

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	6
2.1	Auxiliary Lemmas	6
2.2	Well-formedness Check	7
2.3	Execution	7
3	PDDL and STRIPS Semantics	11
3.1	Utility Functions	12
3.2	Abstract Syntax	12
3.2.1	Generic Entities	12
3.2.2	Domains	13
3.2.3	Problems	13
3.2.4	Plans	14
3.2.5	Ground Actions	14
3.3	Closed-World Assumption, Equality, and Negation	14
3.3.1	Proper Generalization	16

*Author names are alphabetically ordered.

3.4	STRIPS Semantics	16
3.5	Well-Formedness of PDDL	17
3.6	PDDL Semantics	21
3.7	Preservation of Well-Formedness	23
3.7.1	Well-Formed Action Instances	23
3.7.2	Preservation	26
4	Executable PDDL Checker	27
4.1	Generic DFS Reachability Checker	27
4.2	Implementation Refinements	29
4.2.1	Of-Type	29
4.2.2	Application of Effects	32
4.2.3	Well-Formedness	32
4.2.4	Execution of Plan Actions	34
4.2.5	Checking of Plan	36
4.3	Executable Plan Checker	37
4.4	Code Setup	38
4.4.1	Code Equations	38
4.4.2	Setup for Containers Framework	40
4.4.3	More Efficient Distinctness Check for Linorders	40
4.4.4	Code Generation	40
5	Soundness theorem for the STRIPS semantics	40
5.1	Soundness Theorem for PDDL	43

```

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

```

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

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

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

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

end

locale *wf-ast-problem* = *ast-problem* +
 \quad **assumes** *wf*: *well-formed*

begin

lemma *wf-initial*:
 $\quad length\ astI = numVars$
 $\quad \forall x < numVars. astI!x < numVals\ x$
 $\quad \langle proof \rangle$

lemma *wf-goal*: *wf-partial-state astG*
 $\quad \langle proof \rangle$

lemma *wf-operators*:
 $\quad distinct\ (map\ fst\ ast\delta)$
 $\quad \forall \pi \in set\ ast\delta. wf\text{-}operator\ \pi$
 $\quad \langle proof \rangle$

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
 $map\ (\lambda(-,v,vpre,-). (v,the\ vpre))$
 $(filter\ (\lambda(-,v,vpre,-). vpre \neq None)\ effs)$

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 $I\text{-}valid: I \in valid\text{-}states$
 $\langle proof \rangle$

lemma $lookup\text{-}operator\text{-}wf:$
assumes $lookup\text{-}operator\ name = Some\ \pi$
shows $wf\text{-}operator\ \pi\ fst\ \pi = name$
 $\langle proof \rangle$

lemma $execute\text{-}preserves\text{-}valid:$
assumes $s \in valid\text{-}states$
assumes $enabled\ name\ s$
shows $execute\ name\ s \in valid\text{-}states$
 $\langle proof \rangle$

lemma $path\text{-}to\text{-}pres\text{-}valid:$
assumes $s \in valid\text{-}states$
assumes $path\text{-}to\ s\ \pi s\ s'$
shows $s' \in valid\text{-}states$
 $\langle proof \rangle$

end

end

theory $SASP\text{-}Checker$

imports $SASP\text{-}Semantics$

$HOL\text{-}Library.Code\text{-}Target\text{-}Nat$

begin

2 An Executable Checker for Multi-Valued Planning Problem Solutions

2.1 Auxiliary Lemmas

lemma $map\text{-}of\text{-}leI:$
assumes $distinct\ (map\ fst\ l)$
assumes $\bigwedge k\ v. (k,v) \in set\ l \implies m\ k = Some\ v$

shows $\text{map-of } l \subseteq_m m$
 $\langle \text{proof} \rangle$

lemma [*simp*]: $\text{fst} \circ (\lambda(a, b, c, d). (f a b c d, g a b c d)) = (\lambda(a,b,c,d). f a b c d)$
 $\langle \text{proof} \rangle$

lemma map-mp : $m \subseteq_m m' \implies m k = \text{Some } v \implies m' k = \text{Some } v$
 $\langle \text{proof} \rangle$

lemma $\text{map-add-map-of-fold}$:
fixes ps **and** $m :: 'a \rightarrow 'b$
assumes $\text{distinct } (\text{map } \text{fst } ps)$
shows $m ++ \text{map-of } ps = \text{fold } (\lambda(k, v) m. m(k \mapsto v)) ps m$
 $\langle \text{proof} \rangle$

2.2 Well-formedness Check

lemmas $\text{wf-code-thms} =$
 $\text{ast-problem.astDom-def ast-problem.astI-def ast-problem.astG-def ast-problem.ast}\delta\text{-def}$
 $\text{ast-problem.numVars-def ast-problem.numVals-def}$
 $\text{ast-problem.wf-partial-state-def ast-problem.wf-operator-def ast-problem.well-formed-def}$

declare $\text{wf-code-thms}[\text{code}]$

export-code $\text{ast-problem.well-formed}$ **in** SML

2.3 Execution

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

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

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

definition $\text{enabled-opr}' :: \text{ast-operator} \Rightarrow \text{state} \Rightarrow \text{bool}$ **where**
 $\text{enabled-opr}' \equiv \lambda(\text{name}, pres, effs, cost) s. \text{match-pres } pres s \wedge \text{match-implicit-pres}$
 $effs s$

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

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

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

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

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

context *wf-ast-problem begin*

lemma *match-pres-correct*:
assumes *D*: *distinct* (*map fst pres*)
assumes *s* \in *valid-states*
shows *match-pres pres s* \longleftrightarrow *s* \in *subsuming-states* (*map-of pres*)
 \langle *proof* \rangle

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

lemma *enabled-opr'-correct*:
assumes *V*: *s* \in *valid-states*
assumes *lookup-operator name = Some π*
shows *enabled-opr' π s* \longleftrightarrow *enabled name s*
 \langle *proof* \rangle

lemma *eff-enabled'-correct*:
assumes *V*: *s* \in *valid-states*
assumes *case eff of* (*pres,-,-*) \Rightarrow *wf-partial-state pres*
shows *eff-enabled' s eff* \longleftrightarrow *eff-enabled s eff*
 \langle *proof* \rangle

lemma *execute-opr'-correct*:
assumes *V*: *s* \in *valid-states*
assumes *LO*: *lookup-operator name = Some π*

shows *execute-opr'* $\pi s = \text{execute name } s$
<proof>

lemma *lookup-operator'-correct:*
lookup-operator' problem name = lookup-operator name
<proof>

lemma *enabled'-correct:*
assumes *V: s ∈ valid-states*
shows *enabled' problem name s = enabled name s*
<proof>

lemma *execute'-correct:*
assumes *V: s ∈ valid-states*
assumes *enabled name s*
shows *execute' problem name s = execute name s*
<proof>

end

context *ast-problem*
begin

fun *simulate-plan* :: *plan* \Rightarrow *state* \rightarrow *state* **where**
 simulate-plan [] *s* = *Some s*
 | *simulate-plan* ($\pi \# \pi s$) *s* = (
 if enabled πs *then*
 let *s'* = *execute* πs *in*
 simulate-plan $\pi s s'$
 else
 None
)

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

definition *check-plan* :: *plan* \Rightarrow *bool* **where**
 check-plan $\pi s =$ (
 case simulate-plan πs *I of*
 None \Rightarrow *False*
 | *Some s'* $\Rightarrow s' \in G$)

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

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 = (
  if enabled' problem  $\pi$  s then
    let s = execute' problem  $\pi$  s in
    simulate-plan' problem  $\pi s$  s
  else
    None
)
```

Avoiding duplicate lookup.

```
lemma simulate-plan'-code[code]:
  simulate-plan' problem [] s = 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
  )
<proof>
```

```
definition initial-state' :: ast-problem  $\Rightarrow$  state where
  initial-state' problem  $\equiv$  let astI = ast-problem.astI problem in (
     $\lambda v.$  if  $v < \text{length astI}$  then Some (astI!v) else None
  )
```

```
definition check-plan' :: ast-problem  $\Rightarrow$  plan  $\Rightarrow$  bool where
  check-plan' problem  $\pi s$  = (
    case simulate-plan' problem  $\pi s$  (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  $\pi s$  s = simulate-plan  $\pi s$  s
  <proof>
```

```
lemma simulate-plan'-correct-paper:
  assumes s $\in$ valid-states
  shows simulate-plan' problem  $\pi s$  s = Some s'
```

\longleftrightarrow *path-to s π s'*
 ⟨proof⟩

lemma *initial-state'-correct:*
initial-state' problem = I
 ⟨proof⟩

lemma *check-plan'-correct:*
check-plan' problem π s = check-plan π s
 ⟨proof⟩

end

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 ()
 \longleftrightarrow *ast-problem.well-formed problem ∧ ast-problem.valid-plan problem π s*
 ⟨proof⟩

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*

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 ($x \in \text{set } l \wedge f\ x = n$)
<proof>

lemma *index-by-eq-SomeD*:
shows *index-by f l n* = *Some x* \implies ($x \in \text{set } l \wedge f\ x = n$)
<proof>

lemma *lookup-zip-idx-eq*:
assumes *length params* = *length args*
assumes $i < \text{length } \text{args}$
assumes *distinct params*
assumes $k = \text{params } !\ i$
shows *map-of* (*zip params args*) k = *Some* ($\text{args } !\ i$)
<proof>

lemma *rtrancl-image-idem[simp]*: $R^* \text{ `` } R^* \text{ `` } s = R^* \text{ `` } s$
<proof>

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* $M \equiv \lambda \text{predAtm } p \text{ xs} \Rightarrow \text{Atom } (\text{predAtm } p \text{ xs}) \in M \mid \text{Eq } a \ b \Rightarrow a=b$

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* $M = M \cup \{\neg(\text{Atom } (\text{predAtm } p \text{ as})) \mid p \text{ as. } \text{Atom } (\text{predAtm } p \text{ as}) \notin M\} \cup \{\text{Atom } (\text{Eq } a \ a) \mid a. \text{True}\} \cup \{\neg(\text{Atom } (\text{Eq } a \ b)) \mid a \ b. \ a \neq b\}$

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

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

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

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

lemma

close-world-extensive: $M \subseteq \text{close-world } M$ **and**

close-world-idem[simp]: $\text{close-world } (\text{close-world } M) = \text{close-world } M$
(proof)

lemma *in-close-world-conv*:

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

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 $\mathcal{A} = \text{valuation } M$

(proof)

lemma *valuation-aux-2*:

assumes *wm-basic* M

shows $(\forall G \in \text{close-world } M. \text{valuation } M \models G)$

(proof)

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

(proof)

lemma *close-world-imp-val*:

wm-basic $M \implies M \models \varphi \implies \text{valuation } M \models \varphi$

(proof)

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 $\text{valuation } M \models \varphi \longleftrightarrow M \models \varphi$

(proof)

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  $\Rightarrow$  bool where
  is-STRIPS-fmla (Atom (predAtm - -))  $\longleftrightarrow$  True
| is-STRIPS-fmla ( $\perp$ )  $\longleftrightarrow$  True
| is-STRIPS-fmla ( $\varphi_1 \wedge \varphi_2$ )  $\longleftrightarrow$  is-STRIPS-fmla  $\varphi_1 \wedge$  is-STRIPS-fmla  $\varphi_2$ 
| is-STRIPS-fmla ( $\varphi_1 \vee \varphi_2$ )  $\longleftrightarrow$  is-STRIPS-fmla  $\varphi_1 \wedge$  is-STRIPS-fmla  $\varphi_2$ 
| is-STRIPS-fmla ( $\neg \perp$ )  $\longleftrightarrow$  True
| is-STRIPS-fmla -  $\longleftrightarrow$  False

```

```

lemma aux1:  $\llbracket$ wm-basic M; is-STRIPS-fmla  $\varphi$ ; valuation M  $\models$   $\varphi$ ;  $\forall G \in M. \mathcal{A} \models$ 
G $\rrbracket \Rightarrow \mathcal{A} \models \varphi$ 
  <proof>

```

```

lemma aux2:  $\llbracket$ wm-basic M; is-STRIPS-fmla  $\varphi$ ;  $\forall \mathcal{A}. (\forall G \in M. \mathcal{A} \models G) \longrightarrow \mathcal{A} \models$ 
 $\varphi$  $\rrbracket \Rightarrow$  valuation M  $\models \varphi$ 
  <proof>

```

```

lemma valuation-iff-STRIPS:
  assumes wm-basic M
  assumes is-STRIPS-fmla  $\varphi$ 
  shows valuation M  $\models \varphi \longleftrightarrow$  M  $\models \varphi$ 
  <proof>

```

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:
   $\llbracket$ wm-basic M; is-STRIPS-fmla  $\varphi$  $\rrbracket \Rightarrow$  M  $\models \varphi \longleftrightarrow$  M  $\models \varphi$ 
  <proof>

```

3.4 STRIPS Semantics

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

```

locale ast-domain =
  fixes D :: 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 - set d)  $\cup$  (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 $M_0..M_N$ 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* α
 \wedge *ground-action-path* (*execute-ground-action* α *M*) αs *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* α

\wedge (*ground-action-path* (*apply-effect* (*effect* α) *M*) αs *M'*)

<proof>

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'*

<proof>

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* (λ *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

| $wf\text{-fmla } (\varphi 1 \rightarrow \varphi 2) \longleftrightarrow (wf\text{-fmla } \varphi 1 \wedge wf\text{-fmla } \varphi 2)$

lemma $wf\text{-fmla } \varphi = (\forall a \in atoms \varphi. wf\text{-atom } a)$
 ⟨proof⟩

Special case for a well-formed atomic predicate formula

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

lemma $wf\text{-fmla-atom-alt}: wf\text{-fmla-atom } \varphi \longleftrightarrow is\text{-predAtom } \varphi \wedge wf\text{-fmla } \varphi$
 ⟨proof⟩

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**
 $constT \equiv map\text{-of } (consts D)$

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

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

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

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

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

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

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*)
 $\langle \text{proof} \rangle$

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 = ($\forall f \in M. \text{wf-fmla-atom } \text{objT } f$)

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
  ⟨proof⟩

```

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
  ⟨proof⟩

```

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
  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
  ⟨proof⟩

```

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 π s M' =
 (($\forall \pi \in$ set π s. wf-plan-action π)
 \wedge ground-action-path M (map resolve-instantiate π 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 π s = (\exists M'. *plan-action-path* M π s M' \wedge M' $c \models =$ (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* π s \equiv \exists M'. *plan-action-path* I π s M' \wedge M' $c \models =$ (goal P)
 <proof>

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
  ^ 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*
 ⟨*proof*⟩

lemma (in *ast-domain*) *of-type-trans*[*trans*]:
of-type T1 T2 \implies *of-type T2 T3* \implies *of-type T1 T3*
 ⟨*proof*⟩

lemma *is-of-type-map-ofE*:
assumes *is-of-type* (*map-of params*) *x T*
obtains *i xT* **where** $i < \text{length } \text{params}$ $\text{params}!i = (x, xT)$ *of-type xT T*
 ⟨*proof*⟩

lemma *wf-atom-mono*:
assumes *SS*: $\text{tys} \subseteq_m \text{tys}'$
assumes *WF*: *wf-atom tys a*
shows *wf-atom tys' a*
 ⟨*proof*⟩

lemma *wf-fmla-atom-mono*:
assumes *SS*: $\text{tys} \subseteq_m \text{tys}'$
assumes *WF*: *wf-fmla-atom tys a*
shows *wf-fmla-atom tys' a*
 ⟨*proof*⟩

lemma *constT-ss-objT*: $\text{constT} \subseteq_m \text{objT}$
 ⟨*proof*⟩

lemma *wf-atom-constT-imp-objT*: *wf-atom (ty-term Q constT) a* \implies *wf-atom (ty-term Q objT) a*
 ⟨*proof*⟩

lemma *wf-fmla-atom-constT-imp-objT*: *wf-fmla-atom (ty-term Q constT) a* \implies *wf-fmla-atom (ty-term Q objT) a*
 ⟨*proof*⟩

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
 ⟨proof⟩

lemma *is-of-type-const-conv*: *is-of-type (ty-term Q objT) (term.CONST x) T*
 \longleftrightarrow *is-of-type objT x T*
 ⟨proof⟩

lemma *INST'*: *is-of-type (ty-term Q objT) x T* \implies *is-of-type objT (subst-term f x) T*
 ⟨proof⟩

lemma *wf-inst-eq-aux*: *Q x = Some T* \implies *objT (f x) \neq None*
 ⟨proof⟩

lemma *wf-inst-eq-aux'*: *ty-term Q objT x = Some T* \implies *objT (subst-term f x)*
 \neq *None*
 ⟨proof⟩

lemma *wf-inst-atom*:
 assumes *wf-atom (ty-term Q constT) a*
 shows *wf-atom objT (map-atom (subst-term f) a)*
 ⟨proof⟩

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)*
 ⟨proof⟩

lemma *wf-inst-effect*:
 assumes *wf-effect (ty-term Q constT) φ*
 shows *wf-effect objT ((map-ast-effect o subst-term) f φ)*
 ⟨proof⟩

lemma *wf-inst-formula*:
 assumes *wf-fmla (ty-term Q constT) φ*
 shows *wf-fmla objT ((map-formula o map-atom o subst-term) f φ)*
 ⟨proof⟩

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>
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*
<proof>

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'*
<proof>

end — Context of *ast-problem*

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

The initial world model is well-formed

lemma *wf-I*: *wf-world-model* *I*
<proof>

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*)
<proof>

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)

<proof>

theorem *wf-execute-compact-notation:*

plan-action-enabled π $s \implies$ *wf-world-model* s

\implies *wf-world-model* (*execute-plan-action* π s)

<proof>

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'

<proof>

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* $succ \equiv \{ (u,v). v \in set (succ\ u) \}$

lemma *succ-as-E*: $set (succ\ x) = E\text{-of-succ}\ succ\ \{\{x\}\}$
 ⟨proof⟩

context

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

begin

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

definition *dfs-reachable* $D\ w \equiv$

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

context

fixes $w_0 :: 'a\ list$

assumes *finite-dfs-reachable[simp, intro!]*: $finite\ (E^*\ \{\{set\ w_0\}\})$

begin

private abbreviation (*input*) $W_0 \equiv set\ w_0$

definition *dfs-reachable-invar* $D\ V\ W\ brk \longleftrightarrow$

$W_0 \subseteq W \cup V$
 $\wedge\ W \cup V \subseteq E^*\ \{\{W_0\}\}$
 $\wedge\ E^*\ V \subseteq W \cup V$
 $\wedge\ Collect\ D \cap V = \{\}$
 $\wedge\ (brk \longrightarrow Collect\ D \cap E^*\ \{\{W_0 \neq \{\}\}\})$

lemma *card-decreases*:

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

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

$(\forall y\ ys. x2 \neq y \# ys) \longleftrightarrow x2 = []$ ⟨proof⟩

lemma *dfs-reachable-correct*: $dfs\text{-reachable}\ D\ w_0 \longleftrightarrow Collect\ D \cap E^*\ \{\{set\ w_0 \neq \{\}\}\}$

⟨proof⟩

end

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

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

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

end

lemma *finite-imp-finite-dfs-reachable*:
 $[[\text{finite } E; \text{finite } S]] \implies \text{finite } (E^* \text{ } S)$
<proof>

lemma *dfs-reachable-tab-succ-correct*: $\text{dfs-reachable } (\text{tab-succ } l) \text{ } D \text{ } vs_0 \iff \text{Collect } D \cap (\text{set } l)^* \text{ } \{\}$
<proof>

4.2 Implementation Refinements

4.2.1 Of-Type

definition *of-type-impl* $G \text{ } oT \text{ } T \equiv (\forall pt \in \text{set } (\text{primitives } oT). \text{dfs-reachable } G \text{ } ((=) \text{ } pt) \text{ } (\text{primitives } T))$

fun *ty-term'* **where**

ty-term' $\text{varT } objT \text{ } (\text{term.VAR } v) = \text{varT } v$
ty-term' $\text{varT } objT \text{ } (\text{term.CONST } c) = \text{Mapping.lookup } objT \text{ } c$

lemma *ty-term'-correct-aux*: $\text{ty-term}' \text{ } \text{varT } objT \text{ } t = \text{ty-term } \text{varT } (\text{Mapping.lookup } objT \text{ } t)$
<proof>

lemma *ty-term'-correct[simp]*: $\text{ty-term}' \text{ } \text{varT } objT = \text{ty-term } \text{varT } (\text{Mapping.lookup } objT)$
<proof>

context *ast-domain* **begin**

definition *of-type1* $pt \text{ } T \iff pt \in \text{subtype-rel}^* \text{ } \{\}$

lemma *of-type-refine1*: $\text{of-type } oT \text{ } T \iff (\forall pt \in \text{set } (\text{primitives } oT). \text{of-type1 } pt \text{ } T)$

<proof>

definition $STG \equiv (tab-succ (map subtype-edge (types D)))$

lemma *subtype-rel-impl*: $subtype-rel = E-of-succ (tab-succ (map subtype-edge (types D)))$
<proof>

lemma *of-type1-impl*: $of-type1\ pt\ T \longleftrightarrow dfs-reachable (tab-succ (map subtype-edge (types D))) ((=)pt) (primitives\ T)$
<proof>

lemma *of-type-impl-correct*: $of-type-impl\ STG\ oT\ T \longleftrightarrow of-type\ oT\ T$
<proof>

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

lemma *mp-objT-correct[simp]*: $Mapping.lookup\ mp-constT = constT$
<proof>

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'* $stg\ v\ T \longleftrightarrow ($
 $case\ ty-ent\ v\ of$
 $Some\ vT \Rightarrow of-type-impl\ stg\ vT\ T$
 $| None \Rightarrow False)$

lemma *is-of-type'-correct*: $is-of-type'\ STG\ v\ T = is-of-type\ ty-ent\ v\ T$
<proof>

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

lemma *wf-pred-atom'-correct*: $wf-pred-atom'\ STG\ pvs = wf-pred-atom\ ty-ent\ pvs$
<proof>

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

lemma *wf-atom'-correct*: $wf-atom'\ STG\ a = wf-atom\ ty-ent\ a$
<proof>

```

fun wf-fmla' :: - => ('ent atom) formula => bool where
  wf-fmla' stg (Atom a) <-> wf-atom' stg a
| wf-fmla' stg ⊥ <-> True
| wf-fmla' stg (φ1 ∧ φ2) <-> (wf-fmla' stg φ1 ∧ wf-fmla' stg φ2)
| wf-fmla' stg (φ1 ∨ φ2) <-> (wf-fmla' stg φ1 ∧ wf-fmla' stg φ2)
| wf-fmla' stg (φ1 → φ2) <-> (wf-fmla' stg φ1 ∧ wf-fmla' stg φ2)
| wf-fmla' stg (¬φ) <-> wf-fmla' stg φ

```

lemma wf-fmla'-correct: wf-fmla' STG φ <-> wf-fmla ty-ent φ
 ⟨proof⟩

```

fun wf-fmla-atom1' where
  wf-fmla-atom1' stg (Atom (predAtm p vs)) <-> wf-pred-atom' stg (p,vs)
| wf-fmla-atom1' stg - <-> False

```

lemma wf-fmla-atom1'-correct: wf-fmla-atom1' STG φ = wf-fmla-atom ty-ent φ
 ⟨proof⟩

```

fun wf-effect' where
  wf-effect' stg (Effect a d) <->
    (∀ ae∈set a. wf-fmla-atom1' stg ae)
    ∧ (∀ de∈set d. wf-fmla-atom1' stg de)

```

lemma wf-effect'-correct: wf-effect' STG e = wf-effect ty-ent e
 ⟨proof⟩

end — Context fixing ty-ent

```

fun wf-action-schema' :: - => - => ast-action-schema => bool where
  wf-action-schema' stg conT (Action-Schema n params pre eff) <-> (
    let
      tyv = ty-term' (map-of params) conT
    in
      distinct (map fst params)
      ∧ wf-fmla' tyv stg pre
      ∧ wf-effect' tyv stg eff)

```

lemma wf-action-schema'-correct: wf-action-schema' STG mp-constT s = wf-action-schema s
 ⟨proof⟩

```

definition wf-domain' :: - => - => bool where
  wf-domain' stg conT ≡
    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)

```

\wedge *distinct* (*map ast-action-schema.name (actions D)*)
 \wedge ($\forall a \in \text{set } (\text{actions } D). \text{wf-action-schema}' \text{ stg conT } a$)

lemma *wf-domain'-correct*: *wf-domain' STG mp-constT = wf-domain*
 $\langle \text{proof} \rangle$

end — Context of *ast-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*
 $:: \text{object ast-effect} \Rightarrow \text{world-model} \Rightarrow \text{world-model}$
where
apply-effect-exec (*Effect a d*) *s*
 $= \text{fold } (\lambda \text{add } s. \text{Set.insert add } s) a$
 $(\text{fold } (\lambda \text{del } s. \text{Set.remove del } s) d s)$

lemma *apply-effect-exec-refine[simp]*:
apply-effect-exec (*Effect (a) (d)*) *s*
 $= \text{apply-effect } (\text{Effect } (a) (d)) s$
 $\langle \text{proof} \rangle$

lemmas *apply-effect-eq-impl-eq*
 $= \text{apply-effect-exec-refine}[\text{symmetric, unfolded apply-effect-exec.simps}]$

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

lemma *mp-objT-correct[simp]*: $\text{Mapping.lookup mp-objT} = \text{objT}$
 $\langle \text{proof} \rangle$

We refine the typecheck to use the mapping

definition *is-obj-of-type-impl stg mp n T* = (

$\text{case Mapping.lookup mp } n \text{ of None} \Rightarrow \text{False} \mid \text{Some } oT \Rightarrow \text{of-type-impl stg } oT$
 T
 $)$

lemma *is-obj-of-type-impl-correct*[simp]:
 $\text{is-obj-of-type-impl STG mp-objT} = \text{is-obj-of-type}$
 $\langle \text{proof} \rangle$

We refine the well-formedness checks to use the mapping

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

lemma *wf-fact'-correct*[simp]: $\text{wf-fact}' \text{ mp-objT STG} = \text{wf-fact}$
 $\langle \text{proof} \rangle$

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

lemma *wf-fmla-atom2'-correct*[simp]:
 $\text{wf-fmla-atom2}' \text{ mp-objT STG } \varphi = \text{wf-fmla-atom } \text{objT } \varphi$
 $\langle \text{proof} \rangle$

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

lemma *wf-problem'-correct*:
 $\text{wf-problem}' \text{ STG mp-constT mp-objT} = \text{wf-problem}$
 $\langle \text{proof} \rangle$

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

lemma *wf-effect-inst-weak*:
fixes $a \text{ args}$
defines $ai \equiv \text{instantiate-action-schema } a \text{ args}$
assumes $A: \text{action-params-match } a \text{ args}$
 $\text{wf-action-schema } a$
shows $\text{wf-effect-inst } (\text{effect } ai)$
 $\langle \text{proof} \rangle$

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* $n = \text{Some } a$
 shows *wf-action-schema* a
 $\langle \text{proof} \rangle$

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* $s F \longleftrightarrow \text{close-world } s \models F$
 $\langle \text{proof} \rangle$

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

```

    Error-Monad.return (apply-effect (effect ai) s)
  }

```

Justification of implementation.

lemma (in *wf-ast-problem*) *en-exE-return-iff*:
assumes *wm-basic s*
shows *en-exE a s = Inr s'*
 \longleftrightarrow *plan-action-enabled a s \wedge s' = execute-plan-action a s*
<proof>

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 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*
 $:: - \Rightarrow (\text{object, type}) \text{ mapping} \Rightarrow \text{plan-action} \Rightarrow \text{world-model} \Rightarrow \text{-+world-model}$
where
en-exE2 G mp \equiv $\lambda(PAction\ n\ args) \Rightarrow \lambda M. \text{do } \{$
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*
<proof>

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*
<proof>

lemma (in *wf-ast-problem*) *en-exE2-return-iff-compact-notation*:
 $\llbracket \text{wm-basic } s \rrbracket \implies$
en-exE2 STG mp-objT a s = Inr s'
 \longleftrightarrow *plan-action-enabled a s \wedge s' = execute-plan-action a s*
<proof>

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* π *s* = *valid-plan-from1* *s* π *s*
<proof>

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  $\pi$  s
  <+? ( $\lambda e$  -. shows "at step " o shows si o shows "": " o e ());
  valid-plan-fromE stg mp (si+1) s  $\pi$ s
}
```

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*:
 \llbracket *wm-basic* *s*; *wf-plan-action* *a* \rrbracket
 \implies *wm-basic* (*execute-plan-action* *a* *s*)
<proof>

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* π *s* = *Inr* () \longleftrightarrow *valid-plan-from* *s* π *s*
<proof>

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*]: $check-all-list\ P\ l\ msg\ msgf = Inr\ () \longleftrightarrow$
 $(\forall x \in set\ l. P\ x)$
 $\langle proof \rangle$

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*]: $check-wf-types\ D = Inr\ () \longleftrightarrow ast-domain.wf-types$
 D
 $\langle proof \rangle$

definition *check-wf-domain* $D\ stg\ conT \equiv do\ \{$
 $check-wf-types\ D;$
 $check\ (distinct\ (map\ (predicate-decl.pred)\ (predicates\ D)))\ (ERRS\ "Duplicate$
 $predicate\ declaration");$
 $check-all-list\ (ast-domain.wf-predicate-decl\ D)\ (predicates\ D)\ "Malformed\ predi-$
 $cate\ declaration"\ (shows\ o\ predicate.name\ o\ predicate-decl.pred);$
 $check\ (distinct\ (map\ fst\ (consts\ D)))\ (ERRS\ "Duplicate\ constant\ declaration");$
 $check\ (\forall (n,T) \in set\ (consts\ D). ast-domain.wf-type\ D\ T)\ (ERRS\ "Malformed$
 $type");$
 $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*]:
 $check-wf-domain\ D\ stg\ conT = Inr\ () \longleftrightarrow ast-domain.wf-domain'\ D\ stg\ conT$
 $\langle proof \rangle$

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

definition *check-wf-problem* P stg $conT$ $mp \equiv do \{$
 let $D = ast\text{-}problem.domain\ P$;
 check-wf-domain D stg $conT$ $<+? prepend\ err\ msg\ "Domain\ not\ well\text{-}formed"$;
 check (distinct (map fst (objects P) @ map fst (consts D))) (ERRS "Duplicate
 object declaration");
 check (($\forall (n, T) \in set\ (objects\ P).$ ast-domain.wf-type D T)) (ERRS "Malformed
 type");
 check (distinct (init P)) (ERRS "Duplicate fact in initial state");
 check ($\forall f \in set\ (init\ P).$ ast-problem.wf-fmla-atom2' P mp 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]:
 $check\text{-}wf\text{-}problem\ P\ stg\ conT\ mp = Inr\ () \iff ast\text{-}problem.wf\text{-}problem'\ P\ stg$
 $conT\ mp$
 <proof>

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

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]: $check\text{-}plan\ P\ \pi s = Inr\ ()$
 $\iff ast\text{-}problem.wf\text{-}problem\ P \wedge ast\text{-}problem.valid\text{-}plan\ P\ \pi s$
 <proof>

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
  ⟨proof⟩
```

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

```
lemma [code-unfold]: distinct = distinct-ds
  ⟨proof⟩
```

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)
  ⟨proof⟩

```

```

lemma close-eq-Eq[simp]: close-eq s (Eq a b) ⟷ a=b
  ⟨proof⟩

```

```

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-opr opr f =
    ((∀ s. s |= (precondition opr) ⟶
      (∃ s'. f s = (Some s')

```

$$\begin{aligned}
& \wedge (\forall atm. is-predAtom\ atm \wedge atm \notin set(dels\ (effect\ opr)) \wedge s \models atm \\
\longrightarrow s' \models atm) \\
& \wedge (\forall atm. is-predAtom\ atm \wedge atm \notin set(adds\ (effect\ opr)) \wedge s \models \\
Not\ atm \longrightarrow s' \models Not\ atm) \\
& \wedge (\forall atm. atm \in set(adds\ (effect\ opr)) \longrightarrow s' \models atm) \\
& \wedge (\forall fmla. fmla \in set(dels\ (effect\ opr)) \wedge fmla \notin set(adds\ (effect \\
opr)) \longrightarrow s' \models (Not\ fmla)) \\
& \wedge (\forall a\ b. s \models Atom\ (Eq\ a\ b) \longrightarrow s' \models Atom\ (Eq\ a\ b)) \\
& \wedge (\forall a\ b. s \models Not\ (Atom\ (Eq\ a\ b)) \longrightarrow s' \models Not\ (Atom\ (Eq\ a\ b))) \\
&)) \\
& \wedge (\forall fmla \in set(adds\ (effect\ opr)). is-predAtom\ fmla)) \\
\langle proof \rangle
\end{aligned}$$

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

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 $fs \equiv fold\ compose-action\ fs\ Some$

Composing a list of actions satisfies some natural lemmas:

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

compose-actions $[] = Some\ \langle proof \rangle$

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

$f\ s = Some\ s' \implies compose-actions\ (f\ \#fs)\ s = compose-actions\ fs\ s'$
 $\langle proof \rangle$

Soundness Theorem in Lifschitz’s paper [4].

theorem *STRIPS-sema-sound*:

assumes *sound-system* $\Sigma\ M_0\ s_0\ f$

— For a sound system Σ

assumes *set* $\alpha s \subseteq \Sigma$

— And a plan αs

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 $\forall fmla \in \text{close-world } M'. s' \models fmla$
 $\langle \text{proof} \rangle$

More compact notation of the soundness theorem.

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

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

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

lemma *wm-to-state-inv[simp]*: $wm\text{-to-state } (\text{state-to-wm } s) = s$
 $\langle \text{proof} \rangle$

Mapping AST action instances to actions

definition *pddl-opr-to-act* $g\text{-opr } s = (
\text{let } M = \text{state-to-wm } s \text{ in }
\text{if } (wm\text{-to-state } (\text{close-world } M)) \models (\text{precondition } g\text{-opr}) \text{ then }
\text{Some } (wm\text{-to-state } (\text{apply-effect } (\text{effect } g\text{-opr}) M))
\text{else }
\text{None})$

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*:

$s \models_{=} (formula.Atom atm)$
 $\longleftrightarrow ((formula.Atom atm) \in close-eq-M (state-to-wm s))$
 $\langle proof \rangle$

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

$close-eq (wm-to-state M) \models (formula.Atom atm)$
 $\longleftrightarrow ((formula.Atom atm) \in close-eq-M M)$
 $\langle proof \rangle$

lemma *not-dels-preserved*:

assumes $f \notin (set d)$ $f \in M$
shows $f \in apply-effect (Effect a d) M$
 $\langle proof \rangle$

lemma *adds-satisfied*:

assumes $f \in (set a)$
shows $f \in apply-effect (Effect a d) M$
 $\langle proof \rangle$

lemma *dels-unsatisfied*:

assumes $f \in (set d)$ $f \notin set a$
shows $f \notin apply-effect (Effect a d) M$
 $\langle proof \rangle$

lemma *dels-unsatisfied-2*:

assumes $f \in set (dels eff)$ $f \notin set (adds eff)$
shows $f \notin apply-effect eff M$
 $\langle proof \rangle$

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

$\langle proof \rangle$

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

assumes $wf-effect-inst effs ae \in set (adds effs)$
shows $is-predAtom ae$
 $\langle proof \rangle$

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

assumes $wf-effect-inst effs ae \in set (dels effs)$
shows $is-predAtom ae$
 $\langle proof \rangle$

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

$\langle proof \rangle$

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))*
 $\langle proof \rangle$

lemma *wf-eff-impl-wf-eff-inst*: *wf-effect objT eff \implies wf-effect-inst eff*
 $\langle proof \rangle$

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)*
 $\langle proof \rangle$

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)))
 $\langle proof \rangle$

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)))
 $\langle proof \rangle$

lemma *wf-plan-act-is-sound'*:
assumes *wf-plan-action π*
shows *sound-opr*
(resolve-instantiate π)
((pddl-opr-to-act (resolve-instantiate π)))
 $\langle proof \rangle$

lemma *wf-world-model-has-atoms*: *f \in M \implies wf-world-model M \implies is-predAtom*
f
 $\langle proof \rangle$

lemma *wm-to-state-works-for-wf-wm-closed*:
wf-world-model M \implies fmla \in close-world M \implies close-eq (wm-to-state M) \models
fmla
 $\langle proof \rangle$

lemma *wm-to-state-works-for-wf-wm*: *wf-world-model* $M \implies fmla \in M \implies close\text{-}eq$
(wm-to-state $M) \models fmla$
<proof>

lemma *wm-to-state-works-for-I-closed*:
assumes $x \in close\text{-}world\ I$
shows $close\text{-}eq\ (wm\text{-}to\text{-}state\ I) \models x$
<proof>

lemma *wf-wm-imp-basic*: *wf-world-model* $M \implies wm\text{-}basic\ M$
<proof>

theorem *wf-plan-sound-system*:
assumes $\forall \pi \in set\ \pi s. wf\text{-}plan\text{-}action\ \pi$
shows *sound-system*
 (*set* (*map resolve-instantiate* πs))
 I
 (*wm-to-state* I)
 ($(\lambda \alpha. pddl\text{-}opr\text{-}to\text{-}act\ \alpha)$)
<proof>

theorem *wf-plan-soundness-theorem*:
assumes *plan-action-path* $I\ \pi s\ M$
defines $\alpha s \equiv map\ (pddl\text{-}opr\text{-}to\text{-}act \circ resolve\text{-}instantiate)\ \pi s$
defines $s_0 \equiv wm\text{-}to\text{-}state\ I$
shows $\exists s'. compose\text{-}actions\ \alpha s\ s_0 = Some\ s' \wedge (\forall \varphi \in close\text{-}world\ M. s' \models \varphi)$
<proof>

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.