Formalization of Lower-Bound Certificates for Optimal Classical Planning

Dmitriy Traytel

June 28, 2026

Abstract

We formalize the framework of pseudo-Boolean lower-bound certificates for classical planning introduced by Dold, Helmert, Nordström, Röger and Schindler [3]. A lower-bound certificate for a STRIPS (Simplified Theory of International Planning Representation and Scheduling) planning task and a bound B is a pseudo-Boolean circuit together with three proofs in the cutting planes proof system with reification; its validity guarantees that every plan of the task costs at least B, and hence that a plan of cost B is optimal. We prove the soundness of the certificate format (Theorem 1 of the paper) and formalize the paper's case study: certificates extracted from a run of the A* algorithm, with heuristic certificates for pattern database heuristics and for the maximum heuristic h^{max}, including the appendix material on efficiently proof-logging pattern databases. The main results state that a suitable snapshot of a terminated A* run yields a valid certificate and therefore proves the optimality of the plan it found. All locales are shown consistent by concrete interpretations, which also compose the pattern database certificate with the A* certificate, as intended by the paper.

Contents

1 Introduction

Optimal classical planners are complex pieces of software, and bugs that lead to suboptimal plans being reported as optimal are difficult to detect by testing: while a plan itself is easy to validate, its *optimality* is not. Dold et al. [3] address this problem with proof logging: the planner emits, alongside the plan, a *lower-bound certificate* that can be checked by an independent, much simpler verifier in the style of VeriPB [1]. The certificate asserts that no plan of cost less than a bound B exists; together with a plan of cost B this establishes optimality. (The paper also discusses how a certificate with a sufficiently large bound B establishes unsolvability [5]; the formalization here handles arbitrary finite bounds B.)

This entry formalizes the central trust story of that paper: the *soundness* of the certificate format, and the existence of valid certificates for the paper's case study, A* with pattern database [4] and h^{max} [2] heuristics. The development is organized as follows.

Lower_Bound_Certificates formalizes STRIPS planning tasks, pseudo-Boolean constraints and their 0/1 semantics, the cutting planes proof system with reification (CPR), the pseudo-Boolean encoding of a planning task (Definition 1 of the paper), pseudo-Boolean circuits (Definition 2), lower-bound certificates (Definition 3), and the soundness theorem (Theorem 1): a valid certificate for bound B implies that every plan costs at least B.

Encoding_Semantics develops the semantic toolkit used by all later theories. The reverse unit propagation (RUP) rule of the paper is over-approximated by a semantic rule, so every "RUP can derive X" lemma of the paper reduces to a semantic implication over 0/1 models; the bridge lemma *semantic_to_cpr* converts such implications into CPR derivations. The theory also proves characteristic "forcing" lemmas for each constraint of the encoding, and instantiates Theorem 1 at the extended variable type $\alpha + \gamma$, which provides unboundedly many fresh gate names for certificate circuits.

K_Gates formalizes the clipped cost-threshold gates $K_{\geq l}$ and the cost lemmas of Appendix B (monotonicity and step interaction).

Heuristic_Certificates formalizes heuristic certificates (Definition 4): a heuristic-supplied circuit fragment together with per-state state, goal, and inductivity lemmas.

A_Star_Certificates formalizes the proof-logging A* algorithm (equations (9)–(13); the three CPR proof obligations, the paper's Lemmas 1–5). A locale captures the snapshot of a terminated A* run—the closed list with g-values, the open list, the expansion-closure property,

and a valid heuristic certificate—and the main theorems show that the assembled circuit is a valid lower-bound certificate, so any plan costs at least B and a found plan of cost B is optimal.

PDB_Certificates constructs heuristic certificates for pattern database heuristics (equations (14)–(16); the three proof obligations, paper Lemma 6 and Appendix C) from a distance table over the abstract state space of a pattern, assuming exactly the two semantic properties of the table that soundness requires: abstract goal states have distance 0, and the distance is consistent along abstract transitions.

Hmax_Certificates constructs heuristic certificates for the maximum heuristic (equations (17)–(18); the three proof obligations of Appendix D) from the values an implementation computes anyway, assuming the distilled consequences of the h^{max} fixed-point equations.

Efficient_PDB formalizes the appendix material (Definition 5; the state-set extension lemma, Lemma 7 of Appendix E; the efficient-PDB lemmas, Lemmas 8–18 of Appendix F): a pattern database circuit restricted to the abstract states with finite goal distance, with an additional gate r_∞ covering all remaining abstract states.

Locale_Instances interprets every locale of the development with a concrete planning task, ruling out inconsistent assumption sets. The interpretations compose the pattern database certificate with the A* certificate and yield the concrete optimality theorem for the example task.

Two systematic deviations from the paper are worth highlighting; both are documented where they occur. First, the RUP rule is modeled semantically (every constraint derivable by unit propagation to conflict satisfies the semantic condition, so all paper certificates remain expressible), which means that statements about the *length* of derivations—such as the $O(|B|)$ bound of Lemma 7, Appendix E—are out of scope: of such lemmas we formalize the semantic content only. Second, the K-gates referenced by the heuristic circuits are inlined over the cost bits rather than shared with the search's own gates, since certificate circuits may only reference input variables and their own earlier gates. Otherwise, the formalization follows the paper's structure closely; the mapping from paper lemmas to Isabelle facts is given in the text blocks of each theory.

2 Lower-Bound Certificates

theory *Lower-Bound-Certificates*
  imports *Main*
begin

2.1 STRIPS Planning Tasks

type-synonym *'v state = 'v set*

record *('v) action =*
pre :: 'v set
add :: 'v set
del :: 'v set
cost :: nat

record *('v) strips-task =*
— In our formalisation the add and delete lists of an action may overlap. This is intentional the semantics (Def of the successor function) gives add priority over delete, matching the standard STRIPS semantics from the literature.
vars :: 'v set
acts :: ('v action) set
init :: 'v state
goal :: 'v set

definition *evars :: ('v action) ⇒ 'v set* **where**
evars a ≡ add a ∪ del a

definition *applicable :: ('v action) ⇒ 'v state ⇒ bool* **where**
applicable a s ≡ pre a ⊆ s

definition *successor :: ('v action) ⇒ 'v state ⇒ 'v state* **where**
successor a s ≡ (s − del a) ∪ add a

inductive *path ::*
('v action) set ⇒ 'v state ⇒ 'v state ⇒ ('v action) list ⇒ bool
for *A :: ('v action) set*
where
path-nil: path A s s []
| *path-cons: ⟦applicable a s; path A (successor a s) t π; a ∈ A⟧*
⟹ path A s t (a # π)

definition *is-goal-state :: ('v strips-task) ⇒ 'v state ⇒ bool* **where**
is-goal-state Π s ≡ goal Π ⊆ s

definition *is-plan-for :: ('v strips-task) ⇒ ('v action) list ⇒ bool* **where**
is-plan-for Π π ≡ ∃ s. path (acts Π) (init Π) s π ∧ is-goal-state Π s

definition *plan-cost :: ('v action) list ⇒ nat* **where**
plan-cost π ≡ sum-list (map cost π)

definition *optimal-plan :: ('v strips-task) ⇒ ('v action) list ⇒ bool* **where**
optimal-plan Π π ≡ is-plan-for Π π ∧
(∀ π'. is-plan-for Π π' ⟶ plan-cost π ≤ plan-cost π')

definition *solvable :: ('v strips-task) ⇒ bool* **where**

solvable $\Pi \equiv \exists \pi.$ *is-plan-for* $\Pi\ \pi$

2.2 Pseudo-Boolean Formulas

datatype *'v literal* = *Pos 'v* | *Neg 'v*

definition *lit-neg* :: *'v literal* ⇒ *'v literal* **where**
lit-neg l ≡ *case l of Pos v* ⇒ *Neg v* | *Neg v* ⇒ *Pos v*

type-synonym *'v pb-constraint* = (*nat* × *'v literal*) *list* × *nat*

definition *eval-lit* :: *'v literal* ⇒ (*'v* ⇒ *nat*) ⇒ *nat* **where**
eval-lit l rho ≡ *case l of Pos v* ⇒ *rho v* | *Neg v* ⇒ *1* − *rho v*

fun *pb-sum* :: (*nat* × *'v literal*) *list* ⇒ (*'v* ⇒ *nat*) ⇒ *nat* **where**
pb-sum [] *rho* = *0*
| *pb-sum* ((*a*, *l*) # *coeffs*) *rho* = *a* ∗ *eval-lit l rho* + *pb-sum coeffs rho*

definition *satisfies* :: *'v pb-constraint* ⇒ (*'v* ⇒ *nat*) ⇒ *bool* **where**
satisfies C rho ≡
let (*coeffs*, *A*) = *C*
in pb-sum coeffs rho ≥ *A*

definition *models* :: *'v pb-constraint set* ⇒ (*'v* ⇒ *nat*) ⇒ *bool* **where**
models CC rho ≡ ∀ *C* ∈ *CC. satisfies C rho*

definition *unsat-01* :: *'v pb-constraint set* ⇒ *bool* **where**
unsat-01 CC ≡ ¬ (∃ *rho.* (∀ *v. rho v* = *0* ∨ *rho v* = *1*) ∧ *models CC rho*)

definition *implies-constraint* :: *'v pb-constraint set* ⇒ *'v pb-constraint* ⇒ *bool*
(**infix** ⊨ *55*) **where**
CC ⊨ *C* ≡ ∀ *rho. models CC rho* ⟶ *satisfies C rho*

definition *implies-formula* :: *'v pb-constraint set* ⇒ *'v pb-constraint set* ⇒ *bool*
(**infix** ⊨.. *55*) **where**
CC ⊨.. *DD* ≡ ∀ *D* ∈ *DD. CC* ⊨ *D*

definition *constraint-neg* :: *'v pb-constraint* ⇒ *'v pb-constraint* **where**
constraint-neg C ≡ *case C of* (*coeffs*, *A*) ⇒
let M = *sum-list* (*map fst coeffs*)
in (*map* (λ(*a*, *l*). (*a*, *lit-neg l*)) *coeffs*, *M* − (*A* − *1*))

2.3 Cutting Planes with Reification (CPR)

definition *unit-clause* :: *'v literal* ⇒ *'v pb-constraint* **where**
unit-clause l ≡ ([(*1*, *l*)], *1*)

lemma *eval-lit-lit-neg-ge-one*:
eval-lit l rho + *eval-lit* (*lit-neg l*) *rho* ≥ *1*
by (*cases l*; *simp add*: *eval-lit-def lit-neg-def*; *metis add.commute le-zero-eq not-one-le-zero*)

lemma *pb-sum-add-negated-ge-sum*:
pb-sum coeffs rho + pb-sum (map (λ(a, l). (a, lit-neg l)) coeffs) rho
≥ sum-list (map fst coeffs)
proof (*induction coeffs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a-l coeffs*)
obtain *a l* **where** [*simp*]: *a-l* = (*a*, *l*) **by** (*cases a-l*)
have *X-ge-1*: *eval-lit l rho + eval-lit (lit-neg l) rho ≥ 1*
using *eval-lit-lit-neg-ge-one*[*of l rho*] **by** *blast*
have *IH*: *pb-sum coeffs rho + pb-sum (map (λ(a, l). (a, lit-neg l)) coeffs) rho ≥ sum-list (map fst coeffs)*
using *Cons.IH* **by** *blast*
have *a + sum-list (map fst coeffs)*
≤ a ∗ (eval-lit l rho + eval-lit (lit-neg l) rho)
+ (pb-sum coeffs rho + pb-sum (map (λ(a, l). (a, lit-neg l)) coeffs) rho)
using *X-ge-1 IH* **by** (*intro add-mono mult-le-mono order-refl*; *simp*)
also have *... = a ∗ eval-lit l rho + a ∗ eval-lit (lit-neg l) rho*
+ (pb-sum coeffs rho + pb-sum (map (λ(a, l). (a, lit-neg l)) coeffs) rho)
proof −
have *a ∗ (eval-lit l rho + eval-lit (lit-neg l) rho) = a ∗ eval-lit l rho + a ∗ eval-lit (lit-neg l) rho*
by (*simp add*: *distrib-left*)
then show *?thesis* **by** *simp*
qed
also have *... = pb-sum ((a, l) # coeffs) rho + pb-sum (map (λ(a, l). (a, lit-neg l)) ((a, l) # coeffs)) rho*
by (*simp add*: *add-ac*)
finally show *?case* **by** *simp*
qed

definition *linear-combination* :: *nat ⇒ 'v pb-constraint ⇒ nat ⇒ 'v pb-constraint ⇒ 'v pb-constraint* **where**
linear-combination cA C cB D ≡
case (C, D) of ((coeffsA, A), (coeffsB, B)) ⇒
(map (λ(a, l). (cA ∗ a, l)) coeffsA @ map (λ(b, l). (cB ∗ b, l)) coeffsB, cA ∗ A + cB ∗ B)

definition *division* :: *nat ⇒ 'v pb-constraint ⇒ 'v pb-constraint* **where**
division c C ≡ case C of (coeffs, A) ⇒
(map (λ(a, l). ((a + c − 1) div c, l)) coeffs, (A + c − 1) div c)

definition *saturation* :: *'v pb-constraint ⇒ 'v pb-constraint* **where**
saturation C ≡ case C of (coeffs, A) ⇒
(map (λ(a, l). (min a A, l)) coeffs, A)

inductive *cpr-derives* :: *'v pb-constraint set ⇒ 'v pb-constraint ⇒ bool* **where**

hyp: $C \in CC \Longrightarrow$ *cpr-derives CC C*
| *lit-ax*: *cpr-derives CC* ([(*1*, *l*)], *0*)
| *lin-comb*: ⟦*cpr-derives CC C*; *cpr-derives CC D*⟧
$\Longrightarrow$ *cpr-derives CC* (*linear-combination cA C cB D*)
| *div-rule*: ⟦*cpr-derives CC C*; $c \geq 1$⟧ $\Longrightarrow$ *cpr-derives CC* (*division c C*)
| *sat-rule*: *cpr-derives CC C* $\Longrightarrow$ *cpr-derives CC* (*saturation C*)
— The semantic RUP rule over-approximates actual unit-propagation-based RUP, but the over-approximation is still sound for proving unsatisfiability of 0-1-constrained constraint sets.
| *rup*: *unsat-01* (*CC* ∪ {*constraint-neg C*}) $\Longrightarrow$ *cpr-derives CC C*

lemma *hyp-sound*: $C \in CC \Longrightarrow$ *models CC rho* $\Longrightarrow$ *satisfies C rho*
unfolding *models-def* **by** *blast*

lemma *pb-sum-map-mul*: *pb-sum* (*map* (λ(*a*, *l*). ($k * a$, *l*)) *xs*) *rho* = $k *$ *pb-sum xs rho*
proof (*induct xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons p xs*)
then obtain *a l* **where** *p* = (*a*, *l*) **by** (*cases p*) *auto*
then show *?case* **using** *Cons* **by** (*simp add*: *algebra-simps*)
qed

lemma *pb-sum-append*: *pb-sum* (*xs* @ *ys*) *rho* = *pb-sum xs rho* + *pb-sum ys rho*
by (*induct xs*) *auto*

lemma *lin-comb-sound*:
assumes *satisfies C rho satisfies D rho*
shows *satisfies* (*linear-combination nC C nD D*) *rho*
proof −
obtain *coeffsA A* **where** *C*: *C* = (*coeffsA*, *A*) **by** (*cases C*)
obtain *coeffsB B* **where** *D*: *D* = (*coeffsB*, *B*) **by** (*cases D*)
have *sum-distrib*: *pb-sum* (*map* (λ(*a*, *l*). ($nC * a$, *l*)) *coeffsA* @ *map* (λ(*b*, *l*). ($nD * b$, *l*)) *coeffsB*) *rho*
= $nC *$ *pb-sum coeffsA rho* + $nD *$ *pb-sum coeffsB rho*
by (*simp add*: *pb-sum-append pb-sum-map-mul*)
from *assms* **have** *Ap*: *pb-sum coeffsA rho* $\geq A$ **and** *Bp*: *pb-sum coeffsB rho* $\geq B$
unfolding *satisfies-def C D* **by** *auto*
have $nC *$ *pb-sum coeffsA rho* + $nD *$ *pb-sum coeffsB rho* $\geq nC * A + nD * B$
using *Ap Bp* **by** (*simp add*: *add-mono mult-le-mono*)
then show *?thesis*
using *sum-distrib* **unfolding** *satisfies-def linear-combination-def C D*
by *simp*
qed

lemma *pb-sum-ge-term*: (*a*, *l*) ∈ *set coeffs* $\Longrightarrow$ *pb-sum coeffs rho* $\geq a *$ *eval-lit l rho*

```
proof (induct coeffs arbitrary: a l)
  case Nil
  then show ?case by simp
next
  case (Cons p coeffs)
  then obtain a' l' where p-eq: p = (a', l') by (cases p) auto
  show ?case
  proof (cases (a, l) = (a', l'))
    case True
    then show ?thesis unfolding p-eq by simp
  next
    case False
    then have (a, l) ∈ set coeffs using Cons.prems p-eq by auto
    then have pb-sum coeffs rho ≥ a * eval-lit l rho using Cons.hyps by blast
    then show ?thesis unfolding p-eq
      by (simp add: le-trans le-add2)
  qed
qed

lemma ceil-ge:
  fixes a c :: nat
  assumes c > 0
  shows ((a + c − 1) div c) * c ≥ a
 using assms by (metis add.commute add-less-cancel-right bot-nat-0.not-eq-extremum
dec-less-imp-less-eq diff-less
      dividend-less-times-div le-eq-less-or-eq mult.commute zero-less-one)

lemma mul-ge-imp-ceil-div-ge:
  fixes m n c :: nat
  assumes c ≥ 1 m * c ≥ n
  shows m ≥ (n + c − 1) div c
proof (cases n = 0)
  case True    then show ?thesis using ‹c ≥ 1› by simp
next
  case False
  then have n > 0 by simp
  show ?thesis
  proof (rule ccontr)
    assume ¬ m ≥ (n + c − 1) div c
    then have m < (n + c − 1) div c by simp
    then have Suc m ≤ (n + c − 1) div c by simp
    then have Suc m * c ≤ ((n + c − 1) div c) * c
      by (metis ‹Suc m ≤ (n + c − 1) div c› order-refl mult-le-mono)
    also have ... ≤ n + c − 1
    proof −
      have *: ((n + c − 1) div c) * c + (n + c − 1) mod c = n + c − 1
        by (simp add: div-mult-mod-eq)
      show ?thesis by (simp add: add-leE *)
    qed
```

```
    finally have Suc m * c ≤ n + c - 1 .
    then have m * c + c ≤ n + c - 1
      by (metis ‹Suc m * c ≤ n + c - 1› mult-Suc add.commute)
     then have m * c + c ≤ (n - 1) + c using ‹n > 0› by (simp add:
add-diff-inverse-nat)
    then have m * c ≤ n - 1 using add-le-imp-le-right by blast
    then have m * c < n using ‹n > 0› by simp
    with assms(2) show False by simp
  qed
qed

lemma ceil-add-ineq:
  fixes a b c :: nat
  assumes c ≥ 1
  shows (a + c - 1) div c + (b + c - 1) div c ≥ (a + b + c - 1) div c
proof -
  have ((a + c - 1) div c + (b + c - 1) div c) * c = ((a + c - 1) div c) * c +
((b + c - 1) div c) * c
    by (simp add: distrib-right)
  also have ... ≥ a + b
  proof -
    have c > 0 using assms(1) by auto
    then have ((a + c - 1) div c) * c ≥ a and ((b + c - 1) div c) * c ≥ b
      using ceil-ge by blast+
    then show ?thesis by (simp add: add-le-mono)
  qed
  finally have sum-mul-ge: ((a + c - 1) div c + (b + c - 1) div c) * c ≥ a + b
.
  show ?thesis
    using assms sum-mul-ge by (rule mul-ge-imp-ceil-div-ge)
qed

lemma div-rule-sound:
  assumes c ≥ 1 satisfies C rho
  shows satisfies (division c C) rho
proof -
  obtain coeffs A where C: C = (coeffs, A) by (cases C)
  have sum-ge-A: pb-sum coeffs rho ≥ A
    using assms unfolding satisfies-def C by simp
  have pb-sum (map (λ(a,l). ((a + c - 1) div c, l)) coeffs) rho ≥ (pb-sum coeffs
rho + c - 1) div c
  proof (induct coeffs)
    case Nil
    then show ?case using ‹c ≥ 1› by simp
  next
    case (Cons p coeffs)
    obtain a l where p-eq: p = (a, l) by (cases p) auto
    have IH: pb-sum (map (λ(a,l). ((a + c - 1) div c, l)) coeffs) rho ≥ (pb-sum
coeffs rho + c - 1) div c
```

```
    using Cons by auto
  have mul-ineq: ((a + c − 1) div c) ∗ eval-lit l rho ∗ c ≥ a ∗ eval-lit l rho
  proof −
    have ((a + c − 1) div c) ∗ eval-lit l rho ∗ c = ((a + c − 1) div c) ∗ c ∗
eval-lit l rho
      by (simp add: mult.commute mult.left-commute mult.assoc)
    also have ... ≥ a ∗ eval-lit l rho
    proof −
      have c > 0 using assms(1) by auto
      then have ((a + c − 1) div c) ∗ c ≥ a using ceil-ge by blast
      then show ?thesis using mult-le-mono1 by blast
    qed
    finally show ?thesis .
  qed
  have div-ineq: ((a + c − 1) div c) ∗ eval-lit l rho ≥ (a ∗ eval-lit l rho + c −
1) div c
    by (rule mul-ge-imp-ceil-div-ge[OF assms(1) mul-ineq])
  have step1: ((a + c − 1) div c) ∗ eval-lit l rho + pb-sum (map (λ(a,l). ((a +
c − 1) div c, l)) coeffs) rho
    ≥ (a ∗ eval-lit l rho + c − 1) div c + (pb-sum coeffs rho + c − 1) div c
    using div-ineq IH by (simp add: add-le-mono)
  have step2: (a ∗ eval-lit l rho + c − 1) div c + (pb-sum coeffs rho + c − 1)
div c
    ≥ (a ∗ eval-lit l rho + pb-sum coeffs rho + c − 1) div c
    using assms(1) by (rule ceil-add-ineq)
  show ?case unfolding p-eq
    using step1 step2 by auto
qed
then have ∗: (pb-sum coeffs rho + c − 1) div c ≤ pb-sum (map (λ(a,l). ((a +
c − 1) div c, l)) coeffs) rho
  by simp
have A + c − 1 ≤ pb-sum coeffs rho + c − 1 using sum-ge-A by simp
then have (A + c − 1) div c ≤ (pb-sum coeffs rho + c − 1) div c by (rule
div-le-mono)
then have pb-sum (map (λ(a,l). ((a + c − 1) div c, l)) coeffs) rho ≥ (A + c −
1) div c
  using ∗ by (meson le-trans)
then show ?thesis unfolding satisfies-def division-def C by simp
qed

lemma sat-rule-sound:
  assumes satisfies C rho
  shows satisfies (saturation C) rho
proof −
  obtain coeffs A where C: C = (coeffs, A) by (cases C)
  have sum-ge-A: pb-sum coeffs rho ≥ A
    using assms unfolding satisfies-def C by simp
  have pb-sum (map (λ(a,l). (min a A, l)) coeffs) rho ≥ A
  proof (cases ∃(a,l)∈set coeffs. a ≥ A ∧ eval-lit l rho > 0)
```

```
    case True
    then obtain a l where (a,l)∈set coeffs a ≥ A eval-lit l rho > 0 by blast
    have (min a A, l) ∈ set (map (λ(a,l). (min a A, l)) coeffs)
      using ‹(a,l)∈set coeffs› by force
   then have pb-sum (map (λ(a,l). (min a A, l)) coeffs) rho ≥ (min a A) * eval-lit
l rho
      by (rule pb-sum-ge-term)
    moreover have (min a A) * eval-lit l rho ≥ A
      using ‹a ≥ A› ‹eval-lit l rho > 0› by (simp add: min-absorb2 mult-le-mono)
    ultimately show ?thesis by simp
  next
    case False
    then have ∀ (a,l)∈set coeffs. a < A ∨ eval-lit l rho = 0 by auto
    then have pb-sum (map (λ(a,l). (min a A, l)) coeffs) rho = pb-sum coeffs rho
    proof (induct coeffs)
      case Nil
      then show ?case by simp
    next
      case (Cons p coeffs)
      obtain a l where p = (a, l) by (cases p) auto
      then have a < A ∨ eval-lit l rho = 0 using Cons by auto
       moreover have pb-sum (map (λ(a,l). (min a A, l)) coeffs) rho = pb-sum
coeffs rho
        using Cons by auto
      ultimately show ?case unfolding ‹p = (a,l)›
        by (auto simp: min-def)
    qed
    then show ?thesis using sum-ge-A by simp
  qed
  then show ?thesis
    unfolding satisfies-def saturation-def C by simp
qed

theorem cpr-sound:
  assumes cpr-derives CC C and models CC rho and ∀ v. rho v = 0 ∨ rho v =
1
  shows satisfies C rho
  using assms
proof (induct arbitrary: rho rule: cpr-derives.induct)
  case (hyp CC C)
  then show ?case by (simp add: hyp-sound)
next
  case (lit-ax CC l)
  then show ?case by (simp add: satisfies-def eval-lit-def)
next
  case (lin-comb CC C D cA cB)
  then have satisfies C rho and satisfies D rho by blast+
  then show ?case by (blast intro: lin-comb-sound)
next
```

```
  case (div-rule CC C c)
  then show ?case using div-rule-sound by blast
next
  case (sat-rule CC C)
  then show ?case using sat-rule-sound by blast
next
  case (rup CC C)
  then have unsat: unsat-01 (CC ∪ {constraint-neg C}) by simp
  show ?case
  proof (rule ccontr)
    assume ¬ satisfies C rho
    obtain coeffs A where C: C = (coeffs, A) by (cases C)
    from ‹¬ satisfies C rho› have X-lt-A: pb-sum coeffs rho < A
      unfolding satisfies-def C by simp
    let ?X = pb-sum coeffs rho
    let ?Y = pb-sum (map (λ(a, l). (a, lit-neg l)) coeffs) rho
    let ?M = sum-list (map fst coeffs)
    have X-Y-ge-M: ?X + ?Y ≥ ?M
      using pb-sum-add-negated-ge-sum[of coeffs rho] by simp
    have satisfies (constraint-neg C) rho
    proof (cases A = 0)
      case True
      then have ?X < 0 using X-lt-A by simp
      then show ?thesis by simp
    next
      case False
      then have A ≥ 1 by simp
      then have X-le-A-minus-1: ?X ≤ A − 1 using X-lt-A by linarith
      have ?Y ≥ ?M − (A − 1)
      proof −
        have ?M − ?X ≤ ?Y
          using X-Y-ge-M by auto
        moreover have ?M − (A − 1) ≤ ?M − ?X
          using X-le-A-minus-1 by (simp add: diff-le-mono2)
        ultimately show ?thesis by (meson order-trans)
      qed
      then show ?thesis
        by (simp add: constraint-neg-def C satisfies-def Let-def)
    qed
    then have sat-neg: models (CC ∪ {constraint-neg C}) rho
      using ‹models CC rho› unfolding models-def by auto
    have ¬ unsat-01 (CC ∪ {constraint-neg C})
      unfolding unsat-01-def
      using sat-neg ‹∀ v. rho v = 0 ∨ rho v = 1› by blast
    with unsat show False by blast
  qed
qed
```

2.4 PB Task Encoding (Definition 1)

datatype *'v pvar* =
  StateVar 'v
| *CostBit nat* | *PrimedCostBit nat*
| *ReifI* | *ReifG* | *ReifT*
| *ReifEq 'v* | *ReifLeq 'v* | *ReifGeq 'v* | *ReifCostGe nat*
| *ReifAction nat*
| *ReifDeltaCostLower nat* — auxiliary reifying cost difference >= k
| *ReifDeltaCostUpper nat* — auxiliary reifying cost difference <= k
| *ReifDeltaCost nat* — from eq (3): reifies cost difference = k
| *ReifPrimedCostGe nat* — from eq (5): reifies primed cost >= k
| *ReifCert 'v* — fresh certificate variables, wrapped by Original/Primed when lifted

type-synonym *'v pconstraint* = *'v pvar pb-constraint*

definition *reif-fwd* :: *'v pvar literal* ⇒ (*nat* × *'v pvar literal*) *list* ⇒ *nat* ⇒ *'v pconstraint* **where**
  reif-fwd r coeffs A ≡ ([(*A*, *lit-neg r*)] @ *coeffs*, *A*)

definition *reif-bwd* :: *'v pvar literal* ⇒ (*nat* × *'v pvar literal*) *list* ⇒ *nat* ⇒ *'v pconstraint* **where**
  reif-bwd r coeffs A ≡
    let M = *sum-list* (*map fst coeffs*); *M'* = *M* + *1* − *A in*
    ([(*M'*, *r*)] @ *map* (λ(*a*, *l*). (*a*, *lit-neg l*)) *coeffs*, *M'*)

definition *reification* :: *'v pvar literal* ⇒ (*nat* × *'v pvar literal*) *list* ⇒ *nat* ⇒ *'v pconstraint set* **where**
  reification r coeffs A ≡ {*reif-fwd r coeffs A*, *reif-bwd r coeffs A*}

definition *state-lits* :: *'v*::*linorder set* ⇒ (*nat* × *'v pvar literal*) *list* **where**
  state-lits S ≡ *map* (λ*v*. (*1*, *Pos* (*StateVar v*))) (*sorted-list-of-set S*)

definition *neg-state-lits* :: *'v*::*linorder set* ⇒ (*nat* × *'v pvar literal*) *list* **where**
  neg-state-lits S ≡ *map* (λ*v*. (*1*, *Neg* (*StateVar v*))) (*sorted-list-of-set S*)

definition *encode-init* :: *'v*::*linorder strips-task* ⇒ *'v pconstraint set* **where**
  encode-init Π ≡
    let I = *init* Π; *V* = *vars* Π *in*
    reification (*Pos ReifI*)
      (*state-lits I* @ *neg-state-lits* (*V* − *I*)) (*card V*)

definition *encode-goal* :: *'v*::*linorder strips-task* ⇒ *'v pconstraint set* **where**
  encode-goal Π ≡
    let G = *goal* Π *in*
    reification (*Pos ReifG*) (*state-lits G*) (*card G*)

```
definition bits-needed :: nat ⇒ nat where
  bits-needed B ≡ (LEAST k. B < 2^k)

lemma bits-needed-sufficient: B < 2^(bits-needed B)
  unfolding bits-needed-def
proof (rule LeastI-ex)
  show ∃ k. B < 2^k
  proof (induction B)
    case 0
    show ?case by (rule exI[of - 1]) simp
  next
    case (Suc B)
    then obtain k where B < 2^k by blast
    then have Suc B ≤ 2^k by simp
    then have Suc B < 2^(Suc k) by (simp add: less-Suc-eq-le)
    then show ?case by blast
  qed
qed

lemma bits-needed-upper-bound: bits-needed B ≤ B + 1
proof −
  have B < 2^(B+1)
    by (metis add-lessD1 less-exp)
  then have bits-needed B ≤ B+1
    unfolding bits-needed-def by (rule Least-le)
  thus ?thesis by simp
qed

lemma c-mod-bits-needed: c < 2^(bits-needed B) ⟹ c mod (2::nat)^(bits-needed
B) = c
  by simp


definition encode-cost-ge :: nat ⇒ 'v pconstraint set where
  encode-cost-ge k ≡
   reification (Pos (ReifCostGe k)) (map (λi. (2^i, Pos (CostBit i))) [0..<bits-needed
k]) k
```

2.5 Transition Encoding (Equations 3–8)

```
datatype 'v var = Original 'v | Primed 'v

instantiation var :: (linorder) linorder
begin
definition less-eq-var :: 'a var ⇒ 'a var ⇒ bool where
  less-eq-var x y ≡ case (x, y) of
    (Original a, Original b) ⇒ a ≤ b
  | (Original -, Primed -)   ⇒ True
  | (Primed -,   Original -) ⇒ False
```

```
  | (Primed a,  Primed b)    ⇒ a ≤ b
definition less-var :: 'a var ⇒ 'a var ⇒ bool where
  less-var x y ≡ x ≤ y ∧ ¬ y ≤ x
instance by standard
  (auto simp: less-eq-var-def less-var-def split: var.splits)
end


definition prime-var :: 'v var ⇒ 'v var where
  prime-var x ≡ case x of Original v ⇒ Primed v | Primed v ⇒ Primed v


definition primed-pvar :: 'v var pvar ⇒ 'v var pvar where
  primed-pvar x ≡ map-pvar prime-var x




definition lift-to-var :: ('v pvar ⇒ nat) ⇒ ('v var pvar ⇒ nat) where
  lift-to-var f ≡ λx. case x of
    StateVar v      ⇒ f (StateVar (case v of Original w ⇒ w | Primed w ⇒ w))
  | CostBit i       ⇒ f (CostBit i)
  | PrimedCostBit i ⇒ f (CostBit i)
  | y               ⇒ f (map-pvar (case-var id id) y)


definition action-reifs :: 'v action list ⇒ 'v var pvar literal list where
  action-reifs as ≡ map (λi. Pos (ReifAction i)) [0..<length as]


definition encode-delta-cost :: nat ⇒ nat ⇒ 'v var pconstraint set where
  encode-delta-cost k num-bits ≡
    let cost-lits   = map (λi. (2^i, Pos (CostBit i))) [0..<num-bits];
        cost'-lits  = map (λi. (2^i, Pos (PrimedCostBit i))) [0..<num-bits];
        neg-c-lits  = map (λi. (2^i, Neg (CostBit i))) [0..<num-bits];
        neg-c'-lits = map (λi. (2^i, Neg (PrimedCostBit i))) [0..<num-bits];
        M           = 2^num-bits − 1
    in reification (Pos (ReifDeltaCostLower k)) (cost'-lits @ neg-c-lits) (k + M)
     ∪ reification (Pos (ReifDeltaCostUpper k)) (cost-lits @ neg-c'-lits) (M − k)
     ∪ reification (Pos (ReifDeltaCost k))
                   [(1, Pos (ReifDeltaCostLower k)), (1, Pos (ReifDeltaCostUpper k))]
2


definition encode-cost-ge-primed :: nat ⇒ 'v var pconstraint set where
  encode-cost-ge-primed k ≡
    reification (Pos (ReifPrimedCostGe k))
      (map (λi. (2^i, Pos (PrimedCostBit i))) [0..<bits-needed k]) k
```

definition *encode-eq-var* :: *'v* ⇒ *'v var pconstraint set* **where**
encode-eq-var v ≡
reification (*Pos* (*ReifGeq* (*Original v*))) [(*1*, *Pos* (*StateVar* (*Original v*))), (*1*, *Neg* (*StateVar* (*Primed v*)))] *1*
∪ *reification* (*Pos* (*ReifLeq* (*Original v*))) [(*1*, *Neg* (*StateVar* (*Original v*))), (*1*, *Pos* (*StateVar* (*Primed v*)))] *1*
∪ *reification* (*Pos* (*ReifEq* (*Original v*))) [(*1*, *Pos* (*ReifLeq* (*Original v*))), (*1*, *Pos* (*ReifGeq* (*Original v*)))] *2*

definition *action-constraint* ::
'v var pvar literal ⇒ *'v*::*linorder action* ⇒ *'v set* ⇒ *nat* ⇒ *'v var pconstraint* **where**
action-constraint r a V B ≡
let pre-lits = *map* (λ*v*. (*1*, *Pos* (*StateVar* (*Original v*)))) (*sorted-list-of-set* (*pre a*));
add-lits = *map* (λ*v*. (*1*, *Pos* (*StateVar* (*Primed v*)))) (*sorted-list-of-set* (*add a*));
del-lits = *map* (λ*v*. (*1*, *Neg* (*StateVar* (*Primed v*)))) (*sorted-list-of-set* (*del a*));
frame-lits = *map* (λ*v*. (*1*, *Pos* (*ReifEq* (*Original v*)))) (*sorted-list-of-set* (*V* − *evars a*));
delta-lit = [(*1*, *Pos* (*ReifDeltaCost* (*cost a*)))];
bound-lit = [(*1*, *Neg* (*ReifPrimedCostGe B*))];
A = *2* + *card* (*pre a*) + *card V*;
lhs = [(*A*, *lit-neg r*)] @ *delta-lit* @ *pre-lits* @ *add-lits* @ *del-lits* @ *frame-lits* @ *bound-lit*
in (*lhs*, *A*)

definition *action-selection-reif* ::
'v var pvar literal list ⇒ *'v var pconstraint set* **where**
action-selection-reif rs ≡
reification (*Pos ReifT*) (*map* (λ*r*. (*1*, *r*)) *rs*) *1*

definition *encode-transition* ::
'v::*linorder action list* ⇒ *'v set* ⇒ *nat* ⇒ *'v var pconstraint set* **where**
encode-transition as V B ≡
let rs = *action-reifs as*;
action-cs = (⋃ *i*<*length as*. {*action-constraint* (*rs*!*i*) (*as*!*i*) *V B*});
delta-cs = (⋃ *a*∈*set as*. *encode-delta-cost* (*cost a*) (*bits-needed B*));
eq-cs = (⋃ *v*∈*V*. *encode-eq-var v*);
primed-ge-c = *encode-cost-ge-primed B*;
sel-cs = *action-selection-reif rs*
in action-cs ∪ *delta-cs* ∪ *eq-cs* ∪ *primed-ge-c* ∪ *sel-cs*

2.6 PB Circuits (Definition 2)

type-synonym *'v pb-circuit* = (*'v pvar literal* × (*nat* × *'v pvar literal*) *list* × *nat*) *list* × *'v pvar literal*

definition *models-circuit* :: *'v pb-circuit* ⇒ (*'v pvar* ⇒ *nat*) ⇒ *bool* **where**
models-circuit C rho ≡
let (*pairs*, *out*) = *C in*
eval-lit out rho = *1* ∧
(∀ (*r*, *coeffs*, *A*) ∈ *set pairs*. *pb-sum coeffs rho* ≥ *A*)

definition *pvar-of-lit* :: *'v pvar literal* ⇒ *'v pvar* **where**
pvar-of-lit l ≡ *case l of Pos x* ⇒ *x* | *Neg x* ⇒ *x*

definition *is-input-pvar* :: *'v pvar* ⇒ *bool* **where**
is-input-pvar x ≡ (∃ *v*. *x* = *StateVar v*) ∨ (∃ *i*. *x* = *CostBit i*) ∨ (∃ *i*. *x* = *PrimedCostBit i*)

definition *constraint-pvars* :: *'v pconstraint* ⇒ *'v pvar set* **where**
constraint-pvars φ ≡ *pvar-of-lit* ‘ (*snd* ‘ *set* (*fst* φ))

definition *wf-circuit* :: *'v pb-circuit* ⇒ *bool* **where**
wf-circuit C ≡
let (*pairs*, *out*) = *C in*
(∀ *i* < *length pairs*.
let (*r-i*, *cs-i*, *A-i*) = *pairs* ! *i*;
allowed = {*x*. *is-input-pvar x*} ∪
(*pvar-of-lit* ‘ *fst* ‘ *set* (*take i pairs*))
in (*pvar-of-lit* ‘ *snd* ‘ *set cs-i*) ⊆ *allowed*) ∧
(*Pos* (*pvar-of-lit out*) ∈ *fst* ‘ *set pairs* ∨ *Neg* (*pvar-of-lit out*) ∈ *fst* ‘ *set pairs*)

2.7 State-Cost Assignments and Certificate Conditions

definition *state-rho* :: *'v*::*linorder strips-task* ⇒ *'v state* ⇒ *nat* ⇒ (*'v pvar* ⇒ *nat*) **where**
state-rho Π *s c* ≡ λ*x*. *case x of*
StateVar v ⇒ *if v* ∈ *s then 1 else 0*
| *CostBit i* ⇒ (*c div 2*^*i*) *mod 2*
| *ReifI* ⇒ (*let V* = *vars* Π; *I* = *init* Π *in*
if (∀ *v*∈*V*. (*v*∈*I* ⟷ *v*∈*s*)) *then 1 else 0*)
| *ReifG* ⇒ *if is-goal-state* Π *s then 1 else 0*
| *ReifCostGe k* ⇒ *if c* ≥ *k then 1 else 0*
| - ⇒ *0*

2.8 Lifting Circuits to the Primed/Unprimed Context

definition *lift-circuit* :: (*'v pvar* ⇒ *'v var pvar*) ⇒ *'v pb-circuit* ⇒ *'v var pb-circuit*
where
lift-circuit f C ≡
let (*pairs*, *out*) = *C in*
(*map* (λ(*r*, *cs*, *A*). (*map-literal f r*,
map (λ(*a*, *l*). (*a*, *map-literal f l*)) *cs*,
A)) *pairs*,
map-literal f out)

definition *orig-circuit* :: *'v pb-circuit* ⇒ *'v var pb-circuit* **where**
orig-circuit C ≡ *lift-circuit* (*map-pvar Original*) *C*

primrec *primed-pvar-map* :: *'v pvar* ⇒ *'v var pvar* **where**
primed-pvar-map (*StateVar v*) = *StateVar* (*Primed v*)
| *primed-pvar-map* (*CostBit i*) = *PrimedCostBit i*
| *primed-pvar-map* (*PrimedCostBit i*) = *PrimedCostBit i*
| *primed-pvar-map ReifI* = *ReifI*
| *primed-pvar-map ReifG* = *ReifG*
| *primed-pvar-map ReifT* = *ReifT*
| *primed-pvar-map* (*ReifEq v*) = *ReifEq* (*Primed v*)
| *primed-pvar-map* (*ReifLeq v*) = *ReifLeq* (*Primed v*)
| *primed-pvar-map* (*ReifGeq v*) = *ReifGeq* (*Primed v*)
| *primed-pvar-map* (*ReifCostGe n*) = *ReifCostGe n*
| *primed-pvar-map* (*ReifAction n*) = *ReifAction n*
| *primed-pvar-map* (*ReifDeltaCostLower n*) = *ReifDeltaCostLower n*
| *primed-pvar-map* (*ReifDeltaCostUpper n*) = *ReifDeltaCostUpper n*
| *primed-pvar-map* (*ReifDeltaCost n*) = *ReifDeltaCost n*
| *primed-pvar-map* (*ReifPrimedCostGe n*) = *ReifPrimedCostGe n*
| *primed-pvar-map* (*ReifCert x*) = *ReifCert* (*Primed x*)

definition *primed-circuit* :: *'v pb-circuit* ⇒ *'v var pb-circuit* **where**
primed-circuit C ≡ *lift-circuit primed-pvar-map C*

definition *cost-ge-constraint* :: *nat* ⇒ *'v pconstraint* **where**
cost-ge-constraint B ≡
(*map* (λ*i*. (2^*i*, *Pos* (*CostBit i*))) [*0*..<*bits-needed B*], *B*)

lemma *pb-sum-cost-bits*:
pb-sum (*map* (λ*i*. (2^*i*, *Pos* (*CostBit i*))) [*0*..<*k*]) (*state-rho* Π *s c*) = *c mod* 2^*k*
proof (*induct k*)
case *0*
then show *?case* **by** *simp*
next

case (*Suc k*)
have *pb-sum* (*map* (λ*i*. (2^*i*, *Pos* (*CostBit i*))) ([*0*..<*k*] @ [*k*])) (*state-rho* Π *s c*)
= *pb-sum* (*map* (λ*i*. (2^*i*, *Pos* (*CostBit i*))) [*0*..<*k*]) (*state-rho* Π *s c*)
+ *pb-sum* [(2^*k*, *Pos* (*CostBit k*))] (*state-rho* Π *s c*)
by (*simp add*: *pb-sum-append*)
also have ... = *c mod* 2^*k* + 2^*k* ∗ ((*c div* 2^*k*) *mod* 2)
using *Suc* **by** (*simp add*: *eval-lit-def state-rho-def*)
also have ... = *c mod* 2^(*Suc k*)
by (*metis add.commute mod-mult2-eq mult.commute power-Suc*)
finally show *?case* **by** *simp*
qed

2.9 Encoding Soundness Lemmas

lemma *pb-sum-state-lits-aux*:
assumes *distinct xs*
shows *pb-sum* (*map* (λ*v*. (*1*, *Pos* (*StateVar v*))) *xs*) (*state-rho* Π *s c*) = *card* (*set xs* ∩ *s*)
using *assms*
proof (*induct xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then have *a* ∉ *set xs* **and** *distinct xs* **by** *auto*
have *pb-sum* (*map* (λ*v*. (*1*, *Pos* (*StateVar v*))) (*a* # *xs*)) (*state-rho* Π *s c*)
= *1* ∗ *eval-lit* (*Pos* (*StateVar a*)) (*state-rho* Π *s c*)
+ *pb-sum* (*map* (λ*v*. (*1*, *Pos* (*StateVar v*))) *xs*) (*state-rho* Π *s c*)
by *simp*
also have *eval-lit* (*Pos* (*StateVar a*)) (*state-rho* Π *s c*) = (*if a* ∈ *s then 1 else 0*)
by (*simp add*: *eval-lit-def state-rho-def*)
also have *pb-sum* (*map* (λ*v*. (*1*, *Pos* (*StateVar v*))) *xs*) (*state-rho* Π *s c*) = *card* (*set xs* ∩ *s*)
using *Cons* **by** *simp*
also have *1* ∗ (*if a* ∈ *s then 1 else 0*) + *card* (*set xs* ∩ *s*) = *card* (*set* (*a* # *xs*) ∩ *s*)
using ‹*a* ∉ *set xs*› **by** (*auto simp*: *card-insert-if*)
finally show *?case* **.**
qed

lemma *pb-sum-state-lits*:
fixes *S* :: *'v*::*linorder set*
assumes *finite S*
shows *pb-sum* (*state-lits S*) (*state-rho* Π *s c*) = *card* (*S* ∩ *s*)
proof −
have *distinct* (*sorted-list-of-set S*) **by** *simp*
then have *pb-sum* (*map* (λ*v*. (*1*, *Pos* (*StateVar v*))) (*sorted-list-of-set S*)) (*state-rho* Π *s c*)
= *card* (*set* (*sorted-list-of-set S*) ∩ *s*)

```
    by (rule pb-sum-state-lits-aux)
  also have set (sorted-list-of-set S) = S using assms by simp
  finally show ?thesis unfolding state-lits-def by simp
qed

lemma pb-sum-neg-state-lits-aux:
  assumes distinct xs
  shows pb-sum (map (λv. (1, Neg (StateVar v))) xs) (state-rho Π s c) = card (set
xs − s)
  using assms
proof (induct xs)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  have pb-sum (map (λv. (1, Neg (StateVar v))) (a # xs)) (state-rho Π s c)
    = (if a ∈ s then 0 else 1) + pb-sum (map (λv. (1, Neg (StateVar v))) xs)
(state-rho Π s c)
    by (simp add: eval-lit-def state-rho-def)
  also have ... = (if a ∈ s then 0 else 1) + card (set xs − s)
    using Cons by simp
  also have ... = card (set (a # xs) − s)
  proof (cases a ∈ s)
    case True
    then show ?thesis by simp
  next
    case False
    then have notin-s: a ∉ s by simp
    have set (a # xs) − s = insert a (set xs − s) using notin-s by auto
    then show ?thesis using Cons notin-s by (simp add: card-insert-if)
  qed
  finally show ?case .
qed

lemma pb-sum-neg-state-lits:
  fixes S :: 'v::linorder set
  assumes finite S
  shows pb-sum (neg-state-lits S) (state-rho Π s c) = card (S − s)
proof −
  have distinct (sorted-list-of-set S) by simp
 then have pb-sum (map (λv. (1, Neg (StateVar v))) (sorted-list-of-set S)) (state-rho
Π s c)
    = card (set (sorted-list-of-set S) − s)
    by (rule pb-sum-neg-state-lits-aux)
  also have set (sorted-list-of-set S) = S using assms by simp
  finally show ?thesis unfolding neg-state-lits-def by simp
qed

lemma init-encoding-sound:
```

```
  fixes Π :: 'v::linorder strips-task
  assumes finite (vars Π) init Π ⊆ vars Π
  shows models (encode-init Π) (state-rho Π (init Π) 0)
proof −
  let ?r = Pos ReifI
  let ?I = init Π
  let ?V = vars Π
  let ?coeffs = state-lits ?I @ neg-state-lits (?V − ?I)
  let ?A = card ?V
  have reifI-val: eval-lit ?r (state-rho Π ?I 0) = 1
    by (simp add: eval-lit-def state-rho-def)
  have sum-eq: pb-sum ?coeffs (state-rho Π ?I 0) = ?A
  proof −
    have pb-sum (state-lits ?I) (state-rho Π ?I 0) = card (?I ∩ ?I)
      using assms(1) pb-sum-state-lits by (metis assms(2) rev-finite-subset)
    also have ?I ∩ ?I = ?I by simp
    finally have sum1: pb-sum (state-lits ?I) (state-rho Π ?I 0) = card ?I .
    have pb-sum (neg-state-lits (?V − ?I)) (state-rho Π ?I 0) = card ((?V − ?I)
− ?I)
      using assms(1) pb-sum-neg-state-lits by blast
    also have (?V − ?I) − ?I = ?V − ?I by blast
    finally have sum2: pb-sum (neg-state-lits (?V − ?I)) (state-rho Π ?I 0) =
card (?V − ?I) .
    have card-eq: card ?I + card (?V − ?I) = card ?V
    proof −
      have finV: finite ?V using assms(1) by simp
      have subIV: ?I ⊆ ?V using assms(2) by simp
      have finI: finite ?I using finV subIV finite-subset by blast
      have disj: ?I ∩ (?V − ?I) = {} by blast
      have card ?I + card (?V − ?I) = card (?I ∪ (?V − ?I))
        using finI finV by (metis card-Un-disjoint disj finite-Diff)
      also have ?I ∪ (?V − ?I) = ?V using subIV by auto
      finally show ?thesis .
    qed
    show ?thesis
      using sum1 sum2 card-eq by (simp add: pb-sum-append)
  qed
  have fwd-sound: satisfies (reif-fwd ?r ?coeffs ?A) (state-rho Π ?I 0)
  proof −
    have pb-sum (fst (reif-fwd ?r ?coeffs ?A)) (state-rho Π ?I 0)
      = ?A * eval-lit (lit-neg ?r) (state-rho Π ?I 0) + pb-sum ?coeffs (state-rho Π
?I 0)
      by (simp add: reif-fwd-def)
    also have eval-lit (lit-neg ?r) (state-rho Π ?I 0) = 0
      using reifI-val by (simp add: eval-lit-def lit-neg-def)
    also have pb-sum ?coeffs (state-rho Π ?I 0) = ?A by (rule sum-eq)
    finally have pb-sum (fst (reif-fwd ?r ?coeffs ?A)) (state-rho Π ?I 0) = ?A by
simp
    then show ?thesis by (simp add: satisfies-def reif-fwd-def)
```

```
  qed
  have bwd-sound: satisfies (reif-bwd ?r ?coeffs ?A) (state-rho Π ?I 0)
    using reifI-val by (simp add: satisfies-def reif-bwd-def Let-def le-add1)
  show ?thesis
    unfolding models-def encode-init-def reification-def
    by (simp add: fwd-sound bwd-sound Let-def)
qed

lemma goal-encoding-sound:
  fixes Π :: 'v::linorder strips-task
  assumes finite (vars Π) goal Π ⊆ vars Π is-goal-state Π s
  shows models (encode-goal Π) (state-rho Π s c)
proof −
  let ?r = Pos ReifG
  let ?G = goal Π
  let ?coeffs = state-lits ?G
  let ?A = card ?G
  have finG: finite ?G
    using assms(1,2) by (blast intro: finite-subset)
  have reifG-val: eval-lit ?r (state-rho Π s c) = 1
    using assms(3) by (simp add: eval-lit-def state-rho-def)
  have sum-eq: pb-sum ?coeffs (state-rho Π s c) = ?A
  proof −
    have pb-sum ?coeffs (state-rho Π s c) = card (?G ∩ s)
      using finG by (simp add: pb-sum-state-lits)
    also have ?G ∩ s = ?G
      using assms(3) unfolding is-goal-state-def by auto
    finally show ?thesis .
  qed
  have fwd-sound: satisfies (reif-fwd ?r ?coeffs ?A) (state-rho Π s c)
    using reifG-val sum-eq
    by (simp add: satisfies-def reif-fwd-def eval-lit-def lit-neg-def)
  have bwd-sound: satisfies (reif-bwd ?r ?coeffs ?A) (state-rho Π s c)
    using reifG-val by (simp add: satisfies-def reif-bwd-def Let-def le-add1)
  show ?thesis
    unfolding models-def encode-goal-def reification-def
    by (simp add: fwd-sound bwd-sound Let-def)
qed

lemma state-rho-range:
  state-rho Π s c x ≤ 1
  unfolding state-rho-def
  by (cases x; simp)

lemma state-rho-01:
  state-rho Π s c x = 0 ∨ state-rho Π s c x = 1
  unfolding state-rho-def
  by (cases x; simp add: mod-2-eq-odd)
```

lemma *eval-lit-plus-lit-neg*:
eval-lit l (*state-rho* Π *s c*) + *eval-lit* (*lit-neg l*) (*state-rho* Π *s c*) = *1*
using *state-rho-range*[*of* Π *s c*]
by (*simp add*: *eval-lit-def lit-neg-def split*: *literal.splits*)

lemma *pb-sum-add-negated*:
pb-sum coeffs (*state-rho* Π *s c*)
+ *pb-sum* (*map* (λ(*a*,*l*). (*a*, *lit-neg l*)) *coeffs*) (*state-rho* Π *s c*)
= *sum-list* (*map fst coeffs*)
proof (*induction coeffs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons ac coeffs*)
then have *IH*: *pb-sum coeffs* (*state-rho* Π *s c*) + *pb-sum* (*map* (λ(*a*,*l*). (*a*, *lit-neg l*)) *coeffs*) (*state-rho* Π *s c*) = *sum-list* (*map fst coeffs*)
by *blast*
obtain *x l* **where** *ac*: *ac* = (*x*, *l*) **by** (*cases ac*)
have *split*: *pb-sum* (*ac* # *coeffs*) (*state-rho* Π *s c*) = *x* ∗ *eval-lit l* (*state-rho* Π *s c*) + *pb-sum coeffs* (*state-rho* Π *s c*)
and *split-neg*: *pb-sum* (*map* (λ(*a*,*l*). (*a*, *lit-neg l*)) (*ac* # *coeffs*)) (*state-rho* Π *s c*) = *x* ∗ *eval-lit* (*lit-neg l*) (*state-rho* Π *s c*) + *pb-sum* (*map* (λ(*a*,*l*). (*a*, *lit-neg l*)) *coeffs*) (*state-rho* Π *s c*)
by (*simp-all add*: *ac*)
show *?case*
proof −
have *factor*: *x* ∗ *eval-lit l* (*state-rho* Π *s c*) + *x* ∗ *eval-lit* (*lit-neg l*) (*state-rho* Π *s c*) = *x*
by (*simp add*: *distrib-left*[*symmetric*] *eval-lit-plus-lit-neg*)
have *pb-sum* (*ac* # *coeffs*) (*state-rho* Π *s c*) + *pb-sum* (*map* (λ(*a*,*l*). (*a*, *lit-neg l*)) (*ac* # *coeffs*)) (*state-rho* Π *s c*)
= (*x* ∗ *eval-lit l* (*state-rho* Π *s c*) + *pb-sum coeffs* (*state-rho* Π *s c*)) +
(*x* ∗ *eval-lit* (*lit-neg l*) (*state-rho* Π *s c*) + *pb-sum* (*map* (λ(*a*,*l*). (*a*, *lit-neg l*)) *coeffs*) (*state-rho* Π *s c*))
by (*simp add*: *ac split split-neg*)
also have ... = *pb-sum coeffs* (*state-rho* Π *s c*) + *pb-sum* (*map* (λ(*a*,*l*). (*a*, *lit-neg l*)) *coeffs*) (*state-rho* Π *s c*) +
x ∗ *eval-lit l* (*state-rho* Π *s c*) + *x* ∗ *eval-lit* (*lit-neg l*) (*state-rho* Π *s c*)
by (*simp add*: *add-ac*)
also have ... = *sum-list* (*map fst coeffs*) + *x*
using *IH factor* **by** (*simp add*: *add-ac*)
also have ... = *x* + *sum-list* (*map fst coeffs*)
by (*simp add*: *add-ac*)
also have ... = *sum-list* (*map fst* (*ac* # *coeffs*))
by (*simp add*: *ac*)
finally show *?thesis* **.**
qed
qed

```
lemma sum-list-exp: sum-list (map ((^) 2) [0..<B]) = (2 :: nat) ^ B - 1
  by (induct B) auto


lemma cost-ge-encoding-sound:
  assumes c ≥ k c < 2^(bits-needed k)
  shows models (encode-cost-ge k) (state-rho Π s c)
proof −
  let ?r = Pos (ReifCostGe k)
  let ?coeffs = map (λi. (2^i, Pos (CostBit i))) [0..<bits-needed k]
  let ?A = k
  have reif-val: eval-lit ?r (state-rho Π s c) = 1
    using assms(1) by (simp add: eval-lit-def state-rho-def)
  have sum-eq: pb-sum ?coeffs (state-rho Π s c) = c
  proof −
    have pb-sum ?coeffs (state-rho Π s c) = c mod 2^(bits-needed k)
      by (rule pb-sum-cost-bits)
    also have ... = c
      using assms(2) by simp
    finally show ?thesis .
  qed
  have fwd-sound: satisfies (reif-fwd ?r ?coeffs ?A) (state-rho Π s c)
    using reif-val sum-eq assms(1)
    by (simp add: satisfies-def reif-fwd-def eval-lit-def lit-neg-def)
  have bwd-sound: satisfies (reif-bwd ?r ?coeffs ?A) (state-rho Π s c)
    using reif-val by (simp add: satisfies-def reif-bwd-def Let-def le-add1)
  show ?thesis
    using fwd-sound bwd-sound
    unfolding models-def encode-cost-ge-def reification-def by simp
qed

lemma cost-ge-encoding-below:
  assumes c < k
  shows models (encode-cost-ge k) (state-rho Π s c)
proof −
  let ?r = Pos (ReifCostGe k)
  let ?coeffs = map (λi. (2^i, Pos (CostBit i))) [0..<bits-needed k]
  have reif-val: state-rho Π s c (ReifCostGe k) = 0
    using assms by (simp add: state-rho-def)
  have eval-r: eval-lit ?r (state-rho Π s c) = 0
    by (simp add: eval-lit-def reif-val)
  have eval-neg-r: eval-lit (lit-neg ?r) (state-rho Π s c) = 1
    by (simp add: eval-lit-def lit-neg-def reif-val)
  have c-lt-pow: c < 2^(bits-needed k)
  proof −
    have c < k using assms .
    also have k < 2^(bits-needed k) by (rule bits-needed-sufficient)
    finally show ?thesis .
```

qed
have *pow-pos*: $k < (2::nat)\hat{}(bits\text{-}needed\ k)$
using *bits-needed-sufficient* **by** *metis*
have *sum-eq*: *pb-sum ?coeffs* (*state-rho* Π *s c*) = *c*
proof −
have *pb-sum ?coeffs* (*state-rho* Π *s c*) = *c mod* $2\hat{}(bits\text{-}needed\ k)$
by (*rule pb-sum-cost-bits*)
thus *?thesis* **using** *c-lt-pow* **by** *simp*
qed
have *sum-total*: *sum-list* (*map fst ?coeffs*) = $2\hat{}(bits\text{-}needed\ k) - (1 :: nat)$
by (*simp add*: *sum-list-exp o-def*)
have *neg-sum-eq*: *pb-sum* (*map* (λ(*a*, *l*). (*a*, *lit-neg l*)) *?coeffs*) (*state-rho* Π *s c*)
= $2\hat{}(bits\text{-}needed\ k) - 1 - c$
proof −
have *eq*: *pb-sum ?coeffs* (*state-rho* Π *s c*)
+ *pb-sum* (*map* (λ(*a*, *l*). (*a*, *lit-neg l*)) *?coeffs*) (*state-rho* Π *s c*)
= *sum-list* (*map fst ?coeffs*)
by (*metis* (*mono-tags*, *lifting*) *pb-sum-add-negated sum-total*)
thus *?thesis* **using** *sum-eq sum-total c-lt-pow*
by (*metis* (*no-types*, *lifting*) *diff-add-inverse*)
qed
have *fwd-sound*: *satisfies* (*reif-fwd ?r ?coeffs k*) (*state-rho* Π *s c*)
unfolding *satisfies-def reif-fwd-def Let-def*
using *eval-neg-r sum-eq* **by** *simp*
have *bwd-lhs*: *pb-sum* (*fst* (*reif-bwd ?r ?coeffs k*)) (*state-rho* Π *s c*)
= $2\hat{}(bits\text{-}needed\ k) - 1 - c$
proof −
have *thresh*: *sum-list* (*map fst ?coeffs*) − *k* + 1 = $2\hat{}(bits\text{-}needed\ k) - k$
using *sum-total pow-pos*
by (*metis Suc-diff-Suc Suc-eq-plus1 diff-diff-left plus-1-eq-Suc*)
show *?thesis*
unfolding *reif-bwd-def Let-def*
using *eval-r neg-sum-eq* **by** *auto*
qed
have *bwd-thresh*: *snd* (*reif-bwd ?r ?coeffs k*) = $2\hat{}(bits\text{-}needed\ k) - k$
unfolding *reif-bwd-def Let-def* **using** *sum-total pow-pos*
by (*metis One-nat-def Suc-eq-plus1 Suc-pred snd-conv zero-less-numeral zero-less-power*)
have *ineq*: $(2\hat{}(bits\text{-}needed\ k) - 1) - c \geq 2\hat{}(bits\text{-}needed\ k) - k$
proof −
have *c+1* ≤ *k* **using** *assms* **by** *simp*
then have $2\hat{}(bits\text{-}needed\ k) - k \leq 2\hat{}(bits\text{-}needed\ k) - (c+1)$ **by** (*simp add*: *diff-le-mono2*)
also have $2\hat{}(bits\text{-}needed\ k) - (c+1) = (2\hat{}(bits\text{-}needed\ k) - 1) - c$ **by** *simp*
finally show *?thesis* **.**
qed **have** *bwd-sound*: *satisfies* (*reif-bwd ?r ?coeffs k*) (*state-rho* Π *s c*)
by (*metis* (*mono-tags*, *lifting*) *bwd-lhs bwd-thresh ineq satisfies-def split-beta*)
show *?thesis*
using *fwd-sound bwd-sound*
unfolding *models-def encode-cost-ge-def reification-def* **by** *simp*

qed

definition *cost-circuit* :: *nat* ⇒ *'v pb-circuit* **where**
cost-circuit k ≡
let r = *Pos* (*ReifCostGe k*);
coeffs = *map* (λ*i*. (2^*i*, *Pos* (*CostBit i*))) [*0*..<*bits-needed k*]
in ([(*r*, *coeffs*, *k*)], *r*)

lemma *cost-circuit-complete*:
assumes *c* ≥ *k c* < *2*^(*bits-needed k*)
shows *models-circuit* (*cost-circuit k*) (*state-rho* Π *s c*)
proof −
let *?r* = *Pos* (*ReifCostGe k*)
let *?coeffs* = *map* (λ*i*. (*2*^*i*, *Pos* (*CostBit i*))) [*0*..<*bits-needed k*]
have *out*: *eval-lit ?r* (*state-rho* Π *s c*) = *1*
using *assms*(*1*) **by** (*simp add*: *eval-lit-def state-rho-def*)
have *sum-eq*: *pb-sum ?coeffs* (*state-rho* Π *s c*) = *c*
by (*simp add*: *pb-sum-cost-bits assms*(*2*))
show *?thesis*
unfolding *models-circuit-def cost-circuit-def Let-def*
using *out sum-eq assms*(*1*) **by** *simp*
qed

lemma *cost-circuit-sound*:
assumes *models-circuit* (*cost-circuit k*) (*state-rho* Π *s c*)
shows *c* ≥ *k*
proof (*cases c* < *k*)
case *True*
then have *eval-lit* (*Pos* (*ReifCostGe k*)) (*state-rho* Π *s c*) = *0*
by (*simp add*: *eval-lit-def state-rho-def*)
with *assms* **show** *?thesis*
unfolding *models-circuit-def cost-circuit-def Let-def* **by** *simp*
next
case *False*
then show *?thesis* **by** *simp*
qed

2.10 Lower-Bound Certificates (Definition 3)

record (*'v*) *certificate* =
cert-circuit :: *'v pb-circuit*
cert-actions :: *'v action list*

2.11 Paper Theorem 1 from CPR Certificates

definition *circuit-reif-pvars* :: *'v pb-circuit* ⇒ *'v pvar set* **where**
circuit-reif-pvars C ≡ *pvar-of-lit* ‘ *fst* ‘ *set* (*fst C*)

definition *circuit-constraints* :: *'v pb-circuit* ⇒ *'v pconstraint set* **where**

circuit-constraints $C \equiv \bigcup (r,\ cs,\ A) \in set\ (fst\ C).\ reification\ r\ cs\ A$

definition *distinct-reif-vars* :: *'v pb-circuit* ⇒ *bool* **where**
distinct-reif-vars $C \equiv$
let pairs = *fst C*
in $\forall\ i < length\ pairs.\ \forall\ j < length\ pairs.\ i \neq j \longrightarrow$
pvar-of-lit (*fst* (*pairs* ! *i*)) ≠ *pvar-of-lit* (*fst* (*pairs* ! *j*))

definition *neg-cost-ge-one* :: *nat* ⇒ *'v pconstraint* **where**
neg-cost-ge-one $B \equiv$
(*map* (λ*i*. (2^*i*, *Neg* (*CostBit i*))) [*0*..<*bits-needed B*], 2^(*bits-needed B*) − *1*)

definition *certificate-valid-cpr* ::
nat ⇒ *'v*::*linorder strips-task* ⇒ (*'v certificate*) ⇒ *bool* **where**
certificate-valid-cpr B Π *Cert* ≡
let C-φ = *cert-circuit Cert in*
finite (*vars* Π) ∧
init Π ⊆ *vars* Π ∧
goal Π ⊆ *vars* Π ∧
finite (*acts* Π) ∧
acts Π ⊆ *set* (*cert-actions Cert*) ∧
(∀ *a* ∈ *set* (*cert-actions Cert*).
pre a ⊆ *vars* Π ∧ *add a* ⊆ *vars* Π ∧ *del a* ⊆ *vars* Π) ∧
wf-circuit C-φ ∧
distinct-reif-vars C-φ ∧
(∀ (*r*, *φ*) ∈ *set* (*fst C-φ*). ∀ *v* ∈ *constraint-pvars φ*. ∀ *i*. *v* ≠ *PrimedCostBit i*) ∧
(∀ *v* ∈ *circuit-reif-pvars C-φ*.
¬ *is-input-pvar v* ∧ *v* ≠ *ReifI* ∧ (∀ *k*. *v* ≠ *ReifCostGe k*) ∧ *v* ≠ *ReifG* ∧
(∀ *k*. *v* ≠ *ReifDeltaCost k*) ∧ (∀ *k*. *v* ≠ *ReifDeltaCostLower k*) ∧
(∀ *k*. *v* ≠ *ReifDeltaCostUpper k*) ∧
(∀ *k*. *v* ≠ *ReifPrimedCostGe k*) ∧ *v* ≠ *ReifT* ∧
(∀ *u*. *v* ≠ *ReifGeq u*) ∧ (∀ *u*. *v* ≠ *ReifLeq u*) ∧ (∀ *u*. *v* ≠ *ReifEq u*) ∧
(∀ *i*. *v* ≠ *ReifAction i*) ∧ (∀ *i*. *v* ≠ *PrimedCostBit i*)) ∧
cpr-derives
(*encode-init* Π ∪ *circuit-constraints C-φ* ∪ *encode-cost-ge B* ∪
{*unit-clause* (*Pos ReifI*), *neg-cost-ge-one B*})
(*unit-clause* (*snd C-φ*)) ∧
cpr-derives
(*encode-goal* Π ∪ *circuit-constraints C-φ* ∪ *encode-cost-ge B* ∪
{*unit-clause* (*snd C-φ*), *unit-clause* (*Pos ReifG*)})
(*cost-ge-constraint B*) ∧
cpr-derives
(*encode-transition* (*cert-actions Cert*) (*vars* Π) *B* ∪
circuit-constraints (*orig-circuit C-φ*) ∪
circuit-constraints (*primed-circuit C-φ*) ∪
encode-cost-ge B ∪

```
    {unit-clause (snd (orig-circuit C-φ)), unit-clause (Pos ReifT)})
   (unit-clause (snd (primed-circuit C-φ)))

lemma map-pvar-Original-inj:
  map-pvar Original x = map-pvar Original y ⟷ (x :: 'v pvar) = y
  by (cases x; cases y; auto)

lemma map-literal-eq-Pos-conv: map-literal f x = Pos y ⟷ (∃ z. x = Pos z ∧ f
z = y)
  by (cases x; auto)
lemma map-literal-eq-Neg-conv: map-literal f x = Neg y ⟷ (∃ z. x = Neg z ∧ f
z = y)
  by (cases x; auto)
lemmas map-literal-convs = map-literal-eq-Pos-conv map-literal-eq-Neg-conv

lemma not-Pos-Neg: (∀ x. b ≠ Pos x) ⟷ (∃ x. b = Neg x)
  by (cases b; simp)

lemma wf-orig-circuit:
  assumes wf-circuit (C :: 'v pb-circuit)
  shows wf-circuit (orig-circuit C)
proof −
  obtain pairs out where C: C = (pairs, out) by (cases C)
  have F1: ∀ i < length pairs. ∀ r cs A. pairs ! i = (r, cs, A) ⟶
      pvar-of-lit ‘ snd ‘ set cs ⊆
      Collect is-input-pvar ∪ pvar-of-lit ‘ fst ‘ set (take i pairs)
    using assms unfolding wf-circuit-def C Let-def by (fastforce simp: image-iff
split: prod.splits)
  have F2: Pos (pvar-of-lit out) ∈ fst ‘ set pairs ∨ Neg (pvar-of-lit out) ∈ fst ‘ set
pairs
    using assms unfolding wf-circuit-def C Let-def by auto
  have pof: ∀ l :: 'v pvar literal. pvar-of-lit (map-literal (map-pvar Original) l) =
      map-pvar Original (pvar-of-lit l)
    by (simp add: pvar-of-lit-def split: literal.splits)
  have cs-map: ∀ cs :: (nat × 'v pvar literal) list.
     pvar-of-lit ‘ snd ‘ set (map (λ(a,l). (a, map-literal (map-pvar Original) l)) cs)
=
      (map-pvar Original ‘ (pvar-of-lit ‘ snd ‘ set cs) :: 'v var pvar set)
    by (force simp: pvar-of-lit-def image-iff map-literal-convs split: literal.splits)
  have inp: ∀ v :: 'v pvar. is-input-pvar v ⟶ is-input-pvar (map-pvar Original v)
    by (auto simp: is-input-pvar-def split: pvar.splits)
  have tk: ∀ i. pvar-of-lit ‘ fst ‘
      set (take i (map (λ(r, cs, A). (map-literal (map-pvar Original) r,
          map (λ(a, l). (a, map-literal (map-pvar Original) l)) cs, A)) pairs)) =
      map-pvar Original ‘ (pvar-of-lit ‘ fst ‘ set (take i pairs))
    by (induction pairs; auto simp: pof take-Cons' split: if-splits)
  let ?f = map-pvar Original
  let ?lp = map (λ(r, cs, A). (map-literal ?f r,
          map (λ(a, l). (a, map-literal ?f l)) cs, A)) pairs
```

```
  have wf1: ∀ i < length ?lp.
      let (r-i, cs-i, A-i) = ?lp ! i;
          allowed = Collect is-input-pvar ∪ pvar-of-lit ‘ fst ‘ set (take i ?lp)
      in pvar-of-lit ‘ snd ‘ set cs-i ⊆ allowed
  proof (intro allI impI)
    fix i assume i-lt: i < length ?lp
    have i-lt': i < length pairs using i-lt by simp
    obtain r cs A where pair-i: pairs ! i = (r, cs, A)
      by (metis prod-cases3)
    have sub: pvar-of-lit ‘ snd ‘ set cs ⊆
          Collect is-input-pvar ∪ pvar-of-lit ‘ fst ‘ set (take i pairs)
      using F1 i-lt' pair-i by auto
    have lp-i: ?lp ! i = (map-literal ?f r,
        map (λ(a, l). (a, map-literal ?f l)) cs, A)
      by (simp add: pair-i i-lt')
    have cs-eq: pvar-of-lit ‘ snd ‘ set (map (λ(a, l). (a, map-literal ?f l)) cs) =
        map-pvar Original ‘ (pvar-of-lit ‘ snd ‘ set cs)
      using cs-map by auto
    have rhs: pvar-of-lit ‘ fst ‘ set (take i ?lp) =
          map-pvar Original ‘ (pvar-of-lit ‘ fst ‘ set (take i pairs))
      using tk by simp
    show let (r-i, cs-i, A-i) = ?lp ! i;
          allowed = Collect is-input-pvar ∪ pvar-of-lit ‘ fst ‘ set (take i ?lp)
      in pvar-of-lit ‘ snd ‘ set cs-i ⊆ allowed
      unfolding Let-def lp-i fst-conv snd-conv cs-eq rhs
      using sub inp pof by (force simp: image-iff map-pvar-Original-inj)
  qed
  show ?thesis
    unfolding wf-circuit-def orig-circuit-def lift-circuit-def C Let-def prod.case
  proof (intro conjI allI impI, goal-cases)
    case (1 i)
    then show ?case using wf1 by (simp add: split-beta Let-def)
  next
    case 2
     then show ?case using F2 by (force simp: pvar-of-lit-def split: prod.splits
literal.splits)
  qed
qed


definition in-M :: 'v::linorder pb-circuit ⇒ 'v strips-task ⇒ 'v state ⇒ nat ⇒ bool
where
  in-M C Π s c ≡
    ∃ rho. (∀ v. rho v = 0 ∨ rho v = 1)
      ∧ models (circuit-constraints C) rho
      ∧ eval-lit (snd C) rho = 1
      ∧ (∀ v. rho (StateVar v) = (if v ∈ s then 1 else 0))
      ∧ (∀ i. rho (CostBit i) = (c div 2^i) mod 2)
      ∧ (∀ i. rho (PrimedCostBit i) = 0)
```

lemma *state-rho-StateVar-eq*:
state-rho Π *s c* (*StateVar v*) = (*if v* ∈ *s then 1 else 0*)
by (*simp add*: *state-rho-def*)

lemma *state-rho-CostBit-eq*:
state-rho Π *s c* (*CostBit i*) = (*c div* 2^*i*) *mod 2*
by (*simp add*: *state-rho-def*)

definition *extend-rho* :: *'v pb-circuit* ⇒ (*'v pvar* ⇒ *nat*) ⇒ (*'v pvar* ⇒ *nat*) ⇒
(*'v pvar* ⇒ *nat*) **where**
extend-rho C rho-circ rho-base ≡ λ *x*.
if x ∈ *circuit-reif-pvars C then rho-circ x else rho-base x*

lemma *extend-rho-base-on-non-reif*:
assumes *x* ∉ *circuit-reif-pvars C*
shows *extend-rho C rho-circ rho-base x* = *rho-base x*
unfolding *extend-rho-def* **using** *assms* **by** *simp*

lemma *pb-sum-congr*:
assumes ∀ (*a*, *l*) ∈ *set coeffs*. *eval-lit l f* = *eval-lit l g*
shows *pb-sum coeffs f* = *pb-sum coeffs g*
using *assms*
proof (*induction coeffs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons p cs*)
obtain *a l* **where** *pl*: *p* = (*a*, *l*) **by** (*cases p*)
with *Cons* **have** *head*: *eval-lit l f* = *eval-lit l g* **by** *auto*
from *Cons pl* **have** *pb-sum cs f* = *pb-sum cs g* **by** *auto*
with *head pl* **show** *?case* **by** *simp*
qed

lemma *pb-sum-add-negated-gen*:
assumes ∀ *v*. *rho v* = *0* ∨ *rho v* = *1*
shows *pb-sum coeffs rho* + *pb-sum* (*map* (λ(*a*,*l*). (*a*, *lit-neg l*)) *coeffs*) *rho*
= *sum-list* (*map fst coeffs*)
proof (*induction coeffs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons ac cs*)
obtain *x l* **where** *ac*: *ac* = (*x*, *l*) **by** (*cases ac*)
have *step*: *x* ∗ *eval-lit l rho* + *x* ∗ *eval-lit* (*lit-neg l*) *rho* = *x*
proof (*cases l*)
case (*Pos v*)
have *rho v* = *0* ∨ *rho v* = *1* **using** *assms* **by** *simp*
then show *?thesis* **by** (*auto simp*: *Pos eval-lit-def lit-neg-def*)
next
case (*Neg v*)

```
    have rho v = 0 ∨ rho v = 1 using assms by simp
    then show ?thesis by (auto simp: Neg eval-lit-def lit-neg-def)
  qed
  show ?case using Cons.IH ac step by simp
qed

lemma pb-sum-extend-rho-eq-base:
  assumes ∀ (a, l) ∈ set coeffs. pvar-of-lit l ∉ circuit-reif-pvars C
  shows pb-sum coeffs (extend-rho C rho-circ rho-base) = pb-sum coeffs rho-base
  using assms
proof (induction coeffs)
  case Nil then show ?case by simp
next
  case (Cons p coeffs)
  obtain a l where p = (a, l) by (cases p)
  with Cons have pvar-of-lit l ∉ circuit-reif-pvars C by auto
  then have eval-lit l (extend-rho C rho-circ rho-base) = eval-lit l rho-base
    by (cases l; simp add: eval-lit-def extend-rho-def pvar-of-lit-def)
  with Cons show ?case by (simp add: ‹p = (a, l)›)
qed

lemma satisfies-extend-rho-eq-base:
  assumes ∀ v ∈ constraint-pvars φ. v ∉ circuit-reif-pvars C
  shows satisfies φ (extend-rho C rho-circ rho-base) = satisfies φ rho-base
proof −
  obtain coeffs A where φ = (coeffs, A) by (cases φ)
  with assms have ∀ (a, l) ∈ set coeffs. pvar-of-lit l ∉ circuit-reif-pvars C
    unfolding constraint-pvars-def by fastforce
  then show ?thesis
    unfolding ‹φ = (coeffs, A)› satisfies-def
    by (simp add: pb-sum-extend-rho-eq-base)
qed

lemma models-circuit-constraints-extend:
  assumes models (circuit-constraints C) rho-circ
    and ∀ (r, φ) ∈ set (fst C). ∀ v ∈ constraint-pvars φ. is-input-pvar v ∨ v ∈
circuit-reif-pvars C
    and ∀ v. is-input-pvar v ⟶ rho-base v = rho-circ v
  shows models (circuit-constraints C) (extend-rho C rho-circ rho-base)
proof (unfold models-def, intro ballI)
  fix φ assume φ ∈ circuit-constraints C
  then obtain r cs A where rφ: (r, cs, A) ∈ set (fst C)
    and φ-reif: φ ∈ reification r cs A
    unfolding circuit-constraints-def by blast
  have cpvars: ∀ v ∈ constraint-pvars φ. is-input-pvar v ∨ v ∈ circuit-reif-pvars C
  proof (intro ballI)
    fix v assume vin: v ∈ constraint-pvars φ
    have sub: constraint-pvars φ ⊆ {pvar-of-lit r} ∪ pvar-of-lit ‘ snd ‘ set cs
      using φ-reif
```

unfolding *reification-def reif-fwd-def reif-bwd-def constraint-pvars-def*
by (*force simp*: *pvar-of-lit-def lit-neg-def Let-def split*: *if-splits literal.splits*)
then consider $v = pvar\text{-}of\text{-}lit\ r \mid v \in pvar\text{-}of\text{-}lit\ `\ snd\ `\ set\ cs$
using *vin* **by** *blast*
then show *is-input-pvar* $v \vee v \in$ *circuit-reif-pvars C*
proof *cases*
case *1*
have *pvar-of-lit* $r \in$ *circuit-reif-pvars C*
using $r\varphi$ **unfolding** *circuit-reif-pvars-def* **by** (*force simp*: *image-iff*)
with *1* **show** *?thesis* **by** *blast*
next
case *2*
have $\forall\ v \in pvar\text{-}of\text{-}lit\ `\ snd\ `\ set\ cs.$ *is-input-pvar* $v \vee v \in$ *circuit-reif-pvars C*
using *assms*(*2*) $r\varphi$ **unfolding** *constraint-pvars-def* **by** (*fastforce split*: *prod.splits*)
with *2* **show** *?thesis* **by** *blast*
qed
qed
have *sat-circ*: *satisfies* φ *rho-circ*
using *assms*(*1*) $\langle \varphi \in$ *circuit-constraints C*$\rangle$ **unfolding** *models-def* **by** *blast*
have *coeffs-eq*: $\forall\ (a,\ l) \in set\ (fst\ \varphi).$ *eval-lit l* (*extend-rho C rho-circ rho-base*) = *eval-lit l rho-circ*
proof (*safe*)
fix *a l* **assume** *al*: $(a,\ l) \in set\ (fst\ \varphi)$
have *pvar-of-lit l* $\in$ *constraint-pvars* φ
unfolding *constraint-pvars-def* **using** *al* **by** *force*
with *cpvars* **have** *is-input-pvar* (*pvar-of-lit l*) $\vee$ *pvar-of-lit l* $\in$ *circuit-reif-pvars C* **by** *auto*
then show *eval-lit l* (*extend-rho C rho-circ rho-base*) = *eval-lit l rho-circ*
using *assms*(*3*)
by (*cases l*; *auto simp add*: *eval-lit-def extend-rho-def circuit-reif-pvars-def pvar-of-lit-def*)
qed
have *sum-eq*: *pb-sum* (*fst* φ) (*extend-rho C rho-circ rho-base*) = *pb-sum* (*fst* φ) *rho-circ*
using *coeffs-eq* **by** (*rule pb-sum-congr*)
with *sat-circ* **show** *satisfies* φ (*extend-rho C rho-circ rho-base*)
unfolding *satisfies-def* **by** (*cases* φ) *simp*
qed

lemma *eval-output-extend-rho*:
assumes *pvar-of-lit* (*snd C*) $\in$ *circuit-reif-pvars C*
shows *eval-lit* (*snd C*) (*extend-rho C rho-circ rho-base*) = *eval-lit* (*snd C*) *rho-circ*
proof −
have *pvar-of-lit* (*snd C*) $\in$ *circuit-reif-pvars C* **using** *assms* **.**
then show *?thesis*
by (*cases snd C*; *simp add*: *eval-lit-def extend-rho-def circuit-reif-pvars-def pvar-of-lit-def*)
qed

lemma *pb-sum-neg-cost-bits-zero*:
assumes $\forall\ i < k.\ rho\ (CostBit\ i) = 0$
shows *pb-sum* (*map* ($\lambda i.\ (2\hat{}i,\ Neg\ (CostBit\ i))$) $[0..<k]$) *rho* $= 2\hat{}k - 1$
using *assms*
by (*induction k*) (*auto simp*: *pb-sum-append eval-lit-def*)

lemma *satisfies-neg-cost-ge-one*:
assumes $\forall\ i.\ rho\ (CostBit\ i) = 0$
shows *satisfies* (*neg-cost-ge-one B*) *rho*
unfolding *neg-cost-ge-one-def satisfies-def*
using *pb-sum-neg-cost-bits-zero*[*of bits-needed B rho*] *assms* **by** *simp*

lemma *in-M-init*:
fixes $\Pi :: 'v$::*linorder strips-task* **and** *C-*$\varphi :: 'v$ *pb-circuit*
assumes *fin*: *finite* (*vars* Π) **and** *init-sub*: *init* $\Pi \subseteq$ *vars* Π
and *wf*: *wf-circuit C-*φ
and *out-reif*: *pvar-of-lit* (*snd C-*φ) $\in$ *circuit-reif-pvars C-*φ
and *circ-vars*: $\forall\ (r,\ \varphi) \in$ *set* (*fst C-*φ).
$\forall\ v \in$ *constraint-pvars* φ. *is-input-pvar* $v \vee v \in$ *circuit-reif-pvars C-*φ
and *disjoint*: $\forall\ v \in$ *circuit-reif-pvars C-*φ.
$\neg$ *is-input-pvar* $v \wedge v \neq$ *ReifI* $\wedge$ ($\forall\ k.\ v \neq$ *ReifCostGe* k)
and *realiz*: $\forall$ *base*. ($\forall\ v.$ *base* $v = 0 \vee$ *base* $v = 1$) $\longrightarrow$ ($\exists$ *rho*.
models (*circuit-constraints C-*φ) *rho*
$\wedge$ ($\forall\ v.$ *is-input-pvar* $v \longrightarrow$ *rho* $v =$ *base* v)
$\wedge$ ($\forall\ v.$ *rho* $v = 0 \vee$ *rho* $v = 1$))
and *B-pos*: $B \geq 1$
and *cpr-init*: *cpr-derives*
(*encode-init* $\Pi \cup$ *circuit-constraints C-*$\varphi \cup$ *encode-cost-ge* $B \cup$
{*unit-clause* (*Pos ReifI*), *neg-cost-ge-one* B})
(*unit-clause* (*snd C-*φ))
shows *in-M C-*φ Π (*init* Π) *0*
proof $-$
let *?s* $=$ *init* Π
let *?base* $=$ *state-rho* Π *?s 0*
have *base-01*: $\forall\ v.$ *?base* $v = 0 \vee$ *?base* $v = 1$ **using** *state-rho-01* **by** *blast*
obtain *rho-circ* **where**
circ-sat: *models* (*circuit-constraints C-*φ) *rho-circ*
and *inp-eq*: $\forall\ v.$ *is-input-pvar* $v \longrightarrow$ *rho-circ* $v =$ *?base* v
and *circ-01*: $\forall\ v.$ *rho-circ* $v = 0 \vee$ *rho-circ* $v = 1$
using *realiz base-01* **by** *blast*
let *?rho* $=$ *extend-rho C-*φ *rho-circ ?base*
have *inp-sym*: $\forall\ v.$ *is-input-pvar* $v \longrightarrow$ *?base* $v =$ *rho-circ* v
using *inp-eq* **by** *auto*
have *rho-circ-sat*: *models* (*circuit-constraints C-*φ) *?rho*
using *models-circuit-constraints-extend*[*OF circ-sat circ-vars inp-sym*] **.**
have *sv-rho*: $\forall\ v.$ *?rho* (*StateVar* v) $=$ (*if* $v \in$ *?s then 1 else 0*)
proof
fix v

```
    show ?rho (StateVar v) = (if v ∈ ?s then 1 else 0)
    proof (cases StateVar v ∈ circuit-reif-pvars C-φ)
      case True
      then have ?rho (StateVar v) = rho-circ (StateVar v)
        by (simp add: extend-rho-def)
      also have ... = ?base (StateVar v)
         using inp-eq[rule-format, of StateVar v] unfolding is-input-pvar-def by
auto
      also have ... = (if v ∈ ?s then 1 else 0)
        by (simp add: state-rho-def)
      finally show ?thesis .
    next
      case False
      then have ?rho (StateVar v) = ?base (StateVar v)
        by (simp add: extend-rho-base-on-non-reif)
      also have ... = (if v ∈ ?s then 1 else 0)
        by (simp add: state-rho-def)
      finally show ?thesis .
    qed
  qed
  have cb-rho: ∀ i. ?rho (CostBit i) = (0 div 2^i) mod 2
  proof
    fix i
    show ?rho (CostBit i) = (0 div 2^i) mod 2
    proof (cases CostBit i ∈ circuit-reif-pvars C-φ)
      case True
      then have ?rho (CostBit i) = rho-circ (CostBit i)
        by (simp add: extend-rho-def)
      also have ... = ?base (CostBit i)
       using inp-eq[rule-format, of CostBit i] unfolding is-input-pvar-def by auto
      also have ... = (0 div 2^i) mod 2
        by (simp add: state-rho-def)
      finally show ?thesis .
    next
      case False
      then have ?rho (CostBit i) = ?base (CostBit i)
        by (simp add: extend-rho-base-on-non-reif)
      also have ... = (0 div 2^i) mod 2
        by (simp add: state-rho-def)
      finally show ?thesis .
    qed
  qed
  have primed-cb-rho: ∀ i. ?rho (PrimedCostBit i) = (0 div 2^i) mod 2
  proof
    fix i
    show ?rho (PrimedCostBit i) = (0 div 2^i) mod 2
    proof (cases PrimedCostBit i ∈ circuit-reif-pvars C-φ)
      case True
      then have ?rho (PrimedCostBit i) = rho-circ (PrimedCostBit i)
```

```
      by (simp add: extend-rho-def)
    also have ... = ?base (PrimedCostBit i)
      using inp-eq[rule-format, of PrimedCostBit i] unfolding is-input-pvar-def
by auto
    also have ... = (0 div 2^i) mod 2
      by (simp add: state-rho-def)
    finally show ?thesis .
  next
    case False
    then have ?rho (PrimedCostBit i) = ?base (PrimedCostBit i)
      by (simp add: extend-rho-base-on-non-reif)
    also have ... = (0 div 2^i) mod 2
      by (simp add: state-rho-def)
    finally show ?thesis .
  qed
 qed
 have enc-init-sat: models (encode-init Π) ?rho
 proof (unfold models-def, intro ballI)
  fix φ assume φ-in: φ ∈ encode-init Π
  have base-sat: satisfies φ ?base
    using init-encoding-sound[OF fin init-sub] φ-in unfolding models-def by
auto
  have not-reif: ∀ v ∈ constraint-pvars φ. v ∉ circuit-reif-pvars C-φ
  proof
    fix v assume v ∈ constraint-pvars φ
    with φ-in have v = ReifI ∨ (∃ u. v = StateVar u)
      unfolding encode-init-def reification-def constraint-pvars-def
          reif-fwd-def reif-bwd-def state-lits-def neg-state-lits-def pvar-of-lit-def
      by (auto split: literal.splits simp: lit-neg-def Let-def)
    then show v ∉ circuit-reif-pvars C-φ
      using disjoint unfolding is-input-pvar-def by auto
  qed
  from satisfies-extend-rho-eq-base[OF not-reif] base-sat
  show satisfies φ ?rho by simp
 qed
 have base-cost-sat: models (encode-cost-ge B :: 'v pconstraint set) ?base
  by (rule cost-ge-encoding-below) (use B-pos in simp)
 have enc-cost-sat: models (encode-cost-ge B :: 'v pconstraint set) ?rho
 proof (unfold models-def, intro ballI)
  fix φ :: 'v pconstraint assume φ-in: φ ∈ (encode-cost-ge B :: 'v pconstraint set)
  have base-sat: satisfies φ ?base
    using base-cost-sat φ-in unfolding models-def by auto
  have not-reif: ∀ v ∈ constraint-pvars φ. v ∉ circuit-reif-pvars C-φ
  proof
    fix v assume v ∈ constraint-pvars φ
    with φ-in have v = ReifCostGe B ∨ (∃ i. v = CostBit i)
      unfolding encode-cost-ge-def reification-def constraint-pvars-def
          reif-fwd-def reif-bwd-def pvar-of-lit-def
      by (auto split: literal.splits simp: lit-neg-def Let-def)
```

```
    then show v ∉ circuit-reif-pvars C-φ
      using disjoint unfolding is-input-pvar-def by auto
  qed
  from satisfies-extend-rho-eq-base[OF not-reif] base-sat
  show satisfies φ ?rho by simp
 qed
 have reifI-rho: ?rho ReifI = 1
 proof −
  have ReifI ∉ circuit-reif-pvars C-φ using disjoint by auto
 then have ?rho ReifI = ?base ReifI by (simp add: extend-rho-base-on-non-reif)
  then show ?thesis by (simp add: state-rho-def)
 qed
 have axiom-rho: models {unit-clause (Pos ReifI), neg-cost-ge-one B} ?rho
 proof −
  have sat1: satisfies (unit-clause (Pos ReifI)) ?rho
    unfolding satisfies-def unit-clause-def
    by (simp add: eval-lit-def reifI-rho)
  have sat2: satisfies (neg-cost-ge-one B) ?rho
    by (rule satisfies-neg-cost-ge-one) (simp add: cb-rho)
  show ?thesis using sat1 sat2 by (simp add: models-def)
 qed
 have hyp-model: ∀ φ ∈ encode-init Π ∪ circuit-constraints C-φ ∪ encode-cost-ge
B ∪
    {unit-clause (Pos ReifI), neg-cost-ge-one B}. satisfies φ ?rho
  using enc-init-sat rho-circ-sat enc-cost-sat axiom-rho
  unfolding models-def circuit-constraints-def by auto
 have output-sat: satisfies (unit-clause (snd C-φ)) ?rho
  using cpr-sound[OF cpr-init] hyp-model
  using models-def by (metis (no-types, lifting) base-01 circ-01 extend-rho-def)
 have rho-01: ∀ v. ?rho v = 0 ∨ ?rho v = 1
    unfolding extend-rho-def
    using circ-01 by (auto simp: state-rho-def is-input-pvar-def split: pvar.splits
var.splits)
 have output-eval: eval-lit (snd C-φ) ?rho = 1
  using output-sat unfolding satisfies-def unit-clause-def
 proof (cases snd C-φ)
  case (Pos x1)
  have h01: ?rho x1 = 0 ∨ ?rho x1 = 1 using rho-01 by blast
  moreover have hge: 1 ≤ ?rho x1
    using output-sat unfolding satisfies-def unit-clause-def
    by (simp add: Pos eval-lit-def)
  ultimately show ?thesis by (simp add: Pos eval-lit-def)
 next
  case (Neg x2)
  have h01: ?rho x2 = 0 ∨ ?rho x2 = 1 using rho-01 by blast
  moreover have hge: 1 ≤ 1 − ?rho x2
    using output-sat unfolding satisfies-def unit-clause-def
    by (simp add: Neg eval-lit-def)
  ultimately show ?thesis by (simp add: Neg eval-lit-def)
```

qed
show *?thesis*
unfolding *in-M-def*
using *rho-circ-sat output-eval sv-rho cb-rho primed-cb-rho rho-01* **by** *auto*
qed

lemma *wf-circuit-out-reif*:
assumes *wf-circuit* (*C* :: *'v pb-circuit*)
shows *pvar-of-lit* (*snd C*) ∈ *circuit-reif-pvars C*
proof −
obtain *pairs out* **where** *C*: *C* = (*pairs*, *out*) **by** (*cases C*)
with *assms* **have** *Pos* (*pvar-of-lit out*) ∈ *fst ‘ set pairs* ∨ *Neg* (*pvar-of-lit out*) ∈ *fst ‘ set pairs*
unfolding *wf-circuit-def* **by** *auto*
then obtain *r φ* **where** (*r*, *φ*) ∈ *set pairs* **and** *r* = *Pos* (*pvar-of-lit out*) ∨ *r* = *Neg* (*pvar-of-lit out*)
by (*force simp*: *image-iff*)
then have *pvar-of-lit out* = *pvar-of-lit r*
by (*auto simp*: *pvar-of-lit-def*)
with ‹(*r*, *φ*) ∈ *set pairs*› **show** *?thesis*
unfolding *circuit-reif-pvars-def C*
by (*force simp*: *image-iff*)
qed

lemma *satisfies-cong*:
assumes ∀ *v* ∈ *constraint-pvars φ*. *rho1 v* = *rho2 v*
shows *satisfies φ rho1* = *satisfies φ rho2*
proof (*cases φ*)
case (*Pair coeffs A*)
have ∀ (*a*, *l*) ∈ *set coeffs*. *eval-lit l rho1* = *eval-lit l rho2*
proof (*safe*)
fix *a l* **assume** (*a*, *l*) ∈ *set coeffs*
then have *pvar-of-lit l* ∈ *constraint-pvars φ*
unfolding *Pair constraint-pvars-def* **by** *force*
with *assms* **have** *rho1* (*pvar-of-lit l*) = *rho2* (*pvar-of-lit l*) **by** *auto*
then show *eval-lit l rho1* = *eval-lit l rho2* **by** (*cases l*; *simp add*: *eval-lit-def pvar-of-lit-def*)
qed
then show *?thesis*
unfolding *Pair satisfies-def*
by (*auto simp*: *pb-sum-congr*)
qed

lemma *pvar-of-lit-lit-neg*: *pvar-of-lit* (*lit-neg l*) = *pvar-of-lit l*
by (*simp add*: *lit-neg-def pvar-of-lit-def split*: *literal.splits*)

lemma *wf-circuit-realizability*:
fixes *C* :: *'v pb-circuit*
assumes *wf*: *wf-circuit C*

```
    and distinct: distinct-reif-vars C
    and reif-not-input: ∀ v ∈ circuit-reif-pvars C. ¬ is-input-pvar v
  shows ∀ (base :: 'v pvar ⇒ nat). (∀ v. base v = 0 ∨ base v = 1) ⟶
    (∃ rho.
      models (circuit-constraints C) rho
      ∧ (∀ v. is-input-pvar v ⟶ rho v = base v)
      ∧ (∀ v. rho v = 0 ∨ rho v = 1))
proof (intro allI impI)
  fix base :: 'v pvar ⇒ nat
  assume base-01: ∀ v. base v = 0 ∨ base v = 1
  obtain pairs out where C: C = (pairs, out) by (cases C)
  have wf-conj: wf-circuit C ∧ distinct-reif-vars C using wf distinct ..
  have wf-deps: ∀ i < length pairs.
    pvar-of-lit ‘ snd ‘ set (fst (snd (pairs ! i)))
    ⊆ {x. is-input-pvar x} ∪ (pvar-of-lit ‘ fst ‘ set (take i pairs))
    using wf unfolding wf-circuit-def C Let-def by (auto simp: split-beta)
  have distinct': ∀ i < length pairs. ∀ j < length pairs. i ≠ j ⟶
    pvar-of-lit (fst (pairs ! i)) ≠ pvar-of-lit (fst (pairs ! j))
    using distinct unfolding distinct-reif-vars-def C Let-def by auto
  have reif-not-input':
    ∀ i < length pairs. ¬ is-input-pvar (pvar-of-lit (fst (pairs ! i)))
  proof (intro allI impI)
    fix i
    assume i < length pairs
    then have mem: pvar-of-lit (fst (pairs ! i)) ∈ circuit-reif-pvars C
      unfolding circuit-reif-pvars-def C
      by (metis fst-conv image-eqI nth-mem)
    then show ¬ is-input-pvar (pvar-of-lit (fst (pairs ! i)))
      using reif-not-input by meson
  qed
  have ∀ k ≤ length pairs. ∃ rho. (∀ v. rho v = 0 ∨ rho v = 1) ∧
    (∀ v. is-input-pvar v ⟶ rho v = base v) ∧
    (∀ j < k. ∀ φ ∈ reification (fst (pairs ! j)) (fst (snd (pairs ! j))) (snd (snd (pairs
! j))). satisfies φ rho)
  proof (intro allI impI)
    fix k
    assume k ≤ length pairs
    then show ∃ rho. (∀ v. rho v = 0 ∨ rho v = 1) ∧
      (∀ v. is-input-pvar v ⟶ rho v = base v) ∧
      (∀ j < k. ∀ φ ∈ reification (fst (pairs ! j)) (fst (snd (pairs ! j))) (snd (snd
(pairs ! j))). satisfies φ rho)
    proof (induction k)
      case 0
      let ?rho = λv. if is-input-pvar v then base v else 0
      have ∀ v. ?rho v = 0 ∨ ?rho v = 1 using base-01 by (simp split: if-splits)
      moreover have ∀ v. is-input-pvar v ⟶ ?rho v = base v by simp
      moreover have ∀ j < 0. ∀ φ ∈ reification (fst (pairs ! j)) (fst (snd (pairs !
j))) (snd (snd (pairs ! j))). satisfies φ ?rho by simp
      ultimately show ?case by auto
```

```
  next
    case (Suc k)
    then have k-lt: k < length pairs by auto
    from Suc.IH[OF Suc-leD[OF Suc.prems]] obtain rho where
      rho-01: ∀ v. rho v = 0 ∨ rho v = 1
      and rho-base: ∀ v. is-input-pvar v ⟶ rho v = base v
      and rho-sat: ∀ j < k. ∀ φ ∈ reification (fst (pairs ! j)) (fst (snd (pairs ! j)))
(snd (snd (pairs ! j))). satisfies φ rho
      by blast
    let ?r = fst (pairs ! k)
    let ?cs = fst (snd (pairs ! k))
    let ?A = snd (snd (pairs ! k))
    let ?v = pvar-of-lit ?r
    define val :: nat
      where val ≡ if pb-sum ?cs rho ≥ ?A then (case ?r of Pos - ⇒ 1 | Neg - ⇒
0) else (case ?r of Pos - ⇒ 0 | Neg - ⇒ 1)
    have val-01: val = 0 ∨ val = 1
      unfolding val-def by (cases ?r; auto)
    define rho' where rho' ≡ rho(?v := val)
    have rho'-01: ∀ v. rho' v = 0 ∨ rho' v = 1
      using rho-01 val-01 unfolding rho'-def by auto
    have rho'-base: ∀ v. is-input-pvar v ⟶ rho' v = base v
    proof (intro allI impI)
      fix v :: 'v pvar
      assume v-inp: is-input-pvar v
      show rho' v = base v
      proof (cases v = ?v)
        case True
        with reif-not-input' k-lt v-inp show ?thesis by auto
      next
        case False
        thus ?thesis using rho-base v-inp unfolding rho'-def by simp
      qed
    qed
    have eval-r: eval-lit ?r rho' = (if pb-sum ?cs rho ≥ ?A then 1 else 0)
       by (cases ?r; simp add: rho'-def eval-lit-def val-def pvar-of-lit-def split:
if-splits)
    have cs-no-v: ?v ∉ pvar-of-lit ‘ snd ‘ set ?cs
    proof
      assume ?v ∈ pvar-of-lit ‘ snd ‘ set ?cs
      then have ?v ∈ {x. is-input-pvar x} ∪ (pvar-of-lit ‘ fst ‘ set (take k pairs))
        using wf-deps k-lt by (auto simp: split-beta)
      then show False
      proof (elim UnE)
        assume ?v ∈ {x. is-input-pvar x}
        then have is-input-pvar ?v by simp
        with reif-not-input' k-lt show False by auto
      next
        assume ?v ∈ pvar-of-lit ‘ fst ‘ set (take k pairs)
```

```
      then obtain r' where r' ∈ fst ` set (take k pairs) pvar-of-lit r' = ?v
        by (auto simp: pvar-of-lit-def split: literal.splits)
      then obtain i where i < k pvar-of-lit (fst (pairs ! i)) = ?v
        by (auto simp: in-set-conv-nth)
      with distinct' k-lt show False by auto
    qed
  qed
  have rho'-cs-agree: pb-sum ?cs rho' = pb-sum ?cs rho
  proof (intro pb-sum-congr, safe)
    fix a l assume (a, l) ∈ set ?cs
    from cs-no-v this show eval-lit l rho' = eval-lit l rho
      by (force simp: rho'-def eval-lit-def pvar-of-lit-def fun-upd-def image-iff
split: literal.splits)
  qed
  have neg-cs-agree: ∀(a, l) ∈ set (map (λ(a, l). (a, lit-neg l)) ?cs). eval-lit l
rho' = eval-lit l rho
    unfolding set-map
  proof safe
    fix a l assume (a, l) ∈ set ?cs
    then show eval-lit (lit-neg l) rho' = eval-lit (lit-neg l) rho
      unfolding rho'-def eval-lit-def using cs-no-v
      by (cases l) (auto simp: lit-neg-def image-iff pvar-of-lit-def)
  qed
  have neg-sum-eq: pb-sum (map (λ(a, l). (a, lit-neg l)) ?cs) rho' = pb-sum
(map (λ(a, l). (a, lit-neg l)) ?cs) rho
    by (rule pb-sum-congr[OF neg-cs-agree])
  have pb-le-M: pb-sum ?cs rho ≤ sum-list (map fst ?cs)
    using pb-sum-add-negated-gen[OF rho-01, of ?cs] by simp
  have pb-sum-neg-eq: pb-sum (map (λ(a, l). (a, lit-neg l)) ?cs) rho = sum-list
(map fst ?cs) − pb-sum ?cs rho
  proof −
    have eq: sum-list (map fst ?cs) = pb-sum ?cs rho + pb-sum (map (λ(a, l).
(a, lit-neg l)) ?cs) rho
      using pb-sum-add-negated-gen[OF rho-01, of ?cs] by auto
    then show ?thesis by simp
  qed
  have sat-fwd: satisfies (reif-fwd ?r ?cs ?A) rho'
  proof −
    have eval-lit (lit-neg ?r) rho' = (if pb-sum ?cs rho ≥ ?A then 0 else 1)
    using eval-r rho'-01 by (force simp add: eval-lit-def lit-neg-def split: if-splits
literal.splits)
  then have pb-sum (fst (reif-fwd ?r ?cs ?A)) rho' = ?A ∗ (if pb-sum ?cs rho
≥ ?A then 0 else 1) + pb-sum ?cs rho'
      unfolding reif-fwd-def by simp
    also have ... = ?A ∗ (if pb-sum ?cs rho ≥ ?A then 0 else 1) + pb-sum ?cs
rho
      using rho'-cs-agree by auto
    also have ... ≥ ?A
    proof (cases pb-sum ?cs rho ≥ ?A)
```

```
      case True
      then show ?thesis using rho'-cs-agree by auto
    next
      case False
      then have ?A * 1 + pb-sum ?cs rho ≥ ?A by simp
      then show ?thesis using False by auto
    qed
    finally show ?thesis unfolding satisfies-def reif-fwd-def by auto
  qed
  have sat-bwd: satisfies (reif-bwd ?r ?cs ?A) rho'
  proof (cases pb-sum ?cs rho ≥ ?A)
    case True
    then have eval-lit ?r rho' = 1 using eval-r by simp
    then have snd (reif-bwd ?r ?cs ?A) = (sum-list (map fst ?cs) + 1 − ?A)
      unfolding reif-bwd-def Let-def by simp
    also have ... ≤ (sum-list (map fst ?cs) + 1 − ?A) * eval-lit ?r rho' +
pb-sum (map (λ(a, l). (a, lit-neg l)) ?cs) rho'
      using ‹eval-lit ?r rho' = 1› by simp
    finally show ?thesis unfolding satisfies-def reif-bwd-def Let-def by simp
  next
    case False
    then have pb-lt-A: pb-sum ?cs rho < ?A by simp
    have eval-r0: eval-lit ?r rho' = 0 using eval-r False by simp
    have snd (reif-bwd ?r ?cs ?A) = sum-list (map fst ?cs) + 1 − ?A
      unfolding reif-bwd-def Let-def by simp
    also have ... ≤ pb-sum (map (λ(a, l). (a, lit-neg l)) ?cs) rho'
    proof −
      have pb-sum (map (λ(a, l). (a, lit-neg l)) ?cs) rho' = pb-sum (map (λ(a,
l). (a, lit-neg l)) ?cs) rho
        using neg-sum-eq .
      also have ... = sum-list (map fst ?cs) − pb-sum ?cs rho
        using pb-sum-neg-eq .
      also have ... ≥ sum-list (map fst ?cs) + 1 − ?A
      proof −
        have pb-sum ?cs rho + 1 ≤ ?A using pb-lt-A by auto
        then show ?thesis
          using pb-le-M by linarith
      qed
      finally show ?thesis .
    qed
    finally show ?thesis
      unfolding satisfies-def reif-bwd-def Let-def using eval-r0 by auto
  qed
  have sat-k: ∀ φ ∈ reification ?r ?cs ?A. satisfies φ rho'
    using sat-fwd sat-bwd unfolding reification-def by auto
  have rho'-sat: ∀ j < Suc k. ∀ φ ∈ reification (fst (pairs ! j)) (fst (snd (pairs !
j))) (snd (snd (pairs ! j))). satisfies φ rho'
  proof (intro allI impI ballI)
    fix j φ
```

```
      assume j-lt: j < Suc k
       assume φ-reif: φ ∈ reification (fst (pairs ! j)) (fst (snd (pairs ! j))) (snd
(snd (pairs ! j)))
      show satisfies φ rho'
      proof (cases j < k)
        case True
        then have j-lt-k: j < k .
        let ?r-j = fst (pairs ! j)
        let ?cs-j = fst (snd (pairs ! j))
        have not-dep: ?v ∉ constraint-pvars φ
        proof
          assume dep: ?v ∈ constraint-pvars φ
            have constraint-pvars φ ⊆ {pvar-of-lit ?r-j} ∪ (pvar-of-lit ` snd ` set
?cs-j)
            using φ-reif unfolding reification-def
           by (fastforce simp: reif-fwd-def reif-bwd-def constraint-pvars-def Let-def
              lit-neg-def pvar-of-lit-def image-iff split: if-splits literal.splits)
          then have ?v = pvar-of-lit ?r-j ∨ ?v ∈ pvar-of-lit ` snd ` set ?cs-j
            using dep by auto
          then show False
          proof
            assume ?v = pvar-of-lit ?r-j
            with distinct' j-lt-k k-lt show False by auto
          next
            assume v-in: ?v ∈ pvar-of-lit ` snd ` set ?cs-j
             then have ?v ∈ {x. is-input-pvar x} ∪ (pvar-of-lit ` fst ` set (take j
pairs))
            proof −
                  have j-lt-len: j < length pairs using True k-lt by (simp add:
order.strict-trans)
              have pvar-of-lit ` snd ` set ?cs-j ⊆ {x. is-input-pvar x} ∪ (pvar-of-lit
` fst ` set (take j pairs))
                using wf-deps j-lt-len by simp
              then show ?thesis using v-in by blast
            qed
            then have ?v ∈ pvar-of-lit ` fst ` set (take j pairs)
            proof (elim UnE)
              assume ?v ∈ {x. is-input-pvar x}
              then have is-input-pvar ?v by simp
              with reif-not-input' k-lt show ?thesis by auto
            next
              assume ?v ∈ pvar-of-lit ` fst ` set (take j pairs)
              then show ?thesis .
            qed
            then obtain r' where r' ∈ fst ` set (take j pairs) pvar-of-lit r' = ?v
              by (auto simp: pvar-of-lit-def split: literal.splits)
            then obtain i where i < j pvar-of-lit (fst (pairs ! i)) = ?v
              by (auto simp: in-set-conv-nth)
            with distinct' j-lt-k k-lt show False by auto
```

```
        qed
      qed
      have satisfies φ rho
        using rho-sat j-lt-k φ-reif by blast
      moreover have ∀ v' ∈ constraint-pvars φ. rho v' = rho' v'
        using not-dep unfolding rho'-def by auto
      ultimately show ?thesis
        using satisfies-cong by metis
    next
      case False
      then have j = k using j-lt by auto
      then show ?thesis using sat-k φ-reif by simp
    qed
  qed
  thus ?case using rho'-01 rho'-base rho'-sat by blast
 qed
qed
then obtain rho where
  rho-01: ∀ v. rho v = 0 ∨ rho v = 1
  and rho-base: ∀ v. is-input-pvar v ⟶ rho v = base v
  and rho-sat: ∀ j < length pairs. ∀ φ ∈ reification (fst (pairs ! j)) (fst (snd (pairs
! j))) (snd (snd (pairs ! j))). satisfies φ rho
  by auto
have models (circuit-constraints C) rho
  unfolding models-def circuit-constraints-def C fst-conv
proof (intro ballI)
  fix φ
  assume φ ∈ (⋃(r, x, y)∈set pairs. reification r x y)
  then obtain x where x-in: x ∈ set pairs and x-reif: φ ∈ reification (fst x)
(fst (snd x)) (snd (snd x)) by auto
  then obtain j where j < length pairs pairs ! j = x
    by (auto simp: in-set-conv-nth)
  then show satisfies φ rho using rho-sat x-reif by auto
qed
then show ∃ rho. models (circuit-constraints C) rho
  ∧ (∀ v. is-input-pvar v ⟶ rho v = base v)
  ∧ (∀ v. rho v = 0 ∨ rho v = 1)
  using rho-01 rho-base by blast
qed

lemma models-circuit-constraints-cong:
 assumes models (circuit-constraints C) rho1
  and ∀ φ ∈ circuit-constraints C. ∀ v ∈ constraint-pvars φ. rho1 v = rho2 v
 shows models (circuit-constraints C) rho2
 unfolding models-def
proof
 fix φ assume φ ∈ circuit-constraints C
 then have satisfies φ rho1 using assms(1) unfolding models-def by auto
 moreover have satisfies φ rho1 = satisfies φ rho2
```

```
    by (rule satisfies-cong) (insert assms(2) ‹φ ∈ circuit-constraints C›, auto)
  ultimately show satisfies φ rho2 by simp
qed

lemma pb-sum-cost-bits-gen:
  assumes ∀ i < k. rho (CostBit i) = (c div 2^i) mod 2
  shows pb-sum (map (λi. (2^i, Pos (CostBit i))) [0..<k]) rho = c mod 2^k
  using assms
proof (induction k)
  case 0 show ?case by simp
next
  case (Suc k)
  have IH: pb-sum (map (λi. (2^i, Pos (CostBit i))) [0..<k]) rho = c mod 2^k
    using Suc.IH Suc.prems by auto
  have step: rho (CostBit k) = c div 2^k mod 2
    using Suc.prems by simp
  have pb-sum (map (λi. (2^i, Pos (CostBit i))) [0..<Suc k]) rho
      = pb-sum (map (λi. (2^i, Pos (CostBit i))) [0..<k]) rho
        + pb-sum [(2^k, Pos (CostBit k))] rho
    by (simp add: pb-sum-append)
  also have ... = c mod 2^k + 2^k * ((c div 2^k) mod 2)
    using IH step by (simp add: eval-lit-def)
  also have ... = c mod 2^(Suc k)
    by (metis add.commute mod-mult2-eq mult.commute power-Suc)
  finally show ?case by simp
qed

lemma cost-ge-constraint-sound':
  assumes satisfies (cost-ge-constraint B) rho
    and ∀ i < bits-needed B. rho (CostBit i) = (c div 2^i) mod 2
  shows c ≥ B
proof −
  have sum-eq: pb-sum (map (λi. (2^i, Pos (CostBit i))) [0..<bits-needed B]) rho
= c mod 2^(bits-needed B)
    by (rule pb-sum-cost-bits-gen) (use assms(2) in auto)
  have B ≤ c mod 2^(bits-needed B)
  proof −
   from assms(1) have B ≤ pb-sum (map (λi. (2^i, Pos (CostBit i))) [0..<bits-needed
B]) rho
      unfolding satisfies-def cost-ge-constraint-def by simp
    with sum-eq show ?thesis by linarith
  qed
  then show ?thesis by (meson mod-less-eq-dividend order-trans)
qed

lemma in-M-goal-bound:
  fixes Π :: 'v::linorder strips-task
  assumes in-M C-φ Π s c
    and wf-circuit C-φ
```

```
    and circ-vars: ∀(r, φ) ∈ set (fst C-φ). ∀v ∈ constraint-pvars φ. is-input-pvar
v ∨ v ∈ circuit-reif-pvars C-φ
    and disjoint: ∀v ∈ circuit-reif-pvars C-φ. ¬ is-input-pvar v ∧ v ≠ ReifI ∧ (∀k.
v ≠ ReifCostGe k)
    and reifG-not-circ: ReifG ∉ circuit-reif-pvars C-φ
      and goal-cpr: cpr-derives (encode-goal Π ∪ circuit-constraints C-φ ∪ en-
code-cost-ge B ∪
                    {unit-clause (snd C-φ), unit-clause (Pos ReifG)})
                    (cost-ge-constraint B)
    and is-goal-state Π s
    and finite (vars Π)
    and goal Π ⊆ vars Π
  shows c ≥ B
proof (cases c < 2^(bits-needed B))
  case True
  from assms(1) obtain rho-raw where
    rho-raw-circ: models (circuit-constraints C-φ) rho-raw
    and rho-raw-out: eval-lit (snd C-φ) rho-raw = 1
    and rho-raw-sv: ∀v. rho-raw (StateVar v) = (if v ∈ s then 1 else 0)
    and rho-raw-cb: ∀i. rho-raw (CostBit i) = (c div 2^i) mod 2
    and rho-raw-pb: ∀i. rho-raw (PrimedCostBit i) = 0
    and rho-raw-01: ∀v. rho-raw v = 0 ∨ rho-raw v = 1
    unfolding in-M-def by blast
  have out-reif: pvar-of-lit (snd C-φ) ∈ circuit-reif-pvars C-φ
    by (rule wf-circuit-out-reif[OF assms(2)])
  define base where base = state-rho Π s c
  define rho-adj where rho-adj v ≡ if is-input-pvar v then base v else rho-raw v
for v
  have rho-adj-agrees: ∀ φ ∈ circuit-constraints C-φ. ∀ v ∈ constraint-pvars φ.
rho-raw v = rho-adj v
  proof (intro ballI allI impI)
    fix φ v assume φ ∈ circuit-constraints C-φ and v ∈ constraint-pvars φ
    show rho-raw v = rho-adj v
    proof (cases is-input-pvar v)
      case True
      have rho-adj v = base v by (simp add: rho-adj-def True)
      also have base v = rho-raw v
        unfolding base-def state-rho-def
        using True rho-raw-sv rho-raw-cb rho-raw-pb
        by (cases v; auto simp: is-input-pvar-def state-rho-def split: var.splits)
      finally show ?thesis by simp
    next
      case False
      then show ?thesis by (simp add: rho-adj-def)
    qed
  qed
  have rho-circ: models (circuit-constraints C-φ) rho-adj
    by (rule models-circuit-constraints-cong[OF rho-raw-circ rho-adj-agrees])
  have rho-out: eval-lit (snd C-φ) rho-adj = 1
```

proof −
have *eval-lit* (*snd C-φ*) *rho-adj* = *eval-lit* (*snd C-φ*) *rho-raw*
using *out-reif disjoint*
by (*cases snd C-φ*) (*simp-all add*: *eval-lit-def rho-adj-def pvar-of-lit-def*)
also have ... = *1* **by** (*rule rho-raw-out*)
finally show *?thesis* **.**
qed
define *rho-ext* **where** *rho-ext* = *extend-rho C-φ rho-adj base*
have *base-on-input*: ∀ (*v* :: *'v pvar*). *is-input-pvar v* ⟶ *base v* = *rho-adj v*
by (*auto simp*: *rho-adj-def*)
have *rho-ext-circ*: *models* (*circuit-constraints C-φ*) *rho-ext*
unfolding *rho-ext-def*
by (*rule models-circuit-constraints-extend*[*OF rho-circ circ-vars base-on-input*])
have *rho-ext-out*: *eval-lit* (*snd C-φ*) *rho-ext* = *1*
using *eval-output-extend-rho*[*OF out-reif*] *rho-out* **by** (*simp add*: *rho-ext-def*)
have *cost-constraints-not-circ*:
∀ *φ* ∈ *encode-cost-ge B*. ∀ *v* ∈ *constraint-pvars φ*. *v* ∉ *circuit-reif-pvars C-φ*
proof (*intro ballI allI impI*)
fix *φ* **and** *v* :: *'v pvar*
assume *φ* ∈ *encode-cost-ge B* **and** *v* ∈ *constraint-pvars φ*
then have *φ* = *reif-fwd* (*Pos* (*ReifCostGe B*)) (*map* (λ*i*. (*2^i*, *Pos* (*CostBit i*))) [*0..<bits-needed B*]) *B* ∨
φ = *reif-bwd* (*Pos* (*ReifCostGe B*)) (*map* (λ*i*. (*2^i*, *Pos* (*CostBit i*))) [*0..<bits-needed B*]) *B*
unfolding *encode-cost-ge-def reification-def* **by** *auto*
then show *v* ∉ *circuit-reif-pvars C-φ*
proof (*elim disjE*)
assume *φ* = *reif-fwd* (*Pos* (*ReifCostGe B*)) (*map* (λ*i*. (*2^i*, *Pos* (*CostBit i*))) [*0..<bits-needed B*]) *B*
then have *v* = *ReifCostGe B* ∨ (∃ *i* < *bits-needed B*. *v* = *CostBit i*)
using ‹*v* ∈ *constraint-pvars φ*›
unfolding *constraint-pvars-def reif-fwd-def* **by** (*auto simp*: *pvar-of-lit-def lit-neg-def*)
with *disjoint* **show** *?thesis* **by** (*auto simp*: *is-input-pvar-def*)
next
assume *φ* = *reif-bwd* (*Pos* (*ReifCostGe B*)) (*map* (λ*i*. (*2^i*, *Pos* (*CostBit i*))) [*0..<bits-needed B*]) *B*
then have *v* = *ReifCostGe B* ∨ (∃ *i* < *bits-needed B*. *v* = *CostBit i*)
using ‹*v* ∈ *constraint-pvars φ*› **unfolding** *constraint-pvars-def reif-bwd-def*
by (*clarsimp simp*: *pvar-of-lit-def Let-def lit-neg-def split*: *literal.splits*; *arith*)
with *disjoint* **show** *?thesis* **by** (*auto simp*: *is-input-pvar-def*)
qed
qed
have *goal-constraints-not-circ*:
∀ *φ* ∈ *encode-goal* Π. ∀ *v* ∈ *constraint-pvars φ*. *v* ∉ *circuit-reif-pvars C-φ*
proof (*intro ballI allI impI*)
fix *φ* **and** *v* :: *'v pvar*
assume *φ* ∈ *encode-goal* Π **and** *v* ∈ *constraint-pvars φ*

```
    then have φ = reif-fwd (Pos ReifG) (state-lits (goal Π)) (card (goal Π)) ∨
            φ = reif-bwd (Pos ReifG) (state-lits (goal Π)) (card (goal Π))
      unfolding encode-goal-def reification-def
      by (auto simp: Let-def)
    then show v ∉ circuit-reif-pvars C-φ
    proof (elim disjE)
      assume φ = reif-fwd (Pos ReifG) (state-lits (goal Π)) (card (goal Π))
      then have v = ReifG ∨ (∃ v' ∈ goal Π. v = StateVar v')
        using ‹v ∈ constraint-pvars φ›
        unfolding constraint-pvars-def reif-fwd-def state-lits-def
       by (auto simp: pvar-of-lit-def lit-neg-def finite-subset[OF assms(9) assms(8)]
split: literal.splits)
      with disjoint reifG-not-circ show ?thesis by (auto simp: is-input-pvar-def)
    next
      assume φ = reif-bwd (Pos ReifG) (state-lits (goal Π)) (card (goal Π))
      then have v = ReifG ∨ (∃ v' ∈ goal Π. v = StateVar v')
        using ‹v ∈ constraint-pvars φ›
        unfolding constraint-pvars-def reif-bwd-def state-lits-def
        by (auto simp: pvar-of-lit-def lit-neg-def Let-def finite-subset[OF assms(9)
assms(8)] split: literal.splits)
      with disjoint reifG-not-circ show ?thesis by (auto simp: is-input-pvar-def)
    qed
  qed
  have enc-cost-sat: models (encode-cost-ge B) base
  proof (cases c ≥ B)
    case True
    from cost-ge-encoding-sound[OF True ‹c < 2^(bits-needed B)›]
    show ?thesis unfolding base-def .
  next
    case False
    then have c < B by (simp add: not-le)
    from cost-ge-encoding-below[OF this]
    show ?thesis unfolding base-def .
  qed
  have goal-sat: models (encode-goal Π) base
    unfolding base-def
    by (rule goal-encoding-sound[OF assms(8) assms(9) assms(7)])
  have lit-axiom-sat: satisfies (unit-clause (Pos ReifG)) rho-ext
  proof −
    have rho-eq: rho-ext ReifG = 1
      by (simp add: rho-ext-def extend-rho-def base-def state-rho-def
                    is-input-pvar-def assms(7) reifG-not-circ split: if-splits)
    then show ?thesis
      unfolding satisfies-def unit-clause-def
      by (simp add: eval-lit-def)
  qed
  have hyp-model:
    ∀ ψ ∈ encode-goal Π ∪ circuit-constraints C-φ ∪ encode-cost-ge B ∪
            {unit-clause (snd C-φ), unit-clause (Pos ReifG)}. satisfies ψ rho-ext
```

proof (*intro ballI*)
fix ψ
assume $\psi \in$ *encode-goal* $\Pi \cup$ *circuit-constraints C-*$\varphi \cup$ *encode-cost-ge B* $\cup$
$\{$*unit-clause* (*snd C-*φ), *unit-clause* (*Pos ReifG*)$\}$
then consider (*goal*) $\psi \in$ *encode-goal* Π
| (*circ*) $\psi \in$ *circuit-constraints C-*φ
| (*cost*) $\psi \in$ *encode-cost-ge B*
| (*ax1*) $\psi =$ *unit-clause* (*snd C-*φ)
| (*ax2*) $\psi =$ *unit-clause* (*Pos ReifG*)
by *auto*
then show *satisfies* ψ *rho-ext*
proof *cases*
case *goal*
then show *?thesis*
using *goal-sat goal-constraints-not-circ*
unfolding *rho-ext-def models-def*
by (*auto simp*: *satisfies-extend-rho-eq-base*)
next
case *circ*
then show *?thesis*
using *rho-ext-circ* **by** (*auto simp*: *models-def*)
next
case *cost*
then show *?thesis*
using *enc-cost-sat cost-constraints-not-circ*
unfolding *rho-ext-def models-def*
by (*auto simp*: *satisfies-extend-rho-eq-base*)
next
case *ax1*
then show *?thesis*
using *rho-ext-out* **by** (*simp add*: *satisfies-def unit-clause-def*)
next
case *ax2*
then show *?thesis*
by (*simp add*: *lit-axiom-sat*)
qed
qed
have *rho-adj-01*: $\forall\ v.$ *rho-adj* $v = 0 \vee$ *rho-adj* $v = 1$
unfolding *rho-adj-def base-def state-rho-def*
using *rho-raw-01* **by** (*auto simp add*: *is-input-pvar-def split*: *pvar.splits*)
then have *rho-ext-01*: $\forall\ v.$ *rho-ext* $v = 0 \vee$ *rho-ext* $v = 1$
unfolding *rho-ext-def extend-rho-def base-def* **using** *state-rho-01* **by** *metis*
have *bound-sat*: *satisfies* (*cost-ge-constraint B*) *rho-ext*
using *cpr-sound goal-cpr hyp-model models-def rho-ext-01* **by** *blast*
then show *?thesis*
proof (*rule cost-ge-constraint-sound'*)
show $\forall\, i <$ *bits-needed B. rho-ext* (*CostBit i*) = (*c div* $2 \mathbin{\hat{}} i$) *mod 2*
using *rho-raw-cb disjoint base-on-input*
by (*auto simp*: *rho-ext-def extend-rho-def rho-adj-def base-def state-rho-def*

is-input-pvar-def)
qed
next
case *False*
then have $c \geq 2\hat{}(\textit{bits-needed } B)$ **by** *simp*
then show *?thesis* **by** (*meson bits-needed-sufficient order.strict-implies-order order-less-le-trans*)
qed

2.12 Transition Step Soundness and CPR Theorem

definition *two-state-rho* ::
$'v$::*linorder strips-task* $\Rightarrow$ $'v$ *state* $\Rightarrow$ *nat* $\Rightarrow$ $'v$ *state* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ ($'v$ *action*) *list* $\Rightarrow$
($'v$ *var pvar* $\Rightarrow$ *nat*) **where**
two-state-rho Π *s c s' c' sel-i as* $\equiv$ $\lambda x.$ *case x of*
StateVar (*Original v*) $\Rightarrow$ *if v* $\in$ *s then 1 else 0*
| *StateVar* (*Primed v*) $\Rightarrow$ *if v* $\in$ *s' then 1 else 0*
| *CostBit i* $\Rightarrow$ (*c div 2*$\hat{}$*i*) *mod 2*
| *PrimedCostBit i* $\Rightarrow$ (*c' div 2*$\hat{}$*i*) *mod 2*
| *ReifT* $\Rightarrow$ *1*
| *ReifDeltaCostLower k* $\Rightarrow$ *if c'* $\geq$ *c* + *k then 1 else 0*
| *ReifDeltaCostUpper k* $\Rightarrow$ *if c'* $\leq$ *c* + *k then 1 else 0*
| *ReifDeltaCost k* $\Rightarrow$ *if k* = *c'* − *c then 1 else 0*
| *ReifPrimedCostGe k* $\Rightarrow$ *if c'* $\geq$ *k then 1 else 0*
| *ReifGeq* (*Original v*) $\Rightarrow$ *if v* $\in$ *s* $\vee$ *v* $\notin$ *s' then 1 else 0*
| *ReifLeq* (*Original v*) $\Rightarrow$ *if v* $\notin$ *s* $\vee$ *v* $\in$ *s' then 1 else 0*
| *ReifEq* (*Original v*) $\Rightarrow$ *if* (*v* $\in$ *s* $\longleftrightarrow$ *v* $\in$ *s'*) *then 1 else 0*
| *ReifAction i* $\Rightarrow$ *if i* = *sel-i then 1 else 0*
| - $\Rightarrow$ *0*

lemma *two-state-rho-range*:
$\forall$ *v. two-state-rho* Π *s c s' c' sel-i as v* = *0* $\vee$ *two-state-rho* Π *s c s' c' sel-i as v* = *1*
by (*auto simp*: *two-state-rho-def split*: *pvar.splits var.splits*)

lemma *pb-sum-pos-pvar-aux*:
assumes *distinct xs*
and $\forall$ *v* $\in$ *set xs. rho* (*g v*) = (*if v* $\in$ *T then 1 else 0*)
shows *pb-sum* (*map* (λ*v.* (*1*, *Pos* (*g v*))) *xs*) *rho* = *card* (*set xs* $\cap$ *T*)
using *assms*
proof (*induction xs*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons a xs*)
then have *a-not-xs*: *a* $\notin$ *set xs* **and** *distinct xs*
and *vals*: $\forall$ *v* $\in$ *set* (*a* # *xs*). *rho* (*g v*) = (*if v* $\in$ *T then 1 else 0*) **by** *auto*
have *step*: *pb-sum* (*map* (λ*v.* (*1*, *Pos* (*g v*))) (*a* # *xs*)) *rho*
= (*if a* $\in$ *T then 1 else 0*) + *pb-sum* (*map* (λ*v.* (*1*, *Pos* (*g v*))) *xs*) *rho*

```
    using vals by (simp add: eval-lit-def)
  have IH: pb-sum (map (λv. (1, Pos (g v))) xs) rho = card (set xs ∩ T)
    by (rule Cons.IH) (use Cons.prems in auto)
  have card-step: (if a ∈ T then 1 else 0) + card (set xs ∩ T) = card (set (a #
xs) ∩ T)
    using a-not-xs by (auto simp: card-insert-if)
  show ?case using step IH card-step by linarith
qed

lemma pb-sum-neg-pvar-aux:
  assumes distinct xs
    and ∀ v ∈ set xs. rho (g v) = (if v ∈ T then 1 else 0)
  shows pb-sum (map (λv. (1, Neg (g v))) xs) rho = card (set xs − T)
  using assms
proof (induction xs)
  case Nil show ?case by simp
next
  case (Cons a xs)
  then have a-not-xs: a ∉ set xs and distinct xs
    and vals: ∀ v ∈ set (a # xs). rho (g v) = (if v ∈ T then 1 else 0) by auto
  have step: pb-sum (map (λv. (1, Neg (g v))) (a # xs)) rho
    = (if a ∉ T then 1 else 0) + pb-sum (map (λv. (1, Neg (g v))) xs) rho
    using vals by (simp add: eval-lit-def)
  have IH: pb-sum (map (λv. (1, Neg (g v))) xs) rho = card (set xs − T)
    by (rule Cons.IH) (use Cons.prems in auto)
  have card-step: (if a ∉ T then 1 else 0) + card (set xs − T) = card (set (a #
xs) − T)
    using a-not-xs by (auto simp: insert-Diff-if)
  show ?case using step IH card-step by linarith
qed

lemma pb-sum-pos-pvar-sorted:
  assumes fin: finite S
    and vals: ∀ v ∈ S. rho (g v) = (if v ∈ T then 1 else 0)
  shows pb-sum (map (λv. (1, Pos (g v))) (sorted-list-of-set S)) rho = card (S ∩
T)
proof −
  have pb-sum (map (λv. (1, Pos (g v))) (sorted-list-of-set S)) rho
      = card (set (sorted-list-of-set S) ∩ T)
    by (rule pb-sum-pos-pvar-aux) (use fin vals in simp-all)
  also have set (sorted-list-of-set S) = S using fin by simp
  finally show ?thesis .
qed

lemma pb-sum-neg-pvar-sorted:
  assumes fin: finite S
    and vals: ∀ v ∈ S. rho (g v) = (if v ∈ T then 1 else 0)
  shows pb-sum (map (λv. (1, Neg (g v))) (sorted-list-of-set S)) rho = card (S −
T)
```

```
proof −
  have pb-sum (map (λv. (1, Neg (g v))) (sorted-list-of-set S)) rho
      = card (set (sorted-list-of-set S) − T)
    by (rule pb-sum-neg-pvar-aux) (use fin vals in simp-all)
  also have set (sorted-list-of-set S) = S using fin by simp
  finally show ?thesis .
qed

lemma pb-sum-primed-cost-bits-gen:
  assumes ∀ i < k. rho (PrimedCostBit i) = (c div 2^i) mod 2
  shows pb-sum (map (λi. (2^i, Pos (PrimedCostBit i))) [0..<k]) rho = c mod
2^k
  using assms
proof (induction k)
  case 0 show ?case by simp
next
  case (Suc k)
  have IH: pb-sum (map (λi. (2^i, Pos (PrimedCostBit i))) [0..<k]) rho = c mod
2^k
    using Suc.IH Suc.prems by auto
  have step: rho (PrimedCostBit k) = c div 2^k mod 2
    using Suc.prems by simp
  have pb-sum (map (λi. (2^i, Pos (PrimedCostBit i))) [0..<Suc k]) rho
      = pb-sum (map (λi. (2^i, Pos (PrimedCostBit i))) [0..<k]) rho
        + pb-sum [(2^k, Pos (PrimedCostBit k))] rho
    by (simp add: pb-sum-append)
  also have ... = c mod 2^k + 2^k * ((c div 2^k) mod 2)
    using IH step by (simp add: eval-lit-def)
  also have ... = c mod 2^(Suc k)
    by (metis add.commute mod-mult2-eq mult.commute power-Suc)
  finally show ?case by simp
qed

lemma encode-transition-sound:
  fixes Π :: 'v::linorder strips-task
  assumes fin-V: finite V
    and sel-lt: sel-i < length as
    and appl: applicable (as ! sel-i) s
    and succ-eq: s' = successor (as ! sel-i) s
    and cost-eq: c' = c + cost (as ! sel-i)
    and c'-lt-B: c' < B
    and pre-sub: pre (as ! sel-i) ⊆ V
    and add-sub: add (as ! sel-i) ⊆ V
    and del-sub: del (as ! sel-i) ⊆ V
  shows models (encode-transition as V B)
               (two-state-rho Π s c s' c' sel-i as)
proof −
  have c'-bound: c' < 2^(bits-needed B)
  proof −
```

have $c' < B$ **using** *c'-lt-B* **.**
also have $B < 2\hat{}(\textit{bits-needed}\ B)$ **by** (*rule bits-needed-sufficient*)
finally show *?thesis* **.**
qed
let *?rho* = *two-state-rho* Π *s c s' c' sel-i as*
let *?rs* = *action-reifs as*
have *rho-StateVar*: $\forall v.$ *?rho* (*StateVar* (*Original v*)) = (*if v* $\in$ *s then 1 else 0*)
by (*simp add*: *two-state-rho-def*)
have *rho-CostBit*: $\forall i.$ *?rho* (*CostBit i*) = (*c div* $2\hat{}i$) *mod 2*
by (*simp add*: *two-state-rho-def*)
have *rho-PrimedStateVar*: $\forall v.$ *?rho* (*StateVar* (*Primed v*)) = (*if v* $\in$ *s' then 1 else 0*)
by (*simp add*: *two-state-rho-def*)
have *rho-PrimedCostBit*: $\forall i.$ *?rho* (*PrimedCostBit i*) = (*c' div* $2\hat{}i$) *mod 2*
by (*simp add*: *two-state-rho-def*)
have *rho-ReifT*: *?rho ReifT* = *1*
by (*simp add*: *two-state-rho-def*)
have *rho-ReifDeltaCost*: $\forall k.$ *?rho* (*ReifDeltaCost k*) = (*if k* = *c'* − *c then 1 else 0*)
by (*simp add*: *two-state-rho-def*)
have *rho-ReifPrimedCostGe*: $\forall k.$ *?rho* (*ReifPrimedCostGe k*) = (*if c'* $\geq$ *k then 1 else 0*)
by (*simp add*: *two-state-rho-def*)
have *rho-ReifGeq*: $\forall v.$ *?rho* (*ReifGeq* (*Original v*)) = (*if v* $\in$ *s* $\vee$ *v* $\notin$ *s' then 1 else 0*)
by (*simp add*: *two-state-rho-def*)
have *rho-ReifLeq*: $\forall v.$ *?rho* (*ReifLeq* (*Original v*)) = (*if v* $\notin$ *s* $\vee$ *v* $\in$ *s' then 1 else 0*)
by (*simp add*: *two-state-rho-def*)
have *rho-ReifEq*: $\forall v.$ *?rho* (*ReifEq* (*Original v*)) = (*if* (*v* $\in$ *s* $\longleftrightarrow$ *v* $\in$ *s'*) *then 1 else 0*)
by (*simp add*: *two-state-rho-def*)
have *rho-ReifAction*: $\forall i.$ *?rho* (*ReifAction i*) = (*if i* = *sel-i then 1 else 0*)
by (*simp add*: *two-state-rho-def*)

have *models-action-cs*:
models ($\bigcup$ *i*<*length as*. {*action-constraint* (*?rs*!*i*) (*as*!*i*) *V B*}) *?rho*
proof (*unfold models-def*, *intro ballI*)
fix φ
assume φ*-in*: $\varphi \in (\bigcup$ *i*<*length as*. {*action-constraint* (*?rs*!*i*) (*as*!*i*) *V B*})
then obtain *i* **where** *i-lt*: *i* < *length as*
and φ*-eq*: φ = *action-constraint* (*?rs*!*i*) (*as*!*i*) *V B*
by *auto*
show *satisfies* φ *?rho*
proof (*cases i* = *sel-i*)
case *True*
have *r-eq*: *?rs* ! *i* = *Pos* (*ReifAction i*)
unfolding *action-reifs-def* **using** *i-lt* **by** *simp*
have *a-eq*: *as*!*i* = *as*!*sel-i* **using** *True* **by** *simp*

have *fin-pre*: *finite* (*pre* (*as*!*i*))
using *fin-V pre-sub i-lt True* **by** (*blast intro*: *finite-subset*)
have *fin-add*: *finite* (*add* (*as*!*i*))
using *fin-V add-sub i-lt True* **by** (*blast intro*: *finite-subset*)
have *fin-del*: *finite* (*del* (*as*!*i*))
using *fin-V del-sub i-lt True* **by** (*blast intro*: *finite-subset*)
have *fin-Vevars*: *finite* (*V* − *evars* (*as*!*i*)) **using** *fin-V* **by** *simp*
have *pre-sub-s*: *pre* (*as*!*i*) ⊆ *s*
using *appl True i-lt* **unfolding** *applicable-def* **by** *auto*
have *add-sub-s′*: *add* (*as*!*i*) ⊆ *s′*
using *succ-eq a-eq* **by** (*simp add*: *successor-def*)
have *delta-val*: *pb-sum* [(*1*, *Pos* (*ReifDeltaCost* (*cost* (*as*!*i*))))] *?rho* = *1*
using *cost-eq rho-ReifDeltaCost True* **by** (*simp add*: *eval-lit-def*)
have *pre-val*: *pb-sum* (*map* (λ*v*. (*1*, *Pos* (*StateVar* (*Original v*)))) (*sorted-list-of-set* (*pre* (*as*!*i*)))) *?rho*
= *card* (*pre* (*as*!*i*))
proof −
have *pb-sum* (*map* (λ*v*. (*1*, *Pos* (*StateVar* (*Original v*)))) (*sorted-list-of-set* (*pre* (*as*!*i*)))) *?rho*
= *card* (*pre* (*as*!*i*) ∩ *s*)
by (*rule pb-sum-pos-pvar-sorted*[*OF fin-pre*, **where** *g*=λ*v*. *StateVar* (*Original v*) **and** *T*=*s*])
(*simp add*: *rho-StateVar*)
also have *pre* (*as*!*i*) ∩ *s* = *pre* (*as*!*i*) **using** *pre-sub-s* **by** *auto*
finally show *?thesis* **by** *simp*
qed
have *add-val*: *pb-sum* (*map* (λ*v*. (*1*, *Pos* (*StateVar* (*Primed v*)))) (*sorted-list-of-set* (*add* (*as*!*i*)))) *?rho*
= *card* (*add* (*as*!*i*))
proof −
have *pb-sum* (*map* (λ*v*. (*1*, *Pos* (*StateVar* (*Primed v*)))) (*sorted-list-of-set* (*add* (*as*!*i*)))) *?rho*
= *card* (*add* (*as*!*i*) ∩ *s′*)
by (*rule pb-sum-pos-pvar-sorted*[*OF fin-add*, **where** *g*=λ*v*. *StateVar* (*Primed v*) **and** *T*=*s′*])
(*simp add*: *rho-PrimedStateVar*)
also have *add* (*as*!*i*) ∩ *s′* = *add* (*as*!*i*) **using** *add-sub-s′* **by** *auto*
finally show *?thesis* **by** *simp*
qed
have *del-val*: *pb-sum* (*map* (λ*v*. (*1*, *Neg* (*StateVar* (*Primed v*)))) (*sorted-list-of-set* (*del* (*as*!*i*)))) *?rho*
= *card* (*del* (*as*!*i*) − *s′*)
by (*rule pb-sum-neg-pvar-sorted*[*OF fin-del*, **where** *g*=λ*v*. *StateVar* (*Primed v*) **and** *T*=*s′*])
(*simp add*: *rho-PrimedStateVar*)
have *not-evars-unchanged*: ∀ *v* ∈ *V* − *evars* (*as*!*i*). (*v* ∈ *s* ⟷ *v* ∈ *s′*)
unfolding *succ-eq successor-def evars-def* **using** *True a-eq* **by** *auto*
have *frame-val*:
pb-sum (*map* (λ*v*. (*1*, *Pos* (*ReifEq* (*Original v*)))) (*sorted-list-of-set* (*V* −

```
evars (as!i)))) ?rho
        = card (V − evars (as!i))
      proof −
        have pb-sum (map (λv. (1, Pos (ReifEq (Original v)))) (sorted-list-of-set
(V − evars (as!i)))) ?rho
          = card ((V − evars (as!i)) ∩ (V − evars (as!i)))
            by (rule pb-sum-pos-pvar-sorted[OF fin-Vevars, where g=λv. ReifEq
(Original v) and T=V − evars (as!i)])
            (auto simp: rho-ReifEq not-evars-unchanged)
        also have (V − evars (as!i)) ∩ (V − evars (as!i)) = V − evars (as!i) by
auto
        finally show ?thesis by simp
      qed
      have bound-val: pb-sum [(1, Neg (ReifPrimedCostGe B))] ?rho = 1
        using c′-lt-B rho-ReifPrimedCostGe by (simp add: eval-lit-def)
      have evars-sub-V: evars (as!i) ⊆ V
        unfolding evars-def using add-sub del-sub i-lt True by auto
      have card-add-del-frame:
       card (add (as!i)) + card (del (as!i) − s′) + card (V − evars (as!i)) = card
V
      proof −
        have disj-add-del: add (as!i) ∩ (del (as!i) − add (as!i)) = {} by auto
        have union-add-del: add (as!i) ∪ (del (as!i) − add (as!i)) = evars (as!i)
          unfolding evars-def by auto
        have card-add-del-minus:
          card (add (as!i)) + card (del (as!i) − add (as!i)) = card (evars (as!i))
          using card-Un-disjoint[OF fin-add finite-Diff[OF fin-del] disj-add-del]
          unfolding union-add-del by simp
        have del-s′-eq-del-minus-add: del (as!i) − s′ = del (as!i) − add (as!i)
          unfolding succ-eq successor-def using True a-eq by auto
        have card-V-part: card V = card (V − evars (as!i)) + card (evars (as!i))
          by (metis add.commute card-Int-Diff evars-sub-V fin-V inf.absorb-iff2)
        show ?thesis
          using card-add-del-minus del-s′-eq-del-minus-add card-V-part by simp
      qed
      have A-val: snd φ = 2 + card (pre (as!i)) + card V
        unfolding φ-eq action-constraint-def Let-def by simp
      have lhs-eq: fst φ = [(2 + card (pre (as!i)) + card V, lit-neg (?rs!i))] @
        [(1, Pos (ReifDeltaCost (cost (as!i))))] @
        map (λv. (1, Pos (StateVar (Original v)))) (sorted-list-of-set (pre (as!i)))
@
         map (λv. (1, Pos (StateVar (Primed v)))) (sorted-list-of-set (add (as!i)))
@
        map (λv. (1, Neg (StateVar (Primed v)))) (sorted-list-of-set (del (as!i))) @
          map (λv. (1, Pos (ReifEq (Original v)))) (sorted-list-of-set (V − evars
(as!i))) @
        [(1, Neg (ReifPrimedCostGe B))]
        unfolding φ-eq action-constraint-def Let-def by simp
      have lit-neg-pb: pb-sum [(2 + card (pre (as!i)) + card V, lit-neg (?rs!i))]
```

```
?rho = 0
      using r-eq rho-ReifAction True by (simp add: lit-neg-def eval-lit-def)
    have total-sum: pb-sum (fst φ) ?rho = 2 + card (pre (as!i)) + card V
    proof −
      have pb-sum (fst φ) ?rho =
        pb-sum [(2 + card (pre (as!i)) + card V, lit-neg (?rs!i))] ?rho +
        pb-sum ([(1, Pos (ReifDeltaCost (cost (as!i))))] @
         map (λv. (1, Pos (StateVar (Original v)))) (sorted-list-of-set (pre (as!i)))
@
         map (λv. (1, Pos (StateVar (Primed v)))) (sorted-list-of-set (add (as!i)))
@
         map (λv. (1, Neg (StateVar (Primed v)))) (sorted-list-of-set (del (as!i)))
@
          map (λv. (1, Pos (ReifEq (Original v)))) (sorted-list-of-set (V − evars
(as!i))) @
           [(1, Neg (ReifPrimedCostGe B))]) ?rho
        by (simp add: lhs-eq pb-sum-append)
      also have ... = 1 + card (pre (as!i)) +
       (card (add (as!i)) + card (del (as!i)) − s′) + card (V − evars (as!i))) + 1
            unfolding pb-sum-append delta-val pre-val add-val del-val frame-val
bound-val lit-neg-pb
        by simp
      also have ... = 1 + card (pre (as!i)) + card V + 1
        by (simp add: card-add-del-frame)
      also have ... = 2 + card (pre (as!i)) + card V by simp
      finally show ?thesis .
    qed
    show satisfies φ ?rho
      using total-sum A-val by (simp add: satisfies-def prod.case-eq-if)
  next
    case False
    have r-eq: ?rs ! i = Pos (ReifAction i)
      unfolding action-reifs-def using i-lt by simp
    have eval-r-neg-1: eval-lit (lit-neg (?rs ! i)) ?rho = 1
      by (simp add: r-eq False lit-neg-def eval-lit-def rho-ReifAction)
    show satisfies φ ?rho
    proof −
      have mem: (2 + card (pre (as!i)) + card V, lit-neg (?rs!i)) ∈ set (fst φ)
        unfolding φ-eq action-constraint-def Let-def by simp
      have sum-ge: pb-sum (fst φ) ?rho ≥ 2 + card (pre (as!i)) + card V
      proof −
        have pb-sum (fst φ) ?rho ≥
             (2 + card (pre (as!i)) + card V) ∗ eval-lit (lit-neg (?rs!i)) ?rho
          by (rule pb-sum-ge-term[OF mem])
       also have (2 + card (pre (as!i)) + card V) ∗ eval-lit (lit-neg (?rs!i)) ?rho
              = 2 + card (pre (as!i)) + card V
          using eval-r-neg-1 by simp
        finally show ?thesis .
      qed
```

```
      have A-eq: snd φ = 2 + card (pre (as!i)) + card V
        unfolding φ-eq action-constraint-def Let-def by simp
      show satisfies φ ?rho
        using sum-ge A-eq by (simp add: satisfies-def prod.case-eq-if)
    qed
  qed
qed

have models-delta-cs:
  models (⋃a∈set as. encode-delta-cost (cost a) (bits-needed B)) ?rho
unfolding models-def
proof (intro ballI)
  fix φ :: 'v var pconstraint
  assume φ-in: φ ∈ (⋃a∈set as. encode-delta-cost (cost a) (bits-needed B))
  from φ-in obtain a :: 'v action where a-in: a ∈ set as
    and φ-dc: φ ∈ encode-delta-cost (cost a) (bits-needed B) by auto
  let ?k   = cost a
  let ?pcl  = map (λi. (2^i, Pos (PrimedCostBit i))) [0..<bits-needed B]
  let ?cl   = map (λi. (2^i, Pos (CostBit i))) [0..<bits-needed B]
  let ?ncl  = map (λi. (2^i, Neg (CostBit i))) [0..<bits-needed B]
  let ?ncl' = map (λi. (2^i, Neg (PrimedCostBit i))) [0..<bits-needed B]
  let ?M    = 2^(bits-needed B) - (1::nat)
  have c-bound: c < 2^(bits-needed B)
    using add-lessD1 c'-bound cost-eq by blast
  have all-01: ∀ v. ?rho v = 0 ∨ ?rho v = 1 by (rule two-state-rho-range)
  have pb-pcl: pb-sum ?pcl ?rho = c'
    by (metis (mono-tags, lifting) c'-bound mod-less pb-sum-primed-cost-bits-gen
        rho-PrimedCostBit)
  have pb-cl: pb-sum ?cl ?rho = c
  by (metis (mono-tags, lifting) c-bound div-less mod-eq-self-iff-div-eq-0 pb-sum-cost-bits-gen
        rho-CostBit)
  have pb-ncl: pb-sum ?ncl ?rho = ?M - c
  proof -
    have pb-sum ?cl ?rho + pb-sum ?ncl ?rho = sum-list (map fst ?cl)
      using pb-sum-add-negated-gen[OF all-01, of ?cl]
      by (auto simp: lit-neg-def o-def)
     moreover have sum-list (map fst ?cl) = ?M by (auto simp: sum-list-exp
o-def)
    ultimately show ?thesis using pb-cl
      by (metis (no-types, lifting) diff-add-inverse)
  qed
  have pb-ncl': pb-sum ?ncl' ?rho = ?M - c'
  proof -
    have pb-sum ?pcl ?rho + pb-sum ?ncl' ?rho = sum-list (map fst ?pcl)
     using pb-sum-add-negated-gen[OF all-01, of ?pcl] by (auto simp: lit-neg-def
o-def)
     moreover have sum-list (map fst ?pcl) = ?M by (auto simp: sum-list-exp
o-def)
    ultimately show ?thesis using pb-pcl
```

```
      by (metis (no-types, lifting) diff-add-inverse)
    qed
    have rho-rl: ?rho (ReifDeltaCostLower ?k) = (if c' ≥ c + ?k then 1 else 0)
      by (simp add: two-state-rho-def)
    have rho-ru: ?rho (ReifDeltaCostUpper ?k) = (if c' ≤ c + ?k then 1 else 0)
      by (simp add: two-state-rho-def)
    have rho-rd: ?rho (ReifDeltaCost ?k) = (if ?k = c' − c then 1 else 0)
      using rho-ReifDeltaCost by simp
    have rd-iff: ?rho (ReifDeltaCost ?k) = 1 ⟷
                ?rho (ReifDeltaCostLower ?k) = 1 ∧ ?rho (ReifDeltaCostUpper ?k)
= 1
      using rho-rd rho-rl rho-ru cost-eq by (auto split: if-splits)
    from φ-dc have φ-cases:
      φ ∈ reification (Pos (ReifDeltaCostLower ?k)) (?pcl @ ?ncl) (?k + ?M) ∨
       φ ∈ reification (Pos (ReifDeltaCostUpper ?k)) (?cl @ ?ncl') (?M − ?k) ∨
       φ ∈ reification (Pos (ReifDeltaCost ?k))
                  [(1, Pos (ReifDeltaCostLower ?k)), (1, Pos (ReifDeltaCostUpper
?k))] 2
      unfolding encode-delta-cost-def Let-def by auto
    show satisfies φ ?rho using φ-cases
    proof (elim disjE)
      assume h1: φ ∈ reification (Pos (ReifDeltaCostLower ?k)) (?pcl @ ?ncl) (?k
+ ?M)
     from h1 have φ-l: φ = reif-fwd (Pos (ReifDeltaCostLower ?k)) (?pcl @ ?ncl)
(?k + ?M) ∨
                          φ = reif-bwd (Pos (ReifDeltaCostLower ?k)) (?pcl @ ?ncl)
(?k + ?M)
       unfolding reification-def by auto
      show satisfies φ ?rho using φ-l
      proof (elim disjE)
        assume eq: φ = reif-fwd (Pos (ReifDeltaCostLower ?k)) (?pcl @ ?ncl) (?k
+ ?M)
       show ?thesis
         unfolding eq satisfies-def reif-fwd-def
         using rho-rl c-bound c'-bound pb-pcl pb-ncl
         by (auto simp: eval-lit-def lit-neg-def pb-sum-append)
      next
        assume eq: φ = reif-bwd (Pos (ReifDeltaCostLower ?k)) (?pcl @ ?ncl) (?k
+ ?M)
         have c + cost a > c' ⟹ Suc (2^(bits-needed B) − Suc (cost a)) ≤
2^(bits-needed B) + c − Suc c'
         by (smt (verit, best) Nat.diff-cancel add.commute add.left-commute c'-lt-B
             diff-less-mono2 le-trans bits-needed-sufficient linorder-not-less
             not-less-eq plus-1-eq-Suc trans-less-add1)
       then show ?thesis
         unfolding eq satisfies-def reif-bwd-def Let-def
         using rho-rl c-bound c'-bound pb-cl pb-pcl pb-ncl pb-ncl'
         by (auto simp: eval-lit-def lit-neg-def pb-sum-append
           sum-list-exp o-def not-le simp del: upt.simps)
```

```
    qed
  next
    assume h2: φ ∈ reification (Pos (ReifDeltaCostUpper ?k)) (?cl @ ?ncl') (?M
− ?k)
    from h2 have φ-u: φ = reif-fwd (Pos (ReifDeltaCostUpper ?k)) (?cl @ ?ncl')
(?M − ?k) ∨
                          φ = reif-bwd (Pos (ReifDeltaCostUpper ?k)) (?cl @ ?ncl')
(?M − ?k)
      unfolding reification-def by auto
    show satisfies φ ?rho using φ-u
    proof (elim disjE)
      assume eq: φ = reif-fwd (Pos (ReifDeltaCostUpper ?k)) (?cl @ ?ncl') (?M
− ?k)
      show ?thesis
        unfolding eq satisfies-def reif-fwd-def
        using rho-ru c-bound c'-bound pb-cl pb-ncl'
        by (auto simp: eval-lit-def lit-neg-def pb-sum-append)
    next
      assume eq: φ = reif-bwd (Pos (ReifDeltaCostUpper ?k)) (?cl @ ?ncl') (?M
− ?k)
      show ?thesis
        unfolding eq satisfies-def reif-bwd-def Let-def
        using rho-ru c-bound c'-bound pb-pcl pb-ncl
        by (auto simp: eval-lit-def lit-neg-def pb-sum-append
              sum-list-exp o-def simp del: upt.simps)
    qed
  next
    assume h3: φ ∈ reification (Pos (ReifDeltaCost ?k))
                  [(1, Pos (ReifDeltaCostLower ?k)), (1, Pos (ReifDeltaCostUpper
?k))] 2
    from h3 have φ-d:
      φ = reif-fwd (Pos (ReifDeltaCost ?k))
                  [(1, Pos (ReifDeltaCostLower ?k)), (1, Pos (ReifDeltaCostUpper
?k))] 2 ∨
       φ = reif-bwd (Pos (ReifDeltaCost ?k))
                  [(1, Pos (ReifDeltaCostLower ?k)), (1, Pos (ReifDeltaCostUpper
?k))] 2
      unfolding reification-def by auto
     have rl01: ?rho (ReifDeltaCostLower ?k) = 0 ∨ ?rho (ReifDeltaCostLower
?k) = 1
      using all-01 by blast
     have ru01: ?rho (ReifDeltaCostUpper ?k) = 0 ∨ ?rho (ReifDeltaCostUpper
?k) = 1
      using all-01 by blast
    show satisfies φ ?rho using φ-d
    proof (elim disjE)
      assume eq: φ = reif-fwd (Pos (ReifDeltaCost ?k))
                  [(1, Pos (ReifDeltaCostLower ?k)), (1, Pos (ReifDeltaCostUpper
?k))] 2
```

```
      show ?thesis
        unfolding eq satisfies-def reif-fwd-def
        using rd-iff rl01 ru01 rho-rd rho-rl rho-ru
        by (auto simp: eval-lit-def lit-neg-def cost-eq)
    next
      assume eq: φ = reif-bwd (Pos (ReifDeltaCost ?k))
                  [(1, Pos (ReifDeltaCostLower ?k)), (1, Pos (ReifDeltaCostUpper
?k))] 2
      show ?thesis
        unfolding eq satisfies-def reif-bwd-def Let-def
        using rd-iff rl01 ru01
        by (auto simp: eval-lit-def lit-neg-def)
    qed

  qed
 qed

 have models-eq-cs: models (⋃ v∈V. encode-eq-var v) ?rho
 proof (unfold models-def, intro ballI)
  fix φ assume φ-in: φ ∈ (⋃ v∈V. encode-eq-var v)
  then obtain u where u-in: u ∈ V and φ-eq: φ ∈ encode-eq-var u by auto
  from φ-eq have disj:
    φ = reif-fwd (Pos (ReifGeq (Original u))) [(1, Pos (StateVar (Original u))),
(1, Neg (StateVar (Primed u)))] 1 ∨
    φ = reif-bwd (Pos (ReifGeq (Original u))) [(1, Pos (StateVar (Original u))),
(1, Neg (StateVar (Primed u)))] 1 ∨
    φ = reif-fwd (Pos (ReifLeq (Original u))) [(1, Neg (StateVar (Original u))),
(1, Pos (StateVar (Primed u)))] 1 ∨
    φ = reif-bwd (Pos (ReifLeq (Original u))) [(1, Neg (StateVar (Original u))),
(1, Pos (StateVar (Primed u)))] 1 ∨
    φ = reif-fwd (Pos (ReifEq (Original u))) [(1, Pos (ReifLeq (Original u))),
(1, Pos (ReifGeq (Original u)))] 2 ∨
    φ = reif-bwd (Pos (ReifEq (Original u))) [(1, Pos (ReifLeq (Original u))),
(1, Pos (ReifGeq (Original u)))] 2
    unfolding encode-eq-var-def reification-def by auto
  show satisfies φ ?rho using disj
  proof (elim disjE)
    assume eq: φ = reif-fwd (Pos (ReifGeq (Original u))) [(1, Pos (StateVar
(Original u))), (1, Neg (StateVar (Primed u)))] 1
    show ?thesis
      unfolding eq satisfies-def reif-fwd-def
      using rho-ReifGeq rho-StateVar rho-PrimedStateVar
      by (auto simp: eval-lit-def lit-neg-def)
  next
    assume eq: φ = reif-bwd (Pos (ReifGeq (Original u))) [(1, Pos (StateVar
(Original u))), (1, Neg (StateVar (Primed u)))] 1
    show ?thesis
      unfolding eq satisfies-def reif-bwd-def Let-def
      using rho-ReifGeq rho-StateVar rho-PrimedStateVar
```

```
      by (auto simp: eval-lit-def lit-neg-def)
    next
      assume eq: φ = reif-fwd (Pos (ReifLeq (Original u))) [(1, Neg (StateVar
(Original u))), (1, Pos (StateVar (Primed u)))] 1
      show ?thesis
        unfolding eq satisfies-def reif-fwd-def
        using rho-ReifLeq rho-StateVar rho-PrimedStateVar
        by (auto simp: eval-lit-def lit-neg-def)
    next
      assume eq: φ = reif-bwd (Pos (ReifLeq (Original u))) [(1, Neg (StateVar
(Original u))), (1, Pos (StateVar (Primed u)))] 1
      show ?thesis
        unfolding eq satisfies-def reif-bwd-def Let-def
        using rho-ReifLeq rho-StateVar rho-PrimedStateVar
        by (auto simp: eval-lit-def lit-neg-def)
    next
      assume eq: φ = reif-fwd (Pos (ReifEq (Original u))) [(1, Pos (ReifLeq
(Original u))), (1, Pos (ReifGeq (Original u)))] 2
      show ?thesis
        unfolding eq satisfies-def reif-fwd-def
        using rho-ReifEq rho-ReifLeq rho-ReifGeq
        by (auto simp: eval-lit-def lit-neg-def)
    next
      assume eq: φ = reif-bwd (Pos (ReifEq (Original u))) [(1, Pos (ReifLeq
(Original u))), (1, Pos (ReifGeq (Original u)))] 2
      show ?thesis
        unfolding eq satisfies-def reif-bwd-def Let-def
        using rho-ReifEq rho-ReifLeq rho-ReifGeq
        by (auto simp: eval-lit-def lit-neg-def)
    qed
  qed

  have models-primed-ge: models (encode-cost-ge-primed B) ?rho
  proof (unfold models-def, intro ballI)
    fix φ :: 'v var pconstraint assume φ-in: φ ∈ encode-cost-ge-primed B
    from φ-in have disj:
      φ = reif-fwd (Pos (ReifPrimedCostGe B)) (map (λi. (2^i, Pos (PrimedCostBit
i))) [0..<bits-needed B]) B ∨
      φ = reif-bwd (Pos (ReifPrimedCostGe B)) (map (λi. (2^i, Pos (PrimedCostBit
i))) [0..<bits-needed B]) B
      unfolding encode-cost-ge-primed-def reification-def by auto
    show satisfies φ ?rho using disj
    proof (elim disjE)
      assume eq: φ = reif-fwd (Pos (ReifPrimedCostGe B))
            (map (λi. (2^i, Pos (PrimedCostBit i))) [0..<bits-needed B]) B
      show ?thesis
        unfolding eq satisfies-def reif-fwd-def Let-def split-beta snd-conv fst-conv
      proof (cases c' ≥ B)
        case True
```

```
      have r-eval: eval-lit (lit-neg (Pos (ReifPrimedCostGe B))) ?rho = 0
        using rho-ReifPrimedCostGe True by (simp add: eval-lit-def lit-neg-def)
    have pb-coeffs: pb-sum (map (λi. (2^i, Pos (PrimedCostBit i))) [0..<bits-needed
B]) ?rho = c'
        using rho-PrimedCostBit c'-bound
        by (metis (lifting) ext mod-less pb-sum-primed-cost-bits-gen)
      then show pb-sum ([(B, lit-neg (Pos (ReifPrimedCostGe B)))] @
                map (λi. (2^i, Pos (PrimedCostBit i))) [0..<bits-needed B]) ?rho
≥ B
        using r-eval True by (simp add: pb-sum-append)
    next
      case False
      have r-eval: eval-lit (lit-neg (Pos (ReifPrimedCostGe B))) ?rho = 1
        using rho-ReifPrimedCostGe False by (simp add: eval-lit-def lit-neg-def)
      show pb-sum ([(B, lit-neg (Pos (ReifPrimedCostGe B)))] @
                map (λi. (2^i, Pos (PrimedCostBit i))) [0..<bits-needed B]) ?rho
≥ B
        using r-eval by (simp add: pb-sum-append)
    qed
  next
    assume eq: φ = reif-bwd (Pos (ReifPrimedCostGe B))
          (map (λi. (2^i, Pos (PrimedCostBit i))) [0..<bits-needed B]) B
     let ?coeffs = map (λi. (2^i :: nat, Pos (PrimedCostBit i :: 'v var pvar)))
[0..<bits-needed B]
    let ?M = sum-list (map fst ?coeffs)
    have M-val: ?M = 2^(bits-needed B) − 1
      by (auto simp: o-def sum-list-exp)
    have M'-val: ?M − B + 1 = 2^(bits-needed B) − B unfolding M-val
    by (metis Suc-diff-Suc Suc-eq-plus1 diff-Suc-eq-diff-pred bits-needed-sufficient)
    have pb-coeffs: pb-sum ?coeffs ?rho = c'
      using rho-PrimedCostBit c'-bound
      by (metis (lifting) ext mod-less pb-sum-primed-cost-bits-gen)
  have pb-neg: pb-sum (map (λ(a,l). (a, lit-neg l)) ?coeffs) ?rho = (2^(bits-needed
B) − 1) − c'
    proof −
      have all-01: ∀ v. ?rho v = 0 ∨ ?rho v = 1 by (rule two-state-rho-range)
       have pb-sum ?coeffs ?rho + pb-sum (map (λ(a,l). (a, lit-neg l)) ?coeffs)
?rho = ?M
        using all-01 pb-sum-add-negated-gen by blast
      then show ?thesis using pb-coeffs M-val by simp
    qed
    have r-eval: eval-lit (Pos (ReifPrimedCostGe B)) ?rho = 0
      using rho-ReifPrimedCostGe c'-lt-B by (simp add: eval-lit-def)
    have pb-sum ([(?M − B + 1, Pos (ReifPrimedCostGe B))] @
                map (λ(a,l). (a, lit-neg l)) ?coeffs) ?rho
      = (?M − B + 1) * 0 + ((2^(bits-needed B) − 1) − c')
      using r-eval pb-neg M'-val by (simp add: pb-sum-append)
    moreover have Suc c' < 2^(bits-needed B)
    proof −
```

```
      have Suc c' ≤ B using c'-lt-B by simp
      also have B < 2^(bits-needed B) by (rule bits-needed-sufficient)
      finally show ?thesis .
    qed
    ultimately show ?thesis using c'-lt-B
    by (auto simp: satisfies-def reif-bwd-def r-eval eq Let-def o-def mult-2 Suc-le-eq
sum-list-exp
        intro!: diff-less-mono2)
  qed
 qed

 have models-sel: models (action-selection-reif ?rs) ?rho
 proof (unfold models-def, intro ballI)
  fix φ :: 'v var pconstraint assume φ-in: φ ∈ action-selection-reif ?rs
  from φ-in have disj:
   φ = reif-fwd (Pos ReifT) (map (λi. (1, Pos (ReifAction i))) [0..<length as])
1 ∨
   φ = reif-bwd (Pos ReifT) (map (λi. (1, Pos (ReifAction i))) [0..<length as])
1
   unfolding action-selection-reif-def action-reifs-def reification-def
   by (auto simp: o-def)
  show satisfies φ ?rho using disj
  proof (elim disjE)
    assume eq: φ = reif-fwd (Pos ReifT) (map (λi. (1, Pos (ReifAction i)))
[0..<length as]) 1
   have sum-ge: pb-sum (map (λi. (1, Pos (ReifAction i))) [0..<length as]) ?rho
≥ 1
   proof −
    have mem: (1, Pos (ReifAction sel-i)) ∈ set (map (λi. (1, Pos (ReifAction
i))) [0..<length as])
      using sel-lt by auto
    have pb-sum (map (λi. (1, Pos (ReifAction i))) [0..<length as]) ?rho ≥
          1 * eval-lit (Pos (ReifAction sel-i)) ?rho
      by (rule pb-sum-ge-term[OF mem])
    also have 1 * eval-lit (Pos (ReifAction sel-i)) ?rho = 1
      by (simp add: eval-lit-def rho-ReifAction)
    finally show ?thesis by simp
   qed
   show ?thesis
    unfolding eq satisfies-def reif-fwd-def
    using rho-ReifT sum-ge
    by (simp add: eval-lit-def lit-neg-def)
  next
    assume eq: φ = reif-bwd (Pos ReifT) (map (λi. (1, Pos (ReifAction i)))
[0..<length as]) 1
   show ?thesis
    unfolding eq satisfies-def reif-bwd-def Let-def
    using rho-ReifT by (simp add: eval-lit-def lit-neg-def)
  qed
```

qed

show *?thesis*
unfolding *encode-transition-def Let-def models-def*
using *models-action-cs models-delta-cs models-eq-cs models-primed-ge models-sel*
unfolding *models-def* **by** *auto*
qed

lemma *constraint-pvars-reification*:
assumes $\varphi \in$ *reification r coeffs A*
shows *constraint-pvars* $\varphi \subseteq \{$*pvar-of-lit r*$\} \cup$ (*pvar-of-lit* ‘ *snd* ‘ *set coeffs*)
using *assms* **unfolding** *reification-def reif-fwd-def reif-bwd-def constraint-pvars-def*
by (*force simp*: *pvar-of-lit-lit-neg Let-def split*: *if-splits literal.splits*)
lemma *constraint-pvars-action-constraint-mem*:
$v \in$ *constraint-pvars* (*action-constraint r a V B*) $\Longrightarrow$
$v =$ *pvar-of-lit r* $\vee$ $v =$ *ReifDeltaCost* (*cost a*) $\vee$ *is-input-pvar v* $\vee$
$(\exists\, u.\ v =$ *ReifEq u*$) \vee v =$ *ReifPrimedCostGe B*
unfolding *action-constraint-def constraint-pvars-def Let-def*
by (*auto simp*: *pvar-of-lit-def pvar-of-lit-lit-neg is-input-pvar-def lit-neg-def*
split: *literal.splits*)
lemma *enc-trans-pvars-not-circ*:
fixes *C-*φ :: *'v*::*linorder pb-circuit*
assumes φ*-in*: $\varphi \in$ *encode-transition as V B*
and *extra-disj*: $\forall\, v \in$ *circuit-reif-pvars C-*φ.
$(\forall\, k.\ v \neq$ *ReifDeltaCost k*$) \wedge (\forall\, k.\ v \neq$ *ReifDeltaCostLower k*$) \wedge$
$(\forall\, k.\ v \neq$ *ReifDeltaCostUpper k*$) \wedge$
$(\forall\, k.\ v \neq$ *ReifPrimedCostGe k*$) \wedge v \neq$ *ReifT* $\wedge$
$(\forall\, u.\ v \neq$ *ReifGeq u*$) \wedge (\forall\, u.\ v \neq$ *ReifLeq u*$) \wedge (\forall\, u.\ v \neq$ *ReifEq u*$) \wedge (\forall\, i.\ v$
$\neq$ *ReifAction i*$)$
shows $\forall\ v \in$ *constraint-pvars* φ. $(\exists\, k.\ v =$ *ReifDeltaCost k*$) \vee (\exists\, k.\ v =$ *ReifDelta-CostLower k*$) \vee$
$(\exists\, k.\ v =$ *ReifDeltaCostUpper k*$) \vee$
$(\exists\, k.\ v =$ *ReifPrimedCostGe k*$) \vee v =$ *ReifT* $\vee$
$(\exists\, u.\ v =$ *ReifGeq u*$) \vee (\exists\, u.\ v =$ *ReifLeq u*$) \vee (\exists\, u.\ v =$ *ReifEq u*$) \vee$
$(\exists\, i.\ v =$ *ReifAction i*$) \vee$ *is-input-pvar v*
proof
fix *v* **assume** *v-in*: $v \in$ *constraint-pvars* φ
from φ*-in* **show** $(\exists\, k.\ v =$ *ReifDeltaCost k*$) \vee (\exists\, k.\ v =$ *ReifDeltaCostLower k*$)$
$\vee$
$(\exists\, k.\ v =$ *ReifDeltaCostUpper k*$) \vee$
$(\exists\, k.\ v =$ *ReifPrimedCostGe k*$) \vee v =$ *ReifT* $\vee$
$(\exists\, u.\ v =$ *ReifGeq u*$) \vee (\exists\, u.\ v =$ *ReifLeq u*$) \vee (\exists\, u.\ v =$ *ReifEq u*$) \vee$
$(\exists\, i.\ v =$ *ReifAction i*$) \vee$ *is-input-pvar v*
unfolding *encode-transition-def Let-def*
proof (*elim UnE*)
assume $\varphi \in (\bigcup i$<*length as*. $\{$*action-constraint* (*action-reifs as* ! *i*) (*as* ! *i*) *V B*$\})$
then obtain *i* **where** $i <$ *length as* **and** φ*-eq*: $\varphi =$ *action-constraint* (*action-reifs as* ! *i*) (*as* ! *i*) *V B*

by *auto*
with *v-in* **show** *?thesis*
using *constraint-pvars-action-constraint-mem*[*of v action-reifs as ! i as ! i V B*]
by (*auto simp*: *action-reifs-def pvar-of-lit-def*)
next
assume $\varphi \in (\bigcup a{\in}set\ as.\ encode\text{-}delta\text{-}cost\ (cost\ a)\ (bits\text{-}needed\ B))$
then obtain *a* **where** *a-in*: $a \in set\ as$ **and** φ-*eq*: $\varphi \in encode\text{-}delta\text{-}cost\ (cost\ a)\ (bits\text{-}needed\ B)$
by *auto*
let *?k* = *cost a*
let *?M* = $2\hat{}(bits\text{-}needed\ B) - (1{::}nat)$
let *?pcl* = *map* $(\lambda i.\ (2\hat{}i,\ Pos\ (PrimedCostBit\ i)))\ [0..{<}bits\text{-}needed\ B]$
let *?cl* = *map* $(\lambda i.\ (2\hat{}i,\ Pos\ (CostBit\ i)))\ [0..{<}bits\text{-}needed\ B]$
let *?ncl* = *map* $(\lambda i.\ (2\hat{}i,\ Neg\ (CostBit\ i)))\ [0..{<}bits\text{-}needed\ B]$
let *?ncl′* = *map* $(\lambda i.\ (2\hat{}i,\ Neg\ (PrimedCostBit\ i)))\ [0..{<}bits\text{-}needed\ B]$
from φ-*eq* **have** *disj-dc*:
$\varphi \in reification\ (Pos\ (ReifDeltaCostLower\ ?k))\ (?pcl\ @\ ?ncl)\ (?k + ?M) \vee$
$\varphi \in reification\ (Pos\ (ReifDeltaCostUpper\ ?k))\ (?cl\ @\ ?ncl')\ (?M - ?k) \vee$
$\varphi \in reification\ (Pos\ (ReifDeltaCost\ ?k))$
$[(1,\ Pos\ (ReifDeltaCostLower\ ?k)),\ (1,\ Pos\ (ReifDeltaCostUpper\ ?k))]$ *2*
unfolding *encode-delta-cost-def Let-def* **by** *auto*
show *?thesis* **using** *disj-dc*
proof (*elim disjE*)
assume *h*: $\varphi \in reification\ (Pos\ (ReifDeltaCostLower\ ?k))\ (?pcl\ @\ ?ncl)\ (?k + ?M)$
have $v = ReifDeltaCostLower\ ?k \vee (\exists i.\ v = PrimedCostBit\ i) \vee (\exists i.\ v = CostBit\ i)$
using *v-in constraint-pvars-reification*[*OF h*]
by (*auto simp*: *pvar-of-lit-def is-input-pvar-def split*: *literal.splits*)
then show *?thesis* **by** (*auto simp*: *is-input-pvar-def*)
next
assume *h*: $\varphi \in reification\ (Pos\ (ReifDeltaCostUpper\ ?k))\ (?cl\ @\ ?ncl')\ (?M - ?k)$
have $v = ReifDeltaCostUpper\ ?k \vee (\exists i.\ v = CostBit\ i) \vee (\exists i.\ v = PrimedCostBit\ i)$
using *v-in constraint-pvars-reification*[*OF h*]
by (*auto simp*: *pvar-of-lit-def is-input-pvar-def split*: *literal.splits*)
then show *?thesis* **by** (*auto simp*: *is-input-pvar-def*)
next
assume *h*: $\varphi \in reification\ (Pos\ (ReifDeltaCost\ ?k))$
$[(1,\ Pos\ (ReifDeltaCostLower\ ?k)),\ (1,\ Pos\ (ReifDeltaCostUpper\ ?k))]$ *2*
have $v = ReifDeltaCost\ ?k \vee v = ReifDeltaCostLower\ ?k \vee v = ReifDeltaCostUpper\ ?k$
using *v-in constraint-pvars-reification*[*OF h*]
by (*auto simp*: *pvar-of-lit-def split*: *literal.splits*)
then show *?thesis* **by** *auto*

qed
next
assume $\varphi \in (\bigcup v{\in}V.\ \textit{encode-eq-var}\ v)$
then obtain u **where** *u-in*: $u \in V$ **and** φ*-eq*: $\varphi \in \textit{encode-eq-var}\ u$
by *auto*
from φ*-eq* **have** *disj6*:
$\varphi = \textit{reif-fwd}\ (\textit{Pos}\ (\textit{ReifGeq}\ (\textit{Original}\ u)))\ [(1,\ \textit{Pos}\ (\textit{StateVar}\ (\textit{Original}\ u))),\ (1,\ \textit{Neg}\ (\textit{StateVar}\ (\textit{Primed}\ u)))]\ 1 \vee$
$\varphi = \textit{reif-bwd}\ (\textit{Pos}\ (\textit{ReifGeq}\ (\textit{Original}\ u)))\ [(1,\ \textit{Pos}\ (\textit{StateVar}\ (\textit{Original}\ u))),\ (1,\ \textit{Neg}\ (\textit{StateVar}\ (\textit{Primed}\ u)))]\ 1 \vee$
$\varphi = \textit{reif-fwd}\ (\textit{Pos}\ (\textit{ReifLeq}\ (\textit{Original}\ u)))\ [(1,\ \textit{Neg}\ (\textit{StateVar}\ (\textit{Original}\ u))),\ (1,\ \textit{Pos}\ (\textit{StateVar}\ (\textit{Primed}\ u)))]\ 1 \vee$
$\varphi = \textit{reif-bwd}\ (\textit{Pos}\ (\textit{ReifLeq}\ (\textit{Original}\ u)))\ [(1,\ \textit{Neg}\ (\textit{StateVar}\ (\textit{Original}\ u))),\ (1,\ \textit{Pos}\ (\textit{StateVar}\ (\textit{Primed}\ u)))]\ 1 \vee$
$\varphi = \textit{reif-fwd}\ (\textit{Pos}\ (\textit{ReifEq}\ (\textit{Original}\ u)))\ [(1,\ \textit{Pos}\ (\textit{ReifLeq}\ (\textit{Original}\ u))),\ (1,\ \textit{Pos}\ (\textit{ReifGeq}\ (\textit{Original}\ u)))]\ 2 \vee$
$\varphi = \textit{reif-bwd}\ (\textit{Pos}\ (\textit{ReifEq}\ (\textit{Original}\ u)))\ [(1,\ \textit{Pos}\ (\textit{ReifLeq}\ (\textit{Original}\ u))),\ (1,\ \textit{Pos}\ (\textit{ReifGeq}\ (\textit{Original}\ u)))]\ 2$
unfolding *encode-eq-var-def reification-def* **by** *auto*
show *?thesis* **using** *disj6*
proof (*elim disjE*)
assume *eq*: $\varphi = \textit{reif-fwd}\ (\textit{Pos}\ (\textit{ReifGeq}\ (\textit{Original}\ u)))\ [(1,\ \textit{Pos}\ (\textit{StateVar}\ (\textit{Original}\ u))),\ (1,\ \textit{Neg}\ (\textit{StateVar}\ (\textit{Primed}\ u)))]\ 1$
have $v = \textit{ReifGeq}\ (\textit{Original}\ u) \vee v = \textit{StateVar}\ (\textit{Original}\ u) \vee v = \textit{StateVar}\ (\textit{Primed}\ u)$
using *v-in* **by** (*auto simp*: *eq constraint-pvars-def reif-fwd-def pvar-of-lit-def lit-neg-def*
split: *literal.splits*)
then show *?thesis* **by** (*auto simp*: *is-input-pvar-def*)
next
assume *eq*: $\varphi = \textit{reif-bwd}\ (\textit{Pos}\ (\textit{ReifGeq}\ (\textit{Original}\ u)))\ [(1,\ \textit{Pos}\ (\textit{StateVar}\ (\textit{Original}\ u))),\ (1,\ \textit{Neg}\ (\textit{StateVar}\ (\textit{Primed}\ u)))]\ 1$
have $v = \textit{ReifGeq}\ (\textit{Original}\ u) \vee v = \textit{StateVar}\ (\textit{Original}\ u) \vee v = \textit{StateVar}\ (\textit{Primed}\ u)$
using *v-in* **by** (*auto simp*: *eq constraint-pvars-def reif-bwd-def pvar-of-lit-def lit-neg-def*
pvar-of-lit-lit-neg Let-def split: *literal.splits*)
then show *?thesis* **by** (*auto simp*: *is-input-pvar-def*)
next
assume *eq*: $\varphi = \textit{reif-fwd}\ (\textit{Pos}\ (\textit{ReifLeq}\ (\textit{Original}\ u)))\ [(1,\ \textit{Neg}\ (\textit{StateVar}\ (\textit{Original}\ u))),\ (1,\ \textit{Pos}\ (\textit{StateVar}\ (\textit{Primed}\ u)))]\ 1$
have $v = \textit{ReifLeq}\ (\textit{Original}\ u) \vee v = \textit{StateVar}\ (\textit{Original}\ u) \vee v = \textit{StateVar}\ (\textit{Primed}\ u)$
using *v-in* **by** (*auto simp*: *eq constraint-pvars-def reif-fwd-def pvar-of-lit-def lit-neg-def*
split: *literal.splits*)
then show *?thesis* **by** (*auto simp*: *is-input-pvar-def*)
next
assume *eq*: $\varphi = \textit{reif-bwd}\ (\textit{Pos}\ (\textit{ReifLeq}\ (\textit{Original}\ u)))\ [(1,\ \textit{Neg}\ (\textit{StateVar}$

```
(Original u))), (1, Pos (StateVar (Primed u)))] 1
    have v = ReifLeq (Original u) ∨ v = StateVar (Original u) ∨ v = StateVar
(Primed u)
     using v-in by (auto simp: eq constraint-pvars-def reif-bwd-def pvar-of-lit-def
lit-neg-def
                        pvar-of-lit-lit-neg Let-def split: literal.splits)
    then show ?thesis by (auto simp: is-input-pvar-def)
  next
     assume eq: φ = reif-fwd (Pos (ReifEq (Original u))) [(1, Pos (ReifLeq
(Original u))), (1, Pos (ReifGeq (Original u)))] 2
    have v = ReifEq (Original u) ∨ v = ReifLeq (Original u) ∨ v = ReifGeq
(Original u)
     using v-in by (auto simp: eq constraint-pvars-def reif-fwd-def pvar-of-lit-def
lit-neg-def
                        split: literal.splits)
    then show ?thesis by auto
  next
     assume eq: φ = reif-bwd (Pos (ReifEq (Original u))) [(1, Pos (ReifLeq
(Original u))), (1, Pos (ReifGeq (Original u)))] 2
    have v = ReifEq (Original u) ∨ v = ReifLeq (Original u) ∨ v = ReifGeq
(Original u)
     using v-in by (auto simp: eq constraint-pvars-def reif-bwd-def pvar-of-lit-def
lit-neg-def
                        pvar-of-lit-lit-neg Let-def split: literal.splits)
    then show ?thesis by auto
  qed
 next
  assume φ ∈ encode-cost-ge-primed B
  then have disj3:
   φ = reif-fwd (Pos (ReifPrimedCostGe B))
         (map (λi. (2^i, Pos (PrimedCostBit i))) [0..<bits-needed B]) B ∨
   φ = reif-bwd (Pos (ReifPrimedCostGe B))
         (map (λi. (2^i, Pos (PrimedCostBit i))) [0..<bits-needed B]) B
   unfolding encode-cost-ge-primed-def reification-def by auto
  show ?thesis using disj3
  proof (elim disjE)
   assume eq: φ = reif-fwd (Pos (ReifPrimedCostGe B))
                 (map (λi. (2^i, Pos (PrimedCostBit i))) [0..<bits-needed B]) B
   have v = ReifPrimedCostGe B ∨ (∃ i. v = PrimedCostBit i)
    using v-in by (auto simp: eq constraint-pvars-def reif-fwd-def pvar-of-lit-def
lit-neg-def
                        split: literal.splits)
   then show ?thesis by (auto simp: is-input-pvar-def)
  next
   assume eq: φ = reif-bwd (Pos (ReifPrimedCostGe B))
                 (map (λi. (2^i, Pos (PrimedCostBit i))) [0..<bits-needed B]) B
   have v = ReifPrimedCostGe B ∨ (∃ i. v = PrimedCostBit i)
   using v-in by (clarsimp simp: eq constraint-pvars-def reif-bwd-def pvar-of-lit-def
lit-neg-def
```

```
                                    Let-def split: literal.splits; arith)
      then show ?thesis by (auto simp: is-input-pvar-def)
    qed
  next
    assume φ ∈ action-selection-reif (action-reifs as)
    then have disj2-sel:
      φ = reif-fwd (Pos ReifT) (map (λi. (1, Pos (ReifAction i))) [0..<length as])
1 ∨
       φ = reif-bwd (Pos ReifT) (map (λi. (1, Pos (ReifAction i))) [0..<length as])
1
      unfolding action-selection-reif-def action-reifs-def reification-def
      by (auto simp: o-def)
    show ?thesis using disj2-sel
    proof (elim disjE)
       assume eq: φ = reif-fwd (Pos ReifT) (map (λi. (1, Pos (ReifAction i)))
[0..<length as]) 1
      have v = ReifT ∨ (∃ i. v = ReifAction i)
       using v-in by (auto simp: eq constraint-pvars-def reif-fwd-def pvar-of-lit-def
lit-neg-def
                              split: literal.splits)
      then show ?thesis by auto
    next
       assume eq: φ = reif-bwd (Pos ReifT) (map (λi. (1, Pos (ReifAction i)))
[0..<length as]) 1
      have v = ReifT ∨ (∃ i. v = ReifAction i)
       using v-in by (auto simp: eq constraint-pvars-def reif-bwd-def pvar-of-lit-def
lit-neg-def
                              pvar-of-lit-lit-neg Let-def split: literal.splits)
      then show ?thesis by auto
    qed
  qed
qed



lemma two-state-rho-encode-cost-ge-below:
  fixes Π :: 'v::linorder strips-task
  assumes c < B
  shows models (encode-cost-ge B) (two-state-rho Π s c s' c' sel-i as)
proof −
  let ?ϱ = two-state-rho Π s c s' c' sel-i as
  let ?r = Pos (ReifCostGe B)
  let ?coeffs = map (λi. (2^i, Pos (CostBit i))) [0..<bits-needed B] :: (nat × 'v
var pvar literal) list
  have r-val: ?ϱ (ReifCostGe B) = 0
    by (simp add: two-state-rho-def)
  have eval-r: eval-lit ?r ?ϱ = 0
    by (simp add: eval-lit-def r-val)
  have eval-neg-r: eval-lit (lit-neg ?r) ?ϱ = 1
```

```
  by (simp add: eval-lit-def lit-neg-def r-val)
 have cost-bits: ∀ i < bits-needed B. ?ϱ (CostBit i) = (c div 2^i) mod 2
  by (simp add: two-state-rho-def)
 have c-lt-pow: c < 2^(bits-needed B)
 proof −
  have c < B using assms .
  also have B < 2^(bits-needed B) by (rule bits-needed-sufficient)
  finally show ?thesis .
 qed
 have sum-eq: pb-sum ?coeffs ?ϱ = c
 proof −
  have pb-sum ?coeffs ?ϱ = c mod 2^(bits-needed B)
   by (rule pb-sum-cost-bits-gen[OF cost-bits])
  thus ?thesis using c-lt-pow by simp
 qed
 have sum-total: sum-list (map fst ?coeffs) = 2^(bits-needed B) − (1 :: nat)
  by (auto simp: sum-list-exp o-def)
 have all-01: ∀ v. ?ϱ v = 0 ∨ ?ϱ v = 1 by (rule two-state-rho-range)
 have neg-sum-eq: pb-sum (map (λ(a, l). (a, lit-neg l)) ?coeffs) ?ϱ
    = 2^(bits-needed B) − 1 − c
 proof −
  have pb-sum ?coeffs ?ϱ + pb-sum (map (λ(a, l). (a, lit-neg l)) ?coeffs) ?ϱ
       = sum-list (map fst ?coeffs)
    using all-01 pb-sum-add-negated-gen
    by (metis (mono-tags, lifting))
 thus ?thesis using sum-eq sum-total by (metis (no-types, lifting) diff-add-inverse)
 qed
 have fwd-sound: satisfies (reif-fwd ?r ?coeffs B) ?ϱ
  unfolding satisfies-def reif-fwd-def Let-def
  using eval-neg-r sum-eq by simp
 have ineq: (2^(bits-needed B)−1) − c ≥ 2^(bits-needed B) − B
 proof −
  have c+1 ≤ B using assms by simp
  then have 2^(bits-needed B) − B ≤ 2^(bits-needed B) − (c+1) by (simp add:
diff-le-mono2)
  also have 2^(bits-needed B) − (c+1) = (2^(bits-needed B)−1) − c
   by simp
  finally show ?thesis .
 qed
 have M-val: sum-list (map fst ?coeffs) + 1 − B = 2^(bits-needed B) − B
  using sum-total by simp
 have bwd-sound: satisfies (reif-bwd ?r ?coeffs B) ?ϱ
 proof −
  have snd-val: snd (reif-bwd ?r ?coeffs B) = 2^(bits-needed B) − B
   unfolding reif-bwd-def Let-def snd-conv
   using M-val by simp
  have fst-val: pb-sum (fst (reif-bwd ?r ?coeffs B)) ?ϱ = 2^(bits-needed B) − 1
− c
  proof −
```

have *pb-sum* (*fst* (*reif-bwd ?r ?coeffs B*)) *?ϱ* =
pb-sum ([(*sum-list* (*map fst ?coeffs*) + *1* − *B*, *?r*)] @ *map* (λ(*a*,*l*). (*a*, *lit-neg l*)) *?coeffs*) *?ϱ*
by (*auto simp*: *reif-bwd-def Let-def o-def*)
also have ... = (*sum-list* (*map fst ?coeffs*) + *1* − *B*) ∗ *eval-lit ?r ?ϱ* + *pb-sum* (*map* (λ(*a*,*l*). (*a*, *lit-neg l*)) *?coeffs*) *?ϱ*
by (*simp add*: *pb-sum-append eval-r*)
also have ... = *0* + *pb-sum* (*map* (λ(*a*,*l*). (*a*, *lit-neg l*)) *?coeffs*) *?ϱ*
using *eval-r* **by** *simp*
also have ... = *2*^(*bits-needed B*) − *1* − *c*
using *neg-sum-eq* **by** *simp*
finally show *?thesis* **.**
qed
show *?thesis*
unfolding *satisfies-def*
using *fst-val ineq snd-val* **by** *force*
qed
show *?thesis*
unfolding *models-def encode-cost-ge-def reification-def*
using *fwd-sound bwd-sound* **by** *auto*
qed

lemma *lift-to-var-map-pvar-orig*:
assumes ⋀*i*. *p* ≠ *PrimedCostBit i*
shows *lift-to-var rho* (*map-pvar Original p*) = *rho p*
by (*cases p*; *auto simp*: *lift-to-var-def assms split*: *var.splits*)

lemma *eval-lit-map-pvar-lift-to-var-orig*:
assumes ∀ *i*. *pvar-of-lit l* ≠ *PrimedCostBit i*
shows *eval-lit* (*map-literal* (*map-pvar Original*) *l*) (*lift-to-var rho*) = *eval-lit l rho*
using *assms*
by (*cases l*; *simp add*: *eval-lit-def lift-to-var-map-pvar-orig pvar-of-lit-def*)

lemma *eval-lit-neg-lit-map-pvar-lift-to-var-orig*:
assumes ∀ *i*. *pvar-of-lit l* ≠ *PrimedCostBit i*
shows *eval-lit* (*lit-neg* (*map-literal* (*map-pvar Original*) *l*)) (*lift-to-var rho*) = *eval-lit* (*lit-neg l*) *rho*
using *assms*
by (*cases l*; *simp add*: *eval-lit-def lift-to-var-map-pvar-orig pvar-of-lit-def lit-neg-def*)

lemma *lift-to-var-primed-pvar-map*:
assumes ⋀*i*. *p* ≠ *PrimedCostBit i*
shows *lift-to-var rho* (*primed-pvar-map p*) = *rho p*
by (*cases p*; *auto simp*: *lift-to-var-def primed-pvar-map-def assms split*: *var.splits*)

lemma *eval-lit-primed-pvar-map-lift-to-var*:
assumes ∀ *i*. *pvar-of-lit l* ≠ *PrimedCostBit i*
shows *eval-lit* (*map-literal primed-pvar-map l*) (*lift-to-var rho*) = *eval-lit l rho*

using *assms*
by (*cases l*; *simp add*: *eval-lit-def lift-to-var-primed-pvar-map pvar-of-lit-def*)

lemma *pb-sum-map-literal-lift-to-var-orig*:
assumes *no-pcb*: ∀ (*a*, *l*) ∈ *set cs*. ∀ *i*. *pvar-of-lit l* ≠ *PrimedCostBit i*
shows *pb-sum* (*map* (λ(*a*, *l*). (*a*, *map-literal* (*map-pvar Original*) *l*)) *cs*) (*lift-to-var rho*) = *pb-sum cs rho*
using *assms*
by (*induct cs*) (*auto simp*: *eval-lit-map-pvar-lift-to-var-orig*)

lemma *pb-sum-lit-neg-map-literal-lift-to-var-orig*:
assumes *no-pcb*: ∀ (*a*, *l*) ∈ *set cs*. ∀ *i*. *pvar-of-lit l* ≠ *PrimedCostBit i*
shows *pb-sum* (*map* (λ*al*. (*fst al*, *lit-neg* (*map-literal* (*map-pvar Original*) (*snd al*)))) *cs*) (*lift-to-var rho*) =
pb-sum (*map* (λ(*a*, *l*). (*a*, *lit-neg l*)) *cs*) *rho*
using *assms*
by (*induct cs*) (*auto simp*: *eval-lit-neg-lit-map-pvar-lift-to-var-orig*)

lemma *pb-sum-map-literal-lift-to-var-primed*:
assumes *no-pcb*: ∀ (*a*, *l*) ∈ *set cs*. ∀ *i*. *pvar-of-lit l* ≠ *PrimedCostBit i*
shows *pb-sum* (*map* (λ(*a*, *l*). (*a*, *map-literal primed-pvar-map l*)) *cs*) (*lift-to-var rho*) = *pb-sum cs rho*
using *assms*
proof (*induct cs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*)
obtain *a l* **where** *x*: *x* = (*a*, *l*) **by** *force*
have *no-pcb-l*: ∀ *i*. *pvar-of-lit l* ≠ *PrimedCostBit i*
using *Cons.prems* **unfolding** *x* **by** *auto*
have *eval-lit* (*map-literal primed-pvar-map l*) (*lift-to-var rho*) = *eval-lit l rho*
using *no-pcb-l* **by** (*rule eval-lit-primed-pvar-map-lift-to-var*)
then have *hd*: *a* ∗ *eval-lit* (*map-literal primed-pvar-map l*) (*lift-to-var rho*) = *a* ∗ *eval-lit l rho* **by** *simp*
have *IH*: *pb-sum* (*map* (λ(*a*, *l*). (*a*, *map-literal primed-pvar-map l*)) *xs*) (*lift-to-var rho*) = *pb-sum xs rho*
using *Cons.prems* **by** (*intro Cons.hyps*) *auto*
show *?case* **by** (*simp add*: *x hd IH*)
qed

lemma *eval-lit-neg-lit-primed-pvar-map-lift-to-var*:
assumes ∀ *i*. *pvar-of-lit l* ≠ *PrimedCostBit i*
shows *eval-lit* (*lit-neg* (*map-literal primed-pvar-map l*)) (*lift-to-var rho*) = *eval-lit* (*lit-neg l*) *rho*
using *assms*
by (*cases l*; *simp add*: *eval-lit-def lift-to-var-primed-pvar-map pvar-of-lit-def lit-neg-def*)

lemma *pb-sum-lit-neg-map-literal-lift-to-var-primed*:
assumes *no-pcb*: ∀ (*a*, *l*) ∈ *set cs*. ∀ *i*. *pvar-of-lit l* ≠ *PrimedCostBit i*

shows *pb-sum* (*map* (λ*al*. (*fst al*, *lit-neg* (*map-literal primed-pvar-map* (*snd al*)))) *cs*) (*lift-to-var rho*) =
pb-sum (*map* (λ(*a*, *l*). (*a*, *lit-neg l*)) *cs*) *rho*
using *assms*
by (*induct cs*) (*auto simp*: *eval-lit-neg-lit-primed-pvar-map-lift-to-var*)

lemma *lift-to-var-models-circuit-constraints-orig*:
assumes *models* (*circuit-constraints C*) *rho*
and *no-pcb-cs*: ∀ (*r*, *cs*, *A*) ∈ *set* (*fst C*). ∀ *v* ∈ *pvar-of-lit* ‘ *snd* ‘ *set cs*. ∀ *i*. *v* ≠ *PrimedCostBit i*
and *no-pcb-r*: ∀ (*r*, *cs*, *A*) ∈ *set* (*fst C*). ∀ *i*. *pvar-of-lit r* ≠ *PrimedCostBit i*
shows *models* (*circuit-constraints* (*lift-circuit* (*map-pvar Original*) *C*)) (*lift-to-var rho*)
proof (*unfold models-def*, *intro ballI*)
fix ψ **assume** ψ ∈ *circuit-constraints* (*lift-circuit* (*map-pvar Original*) *C*)
then obtain *r cs A* **where** *rcs*: (*r*, *cs*, *A*) ∈ *set* (*fst C*)
and ψ*-reif*: ψ ∈ *reification* (*map-literal* (*map-pvar Original*) *r*)
(*map* (λ(*a*,*l*). (*a*, *map-literal* (*map-pvar Original*) *l*)) *cs*) *A*
unfolding *circuit-constraints-def lift-circuit-def Let-def*
by (*force simp*: *image-iff split-beta*)
have *reif-sat*: ∀ φ ∈ *reification r cs A*. *satisfies* φ *rho*
using *assms*(*1*) *rcs* **unfolding** *models-def circuit-constraints-def* **by** *blast*
have *no-primed-cs*: ∀ (*a*, *l*) ∈ *set cs*. ∀ *i*. *pvar-of-lit l* ≠ *PrimedCostBit i*
using *no-pcb-cs rcs* **by** (*force simp*: *image-iff*)
have *no-primed-r*: ∀ *i*. *pvar-of-lit r* ≠ *PrimedCostBit i*
using *no-pcb-r rcs* **by** *blast*
have *satisfies* ψ (*lift-to-var rho*)
proof (*cases* ψ = *reif-fwd* (*map-literal* (*map-pvar Original*) *r*)
(*map* (λ(*a*,*l*). (*a*, *map-literal* (*map-pvar Original*) *l*)) *cs*) *A*)
case *True*
then have ψ*-fwd*: ψ = *reif-fwd* (*map-literal* (*map-pvar Original*) *r*)
(*map* (λ(*a*,*l*). (*a*, *map-literal* (*map-pvar Original*) *l*)) *cs*) *A* **.**
have *pb-sum* (*fst* ψ) (*lift-to-var rho*) = *pb-sum* (*fst* (*reif-fwd r cs A*)) *rho*
unfolding ψ*-fwd reif-fwd-def* **using** *no-primed-r*
by (*auto simp*: *pb-sum-append pb-sum-map-literal-lift-to-var-orig no-primed-cs*
eval-lit-neg-lit-map-pvar-lift-to-var-orig)
moreover have *snd* ψ = *snd* (*reif-fwd r cs A*)
unfolding ψ*-fwd reif-fwd-def* **by** *simp*
ultimately show *?thesis*
using *reif-sat*[*THEN ballE*, **where** *x* = *reif-fwd r cs A*]
by (*force simp*: *satisfies-def reification-def*)
next
case *False*
then have ψ*-bwd*: ψ = *reif-bwd* (*map-literal* (*map-pvar Original*) *r*)
(*map* (λ(*a*,*l*). (*a*, *map-literal* (*map-pvar Original*) *l*)) *cs*) *A*
using ψ*-reif* **unfolding** *reification-def* **by** *auto*
have *pb-sum* (*fst* ψ) (*lift-to-var rho*) = *pb-sum* (*fst* (*reif-bwd r cs A*)) *rho*
unfolding ψ*-bwd reif-bwd-def Let-def* **using** *no-primed-r no-primed-cs*
by (*auto simp*: *pb-sum-append split-beta o-def*

pb-sum-map-literal-lift-to-var-orig pb-sum-lit-neg-map-literal-lift-to-var-orig
eval-lit-map-pvar-lift-to-var-orig eval-lit-neg-lit-map-pvar-lift-to-var-orig)
moreover have *snd* ψ = *snd* (*reif-bwd r cs A*)
unfolding ψ*-bwd reif-bwd-def Let-def* **by** (*simp add*: *o-def split-beta*)
ultimately show *?thesis*
using *reif-sat*[*THEN ballE*, **where** x = *reif-bwd r cs A*]
by (*force simp*: *satisfies-def reification-def*)
qed
then show *satisfies* ψ (*lift-to-var rho*) **.**
qed

lemma *lift-to-var-models-circuit-constraints-primed*:
assumes *models* (*circuit-constraints C*) *rho*
and *no-pcb-cs*: $\forall$ (*r*, *cs*, *A*) $\in$ *set* (*fst C*). $\forall$ *v* $\in$ *pvar-of-lit* ‘ *snd* ‘ *set cs*. $\forall$ *i*. *v* $\neq$ *PrimedCostBit i*
and *no-pcb-r*: $\forall$ (*r*, *cs*, *A*) $\in$ *set* (*fst C*). $\forall$ *i*. *pvar-of-lit r* $\neq$ *PrimedCostBit i*
shows *models* (*circuit-constraints* (*lift-circuit primed-pvar-map C*)) (*lift-to-var rho*)
proof (*unfold models-def*, *intro ballI*)
fix ψ **assume** ψ $\in$ *circuit-constraints* (*lift-circuit primed-pvar-map C*)
then obtain *r cs A* **where** *rcs*: (*r*, *cs*, *A*) $\in$ *set* (*fst C*)
and ψ*-reif*: ψ $\in$ *reification* (*map-literal primed-pvar-map r*)
(*map* (λ(*a*,*l*). (*a*, *map-literal primed-pvar-map l*)) *cs*) *A*
unfolding *circuit-constraints-def lift-circuit-def Let-def*
by (*force simp*: *image-iff split-beta*)
have *reif-sat*: $\forall$ φ $\in$ *reification r cs A*. *satisfies* φ *rho*
using *assms*(*1*) *rcs* **unfolding** *models-def circuit-constraints-def* **by** *blast*
have *no-primed-cs*: $\forall$ (*a*, *l*) $\in$ *set cs*. $\forall$ *i*. *pvar-of-lit l* $\neq$ *PrimedCostBit i*
using *no-pcb-cs rcs* **by** (*force simp*: *image-iff*)
have *no-primed-r*: $\forall$ *i*. *pvar-of-lit r* $\neq$ *PrimedCostBit i*
using *no-pcb-r rcs* **by** *blast*
have *satisfies* ψ (*lift-to-var rho*)
proof (*cases* ψ = *reif-fwd* (*map-literal primed-pvar-map r*)
(*map* (λ(*a*,*l*). (*a*, *map-literal primed-pvar-map l*)) *cs*) *A*)
case *True*
then have ψ*-fwd*: ψ = *reif-fwd* (*map-literal primed-pvar-map r*)
(*map* (λ(*a*,*l*). (*a*, *map-literal primed-pvar-map l*)) *cs*) *A* **.**
have *pb-sum* (*fst* ψ) (*lift-to-var rho*) = *pb-sum* (*fst* (*reif-fwd r cs A*)) *rho*
unfolding ψ*-fwd reif-fwd-def* **using** *no-primed-r no-primed-cs*
by (*auto simp*: *pb-sum-append pb-sum-map-literal-lift-to-var-primed*
eval-lit-primed-pvar-map-lift-to-var
eval-lit-neg-lit-primed-pvar-map-lift-to-var)
moreover have *snd* ψ = *snd* (*reif-fwd r cs A*)
unfolding ψ*-fwd reif-fwd-def* **by** *simp*
ultimately show *?thesis*
using *reif-sat*[*THEN ballE*, **where** x = *reif-fwd r cs A*]
by (*force simp*: *satisfies-def reification-def*)
next
case *False*

then have *ψ-bwd*: ψ = *reif-bwd* (*map-literal primed-pvar-map r*)
(*map* (λ(*a*,*l*). (*a*, *map-literal primed-pvar-map l*)) *cs*) *A*
using *ψ-reif* **unfolding** *reification-def* **by** *auto* **have** *pb-sum* (*fst* ψ)
(*lift-to-var rho*) = *pb-sum* (*fst* (*reif-bwd r cs A*)) *rho*
unfolding *ψ-bwd reif-bwd-def Let-def* **using** *no-primed-r no-primed-cs*
by (*auto simp*: *pb-sum-append split-beta o-def*
pb-sum-map-literal-lift-to-var-primed
pb-sum-lit-neg-map-literal-lift-to-var-primed
eval-lit-primed-pvar-map-lift-to-var)
moreover have *snd* ψ = *snd* (*reif-bwd r cs A*)
unfolding *ψ-bwd reif-bwd-def Let-def* **by** (*simp add*: *o-def split-beta*)
ultimately show *?thesis*
using *reif-sat*[*THEN ballE*, **where** *x* = *reif-bwd r cs A*]
by (*force simp*: *satisfies-def reification-def*)
qed
then show *satisfies* ψ (*lift-to-var rho*) **.**
qed

lemma *pvar-of-lit-map-literal-gen*: *pvar-of-lit* (*map-literal f l*) = *f* (*pvar-of-lit l*)
by (*cases l*; *simp add*: *pvar-of-lit-def*)

lemma *lift-circuit-reif-eq*[*simp*]:
circuit-reif-pvars (*lift-circuit f C*) = *f* ‘ *circuit-reif-pvars C*
unfolding *circuit-reif-pvars-def lift-circuit-def*
by (*force simp*: *pvar-of-lit-def image-iff split-beta split*: *prod.splits literal.splits*)

lemma *orig-reif-eq*[*simp*]:
circuit-reif-pvars (*orig-circuit C*) = *map-pvar Original* ‘ *circuit-reif-pvars C*
unfolding *orig-circuit-def* **by** *simp*

lemma *primed-reif-eq*[*simp*]:
circuit-reif-pvars (*primed-circuit C*) = *primed-pvar-map* ‘ *circuit-reif-pvars C*
unfolding *primed-circuit-def* **by** *simp*

lemma *snd-lift-circuit*[*simp*]: *snd* (*lift-circuit f C*) = *map-literal f* (*snd C*)
by (*simp add*: *lift-circuit-def Let-def split*: *prod.split*)

lemma *snd-orig-circuit*[*simp*]: *snd* (*orig-circuit C*) = *map-literal* (*map-pvar Original*) (*snd C*)
unfolding *orig-circuit-def* **by** *simp*

lemma *in-M-step*:
fixes Π :: *'v*::*linorder strips-task* **and** *C-φ* :: *'v pb-circuit*
assumes *in-M-s*: *in-M C-φ* Π *s c*
and *appl*: *applicable a s*
and *a-mem*: *a* ∈ *acts* Π
and *as-cover-acts*: *acts* Π ⊆ *set as*
and *wf*: *wf-circuit C-φ*
and *circ-vars*: ∀ (*r*, *φ*) ∈ *set* (*fst C-φ*). ∀ *v* ∈ *constraint-pvars φ*. *is-input-pvar*

$v \vee v \in$ *circuit-reif-pvars C-*φ
and *no-pcb*: $\forall (r, \varphi) \in$ *set* (*fst C-*φ). $\forall v \in$ *constraint-pvars* φ. $\forall i.\ v \neq$ *Primed-CostBit i*
and *disjoint*: $\forall v \in$ *circuit-reif-pvars C-*φ. $\neg$ *is-input-pvar v* $\wedge$ $v \neq$ *ReifI* $\wedge$ ($\forall k.$ $v \neq$ *ReifCostGe k*)
and *extra-disj*: $\forall v \in$ *circuit-reif-pvars C-*φ.
($\forall k.\ v \neq$ *ReifDeltaCost k*) $\wedge$ ($\forall k.\ v \neq$ *ReifDeltaCostLower k*) $\wedge$
($\forall k.\ v \neq$ *ReifDeltaCostUpper k*) $\wedge$
($\forall k.\ v \neq$ *ReifPrimedCostGe k*) $\wedge$ $v \neq$ *ReifT* $\wedge$
($\forall u.\ v \neq$ *ReifGeq u*) $\wedge$ ($\forall u.\ v \neq$ *ReifLeq u*) $\wedge$ ($\forall u.\ v \neq$ *ReifEq u*) $\wedge$ ($\forall i.\ v \neq$ *ReifAction i*) $\wedge$
($\forall i.\ v \neq$ *PrimedCostBit i*) $\wedge$ $v \neq$ *ReifG*
and *realiz*: $\forall$ (*base* :: *'v pvar* $\Rightarrow$ *nat*). ($\forall$ *v. base v = 0* $\vee$ *base v = 1*) $\longrightarrow$ ($\exists$ *rho*.
models (*circuit-constraints C-*φ) *rho*
$\wedge$ ($\forall$ *v. is-input-pvar v* $\longrightarrow$ *rho v = base v*)
$\wedge$ ($\forall$ *v. rho v = 0* $\vee$ *rho v = 1*))
and *fin-V*: *finite V*
and *pre-sub*: $\forall$ *a* $\in$ *set as. pre a* $\subseteq$ *V*
and *add-sub*: $\forall$ *a* $\in$ *set as. add a* $\subseteq$ *V*
and *del-sub*: $\forall$ *a* $\in$ *set as. del a* $\subseteq$ *V*
and *c-lt-B*: *c + cost a < B*
and *cpr-ind*: *cpr-derives*
(*encode-transition as V B* $\cup$ *circuit-constraints* (*orig-circuit C-*φ) $\cup$
circuit-constraints (*primed-circuit C-*φ) $\cup$ *encode-cost-ge B* $\cup$
{*unit-clause* (*snd* (*orig-circuit C-*φ)), *unit-clause* (*Pos ReifT*)})
(*unit-clause* (*snd* (*primed-circuit C-*φ)))
shows *in-M C-*φ Π (*successor a s*) (*c + cost a*)
proof −
let *?s'* = *successor a s*
let *?c'* = *c + cost a*
obtain *sel-i* **where** *sel-i-lt*: *sel-i < length as* **and** *as-sel*: *as* ! *sel-i* = *a*
using *a-mem as-cover-acts* **by** (*metis in-set-conv-nth subset-eq*)

let *?rho-trans* = *two-state-rho* Π *s c ?s' ?c' sel-i as*

have *rho-trans-01*: $\forall$ *v. ?rho-trans v = 0* $\vee$ *?rho-trans v = 1*
by (*rule two-state-rho-range*)

have *appl-as-sel*: *applicable* (*as* ! *sel-i*) *s*
by (*simp add*: *as-sel appl*)
have *rho-trans-sat*: *models* (*encode-transition as V B*) *?rho-trans*
proof (*rule encode-transition-sound*)
show *finite V* **by** (*rule fin-V*)
show *sel-i < length as* **by** (*rule sel-i-lt*)
show *applicable* (*as* ! *sel-i*) *s* **by** (*simp add*: *as-sel appl*)
show *successor a s* = *successor* (*as* ! *sel-i*) *s* **by** (*simp add*: *as-sel*)
show *c + cost a* = *c + cost* (*as* ! *sel-i*) **by** (*simp add*: *as-sel*)
show *c + cost a < B* **by** (*rule c-lt-B*) **show** *pre* (*as* ! *sel-i*) $\subseteq$ *V*

by (*metis pre-sub sel-i-lt in-set-conv-nth*)
show *add* (*as* ! *sel-i*) ⊆ *V*
by (*metis add-sub sel-i-lt in-set-conv-nth*)
show *del* (*as* ! *sel-i*) ⊆ *V*
by (*metis del-sub sel-i-lt in-set-conv-nth*)
qed

have *c-lt-B'*: *c* < *B*
using *c-lt-B* **by** (*simp add*: *trans-less-add2*)
have *rho-trans-cost-ge*: *models* (*encode-cost-ge B*) *?rho-trans*
by (*rule two-state-rho-encode-cost-ge-below*[*OF c-lt-B'*])

have *rho-trans-ReifT*: *satisfies* (*unit-clause* (*Pos ReifT*)) *?rho-trans*
by (*simp add*: *unit-clause-def satisfies-def eval-lit-def two-state-rho-def*)

obtain *rho-circ* **where**
rho-circ-sat: *models* (*circuit-constraints C-φ*) *rho-circ*
and *rho-circ-out*: *eval-lit* (*snd C-φ*) *rho-circ* = *1*
and *rho-circ-inp*: ∀ *v*. *is-input-pvar v* ⟶ *rho-circ v* = *state-rho* Π *s c v*
and *rho-circ-01*: ∀ *v*. *rho-circ v* = *0* ∨ *rho-circ v* = *1*
using *in-M-s*
by (*auto simp*: *in-M-def state-rho-def is-input-pvar-def split*: *pvar.splits*)

let *?base-s'* = *state-rho* Π *?s' ?c'*
have *base-s'-01*: ∀ *v*. *?base-s' v* = *0* ∨ *?base-s' v* = *1* **using** *state-rho-01* **by** *blast*
obtain *rho-circ'* **where**
rho-circ'-sat: *models* (*circuit-constraints C-φ*) *rho-circ'*
and *rho-circ'-inp*: ∀ *v*. *is-input-pvar v* ⟶ *rho-circ' v* = *?base-s' v*
and *rho-circ'-01*: ∀ *v*. *rho-circ' v* = *0* ∨ *rho-circ' v* = *1*
using *realiz base-s'-01* **by** *blast*

let *?base-s'-old-c* = *state-rho* Π *?s' c*
have *base-s'-old-c-01*: ∀ *v*. *?base-s'-old-c v* = *0* ∨ *?base-s'-old-c v* = *1*
using *state-rho-01* **by** *blast*
obtain *rho-circ''* **where**
rho-circ''-sat: *models* (*circuit-constraints C-φ*) *rho-circ''*
and *rho-circ''-inp*: ∀ *v*. *is-input-pvar v* ⟶ *rho-circ'' v* = *?base-s'-old-c v*
and *rho-circ''-01*: ∀ *v*. *rho-circ'' v* = *0* ∨ *rho-circ'' v* = *1*
using *realiz base-s'-old-c-01* **by** *blast*

let *?rho-combined* = λ*x*. *if x* ∈ *circuit-reif-pvars* (*orig-circuit C-φ*) *then lift-to-var rho-circ x else if x* ∈ *circuit-reif-pvars* (*primed-circuit C-φ*) *then lift-to-var rho-circ' x else ?rho-trans x*

have *circ-vars-orig*:
∀ *φ* ∈ *circuit-constraints* (*orig-circuit C-φ*).
∀ *v* ∈ *constraint-pvars φ*. *v* ∈ *circuit-reif-pvars* (*orig-circuit C-φ*) ∨
(∃ *w*. *v* = *StateVar* (*Original w*)) ∨ (∃ *i*. *v* = *CostBit i*)

```
  using circ-vars no-pcb
 proof (intro ballI allI impI)
  fix φ v
  assume φin: φ ∈ circuit-constraints (orig-circuit C-φ)
   and vin: v ∈ constraint-pvars φ
  from φin obtain r0 cs0 A0 where
     r0-in: (r0, cs0, A0) ∈ set (fst C-φ) and
     φ-reif: φ ∈ reification (map-literal (map-pvar Original) r0)
                 (map (λ(a,l). (a, map-literal (map-pvar Original) l)) cs0) A0
   unfolding circuit-constraints-def orig-circuit-def lift-circuit-def
   by (auto simp: split-beta)
  have r0-in-reif: pvar-of-lit r0 ∈ circuit-reif-pvars C-φ
  unfolding circuit-reif-pvars-def using r0-in by (force simp: image-iff split-beta)
  have v-sub: v ∈ {map-pvar Original (pvar-of-lit r0)} ∪
                  map-pvar Original ‘ (pvar-of-lit ‘ snd ‘ set cs0)
   using constraint-pvars-reification[OF φ-reif] vin
   by (force simp: pvar-of-lit-map-literal-gen image-iff split-beta)
  show v ∈ circuit-reif-pvars (orig-circuit C-φ) ∨
       (∃ w. v = StateVar (Original w)) ∨ (∃ i. v = CostBit i)
  proof (cases v = map-pvar Original (pvar-of-lit r0))
   case True
   then have v ∈ map-pvar Original ‘ circuit-reif-pvars C-φ
    using r0-in-reif by blast
   thus ?thesis using orig-reif-eq by auto
  next
   case False
   with v-sub obtain v0 where v0-in-cs: v0 ∈ pvar-of-lit ‘ snd ‘ set cs0
    and v-eq: v = map-pvar Original v0 by auto
   have v0-cpv: v0 ∈ constraint-pvars (cs0, A0)
    using v0-in-cs unfolding constraint-pvars-def by simp
   from circ-vars r0-in v0-cpv have is-input-pvar v0 ∨ v0 ∈ circuit-reif-pvars
C-φ
    by blast
   thus ?thesis
   proof (elim disjE)
    assume is-input-pvar v0
    then consider (sv) ∃ w. v0 = StateVar w | (cb) ∃ i. v0 = CostBit i | (pcb)
∃ i. v0 = PrimedCostBit i
     unfolding is-input-pvar-def by blast
    thus ?thesis
    proof cases
     case sv
     then obtain w where v0 = StateVar w by blast
     hence v = StateVar (Original w) using v-eq by simp
     thus ?thesis by blast
    next
     case cb
     then obtain i where v0 = CostBit i by blast
     hence v = CostBit i using v-eq by simp
```

```
      thus ?thesis by blast
    next
      case pcb
      then obtain i where v0 = PrimedCostBit i by blast
      with no-pcb r0-in v0-cpv show ?thesis by blast
    qed
  next
    assume v0 ∈ circuit-reif-pvars C-φ
    then have map-pvar Original v0 ∈ circuit-reif-pvars (orig-circuit C-φ)
      using orig-reif-eq by auto
    thus ?thesis using v-eq by blast
  qed
 qed
 qed

 have comb-circ: models (circuit-constraints (orig-circuit C-φ)) ?rho-combined
 proof −
  have lv-sat: models (circuit-constraints (orig-circuit C-φ)) (lift-to-var rho-circ)
    using lift-to-var-models-circuit-constraints-orig[OF rho-circ-sat] no-pcb dis-
joint
     unfolding orig-circuit-def by (auto simp: split-beta constraint-pvars-def
is-input-pvar-def circuit-reif-pvars-def)
  show ?thesis
  proof (unfold models-def, intro ballI)
   fix ψ assume ψin: ψ ∈ circuit-constraints (orig-circuit C-φ)
   have satisfies ψ (lift-to-var rho-circ)
    using lv-sat ψin unfolding models-def by auto
    moreover have ∀ v ∈ constraint-pvars ψ. ?rho-combined v = lift-to-var
rho-circ v
   proof (intro ballI)
    fix v assume v ∈ constraint-pvars ψ
    with ψin have v ∈ circuit-reif-pvars (orig-circuit C-φ) ∨
                   (∃ w. v = StateVar (Original w)) ∨ (∃ i. v = CostBit i)
     using circ-vars-orig by blast
    thus ?rho-combined v = lift-to-var rho-circ v
    proof (elim disjE exE)
     assume v ∈ circuit-reif-pvars (orig-circuit C-φ)
     thus ?thesis by simp
    next
     fix w assume v = StateVar (Original w)
      have sv-not-orig: StateVar (Original w) ∉ circuit-reif-pvars (orig-circuit
C-φ)
     proof
      assume StateVar (Original w) ∈ circuit-reif-pvars (orig-circuit C-φ)
      then have StateVar w ∈ circuit-reif-pvars C-φ
       using orig-reif-eq
      proof (auto simp: image-iff)
       fix x assume x ∈ circuit-reif-pvars C-φ
        and StateVar (Original w) = map-pvar Original x
```

```
        thus StateVar w ∈ circuit-reif-pvars C-φ
          by (cases x; auto)
      qed
      then have sv-not-inp: ¬ is-input-pvar (StateVar w)
        using disjoint by (fast dest!: bspec)
      then have is-input-pvar (StateVar w)
        unfolding is-input-pvar-def by auto
      then show False using sv-not-inp by blast
    qed
  have sv-not-primed: StateVar (Original w) ∉ circuit-reif-pvars (primed-circuit
C-φ)
    proof
      assume StateVar (Original w) ∈ circuit-reif-pvars (primed-circuit C-φ)
      then obtain v where v ∈ circuit-reif-pvars C-φ
        and primed-pvar-map v = StateVar (Original w)
        using primed-reif-eq by auto
      thus False by (cases v; simp add: primed-pvar-map-def split: var.splits)
    qed
      have ?rho-combined (StateVar (Original w)) = ?rho-trans (StateVar
(Original w))
      using sv-not-orig sv-not-primed by simp
    also have ... = (if w ∈ s then 1 else 0)
      by (simp add: two-state-rho-def)
    also have ... = rho-circ (StateVar w)
      using rho-circ-inp by (auto simp: is-input-pvar-def state-rho-def)
    also have ... = lift-to-var rho-circ (StateVar (Original w))
      by (simp add: lift-to-var-def)
    finally show ?thesis unfolding ‹v = StateVar (Original w)› .
  next
    fix i assume v = CostBit i
    have cb-not-orig: CostBit i ∉ circuit-reif-pvars (orig-circuit C-φ)
          by (smt (verit, ccfv-threshold) disjoint image-iff is-input-pvar-def
orig-reif-eq
        map-pvar-Original-inj pvar.map(2))
    have cb-not-primed: CostBit i ∉ circuit-reif-pvars (primed-circuit C-φ)
    proof
      assume CostBit i ∈ circuit-reif-pvars (primed-circuit C-φ)
      then obtain v where v ∈ circuit-reif-pvars C-φ
        and primed-pvar-map v = CostBit i
        using primed-reif-eq by auto
      thus False by (cases v; simp add: primed-pvar-map-def split: pvar.splits
var.splits)
    qed
    have ?rho-combined (CostBit i) = ?rho-trans (CostBit i)
      using cb-not-orig cb-not-primed by simp
    also have ... = (c div 2^i) mod 2
      by (simp add: two-state-rho-def)
    also have ... = rho-circ (CostBit i)
      using rho-circ-inp by (auto simp: is-input-pvar-def state-rho-def)
```

```
        also have ... = lift-to-var rho-circ (CostBit i)
          by (simp add: lift-to-var-def)
        finally show ?thesis unfolding ‹v = CostBit i› .
      qed
    qed
    ultimately show satisfies ψ ?rho-combined
      using satisfies-cong by fast
  qed
 qed

 have comb-primed-circ: models (circuit-constraints (primed-circuit C-φ)) ?rho-combined
 proof −
  have lv-sat': models (circuit-constraints (primed-circuit C-φ)) (lift-to-var rho-circ')
      using lift-to-var-models-circuit-constraints-primed[OF rho-circ'-sat] no-pcb
disjoint
      unfolding primed-circuit-def by (auto simp: split-beta constraint-pvars-def
is-input-pvar-def circuit-reif-pvars-def)
   show ?thesis
   proof (unfold models-def, intro ballI)
    fix ψ assume ψin: ψ ∈ circuit-constraints (primed-circuit C-φ)
    have satisfies ψ (lift-to-var rho-circ')
     using lv-sat' ψin unfolding models-def by auto
     moreover have ∀ v ∈ constraint-pvars ψ. ?rho-combined v = lift-to-var
rho-circ' v
    proof (intro ballI)
     fix v assume v ∈ constraint-pvars ψ
     from ψin obtain r0 cs0 A0 where
        r0-in: (r0, cs0, A0) ∈ set (fst C-φ)
        and ψ-reif: ψ ∈ reification (map-literal primed-pvar-map r0)
                        (map (λ(a,l). (a, map-literal primed-pvar-map l)) cs0) A0
      unfolding circuit-constraints-def primed-circuit-def lift-circuit-def
      by (auto simp: split-beta)
     have r0-in-reif: pvar-of-lit r0 ∈ circuit-reif-pvars C-φ
        unfolding circuit-reif-pvars-def using r0-in by (force simp: image-iff
split-beta)
      have psi-pvars-sub: constraint-pvars ψ ⊆ primed-pvar-map ‘ ({pvar-of-lit
r0} ∪ constraint-pvars (cs0, A0))
      using constraint-pvars-reification[OF ψ-reif]
     by (force simp: pvar-of-lit-map-literal-gen image-iff split-beta constraint-pvars-def)
     obtain u where u-in: u ∈ {pvar-of-lit r0} ∪ constraint-pvars (cs0, A0)
      and v-def: v = primed-pvar-map u
      using psi-pvars-sub ‹v ∈ constraint-pvars ψ› by blast

     have ?rho-combined v = lift-to-var rho-circ' v
     proof (cases v ∈ circuit-reif-pvars (primed-circuit C-φ))
      case True
      have u-in-reif: u ∈ circuit-reif-pvars C-φ
      proof (rule ccontr)
       assume u ∉ circuit-reif-pvars C-φ
```

have $u \in$ *constraint-pvars* (*cs0*, *A0*) **using** *u-in r0-in-reif* ‹$u \notin$ *circuit-reif-pvars C-*φ›
by *blast*
with *circ-vars r0-in* ‹$u \notin$ *circuit-reif-pvars C-*φ› **have** *is-input-pvar u*
by *blast*
with *True primed-reif-eq v-def* **show** *False*
proof (*cases u*)
case (*StateVar w*)
with ‹$v \in$ *circuit-reif-pvars* (*primed-circuit C-*φ)› *primed-reif-eq v-def*
obtain *x* **where** $x \in$ *circuit-reif-pvars C-*φ *StateVar* (*Primed w*) = *primed-pvar-map x*
by (*auto simp*: *image-iff*)
with *disjoint* **have** ¬ *is-input-pvar x* **by** *blast*
with ‹*StateVar* (*Primed w*) = *primed-pvar-map x*› **show** *False*
by (*cases x*; *simp-all add*: *is-input-pvar-def primed-pvar-map-def split*: *var.splits*)
next
case (*CostBit i*)
with ‹$v \in$ *circuit-reif-pvars* (*primed-circuit C-*φ)› *primed-reif-eq v-def*
obtain *x* **where** $x \in$ *circuit-reif-pvars C-*φ *PrimedCostBit i* = *primed-pvar-map x*
by (*auto simp*: *image-iff*)
with *disjoint* **have** ¬ *is-input-pvar x* **by** *blast*
with ‹*PrimedCostBit i* = *primed-pvar-map x*› **show** *False*
by (*cases x*; *simp-all add*: *is-input-pvar-def primed-pvar-map-def*)
next
case (*PrimedCostBit i*)
with ‹$v \in$ *circuit-reif-pvars* (*primed-circuit C-*φ)› *primed-reif-eq v-def*
obtain *x* **where** $x \in$ *circuit-reif-pvars C-*φ *PrimedCostBit i* = *primed-pvar-map x*
by (*auto simp*: *image-iff*)
with *disjoint* **have** ¬ *is-input-pvar x* **by** *blast*
with ‹*PrimedCostBit i* = *primed-pvar-map x*› **show** *False*
by (*cases x*; *simp-all add*: *is-input-pvar-def primed-pvar-map-def*)
qed (*simp-all add*: *is-input-pvar-def*)
qed
have *v-not-in-orig*: $v \notin$ *circuit-reif-pvars* (*orig-circuit C-*φ)
proof
assume $v \in$ *circuit-reif-pvars* (*orig-circuit C-*φ)
with *orig-reif-eq v-def* **obtain** u' **where** $u' \in$ *circuit-reif-pvars C-*φ
and *eq*: *map-pvar Original u'* = *primed-pvar-map u*
by (*auto simp*: *image-iff*)
from *disjoint extra-disj u-in-reif*
have ¬ *is-input-pvar u u* ≠ *ReifI* ∀ *k*. *u* ≠ *ReifCostGe k*
u ≠ *ReifG u* ≠ *ReifT*
∀ *k*. *u* ≠ *ReifDeltaCost k* ∀ *k*. *u* ≠ *ReifDeltaCostLower k*
∀ *k*. *u* ≠ *ReifDeltaCostUpper k*
∀ *k*. *u* ≠ *ReifPrimedCostGe k*
∀ *y*. *u* ≠ *ReifGeq y* ∀ *y*. *u* ≠ *ReifLeq y* ∀ *y*. *u* ≠ *ReifEq y*
∀ *i*. *u* ≠ *ReifAction i*

by *auto*
then obtain x **where** $u = \mathit{ReifCert}\ x$ **by** (*cases* u; *auto simp*: *is-input-pvar-def*)
then have *primed-pvar-map* $u = \mathit{ReifCert}\ (\mathit{Primed}\ x)$ **by** (*simp add*: *primed-pvar-map-def*)
with *eq* **have** *map-pvar Original* $u' = \mathit{ReifCert}\ (\mathit{Primed}\ x)$ **by** *simp*
thus *False* **by** (*cases* u'; *auto simp*: *is-input-pvar-def split*: *var.splits*)
qed
thus *?thesis* **using** *True* **by** (*auto simp*: *Let-def*)
next
case *False*
have *is-input-pvar u*
proof (*rule ccontr*)
assume ¬ *is-input-pvar u*
with *circ-vars r0-in u-in r0-in-reif* **have** $u \in$ *circuit-reif-pvars C-*φ **by** *blast*
then have $v \in$ *circuit-reif-pvars* (*primed-circuit C-*φ)
using *primed-reif-eq v-def* **by** (*auto simp*: *image-iff*)
with *False* **show** *False* **by** *blast*
qed
have *not-in-orig*: $v \notin$ *circuit-reif-pvars* (*orig-circuit C-*φ)
proof
assume $v \in$ *circuit-reif-pvars* (*orig-circuit C-*φ)
with *orig-reif-eq v-def* **obtain** u' **where** u'*-reif*: $u' \in$ *circuit-reif-pvars C-*φ
and *eq*: *map-pvar Original* $u' =$ *primed-pvar-map u*
by (*auto simp*: *image-iff*)
with *disjoint* **have** *not-inp-u*$'$: ¬ *is-input-pvar* u' **by** *blast*
from ‹*is-input-pvar u*› *eq not-inp-u*$'$ **show** *False*
by (*cases u*; *cases* u'; *simp add*: *is-input-pvar-def primed-pvar-map-def split*: *var.splits*)
qed
have *?rho-combined v = ?rho-trans v*
using *False not-in-orig* **by** (*auto simp*: *Let-def*)
also have ... = *lift-to-var rho-circ*$'$ *v* **using** ‹*is-input-pvar u*›
proof (*cases u*)
case (*StateVar w*)
hence $v = \mathit{StateVar}\ (\mathit{Primed}\ w)$ **by** (*simp add*: *v-def primed-pvar-map-def*)
hence *?rho-trans v* = (*if w* ∈ *?s*$'$ *then 1 else 0*)
by (*simp add*: *two-state-rho-def*)
also have ... = *rho-circ*$'$ (*StateVar w*)
using *rho-circ*$'$*-inp* ‹*is-input-pvar u*› *StateVar*
by (*simp add*: *is-input-pvar-def state-rho-StateVar-eq*)
also have ... = *lift-to-var rho-circ*$'$ (*StateVar* (*Primed w*))
by (*simp add*: *lift-to-var-def*)
finally show *?thesis* **using** ‹$v = \mathit{StateVar}\ (\mathit{Primed}\ w)$› **by** *blast*
next
case (*CostBit i*)
hence $v = \mathit{PrimedCostBit}\ i$ **by** (*simp add*: *v-def primed-pvar-map-def*)

```
        hence ?rho-trans v = (?c' div 2^i) mod 2
          by (simp add: two-state-rho-def)
        also have ... = rho-circ' (CostBit i)
          using rho-circ'-inp ‹is-input-pvar u› CostBit
          by (simp add: is-input-pvar-def state-rho-CostBit-eq)
        also have ... = lift-to-var rho-circ' (PrimedCostBit i)
          by (simp add: lift-to-var-def)
        finally show ?thesis using ‹v = PrimedCostBit i› by blast
      next
        case (PrimedCostBit i)
        with u-in no-pcb r0-in r0-in-reif extra-disj show ?thesis by blast
      qed (auto simp: is-input-pvar-def)
      finally show ?thesis .
    qed
    thus ?rho-combined v = lift-to-var rho-circ' v .
  qed
  ultimately show satisfies ψ ?rho-combined
    using satisfies-cong by fast
 qed
qed

have enc-trans-not-reif: ∀ φ ∈ encode-transition as V B.
   ∀ v ∈ constraint-pvars φ.
     v ∉ circuit-reif-pvars (orig-circuit C-φ) ∧
     v ∉ circuit-reif-pvars (primed-circuit C-φ)
proof (intro ballI allI impI)
  fix φ v
  assume φ ∈ encode-transition as V B v ∈ constraint-pvars φ
  have v-chars: (∃ k. v = ReifDeltaCost k) ∨ (∃ k. v = ReifDeltaCostLower k) ∨
      (∃ k. v = ReifDeltaCostUpper k) ∨
      (∃ k. v = ReifPrimedCostGe k) ∨ v = ReifT ∨
      (∃ u. v = ReifGeq u) ∨ (∃ u. v = ReifLeq u) ∨ (∃ u. v = ReifEq u) ∨
      (∃ i. v = ReifAction i) ∨ is-input-pvar v
  proof −
    have extra-disj': ∀ v ∈ circuit-reif-pvars C-φ.
      (∀ k. v ≠ ReifDeltaCost k) ∧ (∀ k. v ≠ ReifDeltaCostLower k) ∧
      (∀ k. v ≠ ReifDeltaCostUpper k) ∧
      (∀ k. v ≠ ReifPrimedCostGe k) ∧ v ≠ ReifT ∧
      (∀ u. v ≠ ReifGeq u) ∧ (∀ u. v ≠ ReifLeq u) ∧ (∀ u. v ≠ ReifEq u) ∧ (∀ i. v
≠ ReifAction i)
      using extra-disj by auto
    show ?thesis
    using enc-trans-pvars-not-circ[OF ‹φ ∈ encode-transition as V B› extra-disj']
        ‹v ∈ constraint-pvars φ› by blast
  qed
 show v ∉ circuit-reif-pvars (orig-circuit C-φ) ∧ v ∉ circuit-reif-pvars (primed-circuit
C-φ)
  proof
    show v ∉ circuit-reif-pvars (orig-circuit C-φ)
```

```
    proof
      assume v ∈ circuit-reif-pvars (orig-circuit C-φ)
    then obtain w where w ∈ circuit-reif-pvars C-φ and v = map-pvar Original
w
        using orig-reif-eq by blast
      hence ¬ is-input-pvar w using disjoint by auto
      moreover have is-input-pvar w ∨
        (∃ k. w = ReifDeltaCost k) ∨ (∃ k. w = ReifDeltaCostLower k) ∨
        (∃ k. w = ReifDeltaCostUpper k) ∨
        (∃ k. w = ReifPrimedCostGe k) ∨ w = ReifT ∨
       (∃ u. w = ReifGeq u) ∨ (∃ u. w = ReifLeq u) ∨ (∃ u. w = ReifEq u) ∨ (∃ i.
w = ReifAction i)
      proof (cases w)
        case (ReifCert x)
        with ‹v = map-pvar Original w› v-chars show ?thesis
          by (elim disjE exE) (auto simp: is-input-pvar-def)
     qed (use v-chars ‹v = map-pvar Original w› in ‹auto simp: is-input-pvar-def
split: pvar.splits var.splits›)
       ultimately show False using extra-disj ‹w ∈ circuit-reif-pvars C-φ› by
auto
    qed
    show v ∉ circuit-reif-pvars (primed-circuit C-φ)
    proof
      assume v ∈ circuit-reif-pvars (primed-circuit C-φ)
    then obtain w where w ∈ circuit-reif-pvars C-φ and v = primed-pvar-map
w
        using primed-reif-eq by auto
      hence ¬ is-input-pvar w using disjoint by auto
      moreover have is-input-pvar w ∨
        (∃ k. w = ReifDeltaCost k) ∨ (∃ k. w = ReifDeltaCostLower k) ∨
        (∃ k. w = ReifDeltaCostUpper k) ∨
        (∃ k. w = ReifPrimedCostGe k) ∨ w = ReifT ∨
       (∃ u. w = ReifGeq u) ∨ (∃ u. w = ReifLeq u) ∨ (∃ u. w = ReifEq u) ∨ (∃ i.
w = ReifAction i)
      proof (cases w)
        case (ReifCert x)
        with ‹v = primed-pvar-map w› v-chars show ?thesis
          by (elim disjE exE) (auto simp: is-input-pvar-def)
     qed (use v-chars ‹v = primed-pvar-map w› in ‹auto simp: is-input-pvar-def
split: pvar.splits var.splits›)
       ultimately show False using extra-disj ‹w ∈ circuit-reif-pvars C-φ› by
auto
    qed
  qed
 qed

 have comb-trans: models (encode-transition as V B) ?rho-combined
 proof (unfold models-def, intro ballI)
   fix φ assume φ ∈ encode-transition as V B
```

```
  have satisfies φ ?rho-trans
    using rho-trans-sat[unfolded models-def] ‹φ ∈ encode-transition as V B› by
auto
  moreover have ∀ v ∈ constraint-pvars φ. ?rho-combined v = ?rho-trans v
    using enc-trans-not-reif ‹φ ∈ encode-transition as V B›
    by auto
  ultimately show satisfies φ ?rho-combined
    using satisfies-cong by fast
 qed

 have comb-cost-ge: models (encode-cost-ge B) ?rho-combined
 proof (unfold models-def, intro ballI)
  fix φ :: 'v var pconstraint assume φin: φ ∈ encode-cost-ge B
  have not-reif: ∀ v ∈ constraint-pvars φ.
   v ∉ circuit-reif-pvars (orig-circuit C-φ) ∧ v ∉ circuit-reif-pvars (primed-circuit
C-φ)
  proof (intro ballI)
   fix v assume v ∈ constraint-pvars φ
   have v-from-ge: v = ReifCostGe B ∨ (∃ i. v = CostBit i)
    using ‹φ ∈ encode-cost-ge B› ‹v ∈ constraint-pvars φ›
    by (force simp: encode-cost-ge-def reification-def reif-fwd-def reif-bwd-def
                    constraint-pvars-def pvar-of-lit-def lit-neg-def Let-def
              split: literal.splits)
  show v ∉ circuit-reif-pvars (orig-circuit C-φ) ∧ v ∉ circuit-reif-pvars (primed-circuit
C-φ)
   proof
    show v ∉ circuit-reif-pvars (orig-circuit C-φ)
    proof
     assume v ∈ circuit-reif-pvars (orig-circuit C-φ)
      then obtain w where w ∈ circuit-reif-pvars C-φ and v = map-pvar
Original w
       using orig-reif-eq by blast
     with v-from-ge show False
     proof (elim disjE exE)
      assume v = ReifCostGe B
       with ‹v = map-pvar Original w› ‹w ∈ circuit-reif-pvars C-φ› disjoint
show False
        by (cases w; auto simp add: is-input-pvar-def)
      next
       fix i assume v = CostBit i
        with ‹v = map-pvar Original w› ‹w ∈ circuit-reif-pvars C-φ› disjoint
show False
        by (cases w; auto simp add: is-input-pvar-def)
     qed
    qed
    show v ∉ circuit-reif-pvars (primed-circuit C-φ)
    proof
     assume v ∈ circuit-reif-pvars (primed-circuit C-φ)
    then obtain w where w ∈ circuit-reif-pvars C-φ and v = primed-pvar-map
```

```
w
          using primed-reif-eq by auto
        with v-from-ge show False
        proof (elim disjE exE)
          assume v = ReifCostGe B
            with ‹v = primed-pvar-map w› ‹w ∈ circuit-reif-pvars C-φ› disjoint
show False
            by (cases w; auto simp add: is-input-pvar-def primed-pvar-map-def)
        next
          fix i assume v = CostBit i
          with ‹v = primed-pvar-map w› show False
            by (cases w; simp add: is-input-pvar-def primed-pvar-map-def)
        qed
      qed
    qed
    have satisfies φ ?rho-trans
      using rho-trans-cost-ge[unfolded models-def] φin by auto
    moreover have ∀ v ∈ constraint-pvars φ. ?rho-combined v = ?rho-trans v
      using not-reif by auto
    ultimately show satisfies φ ?rho-combined
      using satisfies-cong by fast
  qed
  have comb-ReifT: satisfies (unit-clause (Pos ReifT)) ?rho-combined
  proof −
    have not-reif: ReifT ∉ circuit-reif-pvars (orig-circuit C-φ) ∧
                   ReifT ∉ circuit-reif-pvars (primed-circuit C-φ)
    proof
      show ReifT ∉ circuit-reif-pvars (orig-circuit C-φ)
      proof
        assume ReifT ∈ circuit-reif-pvars (orig-circuit C-φ)
        then obtain w where w ∈ circuit-reif-pvars C-φ and ReifT = map-pvar
Original w
          using orig-reif-eq by blast
        hence w = ReifT
          by (cases w; simp)
        with extra-disj ‹w ∈ circuit-reif-pvars C-φ› show False by auto
      qed
      show ReifT ∉ circuit-reif-pvars (primed-circuit C-φ)
      proof
        assume ReifT ∈ circuit-reif-pvars (primed-circuit C-φ)
      then obtain w where w ∈ circuit-reif-pvars C-φ and ReifT = primed-pvar-map
w
          using primed-reif-eq by auto
        hence w = ReifT
          by (cases w; simp add: primed-pvar-map-def)
        with extra-disj ‹w ∈ circuit-reif-pvars C-φ› show False by auto
      qed
    qed
```

```
    thus ?thesis
      using rho-trans-ReifT
      by (simp add: unit-clause-def satisfies-def eval-lit-def not-reif)
  qed
  have out-orig-reif:
    pvar-of-lit (snd (orig-circuit C-φ)) ∈ circuit-reif-pvars (orig-circuit C-φ)
  proof −
    have pvar-of-lit (snd C-φ) ∈ circuit-reif-pvars C-φ
      using wf-circuit-out-reif[OF wf] .
    then have map-pvar Original (pvar-of-lit (snd C-φ)) ∈
              map-pvar Original ‘ circuit-reif-pvars C-φ by blast
    moreover
    have pvar-of-lit (snd (orig-circuit C-φ)) = map-pvar Original (pvar-of-lit (snd
C-φ))
      by (simp add: orig-circuit-def lift-circuit-def Let-def split-beta
                    pvar-of-lit-map-literal-gen)
    ultimately show ?thesis using orig-reif-eq by simp
  qed

  have out-orig-not-primed:
    pvar-of-lit (snd (orig-circuit C-φ)) ∉ circuit-reif-pvars (primed-circuit C-φ)
  proof
    assume pvar-of-lit (snd (orig-circuit C-φ)) ∈ circuit-reif-pvars (primed-circuit
C-φ)
    then obtain w where w ∈ circuit-reif-pvars C-φ
      and eq: pvar-of-lit (snd (orig-circuit C-φ)) = primed-pvar-map w
      using primed-reif-eq by blast
    then have map-pvar Original (pvar-of-lit (snd C-φ)) = primed-pvar-map w
      by (simp add: orig-circuit-def lift-circuit-def pvar-of-lit-map-literal-gen
                    Let-def split-beta)
    thus False
      using ‹w ∈ circuit-reif-pvars C-φ› disjoint extra-disj
      by (cases pvar-of-lit (snd C-φ); cases w; auto simp: is-input-pvar-def)
  qed

  have comb-orig-out:
    satisfies (unit-clause (snd (orig-circuit C-φ))) ?rho-combined
  proof −
    let ?l = snd (orig-circuit C-φ)
    have sat-lift: satisfies (unit-clause ?l) (lift-to-var rho-circ)
    proof −
      have no-pcb-out: ∀ i. pvar-of-lit (snd C-φ) ≠ PrimedCostBit i
      proof
        fix i
        have pvar-of-lit (snd C-φ) ∈ circuit-reif-pvars C-φ
          by (rule wf-circuit-out-reif[OF wf])
        with extra-disj show pvar-of-lit (snd C-φ) ≠ PrimedCostBit i
          by blast
      qed
```

```
    have eval-lit ?l (lift-to-var rho-circ) = eval-lit (snd C-φ) rho-circ
      by (simp add: eval-lit-map-pvar-lift-to-var-orig[OF no-pcb-out])
    also have ... = 1 using rho-circ-out .
    finally show ?thesis
      by (auto simp: unit-clause-def satisfies-def)
  qed
    have ∀ v ∈ constraint-pvars (unit-clause ?l). ?rho-combined v = lift-to-var
rho-circ v
  proof
    fix v
    assume v ∈ constraint-pvars (unit-clause ?l)
    then have v = pvar-of-lit ?l
      by (auto simp: unit-clause-def constraint-pvars-def pvar-of-lit-def)
    thus ?rho-combined v = lift-to-var rho-circ v
      using out-orig-reif out-orig-not-primed by simp
  qed
  thus ?thesis using sat-lift satisfies-cong
    by fast
 qed

 have hyp-model:
  ∀ ψ ∈ encode-transition as V B ∪
        circuit-constraints (orig-circuit C-φ) ∪
        circuit-constraints (primed-circuit C-φ) ∪
        encode-cost-ge B ∪
        {unit-clause (snd (orig-circuit C-φ)), unit-clause (Pos ReifT)}. satisfies
ψ ?rho-combined
  using comb-trans comb-circ comb-primed-circ comb-cost-ge comb-orig-out comb-ReifT
  by (auto simp: models-def)

 have rho-comb-01: ∀ v. ?rho-combined v = 0 ∨ ?rho-combined v = 1
  using rho-circ-01 rho-circ'-01 two-state-rho-range unfolding lift-to-var-def
  by (auto 10 0 split: pvar.splits var.splits)
 have output-sat: satisfies (unit-clause (snd (primed-circuit C-φ))) ?rho-combined
  using cpr-sound[OF cpr-ind - rho-comb-01] hyp-model models-def by auto

 have out-reif': pvar-of-lit (snd (primed-circuit C-φ)) ∈ circuit-reif-pvars (primed-circuit
C-φ)
 proof −
  have pvar-of-lit (snd C-φ) ∈ circuit-reif-pvars C-φ
    using wf-circuit-out-reif[OF wf] .
  hence primed-pvar-map (pvar-of-lit (snd C-φ)) ∈
        primed-pvar-map ‘ circuit-reif-pvars C-φ by blast
  moreover
  have pvar-of-lit (snd (primed-circuit C-φ)) = primed-pvar-map (pvar-of-lit (snd
C-φ))
   by (simp add: primed-circuit-def lift-circuit-def Let-def split-beta pvar-of-lit-map-literal-gen)
  ultimately show ?thesis using primed-reif-eq by simp
 qed
```

```
  let ?rho-out = extend-rho C-φ rho-circ' ?base-s'

  have out-circ-sat: models (circuit-constraints C-φ) ?rho-out
    using models-circuit-constraints-extend[OF rho-circ'-sat circ-vars]
    by (simp add: rho-circ'-inp)

  have out-reif-val: eval-lit (snd C-φ) ?rho-out = 1
  proof −
    let ?pov = pvar-of-lit (snd (primed-circuit C-φ))
    let ?ov  = pvar-of-lit (snd C-φ)
    have pov-primed: ?pov ∈ circuit-reif-pvars (primed-circuit C-φ) using out-reif'
.
    have pov-not-orig: ?pov ∉ circuit-reif-pvars (orig-circuit C-φ)
    proof
      assume ?pov ∈ circuit-reif-pvars (orig-circuit C-φ)
      then obtain w where w ∈ circuit-reif-pvars C-φ ?pov = map-pvar Original
w
        using orig-reif-eq by blast
      then have primed-pvar-map (pvar-of-lit (snd C-φ)) = map-pvar Original w
        by (simp add: lift-circuit-def primed-circuit-def prod.case-eq-if
            pvar-of-lit-map-literal-gen)
      thus False
        using ‹w ∈ circuit-reif-pvars C-φ› disjoint extra-disj
        by (cases pvar-of-lit (snd C-φ); cases w; auto)
    qed
    have pov-eq: ?pov = primed-pvar-map ?ov
      by (simp add: lift-circuit-def primed-circuit-def prod.case-eq-if
          pvar-of-lit-map-literal-gen)
    have comb-pov: ?rho-combined ?pov = rho-circ' ?ov
    proof −
      have ?rho-combined ?pov = lift-to-var rho-circ' ?pov
        using pov-primed pov-not-orig by simp
      also have lift-to-var rho-circ' ?pov = rho-circ' ?ov
        by (metis disjoint is-input-pvar-def lift-to-var-primed-pvar-map wf pov-eq
            wf-circuit-out-reif)
      finally show ?thesis .
    qed
    have rho-out-ov: ?rho-out ?ov = rho-circ' ?ov
      by (simp add: extend-rho-def wf wf-circuit-out-reif)
    have eval-eq: eval-lit (snd (primed-circuit C-φ)) ?rho-combined =
        eval-lit (snd C-φ) ?rho-out
      using comb-pov rho-out-ov
      unfolding primed-circuit-def lift-circuit-def Let-def pvar-of-lit-def eval-lit-def
      by (auto split: prod.splits literal.splits simp: map-pvar-def)
   have eval-eq': eval-lit (snd (primed-circuit C-φ)) ?rho-combined = eval-lit (snd
C-φ) rho-circ'
    proof −
      have eval-lit (snd (primed-circuit C-φ)) ?rho-combined = eval-lit (snd C-φ)
```

```
?rho-out
      by (rule eval-eq)
    also have eval-lit (snd C-φ) ?rho-out = eval-lit (snd C-φ) rho-circ'
    proof (cases snd C-φ)
      case (Pos v)
      then show ?thesis using rho-out-ov
        by (simp add: eval-lit-def pvar-of-lit-def)
    next
      case (Neg v)
      then show ?thesis using rho-out-ov
        by (simp add: eval-lit-def pvar-of-lit-def)
    qed
    finally show ?thesis .
  qed
  have eval-lit (snd (primed-circuit C-φ)) ?rho-combined = 1
  proof −
    from output-sat have eval-lit (snd (primed-circuit C-φ)) ?rho-combined ≥ 1
      by (auto simp: unit-clause-def satisfies-def)
    moreover have eval-lit (snd C-φ) rho-circ' ≤ 1
    proof (cases snd C-φ)
      case (Pos v)
      have rho-circ' v = 0 ∨ rho-circ' v = 1 using rho-circ'-01 by auto
      then show ?thesis by (auto simp: Pos eval-lit-def)
    next
      case (Neg v)
      have rho-circ' v = 0 ∨ rho-circ' v = 1 using rho-circ'-01 by auto
      then show ?thesis by (auto simp: Neg eval-lit-def)
    qed
    ultimately show ?thesis using eval-eq' by linarith
  qed
  with eval-eq show ?thesis by simp
qed

have out-sv: ∀ v. ?rho-out (StateVar v) = (if v ∈ ?s' then 1 else 0)
proof
  fix v
  have StateVar v ∉ circuit-reif-pvars C-φ
    using disjoint by (auto simp: is-input-pvar-def)
  thus ?rho-out (StateVar v) = (if v ∈ ?s' then 1 else 0)
    by (simp add: extend-rho-base-on-non-reif state-rho-def)
qed
have out-cb: ∀ i. ?rho-out (CostBit i) = (?c' div 2^i) mod 2
proof
  fix i
  have CostBit i ∉ circuit-reif-pvars C-φ
    using disjoint by (auto simp: is-input-pvar-def)
  thus ?rho-out (CostBit i) = (?c' div 2^i) mod 2
    by (simp add: extend-rho-base-on-non-reif state-rho-def)
qed
```

```
  have out-pcb: ∀ i. ?rho-out (PrimedCostBit i) = 0
  proof
    fix i
    have PrimedCostBit i ∉ circuit-reif-pvars C-φ
      using disjoint by (auto simp: is-input-pvar-def)
    thus ?rho-out (PrimedCostBit i) = 0
      by (simp add: extend-rho-base-on-non-reif state-rho-def)
  qed

  show ?thesis
    unfolding in-M-def
    using out-circ-sat out-reif-val out-sv out-cb out-pcb
    by (smt (verit, best) base-s'-01 extend-rho-def rho-circ'-01)
qed

lemma in-M-path:
  fixes Π :: 'v::linorder strips-task and C-φ :: 'v pb-circuit
  assumes base: in-M C-φ Π s0 0
    and path-p: path (acts Π) s0 sf π
    and wf: wf-circuit C-φ
    and circ-vars: ∀ (r, φ) ∈ set (fst C-φ). ∀ v ∈ constraint-pvars φ. is-input-pvar
v ∨ v ∈ circuit-reif-pvars C-φ
   and no-pcb: ∀ (r, φ) ∈ set (fst C-φ). ∀ v ∈ constraint-pvars φ. ∀ i. v ≠ Primed-
CostBit i
   and disjoint: ∀ v ∈ circuit-reif-pvars C-φ. ¬ is-input-pvar v ∧ v ≠ ReifI ∧ (∀ k.
v ≠ ReifCostGe k)
    and extra-disj: ∀ v ∈ circuit-reif-pvars C-φ.
        (∀ k. v ≠ ReifDeltaCost k) ∧ (∀ k. v ≠ ReifDeltaCostLower k) ∧
        (∀ k. v ≠ ReifDeltaCostUpper k) ∧
        (∀ k. v ≠ ReifPrimedCostGe k) ∧ v ≠ ReifT ∧
        (∀ u. v ≠ ReifGeq u) ∧ (∀ u. v ≠ ReifLeq u) ∧ (∀ u. v ≠ ReifEq u) ∧ (∀ i. v
≠ ReifAction i) ∧
        (∀ i. v ≠ PrimedCostBit i) ∧ v ≠ ReifG
    and realiz: ∀ (base :: 'v pvar ⇒ nat). (∀ v. base v = 0 ∨ base v = 1) ⟶ (∃
rho.
        models (circuit-constraints C-φ) rho
        ∧ (∀ v. is-input-pvar v ⟶ rho v = base v)
        ∧ (∀ v. rho v = 0 ∨ rho v = 1))
    and fin-V: finite V
    and cost-bound: plan-cost π < B
    and pre-sub: ∀ a ∈ set as. pre a ⊆ V
    and add-sub: ∀ a ∈ set as. add a ⊆ V
    and del-sub: ∀ a ∈ set as. del a ⊆ V
    and acts-sub-as: acts Π ⊆ set as
    and cpr-ind: cpr-derives
      (encode-transition as V B ∪ circuit-constraints (orig-circuit C-φ) ∪
       circuit-constraints (primed-circuit C-φ) ∪ encode-cost-ge B ∪
       {unit-clause (snd (orig-circuit C-φ)), unit-clause (Pos ReifT)})
      (unit-clause (snd (primed-circuit C-φ)))
```

```
  shows in-M C-φ Π sf (plan-cost π)
proof −
  have strengthen: ⋀s c sf'. ⟦in-M C-φ Π s c; path (acts Π) s sf' π; c + plan-cost
π < B⟧
    ⟹ in-M C-φ Π sf' (c + plan-cost π)
  proof (induction π)
    case Nil
    from Nil.prems(2) have sf' = s by (cases rule: path.cases) auto
    then show ?case using Nil(1) by (simp add: plan-cost-def)
  next
    case (Cons a π')
    from Cons.prems(2) have applicable a s
      by (cases rule: path.cases) auto
    from Cons.prems(2) have path (acts Π) (successor a s) sf' π'
      by (cases rule: path.cases) auto
    have a-in-acts: a ∈ acts Π
      using Cons.prems(2) by (cases rule: path.cases) auto
    have a-in-as: a ∈ set as
      using a-in-acts acts-sub-as by auto
    have pre-a-V: pre a ⊆ V using pre-sub a-in-as by auto
    have add-a-V: add a ⊆ V using add-sub a-in-as by auto
    have del-a-V: del a ⊆ V using del-sub a-in-as by auto
    have c-lt-B: c + cost a < B
    proof −
      have (c + cost a) + plan-cost π' < B
        using Cons.prems(3) by (simp add: plan-cost-def add-ac)
      then show ?thesis using add-lessD1 by blast
    qed
    have cost-prog: (c + cost a) + plan-cost π' < B
      using Cons.prems(3) by (simp add: plan-cost-def add-ac)
    have step: in-M C-φ Π (successor a s) (c + cost a)
      using in-M-step[OF Cons.prems(1) ‹applicable a s› a-in-acts acts-sub-as
                        wf circ-vars no-pcb disjoint extra-disj realiz fin-V
                        pre-sub add-sub del-sub c-lt-B cpr-ind]
      by blast
    show ?case
      using Cons.IH[OF step ‹path (acts Π) (successor a s) sf' π'› cost-prog]
      by (simp add: plan-cost-def add-ac)
  qed
  show ?thesis
    using strengthen[OF base path-p] cost-bound by simp
qed

theorem theorem-1-from-cpr:
  fixes Π :: 'v::linorder strips-task
  assumes cert: certificate-valid-cpr B Π Cert
    and plan: is-plan-for Π π
  shows plan-cost π ≥ B
proof −
```

let *?as* = *cert-actions Cert*
let *?C-φ* = *cert-circuit Cert*
let *?r-φ* = *snd ?C-φ*
from *cert* **have** *fin*: *finite* (*vars* Π)
unfolding *certificate-valid-cpr-def Let-def* **by** *auto*
from *cert* **have** *init-sub*: *init* Π ⊆ *vars* Π
unfolding *certificate-valid-cpr-def Let-def* **by** *auto*
from *cert* **have** *goal-sub*: *goal* Π ⊆ *vars* Π
unfolding *certificate-valid-cpr-def Let-def* **by** *auto*
from *cert* **have** *fin-acts*: *finite* (*acts* Π)
unfolding *certificate-valid-cpr-def Let-def* **by** *auto*
from *cert* **have** *as-cover*: *acts* Π ⊆ *set ?as*
unfolding *certificate-valid-cpr-def Let-def* **by** *auto*
from *cert* **have** *act-sub*:
∀ *a* ∈ *set ?as*. *pre a* ⊆ *vars* Π ∧ *add a* ⊆ *vars* Π ∧ *del a* ⊆ *vars* Π
unfolding *certificate-valid-cpr-def Let-def* **by** *auto*
from *cert* **have** *wf*: *wf-circuit ?C-φ*
unfolding *certificate-valid-cpr-def Let-def* **by** *auto*
from *cert* **have** *disjoint-full*:
∀ *v* ∈ *circuit-reif-pvars ?C-φ*.
¬ *is-input-pvar v* ∧ *v* ≠ *ReifI* ∧ (∀ *k*. *v* ≠ *ReifCostGe k*) ∧ *v* ≠ *ReifG* ∧
(∀ *k*. *v* ≠ *ReifDeltaCost k*) ∧ (∀ *k*. *v* ≠ *ReifDeltaCostLower k*) ∧
(∀ *k*. *v* ≠ *ReifDeltaCostUpper k*) ∧
(∀ *k*. *v* ≠ *ReifPrimedCostGe k*) ∧ *v* ≠ *ReifT* ∧
(∀ *u*. *v* ≠ *ReifGeq u*) ∧ (∀ *u*. *v* ≠ *ReifLeq u*) ∧ (∀ *u*. *v* ≠ *ReifEq u*) ∧
(∀ *i*. *v* ≠ *ReifAction i*) ∧ (∀ *i*. *v* ≠ *PrimedCostBit i*)
unfolding *certificate-valid-cpr-def Let-def* **by** *auto*
from *cert* **have** *distinct-reif*: *distinct-reif-vars ?C-φ*
unfolding *certificate-valid-cpr-def Let-def* **by** *auto*
have *reif-not-input*: ∀ *v* ∈ *circuit-reif-pvars ?C-φ*. ¬ *is-input-pvar v*
using *disjoint-full* **unfolding** *circuit-reif-pvars-def* **by** *auto*
note *wf-circuit-realizability*[*OF wf distinct-reif reif-not-input*]
then have *realiz'*: ∀ (*base* :: *'v pvar* ⇒ *nat*). (∀ *v*. *base v* = *0* ∨ *base v* = *1*) ⟶
(∃ *rho*.
models (*circuit-constraints ?C-φ*) *rho*
∧ (∀ *v*. *is-input-pvar v* ⟶ *rho v* = *base v*)
∧ (∀ *v*. *rho v* = *0* ∨ *rho v* = *1*))
by *blast*
from *cert* **have** *cpr-init*: *cpr-derives*
(*encode-init* Π ∪ *circuit-constraints ?C-φ* ∪ *encode-cost-ge B* ∪
{*unit-clause* (*Pos ReifI*), *neg-cost-ge-one B*})
(*unit-clause* (*snd ?C-φ*))
unfolding *certificate-valid-cpr-def Let-def* **by** *auto*
from *cert* **have** *cpr-goal*: *cpr-derives*
(*encode-goal* Π ∪ *circuit-constraints ?C-φ* ∪ *encode-cost-ge B* ∪
{*unit-clause* (*snd ?C-φ*), *unit-clause* (*Pos ReifG*)})
(*cost-ge-constraint B*)
unfolding *certificate-valid-cpr-def Let-def* **by** *auto*
from *cert* **have** *cpr-ind*: *cpr-derives*

(*encode-transition ?as* (*vars* Π) *B* ∪ *circuit-constraints* (*orig-circuit ?C-φ*) ∪
circuit-constraints (*primed-circuit ?C-φ*) ∪ *encode-cost-ge B* ∪
{*unit-clause* (*snd* (*orig-circuit ?C-φ*)), *unit-clause* (*Pos ReifT*)})
(*unit-clause* (*snd* (*primed-circuit ?C-φ*)))
unfolding *certificate-valid-cpr-def Let-def* **by** *auto*
have *disjoint*: ∀ *v* ∈ *circuit-reif-pvars ?C-φ*. ¬ *is-input-pvar v* ∧ *v* ≠ *ReifI* ∧ (∀ *k*. *v* ≠ *ReifCostGe k*)
using *disjoint-full* **unfolding** *circuit-reif-pvars-def* **by** *auto*
have *extra-disj*: ∀ *v* ∈ *circuit-reif-pvars ?C-φ*.
(∀ *k*. *v* ≠ *ReifDeltaCost k*) ∧ (∀ *k*. *v* ≠ *ReifDeltaCostLower k*) ∧
(∀ *k*. *v* ≠ *ReifDeltaCostUpper k*) ∧
(∀ *k*. *v* ≠ *ReifPrimedCostGe k*) ∧ *v* ≠ *ReifT* ∧
(∀ *u*. *v* ≠ *ReifGeq u*) ∧ (∀ *u*. *v* ≠ *ReifLeq u*) ∧ (∀ *u*. *v* ≠ *ReifEq u*) ∧ (∀ *i*. *v* ≠ *ReifAction i*) ∧
(∀ *i*. *v* ≠ *PrimedCostBit i*) ∧ *v* ≠ *ReifG*
using *disjoint-full* **unfolding** *circuit-reif-pvars-def* **by** *auto*
have *disjointG*: *ReifG* ∉ *circuit-reif-pvars ?C-φ*
using *disjoint-full* **unfolding** *circuit-reif-pvars-def* **by** *auto*
have *circ-vars*: ∀ (*r*, *φ*) ∈ *set* (*fst ?C-φ*).
∀ *v* ∈ *constraint-pvars φ*. *is-input-pvar v* ∨ *v* ∈ *circuit-reif-pvars ?C-φ*
proof −
{
fix *r φ v*
assume *rin*: (*r*, *φ*) ∈ *set* (*fst ?C-φ*) **and** *vin*: *v* ∈ *constraint-pvars φ*
obtain *pairs out* **where** *cphi*: *?C-φ* = (*pairs*, *out*) **by** (*cases ?C-φ*)
have *rin′*: (*r*, *φ*) ∈ *set pairs* **using** *rin cphi* **by** *simp*
from *rin′* **obtain** *i* **where** *i-lt*: *i* < *length pairs* **and** *pairs-i*: *pairs* ! *i* = (*r*, *φ*)
using *in-set-conv-nth* **by** *metis*
from *wf*[*unfolded cphi wf-circuit-def Let-def*, *simplified*, *THEN conjunct1*, *rule-format*, *OF i-lt*]
have *sub*: *constraint-pvars φ* − {*pvar-of-lit r*} ⊆ *Collect is-input-pvar* ∪ *pvar-of-lit ‘ fst ‘ set* (*take i pairs*)
using *pairs-i* **by** (*auto simp add*: *split-beta constraint-pvars-def*)
have *rv-in*: *pvar-of-lit r* ∈ *circuit-reif-pvars ?C-φ*
unfolding *circuit-reif-pvars-def cphi* **using** *rin′* **by** (*auto simp*: *image-def*)
have *take-sub*: *pvar-of-lit ‘ fst ‘ set* (*take i pairs*) ⊆ *circuit-reif-pvars ?C-φ*
unfolding *circuit-reif-pvars-def cphi* **using** *set-take-subset*[*of i pairs*] **by** *auto*
have *is-input-pvar v* ∨ *v* ∈ *circuit-reif-pvars ?C-φ*
proof (*cases v* = *pvar-of-lit r*)
case *True* **thus** *?thesis* **using** *rv-in* **by** *auto*
next
case *False*
then have *v* ∈ *Collect is-input-pvar* ∪ *pvar-of-lit ‘ fst ‘ set* (*take i pairs*)
using *sub vin* **by** *auto*
thus *?thesis* **using** *take-sub* **by** *auto*
qed
}

then show *?thesis* **by** (*auto simp*: *Ball-def split*: *prod.splits*)
qed
have *no-pcb*: $\forall (r, \varphi) \in$ *set* (*fst ?C-*φ). $\forall\, v \in$ *constraint-pvars* φ. $\forall\, i$. $v \neq$ *Primed-CostBit i*
using *cert* **unfolding** *certificate-valid-cpr-def Let-def* **by** *auto*

from *plan* **obtain** *sf* **where**
path-p: *path* (*acts* Π) (*init* Π) *sf* π
and *goal-sf*: *is-goal-state* Π *sf*
unfolding *is-plan-for-def* **by** *blast*
have *pre-sub*: $\forall\ a \in$ *set ?as*. *pre a* $\subseteq$ *vars* Π **using** *act-sub* **by** *auto*
have *add-sub*: $\forall\ a \in$ *set ?as*. *add a* $\subseteq$ *vars* Π **using** *act-sub* **by** *auto*
have *del-sub*: $\forall\ a \in$ *set ?as*. *del a* $\subseteq$ *vars* Π **using** *act-sub* **by** *auto*

show *plan-cost* $\pi \geq B$
proof (*cases* $B = 0$)
case *True* **thus** *?thesis* **by** *simp*
next
case *False*
then have *B-pos*: $B \geq 1$ **by** *simp*
show *plan-cost* $\pi \geq B$
proof (*rule ccontr*)
assume $\neg$ *plan-cost* $\pi \geq B$
then have *cost-lt-B*: *plan-cost* $\pi < B$ **by** (*simp add*: *not-le*)
have *in-M-0*: *in-M ?C-*φ Π (*init* Π) *0*
using *in-M-init*[*OF fin init-sub wf wf-circuit-out-reif*[*OF wf*] *circ-vars disjoint realiz' B-pos cpr-init*]
.
have *in-M-final*: *in-M ?C-*φ Π *sf* (*plan-cost* π)
using *in-M-path*[*OF in-M-0 path-p wf circ-vars no-pcb disjoint extra-disj realiz' fin cost-lt-B pre-sub add-sub del-sub as-cover cpr-ind*]
.
have *plan-cost* $\pi \geq B$
using *in-M-goal-bound*[*OF in-M-final wf circ-vars disjoint disjointG cpr-goal goal-sf fin goal-sub*]
.
thus *False* **using** *cost-lt-B* **by** *simp*
qed
qed
qed

end

3 Encoding Semantics

theory *Encoding-Semantics*
imports *Lower-Bound-Certificates*

begin

Shared toolkit for formalizing the remainder of arXiv:2504.18443: semantic analysis of 0/1 models of the PB encoding (converse direction of the existing $*$-*sound* lemmas), the bridge between semantic 0/1 implication and *cpr-derives*, and the task embedding into an extended variable type that provides unboundedly many fresh circuit gate names.

3.1 Basic facts about 0/1 assignments

lemma *eval-lit-le-one*:
assumes $\forall$ *v*. *rho v* = *0* $\vee$ *rho v* = *1*
shows *eval-lit l rho* $\leq$ *1*
proof (*cases l*)
case (*Pos v*)
then show *?thesis* **using** *assms*[*rule-format*, *of v*] **by** (*auto simp*: *eval-lit-def*)
next
case (*Neg v*)
then show *?thesis* **by** (*simp add*: *eval-lit-def*)
qed

lemma *eval-lit-01*:
assumes $\forall$ *v*. *rho v* = *0* $\vee$ *rho v* = *1*
shows *eval-lit l rho* = *0* $\vee$ *eval-lit l rho* = *1*
proof (*cases l*)
case (*Pos v*)
then show *?thesis* **using** *assms*[*rule-format*, *of v*] **by** (*auto simp*: *eval-lit-def*)
next
case (*Neg v*)
then show *?thesis* **using** *assms*[*rule-format*, *of v*] **by** (*auto simp*: *eval-lit-def*)
qed

lemma *eval-lit-neg-conv*:
assumes $\forall$ *v*. *rho v* = *0* $\vee$ *rho v* = *1*
shows *eval-lit* (*lit-neg l*) *rho* = *1* $-$ *eval-lit l rho*
proof (*cases l*)
case (*Pos v*)
then show *?thesis* **by** (*simp add*: *eval-lit-def lit-neg-def*)
next
case (*Neg v*)
then show *?thesis* **using** *assms*[*rule-format*, *of v*] **by** (*auto simp*: *eval-lit-def lit-neg-def*)
qed

lemma *eval-lit-neg-sum-one*:
assumes $\forall$ *v*. *rho v* = *0* $\vee$ *rho v* = *1*
shows *eval-lit l rho* + *eval-lit* (*lit-neg l*) *rho* = *1*
proof (*cases l*)
case (*Pos v*)

then show *?thesis* **using** *assms*[*rule-format*, *of v*] **by** (*auto simp*: *eval-lit-def lit-neg-def*)
next
case (*Neg v*)
then show *?thesis* **using** *assms*[*rule-format*, *of v*] **by** (*auto simp*: *eval-lit-def lit-neg-def*)
qed

lemma *pb-sum-le-weight*:
assumes $\forall$ *v*. *rho v* = *0* $\vee$ *rho v* = *1*
shows *pb-sum coeffs rho* $\leq$ *sum-list* (*map fst coeffs*)
proof (*induction coeffs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons p coeffs*)
obtain *a l* **where** *p*: *p* = (*a*, *l*) **by** (*cases p*)
have *a* $*$ *eval-lit l rho* $\leq$ *a* $*$ *1*
by (*rule mult-le-mono2*[*OF eval-lit-le-one*[*OF assms*]])
then have *head*: *a* $*$ *eval-lit l rho* $\leq$ *a* **by** *simp*
have *a* $*$ *eval-lit l rho* + *pb-sum coeffs rho* $\leq$ *a* + *sum-list* (*map fst coeffs*)
using *head Cons.IH* **by** (*rule add-mono*)
then show *?case* **by** (*simp add*: *p*)
qed

lemma *pb-sum-unit-list*:
pb-sum (*map* (λ*v*. (*1*, *lt v*)) *xs*) *rho* = ($\sum$ *v*$\leftarrow$*xs*. *eval-lit* (*lt v*) *rho*)
by (*induction xs*) *auto*

lemma *pb-sum-unit-set*:
assumes *finite S*
shows *pb-sum* (*map* (λ*v*. (*1*, *lt v*)) (*sorted-list-of-set S*)) *rho*
= ($\sum$ *v*$\in$*S*. *eval-lit* (*lt v*) *rho*)
proof −
have *pb-sum* (*map* (λ*v*. (*1*, *lt v*)) (*sorted-list-of-set S*)) *rho*
= ($\sum$ *v*$\leftarrow$*sorted-list-of-set S*. *eval-lit* (*lt v*) *rho*)
by (*rule pb-sum-unit-list*)
also have ... = ($\sum$ *v*$\in$*set* (*sorted-list-of-set S*). *eval-lit* (*lt v*) *rho*)
by (*rule sum-list-distinct-conv-sum-set*) *simp*
also have *set* (*sorted-list-of-set S*) = *S* **using** *assms* **by** *simp*
finally show *?thesis* **.**
qed

lemma *sum-units-all-one*:
fixes *f* :: *'a* $\Rightarrow$ *nat*
assumes *fin*: *finite S*
and *total*: ($\sum$ *v*$\in$*S*. *f v*) $\geq$ *card S*
and *bnd*: $\forall$ *v*$\in$*S*. *f v* $\leq$ *1*
shows $\forall$ *v*$\in$*S*. *f v* = *1*

```
proof (rule ccontr)
  assume ¬ (∀ v∈S. f v = 1)
  then obtain v0 where v0: v0 ∈ S and f v0 ≠ 1 by blast
  with bnd have f-v0: f v0 = 0 by fastforce
  have cardpos: 0 < card S using fin v0 by (auto simp: card-gt-0-iff)
  have bound-rest: (∑ v∈S − {v0}. f v) ≤ card (S − {v0})
    using sum-bounded-above[of S − {v0} f 1] bnd by simp
  have (∑ v∈S. f v) = f v0 + (∑ v∈S − {v0}. f v)
    using fin v0 by (simp add: sum.remove)
  also have ... ≤ 0 + card (S − {v0})
    using f-v0 bound-rest by simp
  also have ... = card S − 1
    using fin v0 by (simp add: card-Diff-singleton)
  also have ... < card S
    using cardpos by linarith
  finally show False using total by simp
qed

lemma pb-sum-pos-ex:
  assumes pb-sum coeffs rho ≥ 1
  shows ∃(a, l) ∈ set coeffs. a * eval-lit l rho ≥ 1
  using assms
proof (induction coeffs)
  case Nil
  then show ?case by simp
next
  case (Cons p coeffs)
  obtain a l where p: p = (a, l) by (cases p)
  show ?case
  proof (cases a * eval-lit l rho ≥ 1)
    case True
    then show ?thesis by (auto simp: p)
  next
    case False
    then have zero: a * eval-lit l rho = 0 by linarith
    have pb-sum coeffs rho ≥ 1 using Cons.prems by (simp add: p zero)
    from Cons.IH[OF this] show ?thesis by (auto simp: p)
  qed
qed
```

3.2 Semantic implication and CPR derivability

Because the formal CPR system contains the semantic *unsat-01* (*?CC* ∪ {*constraint-neg ?C*}) ⟹ *cpr-derives ?CC ?C* rule, derivability of a constraint with positive threshold is *equivalent* to semantic implication over 0/1 assignments. All "it is possible to derive" lemmas of the paper are proved through this bridge.

```
lemma implies01-unsat-neg:
```

assumes *impl*: $\forall$ *rho*. ($\forall$ *v*. *rho v* = *0* $\vee$ *rho v* = *1*) $\longrightarrow$ *models CC rho* $\longrightarrow$ *satisfies C rho*
and *A-pos*: *snd C* $\geq$ (*1*::*nat*)
shows *unsat-01* (*CC* $\cup$ {*constraint-neg C*})
proof (*rule ccontr*)
assume $\neg$ *unsat-01* (*CC* $\cup$ {*constraint-neg C*})
then obtain *rho* **where** *rho01*: $\forall$ *v*. *rho v* = *0* $\vee$ *rho v* = *1*
and *m*: *models* (*CC* $\cup$ {*constraint-neg C*}) *rho*
unfolding *unsat-01-def* **by** *blast*
obtain *coeffs A* **where** *C*: *C* = (*coeffs*, *A*) **by** (*cases C*)
have *A1*: *A* $\geq$ *1* **using** *A-pos C* **by** *simp*
have *mCC*: *models CC rho* **using** *m* **by** (*simp add*: *models-def*)
have *satC*: *satisfies C rho* **using** *impl rho01 mCC* **by** *blast*
have *pos*: *pb-sum coeffs rho* $\geq$ *A* **using** *satC* **by** (*simp add*: *C satisfies-def*)
let *?neg* = *map* (λ(*a*, *l*). (*a*, *lit-neg l*)) *coeffs*
let *?M* = *sum-list* (*map fst coeffs*)
have *satN*: *satisfies* (*constraint-neg C*) *rho* **using** *m* **by** (*simp add*: *models-def*)
have *neg*: *pb-sum ?neg rho* $\geq$ *?M* $-$ (*A* $-$ *1*)
using *satN* **by** (*simp add*: *constraint-neg-def C satisfies-def Let-def*)
have *sum-eq*: *pb-sum coeffs rho* + *pb-sum ?neg rho* = *?M*
by (*rule pb-sum-add-negated-gen*[*OF rho01*])
show *False*
proof (*cases A* $\leq$ *?M*)
case *True*
have *?M* $-$ (*A* $-$ *1*) = *?M* $-$ *A* + *1*
using *A1 True* **by** *simp*
then have *A* + (*?M* $-$ *A* + *1*) $\leq$ *pb-sum coeffs rho* + *pb-sum ?neg rho*
using *pos neg* **by** (*intro add-mono*) *auto*
then have *?M* + *1* $\leq$ *?M* **using** *sum-eq True* **by** *simp*
then show *False* **by** *simp*
next
case *False*
have *pb-sum coeffs rho* $\leq$ *?M* **by** (*rule pb-sum-le-weight*[*OF rho01*])
with *pos False* **show** *False* **by** *simp*
qed
qed

lemma *semantic-to-cpr*:
assumes $\forall$ *rho*. ($\forall$ *v*. *rho v* = *0* $\vee$ *rho v* = *1*) $\longrightarrow$ *models CC rho* $\longrightarrow$ *satisfies C rho*
and *snd C* $\geq$ (*1*::*nat*)
shows *cpr-derives CC C*
by (*rule cpr-derives.rup*[*OF implies01-unsat-neg*[*OF assms*]])

lemma *cpr-derives-iff-semantic*:
assumes *snd C* $\geq$ (*1*::*nat*)
shows *cpr-derives CC C* $\longleftrightarrow$
($\forall$ *rho*. ($\forall$ *v*. *rho v* = *0* $\vee$ *rho v* = *1*) $\longrightarrow$ *models CC rho* $\longrightarrow$ *satisfies C rho*)
using *cpr-sound semantic-to-cpr*[*OF - assms*] **by** *blast*

3.3 Reification semantics

In any 0/1 model of a reification pair, the gate literal carries exactly the truth value of the reified body the semantic core of every RUP argument in the paper.

```
lemma reification-forces:
  assumes rho01: ∀ v. rho v = 0 ∨ rho v = 1
    and m: models (reification r coeffs A) rho
  shows eval-lit r rho = 1 ⟷ pb-sum coeffs rho ≥ A
proof
  assume out: eval-lit r rho = 1
  have fwd: satisfies (reif-fwd r coeffs A) rho
    using m by (simp add: models-def reification-def)
  have negr: eval-lit (lit-neg r) rho = 0
    using eval-lit-neg-conv[OF rho01, of r] out by simp
  show pb-sum coeffs rho ≥ A
    using fwd by (simp add: reif-fwd-def satisfies-def negr)
next
  assume body: pb-sum coeffs rho ≥ A
  let ?M = sum-list (map fst coeffs)
  let ?neg = map (λ(a, l). (a, lit-neg l)) coeffs
  have bwd: satisfies (reif-bwd r coeffs A) rho
    using m by (simp add: models-def reification-def)
  have bwd': (?M + 1 − A) ∗ eval-lit r rho + pb-sum ?neg rho ≥ ?M + 1 − A
    using bwd by (simp add: reif-bwd-def satisfies-def Let-def)
  have sum-eq: pb-sum coeffs rho + pb-sum ?neg rho = ?M
    by (rule pb-sum-add-negated-gen[OF rho01])
  show eval-lit r rho = 1
  proof (rule ccontr)
    assume eval-lit r rho ≠ 1
    with eval-lit-01[OF rho01, of r] have r0: eval-lit r rho = 0 by auto
    have negsum: pb-sum ?neg rho ≥ ?M + 1 − A
      using bwd' by (simp add: r0)
    have pos-le: pb-sum coeffs rho ≤ ?M
      using sum-eq by linarith
    show False
    proof (cases A ≤ ?M)
      case True
      have A + (?M + 1 − A) ≤ pb-sum coeffs rho + pb-sum ?neg rho
        using body negsum by (rule add-mono)
      also have ... = ?M by (rule sum-eq)
      finally show False using True by simp
    next
      case False
      then show False using body pos-le by simp
    qed
  qed
qed
```

3.4 Binary cost values

The numeric value carried by a block of cost bits in an assignment. *cb* selects the bit variables (e.g. *CostBit* or *PrimedCostBit*).

definition *bits-val* :: *nat* ⇒ (*nat* ⇒ *'w*) ⇒ (*'w* ⇒ *nat*) ⇒ *nat* **where**
bits-val k cb rho ≡ ∑ *i*<*k*. *2^i* ∗ *rho* (*cb i*)

lemma *bits-val-Suc*: *bits-val* (*Suc k*) *cb rho* = *bits-val k cb rho* + *2^k* ∗ *rho* (*cb k*)
by (*simp add*: *bits-val-def*)

lemma *pb-sum-bits-val*:
pb-sum (*map* (λ*i*. (*2^i*, *Pos* (*cb i*))) [*0*..<*k*]) *rho* = *bits-val k cb rho*
proof (*induction k*)
case *0*
then show *?case* **by** (*simp add*: *bits-val-def*)
next
case (*Suc k*)
have *pb-sum* (*map* (λ*i*. (*2^i*, *Pos* (*cb i*))) [*0*..<*Suc k*]) *rho*
= *pb-sum* (*map* (λ*i*. (*2^i*, *Pos* (*cb i*))) [*0*..<*k*]) *rho* + *2^k* ∗ *rho* (*cb k*)
by (*simp add*: *pb-sum-append eval-lit-def*)
then show *?case* **using** *Suc.IH* **by** (*simp add*: *bits-val-Suc*)
qed

lemma *bits-val-lt*:
assumes ∀ *i*. *rho* (*cb i*) = *0* ∨ *rho* (*cb i*) = *1*
shows *bits-val k cb rho* < *2^k*
proof (*induction k*)
case *0*
then show *?case* **by** (*simp add*: *bits-val-def*)
next
case (*Suc k*)
have *rho* (*cb k*) ≤ *1* **using** *assms*[*rule-format*, *of k*] **by** *auto*
then have *bnd*: *2^k* ∗ *rho* (*cb k*) ≤ *2^k* **by** *simp*
have *bits-val* (*Suc k*) *cb rho* < *2^k* + *2^k*
unfolding *bits-val-Suc* **by** (*rule add-less-le-mono*[*OF Suc.IH bnd*])
then show *?case* **by** *simp*
qed

lemma *bits-val-eq-of-binary*:
assumes ∀ *i* < *k*. *rho* (*cb i*) = (*c div 2^i*) *mod 2*
shows *bits-val k cb rho* = *c mod 2^k*
using *assms*
proof (*induction k*)
case *0*
then show *?case* **by** (*simp add*: *bits-val-def*)
next
case (*Suc k*)
have *IH*: *bits-val k cb rho* = *c mod 2^k* **using** *Suc.IH Suc.prems* **by** *simp*
have *bk*: *rho* (*cb k*) = (*c div 2^k*) *mod 2* **using** *Suc.prems* **by** *simp*

```
  have bits-val (Suc k) cb rho = c mod 2^k + 2^k * ((c div 2^k) mod 2)
    by (simp add: bits-val-Suc IH bk)
  also have ... = c mod 2^(Suc k)
    by (metis add.commute mod-mult2-eq mult.commute power-Suc)
  finally show ?case .
qed

lemma pb-sum-neg-bits-val:
  assumes rho01: ∀ v. rho v = 0 ∨ rho v = 1
  shows pb-sum (map (λi. (2^i, Neg (cb i))) [0..<k]) rho
      = (2^k − 1) − bits-val k cb rho
proof −
  let ?pos = map (λi. (2^i, Pos (cb i))) [0..<k]
  let ?neg = map (λi. (2^i, Neg (cb i))) [0..<k]
  have neg-eq: map (λ(a, l). (a, lit-neg l)) ?pos = ?neg
    by (simp add: lit-neg-def)
  have fst-eq: map fst ?pos = map ((^) 2) [0..<k]
    by simp
  have sum01: pb-sum ?pos rho + pb-sum (map (λ(a, l). (a, lit-neg l)) ?pos) rho
      = sum-list (map fst ?pos)
    by (rule pb-sum-add-negated-gen[OF rho01])
  have key: bits-val k cb rho + pb-sum ?neg rho = 2^k − 1
    using sum01 unfolding neg-eq fst-eq pb-sum-bits-val sum-list-exp by simp
  then show ?thesis by linarith
qed
```

Semantics of a threshold gate over a block of cost bits, and of the encoding reifications (4) and (5).

```
lemma cost-threshold-gate-forces:
  assumes rho01: ∀ v. rho v = 0 ∨ rho v = 1
    and m: models (reification r (map (λi. (2^i, Pos (cb i))) [0..<k]) A) rho
  shows eval-lit r rho = 1 ⟷ bits-val k cb rho ≥ A
  using reification-forces[OF rho01 m] by (simp add: pb-sum-bits-val)

lemma encode-cost-ge-forces:
  assumes rho01: ∀ v. rho v = 0 ∨ rho v = 1
    and m: models (encode-cost-ge k) rho
  shows rho (ReifCostGe k) = 1 ⟷ bits-val (bits-needed k) CostBit rho ≥ k
proof −
  have eval-lit (Pos (ReifCostGe k)) rho = 1 ⟷ bits-val (bits-needed k) CostBit
rho ≥ k
    using m unfolding encode-cost-ge-def
    by (intro cost-threshold-gate-forces[OF rho01]) simp
  then show ?thesis by (simp add: eval-lit-def)
qed

lemma encode-cost-ge-primed-forces:
  assumes rho01: ∀ v. rho v = 0 ∨ rho v = 1
    and m: models (encode-cost-ge-primed k) rho
```

shows *rho* (*ReifPrimedCostGe k*) = *1* ⟷ *bits-val* (*bits-needed k*) *PrimedCostBit rho* ≥ *k*
proof −
have *eval-lit* (*Pos* (*ReifPrimedCostGe k*)) *rho* = *1*
⟷ *bits-val* (*bits-needed k*) *PrimedCostBit rho* ≥ *k*
using *m* **unfolding** *encode-cost-ge-primed-def*
by (*intro cost-threshold-gate-forces*[*OF rho01*]) *simp*
then show *?thesis* **by** (*simp add*: *eval-lit-def*)
qed

lemma *models-mono*:
models CC rho ⟹ *DD* ⊆ *CC* ⟹ *models DD rho*
unfolding *models-def* **by** *blast*

3.5 Semantics of the transition encoding gates

Equation (3): the three-gate circuit for $\Delta c{=}k$ pins *ReifDeltaCost* to the truth of $c' = c + k$.

lemma *encode-delta-cost-lower-forces*:
assumes *rho01*: ∀ *v*. *rho v* = *0* ∨ *rho v* = *1*
and *m*: *models* (*encode-delta-cost k nb*) *rho*
shows *rho* (*ReifDeltaCostLower k*) = *1*
⟷ *bits-val nb PrimedCostBit rho* ≥ *bits-val nb CostBit rho* + *k*
proof −
let *?c* = *bits-val nb CostBit rho*
let *?c'* = *bits-val nb PrimedCostBit rho*
let *?M* = (*2*::*nat*)^*nb* − *1*
have *c-le*: *?c* ≤ *?M*
using *bits-val-lt*[*of rho CostBit nb*] *rho01* **by** *fastforce*
have *mlow*: *models* (*reification* (*Pos* (*ReifDeltaCostLower k*))
(*map* (λ*i*. (*2*^*i*, *Pos* (*PrimedCostBit i*))) [*0*..<*nb*]
@ *map* (λ*i*. (*2*^*i*, *Neg* (*CostBit i*))) [*0*..<*nb*]) (*k* + *?M*)) *rho*
by (*rule models-mono*[*OF m*]) (*auto simp*: *encode-delta-cost-def Let-def*)
have *sum-eq*: *pb-sum* (*map* (λ*i*. (*2*^*i*, *Pos* (*PrimedCostBit i*))) [*0*..<*nb*]
@ *map* (λ*i*. (*2*^*i*, *Neg* (*CostBit i*))) [*0*..<*nb*]) *rho* = *?c'* + (*?M* − *?c*)
by (*simp add*: *pb-sum-append pb-sum-bits-val pb-sum-neg-bits-val*[*OF rho01*])
have *eval-lit* (*Pos* (*ReifDeltaCostLower k*)) *rho* = *1* ⟷ *?c'* + (*?M* − *?c*) ≥ *k* + *?M*
using *reification-forces*[*OF rho01 mlow*] **by** (*simp add*: *sum-eq*)
moreover have *?c'* + (*?M* − *?c*) ≥ *k* + *?M* ⟷ *?c'* ≥ *?c* + *k*
using *c-le* **by** *linarith*
ultimately show *?thesis* **by** (*simp add*: *eval-lit-def*)
qed

lemma *encode-delta-cost-upper-forces*:
assumes *rho01*: ∀ *v*. *rho v* = *0* ∨ *rho v* = *1*
and *m*: *models* (*encode-delta-cost k nb*) *rho*
shows *rho* (*ReifDeltaCostUpper k*) = *1*
⟷ *bits-val nb PrimedCostBit rho* ≤ *bits-val nb CostBit rho* + *k*

```
proof −
  let ?c = bits-val nb CostBit rho
  let ?c' = bits-val nb PrimedCostBit rho
  let ?M = (2::nat)^nb − 1
  have c'-le: ?c' ≤ ?M
    using bits-val-lt[of rho PrimedCostBit nb] rho01 by fastforce
  have mup: models (reification (Pos (ReifDeltaCostUpper k))
      (map (λi. (2^i, Pos (CostBit i))) [0..<nb]
       @ map (λi. (2^i, Neg (PrimedCostBit i))) [0..<nb]) (?M − k)) rho
    by (rule models-mono[OF m]) (auto simp: encode-delta-cost-def Let-def)
  have sum-eq: pb-sum (map (λi. (2^i, Pos (CostBit i))) [0..<nb]
      @ map (λi. (2^i, Neg (PrimedCostBit i))) [0..<nb]) rho = ?c + (?M − ?c')
    by (simp add: pb-sum-append pb-sum-bits-val pb-sum-neg-bits-val[OF rho01])
  have eval-lit (Pos (ReifDeltaCostUpper k)) rho = 1 ⟷ ?c + (?M − ?c') ≥
?M − k
    using reification-forces[OF rho01 mup] by (simp add: sum-eq)
  moreover have ?c + (?M − ?c') ≥ ?M − k ⟷ ?c' ≤ ?c + k
    using c'-le by linarith
  ultimately show ?thesis by (simp add: eval-lit-def)
qed

lemma encode-delta-cost-forces:
  assumes rho01: ∀ v. rho v = 0 ∨ rho v = 1
    and m: models (encode-delta-cost k nb) rho
  shows rho (ReifDeltaCost k) = 1
      ⟷ bits-val nb PrimedCostBit rho = bits-val nb CostBit rho + k
proof −
  let ?L = ReifDeltaCostLower k
  let ?U = ReifDeltaCostUpper k
  have mdelta: models (reification (Pos (ReifDeltaCost k))
      [(1, Pos ?L), (1, Pos ?U)] 2) rho
    by (rule models-mono[OF m]) (auto simp: encode-delta-cost-def Let-def)
  have L-le: rho ?L ≤ 1 and U-le: rho ?U ≤ 1
    using rho01[rule-format, of ?L] rho01[rule-format, of ?U] by auto
  have eval-lit (Pos (ReifDeltaCost k)) rho = 1 ⟷ rho ?L + rho ?U ≥ 2
    using reification-forces[OF rho01 mdelta] by (simp add: eval-lit-def)
  moreover have rho ?L + rho ?U ≥ 2 ⟷ rho ?L = 1 ∧ rho ?U = 1
    using L-le U-le by linarith
  ultimately have rho (ReifDeltaCost k) = 1 ⟷ rho ?L = 1 ∧ rho ?U = 1
    by (simp add: eval-lit-def)
  then show ?thesis
    using encode-delta-cost-lower-forces[OF rho01 m]
      encode-delta-cost-upper-forces[OF rho01 m]
    by auto
qed
```

Equation (6): the equality gate *ReifEq* pins the truth of $v = v'$.

```
lemma encode-eq-var-geq-forces:
  assumes rho01: ∀ v. rho v = 0 ∨ rho v = 1
```

```
    and m: models (encode-eq-var v) rho
  shows rho (ReifGeq (Original v)) = 1
      ⟷ rho (StateVar (Primed v)) ≤ rho (StateVar (Original v))
proof −
  let ?x = rho (StateVar (Original v))
  let ?y = rho (StateVar (Primed v))
  have mg: models (reification (Pos (ReifGeq (Original v)))
      [(1, Pos (StateVar (Original v))), (1, Neg (StateVar (Primed v)))] 1) rho
    by (rule models-mono[OF m]) (auto simp: encode-eq-var-def)
  have y-le: ?y ≤ 1 using rho01[rule-format, of StateVar (Primed v)] by auto
  have eval-lit (Pos (ReifGeq (Original v))) rho = 1 ⟷ ?x + (1 − ?y) ≥ 1
    using reification-forces[OF rho01 mg] by (simp add: eval-lit-def)
  moreover have ?x + (1 − ?y) ≥ 1 ⟷ ?y ≤ ?x
    using y-le rho01[rule-format, of StateVar (Original v)] by auto
  ultimately show ?thesis by (simp add: eval-lit-def)
qed

lemma encode-eq-var-leq-forces:
  assumes rho01: ∀ v. rho v = 0 ∨ rho v = 1
    and m: models (encode-eq-var v) rho
  shows rho (ReifLeq (Original v)) = 1
      ⟷ rho (StateVar (Original v)) ≤ rho (StateVar (Primed v))
proof −
  let ?x = rho (StateVar (Original v))
  let ?y = rho (StateVar (Primed v))
  have ml: models (reification (Pos (ReifLeq (Original v)))
      [(1, Neg (StateVar (Original v))), (1, Pos (StateVar (Primed v)))] 1) rho
    by (rule models-mono[OF m]) (auto simp: encode-eq-var-def)
  have x-le: ?x ≤ 1 using rho01[rule-format, of StateVar (Original v)] by auto
  have eval-lit (Pos (ReifLeq (Original v))) rho = 1 ⟷ (1 − ?x) + ?y ≥ 1
    using reification-forces[OF rho01 ml] by (simp add: eval-lit-def)
  moreover have (1 − ?x) + ?y ≥ 1 ⟷ ?x ≤ ?y
    using x-le rho01[rule-format, of StateVar (Primed v)] by auto
  ultimately show ?thesis by (simp add: eval-lit-def)
qed

lemma encode-eq-var-forces:
  assumes rho01: ∀ v. rho v = 0 ∨ rho v = 1
    and m: models (encode-eq-var v) rho
  shows rho (ReifEq (Original v)) = 1
      ⟷ rho (StateVar (Original v)) = rho (StateVar (Primed v))
proof −
  let ?L = ReifLeq (Original v)
  let ?G = ReifGeq (Original v)
  have meq: models (reification (Pos (ReifEq (Original v)))
      [(1, Pos ?L), (1, Pos ?G)] 2) rho
    by (rule models-mono[OF m]) (auto simp: encode-eq-var-def)
  have L-le: rho ?L ≤ 1 and G-le: rho ?G ≤ 1
    using rho01[rule-format, of ?L] rho01[rule-format, of ?G] by auto
```

have *eval-lit* (*Pos* (*ReifEq* (*Original v*))) *rho* = *1* ⟷ *rho ?L* + *rho ?G* ≥ *2*
using *reification-forces*[*OF rho01 meq*] **by** (*simp add*: *eval-lit-def*)
moreover have *rho ?L* + *rho ?G* ≥ *2* ⟷ *rho ?L* = *1* ∧ *rho ?G* = *1*
using *L-le G-le* **by** *linarith*
ultimately have *rho* (*ReifEq* (*Original v*)) = *1* ⟷ *rho ?L* = *1* ∧ *rho ?G* = *1*
by (*simp add*: *eval-lit-def*)
then show *?thesis*
using *encode-eq-var-leq-forces*[*OF rho01 m*] *encode-eq-var-geq-forces*[*OF rho01 m*]
by *auto*
qed

Pointwise sums of unit literals over a finite variable set.

lemma *rho01-le-one*:
fixes *rho* :: *'w* ⇒ *nat*
assumes ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
shows *rho y* ≤ *1*
using *assms*[*rule-format*, *of y*] **by** *auto*

lemma *pb-sum-unit-set-pos*:
assumes *finite S*
shows *pb-sum* (*map* (λ*v*. (*1*, *Pos* (*f v*))) (*sorted-list-of-set S*)) *rho* = (∑ *v*∈*S*. *rho* (*f v*))
using *pb-sum-unit-set*[*OF assms*, *of* λ*v*. *Pos* (*f v*) *rho*]
by (*simp add*: *eval-lit-def*)

lemma *pb-sum-unit-set-neg*:
assumes *finite S*
shows *pb-sum* (*map* (λ*v*. (*1*, *Neg* (*f v*))) (*sorted-list-of-set S*)) *rho* = (∑ *v*∈*S*. *1* − *rho* (*f v*))
using *pb-sum-unit-set*[*OF assms*, *of* λ*v*. *Neg* (*f v*) *rho*]
by (*simp add*: *eval-lit-def*)

lemma *sum-le-card*:
fixes *f* :: *'a* ⇒ *nat*
assumes *finite S* **and** ∀ *v*∈*S*. *f v* ≤ *1*
shows (∑ *v*∈*S*. *f v*) ≤ *card S*
proof −
have (∑ *v*∈*S*. *f v*) ≤ (∑ *v*∈*S*. *1*) **using** *assms*(*2*) **by** (*intro sum-mono*) *auto*
then show *?thesis* **by** *simp*
qed

lemma *sum-eq-card-ones*:
fixes *f* :: *'a* ⇒ *nat*
assumes ∀ *v*∈*S*. *f v* = *1*
shows (∑ *v*∈*S*. *f v*) = *card S*
proof −
have (∑ *v*∈*S*. *f v*) = (∑ *v*∈*S*. *1*) **using** *assms* **by** (*intro sum.cong*) *auto*
then show *?thesis* **by** *simp*

qed

lemma *sum-rho-all-one*:
assumes *rho01*: $\forall x.\ rho\ x = 0 \vee rho\ x = 1$
and *fin*: *finite S*
and *total*: $(\sum v{\in}S.\ rho\ (f\ v)) \geq card\ S$
shows $\forall v{\in}S.\ rho\ (f\ v) = 1$
proof −
have *bnd*: $\forall v{\in}S.\ rho\ (f\ v) \leq 1$
by (*intro ballI rho01-le-one*[*OF rho01*])
show *?thesis* **by** (*rule sum-units-all-one*[*OF fin total bnd*])
qed

lemma *sum-rho-all-zero*:
assumes *rho01*: $\forall x.\ rho\ x = 0 \vee rho\ x = 1$
and *fin*: *finite S*
and *total*: $(\sum v{\in}S.\ 1 - rho\ (f\ v)) \geq card\ S$
shows $\forall v{\in}S.\ rho\ (f\ v) = 0$
proof −
have $\forall v{\in}S.\ 1 - rho\ (f\ v) = 1$
by (*rule sum-units-all-one*[*OF fin total*]) *auto*
then show *?thesis* **using** *rho01* **by** (*metis diff-self-eq-0 zero-neq-one*)
qed

lemma *sum-rho-le-card*:
assumes *rho01*: $\forall x.\ rho\ x = 0 \vee rho\ x = 1$
and *fin*: *finite S*
shows $(\sum v{\in}S.\ rho\ (f\ v)) \leq card\ S$
by (*rule sum-le-card*[*OF fin*]) (*intro ballI rho01-le-one*[*OF rho01*])

lemma *sum-one-minus-le-card*:
fixes *rho* :: $'w \Rightarrow nat$
assumes *fin*: *finite S*
shows $(\sum v{\in}S.\ 1 - rho\ (f\ v)) \leq card\ S$
by (*rule sum-le-card*[*OF fin*]) *auto*

3.6 Semantics of the initial state, goal, and action selection gates

Equation (1): *ReifI* is true iff the state variables encode exactly the initial state on *vars* Π.

lemma *encode-init-forces*:
fixes Π :: *'v*::*linorder strips-task*
assumes *rho01*: $\forall x.\ rho\ x = 0 \vee rho\ x = 1$
and *m*: *models* (*encode-init* Π) *rho*
and *fin*: *finite* (*vars* Π)
and *init-sub*: *init* Π ⊆ *vars* Π
shows *rho ReifI* = *1*
$\longleftrightarrow$ $(\forall v \in vars\ \Pi.\ rho\ (StateVar\ v) = (if\ v \in init\ \Pi\ then\ 1\ else\ 0))$

proof −
let *?I* = *init* Π
let *?V* = *vars* Π
let *?body* = *state-lits ?I* @ *neg-state-lits* (*?V* − *?I*)
have *fin-I*: *finite ?I* **and** *fin-VI*: *finite* (*?V* − *?I*)
using *fin init-sub* **by** (*auto intro*: *finite-subset*)
have *m′*: *models* (*reification* (*Pos ReifI*) *?body* (*card ?V*)) *rho*
using *m* **by** (*simp add*: *encode-init-def Let-def*)
have *s-pos*: *pb-sum* (*state-lits ?I*) *rho* = (∑ *v*∈*?I*. *rho* (*StateVar v*))
unfolding *state-lits-def* **by** (*rule pb-sum-unit-set-pos*[*OF fin-I*])
have *s-neg*: *pb-sum* (*neg-state-lits* (*?V* − *?I*)) *rho* = (∑ *v*∈*?V* − *?I*. *1* − *rho* (*StateVar v*))
unfolding *neg-state-lits-def* **by** (*rule pb-sum-unit-set-neg*[*OF fin-VI*])
have *body-sum*: *pb-sum ?body rho*
= (∑ *v*∈*?I*. *rho* (*StateVar v*)) + (∑ *v*∈*?V* − *?I*. *1* − *rho* (*StateVar v*))
by (*simp add*: *pb-sum-append s-pos s-neg*)
have *card-split*: *card ?I* + *card* (*?V* − *?I*) = *card ?V*
using *fin init-sub*
by (*metis card-Diff-subset card-mono finite-subset le-add-diff-inverse*)
show *?thesis*
proof
assume *rho ReifI* = *1*
then have *pb-sum ?body rho* ≥ *card ?V*
using *reification-forces*[*OF rho01 m′*] **by** (*simp add*: *eval-lit-def*)
then have *total*: (∑ *v*∈*?I*. *rho* (*StateVar v*)) + (∑ *v*∈*?V* − *?I*. *1* − *rho* (*StateVar v*))
≥ *card ?I* + *card* (*?V* − *?I*)
using *body-sum card-split* **by** *simp*
have *bnd1*: (∑ *v*∈*?I*. *rho* (*StateVar v*)) ≤ *card ?I*
by (*rule sum-rho-le-card*[*OF rho01 fin-I*])
have *bnd2*: (∑ *v*∈*?V* − *?I*. *1* − *rho* (*StateVar v*)) ≤ *card* (*?V* − *?I*)
by (*rule sum-one-minus-le-card*[*OF fin-VI*])
have *t1*: (∑ *v*∈*?I*. *rho* (*StateVar v*)) ≥ *card ?I*
using *total bnd1 bnd2* **by** *linarith*
have *t2*: (∑ *v*∈*?V* − *?I*. *1* − *rho* (*StateVar v*)) ≥ *card* (*?V* − *?I*)
using *total bnd1 bnd2* **by** *linarith*
have *ones*: ∀ *v*∈*?I*. *rho* (*StateVar v*) = *1*
by (*rule sum-rho-all-one*[*OF rho01 fin-I t1*])
have *zeros*: ∀ *v*∈*?V* − *?I*. *rho* (*StateVar v*) = *0*
by (*rule sum-rho-all-zero*[*OF rho01 fin-VI t2*])
show ∀ *v* ∈ *?V*. *rho* (*StateVar v*) = (*if v* ∈ *?I then 1 else 0*)
using *ones zeros init-sub* **by** *auto*
next
assume *enc*: ∀ *v* ∈ *?V*. *rho* (*StateVar v*) = (*if v* ∈ *?I then 1 else 0*)
have *s1*: (∑ *v*∈*?I*. *rho* (*StateVar v*)) = *card ?I*
using *enc init-sub* **by** (*intro sum-eq-card-ones*) *auto*
have *s2*: (∑ *v*∈*?V* − *?I*. *1* − *rho* (*StateVar v*)) = *card* (*?V* − *?I*)
using *enc* **by** (*intro sum-eq-card-ones*) *auto*
have *pb-sum ?body rho* = *card ?V*

```
    using body-sum s1 s2 card-split by simp
  then have eval-lit (Pos ReifI) rho = 1
    using reification-forces[OF rho01 m'] by simp
  then show rho ReifI = 1 by (simp add: eval-lit-def)
 qed
qed
```

Equation (2): *ReifG* is true iff all goal variables are true.

```
lemma encode-goal-forces:
 fixes Π :: 'v::linorder strips-task
 assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
   and m: models (encode-goal Π) rho
   and fin: finite (goal Π)
 shows rho ReifG = 1 ⟷ (∀ v ∈ goal Π. rho (StateVar v) = 1)
proof −
 let ?G = goal Π
 have m': models (reification (Pos ReifG) (state-lits ?G) (card ?G)) rho
   using m by (simp add: encode-goal-def Let-def)
 have body-sum: pb-sum (state-lits ?G) rho = (∑ v∈?G. rho (StateVar v))
   unfolding state-lits-def by (rule pb-sum-unit-set-pos[OF fin])
 show ?thesis
 proof
   assume rho ReifG = 1
   then have (∑ v∈?G. rho (StateVar v)) ≥ card ?G
     using reification-forces[OF rho01 m'] body-sum by (simp add: eval-lit-def)
   then show ∀ v ∈ ?G. rho (StateVar v) = 1
     by (rule sum-rho-all-one[OF rho01 fin])
 next
   assume ∀ v ∈ ?G. rho (StateVar v) = 1
   then have (∑ v∈?G. rho (StateVar v)) = card ?G
     by (intro sum-eq-card-ones) auto
   then have eval-lit (Pos ReifG) rho = 1
     using reification-forces[OF rho01 m'] body-sum by simp
   then show rho ReifG = 1 by (simp add: eval-lit-def)
 qed
qed
```

Equation (8): if *ReifT* is true, some action gate is selected, and conversely a selected action gate forces *ReifT*.

```
lemma action-selection-forces:
 assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
   and m: models (action-selection-reif rs) rho
 shows rho ReifT = 1 ⟷ (∃ r ∈ set rs. eval-lit r rho = 1)
proof −
 have m': models (reification (Pos ReifT) (map (λr. (1, r)) rs) 1) rho
   using m by (simp add: action-selection-reif-def)
 have iff1: eval-lit (Pos ReifT) rho = 1 ⟷ pb-sum (map (λr. (1, r)) rs) rho
≥ 1
   by (rule reification-forces[OF rho01 m'])
```

```
  show ?thesis
  proof
    assume rho ReifT = 1
    then have pb-sum (map (λr. (1, r)) rs) rho ≥ 1
      using iff1 by (simp add: eval-lit-def)
    then have ∃(a, l) ∈ set (map (λr. (1, r)) rs). a * eval-lit l rho ≥ 1
      by (rule pb-sum-pos-ex)
    then obtain a l where al-in: (a, l) ∈ set (map (λr. (1, r)) rs)
      and al-pos: a * eval-lit l rho ≥ 1 by auto
    have r-in: l ∈ set rs and a1: a = 1 using al-in by auto
    have eval-lit l rho ≥ 1 using al-pos a1 by simp
    then have eval-lit l rho = 1
      using eval-lit-le-one[OF rho01, of l] by simp
    then show ∃ r ∈ set rs. eval-lit r rho = 1 using r-in by blast
  next
    assume ∃ r ∈ set rs. eval-lit r rho = 1
    then obtain r where r-in: r ∈ set rs and r1: eval-lit r rho = 1 by blast
    have pb-sum (map (λr. (1, r)) rs) rho ≥ 1
      using r-in r1
    proof (induction rs)
      case Nil
      then show ?case by simp
    next
      case (Cons q rs)
      then show ?case by (cases r = q) auto
    qed
    then have eval-lit (Pos ReifT) rho = 1 using iff1 by simp
    then show rho ReifT = 1 by (simp add: eval-lit-def)
  qed
qed
```

3.7 Semantics of the action constraint (equation (7))

A selected action gate forces all conjuncts of the action constraint: the cost-delta gate, the preconditions on the unprimed side, the effects on the primed side, the frame gates, and the negated cost bound. Variables in $add\ a \cap del\ a$ are unconstrained by the encoding (their two literals always contribute exactly 1), which matches the relaxation discussed in the faithfulness assessment.

```
lemma action-constraint-extract:
  fixes a :: 'v::linorder action and V :: 'v set and rho :: 'v var pvar ⇒ nat
  assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
    and sat: satisfies (action-constraint r a V B) rho
    and out: eval-lit r rho = 1
    and finV: finite V
    and pre-sub: pre a ⊆ V and add-sub: add a ⊆ V and del-sub: del a ⊆ V
  shows rho (ReifDeltaCost (cost a)) = 1
      ∧ rho (ReifPrimedCostGe B) = 0
```

$\wedge\ (\forall\, v \in \mathit{pre}\ a.\ \mathit{rho}\ (\mathit{StateVar}\ (\mathit{Original}\ v)) = 1)$
$\wedge\ (\forall\, v \in \mathit{add}\ a - \mathit{del}\ a.\ \mathit{rho}\ (\mathit{StateVar}\ (\mathit{Primed}\ v)) = 1)$
$\wedge\ (\forall\, v \in \mathit{del}\ a - \mathit{add}\ a.\ \mathit{rho}\ (\mathit{StateVar}\ (\mathit{Primed}\ v)) = 0)$
$\wedge\ (\forall\, v \in V - \mathit{evars}\ a.\ \mathit{rho}\ (\mathit{ReifEq}\ (\mathit{Original}\ v)) = 1)$
proof −
have *fin-pre*: *finite* (*pre a*) **and** *fin-add*: *finite* (*add a*)
and *fin-del*: *finite* (*del a*) **and** *fin-frame*: *finite* (*V − evars a*)
using *finV pre-sub add-sub del-sub* **by** (*auto intro*: *finite-subset*)
have *fin-amd*: *finite* (*add a − del a*) **and** *fin-dma*: *finite* (*del a − add a*)
and *fin-int*: *finite* (*add a ∩ del a*)
using *fin-add fin-del* **by** *auto*
define *A* **where** *A* = *2* + *card* (*pre a*) + *card V*
define *D* **where** *D* = *rho* (*ReifDeltaCost* (*cost a*))
define *Bv* **where** *Bv* = *rho* (*ReifPrimedCostGe B*)
define *Spre* **where** *Spre* = (∑ *v*∈*pre a*. *rho* (*StateVar* (*Original v*)))
define *Sadd* **where** *Sadd* = (∑ *v*∈*add a*. *rho* (*StateVar* (*Primed v*)))
define *Sdel* **where** *Sdel* = (∑ *v*∈*del a*. *1 − rho* (*StateVar* (*Primed v*)))
define *Sfr* **where** *Sfr* = (∑ *v*∈*V − evars a*. *rho* (*ReifEq* (*Original v*)))
have *negr*: *eval-lit* (*lit-neg r*) *rho* = *0*
using *eval-lit-neg-conv*[*OF rho01*, *of r*] *out* **by** *simp*
have *e-pre*: *pb-sum* (*map* (λ*v*. (*1*, *Pos* (*StateVar* (*Original v*)))) (*sorted-list-of-set* (*pre a*))) *rho* = *Spre*
unfolding *Spre-def* **by** (*rule pb-sum-unit-set-pos*[*OF fin-pre*])
have *e-add*: *pb-sum* (*map* (λ*v*. (*1*, *Pos* (*StateVar* (*Primed v*)))) (*sorted-list-of-set* (*add a*))) *rho* = *Sadd*
unfolding *Sadd-def* **by** (*rule pb-sum-unit-set-pos*[*OF fin-add*])
have *e-del*: *pb-sum* (*map* (λ*v*. (*1*, *Neg* (*StateVar* (*Primed v*)))) (*sorted-list-of-set* (*del a*))) *rho* = *Sdel*
unfolding *Sdel-def* **by** (*rule pb-sum-unit-set-neg*[*OF fin-del*])
have *e-fr*: *pb-sum* (*map* (λ*v*. (*1*, *Pos* (*ReifEq* (*Original v*)))) (*sorted-list-of-set* (*V − evars a*))) *rho* = *Sfr*
unfolding *Sfr-def* **by** (*rule pb-sum-unit-set-pos*[*OF fin-frame*])
have *snd-ac*: *snd* (*action-constraint r a V B*) = *A*
unfolding *action-constraint-def Let-def A-def* **by** *simp*
have *ge0*: *pb-sum* (*fst* (*action-constraint r a V B*)) *rho* ≥ *A*
using *sat snd-ac* **unfolding** *satisfies-def* **by** (*simp add*: *split-beta*)
have *lhs-eq*: *pb-sum* (*fst* (*action-constraint r a V B*)) *rho*
= *A* ∗ *eval-lit* (*lit-neg r*) *rho* + *D* + *Spre* + *Sadd* + *Sdel* + *Sfr* + (*1* − *Bv*)
unfolding *action-constraint-def fst-conv Let-def*
by (*simp add*: *pb-sum-append A-def D-def Bv-def eval-lit-def*
e-pre[*symmetric*] *e-add*[*symmetric*] *e-del*[*symmetric*] *e-fr*[*symmetric*] *add-ac*)
have *main-ge*: *D* + *Spre* + *Sadd* + *Sdel* + *Sfr* + (*1* − *Bv*) ≥ *A*
using *ge0 lhs-eq negr* **by** *simp*
— Bounds for each component.
have *D-le*: *D* ≤ *1* **unfolding** *D-def* **by** (*rule rho01-le-one*[*OF rho01*])
have *Bv-le*: *Bv* ≤ *1* **unfolding** *Bv-def* **by** (*rule rho01-le-one*[*OF rho01*])
have *Spre-le*: *Spre* ≤ *card* (*pre a*)
unfolding *Spre-def* **by** (*rule sum-rho-le-card*[*OF rho01 fin-pre*])
have *Sfr-le*: *Sfr* ≤ *card* (*V − evars a*)

unfolding *Sfr-def* **by** (*rule sum-rho-le-card*[*OF rho01 fin-frame*])
— Split the add/del sums along the overlap.
have *Sadd-split*: $Sadd = (\sum v{\in}add\ a - del\ a.\ rho\ (StateVar\ (Primed\ v)))$
$+ (\sum v{\in}add\ a \cap del\ a.\ rho\ (StateVar\ (Primed\ v)))$
proof −
have $(\sum v{\in}(add\ a - del\ a) \cup (add\ a \cap del\ a).\ rho\ (StateVar\ (Primed\ v)))$
$= (\sum v{\in}add\ a - del\ a.\ rho\ (StateVar\ (Primed\ v)))$
$+ (\sum v{\in}add\ a \cap del\ a.\ rho\ (StateVar\ (Primed\ v)))$
by (*rule sum.union-disjoint*) (*use fin-amd fin-int* **in** *auto*)
moreover have $(add\ a - del\ a) \cup (add\ a \cap del\ a) = add\ a$ **by** *auto*
ultimately show *?thesis* **unfolding** *Sadd-def* **by** *simp*
qed
have *Sdel-split*: $Sdel = (\sum v{\in}del\ a - add\ a.\ 1 - rho\ (StateVar\ (Primed\ v)))$
$+ (\sum v{\in}add\ a \cap del\ a.\ 1 - rho\ (StateVar\ (Primed\ v)))$
proof −
have $(\sum v{\in}(del\ a - add\ a) \cup (add\ a \cap del\ a).\ 1 - rho\ (StateVar\ (Primed\ v)))$
$= (\sum v{\in}del\ a - add\ a.\ 1 - rho\ (StateVar\ (Primed\ v)))$
$+ (\sum v{\in}add\ a \cap del\ a.\ 1 - rho\ (StateVar\ (Primed\ v)))$
by (*rule sum.union-disjoint*) (*use fin-dma fin-int* **in** *auto*)
moreover have $(del\ a - add\ a) \cup (add\ a \cap del\ a) = del\ a$ **by** *auto*
ultimately show *?thesis* **unfolding** *Sdel-def* **by** *simp*
qed
have *pair-sum*: $(\sum v{\in}add\ a \cap del\ a.\ rho\ (StateVar\ (Primed\ v)))$
$+ (\sum v{\in}add\ a \cap del\ a.\ 1 - rho\ (StateVar\ (Primed\ v))) = card\ (add\ a \cap del\ a)$
proof −
have *each*: $\forall\, v{\in}add\ a \cap del\ a.$
$rho\ (StateVar\ (Primed\ v)) + (1 - rho\ (StateVar\ (Primed\ v))) = 1$
proof
fix v **assume** $v \in add\ a \cap del\ a$
show $rho\ (StateVar\ (Primed\ v)) + (1 - rho\ (StateVar\ (Primed\ v))) = 1$
using *rho01-le-one*[*OF rho01*, *of StateVar* (*Primed v*)] **by** *linarith*
qed
have $(\sum v{\in}add\ a \cap del\ a.\ rho\ (StateVar\ (Primed\ v)))$
$+ (\sum v{\in}add\ a \cap del\ a.\ 1 - rho\ (StateVar\ (Primed\ v)))$
$= (\sum v{\in}add\ a \cap del\ a.\ rho\ (StateVar\ (Primed\ v)) + (1 - rho\ (StateVar\ (Primed\ v))))$
by (*rule sum.distrib*[*symmetric*])
also have $... = card\ (add\ a \cap del\ a)$
using *each* **by** (*rule sum-eq-card-ones*)
finally show *?thesis* **.**
qed
have *amd-le*: $(\sum v{\in}add\ a - del\ a.\ rho\ (StateVar\ (Primed\ v))) \leq card\ (add\ a - del\ a)$
by (*rule sum-rho-le-card*[*OF rho01 fin-amd*])
have *dma-le*: $(\sum v{\in}del\ a - add\ a.\ 1 - rho\ (StateVar\ (Primed\ v))) \leq card\ (del\ a - add\ a)$
by (*rule sum-one-minus-le-card*[*OF fin-dma*])
— Cardinality bookkeeping.

have *evars-sub*: *evars a* $\subseteq$ *V* **using** *add-sub del-sub* **by** (*auto simp*: *evars-def*)
have *fin-evars*: *finite* (*evars a*) **using** *fin-add fin-del* **by** (*simp add*: *evars-def*)
have *card-evars*: *card* (*add a − del a*) + *card* (*del a − add a*) + *card* (*add a* ∩ *del a*)
= *card* (*evars a*)
proof −
have *d1*: (*add a − del a*) ∩ (*add a* ∩ *del a*) = {} **by** *auto*
have *u1*: (*add a − del a*) ∪ (*add a* ∩ *del a*) = *add a* **by** *auto*
have *c1*: *card* (*add a − del a*) + *card* (*add a* ∩ *del a*) = *card* (*add a*)
using *card-Un-disjoint*[*OF fin-amd fin-int d1*] *u1* **by** *simp*
have *d2*: *add a* ∩ (*del a − add a*) = {} **by** *auto*
have *u2*: *add a* ∪ (*del a − add a*) = *evars a* **by** (*auto simp*: *evars-def*)
have *c2*: *card* (*add a*) + *card* (*del a − add a*) = *card* (*evars a*)
using *card-Un-disjoint*[*OF fin-add fin-dma d2*] *u2* **by** *simp*
show *?thesis* **using** *c1 c2* **by** *linarith*
qed
have *card-V-split*: *card* (*evars a*) + *card* (*V − evars a*) = *card V*
using *finV evars-sub*
by (*metis card-Diff-subset card-mono finite-subset le-add-diff-inverse*)
— All inequalities are tight.
have *t-D*: *D* = *1*
using *main-ge D-le Bv-le Spre-le Sfr-le Sadd-split Sdel-split pair-sum amd-le dma-le card-evars card-V-split A-def* **by** *linarith*
have *t-B*: *Bv* = *0*
using *main-ge D-le Bv-le Spre-le Sfr-le Sadd-split Sdel-split pair-sum amd-le dma-le card-evars card-V-split A-def* **by** *linarith*
have *t-pre*: *Spre* ≥ *card* (*pre a*)
using *main-ge D-le Bv-le Spre-le Sfr-le Sadd-split Sdel-split pair-sum amd-le dma-le card-evars card-V-split A-def* **by** *linarith*
have *t-amd*: (∑ *v*∈*add a − del a*. *rho* (*StateVar* (*Primed v*))) ≥ *card* (*add a − del a*)
using *main-ge D-le Bv-le Spre-le Sfr-le Sadd-split Sdel-split pair-sum amd-le dma-le card-evars card-V-split A-def* **by** *linarith*
have *t-dma*: (∑ *v*∈*del a − add a*. *1 − rho* (*StateVar* (*Primed v*))) ≥ *card* (*del a − add a*)
using *main-ge D-le Bv-le Spre-le Sfr-le Sadd-split Sdel-split pair-sum amd-le dma-le card-evars card-V-split A-def* **by** *linarith*
have *t-fr*: *Sfr* ≥ *card* (*V − evars a*)
using *main-ge D-le Bv-le Spre-le Sfr-le Sadd-split Sdel-split pair-sum amd-le dma-le card-evars card-V-split A-def* **by** *linarith*
— Pointwise consequences.
have *pre-ones*: ∀ *v* ∈ *pre a*. *rho* (*StateVar* (*Original v*)) = *1*
using *t-pre* **unfolding** *Spre-def* **by** (*rule sum-rho-all-one*[*OF rho01 fin-pre*])
have *add-ones*: ∀ *v* ∈ *add a − del a*. *rho* (*StateVar* (*Primed v*)) = *1*
using *t-amd* **by** (*rule sum-rho-all-one*[*OF rho01 fin-amd*])
have *del-zeros*: ∀ *v* ∈ *del a − add a*. *rho* (*StateVar* (*Primed v*)) = *0*
using *t-dma* **by** (*rule sum-rho-all-zero*[*OF rho01 fin-dma*])
have *fr-ones*: ∀ *v* ∈ *V − evars a*. *rho* (*ReifEq* (*Original v*)) = *1*
using *t-fr* **unfolding** *Sfr-def* **by** (*rule sum-rho-all-one*[*OF rho01 fin-frame*])

show *?thesis*
using *t-D t-B pre-ones add-ones del-zeros fr-ones*
unfolding *D-def Bv-def* **by** *blast*
qed

3.8 Task embedding into an extended variable type

The certificate format restricts circuit gate names to *ReifCert x* with $x :: 'v$ at most as many names as the task has variables. The A*, PDB and hmax circuits need unboundedly many gates. Since Theorem 1 is polymorphic in the variable type, we instantiate it at the sum type $'v + 'g$: the task lives in the *Inl* part, and certificate gate names *ReifCert* (*Inr i*) are guaranteed fresh.

instantiation *sum* :: (*linorder*, *linorder*) *linorder*
begin

definition *less-eq-sum* :: $'a + 'b \Rightarrow 'a + 'b \Rightarrow$ *bool* **where**
less-eq-sum x y ≡ *case* (*x*, *y*) *of*
(*Inl a*, *Inl b*) ⇒ *a* ≤ *b*
| (*Inl* -, *Inr* -) ⇒ *True*
| (*Inr* -, *Inl* -) ⇒ *False*
| (*Inr a*, *Inr b*) ⇒ *a* ≤ *b*

definition *less-sum* :: $'a + 'b \Rightarrow 'a + 'b \Rightarrow$ *bool* **where**
less-sum x y ≡ *x* ≤ *y* ∧ ¬ *y* ≤ *x*

instance
by *standard* (*auto simp*: *less-eq-sum-def less-sum-def split*: *sum.splits*)

end

definition *embed-act* :: $'v$ *action* ⇒ $('v + 'g)$ *action* **where**
embed-act a ≡ (| *pre* = *Inl* ‘ *pre a*, *add* = *Inl* ‘ *add a*, *del* = *Inl* ‘ *del a*,
cost = *cost a* |)

definition *embed-task* :: $'v$ *strips-task* ⇒ $('v + 'g)$ *strips-task* **where**
embed-task Π ≡ (| *vars* = *Inl* ‘ *vars* Π, *acts* = *embed-act* ‘ *acts* Π,
init = *Inl* ‘ *init* Π, *goal* = *Inl* ‘ *goal* Π |)

lemma *embed-act-applicable*:
applicable (*embed-act a*) (*Inl* ‘ *s*) ⟷ *applicable a s*
by (*simp add*: *applicable-def embed-act-def inj-image-subset-iff*)

lemma *embed-act-successor*:
successor (*embed-act a*) (*Inl* ‘ *s*) = *Inl* ‘ *successor a s*
by (*simp add*: *successor-def embed-act-def image-Un image-set-diff*)

lemma *embed-path*:

```
  assumes path (acts Π) s t π
  shows path (acts (embed-task Π)) (Inl ‘ s) (Inl ‘ t) (map embed-act π)
  using assms
proof (induction rule: path.induct)
  case (path-nil s)
  show ?case by (simp add: path.path-nil)
next
  case (path-cons a s t π)
  then show ?case
    by (auto intro!: path.path-cons
        simp: embed-act-applicable embed-act-successor embed-task-def)
qed

lemma embed-plan-cost:
  plan-cost (map embed-act π) = plan-cost π
  by (simp add: plan-cost-def embed-act-def o-def)

lemma embed-is-plan-for:
  assumes is-plan-for Π π
  shows is-plan-for (embed-task Π) (map embed-act π)
proof −
  from assms obtain s where p: path (acts Π) (init Π) s π
    and g: is-goal-state Π s
    unfolding is-plan-for-def by blast
  have ip: init (embed-task Π) = Inl ‘ init Π
    by (simp add: embed-task-def)
  have path (acts (embed-task Π)) (Inl ‘ init Π) (Inl ‘ s) (map embed-act π)
    by (rule embed-path[OF p])
  moreover have is-goal-state (embed-task Π) (Inl ‘ s)
    using g by (auto simp: is-goal-state-def embed-task-def)
  ultimately show ?thesis unfolding is-plan-for-def ip by blast
qed
```

Theorem 1 transported back to the original task: a valid certificate over the extended variable type bounds the cost of every plan of the original task.

```
theorem embedded-certificate-lower-bound:
  fixes Π :: 'v::linorder strips-task
    and Cert :: (('v + 'g::linorder)) certificate
  assumes cert: certificate-valid-cpr B (embed-task Π) Cert
    and plan: is-plan-for Π π
  shows plan-cost π ≥ B
proof −
  have is-plan-for (embed-task Π) (map embed-act π)
    by (rule embed-is-plan-for[OF plan])
  from theorem-1-from-cpr[OF cert this]
  show ?thesis by (simp add: embed-plan-cost)
qed

corollary embedded-certificate-optimality:
```

fixes Π :: *'v::linorder strips-task*
and *Cert* :: (('*v* + '*g*::*linorder*)) *certificate*
assumes *cert*: *certificate-valid-cpr B* (*embed-task* Π) *Cert*
and *plan*: *is-plan-for* Π π **and** *cost*: *plan-cost* π = *B*
shows *optimal-plan* Π π
unfolding *optimal-plan-def*
using *plan cost embedded-certificate-lower-bound*[*OF cert*] **by** *auto*

end

4 K-Gates

theory *K-Gates*
imports *Encoding-Semantics*
begin

The paper's placeholder variables $K{\geq}l$ (equations (9)/(10)) reify the clipped cost condition $cost \geq min\{B, max\{0, l\}\}$ with $l \in \mathbb{Z}$. In the paper they are defined via the reification variables $cost{\geq}k$ from the encoding family (4)/(5); since the certificate format keeps circuit gates disjoint from the encoding's reification variables, we inline that composition: a K-gate reifies the clipped threshold *directly over the cost bits*. The primed copy of such a gate (under *primed-circuit*) then constrains *PrimedCostBit* exactly the paper's $K'{\geq}l$.

The paper proves its Lemmas 1 and 2 with the RED rule (empty witness). RED with an empty witness concludes C from $C \cup \{\neg C\} \models C$, i.e. from semantic implication which is subsumed by the semantic *unsat-01* (*?CC* $\cup$ {*constraint-neg ?C*}) $\Longrightarrow$ *cpr-derives ?CC ?C* rule of the formal system, so no additional proof rule is needed.

4.1 Clipping arithmetic

definition *clip* :: *nat* $\Rightarrow$ *int* $\Rightarrow$ *nat* **where**
clip B l $\equiv$ *min B* (*nat l*)

lemma *clip-nonpos*[*simp*]: *l* $\leq$ *0* $\Longrightarrow$ *clip B l* = *0*
by (*simp add*: *clip-def*)

lemma *clip-ge-B*[*simp*]: *l* $\geq$ *int B* $\Longrightarrow$ *clip B l* = *B*
by (*simp add*: *clip-def min-def nat-le-eq-zle*)

lemma *clip-in-range*[*simp*]: *0* $\leq$ *l* $\Longrightarrow$ *l* $\leq$ *int B* $\Longrightarrow$ *clip B l* = *nat l*
proof −
assume *0* $\leq$ *l* **and** *l* $\leq$ *int B*
then have *nat l* $\leq$ *B* **by** *simp*
then show *?thesis* **by** (*simp add*: *clip-def min-def*)
qed

lemma *clip-le-B*: *clip B l* $\leq$ *B*
by (*simp add*: *clip-def*)

lemma *clip-mono*:
assumes $j \leq j'$
shows *clip B j* $\leq$ *clip B j'*
proof −
have *nat j* $\leq$ *nat j'* **using** *assms* **by** (*rule nat-mono*)
then show *?thesis* **unfolding** *clip-def* **by** (*intro min.mono*) *auto*
qed

lemma *nat-add-int-le*: *nat* (*l* + *int m*) $\leq$ *nat l* + *m*
by (*cases l* $\geq$ *0*) *auto*

lemma *clip-add-le*:
clip B (*l* + *int m*) $\leq$ *clip B l* + *m*
proof (*cases nat l* $\leq$ *B*)
case *True*
have *clip B* (*l* + *int m*) $\leq$ *nat* (*l* + *int m*) **by** (*simp add*: *clip-def*)
also have ... $\leq$ *nat l* + *m* **by** (*rule nat-add-int-le*)
also have ... = *clip B l* + *m* **using** *True* **by** (*simp add*: *clip-def min-def*)
finally show *?thesis* **.**
next
case *False*
have *clip B* (*l* + *int m*) $\leq$ *B* **by** (*rule clip-le-B*)
also have ... $\leq$ *clip B l* + *m* **using** *False* **by** (*simp add*: *clip-def min-def*)
finally show *?thesis* **.**
qed

4.2 K-gates and their semantics

The gate triple for a K-gate with name literal *r*: it reifies Σ *2^i•c* $\geq$ *clip B l* over the *bits-needed B*-bit cost block. All new cost bodies use this width so they stay aligned with the *encode-delta-cost* gates of the transition encoding.

definition *k-gate-body* :: *nat* $\Rightarrow$ (*nat* $\times$ *'w pvar literal*) *list* **where**
k-gate-body B $\equiv$ *map* (λ*i*. (*2^i*, *Pos* (*CostBit i*))) [*0..<bits-needed B*]

definition *k-gate* :: *'w pvar literal* $\Rightarrow$ *nat* $\Rightarrow$ *int* $\Rightarrow$
'w pvar literal $\times$ (*nat* $\times$ *'w pvar literal*) *list* $\times$ *nat* **where**
k-gate r B l $\equiv$ (*r*, *k-gate-body B*, *clip B l*)

lemma *k-gate-forces*:
assumes *rho01*: $\forall$*x*. *rho x* = *0* $\vee$ *rho x* = *1*
and *m*: *models* (*reification r* (*k-gate-body B*) (*clip B l*)) *rho*
shows *eval-lit r rho* = *1* $\longleftrightarrow$ *bits-val* (*bits-needed B*) *CostBit rho* $\geq$ *clip B l*
using *cost-threshold-gate-forces*[*OF rho01 m*[*unfolded k-gate-body-def*]] **.**

Lemma 1 of the paper, semantically: if the cost bits witness the larger

clipped threshold *clip B (j + k)*, they witness the smaller one *clip B j*. At the gate level: a true K-gate for $j + k$ forces the K-gate for j.

lemma *k-gate-mono-semantic*:
  assumes *rho01*: $\forall x.$ *rho x = 0 ∨ rho x = 1*
    and *m-hi*: *models (reification r-hi (k-gate-body B) (clip B (j + int k))) rho*
    and *m-lo*: *models (reification r-lo (k-gate-body B) (clip B j)) rho*
    and *hi*: *eval-lit r-hi rho = 1*
  shows *eval-lit r-lo rho = 1*
proof −
  have *bits-val (bits-needed B) CostBit rho ≥ clip B (j + int k)*
    using *k-gate-forces*[*OF rho01 m-hi*] *hi* **by** *simp*
  moreover have *clip B j ≤ clip B (j + int k)*
    by (*rule clip-mono*) *simp*
  ultimately have *bits-val (bits-needed B) CostBit rho ≥ clip B j*
    by *simp*
  then show *?thesis* **using** *k-gate-forces*[*OF rho01 m-lo*] **by** *simp*
qed

Lemma 2 of the paper, semantically: from *cost ≥ clip B l* and $\Delta c = m$ conclude *cost′ ≥ clip B (l + m)*. The primed K-gate body is the *Primed-CostBit* block, which is what *primed-circuit* produces from a K-gate.

definition *k-gate-body-primed* :: *nat ⇒ (nat × 'w pvar literal) list* **where**
  k-gate-body-primed B ≡ map (λi. (2^i, Pos (PrimedCostBit i))) [0..<bits-needed B]

lemma *k-gate-primed-forces*:
  assumes *rho01*: $\forall x.$ *rho x = 0 ∨ rho x = 1*
    and *m*: *models (reification r (k-gate-body-primed B) (clip B l)) rho*
  shows *eval-lit r rho = 1 ⟷ bits-val (bits-needed B) PrimedCostBit rho ≥ clip B l*
  using *cost-threshold-gate-forces*[*OF rho01 m*[*unfolded k-gate-body-primed-def*]] **.**

lemma *k-gate-step-semantic*:
  assumes *rho01*: $\forall x.$ *rho x = 0 ∨ rho x = 1*
    and *mK*: *models (reification rK (k-gate-body B) (clip B l)) rho*
    and *mK′*: *models (reification rK′ (k-gate-body-primed B) (clip B (l + int m))) rho*
    and *mD*: *models (encode-delta-cost m (bits-needed B)) rho*
    and *K1*: *eval-lit rK rho = 1*
    and *D1*: *rho (ReifDeltaCost m) = 1*
  shows *eval-lit rK′ rho = 1*
proof −
  let *?c = bits-val (bits-needed B) CostBit rho*
  let *?c′ = bits-val (bits-needed B) PrimedCostBit rho*
  have *c-ge*: *?c ≥ clip B l*
    using *k-gate-forces*[*OF rho01 mK*] *K1* **by** *simp*
  have *c′-eq*: *?c′ = ?c + m*
    using *encode-delta-cost-forces*[*OF rho01 mD*] *D1* **by** *simp*
  have *clip B (l + int m) ≤ clip B l + m*

by (*rule clip-add-le*)
also have ... $\leq$ *?c* + *m* **using** *c-ge* **by** *simp*
also have ... = *?c′* **using** *c′-eq* **by** *simp*
finally show *?thesis*
using *k-gate-primed-forces*[*OF rho01 mK′*] **by** *simp*
qed

Bit-level form of the premise *cost* $\geq$ *clip B l* as a PB constraint used when a deduction-style hypothesis set assumes a clipped cost bound directly (the analogue of *cost-ge-constraint* for clipped thresholds).

definition *clip-cost-constraint* :: *nat* $\Rightarrow$ *int* $\Rightarrow$ *′w pconstraint* **where**
clip-cost-constraint B l $\equiv$ (*k-gate-body B*, *clip B l*)

lemma *clip-cost-constraint-sat*:
assumes *rho01*: $\forall$ *x*. *rho x* = *0* $\vee$ *rho x* = *1*
shows *satisfies* (*clip-cost-constraint B l*) *rho*
$\longleftrightarrow$ *bits-val* (*bits-needed B*) *CostBit rho* $\geq$ *clip B l*
by (*simp add*: *clip-cost-constraint-def k-gate-body-def satisfies-def*
pb-sum-bits-val)

end

5 Heuristic Certificates

theory *Heuristic-Certificates*
imports *K-Gates*
begin

Definition 4 of the paper: heuristic certificates. A heuristic maintains a PB circuit H (a gate list shared across all evaluated states) and, per evaluated state s, an output literal $r^h{}_s$ and an estimate $h\ s$. The three obligations (state, goal, inductivity lemma) are stated semantically over 0/1 models; by *1* $\leq$ *snd ?C* $\Longrightarrow$ *cpr-derives ?CC ?C* = ($\forall$ *rho*. ($\forall$ *v*. *rho v* = *0* $\vee$ *rho v* = *1*) $\longrightarrow$ *models ?CC rho* $\longrightarrow$ *satisfies ?C rho*) this is interchangeable with the paper's CPR-derivability formulation (a bridge for the state lemma is proved at the end of this theory).

5.1 Renaming assignments along literal maps

lemma *eval-lit-map-literal*:
eval-lit (*map-literal f l*) *rho* = *eval-lit l* (*rho* $\circ$ *f*)
by (*cases l*) (*simp-all add*: *eval-lit-def*)

lemma *lit-neg-map-literal*:
lit-neg (*map-literal f l*) = *map-literal f* (*lit-neg l*)
by (*cases l*) (*simp-all add*: *lit-neg-def*)

lemma *pb-sum-map-literal*:

pb-sum (*map* (λ(*a*, *l*). (*a*, *map-literal f l*)) *cs*) *rho* = *pb-sum cs* (*rho* ∘ *f*)
by (*induction cs*) (*auto simp*: *eval-lit-map-literal o-def*)

lemma *models-Union-iff*:
models (⋃ *x*∈*S*. *F x*) *rho* ⟷ (∀ *x*∈*S*. *models* (*F x*) *rho*)
unfolding *models-def* **by** *blast*

lemma *models-reification-lift*:
models (*reification* (*map-literal f r*) (*map* (λ(*a*, *l*). (*a*, *map-literal f l*)) *cs*) *A*) *rho*
⟷ *models* (*reification r cs A*) (*rho* ∘ *f*)
proof −
have *pos-sum*: *pb-sum* (*map* (λ(*a*, *l*). (*a*, *map-literal f l*)) *cs*) *rho* = *pb-sum cs* (*rho* ∘ *f*)
by (*rule pb-sum-map-literal*)
have *fst-eq*: *map* (*fst* ∘ (λ(*a*, *l*). (*a*, *map-literal f l*))) *cs* = *map fst cs*
by (*induction cs*) *auto*
have *neg-sum*: *pb-sum* (*map* ((λ(*a*, *l*). (*a*, *lit-neg l*)) ∘ (λ(*a*, *l*). (*a*, *map-literal f l*))) *cs*) *rho*
= *pb-sum* (*map* (λ(*a*, *l*). (*a*, *lit-neg l*)) *cs*) (*rho* ∘ *f*)
by (*induction cs*) (*auto simp*: *lit-neg-map-literal eval-lit-map-literal o-def*)
have *fwd*: *satisfies* (*reif-fwd* (*map-literal f r*) (*map* (λ(*a*, *l*). (*a*, *map-literal f l*)) *cs*) *A*) *rho*
⟷ *satisfies* (*reif-fwd r cs A*) (*rho* ∘ *f*)
by (*simp add*: *reif-fwd-def satisfies-def pos-sum*
lit-neg-map-literal eval-lit-map-literal)
have *bwd*: *satisfies* (*reif-bwd* (*map-literal f r*) (*map* (λ(*a*, *l*). (*a*, *map-literal f l*)) *cs*) *A*) *rho*
⟷ *satisfies* (*reif-bwd r cs A*) (*rho* ∘ *f*)
by (*simp add*: *reif-bwd-def satisfies-def Let-def fst-eq neg-sum eval-lit-map-literal*)
show *?thesis*
by (*simp add*: *reification-def models-def fwd bwd*)
qed

lemma *models-circuit-constraints-lift*:
models (*circuit-constraints* (*lift-circuit f C*)) *rho*
⟷ *models* (*circuit-constraints C*) (*rho* ∘ *f*)
proof −
obtain *pairs out* **where** *C*: *C* = (*pairs*, *out*) **by** (*cases C*)
have *set-lift*: *set* (*fst* (*lift-circuit f C*))
= (λ(*r*, *cs*, *A*). (*map-literal f r*, *map* (λ(*a*, *l*). (*a*, *map-literal f l*)) *cs*, *A*)) ‘ *set pairs*
by (*simp add*: *C lift-circuit-def*)
have *models* (*circuit-constraints* (*lift-circuit f C*)) *rho*
⟷ (∀ (*r*, *cs*, *A*) ∈ *set pairs*.
models (*reification* (*map-literal f r*) (*map* (λ(*a*, *l*). (*a*, *map-literal f l*)) *cs*) *A*) *rho*)
unfolding *circuit-constraints-def set-lift*
by (*fastforce simp*: *models-Union-iff split-beta*)
also have ... ⟷ (∀ (*r*, *cs*, *A*) ∈ *set pairs*. *models* (*reification r cs A*) (*rho* ∘ *f*))

```
    using models-reification-lift by (fastforce simp: split-beta)
  also have ... ⟷ models (circuit-constraints C) (rho ∘ f)
    unfolding C circuit-constraints-def
    by (fastforce simp: models-Union-iff split-beta)
  finally show ?thesis .
qed

lemma rho01-comp:
  (∀ x. rho x = 0 ∨ rho x = 1) ⟹ (∀ x. (rho ∘ f) x = 0 ∨ (rho ∘ f) x = 1)
  by simp
```

5.2 Heuristic certificates (Definition 4)

```
record ('u) heuristic-cert =
  hc-gates :: ('u pvar literal × (nat × 'u pvar literal) list × nat) list
  hc-out   :: 'u state ⇒ 'u pvar literal
  hc-h     :: 'u state ⇒ nat

definition hc-constraints :: ('u) heuristic-cert ⇒ 'u pconstraint set where
  hc-constraints HC ≡ ⋃(r, cs, A) ∈ set (hc-gates HC). reification r cs A

lemma hc-constraints-eq-circuit:
  hc-constraints HC = circuit-constraints (hc-gates HC, out)
  by (simp add: hc-constraints-def circuit-constraints-def)
```

State lemma: if the state variables encode exactly s on the task variables and the cost bits witness *clip B (B − h s)*, the output gate for s is forced. (Paper: $(r_s \wedge cost{\geq}max\{0,Bh(s)\}) \rightarrow r^h{}_s$.)

```
definition hc-state-lemma ::
  'u strips-task ⇒ nat ⇒ ('u) heuristic-cert ⇒ 'u state ⇒ bool where
  hc-state-lemma Πe B HC s ≡
    ∀ rho. (∀ x. rho x = 0 ∨ rho x = 1) ⟶
      models (hc-constraints HC) rho ⟶
      (∀ v ∈ vars Πe. rho (StateVar v) = (if v ∈ s then 1 else 0)) ⟶
      bits-val (bits-needed B) CostBit rho ≥ clip B (int B − int (hc-h HC s)) ⟶
      eval-lit (hc-out HC s) rho = 1

lemma hc-state-lemmaD:
  assumes hc-state-lemma Πe B HC s
    and ∀ x. rho x = 0 ∨ rho x = 1
    and models (hc-constraints HC) rho
    and ∀ v ∈ vars Πe. rho (StateVar v) = (if v ∈ s then 1 else 0)
    and bits-val (bits-needed B) CostBit rho ≥ clip B (int B − int (hc-h HC s))
  shows eval-lit (hc-out HC s) rho = 1
  using assms unfolding hc-state-lemma-def by blast
```

Goal lemma: a true output gate together with a satisfied goal forces the cost bits to reach B. (Paper: $(r_G \wedge r^h{}_s) \rightarrow cost{\geq}B$.)

```
definition hc-goal-lemma ::
```

'u strips-task ⇒ nat ⇒ ('u) heuristic-cert ⇒ 'u state ⇒ bool **where**
hc-goal-lemma Π*e B HC s* ≡
∀ *rho*. (∀ *x*. *rho x* = *0* ∨ *rho x* = *1*) ⟶
models (*hc-constraints HC*) *rho* ⟶
(∀ *v* ∈ *goal* Π*e*. *rho* (*StateVar v*) = *1*) ⟶
eval-lit (*hc-out HC s*) *rho* = *1* ⟶
bits-val (*bits-needed B*) *CostBit rho* ≥ *B*

lemma *hc-goal-lemmaD*:
assumes *hc-goal-lemma* Π*e B HC s*
and ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
and *models* (*hc-constraints HC*) *rho*
and ∀ *v* ∈ *goal* Π*e*. *rho* (*StateVar v*) = *1*
and *eval-lit* (*hc-out HC s*) *rho* = *1*
shows *bits-val* (*bits-needed B*) *CostBit rho* ≥ *B*
using *assms* **unfolding** *hc-goal-lemma-def* **by** *blast*

Inductivity lemma: across one encoded transition, a true (unprimed) output gate forces the primed copy. (Paper: $(r^h{}_s \wedge r_T) \rightarrow r^h{}_s{}'$ from $C_{trans} \cup H \cup H' \cup C_{\geq} \cup C_K$.)

definition *hc-ind-lemma* ::
'u::*linorder strips-task ⇒ nat ⇒ 'u action list ⇒ ('u) heuristic-cert ⇒ 'u state ⇒ bool* **where**
hc-ind-lemma Π*e B as HC s* ≡
∀ *rho*. (∀ *x*. *rho x* = *0* ∨ *rho x* = *1*) ⟶
models (*circuit-constraints* (*orig-circuit* (*hc-gates HC*, *hc-out HC s*))) *rho* ⟶
models (*circuit-constraints* (*primed-circuit* (*hc-gates HC*, *hc-out HC s*))) *rho* ⟶
models (*encode-transition as* (*vars* Π*e*) *B*) *rho* ⟶
models (*encode-cost-ge B*) *rho* ⟶
rho ReifT = *1* ⟶
eval-lit (*map-literal* (*map-pvar Original*) (*hc-out HC s*)) *rho* = *1* ⟶
eval-lit (*map-literal primed-pvar-map* (*hc-out HC s*)) *rho* = *1*

lemma *hc-ind-lemmaD*:
assumes *hc-ind-lemma* Π*e B as HC s*
and ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
and *models* (*circuit-constraints* (*orig-circuit* (*hc-gates HC*, *hc-out HC s*))) *rho*
and *models* (*circuit-constraints* (*primed-circuit* (*hc-gates HC*, *hc-out HC s*))) *rho*
and *models* (*encode-transition as* (*vars* Π*e*) *B*) *rho*
and *models* (*encode-cost-ge B*) *rho*
and *rho ReifT* = *1*
and *eval-lit* (*map-literal* (*map-pvar Original*) (*hc-out HC s*)) *rho* = *1*
shows *eval-lit* (*map-literal primed-pvar-map* (*hc-out HC s*)) *rho* = *1*
using *assms* **unfolding** *hc-ind-lemma-def* **by** *blast*

definition *hc-valid* ::
'u::*linorder strips-task ⇒ nat ⇒ 'u action list ⇒ ('u) heuristic-cert ⇒ 'u state*

set ⇒ *bool* **where**
hc-valid Π*e B as HC S* ≡
∀ *s* ∈ *S*. *hc-state-lemma* Π*e B HC s* ∧ *hc-goal-lemma* Π*e B HC s* ∧ *hc-ind-lemma* Π*e B as HC s*

5.3 The state lemma in primed variables

The paper requires the heuristic to "log a proof for the state lemma in terms of the primed variables". Formally this is a consequence of the unprimed state lemma, by precomposing a model of the primed circuit copy with the renaming *primed-pvar-map*.

lemma *hc-state-lemma-primed*:
fixes Π*e* :: *'u*::*linorder strips-task*
assumes *sl*: *hc-state-lemma* Π*e B HC s*
and *rho01*: ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
and *m*: *models* (*circuit-constraints* (*primed-circuit* (*hc-gates HC*, *out*))) *rho*
and *sv*: ∀ *v* ∈ *vars* Π*e*. *rho* (*StateVar* (*Primed v*)) = (*if v* ∈ *s then 1 else 0*)
and *cb*: *bits-val* (*bits-needed B*) *PrimedCostBit rho* ≥ *clip B* (*int B* − *int* (*hc-h HC s*))
shows *eval-lit* (*map-literal primed-pvar-map* (*hc-out HC s*)) *rho* = *1*
proof −
let *?rho'* = *rho* ∘ *primed-pvar-map*
have *rho'01*: ∀ *x*. *?rho' x* = *0* ∨ *?rho' x* = *1*
by (*rule rho01-comp*[*OF rho01*])
have *m'*: *models* (*hc-constraints HC*) *?rho'*
using *m models-circuit-constraints-lift*[*of primed-pvar-map* (*hc-gates HC*, *out*) *rho*]
by (*simp add*: *primed-circuit-def hc-constraints-eq-circuit*[*of HC out*])
have *sv'*: ∀ *v* ∈ *vars* Π*e*. *?rho'* (*StateVar v*) = (*if v* ∈ *s then 1 else 0*)
using *sv* **by** *simp*
have *cb'*: *bits-val* (*bits-needed B*) *CostBit ?rho'* ≥ *clip B* (*int B* − *int* (*hc-h HC s*))
using *cb* **by** (*simp add*: *bits-val-def*)
have *eval-lit* (*hc-out HC s*) *?rho'* = *1*
by (*rule hc-state-lemmaD*[*OF sl rho'01 m' sv' cb'*])
then show *?thesis* **by** (*simp add*: *eval-lit-map-literal*)
qed

The analogous fact for the unprimed copy embedded by *orig-circuit*.

lemma *hc-state-lemma-orig*:
fixes Π*e* :: *'u*::*linorder strips-task*
assumes *sl*: *hc-state-lemma* Π*e B HC s*
and *rho01*: ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
and *m*: *models* (*circuit-constraints* (*orig-circuit* (*hc-gates HC*, *out*))) *rho*
and *sv*: ∀ *v* ∈ *vars* Π*e*. *rho* (*StateVar* (*Original v*)) = (*if v* ∈ *s then 1 else 0*)
and *cb*: *bits-val* (*bits-needed B*) *CostBit rho* ≥ *clip B* (*int B* − *int* (*hc-h HC s*))
shows *eval-lit* (*map-literal* (*map-pvar Original*) (*hc-out HC s*)) *rho* = *1*
proof −
let *?rho'* = *rho* ∘ *map-pvar Original*

```
  have rho'01: ∀x. ?rho' x = 0 ∨ ?rho' x = 1
    by (rule rho01-comp[OF rho01])
  have m': models (hc-constraints HC) ?rho'
   using m models-circuit-constraints-lift[of map-pvar Original (hc-gates HC, out)
rho]
    by (simp add: orig-circuit-def hc-constraints-eq-circuit[of HC out])
  have sv': ∀v ∈ vars Πe. ?rho' (StateVar v) = (if v ∈ s then 1 else 0)
    using sv by simp
  have cb': bits-val (bits-needed B) CostBit ?rho' ≥ clip B (int B − int (hc-h HC
s))
    using cb by (simp add: bits-val-def)
  have eval-lit (hc-out HC s) ?rho' = 1
    by (rule hc-state-lemmaD[OF sl rho'01 m' sv' cb'])
  then show ?thesis by (simp add: eval-lit-map-literal)
qed
```

5.4 Faithfulness bridge: the state lemma as a CPR derivation

The paper states Definition 4 via CPR proofs. The semantic conditions above are interchangeable with that formulation; we make this explicit for the state lemma (the other two obligations are analogous). The state description C_s of the paper becomes a set of unit clauses, and the clipped cost premise its bit-level constraint.

```
definition state-unit-clauses :: 'u strips-task ⇒ 'u state ⇒ 'u pconstraint set where
  state-unit-clauses Πe s ≡
     (λv. unit-clause (Pos (StateVar v))) ‘ (vars Πe ∩ s)
   ∪ (λv. unit-clause (Neg (StateVar v))) ‘ (vars Πe − s)

lemma unit-clause-pos-sat:
  assumes rho01: ∀x. rho x = 0 ∨ rho x = 1
  shows satisfies (unit-clause (Pos w)) rho ⟷ rho w = 1
proof −
  have rho w ≤ 1 by (rule rho01-le-one[OF rho01])
  then show ?thesis
    by (auto simp: unit-clause-def satisfies-def eval-lit-def)
qed

lemma unit-clause-neg-sat:
  assumes rho01: ∀x. rho x = 0 ∨ rho x = 1
  shows satisfies (unit-clause (Neg w)) rho ⟷ rho w = 0
  using rho01[rule-format, of w]
  by (auto simp: unit-clause-def satisfies-def eval-lit-def)

lemma state-unit-clauses-sat:
  assumes rho01: ∀x. rho x = 0 ∨ rho x = 1
  shows models (state-unit-clauses Πe s) rho
      ⟷ (∀v ∈ vars Πe. rho (StateVar v) = (if v ∈ s then 1 else 0))
proof
```

```
  assume m: models (state-unit-clauses Πe s) rho
  show ∀ v ∈ vars Πe. rho (StateVar v) = (if v ∈ s then 1 else 0)
  proof
    fix v assume vV: v ∈ vars Πe
    show rho (StateVar v) = (if v ∈ s then 1 else 0)
    proof (cases v ∈ s)
      case True
      then have unit-clause (Pos (StateVar v)) ∈ state-unit-clauses Πe s
        using vV by (auto simp: state-unit-clauses-def)
      then have satisfies (unit-clause (Pos (StateVar v))) rho
        using m by (auto simp: models-def)
      then show ?thesis using True unit-clause-pos-sat[OF rho01] by simp
    next
      case False
      then have unit-clause (Neg (StateVar v)) ∈ state-unit-clauses Πe s
        using vV by (auto simp: state-unit-clauses-def)
      then have satisfies (unit-clause (Neg (StateVar v))) rho
        using m by (auto simp: models-def)
      then show ?thesis using False unit-clause-neg-sat[OF rho01] by simp
    qed
  qed
next
  assume enc: ∀ v ∈ vars Πe. rho (StateVar v) = (if v ∈ s then 1 else 0)
  show models (state-unit-clauses Πe s) rho
    unfolding state-unit-clauses-def models-def
    using enc unit-clause-pos-sat[OF rho01] unit-clause-neg-sat[OF rho01]
    by auto
qed

theorem hc-state-lemma-cpr:
  fixes Πe :: 'u::linorder strips-task
  assumes sl: hc-state-lemma Πe B HC s
  shows cpr-derives
    (hc-constraints HC ∪ state-unit-clauses Πe s
     ∪ {clip-cost-constraint B (int B − int (hc-h HC s))})
    (unit-clause (hc-out HC s))
proof (rule semantic-to-cpr)
  show snd (unit-clause (hc-out HC s)) ≥ (1::nat)
    by (simp add: unit-clause-def)
  show ∀ rho. (∀ x. rho x = 0 ∨ rho x = 1) ⟶
      models (hc-constraints HC ∪ state-unit-clauses Πe s
        ∪ {clip-cost-constraint B (int B − int (hc-h HC s))}) rho ⟶
      satisfies (unit-clause (hc-out HC s)) rho
  proof (intro allI impI)
    fix rho :: 'u pvar ⇒ nat
    assume rho01: ∀ x. rho x = 0 ∨ rho x = 1
      and m: models (hc-constraints HC ∪ state-unit-clauses Πe s
        ∪ {clip-cost-constraint B (int B − int (hc-h HC s))}) rho
    have m1: models (hc-constraints HC) rho
```

```
    and m2: models (state-unit-clauses Πe s) rho
    and m3: satisfies (clip-cost-constraint B (int B − int (hc-h HC s))) rho
    using m by (auto simp: models-def)
  have sv: ∀ v ∈ vars Πe. rho (StateVar v) = (if v ∈ s then 1 else 0)
    using state-unit-clauses-sat[OF rho01] m2 by blast
  have cb: bits-val (bits-needed B) CostBit rho ≥ clip B (int B − int (hc-h HC
s))
    using clip-cost-constraint-sat[OF rho01] m3 by blast
  have eval-lit (hc-out HC s) rho = 1
    by (rule hc-state-lemmaD[OF sl rho01 m1 sv cb])
  then show satisfies (unit-clause (hc-out HC s)) rho
    by (simp add: unit-clause-def satisfies-def)
 qed
qed
```

5.5 Generic conjunction and disjunction gates

The circuits of the paper's case study are built almost exclusively from two gate forms over unit-coefficient literal lists: conjunction gates $r \Leftrightarrow \Sigma\ \geq n$ (all of the n literals true) and disjunction gates $r \Leftrightarrow \Sigma\ \geq 1$ (some literal true).

```
lemma sum-list-units-all-one:
 fixes f :: 'a ⇒ nat
 assumes (∑ l←ls. f l) ≥ length ls and ∀ l∈set ls. f l ≤ 1
 shows ∀ l∈set ls. f l = 1
 using assms
proof (induction ls)
 case Nil
 then show ?case by simp
next
 case (Cons x ls)
 have x-le: f x ≤ 1 and ls-le: ∀ l∈set ls. f l ≤ 1
   using Cons.prems(2) by auto
 have ls-sum-le: (∑ l←ls. f l) ≤ length ls
   using ls-le by (induction ls) fastforce+
 have x1: f x = 1 and rest: (∑ l←ls. f l) ≥ length ls
   using Cons.prems(1) x-le ls-sum-le by auto
 show ?case using x1 Cons.IH[OF rest ls-le] by simp
qed

lemma conj-gate-forces:
 assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
   and m: models (reification r (map (λl. (1, l)) ls) (length ls)) rho
 shows eval-lit r rho = 1 ⟷ (∀ l∈set ls. eval-lit l rho = 1)
proof −
 have sum-eq: pb-sum (map (λl. (1, l)) ls) rho = (∑ l←ls. eval-lit l rho)
   by (induction ls) auto
 have le1: ∀ l∈set ls. eval-lit l rho ≤ 1
   by (intro ballI eval-lit-le-one[OF rho01])
```

```
  have iff1: eval-lit r rho = 1 ⟷ (∑ l←ls. eval-lit l rho) ≥ length ls
    using reification-forces[OF rho01 m] unfolding sum-eq .
  show ?thesis
  proof
    assume eval-lit r rho = 1
    then have (∑ l←ls. eval-lit l rho) ≥ length ls using iff1 by simp
    then show ∀ l∈set ls. eval-lit l rho = 1
      by (rule sum-list-units-all-one[OF - le1])
  next
    assume all1: ∀ l∈set ls. eval-lit l rho = 1
    have (∑ l←ls. eval-lit l rho) = length ls
      using all1 by (induction ls) auto
    then show eval-lit r rho = 1 using iff1 by simp
  qed
qed

lemma disj-gate-forces:
  assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
    and m: models (reification r (map (λl. (1, l)) ls) 1) rho
  shows eval-lit r rho = 1 ⟷ (∃ l∈set ls. eval-lit l rho = 1)
proof −
  have iff1: eval-lit r rho = 1 ⟷ pb-sum (map (λl. (1, l)) ls) rho ≥ 1
    by (rule reification-forces[OF rho01 m])
  show ?thesis
  proof
    assume eval-lit r rho = 1
    then have pb-sum (map (λl. (1, l)) ls) rho ≥ 1 using iff1 by simp
    then have ∃ (a, l) ∈ set (map (λl. (1, l)) ls). a ∗ eval-lit l rho ≥ 1
      by (rule pb-sum-pos-ex)
    then obtain a l where (a, l) ∈ set (map (λl. (1, l)) ls)
      and a ∗ eval-lit l rho ≥ 1 by auto
    then have l ∈ set ls and eval-lit l rho ≥ 1 by auto
    then show ∃ l∈set ls. eval-lit l rho = 1
      using eval-lit-le-one[OF rho01] by (metis le-antisym)
  next
    assume ∃ l∈set ls. eval-lit l rho = 1
    then obtain l where l-in: l ∈ set ls and l1: eval-lit l rho = 1 by blast
    have pb-sum (map (λl. (1, l)) ls) rho ≥ 1
      using l-in l1
    proof (induction ls)
      case Nil
      then show ?case by simp
    next
      case (Cons q ls)
      then show ?case by (cases l = q) auto
    qed
    then show eval-lit r rho = 1 using iff1 by simp
  qed
qed
```

end

6 A* Certificates

theory *A-Star-Certificates*
imports *Heuristic-Certificates*
begin

Proof-logging A* (paper equations (9)(13) and Lemmas 3–7). The locale *astar-run* captures the snapshot of a terminated proof-logging A* search: the closed list with g-values, the open list with a valid heuristic certificate, and the expansion-closure property (every applicable action from a closed state leads to a state covered by the closed list, by duplicate detection, or by an open state whose f-value reaches the bound the paper's "A* closes all states with f < B"). From this snapshot we assemble the certificate circuit $\langle A,\ r\text{-}A* \rangle$ over the extended variable type $'v + nat$ and prove it a valid lower-bound certificate.

6.1 Extracting components of the transition encoding

lemma *action-reifs-nth*: $j < length\ as \Longrightarrow action\text{-}reifs\ as\ !\ j = Pos\ (ReifAction\ j)$
by (*simp add*: *action-reifs-def*)

lemma *trans-sel*:
assumes *models* (*encode-transition as V B*) *rho*
shows *models* (*action-selection-reif* (*action-reifs as*)) *rho*
by (*rule models-mono*[*OF assms*]) (*auto simp*: *encode-transition-def Let-def*)

lemma *trans-action-constraint*:
assumes *m*: *models* (*encode-transition as V B*) *rho* **and** *j*: $j < length\ as$
shows *satisfies* (*action-constraint* (*Pos* (*ReifAction j*)) (*as* ! *j*) *V B*) *rho*
proof −
have *action-constraint* (*action-reifs as* ! *j*) (*as* ! *j*) *V B* ∈ *encode-transition as V B*
using *j* **unfolding** *encode-transition-def Let-def* **by** *auto*
then show *?thesis*
using *m action-reifs-nth*[*OF j*] **unfolding** *models-def* **by** *auto*
qed

lemma *trans-delta*:
assumes *m*: *models* (*encode-transition as V B*) *rho* **and** *a-in*: $a \in set\ as$
shows *models* (*encode-delta-cost* (*cost a*) (*bits-needed B*)) *rho*
by (*rule models-mono*[*OF m*]) (*use a-in* **in** ‹*auto simp*: *encode-transition-def Let-def*›)

lemma *trans-eq-var*:
assumes *m*: *models* (*encode-transition as V B*) *rho* **and** *v-in*: $v \in V$

shows *models* (*encode-eq-var v*) *rho*
by (*rule models-mono*[*OF m*]) (*use v-in* **in** ‹*auto simp*: *encode-transition-def Let-def*›)

lemma *trans-primed-ge*:
assumes *m*: *models* (*encode-transition as V B*) *rho*
shows *models* (*encode-cost-ge-primed B*) *rho*
by (*rule models-mono*[*OF m*]) (*auto simp*: *encode-transition-def Let-def*)

6.2 Embedded actions

lemma *pre-embed*: *pre* (*embed-act a*) = *Inl* ‘ *pre a*
and *add-embed*: *add* (*embed-act a*) = *Inl* ‘ *add a*
and *del-embed*: *del* (*embed-act a*) = *Inl* ‘ *del a*
and *cost-embed*: *cost* (*embed-act a*) = *cost a*
by (*simp-all add*: *embed-act-def*)

lemma *evars-embed*: *evars* (*embed-act a*) = *Inl* ‘ *evars a*
by (*simp add*: *evars-def embed-act-def image-Un*)

lemma *acts-embed*: *acts* (*embed-task* Π) = *embed-act* ‘ *acts* Π
by (*simp add*: *embed-task-def*)

locale *astar-run* =
fixes Π :: *'v*::*linorder strips-task*
and *B* :: *nat*
and *closed-list* :: (*'v state* × *nat*) *list*
and *open-list* :: *'v state list*
and *HC* :: ((*'v* + *nat*)) *heuristic-cert*
and *cas* :: (*'v* + *nat*) *action list*
assumes *fin-vars*: *finite* (*vars* Π)
and *init-sub*: *init* Π ⊆ *vars* Π
and *goal-sub*: *goal* Π ⊆ *vars* Π
and *fin-acts*: *finite* (*acts* Π)
and *acts-disjoint*: ∀ *a* ∈ *acts* Π. *add a* ∩ *del a* = {}
and *acts-states-sub*:
∀ *a* ∈ *acts* Π. *pre a* ⊆ *vars* Π ∧ *add a* ⊆ *vars* Π ∧ *del a* ⊆ *vars* Π
and *cas-eq*: *set cas* = *acts* (*embed-task* Π)
and *B-pos*: *B* ≥ *1*
— A* snapshot conditions:
and *closed-init*: (*init* Π, *0*) ∈ *set closed-list*
and *closed-sub*: ∀ (*s*, *g*) ∈ *set closed-list*. *s* ⊆ *vars* Π
and *open-sub*: ∀ *s* ∈ *set open-list*. *s* ⊆ *vars* Π
and *closed-goal-ge*: ∀ (*s*, *g*) ∈ *set closed-list*. *is-goal-state* Π *s* ⟶ *g* ≥ *B*
and *expansion*: ∀ (*s*, *g*) ∈ *set closed-list*. ∀ *a* ∈ *acts* Π. *applicable a s* ⟶
(*is-goal-state* Π *s* ∧ *g* ≥ *B*)
∨ (∃ *g'*. (*successor a s*, *g'*) ∈ *set closed-list* ∧ *g'* ≤ *g* + *cost a*)
∨ (*successor a s* ∈ *set open-list* ∧

int (*g* + *cost a*) ≥ *int B* − *int* (*hc-h HC* (*Inl* ‘ *successor a s*)))
— Heuristic certificate conditions:
and *hc-ok*: *hc-valid* (*embed-task* Π) *B cas HC* ((λ*s*. *Inl* ‘ *s*) ‘ *set open-list*)
and *hc-names*: ∀ (*r*, *cs*, *A*) ∈ *set* (*hc-gates HC*). ∃ *j*. *r* = *Pos* (*ReifCert* (*Inr* (*2* ∗ *j* + *1*)))
and *hc-distinct*: *distinct* (*map* (λ(*r*, *cs*, *A*). *pvar-of-lit r*) (*hc-gates HC*))
and *hc-wf*: ∀ *i* < *length* (*hc-gates HC*). *case hc-gates HC* ! *i of* (*r*, *cs*, *A*) ⇒
(∀ *x* ∈ *pvar-of-lit* ‘ *snd* ‘ *set cs*.
(∃ *v*. *x* = *StateVar v*) ∨ (∃ *j*. *x* = *CostBit j*)
∨ *x* ∈ *pvar-of-lit* ‘ *fst* ‘ *set* (*take i* (*hc-gates HC*)))
and *hc-out-in*: ∀ *s* ∈ *set open-list*. *hc-out HC* (*Inl* ‘ *s*) ∈ *fst* ‘ *set* (*hc-gates HC*)
begin

abbreviation Π*e* :: (′*v* + *nat*) *strips-task* **where**
Π*e* ≡ *embed-task* Π

definition *n-cl* :: *nat* **where**
n-cl = *length closed-list*

definition *n-hc* :: *nat* **where**
n-hc = *length* (*hc-gates HC*)

definition *cl-state* :: *nat* ⇒ ′*v state* **where**
cl-state i = *fst* (*closed-list* ! *i*)

definition *cl-g* :: *nat* ⇒ *nat* **where**
cl-g i = *snd* (*closed-list* ! *i*)

6.2.1 Gate names

definition *k-name* :: *nat* ⇒ (′*v* + *nat*) *pvar* **where**
k-name i = *ReifCert* (*Inr* (*2* ∗ *i*))

definition *cl-name* :: *nat* ⇒ (′*v* + *nat*) *pvar* **where**
cl-name i = *ReifCert* (*Inr* (*2* ∗ (*n-cl* + *i*)))

definition *out-name* :: (′*v* + *nat*) *pvar* **where**
out-name = *ReifCert* (*Inr* (*2* ∗ (*2* ∗ *n-cl*)))

6.2.2 Gates

K-gates: for each closed pair (*s*, *g*) the clipped cost threshold gate $K{\geq}g$ (paper equation (9), inlined over the cost bits).

definition *kg* :: *nat* ⇒ (′*v* + *nat*) *pvar literal* × (*nat* × (′*v* + *nat*) *pvar literal*) *list* × *nat* **where**
kg i = *k-gate* (*Pos* (*k-name i*)) *B* (*int* (*cl-g i*))

Closed-state gates (paper equation (11)): the conjunction of the exact state description of *cl-state i* and the K-gate for *cl-g i*.

definition *state-lits-exact* :: *'v state* ⇒ (*'v* + *nat*) *pvar literal list* **where**
state-lits-exact s =
map (λ*v*. *if v* ∈ *s then Pos* (*StateVar* (*Inl v*)) *else Neg* (*StateVar* (*Inl v*)))
(*sorted-list-of-set* (*vars* Π))

definition *cl-lits* :: *nat* ⇒ (*'v* + *nat*) *pvar literal list* **where**
cl-lits i = *Pos* (*k-name i*) # *state-lits-exact* (*cl-state i*)

definition *clg* :: *nat* ⇒ (*'v* + *nat*) *pvar literal* × (*nat* × (*'v* + *nat*) *pvar literal*) *list* × *nat* **where**
clg i = (*Pos* (*cl-name i*), *map* (λ*l*. (*1*, *l*)) (*cl-lits i*), *length* (*cl-lits i*))

The output gate (paper equation (13)): some closed-state gate or some open-state heuristic output is true.

definition *out-lits* :: (*'v* + *nat*) *pvar literal list* **where**
out-lits = *map* (λ*i*. *Pos* (*cl-name i*)) [*0*..<*n-cl*]
@ *map* (λ*s*. *hc-out HC* (*Inl* ‘ *s*)) *open-list*

definition *outg* :: (*'v* + *nat*) *pvar literal* × (*nat* × (*'v* + *nat*) *pvar literal*) *list* × *nat* **where**
outg = (*Pos out-name*, *map* (λ*l*. (*1*, *l*)) *out-lits*, *1*)

definition *astar-gates* ::
((*'v* + *nat*) *pvar literal* × (*nat* × (*'v* + *nat*) *pvar literal*) *list* × *nat*) *list* **where**
astar-gates = *map kg* [*0*..<*n-cl*] @ *map clg* [*0*..<*n-cl*] @ *hc-gates HC* @ [*outg*]

definition *astar-circuit* :: (*'v* + *nat*) *pb-circuit* **where**
astar-circuit = (*astar-gates*, *Pos out-name*)

definition *astar-cert* :: ((*'v* + *nat*)) *certificate* **where**
astar-cert = (| *cert-circuit* = *astar-circuit*, *cert-actions* = *cas* |)

6.2.3 Basic structure of the gate list

lemma *length-astar-gates*: *length astar-gates* = *2* ∗ *n-cl* + *n-hc* + *1*
by (*simp add*: *astar-gates-def n-hc-def*)

lemma *astar-gates-nth-k*:
i < *n-cl* ⟹ *astar-gates* ! *i* = *kg i*
by (*simp add*: *astar-gates-def nth-append*)

lemma *astar-gates-nth-cl*:
i < *n-cl* ⟹ *astar-gates* ! (*n-cl* + *i*) = *clg i*
by (*simp add*: *astar-gates-def nth-append*)

lemma *astar-gates-nth-hc*:
i < *n-hc* ⟹ *astar-gates* ! (*2* ∗ *n-cl* + *i*) = *hc-gates HC* ! *i*
by (*simp add*: *astar-gates-def n-hc-def nth-append*)

lemma *astar-gates-nth-out*:
astar-gates ! (2 ∗ *n-cl* + *n-hc*) = *outg*
by (*simp add*: *astar-gates-def n-hc-def nth-append*)

lemma *kg-in-gates*: *i* < *n-cl* ⟹ *kg i* ∈ *set astar-gates*
by (*simp add*: *astar-gates-def*)

lemma *clg-in-gates*: *i* < *n-cl* ⟹ *clg i* ∈ *set astar-gates*
by (*simp add*: *astar-gates-def*)

lemma *hc-in-gates*: *g* ∈ *set* (*hc-gates HC*) ⟹ *g* ∈ *set astar-gates*
by (*simp add*: *astar-gates-def*)

lemma *outg-in-gates*: *outg* ∈ *set astar-gates*
by (*simp add*: *astar-gates-def*)

6.2.4 Extracting per-gate models from a circuit model

lemma *models-gate*:
assumes *m*: *models* (*circuit-constraints astar-circuit*) *rho*
and *g-in*: (*r*, *cs*, *A*) ∈ *set astar-gates*
shows *models* (*reification r cs A*) *rho*
proof (*rule models-mono*[*OF m*])
show *reification r cs A* ⊆ *circuit-constraints astar-circuit*
using *g-in* **unfolding** *circuit-constraints-def astar-circuit-def* **by** *fastforce*
qed

lemma *models-kg*:
assumes *models* (*circuit-constraints astar-circuit*) *rho* **and** *i* < *n-cl*
shows *models* (*reification* (*Pos* (*k-name i*)) (*k-gate-body B*) (*clip B* (*int* (*cl-g i*)))) *rho*
using *models-gate*[*OF assms*(*1*)] *kg-in-gates*[*OF assms*(*2*)]
by (*simp add*: *kg-def k-gate-def*)

lemma *models-clg*:
assumes *models* (*circuit-constraints astar-circuit*) *rho* **and** *i* < *n-cl*
shows *models* (*reification* (*Pos* (*cl-name i*)) (*map* (λ*l*. (*1*, *l*)) (*cl-lits i*)) (*length* (*cl-lits i*))) *rho*
using *models-gate*[*OF assms*(*1*)] *clg-in-gates*[*OF assms*(*2*)]
by (*simp add*: *clg-def*)

lemma *models-outg*:
assumes *models* (*circuit-constraints astar-circuit*) *rho*
shows *models* (*reification* (*Pos out-name*) (*map* (λ*l*. (*1*, *l*)) *out-lits*) *1*) *rho*
using *models-gate*[*OF assms*] *outg-in-gates*
by (*simp add*: *outg-def*)

lemma *models-hc-constraints*:
assumes *models* (*circuit-constraints astar-circuit*) *rho*

shows *models* (*hc-constraints HC*) *rho*
proof −
have *hc-constraints HC* ⊆ *circuit-constraints astar-circuit*
unfolding *hc-constraints-def circuit-constraints-def astar-circuit-def astar-gates-def*
by *fastforce*
then show *?thesis* **by** (*rule models-mono*[*OF assms*])
qed

6.2.5 Semantics of the closed-state gates

lemma *state-lits-exact-sem*:
assumes *rho01*: ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
shows (∀ *l* ∈ *set* (*state-lits-exact s*). *eval-lit l rho* = *1*)
⟷ (∀ *v* ∈ *vars* Π. *rho* (*StateVar* (*Inl v*)) = (*if v* ∈ *s then 1 else 0*))
proof −
have *set-eq*: *set* (*sorted-list-of-set* (*vars* Π)) = *vars* Π
using *fin-vars* **by** *simp*
have *lit-sem*: ⋀*v*. *eval-lit* (*if v* ∈ *s then Pos* (*StateVar* (*Inl v*))
else Neg (*StateVar* (*Inl v*))) *rho* = *1*
⟷ *rho* (*StateVar* (*Inl v*)) = (*if v* ∈ *s then 1 else 0*)
proof −
fix *v*
show *eval-lit* (*if v* ∈ *s then Pos* (*StateVar* (*Inl v*))
else Neg (*StateVar* (*Inl v*))) *rho* = *1*
⟷ *rho* (*StateVar* (*Inl v*)) = (*if v* ∈ *s then 1 else 0*)
using *rho01*[*rule-format*, *of StateVar* (*Inl v*)]
by (*cases v* ∈ *s*) (*auto simp*: *eval-lit-def*)
qed
show *?thesis*
proof
assume *all1*: ∀ *l* ∈ *set* (*state-lits-exact s*). *eval-lit l rho* = *1*
show ∀ *v* ∈ *vars* Π. *rho* (*StateVar* (*Inl v*)) = (*if v* ∈ *s then 1 else 0*)
proof
fix *v* **assume** *vV*: *v* ∈ *vars* Π
have (*if v* ∈ *s then Pos* (*StateVar* (*Inl v*)) *else Neg* (*StateVar* (*Inl v*)))
∈ *set* (*state-lits-exact s*)
unfolding *state-lits-exact-def* **using** *vV set-eq* **by** *auto*
then have *eval-lit* (*if v* ∈ *s then Pos* (*StateVar* (*Inl v*))
else Neg (*StateVar* (*Inl v*))) *rho* = *1*
using *all1* **by** *blast*
then show *rho* (*StateVar* (*Inl v*)) = (*if v* ∈ *s then 1 else 0*)
using *lit-sem*[*of v*] **by** *simp*
qed
next
assume *enc*: ∀ *v* ∈ *vars* Π. *rho* (*StateVar* (*Inl v*)) = (*if v* ∈ *s then 1 else 0*)
show ∀ *l* ∈ *set* (*state-lits-exact s*). *eval-lit l rho* = *1*
proof
fix *l* **assume** *l* ∈ *set* (*state-lits-exact s*)
then obtain *v* **where** *vV*: *v* ∈ *vars* Π

```
      and l-eq: l = (if v ∈ s then Pos (StateVar (Inl v)) else Neg (StateVar (Inl
v)))
      unfolding state-lits-exact-def using set-eq by auto
    show eval-lit l rho = 1
      using enc vV lit-sem[of v] unfolding l-eq by simp
  qed
 qed
qed

lemma clg-forces:
 assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
   and m: models (circuit-constraints astar-circuit) rho
   and i-lt: i < n-cl
 shows rho (cl-name i) = 1
   ⟷ (rho (k-name i) = 1 ∧
      (∀ v ∈ vars Π. rho (StateVar (Inl v)) = (if v ∈ cl-state i then 1 else 0)))
proof −
 have eval-lit (Pos (cl-name i)) rho = 1 ⟷ (∀ l ∈ set (cl-lits i). eval-lit l rho =
1)
   by (rule conj-gate-forces[OF rho01 models-clg[OF m i-lt]])
 also have ... ⟷ (eval-lit (Pos (k-name i)) rho = 1 ∧
     (∀ l ∈ set (state-lits-exact (cl-state i)). eval-lit l rho = 1))
   by (simp add: cl-lits-def)
 also have ... ⟷ (rho (k-name i) = 1 ∧
     (∀ l ∈ set (state-lits-exact (cl-state i)). eval-lit l rho = 1))
   by (simp add: eval-lit-def)
 also have ... ⟷ (rho (k-name i) = 1 ∧
     (∀ v ∈ vars Π. rho (StateVar (Inl v)) = (if v ∈ cl-state i then 1 else 0)))
   using state-lits-exact-sem[OF rho01, of cl-state i] by blast
 finally show ?thesis by (simp add: eval-lit-def)
qed

lemma kg-forces:
 assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
   and m: models (circuit-constraints astar-circuit) rho
   and i-lt: i < n-cl
 shows rho (k-name i) = 1
   ⟷ bits-val (bits-needed B) CostBit rho ≥ clip B (int (cl-g i))
 using k-gate-forces[OF rho01 models-kg[OF m i-lt]]
 by (simp add: eval-lit-def)

lemma outg-forces:
 assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
   and m: models (circuit-constraints astar-circuit) rho
 shows rho out-name = 1 ⟷ (∃ l ∈ set out-lits. eval-lit l rho = 1)
 using disj-gate-forces[OF rho01 models-outg[OF m]]
 by (simp add: eval-lit-def)
```

6.2.6 Embedded task bookkeeping

lemma *vars-e*: *vars* Πe = *Inl* ‘ *vars* Π
by (*simp add*: *embed-task-def*)

lemma *goal-e*: *goal* Πe = *Inl* ‘ *goal* Π
by (*simp add*: *embed-task-def*)

lemma *init-e*: *init* Πe = *Inl* ‘ *init* Π
by (*simp add*: *embed-task-def*)

lemma *fin-vars-e*: *finite* (*vars* Πe)
using *fin-vars* **by** (*simp add*: *vars-e*)

lemma *init-sub-e*: *init* $\Pi e \subseteq$ *vars* Πe
using *init-sub* **by** (*auto simp*: *init-e vars-e*)

lemma *goal-sub-e*: *goal* $\Pi e \subseteq$ *vars* Πe
using *goal-sub* **by** (*auto simp*: *goal-e vars-e*)

lemma *fin-goal-e*: *finite* (*goal* Πe)
using *goal-sub-e fin-vars-e* **by** (*rule finite-subset*)

lemma *closed-nth-in*: $i <$ *n-cl* $\Longrightarrow$ (*cl-state i*, *cl-g i*) $\in$ *set closed-list*
unfolding *cl-state-def cl-g-def n-cl-def* **by** (*metis nth-mem prod.collapse*)

lemma *closed-mem-nth*: (*s*, *g*) $\in$ *set closed-list* $\Longrightarrow \exists\, i <$ *n-cl*. *cl-state* $i = s \wedge$ *cl-g* $i = g$
unfolding *cl-state-def cl-g-def n-cl-def* **by** (*metis fst-conv snd-conv in-set-conv-nth*)

lemma *hc-ok-state*: *s* $\in$ *set open-list* $\Longrightarrow$ *hc-state-lemma* Πe *B HC* (*Inl* ‘ *s*)
using *hc-ok* **unfolding** *hc-valid-def* **by** *blast*

lemma *hc-ok-goal*: *s* $\in$ *set open-list* $\Longrightarrow$ *hc-goal-lemma* Πe *B HC* (*Inl* ‘ *s*)
using *hc-ok* **unfolding** *hc-valid-def* **by** *blast*

lemma *hc-ok-ind*: *s* $\in$ *set open-list* $\Longrightarrow$ *hc-ind-lemma* Πe *B cas HC* (*Inl* ‘ *s*)
using *hc-ok* **unfolding** *hc-valid-def* **by** *blast*

lemma *state-descr-translate*:
($\forall\, w \in$ *vars* Πe. *rho* (*StateVar w*) = (*if w* $\in$ *Inl* ‘ *s then 1 else 0*))
$\longleftrightarrow$ ($\forall\, v \in$ *vars* Π. *rho* (*StateVar* (*Inl v*)) = (*if v* $\in$ *s then 1 else 0*))
by (*auto simp*: *vars-e inj-image-mem-iff*)

lemma *neg-cost-ge-one-zero*:
assumes *rho01*: $\forall\, x$. (*rho* :: (*'v* + *nat*) *pvar* $\Rightarrow$ *nat*) *x* = *0* $\vee$ *rho x* = *1*
and *sat*: *satisfies* (*neg-cost-ge-one B*) *rho*
shows *bits-val* (*bits-needed B*) *CostBit rho* = *0*
proof −
have *neg-sum*: *pb-sum* (*map* (λi. (*2^i*, *Neg* (*CostBit i*))) [*0..<bits-needed B*]) *rho*

```
    = (2^(bits-needed B) - 1) - bits-val (bits-needed B) CostBit rho
   by (rule pb-sum-neg-bits-val[OF rho01])
 have ge: pb-sum (map (λi. (2^i, Neg (CostBit i))) [0..<bits-needed B]) rho
    ≥ 2^(bits-needed B) - 1
   using sat by (simp add: neg-cost-ge-one-def satisfies-def)
 have lt: bits-val (bits-needed B) CostBit rho < 2^(bits-needed B)
   by (rule bits-val-lt) (use rho01 in blast)
 show ?thesis using neg-sum ge lt by linarith
qed
```

6.2.7 The initial state lemma (paper Lemma 3)

```
lemma astar-init-semantic:
 assumes rho01: ∀x. rho x = 0 ∨ rho x = 1
   and mI: models (encode-init Πe) rho
   and mC: models (circuit-constraints astar-circuit) rho
   and rI: rho ReifI = 1
   and bits0: satisfies (neg-cost-ge-one B) rho
 shows rho out-name = 1
proof -
 have c0: bits-val (bits-needed B) CostBit rho = 0
   by (rule neg-cost-ge-one-zero[OF rho01 bits0])
 have sv-e: ∀w ∈ vars Πe. rho (StateVar w) = (if w ∈ init Πe then 1 else 0)
   using encode-init-forces[OF rho01 mI fin-vars-e init-sub-e] rI by blast
 have sv: ∀v ∈ vars Π. rho (StateVar (Inl v)) = (if v ∈ init Π then 1 else 0)
   using sv-e state-descr-translate unfolding init-e by blast
 obtain i0 where i0-lt: i0 < n-cl and i0-s: cl-state i0 = init Π
   and i0-g: cl-g i0 = 0
   using closed-mem-nth[OF closed-init] by blast
 have k1: rho (k-name i0) = 1
 proof -
   have clip B (int (cl-g i0)) = 0 by (simp add: i0-g)
   then show ?thesis using kg-forces[OF rho01 mC i0-lt] by simp
 qed
 have cl1: rho (cl-name i0) = 1
   using clg-forces[OF rho01 mC i0-lt] k1 sv i0-s by simp
 have Pos (cl-name i0) ∈ set out-lits
   unfolding out-lits-def using i0-lt by simp
 moreover have eval-lit (Pos (cl-name i0)) rho = 1
   using cl1 by (simp add: eval-lit-def)
 ultimately show ?thesis
   using outg-forces[OF rho01 mC] by blast
qed
```

6.2.8 The goal lemma (paper Lemma 4)

```
lemma astar-goal-semantic:
 assumes rho01: ∀x. rho x = 0 ∨ rho x = 1
   and mG: models (encode-goal Πe) rho
   and mC: models (circuit-constraints astar-circuit) rho
```

```
    and rG: rho ReifG = 1
    and out1: rho out-name = 1
  shows bits-val (bits-needed B) CostBit rho ≥ B
proof −
  have gv: ∀ w ∈ goal Π e. rho (StateVar w) = 1
    using encode-goal-forces[OF rho01 mG fin-goal-e] rG by blast
  obtain l where l-in: l ∈ set out-lits and l1: eval-lit l rho = 1
    using outg-forces[OF rho01 mC] out1 by blast
  from l-in consider
    (ClosedGate) i where i < n-cl and l = Pos (cl-name i)
  | (OpenState) s where s ∈ set open-list and l = hc-out HC (Inl ‘ s)
    unfolding out-lits-def by auto
  then show ?thesis
  proof cases
    case ClosedGate
    have cl1: rho (cl-name i) = 1
      using l1 ClosedGate by (simp add: eval-lit-def)
    have k1: rho (k-name i) = 1
     and sv: ∀ v ∈ vars Π. rho (StateVar (Inl v)) = (if v ∈ cl-state i then 1 else 0)
      using clg-forces[OF rho01 mC ClosedGate(1)] cl1 by auto
    show ?thesis
    proof (cases is-goal-state Π (cl-state i))
      case True
      have cl-g i ≥ B
        using closed-goal-ge closed-nth-in[OF ClosedGate(1)] True by auto
      then have clip B (int (cl-g i)) = B by simp
     then show ?thesis using kg-forces[OF rho01 mC ClosedGate(1)] k1 by simp
    next
      case False
      then obtain v where vG: v ∈ goal Π and v-not: v ∉ cl-state i
        unfolding is-goal-state-def by auto
      have vV: v ∈ vars Π using vG goal-sub by auto
      have rho (StateVar (Inl v)) = 0 using sv vV v-not by simp
      moreover have rho (StateVar (Inl v)) = 1
        using gv vG unfolding goal-e by auto
      ultimately show ?thesis by simp
    qed
  next
    case OpenState
    show ?thesis
      by (rule hc-goal-lemmaD[OF hc-ok-goal[OF OpenState(1)] rho01
            models-hc-constraints[OF mC] gv l1[unfolded OpenState(2)]])
  qed
qed
```

6.2.9 The inductivity lemma (paper Lemmas 5–7)

Any 0/1 model of the transition encoding together with both lifted copies of the circuit that selects an action ($r_T = 1$) and satisfies the unprimed output

gate also satisfies the primed output gate. The proof follows the paper: the selected action is applicable in the closed state of the witnessing gate, and the successor is covered by the closed list, by duplicate detection, or by an open state whose heuristic state lemma (in primed variables) fires. Models of the lifted circuits are analysed by precomposition with the renamings.

lemma *astar-ind-semantic*:
  assumes *rho01*: $\forall x.$ *rho* $x = 0 \vee$ *rho* $x = 1$
    and *mT*: *models* (*encode-transition cas* (*vars* Πe) *B*) *rho*
    and *mO*: *models* (*circuit-constraints* (*orig-circuit astar-circuit*)) *rho*
    and *mP*: *models* (*circuit-constraints* (*primed-circuit astar-circuit*)) *rho*
    and *mB*: *models* (*encode-cost-ge B*) *rho*
    and *rT*: *rho ReifT* = *1*
    and *outO*: *rho* (*map-pvar Original out-name*) = *1*
  shows *rho* (*primed-pvar-map out-name*) = *1*
proof −
  define *rho-o* **where** *rho-o* = *rho* ∘ *map-pvar Original*
  define *rho-p* **where** *rho-p* = *rho* ∘ *primed-pvar-map*
  have *rho-o01*: $\forall x.$ *rho-o* $x = 0 \vee$ *rho-o* $x = 1$
    unfolding *rho-o-def* **by** (*rule rho01-comp*[*OF rho01*])
  have *rho-p01*: $\forall x.$ *rho-p* $x = 0 \vee$ *rho-p* $x = 1$
    unfolding *rho-p-def* **by** (*rule rho01-comp*[*OF rho01*])
  have *mO′*: *models* (*circuit-constraints astar-circuit*) *rho-o*
   using *mO models-circuit-constraints-lift*[*of map-pvar Original astar-circuit rho*]
    unfolding *rho-o-def orig-circuit-def* **by** *blast*
  have *mP′*: *models* (*circuit-constraints astar-circuit*) *rho-p*
    using *mP models-circuit-constraints-lift*[*of primed-pvar-map astar-circuit rho*]
    unfolding *rho-p-def primed-circuit-def* **by** *blast*
  define *c* **where** *c* = *bits-val* (*bits-needed B*) *CostBit rho*
  define *c′* **where** *c′* = *bits-val* (*bits-needed B*) *PrimedCostBit rho*
  have *c-o*: *bits-val* (*bits-needed B*) *CostBit rho-o* = *c*
    by (*simp add*: *rho-o-def c-def bits-val-def*)
  have *c-p*: *bits-val* (*bits-needed B*) *CostBit rho-p* = *c′*
    by (*simp add*: *rho-p-def c′-def bits-val-def*)
  — The selected action.
  have *mSel*: *models* (*action-selection-reif* (*action-reifs cas*)) *rho*
    by (*rule trans-sel*[*OF mT*])
  obtain *rA* **where** *rA-in*: *rA* ∈ *set* (*action-reifs cas*) **and** *rA1*: *eval-lit rA rho* = *1*
    using *action-selection-forces*[*OF rho01 mSel*] *rT* **by** *blast*
  obtain *j* **where** *j-lt*: *j* < *length cas* **and** *rA-eq*: *rA* = *Pos* (*ReifAction j*)
    using *rA-in* **unfolding** *action-reifs-def* **by** *auto*
  have *satAC*: *satisfies* (*action-constraint* (*Pos* (*ReifAction j*)) (*cas* ! *j*) (*vars* Πe) *B*) *rho*
    by (*rule trans-action-constraint*[*OF mT j-lt*])
  obtain *a* **where** *aΠ*: *a* ∈ *acts* Π **and** *caj*: *cas* ! *j* = *embed-act a*
    using *nth-mem*[*OF j-lt*] *cas-eq* **unfolding** *acts-embed* **by** *auto*
  have *preS*: *pre* (*cas* ! *j*) ⊆ *vars* Πe
    and *addS*: *add* (*cas* ! *j*) ⊆ *vars* Πe

```
    and delS: del (cas ! j) ⊆ vars Πe
    using caj bspec[OF acts-states-sub aΠ]
    by (auto simp: pre-embed add-embed del-embed vars-e)
  have outj: eval-lit (Pos (ReifAction j)) rho = 1
    using rA1 rA-eq by simp
  note ext = action-constraint-extract[OF rho01 satAC outj fin-vars-e preS addS
delS]
  have delta1: rho (ReifDeltaCost (cost (cas ! j))) = 1 using ext by blast
  have bound0: rho (ReifPrimedCostGe B) = 0 using ext by blast
  have preO: ∀ w ∈ pre (cas ! j). rho (StateVar (Original w)) = 1 using ext by
blast
  have addP: ∀ w ∈ add (cas ! j) − del (cas ! j). rho (StateVar (Primed w)) = 1
    using ext by blast
  have delP: ∀ w ∈ del (cas ! j) − add (cas ! j). rho (StateVar (Primed w)) = 0
    using ext by blast
  have frameO: ∀ w ∈ vars Πe − evars (cas ! j). rho (ReifEq (Original w)) = 1
    using ext by blast
  — Cost step and cost bound for the transition.
  have mD: models (encode-delta-cost (cost (cas ! j)) (bits-needed B)) rho
    by (rule trans-delta[OF mT nth-mem[OF j-lt]])
  have c'-eq: c' = c + cost a
    using encode-delta-cost-forces[OF rho01 mD] delta1
    by (simp add: c-def c'-def caj cost-embed)
  have c'-lt-B: c' < B
  proof −
    have rho (ReifPrimedCostGe B) = 1 ⟷ c' ≥ B
      using encode-cost-ge-primed-forces[OF rho01 trans-primed-ge[OF mT]]
      by (simp add: c'-def)
    then show ?thesis using bound0 by auto
  qed
  — Common final step: a true output literal on the primed side suffices.
  have to-out: ⋀lit. lit ∈ set out-lits ⟹ eval-lit lit rho-p = 1
      ⟹ rho (primed-pvar-map out-name) = 1
  proof −
    fix lit assume lit ∈ set out-lits and eval-lit lit rho-p = 1
    then have rho-p out-name = 1
      using outg-forces[OF rho-p01 mP'] by blast
    then show rho (primed-pvar-map out-name) = 1 by (simp add: rho-p-def)
  qed
  — Disjunction over the unprimed output gate.
  have outO': rho-o out-name = 1 using outO by (simp add: rho-o-def)
  obtain l where l-in: l ∈ set out-lits and l1: eval-lit l rho-o = 1
    using outg-forces[OF rho-o01 mO'] outO' by blast
  from l-in consider
    (ClosedGate) i where i < n-cl and l = Pos (cl-name i)
  | (OpenState) s where s ∈ set open-list and l = hc-out HC (Inl ‘ s)
    unfolding out-lits-def by auto
  then show ?thesis
  proof cases
```

case *OpenState*
have *outO-hc*: *eval-lit* (*map-literal* (*map-pvar Original*) (*hc-out HC* (*Inl* ‘ *s*))) *rho* = *1*
using *l1* **unfolding** *OpenState*(*2*) **by** (*simp add*: *eval-lit-map-literal rho-o-def*)
have *m-hc-o*: *models* (*circuit-constraints*
(*orig-circuit* (*hc-gates HC*, *hc-out HC* (*Inl* ‘ *s*)))) *rho*
proof −
have *models* (*circuit-constraints* (*hc-gates HC*, *hc-out HC* (*Inl* ‘ *s*))) *rho-o*
using *models-hc-constraints*[*OF mO'*]
by (*simp add*: *hc-constraints-eq-circuit*[*of HC hc-out HC* (*Inl* ‘ *s*), *symmetric*])
then show *?thesis*
using *models-circuit-constraints-lift*[*of map-pvar Original*
(*hc-gates HC*, *hc-out HC* (*Inl* ‘ *s*)) *rho*]
unfolding *rho-o-def orig-circuit-def* **by** *blast*
qed
have *m-hc-p*: *models* (*circuit-constraints*
(*primed-circuit* (*hc-gates HC*, *hc-out HC* (*Inl* ‘ *s*)))) *rho*
proof −
have *models* (*circuit-constraints* (*hc-gates HC*, *hc-out HC* (*Inl* ‘ *s*))) *rho-p*
using *models-hc-constraints*[*OF mP'*]
by (*simp add*: *hc-constraints-eq-circuit*[*of HC hc-out HC* (*Inl* ‘ *s*), *symmetric*])
then show *?thesis*
using *models-circuit-constraints-lift*[*of primed-pvar-map*
(*hc-gates HC*, *hc-out HC* (*Inl* ‘ *s*)) *rho*]
unfolding *rho-p-def primed-circuit-def* **by** *blast*
qed
have *prim-hc*: *eval-lit* (*map-literal primed-pvar-map* (*hc-out HC* (*Inl* ‘ *s*))) *rho* = *1*
by (*rule hc-ind-lemmaD*[*OF hc-ok-ind*[*OF OpenState*(*1*)] *rho01 m-hc-o m-hc-p mT mB rT outO-hc*])
have *eval-lit* (*hc-out HC* (*Inl* ‘ *s*)) *rho-p* = *1*
using *prim-hc* **by** (*simp add*: *eval-lit-map-literal rho-p-def*)
moreover have *hc-out HC* (*Inl* ‘ *s*) ∈ *set out-lits*
unfolding *out-lits-def* **using** *OpenState*(*1*) **by** *auto*
ultimately show *?thesis* **by** (*rule to-out*[*rotated*])
next
case *ClosedGate*
define *s* **where** *s* = *cl-state i*
define *g* **where** *g* = *cl-g i*
have *cl1*: *rho-o* (*cl-name i*) = *1*
using *l1 ClosedGate* **by** (*simp add*: *eval-lit-def*)
have *k1o*: *rho-o* (*k-name i*) = *1*
and *svO*: ∀ *v* ∈ *vars* Π. *rho-o* (*StateVar* (*Inl v*)) = (*if v* ∈ *s then 1 else 0*)
using *clg-forces*[*OF rho-o01 mO' ClosedGate*(*1*)] *cl1* **unfolding** *s-def* **by** *auto*
have *c-ge*: *c* ≥ *clip B* (*int g*)
using *kg-forces*[*OF rho-o01 mO' ClosedGate*(*1*)] *k1o c-o* **unfolding** *g-def* **by** *simp*
— The selected action is applicable in *s*.

```
    have appl: applicable a s
    proof −
      have pre a ⊆ s
      proof
        fix v assume vpre: v ∈ pre a
        have vV: v ∈ vars Π using vpre aΠ acts-states-sub by auto
        have Inl v ∈ pre (cas ! j) using vpre by (simp add: caj pre-embed)
        then have rho (StateVar (Original (Inl v))) = 1 using preO by blast
        then have one: rho-o (StateVar (Inl v)) = 1 by (simp add: rho-o-def)
        have eq: rho-o (StateVar (Inl v)) = (if v ∈ s then 1 else 0)
          using svO vV by blast
        show v ∈ s using one eq by (cases v ∈ s) auto
      qed
      then show ?thesis by (simp add: applicable-def)
    qed
    define s′ where s′ = successor a s
    — The primed state variables encode the successor exactly.
    have svP: ∀ v ∈ vars Π. rho-p (StateVar (Inl v)) = (if v ∈ s′ then 1 else 0)
    proof
      fix v assume vV: v ∈ vars Π
      have rp: rho-p (StateVar (Inl v)) = rho (StateVar (Primed (Inl v)))
        by (simp add: rho-p-def)
       consider (AddC) v ∈ add a | (DelC) v ∈ del a v ∉ add a | (FrameC) v ∉
evars a
        by (auto simp: evars-def)
      then show rho-p (StateVar (Inl v)) = (if v ∈ s′ then 1 else 0)
      proof cases
        case AddC
        have v ∉ del a using AddC acts-disjoint aΠ by auto
        then have Inl v ∈ add (cas ! j) − del (cas ! j)
          using AddC by (auto simp: caj add-embed del-embed)
        then have rho (StateVar (Primed (Inl v))) = 1 using addP by blast
        moreover have v ∈ s′ using AddC by (simp add: s′-def successor-def)
        ultimately show ?thesis using rp by simp
      next
        case DelC
        have Inl v ∈ del (cas ! j) − add (cas ! j)
          using DelC by (auto simp: caj add-embed del-embed)
        then have rho (StateVar (Primed (Inl v))) = 0 using delP by blast
        moreover have v ∉ s′ using DelC by (simp add: s′-def successor-def)
        ultimately show ?thesis using rp by simp
      next
        case FrameC
        have w-in: Inl v ∈ vars Πe − evars (cas ! j)
          using FrameC vV by (auto simp: vars-e caj evars-embed)
       then have eq1: rho (ReifEq (Original (Inl v))) = 1 using frameO by blast
        have meq: models (encode-eq-var (Inl v)) rho
          by (rule trans-eq-var[OF mT]) (use vV in ‹simp add: vars-e›)
        have eq2: rho (StateVar (Original (Inl v))) = rho (StateVar (Primed (Inl
```

```
v)))
        using encode-eq-var-forces[OF rho01 meq] eq1 by simp
      have eq3: rho (StateVar (Original (Inl v))) = (if v ∈ s then 1 else 0)
        using bspec[OF svO vV] by (simp add: rho-o-def)
      have eq4: v ∈ s' ⟷ v ∈ s
        using FrameC by (auto simp: s'-def successor-def evars-def)
      show ?thesis using rp eq2 eq3 eq4 by simp
    qed
  qed
  — Case analysis along the expansion-closure condition.
  have pair-in: (s, g) ∈ set closed-list
    using closed-nth-in[OF ClosedGate(1)] by (simp add: s-def g-def)
  from expansion pair-in aΠ appl consider
    (GoalCase) is-goal-state Π s and g ≥ B
  | (ClosedSucc) g'' where (s', g'') ∈ set closed-list and g'' ≤ g + cost a
  | (OpenSucc) s' ∈ set open-list and
      int (g + cost a) ≥ int B − int (hc-h HC (Inl ‘ s'))
    unfolding s'-def by blast
  then show ?thesis
  proof cases
    case GoalCase
    have clip B (int g) = B using GoalCase(2) by simp
    then have c ≥ B using c-ge by simp
    then have c' ≥ B using c'-eq by simp
    then have False using c'-lt-B by simp
    then show ?thesis ..
  next
    case ClosedSucc
    obtain i'' where i''-lt: i'' < n-cl and i''-s: cl-state i'' = s'
      and i''-g: cl-g i'' = g''
      using closed-mem-nth[OF ClosedSucc(1)] by blast
    have c'-ge: c' ≥ clip B (int (cl-g i''))
    proof −
      have clip B (int g'') ≤ clip B (int (g + cost a))
        by (rule clip-mono) (use ClosedSucc(2) in simp)
      also have ... = clip B (int g + int (cost a)) by simp
      also have ... ≤ clip B (int g) + cost a by (rule clip-add-le)
      also have ... ≤ c + cost a using c-ge by simp
      also have ... = c' using c'-eq by simp
      finally show ?thesis by (simp add: i''-g)
    qed
    have k1p: rho-p (k-name i'') = 1
      using kg-forces[OF rho-p01 mP' i''-lt] c'-ge c-p by simp
    have cl1p: rho-p (cl-name i'') = 1
      using clg-forces[OF rho-p01 mP' i''-lt] k1p svP i''-s by simp
    have Pos (cl-name i'') ∈ set out-lits
      unfolding out-lits-def using i''-lt by simp
    moreover have eval-lit (Pos (cl-name i'')) rho-p = 1
      using cl1p by (simp add: eval-lit-def)
```

```
    ultimately show ?thesis by (rule to-out)
  next
    case OpenSucc
    define h where h = hc-h HC (Inl ‘ s′)
    have m-hc-p: models (circuit-constraints
      (primed-circuit (hc-gates HC, hc-out HC (Inl ‘ s′)))) rho
    proof −
      have models (circuit-constraints (hc-gates HC, hc-out HC (Inl ‘ s′))) rho-p
        using models-hc-constraints[OF mP′]
        by (simp add: hc-constraints-eq-circuit[of HC hc-out HC (Inl ‘ s′), sym-
metric])
      then show ?thesis
        using models-circuit-constraints-lift[of primed-pvar-map
          (hc-gates HC, hc-out HC (Inl ‘ s′)) rho]
        unfolding rho-p-def primed-circuit-def by blast
    qed
    have svP-e: ∀ w ∈ vars Π e. rho (StateVar (Primed w)) = (if w ∈ Inl ‘ s′ then
1 else 0)
    proof
      fix w assume w ∈ vars Π e
      then obtain v where wv: w = Inl v and vV: v ∈ vars Π
        by (auto simp: vars-e)
      have rho (StateVar (Primed (Inl v))) = rho-p (StateVar (Inl v))
        by (simp add: rho-p-def)
      then show rho (StateVar (Primed w)) = (if w ∈ Inl ‘ s′ then 1 else 0)
        using bspec[OF svP vV] wv by (simp add: inj-image-mem-iff)
    qed
    have cb: bits-val (bits-needed B) PrimedCostBit rho
      ≥ clip B (int B − int (hc-h HC (Inl ‘ s′)))
    proof −
      have clip B (int B − int h) ≤ clip B (int (g + cost a))
        by (rule clip-mono) (use OpenSucc(2) in ‹simp add: h-def›)
      also have ... = clip B (int g + int (cost a)) by simp
      also have ... ≤ clip B (int g) + cost a by (rule clip-add-le)
      also have ... ≤ c + cost a using c-ge by simp
      also have ... = c′ using c′-eq by simp
      finally show ?thesis by (simp add: c′-def h-def)
    qed
    have prim-hc: eval-lit (map-literal primed-pvar-map (hc-out HC (Inl ‘ s′)))
rho = 1
      by (rule hc-state-lemma-primed[OF hc-ok-state[OF OpenSucc(1)] rho01
m-hc-p svP-e cb])
    have eval-lit (hc-out HC (Inl ‘ s′)) rho-p = 1
      using prim-hc by (simp add: eval-lit-map-literal rho-p-def)
    moreover have hc-out HC (Inl ‘ s′) ∈ set out-lits
      unfolding out-lits-def using OpenSucc(1) by auto
    ultimately show ?thesis by (rule to-out[rotated])
  qed
qed
```

qed

6.2.10 Gate names: distinctness and freshness

lemma *fst-kg*: *fst* (*kg i*) = *Pos* (*k-name i*)
by (*simp add*: *kg-def k-gate-def*)

lemma *fst-clg*: *fst* (*clg i*) = *Pos* (*cl-name i*)
by (*simp add*: *clg-def*)

lemma *fst-outg*: *fst outg* = *Pos out-name*
by (*simp add*: *outg-def*)

lemma *hc-name-form*:
g ∈ *set* (*hc-gates HC*) ⟹ ∃ *j*. *pvar-of-lit* (*fst g*) = *ReifCert* (*Inr* (*2* ∗ *j* + *1*))
using *hc-names* **by** (*cases g*) (*fastforce simp*: *pvar-of-lit-def*)

lemma *names-eq*:
map (λ*g*. *pvar-of-lit* (*fst g*)) *astar-gates*
= *map k-name* [*0*..<*n-cl*] @ *map cl-name* [*0*..<*n-cl*]
@ *map* (λ*g*. *pvar-of-lit* (*fst g*)) (*hc-gates HC*) @ [*out-name*]
by (*simp add*: *astar-gates-def fst-kg fst-clg fst-outg pvar-of-lit-def o-def*)

lemma *parity-neq*: (*2*::*nat*) ∗ *m* ≠ *2* ∗ *j* + *1*
by *presburger*

lemma *hc-names-odd-set*:
assumes *x* ∈ *set* (*map* (λ*g*. *pvar-of-lit* (*fst g*)) (*hc-gates HC*))
shows ∃ *j*. *x* = *ReifCert* (*Inr* (*2* ∗ *j* + *1*))
proof −
obtain *g* **where** *g-in*: *g* ∈ *set* (*hc-gates HC*) **and** *x-eq*: *x* = *pvar-of-lit* (*fst g*)
using *assms* **by** *auto*
show *?thesis* **using** *hc-name-form*[*OF g-in*] *x-eq* **by** *simp*
qed

lemma *distinct-gate-names*: *distinct* (*map* (λ*g*. *pvar-of-lit* (*fst g*)) *astar-gates*)
proof −
have *d1*: *distinct* (*map k-name* [*0*..<*n-cl*])
by (*auto simp*: *distinct-map inj-on-def k-name-def*)
have *d2*: *distinct* (*map cl-name* [*0*..<*n-cl*])
by (*auto simp*: *distinct-map inj-on-def cl-name-def*)
have *d3*: *distinct* (*map* (λ*g*. *pvar-of-lit* (*fst g*)) (*hc-gates HC*))
proof −
have (λ(*r*, *cs*, *A*). *pvar-of-lit r*) = (λ*g* :: (′*v* + *nat*) *pvar literal* ×
(*nat* × (′*v* + *nat*) *pvar literal*) *list* × *nat*. *pvar-of-lit* (*fst g*))
by (*rule ext*) (*simp add*: *split-beta*)
then show *?thesis* **using** *hc-distinct* **by** *simp*
qed
have *inK*: ⋀*x*. *x* ∈ *set* (*map k-name* [*0*..<*n-cl*]) ⟹ ∃ *i*<*n-cl*. *x* = *ReifCert* (*Inr*

```
(2 * i))
    by (auto simp: k-name-def)
  have inCl: ⋀x. x ∈ set (map cl-name [0..<n-cl]) ⟹ ∃i<n-cl. x = ReifCert
(Inr (2 * (n-cl + i)))
    by (auto simp: cl-name-def)
  have disj-k-cl: set (map k-name [0..<n-cl]) ∩ set (map cl-name [0..<n-cl]) = {}
    by (auto simp: k-name-def cl-name-def)
  have disj-k-hc: set (map k-name [0..<n-cl])
     ∩ set (map (λg. pvar-of-lit (fst g)) (hc-gates HC)) = {}
  proof −
    { fix x assume xK: x ∈ set (map k-name [0..<n-cl])
        and xH: x ∈ set (map (λg. pvar-of-lit (fst g)) (hc-gates HC))
      obtain i where xi: x = ReifCert (Inr (2 * i)) using inK[OF xK] by blast
     obtain j where xj: x = ReifCert (Inr (2 * j + 1)) using hc-names-odd-set[OF
xH] by blast
      from xi xj have 2 * i = 2 * j + 1 by simp
      then have False by presburger }
    then show ?thesis by blast
  qed
  have disj-k-out: out-name ∉ set (map k-name [0..<n-cl])
    by (auto simp: k-name-def out-name-def)
  have disj-cl-hc: set (map cl-name [0..<n-cl])
     ∩ set (map (λg. pvar-of-lit (fst g)) (hc-gates HC)) = {}
  proof −
    { fix x assume xC: x ∈ set (map cl-name [0..<n-cl])
        and xH: x ∈ set (map (λg. pvar-of-lit (fst g)) (hc-gates HC))
      obtain i where xi: x = ReifCert (Inr (2 * (n-cl + i))) using inCl[OF xC]
by blast
     obtain j where xj: x = ReifCert (Inr (2 * j + 1)) using hc-names-odd-set[OF
xH] by blast
      from xi xj have 2 * (n-cl + i) = 2 * j + 1 by simp
      then have False by presburger }
    then show ?thesis by blast
  qed
  have disj-cl-out: out-name ∉ set (map cl-name [0..<n-cl])
    by (auto simp: cl-name-def out-name-def)
  have disj-hc-out: out-name ∉ set (map (λg. pvar-of-lit (fst g)) (hc-gates HC))
  proof
    assume out-name ∈ set (map (λg. pvar-of-lit (fst g)) (hc-gates HC))
    then obtain j where out-name = ReifCert (Inr (2 * j + 1))
      using hc-names-odd-set by blast
    then have 2 * (2 * n-cl) = 2 * j + 1 by (simp add: out-name-def)
    then show False by presburger
  qed
  show ?thesis
    unfolding names-eq
    using d1 d2 d3 disj-k-cl disj-k-hc disj-k-out disj-cl-hc disj-cl-out disj-hc-out
    by auto
qed
```

lemma *distinct-reif-astar*: *distinct-reif-vars astar-circuit*
proof −
have *distinct* (*map* (λ*g*. *pvar-of-lit* (*fst g*)) *astar-gates*)
by (*rule distinct-gate-names*)
then show *?thesis*
unfolding *distinct-reif-vars-def astar-circuit-def fst-conv Let-def*
by (*simp add*: *distinct-conv-nth*)
qed

lemma *names-are-cert*:
v ∈ *circuit-reif-pvars astar-circuit* ⟹ ∃ *w*. *v* = *ReifCert w*
proof −
assume *v* ∈ *circuit-reif-pvars astar-circuit*
then have *v* ∈ *set* (*map* (λ*g*. *pvar-of-lit* (*fst g*)) *astar-gates*)
unfolding *circuit-reif-pvars-def astar-circuit-def* **by** *force*
then consider
v ∈ *set* (*map k-name* [*0*..<*n-cl*]) | *v* ∈ *set* (*map cl-name* [*0*..<*n-cl*])
| *v* ∈ *set* (*map* (λ*g*. *pvar-of-lit* (*fst g*)) (*hc-gates HC*)) | *v* = *out-name*
unfolding *names-eq* **by** *auto*
then show ∃ *w*. *v* = *ReifCert w*
proof *cases*
case *1* **then show** *?thesis* **by** (*auto simp*: *k-name-def*)
next
case *2* **then show** *?thesis* **by** (*auto simp*: *cl-name-def*)
next
case *3* **then show** *?thesis* **using** *hc-names-odd-set* **by** *blast*
next
case *4* **then show** *?thesis* **by** (*simp add*: *out-name-def*)
qed
qed

lemma *astar-freshness*:
∀ *v* ∈ *circuit-reif-pvars astar-circuit*.
¬ *is-input-pvar v* ∧ *v* ≠ *ReifI* ∧ (∀ *k*. *v* ≠ *ReifCostGe k*) ∧ *v* ≠ *ReifG* ∧
(∀ *k*. *v* ≠ *ReifDeltaCost k*) ∧ (∀ *k*. *v* ≠ *ReifDeltaCostLower k*) ∧
(∀ *k*. *v* ≠ *ReifDeltaCostUpper k*) ∧
(∀ *k*. *v* ≠ *ReifPrimedCostGe k*) ∧ *v* ≠ *ReifT* ∧
(∀ *u*. *v* ≠ *ReifGeq u*) ∧ (∀ *u*. *v* ≠ *ReifLeq u*) ∧ (∀ *u*. *v* ≠ *ReifEq u*) ∧
(∀ *i*. *v* ≠ *ReifAction i*) ∧ (∀ *i*. *v* ≠ *PrimedCostBit i*)
proof
fix *v* **assume** *v* ∈ *circuit-reif-pvars astar-circuit*
then obtain *w* **where** *v* = *ReifCert w* **using** *names-are-cert* **by** *blast*
then show ¬ *is-input-pvar v* ∧ *v* ≠ *ReifI* ∧ (∀ *k*. *v* ≠ *ReifCostGe k*) ∧ *v* ≠ *ReifG*
∧
(∀ *k*. *v* ≠ *ReifDeltaCost k*) ∧ (∀ *k*. *v* ≠ *ReifDeltaCostLower k*) ∧
(∀ *k*. *v* ≠ *ReifDeltaCostUpper k*) ∧
(∀ *k*. *v* ≠ *ReifPrimedCostGe k*) ∧ *v* ≠ *ReifT* ∧
(∀ *u*. *v* ≠ *ReifGeq u*) ∧ (∀ *u*. *v* ≠ *ReifLeq u*) ∧ (∀ *u*. *v* ≠ *ReifEq u*) ∧

$(\forall\, i.\ v \neq \mathit{ReifAction}\ i) \wedge (\forall\, i.\ v \neq \mathit{PrimedCostBit}\ i)$
by (*simp add*: *is-input-pvar-def*)
qed

6.2.11 Well-formedness of the assembled circuit

lemma *nth-in-take*: $j < i \Longrightarrow i \leq \mathit{length}\ \mathit{xs} \Longrightarrow \mathit{xs}\ !\ j \in \mathit{set}\ (\mathit{take}\ i\ \mathit{xs})$
by (*metis length-take min.absorb2 nth-mem nth-take*)

lemma *take-nth-ex*: $g' \in \mathit{set}\ (\mathit{take}\ j\ \mathit{xs}) \Longrightarrow \exists\, p.\ p < j \wedge p < \mathit{length}\ \mathit{xs} \wedge g' = \mathit{xs}\ !\ p$
by (*metis in-set-conv-nth length-take min-less-iff-conj nth-take*)

lemma *earlier-name*:
assumes $j < i$ **and** $i \leq \mathit{length}\ \mathit{astar\text{-}gates}$
shows $\mathit{pvar\text{-}of\text{-}lit}\ (\mathit{fst}\ (\mathit{astar\text{-}gates}\ !\ j)) \in \mathit{pvar\text{-}of\text{-}lit}\ `\ \mathit{fst}\ `\ \mathit{set}\ (\mathit{take}\ i\ \mathit{astar\text{-}gates})$
using *nth-in-take*[*OF assms*] **by** *force*

lemma *wf-astar-circuit*: *wf-circuit astar-circuit*
proof −
have *len*: *length astar-gates* = 2 ∗ *n-cl* + *n-hc* + 1 **by** (*rule length-astar-gates*)
have *main*: $\forall\, i < \mathit{length}\ \mathit{astar\text{-}gates}.$
$\mathit{pvar\text{-}of\text{-}lit}\ `\ \mathit{snd}\ `\ \mathit{set}\ (\mathit{fst}\ (\mathit{snd}\ (\mathit{astar\text{-}gates}\ !\ i)))$
$\subseteq \{x.\ \mathit{is\text{-}input\text{-}pvar}\ x\} \cup \mathit{pvar\text{-}of\text{-}lit}\ `\ \mathit{fst}\ `\ \mathit{set}\ (\mathit{take}\ i\ \mathit{astar\text{-}gates})$
proof (*intro allI impI*)
fix *i* **assume** *i-lt*: $i < \mathit{length}\ \mathit{astar\text{-}gates}$
have *i-le*: $i \leq \mathit{length}\ \mathit{astar\text{-}gates}$ **using** *i-lt* **by** *simp*
consider (*K*) $i < \mathit{n\text{-}cl}$ | (*CL*) $\mathit{n\text{-}cl} \leq i\ i < 2 * \mathit{n\text{-}cl}$
| (*HCC*) $2 * \mathit{n\text{-}cl} \leq i\ i < 2 * \mathit{n\text{-}cl} + \mathit{n\text{-}hc}$ | (*OUT*) $i = 2 * \mathit{n\text{-}cl} + \mathit{n\text{-}hc}$
using *i-lt len* **by** *linarith*
then show $\mathit{pvar\text{-}of\text{-}lit}\ `\ \mathit{snd}\ `\ \mathit{set}\ (\mathit{fst}\ (\mathit{snd}\ (\mathit{astar\text{-}gates}\ !\ i)))$
$\subseteq \{x.\ \mathit{is\text{-}input\text{-}pvar}\ x\} \cup \mathit{pvar\text{-}of\text{-}lit}\ `\ \mathit{fst}\ `\ \mathit{set}\ (\mathit{take}\ i\ \mathit{astar\text{-}gates})$
proof *cases*
case *K*
have *g*: *astar-gates* ! *i* = *kg i* **by** (*rule astar-gates-nth-k*[*OF K*])
show *?thesis*
unfolding *g kg-def k-gate-def*
by (*auto simp*: *k-gate-body-def pvar-of-lit-def is-input-pvar-def*)
next
case *CL*
define *j* **where** $j = i - \mathit{n\text{-}cl}$
have *j-lt*: $j < \mathit{n\text{-}cl}$ **and** *i-eq*: $i = \mathit{n\text{-}cl} + j$ **using** *CL* **by** (*auto simp*: *j-def*)
have *g*: *astar-gates* ! *i* = *clg j* **using** *astar-gates-nth-cl*[*OF j-lt*] *i-eq* **by** *simp*
have *kname-in*: *k-name j* ∈ *pvar-of-lit* ‘ *fst* ‘ *set* (*take i astar-gates*)
proof −
have *j* < *i* **using** *j-lt CL i-eq* **by** *linarith*
from *earlier-name*[*OF this i-le*] **show** *?thesis*
using *astar-gates-nth-k*[*OF j-lt*] *fst-kg* **by** (*simp add*: *pvar-of-lit-def*)
qed

show *?thesis*
unfolding *g clg-def*
using *kname-in*
by (*auto simp*: *cl-lits-def state-lits-exact-def pvar-of-lit-def is-input-pvar-def split*: *if-split-asm*)
next
case *HCC*
define *j* **where** *j* = *i* − *2* ∗ *n-cl*
have *j-lt*: *j* < *n-hc* **and** *i-eq*: *i* = *2* ∗ *n-cl* + *j* **using** *HCC* **by** (*auto simp*: *j-def*)
have *g*: *astar-gates* ! *i* = *hc-gates HC* ! *j*
using *astar-gates-nth-hc*[*OF j-lt*] *i-eq* **by** *simp*
obtain *r cs A* **where** *hg*: *hc-gates HC* ! *j* = (*r*, *cs*, *A*) **by** (*metis prod-cases3*)
have *body*: ∀ *x* ∈ *pvar-of-lit* ‘ *snd* ‘ *set cs*.
(∃ *v*. *x* = *StateVar v*) ∨ (∃ *jj*. *x* = *CostBit jj*)
∨ *x* ∈ *pvar-of-lit* ‘ *fst* ‘ *set* (*take j* (*hc-gates HC*))
proof −
have *case hc-gates HC* ! *j of* (*r*, *cs*, *A*) ⇒
(∀ *x* ∈ *pvar-of-lit* ‘ *snd* ‘ *set cs*.
(∃ *v*. *x* = *StateVar v*) ∨ (∃ *jj*. *x* = *CostBit jj*)
∨ *x* ∈ *pvar-of-lit* ‘ *fst* ‘ *set* (*take j* (*hc-gates HC*)))
using *hc-wf*[*rule-format*, *OF j-lt*[*unfolded n-hc-def*]] **.**
then show *?thesis* **by** (*simp add*: *hg*)
qed
have *earlier-sub*: *pvar-of-lit* ‘ *fst* ‘ *set* (*take j* (*hc-gates HC*))
⊆ *pvar-of-lit* ‘ *fst* ‘ *set* (*take i astar-gates*)
proof
fix *x* **assume** *x* ∈ *pvar-of-lit* ‘ *fst* ‘ *set* (*take j* (*hc-gates HC*))
then obtain *g′* **where** *g′-in*: *g′* ∈ *set* (*take j* (*hc-gates HC*))
and *x-eq*: *x* = *pvar-of-lit* (*fst g′*) **by** *auto*
obtain *p* **where** *p-lt*: *p* < *j* **and** *p-len*: *p* < *length* (*hc-gates HC*)
and *g′-eq*: *g′* = *hc-gates HC* ! *p*
using *take-nth-ex*[*OF g′-in*] **by** *blast*
have *pos*: *astar-gates* ! (*2* ∗ *n-cl* + *p*) = *hc-gates HC* ! *p*
using *astar-gates-nth-hc p-len n-hc-def* **by** *simp*
have *2* ∗ *n-cl* + *p* < *i* **using** *p-lt i-eq* **by** *simp*
from *earlier-name*[*OF this i-le*] **show** *x* ∈ *pvar-of-lit* ‘ *fst* ‘ *set* (*take i astar-gates*)
using *pos g′-eq x-eq* **by** *simp*
qed
show *?thesis*
proof (*intro subsetI*)
fix *x* **assume** *x* ∈ *pvar-of-lit* ‘ *snd* ‘ *set* (*fst* (*snd* (*astar-gates* ! *i*)))
then have *x-in*: *x* ∈ *pvar-of-lit* ‘ *snd* ‘ *set cs* **by** (*simp add*: *g hg*)
from *bspec*[*OF body x-in*]
show *x* ∈ {*x*. *is-input-pvar x*} ∪ *pvar-of-lit* ‘ *fst* ‘ *set* (*take i astar-gates*)
using *earlier-sub* **by** (*auto simp*: *is-input-pvar-def*)
qed
next

case *OUT*
have *g*: *astar-gates* ! *i* = *outg* **using** *astar-gates-nth-out OUT* **by** *simp*
have *cl-in*: ⋀*jj*. *jj* < *n-cl* ⟹ *cl-name jj* ∈ *pvar-of-lit* ‘ *fst* ‘ *set* (*take i astar-gates*)
proof −
fix *jj* **assume** *jj*: *jj* < *n-cl*
have *n-cl* + *jj* < *i* **using** *jj OUT* **by** *simp*
from *earlier-name*[*OF this i-le*] **show**
cl-name jj ∈ *pvar-of-lit* ‘ *fst* ‘ *set* (*take i astar-gates*)
using *astar-gates-nth-cl*[*OF jj*] *fst-clg* **by** (*simp add*: *pvar-of-lit-def*)
qed
have *hcl-in*: ⋀*s*. *s* ∈ *set open-list* ⟹
pvar-of-lit (*hc-out HC* (*Inl* ‘ *s*)) ∈ *pvar-of-lit* ‘ *fst* ‘ *set* (*take i astar-gates*)
proof −
fix *s* **assume** *sO*: *s* ∈ *set open-list*
have *hc-out HC* (*Inl* ‘ *s*) ∈ *fst* ‘ *set* (*hc-gates HC*) **using** *hc-out-in sO* **by** *blast*
then obtain *g′* **where** *g′-in*: *g′* ∈ *set* (*hc-gates HC*)
and *out-eq*: *hc-out HC* (*Inl* ‘ *s*) = *fst g′* **by** *auto*
obtain *p* **where** *p-len*: *p* < *length* (*hc-gates HC*) **and** *g′-eq*: *g′* = *hc-gates HC* ! *p*
using *g′-in* **by** (*metis in-set-conv-nth*)
have *pos*: *astar-gates* ! (2 ∗ *n-cl* + *p*) = *hc-gates HC* ! *p*
using *astar-gates-nth-hc p-len n-hc-def* **by** *simp*
have 2 ∗ *n-cl* + *p* < *i* **using** *p-len OUT n-hc-def* **by** *simp*
from *earlier-name*[*OF this i-le*] **show**
pvar-of-lit (*hc-out HC* (*Inl* ‘ *s*)) ∈ *pvar-of-lit* ‘ *fst* ‘ *set* (*take i astar-gates*)
using *pos g′-eq out-eq* **by** *simp*
qed
show *?thesis*
unfolding *g outg-def*
using *cl-in hcl-in* **by** (*fastforce simp*: *out-lits-def pvar-of-lit-def*)
qed
qed
have *out-cond*: *Pos* (*pvar-of-lit* (*Pos out-name*)) ∈ *fst* ‘ *set astar-gates*
∨ *Neg* (*pvar-of-lit* (*Pos out-name*)) ∈ *fst* ‘ *set astar-gates*
proof −
have *Pos out-name* ∈ *fst* ‘ *set astar-gates*
using *outg-in-gates fst-outg* **by** *force*
then show *?thesis* **by** (*simp add*: *pvar-of-lit-def*)
qed
show *?thesis*
unfolding *wf-circuit-def astar-circuit-def Let-def*
using *main out-cond* **by** (*auto simp*: *split-beta*)
qed

lemma *astar-body-pvars*:
assumes *g-in*: (*r*, *cs*, *A*) ∈ *set astar-gates*
and *x-in*: *x* ∈ *pvar-of-lit* ‘ *snd* ‘ *set cs*

```
  shows (∃ v. x = StateVar v) ∨ (∃ i. x = CostBit i) ∨ (∃ w. x = ReifCert w)
proof −
  from g-in consider
    (K) i where i < n-cl and (r, cs, A) = kg i
  | (CL) i where i < n-cl and (r, cs, A) = clg i
  | (HCC) (r, cs, A) ∈ set (hc-gates HC)
  | (OUT) (r, cs, A) = outg
    unfolding astar-gates-def by auto
  then show ?thesis
  proof cases
    case K
    then have cs = k-gate-body B by (simp add: kg-def k-gate-def)
    then show ?thesis using x-in by (auto simp: k-gate-body-def pvar-of-lit-def)
  next
    case CL
    then have cs = map (λl. (1, l)) (cl-lits i) by (simp add: clg-def)
    then show ?thesis using x-in
      by (auto simp: cl-lits-def state-lits-exact-def k-name-def pvar-of-lit-def
          split: if-split-asm)
  next
    case HCC
    then obtain j where j-len: j < length (hc-gates HC)
      and hg: hc-gates HC ! j = (r, cs, A)
      by (metis in-set-conv-nth)
    have body: ∀ x ∈ pvar-of-lit ‘ snd ‘ set cs.
        (∃ v. x = StateVar v) ∨ (∃ jj. x = CostBit jj)
        ∨ x ∈ pvar-of-lit ‘ fst ‘ set (take j (hc-gates HC))
    proof −
      have case hc-gates HC ! j of (r, cs, A) ⇒
          (∀ x ∈ pvar-of-lit ‘ snd ‘ set cs.
            (∃ v. x = StateVar v) ∨ (∃ jj. x = CostBit jj)
            ∨ x ∈ pvar-of-lit ‘ fst ‘ set (take j (hc-gates HC)))
        using hc-wf[rule-format, OF j-len] .
      then show ?thesis by (simp add: hg)
    qed
    from body x-in consider
        (SV) (∃ v. x = StateVar v) ∨ (∃ jj. x = CostBit jj)
      | (EN) x ∈ pvar-of-lit ‘ fst ‘ set (take j (hc-gates HC))
      by blast
    then show ?thesis
    proof cases
      case SV then show ?thesis by blast
    next
      case EN
      then obtain g′ where g′-tk: g′ ∈ set (take j (hc-gates HC))
        and x-eq: x = pvar-of-lit (fst g′) by auto
      have g′ ∈ set (hc-gates HC) using g′-tk by (rule in-set-takeD)
      then show ?thesis using hc-name-form x-eq by blast
    qed
```

```
  next
    case OUT
    then have cs = map (λl. (1, l)) out-lits by (simp add: outg-def)
    then have x ∈ pvar-of-lit ‘ set out-lits using x-in by auto
    then consider
        (CLN) i where i < n-cl and x = cl-name i
      | (HCN) s where s ∈ set open-list and x = pvar-of-lit (hc-out HC (Inl ‘ s))
      by (auto simp: out-lits-def pvar-of-lit-def)
    then show ?thesis
    proof cases
      case CLN then show ?thesis by (simp add: cl-name-def)
    next
      case HCN
      have hc-out HC (Inl ‘ s) ∈ fst ‘ set (hc-gates HC) using hc-out-in HCN(1)
by blast
      then obtain g′ where g′ ∈ set (hc-gates HC) and hc-out HC (Inl ‘ s) = fst
g′ by auto
      then show ?thesis using hc-name-form HCN(2) by fastforce
    qed
  qed
qed

lemma astar-no-pcb:
  ∀ (r, φ) ∈ set (fst astar-circuit). ∀ v ∈ constraint-pvars φ. ∀ i. v ≠ PrimedCostBit
i
proof −
  { fix r cs A v
    assume g-in: (r, cs, A) ∈ set astar-gates
      and v-in: v ∈ constraint-pvars (cs, A)
    have v ∈ pvar-of-lit ‘ snd ‘ set cs
      using v-in by (simp add: constraint-pvars-def)
    from astar-body-pvars[OF g-in this]
    have ∀ i. v ≠ PrimedCostBit i by auto }
  then show ?thesis
    unfolding astar-circuit-def by fastforce
qed
```

6.2.12 Output literal of the assembled circuit

```
lemma snd-circ: snd astar-circuit = Pos out-name
  by (simp add: astar-circuit-def)

lemma snd-orig-astar: snd (orig-circuit astar-circuit) = Pos (map-pvar Original
out-name)
  by (simp add: snd-circ)

lemma snd-primed-astar: snd (primed-circuit astar-circuit) = Pos (primed-pvar-map
out-name)
  by (simp add: snd-circ primed-circuit-def)
```

6.2.13 CPR derivability of the three certificate conditions

lemma *astar-init-cpr*:
cpr-derives
(*encode-init* Π*e* ∪ *circuit-constraints astar-circuit* ∪ *encode-cost-ge B* ∪
{*unit-clause* (*Pos ReifI*), *neg-cost-ge-one B*})
(*unit-clause* (*snd astar-circuit*))
proof (*rule semantic-to-cpr*)
show *snd* (*unit-clause* (*snd astar-circuit*)) ≥ (*1*::*nat*)
by (*simp add*: *unit-clause-def*)
show ∀ *rho*. (∀ *v*. *rho v* = *0* ∨ *rho v* = *1*) ⟶
models (*encode-init* Π*e* ∪ *circuit-constraints astar-circuit* ∪ *encode-cost-ge B*
∪
{*unit-clause* (*Pos ReifI*), *neg-cost-ge-one B*}) *rho* ⟶
satisfies (*unit-clause* (*snd astar-circuit*)) *rho*
proof (*intro allI impI*)
fix *rho* :: ('*v* + *nat*) *pvar* ⇒ *nat*
assume *rho01*: ∀ *v*. *rho v* = *0* ∨ *rho v* = *1*
and *m*: *models* (*encode-init* Π*e* ∪ *circuit-constraints astar-circuit* ∪
encode-cost-ge B ∪ {*unit-clause* (*Pos ReifI*), *neg-cost-ge-one B*}) *rho*
have *mI*: *models* (*encode-init* Π*e*) *rho*
and *mC*: *models* (*circuit-constraints astar-circuit*) *rho*
and *satRI*: *satisfies* (*unit-clause* (*Pos ReifI*)) *rho*
and *bits0*: *satisfies* (*neg-cost-ge-one B*) *rho*
using *m* **by** (*auto simp*: *models-def*)
have *rI*: *rho ReifI* = *1*
using *satRI* **by** (*simp add*: *unit-clause-pos-sat*[*OF rho01*])
have *rho out-name* = *1*
by (*rule astar-init-semantic*[*OF rho01 mI mC rI bits0*])
then show *satisfies* (*unit-clause* (*snd astar-circuit*)) *rho*
by (*simp add*: *snd-circ unit-clause-pos-sat*[*OF rho01*])
qed
qed

lemma *astar-goal-cpr*:
cpr-derives
(*encode-goal* Π*e* ∪ *circuit-constraints astar-circuit* ∪ *encode-cost-ge B* ∪
{*unit-clause* (*snd astar-circuit*), *unit-clause* (*Pos ReifG*)})
(*cost-ge-constraint B*)
proof (*rule semantic-to-cpr*)
show *snd* (*cost-ge-constraint B*) ≥ (*1*::*nat*)
using *B-pos* **by** (*simp add*: *cost-ge-constraint-def*)
show ∀ *rho*. (∀ *v*. *rho v* = *0* ∨ *rho v* = *1*) ⟶
models (*encode-goal* Π*e* ∪ *circuit-constraints astar-circuit* ∪ *encode-cost-ge B*
∪
{*unit-clause* (*snd astar-circuit*), *unit-clause* (*Pos ReifG*)}) *rho* ⟶
satisfies (*cost-ge-constraint B*) *rho*
proof (*intro allI impI*)
fix *rho* :: ('*v* + *nat*) *pvar* ⇒ *nat*
assume *rho01*: ∀ *v*. *rho v* = *0* ∨ *rho v* = *1*

```
      and m: models (encode-goal Πe ∪ circuit-constraints astar-circuit ∪
       encode-cost-ge B ∪ {unit-clause (snd astar-circuit), unit-clause (Pos ReifG)})
rho
    have mG: models (encode-goal Πe) rho
      and mC: models (circuit-constraints astar-circuit) rho
      and satOut: satisfies (unit-clause (snd astar-circuit)) rho
      and satRG: satisfies (unit-clause (Pos ReifG)) rho
      using m by (auto simp: models-def)
    have out1: rho out-name = 1
      using satOut by (simp add: snd-circ unit-clause-pos-sat[OF rho01])
    have rG: rho ReifG = 1
      using satRG by (simp add: unit-clause-pos-sat[OF rho01])
    have bits-val (bits-needed B) CostBit rho ≥ B
      by (rule astar-goal-semantic[OF rho01 mG mC rG out1])
    then show satisfies (cost-ge-constraint B) rho
      by (simp add: cost-ge-constraint-def satisfies-def pb-sum-bits-val)
  qed
qed

lemma astar-ind-cpr:
  cpr-derives
     (encode-transition cas (vars Πe) B ∪
      circuit-constraints (orig-circuit astar-circuit) ∪
      circuit-constraints (primed-circuit astar-circuit) ∪
      encode-cost-ge B ∪
      {unit-clause (snd (orig-circuit astar-circuit)), unit-clause (Pos ReifT)})
     (unit-clause (snd (primed-circuit astar-circuit)))
proof (rule semantic-to-cpr)
  show snd (unit-clause (snd (primed-circuit astar-circuit))) ≥ (1::nat)
    by (simp add: unit-clause-def)
  show ∀ rho. (∀ v. rho v = 0 ∨ rho v = 1) ⟶
      models (encode-transition cas (vars Πe) B ∪
        circuit-constraints (orig-circuit astar-circuit) ∪
        circuit-constraints (primed-circuit astar-circuit) ∪
        encode-cost-ge B ∪
       {unit-clause (snd (orig-circuit astar-circuit)), unit-clause (Pos ReifT)}) rho
⟶
      satisfies (unit-clause (snd (primed-circuit astar-circuit))) rho
  proof (intro allI impI)
    fix rho :: (('v + nat) var) pvar ⇒ nat
    assume rho01: ∀ v. rho v = 0 ∨ rho v = 1
      and m: models (encode-transition cas (vars Πe) B ∪
        circuit-constraints (orig-circuit astar-circuit) ∪
        circuit-constraints (primed-circuit astar-circuit) ∪
        encode-cost-ge B ∪
       {unit-clause (snd (orig-circuit astar-circuit)), unit-clause (Pos ReifT)}) rho
    have mT: models (encode-transition cas (vars Πe) B) rho
      and mO: models (circuit-constraints (orig-circuit astar-circuit)) rho
      and mP: models (circuit-constraints (primed-circuit astar-circuit)) rho
```

and *mB*: *models* (*encode-cost-ge B*) *rho*
and *satO*: *satisfies* (*unit-clause* (*snd* (*orig-circuit astar-circuit*))) *rho*
and *satRT*: *satisfies* (*unit-clause* (*Pos ReifT*)) *rho*
using *m* **by** (*auto simp*: *models-def*)
have *outO*: *rho* (*map-pvar Original out-name*) = *1*
using *satO* **unfolding** *snd-orig-astar* **by** (*simp add*: *unit-clause-pos-sat*[*OF rho01*])
have *rT*: *rho ReifT* = *1*
using *satRT* **by** (*simp add*: *unit-clause-pos-sat*[*OF rho01*])
have *rho* (*primed-pvar-map out-name*) = *1*
by (*rule astar-ind-semantic*[*OF rho01 mT mO mP mB rT outO*])
then show *satisfies* (*unit-clause* (*snd* (*primed-circuit astar-circuit*))) *rho*
by (*simp add*: *snd-primed-astar unit-clause-pos-sat*[*OF rho01*])
qed
qed

6.2.14 Main results: the A* snapshot yields a valid certificate

theorem *astar-certificate-valid*: *certificate-valid-cpr B* Π*e astar-cert*
proof −
have *fin-acts-e*: *finite* (*acts* Π*e*)
by (*simp add*: *acts-embed fin-acts*)
have *acts-sub*: *acts* Π*e* ⊆ *set cas*
using *cas-eq* **by** *simp*
have *cas-states*: ∀ *a* ∈ *set cas*. *pre a* ⊆ *vars* Π*e* ∧ *add a* ⊆ *vars* Π*e* ∧ *del a* ⊆ *vars* Π*e*
proof
fix *a′* **assume** *a′* ∈ *set cas*
then obtain *a0* **where** *a0-in*: *a0* ∈ *acts* Π **and** *a′-eq*: *a′* = *embed-act a0*
using *cas-eq* **unfolding** *acts-embed* **by** *auto*
have *pre a0* ⊆ *vars* Π ∧ *add a0* ⊆ *vars* Π ∧ *del a0* ⊆ *vars* Π
using *bspec*[*OF acts-states-sub a0-in*] **.**
then show *pre a′* ⊆ *vars* Π*e* ∧ *add a′* ⊆ *vars* Π*e* ∧ *del a′* ⊆ *vars* Π*e*
by (*auto simp*: *a′-eq pre-embed add-embed del-embed vars-e*)
qed
show *?thesis*
unfolding *certificate-valid-cpr-def Let-def*
using *fin-vars-e init-sub-e goal-sub-e fin-acts-e acts-sub cas-states*
wf-astar-circuit distinct-reif-astar astar-no-pcb astar-freshness
astar-init-cpr astar-goal-cpr astar-ind-cpr
by (*simp add*: *astar-cert-def*)
qed

theorem *astar-lower-bound*:
assumes *is-plan-for* Π *π*
shows *plan-cost π* ≥ *B*
by (*rule embedded-certificate-lower-bound*[*OF astar-certificate-valid assms*])

corollary *astar-optimal*:

assumes *is-plan-for* $\Pi\ \pi$ **and** *plan-cost* $\pi = B$
shows *optimal-plan* $\Pi\ \pi$
by (*rule embedded-certificate-optimality*[*OF astar-certificate-valid assms*])

end

end

7 Pattern Database Certificates

theory *PDB-Certificates*
imports *A-Star-Certificates*
begin

Proof-logging pattern database heuristics (paper equations (14)–(16) and Lemmas 8–13). A PDB heuristic for a pattern $P \subseteq V$ abstracts each state s to $\alpha(s) = s \cap P$ and returns the precomputed abstract goal distance $d(\alpha(s))$. The locale *pdb-heuristic* captures a PDB table over the abstract state space *Pow P*; the two semantic conditions on the table — goal states have distance 0, and the distance is consistent along abstract transitions — are exactly what the soundness of the generated certificate requires (the paper's "relies on the correctness and admissibility of the PDB heuristic"). From the table we assemble a *heuristic-cert* whose gates realize equations (14)–(16), with the K-gates of equation (15) inlined over the cost bits as everywhere else in this formalization, and prove it valid in the sense of Definition 4.

7.1 The PDB locale

locale *pdb-heuristic* =
fixes Πe :: *'u*::*linorder strips-task*
and B :: *nat*
and P :: *'u set*
and d :: *'u state* $\Rightarrow$ *nat*
and *Ss* :: *'u state list*
and *as* :: *'u action list*
and *nm* :: *nat* $\Rightarrow$ *'u*
assumes *fin-vars*: *finite* (*vars* Πe)
and *P-sub*: $P \subseteq$ *vars* Πe
and *Ss-set*: *set Ss* = *Pow P*
and *Ss-dist*: *distinct Ss*
and *as-states*: $\forall a \in$ *set as*. *pre a* $\subseteq$ *vars* $\Pi e \wedge$ *add a* $\subseteq$ *vars* $\Pi e \wedge$ *del a* $\subseteq$ *vars* Πe
and *as-disjoint*: $\forall a \in$ *set as*. *add a* $\cap$ *del a* = $\{\}$
and *d-goal*: $\bigwedge sa$. *sa* $\subseteq P \Longrightarrow$ *goal* $\Pi e \cap P \subseteq sa \Longrightarrow d\ sa = 0$
and *d-triangle*: $\bigwedge sa\ a$. *sa* $\subseteq P \Longrightarrow a \in$ *set as* $\Longrightarrow$ *pre a* $\cap P \subseteq sa \Longrightarrow$
$d\ sa \leq d\ ((sa - del\ a) \cup (add\ a \cap P)) + cost\ a$
and *nm-inj*: *inj nm*
begin

lemma *fin-P*: *finite P*
using *fin-vars P-sub* **by** (*rule rev-finite-subset*)

definition *n-s* :: *nat* **where**
n-s = *length Ss*

definition *sa-i* :: *nat* ⇒ *'u state* **where**
sa-i i = *Ss* ! *i*

lemma *sa-i-sub*: *i* < *n-s* ⟹ *sa-i i* ⊆ *P*
using *Ss-set nth-mem* **unfolding** *n-s-def sa-i-def* **by** *fastforce*

lemma *abs-mem-nth*:
assumes *sa* ⊆ *P*
shows ∃ *i*. *i* < *n-s* ∧ *sa-i i* = *sa*
proof −
have *sa* ∈ *set Ss* **using** *assms Ss-set* **by** *auto*
then show *?thesis*
unfolding *n-s-def sa-i-def* **by** (*auto simp*: *in-set-conv-nth*)
qed

7.1.1 Gate names

definition *abs-name* :: *nat* ⇒ *'u pvar* **where**
abs-name i = *ReifCert* (*nm i*)

definition *kk-name* :: *nat* ⇒ *'u pvar* **where**
kk-name i = *ReifCert* (*nm* (*n-s* + *i*))

definition *thr-name* :: *nat* ⇒ *'u pvar* **where**
thr-name i = *ReifCert* (*nm* (*2* ∗ *n-s* + *i*))

definition *pout-name* :: *'u pvar* **where**
pout-name = *ReifCert* (*nm* (*3* ∗ *n-s*))

7.1.2 Gates

Equation (14): the abstract-state gate is true iff the state variables restricted to the pattern encode exactly the abstract state.

definition *abs-lits* :: *nat* ⇒ *'u pvar literal list* **where**
abs-lits i =
map (λ*v*. *if v* ∈ *sa-i i then Pos* (*StateVar v*) *else Neg* (*StateVar v*))
(*sorted-list-of-set P*)

definition *absg* :: *nat* ⇒ *'u pvar literal* × (*nat* × *'u pvar literal*) *list* × *nat* **where**
absg i = (*Pos* (*abs-name i*), *map* (λ*l*. (*1*, *l*)) (*abs-lits i*), *length* (*abs-lits i*))

The K-gate part of equation (15): the clipped cost threshold for $B - d(s\alpha)$, inlined over the cost bits.

definition *kgg* :: *nat* ⇒ *'u pvar literal* × (*nat* × *'u pvar literal*) *list* × *nat* **where**
kgg i = *k-gate* (*Pos* (*kk-name i*)) *B* (*int B* − *int* (*d* (*sa-i i*)))

Equation (15): the conjunction of the abstract-state gate and its K-gate.

definition *thr-lits* :: *nat* ⇒ *'u pvar literal list* **where**
thr-lits i = [*Pos* (*abs-name i*), *Pos* (*kk-name i*)]

definition *thrg* :: *nat* ⇒ *'u pvar literal* × (*nat* × *'u pvar literal*) *list* × *nat* **where**
thrg i = (*Pos* (*thr-name i*), *map* (λ*l*. (*1*, *l*)) (*thr-lits i*), *length* (*thr-lits i*))

Equation (16): the output disjunction over all abstract states.

definition *pout-lits* :: *'u pvar literal list* **where**
pout-lits = *map* (λ*i*. *Pos* (*thr-name i*)) [*0*..<*n-s*]

definition *poutg* :: *'u pvar literal* × (*nat* × *'u pvar literal*) *list* × *nat* **where**
poutg = (*Pos pout-name*, *map* (λ*l*. (*1*, *l*)) *pout-lits*, *1*)

definition *pdb-gates* ::
(*'u pvar literal* × (*nat* × *'u pvar literal*) *list* × *nat*) *list* **where**
pdb-gates = *map absg* [*0*..<*n-s*] @ *map kgg* [*0*..<*n-s*] @ *map thrg* [*0*..<*n-s*] @ [*poutg*]

definition *pdb-cert* :: (*'u*) *heuristic-cert* **where**
pdb-cert = (| *hc-gates* = *pdb-gates*,
hc-out = (λ*s*. *Pos pout-name*),
hc-h = (λ*s*. *d* (*s* ∩ *P*)) |)

7.1.3 Basic structure of the gate list

lemma *length-pdb-gates*: *length pdb-gates* = *3* ∗ *n-s* + *1*
by (*simp add*: *pdb-gates-def*)

lemma *absg-in-gates*: *i* < *n-s* ⟹ *absg i* ∈ *set pdb-gates*
by (*simp add*: *pdb-gates-def*)

lemma *kgg-in-gates*: *i* < *n-s* ⟹ *kgg i* ∈ *set pdb-gates*
by (*simp add*: *pdb-gates-def*)

lemma *thrg-in-gates*: *i* < *n-s* ⟹ *thrg i* ∈ *set pdb-gates*
by (*simp add*: *pdb-gates-def*)

lemma *poutg-in-gates*: *poutg* ∈ *set pdb-gates*
by (*simp add*: *pdb-gates-def*)

lemma *models-pdb-gate*:
assumes *m*: *models* (*hc-constraints pdb-cert*) *rho*
and *g-in*: (*r*, *cs*, *A*) ∈ *set pdb-gates*
shows *models* (*reification r cs A*) *rho*
proof (*rule models-mono*[*OF m*])
show *reification r cs A* ⊆ *hc-constraints pdb-cert*

using *g-in* **unfolding** *hc-constraints-def pdb-cert-def* **by** *fastforce*
qed

lemma *models-absg*:
assumes *models* (*hc-constraints pdb-cert*) *rho* **and** $i < n\text{-}s$
shows *models* (*reification* (*Pos* (*abs-name i*))
(*map* (λ*l*. (*1*, *l*)) (*abs-lits i*)) (*length* (*abs-lits i*))) *rho*
using *models-pdb-gate*[*OF assms*(*1*)] *absg-in-gates*[*OF assms*(*2*)]
by (*simp add*: *absg-def*)

lemma *models-kgg*:
assumes *models* (*hc-constraints pdb-cert*) *rho* **and** $i < n\text{-}s$
shows *models* (*reification* (*Pos* (*kk-name i*)) (*k-gate-body B*)
(*clip B* (*int B* − *int* (*d* (*sa-i i*))))) *rho*
using *models-pdb-gate*[*OF assms*(*1*)] *kgg-in-gates*[*OF assms*(*2*)]
by (*simp add*: *kgg-def k-gate-def*)

lemma *models-thrg*:
assumes *models* (*hc-constraints pdb-cert*) *rho* **and** $i < n\text{-}s$
shows *models* (*reification* (*Pos* (*thr-name i*))
(*map* (λ*l*. (*1*, *l*)) (*thr-lits i*)) (*length* (*thr-lits i*))) *rho*
using *models-pdb-gate*[*OF assms*(*1*)] *thrg-in-gates*[*OF assms*(*2*)]
by (*simp add*: *thrg-def*)

lemma *models-poutg*:
assumes *models* (*hc-constraints pdb-cert*) *rho*
shows *models* (*reification* (*Pos pout-name*) (*map* (λ*l*. (*1*, *l*)) *pout-lits*) *1*) *rho*
using *models-pdb-gate*[*OF assms*] *poutg-in-gates*
by (*simp add*: *poutg-def*)

7.1.4 Semantics of the gates

lemma *abs-lits-sem*:
assumes *rho01*: ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
shows (∀ *l* ∈ *set* (*abs-lits i*). *eval-lit l rho* = *1*)
⟷ (∀ *v* ∈ *P*. *rho* (*StateVar v*) = (*if v* ∈ *sa-i i then 1 else 0*))
proof −
have *set-eq*: *set* (*sorted-list-of-set P*) = *P*
using *fin-P* **by** *simp*
have *lit-sem*: ⋀*v*. *eval-lit* (*if v* ∈ *sa-i i then Pos* (*StateVar v*)
else Neg (*StateVar v*)) *rho* = *1*
⟷ *rho* (*StateVar v*) = (*if v* ∈ *sa-i i then 1 else 0*)
proof −
fix *v*
show *eval-lit* (*if v* ∈ *sa-i i then Pos* (*StateVar v*) *else Neg* (*StateVar v*)) *rho* =
1
⟷ *rho* (*StateVar v*) = (*if v* ∈ *sa-i i then 1 else 0*)
using *rho01*[*rule-format*, *of StateVar v*] **by** (*auto simp*: *eval-lit-def*)
qed

```
  show ?thesis
  proof
    assume all1: ∀ l ∈ set (abs-lits i). eval-lit l rho = 1
    show ∀ v ∈ P. rho (StateVar v) = (if v ∈ sa-i i then 1 else 0)
    proof
      fix v assume vP: v ∈ P
      have (if v ∈ sa-i i then Pos (StateVar v) else Neg (StateVar v))
           ∈ set (abs-lits i)
        unfolding abs-lits-def using vP set-eq by auto
      then have eval-lit (if v ∈ sa-i i then Pos (StateVar v)
        else Neg (StateVar v)) rho = 1
        using all1 by blast
      then show rho (StateVar v) = (if v ∈ sa-i i then 1 else 0)
        using lit-sem[of v] by simp
    qed
  next
    assume enc: ∀ v ∈ P. rho (StateVar v) = (if v ∈ sa-i i then 1 else 0)
    show ∀ l ∈ set (abs-lits i). eval-lit l rho = 1
    proof
      fix l assume l ∈ set (abs-lits i)
      then obtain v where vP: v ∈ P
        and l-eq: l = (if v ∈ sa-i i then Pos (StateVar v) else Neg (StateVar v))
        unfolding abs-lits-def using set-eq by auto
      show eval-lit l rho = 1
        using enc vP lit-sem[of v] unfolding l-eq by simp
    qed
  qed
qed

lemma absg-forces:
  assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
    and m: models (hc-constraints pdb-cert) rho
    and i-lt: i < n-s
  shows rho (abs-name i) = 1
    ⟷ (∀ v ∈ P. rho (StateVar v) = (if v ∈ sa-i i then 1 else 0))
proof −
  have eval-lit (Pos (abs-name i)) rho = 1
    ⟷ (∀ l ∈ set (abs-lits i). eval-lit l rho = 1)
    by (rule conj-gate-forces[OF rho01 models-absg[OF m i-lt]])
  then show ?thesis
    unfolding abs-lits-sem[OF rho01] by (simp add: eval-lit-def)
qed

lemma kgg-forces:
  assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
    and m: models (hc-constraints pdb-cert) rho
    and i-lt: i < n-s
  shows rho (kk-name i) = 1
    ⟷ bits-val (bits-needed B) CostBit rho ≥ clip B (int B − int (d (sa-i i)))
```

```
proof −
  have eval-lit (Pos (kk-name i)) rho = 1
      ⟷ bits-val (bits-needed B) CostBit rho ≥ clip B (int B − int (d (sa-i i)))
    by (rule k-gate-forces[OF rho01 models-kgg[OF m i-lt]])
  then show ?thesis by (simp add: eval-lit-def)
qed

lemma thrg-forces:
  assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
    and m: models (hc-constraints pdb-cert) rho
    and i-lt: i < n-s
  shows rho (thr-name i) = 1 ⟷ rho (abs-name i) = 1 ∧ rho (kk-name i) = 1
proof −
  have eval-lit (Pos (thr-name i)) rho = 1
      ⟷ (∀ l ∈ set (thr-lits i). eval-lit l rho = 1)
    by (rule conj-gate-forces[OF rho01 models-thrg[OF m i-lt]])
  then show ?thesis by (simp add: thr-lits-def eval-lit-def)
qed

lemma poutg-forces:
  assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
    and m: models (hc-constraints pdb-cert) rho
  shows rho pout-name = 1 ⟷ (∃ i < n-s. rho (thr-name i) = 1)
proof −
  have eval-lit (Pos pout-name) rho = 1
      ⟷ (∃ l ∈ set pout-lits. eval-lit l rho = 1)
    by (rule disj-gate-forces[OF rho01 models-poutg[OF m]])
  then show ?thesis by (auto simp: pout-lits-def eval-lit-def)
qed
```

7.2 Lemma 8: the state lemma

```
lemma pdb-state-lemma: hc-state-lemma Πe B pdb-cert s
  unfolding hc-state-lemma-def
proof (intro allI impI)
  fix rho :: 'u pvar ⇒ nat
  assume rho01: ∀ x. rho x = 0 ∨ rho x = 1
    and m: models (hc-constraints pdb-cert) rho
    and sv: ∀ v ∈ vars Πe. rho (StateVar v) = (if v ∈ s then 1 else 0)
   and cb: bits-val (bits-needed B) CostBit rho ≥ clip B (int B − int (hc-h pdb-cert
s))
  obtain i where i-lt: i < n-s and i-s: sa-i i = s ∩ P
    using abs-mem-nth[of s ∩ P] by auto
  have abs1: rho (abs-name i) = 1
  proof −
    have ∀ v ∈ P. rho (StateVar v) = (if v ∈ sa-i i then 1 else 0)
    proof
      fix v assume vP: v ∈ P
      have rho (StateVar v) = (if v ∈ s then 1 else 0)
```

```
    using sv P-sub vP by blast
  then show rho (StateVar v) = (if v ∈ sa-i i then 1 else 0)
    using vP by (simp add: i-s)
 qed
 then show ?thesis using absg-forces[OF rho01 m i-lt] by blast
qed
have kk1: rho (kk-name i) = 1
proof −
 have hc-h pdb-cert s = d (sa-i i) by (simp add: pdb-cert-def i-s)
 then show ?thesis using kgg-forces[OF rho01 m i-lt] cb by simp
qed
have thr1: rho (thr-name i) = 1
 using thrg-forces[OF rho01 m i-lt] abs1 kk1 by blast
have rho pout-name = 1
 using poutg-forces[OF rho01 m] thr1 i-lt by blast
then show eval-lit (hc-out pdb-cert s) rho = 1
 by (simp add: pdb-cert-def eval-lit-def)
qed
```

7.3 Lemma 9: the goal lemma

```
lemma pdb-goal-lemma: hc-goal-lemma Πe B pdb-cert s
 unfolding hc-goal-lemma-def
proof (intro allI impI)
 fix rho :: 'u pvar ⇒ nat
 assume rho01: ∀ x. rho x = 0 ∨ rho x = 1
  and m: models (hc-constraints pdb-cert) rho
  and gv: ∀ v ∈ goal Πe. rho (StateVar v) = 1
  and out1: eval-lit (hc-out pdb-cert s) rho = 1
 have p1: rho pout-name = 1
  using out1 by (simp add: pdb-cert-def eval-lit-def)
 obtain i where i-lt: i < n-s and thr1: rho (thr-name i) = 1
  using poutg-forces[OF rho01 m] p1 by blast
 have abs1: rho (abs-name i) = 1 and kk1: rho (kk-name i) = 1
  using thrg-forces[OF rho01 m i-lt] thr1 by blast+
 have absv: ∀ v ∈ P. rho (StateVar v) = (if v ∈ sa-i i then 1 else 0)
  using absg-forces[OF rho01 m i-lt] abs1 by blast
 have goal-in: goal Πe ∩ P ⊆ sa-i i
 proof
  fix v assume vGP: v ∈ goal Πe ∩ P
  have rho (StateVar v) = 1 using gv vGP by blast
  moreover have rho (StateVar v) = (if v ∈ sa-i i then 1 else 0)
   using absv vGP by blast
  ultimately show v ∈ sa-i i by (cases v ∈ sa-i i) auto
 qed
 have d0: d (sa-i i) = 0
  using d-goal[OF sa-i-sub[OF i-lt] goal-in] .
 have clip B (int B − int (d (sa-i i))) = B
  using d0 by simp
```

then show *bits-val* (*bits-needed B*) *CostBit rho* ≥ *B*
using *kgg-forces*[*OF rho01 m i-lt*] *kk1* **by** *simp*
qed

7.4 Lemmas 10–13: the inductivity lemma

Paper Lemmas 10–13 build the inductivity lemma in four steps: the abstract transition for a single action (Lemma 10), the per-action invariant step (Lemma 11), the generalization over the transition relation (Lemma 12), and over all abstract states (Lemma 13). Semantically these collapse into one chase through the encoded transition; the proof below follows the same steps: the selected action, the abstract successor state (Lemma 10), the cost step along the K-gates (Lemma 11), quantified over the selected action (Lemma 12) and the true abstract-state gate (Lemma 13).

lemma *pdb-ind-lemma*: *hc-ind-lemma* Π*e B as pdb-cert s*
unfolding *hc-ind-lemma-def*
proof (*intro allI impI*)
fix *rho* :: *'u var pvar* ⇒ *nat*
assume *rho01*: ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
and *mO*: *models* (*circuit-constraints*
(*orig-circuit* (*hc-gates pdb-cert*, *hc-out pdb-cert s*))) *rho*
and *mP*: *models* (*circuit-constraints*
(*primed-circuit* (*hc-gates pdb-cert*, *hc-out pdb-cert s*))) *rho*
and *mT*: *models* (*encode-transition as* (*vars* Π*e*) *B*) *rho*
and *mB*: *models* (*encode-cost-ge B*) *rho*
and *rT*: *rho ReifT* = *1*
and *outO*: *eval-lit* (*map-literal* (*map-pvar Original*) (*hc-out pdb-cert s*)) *rho* = *1*
define *rho-o* **where** *rho-o* = *rho* ∘ *map-pvar Original*
define *rho-p* **where** *rho-p* = *rho* ∘ *primed-pvar-map*
have *rho-o01*: ∀ *x*. *rho-o x* = *0* ∨ *rho-o x* = *1*
unfolding *rho-o-def* **by** (*rule rho01-comp*[*OF rho01*])
have *rho-p01*: ∀ *x*. *rho-p x* = *0* ∨ *rho-p x* = *1*
unfolding *rho-p-def* **by** (*rule rho01-comp*[*OF rho01*])
have *mO''*: *models* (*circuit-constraints* (*hc-gates pdb-cert*, *hc-out pdb-cert s*)) *rho-o*
using *mO models-circuit-constraints-lift*[*of map-pvar Original*
(*hc-gates pdb-cert*, *hc-out pdb-cert s*) *rho*]
unfolding *rho-o-def orig-circuit-def* **by** *blast*
have *mO'*: *models* (*hc-constraints pdb-cert*) *rho-o*
using *mO'' hc-constraints-eq-circuit*[*of pdb-cert hc-out pdb-cert s*] **by** *simp*
have *mP''*: *models* (*circuit-constraints* (*hc-gates pdb-cert*, *hc-out pdb-cert s*)) *rho-p*
using *mP models-circuit-constraints-lift*[*of primed-pvar-map*
(*hc-gates pdb-cert*, *hc-out pdb-cert s*) *rho*]
unfolding *rho-p-def primed-circuit-def* **by** *blast*
have *mP'*: *models* (*hc-constraints pdb-cert*) *rho-p*
using *mP'' hc-constraints-eq-circuit*[*of pdb-cert hc-out pdb-cert s*] **by** *simp*

define *c* **where** *c* = *bits-val* (*bits-needed B*) *CostBit rho*
define *c′* **where** *c′* = *bits-val* (*bits-needed B*) *PrimedCostBit rho*
have *c-o*: *bits-val* (*bits-needed B*) *CostBit rho-o* = *c*
by (*simp add*: *rho-o-def c-def bits-val-def*)
have *c-p*: *bits-val* (*bits-needed B*) *CostBit rho-p* = *c′*
by (*simp add*: *rho-p-def c′-def bits-val-def*)
— The selected action (paper Lemma 12 quantifies over it).
have *mSel*: *models* (*action-selection-reif* (*action-reifs as*)) *rho*
by (*rule trans-sel*[*OF mT*])
obtain *rA* **where** *rA-in*: *rA* ∈ *set* (*action-reifs as*) **and** *rA1*: *eval-lit rA rho* = *1*
using *action-selection-forces*[*OF rho01 mSel*] *rT* **by** *blast*
obtain *j* **where** *j-lt*: *j* < *length as* **and** *rA-eq*: *rA* = *Pos* (*ReifAction j*)
using *rA-in* **unfolding** *action-reifs-def* **by** *auto*
define *a* **where** *a* = *as* ! *j*
have *a-in*: *a* ∈ *set as* **unfolding** *a-def* **by** (*rule nth-mem*[*OF j-lt*])
have *satAC*: *satisfies* (*action-constraint* (*Pos* (*ReifAction j*)) *a* (*vars* Π*e*) *B*) *rho*
unfolding *a-def* **by** (*rule trans-action-constraint*[*OF mT j-lt*])
have *preS*: *pre a* ⊆ *vars* Π*e* **and** *addS*: *add a* ⊆ *vars* Π*e* **and** *delS*: *del a* ⊆ *vars* Π*e*
using *bspec*[*OF as-states a-in*] **by** *auto*
have *outj*: *eval-lit* (*Pos* (*ReifAction j*)) *rho* = *1*
using *rA1 rA-eq* **by** *simp*
note *ext* = *action-constraint-extract*[*OF rho01 satAC outj fin-vars preS addS delS*]
have *delta1*: *rho* (*ReifDeltaCost* (*cost a*)) = *1* **using** *ext* **by** *blast*
have *preO*: ∀ *w* ∈ *pre a*. *rho* (*StateVar* (*Original w*)) = *1* **using** *ext* **by** *blast*
have *addP*: ∀ *w* ∈ *add a* − *del a*. *rho* (*StateVar* (*Primed w*)) = *1* **using** *ext* **by** *blast*
have *delP*: ∀ *w* ∈ *del a* − *add a*. *rho* (*StateVar* (*Primed w*)) = *0* **using** *ext* **by** *blast*
have *frameO*: ∀ *w* ∈ *vars* Π*e* − *evars a*. *rho* (*ReifEq* (*Original w*)) = *1* **using** *ext* **by** *blast*
have *mD*: *models* (*encode-delta-cost* (*cost a*) (*bits-needed B*)) *rho*
by (*rule trans-delta*[*OF mT a-in*])
have *c′-eq*: *c′* = *c* + *cost a*
using *encode-delta-cost-forces*[*OF rho01 mD*] *delta1*
by (*simp add*: *c-def c′-def*)
— The true abstract-state gate on the unprimed side (paper Lemma 13 quantifies over it).
have *outO′*: *rho-o pout-name* = *1*
using *outO* **unfolding** *pdb-cert-def*
by (*simp add*: *eval-lit-map-literal rho-o-def eval-lit-def*)
obtain *i* **where** *i-lt*: *i* < *n-s* **and** *thr1*: *rho-o* (*thr-name i*) = *1*
using *poutg-forces*[*OF rho-o01 mO′*] *outO′* **by** *blast*
have *abs1*: *rho-o* (*abs-name i*) = *1* **and** *kk1*: *rho-o* (*kk-name i*) = *1*
using *thrg-forces*[*OF rho-o01 mO′ i-lt*] *thr1* **by** *blast+*
define *sa* **where** *sa* = *sa-i i*
have *sa-sub*: *sa* ⊆ *P* **unfolding** *sa-def* **by** (*rule sa-i-sub*[*OF i-lt*])

```
  have absv: ∀ v ∈ P. rho-o (StateVar v) = (if v ∈ sa then 1 else 0)
    using absg-forces[OF rho-o01 mO' i-lt] abs1 unfolding sa-def by blast
  have c-ge: c ≥ clip B (int B − int (d sa))
    using kgg-forces[OF rho-o01 mO' i-lt] kk1 c-o unfolding sa-def by simp
  — The induced abstract action is applicable in the abstract state.
  have pre-sa: pre a ∩ P ⊆ sa
  proof
    fix v assume vp: v ∈ pre a ∩ P
    have rho (StateVar (Original v)) = 1 using preO vp by blast
    then have rho-o (StateVar v) = 1 by (simp add: rho-o-def)
    moreover have rho-o (StateVar v) = (if v ∈ sa then 1 else 0)
      using absv vp by blast
    ultimately show v ∈ sa by (cases v ∈ sa) auto
  qed
  define sa' where sa' = (sa − del a) ∪ (add a ∩ P)
  have sa'-sub: sa' ⊆ P using sa-sub unfolding sa'-def by auto
  obtain i' where i'-lt: i' < n-s and i'-s: sa-i i' = sa'
    using abs-mem-nth[OF sa'-sub] by auto
  — Paper Lemma 10: the primed state variables encode the abstract successor.
  have absv': ∀ v ∈ P. rho-p (StateVar v) = (if v ∈ sa' then 1 else 0)
  proof
    fix v assume vP: v ∈ P
    have rp: rho-p (StateVar v) = rho (StateVar (Primed v))
      by (simp add: rho-p-def)
    consider (AddC) v ∈ add a | (DelC) v ∈ del a v ∉ add a | (FrameC) v ∉ evars
a
      by (auto simp: evars-def)
    then show rho-p (StateVar v) = (if v ∈ sa' then 1 else 0)
    proof cases
      case AddC
      have v ∉ del a using AddC bspec[OF as-disjoint a-in] by auto
      then have rho (StateVar (Primed v)) = 1 using addP AddC by blast
      moreover have v ∈ sa' using AddC vP by (simp add: sa'-def)
      ultimately show ?thesis using rp by simp
    next
      case DelC
      have rho (StateVar (Primed v)) = 0 using delP DelC by blast
      moreover have v ∉ sa' using DelC by (auto simp: sa'-def)
      ultimately show ?thesis using rp by simp
    next
      case FrameC
      have w-in: v ∈ vars Πe − evars a using FrameC vP P-sub by auto
      then have eq1: rho (ReifEq (Original v)) = 1 using frameO by blast
      have meq: models (encode-eq-var v) rho
        by (rule trans-eq-var[OF mT]) (use w-in in blast)
      have eq2: rho (StateVar (Original v)) = rho (StateVar (Primed v))
        using encode-eq-var-forces[OF rho01 meq] eq1 by simp
      have eq3: rho (StateVar (Original v)) = (if v ∈ sa then 1 else 0)
        using bspec[OF absv vP] by (simp add: rho-o-def)
```

```
    have eq4: v ∈ sa' ⟷ v ∈ sa
      using FrameC by (auto simp: sa'-def evars-def)
    show ?thesis using rp eq2 eq3 eq4 by simp
   qed
  qed
  have abs1': rho-p (abs-name i') = 1
   using absg-forces[OF rho-p01 mP' i'-lt, unfolded i'-s] absv' by blast
  — Paper Lemma 11: the cost step along the K-gates.
  have c'-ge: c' ≥ clip B (int B − int (d sa'))
  proof −
   have tri: d sa ≤ d sa' + cost a
    using d-triangle[OF sa-sub a-in pre-sa] unfolding sa'-def .
   have int B − int (d sa') ≤ (int B − int (d sa)) + int (cost a)
    using tri by linarith
   then have clip B (int B − int (d sa')) ≤ clip B ((int B − int (d sa)) + int
(cost a))
    by (rule clip-mono)
   also have ... ≤ clip B (int B − int (d sa)) + cost a
    by (rule clip-add-le)
   also have ... ≤ c + cost a using c-ge by simp
   also have ... = c' using c'-eq by simp
   finally show ?thesis .
  qed
  have kk1': rho-p (kk-name i') = 1
   using kgg-forces[OF rho-p01 mP' i'-lt, unfolded i'-s] c'-ge c-p by simp
  have thr1': rho-p (thr-name i') = 1
   using thrg-forces[OF rho-p01 mP' i'-lt] abs1' kk1' by blast
  have rho-p pout-name = 1
   using poutg-forces[OF rho-p01 mP'] thr1' i'-lt by blast
  then show eval-lit (map-literal primed-pvar-map (hc-out pdb-cert s)) rho = 1
   unfolding pdb-cert-def
   by (simp add: eval-lit-map-literal rho-p-def eval-lit-def)
qed
```

The PDB certificate is a valid heuristic certificate in the sense of Definition 4, for any set of evaluated states.

```
theorem pdb-hc-valid: hc-valid Πe B as pdb-cert S
 unfolding hc-valid-def
 using pdb-state-lemma pdb-goal-lemma pdb-ind-lemma by blast
```

7.5 Structural conditions for use in the A* certificate

The remaining conditions of the *astar-run* locale on the heuristic certificate: every gate is named *ReifCert* (*nm p*) (instantiating *nm* with $\lambda j.\ Inr\ (2 * j + 1)$ at type $'v + nat$ yields exactly the odd gate names required there), the names are distinct, gate bodies only mention state variables, cost bits and earlier gate names, and the output literal is a gate name.

```
lemma pdb-gates-nth-abs: i < n-s ⟹ pdb-gates ! i = absg i
```

```
  by (simp add: pdb-gates-def nth-append)

lemma pdb-gates-nth-kgg: i < n-s ⟹ pdb-gates ! (n-s + i) = kgg i
  by (simp add: pdb-gates-def nth-append)

lemma pdb-gates-nth-thrg: i < n-s ⟹ pdb-gates ! (2 * n-s + i) = thrg i
  by (simp add: pdb-gates-def nth-append)

lemma pdb-gates-nth-out: pdb-gates ! (3 * n-s) = poutg
  by (simp add: pdb-gates-def nth-append)

lemma pdb-gate-name-nth:
  assumes p-lt: p < length pdb-gates
  shows fst (pdb-gates ! p) = Pos (ReifCert (nm p))
proof −
  consider (A) p < n-s | (K) n-s ≤ p p < 2 * n-s
    | (T) 2 * n-s ≤ p p < 3 * n-s | (OUT) p = 3 * n-s
    using p-lt length-pdb-gates by linarith
  then show ?thesis
  proof cases
    case A
    then show ?thesis
      using pdb-gates-nth-abs[OF A] by (simp add: absg-def abs-name-def)
  next
    case K
    then obtain q where p-eq: p = n-s + q using le-iff-add by auto
    have q-lt: q < n-s using K p-eq by simp
    show ?thesis
      unfolding p-eq using pdb-gates-nth-kgg[OF q-lt]
      by (simp add: kgg-def k-gate-def kk-name-def)
  next
    case T
    then obtain q where p-eq: p = 2 * n-s + q using le-iff-add by auto
    have q-lt: q < n-s using T p-eq by simp
    show ?thesis
      unfolding p-eq using pdb-gates-nth-thrg[OF q-lt]
      by (simp add: thrg-def thr-name-def)
  next
    case OUT
    then show ?thesis
      using pdb-gates-nth-out by (simp add: poutg-def pout-name-def)
  qed
qed

lemma pdb-names:
  ∀(r, cs, A) ∈ set (hc-gates pdb-cert). ∃j. r = Pos (ReifCert (nm j))
proof −
  have ⋀g. g ∈ set pdb-gates ⟹ ∃j. fst g = Pos (ReifCert (nm j))
  proof −
```

```
    fix g assume g ∈ set pdb-gates
    then obtain p where p-lt: p < length pdb-gates and g-eq: g = pdb-gates ! p
      by (auto simp: in-set-conv-nth)
    show ∃ j. fst g = Pos (ReifCert (nm j))
      using pdb-gate-name-nth[OF p-lt] g-eq by auto
  qed
  then show ?thesis by (fastforce simp: pdb-cert-def)
qed

lemma pdb-distinct:
  distinct (map (λ(r, cs, A). pvar-of-lit r) (hc-gates pdb-cert))
proof −
  note len = length-pdb-gates
  have map-eq: map (λ(r, cs, A). pvar-of-lit r) pdb-gates
      = map (λp. ReifCert (nm p)) [0..<3 * n-s + 1]
  proof (rule nth-equalityI)
    show length (map (λ(r, cs, A). pvar-of-lit r) pdb-gates)
        = length (map (λp. ReifCert (nm p)) [0..<3 * n-s + 1])
      by (simp add: len)
    fix p assume p < length (map (λ(r, cs, A). pvar-of-lit r) pdb-gates)
    then have p-lt: p < length pdb-gates by simp
     have (λ(r, cs, A). pvar-of-lit r) (pdb-gates ! p) = pvar-of-lit (fst (pdb-gates !
p))
      by (simp add: split-beta)
    also have ... = ReifCert (nm p)
      using pdb-gate-name-nth[OF p-lt] by (simp add: pvar-of-lit-def)
    finally show map (λ(r, cs, A). pvar-of-lit r) pdb-gates ! p
        = map (λp. ReifCert (nm p)) [0..<3 * n-s + 1] ! p
      using p-lt len by (auto simp: nth-append less-Suc-eq)
  qed
  have distinct (map (λp. ReifCert (nm p)) [0..<3 * n-s + 1])
    using nm-inj by (auto simp: distinct-map intro!: inj-onI dest: injD)
  then show ?thesis using map-eq by (simp add: pdb-cert-def)
qed

lemma pdb-out-in: hc-out pdb-cert s ∈ fst ‘ set (hc-gates pdb-cert)
proof −
  have fst poutg = Pos pout-name by (simp add: poutg-def)
  then show ?thesis using poutg-in-gates by (force simp: pdb-cert-def)
qed

lemma nth-in-take-pdb: j < i ⟹ i ≤ length xs ⟹ xs ! j ∈ set (take i xs)
  by (metis length-take min.absorb2 nth-mem nth-take)

lemma pdb-wf:
  ∀ i < length (hc-gates pdb-cert). case hc-gates pdb-cert ! i of (r, cs, A) ⇒
      (∀ x ∈ pvar-of-lit ‘ snd ‘ set cs.
        (∃ v. x = StateVar v) ∨ (∃ j. x = CostBit j)
        ∨ x ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates pdb-cert)))
```

```
proof (intro allI impI)
  fix i assume i-lt: i < length (hc-gates pdb-cert)
  have hcg: hc-gates pdb-cert = pdb-gates by (simp add: pdb-cert-def)
  have i-lt': i < length pdb-gates using i-lt by (simp add: hcg)
  have i-le: i ≤ length pdb-gates using i-lt' by simp
  consider (A) i < n-s | (K) n-s ≤ i i < 2 * n-s
    | (T) 2 * n-s ≤ i i < 3 * n-s | (OUT) i = 3 * n-s
    using i-lt' length-pdb-gates by linarith
  then show case hc-gates pdb-cert ! i of (r, cs, A) ⇒
      (∀ x ∈ pvar-of-lit ‘ snd ‘ set cs.
        (∃ v. x = StateVar v) ∨ (∃ j. x = CostBit j)
        ∨ x ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates pdb-cert)))
  proof cases
    case A
    have g: hc-gates pdb-cert ! i = absg i
      using pdb-gates-nth-abs[OF A] hcg by simp
    have ∀ x ∈ pvar-of-lit ‘ snd ‘ set (map (λl. (1, l)) (abs-lits i)).
        (∃ v. x = StateVar v)
      by (auto simp: abs-lits-def pvar-of-lit-def split: if-splits)
    then show ?thesis unfolding g absg-def by auto
  next
    case K
    then obtain q where i-eq: i = n-s + q using le-iff-add by auto
    have q-lt: q < n-s using K i-eq by simp
    have g: hc-gates pdb-cert ! i = kgg q
      using pdb-gates-nth-kgg[OF q-lt] hcg i-eq by simp
    have ∀ x ∈ pvar-of-lit ‘ snd ‘ set (k-gate-body B). (∃ j. x = CostBit j)
      by (auto simp: k-gate-body-def pvar-of-lit-def)
    then show ?thesis unfolding g kgg-def k-gate-def by auto
  next
    case T
    then obtain q where i-eq: i = 2 * n-s + q using le-iff-add by auto
    have q-lt: q < n-s using T i-eq by simp
    have g: hc-gates pdb-cert ! i = thrg q
      using pdb-gates-nth-thrg[OF q-lt] hcg i-eq by simp
    have abs-in: abs-name q ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates pdb-cert))
    proof −
      have q < i using q-lt i-eq by simp
      then have pdb-gates ! q ∈ set (take i pdb-gates)
        using nth-in-take-pdb i-le by blast
      moreover have fst (pdb-gates ! q) = Pos (abs-name q)
        using pdb-gates-nth-abs[OF q-lt] by (simp add: absg-def)
      ultimately show ?thesis using hcg by (force simp: pvar-of-lit-def)
    qed
    have kk-in: kk-name q ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates pdb-cert))
    proof −
      have n-s + q < i using q-lt i-eq by simp
      then have pdb-gates ! (n-s + q) ∈ set (take i pdb-gates)
        using nth-in-take-pdb i-le by blast
```

```
    moreover have fst (pdb-gates ! (n-s + q)) = Pos (kk-name q)
      using pdb-gates-nth-kgg[OF q-lt] by (simp add: kgg-def k-gate-def)
    ultimately show ?thesis using hcg by (force simp: pvar-of-lit-def)
  qed
  have ∀ x ∈ pvar-of-lit ‘ snd ‘ set (map (λl. (1, l)) (thr-lits q)).
      x ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates pdb-cert))
    using abs-in kk-in by (auto simp: thr-lits-def pvar-of-lit-def)
  then show ?thesis unfolding g thrg-def by auto
next
  case OUT
  have g: hc-gates pdb-cert ! i = poutg
    using pdb-gates-nth-out hcg OUT by simp
  have ∀ x ∈ pvar-of-lit ‘ snd ‘ set (map (λl. (1, l)) pout-lits).
      x ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates pdb-cert))
  proof
    fix x assume x ∈ pvar-of-lit ‘ snd ‘ set (map (λl. (1, l)) pout-lits)
    then obtain q where q-lt: q < n-s and x-eq: x = thr-name q
      by (auto simp: pout-lits-def pvar-of-lit-def)
    have 2 * n-s + q < i using q-lt OUT by simp
    then have pdb-gates ! (2 * n-s + q) ∈ set (take i pdb-gates)
      using nth-in-take-pdb i-le by blast
    moreover have fst (pdb-gates ! (2 * n-s + q)) = Pos (thr-name q)
      using pdb-gates-nth-thrg[OF q-lt] by (simp add: thrg-def)
    ultimately show x ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates pdb-cert))
      using hcg x-eq by (force simp: pvar-of-lit-def)
  qed
  then show ?thesis unfolding g poutg-def by auto
 qed
qed

end

end
```

8 Maximum Heuristic Certificates

```
theory Hmax-Certificates
 imports A-Star-Certificates
begin
```

Proof-logging the maximum heuristic (paper equations (17)–(18) and Lemmas 14–17). The certificate is built per evaluated state s from the values an hmax implementation computes anyway: the heuristic estimate $h = hmax(s)$ and the clipped max values $W(v) = min\{ Vmax(v),\ hmax(s)\}$. The locale *hmax-heuristic* captures exactly the properties of this table that the soundness of the certificate requires (all consequences of the hmax fixed-point equations): W vanishes on s, some goal variable carries the full value h (or the goal is empty and $h = 0$), and W satisfies the action-step recurrence

$W(v) \leq W(p) + cost(a)$ for add effects v and some precondition p (or $W(v) \leq cost(a)$ for actions without preconditions). The K-gates referenced by equations (17) and (18) are inlined over the cost bits, as everywhere else in this formalization. The resulting *heuristic-cert* is valid in the sense of Definition 4 for the singleton set of evaluated states $\{s\}$ (the paper circuit $\langle Hmax, s, rmax\text{-}s \rangle$ is likewise per-state).

8.1 The hmax locale

locale *hmax-heuristic* =
  fixes Πe :: *'u*::*linorder strips-task*
    and B :: *nat*
    and s :: *'u state*
    and h :: *nat*
    and W :: *'u* $\Rightarrow$ *nat*
    and *vs* :: *'u list*
    and *as* :: *'u action list*
    and *nm* :: *nat* $\Rightarrow$ *'u*
  assumes *fin-vars*: *finite* (*vars* Πe)
    and *goal-sub*: *goal* $\Pi e \subseteq$ *vars* Πe
    and *s-sub*: $s \subseteq$ *vars* Πe
    and *vs-set*: *set vs* = *vars* Πe
    and *vs-dist*: *distinct vs*
    and *as-states*: $\forall a \in$ *set as*. *pre a* $\subseteq$ *vars* $\Pi e \wedge$ *add a* $\subseteq$ *vars* $\Pi e \wedge$ *del a* $\subseteq$ *vars* Πe
    and *as-disjoint*: $\forall a \in$ *set as*. *add a* $\cap$ *del a* = $\{\}$
    and *W-zero*: $\bigwedge v.\ v \in s \Longrightarrow W\ v = 0$
    and *goal-W*: *goal* $\Pi e \neq \{\} \Longrightarrow \exists v \in$ *goal* $\Pi e.\ W\ v = h$
    and *goal-empty*: *goal* $\Pi e = \{\} \Longrightarrow h = 0$
    and *W-step-pre*: $\bigwedge a\ v.\ a \in$ *set as* $\Longrightarrow v \in$ *add a* $\Longrightarrow$ *pre a* $\neq \{\} \Longrightarrow$
      $\exists p \in$ *pre a*. $W\ v \leq W\ p +$ *cost a*
    and *W-step-empty*: $\bigwedge a\ v.\ a \in$ *set as* $\Longrightarrow v \in$ *add a* $\Longrightarrow$ *pre a* $= \{\} \Longrightarrow$
      $W\ v \leq$ *cost a*
    and *nm-inj*: *inj nm*
begin

definition *n-v* :: *nat* **where**
  n-v = *length vs*

definition *v-i* :: *nat* $\Rightarrow$ *'u* **where**
  v-i i = *vs* ! *i*

lemma *v-i-in*: *i* < *n-v* $\Longrightarrow$ *v-i i* $\in$ *vars* Πe
  using *vs-set nth-mem* **unfolding** *n-v-def v-i-def* **by** *fastforce*

lemma *var-mem-nth*:
  assumes $v \in$ *vars* Πe
  shows $\exists i.\ i <$ *n-v* $\wedge$ *v-i i* = v

proof −
have *v* ∈ *set vs* **using** *assms vs-set* **by** *auto*
then show *?thesis*
unfolding *n-v-def v-i-def* **by** (*auto simp*: *in-set-conv-nth*)
qed

8.1.1 Gate names

definition *kv-name* :: *nat* ⇒ *'u pvar* **where**
kv-name i = *ReifCert* (*nm i*)

definition *rv-name* :: *nat* ⇒ *'u pvar* **where**
rv-name i = *ReifCert* (*nm* (*n-v* + *i*))

definition *kb-name* :: *'u pvar* **where**
kb-name = *ReifCert* (*nm* (*2* ∗ *n-v*))

definition *max-name* :: *'u pvar* **where**
max-name = *ReifCert* (*nm* (*2* ∗ *n-v* + *1*))

8.1.2 Gates

The K-gate part of equation (17): the clipped cost threshold for $B - hmax(s) + Wmax(v)$, inlined over the cost bits.

definition *kvg* :: *nat* ⇒ *'u pvar literal* × (*nat* × *'u pvar literal*) *list* × *nat* **where**
kvg i = *k-gate* (*Pos* (*kv-name i*)) *B* (*int B* − *int h* + *int* (*W* (*v-i i*)))

Equation (17): the variable gate is the disjunction of the negated state variable and its K-gate.

definition *rv-lits* :: *nat* ⇒ *'u pvar literal list* **where**
rv-lits i = [*Neg* (*StateVar* (*v-i i*)), *Pos* (*kv-name i*)]

definition *rvg* :: *nat* ⇒ *'u pvar literal* × (*nat* × *'u pvar literal*) *list* × *nat* **where**
rvg i = (*Pos* (*rv-name i*), *map* (λ*l*. (*1*, *l*)) (*rv-lits i*), *1*)

The base K-gate for $B - hmax(s)$ used by equation (18).

definition *kbg* :: *'u pvar literal* × (*nat* × *'u pvar literal*) *list* × *nat* **where**
kbg = *k-gate* (*Pos kb-name*) *B* (*int B* − *int h*)

Equation (18): the output is the conjunction of the base K-gate and all variable gates (the paper's threshold $|V| + 1$ over 0/1 summands).

definition *max-lits* :: *'u pvar literal list* **where**
max-lits = *Pos kb-name* # *map* (λ*i*. *Pos* (*rv-name i*)) [*0*..<*n-v*]

definition *mxg* :: *'u pvar literal* × (*nat* × *'u pvar literal*) *list* × *nat* **where**
mxg = (*Pos max-name*, *map* (λ*l*. (*1*, *l*)) *max-lits*, *length max-lits*)

definition *hmax-gates* ::
(*'u pvar literal* × (*nat* × *'u pvar literal*) *list* × *nat*) *list* **where**

hmax-gates = *map kvg* [*0*..<*n-v*] @ *map rvg* [*0*..<*n-v*] @ [*kbg*, *mxg*]

definition *hmax-cert* :: ('*u*) *heuristic-cert* **where**
hmax-cert = (| *hc-gates* = *hmax-gates*,
hc-out = (λ*s'*. *Pos max-name*),
hc-h = (λ*s'*. *h*) |)

8.1.3 Basic structure of the gate list

lemma *length-hmax-gates*: *length hmax-gates* = *2* ∗ *n-v* + *2*
by (*simp add*: *hmax-gates-def*)

lemma *kvg-in-gates*: *i* < *n-v* ⟹ *kvg i* ∈ *set hmax-gates*
by (*simp add*: *hmax-gates-def*)

lemma *rvg-in-gates*: *i* < *n-v* ⟹ *rvg i* ∈ *set hmax-gates*
by (*simp add*: *hmax-gates-def*)

lemma *kbg-in-gates*: *kbg* ∈ *set hmax-gates*
by (*simp add*: *hmax-gates-def*)

lemma *mxg-in-gates*: *mxg* ∈ *set hmax-gates*
by (*simp add*: *hmax-gates-def*)

lemma *models-hmax-gate*:
assumes *m*: *models* (*hc-constraints hmax-cert*) *rho*
and *g-in*: (*r*, *cs*, *A*) ∈ *set hmax-gates*
shows *models* (*reification r cs A*) *rho*
proof (*rule models-mono*[*OF m*])
show *reification r cs A* ⊆ *hc-constraints hmax-cert*
using *g-in* **unfolding** *hc-constraints-def hmax-cert-def* **by** *fastforce*
qed

lemma *models-kvg*:
assumes *models* (*hc-constraints hmax-cert*) *rho* **and** *i* < *n-v*
shows *models* (*reification* (*Pos* (*kv-name i*)) (*k-gate-body B*)
(*clip B* (*int B* − *int h* + *int* (*W* (*v-i i*))))) *rho*
using *models-hmax-gate*[*OF assms*(*1*)] *kvg-in-gates*[*OF assms*(*2*)]
by (*simp add*: *kvg-def k-gate-def*)

lemma *models-rvg*:
assumes *models* (*hc-constraints hmax-cert*) *rho* **and** *i* < *n-v*
shows *models* (*reification* (*Pos* (*rv-name i*)) (*map* (λ*l*. (*1*, *l*)) (*rv-lits i*)) *1*) *rho*
using *models-hmax-gate*[*OF assms*(*1*)] *rvg-in-gates*[*OF assms*(*2*)]
by (*simp add*: *rvg-def*)

lemma *models-kbg*:
assumes *models* (*hc-constraints hmax-cert*) *rho*
shows *models* (*reification* (*Pos kb-name*) (*k-gate-body B*) (*clip B* (*int B* − *int*

h))) *rho*
using *models-hmax-gate*[*OF assms*] *kbg-in-gates*
by (*simp add*: *kbg-def k-gate-def*)

lemma *models-mxg*:
assumes *models* (*hc-constraints hmax-cert*) *rho*
shows *models* (*reification* (*Pos max-name*)
(*map* (λ*l*. (*1*, *l*)) *max-lits*) (*length max-lits*)) *rho*
using *models-hmax-gate*[*OF assms*] *mxg-in-gates*
by (*simp add*: *mxg-def*)

8.1.4 Semantics of the gates

lemma *kvg-forces*:
assumes *rho01*: ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
and *m*: *models* (*hc-constraints hmax-cert*) *rho*
and *i-lt*: *i* < *n-v*
shows *rho* (*kv-name i*) = *1*
⟷ *bits-val* (*bits-needed B*) *CostBit rho* ≥ *clip B* (*int B* − *int h* + *int* (*W* (*v-i i*)))
proof −
have *eval-lit* (*Pos* (*kv-name i*)) *rho* = *1*
⟷ *bits-val* (*bits-needed B*) *CostBit rho* ≥ *clip B* (*int B* − *int h* + *int* (*W* (*v-i i*)))
by (*rule k-gate-forces*[*OF rho01 models-kvg*[*OF m i-lt*]])
then show *?thesis* **by** (*simp add*: *eval-lit-def*)
qed

lemma *rvg-forces*:
assumes *rho01*: ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
and *m*: *models* (*hc-constraints hmax-cert*) *rho*
and *i-lt*: *i* < *n-v*
shows *rho* (*rv-name i*) = *1*
⟷ *rho* (*StateVar* (*v-i i*)) = *0* ∨ *rho* (*kv-name i*) = *1*
proof −
have *eval-lit* (*Pos* (*rv-name i*)) *rho* = *1*
⟷ (∃ *l* ∈ *set* (*rv-lits i*). *eval-lit l rho* = *1*)
by (*rule disj-gate-forces*[*OF rho01 models-rvg*[*OF m i-lt*]])
then show *?thesis* **by** (*auto simp*: *rv-lits-def eval-lit-def*)
qed

lemma *kbg-forces*:
assumes *rho01*: ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
and *m*: *models* (*hc-constraints hmax-cert*) *rho*
shows *rho kb-name* = *1*
⟷ *bits-val* (*bits-needed B*) *CostBit rho* ≥ *clip B* (*int B* − *int h*)
proof −
have *eval-lit* (*Pos kb-name*) *rho* = *1*
⟷ *bits-val* (*bits-needed B*) *CostBit rho* ≥ *clip B* (*int B* − *int h*)

```
    by (rule k-gate-forces[OF rho01 models-kbg[OF m]])
  then show ?thesis by (simp add: eval-lit-def)
qed

lemma mxg-forces:
  assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
    and m: models (hc-constraints hmax-cert) rho
  shows rho max-name = 1
     ⟷ rho kb-name = 1 ∧ (∀ i < n-v. rho (rv-name i) = 1)
proof −
  have eval-lit (Pos max-name) rho = 1
      ⟷ (∀ l ∈ set max-lits. eval-lit l rho = 1)
    by (rule conj-gate-forces[OF rho01 models-mxg[OF m]])
  then show ?thesis by (auto simp: max-lits-def eval-lit-def)
qed
```

8.2 Lemma 14: the state lemma

```
lemma hmax-state-lemma: hc-state-lemma Πe B hmax-cert s
  unfolding hc-state-lemma-def
proof (intro allI impI)
  fix rho :: 'u pvar ⇒ nat
  assume rho01: ∀ x. rho x = 0 ∨ rho x = 1
    and m: models (hc-constraints hmax-cert) rho
    and sv: ∀ v ∈ vars Πe. rho (StateVar v) = (if v ∈ s then 1 else 0)
     and cb: bits-val (bits-needed B) CostBit rho ≥ clip B (int B − int (hc-h
hmax-cert s))
  have cb': bits-val (bits-needed B) CostBit rho ≥ clip B (int B − int h)
    using cb by (simp add: hmax-cert-def)
  have kb1: rho kb-name = 1
    using kbg-forces[OF rho01 m] cb' by simp
  have rv-all: ∀ i < n-v. rho (rv-name i) = 1
  proof (intro allI impI)
    fix i assume i-lt: i < n-v
    show rho (rv-name i) = 1
    proof (cases v-i i ∈ s)
      case True
      have clip B (int B − int h + int (W (v-i i))) = clip B (int B − int h)
        using W-zero[OF True] by simp
      then have rho (kv-name i) = 1
        using kvg-forces[OF rho01 m i-lt] cb' by simp
      then show ?thesis using rvg-forces[OF rho01 m i-lt] by blast
    next
      case False
      have rho (StateVar (v-i i)) = 0
        using sv v-i-in[OF i-lt] False by simp
      then show ?thesis using rvg-forces[OF rho01 m i-lt] by blast
    qed
  qed
```

```
  have rho max-name = 1
    using mxg-forces[OF rho01 m] kb1 rv-all by blast
  then show eval-lit (hc-out hmax-cert s) rho = 1
    by (simp add: hmax-cert-def eval-lit-def)
qed
```

8.3 Lemma 15: the goal lemma

```
lemma hmax-goal-lemma: hc-goal-lemma Πe B hmax-cert s'
  unfolding hc-goal-lemma-def
proof (intro allI impI)
  fix rho :: 'u pvar ⇒ nat
  assume rho01: ∀ x. rho x = 0 ∨ rho x = 1
    and m: models (hc-constraints hmax-cert) rho
    and gv: ∀ v ∈ goal Πe. rho (StateVar v) = 1
    and out1: eval-lit (hc-out hmax-cert s') rho = 1
  have mx1: rho max-name = 1
    using out1 by (simp add: hmax-cert-def eval-lit-def)
  have kb1: rho kb-name = 1 and rv-all: ∀ i < n-v. rho (rv-name i) = 1
    using mxg-forces[OF rho01 m] mx1 by blast+
  show bits-val (bits-needed B) CostBit rho ≥ B
  proof (cases goal Πe = {})
    case True
    have h = 0 by (rule goal-empty[OF True])
    then have clip B (int B − int h) = B by simp
    then show ?thesis using kbg-forces[OF rho01 m] kb1 by simp
  next
    case False
    obtain v where vG: v ∈ goal Πe and vW: W v = h
      using goal-W[OF False] by blast
    obtain i where i-lt: i < n-v and i-v: v-i i = v
      using var-mem-nth goal-sub vG by blast
    have rho (StateVar v) = 1 using gv vG by blast
    then have rho (kv-name i) = 1
      using rvg-forces[OF rho01 m i-lt] rv-all i-lt i-v by auto
    moreover have clip B (int B − int h + int (W (v-i i))) = B
      by (simp add: i-v vW)
    ultimately show ?thesis using kvg-forces[OF rho01 m i-lt] by simp
  qed
qed
```

8.4 Lemmas 16–17: the inductivity lemma

Paper Lemma 16 establishes the step for a single action by a case analysis over how each variable occurs in the action's effects; Lemma 17 generalizes over the selected action. Semantically both collapse into one chase: the selected action of the encoded transition is fixed, and each variable gate of the primed copy is established by the add / delete / frame case analysis, using the W-recurrence for add effects.

lemma *hmax-ind-lemma*: *hc-ind-lemma* Πe *B as hmax-cert s′*
unfolding *hc-ind-lemma-def*
proof (*intro allI impI*)
fix *rho* :: *′u var pvar* ⇒ *nat*
assume *rho01*: ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
and *mO*: *models* (*circuit-constraints*
(*orig-circuit* (*hc-gates hmax-cert*, *hc-out hmax-cert s′*))) *rho*
and *mP*: *models* (*circuit-constraints*
(*primed-circuit* (*hc-gates hmax-cert*, *hc-out hmax-cert s′*))) *rho*
and *mT*: *models* (*encode-transition as* (*vars* Πe) *B*) *rho*
and *mB*: *models* (*encode-cost-ge B*) *rho*
and *rT*: *rho ReifT* = *1*
and *outO*: *eval-lit* (*map-literal* (*map-pvar Original*) (*hc-out hmax-cert s′*)) *rho*
= *1*
define *rho-o* **where** *rho-o* = *rho* ∘ *map-pvar Original*
define *rho-p* **where** *rho-p* = *rho* ∘ *primed-pvar-map*
have *rho-o01*: ∀ *x*. *rho-o x* = *0* ∨ *rho-o x* = *1*
unfolding *rho-o-def* **by** (*rule rho01-comp*[*OF rho01*])
have *rho-p01*: ∀ *x*. *rho-p x* = *0* ∨ *rho-p x* = *1*
unfolding *rho-p-def* **by** (*rule rho01-comp*[*OF rho01*])
have *mO′′*: *models* (*circuit-constraints* (*hc-gates hmax-cert*, *hc-out hmax-cert s′*))
rho-o
using *mO models-circuit-constraints-lift*[*of map-pvar Original*
(*hc-gates hmax-cert*, *hc-out hmax-cert s′*) *rho*]
unfolding *rho-o-def orig-circuit-def* **by** *blast*
have *mO′*: *models* (*hc-constraints hmax-cert*) *rho-o*
using *mO′′ hc-constraints-eq-circuit*[*of hmax-cert hc-out hmax-cert s′*] **by** *simp*
have *mP′′*: *models* (*circuit-constraints* (*hc-gates hmax-cert*, *hc-out hmax-cert s′*))
rho-p
using *mP models-circuit-constraints-lift*[*of primed-pvar-map*
(*hc-gates hmax-cert*, *hc-out hmax-cert s′*) *rho*]
unfolding *rho-p-def primed-circuit-def* **by** *blast*
have *mP′*: *models* (*hc-constraints hmax-cert*) *rho-p*
using *mP′′ hc-constraints-eq-circuit*[*of hmax-cert hc-out hmax-cert s′*] **by** *simp*
define *c* **where** *c* = *bits-val* (*bits-needed B*) *CostBit rho*
define *c′* **where** *c′* = *bits-val* (*bits-needed B*) *PrimedCostBit rho*
have *c-o*: *bits-val* (*bits-needed B*) *CostBit rho-o* = *c*
by (*simp add*: *rho-o-def c-def bits-val-def*)
have *c-p*: *bits-val* (*bits-needed B*) *CostBit rho-p* = *c′*
by (*simp add*: *rho-p-def c′-def bits-val-def*)
— The selected action (paper Lemma 17 quantifies over it).
have *mSel*: *models* (*action-selection-reif* (*action-reifs as*)) *rho*
by (*rule trans-sel*[*OF mT*])
obtain *rA* **where** *rA-in*: *rA* ∈ *set* (*action-reifs as*) **and** *rA1*: *eval-lit rA rho* =
1
using *action-selection-forces*[*OF rho01 mSel*] *rT* **by** *blast*
obtain *j* **where** *j-lt*: *j* < *length as* **and** *rA-eq*: *rA* = *Pos* (*ReifAction j*)
using *rA-in* **unfolding** *action-reifs-def* **by** *auto*
define *a* **where** *a* = *as* ! *j*

have *a-in*: $a \in set\ as$ **unfolding** *a-def* **by** (*rule nth-mem*[*OF j-lt*])
have *satAC*: *satisfies* (*action-constraint* (*Pos* (*ReifAction j*)) *a* (*vars* Π*e*) *B*) *rho*
unfolding *a-def* **by** (*rule trans-action-constraint*[*OF mT j-lt*])
have *preS*: *pre a* ⊆ *vars* Π*e* **and** *addS*: *add a* ⊆ *vars* Π*e* **and** *delS*: *del a* ⊆ *vars* Π*e*
using *bspec*[*OF as-states a-in*] **by** *auto*
have *outj*: *eval-lit* (*Pos* (*ReifAction j*)) *rho* = *1*
using *rA1 rA-eq* **by** *simp*
note *ext* = *action-constraint-extract*[*OF rho01 satAC outj fin-vars preS addS delS*]
have *delta1*: *rho* (*ReifDeltaCost* (*cost a*)) = *1* **using** *ext* **by** *blast*
have *preO*: ∀ *w* ∈ *pre a*. *rho* (*StateVar* (*Original w*)) = *1* **using** *ext* **by** *blast*
have *addP*: ∀ *w* ∈ *add a* − *del a*. *rho* (*StateVar* (*Primed w*)) = *1* **using** *ext* **by** *blast*
have *delP*: ∀ *w* ∈ *del a* − *add a*. *rho* (*StateVar* (*Primed w*)) = *0* **using** *ext* **by** *blast*
have *frameO*: ∀ *w* ∈ *vars* Π*e* − *evars a*. *rho* (*ReifEq* (*Original w*)) = *1* **using** *ext* **by** *blast*
have *mD*: *models* (*encode-delta-cost* (*cost a*) (*bits-needed B*)) *rho*
by (*rule trans-delta*[*OF mT a-in*])
have *c′-eq*: $c' = c + cost\ a$
using *encode-delta-cost-forces*[*OF rho01 mD*] *delta1*
by (*simp add*: *c-def c′-def*)
— The true output gate on the unprimed side.
have *outO′*: *rho-o max-name* = *1*
using *outO* **unfolding** *hmax-cert-def*
by (*simp add*: *eval-lit-map-literal rho-o-def eval-lit-def*)
have *kb-o*: *rho-o kb-name* = *1* **and** *rv-o*: ∀ *i* < *n-v*. *rho-o* (*rv-name i*) = *1*
using *mxg-forces*[*OF rho-o01 mO′*] *outO′* **by** *blast+*
have *c-base*: $c \geq clip\ B\ (int\ B - int\ h)$
using *kbg-forces*[*OF rho-o01 mO′*] *kb-o c-o* **by** *simp*
— A true unprimed state variable carries its cost threshold.
have *carry*: $\bigwedge v.\ v \in vars\ \Pi e \Longrightarrow rho\text{-}o\ (StateVar\ v) = 1 \Longrightarrow$
$c \geq clip\ B\ (int\ B - int\ h + int\ (W\ v))$
proof −
fix *v* **assume** *vV*: *v* ∈ *vars* Π*e* **and** *v1*: *rho-o* (*StateVar v*) = *1*
obtain *i* **where** *i-lt*: *i* < *n-v* **and** *i-v*: *v-i i* = *v*
using *var-mem-nth*[*OF vV*] **by** *blast*
have *rho-o* (*kv-name i*) = *1*
using *rvg-forces*[*OF rho-o01 mO′ i-lt*] *rv-o i-lt i-v v1* **by** *auto*
then show $c \geq clip\ B\ (int\ B - int\ h + int\ (W\ v))$
using *kvg-forces*[*OF rho-o01 mO′ i-lt*] *c-o i-v* **by** *simp*
qed
— Base K-gate of the primed copy.
have *kb-p*: *rho-p kb-name* = *1*
proof −
have $c' \geq clip\ B\ (int\ B - int\ h)$ **using** *c-base c′-eq* **by** *simp*
then show *?thesis* **using** *kbg-forces*[*OF rho-p01 mP′*] *c-p* **by** *simp*
qed

— Paper Lemma 16: the variable gates of the primed copy, by the add / delete / frame case analysis.

```
have rv-p: ∀ i < n-v. rho-p (rv-name i) = 1
proof (intro allI impI)
  fix i assume i-lt: i < n-v
  define v where v = v-i i
  have vV: v ∈ vars Πe unfolding v-def by (rule v-i-in[OF i-lt])
  show rho-p (rv-name i) = 1
  proof (cases rho-p (StateVar v) = 0)
    case True
    then show ?thesis
      using rvg-forces[OF rho-p01 mP' i-lt] unfolding v-def by blast
  next
    case False
    then have v1p: rho-p (StateVar v) = 1
      using rho-p01[rule-format, of StateVar v] by simp
    have c'-thr: c' ≥ clip B (int B − int h + int (W v))
    proof −
      consider (AddC) v ∈ add a | (DelC) v ∈ del a v ∉ add a
        | (FrameC) v ∉ evars a
        by (auto simp: evars-def)
      then show ?thesis
      proof cases
        case AddC
        show ?thesis
        proof (cases pre a = {})
          case True
          have Wv: W v ≤ cost a by (rule W-step-empty[OF a-in AddC True])
          have int B − int h + int (W v) ≤ (int B − int h) + int (cost a)
            using Wv by linarith
          then have clip B (int B − int h + int (W v))
              ≤ clip B ((int B − int h) + int (cost a))
            by (rule clip-mono)
          also have ... ≤ clip B (int B − int h) + cost a
            by (rule clip-add-le)
          also have ... ≤ c + cost a using c-base by simp
          also have ... = c' using c'-eq by simp
          finally show ?thesis .
        next
          case False
          obtain p where pPre: p ∈ pre a and Wv: W v ≤ W p + cost a
            using W-step-pre[OF a-in AddC False] by blast
          have pV: p ∈ vars Πe using preS pPre by blast
          have rho (StateVar (Original p)) = 1 using preO pPre by blast
          then have p1: rho-o (StateVar p) = 1 by (simp add: rho-o-def)
          have c-p-thr: c ≥ clip B (int B − int h + int (W p))
            by (rule carry[OF pV p1])
          have int B − int h + int (W v)
              ≤ (int B − int h + int (W p)) + int (cost a)
```

```
        using Wv by linarith
      then have clip B (int B − int h + int (W v))
          ≤ clip B ((int B − int h + int (W p)) + int (cost a))
        by (rule clip-mono)
      also have ... ≤ clip B (int B − int h + int (W p)) + cost a
        by (rule clip-add-le)
      also have ... ≤ c + cost a using c-p-thr by simp
      also have ... = c' using c'-eq by simp
      finally show ?thesis .
    qed
  next
    case DelC
    have rho (StateVar (Primed v)) = 0 using delP DelC by blast
    then have rho-p (StateVar v) = 0 by (simp add: rho-p-def)
    then show ?thesis using v1p by simp
  next
    case FrameC
    have w-in: v ∈ vars Πe − evars a using FrameC vV by auto
    then have eq1: rho (ReifEq (Original v)) = 1 using frameO by blast
    have meq: models (encode-eq-var v) rho
      by (rule trans-eq-var[OF mT]) (use w-in in blast)
    have eq2: rho (StateVar (Original v)) = rho (StateVar (Primed v))
      using encode-eq-var-forces[OF rho01 meq] eq1 by simp
    have rho-o (StateVar v) = 1
      using v1p eq2 by (simp add: rho-o-def rho-p-def)
    then have c ≥ clip B (int B − int h + int (W v))
      by (rule carry[OF vV])
    then show ?thesis using c'-eq by simp
  qed
 qed
 then have rho-p (kv-name i) = 1
  using kvg-forces[OF rho-p01 mP' i-lt] c-p unfolding v-def by simp
 then show ?thesis using rvg-forces[OF rho-p01 mP' i-lt] by blast
qed
qed
have rho-p max-name = 1
 using mxg-forces[OF rho-p01 mP'] kb-p rv-p by blast
then show eval-lit (map-literal primed-pvar-map (hc-out hmax-cert s')) rho = 1
 unfolding hmax-cert-def
 by (simp add: eval-lit-map-literal rho-p-def eval-lit-def)
qed
```

The hmax certificate is a valid heuristic certificate in the sense of Definition 4 for the evaluated state s.

```
theorem hmax-hc-valid: hc-valid Πe B as hmax-cert {s}
 unfolding hc-valid-def
 using hmax-state-lemma hmax-goal-lemma hmax-ind-lemma by blast
```

8.5 Structural conditions for use in the A* certificate

lemma *hmax-gates-nth-kv*: $i < n\text{-}v \Longrightarrow hmax\text{-}gates\ !\ i = kvg\ i$
by (*simp add*: *hmax-gates-def nth-append*)

lemma *hmax-gates-nth-rv*: $i < n\text{-}v \Longrightarrow hmax\text{-}gates\ !\ (n\text{-}v + i) = rvg\ i$
by (*simp add*: *hmax-gates-def nth-append*)

lemma *hmax-gates-nth-kb*: $hmax\text{-}gates\ !\ (2 * n\text{-}v) = kbg$
by (*simp add*: *hmax-gates-def nth-append*)

lemma *hmax-gates-nth-mx*: $hmax\text{-}gates\ !\ (2 * n\text{-}v + 1) = mxg$
by (*simp add*: *hmax-gates-def nth-append*)

lemma *hmax-gate-name-nth*:
assumes *p-lt*: $p < length\ hmax\text{-}gates$
shows $fst\ (hmax\text{-}gates\ !\ p) = Pos\ (ReifCert\ (nm\ p))$
proof −
consider $(KV)\ p < n\text{-}v \mid (RV)\ n\text{-}v \le p\ p < 2 * n\text{-}v$
$\mid (KB)\ p = 2 * n\text{-}v \mid (MX)\ p = 2 * n\text{-}v + 1$
using *p-lt length-hmax-gates* **by** *linarith*
then show *?thesis*
proof *cases*
case *KV*
then show *?thesis*
using *hmax-gates-nth-kv*[*OF KV*] **by** (*simp add*: *kvg-def k-gate-def kv-name-def*)
next
case *RV*
then obtain *q* **where** *p-eq*: $p = n\text{-}v + q$ **using** *le-iff-add* **by** *auto*
have *q-lt*: $q < n\text{-}v$ **using** *RV p-eq* **by** *simp*
show *?thesis*
unfolding *p-eq* **using** *hmax-gates-nth-rv*[*OF q-lt*]
by (*simp add*: *rvg-def rv-name-def*)
next
case *KB*
then show *?thesis*
using *hmax-gates-nth-kb* **by** (*simp add*: *kbg-def k-gate-def kb-name-def*)
next
case *MX*
then show *?thesis*
using *hmax-gates-nth-mx* **by** (*simp add*: *mxg-def max-name-def*)
qed
qed

lemma *hmax-names*:
$\forall (r,\ cs,\ A) \in set\ (hc\text{-}gates\ hmax\text{-}cert).\ \exists j.\ r = Pos\ (ReifCert\ (nm\ j))$
proof −
have $\bigwedge g.\ g \in set\ hmax\text{-}gates \Longrightarrow \exists j.\ fst\ g = Pos\ (ReifCert\ (nm\ j))$
proof −
fix *g* **assume** $g \in set\ hmax\text{-}gates$

```
    then obtain p where p-lt: p < length hmax-gates and g-eq: g = hmax-gates !
p
      by (auto simp: in-set-conv-nth)
    show ∃ j. fst g = Pos (ReifCert (nm j))
      using hmax-gate-name-nth[OF p-lt] g-eq by auto
  qed
  then show ?thesis by (fastforce simp: hmax-cert-def)
qed

lemma hmax-distinct:
  distinct (map (λ(r, cs, A). pvar-of-lit r) (hc-gates hmax-cert))
proof −
  note len = length-hmax-gates
  have map-eq: map (λ(r, cs, A). pvar-of-lit r) hmax-gates
      = map (λp. ReifCert (nm p)) [0..<2 * n-v + 2]
  proof (rule nth-equalityI)
    show length (map (λ(r, cs, A). pvar-of-lit r) hmax-gates)
        = length (map (λp. ReifCert (nm p)) [0..<2 * n-v + 2])
      by (simp add: len)
    fix p assume p < length (map (λ(r, cs, A). pvar-of-lit r) hmax-gates)
    then have p-lt: p < length hmax-gates by simp
    have (λ(r, cs, A). pvar-of-lit r) (hmax-gates ! p) = pvar-of-lit (fst (hmax-gates
! p))
      by (simp add: split-beta)
    also have ... = ReifCert (nm p)
      using hmax-gate-name-nth[OF p-lt] by (simp add: pvar-of-lit-def)
    finally show map (λ(r, cs, A). pvar-of-lit r) hmax-gates ! p
        = map (λp. ReifCert (nm p)) [0..<2 * n-v + 2] ! p
      using p-lt len by (auto simp: nth-append less-Suc-eq)
  qed
  have distinct (map (λp. ReifCert (nm p)) [0..<2 * n-v + 2])
    using nm-inj by (auto simp: distinct-map intro!: inj-onI dest: injD)
  then show ?thesis using map-eq by (simp add: hmax-cert-def)
qed

lemma hmax-out-in: hc-out hmax-cert s' ∈ fst ‘ set (hc-gates hmax-cert)
proof −
  have fst mxg = Pos max-name by (simp add: mxg-def)
  then show ?thesis using mxg-in-gates by (force simp: hmax-cert-def)
qed

lemma nth-in-take-hmax: j < i ⟹ i ≤ length xs ⟹ xs ! j ∈ set (take i xs)
  by (metis length-take min.absorb2 nth-mem nth-take)

lemma hmax-wf:
  ∀ i < length (hc-gates hmax-cert). case hc-gates hmax-cert ! i of (r, cs, A) ⇒
      (∀ x ∈ pvar-of-lit ‘ snd ‘ set cs.
        (∃ v. x = StateVar v) ∨ (∃ j. x = CostBit j)
        ∨ x ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates hmax-cert)))
```

```
proof (intro allI impI)
  fix i assume i-lt: i < length (hc-gates hmax-cert)
  have hcg: hc-gates hmax-cert = hmax-gates by (simp add: hmax-cert-def)
  have i-lt': i < length hmax-gates using i-lt by (simp add: hcg)
  have i-le: i ≤ length hmax-gates using i-lt' by simp
  consider (KV) i < n-v | (RV) n-v ≤ i i < 2 * n-v
    | (KB) i = 2 * n-v | (MX) i = 2 * n-v + 1
    using i-lt' length-hmax-gates by linarith
  then show case hc-gates hmax-cert ! i of (r, cs, A) ⇒
      (∀ x ∈ pvar-of-lit ` snd ` set cs.
        (∃ v. x = StateVar v) ∨ (∃ j. x = CostBit j)
        ∨ x ∈ pvar-of-lit ` fst ` set (take i (hc-gates hmax-cert)))
  proof cases
    case KV
    have g: hc-gates hmax-cert ! i = kvg i
      using hmax-gates-nth-kv[OF KV] hcg by simp
    have ∀ x ∈ pvar-of-lit ` snd ` set (k-gate-body B). (∃ j. x = CostBit j)
      by (auto simp: k-gate-body-def pvar-of-lit-def)
    then show ?thesis unfolding g kvg-def k-gate-def by auto
  next
    case RV
    then obtain q where i-eq: i = n-v + q using le-iff-add by auto
    have q-lt: q < n-v using RV i-eq by simp
    have g: hc-gates hmax-cert ! i = rvg q
      using hmax-gates-nth-rv[OF q-lt] hcg i-eq by simp
    have kv-in: kv-name q ∈ pvar-of-lit ` fst ` set (take i (hc-gates hmax-cert))
    proof −
      have q < i using q-lt i-eq by simp
      then have hmax-gates ! q ∈ set (take i hmax-gates)
        using nth-in-take-hmax i-le by blast
      moreover have fst (hmax-gates ! q) = Pos (kv-name q)
        using hmax-gates-nth-kv[OF q-lt] by (simp add: kvg-def k-gate-def)
      ultimately show ?thesis using hcg by (force simp: pvar-of-lit-def)
    qed
    have ∀ x ∈ pvar-of-lit ` snd ` set (map (λl. (1, l)) (rv-lits q)).
        (∃ v. x = StateVar v)
        ∨ x ∈ pvar-of-lit ` fst ` set (take i (hc-gates hmax-cert))
      using kv-in by (auto simp: rv-lits-def pvar-of-lit-def)
    then show ?thesis unfolding g rvg-def by auto
  next
    case KB
    have g: hc-gates hmax-cert ! i = kbg
      using hmax-gates-nth-kb hcg KB by simp
    have ∀ x ∈ pvar-of-lit ` snd ` set (k-gate-body B). (∃ j. x = CostBit j)
      by (auto simp: k-gate-body-def pvar-of-lit-def)
    then show ?thesis unfolding g kbg-def k-gate-def by auto
  next
    case MX
    have g: hc-gates hmax-cert ! i = mxg
```

```
    using hmax-gates-nth-mx hcg MX by simp
  have ∀ x ∈ pvar-of-lit ‘ snd ‘ set (map (λl. (1, l)) max-lits).
     x ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates hmax-cert))
  proof
    fix x assume x ∈ pvar-of-lit ‘ snd ‘ set (map (λl. (1, l)) max-lits)
    then consider (Base) x = kb-name
     | (Var) q where q < n-v x = rv-name q
     by (auto simp: max-lits-def pvar-of-lit-def)
    then show x ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates hmax-cert))
    proof cases
     case Base
     have 2 * n-v < i using MX by simp
     then have hmax-gates ! (2 * n-v) ∈ set (take i hmax-gates)
       using nth-in-take-hmax i-le by blast
     moreover have fst (hmax-gates ! (2 * n-v)) = Pos kb-name
       using hmax-gates-nth-kb by (simp add: kbg-def k-gate-def)
     ultimately show ?thesis using hcg Base by (force simp: pvar-of-lit-def)
    next
     case Var
     have n-v + q < i using Var(1) MX by simp
     then have hmax-gates ! (n-v + q) ∈ set (take i hmax-gates)
       using nth-in-take-hmax i-le by blast
     moreover have fst (hmax-gates ! (n-v + q)) = Pos (rv-name q)
       using hmax-gates-nth-rv[OF Var(1)] by (simp add: rvg-def)
     ultimately show ?thesis using hcg Var(2) by (force simp: pvar-of-lit-def)
    qed
  qed
  then show ?thesis unfolding g mxg-def by auto
 qed
qed

end

end
```

9 Efficient Pattern Databases

```
theory Efficient-PDB
 imports PDB-Certificates
begin
```

Efficiently proof-logging PDB heuristics (paper appendix: Lemma 18, Definition 5 and Lemmas 19–29). Definition 5 restricts the PDB circuit to the abstract states with finite goal distance — the part of the abstract state space an efficient PDB implementation actually traverses — and adds a gate $r\infty$ (equation (24)) that is true exactly when the current abstract state is none of the finite-distance ones. The output (equation (25)) is the disjunction of $r\infty$ and the per-state threshold gates of equation (23).

The distance table is now partial: $d\ s\alpha$ together with a finiteness predicate *fin* $s\alpha$ ($d(s\alpha) < \infty$ in the paper). The two table conditions become: abstract goal states are finite with distance 0, and finiteness propagates backwards along applicable abstract transitions together with the triangle inequality (so an infinite-distance state can never reach a finite-distance one).

Lemma 18 is a statement about the *length* of a CPR derivation of the covering disjunction over all extensions of a partial assignment; under the semantic RUP over-approximation used throughout this formalization its content reduces to the existence of the witnessing extension in any 0/1 model (*assignment-extension-witness* below); the step-count aspect is out of scope, as everywhere else.

9.1 Lemma 18, semantic form

Any 0/1 model whose Y-variables follow the pattern of a partial assignment determines a unique total extension on $Z \supseteq Y$: the set of Z-variables that are true. With exact-state gates for all extensions this witness forces the covering disjunction (21) of the paper.

```
lemma assignment-extension-witness:
  fixes rho :: 'w ⇒ nat and f :: 'u ⇒ 'w
  assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
    and YZ: Y ⊆ Z
    and al-pat: ∀ v ∈ Y. rho (f v) = (if v ∈ al then 1 else 0)
  shows ∃ t ⊆ Z. t ∩ Y = al ∩ Y ∧ (∀ v ∈ Z. rho (f v) = (if v ∈ t then 1 else 0))
proof −
  define t where t = {v ∈ Z. rho (f v) = 1}
  have t-sub: t ⊆ Z unfolding t-def by auto
  have t-Y: t ∩ Y = al ∩ Y
  proof (rule set-eqI)
    fix v
    show v ∈ t ∩ Y ⟷ v ∈ al ∩ Y
    proof (cases v ∈ Y)
      case True
      then have rho (f v) = (if v ∈ al then 1 else 0) using al-pat by blast
      then show ?thesis using True YZ unfolding t-def by (auto split: if-splits)
    next
      case False
      then show ?thesis by auto
    qed
  qed
  have t-pat: ∀ v ∈ Z. rho (f v) = (if v ∈ t then 1 else 0)
  proof
    fix v assume vZ: v ∈ Z
    show rho (f v) = (if v ∈ t then 1 else 0)
    proof (cases v ∈ t)
      case True
```

then show *?thesis* **unfolding** *t-def* **by** *simp*
next
case *False*
then have *rho* (*f v*) ≠ *1* **using** *vZ* **unfolding** *t-def* **by** *simp*
then show *?thesis* **using** *rho01*[*rule-format*, *of f v*] *False* **by** *simp*
qed
qed
show *?thesis* **using** *t-sub t-Y t-pat* **by** *blast*
qed

9.2 The efficient PDB locale

locale *efficient-pdb* =
fixes Π*e* :: *'u*::*linorder strips-task*
and *B* :: *nat*
and *P* :: *'u set*
and *d* :: *'u state* ⇒ *nat*
and *fin* :: *'u state* ⇒ *bool*
and *Ss* :: *'u state list*
and *as* :: *'u action list*
and *nm* :: *nat* ⇒ *'u*
assumes *fin-vars*: *finite* (*vars* Π*e*)
and *P-sub*: *P* ⊆ *vars* Π*e*
and *Ss-set*: *set Ss* = {*sa*. *sa* ⊆ *P* ∧ *fin sa*}
and *Ss-dist*: *distinct Ss*
and *as-states*: ∀ *a* ∈ *set as*. *pre a* ⊆ *vars* Π*e* ∧ *add a* ⊆ *vars* Π*e* ∧ *del a* ⊆ *vars* Π*e*
and *as-disjoint*: ∀ *a* ∈ *set as*. *add a* ∩ *del a* = {}
and *fin-goal*: ⋀*sa*. *sa* ⊆ *P* ⟹ *goal* Π*e* ∩ *P* ⊆ *sa* ⟹ *fin sa* ∧ *d sa* = *0*
and *d-triangle*: ⋀*sa a*. *sa* ⊆ *P* ⟹ *a* ∈ *set as* ⟹ *pre a* ∩ *P* ⊆ *sa* ⟹
fin ((*sa* − *del a*) ∪ (*add a* ∩ *P*)) ⟹
fin sa ∧ *d sa* ≤ *d* ((*sa* − *del a*) ∪ (*add a* ∩ *P*)) + *cost a*
and *nm-inj*: *inj nm*
begin

lemma *fin-P*: *finite P*
using *fin-vars P-sub* **by** (*rule rev-finite-subset*)

definition *n-s* :: *nat* **where**
n-s = *length Ss*

definition *sa-i* :: *nat* ⇒ *'u state* **where**
sa-i i = *Ss* ! *i*

lemma *sa-i-sub*: *i* < *n-s* ⟹ *sa-i i* ⊆ *P*
using *Ss-set nth-mem* **unfolding** *n-s-def sa-i-def* **by** *fastforce*

lemma *sa-i-fin*: *i* < *n-s* ⟹ *fin* (*sa-i i*)
using *Ss-set nth-mem* **unfolding** *n-s-def sa-i-def* **by** *fastforce*

lemma *fin-mem-nth*:
  assumes $sa \subseteq P$ **and** *fin sa*
  shows $\exists i.\ i < n\text{-}s \wedge sa\text{-}i\ i = sa$
proof −
  have $sa \in set\ Ss$ **using** *assms Ss-set* **by** *auto*
  then show *?thesis*
    unfolding *n-s-def sa-i-def* **by** (*auto simp*: *in-set-conv-nth*)
qed

9.2.1 Gate names

definition *abs-name* :: *nat* ⇒ *'u pvar* **where**
  abs-name i = *ReifCert* (*nm i*)

definition *kk-name* :: *nat* ⇒ *'u pvar* **where**
  kk-name i = *ReifCert* (*nm* (*n-s* + *i*))

definition *thr-name* :: *nat* ⇒ *'u pvar* **where**
  thr-name i = *ReifCert* (*nm* (*2* ∗ *n-s* + *i*))

definition *inf-name* :: *'u pvar* **where**
  inf-name = *ReifCert* (*nm* (*3* ∗ *n-s*))

definition *pout-name* :: *'u pvar* **where**
  pout-name = *ReifCert* (*nm* (*3* ∗ *n-s* + *1*))

9.2.2 Gates

Equations (22), (23) as in the plain PDB circuit, but only for the finite-distance abstract states.

definition *abs-lits* :: *nat* ⇒ *'u pvar literal list* **where**
  abs-lits i =
    map (λ*v*. *if v* ∈ *sa-i i then Pos* (*StateVar v*) *else Neg* (*StateVar v*))
      (*sorted-list-of-set P*)

definition *absg* :: *nat* ⇒ *'u pvar literal* × (*nat* × *'u pvar literal*) *list* × *nat* **where**
  absg i = (*Pos* (*abs-name i*), *map* (λ*l*. (*1*, *l*)) (*abs-lits i*), *length* (*abs-lits i*))

definition *kgg* :: *nat* ⇒ *'u pvar literal* × (*nat* × *'u pvar literal*) *list* × *nat* **where**
  kgg i = *k-gate* (*Pos* (*kk-name i*)) *B* (*int B* − *int* (*d* (*sa-i i*)))

definition *thr-lits* :: *nat* ⇒ *'u pvar literal list* **where**
  thr-lits i = [*Pos* (*abs-name i*), *Pos* (*kk-name i*)]

definition *thrg* :: *nat* ⇒ *'u pvar literal* × (*nat* × *'u pvar literal*) *list* × *nat* **where**
  thrg i = (*Pos* (*thr-name i*), *map* (λ*l*. (*1*, *l*)) (*thr-lits i*), *length* (*thr-lits i*))

Equation (24): $r\infty$ is the conjunction of the negated abstract-state gates — true iff the current abstract state is none of the finite-distance ones.

definition *inf-lits* :: *'u pvar literal list* **where**
inf-lits = *map* (λ*i*. *Neg* (*abs-name i*)) [*0*..<*n-s*]

definition *infg* :: *'u pvar literal* × (*nat* × *'u pvar literal*) *list* × *nat* **where**
infg = (*Pos inf-name*, *map* (λ*l*. (*1*, *l*)) *inf-lits*, *length inf-lits*)

Equation (25): the output disjunction of $r\infty$ and the threshold gates.

definition *pout-lits* :: *'u pvar literal list* **where**
pout-lits = *Pos inf-name* # *map* (λ*i*. *Pos* (*thr-name i*)) [*0*..<*n-s*]

definition *poutg* :: *'u pvar literal* × (*nat* × *'u pvar literal*) *list* × *nat* **where**
poutg = (*Pos pout-name*, *map* (λ*l*. (*1*, *l*)) *pout-lits*, *1*)

definition *epdb-gates* ::
(*'u pvar literal* × (*nat* × *'u pvar literal*) *list* × *nat*) *list* **where**
epdb-gates = *map absg* [*0*..<*n-s*] @ *map kgg* [*0*..<*n-s*] @ *map thrg* [*0*..<*n-s*]
@ [*infg*, *poutg*]

definition *epdb-cert* :: (*'u*) *heuristic-cert* **where**
epdb-cert = (| *hc-gates* = *epdb-gates*,
hc-out = (λ*s*. *Pos pout-name*),
hc-h = (λ*s*. *if fin* (*s* ∩ *P*) *then d* (*s* ∩ *P*) *else B*) |)

9.2.3 Basic structure of the gate list

lemma *length-epdb-gates*: *length epdb-gates* = *3* ∗ *n-s* + *2*
by (*simp add*: *epdb-gates-def*)

lemma *absg-in-gates*: *i* < *n-s* ⟹ *absg i* ∈ *set epdb-gates*
by (*simp add*: *epdb-gates-def*)

lemma *kgg-in-gates*: *i* < *n-s* ⟹ *kgg i* ∈ *set epdb-gates*
by (*simp add*: *epdb-gates-def*)

lemma *thrg-in-gates*: *i* < *n-s* ⟹ *thrg i* ∈ *set epdb-gates*
by (*simp add*: *epdb-gates-def*)

lemma *infg-in-gates*: *infg* ∈ *set epdb-gates*
by (*simp add*: *epdb-gates-def*)

lemma *poutg-in-gates*: *poutg* ∈ *set epdb-gates*
by (*simp add*: *epdb-gates-def*)

lemma *models-epdb-gate*:
assumes *m*: *models* (*hc-constraints epdb-cert*) *rho*
and *g-in*: (*r*, *cs*, *A*) ∈ *set epdb-gates*
shows *models* (*reification r cs A*) *rho*
proof (*rule models-mono*[*OF m*])
show *reification r cs A* ⊆ *hc-constraints epdb-cert*
using *g-in* **unfolding** *hc-constraints-def epdb-cert-def* **by** *fastforce*

qed

lemma *models-absg*:
assumes *models* (*hc-constraints epdb-cert*) *rho* **and** $i < n\text{-}s$
shows *models* (*reification* (*Pos* (*abs-name i*))
(*map* (λ*l*. (*1*, *l*)) (*abs-lits i*)) (*length* (*abs-lits i*))) *rho*
using *models-epdb-gate*[*OF assms*(*1*)] *absg-in-gates*[*OF assms*(*2*)]
by (*simp add*: *absg-def*)

lemma *models-kgg*:
assumes *models* (*hc-constraints epdb-cert*) *rho* **and** $i < n\text{-}s$
shows *models* (*reification* (*Pos* (*kk-name i*)) (*k-gate-body B*)
(*clip B* (*int B* − *int* (*d* (*sa-i i*))))) *rho*
using *models-epdb-gate*[*OF assms*(*1*)] *kgg-in-gates*[*OF assms*(*2*)]
by (*simp add*: *kgg-def k-gate-def*)

lemma *models-thrg*:
assumes *models* (*hc-constraints epdb-cert*) *rho* **and** $i < n\text{-}s$
shows *models* (*reification* (*Pos* (*thr-name i*))
(*map* (λ*l*. (*1*, *l*)) (*thr-lits i*)) (*length* (*thr-lits i*))) *rho*
using *models-epdb-gate*[*OF assms*(*1*)] *thrg-in-gates*[*OF assms*(*2*)]
by (*simp add*: *thrg-def*)

lemma *models-infg*:
assumes *models* (*hc-constraints epdb-cert*) *rho*
shows *models* (*reification* (*Pos inf-name*)
(*map* (λ*l*. (*1*, *l*)) *inf-lits*) (*length inf-lits*)) *rho*
using *models-epdb-gate*[*OF assms*] *infg-in-gates*
by (*simp add*: *infg-def*)

lemma *models-poutg*:
assumes *models* (*hc-constraints epdb-cert*) *rho*
shows *models* (*reification* (*Pos pout-name*) (*map* (λ*l*. (*1*, *l*)) *pout-lits*) *1*) *rho*
using *models-epdb-gate*[*OF assms*] *poutg-in-gates*
by (*simp add*: *poutg-def*)

9.2.4 Semantics of the gates

The abstract state encoded by a 0/1 assignment of the state variables (the witness of Lemma 18 for $Z = P$).

definition *abs-state* :: (′*u pvar* ⇒ *nat*) ⇒ ′*u state* **where**
abs-state rho = {*v* ∈ *P*. *rho* (*StateVar v*) = *1*}

lemma *abs-state-sub*: *abs-state rho* ⊆ *P*
unfolding *abs-state-def* **by** *auto*

lemma *abs-lits-sem*:
assumes *rho01*: ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
shows (∀ *l* ∈ *set* (*abs-lits i*). *eval-lit l rho* = *1*)

```
    ⟷ (∀ v ∈ P. rho (StateVar v) = (if v ∈ sa-i i then 1 else 0))
proof −
  have set-eq: set (sorted-list-of-set P) = P
    using fin-P by simp
  have lit-sem: ⋀v. eval-lit (if v ∈ sa-i i then Pos (StateVar v)
       else Neg (StateVar v)) rho = 1
     ⟷ rho (StateVar v) = (if v ∈ sa-i i then 1 else 0)
  proof −
    fix v
    show eval-lit (if v ∈ sa-i i then Pos (StateVar v) else Neg (StateVar v)) rho =
1
     ⟷ rho (StateVar v) = (if v ∈ sa-i i then 1 else 0)
     using rho01[rule-format, of StateVar v] by (auto simp: eval-lit-def)
  qed
  show ?thesis
  proof
    assume all1: ∀ l ∈ set (abs-lits i). eval-lit l rho = 1
    show ∀ v ∈ P. rho (StateVar v) = (if v ∈ sa-i i then 1 else 0)
    proof
      fix v assume vP: v ∈ P
      have (if v ∈ sa-i i then Pos (StateVar v) else Neg (StateVar v))
           ∈ set (abs-lits i)
        unfolding abs-lits-def using vP set-eq by auto
      then have eval-lit (if v ∈ sa-i i then Pos (StateVar v)
         else Neg (StateVar v)) rho = 1
        using all1 by blast
      then show rho (StateVar v) = (if v ∈ sa-i i then 1 else 0)
        using lit-sem[of v] by simp
    qed
  next
    assume enc: ∀ v ∈ P. rho (StateVar v) = (if v ∈ sa-i i then 1 else 0)
    show ∀ l ∈ set (abs-lits i). eval-lit l rho = 1
    proof
      fix l assume l ∈ set (abs-lits i)
      then obtain v where vP: v ∈ P
        and l-eq: l = (if v ∈ sa-i i then Pos (StateVar v) else Neg (StateVar v))
        unfolding abs-lits-def using set-eq by auto
      show eval-lit l rho = 1
        using enc vP lit-sem[of v] unfolding l-eq by simp
    qed
  qed
qed

lemma absg-forces:
  assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
    and m: models (hc-constraints epdb-cert) rho
    and i-lt: i < n-s
  shows rho (abs-name i) = 1
     ⟷ (∀ v ∈ P. rho (StateVar v) = (if v ∈ sa-i i then 1 else 0))
```

```
proof −
  have eval-lit (Pos (abs-name i)) rho = 1
      ⟷ (∀ l ∈ set (abs-lits i). eval-lit l rho = 1)
    by (rule conj-gate-forces[OF rho01 models-absg[OF m i-lt]])
  then show ?thesis
    unfolding abs-lits-sem[OF rho01] by (simp add: eval-lit-def)
qed

lemma absg-forces-eq:
  assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
    and m: models (hc-constraints epdb-cert) rho
    and i-lt: i < n-s
  shows rho (abs-name i) = 1 ⟷ abs-state rho = sa-i i
proof −
  have rho (abs-name i) = 1
      ⟷ (∀ v ∈ P. rho (StateVar v) = (if v ∈ sa-i i then 1 else 0))
    by (rule absg-forces[OF rho01 m i-lt])
  also have ... ⟷ abs-state rho = sa-i i
  proof
    assume A: ∀ v ∈ P. rho (StateVar v) = (if v ∈ sa-i i then 1 else 0)
    show abs-state rho = sa-i i
    proof (rule set-eqI)
      fix v
      show v ∈ abs-state rho ⟷ v ∈ sa-i i
      proof (cases v ∈ P)
        case True
       then show ?thesis using A unfolding abs-state-def by (auto split: if-splits)
      next
        case False
       then show ?thesis using sa-i-sub[OF i-lt] unfolding abs-state-def by auto
      qed
    qed
  next
    assume E: abs-state rho = sa-i i
    show ∀ v ∈ P. rho (StateVar v) = (if v ∈ sa-i i then 1 else 0)
    proof
      fix v assume vP: v ∈ P
      show rho (StateVar v) = (if v ∈ sa-i i then 1 else 0)
      proof (cases v ∈ sa-i i)
        case True
        then have v ∈ abs-state rho using E by simp
        then show ?thesis using True unfolding abs-state-def by simp
      next
        case False
        then have v ∉ abs-state rho using E by simp
       then have rho (StateVar v) ≠ 1 using vP unfolding abs-state-def by simp
        then show ?thesis using rho01[rule-format, of StateVar v] False by simp
      qed
    qed
```

```
  qed
  finally show ?thesis .
qed

lemma kgg-forces:
  assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
    and m: models (hc-constraints epdb-cert) rho
    and i-lt: i < n-s
  shows rho (kk-name i) = 1
     ⟷ bits-val (bits-needed B) CostBit rho ≥ clip B (int B − int (d (sa-i i)))
proof −
  have eval-lit (Pos (kk-name i)) rho = 1
       ⟷ bits-val (bits-needed B) CostBit rho ≥ clip B (int B − int (d (sa-i i)))
    by (rule k-gate-forces[OF rho01 models-kgg[OF m i-lt]])
  then show ?thesis by (simp add: eval-lit-def)
qed

lemma thrg-forces:
  assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
    and m: models (hc-constraints epdb-cert) rho
    and i-lt: i < n-s
  shows rho (thr-name i) = 1 ⟷ rho (abs-name i) = 1 ∧ rho (kk-name i) = 1
proof −
  have eval-lit (Pos (thr-name i)) rho = 1
       ⟷ (∀ l ∈ set (thr-lits i). eval-lit l rho = 1)
    by (rule conj-gate-forces[OF rho01 models-thrg[OF m i-lt]])
  then show ?thesis by (simp add: thr-lits-def eval-lit-def)
qed

lemma infg-forces:
  assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
    and m: models (hc-constraints epdb-cert) rho
  shows rho inf-name = 1 ⟷ (∀ i < n-s. rho (abs-name i) = 0)
proof −
  have neg-sem: ⋀i. eval-lit (Neg (abs-name i)) rho = 1 ⟷ rho (abs-name i) =
0
  proof −
    fix i
    show eval-lit (Neg (abs-name i)) rho = 1 ⟷ rho (abs-name i) = 0
      by (cases rho (abs-name i)) (auto simp: eval-lit-def)
  qed
  have eval-lit (Pos inf-name) rho = 1
       ⟷ (∀ l ∈ set inf-lits. eval-lit l rho = 1)
    by (rule conj-gate-forces[OF rho01 models-infg[OF m]])
  also have ... ⟷ (∀ i < n-s. rho (abs-name i) = 0)
  proof
    assume L: ∀ l ∈ set inf-lits. eval-lit l rho = 1
    show ∀ i < n-s. rho (abs-name i) = 0
    proof (intro allI impI)
```

```
    fix i assume i < n-s
    then have Neg (abs-name i) ∈ set inf-lits by (auto simp: inf-lits-def)
    then have eval-lit (Neg (abs-name i)) rho = 1 using L by blast
    then show rho (abs-name i) = 0 using neg-sem by blast
  qed
 next
  assume R: ∀ i < n-s. rho (abs-name i) = 0
  show ∀ l ∈ set inf-lits. eval-lit l rho = 1
  proof
    fix l assume l ∈ set inf-lits
    then obtain i where i < n-s and l = Neg (abs-name i)
      by (auto simp: inf-lits-def)
    then show eval-lit l rho = 1 using R neg-sem by blast
  qed
 qed
 finally show ?thesis by (simp add: eval-lit-def)
qed

lemma poutg-forces:
 assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
   and m: models (hc-constraints epdb-cert) rho
 shows rho pout-name = 1
   ⟷ rho inf-name = 1 ∨ (∃ i < n-s. rho (thr-name i) = 1)
proof −
 have eval-lit (Pos pout-name) rho = 1
    ⟷ (∃ l ∈ set pout-lits. eval-lit l rho = 1)
   by (rule disj-gate-forces[OF rho01 models-poutg[OF m]])
 then show ?thesis by (auto simp: pout-lits-def eval-lit-def)
qed
```

If the encoded abstract state has infinite distance, all abstract-state gates are false and $r\infty$ fires (second case of Lemma 19).

```
lemma not-fin-inf:
 assumes rho01: ∀ x. rho x = 0 ∨ rho x = 1
   and m: models (hc-constraints epdb-cert) rho
   and nf: ¬ fin (abs-state rho)
 shows rho inf-name = 1
proof −
 have ∀ i < n-s. rho (abs-name i) = 0
 proof (intro allI impI)
   fix i assume i-lt: i < n-s
   have abs-state rho ≠ sa-i i using nf sa-i-fin[OF i-lt] by auto
    then have rho (abs-name i) ≠ 1 using absg-forces-eq[OF rho01 m i-lt] by
simp
   then show rho (abs-name i) = 0 using rho01[rule-format] by blast
 qed
 then show ?thesis using infg-forces[OF rho01 m] by blast
qed
```

9.3 Lemma 19: the state lemma

lemma *epdb-state-lemma*: *hc-state-lemma* Π*e B epdb-cert s*
unfolding *hc-state-lemma-def*
proof (*intro allI impI*)
fix *rho* :: *'u pvar* ⇒ *nat*
assume *rho01*: ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
and *m*: *models* (*hc-constraints epdb-cert*) *rho*
and *sv*: ∀ *v* ∈ *vars* Π*e*. *rho* (*StateVar v*) = (*if v* ∈ *s then 1 else 0*)
and *cb*: *bits-val* (*bits-needed B*) *CostBit rho* ≥ *clip B* (*int B* − *int* (*hc-h epdb-cert s*))
have *abs-s*: *abs-state rho* = *s* ∩ *P*
proof (*rule set-eqI*)
fix *v*
show *v* ∈ *abs-state rho* ⟷ *v* ∈ *s* ∩ *P*
proof (*cases v* ∈ *P*)
case *True*
then have *rho* (*StateVar v*) = (*if v* ∈ *s then 1 else 0*)
using *sv P-sub* **by** *blast*
then show *?thesis* **using** *True* **unfolding** *abs-state-def* **by** (*auto split*: *if-splits*)
next
case *False*
then show *?thesis* **unfolding** *abs-state-def* **by** *auto*
qed
qed
show *eval-lit* (*hc-out epdb-cert s*) *rho* = *1*
proof (*cases fin* (*s* ∩ *P*))
case *True*
obtain *i* **where** *i-lt*: *i* < *n-s* **and** *i-s*: *sa-i i* = *s* ∩ *P*
using *fin-mem-nth*[*OF - True*] **by** *auto*
have *abs1*: *rho* (*abs-name i*) = *1*
using *absg-forces-eq*[*OF rho01 m i-lt*] *abs-s i-s* **by** *simp*
have *kk1*: *rho* (*kk-name i*) = *1*
proof −
have *hc-h epdb-cert s* = *d* (*sa-i i*) **using** *True* **by** (*simp add*: *epdb-cert-def i-s*)
then show *?thesis* **using** *kgg-forces*[*OF rho01 m i-lt*] *cb* **by** *simp*
qed
have *thr1*: *rho* (*thr-name i*) = *1*
using *thrg-forces*[*OF rho01 m i-lt*] *abs1 kk1* **by** *blast*
then have *rho pout-name* = *1*
using *poutg-forces*[*OF rho01 m*] *i-lt* **by** *blast*
then show *?thesis* **by** (*simp add*: *epdb-cert-def eval-lit-def*)
next
case *False*
have *rho inf-name* = *1*
by (*rule not-fin-inf*[*OF rho01 m*]) (*simp add*: *abs-s False*)
then have *rho pout-name* = *1*
using *poutg-forces*[*OF rho01 m*] **by** *blast*

then show *?thesis* **by** (*simp add*: *epdb-cert-def eval-lit-def*)
qed
qed

9.4 Lemma 20: the goal lemma

lemma *epdb-goal-lemma*: *hc-goal-lemma* Π*e B epdb-cert s*
unfolding *hc-goal-lemma-def*
proof (*intro allI impI*)
fix *rho* :: *'u pvar* ⇒ *nat*
assume *rho01*: ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
and *m*: *models* (*hc-constraints epdb-cert*) *rho*
and *gv*: ∀ *v* ∈ *goal* Π*e*. *rho* (*StateVar v*) = *1*
and *out1*: *eval-lit* (*hc-out epdb-cert s*) *rho* = *1*
have *p1*: *rho pout-name* = *1*
using *out1* **by** (*simp add*: *epdb-cert-def eval-lit-def*)
have *goal-t*: *goal* Π*e* ∩ *P* ⊆ *abs-state rho*
using *gv* **unfolding** *abs-state-def* **by** *auto*
from *poutg-forces*[*OF rho01 m*] *p1*
consider (*Inf*) *rho inf-name* = *1* | (*Thr*) *i* **where** *i* < *n-s rho* (*thr-name i*) = *1*
by *blast*
then show *bits-val* (*bits-needed B*) *CostBit rho* ≥ *B*
proof *cases*
case *Inf*
— Impossible: the encoded abstract state contains the abstract goal and is therefore a finite-distance state, contradicting *r*∞.
have *fin-t*: *fin* (*abs-state rho*)
using *fin-goal*[*OF abs-state-sub goal-t*] **by** *blast*
obtain *i* **where** *i-lt*: *i* < *n-s* **and** *i-s*: *sa-i i* = *abs-state rho*
using *fin-mem-nth*[*OF abs-state-sub fin-t*] **by** *auto*
have *rho* (*abs-name i*) = *1*
using *absg-forces-eq*[*OF rho01 m i-lt*] *i-s* **by** *simp*
moreover have *rho* (*abs-name i*) = *0*
using *infg-forces*[*OF rho01 m*] *Inf i-lt* **by** *blast*
ultimately have *False* **by** *simp*
then show *?thesis* **..**
next
case *Thr*
have *abs1*: *rho* (*abs-name i*) = *1* **and** *kk1*: *rho* (*kk-name i*) = *1*
using *thrg-forces*[*OF rho01 m Thr*(*1*)] *Thr*(*2*) **by** *blast*+
have *i-s*: *sa-i i* = *abs-state rho*
using *absg-forces-eq*[*OF rho01 m Thr*(*1*)] *abs1* **by** *simp*
have *goal-in*: *goal* Π*e* ∩ *P* ⊆ *sa-i i*
using *goal-t i-s* **by** *simp*
have *d0*: *d* (*sa-i i*) = *0*
using *fin-goal*[*OF sa-i-sub*[*OF Thr*(*1*)] *goal-in*] **by** *blast*
have *clip B* (*int B* − *int* (*d* (*sa-i i*))) = *B*
using *d0* **by** *simp*
then show *?thesis* **using** *kgg-forces*[*OF rho01 m Thr*(*1*)] *kk1* **by** *simp*

qed
qed

9.5 Lemmas 21–29: the inductivity lemma

Paper Lemmas 21–24 treat the threshold-gate cases (finite source state, finite or infinite successor), Lemmas 25–28 the $r\infty$ cases, and Lemma 29 combines them. Semantically all of them collapse into one chase with a case analysis on which disjunct of the output gate is true on the unprimed side and on whether the abstract successor state has finite distance.

lemma *epdb-ind-lemma*: *hc-ind-lemma* Π*e B as epdb-cert s*
unfolding *hc-ind-lemma-def*
proof (*intro allI impI*)
fix *rho* :: *'u var pvar* ⇒ *nat*
assume *rho01*: ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
and *mO*: *models* (*circuit-constraints*
(*orig-circuit* (*hc-gates epdb-cert*, *hc-out epdb-cert s*))) *rho*
and *mP*: *models* (*circuit-constraints*
(*primed-circuit* (*hc-gates epdb-cert*, *hc-out epdb-cert s*))) *rho*
and *mT*: *models* (*encode-transition as* (*vars* Π*e*) *B*) *rho*
and *mB*: *models* (*encode-cost-ge B*) *rho*
and *rT*: *rho ReifT* = *1*
and *outO*: *eval-lit* (*map-literal* (*map-pvar Original*) (*hc-out epdb-cert s*)) *rho* = *1*
define *rho-o* **where** *rho-o* = *rho* ∘ *map-pvar Original*
define *rho-p* **where** *rho-p* = *rho* ∘ *primed-pvar-map*
have *rho-o01*: ∀ *x*. *rho-o x* = *0* ∨ *rho-o x* = *1*
unfolding *rho-o-def* **by** (*rule rho01-comp*[*OF rho01*])
have *rho-p01*: ∀ *x*. *rho-p x* = *0* ∨ *rho-p x* = *1*
unfolding *rho-p-def* **by** (*rule rho01-comp*[*OF rho01*])
have *mO''*: *models* (*circuit-constraints* (*hc-gates epdb-cert*, *hc-out epdb-cert s*)) *rho-o*
using *mO models-circuit-constraints-lift*[*of map-pvar Original*
(*hc-gates epdb-cert*, *hc-out epdb-cert s*) *rho*]
unfolding *rho-o-def orig-circuit-def* **by** *blast*
have *mO'*: *models* (*hc-constraints epdb-cert*) *rho-o*
using *mO'' hc-constraints-eq-circuit*[*of epdb-cert hc-out epdb-cert s*] **by** *simp*
have *mP''*: *models* (*circuit-constraints* (*hc-gates epdb-cert*, *hc-out epdb-cert s*)) *rho-p*
using *mP models-circuit-constraints-lift*[*of primed-pvar-map*
(*hc-gates epdb-cert*, *hc-out epdb-cert s*) *rho*]
unfolding *rho-p-def primed-circuit-def* **by** *blast*
have *mP'*: *models* (*hc-constraints epdb-cert*) *rho-p*
using *mP'' hc-constraints-eq-circuit*[*of epdb-cert hc-out epdb-cert s*] **by** *simp*
define *c* **where** *c* = *bits-val* (*bits-needed B*) *CostBit rho*
define *c'* **where** *c'* = *bits-val* (*bits-needed B*) *PrimedCostBit rho*
have *c-o*: *bits-val* (*bits-needed B*) *CostBit rho-o* = *c*
by (*simp add*: *rho-o-def c-def bits-val-def*)

have *c-p*: *bits-val* (*bits-needed B*) *CostBit rho-p* = c'
by (*simp add*: *rho-p-def c'-def bits-val-def*)
— The selected action.
have *mSel*: *models* (*action-selection-reif* (*action-reifs as*)) *rho*
by (*rule trans-sel*[*OF mT*])
obtain *rA* **where** *rA-in*: *rA* ∈ *set* (*action-reifs as*) **and** *rA1*: *eval-lit rA rho* = *1*
using *action-selection-forces*[*OF rho01 mSel*] *rT* **by** *blast*
obtain *j* **where** *j-lt*: *j* < *length as* **and** *rA-eq*: *rA* = *Pos* (*ReifAction j*)
using *rA-in* **unfolding** *action-reifs-def* **by** *auto*
define *a* **where** *a* = *as* ! *j*
have *a-in*: *a* ∈ *set as* **unfolding** *a-def* **by** (*rule nth-mem*[*OF j-lt*])
have *satAC*: *satisfies* (*action-constraint* (*Pos* (*ReifAction j*)) *a* (*vars* Π*e*) *B*) *rho*
unfolding *a-def* **by** (*rule trans-action-constraint*[*OF mT j-lt*])
have *preS*: *pre a* ⊆ *vars* Π*e* **and** *addS*: *add a* ⊆ *vars* Π*e* **and** *delS*: *del a* ⊆ *vars* Π*e*
using *bspec*[*OF as-states a-in*] **by** *auto*
have *outj*: *eval-lit* (*Pos* (*ReifAction j*)) *rho* = *1*
using *rA1 rA-eq* **by** *simp*
note *ext* = *action-constraint-extract*[*OF rho01 satAC outj fin-vars preS addS delS*]
have *delta1*: *rho* (*ReifDeltaCost* (*cost a*)) = *1* **using** *ext* **by** *blast*
have *preO*: ∀ *w* ∈ *pre a*. *rho* (*StateVar* (*Original w*)) = *1* **using** *ext* **by** *blast*
have *addP*: ∀ *w* ∈ *add a* − *del a*. *rho* (*StateVar* (*Primed w*)) = *1* **using** *ext* **by** *blast*
have *delP*: ∀ *w* ∈ *del a* − *add a*. *rho* (*StateVar* (*Primed w*)) = *0* **using** *ext* **by** *blast*
have *frameO*: ∀ *w* ∈ *vars* Π*e* − *evars a*. *rho* (*ReifEq* (*Original w*)) = *1* **using** *ext* **by** *blast*
have *mD*: *models* (*encode-delta-cost* (*cost a*) (*bits-needed B*)) *rho*
by (*rule trans-delta*[*OF mT a-in*])
have *c'-eq*: c' = *c* + *cost a*
using *encode-delta-cost-forces*[*OF rho01 mD*] *delta1*
by (*simp add*: *c-def c'-def*)
— The encoded abstract states of both sides, and the abstract transition between them (paper Lemmas 21, 25, 26).
define *t* **where** *t* = *abs-state rho-o*
define t' **where** t' = *abs-state rho-p*
have *t-sub*: *t* ⊆ *P* **unfolding** *t-def* **by** (*rule abs-state-sub*)
have *pre-t*: *pre a* ∩ *P* ⊆ *t*
proof
fix *v* **assume** *vp*: *v* ∈ *pre a* ∩ *P*
have *rho* (*StateVar* (*Original v*)) = *1* **using** *preO vp* **by** *blast*
then have *rho-o* (*StateVar v*) = *1* **by** (*simp add*: *rho-o-def*)
then show *v* ∈ *t* **using** *vp* **unfolding** *t-def abs-state-def* **by** *auto*
qed
have *step*: t' = (*t* − *del a*) ∪ (*add a* ∩ *P*)
proof (*rule set-eqI*)
fix *v*

```
    show v ∈ t' ⟷ v ∈ (t − del a) ∪ (add a ∩ P)
    proof (cases v ∈ P)
      case False
      then show ?thesis
        unfolding t-def t'-def abs-state-def by auto
    next
      case True
      have rp: rho-p (StateVar v) = rho (StateVar (Primed v))
        by (simp add: rho-p-def)
      consider (AddC) v ∈ add a | (DelC) v ∈ del a v ∉ add a
        | (FrameC) v ∉ evars a
        by (auto simp: evars-def)
      then show ?thesis
      proof cases
        case AddC
        have v ∉ del a using AddC bspec[OF as-disjoint a-in] by auto
        then have rho (StateVar (Primed v)) = 1 using addP AddC by blast
        then have v ∈ t' using True rp unfolding t'-def abs-state-def by simp
        then show ?thesis using AddC True by simp
      next
        case DelC
        have rho (StateVar (Primed v)) = 0 using delP DelC by blast
        then have v ∉ t' using rp unfolding t'-def abs-state-def by simp
        moreover have v ∉ (t − del a) ∪ (add a ∩ P) using DelC by auto
        ultimately show ?thesis by simp
      next
        case FrameC
        have w-in: v ∈ vars Πe − evars a using FrameC True P-sub by auto
        then have eq1: rho (ReifEq (Original v)) = 1 using frameO by blast
        have meq: models (encode-eq-var v) rho
          by (rule trans-eq-var[OF mT]) (use w-in in blast)
        have eq2: rho (StateVar (Original v)) = rho (StateVar (Primed v))
          using encode-eq-var-forces[OF rho01 meq] eq1 by simp
        have mem-eq: v ∈ t' ⟷ v ∈ t
          using True eq2 unfolding t-def t'-def abs-state-def rho-o-def rho-p-def
          by simp
        have v ∈ (t − del a) ∪ (add a ∩ P) ⟷ v ∈ t
          using FrameC by (auto simp: evars-def)
        then show ?thesis using mem-eq by simp
      qed
    qed
  qed
  — Common final step: with a true gate of the primed copy, the primed output
fires.
  have to-out: rho-p pout-name = 1 ⟹
      eval-lit (map-literal primed-pvar-map (hc-out epdb-cert s)) rho = 1
    unfolding epdb-cert-def
    by (simp add: eval-lit-map-literal rho-p-def eval-lit-def)
  — If the successor state has infinite distance, the primed r∞-gate fires (paper
```

Lemmas 23, 25–27).
have *inf-succ*: ¬ *fin* $t' \Longrightarrow$ *rho-p pout-name* = *1*
proof −
assume ¬ *fin* t'
then have ¬ *fin* (*abs-state rho-p*) **unfolding** t'*-def* **by** *simp*
then have *rho-p inf-name* = *1* **by** (*rule not-fin-inf*[*OF rho-p01 mP'*])
then show *rho-p pout-name* = *1* **using** *poutg-forces*[*OF rho-p01 mP'*] **by** *blast*
qed
— Disjunction over the unprimed output gate.
have *outO'*: *rho-o pout-name* = *1*
using *outO* **unfolding** *epdb-cert-def*
by (*simp add*: *eval-lit-map-literal rho-o-def eval-lit-def*)
from *poutg-forces*[*OF rho-o01 mO'*] *outO'*
consider (*Inf*) *rho-o inf-name* = *1*
| (*Thr*) *i* **where** *i* < *n-s rho-o* (*thr-name i*) = *1*
by *blast*
then have *rho-p pout-name* = *1*
proof *cases*
case *Inf*
— Paper Lemmas 25–28: the unprimed state has infinite distance, hence so does the successor (finiteness would propagate backwards).
have *nf-t*: ¬ *fin t*
proof
assume *fin t*
then obtain *i* **where** *i-lt*: *i* < *n-s* **and** *i-s*: *sa-i i* = *t*
using *fin-mem-nth*[*OF t-sub*] **by** *auto*
have *rho-o* (*abs-name i*) = *1*
using *absg-forces-eq*[*OF rho-o01 mO' i-lt*] *i-s* **unfolding** *t-def* **by** *simp*
moreover have *rho-o* (*abs-name i*) = *0*
using *infg-forces*[*OF rho-o01 mO'*] *Inf i-lt* **by** *blast*
ultimately show *False* **by** *simp*
qed
have ¬ *fin* t'
proof
assume *fin* t'
then have *fin* ((*t* − *del a*) ∪ (*add a* ∩ *P*)) **using** *step* **by** *simp*
then have *fin t* **using** *d-triangle*[*OF t-sub a-in pre-t*] **by** *blast*
then show *False* **using** *nf-t* **by** *simp*
qed
then show *?thesis* **by** (*rule inf-succ*)
next
case *Thr*
— Paper Lemmas 21–24: the unprimed state is a finite-distance abstract state with a true threshold gate.
have *abs1*: *rho-o* (*abs-name i*) = *1* **and** *kk1*: *rho-o* (*kk-name i*) = *1*
using *thrg-forces*[*OF rho-o01 mO' Thr(1)*] *Thr(2)* **by** *blast+*
have *i-t*: *sa-i i* = *t*
using *absg-forces-eq*[*OF rho-o01 mO' Thr(1)*] *abs1* **unfolding** *t-def* **by** *simp*
have *c-ge*: *c* ≥ *clip B* (*int B* − *int* (*d t*))

```
    using kgg-forces[OF rho-o01 mO' Thr(1)] kk1 c-o i-t by simp
  show ?thesis
  proof (cases fin t')
    case False
    then show ?thesis by (rule inf-succ)
  next
    case True
    — Paper Lemma 22: finite successor, threshold gate of the primed copy.
    have t'-sub: t' ⊆ P unfolding t'-def by (rule abs-state-sub)
    obtain i' where i'-lt: i' < n-s and i'-t: sa-i i' = t'
      using fin-mem-nth[OF t'-sub True] by auto
    have abs1': rho-p (abs-name i') = 1
      using absg-forces-eq[OF rho-p01 mP' i'-lt] i'-t unfolding t'-def by simp
    have tri: d t ≤ d t' + cost a
      using d-triangle[OF t-sub a-in pre-t] True step by auto
    have c'-ge: c' ≥ clip B (int B − int (d t'))
    proof −
      have int B − int (d t') ≤ (int B − int (d t)) + int (cost a)
        using tri by linarith
      then have clip B (int B − int (d t'))
          ≤ clip B ((int B − int (d t)) + int (cost a))
        by (rule clip-mono)
      also have ... ≤ clip B (int B − int (d t)) + cost a
        by (rule clip-add-le)
      also have ... ≤ c + cost a using c-ge by simp
      also have ... = c' using c'-eq by simp
      finally show ?thesis .
    qed
    have kk1': rho-p (kk-name i') = 1
      using kgg-forces[OF rho-p01 mP' i'-lt] c'-ge c-p i'-t by simp
    have rho-p (thr-name i') = 1
      using thrg-forces[OF rho-p01 mP' i'-lt] abs1' kk1' by blast
    then show ?thesis using poutg-forces[OF rho-p01 mP'] i'-lt by blast
  qed
qed
then show eval-lit (map-literal primed-pvar-map (hc-out epdb-cert s)) rho = 1
  by (rule to-out)
qed
```

The efficient PDB certificate is a valid heuristic certificate in the sense of Definition 4, for any set of evaluated states.

```
theorem epdb-hc-valid: hc-valid Πe B as epdb-cert S
  unfolding hc-valid-def
  using epdb-state-lemma epdb-goal-lemma epdb-ind-lemma by blast
```

9.6 Structural conditions for use in the A* certificate

```
lemma epdb-gates-nth-abs: i < n-s ⟹ epdb-gates ! i = absg i
  by (simp add: epdb-gates-def nth-append)
```

lemma *epdb-gates-nth-kgg*: $i < n\text{-}s \Longrightarrow epdb\text{-}gates\ !\ (n\text{-}s + i) = kgg\ i$
by (*simp add*: *epdb-gates-def nth-append*)

lemma *epdb-gates-nth-thrg*: $i < n\text{-}s \Longrightarrow epdb\text{-}gates\ !\ (2 * n\text{-}s + i) = thrg\ i$
by (*simp add*: *epdb-gates-def nth-append*)

lemma *epdb-gates-nth-inf*: $epdb\text{-}gates\ !\ (3 * n\text{-}s) = infg$
by (*simp add*: *epdb-gates-def nth-append*)

lemma *epdb-gates-nth-out*: $epdb\text{-}gates\ !\ (3 * n\text{-}s + 1) = poutg$
by (*simp add*: *epdb-gates-def nth-append*)

lemma *epdb-gate-name-nth*:
assumes *p-lt*: $p < length\ epdb\text{-}gates$
shows $fst\ (epdb\text{-}gates\ !\ p) = Pos\ (ReifCert\ (nm\ p))$
proof −
consider $(A)\ p < n\text{-}s \mid (K)\ n\text{-}s \leq p\ p < 2 * n\text{-}s$
$\mid (T)\ 2 * n\text{-}s \leq p\ p < 3 * n\text{-}s \mid (INF)\ p = 3 * n\text{-}s \mid (OUT)\ p = 3 * n\text{-}s + 1$
using *p-lt length-epdb-gates* **by** *linarith*
then show *?thesis*
proof *cases*
case *A*
then show *?thesis*
using *epdb-gates-nth-abs*[*OF A*] **by** (*simp add*: *absg-def abs-name-def*)
next
case *K*
then obtain *q* **where** *p-eq*: $p = n\text{-}s + q$ **using** *le-iff-add* **by** *auto*
have *q-lt*: $q < n\text{-}s$ **using** *K p-eq* **by** *simp*
show *?thesis*
unfolding *p-eq* **using** *epdb-gates-nth-kgg*[*OF q-lt*]
by (*simp add*: *kgg-def k-gate-def kk-name-def*)
next
case *T*
then obtain *q* **where** *p-eq*: $p = 2 * n\text{-}s + q$ **using** *le-iff-add* **by** *auto*
have *q-lt*: $q < n\text{-}s$ **using** *T p-eq* **by** *simp*
show *?thesis*
unfolding *p-eq* **using** *epdb-gates-nth-thrg*[*OF q-lt*]
by (*simp add*: *thrg-def thr-name-def*)
next
case *INF*
then show *?thesis*
using *epdb-gates-nth-inf* **by** (*simp add*: *infg-def inf-name-def*)
next
case *OUT*
then show *?thesis*
using *epdb-gates-nth-out* **by** (*simp add*: *poutg-def pout-name-def*)
qed
qed

```
lemma epdb-names:
  ∀(r, cs, A) ∈ set (hc-gates epdb-cert). ∃j. r = Pos (ReifCert (nm j))
proof −
  have ⋀g. g ∈ set epdb-gates ⟹ ∃j. fst g = Pos (ReifCert (nm j))
  proof −
    fix g assume g ∈ set epdb-gates
    then obtain p where p-lt: p < length epdb-gates and g-eq: g = epdb-gates ! p
      by (auto simp: in-set-conv-nth)
    show ∃j. fst g = Pos (ReifCert (nm j))
      using epdb-gate-name-nth[OF p-lt] g-eq by auto
  qed
  then show ?thesis by (fastforce simp: epdb-cert-def)
qed

lemma epdb-distinct:
  distinct (map (λ(r, cs, A). pvar-of-lit r) (hc-gates epdb-cert))
proof −
  note len = length-epdb-gates
  have map-eq: map (λ(r, cs, A). pvar-of-lit r) epdb-gates
     = map (λp. ReifCert (nm p)) [0..<3 * n-s + 2]
  proof (rule nth-equalityI)
    show length (map (λ(r, cs, A). pvar-of-lit r) epdb-gates)
       = length (map (λp. ReifCert (nm p)) [0..<3 * n-s + 2])
      by (simp add: len)
    fix p assume p < length (map (λ(r, cs, A). pvar-of-lit r) epdb-gates)
    then have p-lt: p < length epdb-gates by simp
    have (λ(r, cs, A). pvar-of-lit r) (epdb-gates ! p) = pvar-of-lit (fst (epdb-gates !
p))
      by (simp add: split-beta)
    also have ... = ReifCert (nm p)
      using epdb-gate-name-nth[OF p-lt] by (simp add: pvar-of-lit-def)
    finally show map (λ(r, cs, A). pvar-of-lit r) epdb-gates ! p
       = map (λp. ReifCert (nm p)) [0..<3 * n-s + 2] ! p
      using p-lt len by (auto simp: nth-append less-Suc-eq)
  qed
  have distinct (map (λp. ReifCert (nm p)) [0..<3 * n-s + 2])
    using nm-inj by (auto simp: distinct-map intro!: inj-onI dest: injD)
  then show ?thesis using map-eq by (simp add: epdb-cert-def)
qed

lemma epdb-out-in: hc-out epdb-cert s ∈ fst ‘ set (hc-gates epdb-cert)
proof −
  have fst poutg = Pos pout-name by (simp add: poutg-def)
  then show ?thesis using poutg-in-gates by (force simp: epdb-cert-def)
qed

lemma nth-in-take-epdb: j < i ⟹ i ≤ length xs ⟹ xs ! j ∈ set (take i xs)
  by (metis length-take min.absorb2 nth-mem nth-take)
```

```
lemma epdb-wf:
  ∀ i < length (hc-gates epdb-cert). case hc-gates epdb-cert ! i of (r, cs, A) ⇒
      (∀ x ∈ pvar-of-lit ‘ snd ‘ set cs.
        (∃ v. x = StateVar v) ∨ (∃ j. x = CostBit j)
        ∨ x ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates epdb-cert)))
proof (intro allI impI)
  fix i assume i-lt: i < length (hc-gates epdb-cert)
  have hcg: hc-gates epdb-cert = epdb-gates by (simp add: epdb-cert-def)
  have i-lt': i < length epdb-gates using i-lt by (simp add: hcg)
  have i-le: i ≤ length epdb-gates using i-lt' by simp
  have abs-take: ⋀q. q < n-s ⟹ q < i ⟹
      abs-name q ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates epdb-cert))
  proof −
    fix q assume q-lt: q < n-s and q-i: q < i
    have epdb-gates ! q ∈ set (take i epdb-gates)
      using nth-in-take-epdb q-i i-le by blast
    moreover have fst (epdb-gates ! q) = Pos (abs-name q)
      using epdb-gates-nth-abs[OF q-lt] by (simp add: absg-def)
   ultimately show abs-name q ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates epdb-cert))
      using hcg by (force simp: pvar-of-lit-def)
  qed
  consider (A) i < n-s | (K) n-s ≤ i i < 2 * n-s
    | (T) 2 * n-s ≤ i i < 3 * n-s | (INF) i = 3 * n-s | (OUT) i = 3 * n-s + 1
    using i-lt' length-epdb-gates by linarith
  then show case hc-gates epdb-cert ! i of (r, cs, A) ⇒
      (∀ x ∈ pvar-of-lit ‘ snd ‘ set cs.
        (∃ v. x = StateVar v) ∨ (∃ j. x = CostBit j)
        ∨ x ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates epdb-cert)))
  proof cases
    case A
    have g: hc-gates epdb-cert ! i = absg i
      using epdb-gates-nth-abs[OF A] hcg by simp
    have ∀ x ∈ pvar-of-lit ‘ snd ‘ set (map (λl. (1, l)) (abs-lits i)).
        (∃ v. x = StateVar v)
      by (auto simp: abs-lits-def pvar-of-lit-def split: if-splits)
    then show ?thesis unfolding g absg-def by auto
  next
    case K
    then obtain q where i-eq: i = n-s + q using le-iff-add by auto
    have q-lt: q < n-s using K i-eq by simp
    have g: hc-gates epdb-cert ! i = kgg q
      using epdb-gates-nth-kgg[OF q-lt] hcg i-eq by simp
    have ∀ x ∈ pvar-of-lit ‘ snd ‘ set (k-gate-body B). (∃ j. x = CostBit j)
      by (auto simp: k-gate-body-def pvar-of-lit-def)
    then show ?thesis unfolding g kgg-def k-gate-def by auto
  next
    case T
    then obtain q where i-eq: i = 2 * n-s + q using le-iff-add by auto
```

```
  have q-lt: q < n-s using T i-eq by simp
  have g: hc-gates epdb-cert ! i = thrg q
    using epdb-gates-nth-thrg[OF q-lt] hcg i-eq by simp
  have abs-in: abs-name q ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates epdb-cert))
    using abs-take q-lt i-eq by simp
  have kk-in: kk-name q ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates epdb-cert))
  proof −
    have n-s + q < i using q-lt i-eq by simp
    then have epdb-gates ! (n-s + q) ∈ set (take i epdb-gates)
      using nth-in-take-epdb i-le by blast
    moreover have fst (epdb-gates ! (n-s + q)) = Pos (kk-name q)
      using epdb-gates-nth-kgg[OF q-lt] by (simp add: kgg-def k-gate-def)
    ultimately show ?thesis using hcg by (force simp: pvar-of-lit-def)
  qed
  have ∀ x ∈ pvar-of-lit ‘ snd ‘ set (map (λl. (1, l)) (thr-lits q)).
      x ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates epdb-cert))
    using abs-in kk-in by (auto simp: thr-lits-def pvar-of-lit-def)
  then show ?thesis unfolding g thrg-def by auto
next
  case INF
  have g: hc-gates epdb-cert ! i = infg
    using epdb-gates-nth-inf hcg INF by simp
  have ∀ x ∈ pvar-of-lit ‘ snd ‘ set (map (λl. (1, l)) inf-lits).
      x ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates epdb-cert))
  proof
    fix x assume x ∈ pvar-of-lit ‘ snd ‘ set (map (λl. (1, l)) inf-lits)
    then obtain q where q-lt: q < n-s and x-eq: x = abs-name q
      by (auto simp: inf-lits-def pvar-of-lit-def)
    have q < i using q-lt INF by simp
    then show x ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates epdb-cert))
      using abs-take q-lt x-eq by simp
  qed
  then show ?thesis unfolding g infg-def by auto
next
  case OUT
  have g: hc-gates epdb-cert ! i = poutg
    using epdb-gates-nth-out hcg OUT by simp
  have ∀ x ∈ pvar-of-lit ‘ snd ‘ set (map (λl. (1, l)) pout-lits).
      x ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates epdb-cert))
  proof
    fix x assume x ∈ pvar-of-lit ‘ snd ‘ set (map (λl. (1, l)) pout-lits)
    then consider (Base) x = inf-name
      | (Var) q where q < n-s x = thr-name q
      by (auto simp: pout-lits-def pvar-of-lit-def)
    then show x ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates epdb-cert))
    proof cases
      case Base
      have 3 ∗ n-s < i using OUT by simp
      then have epdb-gates ! (3 ∗ n-s) ∈ set (take i epdb-gates)
```

```
      using nth-in-take-epdb i-le by blast
    moreover have fst (epdb-gates ! (3 * n-s)) = Pos inf-name
      using epdb-gates-nth-inf by (simp add: infg-def)
    ultimately show ?thesis using hcg Base by (force simp: pvar-of-lit-def)
   next
    case Var
    have 2 * n-s + q < i using Var(1) OUT by simp
    then have epdb-gates ! (2 * n-s + q) ∈ set (take i epdb-gates)
      using nth-in-take-epdb i-le by blast
    moreover have fst (epdb-gates ! (2 * n-s + q)) = Pos (thr-name q)
      using epdb-gates-nth-thrg[OF Var(1)] by (simp add: thrg-def)
    ultimately show ?thesis using hcg Var(2) by (force simp: pvar-of-lit-def)
   qed
  qed
  then show ?thesis unfolding g poutg-def by auto
 qed
qed

end

end
```

10 Locale Instances

```
theory Locale-Instances
 imports Hmax-Certificates Efficient-PDB
begin
```

Consistency witnesses for all locales of the development. A locale whose assumptions are contradictory makes every theorem proved inside it vacuous; the interpretations below rule this out by exhibiting one concrete model for each of *pdb-heuristic*, *hmax-heuristic*, *efficient-pdb* and *astar-run*.

The witness is the smallest non-degenerate planning task: one variable *0*, initially false, required by the goal, and one action that adds it at cost 1. The optimal plan is the single-action plan of cost 1, and we use the bound $B = 1$. All interesting locale assumptions (the PDB distance conditions *d-goal*/*d-triangle*, the hmax table conditions, the A* expansion-closure condition) are discharged non-vacuously.

The A* interpretation is additionally instantiated with the *interpreted* PDB certificate at the embedded variable type *nat* + *nat* (gate names *Inr* (*2***j*+*1*)), i.e. the locales compose exactly as the paper's PDB-into-A* case study intends. As a final sanity check the composition yields the concrete theorem *optimal-plan demo-task* [*demo-act*].

10.1 The demo task

```
definition demo-act :: nat action where
```

demo-act = (| *pre* = {}, *add* = {*0*}, *del* = {}, *cost* = *1* |)

definition *demo-task* :: *nat strips-task* **where**
demo-task = (| *vars* = {*0*}, *acts* = {*demo-act*}, *init* = {}, *goal* = {*0*} |)

lemma *demo-pow*: *set* [{}, {*0*::*nat*}] = *Pow* {*0*}
by (*auto dest*: *subset-singletonD*)

10.2 Interpretation of *pdb-heuristic*

interpretation *demo-pdb*: *pdb-heuristic demo-task 1* {*0*::*nat*}
λ*sa*. *if 0* ∈ *sa then 0 else 1* [{}, {*0*}] [*demo-act*] *id*
proof *unfold-locales*
show *finite* (*vars demo-task*) **by** (*simp add*: *demo-task-def*)
show {*0*::*nat*} ⊆ *vars demo-task* **by** (*simp add*: *demo-task-def*)
show *set* [{}, {*0*::*nat*}] = *Pow* {*0*} **by** (*rule demo-pow*)
show *distinct* [{}, {*0*::*nat*}] **by** *simp*
show ∀ *a* ∈ *set* [*demo-act*].
pre a ⊆ *vars demo-task* ∧ *add a* ⊆ *vars demo-task* ∧ *del a* ⊆ *vars demo-task*
by (*simp add*: *demo-act-def demo-task-def*)
show ∀ *a* ∈ *set* [*demo-act*]. *add a* ∩ *del a* = {}
by (*simp add*: *demo-act-def*)
show ⋀*sa*. *sa* ⊆ {*0*::*nat*} ⟹ *goal demo-task* ∩ {*0*} ⊆ *sa* ⟹
(*if 0* ∈ *sa then 0 else 1*) = (*0*::*nat*)
by (*auto simp*: *demo-task-def*)
show ⋀*sa a*. *sa* ⊆ {*0*::*nat*} ⟹ *a* ∈ *set* [*demo-act*] ⟹ *pre a* ∩ {*0*} ⊆ *sa* ⟹
(*if 0* ∈ *sa then 0 else 1*)
≤ (*if 0* ∈ (*sa* − *del a*) ∪ (*add a* ∩ {*0*}) *then 0 else 1*) + *cost a*
by (*auto simp*: *demo-act-def*)
show *inj* (*id* :: *nat* ⇒ *nat*) **by** *simp*
qed

The interpreted facts are available, e.g. the validity of the PDB certificate for the demo table:

lemma *hc-valid demo-task 1* [*demo-act*] *demo-pdb.pdb-cert S*
by (*rule demo-pdb.pdb-hc-valid*)

10.3 Interpretation of *hmax-heuristic*

The evaluated state is the initial state {}; its hmax value is 1 (the single goal variable costs 1 to achieve), and the clipped max value of every variable is 1.

interpretation *demo-hmax*: *hmax-heuristic demo-task 1* {} :: *nat state 1*
λ*v*. *1* [*0*] [*demo-act*] *id*
proof *unfold-locales*
show *finite* (*vars demo-task*) **by** (*simp add*: *demo-task-def*)
show *goal demo-task* ⊆ *vars demo-task* **by** (*simp add*: *demo-task-def*)
show {} ⊆ *vars demo-task* **by** *simp*
show *set* [*0*] = *vars demo-task* **by** (*simp add*: *demo-task-def*)

show *distinct* [*0*::*nat*] **by** *simp*
show ∀ *a* ∈ *set* [*demo-act*].
pre a ⊆ *vars demo-task* ∧ *add a* ⊆ *vars demo-task* ∧ *del a* ⊆ *vars demo-task*
by (*simp add*: *demo-act-def demo-task-def*)
show ∀ *a* ∈ *set* [*demo-act*]. *add a* ∩ *del a* = {}
by (*simp add*: *demo-act-def*)
show ⋀*v*. *v* ∈ ({} :: *nat state*) ⟹ (*1*::*nat*) = *0* **by** *simp*
show *goal demo-task* ≠ {} ⟹ ∃ *v* ∈ *goal demo-task*. (*1*::*nat*) = *1*
by (*simp add*: *demo-task-def*)
show *goal demo-task* = {} ⟹ (*1*::*nat*) = *0* **by** (*simp add*: *demo-task-def*)
show ⋀*a v*. *a* ∈ *set* [*demo-act*] ⟹ *v* ∈ *add a* ⟹ *pre a* ≠ {} ⟹
∃ *p* ∈ *pre a*. (*1*::*nat*) ≤ *1* + *cost a*
by (*simp add*: *demo-act-def*)
show ⋀*a v*. *a* ∈ *set* [*demo-act*] ⟹ *v* ∈ *add a* ⟹ *pre a* = {} ⟹
(*1*::*nat*) ≤ *cost a*
by (*simp add*: *demo-act-def*)
show *inj* (*id* :: *nat* ⇒ *nat*) **by** *simp*
qed

lemma *hc-valid demo-task 1* [*demo-act*] *demo-hmax.hmax-cert* {{}}
by (*rule demo-hmax.hmax-hc-valid*)

10.4 Interpretation of *efficient-pdb*

In the demo task every abstract state reaches the abstract goal, so the finiteness predicate is constantly true and the table coincides with the plain PDB table.

interpretation *demo-epdb*: *efficient-pdb demo-task 1* {*0*::*nat*}
λ*sa*. *if 0* ∈ *sa then 0 else 1* λ*sa*. *True* [{}, {*0*}] [*demo-act*] *id*
proof *unfold-locales*
show *finite* (*vars demo-task*) **by** (*simp add*: *demo-task-def*)
show {*0*::*nat*} ⊆ *vars demo-task* **by** (*simp add*: *demo-task-def*)
show *set* [{}, {*0*::*nat*}] = {*sa*. *sa* ⊆ {*0*} ∧ *True*}
using *demo-pow* **by** *auto*
show *distinct* [{}, {*0*::*nat*}] **by** *simp*
show ∀ *a* ∈ *set* [*demo-act*].
pre a ⊆ *vars demo-task* ∧ *add a* ⊆ *vars demo-task* ∧ *del a* ⊆ *vars demo-task*
by (*simp add*: *demo-act-def demo-task-def*)
show ∀ *a* ∈ *set* [*demo-act*]. *add a* ∩ *del a* = {}
by (*simp add*: *demo-act-def*)
show ⋀*sa*. *sa* ⊆ {*0*::*nat*} ⟹ *goal demo-task* ∩ {*0*} ⊆ *sa* ⟹
True ∧ (*if 0* ∈ *sa then 0 else 1*) = (*0*::*nat*)
by (*auto simp*: *demo-task-def*)
show ⋀*sa a*. *sa* ⊆ {*0*::*nat*} ⟹ *a* ∈ *set* [*demo-act*] ⟹ *pre a* ∩ {*0*} ⊆ *sa* ⟹
True ⟹ *True* ∧ (*if 0* ∈ *sa then 0 else 1*)
≤ (*if 0* ∈ (*sa* − *del a*) ∪ (*add a* ∩ {*0*}) *then 0 else 1*) + *cost a*
by (*auto simp*: *demo-act-def*)
show *inj* (*id* :: *nat* ⇒ *nat*) **by** *simp*
qed

lemma *hc-valid demo-task 1* [*demo-act*] *demo-epdb.epdb-cert S*
by (*rule demo-epdb.epdb-hc-valid*)

10.5 Interpretation of *pdb-heuristic* at the embedded type

For the A* interpretation the heuristic certificate must live at the extended variable type *nat* + *nat* with the odd gate names *Inr* (*2∗j+1*) that *astar-run* reserves for the heuristic.

lemma *demo-taskE-simps*:
vars (*embed-task demo-task* :: (*nat* + *nat*) *strips-task*) = {*Inl 0*}
goal (*embed-task demo-task* :: (*nat* + *nat*) *strips-task*) = {*Inl 0*}
init (*embed-task demo-task* :: (*nat* + *nat*) *strips-task*) = {}
acts (*embed-task demo-task* :: (*nat* + *nat*) *strips-task*) = {*embed-act demo-act*}
by (*simp-all add*: *embed-task-def demo-task-def*)

lemma *demo-actE-simps*:
pre (*embed-act demo-act* :: (*nat* + *nat*) *action*) = {}
add (*embed-act demo-act* :: (*nat* + *nat*) *action*) = {*Inl 0*}
del (*embed-act demo-act* :: (*nat* + *nat*) *action*) = {}
cost (*embed-act demo-act* :: (*nat* + *nat*) *action*) = *1*
by (*simp-all add*: *embed-act-def demo-act-def*)

lemma *demo-powE*: *set* [{}, {*Inl 0*} :: (*nat* + *nat*) *state*] = *Pow* {*Inl 0*}
by (*auto dest*: *subset-singletonD*)

interpretation *demoE-pdb*: *pdb-heuristic*
embed-task demo-task :: (*nat* + *nat*) *strips-task 1* {*Inl 0*}
λsa. if Inl 0 ∈ *sa then 0 else 1* [{}, {*Inl 0*}] [*embed-act demo-act*]
λj. Inr (*2* ∗ *j* + *1*)
proof *unfold-locales*
show *finite* (*vars* (*embed-task demo-task* :: (*nat* + *nat*) *strips-task*))
by (*simp add*: *demo-taskE-simps*)
show {*Inl 0*} ⊆ *vars* (*embed-task demo-task* :: (*nat* + *nat*) *strips-task*)
by (*simp add*: *demo-taskE-simps*)
show *set* [{}, {*Inl 0*} :: (*nat* + *nat*) *state*] = *Pow* {*Inl 0*}
by (*rule demo-powE*)
show *distinct* [{}, {*Inl 0*} :: (*nat* + *nat*) *state*] **by** *simp*
show ∀ *a* ∈ *set* [*embed-act demo-act* :: (*nat* + *nat*) *action*].
pre a ⊆ *vars* (*embed-task demo-task*)
∧ *add a* ⊆ *vars* (*embed-task demo-task*)
∧ *del a* ⊆ *vars* (*embed-task demo-task*)
by (*simp add*: *demo-actE-simps demo-taskE-simps*)
show ∀ *a* ∈ *set* [*embed-act demo-act* :: (*nat* + *nat*) *action*]. *add a* ∩ *del a* = {}
by (*simp add*: *demo-actE-simps*)
show ⋀*sa. sa* ⊆ {*Inl 0*} ⟹
goal (*embed-task demo-task* :: (*nat* + *nat*) *strips-task*) ∩ {*Inl 0*} ⊆ *sa* ⟹
(*if Inl 0* ∈ *sa then 0 else 1*) = (*0*::*nat*)
by (*auto simp*: *demo-taskE-simps*)

show $\bigwedge sa\ a.\ sa \subseteq \{Inl\ 0\} \Longrightarrow a \in set\ [embed\text{-}act\ demo\text{-}act :: (nat + nat)\ action]$
$\Longrightarrow$
$pre\ a \cap \{Inl\ 0\} \subseteq sa \Longrightarrow$
$(if\ Inl\ 0 \in sa\ then\ 0\ else\ 1)$
$\leq (if\ Inl\ 0 \in (sa - del\ a) \cup (add\ a \cap \{Inl\ 0\})\ then\ 0\ else\ 1) + cost\ a$
by (*auto simp*: *demo-actE-simps*)
show $inj\ (\lambda j.\ Inr\ (2 * j + 1) :: nat + nat)$
by (*auto simp*: *inj-def*)
qed

10.6 Interpretation of *astar-run*

The A* snapshot after expanding the initial state: the closed list holds $(\{\}, 0)$, the open list holds the goal state $\{0\}$, and the heuristic certificate is the interpreted embedded PDB certificate.

interpretation *demo-astar*: *astar-run demo-task 1* $[(\{\}, 0)]$ $[\{0::nat\}]$
demoE-pdb.pdb-cert $[embed\text{-}act\ demo\text{-}act]$
proof *unfold-locales*
show *finite* (*vars demo-task*) **by** (*simp add*: *demo-task-def*)
show *init demo-task* $\subseteq$ *vars demo-task* **by** (*simp add*: *demo-task-def*)
show *goal demo-task* $\subseteq$ *vars demo-task* **by** (*simp add*: *demo-task-def*)
show *finite* (*acts demo-task*) **by** (*simp add*: *demo-task-def*)
show $\forall a \in acts\ demo\text{-}task.\ add\ a \cap del\ a = \{\}$
by (*simp add*: *demo-task-def demo-act-def*)
show $\forall a \in acts\ demo\text{-}task.$
$pre\ a \subseteq vars\ demo\text{-}task \wedge add\ a \subseteq vars\ demo\text{-}task \wedge del\ a \subseteq vars\ demo\text{-}task$
by (*simp add*: *demo-task-def demo-act-def*)
show $set\ [embed\text{-}act\ demo\text{-}act] = acts\ (embed\text{-}task\ demo\text{-}task)$
by (*simp add*: *embed-task-def demo-task-def*)
show $(1::nat) \geq 1$ **by** *simp*
show $(init\ demo\text{-}task,\ 0) \in set\ [(\{\},\ 0)]$ **by** (*simp add*: *demo-task-def*)
show $\forall (s,\ g) \in set\ [(\{\},\ 0)].\ s \subseteq vars\ demo\text{-}task$ **by** (*simp add*: *demo-task-def*)
show $\forall s \in set\ [\{0::nat\}].\ s \subseteq vars\ demo\text{-}task$ **by** (*simp add*: *demo-task-def*)
show $\forall (s,\ g) \in set\ [(\{\},\ 0::nat)].\ is\text{-}goal\text{-}state\ demo\text{-}task\ s \longrightarrow g \geq 1$
by (*simp add*: *is-goal-state-def demo-task-def*)
show $\forall (s,\ g) \in set\ [(\{\},\ 0::nat)].\ \forall a \in acts\ demo\text{-}task.\ applicable\ a\ s \longrightarrow$
$(is\text{-}goal\text{-}state\ demo\text{-}task\ s \wedge g \geq 1)$
$\vee\ (\exists g'.\ (successor\ a\ s,\ g') \in set\ [(\{\},\ 0)] \wedge g' \leq g + cost\ a)$
$\vee\ (successor\ a\ s \in set\ [\{0::nat\}] \wedge$
$int\ (g + cost\ a) \geq int\ 1 - int\ (hc\text{-}h\ demoE\text{-}pdb.pdb\text{-}cert\ (Inl\ `\ successor\ a\ s)))$
proof −
have *succ*: $successor\ demo\text{-}act\ \{\} = \{0\}$
by (*simp add*: *successor-def demo-act-def*)
have *h0*: $hc\text{-}h\ demoE\text{-}pdb.pdb\text{-}cert\ \{Inl\ (0::nat)\} = 0$
unfolding *demoE-pdb.pdb-cert-def* **by** *simp*
have *cost-a*: $cost\ demo\text{-}act = 1$
by (*simp add*: *demo-act-def*)
have *third*: $successor\ demo\text{-}act\ \{\} \in set\ [\{0::nat\}] \wedge$

```
      int (0 + cost demo-act)
        ≥ int 1 − int (hc-h demoE-pdb.pdb-cert (Inl ‘ successor demo-act {}))
    by (simp add: succ h0 cost-a)
  have inner: ∀ a ∈ acts demo-task. applicable a {} ⟶
      (is-goal-state demo-task {} ∧ (0::nat) ≥ 1)
    ∨ (∃ g'. (successor a {}, g') ∈ set [({}, 0)] ∧ g' ≤ 0 + cost a)
    ∨ (successor a {} ∈ set [{0::nat}] ∧
       int (0 + cost a) ≥ int 1 − int (hc-h demoE-pdb.pdb-cert (Inl ‘ successor a
{})))
  proof
    fix a assume a ∈ acts demo-task
    then have a-eq: a = demo-act by (simp add: demo-task-def)
    show applicable a {} ⟶
        (is-goal-state demo-task {} ∧ (0::nat) ≥ 1)
      ∨ (∃ g'. (successor a {}, g') ∈ set [({}, 0)] ∧ g' ≤ 0 + cost a)
      ∨ (successor a {} ∈ set [{0::nat}] ∧
         int (0 + cost a) ≥ int 1 − int (hc-h demoE-pdb.pdb-cert (Inl ‘ successor
a {})))
      unfolding a-eq using third by blast
  qed
  show ?thesis using inner by simp
 qed
 show hc-valid (embed-task demo-task) 1 [embed-act demo-act] demoE-pdb.pdb-cert
     ((λs. Inl ‘ s) ‘ set [{0::nat}])
   by (rule demoE-pdb.pdb-hc-valid)
 show ∀ (r, cs, A) ∈ set (hc-gates demoE-pdb.pdb-cert).
     ∃ j. r = Pos (ReifCert (Inr (2 ∗ j + 1)))
   using demoE-pdb.pdb-names by auto
 show distinct (map (λ(r, cs, A). pvar-of-lit r) (hc-gates demoE-pdb.pdb-cert))
   by (rule demoE-pdb.pdb-distinct)
 show ∀ i < length (hc-gates demoE-pdb.pdb-cert).
     case hc-gates demoE-pdb.pdb-cert ! i of (r, cs, A) ⇒
       (∀ x ∈ pvar-of-lit ‘ snd ‘ set cs.
         (∃ v. x = StateVar v) ∨ (∃ j. x = CostBit j)
         ∨ x ∈ pvar-of-lit ‘ fst ‘ set (take i (hc-gates demoE-pdb.pdb-cert)))
   by (rule demoE-pdb.pdb-wf)
 show ∀ s ∈ set [{0::nat}].
     hc-out demoE-pdb.pdb-cert (Inl ‘ s) ∈ fst ‘ set (hc-gates demoE-pdb.pdb-cert)
   using demoE-pdb.pdb-out-in by simp
qed
```

10.7 End-to-end sanity check

The composed interpretations yield a concrete, non-vacuous consequence: the single-action plan is optimal for the demo task.

```
lemma demo-plan: is-plan-for demo-task [demo-act]
proof −
 have succ: successor demo-act {} = {0}
   by (simp add: successor-def demo-act-def)
```

```
  have p0: path (acts demo-task) (successor demo-act {}) {0} []
    unfolding succ by (rule path.path-nil)
  have appl: applicable demo-act {}
    by (simp add: applicable-def demo-act-def)
  have mem: demo-act ∈ acts demo-task
    by (simp add: demo-task-def)
  have path (acts demo-task) {} {0} [demo-act]
    by (rule path.path-cons[OF appl p0 mem])
  then show ?thesis
    unfolding is-plan-for-def is-goal-state-def
    by (intro exI[of - {0}]) (simp add: demo-task-def)
qed

lemma demo-cost: plan-cost [demo-act] = 1
  by (simp add: plan-cost-def demo-act-def)

theorem demo-optimal: optimal-plan demo-task [demo-act]
  by (rule demo-astar.astar-optimal[OF demo-plan demo-cost])
```

And the lower bound directly: every plan for the demo task costs at least 1.

```
theorem demo-lower-bound: is-plan-for demo-task π ⟹ plan-cost π ≥ 1
  by (rule demo-astar.astar-lower-bound)

end
```

References

[1] B. Bogaerts, S. Gocht, C. McCreesh, and J. Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *Journal of Artificial Intelligence Research*, 77:1539–1589, 2023.

[2] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5–33, 2001.

[3] S. Dold, M. Helmert, J. Nordström, G. Röger, and T. Schindler. Pseudo-boolean proof logging for optimal classical planning. In *Proceedings of the 35th International Conference on Automated Planning and Scheduling (ICAPS 2025)*. AAAI Press, 2025. Extended version with appendix: arXiv:2504.18443.

[4] S. Edelkamp. Planning with pattern databases. In *Proceedings of the 6th European Conference on Planning (ECP 2001)*, pages 84–90, 2001.

[5] S. Eriksson, G. Röger, and M. Helmert. Unsolvability certificates for classical planning. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS 2017)*, pages 88–97. AAAI Press, 2017.