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 Plans ......................................................... 16
    3.2.4 Ground Actions ............................................. 17
  3.3 Closed-World Assumption, Equality, and Negation .......... 17
    3.3.1 Proper Generalization ..................................... 19

*Author names are alphabetically ordered.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4  STRIPS Semantics</td>
<td>20</td>
</tr>
<tr>
<td>3.5  Well-Formedness of PDDL</td>
<td>21</td>
</tr>
<tr>
<td>3.6  PDDL Semantics</td>
<td>25</td>
</tr>
<tr>
<td>3.7  Preservation of Well-Formedness</td>
<td>27</td>
</tr>
<tr>
<td>3.7.1 Well-Formed Action Instances</td>
<td>27</td>
</tr>
<tr>
<td>3.7.2 Preservation</td>
<td>31</td>
</tr>
<tr>
<td>4    Executable PDDL Checker</td>
<td>33</td>
</tr>
<tr>
<td>4.1  Generic DFS Reachability Checker</td>
<td>34</td>
</tr>
<tr>
<td>4.2  Implementation Refinements</td>
<td>36</td>
</tr>
<tr>
<td>4.2.1 Of-Type</td>
<td>36</td>
</tr>
<tr>
<td>4.2.2 Application of Effects</td>
<td>39</td>
</tr>
<tr>
<td>4.2.3 Well-Formedness</td>
<td>40</td>
</tr>
<tr>
<td>4.2.4 Execution of Plan Actions</td>
<td>41</td>
</tr>
<tr>
<td>5    Soundness theorem for the STRIPS semantics</td>
<td>42</td>
</tr>
<tr>
<td>5.1  Soundness Theorem for PDDL</td>
<td>46</td>
</tr>
</tbody>
</table>
theory SASP-Semantics
imports Main
begin

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

1.1 Syntax

\begin{itemize}
\item \textbf{type-synonym} \textit{name} = string
\item \textbf{type-synonym} \textit{ast-variable} = \textit{name} × nat option × \textit{name} list
\item \textbf{type-synonym} \textit{ast-variable-section} = \textit{ast-variable} list
\item \textbf{type-synonym} \textit{ast-initial-state} = nat list
\item \textbf{type-synonym} \textit{ast-goal} = (nat × nat) list
\item \textbf{type-synonym} \textit{ast-precond} = (nat × nat)
\item \textbf{type-synonym} \textit{ast-effect} = \textit{ast-precond} list × nat × nat option × nat
\item \textbf{type-synonym} \textit{ast-operator} = \textit{name} × \textit{ast-precond} list × \textit{ast-effect} list × nat
\item \textbf{type-synonym} \textit{ast-operator-section} = \textit{ast-operator} list
\item \textbf{type-synonym} \textit{ast-problem} = \textit{ast-variable-section} × \textit{ast-initial-state} × \textit{ast-goal} × \textit{ast-operator-section}
\item \textbf{type-synonym} \textit{plan} = name list
\end{itemize}

1.1.1 Well-Formedness

\begin{itemize}
\item \textbf{locale} \textit{ast-problem} =
\item \textbf{fixes} \textit{problem} :: \textit{ast-problem}
\item \textbf{begin}
\item \textbf{definition} \textit{astDom} :: \textit{ast-variable-section}
\item \textbf{where} \textit{astDom} \equiv case problem of \((D,I,G,\delta) \Rightarrow D\)
\item \textbf{definition} \textit{astI} :: \textit{ast-initial-state}
\item \textbf{where} \textit{astI} \equiv case problem of \((D,I,G,\delta) \Rightarrow I\)
\item \textbf{definition} \textit{astG} :: \textit{ast-goal}
\item \textbf{where} \textit{astG} \equiv case problem of \((D,I,G,\delta) \Rightarrow G\)
\item \textbf{definition} \textit{ast\delta} :: \textit{ast-operator-section}
\item \textbf{where} \textit{ast\delta} \equiv case problem of \((D,I,G,\delta) \Rightarrow \delta\)
\item \textbf{definition} \textit{numVars} \equiv length \textit{astDom}
\item \textbf{definition} \textit{numVals} \textit{x} \equiv length \textit{(snd (\textit{snd (astDom!x)}))}
\item \textbf{definition} \textit{wf-partial-state} \textit{ps} \equiv
\item \textbf{distinct} \textit{(map \textit{fst} \textit{ps})}
\item \(\land (\forall (x,v) \in \textit{set ps}. x < \textit{numVars} \land v < \textit{numVals} \textit{x})\)
\item \textbf{definition} \textit{wf-operator} :: \textit{ast-operator} \Rightarrow \textit{bool}
\item \textbf{where} \textit{wf-operator} \equiv \lambda (\textit{name}, \textit{pres}, \textit{effs}, \textit{cost}).
\item \textit{wf-partial-state} \textit{pres}
\item \textbf{distinct} \textit{(map (\lambda (\textit{\_}, \textit{v}, \textit{\_}, \textit{\_}). \textit{v}) \textit{effs})} — This may be too restrictive
\end{itemize}
\[ \forall (epres, x, vp, v) \in \text{set effs}. \]
\[ \text{wf-partial-state epres} \]
\[ \wedge \ x < \text{numVars} \wedge \ v < \text{numVals} \]
\[ \wedge (\text{case } vp \text{ of } \text{None} \Rightarrow \text{True} | \text{Some } v \Rightarrow v < \text{numVals}) \]

**definition** well-formed \( \equiv \)

— Initial state
\[ \text{length } \text{astI} = \text{numVars} \]
\[ \wedge (\forall x < \text{numVars}. \text{astI}!x < \text{numVals} x) \]

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

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

**end**

**locale** wf-ast-problem = ast-problem +

**assumes** \( \text{wf} : \) well-formed

**begin**

**lemma** \( \text{wf-initial} : \)
\[ \text{length } \text{astI} = \text{numVars} \]
\[ \forall x < \text{numVars}. \text{astI}!x < \text{numVals} x \]
using \( \text{wf} \) unfolding well-formed-def by auto

**lemma** \( \text{wf-goal} : \) wf-partial-state \( \text{astG} \)
using \( \text{wf} \) unfolding well-formed-def by auto

**lemma** \( \text{wf-operators} : \)
distinct (map fst ast\( \delta \))
\[ \forall \pi \in \text{set } \text{ast}\delta. \text{wf-operator } \pi \]
using \( \text{wf} \) 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** \( \text{Dom} :: \) nat set where \( \text{Dom} = \{0..<\text{numVars}\} \)


definition range-of-var where range-of-var x ≡ \{0..<\text{numVals }x\}

definition valid-states :: state set where valid-states ≡ \{s. \text{dom }s = \text{Dom }\land (\forall x \in \text{Dom. the } (s x) \in \text{range-of-var }x)\}

definition I :: state where I v ≡ if v < length astI then Some (astI!v) else None

definition subsuming-states :: pstate ⇒ state set where subsuming-states partial ≡ \{s \in \text{valid-states. partial }\subseteq_m s\}

definition G :: state set where G ≡ subsuming-states (map-of astG)

definition implicit-pres :: ast-effect list ⇒ ast-precond list where implicit-pres effs ≡ map (λ(-,v,vpre,-). (v,\text{the } vpre)) (filter (λ(-,vpre,-). vpre ≠ None) effs)

collection ast-problem

begin

definition lookup-operator :: name ⇒ ast-operator option where lookup-operator name ≡ find (λ(n,-,-,-). n = name) astδ

definition enabled :: name ⇒ state ⇒ bool where enabled name s ≡ case lookup-operator name of Some (-,pres,effs,-) ⇒ s \in \text{subsuming-states (map-of pres)} \land s \in \text{subsuming-states (map-of (implicit-pres effs))} \mid \text{None }⇒ \text{False}

definition eff-enabled :: state ⇒ ast-effect ⇒ bool where eff-enabled s ≡ λ(pres,-,-,-). s \in \text{subsuming-states (map-of pres)}

definition execute :: name ⇒ state ⇒ state where execute name s ≡ case lookup-operator name of Some (-,effs,-) ⇒ s ++ map-of (map (λ(-,x,-,v). (x,v)) (filter (eff-enabled s) effs)) \mid \text{None }⇒ \text{undefined}

fun path-to where path-to s [] s′ ←→ s′ = s
\[ \text{path-to } s (\pi \# \pi s) \iff \text{enabled } \pi s \land \text{path-to } (\text{execute } \pi s) \pi s s' \]

**Definition**

\[
\text{valid-plan :: } \text{plan } \Rightarrow \text{bool}
\]

where \[\text{valid-plan } \pi s \equiv \exists s' \in G. \text{path-to } I \pi s s'\]

**1.2.1 Preservation of well-formedness**

**Context**

\text{wf-ast-problem}

**Begin**

**Lemma**

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

**Using**

\text{wf-initial}

**Unfolding**

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

**By**

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

**Lemma**

\text{lookup-operator-wf:}

**Assumes**

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

**Shows**

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

**Proof**

**Obtain**

\text{name' pres effs cost where [simp]: } \pi = (name', \text{pres}, \text{effs}, \text{cost}) \text{ by (cases } \pi\text{)}

**Hence**

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

**Using**

\text{assms}

**Unfolding**

\text{lookup-operator-def}

**Apply**

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

**By**

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

**From**

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

**Unfolding**

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

**Qed**

**Lemma**

\text{execute-preserves-valid:}

**Assumes**

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

**Assumes**

\text{enabled name s}

**Shows**

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

**Proof**

**From**

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

**Unfolding**

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

**From**

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

**By**

\text{simp}

**Have**

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

**Using**

\text{that } (s \in \text{valid-states})
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 \( \text{map-of} (\lambda (-, x, - , y). (x, y)) (\text{filter (eff-enabled s) effs})) \) \( x = \text{Some} \)
  for \( x \) \( v \)
proof –
  from that obtain 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 valid-states
  assumes path-to s π s s'
  shows s'\in valid-states
  using assms
  by (induction s π 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 \( \forall k v. (k,v) \in \text{set l} \implies m k = \text{Some} v \)
  shows map-of l \( \subseteq m \)
  using assms
  by (metis (no-types, hide-lams) domIff map-le-def map-of-SomeD not-Some-eq)
lemma [simp]: \( \text{fst} \circ \lambda(a, b, c, d). (f a b c d, g a b c d) = \lambda(a,b,c,d). f a b c d \)  
by auto

lemma map-mp: \( m \subseteq m' \Rightarrow m k = \text{Some } v \Rightarrow m' k = \text{Some } v \)  
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)
sows 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 \text{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.numVars-def ast-problem.numVals-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 = \text{Some } v

definition match-pres :: ast-precond list \Rightarrow state \Rightarrow bool where  
match-pres ps s \equiv \forall \text{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 (\text{-x,vp,-})\in set effs.  
(case vp of None \Rightarrow \text{True } | \text{Some } v \Rightarrow s x = \text{Some } v)
definition enabled-opr' :: ast-operator ⇒ state ⇒ bool where
   enabled-opr' ≡ λ(name, pres, effs, cost) s. match-pres pres s ∧ match-implicit-pres effs s

definition eff-enabled' :: state ⇒ ast-effect ⇒ bool where
   eff-enabled' s ≡ λ(pres, - , - , - ) . match-pres pres s

definition execute-opr' ≡ λ(name, - , effs, - ) s.
   let effs = filter (eff-enabled' s) effs
   in fold (λ(-, x, - , v) . s(x=v)) effs s

definition lookup-operator' :: ast-problem ⇒ name ⇒ ast-operator
   where lookup-operator' ≡ λ(D, I, G, δ) name . find (λ(n, - , - , - ) . n = name) δ

definition enabled' :: ast-problem ⇒ name ⇒ state ⇒ bool where
   enabled' problem name s ≡
     case lookup-operator' problem name of
     Some π ⇒ enabled-opr' π s
     | None ⇒ False

definition execute' :: ast-problem ⇒ name ⇒ state ⇒ state where
   execute' problem name s ≡
     case lookup-operator' problem name of
     Some π ⇒ execute-opr' π s
     | None ⇒ undefined

context wf-ast-problem begin

lemma match-pres-correct:
   assumes D: distinct (map fst pres)
   assumes s ∈ valid-states
   shows match-pres pres s ←→ s ∈ subsuming-states (map-of pres)
proof –
   have match-pres pres s ←→ map-of pres ⊆ m s
     unfolding match-pres-def match-pre-def
     apply (auto split: prod.splits)
     using map-le-def map-of-SomeD apply fastforce
   by (metis (full-types) D domIff map-le-def map-of-eq-Some iff option.simps(3))

   with assms show ?thesis
     unfolding subsuming-states-def
   by auto
qed

lemma match-implicit-pres-correct:
   assumes D: distinct (map (λ(-, v, - , - ) . v) effs)
   assumes s ∈ valid-states

shows match-implicit-pres effs s ←→ s ∈ subsuming-states (map-of (implicit-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: 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 $(\pi \# \pi s) \ s = ( $
| \quad \text{if enabled } \pi \ s \text{ then} 
| \quad \text{let } s' = \text{execute } \pi \ s \text{ in} 
| \quad \text{simulate-plan } \pi s \ s' 
| \quad \text{else} 
| \quad \text{None} 
| )$

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

**By:** (induction $s \ \pi s \ s'$ rule: path-to.induct) auto

**Definition check-plan ::** plan $\Rightarrow$ bool where

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

**Lemma check-plan-correct:** 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' :: ast-problem $\Rightarrow$ plan $\Rightarrow$ state $\Rightarrow$ state where

simulate-plan' problem $[\ ] \ s =$ Some $s$

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

Avoiding duplicate lookup.

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

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

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

**Definition** initial-state' :: ast-problem $\Rightarrow$ state where
initial-state’ problem ≡ let astI = ast-problem.astI in (λv. if v<length astI then Some (astI!v) else None)

definition check-plan’ :: ast-problem ⇒ plan ⇒ bool where
  check-plan’ problem π s = (case simulate-plan’ problem π s (initial-state’ problem) of
    None ⇒ False
  | Some s' ⇒ match-pres (ast-problem.astG problem) s')

context wf-ast-problem
begin

lemma simulate-plan’-correct:
  assumes s∈valid-states
  shows simulate-plan’ problem π s s = simulate-plan π s s using assms
  apply (induction π s s rule: simulate-plan.induct)
  apply (auto simp: enabled’-correct execute’-correct execute-preserves-valid)
  done

lemma simulate-plan’-correct-paper:
  assumes s∈valid-states
  shows simulate-plan’ problem π s s = Some s’ ←→ path-to s π s 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 π s = check-plan π s
  proof –
    have D: distinct (map fst astG) using wf-goal unfolding wf-partial-state-def by auto

    have S'V: s'∈valid-states if simulate-plan π s 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]
definition verify-plan :: ast-problem ⇒ plan ⇒ String.literal + unit where
  verify-plan problem π s = (  
    if ast-problem.well-formed problem then  
      if check-plan′ problem π s then Inr () else Inl (STR "Invalid plan")  
    else Inl (STR "Problem not well formed")  
  )

lemma verify-plan-correct:
  verify-plan problem π s = Inr () ←→ ast-problem.well-formed problem ∧ ast-problem.valid-plan problem π s
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 π s = valid-plan π s by simp
}
then show ?thesis
  unfolding verify-plan-def
  by auto
qed

definition nat-opt-of-integer :: integer ⇒ nat option where
  nat-opt-of-integer i = (if (i ≥ 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
file code/SASP-Checker-Exported.sml

end

3 PDDL and STRIPS Semantics

theory PDDL-STRIPS-Semantics
imports
  Propositional-Proof-Systems.Formulas
  Propositional-Proof-Systems.Sema
3.1 Utility Functions

**Definition** \( \text{index-by } f \ l \equiv \text{map-of } (\lambda x. (f x, x)) \ l \)

**Lemma** \( \text{index-by-eq-Some-eq[simp]} \):
\[
\begin{align*}
\text{assumes} & \quad \text{distinct } (\text{map } f \ l) \\
\text{shows} & \quad \text{index-by } f \ l \ n = \text{Some } x \iff (x \in \text{set } l \land f x = n)
\end{align*}
\]
\text{unfolding index-by-def using assms by (auto simp: o-def)}

**Lemma** \( \text{index-by-eq-SomeD} \):
\[
\begin{align*}
\text{shows} & \quad \text{index-by } f \ l \ n = \text{Some } x \implies (x \in \text{set } l \land f x = n)
\end{align*}
\]
\text{unfolding index-by-def by (auto dest: map-of-SomeD)}

**Lemma** \( \text{lookup-zip-idx-eq} \):
\[
\begin{align*}
\text{assumes} & \quad \text{length } \text{params} = \text{length } \text{args} \\
\text{assumes} & \quad i < \text{length } \text{args} \\
\text{assumes} & \quad \text{distinct } \text{params} \\
\text{assumes} & \quad k = \text{params} ! i \\
\text{shows} & \quad \text{map-of } (\text{zip } \text{params } \text{args}) \ k = \text{Some } (\text{args} ! i)
\end{align*}
\]
\text{using assms by (auto simp: in-set-conv-nth)}

**Lemma** \( \text{rtrancl-image-idem[simp]} \):
\[
\begin{align*}
R^* \ " " R^* \ " " s = R^* \ " " s
\end{align*}
\]
\text{by (metis relcomp-Image rtrancl-idemp-self-comp)}

3.2 Abstract Syntax

3.2.1 Generic Entities

**Type-Synonym** \( \text{name } = \text{string} \)

**Datatype** \( \text{predicate } = \text{Pred } (\text{name: name}) \)

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** \( \text{ent atom } = \text{predAtm } (\text{predicate: predicate}) \ (\text{arguments: ent list}) \)
\[
| \ \text{Eq } (\text{lhs: ent}) \ (\text{rhs: ent})
\]

15
A type is a list of primitive type names. To model a primitive type, we use a singleton list.

\[ \text{datatype type} = \text{Either (primitives: name list)} \]

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

\[ \text{datatype 'ent ast-effect} = \text{Effect (adds: ('ent atom formula list) (dels: ('ent atom formula list))} \]

Variables are identified by their names.

\[ \text{datatype variable} = \text{varname: Var name} \]

Objects and constants are identified by their names.

\[ \text{datatype object} = \text{name: Obj name} \]

\[ \text{datatype term} = \text{VAR variable | CONST object} \]

\[ \text{hide-const (open) VAR CONST} — \text{Refer to constructors by qualified names only} \]

### 3.2.2 Domains

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

\[ \text{datatype ast-action-schema} = \text{Action-Schema (name: name)} \]
\[ (\text{parameters: (variable \times type list)}) \]
\[ (\text{precondition: term atom formula}) \]
\[ (\text{effect: term ast-effect}) \]

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

\[ \text{datatype predicate-decl} = \text{PredDecl (pred: predicate)} \]
\[ (\text{argTs: type list}) \]

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

\[ \text{datatype ast-domain} = \text{Domain (types: (name \times name list) — (type, supertype) declarations. (predicates: predicate-decl list) (consts: (object \times type list) (actions: ast-action-schema list))} \]

### 3.2.3 Problems

A fact is a predicate applied to objects.

\[ \text{type-synonym fact} = \text{predicate \times object list} \]
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 ≡ ∀ a∈M. is-predAtom a

A valuation extracted from the atoms of the world model

**definition** valuation :: world-model ⇒ object atom valuation
where valuation M ≡ λpredAtm p xs ⇒ Atom (predAtm p xs) ∈ M | Eq a b
⇒ 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 ⇒ world-model where close-world M =
M ∪ {¬(Atom (predAtm p as)) | p as. Atom (predAtm p as) /∈ M}
∪ \{\text{Atom} \ (\text{Eq} \ a \ a) \mid a. \ \text{True}\} \cup \{\neg(\text{Atom} \ (\text{Eq} \ a \ b)) \mid a \ b. \ a \neq b\}

definition close-neg M \equiv M \cup \{\neg(\text{Atom} \ a) \mid a. \ \text{Atom} \ a \not\in M\}

lemma wm-basic M \implies \text{close-world} M = \text{close-neg} (M \cup \{\text{Atom} \ (\text{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 $\models\models$ 53) where
$M \models\models \varphi \equiv \text{close-world} M \models \varphi$

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:
$\varphi \in \text{close-world} M \iff (\varphi \in M \land \varphi = \neg(\text{Atom} \ (\text{predAtm} \ p \ as)) \land \text{Atom} (\text{predAtm} \ p \ as) \not\in M)$
$\lor (\varphi = \text{Atom} (\text{Eq} \ a \ a))$
$\lor (\exists a \ b. \varphi = \neg(\text{Atom} \ (\text{Eq} \ a \ b)) \land a \neq b)$
by (auto simp: close-world-def)

lemma valuation-aux-1:
fixes $M$ :: world-model and $\varphi$ :: object atom formula
defines $C \equiv \text{close-world} M$
assumes $A$: $\forall \varphi \in C. \ A \models \varphi$
sows $A = \text{valuation} M$
using $A$ unfolding $C$-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 $\text{wm-basic} M$
sows ($\forall G \in \text{close-world} M. \ \text{valuation} M \models G$)
using assms unfolding $\text{wm-basic-def}$
by (force simp: in-close-world-conv valuation-def elim: is-predAtom.elims)
lemma val-imp-close-world: valuation $M \models \varphi$ $\implies$ $|\Sigma|\models \varphi$
  unfolding entailment-def
  using valuation-aux-1
  by blast

lemma close-world-imp-val:
  $\text{wm-basic } M \implies M \models\varphi \implies \text{valuation } M \models \varphi$
  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 $\varphi$ if and only if the closure of $M$ entails $\varphi$.
Note that there are no syntactic restrictions on $\varphi$, in particular, $\varphi$ may contain negation.

theorem valuation-iff-close-world:
  assumes $\text{wm-basic } M$
  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 ⇒ bool where
  is-STRIPS-fmla (Atom (predAtm - -)) ←→ True
  | is-STRIPS-fmla (⊥) ←→ True
  | is-STRIPS-fmla ($\varphi_1 \land \varphi_2$) ←→ is-STRIPS-fmla $\varphi_1 \land$ is-STRIPS-fmla $\varphi_2$
  | is-STRIPS-fmla ($\varphi_1 \lor \varphi_2$) ←→ is-STRIPS-fmla $\varphi_1 \land$ is-STRIPS-fmla $\varphi_2$
  | is-STRIPS-fmla ($\neg \bot$) ←→ True
  | is-STRIPS-fmla - ←→ False

lemma aux1: [wm-basic $M$; is-STRIPS-fmla $\varphi$; valuation $M \models \varphi$; $\forall G\in M. A \models G$] $\implies A \models \varphi$
  apply(induction $\varphi$ rule: is-STRIPS-fmla.induct)
  by (auto simp: valuation-def)

lemma aux2: [wm-basic $M$; is-STRIPS-fmla $\varphi$; $\forall A. (\forall G\in M. A \models G) \longrightarrow A \models \varphi$] $\implies$ valuation $M \models \varphi$
  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

lemma valuation-iff-STRIPS:
  assumes $\text{wm-basic } M$
assumes is-STRIPS-fmla \( \varphi \)
shows valuation \( M \models \varphi \) \( \iff \) \( M \models \varphi \)

proof

have aux1: \( \forall A. \ [ \text{valuation } M \models \varphi; \forall G \in M \cdot A \models G ] \implies A \models \varphi \)
using assms apply(induction \( \varphi \) rule: is-STRIPS-fmla.induct)
by (auto simp: valuation-def)

have aux2: \( \forall A. \ [ \forall G \in M \cdot A \models G ] \implies A \models \varphi \) \( \implies \) 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 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

\textbf{theorem} proper-STRIPS-generalization:
\[ [\text{wm-basic } M; \text{is-STRIPS-fmla } \varphi] \implies M \models^c \models^c \varphi \iff M \models \varphi \]
by (simp add: valuation-iff-close-world[symmetric] valuation-iff-STRIPS)

\section{3.4 STRIPS Semantics}

For this section, we fix a domain \( D \), using Isabelle’s locale mechanism.

\textbf{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.

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

Execute a ground action

\textbf{definition} execute-ground-action :: ground-action \( \Rightarrow \) world-model \( \Rightarrow \) world-model
\textbf{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 \( M_0, \ldots M_N \) of intermediate world models, as done in [Lif87].
fun ground-action-path
:: world-model ⇒ ground-action list ⇒ world-model ⇒ bool
where
  ground-action-path M [] M' ←→ (M = M')
| ground-action-path M (α#α s) M' ←→ M s |== precondition α
  ∧ ground-action-path (execute-ground-action α M) as M'

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

lemma ground-action-path-in-paper:
  ground-action-path M [] M' ←→ (M = M')
  ground-action-path M (α#α s) M' ←→ M s |== precondition α
  ∧ (ground-action-path (apply-effect (effect α) M) as 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 ⊆ₚ varT' ⇒ objT ⊆ₚ objT' ⇒
  ty-term varT objT ⊆ₚ ty-term varT' objT'
apply (rule map-leI)
subgoal for x v
  apply (cases x)
  apply (auto dest: map-leD)
done
done
done

calendar ast-domain begin

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

definition sig :: predicate ⇒ type list where
  sig ≡ map-of (map (λPredDecl p n ⇒ (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 ≡ set (map subtype-edge (types D))
**definition** of-type :: type ⇒ type ⇒ bool where

of-type oT T ≡ set (primitives oT) ⊆ 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 ⇒ type — Entity’s type, None if invalid

**begin**

Checks whether an entity has a given type

**definition** is-of-type :: 'ent ⇒ type ⇒ bool where

is-of-type v T ←→ (case ty-ent v of

| Some vT ⇒ of-type vT T |
| None ⇒ False)

**fun** wf-pred-atom :: predicate × 'ent list ⇒ bool where

wf-pred-atom (p,vs) ←→ (case sig p of

| None ⇒ False |
| Some Ts ⇒ 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 ⇒ bool where

wf-atom (predAtm p vs) ←→ wf-pred-atom (p,vs) |

wf-atom (Eq a b) ←→ ty-ent a ≠ None ∧ ty-ent b ≠ None

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

**fun** wf-fmla :: ('ent atom) formula ⇒ bool where

wf-fmla (Atom a) ←→ wf-atom a |

wf-fmla (⊥) ←→ True |

wf-fmla (ϕ1 ∧ ϕ2) ←→ (wf-fmla ϕ1 ∧ wf-fmla ϕ2) |

wf-fmla (ϕ1 ∨ ϕ2) ←→ (wf-fmla ϕ1 ∧ wf-fmla ϕ2) |

wf-fmla (¬ϕ) ←→ wf-fmla ϕ |

wf-fmla (ϕ1 → ϕ2) ←→ (wf-fmla ϕ1 ∧ wf-fmla ϕ2)

**lemma** wf-fmla ϕ = (∀ a∈atoms ϕ. wf-atom a)

**by** (induction ϕ) auto
Special case for a well-formed atomic predicate formula

fun wf-fmla-atom where
  wf-fmla-atom (Atom (predAtm a vs)) ←→ wf-pred-atom (a,vs)
| wf-fmla-atom - ←→ False

lemma wf-fmla-atom-alt: wf-fmla-atom ϕ ←→ is-predAtom ϕ ∧ wf-fmla ϕ
  by (cases ϕ rule: wf-fmla-atom.cases) auto

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

fun wf-effect where
  wf-effect (Effect a d) ←→
  (∀ ae ∈ set a. wf-fmla-atom ae)
∧ (∀ de ∈ set d. wf-fmla-atom de)

end — Context fixing ty-ent

definition constT :: object ⇒ type where
  constT ≡ map-of (consts 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-action-schema :: ast-action-schema ⇒ bool where
  wf-action-schema (Action-Schema n params pre eff) ←→
  (let
tyt = ty-term (map-of params) constT
  in
  distinct (map fst params)
∧ wf-fmla tyt pre
∧ wf-effect tyt eff)

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

fun wf-type where
  wf-type (Either Ts) ←→ set Ts ⊆ insert "object" (fst'set (types D))

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

fun wf-predicate-decl where
  wf-predicate-decl (PredDecl p Ts) ←→ (∀ T∈set Ts. wf-type T)

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

definition wf-types ≡ snd'set (types D) ⊆ 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 ≡
    wf-types ∧ distinct (map (predicate-decl.pred) (predicates D))
∧ (∀ p∈set (predicates D). wf-predicate-decl p)
∧ distinct (map fst (consts D))
∧ (∀ (n,T)∈set (consts D). wf-type T)
∧ distinct (map ast-action-schema.name (actions D))
∧ (∀ a∈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 ≡ ast-problem.domain P

definition objT :: object ⇒ type where
objT ≡ 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 ⇒ 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 = (∀ f∈M. wf-fmla-atom objT f)

definition wf-problem where
wf-problem ≡
    wf-domain
∧ distinct (map fst (objects P) @ map fst (consts D))
∧ (∀ (n, T) ∈ set (objects P). wf-type T)
∧ distinct (init P)
∧ wf-world-model (set (init P))
∧ wf-fmla objT (goal P)

fun wf-effect-inst :: object ast-effect ⇒ bool where
  wf-effect-inst (Effect (a) (d))
  ←→ (∀ a ∈ set a ∪ set d. wf-fmla-atom 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 ⇒ 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 ⇒ object list ⇒ ground-action
where

\begin{align*}
\text{instantiate-action-schema} \ (\text{Action-Schema } n \ \text{params pre eff}) \ \text{args} = (\text{let} \\
\quad \text{tsubst} = \text{subst-term} \ (\text{the o (map-of (zip (map fst params) args)))}; \\
\quad \text{pre-inst} = (\text{map-formula o map-atom}) \ \text{tsubst pre}; \\
\quad \text{eff-inst} = (\text{map-ast-effect}) \ \text{tsubst eff} \\
\text{in} \\
\quad \text{Ground-Action} \ \text{pre-inst eff-inst} \\
\end{align*}

end — Context of ast-domain

context ast-problem begin

Initial model

\begin{align*}
\text{definition} \ I :: \ world-model \ \text{where} \\
I & \equiv \text{set (init P)}
\end{align*}

Resolve a plan action and instantiate the referenced action schema.

\begin{align*}
\text{fun} \ \text{resolve-instantiate} :: \ \text{plan-action} \Rightarrow \ \text{ground-action} \ \text{where} \\
\text{resolve-instantiate} \ (P\text{Action } n \ \text{args}) = \\
\quad \text{instantiate-action-schema} \\
\quad \ (\text{the (resolve-action-schema } n)) \\
\quad \text{args}
\end{align*}

Check whether object has specified type

\begin{align*}
\text{definition} \ \text{is-obj-of-type } n \ T \ \equiv \ \text{case} \ \text{objT } n \ \text{of} \\
\quad \text{None} & \Rightarrow \ \text{False} \\
| \ \text{Some} \ oT & \Rightarrow \ \text{of-type } oT \ T
\end{align*}

We can also use the generic \text{is-of-type} function.

\begin{itemize}
\item \text{lemma} \ \text{is-obj-of-type-alt: is-obj-of-type } = \ \text{is-of-type objT} \\
\item \text{apply} \ (\text{intro ext}) \\
\item \text{unfolding} \ \text{is-obj-of-type-def is-of-type-def by auto}
\end{itemize}

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

\begin{align*}
\text{definition} \ \text{action-params-match } a \ \text{args} \\
& \equiv \ \text{list-all2 is-obj-of-type args (map snd (parameters a))}
\end{align*}

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.

\begin{align*}
\text{fun} \ \text{wf-plan-action} :: \ \text{plan-action} \Rightarrow \ \text{bool} \ \text{where}
\end{align*}
wf-plan-action \((P\text{Action } n \text{ args})\) = (  
  case \text{resolve-action-schema } n \text{ of}  
  None \Rightarrow False  
  \mid \text{Some } a \Rightarrow  
  \text{action-params-match } a \text{ args}  
  \land \text{wf-effect-inst} \left( \text{effect} \left( \text{instantiate-action-schema } a \text{ args} \right) \right)  
)

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.

\text{definition} \text{plan-action-path} :: \text{world-model } \Rightarrow \text{plan-action list } \Rightarrow \text{world-model } \Rightarrow \text{bool}  
\text{where}  
\text{plan-action-path } M \pi s M' =  
(\forall \pi \in \text{set } \pi s. \text{wf-plan-action } \pi)  
\land \text{ground-action-path } M \left( \text{map resolve-instantiate } \pi s \right) M'

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

\text{definition} \text{valid-plan-from} :: \text{world-model } \Rightarrow \text{plan } \Rightarrow \text{bool} \text{ where}  
\text{valid-plan-from } M \pi s = (\exists M'. \text{plan-action-path } M \pi s M' \land M' \models (\text{goal } P))

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

\text{definition} \text{valid-plan} :: \text{plan } \Rightarrow \text{bool}  
\text{where} \text{valid-plan} \equiv \text{valid-plan-from } I

Concise definition used in paper:

\text{lemma} \text{valid-plan } \pi s \equiv \exists M'. \text{plan-action-path } I \pi s M' \land M' \models (\text{goal } P)  
\text{unfolding} \text{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.

\text{context} ast-problem \text{ 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.

\text{fun} \text{wf-ground-action} :: \text{ground-action } \Rightarrow \text{bool} \text{ where}
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, hide-lams)

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

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

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

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

lemma \( \text{is-of-type-var-conv}: \text{is-of-type} \ (\text{ty-term} \ Q \ \text{objT}) \ (\text{term}.\text{VAR} \ x) \ T \iff \text{is-of-type} \ Q \ x \ T \)
unfolding is-of-type-def by (auto)

lemma \( \text{is-of-type-const-conv}: \text{is-of-type} \ (\text{ty-term} \ Q \ \text{objT}) \ (\text{term}.\text{CONST} \ x) \ T \iff \text{is-of-type} \ \text{objT} \ x \ T \)
unfolding is-of-type-def by (auto split: option.splits)

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

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

lemma \( \text{wf-inst-eq-aux'}: \text{ty-term} \ Q \ \text{objT} \ x = \text{Some} \ T \implies \text{objT} \ (\text{subst-term} \ f \ x) \neq \text{None} \)
by (cases x) (auto simp: wf-inst-eq-aux)
lemma wf-inst-atom:
assumes wf-atom \((\text{ty-term } Q \text{ constT})\) a
shows wf-atom objT \((\text{map-atom (subst-term } f)\ a)\)
proof
  have X1: list-all2 \((\text{is-of-type } objT)\) \((\text{map (subst-term } f)\) xs) Ts if
  list-all2 \((\text{is-of-type } (\text{ty-term } Q \text{ objT}))\) 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 \((\text{ty-term } Q \text{ constT})\) a
shows wf-fmla-atom objT \(((\text{map-formula o map-atom o subst-term})\ f\ a)\)
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 \((\text{ty-term } Q \text{ constT})\) \(\varphi\)
shows wf-effect objT \(((\text{map-ast-effect o subst-term})\ f\ \varphi)\)
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 \((\text{ty-term } Q \text{ constT})\) \(\varphi\)
shows wf-fmla objT \(((\text{map-formula o map-atom o subst-term})\ f\ \varphi)\)
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 \([\text{simp}]\): (Action-Schema name params pre eff)
have INST:
-is-of-type \( \text{objT} \ (\text{the} \circ \text{map-of} \ (\text{zip (map fst params) args}) \ x) \ T \)
if \is-of-type\ (\text{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 \( i \ x T \)
unfolding action-params-match-def
apply (subst lookup-zip-idx-eq[where \( i = i \)];
在玩家中 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 \( \pi \ M \) \iff \( \text{wf-plan-action} \ \pi \ \land \ M \models \text{precondition} \ (\text{resolve-instantiate} \ \pi) \)

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 \( \pi \ M \) = (\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' \) \iff \( M' = M \)
by (auto simp: plan-action-path-def)

lemma plan-action-path-Cons[simp]; plan-action-path \( \pi \# \pi s \) \( M' \)

plan-action-enabled \( \pi M \)
\( \land \) plan-action-path \( (\text{execute-plan-action} \ \pi M) \pi s M' \)
by (auto

\( \text{simp: plan-action-path-def execute-plan-action-def execute-ground-action-def plan-action-enabled-def} \))

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 \( \text{objT} e \)
assumes wf-world-model \( s \)
shows wf-world-model \( (\text{apply-effect} \ e \ s) \)
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 \( \pi s \)
assumes wf-world-model \( s \)
shows wf-world-model \( (\text{execute-plan-action} \ \pi s) \)
using assms

proof (cases \( \pi \))

case [simp]: \( (P\text{Action name args}) \)

from \( \text{plan-action-enabled} \ \pi s \) have \( \text{wf-plan-action} \ \pi \)

unfolding plan-action-enabled-def by auto

then obtain \( a \) where
resolve-action-schema name = Some \( a \) and
\( T: \text{action-params-match} \ a \ \text{args} \)
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 \( \text{resolve-action-schema name} = \text{Some} \ a \) have
a ∈ 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 π)
using (resolve-action-schema name = Some a: 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 π; simp)
by (rule (wf-world-model s))
qed

theorem wf-execute-compact-notation:
plan-action-enabled π s ⇒ wf-world-model s
⇒ wf-world-model (execute-plan-action π 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 π s M'
shows wf-world-model M'
using assms
by (induction π 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
4.1 Generic DFS Reachability Checker

Used for subtype checks

definition E-of-succ succ ≡ \{(u,v). v∈set (succ u)\}
lemma succ-as-E: set (succ x) = E-of-succ succ "x"
  unfolding E-of-succ-def by auto

context
  fixes succ :: 'a ⇒ 'a list
begin

  private abbreviation (input) E ≡ E-of-succ succ

definition dfs-reachable D w ≡
  let (V,w,brk) = while (λ(V,w,brk). ¬brk ∧ w≠[])) (λ(V,w,−)).
  case w of v#
    if D v then (V,v#w, True)
    else if v∈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 ≡ set w₀

definition dfs-reachable-invar D V W brk ⟷
  W₀ ⊆ W ∪ V
  ∧ W ∪ V ⊆ E* " W₀
  ∧ E"V ⊆ W ∪ V
  ∧ Collect D ∩ V = {}
  ∧ (brk ⟷ Collect D ∩ E* " W₀ ≠ {})

lemma card-decreases:
  \[finite V; y ∉ V; dfs-reachable-invar D V (Set.insert y W) brk \]
  ⟷ card (E* " W₀ − Set.insert y V) < card (E* " W₀ − V)
apply (rule psubset-card-mono)
apply (auto simp: dfs-reachable-invar-def)

34
lemma all-neq-Cons-is-Nil[simp]:
(∀ y ys. x2 ≠ y # ys) ⟷ x2 = [] by (cases x2) auto

lemma dfs-reachable-correct: dfs-reachable D w0 ⟷ Collect D ∩ E∗ ⊆ set w0 ≠ {}
unfolding dfs-reachable-def
apply (rule while-rule)[where
P=λ(V,w,brk). dfs-reachable-invar D V (set w) brk ∧ finite V
and r=measure (λV. card (E∗ ⊆ (set w0) − V)) <∗lex*> measure length
<∗lex*> measure (λTrue⇒0 | False⇒1)
]
subgoal by (auto simp: dfs-reachable-invar-def)
subgoal by (auto simp: neq-Nil-conv succ-as-E)
by (auto simp: dfs-reachable-invar-def Image-iff intro: rtrancl rtrancl-into-rtrancl)
subgoal by blast
subgoal by (auto simp: neq-Nil-conv card-decreases)
done

definition tab-succ l ≡ Mapping.lookup-default [] (fold (λ(u,v). Mapping.map-default u [] (Cons v)) l Mapping.empty)

lemma Some-eq-map-option [iff]: (Some y = map-option f xo) = (∃ z. xo = Some z ∧ f z = y)
by (auto simp add: map-option-case split: option.split)

lemma tab-succ-correct: E-of-succ (tab-succ l) = set l
proof –
  have set (Mapping.lookup-default [] (fold (λ(u,v). Mapping.map-default u [] (Cons v)) l m) u) = set l " {u} ∪ set (Mapping.lookup-default [] m u)
    for m u
  apply (induction l arbitrary: m)
  by (auto
    simp: lookup-map-entry' lookup-update' keys-is-none-rep Option.is-none-def
    split: if-splits)
  from this[where m=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:
\[ \text{finite } E; \text{finite } S \implies \text{finite } (E^{*}''S) \]
apply (rule finite-subset\[where \text{B=S} \cup (\text{Relation.Domin}_E \cup \text{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 vs\_0 \iff Collect D \cap (set l)^*''set vs\_0 \neq \{\}
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 \equiv (\forall pt \in set \text{ (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 \iff pt \in subtype-rel^{*}'' set (primitives T)
lemma of-type-refine1: of-type oT T \iff (\forall pt \in set \text{ (primitives oT)}. of-type1 pt T)
  unfolding of-type-def of-type1-def by auto

definition STG \equiv (tab-succ (map subtype-edge \text{ (types D)}))

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

lemma of-type1-impl: of-type1 pt T \iff dfs-reachable (tab-succ (map subtype-edge \text{ (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: \( \text{of-type-impl STG} \ oT \ T \iff \text{of-type} \ oT \ T \)
unfolding of-type1-impl STG-def of-type-impl-def of-type-refine1 ..

definition mp-constT :: (object, type) mapping where
\( mp-constT = \text{Mapping.of-alist (consts} \ D) \)

lemma mp-objT-correct\[\text{simp}\]: \( \text{Mapping.lookup mp-constT} = \text{constT} \)
unfolding mp-constT-def constT-def
by transfer (simp add: Map-To-Mapping.map-apply-def)

Lifting the subtype-graph through wf-checker

context

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

begin

definition is-of-type’ STG v T ←→ (case ty-ent v of
Some vT \ \Rightarrow \ \text{of-type-impl stg vT} T
| None \ \Rightarrow \ \text{False})

lemma is-of-type’-correct: \( \text{is-of-type’ STG} \ v \ T = \text{is-of-type} \ ty-ent \ v \ T \)
unfolding is-of-type’-def is-of-type-def of-type-impl-correct ..

fun wf-pred-atom’ where
wf-pred-atom’ stg (p,vs) ←→ (case sig p of
None \ \Rightarrow \ \text{False}
| Some Ts \Rightarrow \text{list-all2} (is-of-type’ stg) vs Ts)

lemma wf-pred-atom’-correct: \( \text{wf-pred-atom’ STG} \ pvs = \text{wf-pred-atom} \ ty-ent \ pvs \)
by (cases pvs) (auto simp: is-of-type’-correct[abs-def] split:option.split)

fun wf-atom’ :: - \Rightarrow (ent atom) bool where
wf-atom’ stg (atom.predAtm p vs) ←→ \text{wf-pred-atom’ stg} (p,vs)
| \text{wf-atom’ stg} (atom.Eq a b) = (ty-ent a \not= None \land ty-ent b \not= None)

lemma wf-atom’-correct: \( \text{wf-atom’ STG} \ a = \text{wf-atom} ty-ent \ a \)
by (cases a) (auto simp: \text{wf-pred-atom’-correct is-of-type’-correct[abs-def] split:option.split})

fun wf-fmla’ :: - \Rightarrow (ent atom) formula \Rightarrow bool where
wf-fmla’ stg (Atom a) ←→ \text{wf-atom’ stg} a
| \text{wf-fmla’ stg} \bot \iff \text{True}
| \text{wf-fmla’ stg} (\varphi_1 \land \varphi_2) \iff (\text{wf-fmla’ stg} \varphi_1 \land \text{wf-fmla’ stg} \varphi_2)
| \text{wf-fmla’ stg} (\varphi_1 \lor \varphi_2) \iff (\text{wf-fmla’ stg} \varphi_1 \land \text{wf-fmla’ stg} \varphi_2)
| \text{wf-fmla’ stg} (\varphi_1 \rightarrow \varphi_2) \iff (\text{wf-fmla’ stg} \varphi_1 \land \text{wf-fmla’ stg} \varphi_2)
| \text{wf-fmla’ stg} (\neg \varphi) \iff \text{wf-fmla’ stg} \varphi

lemma wf-fmla’-correct: \( \text{wf-fmla’ STG} \varphi \iff \text{wf-fmla} ty-ent \varphi \)
by (induction \( \varphi \) rule: wf-fmla.induct) (auto simp: wf-atom'\-correct)

fun \texttt{wf-fmla-atom1'} where
\(
\texttt{wf-fmla-atom1'} \; \texttt{stg} \; (\texttt{Atom} (\texttt{predAtm} \; p \; \texttt{vs})) \quad \quad \textbf{\iff} \quad \texttt{wf-pred-atom'} \; \texttt{stg} \; (p, \texttt{vs})
\)
| \texttt{wf-fmla-atom1'} \; \texttt{stg} \; \textbf{\iff} \quad \texttt{False}

lemma \texttt{wf-fmla-atom1'}\-correct: \texttt{wf-fmla-atom1'} \; \texttt{STG} \; \varphi = \texttt{wf-fmla-atom ty-ent} \; \varphi
by (cases \( \varphi \) rule: \texttt{wf-fmla-atom}.cases) (auto simp: wf-atom'\-correct is-of-type'\-correct [abs-def] split: option.splits)

fun \texttt{wf-effect'} where
\(
\texttt{wf-effect'} \; \texttt{stg} \; (\texttt{Effect} \; a \; d) \quad \textbf{\iff} \quad (\forall \; a e \in \texttt{set} \; a. \; \texttt{wf-fmla-atom1'} \; \texttt{stg} \; ae) \\
\wedge (\forall \; d e \in \texttt{set} \; d. \; \texttt{wf-fmla-atom1'} \; \texttt{stg} \; de)
\)

lemma \texttt{wf-effect'}\-correct: \texttt{wf-effect'} \; \texttt{STG} \; e = \texttt{wf-effect ty-ent} \; e
by (cases e) (auto simp: \texttt{wf-fmla-atom1'}\-correct)

end — Context fixing \( \texttt{ty-ent} \)

fun \texttt{wf-action-schema'} :: \texttt{-} \Rightarrow \texttt{-} \Rightarrow \texttt{ast-action-schema} \Rightarrow \texttt{bool} where
\(
\texttt{wf-action-schema'} \; \texttt{stg} \; \texttt{conT} \; (\texttt{Action-Schema} \; n \; \texttt{params} \; \texttt{pre} \; \texttt{eff}) \quad \textbf{\iff} \quad (\begin{aligned}
& \texttt{let} \\
& \quad \texttt{tyv} = \texttt{ty-term'} \; (\texttt{map-of} \; \texttt{params}) \; \texttt{conT} \\
& \quad \texttt{in} \\
& \quad \texttt{distinct} \; (\texttt{map} \; \texttt{fst} \; \texttt{params}) \\
& \quad \wedge \; \texttt{wf-fmla'} \; \texttt{tyv} \; \texttt{stg} \; \texttt{pre} \\
& \quad \wedge \; \texttt{wf-effect'} \; \texttt{tyv} \; \texttt{stg} \; \texttt{eff}
\end{aligned})
\)

lemma \texttt{wf-action-schema'}\-correct: \texttt{wf-action-schema'} \; \texttt{STG} \; \texttt{mp-constT} \; s = \texttt{wf-action-schema} \; s
by (cases s) (auto simp: \texttt{wf-fmla'}\-correct \texttt{wf-effect'}\-correct)

definition \texttt{wf-domain'} :: \texttt{-} \Rightarrow \texttt{-} \Rightarrow \texttt{bool} where
\(
\texttt{wf-domain'} \; \texttt{stg} \; \texttt{conT} \equiv \quad \begin{aligned}
& \texttt{wf-types} \\
& \wedge \; \texttt{distinct} \; (\texttt{map} \; \texttt{predicate-decl}.\texttt{pred}) \; \texttt{predicates} \; \texttt{D}) \\
& \wedge (\forall p \in \texttt{set} \; \texttt{predicates} \; \texttt{D}). \; \texttt{wf-predicate-decl} \; p \\
& \wedge \; \texttt{distinct} \; (\texttt{map} \; \texttt{fst} \; \texttt{consts} \; \texttt{D}) \\
& \wedge (\forall (n, T) \in \texttt{set} \; \texttt{consts} \; \texttt{D}). \; \texttt{wf-type} \; T \\
& \wedge \; \texttt{distinct} \; (\texttt{map} \; \texttt{ast-action-schema}.\texttt{name} \; \texttt{actions} \; \texttt{D}) \\
& \wedge (\forall a \in \texttt{set} \; \texttt{actions} \; \texttt{D}). \; \texttt{wf-action-schema'} \; \texttt{stg} \; \texttt{conT} \; a
\end{aligned}
\)

lemma \texttt{wf-domain'}\-correct: \texttt{wf-domain'} \; \texttt{STG} \; \texttt{mp-constT} = \texttt{wf-domain}
unfolding \texttt{wf-domain-def} \texttt{wf-domain'}\-def
by (auto simp: \texttt{wf-action-schema'}\-correct)
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 ~⇒~ world-model ⇒ world-model

where

apply-effect-exec (Effect a d) s
  = fold (λadd s. Set.insert add s) a
   (fold (λdel 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 framework 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 ⇒ False | Some oT ⇒ of-type-impl stg oT)

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 ⇒ - ⇒ fact ⇒ bool
  where
    wf-fact' ot stg ≡ 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) ⇒ (wf-fact' mp stg (p,vs)) | - ⇒ False)

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

definition wf-problem' stg conT mp ≡
  wf-domain' stg conT
  ∧ distinct (map fst (objects P) @ map fst (consts D))

\[ \forall (n, T) \in \text{set (objects } P) \text{. } \text{wf-type } T \] 
\[ \land \text{ distinct (init } P) \] 
\[ \land \left( \forall f \in \text{set (init } P) \text{. } \text{wf-fmla-atom2'} mp \text{ stg } f \right) \] 
\[ \land \text{ wf-fmla'} (\text{Mapping.lookup mp}) \text{ stg } (\text{goal } P) \]

**lemma** \( \text{wf-problem'}-\text{correct} \):

\[ \text{wf-problem'} \text{ STG mp-constT mp-objT } = \text{wf-problem} \]

**unfolding** \( \text{wf-problem-def} \) \( \text{wf-problem'}-\text{def} \) \( \text{wf-world-model-def} \)

**by** (auto simp: \( \text{wf-domain'}-\text{correct} \) \( \text{wf-fmla'}-\text{correct} \))

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

**lemma** \( \text{wf-effect-inst-weak} \):

**fixes** a args

**defines** ai \( \equiv \text{instantiate-action-schema a args} \)

**assumes** A: \( \text{action-params-match a args} \)

**wf-action-schema a**

**shows** \( \text{wf-effect-inst (effect ai)} \)

**using** \( \text{wf-instantiate-action-schema[OF A]} \) **unfolding** ai-def[symmetric]

**by** (cases ai) (auto simp: \( \text{wf-effect-inst-alt} \))

**end** — Context of \( \text{ast-problem} \)

**context** \( \text{wf-ast-domain begin} \)

Resolving an action yields a well-founded action schema.

**lemma** \( \text{resolve-action-uf} \):

**assumes** resolve-action-schema \( n = \text{Some } a \)

**shows** \( \text{wf-action-schema a} \)

**proof** —

**from** \( \text{wf-domain have} \)

\( \text{X1: distinct } (\text{map ast-action-schema.name (actions } D)) \)

\( \text{and } \text{X2: } \forall a \in \text{set (actions } D) \text{. } \text{wf-action-schema a} \)

**unfolding** \( \text{wf-domain-def} \) **by** auto

**show** ?thesis

**using** \( \text{assms unfolding resolve-action-schema-def} \)

**by** (auto simp add: \( \text{index-by-eq-Some-eq[OF X1] X2} \))

**qed**

**end** — Context of \( \text{ast-domain} \)

**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.

\[
\text{context ast-domain begin}
\begin{align*}
\text{definition} &\quad \text{resolve-action-schemaE n} \equiv \\
&\quad \text{lift-opt} \\
&\quad (\text{resolve-action-schema n}) \\
&\quad (\text{ERR (shows "No such action schema" o shows n)})
\end{align*}
\text{end — Context of ast-domain}
\]

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.

\[
\text{type-synonym state = (predicate \times \text{object list}) valuation}
\]

context ast-domain begin

An action is a partial function from states to states.

\[
\text{type-synonym action} = \text{state} \rightarrow \text{state}
\]

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]

\[
\text{fun is-NegPredAtom where} \\
\quad \text{is-NegPredAtom (Not x) = is-predAtom x | is-NegPredAtom - = False}
\]

\[
\text{definition close-eq s} = (\lambda \text{predAtm p xs} \Rightarrow s (p, xs) | \text{Eq a b} \Rightarrow a = b)
\]

\[
\text{lemma close-eq-predAtm[simp]: close-eq s (predAtm p xs) \iff s (p, xs)}
\]
\by (auto simp: close-eq-def)

\[
\text{lemma close-eq-Eq[simp]: close-eq s (Eq a b) \iff a = b}
\]
\by (auto simp: close-eq-def)

abbreviation entail-eq :: \text{state} \Rightarrow \text{object atom formula} \Rightarrow \text{bool} (\text{infix} \models) \\
\quad \text{where} \quad \text{entail-eq s f} \equiv \text{close-eq s} \models f
\]

\[
\text{fun sound-opr :: ground-action} \Rightarrow \text{action} \Rightarrow \text{bool where} \\
\quad \text{sound-opr (Ground-Action pre (Effect add del)) f}
\]

42
\[
(\forall s. s \vdash_{=} \text{pre} \longrightarrow \\
(\exists s'. f s = \text{Some } s' \land (\forall \text{atm. is-predAtom atm} \land \text{atm} \not\in \text{set del} \land s \vdash_{=} \text{atm} \longrightarrow s' \vdash_{=} \text{atm}) \\
\land (\forall \text{atm. is-predAtom atm} \land \text{atm} \not\in \text{set add} \land s \vdash_{=} \text{Not atm} \longrightarrow s' \\
\vdash_{=} \text{Not atm}) \\
\land (\forall \text{fmla. fmla} \in \text{set del} \longrightarrow s' \vdash_{=} \text{fmla}) \\
\land (\forall \text{fmla. fmla} \in \text{set del} \land \text{fmla} \not\in \text{set add} \longrightarrow s' \vdash_{=} \text{Not fmla}))
\])
\land (\forall \text{fmla} \in \text{set add. is-predAtom fmla})
\]

**Lemma** sound-opr-alt:

\[
\text{sound-opr opr f} = \\
((\forall s. s \vdash_{=} (\text{precondition opr}) \longrightarrow \\
(\exists s'. f s = \text{Some } s' \\
\land (\forall \text{atm. is-predAtom atm} \land \text{atm} \not\in \text{set del (effect opr)}) \land s \vdash_{=} \text{atm} \longrightarrow s' \vdash_{=} \text{atm}) \\
\land (\forall \text{atm. is-predAtom atm} \land \text{atm} \not\in \text{set add (effect opr)}) \land s \vdash_{=} \text{Not atm} \longrightarrow s' \vdash_{=} \text{Not atm}) \\
\land (\forall \text{atm. atm} \in \text{set (adds (effect opr))} \longrightarrow s' \vdash_{=} \text{atm}) \\
\land (\forall \text{fmla. fmla} \in \text{set (dels (effect opr))} \land \text{fmla} \not\in \text{set (adds (effect opr))} \longrightarrow s' \vdash_{=} \text{Not fmla}))
\])
\land (\forall \text{fmla} \in \text{set (adds (effect opr)). is-predAtom fmla})
\]

*by (cases (opr f) rule: sound-opr.cases) auto

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

**Definition** sound-system

\[
\Rightarrow \text{ground-action set} \\
\Rightarrow \text{world-model} \\
\Rightarrow \text{state} \\
\Rightarrow (\text{ground-action } \Rightarrow \text{action}) \\
\Rightarrow \text{bool}
\]

**where**

\[
\text{sound-system } \Sigma M_0 s_0 f \equiv \\
(\text{((fmla} \in \text{close-world } M_0. s_0 \vdash_{=} \text{fmla}) \\
\land \text{wm-basic } M_0 \\
\land (\forall \alpha \in \Sigma. \text{sound-opr } \alpha (f \alpha)))
\]

Composing two actions

**Definition** compose-action :: action \Rightarrow action \Rightarrow action where

\[
\text{compose-action } f1 f2 x = (\text{case } f2 x \text{ of Some } y \Rightarrow f1 y \mid \text{None } \Rightarrow \text{None})
\]

Composing a list of actions

**Definition** compose-actions :: action list \Rightarrow action where

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

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' ==> compose-actions (f#fs) s = compose-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 = 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 Σ M0 s0 f
  — For a sound system Σ
  assumes set αs ⊆ Σ
  — And a plan αs
  assumes ground-action-path M0 αs M'
  — Which is accepted by the system, yielding result M' (called R(αs) in Lifschitz's paper [4].)
  obtains s'
  — We have that f(αs) is applicable in initial state, yielding state s' (called fαs(s0) in Lifschitz's paper [4].)
  where compose-actions (map f αs) s0 = Some s'
  — The result world model M' is satisfied in state s'
  and ∀ fmla ∈ close-world M'. s' |= fmla
proof
  have (valuation M' |= fmla) if wm-basic M' fmla∈M' for fmla
    using that apply (induction fmla)
    by (auto simp: valuation-def wm-basic-def split: atom.split)
  have ∃ s'. compose-actions (map f αs) s0 = Some s' ∧ (∀ fmla ∈ close-world M'). s' |= fmla
    using assms
proof(induction αs arbitrary: s0 M0)
  case Nil
  then show ?case by (auto simp add: close-world-def compose-action-def sound-system-def)
next
  case ass: (Cons α αs)
  then obtain pre add del where a: α = Ground-Action pre (Effect add del)
    using ground-action.exhaust ast-effect.exhaust by metis
  let ?M1 = execute-ground-action α M0
  have close-world M0 |= precondition α
    using ass(4)
    by auto
moreover have s0-ent-cwM0: ∀ fmla∈(close-world M_0). close-eq s_0 ⊨ fmla
  using ass(2)
  unfolding sound-system-def
  by auto
ultimately have s0-ent-alpha-precond: close-eq s_0 ⊨ precondition α
  unfolding entailment-def
  by auto
then obtain s_1 where s1: (∀ α s_0 = Some s_1
  (∀ atm. is-predAtom atm → atm /∈ set(dels (effect α))
    → close-eq s_0 ⊨ atm
    → close-eq s_1 ⊨ atm)
  (∀ stmt. stmt ∈ set(adds (effect α))
    → close-eq s_1 ⊨ stmt)
  (∀ atm. is-predAtom atm ∧ atm /∈ set (adds (effect α)) ∧ close-eq s_0 ⊨ Not atm
    → close-eq s_1 ⊨ Not atm)
  (∀ stmt. stmt ∈ set (dels (effect α)) ∧ stmt /∈ set (adds (effect α)) → close-eq s_1 ⊨ stmt)
  (∀ a b. close-eq s_0 ⊨ Atom (Eq a b) → close-eq s_1 ⊨ Atom (Eq a b))
  (∀ a b. close-eq s_0 ⊨ Not (Atom (Eq a b)) → close-eq s_1 ⊨ Not (Atom (Eq a b))))
  using ass(2−4)
  unfolding sound-system-def sound-opr-alt by force
have close-eq s_1 ⊨ stmt if stmt∈close-world ?M_1 for stmt
  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 stmt.ast-effect.sel(1) ast-effect.sel(2) formula-semantics.simps(3)
    in-close-world-conv is-predAtom.simps(1))
done
moreover have (∀ atm. stmt ≠ formula.Atom atm) → s ⊨ stmt if stmt∈?M_1
for stmt s
proof−
  have alpha: (∀ s.∀ stmt∈set(adds (effect α)). ¬ is-predAtom stmt → s ⊨ 
    stmt)
    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.prems(1)[unfolded sound-system-def]
  by(cases stmt; fastforce simp: wm-basic-def)
qed

moreover have \((\forall \text{opr} \in \Sigma. \text{sound-opr opr (f opr))}\)

using ass(2) unfolding sound-system-def
by (auto simp add:)

moreover have \(\text{wm-basic} \ ?M_1\)
using ass(2,3)
unfolding sound-system-def execute-ground-action-def
thm sound-opr.cases
apply (cases (\alpha.f \alpha) rule: sound-opr.cases)
apply (auto simp: wm-basic-def)
done

ultimately have \(\text{sound-system} \ \Sigma \ ?M_1 \ s_1 \ f\)
unfolding sound-system-def
by (auto simp: wm-basic-def)

from ass.IH[\{OF this\}] ass.prems obtain \(s'\) where
compose-actions (map f as) \(s_0 = \text{Some} \ s' \land (\forall a \in \text{close-world} \ M', s' \models a)\)
by auto
thus \(?\text{case}\) by (auto simp: s1(1))
qed
with that show \(?\text{thesis}\) by blast
qed

More compact notation of the soundness theorem.

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

end — Context of \text{ast-domain}

5.1 Soundness Theorem for PDDL

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

lemma \text{wm-to-state-eq}[simp]: \text{wm-to-state} M (p, as) \longleftrightarrow \text{Atom (predAtm p as)} \in M
by (auto simp: \text{wm-to-state-def})
Mapping AST action instances to actions

definition pddl-opr-to-act g-opr s = (let M = state-to-wm s in
  if (wm-to-state (close-world M)) |= (precondition g-opr) then
    Some (wm-to-state (apply-effect (effect g-opr) M))
  else
    None)

definition close-eq-M M = (M ∩ {Atom (predAtm p xs) | p xs. True } ∪ {Atom (Eq a a) | a. True} ∪ {¬(Atom (Eq a b)) | a b. a≠b})

lemma atom-in-wm-eq:
s |= (formula.Atom atm)
  ←→ ((formula.Atom atm) ∈ 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) |= (formula.Atom atm)
  ←→ ((formula.Atom atm) ∈ close-eq-M M)
by (auto simp: wm-to-state-def state-to-wm-def image-def close-eq-M-def close-eq-def split: atom.splits)

lemma not-dels-preserved:
  assumes f /∈ (set d) f ∈ M
  shows f ∈ apply-effect (Effect a d) M
  using assms
  by auto

lemma adds-satisfied:
  assumes f ∈ (set a)
  shows f ∈ apply-effect (Effect a d) M
  using assms
  by auto

lemma dels-unsatisfied:
  assumes f ∈ (set d) f /∈ set a
  shows f /∈ apply-effect (Effect a d) M
  using assms
  by auto

lemma dels-unsatisfied-2:
  assumes f ∈ set (dels eff) f /∈ set (adds eff)
  shows f /∈ 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-atm.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 \in 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 \texttt{assms}
by \{cases \texttt{eff}; \texttt{cases fmla rule: wf-fmla-atom.cases}; \texttt{force}\}
done

\begin{verbatim}
lemma \texttt{wf-eff-impt-wf-eff-inst}: \texttt{wf-effect objT eff \rightarrow wf-effect-inst eff}
by \{cases \texttt{eff}; \texttt{auto simp add: wf-fmla-atom-alt}\}

lemma \texttt{wf-pddl-ground-act-is-sound-opr}:
  \texttt{assumes wf-ground-action g-opr}
  \texttt{shows sound-opr g-opr (pddl-opr-to-act g-opr)}
  \texttt{using \texttt{wf-eff-impt-wf-eff-inst \texttt{wf-eff-pddl-ground-act-is-sound-opr \texttt{assms}}}}
  \texttt{by \{cases \texttt{g-opr}; \texttt{auto}\}}

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

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

lemma \texttt{wf-plan-act-is-sound'}:
  \texttt{assumes \texttt{wf-plan-action \pi}}
  \texttt{shows sound-opr}
  (\texttt{resolve-instantiate \pi})
  (\texttt{pddl-opr-to-act (resolve-instantiate \pi)})
  \texttt{using \texttt{assms \texttt{wf-plan-act-is-sound}}} 
  \texttt{by \{cases \pi; \texttt{auto }\}}

lemma \texttt{wf-world-model-has-atoms}: \texttt{f \in M \rightarrow wf-world-model M \Rightarrow is-predAtom f}
using \texttt{wf-fmla-atm-is-atom}
unfolding \texttt{wf-world-model-def}
by \texttt{auto}
\end{verbatim}
lemma \textit{wm-to-state-works-for-wf-wm-closed}:
\begin{align*}
\text{wf-world-model } M &\implies \text{fmla} \subseteq \text{close-world } M \implies \text{close-eq (wm-to-state } M) \\
\\text{apply} \ (\text{cases fmla rule: wf-fmla-atom.cases}) &\implies \text{by (auto simp: wf-world-model-def close-eq-def wm-to-state-def close-world-def)}
\end{align*}

lemma \textit{wm-to-state-works-for-wf-wm}:
\begin{align*}
\text{wf-world-model } M &\implies \text{fmla} \subseteq M \implies \text{close-eq (wm-to-state } M) \\
\\text{apply} \ (\text{cases fmla rule: wf-fmla-atom.cases}) &\implies \text{by (auto simp: wf-world-model-def close-eq-def wm-to-state-def)}
\end{align*}

lemma \textit{wm-to-state-works-for-I-closed}:
\begin{align*}
\text{assumes } x &\in \text{close-world } I \\
\text{shows } \text{close-eq (wm-to-state } I) &\equiv x \\
\text{apply} \ (\text{rule: wm-to-state-works-for-wf-wm-closed}) &\implies \text{using asms wf-I by auto}
\end{align*}

lemma \textit{wf-wm-imp-basic}:
\begin{align*}
\text{wf-world-model } M &\implies \text{wm-basic } M \\
\text{by (auto simp: wf-world-model-def close-eq-def wf-fmla-atm-is-atom)}
\end{align*}

theorem \textit{wf-plan-sound-system}:
\begin{align*}
\text{assumes } \forall \pi \in \text{set } \pi s. \text{wf-plan-action } \pi \\
\text{shows } \text{sound-system} &\equiv \text{close-opr (wm-to-state } I) \\
\text{unfolding } \text{sound-system-def} &\implies \text{by (intro conjI ballI)}
\end{align*}

proof
\begin{align*}
\text{show } \text{close-opr (wm-to-state } I) &\equiv x \text{ if } x \in \text{close-world } I \text{ for } x \text{ using that [unfolded in-close-world-cone]} \\
\text{wm-to-state-works-for-I-closed &wm-to-state-works-for-wf-wm} &\implies \text{by (auto simp: wf-I)}
\end{align*}

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

\text{show} \text{sound-opr } \alpha (\text{pddl-opr-to-act } \alpha) \text{ if } \alpha \in \text{set (map resolve-instantiate } \pi s) \text{ for } \alpha \text{ using that using wf-plan-act-is-sound’ asms by auto}

qed

theorem \textit{wf-plan-soundness-theorem}:
\begin{align*}
\text{assumes } \text{plan-action-path } I \pi s M \\
\text{defines } \alpha s &\equiv \text{map (pddl-opr-to-act } \circ \text{ resolve-instantiate } \pi s
\end{align*}
defines \( s_0 \equiv \text{wm-to-state I} \)

shows \( \exists s'. \text{compose-actions as } s_0 = \text{Some } s' \land (\forall \varphi \in \text{close-world } M. \ s' \models \varphi) \)

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


