A Formal Model of Extended Finite State Machines

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

September 19, 2020

*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):

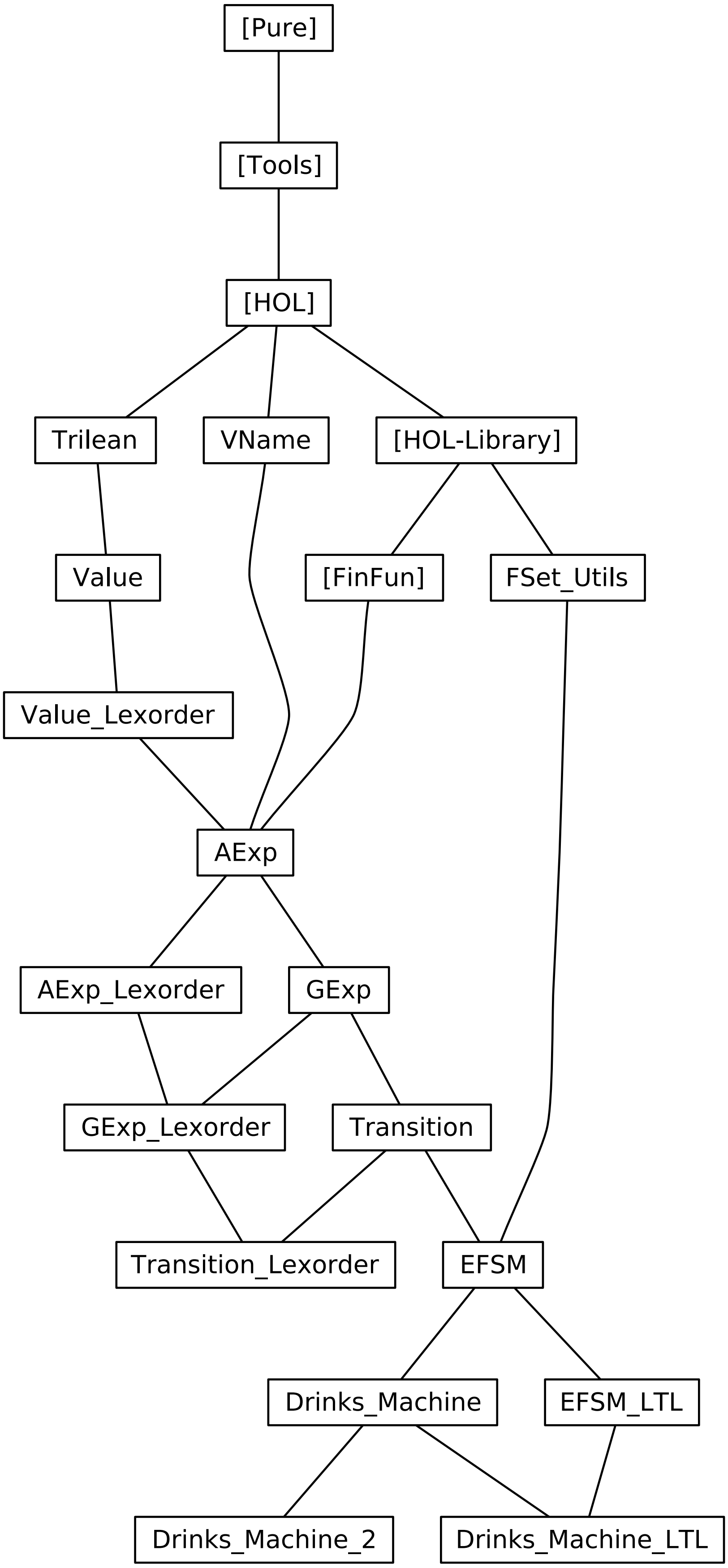

Figure 1.1: The Dependency Graph of the Isabelle Theories.

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⟩

lemma plus_trilean_commutative: "a ∨? b = b ∨? a"
⟨proof⟩

lemma times_trilean_commutative: "a ∧? b = b ∧? a"
  ⟨proof⟩

lemma times_trilean_assoc:
  "a ∧? b ∧? c = a ∧? (b ∧? c)"
⟨proof⟩
```

lemma *trilean_distributivity_1:*
  "(a ∨? b) ∧? c = a ∧? c ∨? b ∧? c"
⟨*proof*⟩

instance
  ⟨*proof*⟩
end

lemma *maybe_or_idempotent: "a ∨? a = a"*
  ⟨*proof*⟩

lemma *maybe_and_idempotent: "a ∧? a = a"*
  ⟨*proof*⟩

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
  ⟨*proof*⟩
end

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

instance
  ⟨*proof*⟩
end

lemma *maybe_and_one: "true ∧? x = x"*
  ⟨*proof*⟩

lemma *maybe_or_zero: "false ∨? x = x"*
  ⟨*proof*⟩

lemma *maybe_double_negation: "¬? ¬? x = x"*
  ⟨*proof*⟩

lemma *maybe_negate_true: "(¬? x = true) = (x = false)"*
  ⟨*proof*⟩

lemma *maybe_negate_false: "(¬? x = false) = (x = true)"*
  ⟨*proof*⟩

lemma *maybe_and_true: "(x ∧? y = true) = (x = true ∧ y = true)"*
  ⟨*proof*⟩

lemma *maybe_and_not_true:*
  "(x ∧? y ≠ true) = (x ≠ true ∨ y ≠ true)"
  ⟨*proof*⟩

lemma *negate_valid: "(¬? x ≠ invalid) = (x ≠ invalid)"*
  ⟨*proof*⟩

lemma *maybe_and_valid:*

"x ∧? y ≠ invalid ⟹ x ≠ invalid ∧ y ≠ invalid"
⟨proof⟩

lemma *maybe_or_valid:*
"x ∨? y ≠ invalid ⟹ x ≠ invalid ∧ y ≠ invalid"
⟨proof⟩

lemma *maybe_or_false:*
"(x ∨? y = false) = (x = false ∧ y = false)"
⟨proof⟩

lemma *maybe_or_true:*
"(x ∨? y = true) = ((x = true ∨ y = true) ∧ x ≠ invalid ∧ y ≠ invalid)"
⟨proof⟩

lemma *maybe_not_invalid:* "(¬? x = invalid) = (x = invalid)"
⟨proof⟩

lemma *maybe_or_invalid:*
"(x ∨? y = invalid) = (x = invalid ∨ y = invalid)"
⟨proof⟩

lemma *maybe_and_invalid:*
"(x ∧? y = invalid) = (x = invalid ∨ y = invalid)"
⟨proof⟩

lemma *maybe_and_false:*
"(x ∧? y = false) = ((x = false ∨ y = false) ∧ x ≠ invalid ∧ y ≠ invalid)"
⟨proof⟩

lemma *invalid_maybe_and:* "invalid ∧? x = invalid"
⟨proof⟩

lemma *maybe_not_eq:* "(¬? x = ¬? y) = (x = y)"
⟨proof⟩

lemma *de_morgans_1:*
"¬? (a ∨? b) = (¬?a) ∧? (¬?b)"
⟨proof⟩

lemma *de_morgans_2:*
"¬? (a ∧? b) = (¬?a) ∨? (¬?b)"
⟨proof⟩

lemma *not_true:* "(x ≠ true) = (x = false ∨ x = invalid)"
⟨proof⟩

lemma *pull_negation:* "(x = ¬? y) = (¬? x = y)"
⟨proof⟩

lemma *comp_fun_commute_maybe_or:* "comp_fun_commute maybe_or"
⟨proof⟩

lemma *comp_fun_commute_maybe_and:* "comp_fun_commute maybe_and"
⟨proof⟩

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'))"
  ⟨proof⟩

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)"
  ⟨proof⟩

lemma maybe_arith_int_None:
  "(maybe_arith_int f a1 a2 = None) = (∄n n'. a1 = Some (Num n) ∧ a2 = Some (Num n'))"
  ⟨proof⟩

lemma maybe_arith_int_Not_Num:
  "(∀n. maybe_arith_int f a1 a2 ≠ Some (Num n)) = (maybe_arith_int f a1 a2 = None)"
  ⟨proof⟩

lemma maybe_arith_int_never_string: "maybe_arith_int f a b ≠ Some (Str x)"
  ⟨proof⟩

definition "value_plus = maybe_arith_int (+)"

lemma value_plus_never_string: "value_plus a b ≠ Some (Str x)"
  ⟨proof⟩

lemma value_plus_symmetry: "value_plus x y = value_plus y x"
  ⟨proof⟩

definition "value_minus = maybe_arith_int (-)"

lemma value_minus_never_string: "value_minus a b ≠ Some (Str x)"
  ⟨proof⟩

definition "value_times = maybe_arith_int (*)"

lemma value_times_never_string: "value_times a b ≠ Some (Str x)"
  ⟨proof⟩

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"
  ⟨proof⟩

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)"
  ⟨proof⟩

lemma value_eq_false: "(value_eq a b = false) = (∃x y. a = Some x ∧ b = Some y ∧ x ≠ y)"
  ⟨proof⟩

lemma value_gt_true_Some: "value_gt a b = true ⟹ (∃x. a = Some x) ∧ (∃y. b = Some y)"
  ⟨proof⟩

lemma value_gt_true: "(value_gt a b = true) = (∃x y. a = Some (Num x) ∧ b = Some (Num y) ∧ x > y)"
  ⟨proof⟩

lemma value_gt_false_Some: "value_gt a b = false ⟹ (∃x. a = Some x) ∧ (∃y. b = Some y)"
  ⟨proof⟩

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
  ⟨proof⟩
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
  ⟨proof⟩

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"
  ⟨proof⟩

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"
  ⟨proof⟩
```

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)"
  ⟨proof⟩

lemma apply_empty_None [simp]: "<> $ x2 = None"
  ⟨proof⟩

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)"
  ⟨proof⟩

lemma input2state_out_of_bounds:
  "i ≥ length ia ⟹ input2state ia $ i = None"
⟨proof⟩

lemma input2state_within_bounds:
  "input2state i $ x = Some a ⟹ x < length i"
  ⟨proof⟩

lemma input2state_empty: "input2state [] $ x1 = None"
  ⟨proof⟩

lemma input2state_nth:
  "i < length ia ⟹ input2state ia $ i = Some (ia ! i)"
⟨proof⟩

lemma input2state_some:
  "i < length ia ⟹
   ia ! i = x ⟹
   input2state ia $ i = Some x"
  ⟨proof⟩

lemma input2state_take: "x1 < A ⟹
   A ≤ length i ⟹
   x = vname.I x1 ⟹
   input2state i $ x1 = input2state (take A i) $ x1"
⟨proof⟩

lemma input2state_not_None:
  "(input2state i $ x ≠ None) ⟹ (x < length i)"
  ⟨proof⟩

lemma input2state_Some:
  "(∃v. input2state i $ x = Some v) = (x < length i)"
```

⟨proof⟩

lemma *input2state_cons: "x1 > 0 ⟹
x1 < length ia ⟹
input2state (a # ia) $ x1 = input2state ia $ (x1-1)"*
⟨proof⟩

lemma *input2state_cons_shift:
"input2state i $ x1 = Some a ⟹ input2state (b # i) $ (Suc x1) = Some a"*
⟨proof⟩

lemma *input2state_exists: "∃ i. input2state i $ x1 = Some a"*
⟨proof⟩

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⟩

lemma *length_append_repeat: "length (i@(repeat a y)) ≥ length i"*
⟨proof⟩

lemma *length_input2state_repeat:
"input2state i $ x = Some a ⟹ y < length (i @ repeat y a)"*
⟨proof⟩

lemma *input2state_double_exists:
"∃ i. input2state i $ x = Some a ∧ input2state i $ y = Some a"*
⟨proof⟩

lemma *input2state_double_exists_2:
"x ≠ y ⟹ ∃ i. input2state i $ x = Some a ∧ input2state i $ y = Some a'"*
⟨proof⟩

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)"*
⟨proof⟩

lemma *join_ir_R [simp]: "(join_ir i r) (R n) = r $ n"*
⟨proof⟩

lemma *join_ir_double_exists:
"∃ i r. join_ir i r v = Some a ∧ join_ir i r v' = Some a"*
⟨proof⟩

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⟩

lemma *exists_join_ir_ext: "∃ i r. join_ir i r v = s v"*
⟨proof⟩

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

⟨proof⟩

```
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_constrain
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⟩

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)"
  ⟨proof⟩

lemma no_variables_aval: "enumerate_aexp_inputs a = {} ⟹
   enumerate_regs a = {} ⟹
   aval a s = aval a s'"
  ⟨proof⟩

lemma enumerate_aexp_inputs_not_empty:
  "(enumerate_aexp_inputs a ≠ {}) = (∃b c. enumerate_aexp_inputs a = set (b#c))"
  ⟨proof⟩

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⟩

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"*
⟨*proof*⟩

lemma *max_reg_V_R: "max_reg (V (R n)) = Some n"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *max_reg_Minus: "max_reg (Minus a1 a2) = max (max_reg a1) (max_reg a2)"*
⟨*proof*⟩

lemma *max_reg_Times: "max_reg (Times a1 a2) = max (max_reg a1) (max_reg a2)"*
⟨*proof*⟩

lemma *no_reg_aval_swap_regs:*
"max_reg a = None ⟹ aval a (join_ir i r) = aval a (join_ir i r')"
⟨*proof*⟩

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*⟩

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*⟩

lemma *enumerate_regs_empty_reg_unconstrained:*
"enumerate_regs a = {} ⟹ ∀r. ¬ aexp_constrains a (V (R r))"
⟨*proof*⟩

lemma *enumerate_aexp_inputs_empty_input_unconstrained:*
"enumerate_aexp_inputs a = {} ⟹ ∀r. ¬ aexp_constrains a (V (I r))"
⟨*proof*⟩

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)"
⟨*proof*⟩

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')"
⟨*proof*⟩

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'"
⟨*proof*⟩

lemma *max_input_I: "max_input (V (vname.I i)) = Some i"*
⟨*proof*⟩

lemma *max_input_Plus:*
"max_input (Plus a1 a2) = max (max_input a1) (max_input a2)"

⟨proof⟩

```
lemma max_input_Minus:
  "max_input (Minus a1 a2) = max (max_input a1) (max_input a2)"
  ⟨proof⟩

lemma max_input_Times:
  "max_input (Times a1 a2) = max (max_input a1) (max_input a2)"
  ⟨proof⟩

lemma aval_take:
  "max_input x < Some a ⟹
   aval x (join_ir i r) = aval x (join_ir (take a i) r)"
⟨proof⟩

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⟩

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))"
  ⟨proof⟩

lemma less_aexp_antisym: "(x::'a aexp) < y = (¬(y < x) ∧ (x ≠ y))"
  ⟨proof⟩

lemma less_aexp_aux_trans: "less_aexp_aux x y ⟹ less_aexp_aux y z ⟹ less_aexp_aux x z"
⟨proof⟩

lemma less_aexp_trans: "(x::'a aexp) < y ⟹ y < z ⟹ x < z"
  ⟨proof⟩

instance ⟨proof⟩
end

lemma smaller_height: "height a1 < height a2 ⟹ a1 < a2"
  ⟨proof⟩
```

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)"
  ⟨proof⟩

lemma gval_Le [simp]:
  "gval (Le a1 a2) s = ¬? (value_gt (aval a1 s) (aval a2 s))"
  ⟨proof⟩

lemma gval_Ge [simp]:
  "gval (Ge a1 a2) s = ¬? (value_gt (aval a2 s) (aval a1 s))"
  ⟨proof⟩

lemma gval_Ne [simp]:
  "gval (Ne a1 a2) s = ¬? (value_eq (aval a1 s) (aval a2 s))"
  ⟨proof⟩
```

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)"*
⟨*proof*⟩

lemma *gval_gNot [simp]: "gval (gNot x) s = ¬? (gval x s)"*
⟨*proof*⟩

lemma *gval_gAnd [simp]:*
"gval (gAnd g1 g2) s = (gval g1 s) ∧? (gval g2 s)"
⟨*proof*⟩

lemma *gAnd_commute: "gval (gAnd a b) s = gval (gAnd b a) s"*
⟨*proof*⟩

lemma *gOr_commute: "gval (gOr a b) s = gval (gOr b a) s"*
⟨*proof*⟩

lemma *gval_gAnd_True:*
"(gval (gAnd g1 g2) s = true) = ((gval g1 s = true) ∧ gval g2 s = true)"
⟨*proof*⟩

lemma *nor_equiv: "gval (gNot (gOr a b)) s = gval (Nor a b) s"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *satisfiable_true: "satisfiable (Bc True)"*
⟨*proof*⟩

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

lemma *valid_true: "valid (Bc True)"*
⟨*proof*⟩

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*⟩

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*⟩

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)"
```
  ⟨*proof*⟩

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)"*
  ⟨*proof*⟩

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"*
  ⟨*proof*⟩

lemma *max_reg_Eq: "max_reg (Eq a b) = max (AExp.max_reg a) (AExp.max_reg b)"*
  ⟨*proof*⟩

lemma *max_reg_Gt: "max_reg (Gt a b) = max (AExp.max_reg a) (AExp.max_reg b)"*
  ⟨*proof*⟩

lemma *max_reg_Nor: "max_reg (Nor a b) = max (max_reg a) (max_reg b)"*
  ⟨*proof*⟩

lemma *gval_In_cons:*
```
  "gval (In v (a # as)) s = (gval (Eq (V v) (L a)) s ∨? gval (In v as) s)"
```
  ⟨*proof*⟩

lemma *possible_to_be_in: "s ≠ [] ⟹ satisfiable (In v s)"*
⟨*proof*⟩

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)"
  ⟨proof⟩
```

lemma *max_reg_list_append_singleton:*

```
  "max_reg_list (as@[bs]) = max (max_reg_list as) (max_reg_list [bs])"
  ⟨proof⟩
```

lemma *max_reg_list_append:*

```
  "max_reg_list (as@bs) = max (max_reg_list as) (max_reg_list bs)"
⟨proof⟩
```

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)"*

```
  ⟨proof⟩
```

lemma *apply_guards_empty [simp]: "apply_guards [] s"*

```
  ⟨proof⟩
```

lemma *apply_guards_cons:*

```
  "apply_guards (a # G) c = (gval a c = true ∧ apply_guards G c)"
  ⟨proof⟩
```

lemma *apply_guards_double_cons:*

```
  "apply_guards (y # x # G) s = (gval (gAnd y x) s = true ∧ apply_guards G s)"
  ⟨proof⟩
```

lemma *apply_guards_append:*

```
  "apply_guards (a@a') s = (apply_guards a s ∧ apply_guards a' s)"
  ⟨proof⟩
```

lemma *apply_guards_foldr:*

```
  "apply_guards G s = (gval (foldr gAnd G (Bc True)) s = true)"
⟨proof⟩
```

lemma *rev_apply_guards: "apply_guards (rev G) s = apply_guards G s"*

```
  ⟨proof⟩
```

lemma *apply_guards_fold:*

```
  "apply_guards G s = (gval (fold gAnd G (Bc True)) s = true)"
  ⟨proof⟩
```

lemma *fold_apply_guards:*

```
  "(gval (fold gAnd G (Bc True)) s = true) = apply_guards G s"
  ⟨proof⟩
```

lemma *foldr_apply_guards:*

```
  "(gval (foldr gAnd G (Bc True)) s = true) = apply_guards G s"
  ⟨proof⟩
```

lemma *apply_guards_subset:*

```
  "set g' ⊆ set g ⟹ apply_guards g c ⟶ apply_guards g' c"
⟨proof⟩
```

lemma *apply_guards_subset_append:*

```
  "set G ⊆ set G' ⟹ apply_guards (G @ G') s = apply_guards (G') s"
  ⟨proof⟩
```

lemma *apply_guards_rearrange:*

```
  "x ∈ set G ⟹ apply_guards G s = apply_guards (x#G) s"
  ⟨proof⟩
```

lemma *apply_guards_condense: "∃g. apply_guards G s = (gval g s = true)"*
⟨*proof*⟩

lemma *apply_guards_false_condense: "∃g. (¬apply_guards G s) = (gval g s = false)"*
⟨*proof*⟩

lemma *max_input_Bc: "max_input (Bc x) = None"*
⟨*proof*⟩

lemma *max_input_Eq:*
"max_input (Eq a1 a2) = max (AExp.max_input a1) (AExp.max_input a2)"
⟨*proof*⟩

lemma *max_input_Gt:*
"max_input (Gt a1 a2) = max (AExp.max_input a1) (AExp.max_input a2)"
⟨*proof*⟩

lemma *gexp_max_input_Nor:*
"max_input (Nor g1 g2) = max (max_input g1) (max_input g2)"
⟨*proof*⟩

lemma *gexp_max_input_In: "max_input (In v l) = AExp.max_input (V v)"*
⟨*proof*⟩

lemma *gval_foldr_gOr_invalid:*
"(gval (fold gOr l g) s = invalid) = (∃g' ∈ (set (g#l)). gval g' s = invalid)"
⟨*proof*⟩

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*⟩

lemma *gval_foldr_gOr_false:*
"(gval (fold gOr l g) s = false) = (∀g' ∈ (set (g#l)). gval g' s = false)"
⟨*proof*⟩

lemma *gval_fold_gOr_rev: "gval (fold gOr (rev l) g) s = gval (fold gOr l g) s"*
⟨*proof*⟩

lemma *gval_fold_gOr_foldr: "gval (fold gOr l g) s = gval (foldr gOr l g) s"*
⟨*proof*⟩

lemma *gval_fold_gOr:*
"gval (fold gOr (a # l) g) s = (gval a s ∨? gval (fold gOr l g) s)"
⟨*proof*⟩

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*⟩

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*⟩

lemma *fold_maybe_or_invalid_base: "fold (∨?) l invalid = invalid"*
⟨*proof*⟩

lemma *fold_maybe_or_true_base_never_false:*

```
  "fold (∨?) l true ≠ false"
⟨proof⟩

lemma fold_true_fold_false_not_invalid:
  "fold (∨?) l true = true ⟹
   fold (∨?) (rev l) false ≠ invalid"
⟨proof⟩

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

lemma fold_maybe_or_rev:
  "fold (∨?) l b = fold (∨?) (rev l) b"
⟨proof⟩

lemma fold_maybe_or_cons:
  "fold (∨?) (a#l) b = a ∨? (fold (∨?) l b)"
  ⟨proof⟩

lemma gval_fold_gOr_map:
  "gval (fold gOr l (Bc False)) s = fold (∨?) (map (λg. gval g s) l) (false)"
⟨proof⟩

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⟩

lemma fold_Eq_true:
  "∀v. fold (∨?) (map (λx. if v = x then true else false) vs) true = true"
  ⟨proof⟩

lemma x_in_set_fold_eq:
  "x ∈ set ll ⟹
   fold (∨?) (map (λxa. if x = xa then true else false) ll) false = true"
⟨proof⟩

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"
  ⟨proof⟩

lemma gval_take: "max_input g < Some a ⟹
   gval g (join_ir i r) = gval g (join_ir (take a i) r)"
⟨proof⟩

lemma gval_fold_gAnd_append_singleton:
  "gval (fold gAnd (a @ [G]) (Bc True)) s = gval (fold gAnd a (Bc True)) s ∧? gval G s"
  ⟨proof⟩

lemma gval_fold_rev_true:
  "gval (fold gAnd (rev G) (Bc True)) s = true ⟹
   gval (fold gAnd G (Bc True)) s = true"
  ⟨proof⟩

lemma gval_fold_not_invalid_all_valid_contra:
  "∃g ∈ set G. gval g s = invalid ⟹
   gval (fold gAnd G (Bc True)) s = invalid"
⟨proof⟩

lemma gval_fold_not_invalid_all_valid:
```

```
"gval (fold gAnd G (Bc True)) s ≠ invalid ⟹
 ∀g ∈ set G. gval g s ≠ invalid"
⟨proof⟩
```

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)"
⟨proof⟩
```

lemma *must_have_one_false_contra:*
```
"∀g ∈ set G. gval g s ≠ false ⟹
 gval (fold gAnd G (Bc True)) s ≠ false"
⟨proof⟩
```

lemma *must_have_one_false:*
```
"gval (fold gAnd G (Bc True)) s = false ⟹
 ∃g ∈ set G. gval g s = false"
⟨proof⟩
```

lemma *all_valid_fold:*
```
"∀g ∈ set G. gval g s ≠ invalid ⟹
 gval (fold gAnd G (Bc True)) s ≠ invalid"
⟨proof⟩
```

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"
⟨proof⟩
```

lemma *gval_fold_rev_false:*
```
"gval (fold gAnd (rev G) (Bc True)) s = false ⟹
 gval (fold gAnd G (Bc True)) s = false"
⟨proof⟩
```

lemma *fold_invalid_means_one_invalid:*
```
"gval (fold gAnd G (Bc True)) s = invalid ⟹
 ∃g ∈ set G. gval g s = invalid"
⟨proof⟩
```

lemma *gval_fold_rev_invalid:*
```
"gval (fold gAnd (rev G) (Bc True)) s = invalid ⟹
 gval (fold gAnd G (Bc True)) s = invalid"
⟨proof⟩
```

lemma *gval_fold_rev_equiv_fold:*
```
"gval (fold gAnd (rev G) (Bc True)) s =  gval (fold gAnd G (Bc True)) s"
⟨proof⟩
```

lemma *gval_fold_equiv_fold_rev:*
```
"gval (fold gAnd G (Bc True)) s = gval (fold gAnd (rev G) (Bc True)) s"
⟨proof⟩
```

lemma *gval_fold_equiv_gval_foldr:*
```
"gval (fold gAnd G (Bc True)) s = gval (foldr gAnd G (Bc True)) s"
⟨proof⟩
```

lemma *gval_foldr_equiv_gval_fold:*
```
"gval (foldr gAnd G (Bc True)) s = gval (fold gAnd G (Bc True)) s"
⟨proof⟩
```

lemma *gval_fold_cons:*
```
"gval (fold gAnd (g # gs) (Bc True)) s = gval g s ∧? gval (fold gAnd gs (Bc True)) s"
⟨proof⟩
```

```
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⟩

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"
  ⟨proof⟩

lemma length_take_or_pad: "length (take_or_pad a n) = n"
⟨proof⟩

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))"
  ⟨proof⟩

lemma not_restricted_cons:
  "not_restricted v (g#gs) = ((¬ gexp_constrains g (V v)) ∧ not_restricted v gs)"
  ⟨proof⟩

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⟩
```

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')"
  ⟨proof⟩
```

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⟩

lemma less_gexp_antisym: "(x::'a gexp) < y = (¬(y < x) ∧ (x ≠ y))"
  ⟨proof⟩

lemma less_gexp_aux_trans: "less_gexp_aux x y ⟹ less_gexp_aux y z ⟹ less_gexp_aux x z"
⟨proof⟩

lemma less_gexp_trans: "(x::'a gexp) < y ⟹ y < z ⟹ x < z"
  ⟨proof⟩

instance ⟨proof⟩
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"*
⟨*proof*⟩

definition *"fSum ≡ fsum (λx. x)"*

lemma *fset_both_sides: "(Abs_fset s = f) = (fset (Abs_fset s) = fset f)"*
⟨*proof*⟩

lemma *Abs_ffilter: "(ffilter f s = s') = ({e ∈ (fset s). f e} = (fset s'))"*
⟨*proof*⟩

lemma *size_ffilter_card: "size (ffilter f s) = card ({e ∈ (fset s). f e})"*
⟨*proof*⟩

lemma *ffilter_empty [simp]: "ffilter f {||} = {||}"*
⟨*proof*⟩

lemma *ffilter_finsert:*
"ffilter f (finsert a s) = (if f a then finsert a (ffilter f s) else (ffilter f s))"
⟨*proof*⟩

lemma *fset_equiv: "(f1 = f2) = (fset f1 = fset f2)"*
⟨*proof*⟩

lemma *finsert_equiv: "(finsert e f = f') = (insert e (fset f) = (fset f'))"*
⟨*proof*⟩

lemma *filter_elements:*
"x |∈| Abs_fset (Set.filter f (fset s)) = (x ∈ (Set.filter f (fset s)))"
⟨*proof*⟩

lemma *sorted_list_of_fempty [simp]: "sorted_list_of_fset {||} = []"*
⟨*proof*⟩

lemma *fmember_implies_member: "e |∈| f ⟹ e ∈ fset f"*
⟨*proof*⟩

lemma *fold_union_ffUnion: "fold (|∪|) l {||} = ffUnion (fset_of_list l)"*
⟨*proof*⟩

lemma *filter_filter:*
"ffilter P (ffilter Q xs) = ffilter (λx. Q x ∧ P x) xs"
⟨*proof*⟩

lemma *fsubset_strict:*
"x2 |⊂| x1 ⟹ ∃e. e |∈| x1 ∧ e |∉| x2"
⟨*proof*⟩

lemma *fsubset:*
"x2 |⊂| x1 ⟹ ∄e. e |∈| x2 ∧ e |∉| x1"
⟨*proof*⟩

lemma *size_fsubset_elem:*
assumes *"∃e. e |∈| x1 ∧ e |∉| x2"*
and *"∄e. e |∈| x2 ∧ e |∉| x1"*
shows *"size x2 < size x1"*
⟨*proof*⟩

lemma *size_fsubset: "x2 |⊂| x1 ⟹ size x2 < size x1"*
⟨*proof*⟩

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"
  ⟨proof⟩

lemma ffilter_singleton: "f e ⟹ ffilter f {|e|} = {|e|}"
  ⟨proof⟩

lemma fset_eq_alt: "(x = y) = (x |⊆| y ∧ size x = size y)"
  ⟨proof⟩

lemma ffold_empty [simp]: "ffold f b {||} = b"
  ⟨proof⟩

lemma sorted_list_of_fset_sort:
  "sorted_list_of_fset (fset_of_list l) = sort (remdups l)"
  ⟨proof⟩

lemma fMin_Min: "fMin (fset_of_list l) = Min (set l)"
  ⟨proof⟩

lemma sorted_hd_Min:
  "sorted l ⟹
   l ≠ [] ⟹
   hd l = Min (set l)"
  ⟨proof⟩

lemma hd_sort_Min: "l ≠ [] ⟹ hd (sort l) = Min (set l)"
  ⟨proof⟩

lemma hd_sort_remdups: "hd (sort (remdups l)) = hd (sort l)"
  ⟨proof⟩

lemma exists_fset_of_list: "∃l. f = fset_of_list l"
  ⟨proof⟩

lemma hd_sorted_list_of_fset:
  "s ≠ {||} ⟹ hd (sorted_list_of_fset s) = (fMin s)"
  ⟨proof⟩

lemma fminus_filter_singleton:
  "fset_of_list l |-| {|x|} = fset_of_list (filter (λe. e ≠ x) l)"
  ⟨proof⟩

lemma card_minus_fMin:
  "s ≠ {||} ⟹ card (fset s - {fMin s}) < card (fset s)"
  ⟨proof⟩


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)
  )"
  ⟨proof⟩
termination
```

⟨proof⟩

lemma *sorted_list_of_fset_Cons:*
"∃h t. (sorted_list_of_fset (finsert s ss)) = h#t"
⟨proof⟩

lemma *list_eq_hd_tl:*
"l ≠ [] ⟹
hd l = h ⟹
tl l = t ⟹
l = (h#t)"
⟨proof⟩

lemma *fset_of_list_sort:* "fset_of_list l = fset_of_list (sort l)"
⟨proof⟩

lemma *exists_sorted_distinct_fset_of_list:*
"∃l. sorted l ∧ distinct l ∧ f = fset_of_list l"
⟨proof⟩

lemma *fset_of_list_empty [simp]:* "(fset_of_list l = {||}) = (l = [])"
⟨proof⟩

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⟩

lemma *sorted_distinct_ffold_ord:* **assumes** "sorted l"
and "distinct l"
shows "ffold_ord f (fset_of_list l) b = fold f l b"
⟨proof⟩

lemma *ffold_ord_fold_sorted:* "ffold_ord f s b = fold f (sorted_list_of_fset s) b"
⟨proof⟩

context includes *fset.lifting* **begin**
lift_definition *fprod* :: "'a fset ⇒ 'b fset ⇒ ('a × 'b) fset " (**infixr** "|×|" 80) **is** "λa b. fset a × fset b"
⟨proof⟩

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

lemma *fprod_empty_l:* "{||} |×| a = {||}"
⟨proof⟩

lemma *fprod_empty_r:* "a |×| {||} = {||}"
⟨proof⟩

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))"
⟨proof⟩

lemma *fprod_member:*
"x |∈| xs ⟹
y |∈| ys ⟹
(x, y) |∈| xs |×| ys"
⟨proof⟩

lemma *fprod_subseteq:*

"x |⊆| x' ∧ y |⊆| y' ⟹ x |×| y |⊆| x' |×| y'"
⟨proof⟩

lemma *fimage_fprod:*
"(a, b) |∈| A |×| B ⟹ f a b |∈| (λ(x, y). f x y) |`| (A |×| B)"
⟨proof⟩

lemma *fprod_singletons:* "{|a|} |×| {|b|} = {|(a, b)|}"
⟨proof⟩

lemma *fprod_equiv:*
"(fset (f |×| f') = s) = (((fset f) × (fset f')) = s)"
⟨proof⟩

lemma *fis_singleton_alt:* "fis_singleton f = (∃ e. f = {|e|})"
⟨proof⟩

lemma *singleton_singleton [simp]:* "fis_singleton {|a|}"
⟨proof⟩

lemma *not_singleton_empty [simp]:* "¬ fis_singleton {||}"
⟨proof⟩

lemma *fis_singleton_fthe_elem:*
"fis_singleton A ⟷ A = {|fthe_elem A|}"
⟨proof⟩

lemma *fBall_ffilter:*
"∀ x |∈| X. f x ⟹ ffilter f X = X"
⟨proof⟩

lemma *fBall_ffilter2:*
"X = Y ⟹
∀ x |∈| X. f x ⟹
ffilter f X = Y"
⟨proof⟩

lemma *size_fset_of_list:* "size (fset_of_list l) = length (remdups l)"
⟨proof⟩

lemma *size_fsingleton:* "(size f = 1) = (∃ e. f = {|e|})"
⟨proof⟩

lemma *ffilter_mono:* "(ffilter X xs = f) ⟹ ∀ x |∈| xs. X x = Y x ⟹ (ffilter Y xs = f)"
⟨proof⟩

lemma *size_fimage:* "size (fimage f s) ≤ size s"
⟨proof⟩

lemma *size_ffilter:* "size (ffilter P f) ≤ size f"
⟨proof⟩

lemma *fimage_size_le:* "⋀f s. size s ≤ n ⟹ size (fimage f s) ≤ n"
⟨proof⟩

lemma *ffilter_size_le:* "⋀f s. size s ≤ n ⟹ size (ffilter f s) ≤ n"
⟨proof⟩

lemma *set_membership_eq:* "A = B ⟷ (λx. Set.member x A) = (λx. Set.member x B)"
⟨proof⟩

lemmas *ffilter_eq_iff = Abs_ffilter set_membership_eq fun_eq_iff*

lemma *size_le_1:* "size f $\leq$ 1 = (f = {||} $\vee$ ($\exists$ e. f = {|e|}))"
⟨*proof*⟩

lemma *size_gt_1:* "1 < size f $\Longrightarrow$ $\exists$ e1 e2 f'. e1 $\neq$ e2 $\wedge$ f = finsert e1 (finsert e2 f')"
⟨*proof*⟩

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"
  ⟨proof⟩

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"
  ⟨proof⟩

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"
  ⟨proof⟩

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"
  ⟨proof⟩

lemma valid_list_can_take:
  "∀g ∈ set (Guards t). valid g ⟹ ∃i. can_take_transition t i c"
  ⟨proof⟩

lemma cant_take_if:
  "∃g ∈ set (Guards t). gval g (join_ir i r) ≠ true ⟹
  ¬ can_take_transition t i r"
  ⟨proof⟩

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"
  ⟨proof⟩

lemmas apply_outputs = datastate apply_outputs_def value_plus_def value_minus_def value_times_def

lemma apply_outputs_empty [simp]: "apply_outputs [] s = []"
  ⟨proof⟩

lemma apply_outputs_preserves_length: "length (apply_outputs p s) = length p"
```

⟨proof⟩

lemma *apply_outputs_literal:* **assumes** "P ! r = L v"
and "r < length P"
shows "apply_outputs P s ! r = Some v"
⟨proof⟩

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"
⟨proof⟩

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"
⟨proof⟩

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⟩

lemma *update_twice:*
"apply_updates [(r, a), (r, b)] s regs = regs (r $:= aval b s)"
⟨proof⟩

lemma *r_not_updated_stays_the_same:*
"r ∉ fst ‘ set U ⟹ apply_updates U c d $ r = d $ r"
⟨proof⟩

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
  ⟨proof⟩
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"
  ⟨proof⟩

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)"
```

⟨proof⟩

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
)"
```
⟨proof⟩

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)"
⟨proof⟩

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)"
⟨proof⟩

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 = {||}"
⟨proof⟩

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)})"*
⟨proof⟩

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⟩

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)|})"
```
⟨proof⟩

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)|})"
```
⟨proof⟩

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)|})"
```
⟨proof⟩

```
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)|}))"
  ⟨proof⟩

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)|}"
  ⟨proof⟩

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)})"
  ⟨proof⟩

lemma possible_steps_apply_guards:
  "possible_steps e s r l i = {|(s', t)|} ⟹
   apply_guards (Guards t) (join_ir i r)"
  ⟨proof⟩

lemma possible_steps_empty:
  "(possible_steps e s r l i = {||}) = (∀ ((origin, dest), t) ∈ fset e. origin ≠ s ∨ Label t ≠ l ∨ ¬
can_take_transition t i r)"
  ⟨proof⟩

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"
  ⟨proof⟩

lemma no_outgoing_transitions:
"ffilter (λ((s', _), _). s = s') e = {||} ⟹
possible_steps e s r l i = {||}"
  ⟨proof⟩

lemma ffilter_split: "ffilter (λ((origin, dest), t). origin = s ∧ Label t = l ∧ length i = Arity t ∧
apply_guards (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)"
  ⟨proof⟩

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⟩
```

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'"
  ⟨proof⟩
```

lemma *choice_symmetry: "choice x y = choice y x"*
⟨*proof*⟩

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)|}))"
⟨*proof*⟩

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)|})
)"
⟨*proof*⟩

lemma *size_le_1: "size f ≤ 1 = (f = {||} ∨ (∃ e. f = {|e|}))"*
⟨*proof*⟩

lemma *ffilter_empty_if: "∀ x |∈| xs. ¬ P x ⟹ ffilter P xs = {||}"*
⟨*proof*⟩

lemma *empty_ffilter: "ffilter P xs = {||} = (∀ x |∈| xs. ¬ P x)"*
⟨*proof*⟩

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"
⟨*proof*⟩

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)"
⟨*proof*⟩

lemma *ffilter_fBall: "(∀ x |∈| xs. P x) = (ffilter P xs = xs)"*
⟨*proof*⟩

lemma *fsubset_if: "∀ x. x |∈| f1 ⟶ x |∈| f2 ⟹ f1 |⊆| f2"*
⟨*proof*⟩

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)"*
⟨*proof*⟩

lemma *possible_steps_can_take_transition:*
"(s2, t1) |∈| possible_steps e1 s1 r l i ⟹ can_take_transition t1 i r"
⟨*proof*⟩

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"
  ⟨proof⟩

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"
  ⟨proof⟩

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"
  ⟨proof⟩

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"
  ⟨proof⟩

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)"
  ⟨proof⟩

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"
  ⟨proof⟩

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"
  ⟨proof⟩

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"
```
⟨proof⟩

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)"
```
⟨proof⟩

```
lemma random_member_singleton [simp]: "random_member {|a|} = Some a"
```
⟨proof⟩

```
lemma random_member_is_member:
  "random_member ss = Some s ⟹ s |∈| ss"
```
⟨proof⟩

```
lemma random_member_None[simp]: "random_member ss = None = (ss = {||})"
```
⟨proof⟩

```
lemma random_member_empty[simp]: "random_member {||} = None"
```
⟨proof⟩

```
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"
```
⟨proof⟩

```
lemma step_member: "step e s r l i = Some (t, s', p, r') ⟹ (s', t) |∈| possible_steps e s r l i"
```
⟨proof⟩

```
lemma step_outputs: "step e s r l i = Some (t, s', p, r') ⟹ evaluate_outputs t i r = p"
```
⟨proof⟩

```
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')"
```
⟨proof⟩

```
lemma step_None: "step e s r l i = None = (possible_steps e s r l i = {||})"
```
⟨proof⟩

```
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'
  )"
```
⟨proof⟩

```
lemma no_possible_steps_1:
  "possible_steps e s r l i = {||} ⟹ step e s r l i = None"
```
⟨proof⟩

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)
    )"
  ⟨proof⟩

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"
  ⟨proof⟩

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"
  ⟨proof⟩

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"
  ⟨proof⟩

lemma observe_execution_no_possible_step:
  "possible_steps e s r (fst h) (snd h) = {||} ⟹
   observe_execution e s r (h#es) = []"
  ⟨proof⟩

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))"
  ⟨proof⟩

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))"
```

⟨*proof*⟩

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)"
```

⟨*proof*⟩

```
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)"
```

⟨*proof*⟩

```
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)"
```

⟨*proof*⟩

```
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*⟩

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)"*
  ⟨*proof*⟩

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)"*
  ⟨*proof*⟩

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)"
  ⟨*proof*⟩

lemma *recognises_possible_steps_not_empty:*
  "recognises_execution e s r (h#t) ⟹ possible_steps e s r (fst h) (snd h) ≠ {||}"
  ⟨*proof*⟩

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')"
  ⟨*proof*⟩

lemma *recognises_cons_step:*
  "recognises_execution e s r (h # t) ⟹ step e s r (fst h) (snd h) ≠ None"
  ⟨*proof*⟩

lemma *no_step_none:*
  "step e s r aa ba = None ⟹ ¬ recognises_execution e s r ((aa, ba) # p)"
  ⟨*proof*⟩

lemma *step_none_rejects:*
  "step e s r (fst h) (snd h) = None ⟹ ¬ recognises_execution e s r (h#t)"
  ⟨*proof*⟩

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))"
  ⟨*proof*⟩

lemma *trace_reject_no_possible_steps_atomic:*
  "possible_steps e s r (fst a) (snd a) = {||} ⟹ ¬ recognises_execution e s r (a#t)"
  ⟨*proof*⟩

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)"
  ⟨*proof*⟩

lemma *recognition_prefix_closure: "recognises_execution e s r (t@t') ⟹ recognises_execution e s r t"*
⟨*proof*⟩

lemma *rejects_prefix: "¬ recognises_execution e s r t ⟹ ¬ recognises_execution e s r (t @ t')"*
  ⟨*proof*⟩

lemma *recognises_head: "recognises_execution e s r (h#t) ⟹ recognises_execution e s r [h]"*
  ⟨*proof*⟩

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)"
  ⟨proof⟩

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"
  ⟨proof⟩

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_up
T i r) t)
)"
  ⟨proof⟩

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⟩
```

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⟩
```

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))
)"
  ⟨proof⟩

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)"
  ⟨proof⟩

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⟩

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"
  ⟨proof⟩
```

lemma *simulates_trace_subset: "trace_simulates e1 e2 ⟹ T e1 ⊆ T e2"*
⟨*proof*⟩

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"

⟨*proof*⟩

lemma *trace_equivalent_reflexive: "trace_equivalent e1 e1"*
⟨*proof*⟩

lemma *trace_equivalent_symmetric:*
"trace_equivalent e1 e2 = trace_equivalent e2 e1"
⟨*proof*⟩

lemma *trace_equivalent_transitive:*
```
  "trace_equivalent e1 e2 ⟹
   trace_equivalent e2 e3 ⟹
   trace_equivalent e1 e3"
  ⟨proof⟩
```

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"
⟨*proof*⟩

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)"
  ⟨proof⟩
```

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))
)"
```
⟨*proof*⟩

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*⟩

lemma *execution_simulates_trace_simulates:*
```
"execution_simulates e1 e2 ⟹ trace_simulates e1 e2"
```
⟨*proof*⟩

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)))"
```
⟨*proof*⟩

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)"
```
⟨*proof*⟩

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)"
```

⟨proof⟩

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⟩

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"
```
⟨proof⟩

lemma *executionally_equivalent_trace_equivalent:*
```
"∀x. executionally_equivalent e1 0 <> e2 0 <> x ⟹ trace_equivalent e1 e2"
```
⟨proof⟩

lemma *executionally_equivalent_symmetry:*
```
"executionally_equivalent e1 s1 r1 e2 s2 r2 x ⟹
 executionally_equivalent e2 s2 r2 e1 s1 r1 x"
```
⟨proof⟩

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⟩

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 []"
```
⟨proof⟩

lemma *visits_base:* `"visits target e s r [] = (s = target)"`
⟨proof⟩

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)"
```
⟨proof⟩

lemma *reachable_initial:* `"reachable 0 e"`
⟨proof⟩

lemma *visits_finsert:*
```
"visits s e s' r t ⟹ visits s (finsert ((aa, ba), b) e) s' r t"
```
⟨proof⟩

lemma *reachable_finsert:*
```
"reachable s e ⟹ reachable s (finsert ((aa, ba), b) e)"
```
⟨proof⟩

lemma *reachable_finsert_contra:*
"¬ reachable s (finsert ((aa, ba), b) e) ⟹ ¬reachable s e"
⟨proof⟩

lemma *visits_empty:* "visits s e s' r [] = (s = s')"
⟨proof⟩

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"
⟨proof⟩

lemma *obtains_base:* "obtains s r e s' r' [] = (s = s' ∧ r = r')"
⟨proof⟩

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)"
⟨proof⟩

lemma *obtains_recognises:*
"obtains s c e s' r t ⟹ recognises_execution e s' r t"
⟨proof⟩

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)"
⟨proof⟩

lemma *recognises_execution_obtains:*
"recognises_execution e s' r t ⟹ ∃c1 s. obtains s c1 e s' r t"
⟨proof⟩

lemma *obtainable_empty_efsm:*
"obtainable s c {||} = (s=0 ∧ c = <>)"
⟨proof⟩

lemma *obtains_visits:* "obtains s r e s' r' t ⟹ visits s e s' r' t"
⟨proof⟩

lemma *unobtainable_if:* "¬ visits s e s' r' t ⟹ ¬ obtains s r e s' r' t"
⟨proof⟩

lemma *obtainable_if_unreachable:* "¬reachable s e ⟹ ¬obtainable s r e"
⟨proof⟩

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⟩

lemma *reachable_if_obtainable_step:*
"obtainable s r e ⟹ ∃l i t. (s', t) |∈| possible_steps e s r l i ⟹ reachable s' e"
⟨proof⟩

```
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"
  ⟨proof⟩
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⟩
lemma executionally_equivalent_remove_unreachable_state:
  "¬ reachable s' e ⟹ executionally_equivalent e 0 <> (remove_state s' e) 0 <> x"

  ⟨proof⟩
```

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)))"
  ⟨proof⟩

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"
  ⟨proof⟩

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)"
⟨*proof*⟩

lemma *ltl_step_none: "possible_steps e s r a b = {||} ⟹ ltl_step e (Some s) r (a, b) = (None, [], r)"*
⟨*proof*⟩

lemma *ltl_step_none_2: "possible_steps e s r (fst ie) (snd ie) = {||} ⟹ ltl_step e (Some s) r ie = (None, [], r)"*
⟨*proof*⟩

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))
)"
⟨*proof*⟩

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')"*
⟨*proof*⟩

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))"*
⟨*proof*⟩

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)"
  ⟨proof⟩
```

```
lemma state_eq_None_not_Some: "state_eq None s ⟹ ¬ state_eq (Some n) s"
  ⟨proof⟩
```

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)"
  ⟨proof⟩
```

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"*
  ⟨*proof*⟩

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))"*
  ⟨*proof*⟩

lemma *alw_state_eq_smap:*
  "alw (state_eq s) ss = alw (λss. shd ss = s) (smap statename ss)"
  ⟨*proof*⟩

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)"*
  ⟨*proof*⟩

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)))"*
  ⟨*proof*⟩

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"*
  ⟨*proof*⟩

lemma *once_none_always_none: "alw (state_eq None) (make_full_observation e None r p t)"*
  ⟨*proof*⟩

lemma *once_none_nxt_always_none: "alw (nxt (state_eq None)) (make_full_observation e None r p t)"*
  ⟨*proof*⟩

lemma *snth_sconst: "(∀i. s !! i = h) = (s = sconst h)"*
  ⟨*proof*⟩

lemma *alw_sconst: "(alw (λxs. shd xs = h) t) = (t = sconst h)"*
  ⟨*proof*⟩

lemma *smap_statename_None: "smap statename (make_full_observation e None r p i) = sconst None"*
  ⟨*proof*⟩

lemma *alw_not_some: "alw (λxs. statename (shd xs) ≠ Some s) (make_full_observation e None r p t)"*

⟨proof⟩

lemma *state_none:* "((state_eq None) impl nxt (state_eq None)) (make_full_observation e s r p t)"
⟨proof⟩

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))"
⟨proof⟩

lemma *no_output_none_aux:*
assumes "∃ p r i. j = (make_full_observation e None r []) i"
shows "alw (output_eq []) j"
⟨proof⟩

lemma *no_output_none:* "nxt (alw (output_eq [])) (make_full_observation e None r p t)"
⟨proof⟩

lemma *nxt_alw:* "nxt (alw P) s ⟹ alw (nxt P) s"
⟨proof⟩

lemma *no_output_none_nxt:* "alw (nxt (output_eq [])) (make_full_observation e None r p t)"
⟨proof⟩

lemma *no_output_none_if_empty:* "alw (output_eq []) (make_full_observation e None r [] t)"
⟨proof⟩

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"
⟨proof⟩

lemma *no_updates_none:* "alw (λx. datastate (shd x) = r) (make_full_observation e None r p t)"
⟨proof⟩

lemma *action_components:* "(label_eq l aand input_eq i) s = (action (shd s) = (String.implode l, i))"
⟨proof⟩

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"*
  ⟨*proof*⟩

lemma *drinks_states: "S drinks = {|0, 1, 2|}"*
  ⟨*proof*⟩

lemma *possible_steps_0:*

```
  "length i = 1 ⟹
   possible_steps drinks 0 r (STR ''select'') i = {|(1, select)|}"
```
  ⟨*proof*⟩

lemma *first_step_select:*

```
  "(s', t) |∈| possible_steps drinks 0 r aa b ⟹ s' = 1 ∧ t = select"
```
  ⟨*proof*⟩

lemma *drinks_vend_insufficient:*

```
  "r $ 2 = Some (Num x1) ⟹
   x1 < 100 ⟹
   possible_steps drinks 1 r (STR ''vend'') [] = {|(1, vend_fail)|}"
```
  ⟨*proof*⟩

lemma *drinks_vend_invalid:*

```
  "∄n. r $ 2 = Some (Num n) ⟹
   possible_steps drinks 1 r (STR ''vend'') [] = {||}"
```
  ⟨*proof*⟩

lemma *possible_steps_1_coin:*

```
  "length i = 1 ⟹ possible_steps drinks 1 r (STR ''coin'') i = {|(1, coin)|}"
```
  ⟨*proof*⟩

lemma *possible_steps_2_vend:*

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

lemma *recognises_from_2:*

```
  "recognises_execution drinks 1 <1 $:= d, 2 $:= Some (Num 100)> [(STR ''vend'', [])]"
```
  ⟨*proof*⟩

lemma *recognises_from_1a:*

```
  "recognises_execution drinks 1 <1 $:= d, 2 $:= Some (Num 50)> [(STR ''coin'', [Num 50]), (STR ''vend'',
[])]"
```
  ⟨*proof*⟩

```
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'', [])]"
  ⟨proof⟩
```

```
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'')]]"
  ⟨proof⟩
```

```
lemma rejects_input:
  "l ≠ STR ''coin'' ⟹
   l ≠ STR ''vend'' ⟹
   ¬ recognises_execution drinks 1 d' [(l, i)]"
  ⟨proof⟩
```

```
lemma rejects_recognises_prefix: "l ≠ STR ''coin'' ⟹
   l ≠ STR ''vend'' ⟹
   ¬ (recognises drinks [(STR ''select'', [Str ''coke'']), (l, i)])"
  ⟨proof⟩
```

```
lemma rejects_termination:
  "observe_execution drinks 0 <> [(STR ''select'', [Str ''coke'']), (STR ''rejects'', [Num 50]), (STR ''coin'',
[Num 50])] = [[]]"
  ⟨proof⟩
```

```
lemma r2_0_vend:
  "can_take_transition vend i r ⟹
   ∃n. r $ 2 = Some (Num n) ∧ n ≥ 100"
  ⟨proof⟩
```

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

```
lemma drinks_end: "possible_steps drinks 2 r a b = {||}"
  ⟨proof⟩
```

```
lemma drinks_vend_r2_String:
  "r $ 2 = Some (value.Str x2) ⟹
   possible_steps drinks 1 r (STR ''vend'') [] = {||}"
  ⟨proof⟩
```

```
lemma drinks_vend_r2_rejects:
  "∄n. r $ 2 = Some (Num n) ⟹ step drinks 1 r (STR ''vend'') [] = None"
  ⟨proof⟩
```

```
lemma drinks_0_rejects:
  "¬ (fst a = STR ''select'' ∧ length (snd a) = 1) ⟹
    (possible_steps drinks 0 r (fst a) (snd a)) = {||}"
  ⟨proof⟩
```

```
lemma drinks_vend_empty: "(possible_steps drinks 0 <> (STR ''vend'') []) = {||}"
  ⟨proof⟩
```

```
lemma drinks_1_rejects:
  "fst a = STR ''coin'' ⟶ length (snd a) ≠ 1 ⟹
          a ≠ (STR ''vend'', []) ⟹
          possible_steps drinks 1 r (fst a) (snd a) = {||}"
  ⟨proof⟩
```

```
lemma drinks_rejects_future: "¬ recognises_execution drinks 2 d ((l, i)#t)"
  ⟨proof⟩

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⟩

lemma rejects_state_step: "s > 1 ⟹ step drinks s r l i = None"
  ⟨proof⟩

lemma invalid_other_states:
  "s > 1 ⟹ ¬ recognises_execution drinks s r ((aa, b) # t)"
  ⟨proof⟩

lemma vend_ge_100:
  "possible_steps drinks 1 r l i = {|(2, vend)|} ⟹
   ¬? value_gt (Some (Num 100)) (r $ 2) = trilean.true"
  ⟨proof⟩

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 = {||}"
  ⟨proof⟩

lemma possible_steps_0_not_select: "a ≠ STR ''select'' ⟹
       possible_steps drinks 0 <> a b = {||}"
  ⟨proof⟩

lemma possible_steps_select_wrong_arity: "a = STR ''select'' ⟹
       length b ≠ 1 ⟹
       possible_steps drinks 0 <> a b = {||}"
  ⟨proof⟩

lemma possible_steps_0_invalid:
  "¬ (l = STR ''select'' ∧ length i = 1) ⟹
   possible_steps drinks 0 <> l i = {||}"
  ⟨proof⟩

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)|}"
  ⟨proof⟩

lemma possible_steps_1:
  "length i = 1 ⟹
   possible_steps drinks2 1 r ((STR ''coin'')) i = {|(2, coin)|}"
  ⟨proof⟩

lemma possible_steps_2_coin:
  "length i = 1 ⟹
   possible_steps drinks2 2 r ((STR ''coin'')) i = {|(2, coin)|}"
  ⟨proof⟩

lemma possible_steps_2_vend:
  "r $ 2 = Some (Num n) ⟹
   n ≥ 100 ⟹
   possible_steps drinks2 2 r ((STR ''vend'')) [] = {|(3, vend)|}"
  ⟨proof⟩

lemma recognises_first_select:
  "recognises_execution drinks 0 r ((aa, b) # as) ⟹ aa = STR ''select'' ∧ length b = 1"
  ⟨proof⟩

lemma drinks2_vend_insufficient:
  "possible_steps drinks2 1 r ((STR ''vend'')) [] = {|(1, vend_nothing)|}"
  ⟨proof⟩

lemma drinks2_vend_insufficient2:
  "r $ 2 = Some (Num x1) ⟹
   x1 < 100 ⟹
   possible_steps drinks2 2 r ((STR ''vend'')) [] = {|(2, vend_fail)|}"
  ⟨proof⟩

lemma drinks2_vend_sufficient: "r $ 2 = Some (Num x1) ⟹
                ¬ x1 < 100 ⟹
                possible_steps drinks2 2 r ((STR ''vend'')) [] = {|(3, vend)|}"
  ⟨proof⟩

lemma recognises_1_2: "recognises_execution drinks 1 r t ⟶ recognises_execution drinks2 2 r t"
⟨proof⟩

lemma drinks_reject_0_2:
  "∄i. a = (STR ''select'', [i]) ⟹
   possible_steps drinks 0 r (fst a) (snd a) = {||}"
  ⟨proof⟩

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'')]]"
  ⟨proof⟩

lemma drinks2_0_invalid:
  "¬ (aa = (STR ''select'') ∧ length (b) = 1) ⟹
    (possible_steps drinks2 0 <> aa b) = {||}"
  ⟨proof⟩

lemma drinks2_vend_r2_none:
  "r $ 2 = None ⟹ possible_steps drinks2 2 r ((STR ''vend'')) [] = {||}"
  ⟨proof⟩

lemma drinks2_end: "possible_steps drinks2 3 r a b = {||}"
  ⟨proof⟩

lemma drinks2_vend_r2_String: "r $ 2 = Some (value.Str x2) ⟹
               possible_steps drinks2 2 r ((STR ''vend'')) [] = {||}"
  ⟨proof⟩

lemma drinks2_2_invalid:
  "fst a = (STR ''coin'') ⟶ length (snd a) ≠ 1 ⟹
          a ≠ ((STR ''vend''), []) ⟹
          possible_steps drinks2 2 r (fst a) (snd a) = {||}"
  ⟨proof⟩

lemma drinks2_1_invalid:
  "¬(a = (STR ''coin'') ∧ length b = 1) ⟹
      ¬(a = (STR ''vend'') ∧ b = []) ⟹
    possible_steps drinks2 1 r a b = {||}"
  ⟨proof⟩

lemma drinks2_vend_invalid:
  "∄n. r $ 2 = Some (Num n) ⟹
   possible_steps drinks2 2 r (STR ''vend'') [] = {||}"
  ⟨proof⟩

lemma equiv_1_2: "executionally_equivalent drinks 1 r drinks2 2 r x"
⟨proof⟩

lemma equiv_1_1: "r$2 = Some (Num 0) ⟹ executionally_equivalent drinks 1 r drinks2 1 r x"
⟨proof⟩


lemma executional_equivalence: "executionally_equivalent drinks 0 <> drinks2 0 <> t"
⟨proof⟩

lemma observational_equivalence: "trace_equivalent drinks drinks2"
  ⟨proof⟩

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⟩

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⟩

lemma LTL_r2_not_always_gt_100: "not (alw (check_exp (Gt (V (Rg 2)) (L (Num 100))))) (watch drinks i)"
  ⟨proof⟩

lemma drinks_step_2_none: "ltl_step drinks (Some 2) r e = (None, [], r)"
  ⟨proof⟩

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⟩

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"
  ⟨proof⟩

lemma LTL_nxt_2_means_vend:
  "alw (nxt (state_eq (Some 2)) impl (state_eq (Some 1))) (watch drinks i)"
⟨proof⟩

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"
  ⟨proof⟩


lemma LTL_costsMoney:
  "(alw (nxt (state_eq (Some 2)) impl (check_exp (Ge (V (Rg 2)) (L (Num 100)))))) (watch drinks i)"
⟨proof⟩

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)"
  ⟨proof⟩

lemma implode_select: "String.implode ''select'' = STR ''select''"
  ⟨proof⟩

lemma implode_coin: "String.implode ''coin'' = STR ''coin''"
  ⟨proof⟩

lemma implode_vend: "String.implode ''vend'' = STR ''vend''"
  ⟨proof⟩

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)"
```
⟨proof⟩

lemma *ltl_step_not_select:*
```
"∄i. e = (STR ''select'', [i]) ⟹
 ltl_step drinks (Some 0) r e = (None, [], r)"
```
⟨proof⟩

lemma *ltl_step_select:*
```
"ltl_step drinks (Some 0) <> (STR ''select'', [i]) = (Some 1, [], <1 $:= Some i, 2 $:= Some (Num 0)>)"
```
⟨proof⟩

lemma *ltl_step_not_coin_or_vend:*
```
"∄i. e = (STR ''coin'', [i]) ⟹
  e ≠ (STR ''vend'', []) ⟹
  ltl_step drinks (Some 1) r e = (None, [], r)"
```
⟨proof⟩

lemma *ltl_step_coin:*
```
"∃p r'. ltl_step drinks (Some 1) r (STR ''coin'', [i]) = (Some 1, p, r')"
```
⟨proof⟩

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))"
```
⟨proof⟩

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)"
```
⟨proof⟩

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"*
⟨proof⟩

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⟩

lemma *steps_1_invalid:*
```
    "∄i. (a, b) = (STR ''coin'', [i]) ⟹
     ∄i. (a, b) = (STR ''vend'', []) ⟹
     possible_steps drinks 1 r a b = {||}"
```
⟨proof⟩

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"*
⟨proof⟩

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⟩

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)"
```

⟨proof⟩

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`.