Inference 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 formal implementation of a state-merging technique to infer extended finite state machines (EFSMs), complete with output and update functions, from black-box traces. In particular, we define the *subsumption in context* relation as a means of determining whether one transition is able to account for the behaviour of another. Building on this, we define the *direct subsumption* relation, which lifts the *subsumption in context* relation to EFSM level such that we can use it to determine whether it is safe to merge a given pair of transitions. Key proofs include the conditions necessary for subsumption to occur and the that subsumption and direct subsumption are preorder relations.

We also provide a number of different *heuristics* which can be used to abstract away concrete values into *registers* so that more states and transitions can be merged and provide proofs of the various conditions which must hold for these abstractions to subsume their ungeneralised counterparts. A Code Generator setup to create executable Scala code is also defined.

Keywords: EFSMs, Model inference, Reverse engineering

Contents

1 Introduction

This AFP entry provides a formal implementation of a state-merging technique to infer EFSMs from black-box traces and is an accompaniment to work published in [1] and [2]. The inference technique builds off classical FSM inference techniques which work by first building a Prefix Tree Acceptor from traces of the underlying system, and then iteratively merging states which share behaviour to form a smaller model.

Most notably, we formalise the definitions of *subsumption in context* and *direct subsumption.* When merging EFSM transitions, one must *account for* the behaviour of the other. The *subsumption in context* relation from [1] formalises the intuition that, in certain contexts, a transition t_2 reproduces the behaviour of, and updates the data state in a manner consistent with, another transition t_1, meaning that t_2 can be used in place of t_1 with no observable difference in behaviour. This relation requires us to supply a context in which to test subsumption, but there is a problem when we try to apply this to inference: Which context should we use? The *directly subsumes* relation presented in [2] incorporates subsumption into a relation which can be used to determine if it is safe to merge a pair of transitions in an EFSM. It is this which allows us to take the subsumption relation from [1] and use it in the inference process.

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).

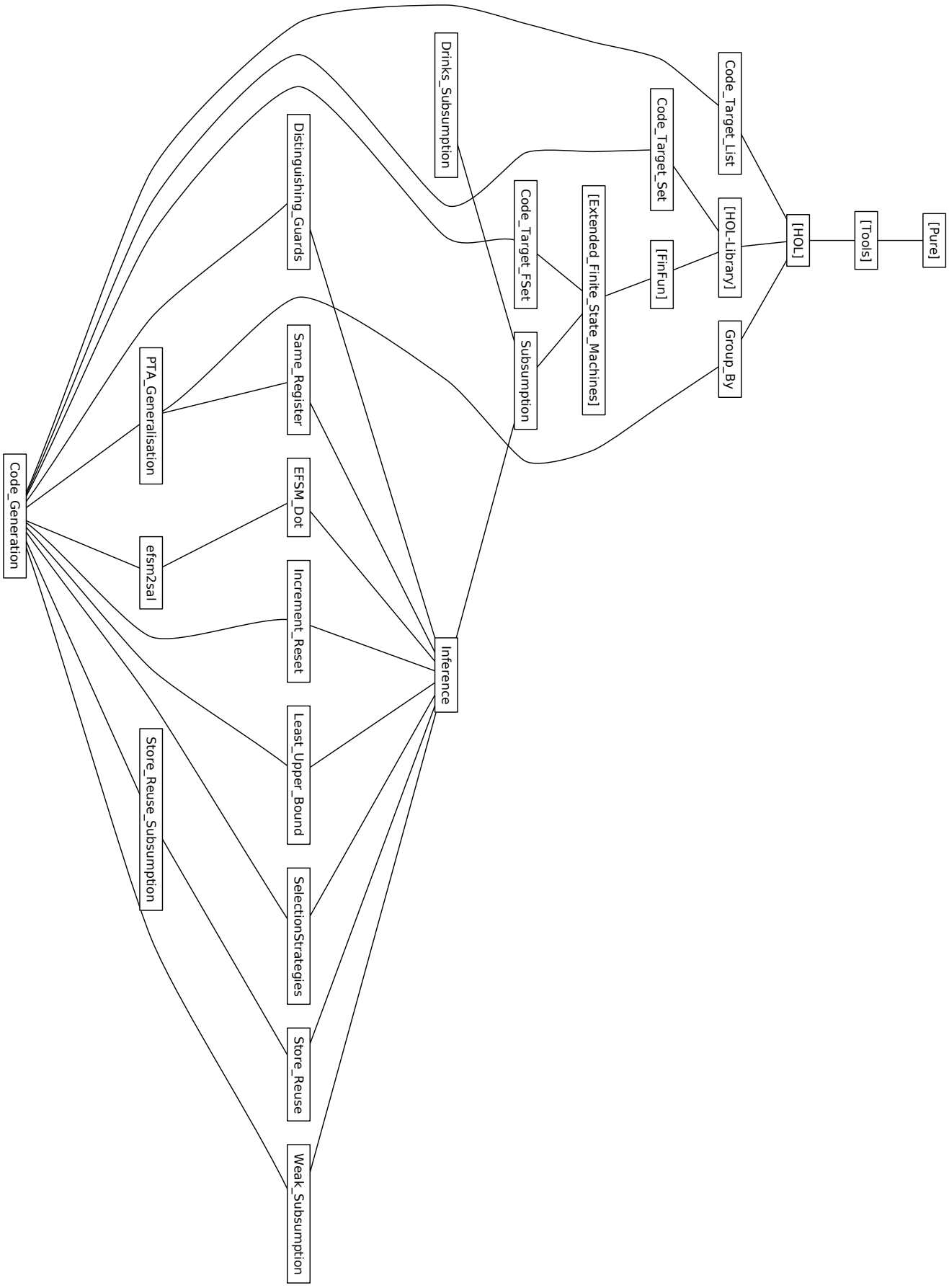

Figure 1.1: The Dependency Graph of the Isabelle Theories.

1.1 Contexts and Subsumption (Subsumption)

This theory uses contexts to extend the idea of transition subsumption from [3] to EFSM transitions with register update functions. The *subsumption in context* relation is the main contribution of [1].

```
theory Subsumption
  imports
    "Extended_Finite_State_Machines.EFSM"
begin

definition posterior_separate :: "nat ⇒ vname gexp list ⇒ update_function list ⇒ inputs ⇒ registers
⇒ registers option" where
  "posterior_separate a g u i r = (if can_take a g i r then Some (apply_updates u (join_ir i r) r) else
None)"

definition posterior :: "transition ⇒ inputs ⇒ registers ⇒ registers option" where
  "posterior t i r = posterior_separate (Arity t) (Guards t) (Updates t) i r"

definition subsumes :: "transition ⇒ registers ⇒ transition ⇒ bool" where
  "subsumes t2 r t1 = (Label t1 = Label t2 ∧ Arity t1 = Arity t2 ∧
                       (∀i. can_take_transition t1 i r ⟶ can_take_transition t2 i r) ∧
                       (∀i. can_take_transition t1 i r ⟶
                            evaluate_outputs t1 i r = evaluate_outputs t2 i r) ∧
                       (∀p1 p2 i. posterior_separate (Arity t1) (Guards t1) (Updates t2) i r = Some p2 ⟶
                                  posterior_separate (Arity t1) (Guards t1) (Updates t1) i r = Some p1 ⟶
                                  (∀P r'. (p1 $ r' = None) ∨ (P (p2 $ r') ⟶ P (p1 $ r'))))
                      )"

lemma no_functionality_subsumed:
  "Label t1 = Label t2 ⟹
   Arity t1 = Arity t2 ⟹
   ∄i. can_take_transition t1 i c ⟹
   subsumes t2 c t1"
  ⟨proof⟩

lemma subsumes_updates:
  "subsumes t2 r t1 ⟹
   can_take_transition t1 i r ⟹
   evaluate_updates t1 i r $ a = Some x ⟹
   evaluate_updates t2 i r $ a = Some x"
  ⟨proof⟩

lemma subsumption:
  "(Label t1 = Label t2 ∧ Arity t1 = Arity t2) ⟹
   (∀i. can_take_transition t1 i r ⟶ can_take_transition t2 i r) ⟹
   (∀i. can_take_transition t1 i r ⟶
        evaluate_outputs t1 i r = evaluate_outputs t2 i r) ⟹

   (∀p1 p2 i. posterior_separate (Arity t1) (Guards t1) (Updates t2) i r = Some p2 ⟶
              posterior_separate (Arity t1) (Guards t1) (Updates t1) i r = Some p1 ⟶
              (∀P r'. (p1 $ r' = None) ∨ (P (p2 $ r') ⟶ P (p1 $ r')))) ⟹
   subsumes t2 r t1"
  ⟨proof⟩

lemma bad_guards:
  "∃i. can_take_transition t1 i r ∧ ¬ can_take_transition t2 i r ⟹
   ¬ subsumes t2 r t1"
  ⟨proof⟩

lemma inconsistent_updates:
  "∃p2 p1. (∃i. posterior_separate (Arity t1) (Guards t1) (Updates t2) i r = Some p2 ∧
                posterior_separate (Arity t1) (Guards t1) (Updates t1) i r = Some p1) ∧
           (∃r' P. P (p2 $ r') ∧ (∃y. p1 $ r' = Some y) ∧ ¬ P (p1 $ r')) ⟹
```

```
    ¬ subsumes t2 r t1"
  ⟨proof⟩

lemma bad_outputs:
  "∃i. can_take_transition t1 i r ∧ evaluate_outputs t1 i r ≠ evaluate_outputs t2 i r ⟹
   ¬ subsumes t2 r t1"
  ⟨proof⟩

lemma no_choice_no_subsumption: "Label t = Label t' ⟹
   Arity t = Arity t' ⟹
   ¬ choice t t' ⟹
   ∃i. can_take_transition t' i c ⟹
  ¬ subsumes t c t'"
  ⟨proof⟩

lemma subsumption_def_alt: "subsumes t1 c t2 = (Label t2 = Label t1 ∧
    Arity t2 = Arity t1 ∧
    (∀i. can_take_transition t2 i c ⟶ can_take_transition t1 i c) ∧
    (∀i. can_take_transition t2 i c ⟶ evaluate_outputs t2 i c = evaluate_outputs t1 i c) ∧
    (∀i. can_take_transition t2 i c ⟶
         (∀r' P.
             P (evaluate_updates t1 i c $ r') ⟶
             evaluate_updates t2 i c $ r' = None ∨ P (evaluate_updates t2 i c $ r'))))"
  ⟨proof⟩

lemma subsumes_update_equality:
  "subsumes t1 c t2 ⟹ (∀i. can_take_transition t2 i c ⟶
         (∀r'.
             ((evaluate_updates t1 i c $ r') = (evaluate_updates t2 i c $ r')) ∨
             evaluate_updates t2 i c $ r' = None))"
  ⟨proof⟩
lemma subsumes_reflexive: "subsumes t c t"⟨proof⟩

  ⟨proof⟩
lemma subsumes_transitive:
  assumes p1: "subsumes t1 c t2"
      and p2: "subsumes t2 c t3"
  shows "subsumes t1 c t3"

  ⟨proof⟩

lemma subsumes_possible_steps_replace:
  "(s2', t2') |∈| possible_steps e2 s2 r2 l i ⟹
   subsumes t2 r2 t1 ⟹
   ((s2, s2'), t2') = ((ss2, ss2'), t1) ⟹
   (s2', t2) |∈| possible_steps (replace e2 ((ss2, ss2'), t1) ((ss2, ss2'), t2)) s2 r2 l i"
⟨proof⟩
```

1.1.1 Direct Subsumption

When merging EFSM transitions, one must *account for* the behaviour of the other. The *subsumption in context* relation formalises the intuition that, in certain contexts, a transition t_2 reproduces the behaviour of, and updates the data state in a manner consistent with, another transition t_1, meaning that t_2 can be used in place of t_1 with no observable difference in behaviour.

The subsumption in context relation requires us to supply a context in which to test subsumption, but there is a problem when we try to apply this to inference: Which context should we use? The *direct subsumption* relation works at EFSM level to determine when and whether one transition is able to account for the behaviour of another such that we can use one in place of another without adversely effecting observable behaviour.

```
definition directly_subsumes :: "transition_matrix ⇒ transition_matrix ⇒ cfstate ⇒ cfstate ⇒ transition
⇒ transition ⇒ bool" where
  "directly_subsumes e1 e2 s1 s2 t1 t2 ≡ (∀c1 c2 t. (obtains s1 c1 e1 0 <> t ∧ obtains s2 c2 e2 0 <> t)
```

```
⟶ subsumes t1 c2 t2)"
```

lemma *subsumes_in_all_contexts_directly_subsumes:*
```
  "(⋀c. subsumes t2 c t1) ⟹ directly_subsumes e1 e2 s s' t2 t1"
```

⟨*proof*⟩

lemma *direct_subsumption:*
```
  "(⋀t c1 c2. obtains s1 c1 e1 0 <> t ⟹ obtains s2 c2 e2 0 <> t ⟹ f c2) ⟹
   (⋀c. f c ⟹ subsumes t1 c t2) ⟹
   directly_subsumes e1 e2 s1 s2 t1 t2"
```

⟨*proof*⟩

lemma *visits_and_not_subsumes:*
```
  "(∃c1 c2 t. obtains s1 c1 e1 0 <> t ∧ obtains s2 c2 e2 0 <> t ∧ ¬ subsumes t1 c2 t2) ⟹
   ¬ directly_subsumes e1 e2 s1 s2 t1 t2"
```

⟨*proof*⟩

lemma *directly_subsumes_reflexive: "directly_subsumes e1 e2 s1 s2 t t"*

⟨*proof*⟩

lemma *directly_subsumes_transitive:*
```
  assumes p1: "directly_subsumes e1 e2 s1 s2 t1 t2"
      and p2: "directly_subsumes e1 e2 s1 s2 t2 t3"
  shows "directly_subsumes e1 e2 s1 s2 t1 t3"
```

⟨*proof*⟩

end

1.1.2 Example

This theory shows how contexts can be used to prove transition subsumption.

theory *Drinks_Subsumption*
imports *"Extended_Finite_State_Machine_Inference.Subsumption" "Extended_Finite_State_Machines.Drinks_Machine_2"*
begin

lemma *stop_at_3: "¬obtains 1 c drinks2 3 r t"*
⟨*proof*⟩

lemma *no_1_2: "¬obtains 1 c drinks2 2 r t"*
⟨*proof*⟩

lemma *no_change_1_1: "obtains 1 c drinks2 1 r t ⟹ c = r"*
⟨*proof*⟩

lemma *obtains_1: "obtains 1 c drinks2 0 <> t ⟹ c $ 2 = Some (Num 0)"*
⟨*proof*⟩

lemma *obtains_1_1_2:*
```
  "obtains 1 c1 drinks2 1 r t ⟹
   obtains 1 c2 drinks 1 r t ⟹
   c1 = r ∧ c2 = r"
```
⟨*proof*⟩

lemma *obtains_1_c2:*
```
  "obtains 1 c1 drinks2 0 <> t ⟹ obtains 1 c2 drinks 0 <> t ⟹ c2 $ 2 = Some (Num 0)"
```
⟨*proof*⟩

lemma *directly_subsumes: "directly_subsumes drinks2 drinks 1 1 vend_fail vend_nothing"*
⟨*proof*⟩

lemma *directly_subsumes_flip: "directly_subsumes drinks2 drinks 1 1 vend_nothing vend_fail"*
⟨*proof*⟩

end

2 EFSM Inference

This chapter presents the definitions necessary for EFSM inference by state-merging.

2.1 Inference by State-Merging (Inference)

This theory sets out the key definitions for the inference of EFSMs from system traces.

```
theory Inference
  imports
    Subsumption
    "Extended_Finite_State_Machines.Transition_Lexorder"
    "HOL-Library.Product_Lexorder"
begin

declare One_nat_def [simp del]
```

2.1.1 Transition Identifiers

We first need to define the `iEFSM` data type which assigns each transition a unique identity. This is necessary because transitions may not occur uniquely in an EFSM. Assigning transitions a unique identifier enables us to look up the origin and destination states of transitions without having to pass them around in the inference functions.

```
type_synonym tid = nat
type_synonym tids = "tid list"
type_synonym iEFSM = "(tids × (cfstate × cfstate) × transition) fset"

definition origin :: "tids ⇒ iEFSM ⇒ nat" where
  "origin uid t = fst (fst (snd (fthe_elem (ffilter (λx. set uid ⊆ set (fst x)) t))))"

definition dest :: "tids ⇒ iEFSM ⇒ nat" where
  "dest uid t = snd (fst (snd (fthe_elem (ffilter (λx. set uid ⊆ set (fst x)) t))))"

definition get_by_id :: "iEFSM ⇒ tid ⇒ transition" where
  "get_by_id e uid = (snd ∘ snd) (fthe_elem (ffilter (λ(tids, _). uid ∈ set tids) e))"

definition get_by_ids :: "iEFSM ⇒ tids ⇒ transition" where
  "get_by_ids e uid = (snd ∘ snd) (fthe_elem (ffilter (λ(tids, _). set uid ⊆ set tids) e))"

definition uids :: "iEFSM ⇒ nat fset" where
  "uids e = ffUnion (fimage (fset_of_list ∘ fst) e)"

definition max_uid :: "iEFSM ⇒ nat option" where
  "max_uid e = (let uids = uids e in if uids = {||} then None else Some (fMax uids))"

definition tm :: "iEFSM ⇒ transition_matrix" where
  "tm e = fimage snd e"

definition all_regs :: "iEFSM ⇒ nat set" where
  "all_regs e = EFSM.all_regs (tm e)"

definition max_reg :: "iEFSM ⇒ nat option" where
  "max_reg e = EFSM.max_reg (tm e)"

definition "max_reg_total e = (case max_reg e of None ⇒ 0 | Some r ⇒ r)"
```

```
definition max_output :: "iEFSM ⇒ nat" where
  "max_output e = EFSM.max_output (tm e)"

definition max_int :: "iEFSM ⇒ int" where
  "max_int e = EFSM.max_int (tm e)"

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

lemma S_alt: "S t = EFSM.S (tm t)"
  ⟨proof⟩

lemma to_in_S:
  "(∃ to from uid. (uid, (from, to), t) |∈| xb ⟶ to |∈| S xb)"
  ⟨proof⟩

lemma from_in_S:
  "(∃ to from uid. (uid, (from, to), t) |∈| xb ⟶ from |∈| S xb)"
  ⟨proof⟩
```

2.1.2 Building the PTA

The first step in EFSM inference is to construct a PTA from the observed traces in the same way as for classical FSM inference. Beginning with the empty EFSM, we iteratively attempt to walk each observed trace in the model. When we reach a point where there is no available transition, one is added. For classical FSMs, this is simply an atomic label. EFSMs deal with data, so we need to add guards which test for the observed input values and outputs which produce the observed values.

```
primrec make_guard :: "value list ⇒ nat ⇒ vname gexp list" where
"make_guard [] _ = []" |
"make_guard (h#t) n = (gexp.Eq (V (vname.I n)) (L h))#(make_guard t (n+1))"

primrec make_outputs :: "value list ⇒ output_function list" where
  "make_outputs [] = []" |
  "make_outputs (h#t) = (L h)#(make_outputs t)"

definition max_uid_total :: "iEFSM ⇒ nat" where
  "max_uid_total e = (case max_uid e of None ⇒ 0 | Some u ⇒ u)"

definition add_transition :: "iEFSM ⇒ cfstate ⇒ label ⇒ value list ⇒ value list ⇒ iEFSM" where
  "add_transition e s label inputs outputs = finsert ([max_uid_total e + 1], (s, (maxS (tm e))+1), ⦇Label=label,
Arity=length inputs, Guards=(make_guard inputs 0), Outputs=(make_outputs outputs), Updates=[]⦈) e"

fun make_branch :: "iEFSM ⇒ cfstate ⇒ registers ⇒ trace ⇒ iEFSM" where
  "make_branch e _ _ [] = e" |
  "make_branch e s r ((label, inputs, outputs)#t) =
    (case (step (tm e) s r label inputs) of
      Some (transition, s', outputs', updated) ⇒
        if outputs' = (map Some outputs) then
          make_branch e s' updated t
        else
          make_branch (add_transition e s label inputs outputs) ((maxS (tm e))+1) r t  |
      None ⇒
          make_branch (add_transition e s label inputs outputs) ((maxS (tm e))+1) r t
    )"

primrec make_pta_aux :: "log ⇒ iEFSM ⇒ iEFSM" where
  "make_pta_aux [] e = e" |
  "make_pta_aux (h#t) e = make_pta_aux t (make_branch e 0 <> h)"

definition "make_pta log = make_pta_aux log {||}"

lemma make_pta_aux_fold [code]:
```

```
  "make_pta_aux l e = fold (λh e. make_branch e 0 <> h) l e"
```
⟨*proof*⟩

2.1.3 Integrating Heuristics

A key contribution of the inference technique presented in [2] is the ability to introduce *internal variables* to the model to generalise behaviours and allow transitions to be merged. This is done by providing the inference technique with a set of *heuristics*. The aim here is not to create a "one size fits all" magic oracle, rather to recognise particular *data usage patterns* which can be abstracted.

```
type_synonym update_modifier = "tids ⇒ tids ⇒ cfstate ⇒ iEFSM ⇒ iEFSM ⇒ iEFSM ⇒ (transition_matrix
⇒ bool) ⇒ iEFSM option"

definition null_modifier :: update_modifier where
  "null_modifier f _ _ _ _ _ _ = None"

definition replace_transition :: "iEFSM ⇒ tids ⇒ transition ⇒ iEFSM" where
  "replace_transition e uid new = (fimage (λ(uids, (from, to), t). if set uid ⊆ set uids then (uids, (from,
to), new) else (uids, (from, to), t)) e)"

definition replace_all :: "iEFSM ⇒ tids list ⇒ transition ⇒ iEFSM" where
  "replace_all e ids new = fold (λid acc. replace_transition acc id new) ids e"

definition replace_transitions :: "iEFSM ⇒ (tids × transition) list ⇒ iEFSM" where
  "replace_transitions e ts = fold (λ(uid, new) acc. replace_transition acc uid new) ts e"

primrec try_heuristics_check :: "(transition_matrix ⇒ bool) ⇒ update_modifier list ⇒ update_modifier"
where
  "try_heuristics_check _ [] = null_modifier" |
  "try_heuristics_check check (h#t) = (λa b c d e f ch.
    case h a b c d e f ch of
      Some e' ⇒ Some e' |
      None ⇒ (try_heuristics_check check t) a b c d e f ch
    )"
```

2.1.4 Scoring State Merges

To tackle the state merging challenge, we need some means of determining which states are compatible for merging. Because states are merged pairwise, we additionally require a way of ordering the state merges. The potential merges are then sorted highest to lowest according to this score such that we can merge states in order of their merge score.

We want to sort first by score (highest to lowest) and then by state pairs (lowest to highest) so we endup merging the states with the highest scores first and then break ties by those state pairs which are closest to the origin.

```
record score =
  Score :: nat
  S1 :: cfstate
  S2 :: cfstate

instantiation score_ext :: (linorder) linorder begin
definition less_score_ext :: "'a::linorder score_ext ⇒ 'a score_ext ⇒ bool" where
"less_score_ext t1 t2 = ((Score t2, S1 t1, S2 t1, more t1) < (Score t1, S1 t2, S2 t2, more t2) )"

definition less_eq_score_ext :: "'a::linorder score_ext ⇒ 'a::linorder score_ext ⇒ bool" where
 "less_eq_score_ext s1 s2 = (s1 < s2 ∨ s1 = s2)"

instance
  ⟨proof⟩
end

type_synonym scoreboard = "score fset"
type_synonym strategy = "tids ⇒ tids ⇒ iEFSM ⇒ nat"
```

definition *outgoing_transitions :: "cfstate ⇒ iEFSM ⇒ (cfstate × transition × tids) fset"* **where**
"outgoing_transitions s e = fimage (λ(uid, (from, to), t'). (to, t', uid)) ((ffilter (λ(uid, (origin, dest), t). origin = s)) e)"

primrec *paths_of_length :: "nat ⇒ iEFSM ⇒ cfstate ⇒ tids list fset"* **where**
"paths_of_length 0 _ _ = {|[]|}" |
"paths_of_length (Suc m) e s = (
let
outgoing = outgoing_transitions s e;
paths = ffUnion (fimage (λ(d, t, id). fimage (λp. id#p) (paths_of_length m e d)) outgoing)
in
ffilter (λl. length l = Suc m) paths
)"

lemma *paths_of_length_1: "paths_of_length 1 e s = fimage (λ(d, t, id). [id]) (outgoing_transitions s e)"*
⟨*proof*⟩

fun *step_score :: "(tids × tids) list ⇒ iEFSM ⇒ strategy ⇒ nat"* **where**
"step_score [] _ _ = 0" |
"step_score ((id1, id2)#t) e s = (
let score = s id1 id2 e in
if score = 0 then
0
else
score + (step_score t e s)
)"

lemma *step_score_foldr [code]:*
"step_score xs e s = foldr (λ(id1, id2) acc. let score = s id1 id2 e in
if score = 0 then
0
else
score + acc) xs 0"
⟨*proof*⟩

definition *score_from_list :: "tids list fset ⇒ tids list fset ⇒ iEFSM ⇒ strategy ⇒ nat"* **where**
"score_from_list P1 P2 e s = (
let
pairs = fimage (λ(l1, l2). zip l1 l2) (P1 |×| P2);
scored_pairs = fimage (λl. step_score l e s) pairs
in
fSum scored_pairs
)"

definition *k_score :: "nat ⇒ iEFSM ⇒ strategy ⇒ scoreboard"* **where**
"k_score k e strat = (
let
states = S e;
pairs_to_score = (ffilter (λ(x, y). x < y) (states |×| states));
paths = fimage (λ(s1, s2). (s1, s2, paths_of_length k e s1, paths_of_length k e s2)) pairs_to_score;
scores = fimage (λ(s1, s2, p1, p2). (|Score = score_from_list p1 p2 e strat, S1 = s1, S2 = s2|)) paths
in
ffilter (λx. Score x > 0) scores
)"

definition *score_state_pair :: "strategy ⇒ iEFSM ⇒ cfstate ⇒ cfstate ⇒ nat"* **where**
"score_state_pair strat e s1 s2 = (
let
T1 = outgoing_transitions s1 e;
T2 = outgoing_transitions s2 e
in
fSum (fimage (λ((_, _, t1), (_, _, t2)). strat t1 t2 e) (T1 |×| T2))

```
  )"

definition score_1 :: "iEFSM ⇒ strategy ⇒ scoreboard" where
  "score_1 e strat = (
    let
      states = S e;
      pairs_to_score = (ffilter (λ(x, y). x < y) (states |×| states));
      scores = fimage (λ(s1, s2). (|Score = score_state_pair strat e s1 s2, S1 = s1, S2 = s2|)) pairs_to_score
    in
      ffilter (λx. Score x > 0) scores
  )"

lemma score_1: "score_1 e s = k_score 1 e s"
⟨proof⟩

fun bool2nat :: "bool ⇒ nat" where
  "bool2nat True = 1" |
  "bool2nat False = 0"

definition score_transitions :: "transition ⇒ transition ⇒ nat" where
  "score_transitions t1 t2 = (
    if Label t1 = Label t2 ∧ Arity t1 = Arity t2 ∧ length (Outputs t1) = length (Outputs t2) then
      1 + bool2nat (t1 = t2) + card ((set (Guards t2)) ∩ (set (Guards t2))) + card ((set (Updates t2)) ∩
(set (Updates t2))) + card ((set (Outputs t2)) ∩ (set (Outputs t2)))
    else
      0
  )"
```

2.1.5 Merging States

```
definition merge_states_aux :: "nat ⇒ nat ⇒ iEFSM ⇒ iEFSM" where
  "merge_states_aux s1 s2 e = fimage (λ(uid, (origin, dest), t). (uid, (if origin = s1 then s2 else origin
, if dest = s1 then s2 else dest), t)) e"

definition merge_states :: "nat ⇒ nat ⇒ iEFSM ⇒ iEFSM" where
  "merge_states x y t = (if x > y then merge_states_aux x y t else merge_states_aux y x t)"

lemma merge_states_symmetry: "merge_states x y t = merge_states y x t"
  ⟨proof⟩

lemma merge_state_self: "merge_states s s t = t"
  ⟨proof⟩

lemma merge_states_self_simp [code]:
  "merge_states x y t = (if x = y then t else if x > y then merge_states_aux x y t else merge_states_aux
y x t)"
  ⟨proof⟩
```

2.1.6 Resolving Nondeterminism

Because EFSM transitions are not simply atomic actions, duplicated behaviours cannot be resolved into a single transition by simply merging destination states, as it can in classical FSM inference. It is now possible for attempts to resolve the nondeterminism introduced by merging states to fail, meaning that two states which initially seemed compatible cannot actually be merged. This is not the case in classical FSM inference.

```
type_synonym nondeterministic_pair = "(cfstate × (cfstate × cfstate) × ((transition × tids) × (transition
× tids)))"

definition state_nondeterminism :: "nat ⇒ (cfstate × transition × tids) fset ⇒ nondeterministic_pair
fset" where
  "state_nondeterminism og nt = (if size nt < 2 then {||} else ffUnion (fimage (λx. let (dest, t) = x in
fimage (λy. let (dest', t') = y in (og, (dest, dest'), (t, t'))) (nt - {|x|})) nt))"
```

lemma *state_nondeterminism_empty [simp]: "state_nondeterminism a {||} = {||}"*
⟨*proof*⟩

lemma *state_nondeterminism_singledestn [simp]: "state_nondeterminism a {|x|} = {||}"*
⟨*proof*⟩

definition *nondeterministic_pairs :: "iEFSM ⇒ nondeterministic_pair fset"* **where**
"nondeterministic_pairs t = ffilter (λ(_, _, (t, _), (t', _)). Label t = Label t' ∧ Arity t = Arity t' ∧ choice t t') (ffUnion (fimage (λs. state_nondeterminism s (outgoing_transitions s t)) (S t)))"

definition *nondeterministic_pairs_labar_dest :: "iEFSM ⇒ nondeterministic_pair fset"* **where**
"nondeterministic_pairs_labar_dest t = ffilter
(λ(_, (d, d'), (t, _), (t', _)).
Label t = Label t' ∧ Arity t = Arity t' ∧ (choice t t' ∨ (Outputs t = Outputs t' ∧ d = d')))
(ffUnion (fimage (λs. state_nondeterminism s (outgoing_transitions s t)) (S t)))"

definition *nondeterministic_pairs_labar :: "iEFSM ⇒ nondeterministic_pair fset"* **where**
"nondeterministic_pairs_labar t = ffilter
(λ(_, (d, d'), (t, _), (t', _)).
Label t = Label t' ∧ Arity t = Arity t' ∧ (choice t t' ∨ Outputs t = Outputs t'))
(ffUnion (fimage (λs. state_nondeterminism s (outgoing_transitions s t)) (S t)))"

definition *deterministic :: "iEFSM ⇒ (iEFSM ⇒ nondeterministic_pair fset) ⇒ bool"* **where**
"deterministic t np = (np t = {||})"

definition *nondeterministic :: "iEFSM ⇒ (iEFSM ⇒ nondeterministic_pair fset) ⇒ bool"* **where**
"nondeterministic t np = (¬ deterministic t np)"

definition *insert_transition :: "tids ⇒ cfstate ⇒ cfstate ⇒ transition ⇒ iEFSM ⇒ iEFSM"* **where**
"insert_transition uid from to t e = (
if ∄(uid, (from', to'), t') |∈| e. from = from' ∧ to = to' ∧ t = t' then
finsert (uid, (from, to), t) e
else
fimage (λ(uid', (from', to'), t').
if from = from' ∧ to = to' ∧ t = t' then
(List.union uid' uid, (from', to'), t')
else
(uid', (from', to'), t')
) e
)"

definition *make_distinct :: "iEFSM ⇒ iEFSM"* **where**
"make_distinct e = ffold_ord (λ(uid, (from, to), t) acc. insert_transition uid from to t acc) e {||}"

— When we replace one transition with another, we need to merge their uids to keep track of which
— transition accounts for which action in the original traces
definition *merge_transitions_aux :: "iEFSM ⇒ tids ⇒ tids ⇒ iEFSM"* **where**
"merge_transitions_aux e oldID newID = (let
(uids1, (origin, dest), old) = fthe_elem (ffilter (λ(uids, _). oldID = uids) e);
(uids2, (origin, dest), new) = fthe_elem (ffilter (λ(uids, _). newID = uids) e) in
make_distinct (finsert (List.union uids1 uids2, (origin, dest), new) (e - {|(uids1, (origin, dest), old), (uids2, (origin, dest), new)|}))
)"

```
definition merge_transitions :: "(cfstate × cfstate) set ⇒ iEFSM ⇒ iEFSM ⇒ iEFSM ⇒ transition ⇒ tids
⇒ transition ⇒ tids ⇒ update_modifier ⇒ (transition_matrix ⇒ bool) ⇒ iEFSM option" where
  "merge_transitions failedMerges oldEFSM preDestMerge destMerge t1 u1 t2 u2 modifier check = (
     if ∀id ∈ set u1. directly_subsumes (tm oldEFSM) (tm destMerge) (origin [id] oldEFSM) (origin u1 destMerge)
t2 t1 then
       — Replace t1 with t2
       Some (merge_transitions_aux destMerge u1 u2)
     else if ∀id ∈ set u2. directly_subsumes (tm oldEFSM) (tm destMerge) (origin [id] oldEFSM) (origin
u2 destMerge) t1 t2 then
       — Replace t2 with t1
       Some (merge_transitions_aux destMerge u2 u1)
     else
        case modifier u1 u2 (origin u1 destMerge) destMerge preDestMerge oldEFSM check of
          None ⇒ None |
          Some e ⇒ Some (make_distinct e)
   )"

definition outgoing_transitions_from :: "iEFSM ⇒ cfstate ⇒ transition fset" where
  "outgoing_transitions_from e s = fimage (λ(_, _, t). t) (ffilter (λ(_, (orig, _), _). orig = s) e)"

definition order_nondeterministic_pairs :: "nondeterministic_pair fset ⇒ nondeterministic_pair list" where
  "order_nondeterministic_pairs s = map snd (sorted_list_of_fset (fimage (λs. let (_, _, (t1, _), (t2, _))
= s in (score_transitions t1 t2, s)) s))"
```

```
function resolve_nondeterminism :: "(cfstate × cfstate) set ⇒ nondeterministic_pair list ⇒ iEFSM ⇒ iEFSM
⇒ update_modifier ⇒ (transition_matrix ⇒ bool) ⇒ (iEFSM ⇒ nondeterministic_pair fset) ⇒ (iEFSM option
× (cfstate × cfstate) set)" where
  "resolve_nondeterminism failedMerges [] _ newEFSM _ check np = (
      if deterministic newEFSM np ∧ check (tm newEFSM) then Some newEFSM else None, failedMerges
  )" |
  "resolve_nondeterminism failedMerges ((from, (dest1, dest2), ((t1, u1), (t2, u2)))#ss) oldEFSM newEFSM
m check np = (
    if (dest1, dest2) ∈ failedMerges ∨ (dest2, dest1) ∈ failedMerges then
      (None, failedMerges)
    else
    let destMerge = merge_states dest1 dest2 newEFSM in
    case merge_transitions failedMerges oldEFSM newEFSM destMerge t1 u1 t2 u2 m check of
      None ⇒ resolve_nondeterminism (insert (dest1, dest2) failedMerges) ss oldEFSM newEFSM m check np
|
      Some new ⇒ (
        let newScores = order_nondeterministic_pairs (np new) in
        if (size new, size (S new), size (newScores)) < (size newEFSM, size (S newEFSM), size ss) then
          case resolve_nondeterminism failedMerges newScores oldEFSM new m check np of
            (Some new', failedMerges) ⇒ (Some new', failedMerges) |
            (None, failedMerges) ⇒ resolve_nondeterminism (insert (dest1, dest2) failedMerges) ss oldEFSM
newEFSM m check np
        else
          (None, failedMerges)
      )
  )"
    ⟨proof⟩
termination
  ⟨proof⟩
```

2.1.7 EFSM Inference

```
definition merge :: "(cfstate × cfstate) set ⇒ iEFSM ⇒ nat ⇒ nat ⇒ update_modifier ⇒ (transition_matrix
⇒ bool) ⇒ (iEFSM ⇒ nondeterministic_pair fset) ⇒ (iEFSM option × (cfstate × cfstate) set)" where
  "merge failedMerges e s1 s2 m check np = (
    if s1 = s2 ∨ (s1, s2) ∈ failedMerges ∨ (s2, s1) ∈ failedMerges then
      (None, failedMerges)
    else
      let e' = make_distinct (merge_states s1 s2 e) in
      resolve_nondeterminism failedMerges (order_nondeterministic_pairs (np e')) e e' m check np
  )"
```

```
function inference_step :: "(cfstate × cfstate) set ⇒ iEFSM ⇒ score fset ⇒ update_modifier ⇒ (transition_matr
⇒ bool) ⇒ (iEFSM ⇒ nondeterministic_pair fset) ⇒ (iEFSM option × (cfstate × cfstate) set)" where
  "inference_step failedMerges e s m check np = (
     if s = {|||} then (None, failedMerges) else
     let
      h = fMin s;
      t = s - {|h|}
    in
    case merge failedMerges e (S1 h) (S2 h) m check np of
      (Some new, failedMerges) ⇒ (Some new, failedMerges) |
      (None, failedMerges) ⇒ inference_step (insert ((S1 h), (S2 h)) failedMerges) e t m check np
  )"
  ⟨proof⟩
termination
  ⟨proof⟩
```

```
function infer :: "(cfstate × cfstate) set ⇒ nat ⇒ iEFSM ⇒ strategy ⇒ update_modifier ⇒ (transition_matrix
⇒ bool) ⇒ (iEFSM ⇒ nondeterministic_pair fset) ⇒ iEFSM" where
  "infer failedMerges k e r m check np = (
    let scores = if k = 1 then score_1 e r else (k_score k e r) in
    case inference_step failedMerges e (ffilter (λs. (S1 s, S2 s) ∉ failedMerges ∧ (S2 s, S1 s) ∉ failedMerges)
scores) m check np of
      (None, _) ⇒ e |
      (Some new, failedMerges) ⇒ if (S new) |⊂| (S e) then infer failedMerges k new r m check np else
e
  )"
  ⟨proof⟩
termination
  ⟨proof⟩

fun get_ints :: "trace ⇒ int list" where
  "get_ints [] = []" |
  "get_ints ((_, inputs, outputs)#t) = (map (λx. case x of Num n ⇒ n) (filter is_Num (inputs@outputs)))"

definition learn :: "nat ⇒ iEFSM ⇒ log ⇒ strategy ⇒ update_modifier ⇒ (iEFSM ⇒ nondeterministic_pair
fset) ⇒ iEFSM" where
  "learn n pta l r m np = (
     let check = accepts_log (set l) in
         (infer {} n pta r m check np)
   )"
```

2.1.8 Evaluating Inferred Models

We need a function to test the EFSMs we infer. The `test_trace` function executes a trace in the model and outputs a more comprehensive trace such that the expected outputs and actual outputs can be compared. If a point is reached where the model does not recognise an action, the remainder of the trace forms the second element of the output pair such that we know the exact point at which the model stopped processing.

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

fun test_trace :: "trace ⇒ iEFSM ⇒ cfstate ⇒ registers ⇒ ((label × inputs × cfstate × cfstate ×
registers × tids × value list × outputs) list × trace)" where
  "test_trace [] _ _ _ = ([], [])" |
  "test_trace ((l, i, expected)#es) e s r = (
    let
      ps = i_possible_steps e s r l i
    in
      if fis_singleton ps then
        let
          (id, s', t) = fthe_elem ps;
          r' = evaluate_updates t i r;
          actual = evaluate_outputs t i r;
          (est, fail) = (test_trace es e s' r')
        in
        ((l, i, s, s', r, id, expected, actual)#est, fail)
      else
        ([], (l, i, expected)#es)
  )"
```

The `test_log` function executes the `test_trace` function on a collection of traces known as the *test set.*

```
definition test_log :: "log ⇒ iEFSM ⇒ ((label × inputs × cfstate × cfstate × registers × tids × value
list × outputs) list × trace) list" where
  "test_log l e = map (λt. test_trace t e 0 <>) l"
```

end

2.2 Selection Strategies (SelectionStrategies)

The strategy used to idenfity and prioritise states to be merged plays a big part in how the final model turns out. This theory file presents a number of different selection strategies.

theory *SelectionStrategies*
imports *Inference*
begin

The simplest strategy is to assign one point for each shared pair of transitions.

```
definition exactly_equal :: strategy where
  "exactly_equal t1ID t2ID e = bool2nat ((get_by_ids e t1ID) = (get_by_ids e t2ID))"
```

Another simple strategy is to look at the labels and arities of outgoing transitions of each state. Pairs of states are ranked by how many transitions with the same label and arity they have in common.

```
definition naive_score :: strategy where
  "naive_score t1ID t2ID e = (
  let
    t1 = get_by_ids e t1ID;
```

```
    t2 = get_by_ids e t2ID
  in
  bool2nat (Label t1 = Label t2 ∧ Arity t1 = Arity t2 ∧ length (Outputs t1) = length (Outputs t2))
)"
```

Building off the above strategy, it makes sense to give transitions an extra "bonus point" if they are exactly equal.

```
definition naive_score_eq_bonus :: strategy where
  "naive_score_eq_bonus t1ID t2ID e = (
  let
    t1 = get_by_ids e t1ID;
    t2 = get_by_ids e t2ID
  in
  score_transitions t1 t2
)"
```

Another strategy is to assign bonus points for each shared output.

```
definition naive_score_outputs :: strategy where
  "naive_score_outputs t1ID t2ID e = (
  let
    t1 = get_by_ids e t1ID;
    t2 = get_by_ids e t2ID
  in
    bool2nat (Label t1 = Label t2) + bool2nat (Arity t1 = Arity t2) + bool2nat (Outputs t1 = Outputs t2)
)"
```

Along similar lines, we can assign additional bonus points for shared guards.

```
definition naive_score_comprehensive :: strategy where
  "naive_score_comprehensive t1ID t2ID e = (
  let
    t1 = get_by_ids e t1ID;
    t2 = get_by_ids e t2ID
  in
  if Label t1 = Label t2 ∧ Arity t1 = Arity t2 then
    if length (Outputs t1) = length (Outputs t2) then
      card (set (Guards t1) ∩ set (Guards t2)) + length (filter (λ(p1, p2). p1 = p2) (zip (Outputs t1)
(Outputs t2)))
    else 0
  else 0
)"
```

This strategy is similar to the one above except that transitions which are exactly equal get 100 bonus points.

```
definition naive_score_comprehensive_eq_high :: strategy where
  "naive_score_comprehensive_eq_high t1ID t2ID e = (
  let
    t1 = get_by_ids e t1ID;
    t2 = get_by_ids e t2ID
  in
  if t1 = t2 then
    100
  else
    if Label t1 = Label t2 ∧ Arity t1 = Arity t2 then
      if length (Outputs t1) = length (Outputs t2) then
        card (set (Guards t1) ∩ set (Guards t2)) + length (filter (λ(p1, p2). p1 = p2) (zip (Outputs t1)
(Outputs t2)))
      else 0
    else 0
)"
```

We can incorporate the subsumption relation into the scoring of merges such that a pair of states receives one point for each pair of transitions where one directly subsumes the other.

```
definition naive_score_subsumption :: "strategy" where
```

```
  "naive_score_subsumption t1ID t2ID e = (
  let
    t1 = get_by_ids e t1ID;
    t2 = get_by_ids e t2ID;
    s = origin t1ID e
  in
  bool2nat (directly_subsumes (tm e) (tm e) s s t1 t2) + bool2nat (directly_subsumes (tm e) (tm e) s s t2
t1)
)"
```

An alternative strategy is to simply score merges based on the states' proximity to the origin.

definition *leaves :: strategy* **where**

```
  "leaves t1ID t2ID e = (
    let
      t1 = get_by_ids e t1ID;
      t2 = get_by_ids e t2ID
    in
    if (Label t1 = Label t2 ∧ Arity t1 = Arity t2 ∧ length (Outputs t1) = length (Outputs t2)) then
      origin t1ID e + origin t2ID e
    else
      0)"
```

end

3 Heuristics

As part of this inference technique, we make use of certain *heuristics* to abstract away concrete values into registers. This allows us to generalise from examples of behaviour. These heuristics are as follows.

Store and Reuse - This heuristic aims to recognise when input values are subsequently used as an output. Such behaviour is generalised by storing the relevant input in a register, and replacing the literal output with the content of the register. This enables the EFSM to *predict* how the underlying system might behave when faced with unseen inputs.

Increment and Reset - This heuristic is a naive attempt to introduce additive behaviour. The idea here is that if we want to merge two transitions with identical input values and different numeric outputs, for example $coin : 1[i_0 = 50]/o_0 := 50$ and $coin : 1[i_0 = 50]/o_0 := 100$, then the behaviour must depend on the value of an internal variable. This heuristic works by dropping the input guard and adding an update to a fresh register, in this case summing the current register value with the input. A similar principle can be applied to other numeric functions such as subtraction.

Same Register - Because of the way heuristics are applied, it is possible for different registers to be introduced to serve the same purpose. This heuristic attempts to identify when this has happened and merge the two registers.

Least Upper Bound - In certain situations, transitions may produce the same output for different inputs. This technique forms the least upper bound of the transition guards — their disjunction — such that they can be merged into a single behaviour.

Distinguishing Guards - Under certain circumstances, we explicitly do not want to merge two transitions into one. This heuristic resolves nondeterminism between transitions by attempting to find apply mutually exclusive guards to each transition such that their behaviour is distinguished.

3.1 Store and Reuse (Store_Reuse)

An obvious candidate for generalisation is the "store and reuse" pattern. This manifests itself when the input of one transition is subsequently used as the output of another. Recognising this usage pattern allows us to introduce a *storage register* to abstract away concrete data values and replace two transitions whose outputs differ with a single transition that outputs the content of the register.

```
theory Store_Reuse
imports "../Inference"
begin
datatype ioTag = In | Out

instantiation ioTag :: linorder begin
fun less_ioTag :: "ioTag ⇒ ioTag ⇒ bool" where
  "In < Out = True" |
  "Out < _ = False" |
  "In < In = False"

definition less_eq_ioTag :: "ioTag ⇒ ioTag ⇒ bool" where
  "less_eq_ioTag x y = (x < y ∨ x = y)"
declare less_eq_ioTag_def [simp]

instance
  ⟨proof⟩
end
```

```
type_synonym index = "nat × ioTag × nat"

fun lookup :: "index ⇒ trace ⇒ value" where
  "lookup (actionNo, In, inx) t = (let (_, inputs, _) = nth t actionNo in nth inputs inx)" |
  "lookup (actionNo, Out, inx) t = (let (_, _, outputs) = nth t actionNo in nth outputs inx)"

abbreviation actionNum :: "index ⇒ nat" where
  "actionNum i ≡ fst i"

abbreviation ioTag :: "index ⇒ ioTag" where
  "ioTag i ≡ fst (snd i)"

abbreviation inx :: "index ⇒ nat" where
  "inx i ≡ snd (snd i)"

primrec index :: "value list ⇒ nat ⇒ ioTag ⇒ nat ⇒ index fset" where
  "index [] _ _ _ = {|}" |
  "index (h#t) actionNo io ind = finsert (actionNo, io, ind) (index t actionNo io (ind + 1))"

definition io_index :: "nat ⇒ value list ⇒ value list ⇒ index fset" where
  "io_index actionNo inputs outputs = (index inputs actionNo In 0) |∪| (index outputs actionNo Out 0)"

definition indices :: "trace ⇒ index fset" where
  "indices e = foldl (|∪|) {|} (map (λ(actionNo, (label, inputs, outputs)). io_index actionNo inputs outputs)
(enumerate 0 e))"

definition get_by_id_intratrace_matches :: "trace ⇒ (index × index) fset" where
  "get_by_id_intratrace_matches e = ffilter (λ(a, b). lookup a e = lookup b e ∧ actionNum a ≤ actionNum
b ∧ a ≠ b) (indices e |×| indices e)"


definition i_step :: "execution ⇒ iEFSM ⇒ cfstate ⇒ registers ⇒ label ⇒ inputs ⇒ (transition × cfstate
× tids × registers) option" where
  "i_step tr e s r l i = (let
    poss_steps = (i_possible_steps e s r l i);
    possibilities = ffilter (λ(u, s', t). recognises_execution (tm e) s' (evaluate_updates t i r) tr) poss_steps
in
    case random_member possibilities of
      None ⇒ None |
      Some (u, s', t) ⇒
      Some (t, s', u, (evaluate_updates t i r))
  )"

type_synonym match = "(((transition × tids) × ioTag × nat) × ((transition × tids) × ioTag × nat))"

definition "exec2trace t = map (λ(label, inputs, _). (label, inputs)) t"
primrec (nonexhaustive) walk_up_to :: "nat ⇒ iEFSM ⇒ nat ⇒ registers ⇒ trace ⇒ (transition × tids)"
where
  "walk_up_to n e s r (h#t) =
    (case (i_step (exec2trace t) e s r (fst h) (fst (snd h))) of
      (Some (transition, s', uid, updated)) ⇒ (case n of 0 ⇒ (transition, uid) | Suc m ⇒ walk_up_to m
e s' updated t)
    )"

definition find_intertrace_matches_aux :: "(index × index) fset ⇒ iEFSM ⇒ trace ⇒ match fset" where
  "find_intertrace_matches_aux intras e t = fimage (λ((e1, io1, inx1), (e2, io2, inx2)). (((walk_up_to e1
e 0 <> t), io1, inx1), ((walk_up_to e2 e 0 <> t), io2, inx2))) intras"

definition find_intertrace_matches :: "log ⇒ iEFSM ⇒ match list" where
  "find_intertrace_matches l e = filter (λ((e1, io1, inx1), (e2, io2, inx2)). e1 ≠ e2) (concat (map (λ(t,
m). sorted_list_of_fset (find_intertrace_matches_aux m e t)) (zip l (map get_by_id_intratrace_matches l))))"

definition total_max_input :: "iEFSM ⇒ nat" where
```

```
  "total_max_input e = (case EFSM.max_input (tm e) of None ⇒ 0 | Some i ⇒ i)"

definition total_max_reg :: "iEFSM ⇒ nat" where
  "total_max_reg e = (case EFSM.max_reg (tm e) of None ⇒ 0 | Some i ⇒ i)"

definition remove_guard_add_update :: "transition ⇒ nat ⇒ nat ⇒ transition" where
  "remove_guard_add_update t inputX outputX = (|
    Label = (Label t), Arity = (Arity t),
    Guards = (filter (λg. ¬ gexp_constrains g (V (vname.I inputX))) (Guards t)),
    Outputs = (Outputs t),
    Updates = (outputX, (V (vname.I inputX)))#(Updates t)
  |)"

definition generalise_output :: "transition ⇒ nat ⇒ nat ⇒ transition" where
  "generalise_output t regX outputX = (|
      Label = (Label t),
      Arity = (Arity t),
      Guards = (Guards t),
      Outputs = list_update (Outputs t) outputX (V (R regX)),
      Updates = (Updates t)
    |)"

definition is_generalised_output_of :: "transition ⇒ transition ⇒ nat ⇒ nat ⇒ bool" where
  "is_generalised_output_of t' t r p = (t' = generalise_output t r p)"

primrec count :: "'a ⇒ 'a list ⇒ nat" where
  "count _ [] = 0" |
  "count a (h#t) = (if a = h then 1+(count a t) else count a t)"

definition replaceAll :: "iEFSM ⇒ transition ⇒ transition ⇒ iEFSM" where
  "replaceAll e old new = fimage (λ(uid, (from, dest), t). if t = old then (uid, (from, dest), new) else
(uid, (from, dest), t)) e"

primrec generalise_transitions :: "((((transition × tids) × ioTag × nat) × (transition × tids) × ioTag
× nat) ×
     ((transition × tids) × ioTag × nat) × (transition × tids) × ioTag × nat) list ⇒ iEFSM ⇒ iEFSM"
where
  "generalise_transitions [] e = e" |
  "generalise_transitions (h#t) e = (let
    ((((orig1, u1), _), (orig2, u2), _), (((gen1, u1'), _), (gen2, u2), _)) = h in
   generalise_transitions t (replaceAll (replaceAll e orig1 gen1) orig2 gen2))"

definition strip_uids :: "(((transition × tids) × ioTag × nat) × (transition × tids) × ioTag × nat)
⇒ ((transition × ioTag × nat) × (transition × ioTag × nat))" where
  "strip_uids x = (let (((t1, u1), io1, in1), (t2, u2), io2, in2) = x in ((t1, io1, in1), (t2, io2, in2)))"

definition modify :: "match list ⇒ tids ⇒ tids ⇒ iEFSM ⇒ iEFSM option" where
  "modify matches u1 u2 old = (let relevant = filter (λ(((_, u1'), io, _), (_, u2'), io', _). io = In ∧
io' = Out ∧ (u1 = u1' ∨ u2 = u1' ∨ u1 = u2' ∨ u2 = u2')) matches;
                                   newReg = case max_reg old of None ⇒ 1 | Some r ⇒ r + 1;
                                   replacements = map (λ(((t1, u1), io1, inx1), (t2, u2), io2, inx2). (((remove_gu
t1 inx1 newReg, u1), io1, inx1), (generalise_output t2 newReg inx2, u2), io2, inx2)) relevant;
                                   comparisons = zip relevant replacements;
                                   stripped_replacements = map strip_uids replacements;
                                   to_replace = filter (λ(_, s). count (strip_uids s) stripped_replacements
> 1) comparisons in
                                 if to_replace = [] then None else Some ((generalise_transitions to_replace
old))
                               )"


definition heuristic_1 :: "log ⇒ update_modifier" where
  "heuristic_1 l t1 t2 s new _ old check = (case modify (find_intertrace_matches l old) t1 t2 new of
```

```
    None ⇒ None |
    Some e ⇒ if check (tm e) then Some e else None
  )"
```

lemma *remove_guard_add_update_preserves_outputs:*
```
  "Outputs (remove_guard_add_update t i r) = Outputs t"
```
⟨*proof*⟩

lemma *remove_guard_add_update_preserves_label:*
```
  "Label (remove_guard_add_update t i r) = Label t"
```
⟨*proof*⟩

lemma *remove_guard_add_update_preserves_arity:*
```
  "Arity (remove_guard_add_update t i r) = Arity t"
```
⟨*proof*⟩

lemmas *remove_guard_add_update_preserves = remove_guard_add_update_preserves_label*
remove_guard_add_update_preserves_arity
remove_guard_add_update_preserves_outputs

definition *is_generalisation_of :: "transition ⇒ transition ⇒ nat ⇒ nat ⇒ bool"* **where**
```
  "is_generalisation_of t' t i r = (t' = remove_guard_add_update t i r ∧
                                    i < Arity t ∧
                                    (∃v. Eq (V (vname.I i)) (L v) ∈ set (Guards t)) ∧
                                    r ∉ set (map fst (Updates t)))"
```

lemma *generalise_output_preserves_label:*
```
  "Label (generalise_output t r p) = Label t"
```
⟨*proof*⟩

lemma *generalise_output_preserves_arity:*
```
  "Arity (generalise_output t r p) = Arity t"
```
⟨*proof*⟩

lemma *generalise_output_preserves_guard:*
```
  "Guards (generalise_output t r p) = Guards t"
```
⟨*proof*⟩

lemma *generalise_output_preserves_output_length:*
```
  "length (Outputs (generalise_output t r p)) = length (Outputs t)"
```
⟨*proof*⟩

lemma *generalise_output_preserves_updates:*
```
  "Updates (generalise_output t r p) = Updates t"
```
⟨*proof*⟩

lemmas *generalise_output_preserves = generalise_output_preserves_label*
generalise_output_preserves_arity
generalise_output_preserves_output_length
generalise_output_preserves_guard
generalise_output_preserves_updates

definition *is_proper_generalisation_of :: "transition ⇒ transition ⇒ iEFSM ⇒ bool"* **where**
```
 "is_proper_generalisation_of t' t e = (∃i ≤ total_max_input e. ∃ r ≤ total_max_reg e.
                                        is_generalisation_of t' t i r ∧
                                        (∀u ∈ set (Updates t). fst u ≠ r) ∧
                                        (∀i ≤ max_input (tm e). ∀u ∈ set (Updates t). fst u ≠ r)
                                       )"
```

definition *generalise_input :: "transition ⇒ nat ⇒ nat ⇒ transition"* **where**
```
  "generalise_input t r i = (|
      Label = Label t,
```

```
      Arity = Arity t,
      Guards = map (λg. case g of Eq (V (I i')) (L _) ⇒ if i = i' then Eq (V (I i)) (V (R r)) else g |
_ ⇒ g) (Guards t),
      Outputs = Outputs t,
      Updates = Updates t
    ⦆"

fun structural_count :: "((transition × ioTag × nat) × (transition × ioTag × nat)) ⇒ ((transition ×
ioTag × nat) × (transition × ioTag × nat)) list ⇒ nat" where
  "structural_count _ [] = 0" |
  "structural_count a (((t1', io1', i1'), (t2', io2', i2'))#t) = (
    let ((t1, io1, i1), (t2, io2, i2)) = a in
    if same_structure t1 t1' ∧ same_structure t2 t2' ∧
       io1 = io1' ∧ io2 = io2' ∧
       i1 = i1' ∧ i2 = i2'
    then
      1+(structural_count a t)
    else
      structural_count a t
    )"

definition remove_guards_add_update :: "transition ⇒ nat ⇒ nat ⇒ transition" where
  "remove_guards_add_update t inputX outputX = ⦇
    Label = (Label t), Arity = (Arity t),
    Guards = [],
    Outputs = (Outputs t),
    Updates = (outputX, (V (vname.I inputX)))#(Updates t)
  ⦆"

definition modify_2 :: "match list ⇒ tids ⇒ tids ⇒ iEFSM ⇒ iEFSM option" where
  "modify_2 matches u1 u2 old = (let relevant = filter (λ(((_, u1'), io, _), (_, u2'), io', _). io = In
∧ io' = In ∧ (u1 = u1' ∨ u2 = u1' ∨ u1 = u2' ∨ u2 = u2')) matches;
                                   newReg = case max_reg old of None ⇒ 1 | Some r ⇒ r + 1;
                                   replacements = map (λ(((t1, u1), io1, inx1), (t2, u2), io2, inx2).
                                                  (((remove_guards_add_update t1 inx1 newReg, u1), io1,
inx1),
                                                    (generalise_input t2 newReg inx2, u2), io2, inx2)) relevant;
                                   comparisons = zip relevant replacements;
                                   stripped_replacements = map strip_uids replacements;
                                   to_replace = filter (λ(_, s). structural_count (strip_uids s) stripped_replacem
> 1) comparisons in
                                 if to_replace = [] then None else Some ((generalise_transitions to_replace
old))
                               )"


definition heuristic_2 :: "log ⇒ update_modifier" where
  "heuristic_2 l t1 t2 s new _ old check = (case modify_2 (find_intertrace_matches l old) t1 t2 new of
    None ⇒ None |
    Some e ⇒ if check (tm e) then Some e else None
  )"
hide_const ioTag.In

end
```

3.1.1 Store and Reuse Subsumption

This theory provides proofs of various properties of the *store and reuse* heuristic, including the preconditions necessary for the transitions it introduces to directly subsume their ungeneralised counterparts.

```
theory Store_Reuse_Subsumption
imports Store_Reuse
begin
```

lemma *generalisation_of_preserves:*
```
"is_generalisation_of t' t i r ⟹
  Label t = Label t' ∧
  Arity t = Arity t' ∧
  (Outputs t) = (Outputs t')"
```
⟨*proof*⟩

lemma *is_generalisation_of_guard_subset:*
```
"is_generalisation_of t' t i r ⟹ set (Guards t') ⊆ set (Guards t)"
```
⟨*proof*⟩

lemma *is_generalisation_of_medial:*
```
"is_generalisation_of t' t i r ⟹
 can_take_transition t ip rg ⟶ can_take_transition t' ip rg"
```
⟨*proof*⟩

lemma *is_generalisation_of_preserves_reg:*
```
"is_generalisation_of t' t i r ⟹
 evaluate_updates t ia c $ r = c $ r"
```
⟨*proof*⟩

lemma *apply_updates_foldr:*
```
"apply_updates u old = foldr (λh r. r(fst h $:= aval (snd h) old)) (rev u)"
```
⟨*proof*⟩

lemma *is_generalisation_of_preserves_reg_2:*
assumes *gen:* `"is_generalisation_of t' t i r"`
and *dif:* `"ra ≠ r"`
shows `"evaluate_updates t ia c $ ra = apply_updates (Updates t') (join_ir ia c) c $ ra"`
⟨*proof*⟩

lemma *is_generalisation_of_apply_guards:*
```
"is_generalisation_of t' t i r ⟹
 apply_guards (Guards t) j ⟹
 apply_guards (Guards t') j"
```
⟨*proof*⟩

If we drop the guard and add an update, and the updated register is undefined in the context, c, then the generalised transition subsumes the specific one.

lemma *is_generalisation_of_subsumes_original:*
```
"is_generalisation_of t' t i r ⟹
 c $ r = None ⟹
 subsumes t' c t"
```
⟨*proof*⟩

lemma *generalise_output_posterior:*
```
"posterior (generalise_output t p r) i ra = posterior t i ra"
```
⟨*proof*⟩

lemma *generalise_output_eq:*
```
"(Outputs t) ! r = L v ⟹
 c $ p = Some v ⟹
 evaluate_outputs t i c = apply_outputs (list_update (Outputs t) r (V (R p))) (join_ir i c)"
```
⟨*proof*⟩

This shows that if we can guarantee that the value of a particular register is the literal output then the generalised output subsumes the specific output.

lemma *generalise_output_subsumes_original:*
```
"Outputs t ! r = L v ⟹
 c $ p = Some v ⟹
 subsumes (generalise_output t p r) c t"
```
⟨*proof*⟩

```
primrec stored_reused_aux_per_reg :: "transition ⇒ transition ⇒ nat ⇒ nat ⇒ (nat × nat) option" where
  "stored_reused_aux_per_reg t' t 0 p = (
    if is_generalised_output_of t' t 0 p then
      Some (0, p)
    else
       None
  )" |
  "stored_reused_aux_per_reg t' t (Suc r) p = (
    if is_generalised_output_of t' t (Suc r) p then
      Some (Suc r, p)
    else
      stored_reused_aux_per_reg t' t r p
  )"

primrec stored_reused_aux :: "transition ⇒ transition ⇒ nat ⇒ nat ⇒ (nat × nat) option" where
  "stored_reused_aux t' t r 0 = stored_reused_aux_per_reg t' t r 0" |
  "stored_reused_aux t' t r (Suc p) = (case stored_reused_aux_per_reg t' t r (Suc p) of
                                         Some x ⇒ Some x |
                                         None ⇒ stored_reused_aux t' t r p
                                       )"

definition stored_reused :: "transition ⇒ transition ⇒ (nat × nat) option" where
  "stored_reused t' t = stored_reused_aux t' t (max (Transition.total_max_reg t) (Transition.total_max_reg
t')) (max (length (Outputs t)) (length (Outputs t')))"

lemma stored_reused_aux_is_generalised_output_of:
"stored_reused_aux t' t mr mp = Some (p, r) ⟹
   is_generalised_output_of t' t p r"
⟨proof⟩

lemma stored_reused_is_generalised_output_of:
  "stored_reused t' t = Some (p, r) ⟹
   is_generalised_output_of t' t p r"
  ⟨proof⟩

lemma is_generalised_output_of_subsumes:
  "is_generalised_output_of t' t r p ⟹
   nth (Outputs t) p = L v ⟹
   c $ r = Some v ⟹
   subsumes t' c t"
  ⟨proof⟩

lemma lists_neq_if:
  "∃i. l ! i ≠ l' ! i ⟹ l ≠ l'"
  ⟨proof⟩

lemma is_generalised_output_of_does_not_subsume:
  "is_generalised_output_of t' t r p ⟹
   p < length (Outputs t) ⟹
   nth (Outputs t) p = L v ⟹
   c $ r ≠ Some v ⟹
   ∃i. can_take_transition t i c ⟹
   ¬subsumes t' c t"
  ⟨proof⟩
```

This shows that we can use the model checker to test whether the relevant register is the correct value for direct subsumption.

```
lemma generalise_output_directly_subsumes_original:
      "stored_reused t' t = Some (r, p) ⟹
       nth (Outputs t) p = L v ⟹
      (∀c1 c2 t. obtains s c1 e1 0 <> t ∧ obtains s' c2 e2 0 <> t ⟶ c2 $ r = Some v) ⟹
       directly_subsumes e1 e2 s s' t' t"
  ⟨proof⟩
```

definition *"generalise_output_context_check v r s1 s2 e1 e2 =*
(∀ c1 c2 t. obtains s1 c1 (tm e1) 0 <> t ∧ obtains s2 c2 (tm e2) 0 <> t ⟶ c2 $ r = Some v)"

lemma *generalise_output_context_check_directly_subsumes_original:*
"stored_reused t' t = Some (r, p) ⟹
nth (Outputs t) p = L v ⟹
generalise_output_context_check v r s s' e1 e2 ⟹
directly_subsumes (tm e1) (tm e2) s s' t' t "
⟨*proof*⟩

definition *generalise_output_direct_subsumption :: "transition ⇒ transition ⇒ iEFSM ⇒ iEFSM ⇒ nat ⇒ nat ⇒ bool"* **where**
"generalise_output_direct_subsumption t' t e e' s s' = (case stored_reused t' t of
None ⇒ False |
Some (r, p) ⇒
(case nth (Outputs t) p of
L v ⇒ generalise_output_context_check v r s s' e e' |
_ ⇒ False)
)"

This allows us to just run the two functions for quick subsumption.

lemma *generalise_output_directly_subsumes_original_executable:*
"generalise_output_direct_subsumption t' t e e' s s' ⟹
directly_subsumes (tm e) (tm e') s s' t' t"
⟨*proof*⟩

lemma *original_does_not_subsume_generalised_output:*
"stored_reused t' t = Some (p, r) ⟹
r < length (Outputs t) ⟹
nth (Outputs t) r = L v ⟹
∃ a c1 tt. obtains s c1 e1 0 <> tt ∧ obtains s' a e 0 <> tt ∧ a $ p ≠ Some v ∧ (∃ i. can_take_transition t i a) ⟹
¬directly_subsumes e1 e s s' t' t"
⟨*proof*⟩

primrec *input_i_stored_in_reg :: "transition ⇒ transition ⇒ nat ⇒ nat ⇒ (nat × nat) option"* **where**
"input_i_stored_in_reg t' t i 0 = (if is_generalisation_of t' t i 0 then Some (i, 0) else None)" |
"input_i_stored_in_reg t' t i (Suc r) = (if is_generalisation_of t' t i (Suc r) then Some (i, (Suc r)) else input_i_stored_in_reg t' t i r)"

primrec *input_stored_in_reg_aux :: "transition ⇒ transition ⇒ nat ⇒ nat ⇒ (nat × nat) option"* **where**
"input_stored_in_reg_aux t' t 0 r = input_i_stored_in_reg t' t 0 r" |
"input_stored_in_reg_aux t' t (Suc i) r = (case input_i_stored_in_reg t' t (Suc i) r of
None ⇒ input_i_stored_in_reg t' t i r |
Some (i, r) ⇒ Some (i, r)
) "

definition *input_stored_in_reg :: "transition ⇒ transition ⇒ iEFSM ⇒ (nat × nat) option"* **where**
"input_stored_in_reg t' t e = (
case input_stored_in_reg_aux t' t (total_max_reg e) (max (Arity t) (Arity t')) of
None ⇒ None |
Some (i, r) ⇒
if length (filter (λ(r', u). r' = r) (Updates t')) = 1 then
Some (i, r)
else None
)"

definition *initially_undefined_context_check :: "transition_matrix ⇒ nat ⇒ nat ⇒ bool"* **where**
"initially_undefined_context_check e r s = (∀ t a. obtains s a e 0 <> t ⟶ a $ r = None)"

lemma *no_incoming_to_zero:*
```
"∀((from, to), t)|∈|e. 0 < to ⟹
     (aaa, ba) |∈| possible_steps e s d l i ⟹
     aaa ≠ 0"
```
⟨*proof*⟩

lemma *no_return_to_zero:*
```
"∀((from, to), t)|∈|e. 0 < to ⟹
 ∀r n. ¬ visits 0 e (Suc n) r t"
```
⟨*proof*⟩

lemma *no_accepting_return_to_zero:*
```
"∀((from, to), t)|∈|e. to ≠ 0 ⟹
 recognises (e) (a#t) ⟹
 ¬visits 0 (e) 0 <> (a#t)"
```
⟨*proof*⟩

lemma *no_return_to_zero_must_be_empty:*
```
"∀((from, to), t)|∈|e. to ≠ 0 ⟹
 obtains 0 a e s r t ⟹
 t = []"
```
⟨*proof*⟩

definition *"no_illegal_updates t r = (∀u ∈ set (Updates t). fst u ≠ r)"*

lemma *input_stored_in_reg_aux_is_generalisation_aux:*
```
"input_stored_in_reg_aux t' t mr mi = Some (i, r) ⟹
 is_generalisation_of t' t i r"
```
⟨*proof*⟩

lemma *input_stored_in_reg_is_generalisation:*
```
"input_stored_in_reg t' t e = Some (i, r) ⟹ is_generalisation_of t' t i r"
```
⟨*proof*⟩

lemma *generalised_directly_subsumes_original:*
```
"input_stored_in_reg t' t e = Some (i, r) ⟹
 initially_undefined_context_check (tm e) r s' ⟹
 no_illegal_updates t r ⟹
 directly_subsumes (tm e1) (tm e) s s' t' t"
```
⟨*proof*⟩

definition *drop_guard_add_update_direct_subsumption :: "transition ⇒ transition ⇒ iEFSM ⇒ nat ⇒ bool"*
where
```
"drop_guard_add_update_direct_subsumption t' t e s' = (
  case input_stored_in_reg t' t e of
    None ⇒ False |
    Some (i, r) ⇒
      if no_illegal_updates t r then
        initially_undefined_context_check (tm e) r s'
      else False
  )"
```

lemma *drop_guard_add_update_direct_subsumption_implies_direct_subsumption:*
```
"drop_guard_add_update_direct_subsumption t' t e s' ⟹
 directly_subsumes (tm e1) (tm e) s s' t' t"
```
⟨*proof*⟩

lemma *is_generalisation_of_constrains_input:*
```
"is_generalisation_of t' t i r ⟹
 ∃v. gexp.Eq (V (vname.I i)) (L v) ∈ set (Guards t)"
```
⟨*proof*⟩

lemma *is_generalisation_of_derestricts_input:*
```
"is_generalisation_of t' t i r ⟹
 ∀g ∈ set (Guards t'). ¬ gexp_constrains g (V (vname.I i))"
```
⟨*proof*⟩

lemma *is_generalisation_of_same_arity:*
```
"is_generalisation_of t' t i r ⟹ Arity t = Arity t'"
```
⟨*proof*⟩

lemma *is_generalisation_of_i_lt_arity:*
```
"is_generalisation_of t' t i r ⟹ i < Arity t"
```
⟨*proof*⟩

lemma
```
"∀i. ¬ can_take_transition t i r ∧ ¬ can_take_transition t' i r ⟹
      Label t = Label t' ⟹
      Arity t = Arity t' ⟹
      subsumes t' r t"
```
⟨*proof*⟩

lemma *input_not_constrained_aval_swap_inputs:*
```
"¬ aexp_constrains a (V (I v)) ⟹ aval a (join_ir i c) = aval a (join_ir (list_update i v x) c)"
```
⟨*proof*⟩

lemma *aval_unconstrained:*
```
" ¬ aexp_constrains a (V (vname.I i)) ⟹
i < length ia ⟹
v = ia ! i ⟹
v' ≠ v ⟹
aval a (join_ir ia c) = aval a (join_ir (list_update ia i v') c)"
```
⟨*proof*⟩

lemma *input_not_constrained_gval_swap_inputs:*
```
"¬ gexp_constrains a (V (I v)) ⟹
 gval a (join_ir i c) = gval a (join_ir (i[v := x]) c)"
```
⟨*proof*⟩

If input i is stored in register r by transition t then if we can take transition, t' then for some input ia then transition t does not subsume t'.

lemma *input_stored_in_reg_not_subsumed:*
```
"input_stored_in_reg t' t e = Some (i, r) ⟹
 ∃ia. can_take_transition t' ia c ⟹
 ¬ subsumes t c t'"
```
⟨*proof*⟩

lemma *aval_updated:*
```
"(r, u) ∈ set U ⟹
 r ∉ set (map fst (removeAll (r, u) U)) ⟹
 apply_updates U s c $ r = aval u s"
```
⟨*proof*⟩

lemma *can_take_append_subset:*
```
"set (Guards t') ⊂ set (Guards t) ⟹
can_take a (Guards t @ Guards t') ia c = can_take a (Guards t) ia c"
```
⟨*proof*⟩

Transitions of the form $t = select : 1[i_0 = x]$ do not subsume transitions of the form $t' = select : 1/r_1 := i_1$.

lemma *general_not_subsume_orig:*
```
"Arity t' = Arity t ⟹
 set (Guards t') ⊂ set (Guards t) ⟹
 (r, (V (I i))) ∈ set (Updates t') ⟹
 r ∉ set (map fst (removeAll (r, V (I i)) (Updates t'))) ⟹
 r ∉ set (map fst (Updates t)) ⟹
 ∃i. can_take_transition t i c ⟹
```

```
   c $ r = None ⟹
   i < Arity t ⟹
   ¬ subsumes t c t'"
```
⟨*proof*⟩

lemma *input_stored_in_reg_updates_reg:*
```
  "input_stored_in_reg t2 t1 a = Some (i, r) ⟹
   (r, V (I i)) ∈ set (Updates t2)"
```
⟨*proof*⟩

definition
```
"diff_outputs_ctx e1 e2 s1 s2 t1 t2 =
  (if Outputs t1 = Outputs t2 then False else
  (∃p c1 r. obtains s1 c1 e1 0 <> p ∧
       obtains s2 r e2 0 <> p ∧
       (∃i. can_take_transition t1 i r ∧ can_take_transition t2 i r ∧
       evaluate_outputs t1 i r ≠ evaluate_outputs t2 i r)
  ))"
```

lemma *diff_outputs_direct_subsumption:*
```
  "diff_outputs_ctx e1 e2 s1 s2 t1 t2 ⟹
   ¬ directly_subsumes e1 e2 s1 s2 t1 t2"
```
⟨*proof*⟩

definition *not_updated :: "nat ⇒ transition ⇒ bool"* **where**
```
  "not_updated r t = (filter (λ(r', _). r' = r) (Updates t) = [])"
```

lemma *not_updated:* **assumes** *"not_updated r t2"*
shows *"apply_updates (Updates t2) s s' $ r = s' $ r"*
⟨*proof*⟩

lemma *one_extra_update_subsumes:*
```
"Label t1 = Label t2 ⟹
   Arity t1 = Arity t2 ⟹
   set (Guards t1) ⊆ set (Guards t2) ⟹
   Outputs t1 = Outputs t2 ⟹
   Updates t1 = (r, u) # Updates t2 ⟹
   not_updated r t2 ⟹
   c $ r = None ⟹
   subsumes t1 c t2"
```
⟨*proof*⟩

lemma *one_extra_update_directly_subsumes:*
```
  "Label t1 = Label t2 ⟹
   Arity t1 = Arity t2 ⟹
   set (Guards t1) ⊆ set (Guards t2) ⟹
   Outputs t1 = Outputs t2 ⟹
   Updates t1 = (r, u)#(Updates t2) ⟹
   not_updated r t2 ⟹
   initially_undefined_context_check e2 r s2 ⟹
   directly_subsumes e1 e2 s1 s2 t1 t2"
```
⟨*proof*⟩

definition
```
"one_extra_update t1 t2 s2 e2 = (
  Label t1 = Label t2 ∧
  Arity t1 = Arity t2 ∧
  set (Guards t1) ⊆ set (Guards t2) ∧
  Outputs t1 = Outputs t2 ∧
  Updates t1 ≠ [] ∧
  tl (Updates t1) = (Updates t2) ∧
  (∃r ∈ set (map fst (Updates t1)). fst (hd (Updates t1)) = r ∧
  not_updated r t2 ∧
  initially_undefined_context_check e2 r s2)
)"
```

lemma *must_be_an_update:*
```
"U1 ≠ [] ⟹
 fst (hd U1) = r ∧ tl U1 = U2 ⟹
 ∃u. U1 = (r, u)#(U2)"
```
⟨*proof*⟩

lemma *one_extra_update_direct_subsumption:*
```
"one_extra_update t1 t2 s2 e2 ⟹ directly_subsumes e1 e2 s1 s2 t1 t2"
```
⟨*proof*⟩

end

3.2 Increment and Reset (Increment_Reset)

The "increment and reset" heuristic proposed in [2] is a naive way of introducing an incrementing register into a model. This this theory implements that heuristic.

theory *Increment_Reset*
imports *"../Inference"*
begin

definition *initialiseReg :: "transition ⇒ nat ⇒ transition"* **where**
```
"initialiseReg t newReg = (|Label = Label t, Arity = Arity t, Guards = Guards t, Outputs = Outputs t, Updates
= ((newReg, L (Num 0))#Updates t)|)"
```

definition *"guardMatch t1 t2 = (∃n n'. Guards t1 = [gexp.Eq (V (vname.I 0)) (L (Num n))] ∧ Guards t2 =
[gexp.Eq (V (vname.I 0)) (L (Num n'))])"*
definition *"outputMatch t1 t2 = (∃m m'. Outputs t1 = [L (Num m)] ∧ Outputs t2 = [L (Num m')])"*

lemma *guard_match_commute: "guardMatch t1 t2 = guardMatch t2 t1"*
⟨*proof*⟩

lemma *guard_match_length:*
```
"length (Guards t1) ≠ 1 ∨ length (Guards t2) ≠ 1 ⟹ ¬ guardMatch t1 t2"
```
⟨*proof*⟩

fun *insert_increment :: update_modifier* **where**
```
"insert_increment t1ID t2ID s new _ old check = (let
   t1 = get_by_ids new t1ID;
   t2 = get_by_ids new t2ID in
   if guardMatch t1 t2 ∧ outputMatch t1 t2 then let
        r = case max_reg new of None ⇒ 1 | Some r ⇒ r+ 1;
        newReg = R r;
        newT1 = (|Label = Label t1, Arity = Arity t1, Guards = [], Outputs = [Plus (V newReg) (V (vname.I
0))], Updates=((r, Plus (V newReg) (V (vname.I 0)))#Updates t1)|);
        newT2 = (|Label = Label t2, Arity = Arity t2, Guards = [], Outputs = [Plus (V newReg) (V (vname.I
0))], Updates=((r, Plus (V newReg) (V (vname.I 0)))#Updates t2)|);
        to_initialise = ffilter (λ(uid, (from, to), t). (to = dest t1ID new ∨ to = dest t2ID new) ∧ t
≠ t1 ∧ t ≠ t2) new;
        initialisedTrans = fimage (λ(uid, (from, to), t). (uid, initialiseReg t r)) to_initialise;
        initialised = replace_transitions new (sorted_list_of_fset initialisedTrans);
        rep = replace_transitions new [(t1ID, newT1), (t2ID, newT2)]
   in
        if check (tm rep) then Some rep else None
   else
     None
   )"
```

definition *struct_replace_all :: "iEFSM ⇒ transition ⇒ transition ⇒ iEFSM"* **where**
```
"struct_replace_all e old new = (let
  to_replace = ffilter (λ(uid, (from, dest), t). same_structure t old) e;
  replacements = fimage (λ(uid, (from, to), t). (uid, new)) to_replace
  in
```

```
    replace_transitions e (sorted_list_of_fset replacements))"

lemma output_match_symmetry: "(outputMatch t1 t2) = (outputMatch t2 t1)"
  ⟨proof⟩

lemma guard_match_symmetry: "(guardMatch t1 t2) = (guardMatch t2 t1)"
  ⟨proof⟩

fun insert_increment_2 :: update_modifier where
  "insert_increment_2 t1ID t2ID s new _ old check = (let
     t1 = get_by_ids new t1ID;
     t2 = get_by_ids new t2ID in
     if guardMatch t1 t2 ∧ outputMatch t1 t2 then let
          r = case max_reg new of None ⇒ 1 | Some r ⇒ r + 1;
          newReg = R r;
          newT1 = (|Label = Label t1, Arity = Arity t1, Guards = [], Outputs = [Plus (V newReg) (V (vname.I
0))], Updates=((r, Plus (V newReg) (V (vname.I 0)))#Updates t1)|);
          newT2 = (|Label = Label t2, Arity = Arity t2, Guards = [], Outputs = [Plus (V newReg) (V (vname.I
0))], Updates=((r, Plus (V newReg) (V (vname.I 0)))#Updates t2)|);
          to_initialise = ffilter (λ(uid, (from, to), t). (to = dest t1ID new ∨ to = dest t2ID new) ∧ t
≠ t1 ∧ t ≠ t2) new;
          initialisedTrans = fimage (λ(uid, (from, to), t). (uid, initialiseReg t r)) to_initialise;
          initialised = replace_transitions new (sorted_list_of_fset initialisedTrans);
          rep = struct_replace_all (struct_replace_all initialised t2 newT2) t1 newT1
      in
          if check (tm rep) then Some rep else None
     else
       None
     )"

fun guardMatch_alt_2 :: "vname gexp list ⇒ bool" where
  "guardMatch_alt_2 [(gexp.Eq (V (vname.I i)) (L (Num n)))] = (i = 1)" |
  "guardMatch_alt_2 _ = False"

fun outputMatch_alt_2 :: "vname aexp list ⇒ bool" where
  "outputMatch_alt_2 [(L (Num n))] = True" |
  "outputMatch_alt_2 _ = False"

end
```

3.3 Same Register (Same_Register)

The `same_register` heuristic aims to replace registers which are used in the same way.

```
theory Same_Register
  imports "../Inference"
begin

definition replace_with :: "iEFSM ⇒ nat ⇒ nat ⇒ iEFSM" where
  "replace_with e r1 r2 = (fimage (λ(u, tf, t). (u, tf,Transition.rename_regs (id(r1:=r2)) t)) e)"

fun merge_if_same :: "iEFSM ⇒ (transition_matrix ⇒ bool) ⇒ (nat × nat) list ⇒ iEFSM" where
  "merge_if_same e _ [] = e" |
  "merge_if_same e check ((r1, r2)#rs) = (
    let transitions = fimage (snd ∘ snd) e in
    if ∃(t1, t2) |∈| ffilter (λ(t1, t2). t1 < t2) (transitions |×| transitions).
      same_structure t1 t2 ∧ r1 ∈ enumerate_regs t1 ∧ r2 ∈ enumerate_regs t2
    then
      let newE = replace_with e r1 r2 in
      if check (tm newE) then
        merge_if_same newE check rs
      else
        merge_if_same e check rs
    else
```

```
    merge_if_same e check rs
  )"

definition merge_regs :: "iEFSM ⇒ (transition_matrix ⇒ bool) ⇒ iEFSM" where
  "merge_regs e check = (
    let
      regs = all_regs e;
      reg_pairs = sorted_list_of_set (Set.filter (λ(r1, r2). r1 < r2) (regs × regs))
    in
    merge_if_same e check reg_pairs
  )"

fun same_register :: update_modifier where
  "same_register t1ID t2ID s new _ old check = (
    let new' = merge_regs new check in
    if new' = new then None else Some new'
   )"

end
```

3.4 Least Upper Bound (Least_Upper_Bound)

The simplest way to merge a pair of transitions with identical outputs and updates is to simply take the least upper bound of their *guards*. This theory presents several variants on this theme.

```
theory Least_Upper_Bound
  imports "../Inference"
begin

fun literal_args :: "'a gexp ⇒ bool" where
  "literal_args (Bc v) = False" |
  "literal_args (Eq (V _) (L _)) = True" |
  "literal_args (In _ _) = True" |
  "literal_args (Eq _ _) = False" |
  "literal_args (Gt va v) = False" |
  "literal_args (Nor v va) = (literal_args v ∧ literal_args va)"

lemma literal_args_eq:
  "literal_args (Eq a b) ⟹ ∃v l. a = (V v) ∧ b = (L l)"
  ⟨proof⟩

definition "all_literal_args t = (∀g ∈ set (Guards t). literal_args g)"

fun merge_in_eq :: "vname ⇒ value ⇒ vname gexp list ⇒ vname gexp list" where
  "merge_in_eq v l [] = [Eq (V v) (L l)]" |
  "merge_in_eq v l ((Eq (V v') (L l'))#t) = (if v = v' ∧ l ≠ l' then (In v [l, l'])#t else (Eq (V v') (L
l'))#(merge_in_eq v l t))" |
  "merge_in_eq v l ((In v' l')#t) = (if v = v' then (In v (remdups (l#l')))#t else (In v' l')#(merge_in_eq
v l t))" |
  "merge_in_eq v l (h#t) = h#(merge_in_eq v l t)"

fun merge_in_in :: "vname ⇒ value list ⇒ vname gexp list ⇒ vname gexp list" where
  "merge_in_in v l [] = [In v l]" |
  "merge_in_in v l ((Eq (V v') (L l'))#t) = (if v = v' then (In v (List.insert l' l))#t else (Eq (V v')
(L l'))#(merge_in_in v l t))" |
  "merge_in_in v l ((In v' l')#t) = (if v = v' then (In v (List.union l l'))#t else (In v' l')#(merge_in_in
v l t))" |
  "merge_in_in v l (h#t) = h#(merge_in_in v l t)"

fun merge_guards :: "vname gexp list ⇒ vname gexp list ⇒ vname gexp list" where
  "merge_guards [] g2 = g2" |
  "merge_guards ((Eq (V v) (L l))#t) g2 =  merge_guards t (merge_in_eq v l g2)" |
  "merge_guards ((In v l)#t) g2 = merge_guards t (merge_in_in v l g2)" |
  "merge_guards (h#t) g2 = h#(merge_guards t g2)"
```

The "least upper bound" (lob) heuristic simply disjoins the guards of two transitions with identical outputs and updates.

```
definition lob_aux :: "transition ⇒ transition ⇒ transition option" where
  "lob_aux t1 t2 = (if Outputs t1 = Outputs t2 ∧ Updates t1 = Updates t2 ∧ all_literal_args t1 ∧ all_literal_args
t2 then
      Some (⦇Label = Label t1, Arity = Arity t1, Guards = remdups (merge_guards (Guards t1) (Guards t2)),
Outputs = Outputs t1, Updates = Updates t1⦈)
     else None)"

fun lob :: update_modifier where
  "lob t1ID t2ID s new _ old _ = (let
     t1 = (get_by_ids new t1ID);
     t2 = (get_by_ids new t2ID) in
     case lob_aux t1 t2 of
       None ⇒ None |
       Some lob_t ⇒
           Some (replace_transitions new [(t1ID, lob_t), (t2ID, lob_t)])
   )"

lemma lob_aux_some: "Outputs t1 = Outputs t2 ⟹
       Updates t1 = Updates t2 ⟹
       all_literal_args t1 ⟹
       all_literal_args t2 ⟹
       Label t = Label t1 ⟹
       Arity t = Arity t1 ⟹
       Guards t = remdups (merge_guards (Guards t1) (Guards t2)) ⟹
       Outputs t = Outputs t1 ⟹
       Updates t = Updates t1 ⟹
       lob_aux t1 t2 = Some t"
  ⟨proof⟩

fun has_corresponding :: "vname gexp ⇒ vname gexp list ⇒ bool" where
  "has_corresponding g [] = False" |
  "has_corresponding (Eq (V v) (L l)) ((Eq (V v') (L l'))#t) = (if v = v' ∧ l = l' then True else has_correspondin
(Eq (V v) (L l)) t)" |
  "has_corresponding (In v' l') ((Eq (V v) (L l))#t) = (if v = v' ∧ l ∈ set l' then True else has_corresponding
(In v' l') t)" |
  "has_corresponding (In v l) ((In v' l')#t) = (if v = v' ∧ set l' ⊆ set l then True else has_corresponding
(In v l) t)" |
  "has_corresponding g (h#t) = has_corresponding g t"

lemma no_corresponding_bc: "¬has_corresponding (Bc x1) G1"
  ⟨proof⟩

lemma no_corresponding_gt: "¬has_corresponding (Gt x1 y1) G1"
  ⟨proof⟩

lemma no_corresponding_nor: "¬has_corresponding (Nor x1 y1) G1"
  ⟨proof⟩

lemma has_corresponding_eq: "has_corresponding (Eq x21 x22) G1 ⟹ (Eq x21 x22) ∈ set G1"
⟨proof⟩

lemma has_corresponding_In: "has_corresponding (In v l) G1 ⟹ (∃l'. (In v l') ∈ set G1 ∧ set l' ⊆ set
l) ∨ (∃l' ∈ set l. (Eq (V v) (L l')) ∈ set G1)"
⟨proof⟩

lemma gval_each_one: "g ∈ set G ⟹ apply_guards G s ⟹ gval g s = true"
  ⟨proof⟩

lemma has_corresponding_apply_guards:
  "∀g∈set G2. has_corresponding g G1 ⟹
```

```
   apply_guards G1 s ⟹
   apply_guards G2 s"
```
⟨*proof*⟩

lemma *correspondence_subsumption:*
```
  "Label t1 = Label t2 ⟹
   Arity t1 = Arity t2 ⟹
   Outputs t1 = Outputs t2 ⟹
   Updates t1 = Updates t2 ⟹
   ∀g ∈ set (Guards t2). has_corresponding g (Guards t1) ⟹
   subsumes t2 c t1"
```
⟨*proof*⟩

definition *"is_lob t1 t2 = (*
```
  Label t1 = Label t2 ∧
  Arity t1 = Arity t2 ∧
  Outputs t1 = Outputs t2 ∧
  Updates t1 = Updates t2 ∧
  (∀g ∈ set (Guards t2). has_corresponding g (Guards t1)))"
```

lemma *is_lob_direct_subsumption:*
```
  "is_lob t1 t2 ⟹ directly_subsumes e1 e2 s s' t2 t1"
```
⟨*proof*⟩

fun *has_distinguishing :: "vname gexp ⇒ vname gexp list ⇒ bool"* **where**
```
  "has_distinguishing g [] = False" |
  "has_distinguishing (Eq (V v) (L l)) ((Eq (V v') (L l'))#t) = (if v = v' ∧ l ≠ l' then True else has_distinguis
(Eq (V v) (L l)) t)" |
  "has_distinguishing (In (I v') l') ((Eq (V (I v)) (L l))#t) = (if v = v' ∧ l ∉ set l' then True else
has_distinguishing (In (I v') l') t)" |
  "has_distinguishing (In (I v) l) ((In (I v') l')#t) = (if v = v' ∧ set l' ⊃ set l then True else has_distinguis
(In (I v) l) t)" |
  "has_distinguishing g (h#t) = has_distinguishing g t"
```

lemma *has_distinguishing: "has_distinguishing g G ⟹ (∃v l. g = (Eq (V v) (L l))) ∨ (∃v l. g = In v
l)"*
⟨*proof*⟩

lemma *has_distinguishing_Eq: "has_distinguishing (Eq (V v) (L l)) G ⟹ ∃l'. (Eq (V v) (L l')) ∈ set
G ∧ l ≠ l'"*
⟨*proof*⟩

lemma *ex_mutex: "Eq (V v) (L l) ∈ set G1 ⟹*
```
       Eq (V v) (L l') ∈ set G2 ⟹
       l ≠ l' ⟹
       apply_guards G1 s ⟹
       ¬ apply_guards G2 s"
```
⟨*proof*⟩

lemma *has_distinguishing_In:*
```
  "has_distinguishing (In v l) G ⟹
   (∃l' i. v = I i ∧ Eq (V v) (L l') ∈ set G ∧ l' ∉ set l) ∨ (∃l' i. v = I i ∧ In v l' ∈ set G ∧ set
l' ⊃ set l)"
```
⟨*proof*⟩

lemma *Eq_apply_guards:*
```
  "Eq (V v) (L l) ∈ set G1 ⟹
   apply_guards G1 s ⟹
   s v = Some l"
```
⟨*proof*⟩

lemma *In_neq_apply_guards:*
```
  "In v l ∈ set G2 ⟹
```

```
   Eq (V v) (L l') ∈ set G1 ⟹
   l' ∉ set l ⟹
   apply_guards G1 s ⟹
   ¬apply_guards G2 s"
```
⟨*proof*⟩

lemma *In_apply_guards:* `"In v l ∈ set G1 ⟹ apply_guards G1 s ⟹ ∃v' ∈ set l. s v = Some v'"`
⟨*proof*⟩

lemma *input_not_constrained_aval_swap_inputs:*
```
  "¬ aexp_constrains a (V (I v)) ⟹
   aval a (join_ir i c) = aval a (join_ir (list_update i v x) c)"
```
⟨*proof*⟩

lemma *input_not_constrained_gval_swap_inputs:*
```
  "¬ gexp_constrains a (V (I v)) ⟹
   gval a (join_ir i c) = gval a (join_ir (i[v := x]) c)"
```
⟨*proof*⟩

lemma *test_aux:*
```
"∀g∈set (removeAll (In (I v) l) G1). ¬ gexp_constrains g (V (I v)) ⟹
      apply_guards G1 (join_ir i c) ⟹
      x ∈ set l ⟹
      apply_guards G1 (join_ir (i[v := x]) c)"
```
⟨*proof*⟩

lemma *test:*
assumes
p1: `"In (I v) l ∈ set G2"` **and**
p2: `"In (I v) l' ∈ set G1"` **and**
p3: `"x ∈ set l'"` **and**
p4: `"x ∉ set l"` **and**
p5: `"apply_guards G1 (join_ir i c)"` **and**
p6: `"length i = a"` **and**
p7: `"∀g ∈ set (removeAll (In (I v) l') G1). ¬ gexp_constrains g (V (I v))"`
shows `"∃i. length i = a ∧ apply_guards G1 (join_ir i c) ∧ (length i = a ⟶ ¬ apply_guards G2 (join_ir i c))"`
⟨*proof*⟩

definition *get_Ins* `:: "vname gexp list ⇒ (nat × value list) list"` **where**
`"get_Ins G = map (λg. case g of (In (I v) l) ⇒ (v, l)) (filter (λg. case g of (In (I _) _ ) ⇒ True | _ ⇒ False) G)"`

lemma *get_Ins_Cons_equiv:* `"∄v l. a = In (I v) l ⟹ get_Ins (a # G) = get_Ins G"`
⟨*proof*⟩

lemma *Ball_Cons:* `"(∀x ∈ set (a#l). P x) = (P a ∧ (∀x ∈ set l. P x))"`
⟨*proof*⟩

lemma *In_in_get_Ins:* `"(In (I v) l ∈ set G) = ((v, l) ∈ set (get_Ins G))"`
⟨*proof*⟩

definition `"check_get_Ins G = (∀(v, l') ∈ set (get_Ins G). ∀g ∈ set (removeAll (In (I v) l') G). ¬ gexp_constrain g (V (I v)))"`

lemma *no_Ins:* `"[] = get_Ins G ⟹ set G - {In (I i) l} = set G"`
⟨*proof*⟩

lemma *test2:*
```
"In (I i) l ∈ set (Guards t2) ⟹
       In (I i) l' ∈ set (Guards t1) ⟹
       length ia = Arity t1 ⟹
       apply_guards (Guards t1) (join_ir ia c) ⟹
       x ∈ set l' ⟹
       x ∉ set l ⟹
```

```
      ∀(v, l')∈insert (0, []) (set (get_Ins (Guards t1))). ∀g∈set (removeAll (In (I v) l') (Guards t1)).
¬ gexp_constrains g (V (I v)) ⟹
      Arity t1 = Arity t2 ⟹
      ∃i. length i = Arity t2 ∧ apply_guards (Guards t1) (join_ir i c) ∧ (length i = Arity t2 ⟶ ¬ apply_guards
(Guards t2) (join_ir i c))"
```
⟨*proof*⟩

lemma *distinguishing_subsumption:*
assumes
p1: "∃g ∈ set (Guards t2). has_distinguishing g (Guards t1)" **and**
p2: "Arity t1 = Arity t2" **and**
p3: "∃i. can_take_transition t1 i c" **and**
p4: "(∀(v, l') ∈ insert (0, []) (set (get_Ins (Guards t1))). ∀g ∈ set (removeAll (In (I v) l') (Guards
t1)). ¬ gexp_constrains g (V (I v)))"
shows
"¬ subsumes t2 c t1"
⟨*proof*⟩

definition "lob_distinguished t1 t2 = (
(∃g ∈ set (Guards t2). has_distinguishing g (Guards t1)) ∧
Arity t1 = Arity t2 ∧
(∀(v, l') ∈ insert (0, []) (set (get_Ins (Guards t1))). ∀g ∈ set (removeAll (In (I v) l') (Guards t1)).
¬ gexp_constrains g (V (I v))))"

lemma *must_be_another:*
"1 < size (fset_of_list b) ⟹
x ∈ set b ⟹
∃x' ∈ set b. x ≠ x'"
⟨*proof*⟩

lemma *another_swap_inputs:*
"apply_guards G (join_ir i c) ⟹
filter (λg. gexp_constrains g (V (I a))) G = [In (I a) b] ⟹
xa ∈ set b ⟹
apply_guards G (join_ir (i[a := xa]) c)"
⟨*proof*⟩

lemma *lob_distinguished_2_not_subsumes:*
"∃(i, l) ∈ set (get_Ins (Guards t2)). filter (λg. gexp_constrains g (V (I i))) (Guards t2) = [(In (I
i) l)] ∧
(∃l' ∈ set l. i < Arity t1 ∧ Eq (V (I i)) (L l') ∈ set (Guards t1) ∧ size (fset_of_list l) > 1) ⟹
Arity t1 = Arity t2 ⟹
∃i. can_take_transition t2 i c ⟹
¬ subsumes t1 c t2"
⟨*proof*⟩

definition "lob_distinguished_2 t1 t2 =
(∃(i, l) ∈ set (get_Ins (Guards t2)). filter (λg. gexp_constrains g (V (I i))) (Guards t2) = [(In (I
i) l)] ∧
(∃l' ∈ set l. i < Arity t1 ∧ Eq (V (I i)) (L l') ∈ set (Guards t1) ∧ size (fset_of_list l) > 1) ∧
Arity t1 = Arity t2)"

lemma *lob_distinguished_3_not_subsumes:*
"∃(i, l) ∈ set (get_Ins (Guards t2)). filter (λg. gexp_constrains g (V (I i))) (Guards t2) = [(In (I
i) l)] ∧
(∃(i', l') ∈ set (get_Ins (Guards t1)). i = i' ∧ set l' ⊂ set l) ⟹
Arity t1 = Arity t2 ⟹
∃i. can_take_transition t2 i c ⟹
¬ subsumes t1 c t2"
⟨*proof*⟩

definition "lob_distinguished_3 t1 t2 = (∃(i, l) ∈ set (get_Ins (Guards t2)). filter (λg. gexp_constrains

```
g (V (I i))) (Guards t2) = [(In (I i) l)] ∧
    (∃(i', l') ∈ set (get_Ins (Guards t1)). i = i' ∧ set l' ⊂ set l) ∧
   Arity t1 = Arity t2)"

fun is_In :: "'a gexp ⇒ bool" where
  "is_In (In _ _) = True" |
  "is_In _ = False"
```

The "greatest upper bound" (gob) heuristic is similar to `lob` but applies a more intellegent approach to guard merging.

```
definition gob_aux :: "transition ⇒ transition ⇒ transition option" where
  "gob_aux t1 t2 = (if Outputs t1 = Outputs t2 ∧ Updates t1 = Updates t2 ∧ all_literal_args t1 ∧ all_literal_args
t2 then
      Some (|Label = Label t1, Arity = Arity t1, Guards = remdups (filter (Not ∘ is_In) (merge_guards (Guards
t1) (Guards t2))), Outputs = Outputs t1, Updates = Updates t1|)
     else None)"

fun gob :: update_modifier where
  "gob t1ID t2ID s new _ old _ = (let
     t1 = (get_by_ids new t1ID);
     t2 = (get_by_ids new t2ID) in
     case gob_aux t1 t2 of
       None ⇒ None |
       Some gob_t ⇒
           Some (replace_transitions new [(t1ID, gob_t), (t2ID, gob_t)])
   )"
```

The "Gung Ho" heuristic simply drops the guards of both transitions, making them identical.

```
definition gung_ho_aux :: "transition ⇒ transition ⇒ transition option" where
  "gung_ho_aux t1 t2 = (if Outputs t1 = Outputs t2 ∧ Updates t1 = Updates t2 ∧ all_literal_args t1 ∧ all_literal_
t2 then
      Some (|Label = Label t1, Arity = Arity t1, Guards = [], Outputs = Outputs t1, Updates = Updates t1|)
     else None)"

fun gung_ho :: update_modifier where
  "gung_ho t1ID t2ID s new _ old _ = (let
     t1 = (get_by_ids new t1ID);
     t2 = (get_by_ids new t2ID) in
     case gung_ho_aux t1 t2 of
       None ⇒ None |
       Some gob_t ⇒
           Some (replace_transitions new [(t1ID, gob_t), (t2ID, gob_t)])
   )"

lemma guard_subset_eq_outputs_updates_subsumption:
  "Label t1 = Label t2 ⟹
   Arity t1 = Arity t2 ⟹
   Outputs t1 = Outputs t2 ⟹
   Updates t1 = Updates t2 ⟹
   set (Guards t2) ⊆ set (Guards t1) ⟹
   subsumes t2 c t1"
  ⟨proof⟩

lemma guard_subset_eq_outputs_updates_direct_subsumption:
  "Label t1 = Label t2 ⟹
   Arity t1 = Arity t2 ⟹
   Outputs t1 = Outputs t2 ⟹
   Updates t1 = Updates t2 ⟹
   set (Guards t2) ⊆ set (Guards t1) ⟹
   directly_subsumes m1 m2 s1 s2 t2 t1"
  ⟨proof⟩

lemma unconstrained_input:
```

```
  "∀g∈set G. ¬ gexp_constrains g (V (I i)) ⟹
   apply_guards G (join_ir ia c) ⟹
   apply_guards G (join_ir (ia[i := x']) c)"
```
⟨*proof*⟩

lemma *each_input_guarded_once_cons:*
```
   "∀i∈⋃ (enumerate_gexp_inputs ` set (a # G)). length (filter (λg. gexp_constrains g (V (I i))) (a # G)) ≤ 1 ⟹
    ∀i∈⋃ (enumerate_gexp_inputs ` set G). length (filter (λg. gexp_constrains g (V (I i))) G) ≤ 1"
```
⟨*proof*⟩

lemma *literal_args_can_take:*
```
  "∀g∈set G. ∃i v s. g = Eq (V (I i)) (L v) ∨ g = In (I i) s ∧ s ≠ [] ⟹
   ∀i∈⋃ (enumerate_gexp_inputs ` set G). i < a ⟹
   ∀i∈⋃ (enumerate_gexp_inputs ` set G). length (filter (λg. gexp_constrains g (V (I i))) G) ≤ 1 ⟹
   ∃i. length i = a ∧ apply_guards G (join_ir i c)"
```
⟨*proof*⟩

lemma *"(SOME x'. x' ≠ (v::value)) ≠ v"*
⟨*proof*⟩

lemma *opposite_gob_subsumption:*
```
"∀g ∈ set (Guards t1). ∃i v s. g = Eq (V (I i)) (L v) ∨ (g = In (I i) s ∧ s ≠ []) ⟹
      ∀g ∈ set (Guards t2). ∃i v s. g = Eq (V (I i)) (L v) ∨ (g = In (I i) s ∧ s ≠ []) ⟹
      ∃ i. ∃v. Eq (V (I i)) (L v) ∈ set (Guards t1) ∧
        (∀g ∈ set (Guards t2). ¬ gexp_constrains g (V (I i))) ⟹
      Arity t1 = Arity t2 ⟹
      ∀i ∈ enumerate_inputs t2. i < Arity t2 ⟹
      ∀i ∈ enumerate_inputs t2. length (filter (λg. gexp_constrains g (V (I i))) (Guards t2)) ≤ 1 ⟹
      ¬ subsumes t1 c t2"
```
⟨*proof*⟩

fun *is_lit_eq :: "vname gexp ⇒ nat ⇒ bool"* **where**
```
  "is_lit_eq (Eq (V (I i)) (L v)) i' = (i = i')" |
  "is_lit_eq _ _ = False"
```

lemma *"(∃v. Eq (V (I i)) (L v) ∈ set G) = (∃g ∈ set G. is_lit_eq g i)"*
⟨*proof*⟩

fun *is_lit_eq_general :: "vname gexp ⇒ bool"* **where**
```
  "is_lit_eq_general (Eq (V (I _)) (L _)) = True" |
  "is_lit_eq_general _ = False"
```

fun *is_input_in :: "vname gexp ⇒ bool"* **where**
```
  "is_input_in (In (I i) s) = (s ≠ [])" |
  "is_input_in _ = False"
```

definition *"opposite_gob t1 t2 = (*
```
      (∀g ∈ set (Guards t1). is_lit_eq_general g ∨ is_input_in g) ∧
      (∀g ∈ set (Guards t2). is_lit_eq_general g ∨ is_input_in g) ∧
      (∃ i ∈ (enumerate_inputs t1 ∪ enumerate_inputs t2). (∃g ∈ set (Guards t1). is_lit_eq g i) ∧
        (∀g ∈ set (Guards t2). ¬ gexp_constrains g (V (I i)))) ∧
      Arity t1 = Arity t2 ∧
      (∀i ∈ enumerate_inputs t2. i < Arity t2) ∧
      (∀i ∈ enumerate_inputs t2. length (filter (λg. gexp_constrains g (V (I i))) (Guards t2)) ≤ 1))"
```

lemma *"is_lit_eq_general g ∨ is_input_in g ⟹*
```
      ∃i v s. g = Eq (V (I i)) (L v) ∨ g = In (I i) s ∧ s ≠ []"
```
⟨*proof*⟩

lemma *opposite_gob_directly_subsumption:*
```
  "opposite_gob t1 t2 ⟹ ¬ subsumes t1 c t2"
```
⟨*proof*⟩

fun *get_in :: "'a gexp ⇒ ('a × value list) option"* **where**
"get_in (In v s) = Some (v, s)" |
"get_in _ = None"

lemma *not_subset_not_in: "(¬ s1 ⊆ s2) = (∃i. i ∈ s1 ∧ i ∉ s2)"*
⟨*proof*⟩

lemma *get_in_is: "(get_in x = Some (v, s1)) = (x = In v s1)"*
⟨*proof*⟩

lemma *gval_rearrange:*
"g ∈ set G ⟹
gval g s = true ⟹
apply_guards (removeAll g G) s ⟹
apply_guards G s"
⟨*proof*⟩

lemma *singleton_list: "(length l = 1) = (∃e. l = [e])"*
⟨*proof*⟩

lemma *remove_restricted:*
"g ∈ set G ⟹
gexp_constrains g (V v) ⟹
restricted_once v G ⟹
not_restricted v (removeAll g G)"
⟨*proof*⟩

lemma *unrestricted_input_swap:*
"not_restricted (I i) G ⟹
apply_guards G (join_ir iaa c) ⟹
apply_guards (removeAll g G) (join_ir (iaa[i := ia]) c)"
⟨*proof*⟩

lemma *apply_guards_remove_restricted:*
"g ∈ set G ⟹
gexp_constrains g (V (I i)) ⟹
restricted_once (I i) G ⟹
apply_guards G (join_ir iaa c) ⟹
apply_guards (removeAll g G) (join_ir (iaa[i := ia]) c)"
⟨*proof*⟩

lemma *In_swap_inputs:*
"In (I i) s2 ∈ set G ⟹
restricted_once (I i) G ⟹
ia ∈ set s2 ⟹
apply_guards G (join_ir iaa c) ⟹
apply_guards G (join_ir (iaa[i := ia]) c)"
⟨*proof*⟩

definition *these :: "'a option list ⇒ 'a list"* **where**
"these as = map (λx. case x of Some y ⇒ y) (filter (λx. x ≠ None) as)"

lemma *these_cons: "these (a#as) = (case a of None ⇒ these as | Some x ⇒ x#(these as))"*
⟨*proof*⟩

definition *get_ins :: "vname gexp list ⇒ (nat × value list) list"* **where**
"get_ins g = map (λ(v, s). case v of I i ⇒ (i, s)) (filter (λ(v, _). case v of I _ ⇒ True | R _ ⇒ False) (these (map get_in g)))"

lemma *in_get_ins:*
"(I x1a, b) ∈ set (these (map get_in G)) ⟹
In (I x1a) b ∈ set G"

⟨*proof*⟩

lemma *restricted_head: "∀v. restricted_once v (Eq (V x2) (L x1) # G) ∨ not_restricted v (Eq (V x2) (L x1) # G) ⟹*
not_restricted x2 G"
⟨*proof*⟩

fun *atomic :: "'a gexp ⇒ bool"* **where**
"atomic (Eq (V _) (L _)) = True" |
"atomic (In _ _) = True" |
"atomic _ = False"

lemma *restricted_max_once_cons: "∀v. restricted_once v (g#gs) ∨ not_restricted v (g#gs) ⟹*
∀v. restricted_once v gs ∨ not_restricted v gs"
⟨*proof*⟩

lemma *not_restricted_swap_inputs:*
"not_restricted (I x1a) G ⟹
apply_guards G (join_ir i r) ⟹
apply_guards G (join_ir (i[x1a := x1]) r)"
⟨*proof*⟩
end

3.5 Distinguishing Guards (Distinguishing_Guards)

If we cannot resolve the nondeterminism which arises from merging states by merging transitions, we might then conclude that those states should not be merged. Alternatively, we could consider the possibility of *value-dependent* behaviour. This theory presents a heuristic which tries to find a guard which distinguishes between a pair of transitions.

theory *Distinguishing_Guards*
imports *"../Inference"*
begin

hide_const *uids*

definition *put_updates :: "tids ⇒ update_function list ⇒ iEFSM ⇒ iEFSM"* **where**
"put_updates uids updates iefsm = fimage (λ(uid, fromTo, tran).
case uid of [u] ⇒
if u ∈ set uids then
(uid, fromTo, (|Label = Label tran, Arity = Arity tran, Guards = Guards tran, Outputs = Outputs tran, Updates = (Updates tran)@updates|))
else
(uid, fromTo, tran)
) iefsm"

definition *transfer_updates :: "iEFSM ⇒ iEFSM ⇒ iEFSM"* **where**
"transfer_updates e pta = fold (λ(tids, (from, to), tran) acc. put_updates tids (Updates tran) acc) (sorted_list_e) pta"

fun *trace_collect_training_sets :: "trace ⇒ iEFSM ⇒ cfstate ⇒ registers ⇒ tids ⇒ tids ⇒ (inputs × registers) list ⇒ (inputs × registers) list ⇒ ((inputs × registers) list × (inputs × registers) list)"*
where
"trace_collect_training_sets [] uPTA s registers T1 T2 G1 G2 = (G1, G2)" |
"trace_collect_training_sets ((label, inputs, outputs)#t) uPTA s registers T1 T2 G1 G2 = (
let
(uids, s', tran) = fthe_elem (ffilter (λ(uids, s', tran). evaluate_outputs tran inputs registers = map Some outputs) (i_possible_steps uPTA s registers label inputs));
updated = (evaluate_updates tran inputs registers)
in
if hd uids ∈ set T1 then
trace_collect_training_sets t uPTA s' updated T1 T2 ((inputs, registers)#G1) G2
else if hd uids ∈ set T2 then
trace_collect_training_sets t uPTA s' updated T1 T2 G1 ((inputs, registers)#G2)

```
    else
      trace_collect_training_sets t uPTA s' updated T1 T2 G1 G2
  )"

primrec collect_training_sets :: "log ⇒ iEFSM ⇒ tids ⇒ tids ⇒ (inputs × registers) list ⇒ (inputs
× registers) list ⇒ ((inputs × registers) list × (inputs × registers) list)" where
  "collect_training_sets [] uPTA T1 T2 G1 G2 = (G1, G2)" |
  "collect_training_sets (h#t) uPTA T1 T2 G1 G2 = (
    let (G1a, G2a) = trace_collect_training_sets h uPTA 0 <> T1 T2 [] [] in
    collect_training_sets t uPTA T1 T2 (List.union G1 G1a) (List.union G2 G2a)
  )"

definition find_distinguishing_guards :: "(inputs × registers) list ⇒ (inputs × registers) list ⇒ (vname
gexp × vname gexp) option" where
  "find_distinguishing_guards G1 G2 = (
    let gs = {(g1, g2).
      (∀(i, r) ∈ set G1. gval g1 (join_ir i r) = true) ∧
      (∀(i, r) ∈ set G2. gval g2 (join_ir i r) = true) ∧
      (∀i r. ¬ (gval g1 (join_ir i r) = true ∧ gval g2 (join_ir i r) = true))
    } in
    if gs = {} then None else Some (Eps (λg. g ∈ gs))
  )"
declare find_distinguishing_guards_def [code del]
code_printing constant find_distinguishing_guards ⇀ (Scala) "Dirties.findDistinguishingGuards"

definition add_guard :: "transition ⇒ vname gexp ⇒ transition" where
  "add_guard t g = t⦇Guards := List.insert g (Guards t)⦈"

definition distinguish :: "log ⇒ update_modifier" where
  "distinguish log t1ID t2ID s destMerge preDestMerge old check = (
    let
      t1 = get_by_ids destMerge t1ID;
      t2 = get_by_ids destMerge t2ID;
      uPTA = transfer_updates destMerge (make_pta log);
      (G1, G2) = collect_training_sets log uPTA t1ID t2ID [] []
    in
      case find_distinguishing_guards G1 G2 of
        None ⇒ None |
        Some (g1, g2) ⇒ (
          let rep = replace_transitions preDestMerge [(t1ID, add_guard t1 g1), (t2ID, add_guard t2 g2)]
in
          if check (tm rep) then Some rep else None
        )
  )"

definition can_still_take_ctx :: "transition_matrix ⇒ transition_matrix ⇒ cfstate ⇒ cfstate ⇒ transition
⇒ transition ⇒ bool" where
  "can_still_take_ctx e1 e2 s1 s2 t1 t2 = (
    ∀t. recognises e1 t ∧ visits s1 e1 0 <> t ∧ recognises e2 t ∧ visits s2 e2 0 <> t ⟶
    (∀a. obtains s2 a e2 0 <> t  ∧ (∀i. can_take_transition t2 i a ⟶ can_take_transition t1 i a))
  )"

lemma distinguishing_guard_subsumption:
"Label t1 = Label t2 ⟹
 Arity t1 = Arity t2 ⟹
 Outputs t1 = Outputs t2 ⟹
 Updates t1 = Updates t2 ⟹
 can_still_take_ctx e1 e2 s1 s2 t1 t2 ⟹
 recognises e1 p ⟹
 visits s1 e1 0 <> p ⟹
 obtains s2 c e2 0 <> p ⟹
 subsumes t1 c t2"
  ⟨proof⟩
```

```
definition "recognises_and_visits_both a b s s' = (
  ∃p c1 c2. obtains s c1 a 0 <> p ∧ obtains s' c2 b 0 <> p)"

definition "can_still_take e1 e2 s1 s2 t1 t2 = (
  Label t1 = Label t2 ∧
  Arity t1 = Arity t2 ∧
  Outputs t1 = Outputs t2 ∧
  Updates t1 = Updates t2 ∧
  can_still_take_ctx e1 e2 s1 s2 t1 t2 ∧
  recognises_and_visits_both e1 e2 s1 s2)"

lemma can_still_take_direct_subsumption:
  "can_still_take e1 e2 s1 s2 t1 t2 ⟹
  directly_subsumes e1 e2 s1 s2 t1 t2"
  ⟨proof⟩

end
```

3.5.1 Weak Subsumption

Unfortunately, the *direct subsumption* relation cannot be transformed into executable code. To solve this problem, [2] advocates for the use of a model checker, but this turns out to be prohibitively slow for all but the smallest of examples. To solve this problem, we must make a practical compromise and use another heuristic: the *weak subsumption* heuristic. This heuristic simply tries to delete each transition in turn and runs the original traces used to build the PTA are still accepted. If so, this is taken as an acceptable substitute for direct subsumption.

The acceptability of this, with respect to model behaviour, is justified by the fact that the original traces are checked for acceptance. In situations where one transition genuinely does directly subsume the other, the merge will go ahead as normal. In situations where one transition does not directly subsume the other, the merge may still go ahead if replacing one transition with the other still allows the model to accept the original traces. In this case, the heuristic makes an overgeneralisation, but this is deemed to be acceptable since this is what heuristics are for. This approach is clearly not as formal as we would like, but the compromise is necessary to allow models to be inferred in reasonable time.

```
theory Weak_Subsumption
imports "../Inference"
begin

definition maxBy :: "('a ⇒ 'b::linorder) ⇒ 'a ⇒ 'a ⇒ 'a" where
  "maxBy f a b = (if (f a > f b) then a else b)"

fun weak_subsumption :: "update_modifier" where
  "weak_subsumption t1ID t2ID s new _ old check = (let
     t1 = get_by_ids new t1ID;
     t2 = get_by_ids new t2ID
     in
     if
      same_structure t1 t2
     then
      let
        maxT = maxBy (λx. ((length ∘ Updates) x, map snd (Updates x))) t1 t2;
        minT = if maxT = t1 then t2 else t1;
        newEFSMmax = replace_all new [t1ID, t2ID] maxT in
      — Most of the time, we'll want the transition with the most updates so start with that one
      if check (tm newEFSMmax) then
        Some newEFSMmax
      else
        — There may be other occasions where we want to try the other transition
        — e.g. if their updates are equal but one has a different guard
        let newEFSMmin = replace_all new [t1ID, t2ID] minT in
        if check (tm newEFSMmin) then
          Some newEFSMmin
```

```
        else
          None
      else
       None
   )"

end
theory Group_By
imports Main
begin

fun group_by :: "('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a list list" where
  "group_by f [] = []" |
  "group_by f (h#t) = (
    let
      group = (takeWhile (f h) t);
      dropped = drop (length group) t
    in
      (h#group)#(group_by f dropped)
  )"

lemma "(group_by f xs = []) = (xs = [])"
  ⟨proof⟩

lemma not_empty_group_by_drop: "∀x ∈ set (group_by f (drop l xs)). x ≠ []"
  ⟨proof⟩

lemma no_empty_groups: "∀x∈set (group_by f xs). x ≠ []"
  ⟨proof⟩

lemma "(drop (length (takeWhile f l)) l) = dropWhile f l"
  ⟨proof⟩

lemma takeWhile_dropWhile: "takeWhile f l @ dropWhile f l = l' ⟹ l' = l"
  ⟨proof⟩

lemma append_pref: "l' = l'' ⟹ (l@l' = l@l'')"
  ⟨proof⟩

lemma dropWhile_drop: "∃x. dropWhile f l = drop x l"
  ⟨proof⟩

lemma group_by_drop_foldr: "drop x l = foldr (@) (group_by f (drop x l)) []"
⟨proof⟩

lemma group_by_inverse: "foldr (@) (group_by f l) [] = l"
⟨proof⟩

end
```

3.6 PTA Generalisation (PTA_Generalisation)

The problem with the simplistic heuristics of [2] is that the performance of the Inference technique is almost entirely dependent on the quality and applicability of the heuristics provided to it. Producing high quality heuristics often requires some inside knowledge of the system under inference. If the user has this knowledge already, they are unlikely to require automated inference. Ideally, we would like something more generally applicable. This theory presents a more abstract *metaheuristic* which can be implemented with genetic programming.

```
theory PTA_Generalisation
  imports "../Inference" Same_Register Group_By
begin
```

```
hide_const I

datatype value_type = N | S

instantiation value_type :: linorder begin
fun less_value_type :: "value_type ⇒ value_type ⇒ bool" where
  "less_value_type N S = True" |
  "less_value_type _ _ = False"

definition less_eq_value_type :: "value_type ⇒ value_type ⇒ bool" where
 "less_eq_value_type v1 v2 ≡ (v1 < v2 ∨ v1 = v2)"

instance
  ⟨proof⟩

end

— This is a very hacky way of making sure that things with differently typed outputs don't get lumped together.
fun typeSig :: "output_function ⇒ value_type" where
  "typeSig (L (value.Str _)) = S" |
  "typeSig _ = N"

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

lemma same_structure_equiv:
  "Outputs t1 = [L (Num m)] ⟹ Outputs t2 = [L (Num n)] ⟹
   same_structure t1 t2 = Transition.same_structure t1 t2"
  ⟨proof⟩

type_synonym transition_group = "(tids × transition) list"

fun observe_all :: "iEFSM ⇒  cfstate ⇒ registers ⇒ trace ⇒ transition_group" where
  "observe_all _ _ _ [] = []" |
  "observe_all e s r ((l, i, _)#es)  =
    (case random_member (i_possible_steps e s r l i)  of
      (Some (ids, s', t)) ⇒ (((ids, t)#(observe_all e s' (evaluate_updates t i r) es))) |
      _ ⇒ []
    )"

definition transition_groups_exec :: "iEFSM ⇒ trace ⇒ (nat × tids × transition) list list" where
  "transition_groups_exec e t = group_by (λ(_, _, t1) (_, _, t2). same_structure t1 t2) (enumerate 0 (observe_all
e 0 <> t))"

type_synonym struct = "(label × arity × value_type list)"
```

We need to take the list of transition groups and tag them with the last transition that was taken which had a different structure.

```
fun tag :: "struct option ⇒ (nat × tids × transition) list list ⇒ (struct option × struct × (nat ×
tids × transition) list) list" where
  "tag _ [] = []" |
  "tag t (g#gs) = (
    let
      (_, _, head) = hd g;
      struct = (Label head, Arity head, map typeSig (Outputs head))
    in
    (t, struct, g)#(tag (Some struct) gs)
  )"
```

We need to group transitions not just by their structure but also by their history - i.e. the last transition which was taken which had a different structure. We need to order these groups by their relative positions within the traces such that output and update functions can be inferred in the correct order.

```
definition transition_groups :: "iEFSM ⇒ log ⇒ transition_group list" where
  "transition_groups e l = (
    let
      trace_groups = map (transition_groups_exec e) l;
      tagged = map (tag None) trace_groups;
      flat =  sort (fold (@) tagged []);
      group_fun = fold (λ(tag, s, gp) f. f((tag, s) $:= gp@(f$(tag, s)))) flat (K$ []);
      grouped = map (λx. group_fun $ x) (finfun_to_list group_fun);
      inx_groups = map (λgp. (Min (set (map fst gp)), map snd gp)) grouped
    in
      map snd (sort inx_groups)
  )"
```

For a given trace group, log, and EFSM, we want to build the training set for that group. That is, the set of inputs, registers, and expected outputs from those transitions. To do this, we must walk the traces in the EFSM to obtain the register values.

```
fun trace_group_training_set :: "transition_group ⇒ iEFSM ⇒ cfstate ⇒ registers ⇒ trace ⇒ (inputs
× registers × value list) list ⇒ (inputs × registers × value list) list" where
  "trace_group_training_set _ _ _ _ [] train = train" |
  "trace_group_training_set gp e s r ((l, i, p)#t) train = (
    let
      (id, s', transition) = fthe_elem (i_possible_steps e s r l i)
    in
    if ∃(id', _) ∈ set gp. id' = id then
      trace_group_training_set gp e s' (evaluate_updates transition i r) t ((i, r, p)#train)
    else
      trace_group_training_set gp e s' (evaluate_updates transition i r) t train
  )"

definition make_training_set :: "iEFSM ⇒ log ⇒ transition_group ⇒ (inputs × registers × value list)
list" where
  "make_training_set e l gp = fold (λh a. trace_group_training_set gp e 0 <> h a) l []"

primrec replace_groups :: "transition_group list ⇒ iEFSM ⇒ iEFSM" where
  "replace_groups [] e = e" |
  "replace_groups (h#t) e = replace_groups t (fold (λ(id, t) acc. replace_transition acc id t) h e)"

lemma replace_groups_fold [code]:
  "replace_groups xs e = fold (λh acc'. (fold (λ(id, t) acc. replace_transition acc id t) h acc')) xs e"
  ⟨proof⟩

definition insert_updates :: "transition ⇒ update_function list ⇒ transition" where
  "insert_updates t u = (
    let
      — Want to filter out null updates of the form rn := rn. It doesn't affect anything but it
      — does make things look cleaner
      necessary_updates = filter (λ(r, u). u ≠ V (R r)) u
    in
    t(|Updates := (filter (λ(r, _). r ∉ set (map fst u)) (Updates t))@necessary_updates|)
  )"

fun add_groupwise_updates_trace :: "trace  ⇒ (tids × update_function list) list ⇒ iEFSM ⇒ cfstate ⇒
registers ⇒ iEFSM" where
  "add_groupwise_updates_trace [] _ e _ _ = e" |
  "add_groupwise_updates_trace ((l, i, _)#trace) funs e s r = (
    let
      (id, s', t) = fthe_elem (i_possible_steps e s r l i);
      updated = evaluate_updates t i r;
      newUpdates = List.maps snd (filter (λ(tids, _). set id ⊆ set tids) funs);
```

```
      t' = insert_updates t newUpdates;
      updated' = apply_updates (Updates t') (join_ir i r) r;
      necessaryUpdates = filter (λ(r, _). updated $ r ≠ updated' $ r) newUpdates;
      t'' = insert_updates t necessaryUpdates;
      e' = replace_transition e id t''
    in
    add_groupwise_updates_trace trace funs e' s' updated'
  )"

primrec add_groupwise_updates :: "log  ⇒ (tids × update_function list) list ⇒ iEFSM ⇒ iEFSM" where
  "add_groupwise_updates [] _ e = e" |
  "add_groupwise_updates (h#t) funs e = add_groupwise_updates t funs (add_groupwise_updates_trace h funs
e 0 <>)"

lemma fold_add_groupwise_updates [code]:
  "add_groupwise_updates log funs e = fold (λtrace acc. add_groupwise_updates_trace trace funs acc 0 <>)
log e"
  ⟨proof⟩
definition get_regs :: "(vname ⇒f String.literal) ⇒ inputs ⇒ vname aexp ⇒ value ⇒ registers" where
  "get_regs types inputs expression output = Eps (λr. aval expression (join_ir inputs r) = Some output)"

declare get_regs_def [code del]
code_printing constant get_regs ⇀ (Scala) "Dirties.getRegs"

type_synonym action_info = "(cfstate × registers × registers × inputs × tids × transition)"
type_synonym run_info = "action_info list"
type_synonym targeted_run_info = "(registers × action_info) list"

fun everything_walk :: "output_function ⇒ nat ⇒ (vname ⇒f String.literal) ⇒ trace ⇒ iEFSM ⇒ cfstate
⇒ registers ⇒ transition_group ⇒ run_info" where
  "everything_walk _ _ _ [] _ _ _ _ = []" |
  "everything_walk f fi types ((label, inputs, outputs)#t) oPTA s regs gp  = (
    let (tid, s', ta) = fthe_elem (i_possible_steps oPTA s regs label inputs) in
     — Possible steps with a transition we need to modify
    if ∃(tid', _) ∈ set gp. tid = tid' then
      (s, regs, get_regs types inputs f (outputs!fi), inputs, tid, ta)#(everything_walk f fi types t oPTA
s' (evaluate_updates ta inputs regs) gp)
    else
      let empty = <> in
      (s, regs, empty, inputs, tid, ta)#(everything_walk f fi types t oPTA s' (evaluate_updates ta inputs
regs) gp)
  )"

definition everything_walk_log :: "output_function ⇒ nat ⇒ (vname ⇒f String.literal) ⇒ log ⇒ iEFSM
⇒ transition_group ⇒ run_info list" where
  "everything_walk_log f fi types log e gp = map (λt. everything_walk f fi types t e 0 <> gp) log"

fun target :: "registers ⇒ run_info ⇒ targeted_run_info" where
  "target _ [] = []" |
  "target tRegs ((s, oldregs, regs, inputs, tid, ta)#t) = (
    let newTarget = if finfun_to_list regs = [] then tRegs else regs in
    (tRegs, s, oldregs, regs, inputs, tid, ta)#target newTarget t
  )"

fun target_tail :: "registers ⇒ run_info ⇒ targeted_run_info ⇒ targeted_run_info" where
  "target_tail _ [] tt = rev tt" |
  "target_tail tRegs ((s, oldregs, regs, inputs, tid, ta)#t) tt = (
    let newTarget = if finfun_to_list regs = [] then tRegs else regs in
    target_tail newTarget t ((tRegs, s, oldregs, regs, inputs, tid, ta)#tt)
  )"

lemma target_tail: "(rev bs)@(target tRegs ts) = target_tail tRegs ts bs"
⟨proof⟩
```

```
definition "target_fold tRegs ts b = fst (fold (λ(s, oldregs, regs, inputs, tid, ta) (acc, tRegs).
let newTarget = if finfun_to_list regs = [] then tRegs else regs in
    (acc@[(tRegs, s, oldregs, regs, inputs, tid, ta)], newTarget)
) ts (rev b, tRegs))"

lemma target_tail_fold: "target_tail tRegs ts b = target_fold tRegs ts b"
⟨proof⟩

lemma target_fold [code]: "target tRegs ts = target_fold tRegs ts []"
  ⟨proof⟩
definition get_update :: "label ⇒ nat ⇒ value list ⇒ (inputs × registers × registers) list ⇒ vname
aexp option" where
  "get_update _ reg values train = (let
    possible_funs = {a. ∀(i, r, r') ∈ set train. aval a (join_ir i r) = r' $ reg}
    in
    if possible_funs = {} then None else Some (Eps (λx. x ∈ possible_funs))
  )"

declare get_update_def [code del]
code_printing constant get_update ⇀ (Scala) "Dirties.getUpdate"

definition get_updates_opt :: "label ⇒ value list ⇒ (inputs × registers × registers) list ⇒ (nat ×
vname aexp option) list" where
  "get_updates_opt l values train = (let
    updated_regs = fold List.union (map (finfun_to_list ∘ snd ∘ snd) train) [] in
    map (λr.
      let targetValues = remdups (map (λ(_, _, regs). regs $ r) train) in
      if  (∀(_, anteriorRegs, posteriorRegs) ∈ set train. anteriorRegs $ r = posteriorRegs $ r) then
        (r, Some (V (R r)))
      else if length targetValues = 1 ∧ (∀(inputs, anteriorRegs, _) ∈ set train. finfun_to_list anteriorRegs
= []) then
        case hd targetValues of Some v ⇒
        (r, Some (L v))
      else
        (r, get_update l r values train)
    ) updated_regs
  )"

definition finfun_add :: "(('a::linorder) ⇒f 'b) ⇒ ('a ⇒f 'b) ⇒ ('a ⇒f 'b)" where
  "finfun_add a b = fold (λk f. f(k $:= b $ k)) (finfun_to_list b) a"

definition group_update :: "value list ⇒ targeted_run_info ⇒ (tids × (nat × vname aexp) list) option"
where
  "group_update values l = (
    let
      (_, (_, _, _, _, _, t)) = hd l;
      targeted = filter (λ(regs, _). finfun_to_list regs ≠ []) l;
      maybe_updates = get_updates_opt (Label t) values (map (λ(tRegs, s, oldRegs, regs, inputs, tid, ta).
(inputs, finfun_add oldRegs regs, tRegs)) targeted)
    in
    if ∃(_, f_opt) ∈ set maybe_updates. f_opt = None then
      None
    else
      Some (fold List.union (map (λ(tRegs, s, oldRegs, regs, inputs, tid, ta). tid) l) [], map (λ(r, f_o).
(r, the f_o)) maybe_updates)
  )"

fun groupwise_put_updates :: "transition_group list ⇒ log ⇒ value list ⇒ run_info list ⇒ (nat × (vname
aexp × vname ⇒f String.literal)) ⇒ iEFSM ⇒ iEFSM" where
  "groupwise_put_updates [] _ _ _ _  e = e" |
  "groupwise_put_updates (gp#gps) log values walked (o_inx, (op, types)) e = (
    let
```

```
      targeted = map (λx. filter (λ(_, _, _, _, id, tran). (id, tran) ∈ set gp) x) (map (λw. rev (target
<> (rev w))) walked);
      group = fold List.union targeted []
    in
    case group_update values group of
      None ⇒ groupwise_put_updates gps log values walked (o_inx, (op, types)) e |
      Some u ⇒ groupwise_put_updates gps log values walked (o_inx, (op, types)) (make_distinct (add_groupwise_upda
log [u] e))
  )"

definition updates_for_output :: "log ⇒ value list ⇒ transition_group ⇒ nat ⇒ vname aexp ⇒ vname ⇒f
String.literal ⇒ iEFSM ⇒ iEFSM" where
"updates_for_output log values current o_inx op types e = (
  if AExp.enumerate_regs op = {} then e
  else
    let
      walked = everything_walk_log op o_inx types log e current;
      groups = transition_groups e log
    in
    groupwise_put_updates groups log values walked (o_inx, (op, types)) e
  )"

type_synonym output_types = "(vname aexp × vname ⇒f String.literal)"

fun put_updates :: "log ⇒ value list ⇒ transition_group ⇒ (nat × output_types option) list ⇒ iEFSM
⇒ iEFSM" where
  "put_updates _ _ _ [] e = e" |
  "put_updates log values gp ((_, None)#ops) e = put_updates log values gp ops e" |
  "put_updates log values gp ((o_inx, Some (op, types))#ops) e = (
    let
      gp' = map (λ(id, t). (id, t(|Outputs := list_update (Outputs t) o_inx op|))) gp;
      generalised_model = fold (λ(id, t) acc. replace_transition acc id t) gp' e;
      e' = updates_for_output log values gp o_inx op types generalised_model
    in
    if accepts_log (set log) (tm e') then
     put_updates log values gp' ops e'
    else
     put_updates log values gp ops e
  )"

fun unzip_3 :: "('a × 'b × 'c) list ⇒ ('a list × 'b list × 'c list)" where
  "unzip_3 [] = ([], [], [])" |
  "unzip_3 ((a, b, c)#l) = (
    let (as, bs, cs) = unzip_3 l in
    (a#as, b#bs, c#cs)
  )"

lemma unzip_3: "unzip_3 l = (map fst l, map (fst ∘ snd) l, map (snd ∘ snd) l)"
  ⟨proof⟩

fun unzip_3_tailrec_rev :: "('a × 'b × 'c) list ⇒ ('a list × 'b list × 'c list) ⇒ ('a list × 'b list
× 'c list)" where
  "unzip_3_tailrec_rev [] (as, bs, cs) = (as, bs, cs)" |
  "unzip_3_tailrec_rev ((a, b, c)#t) (as, bs, cs) = unzip_3_tailrec_rev t (a#as, b#bs, c#cs)"

lemma unzip_3_tailrec_rev: "unzip_3_tailrec_rev l (as, bs, cs) = ((map_tailrec_rev fst l as), (map_tailrec_rev
(fst ∘ snd) l bs), (map_tailrec_rev (snd ∘ snd) l cs))"
  ⟨proof⟩

definition "unzip_3_tailrec l = (let (as, bs, cs) = unzip_3_tailrec_rev l ([],[],[]) in (rev as, rev bs,
rev cs))"

lemma unzip_3_tailrec [code]: "unzip_3 l = unzip_3_tailrec l"
```

⟨*proof*⟩

We want to return an aexp which, when evaluated in the correct context accounts for the literal input-output pairs within the training set. This will be replaced by symbolic regression in the executable

```
definition get_output :: "label ⇒ nat ⇒ value list ⇒ (inputs × registers × value) list ⇒ (vname aexp
× (vname ⇒f String.literal)) option" where
  "get_output _ maxReg values train = (let
    possible_funs = {a. ∀(i, r, p) ∈ set train. aval a (join_ir i r) = Some p}
    in
    if possible_funs = {} then None else Some (Eps (λx. x ∈ possible_funs), (K$ STR ''int''))
  )"
declare get_output_def [code del]
code_printing constant get_output ⇀ (Scala) "Dirties.getOutput"

definition get_outputs :: "label ⇒ nat ⇒ value list ⇒ inputs list ⇒ registers list ⇒ value list list
⇒ (vname aexp × (vname ⇒f String.literal)) option list" where
  "get_outputs l maxReg values I r outputs = map_tailrec (λ(maxReg, ps). get_output l maxReg values (zip
I (zip r ps))) (enumerate maxReg (transpose outputs))"

definition enumerate_exec_values :: "trace ⇒ value list" where
  "enumerate_exec_values vs = fold (λ(_, i, p) I. List.union (List.union i p) I) vs []"

definition enumerate_log_values :: "log ⇒ value list" where
  "enumerate_log_values l = fold (λe I. List.union (enumerate_exec_values e) I) l []"


definition generalise_and_update :: "log ⇒ iEFSM ⇒ transition_group ⇒ iEFSM" where
  "generalise_and_update log e gp = (
    let
      label = Label (snd (hd gp));
      values = enumerate_log_values log;
      new_gp_ts = make_training_set e log gp;
      (I, R, P) = unzip_3 new_gp_ts;
      max_reg = max_reg_total e;
      outputs = get_outputs label max_reg values I R P
    in
      put_updates log values gp (enumerate 0 outputs) e
  )"
```

Splitting structural groups up into subgroups by previous transition can cause different subgroups to get different updates. We ideally want structural groups to have the same output and update functions, as structural groups are likely to be instances of the same underlying behaviour.

```
definition standardise_group :: "iEFSM ⇒ log ⇒ transition_group ⇒ (iEFSM ⇒ log ⇒ transition_group ⇒
transition_group) ⇒ iEFSM" where
  "standardise_group e l gp s = (
    let
      standardised = s e l gp;
      e' = replace_transitions e standardised
    in
      if e' = e then e else
      if accepts_log (set l) (tm e') then e' else e
)"

primrec find_outputs :: "output_function list list ⇒ iEFSM ⇒ log ⇒ transition_group ⇒ output_function
list option" where
  "find_outputs [] _ _ _ = None" |
  "find_outputs (h#t) e l g = (
    let
      outputs = fold (λ(tids, t) acc. replace_transition acc tids (t(|Outputs := h|))) g e
    in
      if accepts_log (set l) (tm outputs) then
        Some h
      else
```

```
      find_outputs t e l g
  )"

primrec find_updates_outputs :: "update_function list list ⇒ output_function list list ⇒ iEFSM ⇒ log
⇒ transition_group ⇒ (output_function list × update_function list) option" where
  "find_updates_outputs [] _ _ _ _ = None" |
  "find_updates_outputs (h#t) p e l g = (
    let
      updates = fold (λ(tids, t) acc. replace_transition acc tids (t(|Updates := h|))) g e
    in
      case find_outputs p updates l (map (λ(id, t). (id,t(|Updates := h|)))  g) of
        Some pp ⇒ Some (pp, h) |
        None ⇒ find_updates_outputs t p e l g
  )"

definition updates_for :: "update_function list ⇒ update_function list list" where
  "updates_for U = (
    let uf = fold (λ(r, u) f. f(r $:= u#(f $ r))) U (K$ []) in
    map (λr. map (λu. (r, u)) (uf $ r)) (finfun_to_list uf)
  )"

definition standardise_group_outputs_updates :: "iEFSM ⇒ log ⇒ transition_group ⇒ transition_group" where
  "standardise_group_outputs_updates e l g = (
    let
      update_groups = product_lists (updates_for (remdups (List.maps (Updates ∘ snd) g)));
      update_groups_subs = fold (List.union ∘ subseqs) update_groups [];
      output_groups = product_lists (transpose (remdups (map (Outputs ∘ snd) g)))
    in
    case find_updates_outputs update_groups_subs output_groups e l g of
      None ⇒ g |
      Some (p, u) ⇒ map (λ(id, t). (id, t(|Outputs := p, Updates := u|))) g
  )"

fun find_first_use_of_trace :: "nat ⇒ trace ⇒ iEFSM ⇒ cfstate ⇒ registers ⇒ tids option" where
  "find_first_use_of_trace _ [] _ _ _ = None" |
  "find_first_use_of_trace rr ((l, i, _)#es) e s r = (
    let
      (id, s', t) = fthe_elem (i_possible_steps e s r l i)
    in
      if (∃p ∈ set (Outputs t). aexp_constrains p (V (R rr))) then
        Some id
      else
        find_first_use_of_trace rr es e s' (evaluate_updates t i r)
  )"

definition find_first_uses_of :: "nat ⇒ log ⇒ iEFSM ⇒ tids list" where
  "find_first_uses_of r l e = List.maps (λx. case x of None ⇒ [] | Some x ⇒ [x]) (map (λt. find_first_use_of_tra
r t e 0 <>) l)"

fun find_initialisation_of_trace :: "nat ⇒ trace ⇒ iEFSM ⇒ cfstate ⇒ registers ⇒ (tids × transition)
option" where
  "find_initialisation_of_trace _ [] _ _ _ = None" |
  "find_initialisation_of_trace r' ((l, i, _)#es) e s r = (
    let
      (tids, s', t) = fthe_elem (i_possible_steps e s r l i)
    in
    if (∃(rr, u) ∈ set (Updates t). rr = r' ∧ is_lit u) then
      Some (tids, t)
    else
      find_initialisation_of_trace r' es e s' (evaluate_updates t i r)
  )"

primrec find_initialisation_of :: "nat ⇒ iEFSM ⇒ log ⇒ (tids × transition) option list" where
```

```
  "find_initialisation_of _ _ [] = []" |
  "find_initialisation_of r e (h#t) = (
    case find_initialisation_of_trace r h e 0 <> of
      None ⇒ find_initialisation_of r e t |
      Some thing ⇒ Some thing#(find_initialisation_of r e t)
  )"

definition delay_initialisation_of :: "nat ⇒ log ⇒ iEFSM ⇒ tids list ⇒ iEFSM" where
  "delay_initialisation_of r l e tids = fold (λx e. case x of
      None ⇒ e |
    Some (i_tids, t) ⇒
      let
        origins = map (λid. origin id e) tids;
        init_val = snd (hd (filter (λ(r', _). r = r') (Updates t)));
        e' = fimage (λ(id, (origin', dest), tr).
        — Add the initialisation update to incoming transitions
        if dest ∈ set origins then
          (id, (origin', dest), tr(|Updates := List.insert (r, init_val) (Updates tr)|))
        — Strip the initialisation update from the original initialising transition
        else if id = i_tids then
          (id, (origin', dest), tr(|Updates := filter (λ(r', _). r ≠ r') (Updates tr)|))
        else
          (id, (origin', dest), tr)
      ) e
      in
      — We don't want to update a register twice so just leave it
      if accepts_log (set l) (tm e') then
        e'
      else
        e
  ) (find_initialisation_of r e l) e"

fun groupwise_generalise_and_update :: "log ⇒ iEFSM ⇒ transition_group list ⇒ iEFSM" where
  "groupwise_generalise_and_update _ e [] = e" |
  "groupwise_generalise_and_update log e (gp#t) = (
        let
          e' = generalise_and_update log e gp;
          rep = snd (hd (gp));
          structural_group = fimage (λ(i, _, t). (i, t)) (ffilter (λ(_, _, t). same_structure rep t) e');
          delayed = fold (λr acc. delay_initialisation_of r log acc (find_first_uses_of r log acc)) (sorted_list_o
(all_regs e')) e';
          standardised = standardise_group delayed log (sorted_list_of_fset structural_group) standardise_group_out
          structural_group2 = fimage (λ(_, _, t). (Outputs t, Updates t)) (ffilter (λ(_, _, t).  Label rep
= Label t ∧ Arity rep = Arity t ∧ length (Outputs rep) = length (Outputs t)) standardised)
        in
        — If we manage to standardise a structural group, we do not need to evolve outputs and updates for the other
historical subgroups so can filter them out.
        if fis_singleton structural_group2 then
          groupwise_generalise_and_update log (merge_regs standardised (accepts_log (set log))) (filter
(λg. set g ∩ fset structural_group = {}) t)
        else
          groupwise_generalise_and_update log (merge_regs standardised (accepts_log (set log))) t
  )"

definition drop_all_guards :: "iEFSM ⇒ iEFSM ⇒ log ⇒ update_modifier ⇒ (iEFSM ⇒ nondeterministic_pair
fset) ⇒ iEFSM" where
"drop_all_guards e pta log m np = (let
      derestricted = fimage (λ(id, tf, tran). (id, tf, tran(|Guards := []|))) e;
      nondeterministic_pairs = sorted_list_of_fset (np derestricted)
    in
    case resolve_nondeterminism {} nondeterministic_pairs pta derestricted m (accepts_log (set log)) np
of
      (None, _) ⇒ pta |
```

```
      (Some resolved, _) ⇒ resolved
  )"

definition updated_regs :: "transition ⇒ nat set" where
  "updated_regs t = set (map fst (Updates t))"

definition fewer_updates :: "transition ⇒ transition fset ⇒ transition option" where
  "fewer_updates t tt = (
    let p = ffilter (λt'. same_structure t t' ∧ Outputs t = Outputs t' ∧ updated_regs t' ⊂ updated_regs
t) tt in
    if p = {||} then None else Some (snd (fMin (fimage (λt. (length (Updates t), t)) p))))"

fun remove_spurious_updates_aux :: "iEFSM ⇒ transition_group ⇒ transition fset ⇒ log ⇒ iEFSM" where
  "remove_spurious_updates_aux e [] _ _ = e" |
  "remove_spurious_updates_aux e ((tid, t)#ts) tt l = (
    case fewer_updates t tt of
      None ⇒ remove_spurious_updates_aux e ts tt l |
      Some t' ⇒ (
        let e' = replace_transition e tid t' in
        if accepts_log (set l) (tm e') then
          remove_spurious_updates_aux e' ts tt l
        else
          remove_spurious_updates_aux e ts tt l
      )
  )"


definition remove_spurious_updates :: "iEFSM ⇒ log ⇒ iEFSM" where
  "remove_spurious_updates e l = (
    let transitions = fimage (λ(tid, _, t). (tid, t)) e in
      remove_spurious_updates_aux e (sorted_list_of_fset transitions) (fimage snd transitions) l
  )"

definition derestrict :: "iEFSM ⇒ log ⇒ update_modifier ⇒ (iEFSM ⇒ nondeterministic_pair fset) ⇒ iEFSM"
where
  "derestrict pta log m np = (
    let
      normalised = groupwise_generalise_and_update log pta (transition_groups pta log)
    in
      drop_all_guards normalised pta log m np
  )"

definition "drop_pta_guards pta log m np = drop_all_guards pta pta log m np"

end
```

4 Output

This chapter provides two different output formats for EFSMs.

4.1 Graphical Output (EFSM_Dot)

It is often more intuitive and aesthetically pleasing to view EFSMs graphically. DOT is a graph layout engine which converts textual representations of graphs to more useful formats, such as SVG or PNG representations. This theory defines functions to convert arbitrary EFSMs to DOT for easier viewing. Here, transitions use the syntactic sugar presented in [1] such that they take the form $label : arity[g_1, \ldots, g_g]/f_1, \ldots, f_f[u_1, \ldots, u_u]$.

theory *EFSM_Dot*
imports *Inference*
begin

```
fun string_of_digit :: "nat ⇒ String.literal" where
  "string_of_digit n = (
        if n = 0 then (STR ''0'')
    else if n = 1 then (STR ''1'')
    else if n = 2 then (STR ''2'')
    else if n = 3 then (STR ''3'')
    else if n = 4 then (STR ''4'')
    else if n = 5 then (STR ''5'')
    else if n = 6 then (STR ''6'')
    else if n = 7 then (STR ''7'')
    else if n = 8 then (STR ''8'')
    else (STR ''9''))"

abbreviation newline :: String.literal where
  "newline ≡ STR ''
''"

abbreviation quote :: String.literal where
  "quote ≡ STR ''"''"

definition shows_string :: "String.literal ⇒ String.literal ⇒ String.literal"
where
  "shows_string = (+)"

fun showsp_nat :: "String.literal ⇒ nat ⇒ String.literal ⇒ String.literal"
where
  "showsp_nat p n =
    (if n < 10 then shows_string (string_of_digit n)
    else showsp_nat p (n div 10) o shows_string (string_of_digit (n mod 10)))"
declare showsp_nat.simps [simp del]

definition showsp_int :: "String.literal ⇒ int ⇒ String.literal ⇒ String.literal"
where
  "showsp_int p i =
    (if i < 0 then shows_string STR ''-'' o showsp_nat p (nat (- i)) else showsp_nat p (nat i))"

definition "show_int n  ≡ showsp_int ((STR '''')) n ((STR ''''))"
definition "show_nat n  ≡ showsp_nat ((STR '''')) n ((STR ''''))"

definition replace_backslash :: "String.literal ⇒ String.literal" where
  "replace_backslash s = String.implode (fold (@) (map (λx. if x = CHR 0x5c then [CHR 0x5c,CHR 0x5c] else
[x]) (String.explode s)) '''')"
```

```
code_printing
  constant replace_backslash ⇀ (Scala) "_.replace("\\", "\\\\")"

fun value2dot :: "value ⇒ String.literal" where
  "value2dot (value.Str s) = quote + replace_backslash s + quote" |
  "value2dot (Num n) = show_int n"

fun vname2dot :: "vname ⇒ String.literal" where
  "vname2dot (vname.I n) = STR ''i<sub>''+(show_nat (n))+STR ''</sub>''" |
  "vname2dot (R n) = STR ''r<sub>''+(show_nat n)+STR ''</sub>''"

fun aexp2dot :: "vname aexp ⇒ String.literal" where
  "aexp2dot (L v) = value2dot v" |
  "aexp2dot (V v) = vname2dot v" |
  "aexp2dot (Plus a1 a2) = (aexp2dot a1)+STR '' + ''+(aexp2dot a2)" |
  "aexp2dot (Minus a1 a2) = (aexp2dot a1)+STR '' - ''+(aexp2dot a2)" |
  "aexp2dot (Times a1 a2) = (aexp2dot a1)+STR '' &times; ''+(aexp2dot a2)"

fun join :: "String.literal list ⇒ String.literal ⇒ String.literal" where
  "join [] _ = (STR '''')" |
  "join [a] _ = a" |
  "join (h#t) s = h+s+(join t s)"

definition show_nats :: "nat list ⇒ String.literal" where
  "show_nats l = join (map show_nat l) STR '', ''"

fun gexp2dot :: "vname gexp ⇒ String.literal" where
  "gexp2dot (GExp.Bc True) = (STR ''True'')" |
  "gexp2dot (GExp.Bc False) = (STR ''False'')" |
  "gexp2dot (GExp.Eq a1 a2) = (aexp2dot a1)+STR '' = ''+(aexp2dot a2)" |
  "gexp2dot (GExp.Gt a1 a2) = (aexp2dot a1)+STR '' &gt; ''+(aexp2dot a2)" |
  "gexp2dot (GExp.In v l) = (vname2dot v)+STR ''&isin;{''+(join (map value2dot l) STR '', '')+STR ''}''"
|
  "gexp2dot (Nor g1 g2) = STR ''!(''+(gexp2dot g1)+STR ''&or;''+(gexp2dot g2)+STR '')''"

primrec guards2dot_aux :: "vname gexp list ⇒ String.literal list" where
  "guards2dot_aux [] = []" |
  "guards2dot_aux (h#t) = (gexp2dot h)#(guards2dot_aux t)"

lemma gexp2dot_aux_code [code]: "guards2dot_aux l = map gexp2dot l"
  ⟨proof⟩

primrec updates2dot_aux :: "update_function list ⇒ String.literal list" where
  "updates2dot_aux [] = []" |
  "updates2dot_aux (h#t) = ((vname2dot (R (fst h)))+STR '' := ''+(aexp2dot (snd h)))#(updates2dot_aux t)"

lemma updates2dot_aux_code [code]:
  "updates2dot_aux l = map (λ(r, u). (vname2dot (R r))+STR '' := ''+(aexp2dot u)) l"
  ⟨proof⟩

primrec outputs2dot :: "output_function list ⇒ nat ⇒ String.literal list" where
  "outputs2dot [] _ = []" |
  "outputs2dot (h#t) n = ((STR ''o<sub>''+(show_nat n))+STR ''</sub> := ''+(aexp2dot h))#(outputs2dot t
(n+1))"

fun updates2dot :: "update_function list ⇒ String.literal" where
  "updates2dot [] = (STR '''')" |
  "updates2dot a = STR ''&#91;''+(join (updates2dot_aux a) STR '', '')+STR ''&#93;''"

fun guards2dot :: "vname gexp list ⇒ String.literal" where
  "guards2dot [] = (STR '''')" |
  "guards2dot a = STR ''&#91;''+(join (guards2dot_aux a) STR '', '')+STR ''&#93;''"
```

```
definition latter2dot :: "transition ⇒ String.literal" where
  "latter2dot t = (let l = (join (outputs2dot (Outputs t) 1) STR '', '')+(updates2dot (Updates t)) in (if
l = (STR '''') then (STR '''') else STR ''/''+l))"

definition transition2dot :: "transition ⇒ String.literal" where
  "transition2dot t = (Label t)+STR '':''+(show_nat (Arity t))+(guards2dot (Guards t))+(latter2dot t)"

definition efsm2dot :: "transition_matrix ⇒ String.literal" where
  "efsm2dot e = STR ''digraph EFSM{''+newline+
                STR ''  graph [rankdir=''+quote+(STR ''LR'')+quote+STR '', fontname=''+quote+STR ''Latin
Modern Math''+quote+STR '']';''+newline+
                STR ''  node [color=''+quote+(STR ''black'')+quote+STR '', fillcolor=''+quote+(STR ''white'')+quote
'', shape=''+quote+(STR ''circle'')+quote+STR '', style=''+quote+(STR ''filled'')+quote+STR '', fontname=''+quote+S
''Latin Modern Math''+quote+STR '']';''+newline+
                STR ''  edge [fontname=''+quote+STR ''Latin Modern Math''+quote+STR '']';''+newline+newline+
                  STR ''  s0[fillcolor=''+quote+STR ''gray''+quote+STR '', label=<s<sub>0</sub>>];''+newline+
                  (join (map (λs. STR ''  s''+show_nat s+STR ''[label=<s<sub>'' +show_nat s+ STR ''</sub>>];'')
(sorted_list_of_fset (EFSM.S e - {|0|}))) (newline))+newline+newline+
                  (join ((map (λ((from, to), t). STR ''  s''+(show_nat from)+STR ''->s''+(show_nat to)+STR
''[label=<<i>''+(transition2dot t)+STR ''</i>>];'') (sorted_list_of_fset e))) newline)+newline+
                STR ''}''"

definition iefsm2dot :: "iEFSM ⇒ String.literal" where
  "iefsm2dot e = STR ''digraph EFSM{''+newline+
                 STR ''  graph [rankdir=''+quote+(STR ''LR'')+quote+STR '', fontname=''+quote+STR ''Latin
Modern Math''+quote+STR '']';''+newline+
                 STR ''  node [color=''+quote+(STR ''black'')+quote+STR '', fillcolor=''+quote+(STR ''white'')+quot
'', shape=''+quote+(STR ''circle'')+quote+STR '', style=''+quote+(STR ''filled'')+quote+STR '', fontname=''+quote+S
''Latin Modern Math''+quote+STR '']';''+newline+
                 STR ''  edge [fontname=''+quote+STR ''Latin Modern Math''+quote+STR '']';''+newline+newline+
                  STR ''  s0[fillcolor=''+quote+STR ''gray''+quote+STR '', label=<s<sub>0</sub>>];''+newline+
                  (join (map (λs. STR ''  s''+show_nat s+STR ''[label=<s<sub>'' +show_nat s+ STR ''</sub>>];'')
(sorted_list_of_fset (S e - {|0|}))) (newline))+newline+newline+
                  (join ((map (λ(uid, (from, to), t). STR ''  s''+(show_nat from)+STR ''->s''+(show_nat
to)+STR ''[label=<<i> [''+show_nats (sort uid)+STR '']''+(transition2dot t)+STR ''</i>>];'') (sorted_list_of_fset
e))) newline)+newline+
                STR ''}''"

abbreviation newline_str :: string where
  "newline_str ≡ ''
''"

abbreviation quote_str :: string where
  "quote_str ≡ ''0x22''"
end
```

4.2 Output to SAL (efsm2sal)

SAL is a framework for combining different tools for abstraction, program analysis, theorem proving, and model checking. It is able to verify and refute properties of EFSMs phrased in LTL. In [2], it is proposed that a model checker be used to assist in checking the conditions necessary for one transition to subsume another. In order to effect this, it is necessary to convert the EFSM into a format that SAL can recognise. This theory file sets out the various definitions needed to do this such that SAL can be used to check subsumption conditions when running the EFSM inference tool generated by the code generator.

```
theory efsm2sal
  imports "EFSM_Dot"
begin

definition replace :: "String.literal ⇒ String.literal ⇒ String.literal ⇒ String.literal" where
  "replace s old new = s"
```

```
code_printing constant replace ⇀ (Scala) "_.replaceAll(_, _)"

definition escape :: "String.literal ⇒ (String.literal × String.literal) list ⇒ String.literal" where
  "escape s replacements = fold (λ(old, new) s'. replace s' old new) replacements s"

definition "replacements = [
  (STR ''/'', STR ''_SOL__''),
  (STR 0x5C+STR 0x5C, STR ''_BSOL__''),
  (STR '' '', STR ''_SPACE__''),
  (STR 0x5C+STR ''('', STR ''_LPAR__''),
  (STR 0x5C+STR '')'', STR ''_RPAR__''),
  (STR 0x5C+STR ''.'', STR ''_PERIOD__''),
  (STR ''@'', STR ''_COMMAT__'')
]"

fun aexp2sal :: "vname aexp ⇒ String.literal" where
  "aexp2sal (L (Num n)) = STR ''Some(Num(''+ show_int n + STR ''))''"|
  "aexp2sal (L (value.Str n)) = STR ''Some(Str(String__''+ (if n = STR '''' then STR ''_EMPTY__'' else escape
n replacements) + STR ''))''" |
  "aexp2sal (V (I i)) = STR ''Some(i('' + show_nat (i) + STR ''))''" |
  "aexp2sal (V (R r)) = STR ''r__'' + show_nat r" |
  "aexp2sal (Plus a1 a2) = STR ''value_plus(''+aexp2sal a1 + STR '', '' + aexp2sal a2 + STR '')''" |
  "aexp2sal (Minus a1 a2) = STR ''value_minus(''+aexp2sal a1 + STR '', '' + aexp2sal a2 + STR '')''" |
  "aexp2sal (Times a1 a2) = STR ''value_times(''+aexp2sal a1 + STR '', '' + aexp2sal a2 + STR '')''"

fun gexp2sal :: "vname gexp ⇒ String.literal" where
  "gexp2sal (Bc True) = STR ''True''" |
  "gexp2sal (Bc False) = STR ''False''" |
  "gexp2sal (Eq a1 a2) = STR ''value_eq('' + aexp2sal a1 + STR '', '' + aexp2sal a2 + STR '')''" |
  "gexp2sal (Gt a1 a2) = STR ''value_gt('' + aexp2sal a1 + STR '', '' + aexp2sal a2 + STR '')''" |
  "gexp2sal (In v l) = join (map (λl'.  STR ''gval(value_eq('' + aexp2sal (V v) + STR '', '' + aexp2sal
(L l') + STR ''))'') l) STR '' OR ''" |
  "gexp2sal (Nor g1 g2) = STR ''NOT (gval('' + gexp2sal g1 + STR '') OR gval( '' + gexp2sal g2 + STR ''))''"

fun guards2sal :: "vname gexp list ⇒ String.literal" where
  "guards2sal [] = STR ''TRUE''" |
  "guards2sal G = join (map gexp2sal G) STR '' AND ''"

fun aexp2sal_num :: "vname aexp ⇒ nat ⇒ String.literal" where
  "aexp2sal_num (L (Num n)) _ = STR ''Some(Num(''+ show_int n + STR ''))''"|
  "aexp2sal_num (L (value.Str n)) _ = STR ''Some(Str(String__''+ (if n = STR '''' then STR ''_EMPTY__''
else escape n replacements) + STR ''))''" |
  "aexp2sal_num (V (vname.I i)) _ = STR ''Some(i('' + show_nat i + STR ''))''" |
  "aexp2sal_num (V (vname.R i)) m = STR ''r__'' + show_nat i + STR ''.'' + show_nat m" |
  "aexp2sal_num (Plus a1 a2) _ = STR ''value_plus(''+aexp2sal a1 + STR '', '' + aexp2sal a2 + STR '')''"
|
  "aexp2sal_num (Minus a1 a2) _ = STR ''value_minus(''+aexp2sal a1 + STR '', '' + aexp2sal a2 + STR '')''"
|
  "aexp2sal_num (Times a1 a2) _ = STR ''value_times(''+aexp2sal a1 + STR '', '' + aexp2sal a2 + STR '')''"

fun gexp2sal_num :: "vname gexp ⇒ nat ⇒ String.literal" where
  "gexp2sal_num (Bc True) _ = STR ''True''" |
  "gexp2sal_num (Bc False) _ = STR ''False''" |
  "gexp2sal_num (Eq a1 a2) m = STR ''gval(value_eq('' + aexp2sal_num a1 m + STR '', '' + aexp2sal_num a2
m + STR ''))''" |
  "gexp2sal_num (Gt a1 a2) m = STR ''gval(value_gt('' + aexp2sal_num a1 m + STR '', '' + aexp2sal_num a2
m + STR ''))''" |
  "gexp2sal_num (In v l) m = join (map (λl'.  STR ''gval(value_eq('' + aexp2sal_num (V v) m + STR '', ''
+ aexp2sal_num (L l') m + STR ''))'') l) STR '' OR ''" |
  "gexp2sal_num (Nor g1 g2) m = STR ''NOT ('' + gexp2sal_num g1 m + STR '' OR '' + gexp2sal_num g2 m + STR
'')''"
```

```
fun guards2sal_num :: "vname gexp list ⇒ nat ⇒ String.literal" where
  "guards2sal_num [] _ = STR ''TRUE'' |
  "guards2sal_num G m = join (map (λg. gexp2sal_num g m) G) STR '' AND ''"

end
```

5 Code Generation

This chapter details the code generator setup to produce executable Scala code for our inference technique.

5.1 Lists (Code_Target_List)

Here we define some equivalent definitions which make for a faster implementation. We also make use of the `code_printing` statement such that native Scala implementations of common list operations are used instead of redefining them. This allows us to use the `par` construct such that the parallel implementations are used, which makes for an even faster implementation.

theory *Code_Target_List*
imports *Main*
begin

declare *List.insert_def [code del]*
declare *member_rec [code del]*

lemma *[code]: "List.insert x xs = (if List.member xs x then xs else x#xs)"*
⟨*proof*⟩

declare *enumerate_eq_zip [code]*
declare *foldr_conv_foldl [code]*
declare *map_filter_map_filter [code_unfold del]*

definition *"flatmap l f = List.maps f l"*

lemma *[code]: "List.maps f l = flatmap l f"*
⟨*proof*⟩

definition *"map_code l f = List.map f l"*
lemma *[code]:"List.map f l = map_code l f"*
⟨*proof*⟩

lemma *[code]: "removeAll a l = filter (λx. x ≠ a) l"*
⟨*proof*⟩

definition *"filter_code l f = List.filter f l"*

lemma *[code]: "List.filter l f = filter_code f l"*
⟨*proof*⟩

definition *all :: "'a list ⇒ ('a ⇒ bool) ⇒ bool"* **where**
"all l f = list_all f l"

lemma *[code]: "list_all f l = all l f"*
⟨*proof*⟩

definition *ex :: "'a list ⇒ ('a ⇒ bool) ⇒ bool"* **where**
"ex l f = list_ex f l"

lemma *[code]: "list_ex f l = ex l f"*
⟨*proof*⟩

declare *foldl_conv_fold[symmetric]*

```
lemma fold_conv_foldl [code]: "fold f xs s = foldl (λx s. f s x) s xs"
  ⟨proof⟩

lemma code_list_eq [code]:
  "HOL.equal xs ys ⟷ length xs = length ys ∧ (∀(x,y) ∈ set (zip xs ys). x = y)"
  ⟨proof⟩

definition take_map :: "nat ⇒ 'a list ⇒ 'a list" where
  "take_map n l = (if length l ≤ n then l else map (λi. l ! i) [0..<n])"

lemma nth_take_map: "i < n ⟹ take_map n xs ! i = xs ! i"
  ⟨proof⟩

lemma [code]: "take n l = take_map n l"
  ⟨proof⟩

fun upt_tailrec :: "nat ⇒ nat ⇒ nat list ⇒ nat list" where
  "upt_tailrec i 0 l = l" |
  "upt_tailrec i (Suc j) l = (if i ≤ j then upt_tailrec i j ([j]@l) else l)"

lemma upt_arbitrary_l: "(upt i j)@l = upt_tailrec i j l"
  ⟨proof⟩

lemma [code]: "upt i j = upt_tailrec i j []"
  ⟨proof⟩

function max_sort :: "('a::linorder) list ⇒ 'a list ⇒ 'a list" where
  "max_sort [] l = l" |
  "max_sort (h#t) l = (let u = (h#t); m = Max (set u) in max_sort (removeAll m u) (m#l))"
  ⟨proof⟩
termination
  ⟨proof⟩

lemma remdups_fold [code]:
  "remdups l = foldr (λi l. if i ∈ set l then l else i#l) l []"
⟨proof⟩

code_printing
  constant Cons ⇀ (Scala) "_::_"
  | constant rev ⇀ (Scala) "_.par.reverse.toList"
  | constant List.member ⇀ (Scala) "_.contains((_))"
  | constant "List.remdups" ⇀ (Scala) "_.par.distinct.toList"
  | constant "List.length" ⇀ (Scala) "Nat.Nata(_.par.length)"
  | constant "zip" ⇀ (Scala) "_.par.zip((_)).toList"
  | constant "flatmap" ⇀ (Scala) "_.par.flatMap((_)).toList"
  | constant "List.null" ⇀ (Scala) "_.isEmpty"
  | constant "map_code" ⇀ (Scala) "_.par.map((_)).toList"
  | constant "filter_code" ⇀ (Scala) "_.par.filter((_)).toList"
  | constant "all" ⇀ (Scala) "_.par.forall((_))"
  | constant "ex" ⇀ (Scala) "_.par.exists((_))"
  | constant "nth" ⇀ (Scala) "_(Code'_Numeral.integer'_of'_nat((_)).toInt)"
  | constant "foldl" ⇀ (Scala) "Dirties.foldl"
  | constant "hd" ⇀ (Scala) "_.head"

end
```

5.2 Sets (Code_Target_Set)

While the default code generator setup for sets works fine, it does not make for particularly readable code. The reason for this is that the default setup needs to work with potentially infinite sets. All of the sets we need to use here are finite so we present an alternative setup for the basic set operations which generates much cleaner code.

```
theory Code_Target_Set
  imports "HOL-Library.Cardinality"
begin

code_datatype set
declare List.union_coset_filter [code del]
declare insert_code [code del]
declare remove_code [code del]
declare card_coset_error [code del]
declare coset_subseteq_set_code [code del]
declare eq_set_code(1) [code del]
declare eq_set_code(2) [code del]
declare eq_set_code(4) [code del]
declare List.subset_code [code del]
declare inter_coset_fold [code del]
declare Cardinality.subset'_code [code del]

declare subset_eq [code]


lemma [code del]:
  "x ∈ List.coset xs ⟷ ¬ List.member xs x"
  ⟨proof⟩

lemma sup_set_append[code]: "(set x) ∪ (set y) = set (x @ y)"
  ⟨proof⟩

declare product_concat_map [code]

lemma [code]: "insert x (set s) = (if x ∈ set s then set s else set (x#s))"
  ⟨proof⟩

lemma [code]:
  "Cardinality.subset' (set l1) (set l2) = ((list_all (λx. List.member l2 x)) l1)"
  ⟨proof⟩

end
```

5.3 Finite Sets (Code_Target_FSet)

Here we define the operations on the `fset` datatype in terms of lists rather than sets. This allows the Scala implementation to skip a case match each time, which makes for cleaner and slightly faster code.

```
theory Code_Target_FSet
  imports "Extended_Finite_State_Machines.FSet_Utils"
begin

code_datatype fset_of_list

declare FSet.fset_of_list.rep_eq [code]

lemma fprod_code [code]:
  "fprod (fset_of_list xs) (fset_of_list ys) = fset_of_list (remdups [(x, y). x ← xs, y ← ys])"
  ⟨proof⟩

lemma fminus_fset_filter [code]:
  "fset_of_list A -  xs = fset_of_list (remdups (filter (λx. x |∉| xs) A))"
  ⟨proof⟩

lemma sup_fset_fold [code]:
  "(fset_of_list f1) |∪| (fset_of_list f2) = fset_of_list (remdups (f1@f2))"
  ⟨proof⟩

lemma bot_fset [code]: "{|||} = fset_of_list []"
```

⟨*proof*⟩

lemma *finsert [code]:*
"finsert a (fset_of_list as) = fset_of_list (List.insert a as)"
⟨*proof*⟩

lemma *ffilter_filter [code]:*
"ffilter f (fset_of_list as) = fset_of_list (List.filter f (remdups as))"
⟨*proof*⟩

lemma *fimage_map [code]:*
"fimage f (fset_of_list as) = fset_of_list (List.map f (remdups as))"
⟨*proof*⟩

lemma *ffUnion_fold [code]:*
"ffUnion (fset_of_list as) = fold (|∪|) as {||}"
⟨*proof*⟩

lemma *fmember [code]: "a |∈| (fset_of_list as) = List.member as a"*
⟨*proof*⟩

lemma *fthe_elem [code]: "fthe_elem (fset_of_list [x]) = x"*
⟨*proof*⟩

lemma *size [code]: "size (fset_of_list as) = length (remdups as)"*
⟨*proof*⟩

lemma *fMax_fold [code]: "fMax (fset_of_list (a#as)) = fold max as a"*
⟨*proof*⟩

lemma *fMin_fold [code]: "fMin (fset_of_list (h#t)) = fold min t h"*
⟨*proof*⟩

lemma *fremove_code [code]:*
"fremove a (fset_of_list A) = fset_of_list (filter (λx. x ≠ a) A)"
⟨*proof*⟩

lemma *fsubseteq [code]:*
"(fset_of_list l) |⊆| A = List.list_all (λx. x |∈| A) l"
⟨*proof*⟩

lemma *fsum_fold [code]: "fSum (fset_of_list l) = fold (+) (remdups l) 0"*
⟨*proof*⟩

lemma *code_fset_eq [code]:*
"HOL.equal X (fset_of_list Y) ⟷ size X = length (remdups Y) ∧ (∀x |∈| X. List.member Y x)"
⟨*proof*⟩

lemma *code_fsubset [code]:*
"s |⊂| s' = (s |⊆| s' ∧ size s < size s')"
⟨*proof*⟩

lemma *code_fset [code]: "fset (fset_of_list l) = fold insert l {}"*
⟨*proof*⟩

lemma *code_fBall [code]: "fBall (fset_of_list l) f = list_all f l"*
⟨*proof*⟩

lemma *code_fBex [code]: "fBex (fset_of_list l) f = list_ex f l"*
⟨*proof*⟩

definition *"nativeSort = sort"*
code_printing constant *nativeSort ⇀ (Scala) "_.sortWith((Orderings.less))"*

lemma *[code]: "sorted_list_of_fset (fset_of_list l) = nativeSort (remdups l)"*
⟨*proof*⟩

lemma *[code]: "sorted_list_of_set (set l) = nativeSort (remdups l)"*
⟨*proof*⟩

lemma *[code]: "fMin (fset_of_list (h#t)) = hd (nativeSort (h#t))"*
⟨*proof*⟩

lemma *sorted_Max_Cons:*
"l ≠ [] ⟹
sorted (a#l) ⟹
Max (set (a#l)) = Max (set l)"
⟨*proof*⟩

lemma *sorted_Max:*
"l ≠ [] ⟹
sorted l ⟹
Max (set l) = hd (rev l)"
⟨*proof*⟩

lemma *[code]: "fMax (fset_of_list (h#t)) = last (nativeSort (h#t))"*
⟨*proof*⟩

definition *"list_max l = fold max l"*
code_printing constant *list_max ⇀ (Scala) "_.par.fold((_))(Orderings.max)"*

lemma *[code]: "fMax (fset_of_list (h#t)) = list_max t h"*
⟨*proof*⟩

definition *"list_min l = fold min l"*
code_printing constant *list_min ⇀ (Scala) "_.par.fold((_))(Orderings.min)"*

lemma *[code]: "fMin (fset_of_list (h#t)) = list_min t h"*
⟨*proof*⟩

lemma *fis_singleton_code [code]: "fis_singleton s = (size s = 1)"*
⟨*proof*⟩

end

5.4 Code Generation (Code_Generation)

This theory is used to generate an executable Scala implementation of the inference tool which can be used to infer real EFSMs from real traces. Certain functions are replaced with native implementations. These can be found at `https://github.com/jmafoster1/efsm-inference/blob/master/inference-tool/src/main/scala/inference/Dirties.scala`.

theory *Code_Generation*
imports
"HOL-Library.Code_Target_Numeral"
Inference
SelectionStrategies
"heuristics/Store_Reuse_Subsumption"
"heuristics/Increment_Reset"
"heuristics/Same_Register"
"heuristics/Distinguishing_Guards"
"heuristics/PTA_Generalisation"
"heuristics/Weak_Subsumption"
"heuristics/Least_Upper_Bound"
EFSM_Dot
"code-targets/Code_Target_FSet"
"code-targets/Code_Target_Set"

```
   "code-targets/Code_Target_List"
efsm2sal
begin

declare One_nat_def [simp del]


code_printing
  constant HOL.conj ⇀ (Scala) "_ && _" |
  constant HOL.disj ⇀ (Scala) "_ || _" |
  constant "HOL.equal :: bool ⇒ bool ⇒ bool" ⇀ (Scala) infix 4 "==" |
  constant "fst" ⇀ (Scala) "_._1" |
  constant "snd" ⇀ (Scala) "_._2"|
  constant "(1::nat)" ⇀ (Scala) "Nat.Nata((1))"


definition "initially_undefined_context_check_full = initially_undefined_context_check"


fun mutex :: "'a gexp ⇒ 'a gexp ⇒ bool" where
  "mutex (Eq (V v) (L l)) (Eq (V v') (L l')) = (if v = v' then l ≠ l' else False)" |
  "mutex (gexp.In v l) (Eq (V v') (L l')) = (v = v' ∧ l' ∉ set l)" |
  "mutex (Eq (V v') (L l')) (gexp.In v l) = (v = v' ∧ l' ∉ set l)" |
  "mutex (gexp.In v l) (gexp.In v' l') = (v = v' ∧ set l ∩ set l' = {})" |
  "mutex _ _ = False"

lemma mutex_not_gval:
  "mutex x y ⟹ gval (gAnd y x) s ≠ true"
  ⟨proof⟩



definition choice_cases :: "transition ⇒ transition ⇒ bool" where
  "choice_cases t1 t2 = (
     if ∃(x, y) ∈ set (List.product (Guards t1) (Guards t2)). mutex x y then
       False
     else if Guards t1 = Guards t2 then
       satisfiable (fold gAnd (rev (Guards t1)) (gexp.Bc True))
     else
       satisfiable ((fold gAnd (rev (Guards t1@Guards t2)) (gexp.Bc True)))
   )"

lemma existing_mutex_not_true:
  "∃x∈set G. ∃y∈set G. mutex x y ⟹ ¬ apply_guards G s"
  ⟨proof⟩

lemma [code]: "choice t t' = choice_cases t t'"
  ⟨proof⟩

fun guardMatch_code :: "vname gexp list ⇒ vname gexp list ⇒ bool" where
  "guardMatch_code [(gexp.Eq (V (vname.I i)) (L (Num n)))] [(gexp.Eq (V (vname.I i')) (L (Num n')))] = (i
= 0 ∧ i' = 0)" |
  "guardMatch_code _ _ = False"

lemma [code]: "guardMatch t1 t2 = guardMatch_code (Guards t1) (Guards t2)"
  ⟨proof⟩

fun outputMatch_code :: "output_function list ⇒ output_function list ⇒ bool" where
  "outputMatch_code [L (Num n)] [L (Num n')] = True" |
  "outputMatch_code _ _ = False"

lemma [code]: "outputMatch t1 t2 = outputMatch_code (Outputs t1) (Outputs t2)"
  ⟨proof⟩
```

```
fun always_different_outputs :: "vname aexp list ⇒ vname aexp list ⇒ bool" where
  "always_different_outputs [] [] = False" |
  "always_different_outputs [] (a#_) = True" |
  "always_different_outputs (a#_) [] = True" |
  "always_different_outputs ((L v)#t) ((L v')#t') = (if v = v' then always_different_outputs t t' else True)"
|
  "always_different_outputs (h#t) (h'#t') = always_different_outputs t t'"

lemma always_different_outputs_outputs_never_equal:
  "always_different_outputs O1 O2 ⟹
   apply_outputs O1 s ≠ apply_outputs O2 s"
  ⟨proof⟩

fun tests_input_equality :: "nat ⇒ vname gexp ⇒ bool" where
  "tests_input_equality i (gexp.Eq (V (vname.I i')) (L _)) = (i = i')" |
  "tests_input_equality _ _ = False"

fun no_illegal_updates_code :: "update_function list ⇒ nat ⇒ bool" where
  "no_illegal_updates_code [] _ = True" |
  "no_illegal_updates_code ((r', u)#t) r = (r ≠ r' ∧ no_illegal_updates_code t r)"

lemma no_illegal_updates_code_aux:
  "(∀u∈set u. fst u ≠ r) = no_illegal_updates_code u r"
⟨proof⟩

lemma no_illegal_updates_code [code]:
  "no_illegal_updates t r = no_illegal_updates_code (Updates t) r"
  ⟨proof⟩

fun input_updates_register_aux :: "update_function list ⇒ nat option" where
  "input_updates_register_aux ((n, V (vname.I n'))#_) = Some n'" |
  "input_updates_register_aux (h#t) = input_updates_register_aux t" |
  "input_updates_register_aux [] = None"

definition input_updates_register :: "transition_matrix ⇒ (nat × String.literal)" where
  "input_updates_register e = (
    case fthe_elem (ffilter (λ(_, t). input_updates_register_aux (Updates t) ≠ None) e) of
      (_, t) ⇒ (case
        input_updates_register_aux (Updates t) of
          Some n ⇒ (n, Label t)
      )
  )"

definition "dirty_directly_subsumes e1 e2 s1 s2 t1 t2 = (if t1 = t2 then True else directly_subsumes e1 e2
s1 s2 t1 t2)"

definition "always_different_outputs_direct_subsumption m1 m2 s s' t2 = ((∃p c1 c. obtains s c1 m1 0 <>
p ∧ obtains s' c m2 0 <> p ∧ (∃i. can_take_transition t2 i c)))"

lemma always_different_outputs_direct_subsumption:
  "always_different_outputs (Outputs t1) (Outputs t2) ⟹
   always_different_outputs_direct_subsumption m1 m2 s s' t2 ⟹
   ¬ directly_subsumes m1 m2 s s' t1 t2"
  ⟨proof⟩

definition negate :: "'a gexp list ⇒ 'a gexp" where
  "negate g = gNot (fold gAnd g (Bc True))"

lemma gval_negate_cons:
  "gval (negate (a # G)) s = gval (gNot a) s ∨? gval (negate G) s"
  ⟨proof⟩
```

lemma *negate_true_guard:*
```
"(gval (negate G) s = true) = (gval (fold gAnd G (Bc True)) s = false)"
```
⟨*proof*⟩

lemma *gval_negate_not_invalid:*
```
"(gval (negate gs) (join_ir i ra) ≠ invalid) = (gval (fold gAnd gs (Bc True)) (join_ir i ra) ≠ invalid)"
```
⟨*proof*⟩

definition *"dirty_always_different_outputs_direct_subsumption = always_different_outputs_direct_subsumption"*

lemma *[code]: "always_different_outputs_direct_subsumption m1 m2 s s' t = (*
```
  if Guards t = [] then
    recognises_and_visits_both m1 m2 s s'
  else
    dirty_always_different_outputs_direct_subsumption m1 m2 s s' t
  )"
```
⟨*proof*⟩

definition *guard_subset_subsumption :: "transition ⇒ transition ⇒ bool"* **where**
```
  "guard_subset_subsumption t1 t2 = (Label t1 = Label t2 ∧ Arity t1 = Arity t2 ∧ set (Guards t1) ⊆ set
(Guards t2) ∧ Outputs t1 = Outputs t2 ∧ Updates t1 = Updates t2)"
```

lemma *guard_subset_subsumption:*
```
  "guard_subset_subsumption t1 t2 ⟹ directly_subsumes a b s s' t1 t2"
```
⟨*proof*⟩

definition *"guard_subset_eq_outputs_updates t1 t2 = (Label t1 = Label t2 ∧*
```
   Arity t1 = Arity t2 ∧
   Outputs t1 = Outputs t2 ∧
   Updates t1 = Updates t2 ∧
   set (Guards t2) ⊆ set (Guards t1))"
```

definition *"guard_superset_eq_outputs_updates t1 t2 = (Label t1 = Label t2 ∧*
```
   Arity t1 = Arity t2 ∧
   Outputs t1 = Outputs t2 ∧
   Updates t1 = Updates t2 ∧
   set (Guards t2) ⊃ set (Guards t1))"
```

definition *is_generalisation_of :: "transition ⇒ transition ⇒ nat ⇒ nat ⇒ bool"* **where**
```
  "is_generalisation_of t' t i r = (
    t' = remove_guard_add_update t i r ∧
    i < Arity t ∧
    r ∉ set (map fst (Updates t)) ∧
    (length (filter (tests_input_equality i) (Guards t)) ≥ 1)
  )"
```

lemma *tests_input_equality:*
```
  "(∃v. gexp.Eq (V (vname.I xb)) (L v) ∈ set G) = (1 ≤ length (filter (tests_input_equality xb) G))"
```
⟨*proof*⟩

lemma *[code]:*
```
  "Store_Reuse.is_generalisation_of x xa xb xc = is_generalisation_of x xa xb xc"
```
⟨*proof*⟩

definition *iEFSM2dot :: "iEFSM ⇒ nat ⇒ unit"* **where**
```
  "iEFSM2dot _ _ = ()"
```

definition *logStates :: "iEFSM ⇒ nat ⇒ unit"* **where**
```
  "logStates _ _ = ()"
```

```
function infer_with_log :: "(cfstate × cfstate) set ⇒ nat ⇒ iEFSM ⇒ strategy ⇒ update_modifier ⇒ (transition
⇒ bool) ⇒ (iEFSM ⇒ nondeterministic_pair fset) ⇒ iEFSM" where
  "infer_with_log failedMerges k e r m check np = (
    let scores = if k = 1 then score_1 e r else (k_score k e r) in
    case inference_step failedMerges e (ffilter (λs. (S1 s, S2 s) ∉ failedMerges ∧ (S2 s, S1 s) ∉ failedMerges)
scores) m check np of
      (None, _) ⇒ e |
      (Some new, failedMerges) ⇒ if (Inference.S new) |⊂| (Inference.S e) then
      let temp2 = logStates new (size (Inference.S e)) in
      infer_with_log failedMerges k new r m check np else e
  )"
  ⟨proof⟩
termination
  ⟨proof⟩

lemma infer_empty: "infer f k {|||} r m check np = {|||}"
  ⟨proof⟩




declare GExp.satisfiable_def [code del]
declare initially_undefined_context_check_full_def [code del]
declare generalise_output_context_check_def [code del]
declare dirty_always_different_outputs_direct_subsumption_def [code del]
declare diff_outputs_ctx_def [code del]
declare random_member_def [code del]
declare dirty_directly_subsumes_def [code del]
declare recognises_and_visits_both_def [code del]
declare initially_undefined_context_check_def [code del]
declare can_still_take_ctx_def [code del]

code_printing
  constant infer ⇀ (Scala) "Code'_Generation.infer'_with'_log" |
  constant recognises_and_visits_both ⇀ (Scala) "Dirties.recognisesAndGetsUsToBoth" |
  constant iEFSM2dot ⇀ (Scala) "PrettyPrinter.iEFSM2dot(_, _)" |
  constant logStates ⇀ (Scala) "Log.logStates(_, _)" |
  constant "dirty_directly_subsumes" ⇀ (Scala) "Dirties.scalaDirectlySubsumes" |
  constant "GExp.satisfiable" ⇀ (Scala) "Dirties.satisfiable" |
  constant "initially_undefined_context_check_full" ⇀ (Scala) "Dirties.initiallyUndefinedContextCheck"
|
  constant "generalise_output_context_check" ⇀ (Scala) "Dirties.generaliseOutputContextCheck" |
  constant "dirty_always_different_outputs_direct_subsumption" ⇀ (Scala) "Dirties.alwaysDifferentOutputsDirectSub
|
  constant "diff_outputs_ctx" ⇀ (Scala) "Dirties.diffOutputsCtx" |
  constant "can_still_take" ⇀ (Scala) "Dirties.canStillTake" |
  constant "random_member" ⇀ (Scala) "Dirties.randomMember"

code_printing
  constant "show_nat" ⇀ (Scala) "Code'_Numeral.integer'_of'_nat((_)).toString()"
  | constant "show_int" ⇀ (Scala) "Code'_Numeral.integer'_of'_int((_)).toString()"
  | constant "join" ⇀ (Scala) "_.mkString((_))"


code_printing
  type_constructor finfun ⇀ (Scala) "Map[_, _]"
  | constant "finfun_const" ⇀ (Scala) "scala.collection.immutable.Map().withDefaultValue((_))"
  | constant "finfun_update" ⇀ (Scala) "_ + (_ -> _)"
  | constant "finfun_apply" ⇀ (Scala) "_((_))"
  | constant "finfun_to_list" ⇀ (Scala) "_.keySet.toList"
declare finfun_to_list_const_code [code del]
```

declare *finfun_to_list_update_code [code del]*

definition *mismatched_updates :: "transition ⇒ transition ⇒ bool"* **where**
"mismatched_updates t1 t2 = (∃r ∈ set (map fst (Updates t1)). r ∉ set (map fst (Updates t2)))"

lemma *[code]:*
"directly_subsumes e1 e2 s1 s2 t1 t2 = (if t1 = t2 then True else dirty_directly_subsumes e1 e2 s1 s2 t1 t2)"
⟨*proof*⟩

export_code

try_heuristics_check
learn
infer_with_log
nondeterministic
make_pta
AExp.enumerate_vars

gAnd
gOr
gNot
Lt
Le
Ge
Ne

naive_score
naive_score_eq_bonus
exactly_equal
naive_score_outputs
naive_score_comprehensive
naive_score_comprehensive_eq_high
leaves

same_register
insert_increment_2
heuristic_1
heuristic_2
distinguish
weak_subsumption
lob

nondeterministic_pairs
nondeterministic_pairs_labar
nondeterministic_pairs_labar_dest

min
max
drop_pta_guards
test_log
iefsm2dot
efsm2dot
guards2sal
guards2sal_num
fold_In
max_int
enumerate_vars
derestrict
in *Scala*

end

Bibliography

[1] 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`.

[2] M. Foster, A. D. Brucker, R. G. Taylor, S. North, and J. Derrick. Incorporating data into efsm inference. In P. C. Ölveczky and G. Salaün, editors, *Software Engineering and Formal Methods (SEFM)*, number 11724 in Lecture Notes in Computer Science, pages 257–272. Springer-Verlag, Heidelberg, 2019. ISBN 3-540-25109-X. doi: 10.1007/978-3-030-30446-1_14. URL `https://www.brucker.ch/bibliography/abstract/foster.ea-incorporating-2019`.

[3] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*, page 501, New York, New York, USA, 2008. ACM Press. ISBN 9781605580791. doi: 10.1145/1368088.1368157. URL `http://portal.acm.org/citation.cfm?doid=1368088.1368157`.