

Semantics of AI Planning Languages

Mohammad Abdulaziz and Peter Lammich*

This is an Isabelle/HOL formalisation of the semantics of the multi-valued planning tasks language that is used by the planning system Fast-Downward [3], the STRIPS [2] fragment of the Planning Domain Definition Language [5] (PDDL), and the STRIPS soundness meta-theory developed by Lifschitz [4]. It also contains formally verified checkers for checking the well-formedness of problems specified in either language as well the correctness of potential solutions. The formalisation in this entry was described in an earlier publication [1].

Contents

1	Semantics of Fast-Downward’s Multi-Valued Planning Tasks Language	3
1.1	Syntax	3
1.1.1	Well-Formedness	3
1.2	Semantics as Transition System	4
1.2.1	Preservation of well-formedness	6
2	An Executable Checker for Multi-Valued Planning Problem Solutions	7
2.1	Auxiliary Lemmas	7
2.2	Well-formedness Check	8
2.3	Execution	8
3	PDDL and STRIPS Semantics	14
3.1	Utility Functions	15
3.2	Abstract Syntax	15
3.2.1	Generic Entities	15
3.2.2	Domains	16
3.2.3	Problems	16
3.2.4	Plans	17
3.2.5	Ground Actions	17
3.3	Closed-World Assumption, Equality, and Negation	17
3.3.1	Proper Generalization	19

*Author names are alphabetically ordered.

3.4	STRIPS Semantics	20
3.5	Well-Formedness of PDDL	21
3.6	PDDL Semantics	25
3.7	Preservation of Well-Formedness	27
3.7.1	Well-Formed Action Instances	27
3.7.2	Preservation	31
4	Executable PDDL Checker	33
4.1	Generic DFS Reachability Checker	34
4.2	Implementation Refinements	36
4.2.1	Of-Type	36
4.2.2	Application of Effects	39
4.2.3	Well-Formedness	39
4.2.4	Execution of Plan Actions	41
4.2.5	Checking of Plan	43
4.3	Executable Plan Checker	45
4.4	Code Setup	47
4.4.1	Code Equations	47
4.4.2	Setup for Containers Framework	49
4.4.3	More Efficient Distinctness Check for Linorders	49
4.4.4	Code Generation	49
5	Soundness theorem for the STRIPS semantics	50
5.1	Soundness Theorem for PDDL	54

```

theory SASP-Semantics
imports Main
begin

```

1 Semantics of Fast-Downward's Multi-Valued Planning Tasks Language

1.1 Syntax

```

type-synonym name = string
type-synonym ast-variable = name × nat option × name list
type-synonym ast-variable-section = ast-variable list
type-synonym ast-initial-state = nat list
type-synonym ast-goal = (nat × nat) list
type-synonym ast-precond = (nat × nat)
type-synonym ast-effect = ast-precond list × nat × nat option × nat
type-synonym ast-operator = name × ast-precond list × ast-effect list × nat
type-synonym ast-operator-section = ast-operator list

```

```

type-synonym ast-problem =
  ast-variable-section × ast-initial-state × ast-goal × ast-operator-section

```

```

type-synonym plan = name list

```

1.1.1 Well-Formedness

```

locale ast-problem =
  fixes problem :: ast-problem
begin
  definition astDom :: ast-variable-section
    where astDom ≡ case problem of (D,I,G,δ) ⇒ D
  definition astI :: ast-initial-state
    where astI ≡ case problem of (D,I,G,δ) ⇒ I
  definition astG :: ast-goal
    where astG ≡ case problem of (D,I,G,δ) ⇒ G
  definition astδ :: ast-operator-section
    where astδ ≡ case problem of (D,I,G,δ) ⇒ δ

  definition numVars ≡ length astDom
  definition numVals x ≡ length (snd (snd (astDom!x)))

```

```

definition wf-partial-state ps ≡
  distinct (map fst ps)
  ∧ (∀(x,v) ∈ set ps. x < numVars ∧ v < numVals x)

```

```

definition wf-operator :: ast-operator ⇒ bool
  where wf-operator ≡ λ(name, pres, effs, cost).
    wf-partial-state pres
    ∧ distinct (map (λ(-, v, -, -). v) effs) — This may be too restrictive

```

```

 $\wedge (\forall (epres, x, vp, v) \in set\ effs.$ 
   $wf\text{-}partial\text{-}state\ epres$ 
 $\wedge x < numVars \wedge v < numVals\ x$ 
 $\wedge (case\ vp\ of\ None \Rightarrow True \mid Some\ v \Rightarrow v < numVals\ x)$ 
 $)$ 

```

```

definition well-formed  $\equiv$ 
  — Initial state
   $length\ astI = numVars$ 
 $\wedge (\forall x < numVars. astI!x < numVals\ x)$ 

  — Goal
 $\wedge wf\text{-}partial\text{-}state\ astG$ 

  — Operators
 $\wedge (distinct\ (map\ fst\ ast\delta))$ 
 $\wedge (\forall \pi \in set\ ast\delta. wf\text{-}operator\ \pi)$ 

```

end

```

locale wf-ast-problem = ast-problem +
  assumes wf: well-formed
begin
  lemma wf-initial:
     $length\ astI = numVars$ 
 $\forall x < numVars. astI!x < numVals\ x$ 
    using wf unfolding well-formed-def by auto

  lemma wf-goal: wf-partial-state astG
    using wf unfolding well-formed-def by auto

  lemma wf-operators:
     $distinct\ (map\ fst\ ast\delta)$ 
 $\forall \pi \in set\ ast\delta. wf\text{-}operator\ \pi$ 
    using wf unfolding well-formed-def by auto
end

```

1.2 Semantics as Transition System

```

type-synonym state =  $nat \rightarrow nat$ 
type-synonym pstate =  $nat \rightarrow nat$ 

```

```

context ast-problem
begin

```

```

  definition Dom ::  $nat\ set$  where  $Dom = \{0..<numVars\}$ 

```

definition *range-of-var* **where** *range-of-var* $x \equiv \{0..<numVals\ x\}$

definition *valid-states* $:: state\ set$ **where** *valid-states* $\equiv \{$
 $s. dom\ s = Dom \wedge (\forall x \in Dom. the\ (s\ x) \in range-of-var\ x)$
 $\}$

definition *I* $:: state$
where *I* $v \equiv if\ v < length\ astI\ then\ Some\ (astI!v)\ else\ None$

definition *subsuming-states* $:: pstate \Rightarrow state\ set$
where *subsuming-states* *partial* $\equiv \{ s \in valid-states. partial \subseteq_m s \}$

definition *G* $:: state\ set$
where *G* $\equiv subsuming-states\ (map-of\ astG)$

end

definition *implicit-pres* $:: ast-effect\ list \Rightarrow ast-precond\ list$ **where**
implicit-pres *effs* \equiv
 $map\ (\lambda(-,v,vpre,-). (v,the\ vpre))$
 $(filter\ (\lambda(-,vpre,-). vpre \neq None)\ effs)$

context *ast-problem*
begin

definition *lookup-operator* $:: name \Rightarrow ast-operator\ option$ **where**
lookup-operator *name* $\equiv find\ (\lambda(n,-,-). n=name)\ ast\delta$

definition *enabled* $:: name \Rightarrow state \Rightarrow bool$
where *enabled* *name* *s* \equiv
 $case\ lookup-operator\ name\ of$
 $Some\ (-,pres,effs,-) \Rightarrow$
 $s \in subsuming-states\ (map-of\ pres)$
 $\wedge\ s \in subsuming-states\ (map-of\ (implicit-pres\ effs))$
 $| None \Rightarrow False$

definition *eff-enabled* $:: state \Rightarrow ast-effect \Rightarrow bool$ **where**
eff-enabled *s* $\equiv \lambda(pres,-,-). s \in subsuming-states\ (map-of\ pres)$

definition *execute* $:: name \Rightarrow state \Rightarrow state$ **where**
execute *name* *s* \equiv
 $case\ lookup-operator\ name\ of$
 $Some\ (-,-,effs,-) \Rightarrow$
 $s ++ map-of\ (map\ (\lambda(-,x,-,v). (x,v))\ (filter\ (eff-enabled\ s)\ effs))$
 $| None \Rightarrow undefined$

fun *path-to* **where**
path-to *s* $\square s' \longleftrightarrow s'=s$

| $path\text{-}to\ s\ (\pi\#\pi s)\ s' \longleftrightarrow enabled\ \pi\ s \wedge path\text{-}to\ (execute\ \pi\ s)\ \pi s\ s'$

definition $valid\text{-}plan :: plan \Rightarrow bool$
where $valid\text{-}plan\ \pi s \equiv \exists s' \in G. path\text{-}to\ I\ \pi s\ s'$

end

1.2.1 Preservation of well-formedness

context $wf\text{-}ast\text{-}problem$

begin

lemma $I\text{-}valid: I \in valid\text{-}states$

using $wf\text{-}initial$

unfolding $valid\text{-}states\text{-}def\ Dom\text{-}def\ I\text{-}def\ range\text{-}of\text{-}var\text{-}def$

by $(auto\ split\text{:}if\text{-}splits)$

lemma $lookup\text{-}operator\text{-}wf:$

assumes $lookup\text{-}operator\ name = Some\ \pi$

shows $wf\text{-}operator\ \pi\ fst\ \pi = name$

proof –

obtain $name'\ pres\ effs\ cost$ **where** $[simp]: \pi = (name', pres, effs, cost)$ **by** $(cases\ \pi)$

hence $[simp]: name' = name$ **and** $IN\text{-}AST: (name, pres, effs, cost) \in set\ ast\delta$

using $assms$

unfolding $lookup\text{-}operator\text{-}def$

apply –

apply $(metis\ (mono\text{-}tags,\ lifting)\ case\text{-}prodD\ find\text{-}Some\text{-}iff)$

by $(metis\ (mono\text{-}tags,\ lifting)\ case\text{-}prodD\ find\text{-}Some\text{-}iff\ nth\text{-}mem)$

from $IN\text{-}AST$ **show** $WF: wf\text{-}operator\ \pi\ fst\ \pi = name$

unfolding $enabled\text{-}def$ **using** $wf\text{-}operators$ **by** $auto$

qed

lemma $execute\text{-}preserves\text{-}valid:$

assumes $s \in valid\text{-}states$

assumes $enabled\ name\ s$

shows $execute\ name\ s \in valid\text{-}states$

proof –

from $\langle enabled\ name\ s \rangle$ **obtain** $name'\ pres\ effs\ cost$ **where**

$[simp]: lookup\text{-}operator\ name = Some\ (name', pres, effs, cost)$

unfolding $enabled\text{-}def$ **by** $(auto\ split\text{:}option.\text{splits})$

from $lookup\text{-}operator\text{-}wf[OF\ this]$ **have** $WF: wf\text{-}operator\ (name, pres, effs, cost)$

by $simp$

have $X1: s ++ m \in valid\text{-}states$ **if** $\forall x\ v. m\ x = Some\ v \longrightarrow x < numVars \wedge v < numVals\ x$ **for** m

using $that\ \langle s \in valid\text{-}states \rangle$

```

    by (auto
        simp: valid-states-def Dom-def range-of-var-def map-add-def dom-def
        split: option.splits)

  have X2:  $x < \text{numVars}$   $v < \text{numVals}$   $x$ 
    if map-of (map ( $\lambda(-, x, -, y). (x, y)$ ) (filter (eff-enabled s) effs))  $x = \text{Some}$ 
    v
    for  $x$   $v$ 
  proof -
    from that obtain  $\text{epres}$   $vp$  where ( $\text{epres}, x, vp, v$ )  $\in \text{set effs}$ 
    by (auto dest!: map-of-SomeD)
    with WF show  $x < \text{numVars}$   $v < \text{numVals}$   $x$ 
    unfolding wf-operator-def by auto
  qed

  show ?thesis
  using assms
  unfolding enabled-def execute-def
  by (auto intro!: X1 X2)
qed

lemma path-to-pres-valid:
  assumes  $s \in \text{valid-states}$ 
  assumes path-to  $s$   $\pi s$   $s'$ 
  shows  $s' \in \text{valid-states}$ 
  using assms
  by (induction  $s$   $\pi s$   $s'$  rule: path-to.induct) (auto simp: execute-preserves-valid)

end

end
theory SASP-Checker
imports SASP-Semantics
      HOL-Library.Code-Target-Nat
begin

```

2 An Executable Checker for Multi-Valued Planning Problem Solutions

2.1 Auxiliary Lemmas

```

lemma map-of-leI:
  assumes distinct (map fst l)
  assumes  $\bigwedge k v. (k, v) \in \text{set } l \implies m k = \text{Some } v$ 
  shows map-of  $l \subseteq_m m$ 
  using assms
  by (metis (no-types, opaque-lifting) domIff map-le-def map-of-SomeD not-Some-eq)

```

lemma *[simp]*: $fst \circ (\lambda(a, b, c, d). (f a b c d, g a b c d)) = (\lambda(a,b,c,d). f a b c d)$
 by *auto*

lemma *map-mp*: $m \subseteq_m m' \implies m k = Some v \implies m' k = Some v$
 by (*auto simp: map-le-def dom-def*)

lemma *map-add-map-of-fold*:
 fixes *ps* and $m :: 'a \rightarrow 'b$
 assumes *distinct* (*map fst ps*)
 shows $m ++ map\ of\ ps = fold (\lambda(k, v) m. m(k \mapsto v)) ps m$
proof –
 have *X1*: $(fold (\lambda(k, v) m. m(k \mapsto v)) ps m)(a \mapsto b)$
 $= fold (\lambda(k, v) m. m(k \mapsto v)) ps (m(a \mapsto b))$
 if $a \notin fst \ `set\ ps$
 for $a b ps$ and $m :: 'a \rightarrow 'b$
 using *that*
 by (*induction ps arbitrary: m*) (*auto simp: fun-upd-twist*)

 show *?thesis*
 using *assms*
 by (*induction ps arbitrary: m*) (*auto simp: X1*)
qed

2.2 Well-formedness Check

lemmas *wf-code-thms* =
ast-problem.astDom-def ast-problem.astI-def ast-problem.astG-def ast-problem.ast δ -def
ast-problem.numVars-def ast-problem.numVals-def
ast-problem.wf-partial-state-def ast-problem.wf-operator-def ast-problem.well-formed-def

declare *wf-code-thms[code]*

export-code *ast-problem.well-formed* in *SML*

2.3 Execution

definition *match-pre* :: *ast-precond* \Rightarrow *state* \Rightarrow *bool* **where**
match-pre $\equiv \lambda(x,v) s. s\ x = Some\ v$

definition *match-pres* :: *ast-precond list* \Rightarrow *state* \Rightarrow *bool* **where**
match-pres pres s $\equiv \forall pre \in set\ pres. match\ pre\ pre\ s$

definition *match-implicit-pres* :: *ast-effect list* \Rightarrow *state* \Rightarrow *bool* **where**
match-implicit-pres effs s $\equiv \forall (-,x,vp,-) \in set\ effs.$
 (*case vp of None* \Rightarrow *True* | *Some v* \Rightarrow $s\ x = Some\ v$)

definition *enabled-opr'* :: *ast-operator* \Rightarrow *state* \Rightarrow *bool* **where**

$enabled\text{-opr}' \equiv \lambda(name, pres, effs, cost) s. match\text{-pres } pres\ s \wedge match\text{-implicit}\text{-pres } effs\ s$

definition $eff\text{-enabled}' :: state \Rightarrow ast\text{-effect} \Rightarrow bool$ **where**
 $eff\text{-enabled}'\ s \equiv \lambda(pres, -, -, -). match\text{-pres } pres\ s$

definition $execute\text{-opr}' \equiv \lambda(name, -, effs, -) s.$
 $let\ effs = filter\ (eff\text{-enabled}'\ s)\ effs$
 $in\ fold\ (\lambda(-, x, -, v) s. s(x \mapsto v))\ effs\ s$

definition $lookup\text{-operator}' :: ast\text{-problem} \Rightarrow name \rightarrow ast\text{-operator}$
where $lookup\text{-operator}' \equiv \lambda(D, I, G, \delta) name. find\ (\lambda(n, -, -, -). n = name)\ \delta$

definition $enabled' :: ast\text{-problem} \Rightarrow name \Rightarrow state \Rightarrow bool$ **where**
 $enabled'\ problem\ name\ s \equiv$
 $case\ lookup\text{-operator}'\ problem\ name\ of$
 $\quad Some\ \pi \Rightarrow enabled\text{-opr}'\ \pi\ s$
 $\quad | None \Rightarrow False$

definition $execute' :: ast\text{-problem} \Rightarrow name \Rightarrow state \Rightarrow state$ **where**
 $execute'\ problem\ name\ s \equiv$
 $case\ lookup\text{-operator}'\ problem\ name\ of$
 $\quad Some\ \pi \Rightarrow execute\text{-opr}'\ \pi\ s$
 $\quad | None \Rightarrow undefined$

context $wf\text{-ast}\text{-problem}\ begin$

lemma $match\text{-pres}\text{-correct}$:
assumes D : $distinct\ (map\ fst\ pres)$
assumes $s \in valid\text{-states}$
shows $match\text{-pres } pres\ s \longleftrightarrow s \in subsuming\text{-states}\ (map\text{-of } pres)$
proof –
have $match\text{-pres } pres\ s \longleftrightarrow map\text{-of } pres \subseteq_m s$
unfolding $match\text{-pres}\text{-def } match\text{-pre}\text{-def}$
apply $(auto\ split: prod.\ splits)$
using $map\text{-le}\text{-def } map\text{-of}\text{-Some}D$ **apply** $fastforce$
by $(metis\ (full\text{-types})\ D\ domIff\ map\text{-le}\text{-def } map\text{-of}\text{-eq}\text{-Some}\text{-iff } option.\ simps(3))$
with $assms$ **show** $?thesis$
unfolding $subsuming\text{-states}\text{-def}$
by $auto$
qed

lemma $match\text{-implicit}\text{-pres}\text{-correct}$:
assumes D : $distinct\ (map\ (\lambda(-, v, -, -). v)\ effs)$
assumes $s \in valid\text{-states}$
shows $match\text{-implicit}\text{-pres } effs\ s \longleftrightarrow s \in subsuming\text{-states}\ (map\text{-of } (implicit\text{-pres}))$

```

effs))
proof –
  from assms show ?thesis
    unfolding subsuming-states-def
    unfolding match-implicit-pres-def implicit-pres-def
    apply (auto
      split: prod.splits option.splits
      simp: distinct-map-filter
      intro!: map-of-leI)
    apply (force simp: distinct-map-filter split: prod.split elim: map-mp)
    done
qed

lemma enabled-opr'-correct:
  assumes V: s∈valid-states
  assumes lookup-operator name = Some π
  shows enabled-opr' π s ⟷ enabled name s
  using lookup-operator-wf[OF assms(2)] assms
  unfolding enabled-opr'-def enabled-def wf-operator-def
  by (auto
    simp: match-pres-correct[OF - V] match-implicit-pres-correct[OF - V]
    simp: wf-partial-state-def
    split: option.split
  )

lemma eff-enabled'-correct:
  assumes V: s∈valid-states
  assumes case eff of (pres,-,-) ⇒ wf-partial-state pres
  shows eff-enabled' s eff ⟷ eff-enabled s eff
  using assms
  unfolding eff-enabled'-def eff-enabled-def wf-partial-state-def
  by (auto simp: match-pres-correct)

lemma execute-opr'-correct:
  assumes V: s∈valid-states
  assumes LO: lookup-operator name = Some π
  shows execute-opr' π s = execute name s
proof (cases π)
  case [simp]: (fields name pres effs)

  have [simp]: filter (eff-enabled' s) effs = filter (eff-enabled s) effs
  apply (rule filter-cong[OF refl])
  apply (rule eff-enabled'-correct[OF V])
  using lookup-operator-wf[OF LO]
  unfolding wf-operator-def by auto

  have X1: distinct (map fst (map (λ(-, x, -, y). (x, y)) (filter (eff-enabled s) effs)))

```

```

using lookup-operator-wf[OF LO]
unfolding wf-operator-def
by (auto simp: distinct-map-filter)

term filter (eff-enabled s) effs

have [simp]:
  fold (λ(-, x, -, v) s. s(x ↦ v)) l s =
  fold (λ(k, v) m. m(k ↦ v)) (map (λ(-, x, -, y). (x, y)) l) s
for l :: ast-effect list
by (induction l arbitrary: s) auto

show ?thesis
  unfolding execute-opr'-def execute-def using LO
  by (auto simp: map-add-map-of-fold[OF X1])
qed

lemma lookup-operator'-correct:
  lookup-operator' problem name = lookup-operator name
  unfolding lookup-operator'-def lookup-operator-def
  unfolding astδ-def
  by (auto split: prod.split)

lemma enabled'-correct:
  assumes V: s∈valid-states
  shows enabled' problem name s = enabled name s
  unfolding enabled'-def
  using enabled-opr'-correct[OF V]
  by (auto split: option.splits simp: enabled-def lookup-operator'-correct)

lemma execute'-correct:
  assumes V: s∈valid-states
  assumes enabled name s
  shows execute' problem name s = execute name s
  unfolding execute'-def
  using execute-opr'-correct[OF V] ⟨enabled name s⟩
  by (auto split: option.splits simp: enabled-def lookup-operator'-correct)

end

context ast-problem
begin

fun simulate-plan :: plan ⇒ state → state where
  simulate-plan [] s = Some s
  | simulate-plan (π#πs) s = (

```

```

    if enabled  $\pi$   $s$  then
      let  $s' = \text{execute } \pi$   $s$  in
        simulate-plan  $\pi$   $s$   $s'$ 
    else
      None
  )

```

lemma *simulate-plan-correct*: $\text{simulate-plan } \pi$ $s = \text{Some } s' \iff \text{path-to } s$ π s

by (*induction* s π s' *rule*: *path-to.induct*) *auto*

definition *check-plan* :: $\text{plan} \Rightarrow \text{bool}$ **where**

```

check-plan  $\pi$   $s = ($ 
  case simulate-plan  $\pi$   $s$   $I$  of
    None  $\Rightarrow$  False
  | Some  $s' \Rightarrow s' \in G$ )

```

lemma *check-plan-correct*: $\text{check-plan } \pi$ $s \iff \text{valid-plan } \pi$ s

unfolding *check-plan-def* *valid-plan-def*

by (*auto split*: *option.split simp*: *simulate-plan-correct[symmetric]*)

end

fun *simulate-plan'* :: $\text{ast-problem} \Rightarrow \text{plan} \Rightarrow \text{state} \rightarrow \text{state}$ **where**

```

simulate-plan' problem []  $s = \text{Some } s$ 
| simulate-plan' problem ( $\pi \# \pi$   $s$ )  $s = ($ 
  if enabled' problem  $\pi$   $s$  then
    let  $s = \text{execute}'$  problem  $\pi$   $s$  in
      simulate-plan' problem  $\pi$   $s$ 
  else
    None
)

```

Avoiding duplicate lookup.

lemma *simulate-plan'-code*[*code*]:

```

simulate-plan' problem []  $s = \text{Some } s$ 
simulate-plan' problem ( $\pi \# \pi$   $s$ )  $s = ($ 
  case lookup-operator' problem  $\pi$  of
    None  $\Rightarrow$  None
  | Some  $\pi \Rightarrow$ 
    if enabled-opr'  $\pi$   $s$  then
      simulate-plan' problem  $\pi$   $s$  (execute-opr'  $\pi$   $s$ )
    else None
)

```

by (*auto simp*: *enabled'-def execute'-def split*: *option.split*)

definition *initial-state'* :: $\text{ast-problem} \Rightarrow \text{state}$ **where**

initial-state' problem \equiv let *astI* = *ast-problem.astI* problem in (

)
 $\lambda v. \text{if } v < \text{length } \text{astI} \text{ then } \text{Some } (\text{astI}!v) \text{ else } \text{None}$
)

definition *check-plan'* :: *ast-problem* \Rightarrow *plan* \Rightarrow *bool* **where**
check-plan' *problem* $\pi s =$ (
 case *simulate-plan'* *problem* πs (*initial-state'* *problem*) of
 None \Rightarrow *False*
 | *Some s'* \Rightarrow *match-pres* (*ast-problem.astG* *problem*) *s'*)

context *wf-ast-problem*
begin

lemma *simulate-plan'-correct*:
assumes *s* \in *valid-states*
shows *simulate-plan'* *problem* πs *s* = *simulate-plan* πs *s*
using *assms*
apply (*induction* πs *s* *rule*: *simulate-plan.induct*)
apply (*auto simp*: *enabled'-correct execute'-correct execute-preserves-valid*)
done

lemma *simulate-plan'-correct-paper*:
assumes *s* \in *valid-states*
shows *simulate-plan'* *problem* πs *s* = *Some s'*
 \longleftrightarrow *path-to s* πs *s'*
using *simulate-plan'-correct*[*OF assms*] *simulate-plan-correct* **by** *simp*

lemma *initial-state'-correct*:
initial-state' *problem* = *I*
unfolding *initial-state'-def* *I-def* **by** (*auto simp*: *Let-def*)

lemma *check-plan'-correct*:
check-plan' *problem* πs = *check-plan* πs
proof –
have *D*: *distinct* (*map fst astG*) **using** *wf-goal* **unfolding** *wf-partial-state-def*
by *auto*

have *S'V*: *s'* \in *valid-states* **if** *simulate-plan* πs *I* = *Some s'* **for** *s'*
using *that* **by** (*auto simp*: *simulate-plan-correct path-to-pres-valid*[*OF I-valid*])

show *?thesis*
unfolding *check-plan'-def* *check-plan-def*
by (*auto*
split: *option.splits*
simp: *initial-state'-correct simulate-plan'-correct*[*OF I-valid*]
simp: *match-pres-correct*[*OF D S'V*] *G-def*
)

qed

end

definition *verify-plan* :: *ast-problem* \Rightarrow *plan* \Rightarrow *String.literal* + *unit* **where**
 verify-plan problem $\pi s =$ (
 if *ast-problem.well-formed problem* then
 if *check-plan' problem* πs then *Inr* () else *Inl* (*STR* "Invalid plan")
 else *Inl* (*STR* "Problem not well formed")
)

lemma *verify-plan-correct*:
 verify-plan problem $\pi s =$ *Inr* ()
 \longleftrightarrow *ast-problem.well-formed problem* \wedge *ast-problem.valid-plan problem* πs
proof –
 {
 assume *ast-problem.well-formed problem*
 then interpret *wf-ast-problem* **by** *unfold-locales*

 from *check-plan'-correct check-plan-correct*
 have *check-plan' problem* $\pi s =$ *valid-plan* πs **by** *simp*
 }
 then show ?*thesis*
 unfolding *verify-plan-def*
 by *auto*
qed

definition *nat-opt-of-integer* :: *integer* \Rightarrow *nat option* **where**
 nat-opt-of-integer i = (if ($i \geq 0$) then *Some* (*nat-of-integer i*) else *None*)

export-code *verify-plan nat-of-integer integer-of-nat nat-opt-of-integer Inl Inr*
String.explode String.implode
 in *SML*
 module-name *SASP-Checker-Exported*

end

3 PDDL and STRIPS Semantics

theory *PDDL-STRIPS-Semantics*
imports
 Propositional-Proof-Systems.Formulas
 Propositional-Proof-Systems.Sema
 Propositional-Proof-Systems.Consistency
 Automatic-Refinement.Misc
 Automatic-Refinement.Refine-Util

begin
no-notation *insert* (- ▷ - [56,55] 55)

3.1 Utility Functions

definition *index-by f l* \equiv *map-of* (*map* ($\lambda x. (f x, x)$) *l*)

lemma *index-by-eq-Some-eq[simp]*:
assumes *distinct* (*map f l*)
shows *index-by f l n = Some x* \longleftrightarrow (*x* \in *set l* \wedge *f x = n*)
unfolding *index-by-def*
using *assms*
by (*auto simp: o-def*)

lemma *index-by-eq-SomeD*:
shows *index-by f l n = Some x* \implies (*x* \in *set l* \wedge *f x = n*)
unfolding *index-by-def*
by (*auto dest: map-of-SomeD*)

lemma *lookup-zip-idx-eq*:
assumes *length params = length args*
assumes *i < length args*
assumes *distinct params*
assumes *k = params ! i*
shows *map-of* (*zip params args*) *k = Some* (*args ! i*)
using *assms*
by (*auto simp: in-set-conv-nth*)

lemma *rtrancl-image-idem[simp]*: $R^* \text{ `` } R^* \text{ `` } s = R^* \text{ `` } s$
by (*metis relcomp-Image rtrancl-idemp-self-comp*)

3.2 Abstract Syntax

3.2.1 Generic Entities

type-synonym *name* = *string*

datatype *predicate* = *Pred* (*name: name*)

Some of the AST entities are defined over a polymorphic *'val* type, which gets either instantiated by variables (for domains) or objects (for problems).

An atom is either a predicate with arguments, or an equality statement.

datatype *'ent atom* = *predAtm* (*predicate: predicate*) (*arguments: 'ent list*)
| *Eq* (*lhs: 'ent*) (*rhs: 'ent*)

A type is a list of primitive type names. To model a primitive type, we use a singleton list.

datatype *type* = *Either* (*primitives: name list*)

An effect contains a list of values to be added, and a list of values to be removed.

datatype *'ent ast-effect* = *Effect* (*adds*: ('ent atom formula) list) (*dels*: ('ent atom formula) list)

Variables are identified by their names.

datatype *variable* = *varname*: *Var name*

Objects and constants are identified by their names

datatype *object* = *name*: *Obj name*

datatype *term* = *VAR variable* | *CONST object*

hide-const (**open**) *VAR CONST* — Refer to constructors by qualified names only

3.2.2 Domains

An action schema has a name, a typed parameter list, a precondition, and an effect.

datatype *ast-action-schema* = *Action-Schema*
(*name*: *name*)
(*parameters*: (*variable* × *type*) list)
(*precondition*: *term atom formula*)
(*effect*: *term ast-effect*)

A predicate declaration contains the predicate's name and its argument types.

datatype *predicate-decl* = *PredDecl*
(*pred*: *predicate*)
(*argTs*: *type list*)

A domain contains the declarations of primitive types, predicates, and action schemas.

datatype *ast-domain* = *Domain*
(*types*: (*name* × *name*) list) — (*type*, *supertype*) declarations.
(*predicates*: *predicate-decl list*)
(*consts*: (*object* × *type*) list)
(*actions*: *ast-action-schema list*)

3.2.3 Problems

A fact is a predicate applied to objects.

type-synonym *fact* = *predicate* × *object list*

A problem consists of a domain, a list of objects, a description of the initial state, and a description of the goal state.

datatype *ast-problem* = *Problem*

(*domain: ast-domain*)
 (*objects: (object × type) list*)
 (*init: object atom formula list*)
 (*goal: object atom formula*)

3.2.4 Plans

datatype *plan-action* = *PAction*
 (*name: name*)
 (*arguments: object list*)

type-synonym *plan* = *plan-action list*

3.2.5 Ground Actions

The following datatype represents an action scheme that has been instantiated by replacing the arguments with concrete objects, also called ground action.

datatype *ground-action* = *Ground-Action*
 (*precondition: (object atom) formula*)
 (*effect: object ast-effect*)

3.3 Closed-World Assumption, Equality, and Negation

Discriminator for atomic predicate formulas.

fun *is-predAtom* **where**
is-predAtom (Atom (predAtm - -)) = True | is-predAtom - = False

The world model is a set of (atomic) formulas

type-synonym *world-model* = *object atom formula set*

It is basic, if it only contains atoms

definition *wm-basic* $M \equiv \forall a \in M. \text{is-predAtom } a$

A valuation extracted from the atoms of the world model

definition *valuation* :: *world-model* \Rightarrow *object atom valuation*
where *valuation* $M \equiv \lambda \text{predAtm } p \text{ } xs \Rightarrow \text{Atom } (\text{predAtm } p \text{ } xs) \in M \mid \text{Eq } a \text{ } b \Rightarrow a=b$

Augment a world model by adding negated versions of all atoms not contained in it, as well as interpretations of equality.

definition *close-world* :: *world-model* \Rightarrow *world-model* **where** *close-world* $M =$
 $M \cup \{\neg(\text{Atom } (\text{predAtm } p \text{ } as)) \mid p \text{ } as. \text{Atom } (\text{predAtm } p \text{ } as) \notin M\}$
 $\cup \{\text{Atom } (\text{Eq } a \text{ } a) \mid a. \text{True}\} \cup \{\neg(\text{Atom } (\text{Eq } a \text{ } b)) \mid a \text{ } b. a \neq b\}$

definition *close-neg* $M \equiv M \cup \{\neg(\text{Atom } a) \mid a. \text{Atom } a \notin M\}$

lemma *wm-basic* $M \implies \text{close-world } M = \text{close-neg } (M \cup \{\text{Atom } (Eq\ a\ a) \mid a.\text{True}\})$

unfolding *close-world-def close-neg-def wm-basic-def*
apply *clarsimp*
apply (*auto 0 3*)
by (*metis atom.exhaust*)

abbreviation *cw-entailment* (**infix** $^c \models =$ 53) **where**

$M ^c \models = \varphi \equiv \text{close-world } M \models \varphi$

lemma

close-world-extensive: $M \subseteq \text{close-world } M$ **and**
close-world-idem[*simp*]: $\text{close-world } (\text{close-world } M) = \text{close-world } M$
by (*auto simp: close-world-def*)

lemma *in-close-world-conv*:

$\varphi \in \text{close-world } M \longleftrightarrow$
 $\varphi \in M$
 $\vee (\exists p\ as.\ \varphi = \neg(\text{Atom } (\text{predAtm } p\ as)) \wedge \text{Atom } (\text{predAtm } p\ as) \notin M)$
 $\vee (\exists a.\ \varphi = \text{Atom } (Eq\ a\ a))$
 $\vee (\exists a\ b.\ \varphi = \neg(\text{Atom } (Eq\ a\ b)) \wedge a \neq b)$
 $)$
by (*auto simp: close-world-def*)

lemma *valuation-aux-1*:

fixes $M :: \text{world-model}$ **and** $\varphi :: \text{object atom formula}$
defines $C \equiv \text{close-world } M$
assumes $A: \forall \varphi \in C. \mathcal{A} \models \varphi$
shows $\mathcal{A} = \text{valuation } M$
using A **unfolding** $C\text{-def}$
apply –
apply (*auto simp: in-close-world-conv valuation-def Ball-def intro!: ext split: atom.split*)
apply (*metis formula-semantics.simps(1) formula-semantics.simps(3)*)
apply (*metis formula-semantics.simps(1) formula-semantics.simps(3)*)
by (*metis atom.collapse(2) formula-semantics.simps(1) is-predAtm-def*)

lemma *valuation-aux-2*:

assumes *wm-basic* M
shows $(\forall G \in \text{close-world } M. \text{valuation } M \models G)$
using *assms* **unfolding** *wm-basic-def*
by (*force simp: in-close-world-conv valuation-def elim: is-predAtom.elims*)

lemma *val-imp-close-world*: $\text{valuation } M \models \varphi \implies M ^c \models = \varphi$

unfolding *entailment-def*

using *valuation-aux-1*
by *blast*

lemma *close-world-imp-val*:
wm-basic $M \implies M \models \varphi \implies \text{valuation } M \models \varphi$
unfolding *entailment-def* **using** *valuation-aux-2* **by** *blast*

Main theorem of this section: If a world model M contains only atoms, its induced valuation satisfies a formula φ if and only if the closure of M entails φ .

Note that there are no syntactic restrictions on φ , in particular, φ may contain negation.

theorem *valuation-iff-close-world*:
assumes *wm-basic* M
shows $\text{valuation } M \models \varphi \iff M \models \varphi$
using *assms val-imp-close-world close-world-imp-val* **by** *blast*

3.3.1 Proper Generalization

Adding negation and equality is a proper generalization of the case without negation and equality

fun *is-STRIPS-fmla* :: '*ent atom formula* \implies *bool* **where**
is-STRIPS-fmla (*Atom* (*predAtm* - -)) \iff *True*
| *is-STRIPS-fmla* (\perp) \iff *True*
| *is-STRIPS-fmla* ($\varphi_1 \wedge \varphi_2$) \iff *is-STRIPS-fmla* $\varphi_1 \wedge$ *is-STRIPS-fmla* φ_2
| *is-STRIPS-fmla* ($\varphi_1 \vee \varphi_2$) \iff *is-STRIPS-fmla* $\varphi_1 \wedge$ *is-STRIPS-fmla* φ_2
| *is-STRIPS-fmla* ($\neg \perp$) \iff *True*
| *is-STRIPS-fmla* - \iff *False*

lemma *aux1*: $\llbracket \text{wm-basic } M; \text{is-STRIPS-fmla } \varphi; \text{valuation } M \models \varphi; \forall G \in M. \mathcal{A} \models G \rrbracket \implies \mathcal{A} \models \varphi$
apply (*induction* φ *rule: is-STRIPS-fmla.induct*)
by (*auto simp: valuation-def*)

lemma *aux2*: $\llbracket \text{wm-basic } M; \text{is-STRIPS-fmla } \varphi; \forall \mathcal{A}. (\forall G \in M. \mathcal{A} \models G) \longrightarrow \mathcal{A} \models \varphi \rrbracket \implies \text{valuation } M \models \varphi$
apply (*induction* φ *rule: is-STRIPS-fmla.induct*)
apply *simp-all*
apply (*metis in-close-world-conv valuation-aux-2*)
using *in-close-world-conv valuation-aux-2* **apply** *blast*
using *in-close-world-conv valuation-aux-2* **by** *auto*

lemma *valuation-iff-STRIPS*:
assumes *wm-basic* M
assumes *is-STRIPS-fmla* φ
shows $\text{valuation } M \models \varphi \iff M \models \varphi$
proof –

```

have aux1:  $\bigwedge \mathcal{A}. \llbracket \text{valuation } M \models \varphi; \forall G \in M. \mathcal{A} \models G \rrbracket \implies \mathcal{A} \models \varphi$ 
  using assms apply(induction  $\varphi$  rule: is-STRIPS-fmla.induct)
  by (auto simp: valuation-def)
have aux2:  $\llbracket \forall \mathcal{A}. (\forall G \in M. \mathcal{A} \models G) \longrightarrow \mathcal{A} \models \varphi \rrbracket \implies \text{valuation } M \models \varphi$ 
  using assms
  apply(induction  $\varphi$  rule: is-STRIPS-fmla.induct)
  apply simp-all
  apply (metis in-close-world-conv valuation-aux-2)
  using in-close-world-conv valuation-aux-2 apply blast
  using in-close-world-conv valuation-aux-2 by auto
show ?thesis
  by (auto simp: entailment-def intro: aux1 aux2)
qed

```

Our extension to negation and equality is a proper generalization of the standard STRIPS semantics for formula without negation and equality

theorem *proper-STRIPS-generalization:*
 $\llbracket \text{um-basic } M; \text{is-STRIPS-fmla } \varphi \rrbracket \implies M \models \varphi \iff M \models \varphi$
by (*simp add: valuation-iff-close-world[symmetric] valuation-iff-STRIPS*)

3.4 STRIPS Semantics

For this section, we fix a domain D , using Isabelle's locale mechanism.

```

locale ast-domain =
  fixes  $D :: \text{ast-domain}$ 
begin

```

It seems to be agreed upon that, in case of a contradictory effect, addition overrides deletion. We model this behaviour by first executing the deletions, and then the additions.

```

fun apply-effect :: object ast-effect  $\Rightarrow$  world-model  $\Rightarrow$  world-model
where
  apply-effect (Effect a d)  $s = (s - \text{set } d) \cup (\text{set } a)$ 

```

Execute a ground action

```

definition execute-ground-action :: ground-action  $\Rightarrow$  world-model  $\Rightarrow$  world-model
where
  execute-ground-action a M = apply-effect (effect a) M

```

Predicate to model that the given list of action instances is executable, and transforms an initial world model M into a final model M' .

Note that this definition over the list structure is more convenient in HOL than to explicitly define an indexed sequence $M_0 \dots M_N$ of intermediate world models, as done in [Lif87].

```

fun ground-action-path
  :: world-model  $\Rightarrow$  ground-action list  $\Rightarrow$  world-model  $\Rightarrow$  bool
where

```

```

    ground-action-path M [] M'  $\longleftrightarrow$  (M = M')
  | ground-action-path M ( $\alpha \# \alpha s$ ) M'  $\longleftrightarrow$  M  $\stackrel{c}{\models}$  precondition  $\alpha$ 
     $\wedge$  ground-action-path (execute-ground-action  $\alpha$  M)  $\alpha s$  M'

```

Function equations as presented in paper, with inlined *execute-ground-action*.

```

lemma ground-action-path-in-paper:
  ground-action-path M [] M'  $\longleftrightarrow$  (M = M')
  ground-action-path M ( $\alpha \# \alpha s$ ) M'  $\longleftrightarrow$  M  $\stackrel{c}{\models}$  precondition  $\alpha$ 
   $\wedge$  (ground-action-path (apply-effect (effect  $\alpha$ ) M)  $\alpha s$  M')
  by (auto simp: execute-ground-action-def)

```

end — Context of *ast-domain*

3.5 Well-Formedness of PDDL

```

fun ty-term where
  ty-term varT objT (term.VAR v) = varT v
  | ty-term varT objT (term.CONST c) = objT c

```

```

lemma ty-term-mono: varT  $\subseteq_m$  varT'  $\implies$  objT  $\subseteq_m$  objT'  $\implies$ 
  ty-term varT objT  $\subseteq_m$  ty-term varT' objT'
apply (rule map-leI)
subgoal for x v
  apply (cases x)
  apply (auto dest: map-leD)
done
done

```

context *ast-domain* **begin**

The signature is a partial function that maps the predicates of the domain to lists of argument types.

```

definition sig :: predicate  $\rightarrow$  type list where
  sig  $\equiv$  map-of (map ( $\lambda$ PredDecl p n  $\Rightarrow$  (p,n)) (predicates D))

```

We use a flat subtype hierarchy, where every type is a subtype of object, and there are no other subtype relations.

Note that we do not need to restrict this relation to declared types, as we will explicitly ensure that all types used in the problem are declared.

```

fun subtype-edge where
  subtype-edge (ty, superty) = (superty, ty)

```

```

definition subtype-rel  $\equiv$  set (map subtype-edge (types D))

```

definition *of-type* :: *type* \Rightarrow *type* \Rightarrow *bool* **where**
of-type *oT* *T* \equiv *set* (*primitives* *oT*) \subseteq *subtype-rel** “ *set* (*primitives* *T*)

This checks that every primitive on the LHS is contained in or a subtype of a primitive on the RHS

For the next few definitions, we fix a partial function that maps a polymorphic entity type *'e* to types. An entity can be instantiated by variables or objects later.

context

fixes *ty-ent* :: *'ent* \rightarrow *type* — Entity’s type, None if invalid

begin

Checks whether an entity has a given type

definition *is-of-type* :: *'ent* \Rightarrow *type* \Rightarrow *bool* **where**
is-of-type *v* *T* \longleftrightarrow (
case *ty-ent* *v* *of*
Some *vT* \Rightarrow *of-type* *vT* *T*
| *None* \Rightarrow *False*)

fun *wf-pred-atom* :: *predicate* \times *'ent* *list* \Rightarrow *bool* **where**
wf-pred-atom (*p,vs*) \longleftrightarrow (
case *sig* *p* *of*
None \Rightarrow *False*
| *Some* *Ts* \Rightarrow *list-all2* *is-of-type* *vs* *Ts*)

Predicate-atoms are well-formed if their arguments match the signature, equalities are well-formed if the arguments are valid objects (have a type).

TODO: We could check that types may actually overlap

fun *wf-atom* :: *'ent* *atom* \Rightarrow *bool* **where**
wf-atom (*predAtm* *p* *vs*) \longleftrightarrow *wf-pred-atom* (*p,vs*)
| *wf-atom* (*Eq* *a* *b*) \longleftrightarrow *ty-ent* *a* \neq *None* \wedge *ty-ent* *b* \neq *None*

A formula is well-formed if it consists of valid atoms, and does not contain negations, except for the encoding $\neg\perp$ of true.

fun *wf-fmla* :: (*'ent* *atom*) *formula* \Rightarrow *bool* **where**
wf-fmla (*Atom* *a*) \longleftrightarrow *wf-atom* *a*
| *wf-fmla* (\perp) \longleftrightarrow *True*
| *wf-fmla* ($\varphi1 \wedge \varphi2$) \longleftrightarrow (*wf-fmla* $\varphi1 \wedge$ *wf-fmla* $\varphi2$)
| *wf-fmla* ($\varphi1 \vee \varphi2$) \longleftrightarrow (*wf-fmla* $\varphi1 \wedge$ *wf-fmla* $\varphi2$)
| *wf-fmla* ($\neg\varphi$) \longleftrightarrow *wf-fmla* φ
| *wf-fmla* ($\varphi1 \rightarrow \varphi2$) \longleftrightarrow (*wf-fmla* $\varphi1 \wedge$ *wf-fmla* $\varphi2$)

lemma *wf-fmla* $\varphi = (\forall a \in \text{atoms } \varphi. \text{wf-atom } a)$
by (*induction* φ) *auto*

Special case for a well-formed atomic predicate formula

fun *wf-fmla-atom* **where**

$wf\text{-fmla-atom } (Atom (predAtm a vs)) \longleftrightarrow wf\text{-pred-atom } (a, vs)$
 $| wf\text{-fmla-atom } - \longleftrightarrow False$

lemma $wf\text{-fmla-atom-alt}$: $wf\text{-fmla-atom } \varphi \longleftrightarrow is\text{-predAtom } \varphi \wedge wf\text{-fmla } \varphi$
by (cases φ rule: $wf\text{-fmla-atom.cases}$) *auto*

An effect is well-formed if the added and removed formulas are atomic

fun $wf\text{-effect}$ **where**
 $wf\text{-effect } (Effect a d) \longleftrightarrow$
 $(\forall ae \in set\ a. wf\text{-fmla-atom } ae)$
 $\wedge (\forall de \in set\ d. wf\text{-fmla-atom } de)$

end — Context fixing $ty\text{-ent}$

definition $constT :: object \rightarrow type$ **where**
 $constT \equiv map\text{-of } (const\ D)$

An action schema is well-formed if the parameter names are distinct, and the precondition and effect is well-formed wrt. the parameters.

fun $wf\text{-action-schema} :: ast\text{-action-schema} \Rightarrow bool$ **where**
 $wf\text{-action-schema } (Action\text{-Schema } n\ params\ pre\ eff) \longleftrightarrow ($
 let
 $tyt = ty\text{-term } (map\text{-of } params)\ constT$
 in
 $distinct (map\ fst\ params)$
 $\wedge wf\text{-fmla } tyt\ pre$
 $\wedge wf\text{-effect } tyt\ eff)$

A type is well-formed if it consists only of declared primitive types, and the type object.

fun $wf\text{-type}$ **where**
 $wf\text{-type } (Either\ Ts) \longleftrightarrow set\ Ts \subseteq insert\ "object"\ (fst\ set\ (types\ D))$

A predicate is well-formed if its argument types are well-formed.

fun $wf\text{-predicate-decl}$ **where**
 $wf\text{-predicate-decl } (PredDecl\ p\ Ts) \longleftrightarrow (\forall T \in set\ Ts. wf\text{-type } T)$

The types declaration is well-formed, if all supertypes are declared types (or object)

definition $wf\text{-types} \equiv snd\ set\ (types\ D) \subseteq insert\ "object"\ (fst\ set\ (types\ D))$

A domain is well-formed if

- there are no duplicate declared predicate names,
- all declared predicates are well-formed,

- there are no duplicate action names,
- and all declared actions are well-formed

definition *wf-domain* :: *bool* **where**

wf-domain \equiv
wf-types
 \wedge *distinct* (*map* (*predicate-decl.pred*) (*predicates D*))
 \wedge ($\forall p \in \text{set}$ (*predicates D*). *wf-predicate-decl p*)
 \wedge *distinct* (*map fst* (*consts D*))
 \wedge ($\forall (n, T) \in \text{set}$ (*consts D*). *wf-type T*)
 \wedge *distinct* (*map ast-action-schema.name* (*actions D*))
 \wedge ($\forall a \in \text{set}$ (*actions D*). *wf-action-schema a*)

end — locale *ast-domain*

We fix a problem, and also include the definitions for the domain of this problem.

locale *ast-problem* = *ast-domain domain P*
for *P* :: *ast-problem*
begin

We refer to the problem domain as *D*

abbreviation *D* \equiv *ast-problem.domain P*

definition *objT* :: *object* \rightarrow *type* **where**
objT \equiv *map-of* (*objects P*) ++ *constT*

lemma *objT-alt*: *objT* = *map-of* (*consts D* @ *objects P*)
unfolding *objT-def constT-def*
apply (*clarsimp*)
done

definition *wf-fact* :: *fact* \Rightarrow *bool* **where**
wf-fact = *wf-pred-atom objT*

This definition is needed for well-formedness of the initial model, and forward-references to the concept of world model.

definition *wf-world-model* **where**
wf-world-model M = ($\forall f \in M$. *wf-fmla-atom objT f*)

definition *wf-problem* **where**

wf-problem \equiv
wf-domain
 \wedge *distinct* (*map fst* (*objects P*) @ *map fst* (*consts D*))
 \wedge ($\forall (n, T) \in \text{set}$ (*objects P*). *wf-type T*)

\wedge *distinct* (*init P*)
 \wedge *wf-world-model* (*set (init P)*)
 \wedge *wf-fmla objT* (*goal P*)

fun *wf-effect-inst* :: *object ast-effect* \Rightarrow *bool* **where**
wf-effect-inst (*Effect (a) (d)*)
 $\longleftrightarrow (\forall a \in \text{set } a \cup \text{set } d. \text{wf-fmla-atom } \text{objT } a)$

lemma *wf-effect-inst-alt*: *wf-effect-inst eff = wf-effect objT eff*
by (*cases eff*) *auto*

end — locale *ast-problem*

Locale to express a well-formed domain

locale *wf-ast-domain* = *ast-domain* +
assumes *wf-domain*: *wf-domain*

Locale to express a well-formed problem

locale *wf-ast-problem* = *ast-problem P* **for** *P* +
assumes *wf-problem*: *wf-problem*

begin

sublocale *wf-ast-domain domain P*
apply *unfold-locales*
using *wf-problem*
unfolding *wf-problem-def* **by** *simp*

end — locale *wf-ast-problem*

3.6 PDDL Semantics

context *ast-domain* **begin**

definition *resolve-action-schema* :: *name* \rightarrow *ast-action-schema* **where**
resolve-action-schema n = index-by ast-action-schema.name (actions D) n

fun *subst-term* **where**

subst-term psubst (term.VAR x) = psubst x
 $|$ *subst-term psubst (term.CONST c) = c*

To instantiate an action schema, we first compute a substitution from parameters to objects, and then apply this substitution to the precondition and effect. The substitution is applied via the *map-xxx* functions generated by the datatype package.

fun *instantiate-action-schema*

:: *ast-action-schema* \Rightarrow *object list* \Rightarrow *ground-action*

where

instantiate-action-schema (Action-Schema n params pre eff) args = (let

```

    tsubst = subst-term (the o (map-of (zip (map fst params) args)));
    pre-inst = (map-formula o map-atom) tsubst pre;
    eff-inst = (map-ast-effect) tsubst eff
  in
    Ground-Action pre-inst eff-inst
)

```

end — Context of *ast-domain*

context *ast-problem* **begin**

Initial model

definition *I* :: *world-model* **where**
I ≡ *set (init P)*

Resolve a plan action and instantiate the referenced action schema.

fun *resolve-instantiate* :: *plan-action* ⇒ *ground-action* **where**
resolve-instantiate (*PAction n args*) =
instantiate-action-schema
 (*the (resolve-action-schema n)*)
args

Check whether object has specified type

definition *is-obj-of-type n T* ≡ *case objT n of*
None ⇒ *False*
 | *Some oT* ⇒ *of-type oT T*

We can also use the generic *is-of-type* function.

lemma *is-obj-of-type-alt*: *is-obj-of-type* = *is-of-type objT*
apply (*intro ext*)
unfolding *is-obj-of-type-def is-of-type-def* **by** *auto*

HOL encoding of matching an action's formal parameters against an argument list. The parameters of the action are encoded as a list of *name × type* pairs, such that we map it to a list of types first. Then, the list relator *list-all2* checks that arguments and types have the same length, and each matching pair of argument and type satisfies the predicate *is-obj-of-type*.

definition *action-params-match a args*
 ≡ *list-all2 is-obj-of-type args (map snd (parameters a))*

At this point, we can define well-formedness of a plan action: The action must refer to a declared action schema, the arguments must be compatible with the formal parameters' types.

fun *wf-plan-action* :: *plan-action* ⇒ *bool* **where**
wf-plan-action (*PAction n args*) = (
case resolve-action-schema n of

```

None  $\Rightarrow$  False
| Some a  $\Rightarrow$ 
  action-params-match a args
   $\wedge$  wf-effect-inst (effect (instantiate-action-schema a args))
)

```

TODO: The second conjunct is redundant, as instantiating a well formed action with valid objects yield a valid effect.

A sequence of plan actions form a path, if they are well-formed and their instantiations form a path.

```

definition plan-action-path
  :: world-model  $\Rightarrow$  plan-action list  $\Rightarrow$  world-model  $\Rightarrow$  bool
where
  plan-action-path M  $\pi$ s M' =
    (( $\forall \pi \in$  set  $\pi$ s. wf-plan-action  $\pi$ )
      $\wedge$  ground-action-path M (map resolve-instantiate  $\pi$ s) M')

```

A plan is valid wrt. a given initial model, if it forms a path to a goal model

```

definition valid-plan-from :: world-model  $\Rightarrow$  plan  $\Rightarrow$  bool where
  valid-plan-from M  $\pi$ s = ( $\exists$  M'. plan-action-path M  $\pi$ s M'  $\wedge$  M'  $c\|=\|$  (goal P))

```

Finally, a plan is valid if it is valid wrt. the initial world model I

```

definition valid-plan :: plan  $\Rightarrow$  bool
where valid-plan  $\equiv$  valid-plan-from I

```

Concise definition used in paper:

```

lemma valid-plan  $\pi$ s  $\equiv$   $\exists$  M'. plan-action-path I  $\pi$ s M'  $\wedge$  M'  $c\|=\|$  (goal P)
unfolding valid-plan-def valid-plan-from-def by auto

```

end — Context of *ast-problem*

3.7 Preservation of Well-Formedness

3.7.1 Well-Formed Action Instances

The goal of this section is to establish that well-formedness of world models is preserved by execution of well-formed plan actions.

context *ast-problem* **begin**

As plan actions are executed by first instantiating them, and then executing the action instance, it is natural to define a well-formedness concept for action instances.

```

fun wf-ground-action :: ground-action  $\Rightarrow$  bool where
  wf-ground-action (Ground-Action pre eff)  $\longleftrightarrow$  (
    wf-fmla objT pre
  )

```

\wedge *wf-effect objT eff*
 $)$

We first prove that instantiating a well-formed action schema will yield a well-formed action instance.

We begin with some auxiliary lemmas before the actual theorem.

lemma (in *ast-domain*) *of-type-refl*[*simp, intro!*]: *of-type T T*
unfolding *of-type-def* **by** *auto*

lemma (in *ast-domain*) *of-type-trans*[*trans*]:
of-type T1 T2 \implies of-type T2 T3 \implies of-type T1 T3
unfolding *of-type-def*
by *clarsimp (metis (no-types, opaque-lifting)*
Image-mono contra-subsetD order-refl rtrancl-image-idem)

lemma *is-of-type-map-ofE*:
assumes *is-of-type (map-of params) x T*
obtains *i xT* **where** *i < length params params!i = (x,xT) of-type xT T*
using *assms*
unfolding *is-of-type-def*
by (*auto split: option.splits dest!: map-of-SomeD simp: in-set-conv-nth*)

lemma *wf-atom-mono*:
assumes *SS: tys \subseteq_m tys'*
assumes *WF: wf-atom tys a*
shows *wf-atom tys' a*
proof –
have *list-all2 (is-of-type tys') xs Ts if list-all2 (is-of-type tys) xs Ts for xs Ts*
using *that*
apply *induction*
by (*auto simp: is-of-type-def split: option.splits dest: map-leD[OF SS]*)
with *WF show ?thesis*
by (*cases a (auto split: option.splits dest: map-leD[OF SS])*)
qed

lemma *wf-fmla-atom-mono*:
assumes *SS: tys \subseteq_m tys'*
assumes *WF: wf-fmla-atom tys a*
shows *wf-fmla-atom tys' a*
proof –
have *list-all2 (is-of-type tys') xs Ts if list-all2 (is-of-type tys) xs Ts for xs Ts*
using *that*
apply *induction*
by (*auto simp: is-of-type-def split: option.splits dest: map-leD[OF SS]*)
with *WF show ?thesis*
by (*cases a rule: wf-fmla-atom.cases (auto split: option.splits dest: map-leD[OF SS])*)
qed

```

lemma constT-ss-objT: constT  $\subseteq_m$  objT
  unfolding constT-def objT-def
  apply rule
  by (auto simp: map-add-def split: option.split)

lemma wf-atom-constT-imp-objT: wf-atom (ty-term Q constT) a  $\implies$  wf-atom
(ty-term Q objT) a
  apply (erule wf-atom-mono[rotated])
  apply (rule ty-term-mono)
  by (simp-all add: constT-ss-objT)

lemma wf-fmla-atom-constT-imp-objT: wf-fmla-atom (ty-term Q constT) a  $\implies$ 
wf-fmla-atom (ty-term Q objT) a
  apply (erule wf-fmla-atom-mono[rotated])
  apply (rule ty-term-mono)
  by (simp-all add: constT-ss-objT)

context
  fixes Q and f :: variable  $\implies$  object
  assumes INST: is-of-type Q x T  $\implies$  is-of-type objT (f x) T
begin

  lemma is-of-type-var-conv: is-of-type (ty-term Q objT) (term.VAR x) T  $\longleftrightarrow$ 
is-of-type Q x T
    unfolding is-of-type-def by (auto)

  lemma is-of-type-const-conv: is-of-type (ty-term Q objT) (term.CONST x) T
 $\longleftrightarrow$  is-of-type objT x T
    unfolding is-of-type-def
    by (auto split: option.split)

  lemma INST': is-of-type (ty-term Q objT) x T  $\implies$  is-of-type objT (subst-term
f x) T
    apply (cases x) using INST apply (auto simp: is-of-type-var-conv is-of-type-const-conv)
    done

  lemma wf-inst-eq-aux: Q x = Some T  $\implies$  objT (f x)  $\neq$  None
    using INST[of x T] unfolding is-of-type-def
    by (auto split: option.splits)

  lemma wf-inst-eq-aux': ty-term Q objT x = Some T  $\implies$  objT (subst-term f x)
 $\neq$  None
    by (cases x) (auto simp: wf-inst-eq-aux)

lemma wf-inst-atom:

```

```

assumes wf-atom (ty-term Q constT) a
shows wf-atom objT (map-atom (subst-term f) a)
proof –
have X1: list-all2 (is-of-type objT) (map (subst-term f) xs) Ts if
  list-all2 (is-of-type (ty-term Q objT)) xs Ts for xs Ts
using that
apply induction
using INST'
by auto
then show ?thesis
using assms[THEN wf-atom-constT-imp-objT] wf-inst-eq-aux'
by (cases a; auto split: option.splits)

```

qed

lemma wf-inst-formula-atom:

```

assumes wf-fmla-atom (ty-term Q constT) a
shows wf-fmla-atom objT ((map-formula o map-atom o subst-term) f a)
using assms[THEN wf-fmla-atom-constT-imp-objT] wf-inst-atom
apply (cases a rule: wf-fmla-atom.cases; auto split: option.splits)
by (simp add: INST' list.rel-map(1) list-all2-mono)

```

lemma wf-inst-effect:

```

assumes wf-effect (ty-term Q constT)  $\varphi$ 
shows wf-effect objT ((map-ast-effect o subst-term) f  $\varphi$ )
using assms
proof (induction  $\varphi$ )
  case (Effect x1a x2a)
  then show ?case using wf-inst-formula-atom by auto
qed

```

lemma wf-inst-formula:

```

assumes wf-fmla (ty-term Q constT)  $\varphi$ 
shows wf-fmla objT ((map-formula o map-atom o subst-term) f  $\varphi$ )
using assms
by (induction  $\varphi$ ) (auto simp: wf-inst-atom dest: wf-inst-eq-aux)

```

end

Instantiating a well-formed action schema with compatible arguments will yield a well-formed action instance.

theorem wf-instantiate-action-schema:

```

assumes action-params-match a args
assumes wf-action-schema a
shows wf-ground-action (instantiate-action-schema a args)
proof (cases a)
case [simp]: (Action-Schema name params pre eff)
have INST:
  is-of-type objT ((the o map-of (zip (map fst params) args)) x) T

```

```

if is-of-type (map-of params) x T for x T
using that
apply (rule is-of-type-map-ofE)
using assms
apply (clarsimp simp: Let-def)
subgoal for i xT
  unfolding action-params-match-def
  apply (subst lookup-zip-idx-eq[where i=i];
    (clarsimp simp: list-all2-lengthD)?)
  apply (frule list-all2-nthD2[where p=i]; simp?)
  apply (auto
    simp: is-obj-of-type-alt is-of-type-def
    intro: of-type-trans
    split: option.splits)
  done
done
then show ?thesis
  using assms(2) wf-inst-formula wf-inst-effect
  by (fastforce split: term.splits simp: Let-def comp-apply[abs-def])
qed
end — Context of ast-problem

```

3.7.2 Preservation

context *ast-problem* **begin**

We start by defining two shorthands for enabledness and execution of a plan action.

Shorthand for enabled plan action: It is well-formed, and the precondition holds for its instance.

definition *plan-action-enabled* :: *plan-action* \Rightarrow *world-model* \Rightarrow *bool* **where**
plan-action-enabled π *M*
 \longleftrightarrow *wf-plan-action* $\pi \wedge M \stackrel{c}{\models} \text{precondition } (\text{resolve-instantiate } \pi)$

Shorthand for executing a plan action: Resolve, instantiate, and apply effect

definition *execute-plan-action* :: *plan-action* \Rightarrow *world-model* \Rightarrow *world-model*
where *execute-plan-action* π *M*
 $= (\text{apply-effect } (\text{effect } (\text{resolve-instantiate } \pi))$ *M*)

The *plan-action-path* predicate can be decomposed naturally using these shorthands:

lemma *plan-action-path-Nil[simp]*: *plan-action-path* *M [] M'* \longleftrightarrow *M'=M*
by (*auto simp: plan-action-path-def*)

lemma *plan-action-path-Cons[simp]*:
plan-action-path *M* ($\pi \# \pi s$) *M'* \longleftrightarrow
plan-action-enabled π *M*
 \wedge *plan-action-path* (*execute-plan-action* π *M*) πs *M'*

by (*auto*
simp: plan-action-path-def execute-plan-action-def
execute-ground-action-def plan-action-enabled-def)

end — Context of *ast-problem*

context *wf-ast-problem* **begin**

The initial world model is well-formed

lemma *wf-I: wf-world-model I*
using *wf-problem*
unfolding *I-def wf-world-model-def wf-problem-def*
apply(*safe*) **subgoal for** *f* **by** (*induction f*) *auto*
done

Application of a well-formed effect preserves well-formedness of the model

lemma *wf-apply-effect:*
assumes *wf-effect objT e*
assumes *wf-world-model s*
shows *wf-world-model (apply-effect e s)*
using *assms wf-problem*
unfolding *wf-world-model-def wf-problem-def wf-domain-def*
by (*cases e*) (*auto split: formula.splits prod.splits*)

Execution of plan actions preserves well-formedness

theorem *wf-execute:*
assumes *plan-action-enabled π s*
assumes *wf-world-model s*
shows *wf-world-model (execute-plan-action π s)*
using *assms*
proof (*cases π*)
case [*simp*]: (*PAction name args*)

from \langle *plan-action-enabled π s* \rangle **have** *wf-plan-action π*
unfolding *plan-action-enabled-def* **by** *auto*
then obtain a where
resolve-action-schema name = Some a **and**
T: action-params-match a args
by (*auto split: option.splits*)

from *wf-domain* **have**
[*simp*]: *distinct (map ast-action-schema.name (actions D))*
unfolding *wf-domain-def* **by** *auto*

from \langle *resolve-action-schema name = Some a* \rangle **have**
a \in set (actions D)
unfolding *resolve-action-schema-def* **by** *auto*


```

with wf-domain have wf-action-schema a
  unfolding wf-domain-def by auto
hence wf-ground-action (resolve-instantiate  $\pi$ )
  using  $\langle$ resolve-action-schema name = Some a $\rangle$  T
  wf-instantiate-action-schema
  by auto
thus ?thesis
  apply (simp add: execute-plan-action-def execute-ground-action-def)
  apply (rule wf-apply-effect)
  apply (cases resolve-instantiate  $\pi$ ; simp)
  by (rule  $\langle$ wf-world-model  $s$  $\rangle$ )
qed

```

```

theorem wf-execute-compact-notation:
  plan-action-enabled  $\pi$  s  $\implies$  wf-world-model s
   $\implies$  wf-world-model (execute-plan-action  $\pi$  s)
  by (rule wf-execute)

```

Execution of a plan preserves well-formedness

```

corollary wf-plan-action-path:
  assumes wf-world-model M and plan-action-path M  $\pi$  s M'
  shows wf-world-model M'
  using assms
  by (induction  $\pi$  s arbitrary: M) (auto intro: wf-execute)

```

end — Context of *wf-ast-problem*

end — Theory

4 Executable PDDL Checker

```

theory PDDL-STRIPS-Checker

```

```

imports

```

```

  PDDL-STRIPS-Semantics

```

```

  Error-Monad-Add

```

```

  HOL.String

```

```

  HOL-Library.Code-Target-Nat

```

```

  HOL-Library.While-Combinator

```

```

  Containers.Containers

```

```

begin

```

4.1 Generic DFS Reachability Checker

Used for subtype checks

definition *E-of-succ succ* $\equiv \{ (u,v). v \in \text{set } (\text{succ } u) \}$
lemma *succ-as-E*: $\text{set } (\text{succ } x) = E\text{-of-succ succ} \text{ `` } \{x\}$
unfolding *E-of-succ-def* **by** *auto*

context

fixes *succ* :: 'a \Rightarrow 'a list

begin

private abbreviation (*input*) *E* $\equiv E\text{-of-succ succ}$

definition *dfs-reachable D w* \equiv
 $\text{let } (V,w,brk) = \text{while } (\lambda(V,w,brk). \neg brk \wedge w \neq []) (\lambda(V,w,-).$
case w of v#w \Rightarrow
if D v then (V,v#w,True)
else if v \in V then (V,w,False)
else
let V = insert v V in
let w = succ v @ w in
(V,w,False)
) ({},w,False)
in brk

context

fixes *w₀* :: 'a list

assumes *finite-dfs-reachable[simp, intro!]*: *finite (E* `` set w₀)*

begin

private abbreviation (*input*) *W₀* $\equiv \text{set } w_0$

definition *dfs-reachable-invar D V W brk* \longleftrightarrow
 $W_0 \subseteq W \cup V$
 $\wedge W \cup V \subseteq E^* \text{ `` } W_0$
 $\wedge E \text{ `` } V \subseteq W \cup V$
 $\wedge \text{Collect } D \cap V = \{\}$
 $\wedge (brk \longrightarrow \text{Collect } D \cap E^* \text{ `` } W_0 \neq \{\})$

lemma *card-decreases*:

$\llbracket \text{finite } V; y \notin V; \text{dfs-reachable-invar } D V (\text{Set.insert } y W) brk \rrbracket$
 $\implies \text{card } (E^* \text{ `` } W_0 - \text{Set.insert } y V) < \text{card } (E^* \text{ `` } W_0 - V)$

apply (*rule psubset-card-mono*)

apply (*auto simp: dfs-reachable-invar-def*)

done

lemma *all-neq-Cons-is-Nil[simp]*:

$(\forall y \text{ ys. } x2 \neq y \# \text{ys}) \longleftrightarrow x2 = []$ **by** (*cases x2*) *auto*

lemma *dfs-reachable-correct*: *dfs-reachable D w₀ \longleftrightarrow Collect D \cap E* “ set w₀ \neq {}*

unfolding *dfs-reachable-def*
apply (*rule while-rule*[**where**
P= $\lambda(V,w,brk). \text{dfs-reachable-invar } D \ V \ (\text{set } w) \ brk \wedge \text{finite } V$
and *r=measure* ($\lambda V. \text{card } (E^* \text{ “ } (\text{set } w_0) - V)$) *<*lex*>* *measure length*
*<*lex*>* *measure* ($\lambda \text{True} \Rightarrow 0 \mid \text{False} \Rightarrow 1$)
])
subgoal by (*auto simp: dfs-reachable-invar-def*)
subgoal
apply (*auto simp: neq-Nil-conv succ-as-E[of succ] split: if-splits*)
by (*auto simp: dfs-reachable-invar-def Image-iff intro: rtrancl.rtrancl-into-rtrancl*)
subgoal by (*fastforce simp: dfs-reachable-invar-def dest: Image-closed-trancl*)
subgoal by *blast*
subgoal by (*auto simp: neq-Nil-conv card-decreases*)
done

end

definition *tab-succ* $l \equiv \text{Mapping.lookup-default } [] \ (\text{fold } (\lambda(u,v). \text{Mapping.map-default } u \ [] \ (\text{Cons } v)) \ l \ \text{Mapping.empty})$

lemma *Some-eq-map-option [iff]*: $(\text{Some } y = \text{map-option } f \ x_0) = (\exists z. x_0 = \text{Some } z \wedge f \ z = y)$

by (*auto simp add: map-option-case split: option.split*)

lemma *tab-succ-correct*: $E\text{-of-succ } (\text{tab-succ } l) = \text{set } l$

proof –

have $\text{set } (\text{Mapping.lookup-default } [] \ (\text{fold } (\lambda(u,v). \text{Mapping.map-default } u \ [] \ (\text{Cons } v)) \ l \ m) \ u) = \text{set } l \text{ “ } \{u\} \cup \text{set } (\text{Mapping.lookup-default } [] \ m \ u)$

for $m \ u$

apply (*induction l arbitrary: m*)

by (*auto*

simp: Mapping.lookup-default-def Mapping.map-default-def Mapping.default-def

simp: lookup-map-entry' lookup-update' keys-is-none-rep Option.is-none-def

split: if-splits

)

from *this*[**where** $m = \text{Mapping.empty}$] **show** *?thesis*

by (*auto simp: E-of-succ-def tab-succ-def lookup-default-empty*)

qed

end

lemma *finite-imp-finite-dfs-reachable*:

$[\text{finite } E; \text{finite } S] \Longrightarrow \text{finite } (E^* \text{ “ } S)$

```

apply (rule finite-subset[where B=S ∪ (Relation.Domain E ∪ Relation.Range
E)])
apply (auto simp: intro: finite-Domain finite-Range elim: rtranclE)
done

```

```

lemma dfs-reachable-tab-succ-correct: dfs-reachable (tab-succ l) D vs0 ⟷ Collect
D ∩ (set l)* “set vs0 ≠ {}
apply (subst dfs-reachable-correct)
by (simp-all add: tab-succ-correct finite-imp-finite-dfs-reachable)

```

4.2 Implementation Refinements

4.2.1 Of-Type

```

definition of-type-impl G oT T ≡ (∀ pt∈set (primitives oT). dfs-reachable G ((=)
pt) (primitives T))

```

```

fun ty-term' where
  ty-term' varT objT (term.VAR v) = varT v
| ty-term' varT objT (term.CONST c) = Mapping.lookup objT c

```

```

lemma ty-term'-correct-aux: ty-term' varT objT t = ty-term varT (Mapping.lookup
objT) t
by (cases t) auto

```

```

lemma ty-term'-correct[simp]: ty-term' varT objT = ty-term varT (Mapping.lookup
objT)
using ty-term'-correct-aux by auto

```

```

context ast-domain begin

```

```

definition of-type1 pt T ⟷ pt ∈ subtype-rel* “ set (primitives T)

```

```

lemma of-type-refine1: of-type oT T ⟷ (∀ pt∈set (primitives oT). of-type1 pt
T)
unfolding of-type-def of-type1-def by auto

```

```

definition STG ≡ (tab-succ (map subtype-edge (types D)))

```

```

lemma subtype-rel-impl: subtype-rel = E-of-succ (tab-succ (map subtype-edge
(types D)))
by (simp add: tab-succ-correct subtype-rel-def)

```

```

lemma of-type1-impl: of-type1 pt T ⟷ dfs-reachable (tab-succ (map subtype-edge
(types D))) ((=)pt) (primitives T)
by (simp add: subtype-rel-impl of-type1-def dfs-reachable-tab-succ-correct tab-succ-correct)

```

```

lemma of-type-impl-correct: of-type-impl STG oT T ⟷ of-type oT T
unfolding of-type1-impl STG-def of-type-impl-def of-type-refine1 ..

```

definition $mp\text{-}constT :: (object, type) \text{ mapping where}$

$mp\text{-}constT = \text{Mapping.of-alist (consts D)}$

lemma $mp\text{-}objT\text{-correct}[simp]: \text{Mapping.lookup } mp\text{-}constT = constT$

unfolding $mp\text{-}constT\text{-def } constT\text{-def}$

by transfer ($simp \text{ add: Map-To-Mapping.map-apply-def}$)

Lifting the subtype-graph through wf-checker

context

fixes $ty\text{-}ent :: 'ent \rightarrow type$ — Entity's type, None if invalid

begin

definition $is\text{-}of\text{-}type' \text{ stg } v \text{ } T \longleftrightarrow ($

$\text{case } ty\text{-}ent \text{ } v \text{ of}$

$\text{Some } vT \Rightarrow \text{of-type-impl stg } vT \text{ } T$

$| \text{None} \Rightarrow \text{False}$)

lemma $is\text{-}of\text{-}type'\text{-correct}: is\text{-}of\text{-}type' \text{ } STG \text{ } v \text{ } T = is\text{-}of\text{-}type \text{ } ty\text{-}ent \text{ } v \text{ } T$

unfolding $is\text{-}of\text{-}type'\text{-def } is\text{-}of\text{-}type\text{-def } of\text{-}type\text{-impl-correct} \dots$

fun $wf\text{-}pred\text{-}atom' \text{ where } wf\text{-}pred\text{-}atom' \text{ stg } (p, vs) \longleftrightarrow (\text{case sig } p \text{ of}$

$\text{None} \Rightarrow \text{False}$

$| \text{Some } Ts \Rightarrow \text{list-all2 (is-of-type' stg) vs Ts}$)

lemma $wf\text{-}pred\text{-}atom'\text{-correct}: wf\text{-}pred\text{-}atom' \text{ } STG \text{ } pvs = wf\text{-}pred\text{-}atom \text{ } ty\text{-}ent$

pvs

by ($\text{cases } pvs$) ($\text{auto simp: is-of-type'\text{-correct}[abs-def] split.option.split}$)

fun $wf\text{-}atom' :: - \Rightarrow 'ent \text{ atom} \Rightarrow \text{bool where}$

$wf\text{-}atom' \text{ stg } (atom.predAtm \text{ } p \text{ } vs) \longleftrightarrow wf\text{-}pred\text{-}atom' \text{ stg } (p, vs)$

$| wf\text{-}atom' \text{ stg } (atom.Eq \text{ } a \text{ } b) = (ty\text{-}ent \text{ } a \neq \text{None} \wedge ty\text{-}ent \text{ } b \neq \text{None})$

lemma $wf\text{-}atom'\text{-correct}: wf\text{-}atom' \text{ } STG \text{ } a = wf\text{-}atom \text{ } ty\text{-}ent \text{ } a$

by ($\text{cases } a$) ($\text{auto simp: wf-pred-atom'\text{-correct} is-of-type'\text{-correct}[abs-def] split.option.splits}$)

fun $wf\text{-}fmla' :: - \Rightarrow ('ent \text{ atom}) \text{ formula} \Rightarrow \text{bool where}$

$wf\text{-}fmla' \text{ stg } (Atom \text{ } a) \longleftrightarrow wf\text{-}atom' \text{ stg } a$

$| wf\text{-}fmla' \text{ stg } \perp \longleftrightarrow \text{True}$

$| wf\text{-}fmla' \text{ stg } (\varphi1 \wedge \varphi2) \longleftrightarrow (wf\text{-}fmla' \text{ stg } \varphi1 \wedge wf\text{-}fmla' \text{ stg } \varphi2)$

$| wf\text{-}fmla' \text{ stg } (\varphi1 \vee \varphi2) \longleftrightarrow (wf\text{-}fmla' \text{ stg } \varphi1 \wedge wf\text{-}fmla' \text{ stg } \varphi2)$

$| wf\text{-}fmla' \text{ stg } (\varphi1 \rightarrow \varphi2) \longleftrightarrow (wf\text{-}fmla' \text{ stg } \varphi1 \wedge wf\text{-}fmla' \text{ stg } \varphi2)$

$| wf\text{-}fmla' \text{ stg } (\neg\varphi) \longleftrightarrow wf\text{-}fmla' \text{ stg } \varphi$

lemma $wf\text{-}fmla'\text{-correct}: wf\text{-}fmla' \text{ } STG \text{ } \varphi \longleftrightarrow wf\text{-}fmla \text{ } ty\text{-}ent \text{ } \varphi$

by ($\text{induction } \varphi \text{ rule: wf-fmla.induct}$) ($\text{auto simp: wf-atom'\text{-correct}}$)

fun $wf\text{-}fmla\text{-}atom1' \text{ where}$

$wf\text{-fmla-atom1}'\ stg\ (Atom\ (predAtm\ p\ vs)) \longleftrightarrow wf\text{-pred-atom}'\ stg\ (p,vs)$
 $| wf\text{-fmla-atom1}'\ stg - \longleftrightarrow False$

lemma $wf\text{-fmla-atom1}'\text{-correct}$: $wf\text{-fmla-atom1}'\ STG\ \varphi = wf\text{-fmla-atom}\ ty\text{-ent}$

φ

by ($cases\ \varphi\ rule: wf\text{-fmla-atom}.cases$) ($auto$
 $simp: wf\text{-atom}'\text{-correct}\ is\text{-of}\text{-type}'\text{-correct}[abs\text{-def}]\ split: option.splits$)

fun $wf\text{-effect}'\ where$

$wf\text{-effect}'\ stg\ (Effect\ a\ d) \longleftrightarrow$
 $(\forall ae \in set\ a. wf\text{-fmla-atom1}'\ stg\ ae)$
 $\wedge (\forall de \in set\ d. wf\text{-fmla-atom1}'\ stg\ de)$

lemma $wf\text{-effect}'\text{-correct}$: $wf\text{-effect}'\ STG\ e = wf\text{-effect}\ ty\text{-ent}\ e$

by ($cases\ e$) ($auto\ simp: wf\text{-fmla-atom1}'\text{-correct}$)

end — Context fixing $ty\text{-ent}$

fun $wf\text{-action-schema}' :: - \Rightarrow - \Rightarrow ast\text{-action-schema} \Rightarrow bool\ where$

$wf\text{-action-schema}'\ stg\ conT\ (Action\text{-Schema}\ n\ params\ pre\ eff) \longleftrightarrow ($
 let
 $tyv = ty\text{-term}'\ (map\text{-of}\ params)\ conT$
 in
 $distinct\ (map\ fst\ params)$
 $\wedge wf\text{-fmla}'\ tyv\ stg\ pre$
 $\wedge wf\text{-effect}'\ tyv\ stg\ eff)$

lemma $wf\text{-action-schema}'\text{-correct}$: $wf\text{-action-schema}'\ STG\ mp\text{-const}\ T\ s = wf\text{-action-schema}$

s

by ($cases\ s$) ($auto\ simp: wf\text{-fmla}'\text{-correct}\ wf\text{-effect}'\text{-correct}$)

definition $wf\text{-domain}' :: - \Rightarrow - \Rightarrow bool\ where$

$wf\text{-domain}'\ stg\ conT \equiv$
 $wf\text{-types}$
 $\wedge distinct\ (map\ (predicate\text{-decl}.pred)\ (predicates\ D))$
 $\wedge (\forall p \in set\ (predicates\ D). wf\text{-predicate}\text{-decl}\ p)$
 $\wedge distinct\ (map\ fst\ (consts\ D))$
 $\wedge (\forall (n, T) \in set\ (consts\ D). wf\text{-type}\ T)$
 $\wedge distinct\ (map\ ast\text{-action}\text{-schema}.name\ (actions\ D))$
 $\wedge (\forall a \in set\ (actions\ D). wf\text{-action}\text{-schema}'\ stg\ conT\ a)$

lemma $wf\text{-domain}'\text{-correct}$: $wf\text{-domain}'\ STG\ mp\text{-const}\ T = wf\text{-domain}$

unfolding $wf\text{-domain}\text{-def}\ wf\text{-domain}'\text{-def}$
by ($auto\ simp: wf\text{-action}\text{-schema}'\text{-correct}$)

end — Context of $ast\text{-domain}$

4.2.2 Application of Effects

context *ast-domain* **begin**

We implement the application of an effect by explicit iteration over the additions and deletions

```
fun apply-effect-exec
  :: object ast-effect  $\Rightarrow$  world-model  $\Rightarrow$  world-model
where
  apply-effect-exec (Effect a d) s
    = fold ( $\lambda$ add s. Set.insert add s) a
      (fold ( $\lambda$ del s. Set.remove del s) d s)

lemma apply-effect-exec-refine[simp]:
  apply-effect-exec (Effect (a) (d)) s
    = apply-effect (Effect (a) (d)) s
proof (induction a arbitrary: s)
  case Nil
  then show ?case
proof (induction d arbitrary: s)
  case Nil
  then show ?case by auto
next
  case (Cons a d)
  then show ?case
    by (auto simp add: image-def)
qed
next
  case (Cons a a)
  then show ?case
proof (induction d arbitrary: s)
  case Nil
  then show ?case by (auto; metis Set.insert-def sup-assoc insert-iff)
next
  case (Cons a d)
  then show ?case
    by (auto simp: Un-commute minus-set-fold union-set-fold)
qed
qed
```

```
lemmas apply-effect-eq-impl-eq
  = apply-effect-exec-refine[symmetric, unfolded apply-effect-exec.simps]
```

end — Context of *ast-domain*

4.2.3 Well-Formedness

context *ast-problem* **begin**

We start by defining a mapping from objects to types. The container frame-

work will generate efficient, red-black tree based code for that later.

type-synonym $objT = (object, type) mapping$

definition $mp-objT :: (object, type) mapping$ **where**
 $mp-objT = Mapping.of-alist (consts D @ objects P)$

lemma $mp-objT-correct[simp]$: $Mapping.lookup mp-objT = objT$
unfolding $mp-objT-def objT-alt$
by transfer ($simp add: Map-To-Mapping.map-apply-def$)

We refine the typecheck to use the mapping

definition $is-obj-of-type-impl stg mp n T = ($
 $case Mapping.lookup mp n of None \Rightarrow False \mid Some oT \Rightarrow of-type-impl stg oT$
 T
 $)$

lemma $is-obj-of-type-impl-correct[simp]$:
 $is-obj-of-type-impl STG mp-objT = is-obj-of-type$
apply ($intro ext$)
apply ($auto simp: is-obj-of-type-impl-def is-obj-of-type-def of-type-impl-correct$
 $split: option.split$)
done

We refine the well-formedness checks to use the mapping

definition $wf-fact' :: objT \Rightarrow - \Rightarrow fact \Rightarrow bool$
where
 $wf-fact' ot stg \equiv wf-pred-atom' (Mapping.lookup ot) stg$

lemma $wf-fact'-correct[simp]$: $wf-fact' mp-objT STG = wf-fact$
by ($auto simp: wf-fact'-def wf-fact-def wf-pred-atom'-correct[abs-def]$)

definition $wf-fmla-atom2' mp stg f$
 $= (case f of formula.Atom (predAtm p vs) \Rightarrow (wf-fact' mp stg (p,vs)) \mid - \Rightarrow$
 $False)$

lemma $wf-fmla-atom2'-correct[simp]$:
 $wf-fmla-atom2' mp-objT STG \varphi = wf-fmla-atom objT \varphi$
apply ($cases \varphi rule: wf-fmla-atom.cases$)
by ($auto simp: wf-fmla-atom2'-def wf-fact-def split: option.splits$)

definition $wf-problem' stg conT mp \equiv$
 $wf-domain' stg conT$
 $\wedge distinct (map fst (objects P) @ map fst (consts D))$
 $\wedge (\forall (n, T) \in set (objects P). wf-type T)$
 $\wedge distinct (init P)$
 $\wedge (\forall f \in set (init P). wf-fmla-atom2' mp stg f)$
 $\wedge wf-fmla' (Mapping.lookup mp) stg (goal P)$

lemma *wf-problem'-correct*:
wf-problem' STG mp-constT mp-objT = wf-problem
unfolding *wf-problem-def wf-problem'-def wf-world-model-def*
by (*auto simp: wf-domain'-correct wf-fmla'-correct*)

Instantiating actions will yield well-founded effects. Corollary of $\llbracket \text{action-params-match } ?a \text{ } ?args; \text{wf-action-schema } ?a \rrbracket \implies \text{wf-ground-action } (\text{instantiate-action-schema } ?a \text{ } ?args)$.

lemma *wf-effect-inst-weak*:
fixes *a args*
defines *ai* \equiv *instantiate-action-schema a args*
assumes *A*: *action-params-match a args*
wf-action-schema a
shows *wf-effect-inst (effect ai)*
using *wf-instantiate-action-schema[OF A]* **unfolding** *ai-def[symmetric]*
by (*cases ai*) (*auto simp: wf-effect-inst-alt*)

end — Context of *ast-problem*

context *wf-ast-domain* **begin**

Resolving an action yields a well-founded action schema.

lemma *resolve-action-wf*:
assumes *resolve-action-schema n = Some a*
shows *wf-action-schema a*
proof –
from *wf-domain* **have**
X1: distinct (map ast-action-schema.name (actions D))
and *X2: $\forall a \in \text{set } (\text{actions } D). \text{wf-action-schema } a$*
unfolding *wf-domain-def* **by** *auto*

show *?thesis*
using *assms* **unfolding** *resolve-action-schema-def*
by (*auto simp add: index-by-eq-Some-eq[OF X1] X2*)
qed

end — Context of *ast-domain*

4.2.4 Execution of Plan Actions

We will perform two refinement steps, to summarize redundant operations

We first lift action schema lookup into the error monad.

context *ast-domain* **begin**
definition *resolve-action-schemaE n* \equiv
lift-opt

```

      (resolve-action-schema n)
      (ERR (shows "No such action schema " o shows n))
end — Context of ast-domain

```

context *ast-problem* **begin**

We define a function to determine whether a formula holds in a world model

definition *holds* $M F \equiv (\text{valuation } M) \models F$

Justification of this function

lemma *holds-for-wf-fmlas*:
assumes *wm-basic* s
shows *holds* $s F \iff \text{close-world } s \models F$
unfolding *holds-def* **using** *assms valuation-iff-close-world*
by *blast*

The first refinement summarizes the enabledness check and the execution of the action. Moreover, we implement the precondition evaluation by our *holds* function. This way, we can eliminate redundant resolving and instantiation of the action.

definition *en-exE* :: *plan-action* \Rightarrow *world-model* \Rightarrow $-+$ *world-model* **where**
en-exE $\equiv \lambda(PAction\ n\ args) \Rightarrow \lambda s. \text{do } \{$
 $a \leftarrow \text{resolve-action-schemaE } n;$
 $\text{check } (\text{action-params-match } a\ args) (\text{ERRS "Parameter mismatch"});$
 $\text{let } ai = \text{instantiate-action-schema } a\ args;$
 $\text{check } (\text{wf-effect-inst } (\text{effect } ai)) (\text{ERRS "Effect not well-formed"});$
 $\text{check } (\text{holds } s (\text{precondition } ai)) (\text{ERRS "Precondition not satisfied"});$
 $\text{Error-Monad.return } (\text{apply-effect } (\text{effect } ai) s)$
 $\}$

Justification of implementation.

lemma (**in** *wf-ast-problem*) *en-exE-return-iff*:
assumes *wm-basic* s
shows *en-exE* $a\ s = \text{Inr } s'$
 $\iff \text{plan-action-enabled } a\ s \wedge s' = \text{execute-plan-action } a\ s$
apply (*cases* a)
using *assms holds-for-wf-fmlas wf-domain*
unfolding *plan-action-enabled-def execute-plan-action-def*
and *execute-ground-action-def en-exE-def wf-domain-def*
by (*auto*
 $\text{split: option.splits}$
 $\text{simp: resolve-action-schemaE-def return-iff}$)

Next, we use the efficient implementation *is-obj-of-type-impl* for the type check, and omit the well-formedness check, as effects obtained from instantiating well-formed action schemas are always well-formed (*wf-effect-inst-weak*).

abbreviation *action-params-match2* $\text{stg } mp\ a\ args$

\equiv *list-all2* (*is-obj-of-type-impl stg mp*)
args (*map snd* (*ast-action-schema.parameters a*))

definition *en-exE2*

$:: - \Rightarrow$ (*object, type*) *mapping* \Rightarrow *plan-action* \Rightarrow *world-model* \Rightarrow $-+$ *world-model*

where

en-exE2 G mp \equiv $\lambda(PAction\ n\ args) \Rightarrow \lambda M. do$ {
a \leftarrow *resolve-action-schemaE n*;
check (*action-params-match2 G mp a args*) (*ERRS "Parameter mismatch"*);
let ai = *instantiate-action-schema a args*;
check (*holds M (precondition ai)*) (*ERRS "Precondition not satisfied"*);
Error-Monad.return (*apply-effect* (*effect ai*) *M*)
}

Justification of refinement

lemma (*in wf-ast-problem*) *wf-en-exE2-eq*:

shows *en-exE2 STG mp-objT pa s* = *en-exE pa s*

apply (*cases pa*; *simp add: en-exE2-def en-exE-def Let-def*)

apply (*auto*

simp: return-iff resolve-action-schemaE-def resolve-action-wf

simp: wf-effect-inst-weak action-params-match-def

split: error-monad-bind-split)

done

Combination of the two refinement lemmas

lemma (*in wf-ast-problem*) *en-exE2-return-iff*:

assumes *wm-basic M*

shows *en-exE2 STG mp-objT a M* = *Inr M'*

\longleftrightarrow *plan-action-enabled a M* \wedge *M'* = *execute-plan-action a M*

unfolding *wf-en-exE2-eq*

apply (*subst en-exE-return-iff*)

using *assms*

by (*auto*)

lemma (*in wf-ast-problem*) *en-exE2-return-iff-compact-notation*:

$\llbracket wm-basic\ s \rrbracket \implies$

en-exE2 STG mp-objT a s = *Inr s'*

\longleftrightarrow *plan-action-enabled a s* \wedge *s'* = *execute-plan-action a s*

using *en-exE2-return-iff* .

end — Context of *ast-problem*

4.2.5 Checking of Plan

context *ast-problem begin*

First, we combine the well-formedness check of the plan actions and their execution into a single iteration.

fun *valid-plan-from1* $:: world-model \Rightarrow plan \Rightarrow bool$ **where**

```

  valid-plan-from1 s []  $\longleftrightarrow$  close-world s  $\models$  (goal P)
| valid-plan-from1 s ( $\pi \# \pi s$ )
   $\longleftrightarrow$  plan-action-enabled  $\pi$  s
     $\wedge$  (valid-plan-from1 (execute-plan-action  $\pi$  s)  $\pi s$ )

```

lemma *valid-plan-from1-refine*: *valid-plan-from* s πs = *valid-plan-from1* s πs

proof (*induction* πs *arbitrary*: s)

case *Nil*

then show ?*case* **by** (*auto simp add: plan-action-path-def valid-plan-from-def*)

next

case (*Cons* a πs)

then show ?*case*

by (*auto*

simp: valid-plan-from-def plan-action-path-def plan-action-enabled-def

simp: execute-ground-action-def execute-plan-action-def)

qed

Next, we use our efficient combined enabledness check and execution function, and transfer the implementation to use the error monad:

fun *valid-plan-fromE*

:: - \Rightarrow (*object, type*) *mapping* \Rightarrow *nat* \Rightarrow *world-model* \Rightarrow *plan* \Rightarrow *+*unit

where

valid-plan-fromE *stg mp si s* []

= *check* (*holds* s (*goal* P)) (*ERRS* "*Postcondition does not hold*")

| *valid-plan-fromE* *stg mp si s* ($\pi \# \pi s$) = *do* {

s \leftarrow *en-exE2* *stg mp* π s

\leftarrow ? (λe -. *shows* "*at step* " *o* *shows* *si* *o* *shows* "': " *o* *e* ());

valid-plan-fromE *stg mp* (*si*+1) s πs

}

For the refinement, we need to show that the world models only contain atoms, i.e., containing only atoms is an invariant under execution of well-formed plan actions.

lemma (*in* *wf-ast-problem*) *wf-actions-only-add-atoms*:

[*wm-basic* s; *wf-plan-action* a]

\implies *wm-basic* (*execute-plan-action* a s)

using *wf-problem wf-domain*

unfolding *wf-problem-def wf-domain-def*

apply (*cases* a)

apply (*clarsimp*

split: option.splits

simp: wf-fmla-atom-alt execute-plan-action-def wm-basic-def

simp: execute-ground-action-def)

subgoal for *n args schema fmla*

apply (*cases effect* (*instantiate-action-schema* *schema args*); *simp*)

by (*metis* *ground-action.sel*(2) *ast-domain.wf-effect.simps*

ast-domain.wf-fmla-atom-alt resolve-action-wf

wf-ground-action.elims(2) *wf-instantiate-action-schema*)

done

Refinement lemma for our plan checking algorithm

```

lemma (in wf-ast-problem) valid-plan-fromE-return-iff[return-iff]:
  assumes wm-basic s
  shows valid-plan-fromE STG mp-objT k s  $\pi$  s = Inr ()  $\longleftrightarrow$  valid-plan-from s
 $\pi$  s
  using assms unfolding valid-plan-from1-refine
proof (induction stg $\equiv$ STG mp $\equiv$ mp-objT k s  $\pi$  s rule: valid-plan-fromE.induct)
  case (1 si s)
  then show ?case
    using wf-problem holds-for-wf-fmlas
    by (auto
      simp: return-iff Let-def wf-en-exE2-eq wf-problem-def
      split: plan-action.split)
  next
  case (2 si s  $\pi$   $\pi$  s)
  then show ?case
    apply (clarsimp
      simp: return-iff en-exE2-return-iff
      split: plan-action.split)
    by (meson ast-problem.plan-action-enabled-def wf-actions-only-add-atoms)
qed

lemmas valid-plan-fromE-return-iff'[return-iff]
  = wf-ast-problem.valid-plan-fromE-return-iff[of P, OF wf-ast-problem.intro]

```

end — Context of *ast-problem*

4.3 Executable Plan Checker

We obtain the main plan checker by combining the well-formedness check and executability check.

```

definition check-all-list P l msg msgf  $\equiv$ 
  forallM ( $\lambda$ x. check (P x) ( $\lambda$ :::unit. shows msg o shows "': " o msgf x) ) l <+?
  snd

```

```

lemma check-all-list-return-iff[return-iff]: check-all-list P l msg msgf = Inr ()  $\longleftrightarrow$ 
  ( $\forall$  x $\in$ set l. P x)
  unfolding check-all-list-def
  by (induction l) (auto)

```

```

definition check-wf-types D  $\equiv$  do {
  check-all-list ( $\lambda$ (-,t). t="object"  $\vee$  t $\in$ fst'set (types D)) (types D) "Undeclared
  supertype" (shows o snd)
}

```

lemma *check-wf-types-return-iff*[*return-iff*]: *check-wf-types* $D = \text{Inr } () \longleftrightarrow \text{ast-domain.wf-types } D$

unfolding *ast-domain.wf-types-def* *check-wf-types-def*
by (*force simp: return-iff*)

definition *check-wf-domain* D *stg conT* \equiv *do* {
check-wf-types D ;
check (*distinct* (*map* (*predicate-decl.pred*) (*predicates* D))) (*ERRS* "*Duplicate predicate declaration*");
check-all-list (*ast-domain.wf-predicate-decl* D) (*predicates* D) "*Malformed predicate declaration*" (*shows* *o* *predicate.name* *o* *predicate-decl.pred*);
check (*distinct* (*map* *fst* (*consts* D))) (*ERRS* "*Duplicate constant declaration*");
check ($\forall (n, T) \in \text{set } (\text{consts } D). \text{ast-domain.wf-type } D \ T$) (*ERRS* "*Malformed type*");
check (*distinct* (*map* *ast-action-schema.name* (*actions* D))) (*ERRS* "*Duplicate action name*");
check-all-list (*ast-domain.wf-action-schema'* D *stg conT*) (*actions* D) "*Malformed action*" (*shows* *o* *ast-action-schema.name*)
}

lemma *check-wf-domain-return-iff*[*return-iff*]:

check-wf-domain D *stg conT* $= \text{Inr } () \longleftrightarrow \text{ast-domain.wf-domain}' D$ *stg conT*

proof –

interpret *ast-domain* D .

show *?thesis*

unfolding *check-wf-domain-def* *wf-domain'-def*

by (*auto simp: return-iff*)

qed

definition *prepend-err-msg* $\text{msg } e \equiv \lambda::\text{unit}. \text{shows } \text{msg } \text{ o } \text{shows } '' : '' \text{ o } e$ ($()$)

definition *check-wf-problem* P *stg conT mp* \equiv *do* {

let $D = \text{ast-problem.domain } P$;

check-wf-domain D *stg conT* $<+ ? \text{prepend-err-msg } ''\text{Domain not well-formed}''$;

check (*distinct* (*map* *fst* (*objects* P) @ *map* *fst* (*consts* D))) (*ERRS* "*Duplicate object declaration*");

check ($(\forall (n, T) \in \text{set } (\text{objects } P). \text{ast-domain.wf-type } D \ T)$) (*ERRS* "*Malformed type*");

check (*distinct* (*init* P)) (*ERRS* "*Duplicate fact in initial state*");

check ($\forall f \in \text{set } (\text{init } P). \text{ast-problem.wf-fmla-atom2}' P \ mp \ \text{stg } f$) (*ERRS* "*Malformed formula in initial state*");

check (*ast-domain.wf-fmla'* D (*Mapping.lookup* mp) *stg* (*goal* P)) (*ERRS* "*Malformed goal formula*");

}

lemma *check-wf-problem-return-iff*[*return-iff*]:
check-wf-problem P stg conT mp = Inr () \longleftrightarrow *ast-problem.wf-problem' P stg conT mp*
proof –
interpret *ast-problem P* .
show *?thesis*
unfolding *check-wf-problem-def wf-problem'-def*
by (*auto simp: return-iff*)
qed

definition *check-plan P* $\pi s \equiv$ *do* {
let stg=ast-domain.STG (ast-problem.domain P);
let conT = ast-domain.mp-constT (ast-problem.domain P);
let mp = ast-problem.mp-objT P;
check-wf-problem P stg conT mp;
ast-problem.valid-plan-fromE P stg mp 1 (ast-problem.I P) πs
*} <+? (λe . *String.implode (e () "")*)*

Correctness theorem of the plan checker: It returns *Inr ()* if and only if the problem is well-formed and the plan is valid.

theorem *check-plan-return-iff*[*return-iff*]: *check-plan P* $\pi s = Inr ()$
 \longleftrightarrow *ast-problem.wf-problem P* \wedge *ast-problem.valid-plan P* πs
proof –
interpret *ast-problem P* .
show *?thesis*
unfolding *check-plan-def*
by (*auto*
simp: return-iff wf-world-model-def wf-fmla-atom-alt I-def wf-problem-def
isOK-iff
simp: wf-problem'-correct ast-problem.I-def ast-problem.valid-plan-def wm-basic-def
 $)$
qed

4.4 Code Setup

In this section, we set up the code generator to generate verified code for our plan checker.

4.4.1 Code Equations

We first register the code equations for the functions of the checker. Note that we not necessarily register the original code equations, but also optimized ones.

lemmas *wf-domain-code =*
ast-domain.sig-def
ast-domain.wf-types-def
ast-domain.wf-type.simps

```

ast-domain.wf-predicate-decl.simps
ast-domain.STG-def
ast-domain.is-of-type'-def
ast-domain.wf-atom'.simps
ast-domain.wf-pred-atom'.simps
ast-domain.wf-fmla'.simps
ast-domain.wf-fmla-atom1'.simps
ast-domain.wf-effect'.simps
ast-domain.wf-action-schema'.simps
ast-domain.wf-domain'-def
ast-domain.subst-term.simps
ast-domain.mp-constT-def

```

```

declare wf-domain-code[code]

```

```

lemmas wf-problem-code =
  ast-problem.wf-problem'-def
  ast-problem.wf-fact'-def

```

```

  ast-problem.is-obj-of-type-alt

```

```

  ast-problem.wf-fact-def
  ast-problem.wf-plan-action.simps

```

```

  ast-domain.subtype-edge.simps

```

```

declare wf-problem-code[code]

```

```

lemmas check-code =
  ast-problem.valid-plan-def
  ast-problem.valid-plan-fromE.simps
  ast-problem.en-exE2-def
  ast-problem.resolve-instantiate.simps
  ast-domain.resolve-action-schema-def
  ast-domain.resolve-action-schemaE-def
  ast-problem.I-def
  ast-domain.instantiate-action-schema.simps
  ast-domain.apply-effect-exec.simps

```

```

  ast-domain.apply-effect-eq-impl-eq

```

```

  ast-problem.holds-def
  ast-problem.mp-objT-def
  ast-problem.is-obj-of-type-impl-def
  ast-problem.wf-fmla-atom2'-def
  valuation-def

```

```

declare check-code[code]

```


4.4.2 Setup for Containers Framework

```
derive ceq predicate atom object formula
derive ccompare predicate atom object formula
derive (rbt) set-impl atom formula
```

```
derive (rbt) mapping-impl object
```

```
derive linorder predicate object atom object atom formula
```

4.4.3 More Efficient Distinctness Check for Linorders

```
fun no-stutter :: 'a list ⇒ bool where
  no-stutter [] = True
| no-stutter [-] = True
| no-stutter (a#b#l) = (a≠b ∧ no-stutter (b#l))
```

```
lemma sorted-no-stutter-eq-distinct: sorted l ⇒ no-stutter l ⇔ distinct l
  apply (induction l rule: no-stutter.induct)
  apply (auto simp: )
done
```

```
definition distinct-ds :: 'a::linorder list ⇒ bool
  where distinct-ds l ≡ no-stutter (quicksort l)
```

```
lemma [code-unfold]: distinct = distinct-ds
  apply (intro ext)
  unfolding distinct-ds-def
  apply (auto simp: sorted-no-stutter-eq-distinct)
done
```

4.4.4 Code Generation

```
export-code
  check-plan
  nat-of-integer integer-of-nat Inl Inr
  predAtm Eq predicate Pred Either Var Obj PredDecl BigAnd BigOr
  formula.Not formula.Bot Effect ast-action-schema.Action-Schema
  map-atom Domain Problem PAction
  term.CONST term.VAR
  String.explode String.implode
  in SML
  module-name PDDL-Checker-Exported
  file PDDL-STRIPS-Checker-Exported.sml
```

```
export-code ast-domain.apply-effect-exec in SML module-name ast-domain
```

```
end — Theory
```

5 Soundness theorem for the STRIPS semantics

We prove the soundness theorem according to [4].

```
theory Lifschitz-Consistency
imports PDDL-STRIPS-Semantics
begin
```

States are modeled as valuations of our underlying predicate logic.

```
type-synonym state = (predicate × object list) valuation
```

```
context ast-domain begin
```

An action is a partial function from states to states.

```
type-synonym action = state → state
```

The Isabelle/HOL formula $f\ s = \text{Some } s'$ means that f is applicable in state s , and the result is s' .

Definition B (i)–(iv) in Lifschitz’s paper [4]

```
fun is-NegPredAtom where
  is-NegPredAtom (Not x) = is-predAtom x | is-NegPredAtom - = False
```

```
definition close-eq s = (λpredAtm p xs ⇒ s (p,xs) | Eq a b ⇒ a=b)
```

```
lemma close-eq-predAtm[simp]: close-eq s (predAtm p xs) ↔ s (p,xs)
by (auto simp: close-eq-def)
```

```
lemma close-eq-Eq[simp]: close-eq s (Eq a b) ↔ a=b
by (auto simp: close-eq-def)
```

```
abbreviation entail-eq :: state ⇒ object atom formula ⇒ bool (infix |= 55)
where entail-eq s f ≡ close-eq s |= f
```

```
fun sound-opr :: ground-action ⇒ action ⇒ bool where
  sound-opr (Ground-Action pre (Effect add del)) f ↔
    (∀ s. s |= pre →
      (∃ s'. f s = Some s' ∧ (∀ atm. is-predAtom atm ∧ atm ∉ set del ∧ s |= atm
        → s' |= atm)
        ∧ (∀ atm. is-predAtom atm ∧ atm ∉ set add ∧ s |= Not atm → s'
          |= Not atm)
        ∧ (∀ fmla. fmla ∈ set add → s' |= fmla)
        ∧ (∀ fmla. fmla ∈ set del ∧ fmla ∉ set add → s' |= (Not fmla))
        ))
    ∧ (∀ fmla ∈ set add. is-predAtom fmla)
```

```
lemma sound-opr-alt:
```

$sound\text{-}opr\ f =$
 $((\forall s. s \models (precondition\ opr) \longrightarrow$
 $(\exists s'. f\ s = (Some\ s')$
 $\quad \wedge (\forall atm. is\text{-}predAtom\ atm \wedge atm \notin set(dels\ (effect\ opr)) \wedge s \models atm$
 $\longrightarrow s' \models atm)$
 $\quad \wedge (\forall atm. is\text{-}predAtom\ atm \wedge atm \notin set(adds\ (effect\ opr)) \wedge s \models$
 $Not\ atm \longrightarrow s' \models Not\ atm)$
 $\quad \wedge (\forall atm. atm \in set(adds\ (effect\ opr)) \longrightarrow s' \models atm)$
 $\quad \wedge (\forall fmla. fmla \in set(dels\ (effect\ opr)) \wedge fmla \notin set(adds\ (effect$
 $opr)) \longrightarrow s' \models (Not\ fmla))$
 $\quad \wedge (\forall a\ b. s \models Atom\ (Eq\ a\ b) \longrightarrow s' \models Atom\ (Eq\ a\ b))$
 $\quad \wedge (\forall a\ b. s \models Not\ (Atom\ (Eq\ a\ b)) \longrightarrow s' \models Not\ (Atom\ (Eq\ a\ b)))$
 $))$
 $\quad \wedge (\forall fmla \in set(adds\ (effect\ opr)). is\text{-}predAtom\ fmla))$
by (*cases* (*opr*,*f*) *rule: sound-opr.cases*) *auto*

Definition B (v)–(vii) in Lifschitz’s paper [4]

definition *sound-system*

$::$ *ground-action set*
 \Rightarrow *world-model*
 \Rightarrow *state*
 \Rightarrow (*ground-action* \Rightarrow *action*)
 \Rightarrow *bool*

where

$sound\text{-}system\ \Sigma\ M_0\ s_0\ f \longleftrightarrow$
 $((\forall fmla \in close\text{-}world\ M_0. s_0 \models fmla)$
 $\wedge\ wm\text{-}basic\ M_0$
 $\wedge (\forall \alpha \in \Sigma. sound\text{-}opr\ \alpha\ (f\ \alpha)))$

Composing two actions

definition *compose-action* $::$ *action* \Rightarrow *action* \Rightarrow *action* **where**

compose-action $f1\ f2\ x = (case\ f2\ x\ of\ Some\ y \Rightarrow f1\ y \mid None \Rightarrow None)$

Composing a list of actions

definition *compose-actions* $::$ *action list* \Rightarrow *action* **where**

compose-actions $fs \equiv fold\ compose\text{-}action\ fs\ Some$

Composing a list of actions satisfies some natural lemmas:

lemma *compose-actions-Nil*[*simp*]:

compose-actions $[] = Some$ **unfolding** *compose-actions-def* **by** *auto*

lemma *compose-actions-Cons*[*simp*]:

$f\ s = Some\ s' \implies compose\text{-}actions\ (f\ \#fs)\ s = compose\text{-}actions\ fs\ s'$

proof –

interpret *monoid-add* *compose-action* *Some*

apply *unfold-locales*

unfolding *compose-action-def*

by (*auto split: option.split*)

```

assume  $f s = \text{Some } s'$ 
then show ?thesis
  unfolding compose-actions-def
  by (simp add: compose-action-def fold-plus-sum-list-rev)
qed

```

Soundness Theorem in Lifschitz's paper [4].

theorem *STRIPS-sema-sound*:

```

assumes sound-system  $\Sigma M_0 s_0 f$ 
  — For a sound system  $\Sigma$ 
assumes set  $\alpha s \subseteq \Sigma$ 
  — And a plan  $\alpha s$ 
assumes ground-action-path  $M_0 \alpha s M'$ 
  — Which is accepted by the system, yielding result  $M'$  (called  $R(\alpha s)$  in Lifschitz's
  paper [4].)

```

```

obtains  $s'$ 
  — We have that  $f(\alpha s)$  is applicable in initial state, yielding state  $s'$  (called
   $f_{\alpha s}(s_0)$  in Lifschitz's paper [4].)

```

```

where compose-actions ( $\text{map } f \alpha s$ )  $s_0 = \text{Some } s'$ 
  — The result world model  $M'$  is satisfied in state  $s'$ 
and  $\forall fmla \in \text{close-world } M'. s' \models fmla$ 

```

proof —

```

have (valuation  $M' \models fmla$ ) if wm-basic  $M' fmla \in M'$  for  $fmla$ 
  using that apply (induction fmla)
  by (auto simp: valuation-def wm-basic-def split: atom.split)
have  $\exists s'. \text{compose-actions } (\text{map } f \alpha s) s_0 = \text{Some } s' \wedge (\forall fmla \in \text{close-world } M'. s' \models fmla)$ 

```

```

using assms

```

```

proof(induction  $\alpha s$  arbitrary: s_0 M_0 )

```

```

  case Nil

```

```

then show ?case by (auto simp add: close-world-def compose-action-def sound-system-def)
next

```

```

  case ass: (Cons  $\alpha \alpha s$ )

```

```

then obtain pre add del where  $a: \alpha = \text{Ground-Action } \text{pre } (\text{Effect } \text{add } \text{del})$ 
  using ground-action.exhaust ast-effect.exhaust by metis

```

```

let  $?M_1 = \text{execute-ground-action } \alpha M_0$ 

```

```

have close-world  $M_0 \models \text{precondition } \alpha$ 

```

```

  using ass(4)

```

```

  by auto

```

```

moreover have s0-ent-cwM0:  $\forall fmla \in (\text{close-world } M_0). \text{close-eq } s_0 \models fmla$ 

```

```

  using ass(2)

```

```

  unfolding sound-system-def

```

```

  by auto

```

```

ultimately have s0-ent-alpha-precond:  $\text{close-eq } s_0 \models \text{precondition } \alpha$ 

```

```

  unfolding entailment-def

```

```

  by auto

```

```

then obtain  $s_1$  where  $s_1: (f \alpha) s_0 = \text{Some } s_1$ 

```

```

  ( $\forall \text{atm. is-predAtom } \text{atm} \longrightarrow \text{atm} \notin \text{set}(\text{dels } (\text{effect } \alpha))$ )

```

```

   $\longrightarrow \text{close-eq } s_0 \models \text{atm}$ 

```

$\longrightarrow \text{close-eq } s_1 \models \text{atm}$
 $(\forall \text{fmla}. \text{fmla} \in \text{set}(\text{adds } (\text{effect } \alpha)))$
 $\longrightarrow \text{close-eq } s_1 \models \text{fmla}$
 $(\forall \text{atm}. \text{is-predAtom } \text{atm} \wedge \text{atm} \notin \text{set}(\text{adds } (\text{effect } \alpha)) \wedge \text{close-eq } s_0 \models \text{Not } \text{atm})$
 $\longrightarrow \text{close-eq } s_1 \models \text{Not } \text{atm}$
 $(\forall \text{fmla}. \text{fmla} \in \text{set}(\text{dels } (\text{effect } \alpha)) \wedge \text{fmla} \notin \text{set}(\text{adds } (\text{effect } \alpha))) \longrightarrow \text{close-eq } s_1 \models (\text{Not } \text{fmla})$
 $(\forall a b. \text{close-eq } s_0 \models \text{Atom } (\text{Eq } a b) \longrightarrow \text{close-eq } s_1 \models \text{Atom } (\text{Eq } a b))$
 $(\forall a b. \text{close-eq } s_0 \models \text{Not } (\text{Atom } (\text{Eq } a b)) \longrightarrow \text{close-eq } s_1 \models \text{Not } (\text{Atom } (\text{Eq } a b)))$
using *ass(2-4)*
unfolding *sound-system-def sound-opr-alt* **by** *force*
have $\text{close-eq } s_1 \models \text{fmla}$ **if** $\text{fmla} \in \text{close-world } ?M_1$ **for** fmla
using *ass(2)*
using *that s1 s0-ent-cwM0*
unfolding *sound-system-def execute-ground-action-def wm-basic-def*
apply (*auto simp: in-close-world-conv*)
subgoal
by (*metis (no-types, lifting) DiffE UnE a apply-effect.simps ground-action.sel(2) ast-effect.sel(1) ast-effect.sel(2) close-world-extensive subsetCE*)
subgoal
by (*metis Diff-iff Un-iff a ground-action.sel(2) ast-domain.apply-effect.simps ast-domain.close-eq-predAtom ast-effect.sel(1) ast-effect.sel(2) formula-antics.simps(1) formula-antics.simps(3) in-close-world-conv is-predAtom.simps(1)*)
done
moreover have $(\forall \text{atm}. \text{fmla} \neq \text{formula.Atom } \text{atm}) \longrightarrow s \models \text{fmla}$ **if** $\text{fmla} \in ?M_1$
for $\text{fmla } s$
proof–
have *alpha*: $(\forall s. \forall \text{fmla} \in \text{set}(\text{adds } (\text{effect } \alpha)). \neg \text{is-predAtom } \text{fmla} \longrightarrow s \models \text{fmla})$
using *ass(2,3)*
unfolding *sound-system-def ast-domain.sound-opr-alt*
by *auto*
then show *?thesis*
using *that*
unfolding *a execute-ground-action-def*
using *ass.prem(1)[unfolded sound-system-def]*
by(*cases fmla; fastforce simp: wm-basic-def*)

qed
moreover have $(\forall \text{opr} \in \Sigma. \text{sound-opr } \text{opr} (f \text{opr}))$
using *ass(2) unfolding sound-system-def*
by (*auto simp add:*)
moreover have *wm-basic ?M₁*
using *ass(2,3)*
unfolding *sound-system-def execute-ground-action-def*
thm *sound-opr.cases*
apply (*cases (α, f α) rule: sound-opr.cases*)
apply (*auto simp: wm-basic-def*)

done
ultimately have *sound-system* Σ ? M_1 s_1 f
unfolding *sound-system-def*
by (*auto simp: wm-basic-def*)
from *ass.IH[OF this] ass.prem* **obtain** s' **where**
compose-actions (*map f* αs) $s_1 = \text{Some } s' \wedge (\forall a \in \text{close-world } M'. s' \models a)$
by *auto*
thus ?*case* **by** (*auto simp: s1(1)*)
qed
with that show ?*thesis* **by** *blast*
qed

More compact notation of the soundness theorem.

theorem *STRIPS-sema-sound-compact-version*:
sound-system Σ M_0 s_0 $f \implies \text{set } \alpha s \subseteq \Sigma$
 $\implies \text{ground-action-path } M_0$ αs M'
 $\implies \exists s'. \text{compose-actions } (\text{map } f \alpha s) s_0 = \text{Some } s'$
 $\wedge (\forall \text{fmla} \in \text{close-world } M'. s' \models \text{fmla})$
using *STRIPS-sema-sound* **by** *metis*

end — Context of *ast-domain*

5.1 Soundness Theorem for PDDL

context *wf-ast-problem* **begin**

Mapping world models to states

definition *state-to-wm* :: *state* \Rightarrow *world-model*
where *state-to-wm* $s = (\{\text{formula.Atom } (\text{predAtm } p \text{ } xs) \mid p \text{ } xs. s (p, xs)\})$
definition *wm-to-state* :: *world-model* \Rightarrow *state*
where *wm-to-state* $M = (\lambda(p, xs). (\text{formula.Atom } (\text{predAtm } p \text{ } xs)) \in M)$

lemma *wm-to-state-eq[simp]*: *wm-to-state* $M (p, as) \longleftrightarrow \text{Atom } (\text{predAtm } p \text{ } as)$
 $\in M$
by (*auto simp: wm-to-state-def*)

lemma *wm-to-state-inv[simp]*: *wm-to-state* (*state-to-wm* s) = s
by (*auto simp: wm-to-state-def*
state-to-wm-def image-def)

Mapping AST action instances to actions

definition *pddl-opr-to-act* $g\text{-opr } s = (
\text{let } M = \text{state-to-wm } s \text{ in }
\text{if } (\text{wm-to-state } (\text{close-world } M)) \models (\text{precondition } g\text{-opr}) \text{ then }
\text{Some } (\text{wm-to-state } (\text{apply-effect } (\text{effect } g\text{-opr}) M))$)

else
None)

definition *close-eq-M* $M = (M \cap \{Atom (predAtm p xs) \mid p xs. True\}) \cup \{Atom (Eq a a) \mid a. True\} \cup \{\neg(Atom (Eq a b)) \mid a b. a \neq b\}$

lemma *atom-in-wm-eq*:

$s \models_{=}$ (formula.Atom atm)
 $\longleftrightarrow ((formula.Atom atm) \in close-eq-M (state-to-wm s))$
by (auto simp: wm-to-state-def
state-to-wm-def image-def close-eq-M-def close-eq-def split: atom.splits)

lemma *atom-in-wm-2-eq*:

$close-eq (wm-to-state M) \models (formula.Atom atm)$
 $\longleftrightarrow ((formula.Atom atm) \in close-eq-M M)$
by (auto simp: wm-to-state-def
state-to-wm-def image-def close-eq-def close-eq-M-def split:atom.splits)

lemma *not-dels-preserved*:

assumes $f \notin (set d)$ $f \in M$
shows $f \in apply-effect (Effect a d) M$
using *assms*
by *auto*

lemma *adds-satisfied*:

assumes $f \in (set a)$
shows $f \in apply-effect (Effect a d) M$
using *assms*
by *auto*

lemma *dels-unsatisfied*:

assumes $f \in (set d)$ $f \notin set a$
shows $f \notin apply-effect (Effect a d) M$
using *assms*
by *auto*

lemma *dels-unsatisfied-2*:

assumes $f \in set (dels eff)$ $f \notin set (adds eff)$
shows $f \notin apply-effect eff M$
using *assms*
by (cases *eff*; *auto*)

lemma *wf-fmla-atm-is-atom*: $wf-fmla-atom\ objT\ f \implies is-predAtom\ f$

by (cases *f* rule: *wf-fmla-atom.cases*) *auto*

lemma *wf-act-adds-are-atoms*:

assumes $wf-effect-inst\ effs\ ae \in set (adds\ effs)$
shows $is-predAtom\ ae$
using *assms*

by (*cases effs*) (*auto simp: wf-fmla-atom-alt*)

lemma *wf-act-adds-dels-atoms*:

assumes *wf-effect-inst effs ae ∈ set (dels effs)*

shows *is-predAtom ae*

using *assms*

by (*cases effs*) (*auto simp: wf-fmla-atom-alt*)

lemma *to-state-close-from-state-eq[simp]*: *wm-to-state (close-world (state-to-wm s)) = s*

by (*auto simp: wm-to-state-def close-world-def state-to-wm-def image-def*)

lemma *wf-eff-pddl-ground-act-is-sound-opr*:

assumes *wf-effect-inst (effect g-opr)*

shows *sound-opr g-opr ((pddl-opr-to-act g-opr))*

unfolding *sound-opr-alt*

apply (*cases g-opr; safe*)

subgoal for *pre eff s*

apply (*rule exI[where x=wm-to-state(apply-effect eff (state-to-wm s))]*)

apply (*auto simp: pddl-opr-to-act-def Let-def split:if-splits*)

subgoal for *atm*

by (*cases eff; cases atm; auto simp: close-eq-def wm-to-state-def state-to-wm-def split: atom.splits*)

subgoal for *atm*

by (*cases eff; cases atm; auto simp: close-eq-def wm-to-state-def state-to-wm-def split: atom.splits*)

subgoal for *atm*

using *assms*

by (*cases eff; cases atm; force simp: close-eq-def wm-to-state-def state-to-wm-def split: atom.splits*)

subgoal for *fmla*

using *assms*

by (*cases eff; cases fmla rule: wf-fmla-atom.cases; force simp: close-eq-def wm-to-state-def state-to-wm-def split: atom.splits*)

done

subgoal for *pre eff fmla*

using *assms*

by (*cases eff; cases fmla rule: wf-fmla-atom.cases; force*)

done

lemma *wf-eff-imp-wf-eff-inst*: *wf-effect objT eff \implies wf-effect-inst eff*

by (*cases eff; auto simp add: wf-fmla-atom-alt*)

lemma *wf-pddl-ground-act-is-sound-opr*:

assumes *wf-ground-action g-opr*
shows *sound-opr g-opr (pddl-opr-to-act g-opr)*
using *wf-eff-impl-wf-eff-inst wf-eff-pddl-ground-act-is-sound-opr assms*
by (*cases g-opr; auto*)

lemma *wf-action-schema-sound-inst:*
assumes *action-params-match act args wf-action-schema act*
shows *sound-opr*
(instantiate-action-schema act args)
((pddl-opr-to-act (instantiate-action-schema act args)))
using
wf-pddl-ground-act-is-sound-opr [
OF wf-instantiate-action-schema[OF assms]]
by *blast*

lemma *wf-plan-act-is-sound:*
assumes *wf-plan-action (PAction n args)*
shows *sound-opr*
(instantiate-action-schema (the (resolve-action-schema n)) args)
((pddl-opr-to-act
(instantiate-action-schema (the (resolve-action-schema n)) args)))
using *assms*
using *wf-action-schema-sound-inst wf-eff-pddl-ground-act-is-sound-opr*
by (*auto split: option.splits*)

lemma *wf-plan-act-is-sound':*
assumes *wf-plan-action π*
shows *sound-opr*
(resolve-instantiate π)
((pddl-opr-to-act (resolve-instantiate π)))
using *assms wf-plan-act-is-sound*
by (*cases π ; auto*)

lemma *wf-world-model-has-atoms: $f \in M \implies wf\text{-world-model } M \implies is\text{-predAtom } f$*
using *wf-fmla-atm-is-atom*
unfolding *wf-world-model-def*
by *auto*

lemma *wm-to-state-works-for-wf-wm-closed:*
wf-world-model $M \implies fmla \in close\text{-world } M \implies close\text{-eq } (wm\text{-to-state } M) \models$
fmla
apply (*cases fmla rule: wf-fmla-atom.cases*)
by (*auto simp: wf-world-model-def close-eq-def wm-to-state-def close-world-def*)

lemma *wm-to-state-works-for-wf-wm: wf-world-model $M \implies fmla \in M \implies close\text{-eq}$*
(wm-to-state M) \models fmla
apply (*cases fmla rule: wf-fmla-atom.cases*)
by (*auto simp: wf-world-model-def close-eq-def wm-to-state-def*)

lemma *wm-to-state-works-for-I-closed*:
assumes $x \in \text{close-world } I$
shows $\text{close-eq } (\text{wm-to-state } I) \models x$
apply (*rule wm-to-state-works-for-wf-wm-closed*)
using *assms wf-I* **by** *auto*

lemma *wf-wm-imp-basic*: $\text{wf-world-model } M \implies \text{wm-basic } M$
by (*auto simp: wf-world-model-def wm-basic-def wf-fmla-atm-is-atom*)

theorem *wf-plan-sound-system*:
assumes $\forall \pi \in \text{set } \pi s. \text{wf-plan-action } \pi$
shows *sound-system*
 $(\text{set } (\text{map } \text{resolve-instantiate } \pi s))$
 I
 $(\text{wm-to-state } I)$
 $((\lambda \alpha. \text{pddl-opr-to-act } \alpha))$
unfolding *sound-system-def*
proof(*intro conjI ballI*)
show $\text{close-eq}(\text{wm-to-state } I) \models x$ **if** $x \in \text{close-world } I$ **for** x
using *that[unfolding in-close-world-conv]*
 $\text{wm-to-state-works-for-I-closed}$ $\text{wm-to-state-works-for-wf-wm}$
by (*auto simp: wf-I*)

show $\text{wm-basic } I$ **using** *wf-wm-imp-basic[OF wf-I]* .

show $\text{sound-opr } \alpha (\text{pddl-opr-to-act } \alpha)$ **if** $\alpha \in \text{set } (\text{map } \text{resolve-instantiate } \pi s)$
for α
using *that*
using *wf-plan-act-is-sound' assms*
by *auto*
qed

theorem *wf-plan-soundness-theorem*:
assumes *plan-action-path* $I \pi s M$
defines $\alpha s \equiv \text{map } (\text{pddl-opr-to-act } \circ \text{resolve-instantiate}) \pi s$
defines $s_0 \equiv \text{wm-to-state } I$
shows $\exists s'. \text{compose-actions } \alpha s s_0 = \text{Some } s' \wedge (\forall \varphi \in \text{close-world } M. s' \models \varphi)$
apply (*rule STRIPS-sema-sound*)
apply (*rule wf-plan-sound-system*)
using *assms*
unfolding *plan-action-path-def*
by (*auto simp add: image-def*)

end — Context of *wf-ast-problem*

end

References

- [1] M. Abdulaziz and P. Lammich. A Formally Verified Validator for Classical Planning Problems and Solutions. In *International Conference on Tools in Artificial Intelligence (ICTAI)*. IEEE, 2018.
- [2] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.
- [3] M. Helmert. The fast downward planning system. *J. Artif. Intell. Res.(JAIR)*, 26:191–246, 2006.
- [4] V. Lifschitz. On the semantics of strips. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pages 1–9, 1987.
- [5] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL: The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.