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*
⟨*proof*⟩

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*⟩

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 ∈ CC ⟹ cpr-derives CC C*
| *lit-ax*: *cpr-derives CC ([(1, l)], 0)*
| *lin-comb*: *⟦cpr-derives CC C; cpr-derives CC D⟧*
⟹ cpr-derives CC (linear-combination cA C cB D)
| *div-rule*: *⟦cpr-derives CC C; c ≥ 1⟧ ⟹ cpr-derives CC (division c C)*
| *sat-rule*: *cpr-derives CC C ⟹ 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}) ⟹ cpr-derives CC C*

lemma *hyp-sound*: *C ∈ CC ⟹ models CC rho ⟹ satisfies C rho*
⟨*proof*⟩

lemma *pb-sum-map-mul*: *pb-sum (map (λ(a, l). (k ∗ a, l)) xs) rho = k ∗ pb-sum xs rho*
⟨*proof*⟩

lemma *pb-sum-append*: *pb-sum (xs @ ys) rho = pb-sum xs rho + pb-sum ys rho*
⟨*proof*⟩

lemma *lin-comb-sound*:
assumes *satisfies C rho satisfies D rho*
shows *satisfies (linear-combination nC C nD D) rho*
⟨*proof*⟩

lemma *pb-sum-ge-term*: *(a, l) ∈ set coeffs ⟹ pb-sum coeffs rho ≥ a ∗ eval-lit l*

rho
⟨*proof*⟩

lemma *ceil-ge*:
fixes *a c* :: *nat*
assumes *c* > *0*
shows ((*a* + *c* − *1*) *div c*) ∗ *c* ≥ *a*
⟨*proof*⟩

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*⟩

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*⟩

lemma *div-rule-sound*:
assumes *c* ≥ *1 satisfies C rho*
shows *satisfies* (*division c C*) *rho*
⟨*proof*⟩

lemma *sat-rule-sound*:
assumes *satisfies C rho*
shows *satisfies* (*saturation C*) *rho*
⟨*proof*⟩

theorem *cpr-sound*:
assumes *cpr-derives CC C* **and** *models CC rho* **and** ∀ *v*. *rho v* = *0* ∨ *rho v* = *1*
shows *satisfies C rho*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bits-needed-upper-bound*: *bits-needed B* ≤ *B* + *1*
⟨*proof*⟩

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

⟨*proof*⟩

definition *encode-cost-ge* :: *nat* ⇒ *'v pconstraint set* **where**
encode-cost-ge k ≡
reification (*Pos* (*ReifCostGe k*)) (*map* (λ*i*. (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 ⟨*proof*⟩
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 ≡ λ*x*. *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* (λ*i*. *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 (λv. (1, Pos (StateVar (Original v)))) (sorted-list-of-set
(pre a));
        add-lits   = map (λv. (1, Pos (StateVar (Primed v)))) (sorted-list-of-set (add
a));
        del-lits   = map (λv. (1, Neg (StateVar (Primed v)))) (sorted-list-of-set (del
a));
        frame-lits = map (λv. (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 (λr. (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 ≡ λx. 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*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *pb-sum-neg-state-lits-aux*:
assumes *distinct xs*
shows *pb-sum* (*map* (λ*v*. (*1*, *Neg* (*StateVar v*))) *xs*) (*state-rho* Π *s c*) = *card* (*set xs* − *s*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *state-rho-range*:
state-rho Π *s c x* ≤ *1*
⟨*proof*⟩

lemma *state-rho-01*:
state-rho Π *s c x* = *0* ∨ *state-rho* Π *s c x* = *1*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *sum-list-exp*: *sum-list* (*map* ((^) *2*) [*0*..<*B*]) = (*2* :: *nat*) ^ *B* − *1*
⟨*proof*⟩

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*⟩

lemma *cost-ge-encoding-below*:
assumes *c* < *k*
shows *models* (*encode-cost-ge k*) (*state-rho* Π *s c*)
⟨*proof*⟩

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*⟩

lemma *cost-circuit-sound*:
assumes *models-circuit* (*cost-circuit k*) (*state-rho* Π *s c*)

shows $c \geq k$
⟨*proof*⟩

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 ≡ ⋃ (*r*, *cs*, *A*) ∈ *set* (*fst C*). *reification r cs A*

definition *distinct-reif-vars* :: $'v$ *pb-circuit* ⇒ *bool* **where**
  distinct-reif-vars C ≡
    let pairs = *fst C*
    in ∀ *i* < *length pairs*. ∀ *j* < *length pairs*. *i* ≠ *j* ⟶
        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 ≡
    (*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* ∧

$(\forall\, u.\ v \neq \mathit{ReifGeq}\ u) \wedge (\forall\, u.\ v \neq \mathit{ReifLeq}\ u) \wedge (\forall\, u.\ v \neq \mathit{ReifEq}\ u) \wedge$
$(\forall\, i.\ v \neq \mathit{ReifAction}\ i) \wedge (\forall\, i.\ v \neq \mathit{PrimedCostBit}\ i)) \wedge$
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)) $\wedge$
cpr-derives
(*encode-goal* $\Pi \cup$ *circuit-constraints C-*$\varphi \cup$ *encode-cost-ge B* $\cup$
{*unit-clause* (*snd C-*φ), *unit-clause* (*Pos ReifG*)})
(*cost-ge-constraint B*) $\wedge$
cpr-derives
(*encode-transition* (*cert-actions Cert*) (*vars* Π) *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)))

lemma *map-pvar-Original-inj*:
map-pvar Original x = *map-pvar Original y* $\longleftrightarrow$ (*x* :: *'v pvar*) = *y*
⟨*proof*⟩

lemma *map-literal-eq-Pos-conv*: *map-literal f x* = *Pos y* $\longleftrightarrow$ ($\exists\, z.\ x = \mathit{Pos}\ z \wedge f\ z = y$)
⟨*proof*⟩
lemma *map-literal-eq-Neg-conv*: *map-literal f x* = *Neg y* $\longleftrightarrow$ ($\exists\, z.\ x = \mathit{Neg}\ z \wedge f\ z = y$)
⟨*proof*⟩
lemmas *map-literal-convs* = *map-literal-eq-Pos-conv map-literal-eq-Neg-conv*

lemma *not-Pos-Neg*: ($\forall\, x.\ b \neq \mathit{Pos}\ x$) $\longleftrightarrow$ ($\exists\, x.\ b = \mathit{Neg}\ x$)
⟨*proof*⟩

lemma *wf-orig-circuit*:
assumes *wf-circuit* (*C* :: *'v pb-circuit*)
shows *wf-circuit* (*orig-circuit C*)
⟨*proof*⟩

definition *in-M* :: *'v*::*linorder pb-circuit* ⇒ *'v strips-task* ⇒ *'v state* ⇒ *nat* ⇒ *bool*
where
in-M C Π *s c* ≡
$\exists$ *rho*. ($\forall$ *v*. *rho v* = *0* $\vee$ *rho v* = *1*)
$\wedge$ *models* (*circuit-constraints C*) *rho*
$\wedge$ *eval-lit* (*snd C*) *rho* = *1*
$\wedge$ ($\forall$ *v*. *rho* (*StateVar v*) = (*if v* $\in$ *s then 1 else 0*))
$\wedge$ ($\forall$ *i*. *rho* (*CostBit i*) = (*c div 2*^*i*) *mod 2*)
$\wedge$ ($\forall$ *i*. *rho* (*PrimedCostBit i*) = *0*)

lemma *state-rho-StateVar-eq*:
state-rho Π *s c* (*StateVar v*) = (*if v* ∈ *s then 1 else 0*)
⟨*proof*⟩

lemma *state-rho-CostBit-eq*:
state-rho Π *s c* (*CostBit i*) = (*c div* 2^*i*) *mod* 2
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

lemma *eval-output-extend-rho*:
assumes *pvar-of-lit* (*snd C*) ∈ *circuit-reif-pvars C*

shows *eval-lit* (*snd C*) (*extend-rho C rho-circ rho-base*) = *eval-lit* (*snd C*) *rho-circ*
⟨*proof*⟩

lemma *pb-sum-neg-cost-bits-zero*:
assumes ∀ *i* < *k*. *rho* (*CostBit i*) = *0*
shows *pb-sum* (*map* (λ*i*. (*2^i*, *Neg* (*CostBit i*))) [*0*..<*k*]) *rho* = *2^k* − *1*
⟨*proof*⟩

lemma *satisfies-neg-cost-ge-one*:
assumes ∀ *i*. *rho* (*CostBit i*) = *0*
shows *satisfies* (*neg-cost-ge-one B*) *rho*
⟨*proof*⟩

lemma *in-M-init*:
fixes Π :: *'v*::*linorder strips-task* **and** *C-φ* :: *'v pb-circuit*
assumes *fin*: *finite* (*vars* Π) **and** *init-sub*: *init* Π ⊆ *vars* Π
and *wf*: *wf-circuit C-φ*
and *out-reif*: *pvar-of-lit* (*snd C-φ*) ∈ *circuit-reif-pvars 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 *realiz*: ∀ *base*. (∀ *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 *B-pos*: *B* ≥ *1*
and *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*))
shows *in-M C-φ* Π (*init* Π) *0*
⟨*proof*⟩

lemma *wf-circuit-out-reif*:
assumes *wf-circuit* (*C* :: *'v pb-circuit*)
shows *pvar-of-lit* (*snd C*) ∈ *circuit-reif-pvars C*
⟨*proof*⟩

lemma *satisfies-cong*:
assumes ∀ *v* ∈ *constraint-pvars φ*. *rho1 v* = *rho2 v*
shows *satisfies φ rho1* = *satisfies φ rho2*
⟨*proof*⟩

lemma *pvar-of-lit-lit-neg*: *pvar-of-lit* (*lit-neg l*) = *pvar-of-lit l*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *pb-sum-cost-bits-gen*:
assumes ∀ *i* < *k*. *rho* (*CostBit i*) = (*c div 2^i*) *mod 2*
shows *pb-sum* (*map* (λ*i*. (*2^i*, *Pos* (*CostBit i*))) [*0*..<*k*]) *rho* = *c mod 2^k*
⟨*proof*⟩

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*⟩

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* ∪ *encode-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*⟩

2.12 Transition Step Soundness and CPR Theorem

definition *two-state-rho* ::

'v::linorder strips-task ⇒ 'v state ⇒ nat ⇒ 'v state ⇒ nat ⇒ nat ⇒ ('v action) list ⇒
('v var pvar ⇒ nat) **where**
two-state-rho Π s c s' c' sel-i as ≡ λx. case x of
StateVar (Original v) ⇒ if v ∈ s then 1 else 0
| *StateVar (Primed v) ⇒ if v ∈ s' then 1 else 0*
| *CostBit i ⇒ (c div 2^i) mod 2*
| *PrimedCostBit i ⇒ (c' div 2^i) mod 2*
| *ReifT ⇒ 1*
| *ReifDeltaCostLower k ⇒ if c' ≥ c + k then 1 else 0*
| *ReifDeltaCostUpper k ⇒ if c' ≤ c + k then 1 else 0*
| *ReifDeltaCost k ⇒ if k = c' − c then 1 else 0*
| *ReifPrimedCostGe k ⇒ if c' ≥ k then 1 else 0*
| *ReifGeq (Original v) ⇒ if v ∈ s ∨ v ∉ s' then 1 else 0*
| *ReifLeq (Original v) ⇒ if v ∉ s ∨ v ∈ s' then 1 else 0*
| *ReifEq (Original v) ⇒ if (v ∈ s ⟷ v ∈ s') then 1 else 0*
| *ReifAction i ⇒ if i = sel-i then 1 else 0*
| *- ⇒ 0*

lemma *two-state-rho-range*:
∀ v. two-state-rho Π s c s' c' sel-i as v = 0 ∨ two-state-rho Π s c s' c' sel-i as v = 1
⟨*proof*⟩

lemma *pb-sum-pos-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, Pos (g v))) xs) rho = card (set xs ∩ T)*
⟨*proof*⟩

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)*
⟨*proof*⟩

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*⟩

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*⟩

lemma *pb-sum-primed-cost-bits-gen*:
assumes $\forall\ i < k.\ rho\ (PrimedCostBit\ i) = (c\ div\ 2\hat{}i)\ mod\ 2$
shows $pb\text{-}sum\ (map\ (\lambda i.\ (2\hat{}i,\ Pos\ (PrimedCostBit\ i)))\ [0..<k])\ rho = c\ mod\ 2\hat{}k$
⟨*proof*⟩

lemma *encode-transition-sound*:
fixes $\Pi :: 'v::linorder\ strips\text{-}task$
assumes *fin-V*: *finite V*
and *sel-lt*: *sel-i* $<$ *length as*
and *appl*: *applicable* (*as* ! *sel-i*) *s*
and *succ-eq*: $s' = successor\ (as\ !\ sel\text{-}i)\ s$
and *cost-eq*: $c' = c + cost\ (as\ !\ sel\text{-}i)$
and $c'\text{-}lt\text{-}B$: $c' < B$
and *pre-sub*: $pre\ (as\ !\ sel\text{-}i) \subseteq V$
and *add-sub*: $add\ (as\ !\ sel\text{-}i) \subseteq V$
and *del-sub*: $del\ (as\ !\ sel\text{-}i) \subseteq V$
shows *models* (*encode-transition as V B*)
(*two-state-rho* Π *s c s' c' sel-i as*)
⟨*proof*⟩

lemma *constraint-pvars-reification*:
assumes $\varphi \in reification\ r\ coeffs\ A$
shows $constraint\text{-}pvars\ \varphi \subseteq \{pvar\text{-}of\text{-}lit\ r\} \cup (pvar\text{-}of\text{-}lit\ `\ snd\ `\ set\ coeffs)$
⟨*proof*⟩
lemma *constraint-pvars-action-constraint-mem*:
$v \in constraint\text{-}pvars\ (action\text{-}constraint\ r\ a\ V\ B) \Longrightarrow$
$v = pvar\text{-}of\text{-}lit\ r \vee v = ReifDeltaCost\ (cost\ a) \vee is\text{-}input\text{-}pvar\ v \vee$
$(\exists u.\ v = ReifEq\ u) \vee v = ReifPrimedCostGe\ B$
⟨*proof*⟩
lemma *enc-trans-pvars-not-circ*:
fixes $C\text{-}\varphi :: 'v::linorder\ pb\text{-}circuit$
assumes $\varphi\text{-}in$: $\varphi \in encode\text{-}transition\ as\ V\ B$
and *extra-disj*: $\forall v \in circuit\text{-}reif\text{-}pvars\ C\text{-}\varphi.$
$(\forall k.\ v \neq ReifDeltaCost\ k) \wedge (\forall k.\ v \neq ReifDeltaCostLower\ k) \wedge$
$(\forall k.\ v \neq ReifDeltaCostUpper\ k) \wedge$
$(\forall k.\ v \neq ReifPrimedCostGe\ k) \wedge v \neq ReifT \wedge$
$(\forall u.\ v \neq ReifGeq\ u) \wedge (\forall u.\ v \neq ReifLeq\ u) \wedge (\forall u.\ v \neq ReifEq\ u) \wedge (\forall i.\ v \neq ReifAction\ i)$
shows $\forall\ v \in constraint\text{-}pvars\ \varphi.\ (\exists k.\ v = ReifDeltaCost\ k) \vee (\exists k.\ v = ReifDeltaCostLower\ k) \vee$
$(\exists k.\ v = ReifDeltaCostUpper\ k) \vee$
$(\exists k.\ v = ReifPrimedCostGe\ k) \vee v = ReifT \vee$
$(\exists u.\ v = ReifGeq\ u) \vee (\exists u.\ v = ReifLeq\ u) \vee (\exists u.\ v = ReifEq\ u) \vee$
$(\exists i.\ v = ReifAction\ i) \vee is\text{-}input\text{-}pvar\ v$
⟨*proof*⟩

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*⟩

lemma *lift-to-var-map-pvar-orig*:
assumes $\bigwedge i.\ p \neq$ *PrimedCostBit i*
shows *lift-to-var rho* (*map-pvar Original p*) = *rho p*
⟨*proof*⟩

lemma *eval-lit-map-pvar-lift-to-var-orig*:
assumes $\forall\, i.$ *pvar-of-lit l* ≠ *PrimedCostBit i*
shows *eval-lit* (*map-literal* (*map-pvar Original*) *l*) (*lift-to-var rho*) = *eval-lit l rho*
⟨*proof*⟩

lemma *eval-lit-neg-lit-map-pvar-lift-to-var-orig*:
assumes $\forall\, i.$ *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*
⟨*proof*⟩

lemma *lift-to-var-primed-pvar-map*:
assumes $\bigwedge i.\ p \neq$ *PrimedCostBit i*
shows *lift-to-var rho* (*primed-pvar-map p*) = *rho p*
⟨*proof*⟩

lemma *eval-lit-primed-pvar-map-lift-to-var*:
assumes $\forall\, i.$ *pvar-of-lit l* ≠ *PrimedCostBit i*
shows *eval-lit* (*map-literal primed-pvar-map l*) (*lift-to-var rho*) = *eval-lit l rho*
⟨*proof*⟩

lemma *pb-sum-map-literal-lift-to-var-orig*:
assumes *no-pcb*: $\forall\,(a,\ l) \in$ *set cs.* $\forall\, i.$ *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*
⟨*proof*⟩

lemma *pb-sum-lit-neg-map-literal-lift-to-var-orig*:
assumes *no-pcb*: $\forall\,(a,\ l) \in$ *set cs.* $\forall\, i.$ *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*
⟨*proof*⟩

lemma *pb-sum-map-literal-lift-to-var-primed*:
assumes *no-pcb*: $\forall\,(a,\ l) \in$ *set cs.* $\forall\, i.$ *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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *lift-to-var-models-circuit-constraints-primed*:
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 primed-pvar-map C*)) (*lift-to-var rho*)
⟨*proof*⟩

lemma *pvar-of-lit-map-literal-gen*: *pvar-of-lit* (*map-literal f l*) = *f* (*pvar-of-lit l*)
⟨*proof*⟩

lemma *lift-circuit-reif-eq*[*simp*]:
circuit-reif-pvars (*lift-circuit f C*) = *f* ‘ *circuit-reif-pvars C*
⟨*proof*⟩

lemma *orig-reif-eq*[*simp*]:
circuit-reif-pvars (*orig-circuit C*) = *map-pvar Original* ‘ *circuit-reif-pvars C*
⟨*proof*⟩

lemma *primed-reif-eq*[*simp*]:
circuit-reif-pvars (*primed-circuit C*) = *primed-pvar-map* ‘ *circuit-reif-pvars C*
⟨*proof*⟩

lemma *snd-lift-circuit*[*simp*]: *snd* (*lift-circuit f C*) = *map-literal f* (*snd C*)
⟨*proof*⟩

lemma *snd-orig-circuit*[*simp*]: *snd* (*orig-circuit C*) = *map-literal* (*map-pvar Original*) (*snd C*)
⟨*proof*⟩

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* ∨ *v* ∈ *circuit-reif-pvars C-φ*
and *no-pcb*: ∀ (*r*, *φ*) ∈ *set* (*fst C-φ*). ∀ *v* ∈ *constraint-pvars φ*. ∀ *i*. *v* ≠ *PrimedCostBit 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 *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 *c-lt-B*: *c* + *cost a* < *B*
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 (*successor a s*) (*c* + *cost a*)
⟨*proof*⟩

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*⟩

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*⟩

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* ≤ 1
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *pb-sum-unit-list*:
  pb-sum (*map* ($\lambda v.$ (*1*, *lt v*)) *xs*) *rho* $= (\sum v \leftarrow xs.$ *eval-lit* (*lt v*) *rho*)
  ⟨*proof*⟩

lemma *pb-sum-unit-set*:
  assumes *finite S*
  shows *pb-sum* (*map* ($\lambda v.$ (*1*, *lt v*)) (*sorted-list-of-set S*)) *rho*
      $= (\sum v \in S.$ *eval-lit* (*lt v*) *rho*)
⟨*proof*⟩

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*⟩

lemma *pb-sum-pos-ex*:
assumes *pb-sum coeffs rho* ≥ *1*
shows ∃ (*a*, *l*) ∈ *set coeffs*. *a* ∗ *eval-lit l rho* ≥ *1*
⟨*proof*⟩

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*: ∀ *rho*. (∀ *v*. *rho v* = *0* ∨ *rho v* = *1*) ⟶ *models CC rho* ⟶ *satisfies C rho*
and *A-pos*: *snd C* ≥ (*1*::*nat*)
shows *unsat-01* (*CC* ∪ {*constraint-neg C*})
⟨*proof*⟩

lemma *semantic-to-cpr*:
assumes ∀ *rho*. (∀ *v*. *rho v* = *0* ∨ *rho v* = *1*) ⟶ *models CC rho* ⟶ *satisfies C rho*
and *snd C* ≥ (*1*::*nat*)
shows *cpr-derives CC C*
⟨*proof*⟩

lemma *cpr-derives-iff-semantic*:
assumes *snd C* ≥ (*1*::*nat*)
shows *cpr-derives CC C* ⟷
(∀ *rho*. (∀ *v*. *rho v* = *0* ∨ *rho v* = *1*) ⟶ *models CC rho* ⟶ *satisfies C rho*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *pb-sum-bits-val*:
pb-sum (*map* (λ*i*. (*2^i*, *Pos* (*cb i*))) [*0*..<*k*]) *rho* = *bits-val k cb rho*
⟨*proof*⟩

lemma *bits-val-lt*:
assumes ∀ *i*. *rho* (*cb i*) = *0* ∨ *rho* (*cb i*) = *1*
shows *bits-val k cb rho* < *2^k*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *pb-sum-neg-bits-val*:
assumes *rho01*: ∀ *v*. *rho v* = *0* ∨ *rho v* = *1*
shows *pb-sum* (*map* (λ*i*. (*2^i*, *Neg* (*cb i*))) [*0*..<*k*]) *rho*
= (*2^k* − *1*) − *bits-val k cb rho*
⟨*proof*⟩

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* (λ*i*. (*2^i*, *Pos* (*cb i*))) [*0*..<*k*]) *A*) *rho*
shows *eval-lit r rho* = *1* ⟷ *bits-val k cb rho* ≥ *A*
⟨*proof*⟩

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*⟩

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*⟩

lemma *models-mono*:
models CC rho ⟹ *DD* ⊆ *CC* ⟹ *models DD rho*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟩

lemma *sum-eq-card-ones*:
fixes *f* :: *'a* ⇒ *nat*
assumes ∀ *v*∈*S*. *f v* = *1*
shows (∑ *v*∈*S*. *f v*) = *card S*
⟨*proof*⟩

lemma *sum-rho-all-one*:
assumes *rho01*: ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
and *fin*: *finite S*
and *total*: (∑ *v*∈*S*. *rho* (*f v*)) ≥ *card S*
shows ∀ *v*∈*S*. *rho* (*f v*) = *1*
⟨*proof*⟩

lemma *sum-rho-all-zero*:
assumes *rho01*: ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
and *fin*: *finite S*
and *total*: (∑ *v*∈*S*. *1* − *rho* (*f v*)) ≥ *card S*
shows ∀ *v*∈*S*. *rho* (*f v*) = *0*
⟨*proof*⟩

lemma *sum-rho-le-card*:
assumes *rho01*: $\forall x.\ rho\ x = 0 \lor rho\ x = 1$
and *fin*: *finite S*
shows $(\sum v{\in}S.\ rho\ (f\ v)) \leq card\ S$
⟨*proof*⟩

lemma *sum-one-minus-le-card*:
fixes *rho* :: $'w \Rightarrow nat$
assumes *fin*: *finite S*
shows $(\sum v{\in}S.\ 1 - rho\ (f\ v)) \leq card\ S$
⟨*proof*⟩

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*: $\forall x.\ rho\ x = 0 \lor rho\ x = 1$
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*⟩

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

lemma *encode-goal-forces*:
fixes Π :: $'v$::*linorder strips-task*
assumes *rho01*: $\forall x.\ rho\ x = 0 \lor rho\ x = 1$
and *m*: *models* (*encode-goal* Π) *rho*
and *fin*: *finite* (*goal* Π)
shows *rho ReifG* = *1* $\longleftrightarrow$ $(\forall v \in goal\ \Pi.\ rho\ (StateVar\ v) = 1)$
⟨*proof*⟩

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*: $\forall x.\ rho\ x = 0 \lor rho\ x = 1$
and *m*: *models* (*action-selection-reif rs*) *rho*
shows *rho ReifT* = *1* $\longleftrightarrow$ $(\exists r \in set\ rs.\ eval\text{-}lit\ r\ rho = 1)$
⟨*proof*⟩

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** $rho :: 'v\ var\ pvar \Rightarrow nat$
  assumes *rho01*: $\forall x.\ rho\ x = 0 \vee rho\ x = 1$
    and *sat*: *satisfies* (*action-constraint r a V B*) *rho*
    and *out*: *eval-lit r rho* $= 1$
    and *finV*: *finite V*
    and *pre-sub*: $pre\ a \subseteq V$ **and** *add-sub*: $add\ a \subseteq V$ **and** *del-sub*: $del\ a \subseteq V$
  shows $rho\ (ReifDeltaCost\ (cost\ a)) = 1$
      $\wedge\ rho\ (ReifPrimedCostGe\ B) = 0$
      $\wedge\ (\forall v \in pre\ a.\ rho\ (StateVar\ (Original\ v)) = 1)$
      $\wedge\ (\forall v \in add\ a - del\ a.\ rho\ (StateVar\ (Primed\ v)) = 1)$
      $\wedge\ (\forall v \in del\ a - add\ a.\ rho\ (StateVar\ (Primed\ v)) = 0)$
      $\wedge\ (\forall v \in V - evars\ a.\ rho\ (ReifEq\ (Original\ v)) = 1)$
⟨*proof*⟩

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*) ⇒ $a \leq b$
  | (*Inl* -, *Inr* -) ⇒ *True*
  | (*Inr* -, *Inl* -) ⇒ *False*
  | (*Inr a*, *Inr b*) ⇒ $a \leq b$

definition *less-sum* :: $'a + 'b \Rightarrow 'a + 'b \Rightarrow bool$ **where**
  less-sum x y ≡ $x \leq y \wedge \neg\ y \leq x$

instance
  ⟨*proof*⟩

end

definition *embed-act* :: $'v\ action \Rightarrow ('v + 'g)\ action$ **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*
⟨*proof*⟩

lemma *embed-act-successor*:
successor (*embed-act a*) (*Inl* ‘ *s*) = *Inl* ‘ *successor a s*
⟨*proof*⟩

lemma *embed-path*:
assumes *path* (*acts* Π) *s t* π
shows *path* (*acts* (*embed-task* Π)) (*Inl* ‘ *s*) (*Inl* ‘ *t*) (*map embed-act* π)
⟨*proof*⟩

lemma *embed-plan-cost*:
plan-cost (*map embed-act* π) = *plan-cost* π
⟨*proof*⟩

lemma *embed-is-plan-for*:
assumes *is-plan-for* Π π
shows *is-plan-for* (*embed-task* Π) (*map embed-act* π)
⟨*proof*⟩

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*⟩

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
⟨*proof*⟩

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* ($?CC \cup \{constraint\text{-}neg\ ?C\}) \Longrightarrow cpr\text{-}derives\ ?CC\ ?C$ rule of the formal system, so no additional proof rule is needed.

4.1 Clipping arithmetic

definition *clip* :: *nat* ⇒ *int* ⇒ *nat* **where**
  clip B l ≡ *min B* (*nat l*)

lemma *clip-nonpos*[*simp*]: *l* ≤ *0* ⟹ *clip B l* = *0*
  ⟨*proof*⟩

lemma *clip-ge-B*[*simp*]: *l* ≥ *int B* ⟹ *clip B l* = *B*
  ⟨*proof*⟩

lemma *clip-in-range*[*simp*]: *0* ≤ *l* ⟹ *l* ≤ *int B* ⟹ *clip B l* = *nat l*
⟨*proof*⟩

lemma *clip-le-B*: *clip B l* ≤ *B*
  ⟨*proof*⟩

lemma *clip-mono*:
  assumes *j* ≤ *j'*
  shows *clip B j* ≤ *clip B j'*
⟨*proof*⟩

lemma *nat-add-int-le*: *nat* (*l* + *int m*) ≤ *nat l* + *m*
  ⟨*proof*⟩

lemma *clip-add-le*:
  clip B (*l* + *int m*) ≤ *clip B l* + *m*
⟨*proof*⟩

4.2 K-gates and their semantics

The gate triple for a K-gate with name literal r: it reifies $\Sigma\ 2\hat{}i \cdot c \geq clip\ B\ l$ 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* ⇒ (*nat* × *'w pvar literal*) *list* **where**
k-gate-body B ≡ *map* (λ*i*. (*2^i*, *Pos* (*CostBit i*))) [*0..<bits-needed B*]

definition *k-gate* :: *'w pvar literal* ⇒ *nat* ⇒ *int* ⇒
'w pvar literal × (*nat* × *'w pvar literal*) *list* × *nat* **where**
k-gate r B l ≡ (*r*, *k-gate-body B*, *clip B l*)

lemma *k-gate-forces*:
assumes *rho01*: ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
and *m*: *models* (*reification r* (*k-gate-body B*) (*clip B l*)) *rho*
shows *eval-lit r rho* = *1* ⟷ *bits-val* (*bits-needed B*) *CostBit rho* ≥ *clip B l*
⟨*proof*⟩

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*: ∀ *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*⟩

Lemma 2 of the paper, semantically: from $cost \geq clip\ B\ l$ and $\Delta c = m$ conclude $cost' \geq clip\ B\ (l + m)$. 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* (λ*i*. (*2^i*, *Pos* (*PrimedCostBit i*))) [*0..<bits-needed B*]

lemma *k-gate-primed-forces*:
assumes *rho01*: ∀ *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*
⟨*proof*⟩

lemma *k-gate-step-semantic*:
assumes *rho01*: ∀ *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*⟩

Bit-level form of the premise *cost* ≥ *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* ⇒ *int* ⇒ *′w pconstraint* **where**
clip-cost-constraint B l ≡ (*k-gate-body B*, *clip B l*)

lemma *clip-cost-constraint-sat*:
assumes *rho01*: ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
shows *satisfies* (*clip-cost-constraint B l*) *rho*
⟷ *bits-val* (*bits-needed B*) *CostBit rho* ≥ *clip B l*
⟨*proof*⟩

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* ≤ *snd ?C* ⟹ *cpr-derives ?CC ?C* = (∀ *rho*. (∀ *v*. *rho v* = *0* ∨ *rho v* = *1*) ⟶ *models ?CC rho* ⟶ *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* ∘ *f*)
⟨*proof*⟩

lemma *lit-neg-map-literal*:
lit-neg (*map-literal f l*) = *map-literal f* (*lit-neg l*)
⟨*proof*⟩

lemma *pb-sum-map-literal*:
pb-sum (*map* (λ(*a*, *l*). (*a*, *map-literal f l*)) *cs*) *rho* = *pb-sum cs* (*rho* ∘ *f*)
⟨*proof*⟩

lemma *models-Union-iff*:
models $(\bigcup x{\in}S.\ F\ x)$ *rho* $\longleftrightarrow$ $(\forall\, x{\in}S.\ models\ (F\ x)\ rho)$
⟨*proof*⟩

lemma *models-reification-lift*:
models (*reification* (*map-literal f r*) (*map* (λ(*a*, *l*). (*a*, *map-literal f l*)) *cs*) *A*) *rho*
$\longleftrightarrow$ *models* (*reification r cs A*) (*rho* ∘ *f*)
⟨*proof*⟩

lemma *models-circuit-constraints-lift*:
models (*circuit-constraints* (*lift-circuit f C*)) *rho*
$\longleftrightarrow$ *models* (*circuit-constraints C*) (*rho* ∘ *f*)
⟨*proof*⟩

lemma *rho01-comp*:
$(\forall\, x.\ rho\ x = 0 \lor rho\ x = 1) \Longrightarrow (\forall\, x.\ (rho \circ f)\ x = 0 \lor (rho \circ f)\ x = 1)$
⟨*proof*⟩

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 ≡ $\bigcup (r,\ cs,\ A) \in set\ (hc\text{-}gates\ HC).\ reification\ r\ cs\ A$

lemma *hc-constraints-eq-circuit*:
hc-constraints HC = *circuit-constraints* (*hc-gates HC*, *out*)
⟨*proof*⟩

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: $(r_s \land cost{\geq}max\{0,Bh(s)\}) \rightarrow r^h{}_s$.)

definition *hc-state-lemma* ::
′*u strips-task* ⇒ *nat* ⇒ (′*u*) *heuristic-cert* ⇒ ′*u state* ⇒ *bool* **where**
hc-state-lemma Π*e B HC s* ≡
∀ *rho*. $(\forall\, x.\ rho\ x = 0 \lor rho\ x = 1) \longrightarrow$
models (*hc-constraints HC*) *rho* $\longrightarrow$
$(\forall\, v \in vars\ \Pi e.\ rho\ (StateVar\ v) = (if\ v \in s\ then\ 1\ else\ 0)) \longrightarrow$
bits-val (*bits-needed B*) *CostBit rho* ≥ *clip B* (*int B* − *int* (*hc-h HC s*)) $\longrightarrow$
eval-lit (*hc-out HC s*) *rho* = *1*

lemma *hc-state-lemmaD*:
assumes *hc-state-lemma* Π*e B HC s*
and $\forall\, x.\ rho\ x = 0 \lor rho\ x = 1$
and *models* (*hc-constraints HC*) *rho*

and $\forall\, v \in vars\ \Pi e.\ rho\ (StateVar\ v) = (if\ v \in s\ then\ 1\ else\ 0)$
and $bits\text{-}val\ (bits\text{-}needed\ B)\ CostBit\ rho \geq clip\ B\ (int\ B - int\ (hc\text{-}h\ HC\ s))$
shows $eval\text{-}lit\ (hc\text{-}out\ HC\ s)\ rho = 1$
⟨*proof*⟩

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* ::
$'u\ strips\text{-}task \Rightarrow nat \Rightarrow ('u)\ heuristic\text{-}cert \Rightarrow 'u\ state \Rightarrow bool$ **where**
$hc\text{-}goal\text{-}lemma\ \Pi e\ B\ HC\ s \equiv$
$\forall\, rho.\ (\forall\, x.\ rho\ x = 0 \vee rho\ x = 1) \longrightarrow$
$models\ (hc\text{-}constraints\ HC)\ rho \longrightarrow$
$(\forall\, v \in goal\ \Pi e.\ rho\ (StateVar\ v) = 1) \longrightarrow$
$eval\text{-}lit\ (hc\text{-}out\ HC\ s)\ rho = 1 \longrightarrow$
$bits\text{-}val\ (bits\text{-}needed\ B)\ CostBit\ rho \geq B$

lemma *hc-goal-lemmaD*:
assumes $hc\text{-}goal\text{-}lemma\ \Pi e\ B\ HC\ s$
and $\forall\, x.\ rho\ x = 0 \vee rho\ x = 1$
and $models\ (hc\text{-}constraints\ HC)\ rho$
and $\forall\, v \in goal\ \Pi e.\ rho\ (StateVar\ v) = 1$
and $eval\text{-}lit\ (hc\text{-}out\ HC\ s)\ rho = 1$
shows $bits\text{-}val\ (bits\text{-}needed\ B)\ CostBit\ rho \geq B$
⟨*proof*⟩

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* ::
$'u{::}linorder\ strips\text{-}task \Rightarrow nat \Rightarrow 'u\ action\ list \Rightarrow ('u)\ heuristic\text{-}cert \Rightarrow 'u\ state \Rightarrow bool$ **where**
$hc\text{-}ind\text{-}lemma\ \Pi e\ B\ as\ HC\ s \equiv$
$\forall\, rho.\ (\forall\, x.\ rho\ x = 0 \vee rho\ x = 1) \longrightarrow$
$models\ (circuit\text{-}constraints\ (orig\text{-}circuit\ (hc\text{-}gates\ HC,\ hc\text{-}out\ HC\ s)))\ rho \longrightarrow$
$models\ (circuit\text{-}constraints\ (primed\text{-}circuit\ (hc\text{-}gates\ HC,\ hc\text{-}out\ HC\ s)))\ rho \longrightarrow$
$models\ (encode\text{-}transition\ as\ (vars\ \Pi e)\ B)\ rho \longrightarrow$
$models\ (encode\text{-}cost\text{-}ge\ B)\ rho \longrightarrow$
$rho\ ReifT = 1 \longrightarrow$
$eval\text{-}lit\ (map\text{-}literal\ (map\text{-}pvar\ Original)\ (hc\text{-}out\ HC\ s))\ rho = 1 \longrightarrow$
$eval\text{-}lit\ (map\text{-}literal\ primed\text{-}pvar\text{-}map\ (hc\text{-}out\ HC\ s))\ rho = 1$

lemma *hc-ind-lemmaD*:
assumes $hc\text{-}ind\text{-}lemma\ \Pi e\ B\ as\ HC\ s$
and $\forall\, x.\ rho\ x = 0 \vee rho\ x = 1$
and $models\ (circuit\text{-}constraints\ (orig\text{-}circuit\ (hc\text{-}gates\ HC,\ hc\text{-}out\ HC\ s)))\ rho$
and $models\ (circuit\text{-}constraints\ (primed\text{-}circuit\ (hc\text{-}gates\ HC,\ hc\text{-}out\ HC\ s)))\ rho$
and $models\ (encode\text{-}transition\ as\ (vars\ \Pi e)\ B)\ rho$

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*
⟨*proof*⟩

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*⟩

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*⟩

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 Π*e s* ≡
(λ*v*. *unit-clause* (*Pos* (*StateVar v*))) ‘ (*vars* Π*e* ∩ *s*)
∪ (λ*v*. *unit-clause* (*Neg* (*StateVar v*))) ‘ (*vars* Π*e* − *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*⟩

lemma *unit-clause-neg-sat*:
assumes *rho01*: ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
shows *satisfies* (*unit-clause* (*Neg w*)) *rho* ⟷ *rho w* = *0*
⟨*proof*⟩

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

theorem *hc-state-lemma-cpr*:
fixes Π*e* :: *'u*::*linorder strips-task*
assumes *sl*: *hc-state-lemma* Π*e B HC s*
shows *cpr-derives*
(*hc-constraints HC* ∪ *state-unit-clauses* Π*e s*
∪ {*clip-cost-constraint B* (*int B* − *int* (*hc-h HC s*))})
(*unit-clause* (*hc-out HC s*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *conj-gate-forces*:
assumes *rho01*: ∀ *x*. *rho x* = *0* ∨ *rho x* = *1*
and *m*: *models* (*reification r* (*map* (λ*l*. (*1*, *l*)) *ls*) (*length ls*)) *rho*
shows *eval-lit r rho* = *1* ⟷ (∀ *l*∈*set ls*. *eval-lit l rho* = *1*)
⟨*proof*⟩

lemma *disj-gate-forces*:

assumes *rho01*: $\forall x.\ rho\ x = 0 \vee rho\ x = 1$
and *m*: *models* (*reification r* (*map* ($\lambda l.\ (1,\ l)$) *ls*) *1*) *rho*
shows *eval-lit r rho* = *1* $\longleftrightarrow$ ($\exists\, l{\in}$*set ls. eval-lit l rho* = *1*)
⟨*proof*⟩

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 ⟨*A*, *r-A∗*⟩ 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* ⟹ *action-reifs as* ! *j* = *Pos* (*ReifAction j*)
⟨*proof*⟩

lemma *trans-sel*:
assumes *models* (*encode-transition as V B*) *rho*
shows *models* (*action-selection-reif* (*action-reifs as*)) *rho*
⟨*proof*⟩

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*⟩

lemma *trans-delta*:
assumes *m*: *models* (*encode-transition as V B*) *rho* **and** *a-in*: *a* ∈ *set as*
shows *models* (*encode-delta-cost* (*cost a*) (*bits-needed B*)) *rho*
⟨*proof*⟩

lemma *trans-eq-var*:
assumes *m*: *models* (*encode-transition as V B*) *rho* **and** *v-in*: *v* ∈ *V*
shows *models* (*encode-eq-var v*) *rho*
⟨*proof*⟩

lemma *trans-primed-ge*:
assumes *m*: *models* (*encode-transition as V B*) *rho*
shows *models* (*encode-cost-ge-primed B*) *rho*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *evars-embed*: *evars* (*embed-act a*) = *Inl* ‘ *evars a*
⟨*proof*⟩

lemma *acts-embed*: *acts* (*embed-task* Π) = *embed-act* ‘ *acts* Π
⟨*proof*⟩

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* ($\lambda(r, cs, A)$. *pvar-of-lit* r) (*hc-gates HC*))
and *hc-wf*: $\forall i <$ *length* (*hc-gates HC*). *case hc-gates HC* ! i *of* $(r, cs, A) \Rightarrow$
($\forall x \in$ *pvar-of-lit* ‘ *snd* ‘ *set cs*.
($\exists v.\ x =$ *StateVar* v) $\vee$ ($\exists j.\ x =$ *CostBit* j)
$\vee\ x \in$ *pvar-of-lit* ‘ *fst* ‘ *set* (*take i* (*hc-gates HC*)))
and *hc-out-in*: $\forall s \in$ *set open-list*. *hc-out HC* (*Inl* ‘ s) $\in$ *fst* ‘ *set* (*hc-gates HC*)
begin

abbreviation $\Pi e ::$ ($'v$ + *nat*) *strips-task* **where**
$\Pi e \equiv$ *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 (λv. *if* $v \in s$ *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*
⟨*proof*⟩

lemma *astar-gates-nth-k*:
i < *n-cl* ⟹ *astar-gates* ! *i* = *kg i*
⟨*proof*⟩

lemma *astar-gates-nth-cl*:
i < *n-cl* ⟹ *astar-gates* ! (*n-cl* + *i*) = *clg i*
⟨*proof*⟩

lemma *astar-gates-nth-hc*:
i < *n-hc* ⟹ *astar-gates* ! (*2* ∗ *n-cl* + *i*) = *hc-gates HC* ! *i*
⟨*proof*⟩

lemma *astar-gates-nth-out*:
astar-gates ! (*2* ∗ *n-cl* + *n-hc*) = *outg*
⟨*proof*⟩

lemma *kg-in-gates*: $i < n\text{-}cl \Longrightarrow kg\ i \in set\ astar\text{-}gates$
⟨*proof*⟩

lemma *clg-in-gates*: $i < n\text{-}cl \Longrightarrow clg\ i \in set\ astar\text{-}gates$
⟨*proof*⟩

lemma *hc-in-gates*: $g \in set\ (hc\text{-}gates\ HC) \Longrightarrow g \in set\ astar\text{-}gates$
⟨*proof*⟩

lemma *outg-in-gates*: $outg \in set\ astar\text{-}gates$
⟨*proof*⟩

6.2.4 Extracting per-gate models from a circuit model

lemma *models-gate*:
assumes *m*: $models\ (circuit\text{-}constraints\ astar\text{-}circuit)\ rho$
and *g-in*: $(r,\ cs,\ A) \in set\ astar\text{-}gates$
shows $models\ (reification\ r\ cs\ A)\ rho$
⟨*proof*⟩

lemma *models-kg*:
assumes $models\ (circuit\text{-}constraints\ astar\text{-}circuit)\ rho$ **and** $i < n\text{-}cl$
shows $models\ (reification\ (Pos\ (k\text{-}name\ i))\ (k\text{-}gate\text{-}body\ B)\ (clip\ B\ (int\ (cl\text{-}g\ i))))\ rho$
⟨*proof*⟩

lemma *models-clg*:
assumes $models\ (circuit\text{-}constraints\ astar\text{-}circuit)\ rho$ **and** $i < n\text{-}cl$
shows $models\ (reification\ (Pos\ (cl\text{-}name\ i))\ (map\ (\lambda l.\ (1,\ l))\ (cl\text{-}lits\ i))\ (length\ (cl\text{-}lits\ i)))\ rho$
⟨*proof*⟩

lemma *models-outg*:
assumes $models\ (circuit\text{-}constraints\ astar\text{-}circuit)\ rho$
shows $models\ (reification\ (Pos\ out\text{-}name)\ (map\ (\lambda l.\ (1,\ l))\ out\text{-}lits)\ 1)\ rho$
⟨*proof*⟩

lemma *models-hc-constraints*:
assumes $models\ (circuit\text{-}constraints\ astar\text{-}circuit)\ rho$
shows $models\ (hc\text{-}constraints\ HC)\ rho$
⟨*proof*⟩

6.2.5 Semantics of the closed-state gates

lemma *state-lits-exact-sem*:
assumes *rho01*: $\forall x.\ rho\ x = 0 \vee rho\ x = 1$
shows $(\forall l \in set\ (state\text{-}lits\text{-}exact\ s).\ eval\text{-}lit\ l\ rho = 1)$
$\longleftrightarrow (\forall v \in vars\ \Pi.\ rho\ (StateVar\ (Inl\ v)) = (if\ v \in s\ then\ 1\ else\ 0))$
⟨*proof*⟩

lemma *clg-forces*:
assumes *rho01*: $\forall x.\ rho\ x = 0 \vee rho\ x = 1$
and *m*: *models* (*circuit-constraints astar-circuit*) *rho*
and *i-lt*: $i < n\text{-}cl$
shows *rho* (*cl-name i*) = *1*
$\longleftrightarrow$ (*rho* (*k-name i*) = *1* $\wedge$
$(\forall v \in vars\ \Pi.\ rho\ (StateVar\ (Inl\ v)) = (if\ v \in cl\text{-}state\ i\ then\ 1\ else\ 0)))$
⟨*proof*⟩

lemma *kg-forces*:
assumes *rho01*: $\forall x.\ rho\ x = 0 \vee rho\ x = 1$
and *m*: *models* (*circuit-constraints astar-circuit*) *rho*
and *i-lt*: $i < n\text{-}cl$
shows *rho* (*k-name i*) = *1*
$\longleftrightarrow$ *bits-val* (*bits-needed B*) *CostBit rho* $\geq$ *clip B* (*int* (*cl-g i*))
⟨*proof*⟩

lemma *outg-forces*:
assumes *rho01*: $\forall x.\ rho\ x = 0 \vee rho\ x = 1$
and *m*: *models* (*circuit-constraints astar-circuit*) *rho*
shows *rho out-name* = *1* $\longleftrightarrow$ $(\exists l \in set\ out\text{-}lits.\ eval\text{-}lit\ l\ rho = 1)$
⟨*proof*⟩

6.2.6 Embedded task bookkeeping

lemma *vars-e*: *vars* Πe = *Inl* ‘ *vars* Π
⟨*proof*⟩

lemma *goal-e*: *goal* Πe = *Inl* ‘ *goal* Π
⟨*proof*⟩

lemma *init-e*: *init* Πe = *Inl* ‘ *init* Π
⟨*proof*⟩

lemma *fin-vars-e*: *finite* (*vars* Πe)
⟨*proof*⟩

lemma *init-sub-e*: *init* $\Pi e \subseteq$ *vars* Πe
⟨*proof*⟩

lemma *goal-sub-e*: *goal* $\Pi e \subseteq$ *vars* Πe
⟨*proof*⟩

lemma *fin-goal-e*: *finite* (*goal* Πe)
⟨*proof*⟩

lemma *closed-nth-in*: $i < n\text{-}cl \Longrightarrow$ (*cl-state i*, *cl-g i*) $\in$ *set closed-list*
⟨*proof*⟩

lemma *closed-mem-nth*: $(s, g) \in$ *set closed-list* $\Longrightarrow \exists\, i <$ *n-cl*. *cl-state* $i = s \wedge$ *cl-g* $i = g$
⟨*proof*⟩

lemma *hc-ok-state*: $s \in$ *set open-list* $\Longrightarrow$ *hc-state-lemma* Π*e B HC* (*Inl* ‘ *s*)
⟨*proof*⟩

lemma *hc-ok-goal*: $s \in$ *set open-list* $\Longrightarrow$ *hc-goal-lemma* Π*e B HC* (*Inl* ‘ *s*)
⟨*proof*⟩

lemma *hc-ok-ind*: $s \in$ *set open-list* $\Longrightarrow$ *hc-ind-lemma* Π*e B cas HC* (*Inl* ‘ *s*)
⟨*proof*⟩

lemma *state-descr-translate*:
$(\forall\, w \in$ *vars* Π*e*. *rho* (*StateVar w*) = (*if w* ∈ *Inl* ‘ *s then 1 else 0*))
$\longleftrightarrow$ $(\forall\, v \in$ *vars* Π. *rho* (*StateVar* (*Inl v*)) = (*if v* ∈ *s then 1 else 0*))
⟨*proof*⟩

lemma *neg-cost-ge-one-zero*:
assumes *rho01*: $\forall\, x$. (*rho* :: (*'v* + *nat*) *pvar* ⇒ *nat*) *x* = *0* ∨ *rho x* = *1*
and *sat*: *satisfies* (*neg-cost-ge-one B*) *rho*
shows *bits-val* (*bits-needed B*) *CostBit rho* = *0*
⟨*proof*⟩

6.2.7 The initial state lemma (paper Lemma 3)

lemma *astar-init-semantic*:
assumes *rho01*: $\forall\, x$. *rho x* = *0* ∨ *rho x* = *1*
and *mI*: *models* (*encode-init* Π*e*) *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*⟩

6.2.8 The goal lemma (paper Lemma 4)

lemma *astar-goal-semantic*:
assumes *rho01*: $\forall\, x$. *rho x* = *0* ∨ *rho x* = *1*
and *mG*: *models* (*encode-goal* Π*e*) *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*⟩

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*⟩

6.2.10 Gate names: distinctness and freshness

lemma *fst-kg*: *fst* (*kg i*) = *Pos* (*k-name i*)
  ⟨*proof*⟩

lemma *fst-clg*: *fst* (*clg i*) = *Pos* (*cl-name i*)
  ⟨*proof*⟩

lemma *fst-outg*: *fst outg* = *Pos out-name*
  ⟨*proof*⟩

lemma *hc-name-form*:
  $g \in$ *set* (*hc-gates HC*) $\Longrightarrow \exists j.$ *pvar-of-lit* (*fst g*) = *ReifCert* (*Inr* (2 ∗ *j* + 1))
  ⟨*proof*⟩

lemma *names-eq*:
  map ($\lambda g.$ *pvar-of-lit* (*fst g*)) *astar-gates*
   = *map k-name* [0..<*n-cl*] @ *map cl-name* [0..<*n-cl*]
     @ *map* ($\lambda g.$ *pvar-of-lit* (*fst g*)) (*hc-gates HC*) @ [*out-name*]
  ⟨*proof*⟩

lemma *parity-neq*: (2::*nat*) ∗ *m* ≠ 2 ∗ *j* + 1
  ⟨*proof*⟩

lemma *hc-names-odd-set*:
  assumes $x \in$ *set* (*map* ($\lambda g.$ *pvar-of-lit* (*fst g*)) (*hc-gates HC*))
  shows $\exists j.$ x = *ReifCert* (*Inr* (2 ∗ *j* + 1))
⟨*proof*⟩

lemma *distinct-gate-names*: *distinct* (*map* ($\lambda g.$ *pvar-of-lit* (*fst g*)) *astar-gates*)
⟨*proof*⟩

lemma *distinct-reif-astar*: *distinct-reif-vars astar-circuit*

⟨*proof*⟩

lemma *names-are-cert*:
  $v \in$ *circuit-reif-pvars astar-circuit* $\Longrightarrow \exists\, w.\ v =$ *ReifCert w*
⟨*proof*⟩

lemma *astar-freshness*:
  $\forall\, v \in$ *circuit-reif-pvars astar-circuit*.
    $\neg$ *is-input-pvar v* $\wedge$ $v \neq$ *ReifI* $\wedge$ $(\forall\, k.\ v \neq$ *ReifCostGe k*$)$ $\wedge$ $v \neq$ *ReifG* $\wedge$
    $(\forall\, k.\ v \neq$ *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*$)$
⟨*proof*⟩

6.2.11 Well-formedness of the assembled circuit

lemma *nth-in-take*: $j < i \Longrightarrow i \leq$ *length xs* $\Longrightarrow$ *xs* ! *j* $\in$ *set* (*take i xs*)
  ⟨*proof*⟩

lemma *take-nth-ex*: $g' \in$ *set* (*take j xs*) $\Longrightarrow \exists\, p.\ p < j \wedge p <$ *length xs* $\wedge\ g' =$ *xs* ! *p*
  ⟨*proof*⟩

lemma *earlier-name*:
  assumes $j < i$ **and** $i \leq$ *length astar-gates*
  shows *pvar-of-lit* (*fst* (*astar-gates* ! *j*)) $\in$ *pvar-of-lit* ‘ *fst* ‘ *set* (*take i astar-gates*)
  ⟨*proof*⟩

lemma *wf-astar-circuit*: *wf-circuit astar-circuit*
⟨*proof*⟩

lemma *astar-body-pvars*:
  assumes *g-in*: $(r,\ cs,\ A) \in$ *set astar-gates*
    and *x-in*: $x \in$ *pvar-of-lit* ‘ *snd* ‘ *set cs*
  shows $(\exists\, v.\ x =$ *StateVar v*$) \vee (\exists\, i.\ x =$ *CostBit i*$) \vee (\exists\, w.\ x =$ *ReifCert w*$)$
⟨*proof*⟩

lemma *astar-no-pcb*:
  $\forall\, (r, \varphi) \in$ *set* (*fst astar-circuit*). $\forall\, v \in$ *constraint-pvars* φ. $\forall\, i.\ v \neq$ *PrimedCostBit i*
⟨*proof*⟩

6.2.12 Output literal of the assembled circuit

lemma *snd-circ*: *snd astar-circuit* = *Pos out-name*
  ⟨*proof*⟩

lemma *snd-orig-astar*: *snd* (*orig-circuit astar-circuit*) = *Pos* (*map-pvar Original out-name*)
⟨*proof*⟩

lemma *snd-primed-astar*: *snd* (*primed-circuit astar-circuit*) = *Pos* (*primed-pvar-map out-name*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *astar-ind-cpr*:
cpr-derives
(*encode-transition cas* (*vars* Π*e*) *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*⟩

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*⟩

theorem *astar-lower-bound*:
assumes *is-plan-for* Π *π*
shows *plan-cost π* ≥ *B*
⟨*proof*⟩

corollary *astar-optimal*:
assumes *is-plan-for* Π *π* **and** *plan-cost π* = *B*
shows *optimal-plan* Π *π*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *abs-mem-nth*:
assumes *sa* ⊆ *P*
shows ∃ *i*. *i* < *n-s* ∧ *sa-i i* = *sa*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *absg-in-gates*: *i* < *n-s* ⟹ *absg i* ∈ *set pdb-gates*
⟨*proof*⟩

lemma *kgg-in-gates*: *i* < *n-s* ⟹ *kgg i* ∈ *set pdb-gates*
⟨*proof*⟩

lemma *thrg-in-gates*: *i* < *n-s* ⟹ *thrg i* ∈ *set pdb-gates*
⟨*proof*⟩

lemma *poutg-in-gates*: *poutg* ∈ *set pdb-gates*
⟨*proof*⟩

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*⟩

lemma *models-absg*:
assumes *models* (*hc-constraints pdb-cert*) *rho* **and** *i* < *n-s*
shows *models* (*reification* (*Pos* (*abs-name i*))
(*map* (λ*l*. (*1*, *l*)) (*abs-lits i*)) (*length* (*abs-lits i*))) *rho*
⟨*proof*⟩

lemma *models-kgg*:
assumes *models* (*hc-constraints pdb-cert*) *rho* **and** *i* < *n-s*

shows *models* (*reification* (*Pos* (*kk-name i*)) (*k-gate-body B*)
(*clip B* (*int B* − *int* (*d* (*sa-i i*))))) *rho*
⟨*proof*⟩

lemma *models-thrg*:
assumes *models* (*hc-constraints pdb-cert*) *rho* **and** *i* < *n-s*
shows *models* (*reification* (*Pos* (*thr-name i*))
(*map* (λ*l*. (*1*, *l*)) (*thr-lits i*)) (*length* (*thr-lits i*))) *rho*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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* $\longleftrightarrow$ $(\exists\, i < n\text{-}s.\ rho\ (thr\text{-}name\ i) = 1)$
⟨*proof*⟩

7.2 Lemma 8: the state lemma

lemma *pdb-state-lemma*: *hc-state-lemma* Π*e B pdb-cert s*
⟨*proof*⟩

7.3 Lemma 9: the goal lemma

lemma *pdb-goal-lemma*: *hc-goal-lemma* Π*e B pdb-cert s*
⟨*proof*⟩

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*
⟨*proof*⟩

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* Π*e B as pdb-cert S*
⟨*proof*⟩

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*: $i < n\text{-}s \Longrightarrow pdb\text{-}gates\ !\ i = absg\ i$
⟨*proof*⟩

lemma *pdb-gates-nth-kgg*: $i < n\text{-}s \Longrightarrow pdb\text{-}gates\ !\ (n\text{-}s + i) = kgg\ i$
⟨*proof*⟩

lemma *pdb-gates-nth-thrg*: $i < n\text{-}s \Longrightarrow pdb\text{-}gates\ !\ (2 * n\text{-}s + i) = thrg\ i$
⟨*proof*⟩

lemma *pdb-gates-nth-out*: *pdb-gates* ! (*3* ∗ *n-s*) = *poutg*
⟨*proof*⟩

lemma *pdb-gate-name-nth*:
assumes *p-lt*: *p* < *length pdb-gates*
shows *fst* (*pdb-gates* ! *p*) = *Pos* (*ReifCert* (*nm p*))
⟨*proof*⟩

lemma *pdb-names*:
∀(*r*, *cs*, *A*) ∈ *set* (*hc-gates pdb-cert*). ∃*j*. *r* = *Pos* (*ReifCert* (*nm j*))
⟨*proof*⟩

lemma *pdb-distinct*:
distinct (*map* (λ(*r*, *cs*, *A*). *pvar-of-lit r*) (*hc-gates pdb-cert*))
⟨*proof*⟩

lemma *pdb-out-in*: *hc-out pdb-cert s* ∈ *fst* ‘ *set* (*hc-gates pdb-cert*)
⟨*proof*⟩

lemma *nth-in-take-pdb*: *j* < *i* ⟹ *i* ≤ *length xs* ⟹ *xs* ! *j* ∈ *set* (*take i xs*)
⟨*proof*⟩

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*⟩

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* Πe. $W\ v = h$
    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
  $\langle proof \rangle$

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

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*
⟨*proof*⟩

lemma *kvg-in-gates*: *i < n-v ⟹ kvg i ∈ set hmax-gates*
⟨*proof*⟩

lemma *rvg-in-gates*: *i < n-v ⟹ rvg i ∈ set hmax-gates*
⟨*proof*⟩

lemma *kbg-in-gates*: *kbg ∈ set hmax-gates*
⟨*proof*⟩

lemma *mxg-in-gates*: *mxg ∈ set hmax-gates*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*: $i < n\text{-}v$
shows *rho* (*kv-name i*) = *1*
$\longleftrightarrow$ *bits-val* (*bits-needed B*) *CostBit rho* $\geq$ *clip B* (*int B* − *int h* + *int* (*W* (*v-i i*)))
⟨*proof*⟩

lemma *rvg-forces*:
assumes *rho01*: $\forall x.$ *rho x* = *0* $\vee$ *rho x* = *1*
and *m*: *models* (*hc-constraints hmax-cert*) *rho*
and *i-lt*: $i < n\text{-}v$
shows *rho* (*rv-name i*) = *1*
$\longleftrightarrow$ *rho* (*StateVar* (*v-i i*)) = *0* $\vee$ *rho* (*kv-name i*) = *1*
⟨*proof*⟩

lemma *kbg-forces*:
assumes *rho01*: $\forall x.$ *rho x* = *0* $\vee$ *rho x* = *1*
and *m*: *models* (*hc-constraints hmax-cert*) *rho*
shows *rho kb-name* = *1*
$\longleftrightarrow$ *bits-val* (*bits-needed B*) *CostBit rho* $\geq$ *clip B* (*int B* − *int h*)
⟨*proof*⟩

lemma *mxg-forces*:
assumes *rho01*: $\forall x.$ *rho x* = *0* $\vee$ *rho x* = *1*
and *m*: *models* (*hc-constraints hmax-cert*) *rho*
shows *rho max-name* = *1*
$\longleftrightarrow$ *rho kb-name* = *1* $\wedge$ ($\forall i < n\text{-}v.$ *rho* (*rv-name i*) = *1*)
⟨*proof*⟩

8.2 Lemma 14: the state lemma

lemma *hmax-state-lemma*: *hc-state-lemma* Π*e B hmax-cert s*
⟨*proof*⟩

8.3 Lemma 15: the goal lemma

lemma *hmax-goal-lemma*: *hc-goal-lemma* Π*e B hmax-cert s′*
⟨*proof*⟩

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* Π*e B as hmax-cert s′*
⟨*proof*⟩

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* $\Pi e\ B$ *as hmax-cert* $\{s\}$
⟨*proof*⟩

8.5 Structural conditions for use in the A* certificate

lemma *hmax-gates-nth-kv*: $i < n\text{-}v \Longrightarrow hmax\text{-}gates\ !\ i = kvg\ i$
⟨*proof*⟩

lemma *hmax-gates-nth-rv*: $i < n\text{-}v \Longrightarrow hmax\text{-}gates\ !\ (n\text{-}v + i) = rvg\ i$
⟨*proof*⟩

lemma *hmax-gates-nth-kb*: $hmax\text{-}gates\ !\ (2 * n\text{-}v) = kbg$
⟨*proof*⟩

lemma *hmax-gates-nth-mx*: $hmax\text{-}gates\ !\ (2 * n\text{-}v + 1) = mxg$
⟨*proof*⟩

lemma *hmax-gate-name-nth*:
assumes *p-lt*: $p < length\ hmax\text{-}gates$
shows $fst\ (hmax\text{-}gates\ !\ p) = Pos\ (ReifCert\ (nm\ p))$
⟨*proof*⟩

lemma *hmax-names*:
$\forall (r,\ cs,\ A) \in set\ (hc\text{-}gates\ hmax\text{-}cert).\ \exists j.\ r = Pos\ (ReifCert\ (nm\ j))$
⟨*proof*⟩

lemma *hmax-distinct*:
$distinct\ (map\ (\lambda(r,\ cs,\ A).\ pvar\text{-}of\text{-}lit\ r)\ (hc\text{-}gates\ hmax\text{-}cert))$
⟨*proof*⟩

lemma *hmax-out-in*: $hc\text{-}out\ hmax\text{-}cert\ s' \in fst\ `\ set\ (hc\text{-}gates\ hmax\text{-}cert)$
⟨*proof*⟩

lemma *nth-in-take-hmax*: $j < i \Longrightarrow i \leq length\ xs \Longrightarrow xs\ !\ j \in set\ (take\ i\ xs)$
⟨*proof*⟩

lemma *hmax-wf*:
$\forall i < length\ (hc\text{-}gates\ hmax\text{-}cert).\ case\ hc\text{-}gates\ hmax\text{-}cert\ !\ i\ of\ (r,\ cs,\ A) \Rightarrow$
$(\forall x \in pvar\text{-}of\text{-}lit\ `\ snd\ `\ set\ cs.$
$(\exists v.\ x = StateVar\ v) \vee (\exists j.\ x = CostBit\ j)$
$\vee\ x \in pvar\text{-}of\text{-}lit\ `\ fst\ `\ set\ (take\ i\ (hc\text{-}gates\ hmax\text{-}cert)))$
⟨*proof*⟩

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* :: $'w \Rightarrow$ *nat* **and** *f* :: $'u \Rightarrow 'w$
  assumes *rho01*: $\forall x.$ *rho* $x = 0 \vee$ *rho* $x = 1$
    and *YZ*: $Y \subseteq Z$
    and *al-pat*: $\forall v \in Y.$ *rho* $(f\ v) =$ (*if* $v \in$ *al then 1 else 0*)
  shows $\exists t \subseteq Z.\ t \cap Y = al \cap Y \wedge (\forall v \in Z.$ *rho* $(f\ v) =$ (*if* $v \in t$ *then 1 else 0*))
⟨*proof*⟩

9.2 The efficient PDB locale

locale *efficient-pdb* =
  fixes Πe :: $'u$::*linorder strips-task*
    and B :: *nat*
    and P :: $'u$ *set*

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

lemma *fin-P*: $finite\ P$
⟨*proof*⟩

definition *n-s* :: *nat* **where**
$n\text{-}s = length\ Ss$

definition *sa-i* :: $nat \Rightarrow 'u\ state$ **where**
$sa\text{-}i\ i = Ss\ !\ i$

lemma *sa-i-sub*: $i < n\text{-}s \Longrightarrow sa\text{-}i\ i \subseteq P$
⟨*proof*⟩

lemma *sa-i-fin*: $i < n\text{-}s \Longrightarrow fin\ (sa\text{-}i\ i)$
⟨*proof*⟩

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*⟩

9.2.1 Gate names

definition *abs-name* :: $nat \Rightarrow 'u\ pvar$ **where**
$abs\text{-}name\ i = ReifCert\ (nm\ i)$

definition *kk-name* :: $nat \Rightarrow 'u\ pvar$ **where**
$kk\text{-}name\ i = ReifCert\ (nm\ (n\text{-}s + i))$

definition *thr-name* :: $nat \Rightarrow 'u\ pvar$ **where**
$thr\text{-}name\ i = ReifCert\ (nm\ (2 * n\text{-}s + i))$

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*
⟨*proof*⟩

lemma *absg-in-gates*: *i* < *n-s* ⟹ *absg i* ∈ *set epdb-gates*
⟨*proof*⟩

lemma *kgg-in-gates*: *i* < *n-s* ⟹ *kgg i* ∈ *set epdb-gates*
⟨*proof*⟩

lemma *thrg-in-gates*: *i* < *n-s* ⟹ *thrg i* ∈ *set epdb-gates*
⟨*proof*⟩

lemma *infg-in-gates*: *infg* ∈ *set epdb-gates*
⟨*proof*⟩

lemma *poutg-in-gates*: *poutg* ∈ *set epdb-gates*
⟨*proof*⟩

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*⟩

lemma *models-absg*:
assumes *models* (*hc-constraints epdb-cert*) *rho* **and** *i* < *n-s*
shows *models* (*reification* (*Pos* (*abs-name i*))
(*map* (λ*l*. (*1*, *l*)) (*abs-lits i*)) (*length* (*abs-lits i*))) *rho*
⟨*proof*⟩

lemma *models-kgg*:
assumes *models* (*hc-constraints epdb-cert*) *rho* **and** *i* < *n-s*
shows *models* (*reification* (*Pos* (*kk-name i*)) (*k-gate-body B*)
(*clip B* (*int B* − *int* (*d* (*sa-i i*))))) *rho*
⟨*proof*⟩

lemma *models-thrg*:
assumes *models* (*hc-constraints epdb-cert*) *rho* **and** *i* < *n-s*
shows *models* (*reification* (*Pos* (*thr-name i*))
(*map* (λ*l*. (*1*, *l*)) (*thr-lits i*)) (*length* (*thr-lits i*))) *rho*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *thrg-forces*:
  assumes *rho01*: $\forall\, x.$ *rho* $x = 0 \vee$ *rho* $x = 1$
    and *m*: *models* (*hc-constraints epdb-cert*) *rho*
    and *i-lt*: $i <$ *n-s*
  shows *rho* (*thr-name i*) $= 1 \longleftrightarrow$ *rho* (*abs-name i*) $= 1 \wedge$ *rho* (*kk-name i*) $= 1$
⟨*proof*⟩

lemma *infg-forces*:
  assumes *rho01*: $\forall\, x.$ *rho* $x = 0 \vee$ *rho* $x = 1$
    and *m*: *models* (*hc-constraints epdb-cert*) *rho*
  shows *rho inf-name* $= 1 \longleftrightarrow (\forall\, i <$ *n-s*. *rho* (*abs-name i*) $= 0)$
⟨*proof*⟩

lemma *poutg-forces*:
  assumes *rho01*: $\forall\, x.$ *rho* $x = 0 \vee$ *rho* $x = 1$
    and *m*: *models* (*hc-constraints epdb-cert*) *rho*
  shows *rho pout-name* $= 1$
    $\longleftrightarrow$ *rho inf-name* $= 1 \vee (\exists\, i <$ *n-s*. *rho* (*thr-name i*) $= 1)$
⟨*proof*⟩

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*: $\forall\, x.$ *rho* $x = 0 \vee$ *rho* $x = 1$
    and *m*: *models* (*hc-constraints epdb-cert*) *rho*
    and *nf*: $\neg$ *fin* (*abs-state rho*)
  shows *rho inf-name* $= 1$
⟨*proof*⟩

9.3 Lemma 19: the state lemma

lemma *epdb-state-lemma*: *hc-state-lemma* Π*e B epdb-cert s*
  ⟨*proof*⟩

9.4 Lemma 20: the goal lemma

lemma *epdb-goal-lemma*: *hc-goal-lemma* Π*e B epdb-cert s*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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* Π*e B as epdb-cert S*
⟨*proof*⟩

9.6 Structural conditions for use in the A* certificate

lemma *epdb-gates-nth-abs*: $i < n\text{-}s \Longrightarrow epdb\text{-}gates\ !\ i = absg\ i$
⟨*proof*⟩

lemma *epdb-gates-nth-kgg*: $i < n\text{-}s \Longrightarrow epdb\text{-}gates\ !\ (n\text{-}s + i) = kgg\ i$
⟨*proof*⟩

lemma *epdb-gates-nth-thrg*: $i < n\text{-}s \Longrightarrow epdb\text{-}gates\ !\ (2 * n\text{-}s + i) = thrg\ i$
⟨*proof*⟩

lemma *epdb-gates-nth-inf*: $epdb\text{-}gates\ !\ (3 * n\text{-}s) = infg$
⟨*proof*⟩

lemma *epdb-gates-nth-out*: $epdb\text{-}gates\ !\ (3 * n\text{-}s + 1) = poutg$
⟨*proof*⟩

lemma *epdb-gate-name-nth*:
assumes *p-lt*: $p < length\ epdb\text{-}gates$
shows $fst\ (epdb\text{-}gates\ !\ p) = Pos\ (ReifCert\ (nm\ p))$
⟨*proof*⟩

lemma *epdb-names*:
$\forall (r, cs, A) \in set\ (hc\text{-}gates\ epdb\text{-}cert).\ \exists j.\ r = Pos\ (ReifCert\ (nm\ j))$
⟨*proof*⟩

lemma *epdb-distinct*:
$distinct\ (map\ (\lambda(r, cs, A).\ pvar\text{-}of\text{-}lit\ r)\ (hc\text{-}gates\ epdb\text{-}cert))$
⟨*proof*⟩

lemma *epdb-out-in*: $hc\text{-}out\ epdb\text{-}cert\ s \in fst\ `\ set\ (hc\text{-}gates\ epdb\text{-}cert)$
⟨*proof*⟩

lemma *nth-in-take-epdb*: $j < i \Longrightarrow i \leq length\ xs \Longrightarrow xs\ !\ j \in set\ (take\ i\ xs)$
⟨*proof*⟩

lemma *epdb-wf*:
$\forall i < length\ (hc\text{-}gates\ epdb\text{-}cert).\ case\ hc\text{-}gates\ epdb\text{-}cert\ !\ i\ of\ (r, cs, A) \Rightarrow$
$(\forall x \in pvar\text{-}of\text{-}lit\ `\ snd\ `\ set\ cs.$
$(\exists v.\ x = StateVar\ v) \vee (\exists j.\ x = CostBit\ j)$
$\vee\ x \in pvar\text{-}of\text{-}lit\ `\ fst\ `\ set\ (take\ i\ (hc\text{-}gates\ epdb\text{-}cert)))$
⟨*proof*⟩

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*}
  ⟨*proof*⟩

10.2 Interpretation of *pdb-heuristic*

interpretation *demo-pdb*: *pdb-heuristic demo-task 1* {*0*::*nat*}
  λsa. if 0 ∈ sa then 0 else 1 [{}, {*0*}] [*demo-act*] *id*
⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  λv. 1 [*0*] [*demo-act*] *id*
⟨*proof*⟩

lemma *hc-valid demo-task 1* [*demo-act*] *demo-hmax.hmax-cert* {{}}
  ⟨*proof*⟩

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*}
  λsa. if 0 ∈ sa then 0 else 1 λsa. True [{}, {*0*}] [*demo-act*] *id*
⟨*proof*⟩

lemma *hc-valid demo-task 1* [*demo-act*] *demo-epdb.epdb-cert S*
  ⟨*proof*⟩

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*}
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *demo-powE*: *set* [{}, {*Inl 0*}] :: (*nat + nat*) *state*] = *Pow* {*Inl 0*}
  ⟨*proof*⟩

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*⟩

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-act demo-act*]
⟨*proof*⟩

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*⟩

lemma *demo-cost*: *plan-cost* [*demo-act*] = *1*
⟨*proof*⟩

theorem *demo-optimal*: *optimal-plan demo-task* [*demo-act*]
⟨*proof*⟩

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*
⟨*proof*⟩

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.