A Formal Model of Extended Finite State Machines

Michael Foster* Achim D. Brucker†
Ramsay G. Taylor* John Derrick*

February 23, 2021

*Department of Computer Science, The University of Sheffield, Sheffield, UK
{jmafoster1,r.g.taylor,j.derrick}@sheffield.ac.uk

†Department of Computer Science, University of Exeter, Exeter, UK
a.brucker@exeter.ac.uk

Abstract

In this AFP entry, we provide a formalisation of extended finite state machines (EFSMs) where models are represented as finite sets of transitions between states. EFSMs execute traces to produce observable outputs. We also define various simulation and equality metrics for EFSMs in terms of traces and prove their strengths in relation to each other. Another key contribution is a framework of function definitions such that LTL properties can be phrased over EFSMs. Finally, we provide a simple example case study in the form of a drinks machine.

Keywords: Extended Finite State Machines, Automata, Linear Temporal Logic

Contents

1 Introduction

This AFP entry formalises extended finite state machines (EFSMs) as defined in [2]. Here, models maintain both a *control flow state* and a *data state*, which takes the form of a set of *registers* to which values may be assigned. Transitions may take additional input parameters, and may impose guard conditions on the values of both inputs and registers. Additionally, transitions may produce observable outputs and update the data state by evaluating arithmetic functions over inputs and registers.

As defined in [2], an EFSM is a tuple, (S, s_0, T) where

S is a finite non-empty set of states.

$s_0 \in S$ is the initial state.

T is the transition matrix $T : (S \times S) \to \mathcal{P}(L \times \mathbb{N} \times G \times F \times U)$ with rows representing origin states and columns representing destination states.

In T

L is a finite set of transition labels

$\mathbb{N}$ gives the transition *arity* (the number of input parameters), which may be zero.

G is a finite set of Boolean guard functions $G : (I \times R) \to \mathbb{B}$.

F is a finite set of *output functions* $F : (I \times R) \to O$.

U is a finite set of *update functions* $U : (I \times R) \to R$.

In G, F, and U

I is a list $[i_0, i_1, \ldots, i_{m-1}]$ of values representing the inputs of a transition, which is empty if the arity is zero.

R is a mapping from variables $[r_0, r_1, \ldots]$, representing each register of the machine, to their values.

O is a list $[o_0, o_1, \ldots, o_{n-1}]$ of values, which may be empty, representing the outputs of a transition.

EFSM transitions have five components: label, arity, guards, outputs, and updates. Transition labels are strings, and the arities natural numbers. Guards have a defined type of *guard expression* (`gexp`) and the outputs and updates are defined using *arithmetic expressions* (`aexp`). Outputs are simply a list of expressions to be evaluated. Updates are a list of pairs with the first element being the index of the register to be updated, and the second element being an arithmetic expression to be evaluated.

The rest of this document is automatically generated from the formalization in Isabelle/HOL, i.e., all content is checked by Isabelle. Overall, the structure of this document follows the theory dependencies (see Figure 1.1):

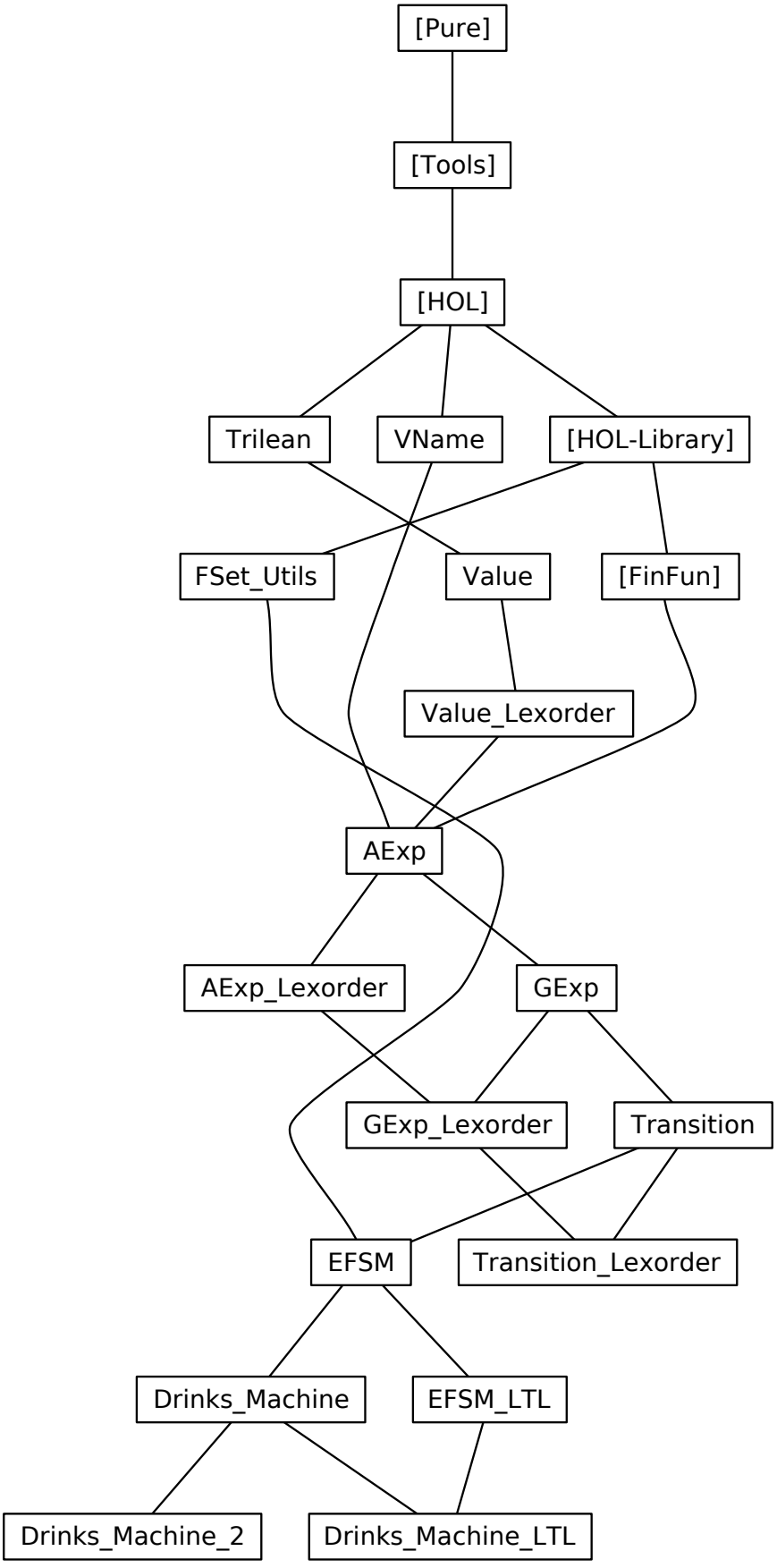

Figure 1.1: The Dependency Graph of the Isabelle Theories.

2 Preliminaries

In this chapter, we introduce the preliminaries, including a three-valued logic, variables, arithmetic expressions and guard expressions.

2.1 Three-Valued Logic (Trilean)

Because our EFSMs are dynamically typed, we cannot rely on conventional Boolean logic when evaluating expressions. For example, we may end up in the situation where we need to evaluate the guard $r_1 > 5$. This is fine if r_1 holds a numeric value, but if r_1 evaluates to a string, this causes problems. We cannot simply evaluate to *false* because then the negation would evaluate to *true.* Instead, we need a three-valued logic such that we can meaningfully evaluate nonsensical guards.

The `trilean` datatype is used to implement three-valued Bochvar logic [1]. Here we prove that the logic is an idempotent semiring, define a partial order, and prove some other useful lemmas.

```
theory Trilean
imports Main
begin

datatype trilean = true | false | invalid

instantiation trilean :: semiring begin
fun times_trilean :: "trilean ⇒ trilean ⇒ trilean" where
  "times_trilean _ invalid = invalid" |
  "times_trilean invalid _ = invalid" |
  "times_trilean true true = true" |
  "times_trilean _ false = false" |
  "times_trilean false _ = false"

fun plus_trilean :: "trilean ⇒ trilean ⇒ trilean" where
  "plus_trilean invalid _ = invalid" |
  "plus_trilean _ invalid = invalid" |
  "plus_trilean true _ = true" |
  "plus_trilean _ true = true" |
  "plus_trilean false false = false"

abbreviation maybe_and :: "trilean ⇒ trilean ⇒ trilean" (infixl "∧?" 70) where
  "maybe_and x y ≡ x * y"

abbreviation maybe_or :: "trilean ⇒ trilean ⇒ trilean" (infixl "∨?" 65) where
  "maybe_or x y ≡ x + y"

lemma plus_trilean_assoc:
  "a ∨? b ∨? c = a ∨? (b ∨? c)"
proof(induct a b  arbitrary: c rule: plus_trilean.induct)
case (1 uu)
  then show ?case
    by simp
next
  case "2_1"
  then show ?case
    by simp
next
  case "2_2"
  then show ?case
    by simp
```

```
next
  case "3_1"
  then show ?case
    by (metis plus_trilean.simps(2) plus_trilean.simps(4) trilean.exhaust)
next
  case "3_2"
  then show ?case
    by (metis plus_trilean.simps(3) plus_trilean.simps(5) plus_trilean.simps(6) plus_trilean.simps(7) trilean.exha
next
  case 4
  then show ?case
    by (metis plus_trilean.simps(2) plus_trilean.simps(3) plus_trilean.simps(4) plus_trilean.simps(5) plus_trilean
trilean.exhaust)
next
  case 5
  then show ?case
    by (metis plus_trilean.simps(6) plus_trilean.simps(7) trilean.exhaust)
qed

lemma plus_trilean_commutative: "a ∨? b = b ∨? a"
proof(induct a b rule: plus_trilean.induct)
  case (1 uu)
  then show ?case
    by (metis plus_trilean.simps(1) plus_trilean.simps(2) plus_trilean.simps(3) trilean.exhaust)
qed auto

lemma times_trilean_commutative: "a ∧? b = b ∧? a"
  by (metis (mono_tags) times_trilean.simps trilean.distinct(5) trilean.exhaust)

lemma times_trilean_assoc:
  "a ∧? b ∧? c = a ∧? (b ∧? c)"
proof(induct a b  arbitrary: c rule: plus_trilean.induct)
  case (1 uu)
  then show ?case
    by (metis (mono_tags, lifting) times_trilean.simps(1) times_trilean_commutative)
next
case "2_1"
  then show ?case
    by (metis (mono_tags, lifting) times_trilean.simps(1) times_trilean_commutative)
next
  case "2_2"
  then show ?case
    by (metis (mono_tags, lifting) times_trilean.simps(1) times_trilean_commutative)
next
  case "3_1"
  then show ?case
    by (metis times_trilean.simps(1) times_trilean.simps(4) times_trilean.simps(5) trilean.exhaust)
next
  case "3_2"
  then show ?case
    by (metis times_trilean.simps(1) times_trilean.simps(5) times_trilean.simps(6) times_trilean.simps(7)
trilean.exhaust)
next
  case 4
  then show ?case
    by (metis times_trilean.simps(1) times_trilean.simps(4) times_trilean.simps(5) times_trilean.simps(7)
trilean.exhaust)
next
case 5
  then show ?case
    by (metis (full_types) times_trilean.simps(1) times_trilean.simps(6) times_trilean.simps(7) trilean.exhaust)
qed
```

```
lemma trilean_distributivity_1:
  "(a ∨? b) ∧? c = a ∧? c ∨? b ∧? c"
proof(induct a b rule: times_trilean.induct)
case (1 uu)
  then show ?case
    by (metis (mono_tags, lifting) plus_trilean.simps(1) plus_trilean_commutative times_trilean.simps(1)
times_trilean_commutative)
next
  case "2_1"
  then show ?case
    by (metis (mono_tags, lifting) plus_trilean.simps(1) times_trilean.simps(1) times_trilean_commutative)
next
  case "2_2"
  then show ?case
    by (metis (mono_tags, lifting) plus_trilean.simps(1) times_trilean.simps(1) times_trilean_commutative)
next
  case 3
  then show ?case
    apply simp
    by (metis (no_types, hide_lams) plus_trilean.simps(1) plus_trilean.simps(4) plus_trilean.simps(7) times_trilea
times_trilean.simps(4) times_trilean.simps(5) trilean.exhaust)
next
  case "4_1"
  then show ?case
    apply simp
    by (metis (no_types, hide_lams) plus_trilean.simps(1) plus_trilean.simps(5) plus_trilean.simps(7) times_trilea
times_trilean.simps(4) times_trilean.simps(5) times_trilean.simps(6) times_trilean.simps(7) trilean.exhaust)
next
  case "4_2"
  then show ?case
    apply simp
    by (metis (no_types, hide_lams) plus_trilean.simps(1) plus_trilean.simps(7) times_trilean.simps(1) times_trile
times_trilean.simps(7) trilean.exhaust)
next
  case 5
  then show ?case
    apply simp
    by (metis (no_types, hide_lams) plus_trilean.simps(1) plus_trilean.simps(6) plus_trilean.simps(7) times_trilea
times_trilean.simps(4) times_trilean.simps(5) times_trilean.simps(6) times_trilean.simps(7) trilean.exhaust)
qed

instance
  apply standard
      apply (simp add: plus_trilean_assoc)
     apply (simp add: plus_trilean_commutative)
    apply (simp add: times_trilean_assoc)
   apply (simp add: trilean_distributivity_1)
  using times_trilean_commutative trilean_distributivity_1 by auto
end

lemma maybe_or_idempotent: "a ∨? a = a"
  by (cases a) auto

lemma maybe_and_idempotent: "a ∧? a = a"
  by (cases a) auto

instantiation trilean :: ord begin
definition less_eq_trilean :: "trilean ⇒ trilean ⇒ bool" where
  "less_eq_trilean a b = (a + b = b)"

definition less_trilean :: "trilean ⇒ trilean ⇒ bool" where
  "less_trilean a b = (a ≤ b ∧ a ≠ b)"
```

```
declare less_trilean_def less_eq_trilean_def [simp]

instance
  by standard
end

instantiation trilean :: uminus begin
  fun maybe_not :: "trilean ⇒ trilean" ("¬? _" [60] 60) where
    "¬? true = false" |
    "¬? false = true" |
    "¬? invalid = invalid"

instance
  by standard
end

lemma maybe_and_one: "true ∧? x = x"
  by (cases x, auto)

lemma maybe_or_zero: "false ∨? x = x"
  by (cases x, auto)

lemma maybe_double_negation: "¬? ¬? x = x"
  by (cases x, auto)

lemma maybe_negate_true: "(¬? x = true) = (x = false)"
  by (cases x, auto)

lemma maybe_negate_false: "(¬? x = false) = (x = true)"
  by (cases x, auto)

lemma maybe_and_true: "(x ∧? y = true) = (x = true ∧ y = true)"
  using times_trilean.elims by blast

lemma maybe_and_not_true:
  "(x ∧? y ≠ true) = (x ≠ true ∨ y ≠ true)"
  by (simp add: maybe_and_true)

lemma negate_valid: "(¬? x ≠ invalid) = (x ≠ invalid)"
  by (metis maybe_double_negation maybe_not.simps(3))

lemma maybe_and_valid:
  "x ∧? y ≠ invalid ⟹ x ≠ invalid ∧ y ≠ invalid"
  using times_trilean.elims by blast

lemma maybe_or_valid:
  "x ∨? y ≠ invalid ⟹ x ≠ invalid ∧ y ≠ invalid"
  using plus_trilean.elims by blast

lemma maybe_or_false:
  "(x ∨? y = false) = (x = false ∧ y = false)"
  using plus_trilean.elims by blast

lemma maybe_or_true:
  "(x ∨? y = true) = ((x = true ∨ y = true) ∧ x ≠ invalid ∧ y ≠ invalid)"
  using plus_trilean.elims by blast

lemma maybe_not_invalid: "(¬? x = invalid) = (x = invalid)"
  by (metis maybe_double_negation maybe_not.simps(3))

lemma maybe_or_invalid:
  "(x ∨? y = invalid) = (x = invalid ∨ y = invalid)"
  using plus_trilean.elims by blast
```

```
lemma maybe_and_invalid:
  "(x ∧? y = invalid) = (x = invalid ∨ y = invalid)"
  using times_trilean.elims by blast

lemma maybe_and_false:
  "(x ∧? y = false) = ((x = false ∨ y = false) ∧ x ≠ invalid ∧ y ≠ invalid)"
  using times_trilean.elims by blast

lemma invalid_maybe_and: "invalid ∧? x = invalid"
  using maybe_and_valid by blast

lemma maybe_not_eq: "(¬? x = ¬? y) = (x = y)"
  by (metis maybe_double_negation)

lemma de_morgans_1:
  "¬? (a ∨? b) = (¬?a) ∧? (¬?b)"
  by (metis (no_types, hide_lams) add.commute invalid_maybe_and maybe_and_idempotent maybe_and_one maybe_not.elims
maybe_not.simps(1) maybe_not.simps(3) maybe_not_invalid maybe_or_zero plus_trilean.simps(1) plus_trilean.simps(4)
times_trilean.simps(1) times_trilean_commutative trilean.exhaust trilean.simps(6))

lemma de_morgans_2:
  "¬? (a ∧? b) = (¬?a) ∨? (¬?b)"
  by (metis de_morgans_1 maybe_double_negation)

lemma not_true: "(x ≠ true) = (x = false ∨ x = invalid)"
  by (metis (no_types, lifting) maybe_not.cases trilean.distinct(1) trilean.distinct(3))

lemma pull_negation: "(x = ¬? y) = (¬? x = y)"
  using maybe_double_negation by auto

lemma comp_fun_commute_maybe_or: "comp_fun_commute maybe_or"
  apply standard
  apply (simp add: comp_def)
  apply (rule ext)
  by (simp add: add.left_commute)

lemma comp_fun_commute_maybe_and: "comp_fun_commute maybe_and"
  apply standard
  apply (simp add: comp_def)
  apply (rule ext)
  by (metis add.left_commute de_morgans_2 maybe_not_eq)

end
```

2.2 Values (Value)

Our EFSM implementation can currently handle integers and strings. Here we define a sum type which combines these. We also define an arithmetic in terms of values such that EFSMs do not need to be strongly typed.

```
theory Value
imports Trilean
begin
datatype "value" = Num int | Str String.literal

fun is_Num :: "value ⇒ bool" where
  "is_Num (Num _) = True" |
  "is_Num (Str _) = False"
fun maybe_arith_int :: "(int ⇒ int ⇒ int) ⇒ value option ⇒ value option ⇒ value option" where
  "maybe_arith_int f (Some (Num x)) (Some (Num y)) = Some (Num (f x y))" |
  "maybe_arith_int _ _ _ = None"
```

```
lemma maybe_arith_int_not_None:
  "maybe_arith_int f a b ≠ None = (∃n n'. a = Some (Num n) ∧ b = Some (Num n'))"
  using maybe_arith_int.elims maybe_arith_int.simps(1) by blast

lemma maybe_arith_int_Some:
  "maybe_arith_int f a b = Some (Num x) = (∃n n'. a = Some (Num n) ∧ b = Some (Num n') ∧ f n n' = x)"
  using maybe_arith_int.elims maybe_arith_int.simps(1) by blast

lemma maybe_arith_int_None:
  "(maybe_arith_int f a1 a2 = None) = (∄n n'. a1 = Some (Num n) ∧ a2 = Some (Num n'))"
  using maybe_arith_int_not_None by blast

lemma maybe_arith_int_Not_Num:
  "(∀n. maybe_arith_int f a1 a2 ≠ Some (Num n)) = (maybe_arith_int f a1 a2 = None)"
  by (metis maybe_arith_int.elims option.distinct(1))

lemma maybe_arith_int_never_string: "maybe_arith_int f a b ≠ Some (Str x)"
  using maybe_arith_int.elims by blast

definition "value_plus = maybe_arith_int (+)"

lemma value_plus_never_string: "value_plus a b ≠ Some (Str x)"
  by (simp add: value_plus_def maybe_arith_int_never_string)

lemma value_plus_symmetry: "value_plus x y = value_plus y x"
  apply (induct x y rule: maybe_arith_int.induct)
  by (simp_all add: value_plus_def)

definition "value_minus = maybe_arith_int (-)"

lemma value_minus_never_string: "value_minus a b ≠ Some (Str x)"
  by (simp add: maybe_arith_int_never_string value_minus_def)

definition "value_times = maybe_arith_int (*)"

lemma value_times_never_string: "value_times a b ≠ Some (Str x)"
  by (simp add: maybe_arith_int_never_string value_times_def)

fun MaybeBoolInt :: "(int ⇒ int ⇒ bool) ⇒ value option ⇒ value option ⇒ trilean" where
  "MaybeBoolInt f (Some (Num a)) (Some (Num b)) = (if f a b then true else false)" |
  "MaybeBoolInt _ _ _ = invalid"

lemma MaybeBoolInt_not_num_1:
  "∀n. r ≠ Some (Num n) ⟹ MaybeBoolInt f n r = invalid"
  using MaybeBoolInt.elims by blast

definition value_gt :: "value option ⇒ value option ⇒ trilean"  where
  "value_gt a b ≡ MaybeBoolInt (>) a b"

fun value_eq :: "value option ⇒ value option ⇒ trilean" where
  "value_eq None _ = invalid" |
  "value_eq _ None = invalid" |
  "value_eq (Some a) (Some b) = (if a = b then true else false)"

lemma value_eq_true: "(value_eq a b = true) = (∃x y. a = Some x ∧ b = Some y ∧ x = y)"
  by (cases a; cases b, auto)

lemma value_eq_false: "(value_eq a b = false) = (∃x y. a = Some x ∧ b = Some y ∧ x ≠ y)"
  by (cases a; cases b, auto)

lemma value_gt_true_Some: "value_gt a b = true ⟹ (∃x. a = Some x) ∧ (∃y. b = Some y)"
  by (cases a; cases b, auto simp: value_gt_def)
```

```
lemma value_gt_true: "(value_gt a b = true) = (∃ x y. a = Some (Num x) ∧ b = Some (Num y) ∧ x > y)"
  apply (cases a)
  using value_gt_true_Some apply blast
  apply (cases b)
  using value_gt_true_Some apply blast
  subgoal for aa bb by (cases aa; cases bb, auto simp: value_gt_def)
  done

lemma value_gt_false_Some: "value_gt a b = false ⟹ (∃ x. a = Some x) ∧ (∃ y. b = Some y)"
  by (cases a; cases b, auto simp: value_gt_def)

end
```

2.3 Variables (VName)

Variables can either be inputs or registers. Here we define the vname datatype which allows us to write expressions in terms of variables and case match during evaluation. We also make the vname datatype a member of linorder such that we can establish a linear order on arithmetic expressions, guards, and subsequently transitions.

```
theory VName
imports Main
begin
datatype vname = I nat | R nat

instantiation vname :: linorder begin
fun less_vname :: "vname ⇒ vname ⇒ bool" where
  "(I n1) < (R n2) = True" |
  "(R n1) < (I n2) = False" |
  "(I n1) < (I n2) = (n1 < n2)" |
  "(R n1) < (R n2) = (n1 < n2)"

definition less_eq_vname :: "vname ⇒ vname ⇒ bool" where
  "less_eq_vname v1 v2 = (v1 < v2 ∨ v1 = v2)"
declare less_eq_vname_def [simp]

instance
  apply standard
      apply (metis (full_types) dual_order.asym less_eq_vname_def less_vname.simps(2) less_vname.simps(3)
less_vname.simps(4) vname.exhaust)
     apply simp
  subgoal for x y z
    apply (induct x y rule: less_vname.induct)
       apply (metis less_eq_vname_def less_vname.elims(2) less_vname.elims(3) vname.simps(4))
      apply simp
     apply (metis less_eq_vname_def less_trans less_vname.elims(3) less_vname.simps(3) vname.simps(4))
    by (metis le_less_trans less_eq_vname_def less_imp_le_nat less_vname.elims(2) less_vname.simps(4) vname.simps(
   apply (metis dual_order.asym less_eq_vname_def less_vname.elims(2) less_vname.simps(3) less_vname.simps(4))
   subgoal for x y
     by (induct x y rule: less_vname.induct, auto)
   done
end

end
```

2.3.1 Value Lexorder

This theory defines a lexicographical ordering on values such that we can build orderings for arithmetic expressions and guards. Here, numbers are defined as less than strings, else the natural ordering on the respective datatypes is used.

```
theory Value_Lexorder
imports Value
begin
```

```
instantiation "value" :: linorder begin
fun less_value :: "value ⇒ value ⇒ bool" where
  "(Num n) < (Str s) = True" |
  "(Str s) < (Num n) = False" |
  "(Str s1) < (Str s2) = (s1 < s2)" |
  "(Num n1) < (Num n2) = (n1 < n2)"

definition less_eq_value :: "value ⇒ value ⇒ bool" where
  "less_eq_value v1 v2 = (v1 < v2 ∨ v1 = v2)"
declare less_eq_value_def [simp]

instance
  apply standard
      apply (simp, metis less_imp_not_less less_value.elims(2) less_value.simps(3) less_value.simps(4))
     apply simp
  subgoal for x y z
    apply (induct x y rule: less_value.induct)
       apply (metis less_eq_value_def less_value.elims(3) less_value.simps(2) value.simps(4))
      apply simp
     apply (metis dual_order.strict_trans less_eq_value_def less_value.elims(2) less_value.simps(3) value.distinct
    by (cases z, auto)
   apply (metis less_eq_value_def less_imp_not_less less_value.elims(2) less_value.simps(3) less_value.simps(4))
  subgoal for x y
    by (induct x y rule: less_value.induct, auto)
  done

end

end
```

2.4 Arithmetic Expressions (AExp)

This theory defines a language of arithmetic expressions over variables and literal values. Here, values are limited to integers and strings. Variables may be either inputs or registers. We also limit ourselves to a simple arithmetic of addition, subtraction, and multiplication as a proof of concept.

```
theory AExp
  imports Value_Lexorder VName FinFun.FinFun "HOL-Library.Option_ord"
begin

declare One_nat_def [simp del]
unbundle finfun_syntax

type_synonym registers = "nat ⇒f value option"
type_synonym 'a datastate = "'a ⇒ value option"
datatype 'a aexp = L "value" | V 'a | Plus "'a aexp" "'a aexp" | Minus "'a aexp" "'a aexp" | Times "'a aexp"
"'a aexp"

fun is_lit :: "'a aexp ⇒ bool" where
  "is_lit (L _) = True" |
  "is_lit _ = False"

lemma aexp_induct_separate_V_cases  [case_names L I R Plus Minus Times]:
  "(⋀x. P (L x)) ⟹
   (⋀x. P (V (I x))) ⟹
   (⋀x. P (V (R x))) ⟹
   (⋀x1a x2a. P x1a ⟹ P x2a ⟹ P (Plus x1a x2a)) ⟹
   (⋀x1a x2a. P x1a ⟹ P x2a ⟹ P (Minus x1a x2a)) ⟹
   (⋀x1a x2a. P x1a ⟹ P x2a ⟹ P (Times x1a x2a)) ⟹
   P a"
  by (metis aexp.induct vname.exhaust)
```

```
fun aval :: "'a aexp ⇒ 'a datastate ⇒ value option" where
  "aval (L x) s = Some x" |
  "aval (V x) s = s x" |
  "aval (Plus a1 a2) s = value_plus (aval a1 s)(aval a2 s)" |
  "aval (Minus a1 a2) s = value_minus (aval a1 s) (aval a2 s)" |
  "aval (Times a1 a2) s = value_times (aval a1 s) (aval a2 s)"

lemma aval_plus_symmetry: "aval (Plus x y) s = aval (Plus y x) s"
  by (simp add: value_plus_symmetry)
```

A little syntax magic to write larger states compactly:

```
definition null_state ("<>") where
  "null_state ≡ (K$ bot)"

no_notation finfun_update ("_'(_ $:= _')" [1000, 0, 0] 1000)
nonterminal fupdbinds and fupdbind

syntax
  "_fupdbind" :: "'a ⇒ 'a ⇒ fupdbind"             ("(2_ $:=/ _)")
  ""          :: "fupdbind ⇒ fupdbinds"           ("_")
  "_fupdbinds":: "fupdbind ⇒ fupdbinds ⇒ fupdbinds" ("_,/ _")
  "_fUpdate"  :: "'a ⇒ fupdbinds ⇒ 'a"            ("_/'((_)')" [1000, 0] 900)
  "_State" :: "fupdbinds => 'a" ("<_>")

translations
  "_fUpdate f (_fupdbinds b bs)" ⇌ "_fUpdate (_fUpdate f b) bs"
  "f(x$:=y)" ⇌ "CONST finfun_update f x y"
  "_State ms" == "_fUpdate <> ms"
  "_State (_updbinds b bs)" <= "_fUpdate (_State b) bs"

lemma empty_None: "<> = (K$ None)"
  by (simp add: null_state_def bot_option_def)

lemma apply_empty_None [simp]: "<> $ x2 = None"
  by (simp add: null_state_def bot_option_def)

definition input2state :: "value list ⇒ registers" where
  "input2state n = fold (λ(k, v) f. f(k $:= Some v)) (enumerate 0 n) (K$ None)"

primrec input2state_prim :: "value list ⇒ nat ⇒ registers" where
  "input2state_prim [] _ = (K$ None)" |
  "input2state_prim (v#t) k = (input2state_prim t (k+1))(k $:= Some v)"

lemma input2state_append:
  "input2state (i @ [a]) = (input2state i)(length i $:= Some a)"
  apply (simp add: eq_finfun_All_ext finfun_All_def finfun_All_except_def)
  apply clarify
  by (simp add: input2state_def enumerate_eq_zip)

lemma input2state_out_of_bounds:
  "i ≥ length ia ⟹ input2state ia $ i = None"
proof(induct ia rule: rev_induct)
  case Nil
  then show ?case
    by (simp add: input2state_def)
next
  case (snoc a as)
  then show ?case
    by (simp add: input2state_def enumerate_eq_zip)
qed

lemma input2state_within_bounds:
```

```
  "input2state i $ x = Some a ⟹ x < length i"
  by (metis input2state_out_of_bounds not_le_imp_less option.distinct(1))

lemma input2state_empty: "input2state [] $ x1 = None"
  by (simp add: input2state_out_of_bounds)

lemma input2state_nth:
  "i < length ia ⟹ input2state ia $ i = Some (ia ! i)"
proof(induct ia rule: rev_induct)
  case Nil
  then show ?case
    by simp
next
  case (snoc a ia)
  then show ?case
    apply (simp add: input2state_def enumerate_eq_zip)
    by (simp add: finfun_upd_apply nth_append)
qed

lemma input2state_some:
  "i < length ia ⟹
   ia ! i = x ⟹
   input2state ia $ i = Some x"
  by (simp add: input2state_nth)

lemma input2state_take: "x1 < A ⟹
   A ≤ length i ⟹
   x = vname.I x1 ⟹
   input2state i $ x1 = input2state (take A i) $ x1"
proof(induct i)
  case Nil
  then show ?case
    by simp
next
  case (Cons a i)
  then show ?case
    by (simp add: input2state_nth)
qed

lemma input2state_not_None:
  "(input2state i $ x ≠ None) ⟹ (x < length i)"
  using input2state_within_bounds by blast

lemma input2state_Some:
  "(∃v. input2state i $ x = Some v) = (x < length i)"
  apply standard
  using input2state_within_bounds apply blast
  by (simp add: input2state_nth)

lemma input2state_cons: "x1 > 0 ⟹
   x1 < length ia ⟹
   input2state (a # ia) $ x1 = input2state ia $ (x1-1)"
  by (simp add: input2state_nth)

lemma input2state_cons_shift:
  "input2state i $ x1 = Some a ⟹ input2state (b # i) $ (Suc x1) = Some a"
proof(induct i rule: rev_induct)
  case Nil
  then show ?case
    by (simp add: input2state_def)
next
  case (snoc x xs)
  then show ?case
```

```
    using input2state_within_bounds[of xs x1 a]
    using input2state_cons[of "Suc x1" "xs @ [x]" b]
    apply simp
    apply (case_tac "x1 < length xs")
     apply simp
    by (metis finfun_upd_apply input2state_append input2state_nth length_Cons length_append_singleton lessI
list.sel(3) nth_tl)
qed

lemma input2state_exists: "∃ i. input2state i $ x1 = Some a"
proof(induct x1)
  case 0
  then show ?case
    apply (rule_tac x="[a]" in exI)
    by (simp add: input2state_def)
next
  case (Suc x1)
  then show ?case
    apply clarify
    apply (rule_tac x="a#i" in exI)
    by (simp add: input2state_cons_shift)
qed

primrec repeat :: "nat ⇒ 'a ⇒ 'a list" where
  "repeat 0 _ = []" |
  "repeat (Suc m) a = a#(repeat m a)"

lemma length_repeat: "length (repeat n a) = n"
proof(induct n)
  case 0
  then show ?case
    by simp
next
  case (Suc a)
  then show ?case
    by simp
qed

lemma length_append_repeat: "length (i@(repeat a y)) ≥ length i"
  by simp

lemma length_input2state_repeat:
  "input2state i $ x = Some a ⟹ y < length (i @ repeat y a)"
  by (metis append.simps(1) append_eq_append_conv input2state_within_bounds length_append length_repeat
list.size(3) neqE not_add_less2 zero_order(3))

lemma input2state_double_exists:
  "∃ i. input2state i $ x = Some a ∧ input2state i $ y = Some a"
  apply (insert input2state_exists[of x a])
  apply clarify
  apply (case_tac "x ≥ y")
  apply (rule_tac x="list_update i y a" in exI)
  apply (metis (no_types, lifting) input2state_within_bounds input2state_nth input2state_out_of_bounds le_trans
length_list_update not_le_imp_less nth_list_update_eq nth_list_update_neq)
  apply (rule_tac x="list_update (i@(repeat y a)) y a" in exI)
  apply (simp add: not_le)
  by (metis length_input2state_repeat input2state_nth input2state_out_of_bounds le_trans length_append_repeat
length_list_update not_le_imp_less nth_append nth_list_update_eq nth_list_update_neq option.distinct(1))

lemma input2state_double_exists_2:
  "x ≠ y ⟹ ∃ i. input2state i $ x = Some a ∧ input2state i $ y = Some a'"
  apply (insert input2state_exists[of x a])
  apply clarify
```

```
  apply (case_tac "x ≥ y")
  apply (rule_tac x="list_update i y a'" in exI)
  apply (metis (no_types, lifting) input2state_within_bounds input2state_nth input2state_out_of_bounds le_trans
length_list_update not_le_imp_less nth_list_update_eq nth_list_update_neq)
  apply (rule_tac x="list_update (i@(repeat y a')) y a'" in exI)
  apply (simp add: not_le)
  apply standard
  apply (metis input2state_nth input2state_within_bounds le_trans length_append_repeat length_list_update
linorder_not_le nth_append nth_list_update_neq order_refl)
  by (metis input2state_nth length_append length_input2state_repeat length_list_update length_repeat nth_list_upda

definition join_ir :: "value list ⇒ registers ⇒ vname datastate" where
  "join_ir i r ≡ (λx. case x of
    R n ⇒ r $ n |
    I n ⇒ (input2state i) $ n
  )"

lemmas datastate = join_ir_def input2state_def

lemma join_ir_empty [simp]: "join_ir [] <> = (λx. None)"
  apply (rule ext)
  apply (simp add: join_ir_def)
  apply (case_tac x)
   apply (simp add: input2state_def)
  by (simp add: empty_None)

lemma join_ir_R [simp]: "(join_ir i r) (R n) = r $ n"
  by (simp add: join_ir_def)

lemma join_ir_double_exists:
  "∃i r. join_ir i r v = Some a ∧ join_ir i r v' = Some a"
proof(cases v)
  case (I x1)
  then show ?thesis
    apply (simp add: join_ir_def)
    apply (cases v')
     apply (simp add: input2state_double_exists input2state_exists)
    using input2state_exists by auto
next
  case (R x2)
  then show ?thesis
    apply (simp add: join_ir_def)
    apply (cases v')
    using input2state_exists apply force
    using input2state_double_exists by fastforce
qed

lemma join_ir_double_exists_2:
  "v ≠ v' ⟹ ∃i r. join_ir i r v = Some a ∧ join_ir i r v' = Some a'"
proof(cases v)
  case (I x1)
  assume "v ≠ v'"
  then show ?thesis using I input2state_exists
    by (cases v', auto simp add: join_ir_def input2state_double_exists_2)
next
  case (R x2)
  assume "v ≠ v'"
  then show ?thesis
    apply (simp add: join_ir_def)
    apply (cases v')
     apply simp
    using R input2state_exists apply auto[1]
    apply (simp add: R)
```

```
    apply (rule_tac x="<x2 $:= Some a,x2a $:= Some a'>" in exI)
    by simp
qed

lemma exists_join_ir_ext: "∃i r. join_ir i r v = s v"
  apply (simp add: join_ir_def)
  apply (case_tac "s v")
   apply (cases v)
    apply (rule_tac x="[]" in exI)
    apply (simp add: input2state_out_of_bounds)
   apply simp
   apply (rule_tac x="<>" in exI)
   apply simp
  apply simp
  apply (cases v)
   apply simp
   apply (simp add: input2state_exists)
  apply simp
  apply (rule_tac x="<x2 $:= Some a>" in exI)
  apply simp
  done

lemma join_ir_nth [simp]:
  "i < length is ⟹ join_ir is r (I i) = Some (is ! i)"
  by (simp add: join_ir_def input2state_nth)

fun aexp_constrains :: "'a aexp ⇒ 'a aexp ⇒ bool" where
  "aexp_constrains (L l) a = (L l = a)" |
  "aexp_constrains (V v) v' = (V v = v')" |
  "aexp_constrains (Plus a1 a2) v = ((Plus a1 a2) = v ∨ (Plus a1 a2) = v ∨ (aexp_constrains a1 v ∨ aexp_constrains
a2 v))" |
  "aexp_constrains (Minus a1 a2) v = ((Minus a1 a2) = v ∨ (aexp_constrains a1 v ∨ aexp_constrains a2 v))"
|
  "aexp_constrains (Times a1 a2) v = ((Times a1 a2) = v ∨ (aexp_constrains a1 v ∨ aexp_constrains a2 v))"

fun aexp_same_structure :: "'a aexp ⇒ 'a aexp ⇒ bool" where
  "aexp_same_structure (L v) (L v') = True" |
  "aexp_same_structure (V v) (V v') = True" |
  "aexp_same_structure (Plus a1 a2) (Plus a1' a2') = (aexp_same_structure a1 a1' ∧ aexp_same_structure a2
a2')" |
  "aexp_same_structure (Minus a1 a2) (Minus a1' a2') = (aexp_same_structure a1 a1' ∧ aexp_same_structure
a2 a2')" |
  "aexp_same_structure _ _ = False"

fun enumerate_aexp_inputs :: "vname aexp ⇒ nat set" where
  "enumerate_aexp_inputs (L _) = {}" |
  "enumerate_aexp_inputs (V (I n)) = {n}" |
  "enumerate_aexp_inputs (V (R n)) = {}" |
  "enumerate_aexp_inputs (Plus v va) = enumerate_aexp_inputs v ∪ enumerate_aexp_inputs va" |
  "enumerate_aexp_inputs (Minus v va) = enumerate_aexp_inputs v ∪ enumerate_aexp_inputs va" |
  "enumerate_aexp_inputs (Times v va) = enumerate_aexp_inputs v ∪ enumerate_aexp_inputs va"

lemma enumerate_aexp_inputs_list: "∃l. enumerate_aexp_inputs a = set l"
proof(induct a)
  case (L x)
  then show ?case
    by simp
next
  case (V x)
  then show ?case
    apply (cases x)
     apply (metis empty_set enumerate_aexp_inputs.simps(2) list.simps(15))
    by simp
```

```
next
  case (Plus a1 a2)
  then show ?case
    by (metis enumerate_aexp_inputs.simps(4) set_append)
next
  case (Minus a1 a2)
  then show ?case
    by (metis enumerate_aexp_inputs.simps(5) set_append)
next
  case (Times a1 a2)
  then show ?case
    by (metis enumerate_aexp_inputs.simps(6) set_append)
qed

fun enumerate_regs :: "vname aexp ⇒ nat set" where
  "enumerate_regs (L _) = {}" |
  "enumerate_regs (V (R n)) = {n}" |
  "enumerate_regs (V (I _)) = {}" |
  "enumerate_regs (Plus v va) = enumerate_regs v ∪ enumerate_regs va" |
  "enumerate_regs (Minus v va) = enumerate_regs v ∪ enumerate_regs va" |
  "enumerate_regs (Times v va) = enumerate_regs v ∪ enumerate_regs va"

lemma finite_enumerate_regs: "finite (enumerate_regs a)"
  by(induct a rule: aexp_induct_separate_V_cases, auto)

lemma no_variables_aval: "enumerate_aexp_inputs a = {} ⟹
   enumerate_regs a = {} ⟹
   aval a s = aval a s'"
  by (induct a rule: aexp_induct_separate_V_cases, auto)

lemma enumerate_aexp_inputs_not_empty:
  "(enumerate_aexp_inputs a ≠ {}) = (∃ b c. enumerate_aexp_inputs a = set (b#c))"
  using enumerate_aexp_inputs_list by fastforce

lemma aval_ir_take: "A ≤ length i ⟹
  enumerate_regs a = {} ⟹
  enumerate_aexp_inputs a ≠ {} ⟹
  Max (enumerate_aexp_inputs a) < A ⟹
  aval a (join_ir (take A i) r) = aval a (join_ir i ra)"
proof(induct a)
  case (L x)
  then show ?case
    by simp
next
  case (V x)
  then show ?case
    apply (cases x)
     apply (simp add: join_ir_def input2state_nth)
    by simp
next
  case (Plus a1 a2)
  then show ?case
    apply (simp only: enumerate_aexp_inputs_not_empty[of "Plus a1 a2"])
    apply (erule exE)+
    apply (simp only: neq_Nil_conv List.linorder_class.Max.set_eq_fold)
    apply (case_tac "fold max c b ≤ length i")
     apply simp
     apply (metis List.finite_set Max.union Plus.prems(4) enumerate_aexp_inputs.simps(4) enumerate_aexp_inputs_not
max_less_iff_conj no_variables_aval sup_bot.left_neutral sup_bot.right_neutral)
    by simp
next
  case (Minus a1 a2)
  then show ?case
```

```
    apply (simp only: enumerate_aexp_inputs_not_empty[of "Minus a1 a2"])
    apply (erule exE)+
    apply (simp only: neq_Nil_conv List.linorder_class.Max.set_eq_fold)
    apply (case_tac "fold max c b ≤ length i")
     apply simp
    apply (metis List.finite_set Max.union Minus.prems(4) enumerate_aexp_inputs.simps(5) enumerate_aexp_inputs_not
max_less_iff_conj no_variables_aval sup_bot.left_neutral sup_bot.right_neutral)
    by simp
next
  case (Times a1 a2)
  then show ?case
    apply (simp only: enumerate_aexp_inputs_not_empty[of "Times a1 a2"])
    apply (erule exE)+
    apply (simp only: neq_Nil_conv List.linorder_class.Max.set_eq_fold)
    apply (case_tac "fold max c b ≤ length i")
     apply simp
    apply (metis List.finite_set Max.union Times.prems(4) enumerate_aexp_inputs.simps(6) enumerate_aexp_inputs_not
max_less_iff_conj no_variables_aval sup_bot.left_neutral sup_bot.right_neutral)
    by simp
qed

definition max_input :: "vname aexp ⇒ nat option" where
  "max_input g = (let inputs = (enumerate_aexp_inputs g) in if inputs = {} then None else Some (Max inputs))"

definition max_reg :: "vname aexp ⇒ nat option" where
  "max_reg g = (let regs = (enumerate_regs g) in if regs = {} then None else Some (Max regs))"

lemma max_reg_V_I: "max_reg (V (I n)) = None"
  by (simp add: max_reg_def)

lemma max_reg_V_R: "max_reg (V (R n)) = Some n"
  by (simp add: max_reg_def)

lemmas max_reg_V = max_reg_V_I max_reg_V_R

lemma max_reg_Plus: "max_reg (Plus a1 a2) = max (max_reg a1) (max_reg a2)"
  apply (simp add: max_reg_def Let_def max_absorb2)
  by (metis Max.union bot_option_def finite_enumerate_regs max_bot2 sup_Some sup_max)

lemma max_reg_Minus: "max_reg (Minus a1 a2) = max (max_reg a1) (max_reg a2)"
  apply (simp add: max_reg_def Let_def max_absorb2)
  by (metis Max.union bot_option_def finite_enumerate_regs max_bot2 sup_Some sup_max)

lemma max_reg_Times: "max_reg (Times a1 a2) = max (max_reg a1) (max_reg a2)"
  apply (simp add: max_reg_def Let_def max_absorb2)
  by (metis Max.union bot_option_def finite_enumerate_regs max_bot2 sup_Some sup_max)

lemma no_reg_aval_swap_regs:
  "max_reg a = None ⟹ aval a (join_ir i r) = aval a (join_ir i r')"
proof(induct a)
  case (V x)
  then show ?case
    apply (cases x)
     apply (simp add: join_ir_def)
    by (simp add: join_ir_def max_reg_def)
next
  case (Plus a1 a2)
  then show ?case
    by (metis (no_types, lifting) aval.simps(3) max.absorb2 max.cobounded2 max_reg_Plus sup_None_2 sup_max)
next
  case (Minus a1 a2)
  then show ?case
    by (metis (no_types, lifting) aval.simps(4) max.cobounded2 max_def_raw max_reg_Minus sup_None_2 sup_max)
```

```
next
  case (Times a1 a2)
  then show ?case
  proof -
    have "bot = max_reg a2"
      by (metis (no_types) Times.prems bot_option_def max.left_commute max_bot2 max_def_raw max_reg_Times)
    then show ?thesis
      by (metis Times.hyps(1) Times.hyps(2) Times.prems aval.simps(5) bot_option_def max_bot2 max_reg_Times)
  qed
qed auto

lemma aval_reg_some_superset:
"∀a. (r $ a ≠ None) ⟶ r $ a = r' $ a ⟹
 aval a (join_ir i r) = Some v ⟹
 aval a (join_ir i r') = Some v"
proof(induct a arbitrary: v rule: aexp_induct_separate_V_cases)
  case (I x)
  then show ?case
    by (simp add: join_ir_def)
next
  case (Plus x1a x2a)
  then show ?case
    apply simp
    by (metis maybe_arith_int_not_None option.simps(3) value_plus_def)
next
  case (Minus x1a x2a)
  then show ?case
    apply simp
    by (metis maybe_arith_int_not_None option.simps(3) value_minus_def)
next
  case (Times x1a x2a)
  then show ?case
    apply simp
    by (metis maybe_arith_int_not_None option.simps(3) value_times_def)
qed auto

lemma aval_reg_none_superset:
"∀a. (r $ a ≠ None) ⟶ r $ a = r' $ a ⟹
 aval a (join_ir i r') = None ⟹
 aval a (join_ir i r) = None"
proof(induct a)
  case (V x)
  then show ?case
    apply (cases x)
     apply (simp add: join_ir_def)
    by auto
next
  case (Plus a1 a2)
  then show ?case
    apply simp
    by (metis (no_types, lifting) maybe_arith_int_None Plus.prems(1) aval_reg_some_superset value_plus_def)
next
  case (Minus a1 a2)
  then show ?case
    apply simp
    by (metis (no_types, lifting) maybe_arith_int_None Minus.prems(1) aval_reg_some_superset value_minus_def)
next
  case (Times a1 a2)
  then show ?case
    apply simp
    by (metis (no_types, lifting) maybe_arith_int_None Times.prems(1) aval_reg_some_superset value_times_def)
qed auto
```

```
lemma enumerate_regs_empty_reg_unconstrained:
  "enumerate_regs a = {} ⟹ ∀r. ¬ aexp_constrains a (V (R r))"
  by (induct a rule: aexp_induct_separate_V_cases, auto)

lemma enumerate_aexp_inputs_empty_input_unconstrained:
  "enumerate_aexp_inputs a = {} ⟹ ∀r. ¬ aexp_constrains a (V (I r))"
  by (induct a rule: aexp_induct_separate_V_cases, auto)

lemma input_unconstrained_aval_input_swap:
  "∀i. ¬ aexp_constrains a (V (I i)) ⟹
   aval a (join_ir i r) = aval a (join_ir i' r)"
  using join_ir_def
  by (induct a rule: aexp_induct_separate_V_cases, auto)

lemma input_unconstrained_aval_register_swap:
  "∀i. ¬ aexp_constrains a (V (R i)) ⟹
   aval a (join_ir i r) = aval a (join_ir i r')"
  using join_ir_def
  by (induct a rule: aexp_induct_separate_V_cases, auto)

lemma unconstrained_variable_swap_aval:
  "∀i. ¬ aexp_constrains a (V (I i)) ⟹
   ∀r. ¬ aexp_constrains a (V (R r)) ⟹
   aval a s = aval a s'"
  by (induct a rule: aexp_induct_separate_V_cases, auto)

lemma max_input_I: "max_input (V (vname.I i)) = Some i"
  by (simp add: max_input_def)

lemma max_input_Plus:
  "max_input (Plus a1 a2) = max (max_input a1) (max_input a2)"
  apply (simp add: max_input_def Let_def max.commute max_absorb2)
  by (metis List.finite_set Max.union enumerate_aexp_inputs_list sup_Some sup_max)

lemma max_input_Minus:
  "max_input (Minus a1 a2) = max (max_input a1) (max_input a2)"
  apply (simp add: max_input_def Let_def max.commute max_absorb2)
  by (metis List.finite_set Max.union enumerate_aexp_inputs_list sup_Some sup_max)

lemma max_input_Times:
  "max_input (Times a1 a2) = max (max_input a1) (max_input a2)"
  apply (simp add: max_input_def Let_def max.commute max_absorb2)
  by (metis List.finite_set Max.union enumerate_aexp_inputs_list sup_Some sup_max)

lemma aval_take:
  "max_input x < Some a ⟹
   aval x (join_ir i r) = aval x (join_ir (take a i) r)"
proof(induct x rule: aexp_induct_separate_V_cases)
  case (I x)
  then show ?case
    by (metis aval.simps(2) input2state_take join_ir_def le_cases less_option_Some max_input_I take_all
vname.simps(5))
next
  case (R x)
  then show ?case
    by (simp add: join_ir_def)
next
  case (Plus x1a x2a)
  then show ?case
    by (simp add: max_input_Plus)
next
  case (Minus x1a x2a)
  then show ?case
```

```
    by (simp add: max_input_Minus)
next
  case (Times x1a x2a)
  then show ?case
    by (simp add: max_input_Times)
qed auto

lemma aval_no_reg_swap_regs: "max_input x < Some a ⟹
   max_reg x = None ⟹
   aval x (join_ir i ra) = aval x (join_ir (take a i) r)"
proof(induct x)
  case (V x)
  then show ?case
    apply (cases x)
     apply (metis aval_take enumerate_regs.simps(3) enumerate_regs_empty_reg_unconstrained input_unconstrained_ava
    by (simp add: max_reg_def)
next
  case (Plus x1 x2)
  then show ?case
    by (metis aval_take no_reg_aval_swap_regs)
next
  case (Minus x1 x2)
  then show ?case
    by (metis aval_take no_reg_aval_swap_regs)
next
  case (Times x1 x2)
  then show ?case
    by (metis aval_take no_reg_aval_swap_regs)
qed auto

fun enumerate_aexp_strings :: "'a aexp ⇒ String.literal set" where
  "enumerate_aexp_strings (L (Str s)) = {s}" |
  "enumerate_aexp_strings (L (Num s)) = {}" |
  "enumerate_aexp_strings (V _) = {}" |
  "enumerate_aexp_strings (Plus a1 a2) = enumerate_aexp_strings a1 ∪ enumerate_aexp_strings a2" |
  "enumerate_aexp_strings (Minus a1 a2) = enumerate_aexp_strings a1 ∪ enumerate_aexp_strings a2" |
  "enumerate_aexp_strings (Times a1 a2) = enumerate_aexp_strings a1 ∪ enumerate_aexp_strings a2"

fun enumerate_aexp_ints :: "'a aexp ⇒ int set" where
  "enumerate_aexp_ints (L (Str s)) = {}" |
  "enumerate_aexp_ints (L (Num s)) = {s}" |
  "enumerate_aexp_ints (V _) = {}" |
  "enumerate_aexp_ints (Plus a1 a2) = enumerate_aexp_ints a1 ∪ enumerate_aexp_ints a2" |
  "enumerate_aexp_ints (Minus a1 a2) = enumerate_aexp_ints a1 ∪ enumerate_aexp_ints a2" |
  "enumerate_aexp_ints (Times a1 a2) = enumerate_aexp_ints a1 ∪ enumerate_aexp_ints a2"

definition enumerate_vars :: "vname aexp ⇒ vname set" where
  "enumerate_vars a = (image I (enumerate_aexp_inputs a)) ∪ (image R (enumerate_regs a))"

fun rename_regs :: "(nat ⇒ nat) ⇒ vname aexp ⇒ vname aexp" where
  "rename_regs _ (L l) = (L l)" |
  "rename_regs f (V (R r)) = (V (R (f r)))" |
  "rename_regs _ (V v) = (V v)" |
  "rename_regs f (Plus a b) = Plus (rename_regs f a) (rename_regs f b)" |
  "rename_regs f (Minus a b) = Minus (rename_regs f a) (rename_regs f b)" |
  "rename_regs f (Times a b) = Times (rename_regs f a) (rename_regs f b)"

definition eq_upto_rename :: "vname aexp ⇒ vname aexp ⇒ bool" where
  "eq_upto_rename a1 a2 = (∃f. bij f ∧ rename_regs f a1 = a2)"
```

end

2.4.1 AExp Lexorder

This theory defines a lexicographical ordering on arithmetic expressions such that we can build orderings for guards and, subsequently, transitions. We make use of the previously established orderings on variable names and values.

```
theory AExp_Lexorder
imports AExp Value_Lexorder
begin
fun height :: "'a aexp ⇒ nat"  where
  "height (L l2) = 1" |
  "height (V v2) = 1" |
  "height (Plus e1 e2) = 1 + max (height e1) (height e2)" |
  "height (Minus e1 e2) = 1 + max (height e1) (height e2)" |
  "height (Times e1 e2) = 1 + max (height e1) (height e2)"

instantiation aexp :: (linorder) linorder begin
fun less_aexp_aux :: "'a aexp ⇒ 'a aexp ⇒ bool"  where
  "less_aexp_aux (L l1) (L l2) = (l1 < l2)" |
  "less_aexp_aux (L l1) _ = True" |

  "less_aexp_aux (V v1) (L l1) = False" |
  "less_aexp_aux (V v1) (V v2) = (v1 < v2)" |
  "less_aexp_aux (V v1) _ = True" |

  "less_aexp_aux (Plus e1 e2) (L l2) = False" |
  "less_aexp_aux (Plus e1 e2) (V v2) = False" |
  "less_aexp_aux (Plus e1 e2) (Plus e1' e2') = ((less_aexp_aux e1 e1') ∨ ((e1 = e1') ∧ (less_aexp_aux e2
e2')))"|
  "less_aexp_aux (Plus e1 e2) _ = True" |

  "less_aexp_aux (Minus e1 e2) (Minus e1' e2') =  ((less_aexp_aux e1 e1') ∨ ((e1 = e1') ∧ (less_aexp_aux
e2 e2')))" |
  "less_aexp_aux (Minus e1 e2) (Times e1' e2') = True" |
  "less_aexp_aux (Minus e1 e2) _ = False" |

  "less_aexp_aux (Times e1 e2) (Times e1' e2') =  ((less_aexp_aux e1 e1') ∨ ((e1 = e1') ∧ (less_aexp_aux
e2 e2')))" |
  "less_aexp_aux (Times e1 e2) _ = False"

definition less_aexp :: "'a aexp ⇒ 'a aexp ⇒ bool" where
  "less_aexp a1 a2 = (
    let
      h1 = height a1;
      h2 = height a2
    in
    if h1 = h2 then
      less_aexp_aux a1 a2
    else
      h1 < h2
  )"

definition less_eq_aexp :: "'a aexp ⇒ 'a aexp ⇒ bool"
  where "less_eq_aexp e1 e2 ≡ (e1 < e2) ∨ (e1 = e2)"

declare less_aexp_def [simp]

lemma less_aexp_aux_antisym: "less_aexp_aux x  y = (¬(less_aexp_aux y x) ∧ (x ≠ y))"
  by (induct x y rule: less_aexp_aux.induct, auto)

lemma less_aexp_antisym: "(x::'a aexp) < y = (¬(y < x) ∧ (x ≠ y))"
  apply (simp add: Let_def)
```

```
  apply standard
  using less_aexp_aux_antisym apply blast
  apply (simp add: not_less)
  apply clarify
  by (induct x, auto)

lemma less_aexp_aux_trans: "less_aexp_aux x y ⟹ less_aexp_aux y z ⟹ less_aexp_aux x z"
proof (induct x y arbitrary: z rule: less_aexp_aux.induct)
  case (1 l1 l2)
  then show ?case by (cases z, auto)
next
  case ("2_1" l1 v)
  then show ?case by (cases z, auto)
next
  case ("2_2" l1 v va)
  then show ?case by (cases z, auto)
next
  case ("2_3" l1 v va)
  then show ?case by (cases z, auto)
next
  case ("2_4" l1 v va)
  then show ?case by (cases z, auto)
next
  case (3 v1 l1)
  then show ?case by (cases z, auto)
next
  case (4 v1 v2)
  then show ?case by (cases z, auto)
next
  case ("5_1" v1 v va)
  then show ?case by (cases z, auto)
next
  case ("5_2" v1 v va)
  then show ?case by (cases z, auto)
next
  case ("5_3" v1 v va)
  then show ?case by (cases z, auto)
next
  case (6 e1 e2 l2)
  then show ?case by (cases z, auto)
next
  case (7 e1 e2 v2)
  then show ?case by (cases z, auto)
next
  case (8 e1 e2 e1' e2')
  then show ?case by (cases z, auto)
next
  case ("9_1" e1 e2 v va)
  then show ?case by (cases z, auto)
next
  case ("9_2" e1 e2 v va)
  then show ?case by (cases z, auto)
next
  case (10 e1 e2 e1' e2')
  then show ?case by (cases z, auto)
next
  case (11 e1 e2 e1' e2')
  then show ?case by (cases z, auto)
next
  case ("12_1" e1 e2 v)
  then show ?case by (cases z, auto)
next
  case ("12_2" e1 e2 v)
```

```
  then show ?case by (cases z, auto)
next
  case ("12_3" e1 e2 v va)
  then show ?case by (cases z, auto)
next
  case (13 e1 e2 e1' e2')
  then show ?case by (cases z, auto)
next
  case ("14_1" e1 e2 v)
  then show ?case by (cases z, auto)
next
  case ("14_2" e1 e2 v)
  then show ?case by (cases z, auto)
next
  case ("14_3" e1 e2 v va)
  then show ?case by (cases z, auto)
next
  case ("14_4" e1 e2 v va)
  then show ?case by (cases z, auto)
qed

lemma less_aexp_trans: "(x::'a aexp) < y ⟹ y < z ⟹ x < z"
  apply (simp add: Let_def)
  apply standard
   apply (metis AExp_Lexorder.less_aexp_aux_trans dual_order.asym)
  by presburger

instance proof
    fix x y z :: "'a aexp"
    show "(x < y) = (x ≤ y ∧ ¬ y ≤ x)"
      by (metis less_aexp_antisym less_eq_aexp_def)
    show "(x ≤ x)"
      by (simp add: less_eq_aexp_def)
    show "x ≤ y ⟹ y ≤ z ⟹ x ≤ z"
      by (metis less_aexp_trans less_eq_aexp_def)
    show "x ≤ y ⟹ y ≤ x ⟹ x = y"
      unfolding less_eq_aexp_def using less_aexp_antisym by blast
    show "x ≤ y ∨ y ≤ x"
      unfolding less_eq_aexp_def using less_aexp_antisym by blast
  qed
end

lemma smaller_height: "height a1 < height a2 ⟹ a1 < a2"
  by simp

end
```

2.4.2 Guards Expressions

This theory defines the guard language of EFSMs which can be translated directly to and from contexts. Boolean values true and false respectively represent the guards which are always and never satisfied. Guards may test for (in)equivalence of two arithmetic expressions or be connected using NOR logic into compound expressions. The use of NOR logic reduces the number of subgoals when inducting over guard expressions.

We also define syntax hacks for the relations less than, less than or equal to, greater than or equal to, and not equal to as well as the expression of logical conjunction, disjunction, and negation in terms of nor logic.

```
theory GExp
imports AExp Trilean
begin
datatype 'a gexp = Bc bool | Eq "'a aexp" "'a aexp" | Gt "'a aexp" "'a aexp" | In 'a "value list" |  Nor
"'a gexp" "'a gexp"

fun gval :: "'a gexp ⇒ 'a datastate ⇒ trilean" where
```

```
  "gval (Bc True) _ = true" |
  "gval (Bc False) _ = false" |
  "gval (Gt a1 a2) s = value_gt (aval a1 s) (aval a2 s)" |
  "gval (Eq a1 a2) s = value_eq (aval a1 s) (aval a2 s)" |
  "gval (In v l) s = (case s v of None ⇒ invalid | Some vv ⇒ if vv ∈ set l then true else false)" |
  "gval (Nor a1 a2) s = ¬? ((gval a1 s) ∨? (gval a2 s))"
definition gNot :: "'a gexp ⇒ 'a gexp"  where
  "gNot g ≡ Nor g g"

definition gOr :: "'a gexp ⇒ 'a gexp ⇒ 'a gexp"  where
  "gOr v va ≡ Nor (Nor v va) (Nor v va)"

definition gAnd :: "'a gexp ⇒ 'a gexp ⇒ 'a gexp"  where
  "gAnd v va ≡ Nor (Nor v v) (Nor va va)"

definition gImplies :: "'a gexp ⇒ 'a gexp ⇒ 'a gexp" where
  "gImplies p q ≡ gOr (gNot p) q"

definition Lt :: "'a aexp ⇒ 'a aexp ⇒ 'a gexp"  where
  "Lt a b ≡ Gt b a"

definition Le :: "'a aexp ⇒ 'a aexp ⇒ 'a gexp"  where
  "Le v va ≡ gNot (Gt v va)"

definition Ge :: "'a aexp ⇒ 'a aexp ⇒ 'a gexp"  where
  "Ge v va ≡ gNot (Lt v va)"

definition Ne :: "'a aexp ⇒ 'a aexp ⇒ 'a gexp"  where
  "Ne v va ≡ gNot (Eq v va)"

lemma gval_Lt [simp]:
  "gval (Lt a1 a2) s = value_gt (aval a2 s) (aval a1 s)"
  by (simp add: Lt_def)

lemma gval_Le [simp]:
  "gval (Le a1 a2) s = ¬? (value_gt (aval a1 s) (aval a2 s))"
  by (simp add: Le_def value_gt_def gNot_def maybe_or_idempotent)

lemma gval_Ge [simp]:
  "gval (Ge a1 a2) s = ¬? (value_gt (aval a2 s) (aval a1 s))"
  by (simp add: Ge_def value_gt_def gNot_def maybe_or_idempotent)

lemma gval_Ne [simp]:
  "gval (Ne a1 a2) s = ¬? (value_eq (aval a1 s) (aval a2 s))"
  by (simp add: Ne_def value_gt_def gNot_def maybe_or_idempotent)

lemmas connectives = gAnd_def gOr_def gNot_def Lt_def Le_def Ge_def Ne_def

lemma gval_gOr [simp]: "gval (gOr x y) r = (gval x r) ∨? (gval y r)"
  by (simp add: maybe_double_negation maybe_or_idempotent gOr_def)

lemma gval_gNot [simp]: "gval (gNot x) s = ¬? (gval x s)"
  by (simp add: maybe_or_idempotent gNot_def)

lemma gval_gAnd [simp]:
  "gval (gAnd g1 g2) s = (gval g1 s) ∧? (gval g2 s)"
  by (simp add: de_morgans_1 maybe_double_negation maybe_or_idempotent gAnd_def)

lemma gAnd_commute: "gval (gAnd a b) s = gval (gAnd b a) s"
  by (simp add: times_trilean_commutative)

lemma gOr_commute: "gval (gOr a b) s = gval (gOr b a) s"
  by (simp add: plus_trilean_commutative gOr_def)
```

```
lemma gval_gAnd_True:
  "(gval (gAnd g1 g2) s = true) = ((gval g1 s = true) ∧ gval g2 s = true)"
  by (simp add: maybe_and_true)

lemma nor_equiv: "gval (gNot (gOr a b)) s = gval (Nor a b) s"
  by simp

definition satisfiable :: "vname gexp ⇒ bool" where
  "satisfiable g ≡ (∃i r. gval g (join_ir i r) = true)"

definition "satisfiable_list l = satisfiable (fold gAnd l (Bc True))"

lemma unsatisfiable_false: "¬ satisfiable (Bc False)"
  by (simp add: satisfiable_def)

lemma satisfiable_true: "satisfiable (Bc True)"
  by (simp add: satisfiable_def)

definition valid :: "vname gexp ⇒ bool" where
  "valid g ≡ (∀s. gval g s = true)"

lemma valid_true: "valid (Bc True)"
  by (simp add: valid_def)

fun gexp_constrains :: "'a gexp ⇒ 'a aexp ⇒ bool" where
  "gexp_constrains (Bc _) _ = False" |
  "gexp_constrains (Eq a1 a2) a = (aexp_constrains a1 a ∨ aexp_constrains a2 a)" |
  "gexp_constrains (Gt a1 a2) a = (aexp_constrains a1 a ∨ aexp_constrains a2 a)" |
  "gexp_constrains (Nor g1 g2) a = (gexp_constrains g1 a ∨ gexp_constrains g2 a)" |
  "gexp_constrains (In v l) a = aexp_constrains (V v) a"

fun contains_bool :: "'a gexp ⇒ bool" where
  "contains_bool (Bc _) = True" |
  "contains_bool (Nor g1 g2) = (contains_bool g1 ∨ contains_bool g2)" |
  "contains_bool _ = False"

fun gexp_same_structure :: "'a gexp ⇒ 'a gexp ⇒ bool" where
  "gexp_same_structure (Bc b) (Bc b') = (b = b')" |
  "gexp_same_structure (Eq a1 a2) (Eq a1' a2') = (aexp_same_structure a1 a1' ∧ aexp_same_structure a2 a2')"
|
  "gexp_same_structure (Gt a1 a2) (Gt a1' a2') = (aexp_same_structure a1 a1' ∧ aexp_same_structure a2 a2')"
|
  "gexp_same_structure (Nor g1 g2) (Nor g1' g2') = (gexp_same_structure g1 g1' ∧ gexp_same_structure g2
g2')" |
  "gexp_same_structure (In v l) (In v' l') = (v = v' ∧ l = l')" |
  "gexp_same_structure _ _ = False"

lemma gval_foldr_true:
  "(gval (foldr gAnd G (Bc True)) s = true) = (∀g ∈ set G. gval g s = true)"
proof(induct G)
  case (Cons a G)
  then show ?case
    apply (simp only: foldr.simps comp_def gval_gAnd maybe_and_true)
    by simp
qed auto

fun enumerate_gexp_inputs :: "vname gexp ⇒ nat set" where
  "enumerate_gexp_inputs (Bc _) = {}" |
  "enumerate_gexp_inputs (Eq v va) = enumerate_aexp_inputs v ∪ enumerate_aexp_inputs va" |
  "enumerate_gexp_inputs (Gt v va) = enumerate_aexp_inputs v ∪ enumerate_aexp_inputs va" |
  "enumerate_gexp_inputs (In v va) = enumerate_aexp_inputs (V v)" |
  "enumerate_gexp_inputs (Nor v va) = enumerate_gexp_inputs v ∪ enumerate_gexp_inputs va"
```

```
lemma enumerate_gexp_inputs_list: "∃l. enumerate_gexp_inputs g = set l"
proof(induct g)
  case (Eq x1a x2)
  then show ?case
    by (metis enumerate_aexp_inputs_list enumerate_gexp_inputs.simps(2) set_append)
next
  case (Gt x1a x2)
  then show ?case
    by (metis enumerate_aexp_inputs_list enumerate_gexp_inputs.simps(3) set_append)
next
  case (In x1a x2)
  then show ?case
    by (simp add: enumerate_aexp_inputs_list)
next
  case (Nor g1 g2)
  then show ?case
    by (metis enumerate_gexp_inputs.simps(5) set_append)
qed auto

definition max_input :: "vname gexp ⇒ nat option" where
  "max_input g = (let inputs = (enumerate_gexp_inputs g) in if inputs = {} then None else Some (Max inputs))"

definition max_input_list :: "vname gexp list ⇒ nat option" where
  "max_input_list g = fold max (map (λg. max_input g) g) None"

lemma max_input_list_cons:
  "max_input_list (a # G) = max (max_input a) (max_input_list G)"
  apply (simp add: max_input_list_def)
  apply (cases "max_input a")
   apply (simp add: max_def_raw)
  by (metis (no_types, lifting) List.finite_set Max.insert Max.set_eq_fold fold_simps(1) list.set(2) max.assoc
set_empty)

fun enumerate_regs :: "vname gexp ⇒ nat set" where
  "enumerate_regs (Bc _) = {}" |
  "enumerate_regs (Eq v va) = AExp.enumerate_regs v ∪ AExp.enumerate_regs va" |
  "enumerate_regs (Gt v va) = AExp.enumerate_regs v ∪ AExp.enumerate_regs va" |
  "enumerate_regs (In v va) = AExp.enumerate_regs (V v)" |
  "enumerate_regs (Nor v va) = enumerate_regs v ∪ enumerate_regs va"

lemma finite_enumerate_regs: "finite (enumerate_regs g)"
  using AExp.finite_enumerate_regs by (induct g, auto)

definition max_reg :: "vname gexp ⇒ nat option" where
  "max_reg g = (let regs = (enumerate_regs g) in if regs = {} then None else Some (Max regs))"

lemma max_reg_gNot: "max_reg (gNot x) = max_reg x"
  by (simp add: max_reg_def gNot_def)

lemma max_reg_Eq: "max_reg (Eq a b) = max (AExp.max_reg a) (AExp.max_reg b)"
  apply (simp add: max_reg_def AExp.max_reg_def Let_def max_absorb2)
  by (metis AExp.finite_enumerate_regs Max.union bot_option_def max_bot2 sup_Some sup_max)

lemma max_reg_Gt: "max_reg (Gt a b) = max (AExp.max_reg a) (AExp.max_reg b)"
  apply (simp add: max_reg_def AExp.max_reg_def Let_def max_absorb2)
  by (metis AExp.finite_enumerate_regs Max.union bot_option_def max_bot2 sup_Some sup_max)

lemma max_reg_Nor: "max_reg (Nor a b) = max (max_reg a) (max_reg b)"
  apply (simp add: max_reg_def AExp.max_reg_def Let_def max_absorb2)
  by (metis GExp.finite_enumerate_regs Max.union bot_option_def max_bot2 sup_Some sup_max)

lemma gval_In_cons:
```

```
  "gval (In v (a # as)) s = (gval (Eq (V v) (L a)) s ∨? gval (In v as) s)"
  by (cases "s v", auto)

lemma possible_to_be_in: "s ≠ [] ⟹ satisfiable (In v s)"
proof-
  assume "s ≠ []"
  have aux: "∃v' i r. join_ir i r v = Some v' ∧ v' ∈ set s ⟹
             ∃i r. (case join_ir i r v of None ⇒ false | Some v ⇒ if v ∈ set s then true else false)
= true"
    by (metis (mono_tags, lifting) option.simps(5))
  show ?thesis
    apply (simp add: satisfiable_def gval_In_cons)
    apply (cases s)
     apply (simp add: ‹s ≠ []›)
    apply (cases v)
     apply (case_tac "∃(i::value list). length i > x1 ∧ i ! x1 = a")
      apply clarsimp
    subgoal for _ _ i by (rule exI[of _ i], intro exI, simp)
     apply (metis gt_ex length_list_update length_repeat nth_list_update_eq)
    apply (rule_tac exI)
    apply (case_tac "∃r. r $ x2 = Some a")
     apply clarsimp
    subgoal for _ _ _ r by (rule exI[of _ r], simp)
    by (metis join_ir_R join_ir_double_exists)
qed

definition max_reg_list :: "vname gexp list ⇒ nat option" where
  "max_reg_list g = (fold max (map (λg. max_reg g) g) None)"

lemma max_reg_list_cons:
  "max_reg_list (a # G) = max (max_reg a) (max_reg_list G)"
  apply (simp add: max_reg_list_def)
  by (metis (no_types, lifting) List.finite_set Max.insert Max.set_eq_fold fold.simps(1) id_apply list.simps(15)
max.assoc set_empty)

lemma max_reg_list_append_singleton:
  "max_reg_list (as@[bs]) = max (max_reg_list as) (max_reg_list [bs])"
  apply (simp add: max_reg_list_def)
  by (metis max.commute sup_None_2 sup_max)

lemma max_reg_list_append:
  "max_reg_list (as@bs) = max (max_reg_list as) (max_reg_list bs)"
proof(induct bs rule: rev_induct)
  case Nil
  then show ?case
    by (metis append_Nil2 fold_simps(1) list.simps(8) max_reg_list_def sup_None_2 sup_max)
next
  case (snoc x xs)
  then show ?case
    by (metis append_assoc max.assoc max_reg_list_append_singleton)
qed

definition apply_guards :: "vname gexp list ⇒ vname datastate ⇒ bool" where
  "apply_guards G s = (∀g ∈ set (map (λg. gval g s) G). g = true)"

lemma apply_guards_singleton[simp]: "(apply_guards [g] s) = (gval g s = true)"
  by (simp add: apply_guards_def)

lemma apply_guards_empty [simp]: "apply_guards [] s"
  by (simp add: apply_guards_def)

lemma apply_guards_cons:
  "apply_guards (a # G) c = (gval a c = true ∧ apply_guards G c)"
```

```
  by (simp add: apply_guards_def)

lemma apply_guards_double_cons:
  "apply_guards (y # x # G) s = (gval (gAnd y x) s = true ∧ apply_guards G s)"
  using apply_guards_cons gval_gAnd_True by blast

lemma apply_guards_append:
  "apply_guards (a@a') s = (apply_guards a s ∧ apply_guards a' s)"
  using apply_guards_def by auto

lemma apply_guards_foldr:
  "apply_guards G s = (gval (foldr gAnd G (Bc True)) s = true)"
proof(induct G)
  case Nil
  then show ?case
    by (simp add: apply_guards_def)
next
  case (Cons a G)
  then show ?case
    by (metis apply_guards_cons foldr.simps(2) gval_gAnd_True o_apply)
qed

lemma rev_apply_guards: "apply_guards (rev G) s = apply_guards G s"
  by (simp add: apply_guards_def)

lemma apply_guards_fold:
  "apply_guards G s = (gval (fold gAnd G (Bc True)) s = true)"
  using rev_apply_guards[symmetric]
  by (simp add: foldr_conv_fold apply_guards_foldr)

lemma fold_apply_guards:
  "(gval (fold gAnd G (Bc True)) s = true) = apply_guards G s"
  by (simp add: apply_guards_fold)

lemma foldr_apply_guards:
  "(gval (foldr gAnd G (Bc True)) s = true) = apply_guards G s"
  by (simp add: apply_guards_foldr)

lemma apply_guards_subset:
  "set g' ⊆ set g ⟹ apply_guards g c ⟶ apply_guards g' c"
proof(induct g)
  case (Cons a g)
  then show ?case
    using apply_guards_def by auto
qed auto

lemma apply_guards_subset_append:
  "set G ⊆ set G' ⟹ apply_guards (G @ G') s = apply_guards (G') s"
  using apply_guards_append apply_guards_subset by blast

lemma apply_guards_rearrange:
  "x ∈ set G ⟹ apply_guards G s = apply_guards (x#G) s"
  using apply_guards_def by auto

lemma apply_guards_condense: "∃g. apply_guards G s = (gval g s = true)"
    using apply_guards_fold by blast

lemma apply_guards_false_condense: "∃g. (¬apply_guards G s) = (gval g s = false)"
  using foldr_apply_guards gval.simps(2) not_true by blast

lemma max_input_Bc: "max_input (Bc x) = None"
  by (simp add: max_input_def)
```

```
lemma max_input_Eq:
  "max_input (Eq a1 a2) = max (AExp.max_input a1) (AExp.max_input a2)"
  apply (simp add: AExp.max_input_def max_input_def Let_def max_absorb2)
  by (metis List.finite_set Max.union bot_option_def enumerate_aexp_inputs_not_empty max_bot2 sup_Some sup_max)

lemma max_input_Gt:
  "max_input (Gt a1 a2) = max (AExp.max_input a1) (AExp.max_input a2)"
  apply (simp add: AExp.max_input_def max_input_def Let_def max_absorb2)
  by (metis List.finite_set Max.union bot_option_def enumerate_aexp_inputs_not_empty max_bot2 sup_Some sup_max)

lemma gexp_max_input_Nor:
  "max_input (Nor g1 g2) = max (max_input g1) (max_input g2)"
  apply (simp add: AExp.max_input_def max_input_def Let_def max_absorb2)
  by (metis List.finite_set Max.union enumerate_gexp_inputs_list less_eq_option_Some_None max_def sup_Some
sup_max)

lemma gexp_max_input_In: "max_input (In v l) = AExp.max_input (V v)"
  by (simp add: AExp.max_input_def GExp.max_input_def)

lemma gval_foldr_gOr_invalid:
  "(gval (fold gOr l g) s = invalid) = (∃g' ∈ (set (g#l)). gval g' s = invalid)"
proof(induct l rule: rev_induct)
  case (snoc x xs)
  then show ?case
    by (simp, metis gval_gOr maybe_or_invalid)
qed auto

lemma gval_foldr_gOr_true:
  "(gval (fold gOr l g) s = true) = ((∃g' ∈ (set (g#l)). gval g' s = true) ∧ (∀g' ∈ (set (g#l)). gval g'
s ≠ invalid))"
proof(induct l rule: rev_induct)
  case (snoc x xs)
  then show ?case
    apply (simp add: maybe_or_true)
    using gval_foldr_gOr_invalid by auto
qed auto

lemma gval_foldr_gOr_false:
  "(gval (fold gOr l g) s = false) = (∀g' ∈ (set (g#l)). gval g' s = false)"
proof(induct l rule: rev_induct)
  case (snoc x xs)
  then show ?case
    by (auto simp add: maybe_or_false)
qed auto

lemma gval_fold_gOr_rev: "gval (fold gOr (rev l) g) s = gval (fold gOr l g) s"
  apply (cases "gval (fold gOr l g) s")
    apply (simp, simp add: gval_foldr_gOr_true)
   apply (simp, simp add: gval_foldr_gOr_false)
  by (simp, simp add: gval_foldr_gOr_invalid)

lemma gval_fold_gOr_foldr: "gval (fold gOr l g) s = gval (foldr gOr l g) s"
  by (simp add: foldr_conv_fold gval_fold_gOr_rev)

lemma gval_fold_gOr:
  "gval (fold gOr (a # l) g) s = (gval a s ∨? gval (fold gOr l g) s)"
  by (simp only: gval_fold_gOr_foldr foldr.simps comp_def gval_gOr)

lemma gval_In_fold:
  "gval (In v l) s = (if s v = None then invalid else gval (fold gOr (map (λx. Eq (V v) (L x)) l) (Bc False))
s)"
proof(induct l)
  case Nil
```

```
  then show ?case
    apply simp
    apply (cases "s v")
     apply simp
    by auto
next
  case (Cons a l)
  then show ?case
    apply (simp only: gval_In_cons)
    apply (cases "s v")
     apply simp
    by (simp add: gval_fold_gOr del: fold.simps)
qed

fun fold_In :: "'a ⇒ value list ⇒ 'a gexp" where
  "fold_In _ [] = Bc False" |
  "fold_In v (l#t) = gOr (Eq (V v) (L l)) (fold_In v t)"

lemma gval_fold_In: "l ≠ [] ⟹ gval (In v l) s = gval (fold_In v l) s"
proof(induct l)
next
  case (Cons a l)
  then show ?case
    apply (case_tac "s v")
     apply simp
    apply simp
    apply safe
       apply simp
       apply (metis fold_In.simps(1) gval.simps(2) plus_trilean.simps(4) plus_trilean.simps(5))
      apply fastforce
     apply fastforce
    by fastforce
qed auto

lemma fold_maybe_or_invalid_base: "fold (∨?) l invalid = invalid"
proof(induct l)
  case (Cons a l)
  then show ?case
    by (metis fold_simps(2) maybe_or_valid)
qed auto

lemma fold_maybe_or_true_base_never_false:
  "fold (∨?) l true ≠ false"
proof(induct l)
  case (Cons a l)
  then show ?case
    by (metis fold_maybe_or_invalid_base fold_simps(2) maybe_not.cases maybe_or_valid plus_trilean.simps(4)
plus_trilean.simps(6))
qed auto

lemma fold_true_fold_false_not_invalid:
  "fold (∨?) l true = true ⟹
   fold (∨?) (rev l) false ≠ invalid"
proof(induct l)
  case (Cons a l)
  then show ?case
    apply simp
    by (metis fold_maybe_or_invalid_base maybe_or_invalid maybe_or_true)
qed auto

lemma fold_true_invalid_fold_rev_false_invalid:
  "fold (∨?) l true = invalid ⟹
   fold (∨?) (rev l) false = invalid"
```

```
proof(induct l)
  case (Cons a l)
  then show ?case
    apply simp
    by (metis maybe_or_true maybe_or_valid)
qed auto

lemma fold_maybe_or_rev:
  "fold (∨?) l b = fold (∨?) (rev l) b"
proof(induct l)
  case (Cons a l)
  then show ?case
  proof(induction a b rule: plus_trilean.induct)
    case (1 uu)
    then show ?case
      by (simp add: fold_maybe_or_invalid_base)
  next
    case "2_1"
    then show ?case
      by (simp add: fold_maybe_or_invalid_base)
  next
    case "2_2"
    then show ?case
      by (simp add: fold_maybe_or_invalid_base)
  next
    case "3_1"
    then show ?case
      apply simp
      by (metis add.assoc fold_maybe_or_true_base_never_false maybe_not.cases maybe_or_idempotent maybe_or_true)
  next
    case "3_2"
    then show ?case
      apply simp
      apply (case_tac "fold (∨?) l true")
        apply (simp add: eq_commute[of true])
        apply (case_tac "fold (∨?) (rev l) false")
          apply simp
         apply simp
        apply (simp add: fold_true_fold_false_not_invalid)
       apply (simp add: fold_maybe_or_true_base_never_false)
      by (simp add: fold_true_invalid_fold_rev_false_invalid)
  next
    case 4
    then show ?case
      by (simp add: maybe_or_zero)
  next
    case 5
    then show ?case
      by (simp add: maybe_or_zero)
  qed
qed auto

lemma fold_maybe_or_cons:
  "fold (∨?) (a#l) b = a ∨? (fold (∨?) l b)"
  by (metis fold_maybe_or_rev foldr.simps(2) foldr_conv_fold o_apply)

lemma gval_fold_gOr_map:
  "gval (fold gOr l (Bc False)) s = fold (∨?) (map (λg. gval g s) l) (false)"
proof(induct l)
  case (Cons a l)
  then show ?case
    by (metis fold_maybe_or_cons gval_fold_gOr list.simps(9))
qed auto
```

```
lemma gval_unfold_first:
  "gval (fold gOr (map (λx. Eq (V v) (L x)) ls) (Eq (V v) (L l))) s =
       gval (fold gOr (map (λx. Eq (V v) (L x)) (l#ls)) (Bc False)) s"
proof(induct ls)
  case Nil
  then show ?case
    apply (cases "s v")
     apply simp
    by (simp add: gOr_def)
next
  case (Cons a ls)
  then show ?case
  proof -
    have "gval (fold gOr (map (λva. Eq (V v) (L va)) ls) (gOr (Eq (V v) (L l)) (Bc False))) s = gval (fold
gOr (map (λva. Eq (V v) (L va)) (l # ls)) (Bc False)) s"
      by simp
    then have "gval (fold gOr (map (λva. Eq (V v) (L va)) (a # ls)) (Eq (V v) (L l))) s = gval (fold gOr
(Eq (V v) (L a) # map (λva. Eq (V v) (L va)) ls) (gOr (Eq (V v) (L l)) (Bc False))) s"
      by (metis (no_types) Cons.hyps gval_fold_gOr list.simps(9))
    then show ?thesis
      by force
  qed
qed

lemma fold_Eq_true:
  "∀v. fold (∨?) (map (λx. if v = x then true else false) vs) true = true"
  by(induct vs, auto)

lemma x_in_set_fold_eq:
  "x ∈ set ll ⟹
   fold (∨?) (map (λxa. if x = xa then true else false) ll) false = true"
proof(induct ll)
  case (Cons a ll)
  then show ?case
    apply simp
    apply standard
     apply (simp add: fold_Eq_true)
    by auto
qed auto

lemma x_not_in_set_fold_eq:
  "s v ∉ Some ` set ll ⟹
   false = fold (∨?) (map (λx. if s v = Some x then true else false) ll) false"
  by(induct ll, auto)

lemma gval_take: "max_input g < Some a ⟹
   gval g (join_ir i r) = gval g (join_ir (take a i) r)"
proof(induct g)
case (Bc x)
  then show ?case
    by (metis (full_types) gval.simps(1) gval.simps(2))
next
  case (Eq x1a x2)
  then show ?case
    by (metis aval_take gval.simps(4) max_input_Eq max_less_iff_conj)
next
  case (Gt x1a x2)
  then show ?case
    by (metis aval_take gval.simps(3) max_input_Gt max_less_iff_conj)
next
  case (Nor g1 g2)
  then show ?case
```

```
    by (simp add: maybe_not_eq gexp_max_input_Nor)
next
  case (In v l)
  then show ?case
    apply (simp add: gexp_max_input_In)
    using aval_take by fastforce
qed

lemma gval_fold_gAnd_append_singleton:
  "gval (fold gAnd (a @ [G]) (Bc True)) s = gval (fold gAnd a (Bc True)) s ∧? gval G s"
  apply simp
  using times_trilean_commutative by blast

lemma gval_fold_rev_true:
  "gval (fold gAnd (rev G) (Bc True)) s = true ⟹
   gval (fold gAnd G (Bc True)) s = true"
  by (metis foldr_conv_fold gval_foldr_true rev_rev_ident set_rev)

lemma gval_fold_not_invalid_all_valid_contra:
  "∃g ∈ set G. gval g s = invalid ⟹
   gval (fold gAnd G (Bc True)) s = invalid"
proof(induct G rule: rev_induct)
  case (snoc a G)
  then show ?case
    apply (simp only: gval_fold_gAnd_append_singleton)
    apply simp
    using maybe_and_valid by blast
qed auto

lemma gval_fold_not_invalid_all_valid:
  "gval (fold gAnd G (Bc True)) s ≠ invalid ⟹
   ∀g ∈ set G. gval g s ≠ invalid"
  using gval_fold_not_invalid_all_valid_contra by blast

lemma all_gval_not_false:
  "(∀g ∈ set G. gval g s ≠ false) = (∀g ∈ set G. gval g s = true) ∨ (∃g ∈ set G. gval g s = invalid)"
  using trilean.exhaust by auto

lemma must_have_one_false_contra:
  "∀g ∈ set G. gval g s ≠ false ⟹
   gval (fold gAnd G (Bc True)) s ≠ false"
  using all_gval_not_false[of G s]
  apply simp
  apply (case_tac "(∀g∈set G. gval g s = true)")
  apply (metis (full_types) foldr_conv_fold gval_fold_rev_true gval_foldr_true not_true)
  by (simp add: gval_fold_not_invalid_all_valid_contra)

lemma must_have_one_false:
  "gval (fold gAnd G (Bc True)) s = false ⟹
   ∃g ∈ set G. gval g s = false"
  using must_have_one_false_contra by blast

lemma all_valid_fold:
  "∀g ∈ set G. gval g s ≠ invalid ⟹
   gval (fold gAnd G (Bc True)) s ≠ invalid"
  apply (induct G rule: rev_induct)
   apply simp
  by (simp add: maybe_and_invalid)

lemma one_false_all_valid_false:
  "∃g∈set G. gval g s = false ⟹
   ∀g∈set G. gval g s ≠ invalid ⟹
   gval (fold gAnd G (Bc True)) s = false"
```

```
  by (metis (full_types) all_valid_fold foldr_conv_fold gval_foldr_true not_true rev_rev_ident set_rev)

lemma gval_fold_rev_false:
  "gval (fold gAnd (rev G) (Bc True)) s = false ⟹
   gval (fold gAnd G (Bc True)) s = false"
  using must_have_one_false[of "rev G" s]
        gval_fold_not_invalid_all_valid[of "rev G" s]
  by (simp add: one_false_all_valid_false)

lemma fold_invalid_means_one_invalid:
  "gval (fold gAnd G (Bc True)) s = invalid ⟹
   ∃g ∈ set G. gval g s = invalid"
  using all_valid_fold by blast

lemma gval_fold_rev_invalid:
  "gval (fold gAnd (rev G) (Bc True)) s = invalid ⟹
   gval (fold gAnd G (Bc True)) s = invalid"
  using fold_invalid_means_one_invalid[of "rev G" s]
  by (simp add: gval_fold_not_invalid_all_valid_contra)

lemma gval_fold_rev_equiv_fold:
  "gval (fold gAnd (rev G) (Bc True)) s =  gval (fold gAnd G (Bc True)) s"
  apply (cases "gval (fold gAnd (rev G) (Bc True)) s")
    apply (simp add: gval_fold_rev_true)
   apply (simp add: gval_fold_rev_false)
  by (simp add: gval_fold_rev_invalid)

lemma gval_fold_equiv_fold_rev:
  "gval (fold gAnd G (Bc True)) s = gval (fold gAnd (rev G) (Bc True)) s"
  by (simp add: gval_fold_rev_equiv_fold)

lemma gval_fold_equiv_gval_foldr:
  "gval (fold gAnd G (Bc True)) s = gval (foldr gAnd G (Bc True)) s"
proof -
  have "gval (fold gAnd G (Bc True)) s = gval (fold gAnd (rev G) (Bc True)) s"
    using gval_fold_equiv_fold_rev by force
  then show ?thesis
  by (simp add: foldr_conv_fold)
qed

lemma gval_foldr_equiv_gval_fold:
  "gval (foldr gAnd G (Bc True)) s = gval (fold gAnd G (Bc True)) s"
  by (simp add: gval_fold_equiv_gval_foldr)

lemma gval_fold_cons:
  "gval (fold gAnd (g # gs) (Bc True)) s = gval g s ∧? gval (fold gAnd gs (Bc True)) s"
  apply (simp only: apply_guards_fold gval_fold_equiv_gval_foldr)
  by (simp only: foldr.simps comp_def gval_gAnd)

lemma gval_fold_take: "max_input_list G < Some a ⟹
   a ≤ length i ⟹
   max_input_list G ≤ Some (length i) ⟹
   gval (fold gAnd G (Bc True)) (join_ir i r) = gval (fold gAnd G (Bc True)) (join_ir (take a i) r)"
proof(induct G)
  case (Cons g gs)
  then show ?case
    apply (simp only: gval_fold_cons)
    apply (simp add: max_input_list_cons)
    using gval_take[of g a i r]
    by simp
qed auto

primrec padding :: "nat ⇒ 'a list" where
```

```
  "padding 0 = []" |
  "padding (Suc m) = (Eps (λx. True))#(padding m)"

definition take_or_pad :: "'a list ⇒ nat ⇒ 'a list" where
  "take_or_pad a n = (if length a ≥ n then take n a else a@(padding (n-length a)))"

lemma length_padding: "length (padding n) = n"
  by (induct n, auto)

lemma length_take_or_pad: "length (take_or_pad a n) = n"
proof(induct n)
  case 0
  then show ?case
    by (simp add: take_or_pad_def)
next
  case (Suc n)
  then show ?case
    apply (simp add: take_or_pad_def)
    apply standard
     apply auto[1]
    by (simp add: length_padding)
qed

fun enumerate_gexp_strings :: "'a gexp ⇒ String.literal set" where
  "enumerate_gexp_strings (Bc _) = {}" |
  "enumerate_gexp_strings (Eq a1 a2) = enumerate_aexp_strings a1 ∪ enumerate_aexp_strings a2" |
  "enumerate_gexp_strings (Gt a1 a2) = enumerate_aexp_strings a1 ∪ enumerate_aexp_strings a2" |
  "enumerate_gexp_strings (In v l) = fold (λx acc. case x of Num n ⇒ acc | Str s ⇒ insert s acc) l {}"
|
  "enumerate_gexp_strings (Nor g1 g2) = enumerate_gexp_strings g1 ∪ enumerate_gexp_strings g2"

fun enumerate_gexp_ints :: "'a gexp ⇒ int set" where
  "enumerate_gexp_ints (Bc _) = {}" |
  "enumerate_gexp_ints (Eq a1 a2) = enumerate_aexp_ints a1 ∪ enumerate_aexp_ints a2" |
  "enumerate_gexp_ints (Gt a1 a2) = enumerate_aexp_ints a1 ∪ enumerate_aexp_ints a2" |
  "enumerate_gexp_ints (In v l) = fold (λx acc. case x of Str s ⇒ acc | Num n ⇒ insert n acc) l {}" |
  "enumerate_gexp_ints (Nor g1 g2) = enumerate_gexp_ints g1 ∪ enumerate_gexp_ints g2"

definition restricted_once :: "'a ⇒ 'a gexp list ⇒ bool" where
  "restricted_once v G = (length (filter (λg. gexp_constrains g (V v)) G) = 1)"

definition not_restricted :: "'a ⇒ 'a gexp list ⇒ bool" where
  "not_restricted v G = (length (filter (λg. gexp_constrains g (V v)) G) = 0)"

lemma restricted_once_cons:
  "restricted_once v (g#gs) = ((gexp_constrains g (V v) ∧ not_restricted v gs) ∨ ((¬ gexp_constrains g
(V v)) ∧ restricted_once v gs))"
  by (simp add: restricted_once_def not_restricted_def)

lemma not_restricted_cons:
  "not_restricted v (g#gs) = ((¬ gexp_constrains g (V v)) ∧ not_restricted v gs)"
  by (simp add: not_restricted_def)

definition enumerate_vars :: "vname gexp ⇒ vname list" where
  "enumerate_vars g = sorted_list_of_set ((image R (enumerate_regs g)) ∪ (image I (enumerate_gexp_inputs
g)))"

fun rename_regs :: "(nat ⇒ nat) ⇒ vname gexp ⇒ vname gexp" where
  "rename_regs _ (Bc b) = Bc b" |
  "rename_regs f (Eq a1 a2) = Eq (AExp.rename_regs f a1) (AExp.rename_regs f a2)" |
  "rename_regs f (Gt a1 a2) = Gt (AExp.rename_regs f a1) (AExp.rename_regs f a2)" |
  "rename_regs f (In (R r) vs) = In (R (f r)) vs" |
  "rename_regs f (In v vs) = In v vs" |
```

```
  "rename_regs f (Nor g1 g2) = Nor (rename_regs f g1) (rename_regs f g2)"

definition eq_upto_rename :: "vname gexp ⇒ vname gexp ⇒ bool" where
  "eq_upto_rename g1 g2 = (∃f. bij f ∧ rename_regs f g1 = g2)"

lemma gval_reg_some_superset:
"∀a. (r $ a  ≠ None) ⟶ r $ a = r' $ a ⟹
  x ≠ invalid ⟹
 gval a (join_ir i r) = x ⟹
 gval a (join_ir i r') = x"
proof(induct a arbitrary: x)
case (Bc b)
  then show ?case by (cases b, auto)
next
  case (Eq x1a x2)
  then show ?case
    apply (cases x)
      apply simp
    using value_eq_true[of "aval x1a (join_ir i r)" "aval x2 (join_ir i r)"]
      apply clarsimp
      apply (simp add: aval_reg_some_superset)
     apply simp
    using value_eq_false[of "aval x1a (join_ir i r)" "aval x2 (join_ir i r)"]
     apply clarsimp
     apply (simp add: aval_reg_some_superset)
    by simp
next
  case (Gt x1a x2)
  then show ?case
    apply (cases x)
      apply simp
    using value_gt_true_Some[of "aval x1a (join_ir i r)" "aval x2 (join_ir i r)"]
      apply clarsimp
      apply (simp add: aval_reg_some_superset)
     apply simp
    using value_gt_false_Some[of "aval x1a (join_ir i r)" "aval x2 (join_ir i r)"]
     apply clarsimp
     apply (simp add: aval_reg_some_superset)
    by simp
next
  case (In x1a x2)
  then show ?case
    apply simp
    apply (case_tac "join_ir i r x1a")
     apply simp
    apply (case_tac "join_ir i r' x1a")
     apply simp
     apply (metis aval_reg_some_superset In.prems(1) aval.simps(2) option.distinct(1))
    apply simp
    by (metis (full_types) aval_reg_some_superset In.prems(1) aval.simps(2) option.inject)
next
  case (Nor a1 a2)
  then show ?case
    apply simp
    apply (cases x)
      apply (simp add: maybe_negate_true maybe_or_false)
     apply (simp add: maybe_negate_false maybe_or_true)
     apply presburger
    by simp
qed

lemma apply_guards_reg_some_superset:
  "∀a. (r $ a  ≠ None) ⟶ r $ a = r' $ a ⟹
```

```
   apply_guards G (join_ir i r) ⟹
   apply_guards G (join_ir i r')"
  apply (induct G)
   apply simp
  apply (simp add: apply_guards_cons)
  using gval_reg_some_superset
  by simp

end
```

2.4.3 GExp Lexorder

This theory defines a lexicographical ordering on guard expressions such that we can build orderings for transitions. We make use of the previously established orderings on arithmetic expressions.

```
theory
GExp_Lexorder
imports
  "GExp"
  "AExp_Lexorder"
  "HOL-Library.List_Lexorder"
begin

fun height :: "'a gexp ⇒ nat" where
  "height (Bc _) = 1" |
  "height (Eq a1 a2) = 1 + max (AExp_Lexorder.height a1) (AExp_Lexorder.height a2)" |
  "height (Gt a1 a2) = 1 + max (AExp_Lexorder.height a1) (AExp_Lexorder.height a2)" |
  "height (In v l) = 2 + size l" |
  "height (Nor g1 g2) = 1 + max (height g1) (height g2)"

instantiation gexp :: (linorder) linorder begin
fun less_gexp_aux :: "'a gexp ⇒ 'a gexp ⇒ bool"  where
  "less_gexp_aux (Bc b1) (Bc b2) = (b1 < b2)" |
  "less_gexp_aux (Bc b1) _ = True" |

  "less_gexp_aux (Eq e1 e2) (Bc b2) = False" |
  "less_gexp_aux (Eq e1 e2) (Eq e1' e2') = ((e1 < e1') ∨ ((e1 = e1') ∧ (e2 < e2')))" |
  "less_gexp_aux (Eq e1 e2) _ = True" |

  "less_gexp_aux (Gt e1 e2) (Bc b2) = False" |
  "less_gexp_aux (Gt e1 e2) (Eq e1' e2') = False" |
  "less_gexp_aux (Gt e1 e2) (Gt e1' e2') = ((e1 < e1') ∨ ((e1 = e1') ∧ (e2 < e2')))" |
  "less_gexp_aux (Gt e1 e2) _ = True" |

  "less_gexp_aux (In vb vc) (Nor v va) = True" |
  "less_gexp_aux (In vb vc) (In v va) = (vb < v ∨ (vb = v ∧ vc < va))" |
  "less_gexp_aux (In vb vc) _ = False" |

  "less_gexp_aux (Nor g1 g2) (Nor g1' g2') = ((less_gexp_aux g1 g1') ∨ ((g1 = g1') ∧ (less_gexp_aux g2 g2')))"
|
  "less_gexp_aux (Nor g1 g2) _ = False"

definition less_gexp :: "'a gexp ⇒ 'a gexp ⇒ bool" where
  "less_gexp a1 a2 = (
    let
      h1 = height a1;
      h2 = height a2
    in
    if h1 = h2 then
      less_gexp_aux a1 a2
    else
      h1 < h2
  )"
```

```
declare less_gexp_def [simp]

definition less_eq_gexp :: "'a gexp ⇒ 'a gexp ⇒ bool" where
  "less_eq_gexp e1 e2 ≡ (e1 < e2) ∨ (e1 = e2)"

lemma less_gexp_aux_antisym: "less_gexp_aux x y = (¬(less_gexp_aux y x) ∧ (x ≠ y))"
proof (induct x y rule: less_gexp_aux.induct)
  case (1 b1 b2)
  then show ?case by auto
next
  case ("2_1" b1 v va)
  then show ?case by auto
next
  case ("2_2" b1 v va)
  then show ?case by auto
next
  case ("2_3" b1 v va)
  then show ?case by auto
next
  case ("2_4" b1 v va)
  then show ?case by auto
next
  case (3 e1 e2 b2)
  then show ?case by auto
next
  case (4 e1 e2 e1' e2')
  then show ?case
    by (metis less_gexp_aux.simps(7) less_imp_not_less less_linear)
next
  case ("5_1" e1 e2 v va)
  then show ?case by auto
next
  case ("5_2" e1 e2 v va)
  then show ?case by auto
next
  case ("5_3" e1 e2 v va)
  then show ?case by auto
next
  case (6 e1 e2 b2)
  then show ?case by auto
next
  case (7 e1 e2 e1' e2')
  then show ?case by auto
next
  case (8 e1 e2 e1' e2')
  then show ?case
    by (metis less_gexp_aux.simps(13) less_imp_not_less less_linear)
next
  case ("9_1" e1 e2 v va)
  then show ?case by auto
next
  case ("9_2" e1 e2 v va)
  then show ?case by auto
next
  case (10 vb vc v va)
  then show ?case by auto
next
  case (11 vb vc v va)
  then show ?case by auto
next
  case ("12_1" vb vc v)
  then show ?case by auto
```

```
next
  case ("12_2" vb vc v va)
  then show ?case by auto
next
  case ("12_3" vb vc v va)
  then show ?case by auto
next
  case (13 g1 g2 g1' g2')
  then show ?case by auto
next
  case ("14_1" g1 g2 v)
  then show ?case by auto
next
  case ("14_2" g1 g2 v va)
  then show ?case by auto
next
  case ("14_3" g1 g2 v va)
  then show ?case by auto
next
  case ("14_4" g1 g2 v va)
  then show ?case by auto
qed

lemma less_gexp_antisym: "(x::'a gexp) < y = (¬(y < x) ∧ (x ≠ y))"
  apply (simp add: Let_def)
  apply standard
  using less_gexp_aux_antisym apply blast
  apply clarsimp
  by (induct x, auto)

lemma less_gexp_aux_trans: "less_gexp_aux x y ⟹ less_gexp_aux y z ⟹ less_gexp_aux x z"
proof(induct x y arbitrary: z rule: less_gexp_aux.induct)
case (1 b1 b2)
  then show ?case by (cases z, auto)
next
  case ("2_1" b1 v va)
  then show ?case by (cases z, auto)
next
  case ("2_2" b1 v va)
  then show ?case by (cases z, auto)
next
  case ("2_3" b1 v va)
  then show ?case by (cases z, auto)
next
  case ("2_4" b1 v va)
  then show ?case by (cases z, auto)
next
  case (3 e1 e2 b2)
  then show ?case by (cases z, auto)
next
  case (4 e1 e2 e1' e2')
  then show ?case
    apply (cases z)
        apply simp
       apply (metis dual_order.strict_trans less_gexp_aux.simps(7))
    by auto
next
  case ("5_1" e1 e2 v va)
  then show ?case by (cases z, auto)
next
  case ("5_2" e1 e2 v va)
  then show ?case by (cases z, auto)
next
```

```
  case ("5_3" e1 e2 v va)
  then show ?case by (cases z, auto)
next
  case (6 e1 e2 b2)
  then show ?case by (cases z, auto)
next
  case (7 e1 e2 e1' e2')
  then show ?case by (cases z, auto)
next
  case (8 e1 e2 e1' e2')
  then show ?case
    apply (cases z)
        apply simp
       apply simp
      apply (metis dual_order.strict_trans less_gexp_aux.simps(13))
    by auto
next
  case ("9_1" e1 e2 v va)
  then show ?case by (cases z, auto)
next
  case ("9_2" e1 e2 v va)
  then show ?case by (cases z, auto)
next
  case (10 vb vc v va)
  then show ?case by (cases z, auto)
next
  case (11 vb vc v va)
  then show ?case by (cases z, auto)
next
  case ("12_1" vb vc v)
  then show ?case by (cases z, auto)
next
  case ("12_2" vb vc v va)
  then show ?case by (cases z, auto)
next
  case ("12_3" vb vc v va)
  then show ?case by (cases z, auto)
next
  case (13 g1 g2 g1' g2')
  then show ?case by (cases z, auto)
next
  case ("14_1" g1 g2 v)
  then show ?case by (cases z, auto)
next
  case ("14_2" g1 g2 v va)
  then show ?case by (cases z, auto)
next
  case ("14_3" g1 g2 v va)
  then show ?case by (cases z, auto)
next
  case ("14_4" g1 g2 v va)
  then show ?case by (cases z, auto)
qed

lemma less_gexp_trans: "(x::'a gexp) < y ⟹ y < z ⟹ x < z"
  apply (simp add: Let_def)
  by (metis (no_types, lifting) dual_order.strict_trans less_gexp_aux_trans less_imp_not_less)

instance proof
  fix x y z :: "'a gexp"
  show "(x < y) = (x ≤ y ∧ ¬ y ≤ x)"
    by (metis less_gexp_antisym less_eq_gexp_def)
  show "(x ≤ x)"
```

```
    by (simp add: less_eq_gexp_def)
  show "x ≤ y ⟹ y ≤ z ⟹ x ≤ z"
    by (metis less_eq_gexp_def less_gexp_trans)
  show "x ≤ y ⟹ y ≤ x ⟹ x = y"
    unfolding less_eq_gexp_def using less_gexp_antisym by blast
  show "x ≤ y ∨ y ≤ x"
    unfolding less_eq_gexp_def using less_gexp_antisym by blast
qed
end

end
```

2.5 FSet Utilities (FSet_Utils)

This theory provides various additional lemmas, definitions, and syntax over the fset data type.

```
theory FSet_Utils
  imports "HOL-Library.FSet"
begin

notation (latex output)
  "FSet.fempty" ("∅") and
  "FSet.fmember" ("∈")

syntax (ASCII)
  "_fBall"       :: "pttrn ⇒ 'a fset ⇒ bool ⇒ bool"      ("(3ALL (_/:_)./ _)" [0, 0, 10] 10)
  "_fBex"        :: "pttrn ⇒ 'a fset ⇒ bool ⇒ bool"      ("(3EX (_/:_)./ _)" [0, 0, 10] 10)
  "_fBex1"       :: "pttrn ⇒ 'a fset ⇒ bool ⇒ bool"      ("(3EX! (_/:_)./ _)" [0, 0, 10] 10)

syntax (input)
  "_fBall"       :: "pttrn ⇒ 'a fset ⇒ bool ⇒ bool"      ("(3! (_/:_)./ _)" [0, 0, 10] 10)
  "_fBex"        :: "pttrn ⇒ 'a fset ⇒ bool ⇒ bool"      ("(3? (_/:_)./ _)" [0, 0, 10] 10)
  "_fBex1"       :: "pttrn ⇒ 'a fset ⇒ bool ⇒ bool"      ("(3?! (_/:_)./ _)" [0, 0, 10] 10)

syntax
  "_fBall"       :: "pttrn ⇒ 'a fset ⇒ bool ⇒ bool"      ("(3∀ (_/|∈|_)./ _)" [0, 0, 10] 10)
  "_fBex"        :: "pttrn ⇒ 'a fset ⇒ bool ⇒ bool"      ("(3∃ (_/|∈|_)./ _)" [0, 0, 10] 10)
  "_fBnex"       :: "pttrn ⇒ 'a fset ⇒ bool ⇒ bool"      ("(3∄ (_/|∈|_)./ _)" [0, 0, 10] 10)
  "_fBex1"       :: "pttrn ⇒ 'a fset ⇒ bool ⇒ bool"      ("(3∃!(_/|∈|_)./ _)" [0, 0, 10] 10)

translations
  "∀x|∈|A. P" ⇌ "CONST fBall A (λx. P)"
  "∃x|∈|A. P" ⇌ "CONST fBex  A (λx. P)"
  "∄x|∈|A. P" ⇌ "CONST fBall A (λx. ¬P)"
  "∃!x|∈|A. P" ⇀ "∃!x. x |∈| A ∧ P"

lemma fset_of_list_remdups [simp]: "fset_of_list (remdups l) = fset_of_list l"
  apply (induct l)
   apply simp
  by (simp add: finsert_absorb fset_of_list_elem)

definition "fSum ≡ fsum (λx. x)"

lemma fset_both_sides: "(Abs_fset s = f) = (fset (Abs_fset s) = fset f)"
  by (simp add: fset_inject)

lemma Abs_ffilter: "(ffilter f s = s') = ({e ∈ (fset s). f e} = (fset s'))"
  by (simp add: ffilter_def fset_both_sides Abs_fset_inverse Set.filter_def)

lemma size_ffilter_card: "size (ffilter f s) = card ({e ∈ (fset s). f e})"
  by (simp add: ffilter_def fset_both_sides Abs_fset_inverse Set.filter_def)

lemma ffilter_empty [simp]: "ffilter f {|||} = {|||}"
```

```
  by auto

lemma ffilter_finsert:
  "ffilter f (finsert a s) = (if f a then finsert a (ffilter f s) else (ffilter f s))"
  apply simp
  apply standard
   apply (simp add: ffilter_def fset_both_sides Abs_fset_inverse)
   apply auto[1]
  apply (simp add: ffilter_def fset_both_sides Abs_fset_inverse)
  by auto

lemma fset_equiv: "(f1 = f2) = (fset f1 = fset f2)"
  by (simp add: fset_inject)

lemma finsert_equiv: "(finsert e f = f') = (insert e (fset f) = (fset f'))"
  by (simp add: finsert_def fset_both_sides Abs_fset_inverse)

lemma filter_elements:
  "x |∈| Abs_fset (Set.filter f (fset s)) = (x ∈ (Set.filter f (fset s)))"
  by (metis ffilter.rep_eq fset_inverse notin_fset)

lemma sorted_list_of_fempty [simp]: "sorted_list_of_fset {|||} = []"
  by (simp add: sorted_list_of_fset_def)

lemma fmember_implies_member: "e |∈| f ⟹ e ∈ fset f"
  by (simp add: fmember_def)

lemma fold_union_ffUnion: "fold (|∪|) l {|||} = ffUnion (fset_of_list l)"
by(induct l rule: rev_induct, auto)

lemma filter_filter:
  "ffilter P (ffilter Q xs) = ffilter (λx. Q x ∧ P x) xs"
  by auto

lemma fsubset_strict:
  "x2 |⊂| x1 ⟹ ∃e. e |∈| x1 ∧ e |∉| x2"
  by auto

lemma fsubset:
  "x2 |⊂| x1 ⟹ ∄e. e |∈| x2 ∧ e |∉| x1"
  by auto

lemma size_fsubset_elem:
  assumes "∃e. e |∈| x1 ∧ e |∉| x2"
      and "∄e. e |∈| x2 ∧ e |∉| x1"
    shows "size x2 < size x1"
  using assms
  apply (simp add: fmember_def)
  by (metis card_seteq finite_fset linorder_not_le subsetI)

lemma size_fsubset: "x2 |⊂| x1 ⟹ size x2 < size x1"
  by (metis fsubset fsubset_strict size_fsubset_elem)

definition fremove :: "'a ⇒ 'a fset ⇒ 'a fset"
  where [code_abbrev]: "fremove x A = A - {|x|}"

lemma arg_cong_ffilter:
  "∀e |∈| f. p e = p' e ⟹ ffilter p f = ffilter p' f"
  by auto

lemma ffilter_singleton: "f e ⟹ ffilter f {|e|} = {|e|}"
  apply (simp add: ffilter_def fset_both_sides Abs_fset_inverse)
  by auto
```

```
lemma fset_eq_alt: "(x = y) = (x |⊆| y ∧ size x = size y)"
  by (metis exists_least_iff le_less size_fsubset)

lemma ffold_empty [simp]: "ffold f b {||} = b"
  by (simp add: ffold_def)

lemma sorted_list_of_fset_sort:
  "sorted_list_of_fset (fset_of_list l) = sort (remdups l)"
  by (simp add: fset_of_list.rep_eq sorted_list_of_fset.rep_eq sorted_list_of_set_sort_remdups)

lemma fMin_Min: "fMin (fset_of_list l) = Min (set l)"
  by (simp add: fMin.F.rep_eq fset_of_list.rep_eq)

lemma sorted_hd_Min:
  "sorted l ⟹
   l ≠ [] ⟹
   hd l = Min (set l)"
  by (metis List.finite_set Min_eqI eq_iff hd_Cons_tl insertE list.set_sel(1) list.simps(15) sorted.simps(2))

lemma hd_sort_Min: "l ≠ [] ⟹ hd (sort l) = Min (set l)"
  by (metis sorted_hd_Min set_empty set_sort sorted_sort)

lemma hd_sort_remdups: "hd (sort (remdups l)) = hd (sort l)"
  by (metis hd_sort_Min remdups_eq_nil_iff set_remdups)

lemma exists_fset_of_list: "∃l. f = fset_of_list l"
  using exists_fset_of_list by fastforce

lemma hd_sorted_list_of_fset:
  "s ≠ {||} ⟹ hd (sorted_list_of_fset s) = (fMin s)"
  apply (insert exists_fset_of_list[of s])
  apply (erule exE)
  apply simp
  apply (simp add: sorted_list_of_fset_sort fMin_Min hd_sort_remdups)
  by (metis fset_of_list_simps(1) hd_sort_Min)

lemma fminus_filter_singleton:
  "fset_of_list l |-| {|x|} = fset_of_list (filter (λe. e ≠ x) l)"
  by auto

lemma card_minus_fMin:
  "s ≠ {||} ⟹ card (fset s - {fMin s}) < card (fset s)"
  by (metis Min_in bot_fset.rep_eq card_Diff1_less fMin.F.rep_eq finite_fset fset_equiv)


function ffold_ord :: "(('a::linorder) ⇒ 'b ⇒ 'b) ⇒ 'a fset ⇒ 'b ⇒ 'b" where
  "ffold_ord f s b = (
    if s = {||} then
      b
    else
      let
        h = fMin s;
        t = s - {|h|}
      in
        ffold_ord f t (f h b)
  )"
  by auto
termination
  apply (relation "measures [λ(a, s, ab). size s]")
   apply simp
  by (simp add: card_minus_fMin)
```

```
lemma sorted_list_of_fset_Cons:
  "∃h t. (sorted_list_of_fset (finsert s ss)) = h#t"
  apply (simp add: sorted_list_of_fset_def)
  by (cases "insort s (sorted_list_of_set (fset ss - {s}))", auto)

lemma list_eq_hd_tl:
  "l ≠ [] ⟹
   hd l = h ⟹
   tl l = t ⟹
   l = (h#t)"
  by auto

lemma fset_of_list_sort: "fset_of_list l = fset_of_list (sort l)"
  by (simp add: fset_of_list.abs_eq)

lemma exists_sorted_distinct_fset_of_list:
  "∃l. sorted l ∧ distinct l ∧ f = fset_of_list l"
  by (metis distinct_sorted_list_of_set sorted_list_of_fset.rep_eq sorted_list_of_fset_simps(2) sorted_sorted_list

lemma fset_of_list_empty [simp]: "(fset_of_list l = {|||}) = (l = [])"
  by (metis fset_of_list.rep_eq fset_of_list_simps(1) set_empty)

lemma ffold_ord_cons: assumes sorted: "sorted (h#t)"
    and distinct: "distinct (h#t)"
  shows "ffold_ord f (fset_of_list (h#t)) b = ffold_ord f (fset_of_list t) (f h b)"
proof-
  have h_is_min: "h = fMin (fset_of_list (h#t))"
    by (metis sorted fMin_Min list.sel(1) list.simps(3) sorted_hd_Min)
  have remove_min: "fset_of_list t = (fset_of_list (h#t)) - {|h|}"
    using distinct fset_of_list_elem by force
  show ?thesis
    apply (simp only: ffold_ord.simps[of f "fset_of_list (h#t)"])
    by (metis h_is_min remove_min fset_of_list_empty list.distinct(1))
qed

lemma sorted_distinct_ffold_ord: assumes "sorted l"
      and "distinct l"
    shows "ffold_ord f (fset_of_list l) b = fold f l b"
  using assms
  apply (induct l arbitrary: b)
   apply simp
  by (metis distinct.simps(2) ffold_ord_cons fold_simps(2) sorted.simps(2))

lemma ffold_ord_fold_sorted: "ffold_ord f s b = fold f (sorted_list_of_fset s) b"
  by (metis exists_sorted_distinct_fset_of_list sorted_distinct_ffold_ord distinct_remdups_id sorted_list_of_fset_
sorted_sort_id)

context includes fset.lifting begin
  lift_definition fprod :: "'a fset ⇒ 'b fset ⇒ ('a × 'b) fset " (infixr "|×|" 80) is "λa b. fset a ×
fset b"
    by simp

  lift_definition fis_singleton :: "'a fset ⇒ bool" is "λA. is_singleton (fset A)".
end

lemma fprod_empty_l: "{|||} |×| a = {|||}"
  using bot_fset_def fprod.abs_eq by force

lemma fprod_empty_r: "a |×| {|||} = {|||}"
  by (simp add: fprod_def bot_fset_def Abs_fset_inverse)

lemmas fprod_empty = fprod_empty_l fprod_empty_r
```

```
lemma fprod_finsert: "(finsert a as) |×| (finsert b bs) =
   finsert (a, b) (fimage (λb. (a, b)) bs |∪| fimage (λa. (a, b)) as |∪| (as |×| bs))"
  apply (simp add: fprod_def fset_both_sides Abs_fset_inverse)
  by auto

lemma fprod_member:
  "x |∈| xs ⟹
   y |∈| ys ⟹
   (x, y) |∈| xs |×| ys"
  by (simp add: fmember_def fprod_def Abs_fset_inverse)

lemma fprod_subseteq:
  "x |⊆| x' ∧ y |⊆| y' ⟹ x |×| y |⊆| x' |×| y'"
  apply (simp add: fprod_def less_eq_fset_def Abs_fset_inverse)
  by auto

lemma fimage_fprod:
  "(a, b) |∈| A |×| B ⟹ f a b |∈| (λ(x, y). f x y) |`| (A |×| B)"
  by force

lemma fprod_singletons: "{|a|} |×| {|b|} = {|(a, b)|}"
  apply (simp add: fprod_def)
  by (metis fset_inverse fset_simps(1) fset_simps(2))

lemma fprod_equiv:
  "(fset (f |×| f') = s) = (((fset f) × (fset f')) = s)"
  by (simp add: fprod_def Abs_fset_inverse)

lemma fis_singleton_alt: "fis_singleton f = (∃e. f = {|e|})"
  by (metis fis_singleton.rep_eq fset_inverse fset_simps(1) fset_simps(2) is_singleton_def)

lemma singleton_singleton [simp]: "fis_singleton {|a|}"
  by (simp add: fis_singleton_def)

lemma not_singleton_empty [simp]: "¬ fis_singleton {||}"
  apply (simp add: fis_singleton_def)
  by (simp add: is_singleton_altdef)

lemma fis_singleton_fthe_elem:
  "fis_singleton A ⟷ A = {|fthe_elem A|}"
  by (metis fis_singleton_alt fthe_felem_eq)

lemma fBall_ffilter:
  "∀x |∈| X. f x ⟹ ffilter f X = X"
  by auto

lemma fBall_ffilter2:
  "X = Y ⟹
   ∀x |∈| X. f x ⟹
   ffilter f X = Y"
  by auto

lemma size_fset_of_list: "size (fset_of_list l) = length (remdups l)"
  apply (induct l)
   apply simp
  by (simp add: fset_of_list.rep_eq insert_absorb)

lemma size_fsingleton: "(size f = 1) = (∃e. f = {|e|})"
  apply (insert exists_fset_of_list[of f])
  apply clarify
  apply (simp only: size_fset_of_list)
  apply (simp add: fset_of_list_def fset_both_sides Abs_fset_inverse)
  by (metis List.card_set One_nat_def card.insert card_1_singletonE card.empty empty_iff finite.intros(1))
```

```
lemma ffilter_mono: "(ffilter X xs = f) ⟹ ∀x |∈| xs. X x = Y x ⟹ (ffilter Y xs = f)"
  by auto

lemma size_fimage: "size (fimage f s) ≤ size s"
  apply (induct s)
   apply simp
  by (simp add: card_insert_if)

lemma size_ffilter: "size (ffilter P f) ≤ size f"
  apply (induct f)
   apply simp
  apply (simp only: ffilter_finsert)
  apply (case_tac "P x")
   apply (simp add: fmember.rep_eq)
  by (simp add: card_insert_if)

lemma fimage_size_le: "⋀f s. size s ≤ n ⟹ size (fimage f s) ≤ n"
  using le_trans size_fimage by blast

lemma ffilter_size_le: "⋀f s. size s ≤ n ⟹ size (ffilter f s) ≤ n"
  using dual_order.trans size_ffilter by blast

lemma set_membership_eq: "A = B ⟷ (λx. Set.member x A) = (λx. Set.member x B)"
  apply standard
   apply simp
  by (meson equalityI subsetI)

lemmas ffilter_eq_iff = Abs_ffilter set_membership_eq fun_eq_iff

lemma size_le_1: "size f ≤ 1 = (f = {||} ∨ (∃e. f = {|e|}))"
  apply standard
   apply (metis bot.not_eq_extremum gr_implies_not0 le_neq_implies_less less_one size_fsingleton size_fsubset)
  by auto

lemma size_gt_1: "1 < size f ⟹ ∃e1 e2 f'. e1 ≠ e2 ∧ f = finsert e1 (finsert e2 f')"
  apply (induct f)
   apply simp
  apply (rule_tac x=x in exI)
  by (metis finsertCI leD not_le_imp_less size_le_1)

end
```

3 Models

In this chapter, we present our formalisation of EFSMs from [2]. We first define transitions, before defining EFSMs as finite sets of transitions between states. Finally, we provide a framework of function definitions and key lemmas such that LTL properties over EFSMs can be more easily specified and proven.

3.1 Transitions (Transition)

Here we define the transitions which make up EFSMs. As per [2], each transition has a label and an arity and, optionally, guards, outputs, and updates. To implement this, we use the record type such that each component of the transition can be accessed.

```
theory Transition
imports GExp
begin

type_synonym label = String.literal
type_synonym arity = nat
type_synonym guard = "vname gexp"
type_synonym inputs = "value list"
type_synonym outputs = "value option list"
type_synonym output_function = "vname aexp"
type_synonym update_function = "nat × vname aexp"
record transition =
  Label :: String.literal
  Arity :: nat
  Guards :: "guard list"
  Outputs :: "output_function list"
  Updates :: "update_function list"

definition same_structure :: "transition ⇒ transition ⇒ bool" where
  "same_structure t1 t2 = (
    Label t1 = Label t2 ∧
    Arity t1 = Arity t2 ∧
    length (Outputs t1) = length (Outputs t2)
  )"

definition enumerate_inputs :: "transition ⇒ nat set" where
  "enumerate_inputs t = (⋃ (set (map enumerate_gexp_inputs (Guards t)))) ∪
                        (⋃ (set (map enumerate_aexp_inputs (Outputs t)))) ∪
                        (⋃ (set (map (λ(_, u). enumerate_aexp_inputs u) (Updates t))))"

definition max_input :: "transition ⇒ nat option" where
  "max_input t = (if enumerate_inputs t = {} then None else Some (Max (enumerate_inputs t)))"

definition total_max_input :: "transition ⇒ nat" where
  "total_max_input t = (case max_input t of None ⇒ 0 | Some a ⇒ a)"

definition enumerate_regs :: "transition ⇒ nat set" where
  "enumerate_regs t = (⋃ (set (map GExp.enumerate_regs (Guards t)))) ∪
                      (⋃ (set (map AExp.enumerate_regs (Outputs t)))) ∪
                      (⋃ (set (map (λ(_, u). AExp.enumerate_regs u) (Updates t)))) ∪
                      (⋃ (set (map (λ(r, _). AExp.enumerate_regs (V (R r))) (Updates t))))"

definition max_reg :: "transition ⇒ nat option" where
  "max_reg t = (if enumerate_regs t = {} then None else Some (Max (enumerate_regs t)))"
```

```
definition total_max_reg :: "transition ⇒ nat" where
  "total_max_reg t = (case max_reg t of None ⇒ 0 | Some a ⇒ a)"

definition enumerate_ints :: "transition ⇒ int set" where
  "enumerate_ints t = (⋃ (set (map enumerate_gexp_ints (Guards t)))) ∪
                      (⋃ (set (map enumerate_aexp_ints (Outputs t)))) ∪
                      (⋃ (set (map (λ(_, u). enumerate_aexp_ints u) (Updates t)))) ∪
                      (⋃ (set (map (λ(r, _). enumerate_aexp_ints (V (R r))) (Updates t))))"

definition valid_transition :: "transition ⇒ bool" where
  "valid_transition t = (∀ i ∈ enumerate_inputs t. i < Arity t)"

definition can_take :: "nat ⇒ vname gexp list ⇒ inputs ⇒ registers ⇒ bool" where
  "can_take a g i r = (length i = a ∧ apply_guards g (join_ir i r))"

lemma can_take_empty [simp]: "length i = a ⟹ can_take a [] i c"
  by (simp add: can_take_def)

lemma can_take_subset_append:
  assumes "set (Guards t) ⊆ set (Guards t')"
  shows "can_take a (Guards t @ Guards t') i c = can_take a (Guards t') i c"
  using assms
  by (simp add: apply_guards_subset_append can_take_def)

definition "can_take_transition t i r = can_take (Arity t) (Guards t) i r"

lemmas can_take = can_take_def can_take_transition_def

lemma can_take_transition_empty_guard:
  "Guards t = [] ⟹ ∃ i. can_take_transition t i c"
  by (simp add: can_take_transition_def can_take_def Ex_list_of_length)

lemma can_take_subset: "length i = Arity t ⟹
   Arity t = Arity t' ⟹
   set (Guards t') ⊆ set (Guards t) ⟹
   can_take_transition t i r ⟹
   can_take_transition t' i r"
  by (simp add: can_take_transition_def can_take_def apply_guards_subset)

lemma valid_list_can_take:
  "∀ g ∈ set (Guards t). valid g ⟹ ∃ i. can_take_transition t i c"
  by (simp add: can_take_transition_def can_take_def apply_guards_def valid_def Ex_list_of_length)

lemma cant_take_if:
  "∃ g ∈ set (Guards t). gval g (join_ir i r) ≠ true ⟹
   ¬ can_take_transition t i r"
  using apply_guards_cons apply_guards_rearrange can_take_def can_take_transition_def by blast

definition apply_outputs :: "'a aexp list ⇒ 'a datastate ⇒ value option list" where
  "apply_outputs p s = map (λp. aval p s) p"

abbreviation "evaluate_outputs t i r ≡ apply_outputs (Outputs t) (join_ir i r)"

lemma apply_outputs_nth:
  "i < length p ⟹ apply_outputs p s ! i = aval (p ! i) s"
  by (simp add: apply_outputs_def)

lemmas apply_outputs = datastate apply_outputs_def value_plus_def value_minus_def value_times_def

lemma apply_outputs_empty [simp]: "apply_outputs [] s = []"
  by (simp add: apply_outputs_def)
```

```
lemma apply_outputs_preserves_length: "length (apply_outputs p s) = length p"
  by (simp add: apply_outputs_def)

lemma apply_outputs_literal: assumes "P ! r = L v"
      and "r < length P"
    shows "apply_outputs P s ! r = Some v"
  by (simp add: assms apply_outputs_nth)

lemma apply_outputs_register: assumes "r < length P"
  shows "apply_outputs (list_update P r (V (R p))) (join_ir i c) ! r = c $ p"
  by (metis apply_outputs_nth assms aval.simps(2) join_ir_R length_list_update nth_list_update_eq)

lemma apply_outputs_unupdated: assumes "ia ≠ r"
      and "ia < length P"
    shows "apply_outputs P j ! ia = apply_outputs (list_update P r v)j ! ia"
  by (metis apply_outputs_nth assms(1) assms(2) length_list_update nth_list_update_neq)

definition apply_updates :: "update_function list ⇒ vname datastate ⇒ registers ⇒ registers" where
  "apply_updates u old = fold (λh r. r(fst h $:= aval (snd h) old)) u"

abbreviation "evaluate_updates t i r ≡ apply_updates (Updates t) (join_ir i r) r"

lemma apply_updates_cons: "ra ≠ r ⟹
       apply_updates u (join_ir ia c) c $ ra = apply_updates ((r, a) # u) (join_ir ia c) c $ ra"
proof(induct u rule: rev_induct)
  case Nil
  then show ?case
    by (simp add: apply_updates_def)
next
  case (snoc u us)
  then show ?case
    apply (cases u)
    apply (simp add: apply_updates_def)
    by (case_tac "ra = aa", auto)
qed

lemma update_twice:
  "apply_updates [(r, a), (r, b)] s regs = regs (r $:= aval b s)"
  by (simp add: apply_updates_def)

lemma r_not_updated_stays_the_same:
  "r ∉ fst ` set U ⟹ apply_updates U c d $ r = d $ r"
  using apply_updates_def
  by (induct U rule: rev_induct, auto)

definition rename_regs :: "(nat ⇒ nat) ⇒ transition ⇒ transition" where
  "rename_regs f t = t⦇
      Guards  := map (GExp.rename_regs f) (Guards t),
      Outputs := map (AExp.rename_regs f) (Outputs t),
      Updates := map (λ(r, u). (f r, AExp.rename_regs f u)) (Updates t)
    ⦈"

definition eq_upto_rename_strong :: "transition ⇒ transition ⇒ bool" where
  "eq_upto_rename_strong t1 t2 = (∃f. bij f ∧ rename_regs f t1 = t2)"

inductive eq_upto_rename :: "transition ⇒ transition ⇒ bool" where
  "Label t1 = Label t2 ⟹
   Arity t2 = Arity t2 ⟹
   apply_guards (map (GExp.rename_regs f) (Guards t1)) = apply_guards (Guards t2) ⟹

   apply_outputs (map (AExp.rename_regs f) (Outputs t1)) = apply_outputs (Outputs t2) ⟹

   apply_updates (map (λ(r, u). (f r, AExp.rename_regs f u)) (Updates t1)) = apply_updates (Updates t2)
```

```
⟹

   eq_upto_rename t1 t2"

end
```

3.1.1 Transition Lexorder

This theory defines a lexicographical ordering on transitions such that we can convert from the set representation of EFSMs to a sorted list that we can recurse over.

```
theory Transition_Lexorder
  imports "Transition"
    GExp_Lexorder
    "HOL-Library.Product_Lexorder"
begin

instantiation "transition_ext" :: (linorder) linorder begin

definition less_transition_ext :: "'a::linorder transition_scheme ⇒ 'a transition_scheme ⇒ bool" where
"less_transition_ext t1 t2 = ((Label t1, Arity t1, Guards t1, Outputs t1, Updates t1, more t1) < (Label
t2, Arity t2, Guards t2, Outputs t2, Updates t2, more t2))"

definition less_eq_transition_ext :: "'a::linorder transition_scheme ⇒ 'a transition_scheme ⇒ bool" where
"less_eq_transition_ext t1 t2 = (t1 < t2 ∨ t1 = t2)"

instance
  apply standard
  unfolding less_eq_transition_ext_def less_transition_ext_def
      apply auto[1]
     apply simp
  using less_trans apply blast
  using less_imp_not_less apply blast
  by (metis Pair_inject equality neqE)
end

end
```

3.2 Extended Finite State Machines (EFSM)

This theory defines extended finite state machines as presented in [2]. States are indexed by natural numbers, however, since transition matrices are implemented by finite sets, the number of reachable states in S is necessarily finite. For ease of implementation, we implicitly make the initial state zero for all EFSMs. This allows EFSMs to be represented purely by their transition matrix which, in this implementation, is a finite set of tuples of the form $((s_1, s_2), t)$ in which s_1 is the origin state, s_2 is the destination state, and t is a transition.

```
theory EFSM
  imports "HOL-Library.FSet" Transition FSet_Utils
begin

declare One_nat_def [simp del]

type_synonym cfstate = nat
type_synonym inputs = "value list"
type_synonym outputs = "value option list"

type_synonym action = "(label × inputs)"
type_synonym execution = "action list"
type_synonym observation = "outputs list"
type_synonym transition_matrix = "((cfstate × cfstate) × transition) fset"

no_notation relcomp (infixr "O" 75) and comp (infixl "o" 55)
```

```
type_synonym event = "(label × inputs × value list)"
type_synonym trace = "event list"
type_synonym log = "trace list"

definition Str :: "string ⇒ value" where
  "Str s ≡ value.Str (String.implode s)"

lemma str_not_num: "Str s ≠ Num x1"
  by (simp add: Str_def)

definition S :: "transition_matrix ⇒ nat fset" where
  "S m = (fimage (λ((s, s'), t). s) m) |∪| fimage (λ((s, s'), t). s') m"

lemma S_ffUnion: "S e = ffUnion (fimage (λ((s, s'), _). {|s, s'|}) e)"
  unfolding S_def
  by(induct e, auto)
```

3.2.1 Possible Steps

From a given state, the possible steps for a given action are those transitions with labels which correspond to the action label, arities which correspond to the number of inputs, and guards which are satisfied by those inputs.

```
definition possible_steps :: "transition_matrix ⇒ cfstate ⇒ registers ⇒ label ⇒ inputs ⇒ (cfstate ×
transition) fset" where
  "possible_steps e s r l i = fimage (λ((origin, dest), t). (dest, t)) (ffilter (λ((origin, dest), t). origin
= s ∧ (Label t) = l ∧ (length i) = (Arity t) ∧ apply_guards (Guards t) (join_ir i r)) e)"

lemma possible_steps_finsert:
"possible_steps (finsert ((s, s'), t) e) ss r l i = (
  if s = ss ∧ (Label t) = l ∧ (length i) = (Arity t) ∧ apply_guards (Guards t) (join_ir i r) then
    finsert (s', t) (possible_steps e s r l i)
  else
    possible_steps e ss r l i
)"
  by (simp add: possible_steps_def ffilter_finsert)


lemma split_origin:
"ffilter (λ((origin, dest), t). origin = s ∧ Label t = l ∧ can_take_transition t i r) e =
ffilter (λ((origin, dest), t). Label t = l ∧ can_take_transition t i r) (ffilter (λ((origin, dest), t).
origin = s) e)"
  by auto

lemma split_label:
"ffilter (λ((origin, dest), t). origin = s ∧ Label t = l ∧ can_take_transition t i r) e =
ffilter (λ((origin, dest), t). origin = s ∧ can_take_transition t i r) (ffilter (λ((origin, dest), t). Label
t = l) e)"
  by auto

lemma possible_steps_empty_guards_false:
  "∀((s1, s2), t) |∈| ffilter (λ((origin, dest), t). Label t = l) e. ¬can_take_transition t i r ⟹
  possible_steps e s r l i = {|||}"
  apply (simp add: possible_steps_def can_take[symmetric] split_label)
  by (simp add: Abs_ffilter fBall_def Ball_def)

lemma fmember_possible_steps: "(s', t) |∈| possible_steps e s r l i = (((s, s'), t) ∈ {((origin, dest),
t) ∈ fset e. origin = s ∧ Label t = l ∧ length i = Arity t ∧ apply_guards (Guards t) (join_ir i r)})"
  apply (simp add: possible_steps_def ffilter_def fimage_def fmember_def Abs_fset_inverse)
  by force

lemma possible_steps_alt_aux:
  "possible_steps e s r l i = {|(d, t)|} ⟹
      ffilter (λ((origin, dest), t). origin = s ∧ Label t = l ∧ length i = Arity t ∧ apply_guards (Guards
t) (join_ir i r)) e = {|((s, d), t)|}"
```

```
proof(induct e)
  case empty
  then show ?case
    by (simp add: fempty_not_finsert possible_steps_def)
next
  case (insert x e)
  then show ?case
    apply (case_tac x)
    subgoal for a b
      apply (case_tac a)
      subgoal for aa _
        apply (simp add: possible_steps_def)
        apply (simp add: ffilter_finsert)
        apply (case_tac "aa = s ∧ Label b = l ∧ length i = Arity b ∧ apply_guards (Guards b) (join_ir i
r)")
        by auto
      done
    done
qed

lemma possible_steps_alt: "(possible_steps e s r l i = {|(d, t)|}) = (ffilter
     (λ((origin, dest), t). origin = s ∧ Label t = l ∧ length i = Arity t ∧ apply_guards (Guards t) (join_ir
i r))
     e = {|((s, d), t)|})"
  apply standard
   apply (simp add: possible_steps_alt_aux)
  by (simp add: possible_steps_def)

lemma possible_steps_alt3: "(possible_steps e s r l i = {|(d, t)|}) = (ffilter
     (λ((origin, dest), t). origin = s ∧ Label t = l ∧ can_take_transition t i r)
     e = {|((s, d), t)|})"
  apply standard
   apply (simp add: possible_steps_alt_aux can_take)
  by (simp add: possible_steps_def can_take)

lemma possible_steps_alt_atom: "(possible_steps e s r l i = {|dt|}) = (ffilter
     (λ((origin, dest), t). origin = s ∧ Label t = l ∧ can_take_transition t i r)
     e = {|((s, fst dt), snd dt)|})"
  apply (cases dt)
  by (simp add: possible_steps_alt can_take_transition_def can_take_def)

lemma possible_steps_alt2: "(possible_steps e s r l i = {|(d, t)|}) = (
     (ffilter (λ((origin, dest), t). Label t = l ∧ length i = Arity t ∧ apply_guards (Guards t) (join_ir
i r)) (ffilter (λ((origin, dest), t). origin = s) e) = {|((s, d), t)|}))"
  apply (simp add: possible_steps_alt)
  apply (simp only: filter_filter)
  apply (rule arg_cong [of "(λ((origin, dest), t). origin = s ∧ Label t = l ∧ length i = Arity t ∧ apply_guards
(Guards t) (join_ir i r))"])
  by (rule ext, auto)

lemma possible_steps_single_out:
"ffilter (λ((origin, dest), t). origin = s) e = {|((s, d), t)|} ⟹
Label t = l ∧ length i = Arity t ∧ apply_guards (Guards t) (join_ir i r) ⟹
possible_steps e s r l i = {|(d, t)|}"
  apply (simp add: possible_steps_alt2 Abs_ffilter)
  by blast

lemma possible_steps_singleton: "(possible_steps e s r l i = {|(d, t)|}) =
    ({((origin, dest), t) ∈ fset e. origin = s ∧ Label t = l ∧ length i = Arity t ∧ apply_guards (Guards
t) (join_ir i r)} = {((s, d), t)})"
  apply (simp add: possible_steps_alt Abs_ffilter Set.filter_def)
  by fast
```

```
lemma possible_steps_apply_guards:
  "possible_steps e s r l i = {|(s', t)|} ⟹
   apply_guards (Guards t) (join_ir i r)"
  apply (simp add: possible_steps_singleton)
  by auto

lemma possible_steps_empty:
  "(possible_steps e s r l i = {||}) = (∀((origin, dest), t) ∈ fset e. origin ≠ s ∨ Label t ≠ l ∨ ¬ can_take_tra
t i r)"
  apply (simp add: can_take_transition_def can_take_def)
  apply (simp add: possible_steps_def Abs_ffilter Set.filter_def)
  by auto

lemma singleton_dest:
  assumes "fis_singleton (possible_steps e s r aa b)"
      and "fthe_elem (possible_steps e s r aa b) = (baa, aba)"
    shows "((s, baa), aba) |∈| e"
  using assms
  apply (simp add: fis_singleton_fthe_elem)
  using possible_steps_alt_aux by force

lemma no_outgoing_transitions:
"ffilter (λ((s', _), _). s = s') e = {||} ⟹
possible_steps e s r l i = {||}"
  apply (simp add: possible_steps_def)
  by auto

lemma ffilter_split: "ffilter (λ((origin, dest), t). origin = s ∧ Label t = l ∧ length i = Arity t ∧ apply_guard
(Guards t) (join_ir i r)) e =
                      ffilter (λ((origin, dest), t). Label t = l ∧ length i = Arity t ∧ apply_guards (Guards
t) (join_ir i r)) (ffilter (λ((origin, dest), t). origin = s) e)"
  by auto

lemma one_outgoing_transition:
  defines "outgoing s ≡ (λ((origin, dest), t). origin = s)"
  assumes prem: "size (ffilter (outgoing s) e) = 1"
  shows "size (possible_steps e s r l i) ≤ 1"
proof-
  have less_eq_1: "⋀x::nat. (x ≤ 1) = (x = 1 ∨ x = 0)"
    by auto
  have size_empty: "⋀f. (size f = 0) = (f = {||})"
    subgoal for f
      by (induct f, auto)
    done
  show ?thesis
    using prem
    apply (simp only: possible_steps_def)
    apply (rule fimage_size_le)
    apply (simp only: ffilter_split outgoing_def[symmetric])
    by (metis (no_types, lifting) size_ffilter)
qed
```

3.2.2 Choice

Here we define the `choice` operator which determines whether or not two transitions are nondeterministic.

```
definition choice :: "transition ⇒ transition ⇒ bool" where
  "choice t t' = (∃ i r. apply_guards (Guards t) (join_ir i r) ∧ apply_guards (Guards t') (join_ir i r))"

definition choice_alt :: "transition ⇒ transition ⇒ bool" where
  "choice_alt t t' = (∃ i r. apply_guards (Guards t@Guards t') (join_ir i r))"

lemma choice_alt: "choice t t' = choice_alt t t'"
```

```
  by (simp add: choice_def choice_alt_def apply_guards_append)

lemma choice_symmetry: "choice x y = choice y x"
  using choice_def by auto

definition deterministic :: "transition_matrix ⇒ bool" where
  "deterministic e = (∀s r l i. size (possible_steps e s r l i) ≤ 1)"

lemma deterministic_alt_aux: "size (possible_steps e s r l i) ≤ 1 =(
       possible_steps e s r l i = {|||} ∨
       (∃s' t.
           ffilter
            (λ((origin, dest), t). origin = s ∧ Label t = l ∧ length i = Arity t ∧ apply_guards (Guards
t) (join_ir i r)) e =
           {|((s, s'), t)|}))"
  apply (case_tac "size (possible_steps e s r l i) = 0")
   apply (simp add: fset_equiv)
  apply (case_tac "possible_steps e s r l i = {|||}")
   apply simp
  apply (simp only: possible_steps_alt[symmetric])
  by (metis le_neq_implies_less le_numeral_extra(4) less_one prod.collapse size_fsingleton)

lemma deterministic_alt: "deterministic e = (
  ∀s r l i.
    possible_steps e s r l i = {|||} ∨
    (∃s' t. ffilter (λ((origin, dest), t). origin = s ∧ (Label t) = l ∧ (length i) = (Arity t) ∧ apply_guards
(Guards t) (join_ir i r)) e = {|((s, s'), t)|})
)"
  using deterministic_alt_aux
  by (simp add: deterministic_def)

lemma size_le_1: "size f ≤ 1 = (f = {|||} ∨ (∃e. f = {|e|}))"
  apply standard
   apply (metis bot.not_eq_extremum gr_implies_not0 le_neq_implies_less less_one size_fsingleton size_fsubset)
  by auto

lemma ffilter_empty_if: "∀x |∈| xs. ¬ P x ⟹ ffilter P xs = {|||}"
  by auto

lemma empty_ffilter: "ffilter P xs = {|||} = (∀x |∈| xs. ¬ P x)"
  by auto

lemma all_states_deterministic:
"(∀s l i r.
  ffilter (λ((origin, dest), t). origin = s ∧ (Label t) = l ∧ can_take_transition t i r) e = {|||} ∨
  (∃x. ffilter (λ((origin, dest), t). origin = s ∧ (Label t) = l ∧ can_take_transition t i r) e = {|x|})
) ⟹ deterministic e"
  unfolding deterministic_def
  apply clarify
  subgoal for s r l i
    apply (erule_tac x=s in allE)
    apply (erule_tac x=l in allE)
    apply (erule_tac x=i in allE)
    apply (erule_tac x=r in allE)
    apply (simp only: size_le_1)
    apply (erule disjE)
     apply (rule_tac disjI1)
     apply (simp add: possible_steps_def can_take_transition_def can_take_def)
    apply (erule exE)
    subgoal for x
      apply (case_tac x)
      subgoal for a b
        apply (case_tac a)
```

```
        apply simp
        apply (induct e)
         apply auto[1]
        subgoal for _ _ _ ba
          apply (rule disjI2)
          apply (rule_tac x=ba in exI)
          apply (rule_tac x=b in exI)
          by (simp add: possible_steps_def can_take_transition_def[symmetric] can_take_def[symmetric])
        done
      done
    done
  done

lemma deterministic_finsert:
"∀i r l.
∀((a, b), t) |∈| ffilter (λ((origin, dest), t). origin = s) (finsert ((s, s'), t') e).
Label t = l ∧ can_take_transition t i r ⟶ ¬ can_take_transition t' i r ⟹
deterministic e ⟹
deterministic (finsert ((s, s'), t') e)"
  apply (simp add: deterministic_def possible_steps_finsert can_take del: size_fset_overloaded_simps)
  apply clarify
  subgoal for r i
    apply (erule_tac x=s in allE)
    apply (erule_tac x=r in allE)
    apply (erule_tac x="Label t'" in allE)
    apply (erule_tac x=i in allE)
    apply (erule_tac x=r in allE)
    apply (erule_tac x=i in allE)
    apply (erule_tac x="Label t'" in allE)
    by auto
  done

lemma ffilter_fBall: "(∀x |∈| xs. P x) = (ffilter P xs = xs)"
  by auto

lemma fsubset_if: "∀x. x |∈| f1 ⟶ x |∈| f2 ⟹ f1 |⊆| f2"
  by auto

lemma in_possible_steps: "(((s, s'), t)|∈|e ∧ Label t = l ∧ can_take_transition t i r) = ((s', t) |∈|
possible_steps e s r l i)"
  apply (simp add: fmember_possible_steps)
  by (simp add: can_take_def can_take_transition_def fmember.rep_eq)

lemma possible_steps_can_take_transition:
  "(s2, t1) |∈| possible_steps e1 s1 r l i ⟹ can_take_transition t1 i r"
  using in_possible_steps by blast

lemma not_deterministic:
  "∃s l i r.
    ∃d1 d2 t1 t2.
      d1 ≠ d2 ∧ t1 ≠ t2 ∧
      ((s, d1), t1) |∈| e ∧
      ((s, d2), t2) |∈| e ∧
      Label t1 = Label t2 ∧
      can_take_transition t1 i r ∧
      can_take_transition t2 i r ⟹
  ¬deterministic e"
  apply (simp add: deterministic_def not_le del: size_fset_overloaded_simps)
  apply clarify
  subgoal for s i r d1 d2 t1 t2
    apply (rule_tac x=s in exI)
    apply (rule_tac x=r in exI)
    apply (rule_tac x="Label t1" in exI)
```

```
    apply (rule_tac x=i in exI)
    apply (case_tac "(d1, t1) |∈| possible_steps e s r (Label t1) i")
     defer using in_possible_steps apply blast
    apply (case_tac "(d2, t2) |∈| possible_steps e s r (Label t1) i")
     apply (metis fempty_iff fsingleton_iff not_le_imp_less prod.inject size_le_1)
    using in_possible_steps by force
  done

lemma not_deterministic_conv:
  "¬deterministic e ⟹
  ∃s l i r.
    ∃d1 d2 t1 t2.
      (d1 ≠ d2 ∨ t1 ≠ t2) ∧
      ((s, d1), t1) |∈| e ∧
      ((s, d2), t2) |∈| e ∧
      Label t1 = Label t2 ∧
      can_take_transition t1 i r ∧
      can_take_transition t2 i r"
  apply (simp add: deterministic_def not_le del: size_fset_overloaded_simps)
  apply clarify
  subgoal for s r l i
    apply (case_tac "∃e1 e2 f'. e1 ≠ e2 ∧ possible_steps e s r l i = finsert e1 (finsert e2 f')")
     defer using size_gt_1 apply blast
    apply (erule exE)+
    subgoal for e1 e2 f'
      apply (case_tac e1, case_tac e2)
      subgoal for a b aa ba
        apply (simp del: size_fset_overloaded_simps)
        apply (rule_tac x=s in exI)
        apply (rule_tac x=i in exI)
        apply (rule_tac x=r in exI)
        apply (rule_tac x=a in exI)
        apply (rule_tac x=aa in exI)
        apply (rule_tac x=b in exI)
        apply (rule_tac x=ba in exI)
        by (metis finsertI1 finsert_commute in_possible_steps)
      done
    done
  done

lemma deterministic_if:
"∄s l i r.
  ∃d1 d2 t1 t2.
    (d1 ≠ d2 ∨ t1 ≠ t2) ∧
    ((s, d1), t1) |∈| e ∧
    ((s, d2), t2) |∈| e ∧
    Label t1 = Label t2 ∧
    can_take_transition t1 i r ∧
    can_take_transition t2 i r ⟹
  deterministic e"
  using not_deterministic_conv by blast

lemma "∀l i r.
  (∀((s, s'), t) |∈| e. Label t = l ∧ can_take_transition t i r ∧
  (∄t' s''. ((s, s''), t') |∈| e ∧ (s' ≠ s'' ∨ t' ≠ t) ∧ Label t' = l ∧ can_take_transition t' i r))
 ⟹ deterministic e"
  apply (simp add: deterministic_def del: size_fset_overloaded_simps)
  apply (rule allI)+
  apply (simp only: size_le_1 possible_steps_empty)
  apply (case_tac "∃t s'. ((s, s'), t)|∈|e ∧ Label t = l ∧ can_take_transition t i r")
   defer using notin_fset apply fastforce
  apply (rule disjI2)
  apply clarify
```

```
  apply (rule_tac x="(s', t)" in exI)
  apply standard
   defer apply (meson fempty_fsubsetI finsert_fsubset in_possible_steps)
  apply standard
  apply (case_tac x)
  apply (simp add: in_possible_steps[symmetric])
  apply (erule_tac x="Label t" in allE)
  apply (erule_tac x=i in allE)
  apply (erule_tac x=r in allE)
  apply (erule_tac x="((s, s'), t)" in fBallE)
   defer apply simp
  apply simp
  apply (erule_tac x=b in allE)
  apply simp
  apply (erule_tac x=a in allE)
  by simp

definition "outgoing_transitions e s = ffilter (λ((o, _), _). o = s) e"

lemma in_outgoing: "((s1, s2), t) |∈| outgoing_transitions e s = (((s1, s2), t) |∈| e ∧ s1 = s)"
  by (simp add: outgoing_transitions_def)

lemma outgoing_transitions_deterministic:
  "∀s.
    ∀((s1, s2), t) |∈| outgoing_transitions e s.
      ∀((s1', s2'), t') |∈| outgoing_transitions e s.
        s2 ≠ s2' ∨ t ≠ t' ⟶ Label t = Label t' ⟶ ¬ choice t t' ⟹ deterministic e"
  apply (rule deterministic_if)
  apply simp
  apply (rule allI)
  subgoal for s
    apply (erule_tac x=s in allE)
    apply (simp add: fBall_def Ball_def)
    apply (rule allI)+
    subgoal for i r d1 d2 t1
      apply (erule_tac x=s in allE)
      apply (erule_tac x=d1 in allE)
      apply (erule_tac x=t1 in allE)
      apply (rule impI, rule allI)
      subgoal for t2
        apply (case_tac "((s, d1), t1) ∈ fset (outgoing_transitions e s)")
         apply simp
         apply (erule_tac x=s in allE)
         apply (erule_tac x=d2 in allE)
         apply (erule_tac x=t2 in allE)
         apply (simp add: outgoing_transitions_def choice_def can_take)
         apply (meson fmember_implies_member)
        apply (simp add: outgoing_transitions_def)
        by (meson fmember_implies_member)
      done
    done
  done

lemma outgoing_transitions_deterministic2: "(⋀s a b ba aa bb bc.
       ((a, b), ba) |∈| outgoing_transitions e s ⟹
       ((aa, bb), bc) |∈| (outgoing_transitions e s) - {|((a, b), ba)|} ⟹ b ≠ bb ∨ ba ≠ bc ⟹ ¬choice
ba bc)
        ⟹ deterministic e"
  apply (rule outgoing_transitions_deterministic)
  by blast

lemma outgoing_transitions_fprod_deterministic:
"(⋀s b ba bb bc.
```

```
(((s, b), ba), ((s, bb), bc)) ∈ fset (outgoing_transitions e s) × fset (outgoing_transitions e s)
⟹ b ≠ bb ∨ ba ≠ bc ⟹ Label ba = Label bc ⟹ ¬choice ba bc)
⟹ deterministic e"
  apply (rule outgoing_transitions_deterministic)
  apply clarify
  by (metis SigmaI fmember_implies_member in_outgoing)
```

The `random_member` function returns a random member from a finite set, or `None`, if the set is empty.

```
definition random_member :: "'a fset ⇒ 'a option" where
  "random_member f = (if f = {||} then None else Some (Eps (λx. x |∈| f)))"

lemma random_member_nonempty: "s ≠ {||} = (random_member s ≠ None)"
  by (simp add: random_member_def)

lemma random_member_singleton [simp]: "random_member {|a|} = Some a"
  by (simp add: random_member_def)

lemma random_member_is_member:
  "random_member ss = Some s ⟹ s |∈| ss"
  apply (simp add: random_member_def)
  by (metis equalsffemptyI option.distinct(1) option.inject verit_sko_ex_indirect)

lemma random_member_None[simp]: "random_member ss = None = (ss = {||})"
  by (simp add: random_member_def)

lemma random_member_empty[simp]: "random_member {||} = None"
  by simp

definition step :: "transition_matrix ⇒ cfstate ⇒ registers ⇒ label ⇒ inputs ⇒ (transition × cfstate
× outputs × registers) option" where
  "step e s r l i = (case random_member (possible_steps e s r l i) of
      None ⇒ None |
      Some (s', t) ⇒  Some (t, s', evaluate_outputs t i r, evaluate_updates t i r)
  )"

lemma possible_steps_not_empty_iff:
  "step e s r a b ≠ None ⟹
   ∃aa ba. (aa, ba) |∈| possible_steps e s r a b"
  apply (simp add: step_def)
  apply (case_tac "possible_steps e s r a b")
   apply (simp add: random_member_def)
  by auto

lemma step_member: "step e s r l i = Some (t, s', p, r') ⟹ (s', t) |∈| possible_steps e s r l i"
  apply (simp add: step_def)
  apply (case_tac "random_member (possible_steps e s r l i)")
   apply simp
  subgoal for a by (case_tac a, simp add: random_member_is_member)
  done

lemma step_outputs: "step e s r l i = Some (t, s', p, r') ⟹ evaluate_outputs t i r = p"
  apply (simp add: step_def)
  apply (case_tac "random_member (possible_steps e s r l i)")
  by auto

lemma step:
  "possibilities = (possible_steps e s r l i) ⟹
   random_member possibilities = Some (s', t) ⟹
   evaluate_outputs t i r = p ⟹
   evaluate_updates t i r = r' ⟹
   step e s r l i = Some (t, s', p, r')"
  by (simp add: step_def)
```

```
lemma step_None: "step e s r l i = None = (possible_steps e s r l i = {||})"
  by (simp add: step_def prod.case_eq_if random_member_def)

lemma step_Some: "step e s r l i = Some (t, s', p, r') =
  (
    random_member (possible_steps e s r l i) = Some (s', t) ∧
    evaluate_outputs t i r = p ∧
    evaluate_updates t i r = r'
  )"
  apply (simp add: step_def)
  apply (case_tac "random_member (possible_steps e s r l i)")
   apply simp
  subgoal for a by (case_tac a, auto)
  done

lemma no_possible_steps_1:
  "possible_steps e s r l i = {||} ⟹ step e s r l i = None"
  by (simp add: step_def random_member_def)
```

3.2.3 Execution Observation

One of the key features of this formalisation of EFSMs is their ability to produce *outputs*, which represent function return values. When action sequences are executed in an EFSM, they produce a corresponding *observation*.

```
fun observe_execution :: "transition_matrix ⇒ cfstate ⇒ registers ⇒ execution ⇒ outputs list" where
  "observe_execution _ _ _ [] = []" |
  "observe_execution e s r ((l, i)#as)  = (
    let viable = possible_steps e s r l i in
    if viable = {||} then
      []
    else
      let (s', t) = Eps (λx. x |∈| viable) in
      (evaluate_outputs t i r)#(observe_execution e s' (evaluate_updates t i r) as)
    )"

lemma observe_execution_step_def: "observe_execution e s r ((l, i)#as)  = (
    case step e s r l i of
      None ⇒ []|
      Some (t, s', p, r') ⇒ p#(observe_execution e s' r' as)
    )"
  apply (simp add: step_def)
  apply (case_tac "possible_steps e s r l i")
   apply simp
  subgoal for x S'
    apply (simp add: random_member_def)
    apply (case_tac "SOME xa. xa = x ∨ xa |∈| S'")
    by simp
  done

lemma observe_execution_first_outputs_equiv:
  "observe_execution e1 s1 r1 ((l, i) # ts) = observe_execution e2 s2 r2 ((l, i) # ts) ⟹
   step e1 s1 r1 l i = Some (t, s', p, r') ⟹
   ∃(s2', t2)|∈|possible_steps e2 s2 r2 l i. evaluate_outputs t2 i r2 = p"
  apply (simp only: observe_execution_step_def)
  apply (case_tac "step e2 s2 r2 l i")
   apply simp
  subgoal for a
    apply simp
    apply (case_tac a)
    apply clarsimp
    by (meson step_member case_prodI rev_fBexI step_outputs)
  done
```

```
lemma observe_execution_step:
  "step e s r (fst h) (snd h) = Some (t, s', p, r') ⟹
   observe_execution e s' r' es = obs ⟹
   observe_execution e s r (h#es) = p#obs"
  apply (cases h, simp add: step_def)
  apply (case_tac "possible_steps e s r a b = {|||}")
   apply simp
  subgoal for a b
    apply (case_tac "SOME x. x |∈| possible_steps e s r a b")
    by (simp add: random_member_def)
  done

lemma observe_execution_possible_step:
  "possible_steps e s r (fst h) (snd h) = {|(s', t)|} ⟹
   apply_outputs (Outputs t) (join_ir (snd h) r) = p ⟹
   apply_updates (Updates t) (join_ir (snd h) r) r = r' ⟹
   observe_execution e s' r' es = obs ⟹
   observe_execution e s r (h#es) = p#obs"
  by (simp add: observe_execution_step step)

lemma observe_execution_no_possible_step:
  "possible_steps e s r (fst h) (snd h) = {||} ⟹
   observe_execution e s r (h#es) = []"
  by (cases h, simp)

lemma observe_execution_no_possible_steps:
  "possible_steps e1 s1 r1 (fst h) (snd h) = {||} ⟹
   possible_steps e2 s2 r2 (fst h) (snd h) = {||} ⟹
   (observe_execution e1 s1 r1 (h#t)) = (observe_execution e2 s2 r2 (h#t))"
  by (simp add: observe_execution_no_possible_step)

lemma observe_execution_one_possible_step:
  "possible_steps e1 s1 r (fst h) (snd h) = {|(s1', t1)|} ⟹
   possible_steps e2 s2 r (fst h) (snd h) = {|(s2', t2)|} ⟹
   apply_outputs (Outputs t1) (join_ir (snd h) r) = apply_outputs (Outputs t2) (join_ir (snd h) r) ⟹

   apply_updates (Updates t1) (join_ir (snd h) r) r = r' ⟹
   apply_updates (Updates t2) (join_ir (snd h) r) r = r' ⟹
   (observe_execution e1 s1' r' t) = (observe_execution e2 s2' r' t) ⟹
   (observe_execution e1 s1 r (h#t)) = (observe_execution e2 s2 r (h#t))"
  by (simp add: observe_execution_possible_step)
```

Utilities

Here we define some utility functions to access the various key properties of a given EFSM.

```
definition max_reg :: "transition_matrix ⇒ nat option" where
  "max_reg e = (let maxes = (fimage (λ(_, t). Transition.max_reg t) e) in if maxes = {||} then None else
fMax maxes)"

definition enumerate_ints :: "transition_matrix ⇒ int set" where
  "enumerate_ints e = ⋃ (image (λ(_, t). Transition.enumerate_ints t) (fset e))"

definition max_int :: "transition_matrix ⇒ int" where
  "max_int e = Max (insert 0 (enumerate_ints e))"

definition max_output :: "transition_matrix ⇒ nat" where
  "max_output e = fMax (fimage (λ(_, t). length (Outputs t)) e)"

definition all_regs :: "transition_matrix ⇒ nat set" where
  "all_regs e = ⋃ (image (λ(_, t). enumerate_regs t) (fset e))"
lemma finite_all_regs: "finite (all_regs e)"
```

```
  apply (simp add: all_regs_def enumerate_regs_def)
  apply clarify
  apply standard
   apply (rule finite_UnI)+
  using GExp.finite_enumerate_regs apply blast
  using AExp.finite_enumerate_regs apply blast
   apply (simp add: AExp.finite_enumerate_regs prod.case_eq_if)
  by auto

definition max_input :: "transition_matrix ⇒ nat option" where
  "max_input e = fMax (fimage (λ(_, t). Transition.max_input t) e)"

fun maxS :: "transition_matrix ⇒ nat" where
  "maxS t = (if t = {|||} then 0 else fMax ((fimage (λ((origin, dest), t). origin) t) |∪| (fimage (λ((origin,
dest), t). dest) t)))"
```

3.2.4 Execution Recognition

The `recognises` function returns true if the given EFSM recognises a given execution. That is, the EFSM is able to respond to each event in sequence. There is no restriction on the outputs produced. When a recognised execution is observed, it produces an accepted trace of the EFSM.

```
inductive recognises_execution :: "transition_matrix ⇒ nat ⇒ registers ⇒ execution ⇒ bool" where
  base [simp]: "recognises_execution e s r []" |
  step: "∃(s', T) |∈| possible_steps e s r l i.
         recognises_execution e s' (evaluate_updates T i r) t ⟹
         recognises_execution e s r ((l, i)#t)"

abbreviation "recognises e t ≡ recognises_execution e 0 <> t"

definition "E e = {x. recognises e x}"

lemma no_possible_steps_rejects:
  "possible_steps e s r l i = {|||} ⟹ ¬ recognises_execution e s r ((l, i)#t)"
  apply clarify
  by (rule recognises_execution.cases, auto)

lemma recognises_step_equiv: "recognises_execution e s r ((l, i)#t) =
   (∃(s', T) |∈| possible_steps e s r l i. recognises_execution e s' (evaluate_updates T i r) t)"
  apply standard
   apply (rule recognises_execution.cases)
  by (auto simp: recognises_execution.step)

fun recognises_prim :: "transition_matrix ⇒ nat ⇒ registers ⇒ execution ⇒ bool" where
  "recognises_prim e s r [] = True" |
  "recognises_prim e s r ((l, i)#t) = (
    let poss_steps = possible_steps e s r l i in
    (∃(s', T) |∈| poss_steps. recognises_prim e s' (evaluate_updates T i r) t)
  )"

lemma recognises_prim [code]: "recognises_execution e s r t = recognises_prim e s r t"
proof(induct t arbitrary: r s)
  case Nil
  then show ?case
    by simp
next
  case (Cons h t)
  then show ?case
    apply (cases h)
    apply simp
    apply standard
     apply (rule recognises_execution.cases, simp)
```

```
      apply simp
     apply auto[1]
    using recognises_execution.step by blast
qed

lemma recognises_single_possible_step:
  assumes "possible_steps e s r l i = {|(s', t)|}"
      and "recognises_execution e s' (evaluate_updates t i r) trace"
    shows "recognises_execution e s r ((l, i)#trace)"
  apply (rule recognises_execution.step)
  using assms by auto

lemma recognises_single_possible_step_atomic:
  assumes "possible_steps e s r (fst h) (snd h) = {|(s', t)|}"
      and "recognises_execution e s' (apply_updates (Updates t) (join_ir (snd h) r) r) trace"
    shows "recognises_execution e s r (h#trace)"
  by (metis assms prod.collapse recognises_single_possible_step)

lemma recognises_must_be_possible_step:
  "recognises_execution e s r (h # t) ⟹
   ∃aa ba. (aa, ba) |∈| possible_steps e s r (fst h) (snd h)"
  using recognises_step_equiv by fastforce

lemma recognises_possible_steps_not_empty:
  "recognises_execution e s r (h#t) ⟹ possible_steps e s r (fst h) (snd h) ≠ {||}"
  apply (rule recognises_execution.cases)
  by auto

lemma recognises_must_be_step:
  "recognises_execution e s r (h#ts) ⟹
   ∃t s' p d'. step e s r (fst h) (snd h) = Some (t, s', p, d')"
  apply (cases h)
  subgoal for a b
    apply (simp add: recognises_step_equiv step_def)
    apply clarify
    apply (case_tac "(possible_steps e s r a b)")
     apply (simp add: random_member_def)
    apply (simp add: random_member_def)
    subgoal for _ _ x S' apply (case_tac "SOME xa. xa = x ∨ xa |∈| S'")
      by simp
    done
  done

lemma recognises_cons_step:
  "recognises_execution e s r (h # t) ⟹ step e s r (fst h) (snd h) ≠  None"
  by (simp add: recognises_must_be_step)

lemma no_step_none:
  "step e s r aa ba = None ⟹ ¬ recognises_execution e s r ((aa, ba) # p)"
  using recognises_cons_step by fastforce

lemma step_none_rejects:
  "step e s r (fst h) (snd h) = None ⟹ ¬ recognises_execution e s r (h#t)"
  using no_step_none surjective_pairing by fastforce

lemma trace_reject:
  "(¬ recognises_execution e s r ((l, i)#t)) = (possible_steps e s r l i = {||} ∨ (∀(s', T) |∈| possible_steps
e s r l i. ¬ recognises_execution e s' (evaluate_updates T i r) t))"
  using recognises_prim by fastforce

lemma trace_reject_no_possible_steps_atomic:
  "possible_steps e s r (fst a) (snd a) = {||} ⟹ ¬ recognises_execution e s r (a#t)"
  using recognises_possible_steps_not_empty by auto
```

```
lemma trace_reject_later:
  "∀(s', T) |∈| possible_steps e s r l i. ¬ recognises_execution e s' (evaluate_updates T i r) t ⟹
   ¬ recognises_execution e s r ((l, i)#t)"
  using trace_reject by auto

lemma recognition_prefix_closure: "recognises_execution e s r (t@t') ⟹ recognises_execution e s r t"
proof(induct t arbitrary: s r)
  case (Cons a t)
  then show ?case
    apply (cases a, clarsimp)
    apply (rule recognises_execution.cases)
      apply simp
     apply simp
    by (rule recognises_execution.step, auto)
qed auto

lemma rejects_prefix: "¬ recognises_execution e s r t ⟹ ¬ recognises_execution e s r (t @ t')"
  using recognition_prefix_closure by blast

lemma recognises_head: "recognises_execution e s r (h#t) ⟹ recognises_execution e s r [h]"
  by (simp add: recognition_prefix_closure)
```

Trace Acceptance

The `accepts` function returns true if the given EFSM accepts a given trace. That is, the EFSM is able to respond to each event in sequence *and* is able to produce the expected output. Accepted traces represent valid runs of an EFSM.

```
inductive accepts_trace :: "transition_matrix ⇒ cfstate ⇒ registers ⇒ trace ⇒ bool" where
  base [simp]: "accepts_trace e s r []" |
  step: "∃(s', T) |∈| possible_steps e s r l i.
         evaluate_outputs T i r = map Some p ∧ accepts_trace e s' (evaluate_updates T i r) t ⟹
         accepts_trace e s r ((l, i, p)#t)"


definition T :: "transition_matrix ⇒ trace set" where
  "T e = {t. accepts_trace e 0 <> t}"

abbreviation "rejects_trace e s r t ≡ ¬ accepts_trace e s r t"

lemma accepts_trace_step:
  "accepts_trace e s r ((l, i, p)#t) = (∃(s', T) |∈| possible_steps e s r l i.
         evaluate_outputs T i r = map Some p ∧
         accepts_trace e s' (evaluate_updates T i r) t)"
  apply standard
  by (rule accepts_trace.cases, auto simp: accepts_trace.step)

lemma accepts_trace_exists_possible_step:
  "accepts_trace e1 s1 r1 ((aa, b, c) # t) ⟹
       ∃(s1', t1)|∈|possible_steps e1 s1 r1 aa b.
          evaluate_outputs t1 b r1 = map Some c"
  using accepts_trace_step by auto

lemma rejects_trace_step:
"rejects_trace e s r ((l, i, p)#t) = (
  (∀(s', T) |∈| possible_steps e s r l i.  evaluate_outputs T i r ≠ map Some p ∨ rejects_trace e s' (evaluate_upd
T i r) t)
)"
  apply (simp add: accepts_trace_step)
  by auto
```

```
definition accepts_log :: "trace set ⇒ transition_matrix ⇒ bool" where
  "accepts_log l e = (∀t ∈ l. accepts_trace e 0 <> t)"
lemma prefix_closure: "accepts_trace e s r (t@t') ⟹ accepts_trace e s r t"

proof(induct t arbitrary: s r)
next
  case (Cons a t)
  then show ?case
    apply (cases a, clarsimp)
    apply (simp add: accepts_trace_step)
    by auto
qed auto
```

For code generation, it is much more efficient to re-implement the `accepts_trace` function primitively than to use the code generator's default setup for inductive definitions.

```
fun accepts_trace_prim :: "transition_matrix ⇒ cfstate ⇒ registers ⇒ trace ⇒ bool" where
  "accepts_trace_prim _ _ _ [] = True" |
  "accepts_trace_prim e s r ((l, i, p)#t) = (
    let poss_steps = possible_steps e s r l i in
    if fis_singleton poss_steps then
      let (s', T) = fthe_elem poss_steps in
      if evaluate_outputs T i r = map Some p then
        accepts_trace_prim e s' (evaluate_updates T i r) t
      else False
    else
      (∃(s', T) |∈| poss_steps.
         evaluate_outputs T i r = (map Some p) ∧
         accepts_trace_prim e s' (evaluate_updates T i r) t))"

lemma accepts_trace_prim [code]: "accepts_trace e s r l = accepts_trace_prim e s r l"
proof(induct l arbitrary: s r)
  case (Cons a l)
  then show ?case
    apply (cases a)
    apply (simp add: accepts_trace_step Let_def fis_singleton_alt)
    by auto
qed auto
```

3.2.5 EFSM Comparison

Here, we define some different metrics of EFSM equality.

State Isomporphism

Two EFSMs are isomorphic with respect to states if there exists a bijective function between the state names of the two EFSMs, i.e. the only difference between the two models is the way the states are indexed.

```
definition isomorphic :: "transition_matrix ⇒ transition_matrix ⇒ bool" where
  "isomorphic e1 e2 = (∃f. bij f ∧ (∀((s1, s2), t) |∈| e1. ((f s1, f s2), t) |∈| e2))"
```

Register Isomporphism

Two EFSMs are isomorphic with respect to registers if there exists a bijective function between the indices of the registers in the two EFSMs, i.e. the only difference between the two models is the way the registers are indexed.

```
definition rename_regs :: "(nat ⇒ nat) ⇒ transition_matrix ⇒ transition_matrix" where
  "rename_regs f e = fimage (λ(tf, t). (tf, Transition.rename_regs f t)) e"

definition eq_upto_rename_strong :: "transition_matrix ⇒ transition_matrix ⇒ bool" where
  "eq_upto_rename_strong e1 e2 = (∃f. bij f ∧ rename_regs f e1 = e2)"
```

Trace Simulation

An EFSM, e_1 simulates another EFSM e_2 if there is a function between the states of the states of e_1 and e_1 such that in each state, if e_1 can respond to the event and produce the correct output, so can e_2.

```
inductive trace_simulation :: "(cfstate ⇒ cfstate) ⇒ transition_matrix ⇒ cfstate ⇒ registers ⇒
transition_matrix ⇒ cfstate ⇒ registers ⇒ trace ⇒ bool" where
  base: "s2 = f s1 ⟹ trace_simulation f e1 s1 r1 e2 s2 r2 []" |
  step: "s2 = f s1 ⟹
        ∀(s1', t1) |∈| ffilter (λ(s1', t1). evaluate_outputs t1 i r1 = map Some o) (possible_steps e1
s1 r1 l i).
          ∃(s2', t2) |∈| possible_steps e2 s2 r2 l i. evaluate_outputs t2 i r2 = map Some o ∧
           trace_simulation f e1 s1' (evaluate_updates t1 i r1) e2 s2' (evaluate_updates t2 i r2) es ⟹
        trace_simulation f e1 s1 r1 e2 s2 r2 ((l, i, o)#es)"

lemma trace_simulation_step:
"trace_simulation f e1 s1 r1 e2 s2 r2 ((l, i, o)#es) = (
  (s2 = f s1) ∧ (∀(s1', t1) |∈| ffilter (λ(s1', t1). evaluate_outputs t1 i r1 = map Some o) (possible_steps
e1 s1 r1 l i).
         (∃(s2', t2) |∈| possible_steps e2 s2 r2 l i. evaluate_outputs t2 i r2 = map Some o ∧
         trace_simulation f e1 s1' (evaluate_updates t1 i r1) e2 s2' (evaluate_updates t2 i r2) es))
)"
  apply standard
   apply (rule trace_simulation.cases, simp+)
  apply (rule trace_simulation.step)
   apply simp
  by blast

lemma trace_simulation_step_none:
  "s2 = f s1 ⟹
   ∄(s1', t1) |∈| possible_steps e1 s1 r1 l i. evaluate_outputs t1 i r1 = map Some o ⟹
   trace_simulation f e1 s1 r1 e2 s2 r2 ((l, i, o)#es)"
  apply (rule trace_simulation.step)
   apply simp
  apply (case_tac "ffilter (λ(s1', t1). evaluate_outputs t1 i r1 = map Some o) (possible_steps e1 s1 r1
l i)")
   apply simp
  by fastforce

definition "trace_simulates e1 e2 = (∃f. ∀t. trace_simulation f e1 0 <> e2 0 <> t)"

lemma rejects_trace_simulation:
  "rejects_trace e2 s2 r2 t ⟹
   accepts_trace e1 s1 r1 t ⟹
   ¬trace_simulation f e1 s1 r1 e2 s2 r2 t"
proof(induct t arbitrary: s1 r1 s2 r2)
  case Nil
  then show ?case
    using accepts_trace.base by blast
next
  case (Cons a t)
  then show ?case
    apply (cases a)
    apply (simp add: rejects_trace_step)
    apply (simp add: accepts_trace_step)
    apply clarify
    apply (rule trace_simulation.cases)
      apply simp
     apply simp
    apply clarsimp
    subgoal for l o _ _ i
      apply (case_tac "ffilter (λ(s1', t1). evaluate_outputs t1 i r1 = map Some o) (possible_steps e1 s1
r1 l i) = {|}")
```

```
      apply auto[1]
     by fastforce
   done
qed

lemma accepts_trace_simulation:
  "accepts_trace e1 s1 r1 t ⟹
   trace_simulation f e1 s1 r1 e2 s2 r2 t ⟹
   accepts_trace e2 s2 r2 t"
  using rejects_trace_simulation by blast

lemma simulates_trace_subset: "trace_simulates e1 e2 ⟹ T e1 ⊆ T e2"
  using T_def accepts_trace_simulation trace_simulates_def by fastforce
```

Trace Equivalence

Two EFSMs are trace equivalent if they accept the same traces. This is the intuitive definition of "observable equivalence" between the behaviours of the two models. If two EFSMs are trace equivalent, there is no trace which can distinguish the two.

```
definition "trace_equivalent e1 e2 = (T e1 = T e2)"


lemma simulation_implies_trace_equivalent:
  "trace_simulates e1 e2 ⟹ trace_simulates e2 e1 ⟹ trace_equivalent e1 e2"

  using simulates_trace_subset trace_equivalent_def by auto

lemma trace_equivalent_reflexive: "trace_equivalent e1 e1"
  by (simp add: trace_equivalent_def)

lemma trace_equivalent_symmetric:
  "trace_equivalent e1 e2 = trace_equivalent e2 e1"
  using trace_equivalent_def by auto

lemma trace_equivalent_transitive:
  "trace_equivalent e1 e2 ⟹
   trace_equivalent e2 e3 ⟹
   trace_equivalent e1 e3"
  by (simp add: trace_equivalent_def)
```

Two EFSMs are trace equivalent if they accept the same traces.

```
lemma trace_equivalent:
  "∀t. accepts_trace e1 0 <> t = accepts_trace e2 0 <> t ⟹ trace_equivalent e1 e2"
  by (simp add: T_def trace_equivalent_def)

lemma accepts_trace_step_2: "(s2', t2) |∈| possible_steps e2 s2 r2 l i ⟹
      accepts_trace e2 s2' (evaluate_updates t2 i r2) t ⟹
      evaluate_outputs t2 i r2 = map Some p ⟹
      accepts_trace e2 s2 r2 ((l, i, p)#t)"
  by (rule accepts_trace.step, auto)
```

Execution Simulation

Execution simulation is similar to trace simulation but for executions rather than traces. Execution simulation has no notion of "expected" output. It simply requires that the simulating EFSM must be able to produce equivalent output for each action.

```
inductive execution_simulation :: "(cfstate ⇒ cfstate) ⇒ transition_matrix ⇒ cfstate ⇒
registers ⇒ transition_matrix ⇒ cfstate ⇒ registers ⇒ execution ⇒ bool" where
  base: "s2 = f s1 ⟹ execution_simulation f e1 s1 r1 e2 s2 r2 []" |
  step: "s2 = f s1 ⟹
```

```
        ∀(s1', t1) |∈| (possible_steps e1 s1 r1 l i).
          ∃(s2', t2) |∈| possible_steps e2 s2 r2 l i.
           evaluate_outputs t1 i r1 = evaluate_outputs t2 i r2 ∧
           execution_simulation f e1 s1' (evaluate_updates t1 i r1) e2 s2' (evaluate_updates t2 i r2) es
⟹
        execution_simulation f e1 s1 r1 e2 s2 r2 ((l, i)#es)"

definition "execution_simulates e1 e2 = (∃f. ∀t. execution_simulation f e1 0 <> e2 0 <> t)"

lemma execution_simulation_step:
"execution_simulation f e1 s1 r1 e2 s2 r2 ((l, i)#es) =
 (s2 = f s1 ∧
 (∀(s1', t1) |∈| (possible_steps e1 s1 r1 l i).
        (∃(s2', t2) |∈| possible_steps e2 s2 r2 l i. evaluate_outputs t1 i r1 = evaluate_outputs t2 i r2
∧
        execution_simulation f e1 s1' (evaluate_updates t1 i r1) e2 s2' (evaluate_updates t2 i r2) es))
)"
  apply standard
   apply (rule execution_simulation.cases)
     apply simp
    apply simp
   apply simp
  apply (rule execution_simulation.step)
   apply simp
  by blast
lemma execution_simulation_trace_simulation:
  "execution_simulation f e1 s1 r1 e2 s2 r2 (map (λ(l, i, o). (l, i)) t) ⟹
   trace_simulation f e1 s1 r1 e2 s2 r2 t"

proof(induct t arbitrary: s1 s2 r1 r2)
case Nil
  then show ?case
    apply (rule execution_simulation.cases)
     apply (simp add: trace_simulation.base)
    by simp
next
  case (Cons a t)
  then show ?case
    apply (cases a, clarsimp)
    apply (rule execution_simulation.cases)
      apply simp
     apply simp
    apply (rule trace_simulation.step)
     apply simp
    apply clarsimp
    subgoal for _ _ _ aa ba
      apply (erule_tac x="(aa, ba)" in fBallE)
       apply clarsimp
       apply blast
      by simp
    done
qed

lemma execution_simulates_trace_simulates:
  "execution_simulates e1 e2 ⟹ trace_simulates e1 e2"
  apply (simp add: execution_simulates_def trace_simulates_def)
  using execution_simulation_trace_simulation by blast
```

Executional Equivalence

Two EFSMs are executionally equivalent if there is no execution which can distinguish between the two. That is, for every execution, they must produce equivalent outputs.

```
inductive executionally_equivalent :: "transition_matrix ⇒ cfstate ⇒ registers ⇒
transition_matrix ⇒ cfstate ⇒ registers ⇒ execution ⇒ bool" where
  base [simp]: "executionally_equivalent e1 s1 r1 e2 s2 r2 []" |
  step: "∀(s1', t1) |∈| possible_steps e1 s1 r1 l i.
           ∃(s2', t2) |∈| possible_steps e2 s2 r2 l i.
             evaluate_outputs t1 i r1 = evaluate_outputs t2 i r2 ∧
             executionally_equivalent e1 s1' (evaluate_updates t1 i r1) e2 s2' (evaluate_updates t2 i r2)
es ⟹
         ∀(s2', t2) |∈| possible_steps e2 s2 r2 l i.
           ∃(s1', t1) |∈| possible_steps e1 s1 r1 l i.
             evaluate_outputs t1 i r1 = evaluate_outputs t2 i r2 ∧
             executionally_equivalent e1 s1' (evaluate_updates t1 i r1) e2 s2' (evaluate_updates t2 i r2)
es ⟹
         executionally_equivalent e1 s1 r1 e2 s2 r2 ((l, i)#es)"

lemma executionally_equivalent_step:
"executionally_equivalent e1 s1 r1 e2 s2 r2 ((l, i)#es) = (
  (∀(s1', t1) |∈| (possible_steps e1 s1 r1 l i). (∃(s2', t2) |∈| possible_steps e2 s2 r2 l i. evaluate_outputs
t1 i r1 = evaluate_outputs t2 i r2 ∧
   executionally_equivalent e1 s1' (evaluate_updates t1 i r1) e2 s2' (evaluate_updates t2 i r2) es)) ∧
  (∀(s2', t2) |∈| (possible_steps e2 s2 r2 l i). (∃(s1', t1) |∈| possible_steps e1 s1 r1 l i. evaluate_outputs
t1 i r1 = evaluate_outputs t2 i r2 ∧
   executionally_equivalent e1 s1' (evaluate_updates t1 i r1) e2 s2' (evaluate_updates t2 i r2) es)))"
  apply standard
   apply (rule executionally_equivalent.cases)
     apply simp
    apply simp
   apply simp
  by (rule executionally_equivalent.step, auto)

lemma execution_end:
  "possible_steps e1 s1 r1 l i = {|||} ⟹
   possible_steps e2 s2 r2 l i = {|||} ⟹
  executionally_equivalent e1 s1 r1 e2 s2 r2 ((l, i)#es)"
  by (simp add: executionally_equivalent_step)

lemma possible_steps_disparity:
  "possible_steps e1 s1 r1 l i ≠ {|||} ⟹
   possible_steps e2 s2 r2 l i = {|||} ⟹
   ¬executionally_equivalent e1 s1 r1 e2 s2 r2 ((l, i)#es)"
  by (simp add: executionally_equivalent_step, auto)

lemma executionally_equivalent_acceptance_map:
  "executionally_equivalent e1 s1 r1 e2 s2 r2 (map (λ(l, i, o). (l, i)) t) ⟹
   accepts_trace e2 s2 r2 t = accepts_trace e1 s1 r1 t"
proof(induct t arbitrary: s1 s2 r1 r2)
  case (Cons a t)
  then show ?case
    apply (cases a, simp)
    apply (rule executionally_equivalent.cases)
      apply simp
     apply simp
    apply clarsimp
    apply standard
    subgoal for i p l
      apply (rule accepts_trace.cases)
        apply simp
       apply simp
      apply clarsimp
      subgoal for aa b
        apply (rule accepts_trace.step)
        apply (erule_tac x="(aa, b)" in fBallE[of "possible_steps e2 s2 r2 l i"])
```

```
        defer apply simp
       apply simp
       by blast
    done
  apply (rule accepts_trace.cases)
    apply simp
   apply simp
  apply clarsimp
  subgoal for _ _ _ aa b
    apply (rule accepts_trace.step)
    apply (erule_tac x="(aa, b)" in fBallE)
     defer apply simp
    apply simp
    by fastforce
  done
qed auto

lemma executionally_equivalent_acceptance:
  "∀x. executionally_equivalent e1 s1 r1 e2 s2 r2 x ⟹ accepts_trace e1 s1 r1  t ⟹ accepts_trace e2
s2 r2 t"
  using executionally_equivalent_acceptance_map by blast

lemma executionally_equivalent_trace_equivalent:
  "∀x. executionally_equivalent e1 0 <> e2 0 <> x ⟹ trace_equivalent e1 e2"
  apply (rule trace_equivalent)
  apply clarify
  subgoal for t apply (erule_tac x="map (λ(l, i, o). (l, i)) t" in allE)
    by (simp add: executionally_equivalent_acceptance_map)
  done

lemma executionally_equivalent_symmetry:
  "executionally_equivalent e1 s1 r1 e2 s2 r2 x ⟹
   executionally_equivalent e2 s2 r2 e1 s1 r1 x"
proof(induct x arbitrary: s1 s2 r1 r2)
  case (Cons a x)
  then show ?case
    apply (cases a, clarsimp)
    apply (simp add: executionally_equivalent_step)
    apply standard
     apply (rule fBallI)
     apply clarsimp
    subgoal for aa b aaa ba
      apply (erule_tac x="(aaa, ba)" in fBallE[of "possible_steps e2 s2 r2 aa b"])
      by (force, simp)
    apply (rule fBallI)
    apply clarsimp
    subgoal for aa b aaa ba
      apply (erule_tac x="(aaa, ba)" in fBallE)
      by (force, simp)
    done
qed auto

lemma executionally_equivalent_transitivity:
  "executionally_equivalent e1 s1 r1 e2 s2 r2 x ⟹
   executionally_equivalent e2 s2 r2 e3 s3 r3 x ⟹
   executionally_equivalent e1 s1 r1 e3 s3 r3 x"
proof(induct x arbitrary: s1 s2 s3 r1 r2 r3)
  case (Cons a x)
  then show ?case
    apply (cases a, clarsimp)
    apply (simp add: executionally_equivalent_step)
    apply clarsimp
    apply standard
```

```
     apply (rule fBallI)
     apply clarsimp
    subgoal for aa b ab ba
      apply (erule_tac x="(ab, ba)" in fBallE[of "possible_steps e1 s1 r1 aa b"])
       prefer 2 apply simp
      apply simp
      apply (erule fBexE)
      subgoal for x apply (case_tac x)
        apply simp
        by blast
      done
    apply (rule fBallI)
    apply clarsimp
    subgoal for aa b ab ba
      apply (erule_tac x="(ab, ba)" in fBallE[of "possible_steps e3 s3 r3 aa b"])
       prefer 2 apply simp
      apply simp
      apply (erule fBexE)
      subgoal for x apply (case_tac x)
        apply clarsimp
        subgoal for aaa baa
          apply (erule_tac x="(aaa, baa)" in fBallE[of "possible_steps e2 s2 r2 aa b"])
           prefer 2 apply simp
          apply simp
          by blast
        done
      done
    done
qed auto
```

3.2.6 Reachability

Here, we define the function `visits` which returns true if the given execution leaves the given EFSM in the given state.

```
inductive visits :: "cfstate ⇒ transition_matrix ⇒ cfstate ⇒ registers ⇒ execution ⇒ bool" where
  base [simp]: "visits s e s r []" |
  step: "∃(s', T) |∈| possible_steps e s r l i. visits target e s' (evaluate_updates T i r) t ⟹
         visits target e s r ((l, i)#t)"

definition "reachable s e = (∃t. visits s e 0 <> t)"

lemma no_further_steps:
  "s ≠ s' ⟹ ¬ visits s e s' r []"
  apply safe
  apply (rule visits.cases)
  by auto

lemma visits_base: "visits target e s r [] = (s = target)"
  by (metis visits.base no_further_steps)

lemma visits_step:
  "visits target e s r (h#t) = (∃(s', T) |∈| possible_steps e s r (fst h) (snd h). visits target e s' (evaluate_up
T (snd h) r) t)"
  apply standard
   apply (rule visits.cases)
     apply simp+
  apply (cases h)
  using visits.step by auto

lemma reachable_initial: "reachable 0 e"
  apply (simp add: reachable_def)
```

```
  apply (rule_tac x="[]" in exI)
  by simp

lemma visits_finsert:
  "visits s e s' r t ⟹ visits s (finsert ((aa, ba), b) e) s' r t"
proof(induct t arbitrary: s' r)
  case Nil
  then show ?case
    by (simp add: visits_base)
next
  case (Cons a t)
  then show ?case
    apply (simp add: visits_step)
    apply (erule fBexE)
    apply (rule_tac x=x in fBexI)
     apply auto[1]
    by (simp add: possible_steps_finsert)
qed

lemma reachable_finsert:
  "reachable s e ⟹ reachable s (finsert ((aa, ba), b) e)"
  apply (simp add: reachable_def)
  by (meson visits_finsert)

lemma reachable_finsert_contra:
  "¬ reachable s (finsert ((aa, ba), b) e) ⟹ ¬reachable s e"
  using reachable_finsert by blast

lemma visits_empty: "visits s e s' r [] = (s = s')"
  apply standard
  by (rule visits.cases, auto)

definition "remove_state s e = ffilter (λ((from, to), t). from ≠ s ∧ to ≠ s) e"
inductive "obtains" :: "cfstate ⇒ registers ⇒ transition_matrix ⇒ cfstate ⇒ registers ⇒ execution ⇒
bool" where
  base [simp]: "obtains s r e s r []" |
  step: "∃(s'', T) |∈| possible_steps e s' r' l i. obtains s r e s'' (evaluate_updates T i r') t ⟹
         obtains s r e s' r' ((l, i)#t)"

definition "obtainable s r e = (∃t. obtains s r e 0 <> t)"

lemma obtains_obtainable:
  "obtains s r e 0 <> t ⟹ obtainable s r e"
  apply (simp add: obtainable_def)
  by auto

lemma obtains_base: "obtains s r e s' r' [] = (s = s' ∧ r = r')"
  apply standard
  by (rule obtains.cases, auto)

lemma obtains_step: "obtains s r e s' r' ((l, i)#t) = (∃(s'', T) |∈| possible_steps e s' r' l i. obtains
s r e s'' (evaluate_updates T i r') t)"
  apply standard
  by (rule obtains.cases, auto simp add: obtains.step)

lemma obtains_recognises:
  "obtains s c e s' r t ⟹ recognises_execution e s' r t"
proof(induct t arbitrary: s' r)
  case Nil
  then show ?case
    by (simp add: obtains_base)
next
  case (Cons a t)
```

```
  then show ?case
    apply (cases a)
    apply simp
    apply (rule obtains.cases)
      apply simp
     apply simp
    apply clarsimp
    using recognises_execution.step by fastforce
qed

lemma ex_comm4:
  "(∃ c1 s a b. (a, b) ∈ fset (possible_steps e s' r l i) ∧ obtains s c1 e a (evaluate_updates b i r) t)
=
   (∃ a b s c1. (a, b) ∈ fset (possible_steps e s' r l i) ∧ obtains s c1 e a (evaluate_updates b i r) t)"
  by auto

lemma recognises_execution_obtains:
  "recognises_execution e s' r t ⟹ ∃ c1 s. obtains s c1 e s' r t"
proof(induct t arbitrary: s' r)
  case Nil
  then show ?case
    by (simp add: obtains_base)
next
  case (Cons a t)
  then show ?case
    apply (cases a)
    apply (simp add: obtains_step)
    apply (rule recognises_execution.cases)
      apply simp
     apply simp
    apply clarsimp
    apply (simp add: fBex_def Bex_def ex_comm4)
    subgoal for _ _ aa ba
      apply (rule_tac x=aa in exI)
      apply (rule_tac x=ba in exI)
      apply (simp add: fmember_implies_member)
      by blast
    done
qed

lemma obtainable_empty_efsm:
  "obtainable s c {||} = (s=0 ∧ c = <>)"
  apply (simp add: obtainable_def)
  apply standard
  apply (metis ffilter_empty no_outgoing_transitions no_step_none obtains.cases obtains_recognises step_None)
  using obtains_base by blast

lemma obtains_visits: "obtains s r e s' r' t ⟹ visits s e s' r' t"
proof(induct t arbitrary: s' r')
  case Nil
  then show ?case
    by (simp add: obtains_base)
next
  case (Cons a t)
  then show ?case
    apply (cases a)
    apply (rule obtains.cases)
      apply simp
     apply simp
    apply clarsimp
    apply (rule visits.step)
    by auto
qed
```

```
lemma unobtainable_if: "¬ visits s e s' r' t ⟹ ¬ obtains s r e s' r' t"
  using obtains_visits by blast

lemma obtainable_if_unreachable: "¬reachable s e ⟹ ¬obtainable s r e"
  by (simp add: reachable_def obtainable_def unobtainable_if)

lemma obtains_step_append:
  "obtains s r e s' r' t ⟹
  (s'', ta) |∈| possible_steps e s r l i ⟹
  obtains s'' (evaluate_updates ta i r) e s' r' (t @ [(l, i)])"
proof(induct t arbitrary: s' r')
  case Nil
  then show ?case
    apply (simp add: obtains_base)
    apply (rule obtains.step)
    apply (rule_tac x="(s'', ta)" in fBexI)
    by auto
next
  case (Cons a t)
  then show ?case
    apply simp
    apply (rule obtains.cases)
      apply simp
     apply simp
    apply clarsimp
    apply (rule obtains.step)
    by auto
qed

lemma reachable_if_obtainable_step:
  "obtainable s r e ⟹ ∃l i t. (s', t) |∈| possible_steps e s r l i ⟹ reachable s' e"
  apply (simp add: reachable_def obtainable_def)
  apply clarify
  subgoal for t l i
    apply (rule_tac x="t@[(l, i)]" in exI)
    using obtains_step_append unobtainable_if by blast
  done

lemma possible_steps_remove_unreachable:
  "obtainable s r e ⟹
  ¬ reachable s' e ⟹
  possible_steps (remove_state s' e) s r l i = possible_steps e s r l i"
  apply standard
  apply (simp add: fsubset_eq)
   apply (rule fBallI)
   apply clarsimp
   apply (metis ffmember_filter in_possible_steps remove_state_def)
  apply (simp add: fsubset_eq)
   apply (rule fBallI)
  apply clarsimp
  subgoal for a b
    apply (case_tac "a = s'")
    using reachable_if_obtainable_step apply blast
    apply (simp add: remove_state_def)
    by (metis (mono_tags, lifting) ffmember_filter in_possible_steps obtainable_if_unreachable old.prod.case)
  done
lemma executionally_equivalent_remove_unreachable_state_arbitrary:
  "obtainable s r e ⟹ ¬ reachable s' e ⟹ executionally_equivalent e s r (remove_state s' e) s r x"

proof(induct x arbitrary: s r)
  case (Cons a x)
  then show ?case
```

```
    apply (cases a, simp)
    apply (rule executionally_equivalent.step)
    apply (simp add: possible_steps_remove_unreachable)
    apply standard
     apply clarsimp
    subgoal for aa b ab ba
      apply (rule_tac x="(ab, ba)" in fBexI)
       apply (metis (mono_tags, lifting) obtainable_def obtains_step_append case_prodI)
      apply simp
      done
    apply (rule fBallI)
    apply clarsimp
    apply (rule_tac x="(ab, ba)" in fBexI)
     apply simp
     apply (metis obtainable_def obtains_step_append possible_steps_remove_unreachable)
    by (simp add: possible_steps_remove_unreachable)
qed auto
lemma executionally_equivalent_remove_unreachable_state:
  "¬ reachable s' e ⟹ executionally_equivalent e 0 <> (remove_state s' e) 0 <> x"

  by (meson executionally_equivalent_remove_unreachable_state_arbitrary
      obtains.simps obtains_obtainable)
```

3.2.7 Transition Replacement

Here, we define the function `replace` to replace one transition with another, and prove some of its properties.

```
definition "replace e1 old new = fimage (λx. if x = old then new else x) e1"

lemma replace_finsert:
  "replace (finsert ((aaa, baa), b) e1) old new = (if ((aaa, baa), b) = old then (finsert new (replace e1
old new)) else (finsert ((aaa, baa), b) (replace e1 old new)))"
  by (simp add: replace_def)

lemma possible_steps_replace_unchanged:
  "((s, aa), ba) ≠ ((s1, s2), t1) ⟹
  (aa, ba) |∈| possible_steps e1 s r l i ⟹
  (aa, ba) |∈| possible_steps (replace e1 ((s1, s2), t1) ((s1, s2), t2)) s r l i"
  apply (simp add: in_possible_steps[symmetric] replace_def)
  by fastforce

end
```

3.3 LTL for EFSMs (EFSM_LTL)

This theory builds off the `Linear_Temporal_Logic_on_Streams` theory from the HOL library and defines functions to ease the expression of LTL properties over EFSMs. Since the LTL operators effectively act over traces of models we must find a way to express models as streams.

```
theory EFSM_LTL
imports "Extended_Finite_State_Machines.EFSM" "HOL-Library.Linear_Temporal_Logic_on_Streams"
begin
record state =
  statename :: "nat option"
  datastate :: registers
  action :: action
  "output" :: outputs


type_synonym whitebox_trace = "state stream"

type_synonym property = "whitebox_trace ⇒ bool"
```

```
abbreviation label :: "state ⇒ String.literal" where
  "label s ≡ fst (action s)"

abbreviation inputs :: "state ⇒ value list" where
  "inputs s ≡ snd (action s)"
fun ltl_step :: "transition_matrix ⇒ cfstate option ⇒ registers ⇒ action ⇒ (nat option × outputs ×
registers)" where
  "ltl_step _ None r _ = (None, [], r)" |
  "ltl_step e (Some s) r (l, i) = (let possibilities = possible_steps e s r l i in
                  if possibilities = {|||} then (None, [], r)
                  else
                    let (s', t) = Eps (λx. x |∈| possibilities) in
                    (Some s', (evaluate_outputs t i r), (evaluate_updates t i r))
                 )"

lemma ltl_step_singleton:
"∃t. possible_steps e n r (fst v) (snd v) = {|(aa, t)|} ∧ evaluate_outputs t (snd v) r  = b ∧ evaluate_updates
t (snd v) r = c⟹
ltl_step e (Some n) r v = (Some aa, b, c)"
  apply (cases v)
  by auto

lemma ltl_step_none: "possible_steps e s r a b = {|||} ⟹ ltl_step e (Some s) r (a, b) = (None, [], r)"
  by simp

lemma ltl_step_none_2: "possible_steps e s r (fst ie) (snd ie) = {|||} ⟹ ltl_step e (Some s) r ie = (None,
[], r)"
  by (metis ltl_step_none prod.exhaust_sel)

lemma ltl_step_alt: "ltl_step e (Some s) r t = (
  let possibilities = possible_steps e s r (fst t) (snd t) in
  if possibilities = {|||} then
    (None, [], r)
  else
  let (s', t') = Eps (λx. x |∈| possibilities) in
  (Some s', (apply_outputs (Outputs t') (join_ir (snd t) r)), (apply_updates (Updates t') (join_ir (snd
t) r) r))
)"
  by (case_tac t, simp add: Let_def)

lemma ltl_step_some:
  assumes "possible_steps e s r l i = {|(s', t)|}"
      and "evaluate_outputs t i r = p"
      and "evaluate_updates t i r = r'"
    shows "ltl_step e (Some s) r (l, i) = (Some s', p, r')"
  by (simp add: assms)

lemma ltl_step_cases:
  assumes invalid: "P (None, [], r)"
      and valid: "∀(s', t) |∈| (possible_steps e s r l i). P (Some s', (evaluate_outputs t i r), (evaluate_update
t i r))"
    shows "P (ltl_step e (Some s) r (l, i))"
  apply simp
  apply (case_tac "possible_steps e s r l i")
   apply (simp add: invalid)
  apply simp
  subgoal for x S'
    apply (case_tac "SOME xa. xa = x ∨ xa |∈| S'")
    apply simp
    apply (insert assms(2))
    apply (simp add: fBall_def Ball_def fmember_def)
    by (metis (mono_tags, lifting) fst_conv prod.case_eq_if snd_conv someI_ex)
  done
```

The `make_full_observation` function behaves similarly to `observe_execution` from the `EFSM` theory. The main difference in behaviour is what is recorded. While the observe execution function simply observes an execution of the EFSM to produce the corresponding output for each action, the intention here is to record every detail of execution, including the values of internal variables.

Thinking of each action as a step forward in time, there are five components which characterise a given point in the execution of an EFSM. At each point, the model has a current control state and data state. Each action has a label and some input parameters, and its execution may produce some observableoutput. It is therefore sufficient to provide a stream of 5-tuples containing the current control state, data state, the label and inputs of the action, and computed output. The make full observation function can then be defined as in Figure 9.1, with an additional function watch defined on top of this which starts the make full observation off in the initial control state with the empty data state.

Careful inspection of the definition reveals another way that `make_full_observation` differs from `observe_execution`. Rather than taking a cfstate, it takes a cfstate option. The reason for this is that we need to make our EFSM models complete. That is, we need them to be able to respond to every action from every state like a DFA. If a model does not recognise a given action in a given state, we cannot simply stop processing because we are working with necessarily infinite traces. Since these traces are generated by observing action sequences, the make full observation function must keep processing whether there is a viable transition or not.

To support this, the make full observation adds an implicit "sink state" to every EFSM it processes by lifting control flow state indices from `nat` to `nat option` such that state n is seen as state `Some` n. The control flow state `None` represents a sink state. If a model is unable to recognise a particular action from its current state, it moves into the `None` state. From here, the behaviour is constant for the rest of the time — the control flow state remains None; the data state does not change, and no output is produced.

```
primcorec make_full_observation :: "transition_matrix ⇒ cfstate option ⇒ registers ⇒ outputs ⇒ action
stream ⇒ whitebox_trace" where
  "make_full_observation e s d p i = (
    let (s', o', d') = ltl_step e s d (shd i) in
    (|statename = s, datastate = d, action=(shd i), output = p|)##(make_full_observation e s' d' o' (stl i))
  )"


abbreviation watch :: "transition_matrix ⇒ action stream ⇒ whitebox_trace" where
  "watch e i ≡ (make_full_observation e (Some 0) <> [] i)"
```

3.3.1 Expressing Properties

In order to simplify the expression and understanding of properties, this theory defines a number of named functions which can be used to express certain properties of EFSMs.

State Equality

The STATE_EQ takes a cfstate option representing a control flow state index and returns true if this is the control flow state at the head of the full observation.

```
abbreviation state_eq :: "cfstate option ⇒ whitebox_trace ⇒ bool" where
  "state_eq v s ≡ statename (shd s) = v"

lemma state_eq_holds: "state_eq s = holds (λx. statename x = s)"
  apply (rule ext)
  by (simp add: holds_def)

lemma state_eq_None_not_Some: "state_eq None s ⟹ ¬ state_eq (Some n) s"
  by simp
```

Label Equality

The LABEL_EQ function takes a string and returns true if this is equal to the label at the head of the full observation.

```
abbreviation "label_eq v s ≡ fst (action (shd s)) = (String.implode v)"
```

```
lemma watch_label: "label_eq l (watch e t) = (fst (shd t) = String.implode l)"
  by (simp add: )
```

Input Equality

The INPUT_EQ function takes a value list and returns true if this is equal to the input at the head of the full observation.

```
abbreviation "input_eq v s ≡ inputs (shd s) = v"
```

Action Equality

The ACTION_EQ function takes a (label, value list) pair and returns true if this is equal to the action at the head of the full observation. This effectively combines `label_eq` and `input_eq` into one function.

```
abbreviation "action_eq e ≡ label_eq (fst e) aand input_eq (snd e)"
```

Output Equality

The OUTPUT_EQ function takes a takes a value option list and returns true if this is equal to the output at the head of the full observation.

```
abbreviation "output_eq v s ≡ output (shd s) = v"
datatype ltl_vname = Ip nat | Op nat | Rg nat
```

Checking Arbitrary Expressions

The CHECK_EXP function takes a guard expression and returns true if the guard expression evaluates to true in the given state.

```
type_synonym ltl_gexp = "ltl_vname gexp"

definition join_iro :: "value list ⇒ registers ⇒ outputs ⇒ ltl_vname datastate" where
  "join_iro i r p = (λx. case x of
    Rg n ⇒ r $ n |
    Ip n ⇒ Some (i ! n) |
    Op n ⇒ p ! n
  )"

lemma join_iro_R [simp]: "join_iro i r p (Rg n) = r $ n"
  by (simp add: join_iro_def)

abbreviation "check_exp g s ≡ (gval g (join_iro (snd (action (shd s))) (datastate (shd s)) (output (shd
s))) = trilean.true)"

lemma alw_ev: "alw f = not (ev (λs. ¬f s))"
  by simp

lemma alw_state_eq_smap:
  "alw (state_eq s) ss = alw (λss. shd ss = s) (smap statename ss)"
  apply standard
   apply (simp add: alw_iff_sdrop )
  by (simp add: alw_mono alw_smap )
```

3.3.2 Sink State

Once the sink state is entered, it cannot be left and there are no outputs or updates henceforth.

```
lemma shd_state_is_none: "(state_eq None) (make_full_observation e None r p t)"
  by (simp add: )

lemma unfold_observe_none: "make_full_observation e None d p t = (⦇statename = None, datastate = d, action=(shd
t), output = p⦈##(make_full_observation e None d [] (stl t)))"
```

```
  by (simp add: stream.expand)

lemma once_none_always_none_aux:
  assumes "∃ p r i. j = (make_full_observation e None r p) i"
  shows "alw (state_eq None) j"
  using assms apply coinduct
  apply simp
  by fastforce

lemma once_none_always_none: "alw (state_eq None) (make_full_observation e None r p t)"
  using once_none_always_none_aux by blast

lemma once_none_nxt_always_none: "alw (nxt (state_eq None)) (make_full_observation e None r p t)"
  using once_none_always_none
  by (simp add: alw_iff_sdrop del: sdrop.simps)

lemma snth_sconst: "(∀ i. s !! i = h) = (s = sconst h)"
  by (metis funpow_code_def id_funpow sdrop_simps(1) sdrop_siterate siterate.simps(1) smap_alt smap_sconst
snth.simps(1) stream.map_id)

lemma alw_sconst: "(alw (λxs. shd xs = h) t) = (t = sconst h)"
  by (simp add: snth_sconst[symmetric] alw_iff_sdrop)

lemma smap_statename_None: "smap statename (make_full_observation e None r p i) = sconst None"
  by (meson EFSM_LTL.alw_sconst alw_state_eq_smap once_none_always_none)

lemma alw_not_some: "alw (λxs. statename (shd xs) ≠ Some s) (make_full_observation e None r p t)"
  by (metis (mono_tags, lifting) alw_mono once_none_always_none option.distinct(1) )

lemma state_none: "((state_eq None) impl nxt (state_eq None)) (make_full_observation e s r p t)"
  by (simp add: )

lemma state_none_2:
  "(state_eq None) (make_full_observation e s r p t) ⟹
   (state_eq None) (make_full_observation e s r p (stl t))"
  by (simp add: )

lemma no_output_none_aux:
  assumes "∃ p r i. j = (make_full_observation e None r []) i"
  shows "alw (output_eq []) j"
  using assms apply coinduct
  apply simp
  by fastforce

lemma no_output_none: "nxt (alw (output_eq [])) (make_full_observation e None r p t)"
  using no_output_none_aux by auto

lemma nxt_alw: "nxt (alw P) s ⟹ alw (nxt P) s"
  by (simp add: alw_iff_sdrop)

lemma no_output_none_nxt: "alw (nxt (output_eq [])) (make_full_observation e None r p t)"
  using nxt_alw no_output_none by blast

lemma no_output_none_if_empty: "alw (output_eq []) (make_full_observation e None r [] t)"
  by (metis (mono_tags, lifting) alw_nxt make_full_observation.simps(1) no_output_none state.select_convs(4))

lemma no_updates_none_aux:
  assumes "∃ p i. j = (make_full_observation e None r p) i"
  shows "alw (λx. datastate (shd x) = r) j"
  using assms apply coinduct
  by fastforce

lemma no_updates_none: "alw (λx. datastate (shd x) = r) (make_full_observation e None r p t)"
```

using *no_updates_none_aux* **by** *blast*

lemma *action_components: "(label_eq l aand input_eq i) s = (action (shd s) = (String.implode l, i))"*
by *(metis fst_conv prod.collapse snd_conv)*

end

4 Examples

In this chapter, we provide some examples of EFSMs and proofs over them. We first present a formalisation of a simple drinks machine. Next, we prove observational equivalence of an alternative model. Finally, we prove some temporal properties of the first example.

4.1 Drinks Machine (Drinks_Machine)

This theory formalises a simple drinks machine. The *select* operation takes one argument - the desired beverage. The *coin* operation also takes one parameter representing the value of the coin. The *vend* operation has two flavours - one which dispenses the drink if the customer has inserted enough money, and one which dispenses nothing if the user has not inserted sufficient funds.

We first define a datatype *statemane* which corresponds to S in the formal definition. Note that, while statename has four elements, the drinks machine presented here only requires three states. The fourth element is included here so that the *statename* datatype may be used in the next example.

```
theory Drinks_Machine
  imports "Extended_Finite_State_Machines.EFSM"
begin
definition select :: "transition" where
"select ≡ ⦇
        Label = STR ''select'',
        Arity = 1,
        Guards = [],
        Outputs = [],
        Updates = [
                    (1, V (I 0)),
                    (2, L (Num 0))
                  ]
      ⦈"


definition coin :: "transition" where
"coin ≡ ⦇
        Label = STR ''coin'',
        Arity = 1,
        Guards = [],
        Outputs = [Plus (V (R 2)) (V (I 0))],
        Updates = [
                    (1, V (R 1)),
                    (2, Plus (V (R 2)) (V (I 0)))
                  ]
      ⦈"


definition vend:: "transition" where
"vend≡ ⦇
        Label = STR ''vend'',
        Arity = 0,
        Guards = [(Ge (V (R 2)) (L (Num 100)))],
        Outputs =  [(V (R 1))],
        Updates = [(1, V (R 1)), (2, V (R 2))]
      ⦈"


definition vend_fail :: "transition" where
```

```
"vend_fail ≡ ⦇
        Label = STR ''vend'',
        Arity = 0,
        Guards = [(Lt (V (R 2)) (L (Num 100)))],
        Outputs =  [],
        Updates = [(1, V (R 1)), (2, V (R 2))]
      ⦈"


definition drinks :: "transition_matrix" where
"drinks ≡ {|
          ((0,1), select),
          ((1,1), coin),
          ((1,1), vend_fail),
          ((1,2), vend)
         |}"

lemmas transitions = select_def coin_def vend_def vend_fail_def

lemma apply_updates_vend: "apply_updates (Updates vend) (join_ir [] r) r = r"
  by (simp add: vend_def apply_updates_def)

lemma drinks_states: "S drinks = {|0, 1, 2|}"
  apply (simp add: S_def drinks_def)
  by auto

lemma possible_steps_0:
  "length i = 1 ⟹
   possible_steps drinks 0 r (STR ''select'') i = {|(1, select)|}"
  apply (simp add: possible_steps_singleton drinks_def)
  apply safe
  by (simp_all add: transitions apply_guards_def)

lemma first_step_select:
  "(s', t) |∈| possible_steps drinks 0 r aa b ⟹ s' = 1 ∧ t = select"
  apply (simp add: possible_steps_def fimage_def ffilter_def fmember_def Abs_fset_inverse Set.filter_def
drinks_def)
  apply safe
  by (simp_all add: transitions)

lemma drinks_vend_insufficient:
  "r $ 2 = Some (Num x1) ⟹
   x1 < 100 ⟹
   possible_steps drinks 1 r (STR ''vend'') [] = {|(1, vend_fail)|}"
  apply (simp add: possible_steps_singleton drinks_def)
  apply safe
  by (simp_all add: transitions apply_guards_def value_gt_def join_ir_def connectives)

lemma drinks_vend_invalid:
  "∄n. r $ 2 = Some (Num n) ⟹
   possible_steps drinks 1 r (STR ''vend'') [] = {||}"
  apply (simp add: possible_steps_empty drinks_def can_take_transition_def can_take_def transitions)
  by (simp add: MaybeBoolInt_not_num_1 value_gt_def)

lemma possible_steps_1_coin:
  "length i = 1 ⟹ possible_steps drinks 1 r (STR ''coin'') i = {|(1, coin)|}"
  apply (simp add: possible_steps_singleton drinks_def)
  apply safe
  by (simp_all add: transitions)

lemma possible_steps_2_vend:
  "∃n. r $ 2 = Some (Num n) ∧ n ≥ 100 ⟹
   possible_steps drinks 1 r (STR ''vend'') [] = {|(2, vend)|}"
```

```
  apply (simp add: possible_steps_singleton drinks_def)
  apply safe
  by (simp_all add: transitions apply_guards_def value_gt_def join_ir_def connectives)

lemma recognises_from_2:
  "recognises_execution drinks 1 <1 $:= d, 2 $:= Some (Num 100)> [(STR ''vend'', [])]"
  apply (rule recognises_execution.step)
  apply (rule_tac x="(2, vend)" in fBexI)
   apply simp
  by (simp add: possible_steps_2_vend)

lemma recognises_from_1a:
  "recognises_execution drinks 1 <1 $:= d, 2 $:= Some (Num 50)> [(STR ''coin'', [Num 50]), (STR ''vend'',
[])]"
  apply (rule recognises_execution.step)
  apply (rule_tac x="(1, coin)" in fBexI)
   apply (simp add: apply_updates_def coin_def finfun_update_twist value_plus_def recognises_from_2)
  by (simp add: possible_steps_1_coin)

lemma recognises_from_1: "recognises_execution drinks 1 <2 $:= Some (Num 0), 1 $:= Some d>
     [(STR ''coin'', [Num 50]), (STR ''coin'', [Num 50]), (STR ''vend'', [])]"
  apply (rule recognises_execution.step)
  apply (rule_tac x="(1, coin)" in fBexI)
   apply (simp add: apply_updates_def coin_def value_plus_def finfun_update_twist recognises_from_1a)
  by (simp add: possible_steps_1_coin)

lemma purchase_coke:
  "observe_execution drinks 0 <> [(STR ''select'', [Str ''coke'']), (STR ''coin'', [Num 50]), (STR ''coin'',
[Num 50]), (STR ''vend'', [])] =
                       [[], [Some (Num 50)], [Some (Num 100)], [Some (Str ''coke'')]]"
  by (simp add: possible_steps_0 possible_steps_1_coin possible_steps_2_vend transitions
                  apply_outputs_def apply_updates_def value_plus_def)

lemma rejects_input:
  "l ≠ STR ''coin'' ⟹
   l ≠ STR ''vend'' ⟹
   ¬ recognises_execution drinks 1 d' [(l, i)]"
  apply (rule no_possible_steps_rejects)
  by (simp add: possible_steps_empty drinks_def can_take_transition_def can_take_def transitions)

lemma rejects_recognises_prefix: "l ≠ STR ''coin'' ⟹
   l ≠ STR ''vend'' ⟹
   ¬ (recognises drinks [(STR ''select'', [Str ''coke'']), (l, i)])"
  apply (rule trace_reject_later)
  apply (simp add: possible_steps_0 select_def join_ir_def input2state_def)
  using rejects_input by blast

lemma rejects_termination:
  "observe_execution drinks 0 <> [(STR ''select'', [Str ''coke'']), (STR ''rejects'', [Num 50]), (STR ''coin'',
[Num 50])] = [[]]"
  apply (rule observe_execution_step)
   apply (simp add: step_def possible_steps_0 select_def)
  apply (rule observe_execution_no_possible_step)
  by (simp add: possible_steps_empty drinks_def can_take_transition_def can_take_def transitions)


lemma r2_0_vend:
  "can_take_transition vend i r ⟹
   ∃n. r $ 2 = Some (Num n) ∧ n ≥ 100"
  apply (simp add: can_take_transition_def can_take_def vend_def apply_guards_def maybe_negate_true maybe_or_false
connectives value_gt_def)
  using MaybeBoolInt.elims by force
```

```
lemma drinks_vend_sufficient: "r $ 2 = Some (Num x1) ⟹
               x1 ≥ 100 ⟹
               possible_steps drinks 1 r (STR ''vend'') [] = {|(2, vend)|}"
  using possible_steps_2_vend by blast

lemma drinks_end: "possible_steps drinks 2 r a b = {||}"
  apply (simp add: possible_steps_def drinks_def transitions)
  by force

lemma drinks_vend_r2_String:
  "r $ 2 = Some (value.Str x2) ⟹
   possible_steps drinks 1 r (STR ''vend'') [] = {||}"
  apply (simp add: possible_steps_empty drinks_def can_take_transition_def can_take_def transitions)
  by (simp add: value_gt_def)

lemma drinks_vend_r2_rejects:
  "∄n. r $ 2 = Some (Num n) ⟹ step drinks 1 r (STR ''vend'') [] = None"
  apply (rule no_possible_steps_1)
  apply (simp add: possible_steps_empty drinks_def can_take_transition_def can_take_def transitions)
  by (simp add: MaybeBoolInt_not_num_1 value_gt_def)

lemma drinks_0_rejects:
  "¬ (fst a = STR ''select'' ∧ length (snd a) = 1) ⟹
    (possible_steps drinks 0 r (fst a) (snd a)) = {||}"
  apply (simp add: drinks_def possible_steps_def transitions)
  by force

lemma drinks_vend_empty: "(possible_steps drinks 0 <> (STR ''vend'') []) = {||}"
  using drinks_0_rejects
  by auto

lemma drinks_1_rejects:
  "fst a = STR ''coin'' ⟶ length (snd a) ≠ 1 ⟹
          a ≠ (STR ''vend'', []) ⟹
          possible_steps drinks 1 r (fst a) (snd a) = {||}"
  apply (simp add: possible_steps_empty drinks_def can_take_transition_def can_take_def transitions)
  by (metis prod.collapse)

lemma drinks_rejects_future: "¬ recognises_execution drinks 2 d ((1, i)#t)"
  apply (rule no_possible_steps_rejects)
  by (simp add: possible_steps_empty drinks_def)

lemma drinks_1_rejects_trace:
  assumes not_vend: "e ≠ (STR ''vend'', [])"
      and not_coin: "∄i. e = (STR ''coin'', [i])"
  shows "¬ recognises_execution drinks 1 r (e # es)"
proof-
  show ?thesis
    apply (cases e, simp)
    subgoal for a b
      apply (rule no_possible_steps_rejects)
      apply (simp add: possible_steps_empty drinks_def can_take_transition_def can_take_def transitions)
      apply (case_tac b)
      using not_vend apply simp
      using not_coin by auto
    done
qed

lemma rejects_state_step: "s > 1 ⟹ step drinks s r 1 i = None"
  apply (rule no_possible_steps_1)
  by (simp add: possible_steps_empty drinks_def)

lemma invalid_other_states:
```

```
  "s > 1 ⟹ ¬ recognises_execution drinks s r ((aa, b) # t)"
  apply (rule no_possible_steps_rejects)
  by (simp add: possible_steps_empty drinks_def)

lemma vend_ge_100:
  "possible_steps drinks 1 r 1 i = {|(2, vend)|} ⟹
   ¬? value_gt (Some (Num 100)) (r $ 2) = trilean.true"
  apply (insert possible_steps_apply_guards[of drinks 1 r 1 i 2 vend])
  by (simp add: possible_steps_def apply_guards_def vend_def)

lemma drinks_no_possible_steps_1:
  assumes not_coin: "¬ (a = STR ''coin'' ∧ length b = 1)"
      and not_vend: "¬ (a = STR ''vend'' ∧ b = [])"
    shows "possible_steps drinks 1 r a b = {||}"
  using drinks_1_rejects not_coin not_vend by auto

lemma possible_steps_0_not_select: "a ≠ STR ''select'' ⟹
       possible_steps drinks 0 <> a b = {||}"
  apply (simp add: possible_steps_def ffilter_def fset_both_sides Abs_fset_inverse Set.filter_def drinks_def)
  apply safe
  by (simp_all add: select_def)

lemma possible_steps_select_wrong_arity: "a = STR ''select'' ⟹
       length b ≠ 1 ⟹
       possible_steps drinks 0 <> a b = {||}"
  apply (simp add: possible_steps_def ffilter_def fset_both_sides Abs_fset_inverse Set.filter_def drinks_def)
  apply safe
  by (simp_all add: select_def)

lemma possible_steps_0_invalid:
  "¬ (1 = STR ''select'' ∧ length i = 1) ⟹
   possible_steps drinks 0 <> 1 i = {||}"
  using possible_steps_0_not_select possible_steps_select_wrong_arity by fastforce

end
```

4.2 An Observationally Equivalent Model (Drinks_Machine_2)

This theory defines a second formalisation of the drinks machine example which produces identical output to the first model. This property is called *observational equivalence* and is discussed in more detail in [2].

```
theory Drinks_Machine_2
  imports Drinks_Machine
begin



definition vend_nothing :: "transition" where
"vend_nothing ≡ (|
        Label = (STR ''vend''),
        Arity = 0,
        Guards = [],
        Outputs =  [],
        Updates = [(1, V (R 1)), (2, V (R 2))]
      |)"

lemmas transitions = Drinks_Machine.transitions vend_nothing_def

definition drinks2 :: transition_matrix where
"drinks2 = {|
              ((0,1), select),
              ((1,1), vend_nothing),
              ((1,2), coin),
```

```
            ((2,2), coin),
            ((2,2), vend_fail),
            ((2,3), vend)
        |}"

lemma possible_steps_0:
  "length i = 1 ⟹
   possible_steps drinks2 0 r ((STR ''select'')) i = {|(1, select)|}"
  apply (simp add: possible_steps_def drinks2_def transitions)
  by force

lemma possible_steps_1:
  "length i = 1 ⟹
   possible_steps drinks2 1 r ((STR ''coin'')) i = {|(2, coin)|}"
  apply (simp add: possible_steps_def drinks2_def transitions)
  by force

lemma possible_steps_2_coin:
  "length i = 1 ⟹
   possible_steps drinks2 2 r ((STR ''coin'')) i = {|(2, coin)|}"
  apply (simp add: possible_steps_def drinks2_def transitions)
  by force

lemma possible_steps_2_vend:
  "r $ 2 = Some (Num n) ⟹
   n ≥ 100 ⟹
   possible_steps drinks2 2 r ((STR ''vend'')) [] = {|(3, vend)|}"
  apply (simp add: possible_steps_singleton drinks2_def)
  apply safe
  by (simp_all add: transitions apply_guards_def value_gt_def join_ir_def connectives)

lemma recognises_first_select:
  "recognises_execution drinks 0 r ((aa, b) # as) ⟹ aa = STR ''select'' ∧ length b = 1"
  using recognises_must_be_possible_step[of drinks 0 r "(aa, b)" as]
  apply simp
  apply clarify
  by (metis first_step_select recognises_possible_steps_not_empty drinks_0_rejects fst_conv snd_conv)

lemma drinks2_vend_insufficient:
  "possible_steps drinks2 1 r ((STR ''vend'')) [] = {|(1, vend_nothing)|}"
  apply (simp add: possible_steps_def drinks2_def transitions)
  by force

lemma drinks2_vend_insufficient2:
  "r $ 2 = Some (Num x1) ⟹
   x1 < 100 ⟹
   possible_steps drinks2 2 r ((STR ''vend'')) [] = {|(2, vend_fail)|}"
  apply (simp add: possible_steps_singleton drinks2_def)
  apply safe
  by (simp_all add: transitions apply_guards_def value_gt_def join_ir_def connectives)

lemma drinks2_vend_sufficient: "r $ 2 = Some (Num x1) ⟹
               ¬ x1 < 100 ⟹
               possible_steps drinks2 2 r ((STR ''vend'')) [] = {|(3, vend)|}"
  apply (simp add: possible_steps_singleton drinks2_def)
  apply safe
  by (simp_all add: transitions apply_guards_def value_gt_def join_ir_def connectives)

lemma recognises_1_2: "recognises_execution drinks 1 r t ⟶ recognises_execution drinks2 2 r t"
proof(induct t arbitrary: r)
  case (Cons a as)
  then show ?case
    apply (cases a)
```

```
    apply (simp add: recognises_step_equiv)
    apply (case_tac "a=(STR ''vend'', [])")
     apply (case_tac "∃n. r$2 = Some (Num n)")
      apply clarify
      apply (case_tac "n < 100")
       apply (simp add: drinks_vend_insufficient drinks2_vend_insufficient2)
      apply (simp add: drinks_vend_sufficient drinks2_vend_sufficient)
      apply (metis recognises_prim recognises_prim.elims(3) drinks_rejects_future)
    using drinks_vend_invalid apply blast
    apply (case_tac "∃i. a=(STR ''coin'', [i])")
     apply clarify
     apply (simp add: possible_steps_1_coin possible_steps_2_coin)
    by (metis recognises_execution.simps drinks_1_rejects_trace)
qed auto

lemma drinks_reject_0_2:
  "∄i. a = (STR ''select'', [i]) ⟹
   possible_steps drinks 0 r (fst a) (snd a) = {||}"
  apply (rule drinks_0_rejects)
  by (cases a, case_tac "snd a", auto)

lemma purchase_coke:
  "observe_execution drinks2 0 <> [((STR ''select''), [Str ''coke'']), ((STR ''coin''), [Num 50]), ((STR
''coin''), [Num 50]), ((STR ''vend''), [])] =
                       [[], [Some (Num 50)], [Some (Num 100)], [Some (Str ''coke'')]]"
  by (simp add: possible_steps_0 select_def apply_updates_def
                possible_steps_1 coin_def apply_outputs finfun_update_twist
                possible_steps_2_coin possible_steps_2_vend vend_def)

lemma drinks2_0_invalid:
  "¬ (aa = (STR ''select'') ∧ length (b) = 1) ⟹
    (possible_steps drinks2 0 <> aa b) = {||}"
  apply (simp add: drinks2_def possible_steps_def transitions)
  by force

lemma drinks2_vend_r2_none:
  "r $ 2 = None ⟹ possible_steps drinks2 2 r ((STR ''vend'')) [] = {||}"
  apply (simp add: possible_steps_empty drinks2_def can_take_transition_def can_take_def transitions)
  by (simp add: value_gt_def)

lemma drinks2_end: "possible_steps drinks2 3 r a b = {||}"
  apply (simp add: possible_steps_def drinks2_def transitions)
  by force

lemma drinks2_vend_r2_String: "r $ 2 = Some (value.Str x2) ⟹
               possible_steps drinks2 2 r ((STR ''vend'')) [] = {||}"
  apply (simp add: possible_steps_empty drinks2_def)
  apply safe
  by (simp_all add: transitions can_take_transition_def can_take_def value_gt_def)

lemma drinks2_2_invalid:
  "fst a = (STR ''coin'') ⟶ length (snd a) ≠ 1 ⟹
         a ≠ ((STR ''vend''), []) ⟹
         possible_steps drinks2 2 r (fst a) (snd a) = {||}"
  apply (simp add: possible_steps_empty drinks2_def transitions can_take_transition_def can_take_def)
  by (metis prod.collapse)

lemma drinks2_1_invalid:
  "¬(a = (STR ''coin'') ∧ length b = 1) ⟹
      ¬(a = (STR ''vend'') ∧ b = []) ⟹
    possible_steps drinks2 1 r a b = {||}"
  apply (simp add: possible_steps_empty drinks2_def)
  apply safe
```

```
  by (simp_all add: transitions can_take_transition_def can_take_def value_gt_def)

lemma drinks2_vend_invalid:
  "∄n. r $ 2 = Some (Num n) ⟹
   possible_steps drinks2 2 r (STR ''vend'') [] = {||}"
  apply (simp add: possible_steps_empty drinks2_def)
  apply safe
  by (simp_all add: transitions can_take_transition_def can_take_def value_gt_def MaybeBoolInt_not_num_1)

lemma equiv_1_2: "executionally_equivalent drinks 1 r drinks2 2 r x"
proof(induct x arbitrary: r)
  case (Cons a t)
  then show ?case
    apply (cases a, clarsimp)
    apply (simp add: executionally_equivalent_step)
    apply (case_tac "fst a = STR ''coin'' ∧ length (snd a) = 1")
     apply (simp add: Drinks_Machine.possible_steps_1_coin possible_steps_2_coin)
    apply (case_tac "a = (STR ''vend'', [])")
     defer using drinks2_2_invalid drinks_no_possible_steps_1 apply auto[1]
    apply (case_tac "∃n. r $ 2 = Some (Num n)")
     defer using drinks_vend_invalid drinks2_vend_invalid apply simp
    apply clarify
    subgoal for aa b n
      apply (case_tac "n < 100")
       apply (simp add: Drinks_Machine.drinks_vend_insufficient drinks2_vend_insufficient2)
      apply (simp add: Drinks_Machine.drinks_vend_sufficient drinks2_vend_sufficient)
      apply (induct t)
       apply simp
      subgoal for aa t apply (case_tac aa, clarsimp)
        by (simp add: executionally_equivalent_step drinks_end drinks2_end)
      done
    done
qed auto

lemma equiv_1_1: "r$2 = Some (Num 0) ⟹ executionally_equivalent drinks 1 r drinks2 1 r x"
proof(induct x)
  case (Cons a t)
  then show ?case
    apply (cases a, clarsimp)
    subgoal for aa b
      apply (simp add: executionally_equivalent_step)
      apply (case_tac "aa = STR ''coin'' ∧ length b = 1")
       apply (simp add: possible_steps_1_coin possible_steps_1 equiv_1_2)
      apply (case_tac "a = (STR ''vend'', [])")
       apply clarsimp
       apply (simp add: drinks_vend_insufficient drinks2_vend_insufficient)
       apply (simp add: vend_fail_def vend_nothing_def apply_updates_def)
       apply (metis finfun_upd_triv)
      by (simp add: drinks2_1_invalid drinks_no_possible_steps_1)
    done
qed auto


lemma executional_equivalence: "executionally_equivalent drinks 0 <> drinks2 0 <> t"
proof(induct t)
  case (Cons a t)
  then show ?case
    apply (cases a, clarify)
    subgoal for aa b
      apply (simp add: executionally_equivalent_step)
      apply (case_tac "aa = STR ''select'' ∧ length b = 1")
       apply (simp add: Drinks_Machine.possible_steps_0 possible_steps_0)
       apply (simp add: apply_updates_def select_def equiv_1_1)
```

```
    by (simp add: drinks2_0_invalid possible_steps_0_invalid)
  done
qed auto

lemma observational_equivalence: "trace_equivalent drinks drinks2"
  by (simp add: executional_equivalence executionally_equivalent_trace_equivalent)

end
```

4.3 Temporal Properties (Drinks_Machine_LTL)

This theory presents some examples of temporal properties over the simple drinks machine.

```
theory Drinks_Machine_LTL
imports "Drinks_Machine" "Extended_Finite_State_Machines.EFSM_LTL"
begin

declare One_nat_def [simp del]

lemma P_ltl_step_0:
  assumes invalid: "P (None, [], <>)"
  assumes select: "l = STR ''select'' ⟶ P (Some 1, [], <1 $:= Some (hd i), 2 $:= Some (Num 0)>)"
  shows "P (ltl_step drinks (Some 0) <> (l, i))"
proof-
  have length_i: "∃d. (l, i) = (STR ''select'', [d]) ⟹ length i = 1"
    by (induct i, auto)
  have length_i_2: "∀d. i ≠ [d] ⟹ length i ≠ 1"
    by (induct i, auto)
  show ?thesis
    apply (case_tac "∃d. (l, i) = (STR ''select'', [d])")
     apply (simp add: possible_steps_0 length_i select_def apply_updates_def)
    using select apply auto[1]
    by (simp add: possible_steps_0_invalid length_i_2 invalid)
qed

lemma P_ltl_step_1:
  assumes invalid: "P (None, [], r)"
  assumes coin: "l = STR ''coin'' ⟶ P (Some 1, [value_plus (r $ 2) (Some (hd i))], r(2 $:= value_plus
(r $ 2) (Some (i ! 0))))"
  assumes vend_fail: "value_gt (Some (Num 100)) (r $ 2) = trilean.true ⟶ P (Some 1, [],r)"
  assumes vend: "¬? value_gt (Some (Num 100)) (r $ 2) = trilean.true ⟶ P (Some 2, [r$1], r)"
  shows "P (ltl_step drinks (Some 1) r (l, i))"
proof-
  have length_i: "⋀s. ∃d. (l, i) = (s, [d]) ⟹ length i = 1"
    by (induct i, auto)
  have length_i_2: "∀d. i ≠ [d] ⟹ length i ≠ 1"
    by (induct i, auto)
  show ?thesis
    apply (case_tac "∃d. (l, i) = (STR ''coin'', [d])")
     apply (simp add: possible_steps_1_coin length_i coin_def apply_outputs_def apply_updates_def)
    using coin apply auto[1]
    apply (case_tac "(l, i) = (STR ''vend'', [])")
     apply (case_tac "∃n. r $ 2 = Some (Num n)")
      apply clarsimp
    subgoal for n
      apply (case_tac "n ≥ 100")
       apply (simp add: drinks_vend_sufficient vend_def apply_updates_def apply_outputs_def)
       apply (metis finfun_upd_triv possible_steps_2_vend vend vend_ge_100)
      apply (simp add: drinks_vend_insufficient vend_fail_def apply_updates_def apply_outputs_def)
      apply (metis MaybeBoolInt.simps(1) finfun_upd_triv not_less value_gt_def vend_fail)
      done
     apply (simp add: drinks_vend_invalid invalid)
    by (simp add: drinks_no_possible_steps_1 length_i_2 invalid)
qed
```

```
lemma LTL_r2_not_always_gt_100: "not (alw (check_exp (Gt (V (Rg 2)) (L (Num 100))))) (watch drinks i)"
  using value_gt_def by auto

lemma drinks_step_2_none: "ltl_step drinks (Some 2) r e = (None, [], r)"
  by (simp add: drinks_end ltl_step_none_2)

lemma one_before_two_2:
  "alw (λx. statename (shd (stl x)) = Some 2 ⟶ statename (shd x) = Some 1) (make_full_observation drinks
(Some 2) r [r $ 1] x2a)"
proof(coinduction)
  case alw
  then show ?case
    apply (simp add: drinks_step_2_none)
    by (metis (mono_tags, lifting) alw_mono nxt.simps once_none_nxt_always_none option.distinct(1))
qed

lemma one_before_two_aux:
  assumes "∃ p r i. j = nxt (make_full_observation drinks (Some 1) r p) i"
  shows "alw (λx. nxt (state_eq (Some 2)) x ⟶ state_eq (Some 1) x) j"
  using assms apply(coinduct)
  apply simp
  apply clarify
  apply standard
   apply simp
  apply simp
  subgoal for r i
    apply (case_tac "shd (stl i)")
    apply (simp del: ltl_step.simps)
    apply (rule P_ltl_step_1)
       apply (rule disjI2)
       apply (rule alw_mono[of "nxt (state_eq None)"])
        apply (simp add: once_none_nxt_always_none)
       apply simp
      apply auto[1]
     apply auto[1]
    apply simp
    by (simp add: one_before_two_2)
  done

lemma LTL_nxt_2_means_vend:
  "alw (nxt (state_eq (Some 2)) impl (state_eq (Some 1))) (watch drinks i)"
proof(coinduction)
  case alw
  then show ?case
    apply (case_tac "shd i")
    apply (simp del: ltl_step.simps)
    apply (rule P_ltl_step_0)
     apply simp
     apply (rule disjI2)
     apply (rule alw_mono[of "nxt (state_eq None)"])
      apply (simp add: once_none_nxt_always_none)
    using one_before_two_aux by auto
qed

lemma costsMoney_aux:
  assumes "∃p r i. j = (nxt (make_full_observation drinks (Some 1) r p) i)"
  shows "alw (λxs. nxt (state_eq (Some 2)) xs ⟶ check_exp (Ge (V (Rg 2)) (L (Num 100))) xs) j"
  using assms apply coinduct
  apply clarsimp
  subgoal for r i
    apply (case_tac "shd (stl i)")
    apply (simp del: ltl_step.simps)
```

```
    apply (rule P_ltl_step_1)
       apply simp
       apply (rule disjI2)
       apply (rule alw_mono[of "nxt (state_eq None)"])
        apply (simp add: once_none_nxt_always_none)
       apply simp
      apply auto[1]
     apply auto[1]
    apply simp
    apply standard
    apply (rule disjI2)
    apply (rule alw_mono[of "nxt (state_eq None)"])
     apply (metis (no_types, lifting) drinks_step_2_none fst_conv make_full_observation.sel(2) nxt.simps
nxt_alw once_none_always_none_aux)
    by simp
  done


lemma LTL_costsMoney:
  "(alw (nxt (state_eq (Some 2)) impl (check_exp (Ge (V (Rg 2)) (L (Num 100)))))) (watch drinks i)"
proof(coinduction)
  case alw
  then show ?case
    apply (cases "shd i")
    subgoal for l ip
      apply (case_tac "l = STR ''select'' ∧ length ip = 1")
       defer
       apply (simp add: possible_steps_0_invalid)
       apply (rule disjI2)
       apply (rule alw_mono[of "nxt (state_eq None)"])
        apply (simp add: once_none_nxt_always_none)
       apply (simp add: )
      apply (simp add: possible_steps_0 select_def)
      apply (rule disjI2)
      apply (simp only: nxt.simps[symmetric])
      using costsMoney_aux by auto
    done
qed

lemma LTL_costsMoney_aux:
  "(alw (not (check_exp (Ge (V (Rg 2)) (L (Num 100)))) impl (not (nxt (state_eq (Some 2)))))) (watch drinks
i)"
  by (metis (no_types, lifting) LTL_costsMoney alw_mono)

lemma implode_select: "String.implode ''select'' = STR ''select''"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_coin: "String.implode ''coin'' = STR ''coin''"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemma implode_vend: "String.implode ''vend'' = STR ''vend''"
  by (metis Literal.rep_eq String.implode_explode_eq zero_literal.rep_eq)

lemmas implode_labels = implode_select implode_coin implode_vend

lemma LTL_neverReachS2:"(((((action_eq (''select'', [Str ''coke'']))) 
                    aand
                    (nxt ((action_eq (''coin'', [Num 100])))))
                    aand
                    (nxt (nxt((label_eq ''vend'' aand (input_eq [])))))
                    impl
                    (nxt (nxt (nxt (state_eq (Some 2)))))
                    (watch drinks i)"
```

```
  apply (simp add: implode_labels)
  apply (cases i)
  apply clarify
  apply simp
  apply (simp add: possible_steps_0 select_def)
  apply (case_tac "shd x2", clarify)
  apply (simp add: possible_steps_1_coin coin_def value_plus_def finfun_update_twist apply_updates_def)
  apply (case_tac "shd (stl x2)", clarify)
  by (simp add: drinks_vend_sufficient )

lemma ltl_step_not_select:
  "∄i. e = (STR ''select'', [i]) ⟹
   ltl_step drinks (Some 0) r e = (None, [], r)"
  apply (cases e, clarify)
  subgoal for a b
    apply (rule ltl_step_none)
    apply (simp add: possible_steps_empty drinks_def can_take_transition_def can_take_def select_def)
    by (cases e, case_tac b, auto)
  done

lemma ltl_step_select:
  "ltl_step drinks (Some 0) <> (STR ''select'', [i]) = (Some 1, [], <1 $:= Some i, 2 $:= Some (Num 0)>)"
  apply (rule  ltl_step_some[of _ _ _ _ _ _ select])
    apply (simp add: possible_steps_0)
   apply (simp add: select_def)
  by (simp add: select_def finfun_update_twist apply_updates_def)

lemma ltl_step_not_coin_or_vend:
  "∄i. e = (STR ''coin'', [i]) ⟹
    e ≠ (STR ''vend'', []) ⟹
    ltl_step drinks (Some 1) r e = (None, [], r)"
  apply (cases e)
  subgoal for a b
    apply (simp del: ltl_step.simps)
    apply (rule ltl_step_none)
    apply (simp add: possible_steps_empty drinks_def can_take_transition_def can_take_def transitions)
    by (case_tac e, case_tac b, auto)
  done

lemma ltl_step_coin:
  "∃p r'. ltl_step drinks (Some 1) r (STR ''coin'', [i]) = (Some 1, p, r')"
  by (simp add: possible_steps_1_coin)

lemma alw_tl:
  "alw φ (make_full_observation e (Some 0) <> [] xs) ⟹
    alw φ
     (make_full_observation e (fst (ltl_step e (Some 0) <> (shd xs))) (snd (snd (ltl_step e (Some 0) <>
(shd xs))))
       (fst (snd (ltl_step e (Some 0) <> (shd xs)))) (stl xs))"
  by auto

lemma stop_at_none:
  "alw (λxs. output (shd (stl xs)) = [Some (EFSM.Str drink)] ⟶ check_exp (Ge (V (Rg 2)) (L (Num 100)))
xs)
           (make_full_observation drinks None r p t)"
  apply (rule alw_mono[of "nxt (output_eq [])"])
   apply (simp add: no_output_none_nxt)
  by simp

lemma drink_costs_money_aux:
  assumes "∃p r t. j = make_full_observation drinks (Some 1) r p t"
  shows "alw (λxs. output (shd (stl xs)) = [Some (EFSM.Str drink)] ⟶ check_exp (Ge (V (Rg 2)) (L (Num
100))) xs) j"
```

```
  using assms apply coinduct
  apply clarsimp
  apply (case_tac "shd t")
  apply (simp del: ltl_step.simps)
  apply (rule P_ltl_step_1)
     apply simp
     apply (rule disjI2)
     apply (rule alw_mono[of "nxt (output_eq [])"])
      apply (simp add: no_output_none_nxt)
     apply simp
    apply (simp add: Str_def value_plus_never_string)
    apply auto[1]
   apply auto[1]
  apply simp
  apply standard
  apply (rule disjI2)
  apply (rule alw_mono[of "nxt (output_eq [])"])
   apply (simp add: drinks_step_2_none no_output_none_if_empty nxt_alw)
  by simp

lemma LTL_drinks_cost_money:
  "alw (nxt (output_eq [Some (Str drink)]) impl (check_exp (Ge (V (Rg 2)) (L (Num 100))))) (watch drinks
t)"
proof(coinduction)
  case alw
  then show ?case
    apply (case_tac "shd t")
    apply (simp del: ltl_step.simps)
    apply (rule P_ltl_step_0)
     apply simp
     apply (rule disjI2)
     apply (rule alw_mono[of "nxt (output_eq [])"])
      apply (simp add: no_output_none_nxt)
     apply simp
    apply simp
    using drink_costs_money_aux
    apply simp
    by blast
qed

lemma steps_1_invalid:
      "∄i. (a, b) = (STR ''coin'', [i]) ⟹
       ∄i. (a, b) = (STR ''vend'', []) ⟹
       possible_steps drinks 1 r a b = {||}"
  apply (simp add: possible_steps_empty drinks_def transitions can_take_transition_def can_take_def)
  by (induct b, auto)

lemma output_vend_aux:
  assumes "∃p r t. j = make_full_observation drinks (Some 1) r p t"
  shows "alw (λxs. label_eq ''vend'' xs ∧ output (shd (stl xs)) = [Some d] ⟶ check_exp (Ge (V (Rg 2))
(L (Num 100))) xs) j"
  using assms apply coinduct
  apply clarsimp
  subgoal for r t
    apply (case_tac "shd t")
    apply (simp add: implode_vend del: ltl_step.simps)
    apply (rule P_ltl_step_1)
       apply simp
       apply (rule disjI2)
       apply (rule alw_mono[of "nxt (output_eq [])"])
        apply (simp add: no_output_none_nxt)
       apply simp
      apply auto[1]
```

```
     apply auto[1]
    apply simp
    apply standard
    apply (rule disjI2)
    apply (rule alw_mono[of "nxt (output_eq [])"])
     apply (simp add: drinks_step_2_none no_output_none_if_empty nxt_alw)
    by simp
  done
lemma LTL_output_vend:
  "alw (((label_eq ''vend'') aand (nxt (output_eq [Some d]))) impl
        (check_exp (Ge (V (Rg 2)) (L (Num 100))))) (watch drinks t)"

proof(coinduction)
  case alw
  then show ?case
    apply (simp add: implode_vend)
    apply (case_tac "shd t")
    apply (simp del: ltl_step.simps)
    apply (rule P_ltl_step_0)
     apply simp
    apply (rule disjI2)
     apply (rule alw_mono[of "nxt (output_eq [])"])
      apply (simp add: no_output_none_nxt)
     apply simp
    apply simp
    subgoal for a b
      using output_vend_aux[of "(make_full_observation drinks (Some 1)
              <1 $:= Some (hd b), 2 $:= Some (Num 0)> [] (stl t))" d]
      using implode_vend by auto
    done
qed
lemma LTL_output_vend_unfolded:
  "alw (λxs. (label (shd xs) = STR ''vend'' ∧
             nxt (λs. output (shd s) = [Some d]) xs) ⟶
              ¬? value_gt (Some (Num 100)) (datastate (shd xs) $ 2) = trilean.true)
     (watch drinks t)"

  apply (insert LTL_output_vend[of d t])
  by (simp add: implode_vend)

end
```

Bibliography

[1] D. A. Bochvar. On a three-valued logical calculus and its application to the analysis of the paradoxes of the classical extended functional calculus. *History and Philosophy of Logic*, 2(1-2):87–112, jan 1981. ISSN 14645149. doi: 10.1080/01445348108837023. URL `http://www.tandfonline.com/doi/abs/10.1080/01445348108837023`.

[2] M. Foster, R. G. Taylor, A. D. Brucker, and J. Derrick. Formalising extended finite state machine transition merging. In J. S. Dong and J. Sun, editors, *ICFEM*, number 11232 in Lecture Notes in Computer Science, pages 373–387. Springer-Verlag, Heidelberg, 2018. ISBN 978-3-030-02449-9. doi: 10.1007/978-3-030-02450-5. URL `https://www.brucker.ch/bibliography/abstract/foster.ea-efsm-2018`.