma Acyclic_Maximal_stable: "system ∈ stable (Acyclic Int Maximal)"
by (simp add: Acyclic_subset_Maximal [THEN Int_absorb2] Acyclic_stable)
```

property 5: existential property

```
lemma Highest_leadsTo_Lowest: "system ∈ Highest i leadsTo Lowest i"
apply (simp add: system_def Component_def mk_total_program_def totalize_JN)
apply (ensures_tac "act i", auto)
done
```

a lowest i can never be in any abover set

```
lemma Lowest_above_subset: "Lowest i ≤ (⋂ k. {s. i ∉ above k s})"
by (auto simp add: image0_r_iff_image0_trancl trancl_converse)
```

property 18: a simpler proof than the original, one which uses psp

```
lemma Highest_escapes_above: "system ∈ Highest i leadsTo (⋂ k. {s. i ∉ above
k s})"
apply (rule leadsTo_weaken_R)
apply (rule_tac [2] Lowest_above_subset)
apply (rule Highest_leadsTo_Lowest)
done
```

```
lemma Highest_escapes_above':
  "system ∈ Highest j Int {s. j ∈ above i s} leadsTo {s. j ∉ above i s}"
by (blast intro: leadsTo_weaken [OF Highest_escapes_above Int_lower1 INT_lower])
```

## 22.3 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 = {}}"
by (auto simp add: image0_trancl_iff_image0_r)
```

```
lemmas above_decreases_lemma =
  psp [THEN leadsTo_weaken, OF Highest_escapes_above' above_not_increase']
```

```

lemma above_decreases:
  "system  $\in$  ( $\bigcup j. \{s. \text{above } i \ s = x\} \text{ Int } \{s. j \in \text{above } i \ s\} \text{ Int Highest } j$ )
    leadsTo  $\{s. \text{above } i \ s < x\}$ "
apply (rule leadsTo_UN)
apply (rule single_leadsTo_I, clarify)
apply (rule_tac x = "above i xa" in above_decreases_lemma)
apply (simp_all (no_asm_use) add: Highest_iff_above0)
apply blast+
done

```

```

lemma Maximal_eq_Maximal': "Maximal = Maximal'"
by (unfold Maximal_def Maximal'_def Highest_def, blast)

```

```

lemma Acyclic_subset:
  " $x \neq \{\}$  ==>
    Acyclic Int  $\{s. \text{above } i \ s = x\} \leq$ 
    ( $\bigcup j. \{s. \text{above } i \ s = x\} \text{ Int } \{s. j \in \text{above } i \ s\} \text{ Int Highest } j$ )"
apply (rule_tac B = "Maximal' Int  $\{s. \text{above } i \ s = x\}$ " in subset_trans)
apply (simp (no_asm) add: Maximal_eq_Maximal' [symmetric])
apply (blast intro: Acyclic_subset_Maximal [THEN subsetD])
apply (simp (no_asm) del: above_def add: Maximal'_def Highest_iff_above0)
apply blast
done

```

```

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':
  " $x \neq \{\}$  ==> system  $\in$  Acyclic Int  $\{s. \text{above } i \ s = x\}$  leadsTo
    Acyclic Int  $\{s. \text{above } i \ s < x\}$ "
by (erule above_decreases_psp [THEN leadsTo_weaken], blast, auto)

```

```

lemmas finite_psubset_induct = wf_finite_psubset [THEN leadsTo_wf_induct]

```

```

lemma Progress: "system  $\in$  Acyclic leadsTo Highest i"
apply (rule_tac f = "%s. above i s" in finite_psubset_induct)
apply (simp del: above_def
  add: Highest_iff_above0 vimage_def finite_psubset_def, clarify)
apply (case_tac "m={}")
apply (rule Int_lower2 [THEN [2] leadsTo_weaken_L])
apply (force simp add: leadsTo_refl)
apply (rule_tac A' = "Acyclic Int  $\{x. \text{above } i \ x < m\}$ " in leadsTo_weaken_R)
apply (blast intro: above_decreases_psp')+
done

```

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

type_synonym '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}"
apply (rule_tac f = count in lessThan_induct, simp)
apply (case_tac "m")
  apply (force intro!: subset_imp_leadsTo)
  apply (unfold Timer_def, ensures_tac "decr")
done

lemma Timer_preserves_snd [iff]: "Timer ∈ preserves snd"
apply (rule preservesI)
apply (unfold Timer_def, safety)
done

declare PLam_stable [simp]

lemma TimerArray_leadsTo_zero:
  "finite I
  ⇒ (plam i: I. Timer) ∈ UNIV leadsTo {(s,uu). ∀ i∈I. s i = 0}"
apply (erule_tac A'1 = "λi. lift_set i ({0} × UNIV)"
  in finite_stable_completion [THEN leadsTo_weaken])
apply auto

  prefer 2
  apply (simp add: Timer_def, safety)

apply (rule_tac f = "sub i o fst" in lessThan_induct)
apply (case_tac "m")

apply (auto intro: subset_imp_leadsTo
  simp add: insert_absorb
    lift_set_Un_distrib [symmetric] lessThan_Suc [symmetric]
    Times_Un_distrib1 [symmetric] Times_Diff_distrib1 [symmetric])

```

```

apply (rename_tac "n")
apply (rule PLam_leadsTo_Basis)
apply (auto simp add: lessThan_Suc [symmetric])
apply (unfold Timer_def mk_total_program_def, safety)
apply (rule_tac act = decr in totalize_transientI, auto)
done

end

```

## 23 Progress Set Examples

theory Progress imports "../UNITY\_Main" begin

### 23.1 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)"

```

#### 23.1.1 Calculating wens\_set FF {k..}

```

lemma Domain_actFF: "Domain (range ( $\lambda x::int. (x, x + 1)$ )) = UNIV"
by force

```

```

lemma FF_eq:
  "FF = mk_program (UNIV, {range ( $\lambda x. (x, x+1)$ )}, UNIV)"
by (simp add: FF_def mk_total_program_def totalize_def totalize_act_def
  program_equalityI Domain_actFF)

```

```

lemma wp_actFF:
  "wp (range ( $\lambda x::int. (x, x + 1)$ )) (atLeast k) = atLeast (k - 1)"
by (force simp add: wp_def)

```

```

lemma wens_FF: "wens FF (range ( $\lambda x. (x, x+1)$ )) (atLeast k) = atLeast (k - 1)"
by (force simp add: FF_eq wens_single_eq wp_actFF)

```

```

lemma single_valued_actFF: "single_valued (range ( $\lambda x::int. (x, x + 1)$ ))"
by (force simp add: single_valued_def)

```

```

lemma wens_single_finite_FF:
  "wens_single_finite (range ( $\lambda x. (x, x+1)$ )) (atLeast k) n =
    atLeast (k - int n)"
apply (induct n, simp)
apply (force simp add: wens_FF
  def_wens_single_finite_Suc_eq_wens [OF FF_eq single_valued_actFF])
done

```

```

lemma wens_single_FF_eq_UNIV:
  "wens_single (range ( $\lambda x::int. (x, x + 1)$ )) (atLeast k) = UNIV"

```

```

apply (auto simp add: wens_single_eq_Union)
apply (rule_tac x="nat (k-x)" in exI)
apply (simp add: wens_single_finite_FF)
done

lemma wens_set_FF:
  "wens_set FF (atLeast k) = insert UNIV (atLeast ' atMost k)"
apply (auto simp add: wens_set_single_eq [OF FF_eq single_valued_actFF]
  wens_single_FF_eq_UNIV wens_single_finite_FF)
apply (erule notE)
apply (rule_tac x="nat (k-xa)" in range_eqI)
apply (simp add: wens_single_finite_FF)
done

```

### 23.1.2 Proving $FF \in UNIV \mapsto \{k..\}$

```

lemma atLeast_ensures: "FF ∈ atLeast (k - 1) ensures atLeast (k::int)"
apply (simp add: Progress.wens_FF [symmetric] wens_ensures)
apply (simp add: wens_ensures FF_eq)
done

lemma atLeast_leadsTo: "FF ∈ atLeast (k - int n) leadsTo atLeast (k::int)"
apply (induct n)
  apply (simp_all add: leadsTo_refl)
  apply (rule_tac A = "atLeast (k - int n - 1)" in leadsTo_weaken_L)
  apply (blast intro: leadsTo_Trans atLeast_ensures, force)
done

lemma UN_atLeast_UNIV: "(⋃ n. atLeast (k - int n)) = UNIV"
apply auto
apply (rule_tac x = "nat (k - x)" in exI, simp)
done

lemma FF_leadsTo: "FF ∈ UNIV leadsTo atLeast (k::int)"
apply (subst UN_atLeast_UNIV [symmetric])
apply (rule leadsTo_UN [OF atLeast_leadsTo])
done

```

Result (4.39): Applying the leadsTo-Join Theorem

```

theorem "FF⊔GG ∈ atLeast 0 leadsTo atLeast (k::int)"
apply (subgoal_tac "FF⊔GG ∈ (atLeast 0 ∩ atLeast 0) leadsTo atLeast k")
  apply simp
  apply (rule_tac T = "atLeast 0" in leadsTo_Join)
    apply (rule leadsTo_weaken_L [OF FF_leadsTo], simp)
    apply (simp add: awp_iff_constrains FF_def, safety)
  apply (simp add: awp_iff_constrains GG_def wens_set_FF, safety)
done

end

```

## 24 Common Declarations for Chandy and Charpentier's Allocator

```
theory AllocBase imports "../UNITY_Main" "HOL-Library.Multiset_Order" begin
```

```
consts Nclients :: nat
```

```
axiomatization NbT :: nat
  where NbT_pos: "0 < NbT"
```

```
abbreviation (input) tokens :: "nat list  $\Rightarrow$  nat"
where
  "tokens  $\equiv$  sum_list"
```

```
abbreviation (input)
  "bag_of  $\equiv$  mset"
```

```
lemma sum_fun_mono:
  fixes f :: "nat  $\Rightarrow$  nat"
  shows " $(\bigwedge i. i < n \implies f\ i \leq g\ i) \implies \text{sum } f\ \{.."
  by (induct n) (auto simp add: lessThan_Suc add_le_mono)$ 
```

```
lemma tokens_mono_prefix: "xs  $\leq$  ys  $\implies$  tokens xs  $\leq$  tokens ys"
  by (induct ys arbitrary: xs) (auto simp add: prefix_Cons)
```

```
lemma mono_tokens: "mono tokens"
  using tokens_mono_prefix by (rule monoI)
```

```
lemma bag_of_append [simp]: "bag_of (l@l') = bag_of l + bag_of l'"
  by (fact mset_append)
```

```
lemma mono_bag_of: "mono (bag_of :: 'a list  $\Rightarrow$  ('a::order) multiset)"
apply (rule monoI)
apply (unfold prefix_def)
apply (erule genPrefix.induct, simp_all add: add_right_mono)
apply (erule order_trans)
apply simp
done
```

```
declare sum.cong [cong]
```

```
lemma bag_of_nthn_lemma:
  " $(\sum_{i \in A} \text{Int lessThan } k. \{\# \text{if } i < k \text{ then } f\ i \text{ else } g\ i\}) =$ "
  " $(\sum_{i \in A} \text{Int lessThan } k. \{\# f\ i\})$ "
by (rule sum.cong, auto)
```

```
lemma bag_of_nthn:
  "bag_of (nthn l A) =
```

```

    ( $\sum i \in A \text{ Int lessThan } (\text{length } l). \{ \# l!i \# \})"$ 
  by (rule_tac xs = l in rev_induct)
    (simp_all add: nth_append Int_insert_right lessThan_Suc nth_append
      bag_of_nth lemma ac_simps)

lemma bag_of_nth_Un_Int:
  "bag_of (nth 1 (A Un B)) + bag_of (nth 1 (A Int B)) =
   bag_of (nth 1 A) + bag_of (nth 1 B)"
apply (subgoal_tac "A Int B Int {..\forall i \in I. \forall j \in I. i \neq j \longrightarrow A \ i \text{ Int } A \ j = \{ \} |]
   ==> bag_of (nth 1 ( $\bigcup (A \ ' I)$ )) =
    ( $\sum i \in I. \text{bag\_of } (\text{nth } 1 (A \ i))$ )"
apply (auto simp add: bag_of_nth)
unfolding UN_simps [symmetric]
apply (subst sum.UNION_disjoint)
apply auto
done

end

theory Alloc
imports AllocBase "../PPROD"
begin

```

## 24.1 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
```

**24.1.1 Resource allocation system specification****definition**

— spec (1)

```
system_safety :: "'a systemState program set"
where "system_safety =
  Always {s. ( $\sum i \in \text{lessThan } Nclients. (\text{tokens } o \text{ giv } o \text{ sub } i \ o \text{ client})s$ )
     $\leq NbT + (\sum i \in \text{lessThan } Nclients. (\text{tokens } o \text{ rel } o \text{ sub } i \ o \text{ client})s$ )}"
```

**definition**

— spec (2)

```
system_progress :: "'a systemState program set"
where "system_progress = (INT i : lessThan Nclients.
  INT h.
    {s. h  $\leq (\text{ask } o \text{ sub } i \ o \text{ client})s$ } 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"
```

**24.1.2 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.  $\forall elt \in \text{set } (\text{ask } s).
elt \leq NbT$ }"
```



**definition**

```

— spec (5)
client_progress :: "'a clientState_d program set"
where "client_progress =
      Increasing giv guarantees
      (INT h. {s. h ≤ giv s & h prefixGe 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 (⋃ (Acts ' preserves (funPair rel ask)))}"
```

**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"
```

**24.1.3 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. (∑ i ∈ lessThan Nclients. (tokens o sub i o allocGiv)s)
        ≤ NbT + (∑ 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. ∀ i < Nclients.
        ∀ 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 (⋃ (Acts ' (preserves allocGiv)))}"

```

**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"

```

**24.1.4 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
    (⋃ (Acts ' (preserves allocRel ∩ (⋂ i<Nclients.
      preserves (giv o sub i o client))))}"

```

**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"

```

**24.1.5 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 ⊔ Network ⊔"

```

(rename sysOfClient
  (plam x: lessThan Nclients. rename client_map Client)))"

declare subset_preserves_o [THEN [2] rev_subsetD, intro]
declare subset_preserves_o [THEN [2] rev_subsetD, simp]
declare funPair_o_distrib [simp]
declare Always_INT_distrib [simp]
declare o_apply [simp del]

lemmas [simp] =
  rename_image_constrains
  rename_image_stable
  rename_image_increasing
  rename_image_invariant
  rename_image_Constrains
  rename_image_Stable
  rename_image_Increasing
  rename_image_Always
  rename_image_leadsTo
  rename_image_LeadsTo
  rename_preserves
  rename_image_preserves
  lift_image_preserves
  bij_image_INT
  bij_is_inj [THEN image_Int]
  bij_image_Collect_eq

ML <
(*Splits up conjunctions & intersections: like CONJUNCTS in the HOL system*)
fun list_of_Int th =
  (list_of_Int (th RS conjunct1) @ list_of_Int (th RS conjunct2))
  handle THM _ => (list_of_Int (th RS @{thm IntD1}) @ list_of_Int (th RS
    @{thm IntD2}))
  handle THM _ => (list_of_Int (th RS @{thm INT_D}))
  handle THM _ => (list_of_Int (th RS @{thm bspec}))
  handle THM _ => [th];
>

lemmas lessThanBspec = lessThan_iff [THEN iffD2, THEN [2] bspec]

attribute_setup normalized = <
let
  fun normalized th =
    normalized (th RS spec
      handle THM _ => th RS @{thm lessThanBspec}
      handle THM _ => th RS @{thm bspec}
      handle THM _ => th RS (@{thm guarantees_INT_right_iff} RS iffD1))
    handle THM _ => th;
in
  Scan.succeed (Thm.rule_attribute [] (K normalized))
end

```

```
end
>
```

```
ML <
fun record_auto_tac ctxt =
  let val ctxt' =
    ctxt addSWrapper Record.split_wrapper
    addsimps
      [{thm sysOfAlloc_def}, @{thm sysOfClient_def},
       @{thm client_map_def}, @{thm non_dummy_def}, @{thm funPair_def},
       @{thm o_apply}, @{thm Let_def}]
  in auto_tac ctxt' end;

>

method_setup record_auto = <Scan.succeed (SIMPLE_METHOD o record_auto_tac)>

lemma inj_sysOfAlloc [iff]: "inj sysOfAlloc"
  apply (unfold sysOfAlloc_def Let_def)
  apply (rule inj_onI)
  apply record_auto
  done
```

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) |)"
  apply (rule inj_sysOfAlloc [THEN inv_f_eq])
  apply record_auto
  done

lemma surj_sysOfAlloc [iff]: "surj sysOfAlloc"
  apply (simp add: surj_iff_all)
  apply record_auto
  done

lemma bij_sysOfAlloc [iff]: "bij sysOfAlloc"
  apply (blast intro: bijI)
  done
```

#### 24.1.6 bijectivity of sysOfClient

```
lemma inj_sysOfClient [iff]: "inj sysOfClient"
  apply (unfold sysOfClient_def)
  apply (rule inj_onI)
  apply record_auto
  done

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|) )"
  apply (rule inj_sysOfClient [THEN inv_f_eq])
  apply record_auto
  done

```

```

lemma surj_sysOfClient [iff]: "surj sysOfClient"
  apply (simp add: surj_iff_all)
  apply record_auto
  done

```

```

lemma bij_sysOfClient [iff]: "bij sysOfClient"
  apply (blast intro: bijI)
  done

```

#### 24.1.7 bijectivity of client\_map

```

lemma inj_client_map [iff]: "inj client_map"
  apply (unfold inj_on_def)
  apply record_auto
  done

```

```

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"
  apply (rule inj_client_map [THEN inv_f_eq])
  apply record_auto
  done

```

```

lemma surj_client_map [iff]: "surj client_map"
  apply (simp add: surj_iff_all)
  apply record_auto
  done

```

```

lemma bij_client_map [iff]: "bij client_map"
  apply (blast intro: bijI)
  done

```

o-simprules for client\_map

```

lemma fst_o_client_map: "fst o client_map = non_dummy"
  apply (unfold client_map_def)
  apply (rule fst_o_funPair)
  done

```

```

ML <ML_Thms.bind_thms ("fst_o_client_map'", make_o_equivs context @{thm fst_o_client_map})>
declare fst_o_client_map' [simp]

```

```

lemma snd_o_client_map: "snd o client_map = clientState_d.dummy"
  apply (unfold client_map_def)
  apply (rule snd_o_funPair)
  done

```

```

ML <ML_Thms.bind_thms ("snd_o_client_map'", make_o_equivs context @{thm snd_o_client_map})>

```

```
declare snd_o_client_map' [simp]
```

## 24.2 o-simprules for *sysOfAlloc* [MUST BE AUTOMATED]

```
lemma client_o_sysOfAlloc: "client o sysOfAlloc = fst o allocState_d.dummy"
"
  apply record_auto
  done
```

```
ML <ML_Thms.bind_thms ("client_o_sysOfAlloc'", make_o_equivs context @{thm
client_o_sysOfAlloc})>
declare client_o_sysOfAlloc' [simp]
```

```
lemma allocGiv_o_sysOfAlloc_eq: "allocGiv o sysOfAlloc = allocGiv"
  apply record_auto
  done
```

```
ML <ML_Thms.bind_thms ("allocGiv_o_sysOfAlloc_eq'", make_o_equivs context
@{thm allocGiv_o_sysOfAlloc_eq})>
declare allocGiv_o_sysOfAlloc_eq' [simp]
```

```
lemma allocAsk_o_sysOfAlloc_eq: "allocAsk o sysOfAlloc = allocAsk"
  apply record_auto
  done
```

```
ML <ML_Thms.bind_thms ("allocAsk_o_sysOfAlloc_eq'", make_o_equivs context
@{thm allocAsk_o_sysOfAlloc_eq})>
declare allocAsk_o_sysOfAlloc_eq' [simp]
```

```
lemma allocRel_o_sysOfAlloc_eq: "allocRel o sysOfAlloc = allocRel"
  apply record_auto
  done
```

```
ML <ML_Thms.bind_thms ("allocRel_o_sysOfAlloc_eq'", make_o_equivs context
@{thm allocRel_o_sysOfAlloc_eq})>
declare allocRel_o_sysOfAlloc_eq' [simp]
```

## 24.3 o-simprules for *sysOfClient* [MUST BE AUTOMATED]

```
lemma client_o_sysOfClient: "client o sysOfClient = fst"
  apply record_auto
  done
```

```
ML <ML_Thms.bind_thms ("client_o_sysOfClient'", make_o_equivs context @{thm
client_o_sysOfClient})>
declare client_o_sysOfClient' [simp]
```

```
lemma allocGiv_o_sysOfClient_eq: "allocGiv o sysOfClient = allocGiv o snd"
"
  apply record_auto
  done
```

```
ML <ML_Thms.bind_thms ("allocGiv_o_sysOfClient_eq'", make_o_equivs context
@{thm allocGiv_o_sysOfClient_eq})>
```

```

declare allocGiv_o_sysOfClient_eq' [simp]

lemma allocAsk_o_sysOfClient_eq: "allocAsk o sysOfClient = allocAsk o snd
"
  apply record_auto
  done

ML <ML_Thms.bind_thms ("allocAsk_o_sysOfClient_eq'", make_o_equivs context
@{thm allocAsk_o_sysOfClient_eq})>
declare allocAsk_o_sysOfClient_eq' [simp]

lemma allocRel_o_sysOfClient_eq: "allocRel o sysOfClient = allocRel o snd
"
  apply record_auto
  done

ML <ML_Thms.bind_thms ("allocRel_o_sysOfClient_eq'", make_o_equivs context
@{thm allocRel_o_sysOfClient_eq})>
declare allocRel_o_sysOfClient_eq' [simp]

lemma allocGiv_o_inv_sysOfAlloc_eq: "allocGiv o inv sysOfAlloc = allocGiv"
  apply (simp add: o_def)
  done

ML <ML_Thms.bind_thms ("allocGiv_o_inv_sysOfAlloc_eq'", make_o_equivs con-
text @{thm allocGiv_o_inv_sysOfAlloc_eq})>
declare allocGiv_o_inv_sysOfAlloc_eq' [simp]

lemma allocAsk_o_inv_sysOfAlloc_eq: "allocAsk o inv sysOfAlloc = allocAsk"
  apply (simp add: o_def)
  done

ML <ML_Thms.bind_thms ("allocAsk_o_inv_sysOfAlloc_eq'", make_o_equivs con-
text @{thm allocAsk_o_inv_sysOfAlloc_eq})>
declare allocAsk_o_inv_sysOfAlloc_eq' [simp]

lemma allocRel_o_inv_sysOfAlloc_eq: "allocRel o inv sysOfAlloc = allocRel"
  apply (simp add: o_def)
  done

ML <ML_Thms.bind_thms ("allocRel_o_inv_sysOfAlloc_eq'", make_o_equivs con-
text @{thm allocRel_o_inv_sysOfAlloc_eq})>
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"
  apply (simp add: o_def drop_map_def)
  done

ML <ML_Thms.bind_thms ("rel_inv_client_map_drop_map'", make_o_equivs con-
text @{thm rel_inv_client_map_drop_map})>
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"
  apply (simp add: o_def drop_map_def)
done

```

```

ML <ML_Thms.bind_thms ("ask_inv_client_map_drop_map'", make_o_equivs con-
text @{thm ask_inv_client_map_drop_map})>
declare ask_inv_client_map_drop_map [simp]

```

Client : <unfolded specification>

```

lemmas client_spec_simps =
  client_spec_def client_increasing_def client_bounded_def
  client_progress_def client_allowed_acts_def client_preserves_def
  guarantees_Int_right

```

```

ML <
val [Client_Increasing_ask, Client_Increasing_rel,
  Client_Bounded, Client_Progress, Client_AllowedActs,
  Client_preserves_giv, Client_preserves_dummy] =
  @{thm Client} |> simplify (context addsimps @{thms client_spec_simps})
  |> list_of_Int;

```

```

ML_Thms.bind_thm ("Client_Increasing_ask", Client_Increasing_ask);
ML_Thms.bind_thm ("Client_Increasing_rel", Client_Increasing_rel);
ML_Thms.bind_thm ("Client_Bounded", Client_Bounded);
ML_Thms.bind_thm ("Client_Progress", Client_Progress);
ML_Thms.bind_thm ("Client_AllowedActs", Client_AllowedActs);
ML_Thms.bind_thm ("Client_preserves_giv", Client_preserves_giv);
ML_Thms.bind_thm ("Client_preserves_dummy", Client_preserves_dummy);
>

```

```

declare
  Client_Increasing_ask [iff]
  Client_Increasing_rel [iff]
  Client_Bounded [iff]
  Client_preserves_giv [iff]
  Client_preserves_dummy [iff]

```

Network : <unfolded specification>

```

lemmas network_spec_simps =
  network_spec_def network_ask_def network_giv_def
  network_rel_def network_allowed_acts_def network_preserves_def
  ball_conj_distrib

```

```

ML <
val [Network_Ask, Network_Giv, Network_Rel, Network_AllowedActs,
  Network_preserves_allocGiv, Network_preserves_rel,
  Network_preserves_ask] =
  @{thm Network} |> simplify (context addsimps @{thms network_spec_simps})
  |> list_of_Int;

```

```

ML_Thms.bind_thm ("Network_Ask", Network_Ask);
ML_Thms.bind_thm ("Network_Giv", Network_Giv);

```

```

ML_Thms.bind_thm ("Network_Rel", Network_Rel);
ML_Thms.bind_thm ("Network_AllowedActs", Network_AllowedActs);
ML_Thms.bind_thm ("Network_preserves_allocGiv", Network_preserves_allocGiv);
ML_Thms.bind_thm ("Network_preserves_rel", Network_preserves_rel);
ML_Thms.bind_thm ("Network_preserves_ask", Network_preserves_ask);
>

```

```

declare Network_preserves_allocGiv [iff]

```

```

declare
  Network_preserves_rel [simp]
  Network_preserves_ask [simp]

```

```

declare
  Network_preserves_rel [simplified o_def, simp]
  Network_preserves_ask [simplified o_def, simp]

```

```

Alloc : <unfolded specification>

```

```

lemmas alloc_spec_simps =
  alloc_spec_def alloc_increasing_def alloc_safety_def
  alloc_progress_def alloc_allowed_acts_def alloc_preserves_def

```

```

ML <
val [Alloc_Increasing_0, Alloc_Safety, Alloc_Progress, Alloc_AllowedActs,
    Alloc_preserves_allocRel, Alloc_preserves_allocAsk,
    Alloc_preserves_dummy] =
  @{thm Alloc} |> simplify (context addsimps @{thms alloc_spec_simps})
  |> list_of_Int;

```

```

ML_Thms.bind_thm ("Alloc_Increasing_0", Alloc_Increasing_0);
ML_Thms.bind_thm ("Alloc_Safety", Alloc_Safety);
ML_Thms.bind_thm ("Alloc_Progress", Alloc_Progress);
ML_Thms.bind_thm ("Alloc_AllowedActs", Alloc_AllowedActs);
ML_Thms.bind_thm ("Alloc_preserves_allocRel", Alloc_preserves_allocRel);
ML_Thms.bind_thm ("Alloc_preserves_allocAsk", Alloc_preserves_allocAsk);
ML_Thms.bind_thm ("Alloc_preserves_dummy", Alloc_preserves_dummy);
>

```

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]

```

## 24.4 Components Lemmas [MUST BE AUTOMATED]

```

lemma Network_component_System: "Network  $\sqsubseteq$ 
  ((rename sysOfClient
    (plam x: (lessThan Nclients). rename client_map Client))  $\sqsubseteq$ 
    rename sysOfAlloc Alloc)
  = System"
by (simp add: System_def Join_ac)

```

```

lemma Client_component_System: "(rename sysOfClient
  (plam x: (lessThan Nclients). rename client_map Client)) ⊑
  (Network ⊑ rename sysOfAlloc Alloc) = System"
  by (simp add: System_def Join_ac)

lemma Alloc_component_System: "rename sysOfAlloc Alloc ⊑
  ((rename sysOfClient (plam x: (lessThan Nclients). rename client_map
  Client)) ⊑
  Network) = System"
  by (simp add: System_def Join_ac)

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"
  by (auto simp add: Allowed_def Client_AllowedActs safety_prop_Acts_iff)

lemma Network_Allowed [simp]: "Allowed Network =
  preserves allocRel Int
  (INT i: lessThan Nclients. preserves(giv o sub i o client))"
  by (auto simp add: Allowed_def Network_AllowedActs safety_prop_Acts_iff)

lemma Alloc_Allowed [simp]: "Allowed Alloc = preserves allocGiv"
  by (auto simp add: Allowed_def Alloc_AllowedActs safety_prop_Acts_iff)

needed in rename_client_map_tac

lemma OK_lift_rename_Client [simp]: "OK I (%i. lift i (rename client_map
Client))"
  apply (rule OK_lift_I)
  apply auto
  apply (drule_tac w1 = rel in subset_preserves_o [THEN [2] rev_subsetD])
  apply (drule_tac [2] w1 = ask in subset_preserves_o [THEN [2] rev_subsetD])
  apply (auto simp add: o_def split_def)
  done

lemma fst_lift_map_eq_fst [simp]: "fst (lift_map i x) i = fst x"
  apply (insert fst_o_lift_map [of i])
  apply (drule fun_cong [where x=x])
  apply (simp add: o_def)
  done

lemma fst_o_lift_map' [simp]:
  "(f o sub i o fst o lift_map i o g) = f o fst o g"
  apply (subst fst_o_lift_map [symmetric])
  apply (simp only: o_assoc)
  done

```

ML

```

<
fun rename_client_map_tac ctxt =
  EVERY [
    simp_tac (ctxt addsimps [@{thm rename_guarantees_eq_rename_inv}]) 1,
    resolve_tac ctxt @{thms guarantees_PLam_I} 1,
    assume_tac ctxt 2,
    (*preserves: routine reasoning*)
    asm_simp_tac (ctxt addsimps [@{thm lift_preserves_sub}]) 2,
    (*the guarantee for "lift i (rename client_map Client)" *)
    asm_simp_tac
      (ctxt addsimps [@{thm lift_guarantees_eq_lift_inv},
                     @{thm rename_guarantees_eq_rename_inv},
                     @{thm bij_imp_bij_inv}, @{thm surj_rename},
                     @{thm inv_inv_eq}]) 1,
    asm_simp_tac
      (ctxt addsimps [@{thm o_def}, @{thm non_dummy_def}, @{thm guarantees_Int_right}])
  ]
>

method_setup rename_client_map = <
  Scan.succeed (fn ctxt => SIMPLE_METHOD (rename_client_map_tac ctxt))
>

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)"
  by rename_client_map

lemma preserves_sub_fst_lift_map: "[| F ∈ preserves w; i ≠ j |]
  ==> F ∈ preserves (sub i o fst o lift_map j o funPair v w)"
  apply (auto simp add: lift_map_def split_def linorder_neq_iff o_def)
  apply (drule_tac [!] subset_preserves_o [THEN [2] rev_subsetD])
  apply (auto simp add: o_def)
  done

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)"
  apply (cases "i=j")
  apply (simp, simp add: o_def non_dummy_def)
  apply (drule Client_preserves_dummy [THEN preserves_sub_fst_lift_map])
  apply (drule_tac [!] subset_preserves_o [THEN [2] rev_subsetD])
  apply (simp add: o_def client_map_def)
  done

lemma rename_sysOfClient_ok_Network:
  "rename sysOfClient (plam x: lessThan Nclients. rename client_map Client)
  ok Network"
  by (auto simp add: ok_iff_Allowed client_preserves_giv_oo_client_map)

```

```

lemma rename_sysOfClient_ok_Alloc:
  "rename sysOfClient (plam x: lessThan Nclients. rename client_map Client)
   ok rename sysOfAlloc Alloc"
  by (simp add: ok_iff_Allowed)

```

```

lemma rename_sysOfAlloc_ok_Network: "rename sysOfAlloc Alloc ok Network"
  by (simp add: ok_iff_Allowed)

```

```

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)"
  apply (rule component_guaranteesD [OF rename_Client_Increasing Client_component_System])
  apply auto
  done

```

```

lemmas rename_guarantees_sysOfAlloc_I =
  bij_sysOfAlloc [THEN rename_rename_guarantees_eq, THEN iffD2]

```

```

lemmas rename_Alloc_Increasing =
  Alloc_Increasing
  [THEN rename_guarantees_sysOfAlloc_I,
   simplified surj_rename 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)"
  apply (unfold System_def)
  apply (simp add: o_def)
  apply (rule rename_Alloc_Increasing [THEN guarantees_Join_I1, THEN guaranteesD])
  apply auto
  done

```

```

ML <
ML_Thms.bind_thms ("System_Increasing'", list_of_Int @ {thm System_Increasing})
>

```

```

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))"
  apply (auto intro!: Network_Rel [THEN component_guaranteesD])
  apply (simp add: ok_commute [of Network])
  done
```

```
lemma System_Follows_ask:
  "i < Nclients ==> System ∈ ((sub i o allocAsk) Fols (ask o sub i o client))"
  apply (auto intro!: Network_Ask [THEN component_guaranteesD])
  apply (simp add: ok_commute [of Network])
  done
```

```
lemma System_Follows_allocGiv:
  "i < Nclients ==> System ∈ (giv o sub i o client) Fols (sub i o allocGiv)"
  apply (auto intro!: Network_Giv [THEN component_guaranteesD])
  rename Alloc_Increasing [THEN component_guaranteesD])
  apply (simp_all add: o_def non_dummy_def ok_commute [of Network])
  apply (auto intro!: rename_Alloc_Increasing [THEN component_guaranteesD])
  done
```

```
lemma Always_giv_le_allocGiv: "System ∈ Always (INT i: lessThan Nclients.
  {s. (giv o sub i o client) s ≤ (sub i o allocGiv) s})"
  apply auto
  apply (erule System_Follows_allocGiv [THEN Follows_Bounded])
  done
```

```
lemma Always_allocAsk_le_ask: "System ∈ Always (INT i: lessThan Nclients.
  {s. (sub i o allocAsk) s ≤ (ask o sub i o client) s})"
  apply auto
  apply (erule System_Follows_ask [THEN Follows_Bounded])
  done
```

```
lemma Always_allocRel_le_rel: "System ∈ Always (INT i: lessThan Nclients.
  {s. (sub i o allocRel) s ≤ (rel o sub i o client) s})"
  by (auto intro!: Follows_Bounded System_Follows_rel)
```

## 24.5 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]
```

safety (1), step 3

```
lemma System_sum_bounded:
  "System ∈ Always {s. (∑ i ∈ lessThan Nclients. (tokens o sub i o allocGiv)
  s)
```

```

      ≤ NbT + (∑ i ∈ lessThan Nclients. (tokens o sub i o allocRel)
s)})"
  apply (simp add: o_apply)
  apply (insert Alloc_Safety [THEN rename_guarantees_sysOfAlloc_I])
  apply (simp add: o_def)
  apply (erule component_guaranteesD)
  apply (auto simp add: System_Increasing_allocRel [simplified sub_apply o_def])
done

```

Follows reasoning

```

lemma Always_tokens_giv_le_allocGiv: "System ∈ Always (INT i: lessThan Nclients.
      {s. (tokens o giv o sub i o client) s
      ≤ (tokens o sub i o allocGiv) s})"
  apply (rule Always_giv_le_allocGiv [THEN Always_weaken])
  apply (auto intro: tokens_mono_prefix simp add: o_apply)
done

```

```

lemma Always_tokens_allocRel_le_rel: "System ∈ Always (INT i: lessThan Nclients.
      {s. (tokens o sub i o allocRel) s
      ≤ (tokens o rel o sub i o client) s})"
  apply (rule Always_allocRel_le_rel [THEN Always_weaken])
  apply (auto intro: tokens_mono_prefix simp add: o_apply)
done

```

safety (1), step 4 (final result!)

```

theorem System_safety: "System ∈ system_safety"
  apply (unfold system_safety_def)
  apply (tactic <resolve_tac context [Always_Int_rule [@{thm System_sum_bounded},
    @{thm Always_tokens_giv_le_allocGiv}, @{thm Always_tokens_allocRel_le_rel}]
RS
    @{thm Always_weaken}] 1>)
  apply auto
  apply (rule sum_fun_mono [THEN order_trans])
  apply (drule_tac [2] order_trans)
  apply (rule_tac [2] add_le_mono [OF order_refl sum_fun_mono])
  prefer 3 apply assumption
  apply auto
done

```

## 24.6 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*]

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. ∀ elt ∈ set ((ask o sub i o client) s). elt ≤ NbT}"
using image_cong_simp [cong del] by rename_client_map

```

```

lemma System_Bounded_ask: "i < Nclients
  ==> System ∈ Always
    {s. ∀elt ∈ set ((ask o sub i o client) s). elt ≤ NbT}"
  apply (rule component_guaranteesD [OF rename_Client_Bounded Client_component_System])
  apply auto
  done

lemma Collect_all_imp_eq: "{x. ∀y. P y → Q x y} = (INT y: {y. P y}. {x.
  Q x y})"
  apply blast
  done

progress (2), step 4

lemma System_Bounded_allocAsk: "System ∈ Always {s. ∀i<Nclients.
  ∀elt ∈ set ((sub i o allocAsk) s). elt ≤ NbT}"
  apply (auto simp add: Collect_all_imp_eq)
  apply (tactic <resolve_tac context [Always_Int_rule [@{thm Always_allocAsk_le_ask},
    @{thm System_Bounded_ask}] RS @{thm Always_weaken}] 1>)
  apply (auto dest: set_mono)
  done

progress (2), step 5 is System_Increasing_allocGiv

progress (2), step 6

lemmas System_Increasing_giv = System_Follows_allocGiv [THEN Follows_Increasing1]

lemma rename_Client_Progress: "i ∈ I
  ==> rename sysOfClient (plam x: I. rename client_map Client)
    ∈ Increasing (giv o sub i o client)
      guarantees
        (INT h. {s. h ≤ (giv o sub i o client) s &
          h pfixGe (ask o sub i o client) s}
          LeadsTo {s. tokens h ≤ (tokens o rel o sub i o client) s})"
  supply image_cong_simp [cong del]
  apply rename_client_map
  apply (simp add: Client_Progress [simplified o_def])
  done

progress (2), step 7

lemma System_Client_Progress:
  "System ∈ (INT i : (lessThan Nclients).
    INT h. {s. h ≤ (giv o sub i o client) s &
      h pfixGe (ask o sub i o client) s}
      LeadsTo {s. tokens h ≤ (tokens o rel o sub i o client) s})"
  apply (rule INT_I)

  apply (rule component_guaranteesD [OF rename_Client_Progress Client_component_System])
  apply (auto simp add: System_Increasing_giv)
  done

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}"
  apply (rule single_LeadsTo_I)
  apply (rule_tac k1 = h and x1 = "(sub i o allocAsk) s"
        in System_lemma2 [THEN LeadsTo_weaken])
  apply auto
  apply (blast intro: trans_Ge [THEN trans_genPrefix, THEN transD] prefix_imp_prefixGe)
done

```

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}"
  apply (rule LeadsTo_Trans)
  prefer 2
  apply (drule System_Follows_rel [THEN
    mono_tokens [THEN mono_Follows_o, THEN [2] rev_subsetD],
    THEN Follows_LeadsTo])
  apply (simp add: o_assoc)
  apply (rule LeadsTo_Trans)
  apply (cut_tac [2] System_Client_Progress)
  prefer 2
  apply (blast intro: LeadsTo_Basis)
  apply (erule System_lemma3)
done

```

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 prefixLe (sub i o allocGiv) s})"
  apply (simp only: o_apply sub_def)
  apply (insert Alloc_Progress [THEN rename_guarantees_sysOfAlloc_I])
  apply (simp add: o_def del: INT_iff)
  apply (drule component_guaranteesD)
  apply (auto simp add:
    System_Increasing_allocRel [simplified sub_apply o_def]
    System_Increasing_allocAsk [simplified sub_apply o_def]
    System_Bounded_allocAsk [simplified sub_apply o_def])

```

```

      System_Alloc_Client_Progress [simplified sub_apply o_def])
done

```

progress (2), step 10 (final result!)

```

lemma System_Progress: "System ∈ system_progress"
  apply (unfold system_progress_def)
  apply (cut_tac System_Alloc_Progress)
  apply auto
  apply (blast intro: LeadsTo_Trans
    System_Follows_allocGiv [THEN Follows_LeadsTo_pfixLe]
    System_Follows_ask [THEN Follows_LeadsTo])
done

```

```

theorem System_correct: "System ∈ system_spec"
  apply (unfold system_spec_def)
  apply (blast intro: System_safety System_Progress)
done

```

Some obsolete lemmas

```

lemma non_dummy_eq_o_funPair: "non_dummy = (% (g,a,r). (| giv = g, ask =
a, rel = r |)) o
                                (funPair giv (funPair ask rel))"
  apply (rule ext)
  apply (auto simp add: o_def non_dummy_def)
done

lemma preserves_non_dummy_eq: "(preserves non_dummy) =
  (preserves rel Int preserves ask Int preserves giv)"
  apply (simp add: non_dummy_eq_o_funPair)
  apply auto
  apply (drule_tac w1 = rel in subset_preserves_o [THEN [2] rev_subsetD])
  apply (drule_tac [2] w1 = ask in subset_preserves_o [THEN [2] rev_subsetD])
  apply (drule_tac [3] w1 = giv in subset_preserves_o [THEN [2] rev_subsetD])
  apply (auto simp add: o_def)
done

```

Could go to Extend.ML

```

lemma bij_fst_inv_inv_eq: "bij f ⇒ fst (inv (%(x, u). inv f x) z) = f z"
  apply (rule fst_inv_equalityI)
  apply (rule_tac f = "%z. (f z, h z)" for h in surjI)
  apply (simp add: bij_is_inj inv_f_f)
  apply (simp add: bij_is_surj surj_f_inv_f)
done

```

end

## 25 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 = (|In = In s, Out = Out s, iOut = iOut s|)"

record 'b distr =
  In   :: "'b list"
  iIn  :: "nat list"
  Out  :: "nat => 'b list"

record ('a,'b) distr_d =
  "'b distr" +
  dummy :: 'a

record allocState =
  giv :: "nat list"
  ask  :: "nat list"
  rel  :: "nat list"

record 'a allocState_d =
  allocState +
  dummy      :: 'a

record 'a systemState =
  allocState +
  mergeRel  :: "nat merge"
  mergeAsk  :: "nat merge"
  distr     :: "nat distr"
  dummy     :: 'a

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 =
  ( $\bigcap i \in \text{lessThan Nclients. Increasing } (\text{sub } i \text{ o merge.In})$ )
  guarantees
  ( $\bigcap i \in \text{lessThan Nclients.}$ 
    (%s. nth s (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 ( $\bigcup (\text{Acts } ' \text{preserves } (\text{funPair merge.Out iOut}))$ )}"
```

**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
  ( $\bigcap i \in \text{lessThan Nclients.}$ 
    (sub i o distr.Out) Fols
    (%s. nth s (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 ( $\bigcup$  (Acts ' (preserves distr.Out)))}"

```

**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)  $\leq$  NbT + tokens (rel s)}"

```

**definition**

```

alloc_progress :: "'a allocState_d program set"
where "alloc_progress =
      Increasing ask Int Increasing rel Int
      Always {s.  $\forall$  elt  $\in$  set (ask s). elt  $\leq$  NbT}
      Int
      ( $\bigcap$  h. {s. h  $\leq$  giv s & h pfixGe (ask s)}
      LeadsTo
      {s. tokens h  $\leq$  tokens (rel s)})
      guarantees ( $\bigcap$  h. {s. h  $\leq$  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 ( $\bigcup$  (Acts ' (preserves giv)))}"

```

**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]

```

## 25.1 Theorems for Merge

```

context Merge
begin

```

```

lemma Merge_Allowed:
  "Allowed M = (preserves merge.Out) Int (preserves merge.iOut)"
apply (cut_tac Merge_spec)
apply (auto simp add: merge_spec_def merge_allowed_acts_def Allowed_def
  safety_prop_Acts_iff)
done

```

```

lemma M_ok_iff [iff]:
  "M ok G = (G ∈ preserves merge.Out & G ∈ preserves merge.iOut &
    M ∈ Allowed G)"
by (auto simp add: Merge_Allowed ok_iff_Allowed)

```

```

lemma Merge_Always_Out_eq_iOut:
  "[| G ∈ preserves merge.Out; G ∈ preserves merge.iOut; M ∈ Allowed G
  |]
  ==> M ⊔ G ∈ Always {s. length (merge.Out s) = length (merge.iOut s)}"
apply (cut_tac Merge_spec)
apply (force dest: guaranteesD simp add: merge_spec_def merge_eqOut_def)
done

```

```

lemma Merge_Bounded:
  "[| G ∈ preserves merge.iOut; G ∈ preserves merge.Out; M ∈ Allowed G
  |]
  ==> M ⊔ G ∈ Always {s. ∀elt ∈ set (merge.iOut s). elt < Nclients}"
apply (cut_tac Merge_spec)
apply (force dest: guaranteesD simp add: merge_spec_def merge_bounded_def)
done

```

```

lemma Merge_Bag_Follows_lemma:
  "[| G ∈ preserves merge.iOut; G ∈ preserves merge.Out; M ∈ Allowed G

```

```

/]
==> M ⊔ G ∈ Always
      {s. (∑ i ∈ lessThan Nclients. bag_of (nth (merge.Out s)
                                             {k. k < length (iOut s) & iOut s ! k = i}))
=
      (bag_of o merge.Out) s}"
apply (rule Always_Compl_Un_eq [THEN iffD1])
apply (blast intro: Always_Int_I [OF Merge_Always_Out_eq_iOut Merge_Bounded])
apply (rule UNIV_AlwaysI, clarify)
apply (subst bag_of_nth UN_disjoint [symmetric])
  apply (simp)
  apply blast
apply (simp add: set_conv_nth)
apply (subgoal_tac
      "(⋃ i ∈ lessThan Nclients. {k. k < length (iOut x) & iOut x ! k = i})
=
      lessThan (length (iOut x))")
  apply (simp (no_asm_simp) add: o_def)
  apply blast
done

lemma 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)"
apply (rule Merge_Bag_Follows_lemma [THEN Always_Follows1, THEN guaranteesI],
auto)
apply (rule Follows_sum)
apply (cut_tac Merge_spec)
apply (auto simp add: merge_spec_def merge_follows_def o_def)
apply (drule guaranteesD)
  prefer 3
  apply (best intro: mono_bag_of [THEN mono_Follows_apply, THEN subsetD],
auto)
done

end

```

## 25.2 Theorems for Distributor

```

context Distrib
begin

```

```

lemma 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))"
apply (cut_tac Distrib_spec)
apply (simp add: distr_spec_def distr_follows_def)
apply clarify
apply (blast intro: guaranteesI Follows_Increasing1 dest: guaranteesD)
done

```

```

lemma Distr_Bag_Follows_lemma:
  "[/ G ∈ preserves distr.Out;
   D ⊔ G ∈ Always {s. ∀elt ∈ set (distr.iIn s). elt < Nclients} /]
  ==> D ⊔ G ∈ Always
    {s. (∑ i ∈ lessThan Nclients. bag_of (nth (distr.In s)
                                              {k. k < length (iIn s) & iIn s ! k = i}))
   =
    bag_of (nth (distr.In s) (lessThan (length (iIn s))))}"
apply (erule Always_Compl_Un_eq [THEN iffD1])
apply (rule UNIV_AlwaysI, clarify)
apply (subst bag_of_nth_UN_disjoint [symmetric])
  apply (simp (no_asm))
  apply blast
apply (simp add: set_conv_nth)
apply (subgoal_tac
  "(⋂ i ∈ lessThan Nclients. {k. k < length (iIn x) & iIn x ! k = i})
  =
  lessThan (length (iIn x))")
  apply (simp (no_asm_simp))
  apply blast
done

lemma D_ok_iff [iff]:
  "D ok G = (G ∈ preserves distr.Out & D ∈ Allowed G)"
apply (cut_tac Distrib_spec)
apply (auto simp add: distr_spec_def distr_allowedActs_def Allowed_def
  safety_prop_Acts_iff ok_iff_Allowed)
done

lemma 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 (nth (distr.In s) (lessThan (length (distr.iIn s)))))"
apply (rule guaranteesI, clarify)
apply (rule Distr_Bag_Follows_lemma [THEN Always_Follows2], auto)
apply (rule Follows_sum)
apply (cut_tac Distrib_spec)
apply (auto simp add: distr_spec_def distr_follows_def o_def)
apply (drule guaranteesD)
  prefer 3
  apply (best intro: mono_bag_of [THEN mono_Follows_apply, THEN subsetD],
  auto)
done

end

```

### 25.3 Theorems for Allocator

```

lemma alloc_refinement_lemma:

```



```

      "!!f::nat=>nat. ( $\bigcap i \in \text{lessThan } n. \{s. f \ i \leq g \ i \ s\}$ )
       $\subseteq \{s. (\sum x \in \text{lessThan } n. f \ x) \leq (\sum x \in \text{lessThan } n. g \ x \ s)\}$ "
    apply (induct_tac "n")
    apply (auto simp add: lessThan_Suc)
  done

lemma alloc_refinement:
  "( $\bigcap i \in \text{lessThan } Nclients. \text{Increasing } (\text{sub } i \ o \ \text{allocAsk}) \ \text{Int}$ 
     $\text{Increasing } (\text{sub } i \ o \ \text{allocRel})$ )
   Int
   Always  $\{s. \forall i. i < Nclients \ \rightarrow$ 
      $(\forall elt \in \text{set } ((\text{sub } i \ o \ \text{allocAsk}) \ s). \text{elt} \leq NbT)\}$ 
   Int
   ( $\bigcap i \in \text{lessThan } Nclients.$ 
     ( $\bigcap h. \{s. h \leq (\text{sub } i \ o \ \text{allocGiv})s \ \& \ h \ \text{pfixGe } (\text{sub } i \ o \ \text{allocAsk})s\}$ 
       LeadsTo  $\{s. \text{tokens } h \leq (\text{tokens } o \ \text{sub } i \ o \ \text{allocRel})s\}$ )
      $\subseteq$ 
     ( $\bigcap i \in \text{lessThan } Nclients. \text{Increasing } (\text{sub } i \ o \ \text{allocAsk}) \ \text{Int}$ 
        $\text{Increasing } (\text{sub } i \ o \ \text{allocRel})$ )
     Int
     Always  $\{s. \forall i. i < Nclients \ \rightarrow$ 
        $(\forall elt \in \text{set } ((\text{sub } i \ o \ \text{allocAsk}) \ s). \text{elt} \leq NbT)\}$ 
     Int
     ( $\bigcap hf. (\bigcap i \in \text{lessThan } Nclients.$ 
        $\{s. hf \ i \leq (\text{sub } i \ o \ \text{allocGiv})s \ \& \ hf \ i \ \text{pfixGe } (\text{sub } i \ o \ \text{allocAsk})s\}$ 
       LeadsTo  $\{s. (\sum i \in \text{lessThan } Nclients. \text{tokens } (hf \ i)) \leq$ 
          $(\sum i \in \text{lessThan } Nclients. (\text{tokens } o \ \text{sub } i \ o \ \text{allocRel})s)\}$ )
     )
  )
  apply (auto simp add: ball_conj_distrib)
  apply (rename_tac F hf)
  apply (rule LeadsTo_weaken_R [OF Finite_stable_completion alloc_refinement_lemma],
    blast, blast)
  apply (subgoal_tac "F  $\in$  Increasing (tokens o (sub i o allocRel))")
  apply (simp add: Increasing_def o_assoc)
  apply (blast intro: mono_tokens [THEN mono_Increasing_o, THEN subsetD])
  done

end

```

## 26 Distributed Resource Management System: the Client

```
theory Client imports "../Rename" AllocBase begin
```

```
type_synonym
```

```
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)"
by (auto simp add: ok_iff_Allowed Client_def [THEN def_total_prg_Allowed])

```

Safety property 1: ask, rel are increasing

```

lemma increasing_ask_rel:
  "Client ∈ UNIV guarantees Increasing ask Int Increasing rel"
apply (auto intro!: increasing_imp_Increasing simp add: guar_def preserves_subset_increasing
[THEN subsetD])
apply (auto simp add: Client_def increasing_def)
apply (safety, auto)+
done

```

```

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 ⊓ G ∈
    Always ({s. tok s ≤ NbT} Int
            {s. ∀ elt ∈ set (ask s). elt ≤ NbT})"
apply auto
apply (rule invariantI [THEN stable_Join_Always2], force)
prefer 2
apply (fast elim!: preserves_subset_stable [THEN subsetD] intro!: stable_Int)

apply (simp add: Client_def, safety)
apply (cut_tac m = "tok s" in NbT_pos [THEN mod_less_divisor], auto)
done

```

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}"
apply (rule guaranteesI)
apply (erule ask_bounded_lemma [THEN Always_weaken])
apply (rule Int_lower2)
done

```

**\*\* Towards proving the liveness property \*\***

```

lemma stable_rel_le_giv: "Client ∈ stable {s. rel s ≤ giv s}"
by (simp add: Client_def, safety, auto)

```

```

lemma Join_Stable_rel_le_giv:
  "[| Client ⊓ G ∈ Increasing giv; G ∈ preserves rel |]
  ==> Client ⊓ G ∈ Stable {s. rel s ≤ giv s}"

```

```

by (rule stable_rel_le_giv [THEN Increasing_preserves_Stable], auto)

lemma Join_Always_rel_le_giv:
  "[| Client  $\sqcup$  G  $\in$  Increasing giv; G  $\in$  preserves rel |]
   ==> Client  $\sqcup$  G  $\in$  Always {s. rel s  $\leq$  giv s}"
by (force intro: AlwaysI Join_Stable_rel_le_giv)

lemma transient_lemma:
  "Client  $\in$  transient {s. rel s = k & k < h & h  $\leq$  giv s & h pfixGe ask s}"
apply (simp add: Client_def mk_total_program_def)
apply (rule_tac act = rel_act in totalize_transientI)
  apply (auto simp add: Domain_unfold Client_def)
  apply (blast intro: less_le_trans prefix_length_le strict_prefix_length_less)
  apply (auto simp add: prefix_def genPrefix_iff_nth Ge_def)
  apply (blast intro: strict_prefix_length_less)
done

lemma induct_lemma:
  "[| Client  $\sqcup$  G  $\in$  Increasing giv; Client ok G |]
   ==> Client  $\sqcup$  G  $\in$  {s. rel s = k & k < h & h  $\leq$  giv s & h pfixGe ask s}
      LeadsTo {s. k < rel s & rel s  $\leq$  giv s &
                h  $\leq$  giv s & h pfixGe ask s}"
apply (rule single_LeadsTo_I)
apply (frule increasing_ask_rel [THEN guaranteesD], auto)
apply (rule transient_lemma [THEN Join_transient_I1, THEN transient_imp_leadsTo,
  THEN leadsTo_imp_LeadsTo, THEN PSP_Stable, THEN LeadsTo_weaken])
  apply (rule Stable_Int [THEN Stable_Int, THEN Stable_Int])
    apply (erule_tac f = giv and x = "giv s" in IncreasingD)
    apply (erule_tac f = ask and x = "ask s" in IncreasingD)
    apply (erule_tac f = rel and x = "rel s" in IncreasingD)
    apply (erule Join_Stable_rel_le_giv, blast)
  apply (blast intro: order_less_imp_le order_trans)
apply (blast intro: sym order_less_le [THEN iffD2] order_trans
  prefix_imp_pfixGe pfixGe_trans)
done

lemma rel_progress_lemma:
  "[| Client  $\sqcup$  G  $\in$  Increasing giv; Client ok G |]
   ==> Client  $\sqcup$  G  $\in$  {s. rel s < h & h  $\leq$  giv s & h pfixGe ask s}
      LeadsTo {s. h  $\leq$  rel s}"
apply (rule_tac f = "%s. size h - size (rel s) " in LessThan_induct)
apply (auto simp add: vimage_def)
apply (rule single_LeadsTo_I)
apply (rule induct_lemma [THEN LeadsTo_weaken], auto)
  apply (blast intro: order_less_le [THEN iffD2] dest: common_prefix_linear)
  apply (drule strict_prefix_length_less)+
apply arith
done

lemma client_progress_lemma:
  "[| Client  $\sqcup$  G  $\in$  Increasing giv; Client ok G |]

```

```

==> Client  $\sqcup$  G  $\in$  {s. h  $\leq$  giv s & h pfixGe ask s}
      LeadsTo {s. h  $\leq$  rel s}"
apply (rule Join_Always_rel_le_giv [THEN Always_LeadsToI], simp_all)
apply (rule LeadsTo_Un [THEN LeadsTo_weaken_L])
  apply (blast intro: rel_progress_lemma)
  apply (rule subset_refl [THEN subset_imp_LeadsTo])
apply (blast intro: order_less_le [THEN iffD2] dest: common_prefix_linear)
done

```

Progress property: all tokens that are given will be released

```

lemma client_progress:
  "Client  $\in$ 
    Increasing giv guarantees
    (INT h. {s. h  $\leq$  giv s & h pfixGe ask s} LeadsTo {s. h  $\leq$  rel s})"
apply (rule guaranteesI, clarify)
apply (blast intro: client_progress_lemma)
done

```

This shows that the Client won't alter other variables in any state that it is combined with

```

lemma client_preserves_dummy: "Client  $\in$  preserves dummy"
by (simp add: Client_def preserves_def, clarify, safety, auto)

```

\* Obsolete lemmas from first version of the Client \*

```

lemma stable_size_rel_le_giv:
  "Client  $\in$  stable {s. size (rel s)  $\leq$  size (giv s)}"
by (simp add: Client_def, safety, auto)

```

clients return the right number of tokens

```

lemma ok_guar_rel_prefix_giv:
  "Client  $\in$  Increasing giv guarantees Always {s. rel s  $\leq$  giv s}"
apply (rule guaranteesI)
apply (rule AlwaysI, force)
apply (blast intro: Increasing_preserves_Stable stable_rel_le_giv)
done

```

end

## 27 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 ==
     $\forall G. \text{extend } h \ F \sqcup G \in X' \ \longrightarrow \ F \sqcup \text{project } h \ (C \ G) \ G \in X"$ 

```

```

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 ==
     $\forall G. \text{extend } h \ F \ \text{ok } G \ \longrightarrow \ F \sqcup \text{project } h \ (C \ G) \ G \in Y"$ 

```

```
--> extend h F ⊔ G ∈ Y'"
```

```
definition subset_closed :: "'a set set => bool" where
  "subset_closed U == ∀ A ∈ U. Pow A ⊆ U"
```

```
context Extend
begin
```

```
lemma project_extend_constrains_I:
  "F ∈ A co B ==> project h C (extend h F) ∈ A co B"
apply (auto simp add: extend_act_def project_act_def constrains_def)
done
```

## 27.1 Safety

```
lemma project_unless:
  "[| G ∈ stable C; project h C G ∈ A unless B |]
  ==> G ∈ (C ∩ extend_set h A) unless (extend_set h B)"
apply (simp add: unless_def project_constrains)
apply (blast dest: stable_constrains_Int intro: constrains_weaken)
done
```

```
lemma Join_project_constrains:
  "(F ⊔ project h C G ∈ A co B) =
   (extend h F ⊔ G ∈ (C ∩ extend_set h A) co (extend_set h B) &
    F ∈ A co B)"
apply (simp (no_asm) add: project_constrains)
apply (blast intro: extend_constrains [THEN iffD2, THEN constrains_weaken]
  dest: constrains_imp_subset)
done
```

```
lemma Join_project_stable:
  "extend h F ⊔ G ∈ stable C
  ==> (F ⊔ project h C G ∈ stable A) =
   (extend h F ⊔ G ∈ stable (C ∩ extend_set h A) &
    F ∈ stable A)"
apply (unfold stable_def)
apply (simp only: Join_project_constrains)
apply (blast intro: constrains_weaken dest: constrains_Int)
done
```

```
lemma project_constrains_I:
  "extend h F ⊔ G ∈ extend_set h A co extend_set h B
  ==> F ⊔ project h C G ∈ A co B"
apply (simp add: project_constrains extend_constrains)
apply (blast intro: constrains_weaken dest: constrains_imp_subset)
done
```

```
lemma project_increasing_I:
```

```

    "extend h F ⊔ G ∈ increasing (func o f)
    ==> F ⊔ project h C G ∈ increasing func"
  apply (unfold increasing_def stable_def)
  apply (simp del: Join_constrains
    add: project_constrains_I extend_set_eq_Collect)
done

lemma Join_project_increasing:
  "(F ⊔ project h UNIV G ∈ increasing func) =
  (extend h F ⊔ G ∈ increasing (func o f))"
  apply (rule iffI)
  apply (erule_tac [2] project_increasing_I)
  apply (simp del: Join_stable
    add: increasing_def Join_project_stable)
  apply (auto simp add: extend_set_eq_Collect extend_stable [THEN iffD1])
done

lemma project_constrains_D:
  "F ⊔ project h UNIV G ∈ A co B
  ==> extend h F ⊔ G ∈ extend_set h A co extend_set h B"
  by (simp add: project_constrains extend_constrains)

end

```

## 27.2 "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)"
  by (unfold projecting_def, blast)

lemma projecting_Un:
  "[| projecting C h F XA' XA; projecting C h F XB' XB |]
  ==> projecting C h F (XA' ∪ XB') (XA ∪ XB)"
  by (unfold projecting_def, blast)

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)"
  by (unfold projecting_def, blast)

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)"
  by (unfold projecting_def, blast)

lemma projecting_weaken:
  "[| projecting C h F X' X; U' ≤ X'; X ⊆ U |] ==> projecting C h F U'
  U"
  by (unfold projecting_def, auto)

lemma projecting_weaken_L:
  "[| projecting C h F X' X; U' ≤ X' |] ==> projecting C h F U' X"

```

```

by (unfold projecting_def, auto)

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)"
by (unfold extending_def, blast)

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)"
by (unfold extending_def, blast)

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)"
by (unfold extending_def, blast)

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)"
by (unfold extending_def, blast)

lemma extending_weaken:
  "[| extending C h F Y' Y;  Y'  $\leq$  V';  V  $\subseteq$  Y |] ==> extending C h F V' V"
by (unfold extending_def, auto)

lemma extending_weaken_L:
  "[| extending C h F Y' Y;  Y'  $\leq$  V' |] ==> extending C h F V' Y"
by (unfold extending_def, auto)

lemma projecting_UNIV: "projecting C h F X' UNIV"
by (simp add: projecting_def)

context Extend
begin

lemma projecting_constrains:
  "projecting C h F (extend_set h A co extend_set h B) (A co B)"
apply (unfold projecting_def)
apply (blast intro: project_constrains_I)
done

lemma projecting_stable:
  "projecting C h F (stable (extend_set h A)) (stable A)"
apply (unfold stable_def)
apply (rule projecting_constrains)
done

lemma projecting_increasing:
  "projecting C h F (increasing (func o f)) (increasing func)"
apply (unfold projecting_def)
apply (blast intro: project_increasing_I)
done

```



```

lemma extending_UNIV: "extending C h F UNIV Y"
apply (simp (no_asm) add: extending_def)
done

lemma extending_constrains:
  "extending (%G. UNIV) h F (extend_set h A co extend_set h B) (A co B)"
apply (unfold extending_def)
apply (blast intro: project_constrains_D)
done

lemma extending_stable:
  "extending (%G. UNIV) h F (stable (extend_set h A)) (stable A)"
apply (unfold stable_def)
apply (rule extending_constrains)
done

lemma extending_increasing:
  "extending (%G. UNIV) h F (increasing (func o f)) (increasing func)"
by (force simp only: extending_def Join_project_increasing)

```

### 27.3 Reachability and project

```

lemma reachable_imp_reachable_project:
  "[| reachable (extend h F⊔G) ⊆ C;
    z ∈ reachable (extend h F⊔G) |]
  ==> f z ∈ reachable (F⊔project h C G)"
apply (erule reachable.induct)
apply (force intro!: reachable.Init simp add: split_extended_all, auto)
apply (rule_tac act = x in reachable.Acts)
apply auto
apply (erule extend_act_D)
apply (rule_tac act1 = "Restrict C act"
  in project_act_I [THEN [3] reachable.Acts], auto)
done

lemma 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)"
apply (unfold Constrains_def)
apply (simp del: Join_constrains
  add: Join_project_constrains, clarify)
apply (erule constrains_weaken)
apply (auto intro: reachable_imp_reachable_project)
done

lemma project_Stable_D:
  "F⊔project h (reachable (extend h F⊔G)) G ∈ Stable A
  ==> extend h F⊔G ∈ Stable (extend_set h A)"
apply (unfold Stable_def)
apply (simp (no_asm_simp) add: project_Constrains_D)
done

lemma project_Always_D:
  "F⊔project h (reachable (extend h F⊔G)) G ∈ Always A

```

```

==> extend h F⊔G ∈ Always (extend_set h A)"
apply (unfold Always_def)
apply (force intro: reachable.Init simp add: project_Stable_D split_extended_all)
done

lemma project_Increasing_D:
  "F⊔project h (reachable (extend h F⊔G)) G ∈ Increasing func
  ==> extend h F⊔G ∈ Increasing (func o f)"
apply (unfold Increasing_def, auto)
apply (subst extend_set_eq_Collect [symmetric])
apply (simp (no_asm_simp) add: project_Stable_D)
done

```

## 27.4 Converse results for weak safety: benefits of the argument C

```

lemma 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)"
apply (erule reachable.induct)
apply (force intro: reachable.Init)
apply (auto simp add: project_act_def)
apply (force del: Id_in_Acts intro: reachable.Acts extend_act_D)+
done

```

```

lemma project_set_reachable_extend_eq:
  "project_set h (reachable (extend h F⊔G)) =
  reachable (F⊔project h (reachable (extend h F⊔G)) G)"
by (auto dest: subset_refl [THEN reachable_imp_reachable_project]
    subset_refl [THEN reachable_project_imp_reachable])

```

```

lemma reachable_extend_Join_subset:
  "reachable (extend h F⊔G) ⊆ C
  ==> reachable (extend h F⊔G) ⊆
  extend_set h (reachable (F⊔project h C G))"
apply (auto dest: reachable_imp_reachable_project)
done

```

```

lemma 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"
apply (unfold Constrains_def)
apply (simp del: Join_constrains
  add: Join_project_constrains extend_set_Int_distrib)
apply (rule conjI)
  prefer 2
  apply (force elim: constrains_weaken_L
    dest!: extend_constrains_project_set
    subset_refl [THEN reachable_project_imp_reachable])
apply (blast intro: constrains_weaken_L)
done

```

```

lemma project_Stable_I:
  "extend h F⊔G ∈ Stable (extend_set h A)
   ==> F⊔project h (reachable (extend h F⊔G)) G ∈ Stable A"
apply (unfold Stable_def)
apply (simp (no_asm_simp) add: project_Constrains_I)
done

lemma project_Always_I:
  "extend h F⊔G ∈ Always (extend_set h A)
   ==> F⊔project h (reachable (extend h F⊔G)) G ∈ Always A"
apply (unfold Always_def)
apply (auto simp add: project_Stable_I)
apply (unfold extend_set_def, blast)
done

lemma project_Increasing_I:
  "extend h F⊔G ∈ Increasing (func o f)
   ==> F⊔project h (reachable (extend h F⊔G)) G ∈ Increasing func"
apply (unfold Increasing_def, auto)
apply (simp (no_asm_simp) add: extend_set_eq_Collect project_Stable_I)
done

lemma 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))"
apply (blast intro: project_Constrains_I project_Constrains_D)
done

lemma project_Stable:
  "(F⊔project h (reachable (extend h F⊔G)) G ∈ Stable A) =
   (extend h F⊔G ∈ Stable (extend_set h A))"
apply (unfold Stable_def)
apply (rule project_Constrains)
done

lemma project_Increasing:
  "(F⊔project h (reachable (extend h F⊔G)) G ∈ Increasing func) =
   (extend h F⊔G ∈ Increasing (func o f))"
apply (simp (no_asm_simp) add: Increasing_def project_Stable extend_set_eq_Collect)
done

```

## 27.5 A lot of redundant theorems: all are proved to facilitate reasoning about guarantees.

```

lemma projecting_Constrains:
  "projecting (%G. reachable (extend h F⊔G)) h F
   (extend_set h A Co extend_set h B) (A Co B)"

apply (unfold projecting_def)
apply (blast intro: project_Constrains_I)
done

lemma projecting_Stable:
  "projecting (%G. reachable (extend h F⊔G)) h F

```

```

      (Stable (extend_set h A)) (Stable A)"
apply (unfold Stable_def)
apply (rule projecting_Constrains)
done

lemma projecting_Always:
  "projecting (%G. reachable (extend h F ⊔ G)) h F
   (Always (extend_set h A)) (Always A)"
apply (unfold projecting_def)
apply (blast intro: project_Always_I)
done

lemma projecting_Increasing:
  "projecting (%G. reachable (extend h F ⊔ G)) h F
   (Increasing (func o f)) (Increasing func)"
apply (unfold projecting_def)
apply (blast intro: project_Increasing_I)
done

lemma extending_Constrains:
  "extending (%G. reachable (extend h F ⊔ G)) h F
   (extend_set h A Co extend_set h B) (A Co B)"
apply (unfold extending_def)
apply (blast intro: project_Constrains_D)
done

lemma extending_Stable:
  "extending (%G. reachable (extend h F ⊔ G)) h F
   (Stable (extend_set h A)) (Stable A)"
apply (unfold extending_def)
apply (blast intro: project_Stable_D)
done

lemma extending_Always:
  "extending (%G. reachable (extend h F ⊔ G)) h F
   (Always (extend_set h A)) (Always A)"
apply (unfold extending_def)
apply (blast intro: project_Always_D)
done

lemma extending_Increasing:
  "extending (%G. reachable (extend h F ⊔ G)) h F
   (Increasing (func o f)) (Increasing func)"
apply (unfold extending_def)
apply (blast intro: project_Increasing_D)
done

```

## 27.6 leadsETo in the precondition (??)

### 27.6.1 transient

```

lemma 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)"

```

```

apply (auto simp add: transient_def Domain_project_act)
apply (subgoal_tac "act '' (C  $\cap$  extend_set h A)  $\subseteq$  - extend_set h A")
  prefer 2
  apply (simp add: stable_def constrains_def, blast)

apply (erule_tac V = "AA  $\subseteq$  -C  $\cup$  BB" for AA BB in thin_rl)
apply (drule bspec, assumption)
apply (simp add: extend_set_def project_act_def, blast)
done

```

```

lemma project_extend_transient_D:
  "project h C (extend h F)  $\in$  transient (project_set h C  $\cap$  D)
  ==> F  $\in$  transient (project_set h C  $\cap$  D)"
apply (simp add: transient_def Domain_project_act, safe)
apply blast+
done

```

### 27.6.2 ensures – a primitive combining progress with safety

```

lemma ensures_extend_set_imp_project_ensures:
  "[| extend h F  $\in$  stable C; G  $\in$  stable C;
    extend h F  $\sqcup$  G  $\in$  A ensures B; A-B = C  $\cap$  extend_set h D |]
  ==> F  $\sqcup$  project h C G
     $\in$  (project_set h C  $\cap$  project_set h A) ensures (project_set h B)"
apply (simp add: ensures_def project_constrains extend_transient,
  clarify)
apply (intro conjI)

apply (blast intro: extend_stable_project_set
  [THEN stableD, THEN constrains_Int, THEN constrains_weaken]

  dest!: extend_constrains_project_set equalityD1)

apply (erule stableD [THEN constrains_Int, THEN constrains_weaken])
  apply assumption
  apply (simp (no_asm_use) add: extend_set_def)
  apply blast
  apply (simp add: extend_set_Int_distrib extend_set_Un_distrib)
  apply (blast intro!: extend_set_project_set [THEN subsetD], blast)

apply auto
  prefer 2
  apply (force dest!: equalityD1
    intro: transient_extend_set_imp_project_transient
      [THEN transient_strengthen])
apply (simp (no_asm_use) add: Int_Diff)
apply (force dest!: equalityD1
  intro: transient_extend_set_imp_project_transient
    [THEN project_extend_transient_D, THEN transient_strengthen])
done

```

Transferring a transient property upwards

```

lemma 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)"
    apply (simp add: transient_def project_set_def extend_set_def project_act_def)
    apply (elim disjE bexE)
    apply (rule_tac x=Id in bexI)
    apply (blast intro!: rev_bexI )+
  done

```

```

lemma project_unless2:
  "[| 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)"
by (auto dest: stable_constrains_Int intro: constrains_weaken
    simp add: unless_def project_constrains Diff_eq Int_assoc
    Int_extend_set_lemma)

```

```

lemma extend_unless:
  "[| extend h F ∈ stable C; F ∈ A unless B |]"
  ==> extend h F ∈ C ∩ extend_set h A unless extend_set h B"
apply (simp add: unless_def stable_def)
apply (drule constrains_Int)
apply (erule extend_constrains [THEN iffD2])
apply (erule constrains_weaken, blast)
apply blast
done

```

```

lemma Join_project_ensures:
  "[| 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)"
apply (auto simp add: ensures_eq extend_unless project_unless)
apply (blast intro: extend_transient [THEN iffD2] transient_strengthen)
apply (blast intro: project_transient_extend_set transient_strengthen)
done

```

Lemma useful for both STRONG and WEAK progress, but the transient condition's very strong

```

lemma 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)"
apply (erule leadsTo_induct)
apply (blast intro: Join_project_ensures)
apply (blast intro: psp_stable2 [THEN leadsTo_weaken_L] leadsTo_Trans)
apply (simp del: UN_simps add: Int_UN_distrib leadsTo_UN extend_set_Union)
done

```

```

lemma 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)"

```

```

apply (rule PLD_lemma [THEN leadsTo_weaken])
apply (auto simp add: split_extended_all)
done

lemma 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)"
by (simp del: Join_stable add: LeadsTo_def project_leadsTo_D_lemma
    project_set_reachable_extend_eq)

```

## 27.7 Towards the theorem *project\_Ensures\_D*

```

lemma 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)"

apply (auto intro!: project_unless2 [unfolded unless_def]
    intro: project_extend_constrains_I
    simp add: ensures_def)

prefer 2
apply (blast intro: project_transient_extend_set)

apply (force elim!: extend_transient [THEN iffD2, THEN transient_strengthen]
    simp add: split_extended_all)
done

lemma 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)"
apply (rule project_ensures_D_lemma [of _ UNIV, elim_format], auto)
done

lemma 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)"
apply (unfold Ensures_def)
apply (rule project_ensures_D_lemma [elim_format])
apply (auto simp add: project_set_reachable_extend_eq [symmetric])
done

```

## 27.8 Guarantees

```

lemma project_act_Restrict_subset_project_act:
  "project_act h (Restrict C act) ⊆ project_act h act"
apply (auto simp add: project_act_def)
done

```

```

lemma subset_closed_ok_extend_imp_ok_project:
  "[| extend h F ok G; subset_closed (AllowedActs F) |]
   ==> F ok project h C G"
  apply (auto simp add: ok_def)
  apply (rename_tac act)
  apply (drule subsetD, blast)
  apply (rule_tac x = "Restrict C (extend_act h act)" in rev_image_eqI)
  apply simp +
  apply (cut_tac project_act_Restrict_subset_project_act)
  apply (auto simp add: subset_closed_def)
done

```

```

lemma project_guarantees_raw:
  assumes xguary: "F ∈ X guarantees Y"
  and closed: "subset_closed (AllowedActs F)"
  and project: "!!G. extend h F ⊔ G ∈ X'
               ==> F ⊔ project h (C G) G ∈ X"
  and extend: "!!G. [| F ⊔ project h (C G) G ∈ Y |]
               ==> extend h F ⊔ G ∈ Y'"
  shows "extend h F ∈ X' guarantees Y'"
  apply (rule xguary [THEN guaranteesD, THEN extend, THEN guaranteesI])
  apply (blast intro: closed subset_closed_ok_extend_imp_ok_project)
  apply (erule project)
done

```

```

lemma 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'"
  apply (rule guaranteesI)
  apply (auto simp add: guaranteesD projecting_def extending_def
    subset_closed_ok_extend_imp_ok_project)
done

```

## 27.9 guarantees corollaries

### 27.9.1 Some could be deleted: the required versions are easy to prove

```

lemma extend_guar_increasing:
  "[| F ∈ UNIV guarantees increasing func;
    subset_closed (AllowedActs F) |]
   ==> extend h F ∈ X' guarantees increasing (func o f)"
  apply (erule project_guarantees)
  apply (rule_tac [3] extending_increasing)
  apply (rule_tac [2] projecting_UNIV, auto)
done

```

```

lemma extend_guar_Increasing:

```



```

    "[| F ∈ UNIV guarantees Increasing func;
      subset_closed (AllowedActs F) |]
    ==> extend h F ∈ X' guarantees Increasing (func o f)"
  apply (erule project_guarantees)
  apply (rule_tac [3] extending_Increasing)
  apply (rule_tac [2] projecting_UNIV, auto)
done

lemma 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)"
  apply (erule project_guarantees)
  apply (rule_tac [3] extending_Always)
  apply (rule_tac [2] projecting_Always, auto)
done

```

### 27.9.2 Guarantees with a leadsTo postcondition

```

lemma project_leadsTo_D:
  "F⊔project h UNIV G ∈ A leadsTo B
  ==> extend h F⊔G ∈ (extend_set h A) leadsTo (extend_set h B)"
  apply (rule_tac C1 = UNIV in project_leadsTo_D_lemma [THEN leadsTo_weaken],
  auto)
done

lemma 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)"
  apply (rule refl [THEN Join_project_LeadsTo], auto)
done

lemma extending_leadsTo:
  "extending (%G. UNIV) h F
    (extend_set h A leadsTo extend_set h B) (A leadsTo B)"
  apply (unfold extending_def)
  apply (blast intro: project_leadsTo_D)
done

lemma extending_LeadsTo:
  "extending (%G. reachable (extend h F⊔G)) h F
    (extend_set h A LeadsTo extend_set h B) (A LeadsTo B)"
  apply (unfold extending_def)
  apply (blast intro: project_LeadsTo_D)
done

end

end

```

## 28 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: "∀ 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"
by (unfold givenBy_def, auto)

```

```

lemma givenBy_eq_all: "(givenBy v) = {A. ∀ x ∈ A. ∀ y. v x = v y ⟶ y ∈ A}"
apply (unfold givenBy_def, safe)
apply (rule_tac [2] x = "v ' _" in image_eqI, auto)
done

```

```

lemma givenByI: "(⋀ x y. [| x ∈ A; v x = v y |] ==> y ∈ A) ==> A ∈ givenBy
v"
by (subst givenBy_eq_all, blast)

```

```

lemma givenByD: "[| A ∈ givenBy v; x ∈ A; v x = v y |] ==> y ∈ A"
by (unfold givenBy_def, auto)

```

```

lemma empty_mem_givenBy [iff]: "{} ∈ givenBy v"
by (blast intro!: givenByI)

```

```

lemma givenBy_imp_eq_Collect: "A ∈ givenBy v ==> ∃ P. A = {s. P(v s)}"

```

```

apply (rule_tac x = "λn. ∃ s. v s = n ∧ s ∈ A" in exI)
apply (simp (no_asm_use) add: givenBy_eq_all)
apply blast
done

```

```

lemma Collect_mem_givenBy: "{s. P(v s)} ∈ givenBy v"
by (unfold givenBy_def, best)

```

```

lemma givenBy_eq_Collect: "givenBy v = {A. ∃ P. A = {s. P(v s)}}"
by (blast intro: Collect_mem_givenBy givenBy_imp_eq_Collect)

```

```

lemma preserves_givenBy_imp_stable:
  "[| F ∈ preserves v; D ∈ givenBy v |] ==> F ∈ stable D"
by (force simp add: preserves_subset_stable [THEN subsetD] givenBy_eq_Collect)

```

```

lemma givenBy_o_subset: "givenBy (w o v) <= givenBy v"
apply (simp (no_asm) add: givenBy_eq_Collect)
apply best
done

```

```

lemma givenBy_DiffI:
  "[| A ∈ givenBy v; B ∈ givenBy v |] ==> A-B ∈ givenBy v"
apply (simp (no_asm_use) add: givenBy_eq_Collect)
apply safe
apply (rule_tac x = "%z. R z & ~ Q z" for R Q in exI)
unfolding set_diff_eq
apply auto
done

```

```

lemma leadsETo_Basis [intro]:
  "[| F ∈ A ensures B; A-B ∈ insert {} CC |] ==> F ∈ A leadsTo[CC] B"
apply (unfold leadsETo_def)
apply (blast intro: elt.Basis)
done

```

```

lemma leadsETo_Trans:
  "[| F ∈ A leadsTo[CC] B; F ∈ B leadsTo[CC] C |] ==> F ∈ A leadsTo[CC] C"
apply (unfold leadsETo_def)
apply (blast intro: elt.Trans)
done

```

```

lemma leadsETo_Un_duplicate:
  "F ∈ A leadsTo[CC] (A' ∪ A') ==> F ∈ A leadsTo[CC] A'"
by (simp add: Un_ac)

```

```

lemma leadsETo_Un_duplicate2:
  "F ∈ A leadsTo[CC] (A' ∪ C ∪ C) ==> F ∈ A leadsTo[CC] (A' Un C)"

```

by (simp add: Un\_ac)

```
lemma leadsETo_Union:
  " $(\bigwedge A. A \in S \implies F \in A \text{ leadsTo}[CC] B) \implies F \in (\bigcup S) \text{ leadsTo}[CC] B$ "
apply (unfold leadsETo_def)
apply (blast intro: elt.Union)
done
```

```
lemma leadsETo_UN:
  " $(\bigwedge i. i \in I \implies F \in (A i) \text{ leadsTo}[CC] B) \implies F \in (\bigcup i:I. A i) \text{ leadsTo}[CC] B$ "
apply (blast intro: leadsETo_Union)
done
```

```
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. ∀A∈S. F ∈ A leadsTo[CC] B & P A B ==> P (⋃ S) B
  |] ==> P za zb"
apply (unfold leadsETo_def)
apply (drule CollectD)
apply (erule elt.induct, blast+)
done
```

```
lemma leadsETo_mono: "CC' <= CC ==> (A leadsTo[CC'] B) <= (A leadsTo[CC] B)"
apply safe
apply (erule leadsETo_induct)
prefer 3 apply (blast intro: leadsETo_Union)
prefer 2 apply (blast intro: leadsETo_Trans)
apply blast
done
```

```
lemma leadsETo_Trans_Un:
  "[| F ∈ A leadsTo[CC] B; F ∈ B leadsTo[DD] C |]
  ==> F ∈ A leadsTo[CC Un DD] C"
by (blast intro: leadsETo_mono [THEN subsetD] leadsETo_Trans)
```

```
lemma leadsETo_Union_Int:
  " $(\bigwedge A. A \in S \implies F \in (A \text{ Int } C) \text{ leadsTo}[CC] B) \implies F \in (\bigcup S \text{ Int } C) \text{ leadsTo}[CC] B$ "
apply (unfold leadsETo_def)
apply (simp only: Int_Union_Union)
apply (blast intro: elt.Union)
done
```

```

lemma leadsETo_Un:
  "[| F ∈ A leadsTo[CC] C; F ∈ B leadsTo[CC] C |]
   ==> F ∈ (A Un B) leadsTo[CC] C"
  using leadsETo_Union [of "{A, B}" F CC C] by auto

lemma single_leadsETo_I:
  "( $\bigwedge x. x \in A \implies F \in \{x\} \text{ leadsTo[CC] } B$ )  $\implies F \in A \text{ leadsTo[CC] } B$ "
  by (subst UN_singleton [symmetric], rule leadsETo_UN, blast)

lemma subset_imp_leadsETo: "A<=B  $\implies F \in A \text{ leadsTo[CC] } B$ "
  by (simp add: subset_imp_ensures [THEN leadsETo_Basis]
      Diff_eq_empty_iff [THEN iffD2])

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'"
  by (blast intro: subset_imp_leadsETo leadsETo_Trans)

lemma leadsETo_weaken_L:
  "[| F ∈ A leadsTo[CC] A'; B<=A |] ==> F ∈ B leadsTo[CC] A'"
  by (blast intro: leadsETo_Trans subset_imp_leadsETo)

lemma leadsETo_Un_distrib:
  "F ∈ (A Un B) leadsTo[CC] C =
   (F ∈ A leadsTo[CC] C  $\wedge$  F ∈ B leadsTo[CC] C)"
  by (blast intro: leadsETo_Un leadsETo_weaken_L)

lemma leadsETo_UN_distrib:
  "F ∈ (UN i:I. A i) leadsTo[CC] B =
   ( $\forall i \in I. F \in (A i) \text{ leadsTo[CC] } B$ )"
  by (blast intro: leadsETo_UN leadsETo_weaken_L)

lemma leadsETo_Union_distrib:
  "F ∈ ( $\bigcup S$ ) leadsTo[CC] B = ( $\forall A \in S. F \in A \text{ leadsTo[CC] } B$ )"
  by (blast intro: leadsETo_Union leadsETo_weaken_L)

lemma leadsETo_weaken:
  "[| F ∈ A leadsTo[CC'] A'; B<=A; A'<=B'; CC' <= CC |]
   ==> F ∈ B leadsTo[CC] B'"
  apply (drule leadsETo_mono [THEN subsetD], assumption)
  apply (blast del: subsetCE
      intro: leadsETo_weaken_R leadsETo_weaken_L leadsETo_Trans)
  done

lemma leadsETo_givenBy:
  "[| F ∈ A leadsTo[CC] A'; CC <= givenBy v |]"

```

```

    ==> F ∈ A leadsTo[givenBy v] A'"
  by (blast intro: leadsETO_weaken)

```

```

lemma leadsETO_Diff:
  "[| F ∈ (A-B) leadsTo[CC] C; F ∈ B leadsTo[CC] C |]"
  ==> F ∈ A leadsTo[CC] C"
  by (blast intro: leadsETO_Un leadsETO_weaken)

```

```

lemma leadsETO_Un_Un:
  "[| F ∈ A leadsTo[CC] A'; F ∈ B leadsTo[CC] B' |]"
  ==> F ∈ (A Un B) leadsTo[CC] (A' Un B'"
  by (blast intro: leadsETO_Un leadsETO_weaken_R)

```

```

lemma leadsETO_cancel2:
  "[| F ∈ A leadsTo[CC] (A' Un B); F ∈ B leadsTo[CC] B' |]"
  ==> F ∈ A leadsTo[CC] (A' Un B'"
  by (blast intro: leadsETO_Un_Un subset_imp_leadsETO leadsETO_Trans)

```

```

lemma leadsETO_cancel1:
  "[| F ∈ A leadsTo[CC] (B Un A'); F ∈ B leadsTo[CC] B' |]"
  ==> F ∈ A leadsTo[CC] (B' Un A'"
  apply (simp add: Un_commute)
  apply (blast intro!: leadsETO_cancel2)
done

```

```

lemma leadsETO_cancel_Diff1:
  "[| F ∈ A leadsTo[CC] (B Un A'); F ∈ (B-A') leadsTo[CC] B' |]"
  ==> F ∈ A leadsTo[CC] (B' Un A'"
  apply (rule leadsETO_cancel1)
  prefer 2 apply assumption
  apply simp_all
done

```

```

lemma e_psp_stable:
  "[| F ∈ A leadsTo[CC] A'; F ∈ stable B; ∀ C ∈ CC. C Int B ∈ CC |]"
  ==> F ∈ (A Int B) leadsTo[CC] (A' Int B)"
  apply (unfold stable_def)
  apply (erule leadsETO_induct)
  prefer 3 apply (blast intro: leadsETO_Union_Int)
  prefer 2 apply (blast intro: leadsETO_Trans)
  apply (rule leadsETO_Basis)
  prefer 2 apply (force simp add: Diff_Int_distrib2 [symmetric])
  apply (simp add: ensures_def Diff_Int_distrib2 [symmetric])

```

```

      Int_Un_distrib2 [symmetric])
apply (blast intro: transient_strengthen constrains_Int)
done

lemma e_psp_stable2:
  "[| F ∈ A leadsTo[CC] A'; F ∈ stable B; ∀ C ∈ CC. C Int B ∈ CC |]
   ==> F ∈ (B Int A) leadsTo[CC] (B Int A')]"
by (simp (no_asm_simp) add: e_psp_stable Int_ac)

lemma e_psp:
  "[| F ∈ A leadsTo[CC] A'; F ∈ B co B';
    ∀ C ∈ CC. C Int B Int B' ∈ CC |]
   ==> F ∈ (A Int B') leadsTo[CC] ((A' Int B) Un (B' - B))]"
apply (erule leadsETO_induct)
prefer 3 apply (blast intro: leadsETO_Union_Int)

apply (rule_tac [2] leadsETO_Un_duplicate2)
apply (erule_tac [2] leadsETO_cancel_Diff1)
prefer 2
  apply (simp add: Int_Diff Diff_triv)
  apply (blast intro: leadsETO_weaken_L dest: constrains_imp_subset)

apply (rule leadsETO_Basis)
apply (blast intro: psp_ensures)
apply (subgoal_tac "A Int B' - (Ba Int B Un (B' - B)) = (A - Ba) Int B Int B'")
apply auto
done

lemma e_psp2:
  "[| F ∈ A leadsTo[CC] A'; F ∈ B co B';
    ∀ C ∈ CC. C Int B Int B' ∈ CC |]
   ==> F ∈ (B' Int A) leadsTo[CC] ((B Int A') Un (B' - B))]"
by (simp add: e_psp Int_ac)

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)]"
apply (erule leadsETO_induct)
prefer 3
  apply (subst Int_Union)
  apply (blast intro: leadsETO_UN)
prefer 2
  apply (blast intro: e_psp_stable2 [THEN leadsETO_weaken_L] leadsETO_Trans)
apply (rule leadsETO_Basis)
apply (auto simp add: Diff_eq_empty_iff [THEN iffD2]
  Int_Diff ensures_def givenBy_eq_Collect)
prefer 3 apply (blast intro: transient_strengthen)
apply (drule_tac [2] P1 = P in preserves_subset_stable [THEN subsetD])

```

```

apply (drule_tac P1 = P in preserves_subset_stable [THEN subsetD])
apply (unfold stable_def)
apply (blast intro: constrains_Int [THEN constrains_weaken])+
done

```

```

lemma leadsETo_subset_leadsTo: "(A leadsTo[CC] B) <= (A leadsTo B)"
apply safe
apply (erule leadsETo_induct)
  prefer 3 apply (blast intro: leadsTo_Union)
  prefer 2 apply (blast intro: leadsTo_Trans, blast)
done

```

```

lemma leadsETo_UNIV_eq_leadsTo: "(A leadsTo[UNIV] B) = (A leadsTo B)"
apply safe
apply (erule leadsETo_subset_leadsTo [THEN subsetD])

apply (erule leadsTo_induct)
  prefer 3 apply (blast intro: leadsETo_Union)
  prefer 2 apply (blast intro: leadsETo_Trans, blast)
done

```

```

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)}"
apply (unfold LeadsETo_def)
apply (blast dest: e_psp_stable2 intro: leadsETo_weaken)
done

```

```

lemma LeadsETo_Trans:
  "[| F ∈ A LeadsTo[CC] B; F ∈ B LeadsTo[CC] C |]
   ==> F ∈ A LeadsTo[CC] C"
apply (simp add: LeadsETo_eq_leadsETo)
apply (blast intro: leadsETo_Trans)
done

```

```

lemma LeadsETo_Union:
  "( $\bigwedge A. A \in S \implies F \in A \text{ LeadsTo[CC] } B$ )  $\implies F \in (\bigcup S) \text{ LeadsTo[CC] } B$ "
apply (simp add: LeadsETo_def)
apply (subst Int_Union)
apply (blast intro: leadsETo_UN)
done

```

```

lemma LeadsETo_UN:
  "( $\bigwedge i. i \in I \implies F \in (A \ i) \text{ LeadsTo[CC] } B$ )"

```



```

    ==> F ∈ (UN i:I. A i) LeadsTo[CC] B"
  apply (blast intro: LeadsETo_Union)
  done

```

```

lemma LeadsETo_Un:
  "[| F ∈ A LeadsTo[CC] C; F ∈ B LeadsTo[CC] C |]
   ==> F ∈ (A Un B) LeadsTo[CC] C"
  using LeadsETo_Union [of "{A, B}" F CC C] by auto

```

```

lemma single_LeadsETo_I:
  "( $\bigwedge s. s \in A \implies F \in \{s\} \text{ LeadsTo[CC] } B$ ) ==> F ∈ A LeadsTo[CC] B"
  by (subst UN_singleton [symmetric], rule LeadsETo_UN, blast)

```

```

lemma subset_imp_LeadsETo:
  "A <= B ==> F ∈ A LeadsTo[CC] B"
  apply (simp (no_asm) add: LeadsETo_def)
  apply (blast intro: subset_imp_leadsETo)
  done

```

```

lemmas empty_LeadsETo = empty_subsetI [THEN subset_imp_LeadsETo]

```

```

lemma LeadsETo_weaken_R:
  "[| F ∈ A LeadsTo[CC] A'; A' <= B' |] ==> F ∈ A LeadsTo[CC] B'"
  apply (simp add: LeadsETo_def)
  apply (blast intro: leadsETo_weaken_R)
  done

```

```

lemma LeadsETo_weaken_L:
  "[| F ∈ A LeadsTo[CC] A'; B <= A |] ==> F ∈ B LeadsTo[CC] A'"
  apply (simp add: LeadsETo_def)
  apply (blast intro: leadsETo_weaken_L)
  done

```

```

lemma LeadsETo_weaken:
  "[| F ∈ A LeadsTo[CC'] A';
     B <= A; A' <= B'; CC' <= CC |]
   ==> F ∈ B LeadsTo[CC] B'"
  apply (simp (no_asm_use) add: LeadsETo_def)
  apply (blast intro: leadsETo_weaken)
  done

```

```

lemma LeadsETo_subset_LeadsTo: "(A LeadsTo[CC] B) <= (A LeadsTo B)"
  apply (unfold LeadsETo_def LeadsTo_def)
  apply (blast intro: leadsETo_subset_leadsTo [THEN subsetD])
  done

```

```

lemma reachable_ensures:
  "F ∈ A ensures B ==> F ∈ (reachable F Int A) ensures B"
  apply (rule stable_ensures_Int [THEN ensures_weaken_R], auto)
  done

```

```

lemma lel_lemma:
  "F ∈ A leadsTo B ⇒ F ∈ (reachable F Int A) leadsTo[Pow(reachable F)]
  B"
apply (erule leadsTo_induct)
  apply (blast intro: reachable_ensures)
  apply (blast dest: e_psp_stable2 intro: leadsETo_Trans leadsETo_weaken_L)
apply (subst Int_Union)
apply (blast intro: leadsETo_UN)
done

lemma LeadsETo_UNIV_eq_LeadsTo: "(A LeadsTo[UNIV] B) = (A LeadsTo B)"
apply safe
apply (erule LeadsETo_subset_LeadsTo [THEN subsetD])

apply (unfold LeadsETo_def LeadsTo_def)
apply (blast intro: lel_lemma [THEN leadsETo_weaken])
done

```

```

context Extend
begin

```

```

lemma givenBy_o_eq_extend_set:
  "givenBy (v o f) = extend_set h ' (givenBy v)"
apply (simp add: givenBy_eq_Collect)
apply (rule equalityI, best)
apply blast
done

lemma givenBy_eq_extend_set: "givenBy f = range (extend_set h)"
by (simp add: givenBy_eq_Collect, best)

lemma extend_set_givenBy_I:
  "D ∈ givenBy v ==> extend_set h D ∈ givenBy (v o f)"
apply (simp (no_asm_use) add: givenBy_eq_all, blast)
done

```

```

lemma 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)"
apply (erule leadsETo_induct)
  apply (force intro: subset_imp_ensures
    simp add: extend_ensures extend_set_Diff_distrib [symmetric])
  apply (blast intro: leadsETo_Trans)
apply (simp add: leadsETo_UN extend_set_Union)
done

```

```

lemma 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)"
  apply (subst Int_extend_set_lemma [symmetric])
  apply (rule Join_project_ensures)
  apply (auto simp add: Int_Diff)
done

```

```

lemma 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"
  apply (rule psp_stable2 [THEN leadsTo_weaken_L])
  apply (auto simp add: project_stable_project_set extend_stable_project_set)
done

```

```

lemma 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)"
  apply (erule leadsETo_induct)
  prefer 3
  apply (simp only: Int_UN_distrib project_set_Union)
  apply (blast intro: leadsTo_UN)
  prefer 2 apply (blast intro: leadsTo_Trans pli_lemma)
  apply (simp add: givenBy_eq_extend_set)
  apply (rule leadsTo_Basis)
  apply (blast intro: ensures_extend_set_imp_project_ensures)
done

```

```

lemma 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"
  apply (rule project_leadsETo_I_lemma [THEN leadsTo_weaken], auto)
done

```

```

lemma 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"
  apply (simp (no_asm_use) add: LeadsTo_def LeadsETO_def)
  apply (rule project_leadsETO_I_lemma [THEN leadsTo_weaken])
  apply (auto simp add: project_set_reachable_extend_eq [symmetric])
done

lemma projecting_leadsTo:
  "projecting (λG. UNIV) h F
    (extend_set h A leadsTo[givenBy f] extend_set h B)
    (A leadsTo B)"
  apply (unfold projecting_def)
  apply (force dest: project_leadsETO_I)
done

lemma projecting_LeadsTo:
  "projecting (λG. reachable (extend h F ⊔ G)) h F
    (extend_set h A LeadsTo[givenBy f] extend_set h B)
    (A LeadsTo B)"
  apply (unfold projecting_def)
  apply (force dest: project_LeadsETO_I)
done

end

end

```