

A Formally Verified Checker for PDDL

This is the proof document generated by Isabelle/HOL while processing the theories. It contains a latex rendering of the Isabelle sources. Isabelle only generates this document after all the proofs have succeeded.

Contents

1	PDDL and STRIPS Semantics	3
1.1	Utility Functions	3
1.2	Abstract Syntax	3
1.2.1	Generic Entities	3
1.2.2	Domains	4
1.2.3	Problems	5
1.2.4	Plans	5
1.2.5	Ground Actions	5
1.3	Closed-World Assumption, Equality, and Negation	5
1.3.1	Proper Generalization	7
1.4	STRIPS Semantics	7
1.5	Well-Formedness of PDDL	8
1.6	PDDL Semantics	12
1.7	Preservation of Well-Formedness	15
1.7.1	Well-Formed Action Instances	15
1.7.2	Preservation	17
2	Executable PDDL Checker	18
2.1	Generic DFS Reachability Checker	19
2.2	Implementation Refinements	20
2.2.1	Of-Type	20
2.2.2	Application of Effects	23
2.2.3	Well-Formedness	23
2.2.4	Execution of Plan Actions	25
2.2.5	Checking of Plan	27
2.3	Executable Plan Checker	28
2.4	Code Setup	30
2.4.1	Code Equations	30
2.4.2	Setup for Containers Framework	31

2.4.3	More Efficient Distinctness Check for Linorders	31
2.4.4	Code Generation	31
2.4.5	Soundness theorem for the STRIPS semantics	32
2.5	Soundness Theorem for PDDL	34
3	Reasoning about Invariants	38

1 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)
```

1.1 Utility Functions

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

lemma $index\text{-}by\text{-}eq\text{-}Some\text{-}eq[simp]$:
assumes $distinct\ (map\ f\ l)$
shows $index\text{-}by\ f\ l\ n = Some\ x \longleftrightarrow (x \in set\ l \wedge f\ x = n)$
 $\langle proof \rangle$

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

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

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

1.2 Abstract Syntax

1.2.1 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

1.2.2 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*)

1.2.3 Problems

A fact is a predicate applied to objects.

type-synonym $fact = predicate \times object\ list$

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

datatype $ast\text{-}problem = Problem$
($domain: ast\text{-}domain$)
($objects: (object \times type)\ list$)
($init: object\ atom\ formula\ list$)
($goal: object\ atom\ formula$)

1.2.4 Plans

datatype $plan\text{-}action = PAction$
($name: name$)
($arguments: object\ list$)

type-synonym $plan = plan\text{-}action\ list$

1.2.5 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 $ground\text{-}action = Ground\text{-}Action$
($precondition: (object\ atom)\ formula$)
($effect: object\ ast\text{-}effect$)

1.3 Closed-World Assumption, Equality, and Negation

Discriminator for atomic predicate formulas.

fun $is\text{-}predAtom$ **where**
 $is\text{-}predAtom\ (Atom\ (predAtm\ -\ -)) = True \mid is\text{-}predAtom\ - = False$

The world model is a set of (atomic) formulas

type-synonym $world\text{-}model = object\ atom\ formula\ set$

It is basic, if it only contains atoms

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

A valuation extracted from the atoms of the world model

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

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

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

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

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

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

lemma

close-world-extensive: $M \subseteq \text{close-world } M$ **and**
close-world-idem[simp]: $\text{close-world } (\text{close-world } M) = \text{close-world } M$
 $\langle \text{proof} \rangle$

lemma *in-close-world-conv*:

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

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 $\mathcal{A} = \text{valuation } M$
 $\langle \text{proof} \rangle$

lemma *valuation-aux-2*:

assumes *wm-basic* M
shows $(\forall G \in \text{close-world } M. \text{valuation } M \models G)$
 $\langle \text{proof} \rangle$

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

lemma *close-world-imp-val*:

wm-basic $M \Longrightarrow M \text{ } ^c \models = \varphi \Longrightarrow \text{valuation } M \models \varphi$

$\langle proof \rangle$

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 *valuation* $M \models \varphi \longleftrightarrow M^c \models \varphi$

$\langle proof \rangle$

1.3.1 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* φ_2

| *is-STRIPS-fmla* ($\varphi_1 \vee \varphi_2$) \longleftrightarrow *is-STRIPS-fmla* $\varphi_1 \wedge$ *is-STRIPS-fmla* φ_2

| *is-STRIPS-fmla* ($\neg \perp$) \longleftrightarrow *True*

| *is-STRIPS-fmla* - \longleftrightarrow *False*

lemma *aux1*: $\llbracket \text{wm-basic } M; \text{is-STRIPS-fmla } \varphi; \text{valuation } M \models \varphi; \forall G \in M. \mathcal{A} \models G \rrbracket \Longrightarrow \mathcal{A} \models \varphi$

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *valuation-iff-STRIPS:*

assumes *wm-basic* M

assumes *is-STRIPS-fmla* φ

shows *valuation* $M \models \varphi \longleftrightarrow M \models \varphi$

$\langle proof \rangle$

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

theorem *proper-STRIPS-generalization:*

$\llbracket \text{wm-basic } M; \text{is-STRIPS-fmla } \varphi \rrbracket \Longrightarrow M^c \models \varphi \longleftrightarrow M \models \varphi$

$\langle proof \rangle$

1.4 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 $M_0 \dots M_N$ of intermediate world models, as done in [Lif87].

```

fun ground-action-path
  :: world-model  $\Rightarrow$  ground-action list  $\Rightarrow$  world-model  $\Rightarrow$  bool
where
  ground-action-path M [] M'  $\longleftrightarrow$  (M = M')
  | ground-action-path M ( $\alpha \# \alpha s$ ) M'  $\longleftrightarrow$  M  $\models_{\text{c}}$  precondition  $\alpha$ 
     $\wedge$  ground-action-path (execute-ground-action  $\alpha$  M)  $\alpha s$  M'

```

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

```

lemma ground-action-path-in-paper:
  ground-action-path M [] M'  $\longleftrightarrow$  (M = M')
  ground-action-path M ( $\alpha \# \alpha s$ ) M'  $\longleftrightarrow$  M  $\models_{\text{c}}$  precondition  $\alpha$ 
     $\wedge$  (ground-action-path (apply-effect (effect  $\alpha$ ) M)  $\alpha s$  M')
  <proof>

```

end — Context of *ast-domain*

1.5 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'  $\Longrightarrow$  objT  $\subseteq_m$  objT'  $\Longrightarrow$ 
  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  $\varphi$ 
| wf-fmla ( $\varphi1 \rightarrow \varphi2$ )  $\longleftrightarrow$  (wf-fmla  $\varphi1 \wedge$  wf-fmla  $\varphi2$ )
```

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

Special case for a well-formed atomic predicate formula

```
fun wf-fmla-atom where
  wf-fmla-atom (Atom (predAtm a vs))  $\longleftrightarrow$  wf-pred-atom (a,vs)
| wf-fmla-atom -  $\longleftrightarrow$  False
```

```
lemma wf-fmla-atom-alt: wf-fmla-atom  $\varphi \longleftrightarrow \text{is-predAtom } \varphi \wedge \text{wf-fmla } \varphi$ 
  <proof>
```

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

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

end — Context fixing *ty-ent*

```
definition constT :: object  $\rightarrow$  type where
  constT  $\equiv$  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  $\Rightarrow$  bool where
  wf-action-schema (Action-Schema n params pre eff)  $\longleftrightarrow$  (
    let
      tyt = ty-term (map-of params) constT
    in
      distinct (map fst params)
       $\wedge$  wf-fmla tyt pre
       $\wedge$  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*) \longleftrightarrow *set* *Ts* \subseteq *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*) \longleftrightarrow ($\forall T \in \text{set } Ts. \text{wf-type } T$)

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

definition *wf-types* \equiv *snd*'*set* (*types* *D*) \subseteq *insert* "object" (*fst*'*set* (*types* *D*))

A domain is well-formed if

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

definition *wf-domain* :: *bool* **where**
wf-domain \equiv
wf-types
 \wedge *distinct* (*map* (*predicate-decl.pred*) (*predicates* *D*))
 \wedge ($\forall p \in \text{set } (\text{predicates } D). \text{wf-predicate-decl } p$)
 \wedge *distinct* (*map* *fst* (*consts* *D*)) (* Required by current semantics that assigns
each object a unique type *)
 \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**

$objT \equiv \text{map-of } (objects\ P) ++ \text{const}T$

lemma *objT-alt*: $objT = \text{map-of } (const\ D @ objects\ P)$
 $\langle proof \rangle$

definition *wf-fact* :: $fact \Rightarrow bool$ **where**
 $wf\text{-}fact = wf\text{-}pred\text{-}atom\ objT$

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

definition *wf-problem* **where**
 $wf\text{-}problem \equiv$
 $wf\text{-}domain$
 $\wedge distinct\ (\text{map}\ fst\ (objects\ P) @ \text{map}\ fst\ (const\ D))\ (*\ \text{Required by current semantics that assigns each object a unique type} *)$
 $\wedge (\forall (n, T) \in set\ (objects\ P). wf\text{-}type\ T)$
 $\wedge distinct\ (init\ P)$
 $\wedge wf\text{-}world\text{-}model\ (set\ (init\ P))$
 $\wedge wf\text{-}fmla\ objT\ (goal\ P)$

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

lemma *wf-effect-inst-alt*: $wf\text{-}effect\text{-}inst\ eff = wf\text{-}effect\ objT\ eff$
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

end — locale *wf-ast-problem*

1.6 PDDL Semantics

context *ast-domain* **begin**

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

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

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

fun *instantiate-action-schema*
:: *ast-action-schema* \Rightarrow *object list* \Rightarrow *ground-action*
where
instantiate-action-schema (Action-Schema n params pre eff) args = (let
tsubst = subst-term (the o (map-of (zip (map fst params) args)));
pre-inst = (map-formula o map-atom) tsubst pre;
eff-inst = (map-ast-effect) tsubst eff
in
Ground-Action pre-inst eff-inst
)

end — Context of *ast-domain*

context *ast-problem* **begin**

Initial model

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

Resolve a plan action and instantiate the referenced action schema.

fun *resolve-instantiate* :: *plan-action* \Rightarrow *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* \equiv *case objT n of*
None \Rightarrow *False*
| *Some oT* \Rightarrow *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* \times *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*
 $\equiv \text{list-all2 } \text{is-obj-of-type } \text{args } (\text{map snd } (\text{parameters } a))$

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* \Rightarrow *bool* **where**
wf-plan-action (*PAction* *n args*) = (
 case *resolve-action-schema* *n* of
 None \Rightarrow *False*
 | *Some* *a* \Rightarrow
 (* *Objects are valid and match parameter types* *)
 action-params-match *a args*
 (* *Effect is valid* *)
 \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'* =
 (($\forall \pi \in \text{set } \pi$ *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'* \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'* \models (*goal* *P*)
 $\langle \text{proof} \rangle$

end — Context of *ast-problem*

1.7 Preservation of Well-Formedness

1.7.1 Well-Formed Action Instances

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

context *ast-problem* **begin**

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

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

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

We begin with some auxiliary lemmas before the actual theorem.

```
lemma (in ast-domain) of-type-refl[simp, intro!]: of-type T T
  <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 < length params params!i = (x,xT) of-type xT T
  <proof>
```

```
lemma wf-atom-mono:
  assumes SS: tys  $\subseteq_m$  tys'
  assumes WF: wf-atom tys a
  shows wf-atom tys' a
  <proof>
```

```
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>
```

```
lemma constT-ss-objT: constT  $\subseteq_m$  objT
  <proof>
```

lemma *wf-atom-constT-imp-objT*: *wf-atom (ty-term Q constT) a \implies wf-atom (ty-term Q objT) a*
 $\langle \text{proof} \rangle$

lemma *wf-fmla-atom-constT-imp-objT*: *wf-fmla-atom (ty-term Q constT) a \implies wf-fmla-atom (ty-term Q objT) a*
 $\langle \text{proof} \rangle$

context

fixes *Q* **and** *f :: variable \Rightarrow 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*
 $\langle \text{proof} \rangle$

lemma *is-of-type-const-conv*: *is-of-type (ty-term Q objT) (term.CONST x) T \longleftrightarrow is-of-type objT x T*
 $\langle \text{proof} \rangle$

lemma *INST'*: *is-of-type (ty-term Q objT) x T \implies is-of-type objT (subst-term f x) T*
 $\langle \text{proof} \rangle$

lemma *wf-inst-eq-aux*: *Q x = Some T \implies objT (f x) \neq None*
 $\langle \text{proof} \rangle$

lemma *wf-inst-eq-aux'*: *ty-term Q objT x = Some T \implies objT (subst-term f x) \neq None*
 $\langle \text{proof} \rangle$

lemma *wf-inst-atom*:

assumes *wf-atom (ty-term Q constT) a*

shows *wf-atom objT (map-atom (subst-term f) a)*

$\langle \text{proof} \rangle$

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)*

$\langle \text{proof} \rangle$

lemma *wf-inst-effect*:

assumes *wf-effect (ty-term Q constT) φ*

shows *wf-effect objT ((map-ast-effect o subst-term) f φ)*

$\langle \text{proof} \rangle$


```

lemma wf-inst-formula:
  assumes wf-fmla (ty-term Q constT)  $\varphi$ 
  shows wf-fmla objT ((map-formula o map-atom o subst-term) f  $\varphi$ )
   $\langle proof \rangle$ 

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)
   $\langle proof \rangle$ 
end — Context of ast-problem

```

1.7.2 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
     $\longleftrightarrow$  wf-plan-action  $\pi \wedge M \models_{\mathcal{C}} \text{precondition } (\text{resolve-instantiate } \pi)$ 

```

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
  = (apply-effect (effect (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'  $\longleftrightarrow M' = M$ 
   $\langle proof \rangle$ 

```

```

lemma plan-action-path-Cons[simp]:
  plan-action-path M ( $\pi \# \pi s$ ) M'  $\longleftrightarrow$ 
    plan-action-enabled  $\pi$  M
     $\wedge$  plan-action-path (execute-plan-action  $\pi$  M)  $\pi s$  M'
   $\langle proof \rangle$ 

```

end — Context of *ast-problem*

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

The initial world model is well-formed

lemma *wf-I: wf-world-model I*
 $\langle proof \rangle$

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)*
 $\langle proof \rangle$

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)*
 $\langle proof \rangle$

theorem *wf-execute-compact-notation:*
 $plan-action-enabled \pi s \implies wf-world-model s$
 $\implies wf-world-model (execute-plan-action \pi s)$
 $\langle proof \rangle$

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'*
 $\langle proof \rangle$

end — Context of *wf-ast-problem*

end — Theory

2 Executable PDDL Checker

theory *PDDL-STRIPS-Checker*

imports

PDDL-STRIPS-Semantics

Error-Monad-Add

HOL-Library.Char-ord
HOL-Library.Code-Char
HOL-Library.Code-Target-Nat

HOL-Library.While-Combinator

Containers.Containers

begin

2.1 Generic DFS Reachability Checker

Used for subtype checks

definition *E-of-succ succ* $\equiv \{ (u,v). v \in \text{set } (\text{succ } u) \}$

lemma *succ-as-E*: $\text{set } (\text{succ } x) = \text{E-of-succ succ} \text{ `` } \{x\}$
 $\langle \text{proof} \rangle$

context

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

begin

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

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 *w₀* :: 'a list

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

begin

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

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

$W_0 \subseteq W \cup V$

$\wedge W \cup V \subseteq E^* \text{ `` } W_0$

$\wedge E \text{ `` } V \subseteq W \cup V$

$\wedge \text{Collect } D \cap V = \{\}$

$\wedge (brk \longrightarrow Collect\ D \cap E^* \text{ `` } W_0 \neq \{\})$

lemma *card-decreases*:

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

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

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

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

end

definition *tab-succ* $l \equiv Mapping.lookup_default\ []\ (fold\ (\lambda(u,v). Mapping.map_default\ u\ []\ (Cons\ v))\ l\ Mapping.empty)$

lemma *Some-eq-map-option [iff]*: $(Some\ y = map_option\ f\ xo) = (\exists z. xo = Some\ z \wedge f\ z = y)$
 $\langle proof \rangle$

lemma *tab-succ-correct*: $E_of_succ\ (tab_succ\ l) = set\ l$
 $\langle proof \rangle$

end

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

$\llbracket finite\ E; finite\ S \rrbracket \implies finite\ (E^* \text{ `` } S)$
 $\langle proof \rangle$

lemma *dfs-reachable-tab-succ-correct*: $dfs_reachable\ (tab_succ\ l)\ D\ vs_0 \longleftrightarrow Collect\ D \cap (set\ l)^* \text{ `` } set\ vs_0 \neq \{\}$
 $\langle proof \rangle$

2.2 Implementation Refinements

2.2.1 Of-Type

definition *of-type-impl* $G\ oT\ T \equiv (\forall pt \in set\ (primitives\ oT). dfs_reachable\ G\ (op=pt)\ (primitives\ T))$

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*
 ⟨proof⟩

lemma *ty-term'-correct[simp]*: *ty-term' varT objT = ty-term varT (Mapping.lookup objT)*
 ⟨proof⟩

context *ast-domain* **begin**

definition *of-type1 pt T* $\longleftrightarrow pt \in \text{subtype-rel}^* \text{ `` set (primitives T) }$

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

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

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* $\longleftrightarrow \text{dfs-reachable (tab-succ (map subtype-edge (types D))) (op=pt) (primitives T)}$
 ⟨proof⟩

lemma *of-type-impl-correct*: *of-type-impl STG oT T* $\longleftrightarrow \text{of-type oT T}$
 ⟨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)* $\longleftrightarrow (\text{case sig p of}$

$None \Rightarrow False$
 $| Some\ Ts \Rightarrow list-all2\ (is-of-type'\ stg)\ vs\ Ts$

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

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*
 $\langle proof \rangle$

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

lemma *wf-fmla'-correct*: *wf-fmla' STG $\varphi \longleftrightarrow wf-fmla\ ty-ent\ \varphi$*
 $\langle proof \rangle$

fun *wf-fmla-atom1'* **where**
 $wf-fmla-atom1'\ stg\ (Atom\ (predAtm\ p\ vs)) \longleftrightarrow wf-pred-atom'\ stg\ (p,vs)$
 $| wf-fmla-atom1'\ stg\ - \longleftrightarrow False$

lemma *wf-fmla-atom1'-correct*: *wf-fmla-atom1' STG $\varphi = wf-fmla-atom\ ty-ent\ \varphi$*
 φ
 $\langle proof \rangle$

fun *wf-effect'* **where**
 $wf-effect'\ stg\ (Effect\ a\ d) \longleftrightarrow$
 $(\forall ae \in set\ a. wf-fmla-atom1'\ stg\ ae)$
 $\wedge (\forall de \in set\ d. wf-fmla-atom1'\ stg\ de)$

lemma *wf-effect'-correct*: *wf-effect' STG e = wf-effect ty-ent e*
 $\langle proof \rangle$

end — Context fixing *ty-ent*

fun *wf-action-schema'* :: $- \Rightarrow - \Rightarrow ast-action-schema \Rightarrow bool$ **where**
 $wf-action-schema'\ stg\ conT\ (Action-Schema\ n\ params\ pre\ eff) \longleftrightarrow ($
 let
 $tyv = ty-term'\ (map-of\ params)\ conT$
 in
 $distinct\ (map\ fst\ params)$

$\wedge \text{wf-fmla}' \text{ tyv stg pre}$
 $\wedge \text{wf-effect}' \text{ tyv stg eff})$

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

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

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

end — Context of *ast-domain*

2.2.2 Application of Effects

context *ast-domain* **begin**

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

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

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

end — Context of *ast-domain*

2.2.3 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 $objT = (object, type) \text{ mapping}$

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

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

We refine the typecheck to use the mapping

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

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

We refine the well-formedness checks to use the mapping

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

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

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

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

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

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

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

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

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

end — Context of *ast-domain*

2.2.4 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 *(valuation M) \models F*

Justification of this function

lemma *holds-for-wf-fmlas*:
assumes *wm-basic s*
shows *holds s F \longleftrightarrow 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* :: *plan-action* \Rightarrow *world-model* \Rightarrow $\neg +$ *world-model* **where**
en-exE $\equiv \lambda(PAction\ n\ args) \Rightarrow \lambda s. do \{$
 a $\leftarrow resolve-action-schemaE\ n;$
 check (*action-params-match* *a* *args*) (*ERRS* "Parameter mismatch");
 let *ai* = *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*
 $:: - \Rightarrow (object, type)\ mapping \Rightarrow plan-action \Rightarrow world-model \Rightarrow \neg + world-model$
where
en-exE2 *G* *mp* $\equiv \lambda(PAction\ n\ args) \Rightarrow \lambda M. do \{$
 a $\leftarrow resolve-action-schemaE\ n;$
 check (*action-params-match2* *G* *mp* *a* *args*) (*ERRS* "Parameter mismatch");
 let *ai* = *instantiate-action-schema* *a* *args*;
 check (*holds* *M* (*precondition* *ai*)) (*ERRS* "Precondition not satisfied");
 Error-Monad.return (*apply-effect* (*effect* *ai*) *M*)
 $\}$

Justification of refinement

lemma (**in** *wf-ast-problem*) *wf-en-exE2-eq*:
shows *en-exE2* *STG* *mp-objT* *pa* *s* = *en-exE* *pa* *s*
<proof>

Combination of the two refinement lemmas

lemma (**in** *wf-ast-problem*) *en-exE2-return-iff*:
assumes *wm-basic* *M*

shows $en-exE2\ STG\ mp-objT\ a\ M = Inr\ M'$
 $\longleftrightarrow plan-action-enabled\ a\ M \wedge M' = execute-plan-action\ a\ M$
 $\langle proof \rangle$

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

end — Context of *ast-problem*

2.2.5 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*: $valid-plan-from\ s\ \pi s = valid-plan-from1\ s\ \pi s$
 $\langle proof \rangle$

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
 $valid-plan-fromE\ stg\ mp\ si\ s\ []$
 $= check\ (holds\ s\ (goal\ P))\ (ERRS\ "Postcondition\ does\ not\ hold")$
 $| valid-plan-fromE\ stg\ mp\ si\ s\ (\pi \# \pi s) = do\ \{$
 $s \leftarrow en-exE2\ stg\ mp\ \pi\ s$
 $<+? (\lambda e -. shows\ "at\ step\ " \circ shows\ si \circ shows\ ":\ " \circ e\ ());$
 $valid-plan-fromE\ stg\ mp\ (si+1)\ s\ \pi s$
 $\}$

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*:
 $\llbracket wm-basic\ s; wf-plan-action\ a \rrbracket$
 $\implies wm-basic\ (execute-plan-action\ a\ s)$
 $\langle proof \rangle$

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*
 πs
 $\langle \text{proof} \rangle$

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*

2.3 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 ($\lambda x. \text{check } (P \ x) (\lambda :: \text{unit}. \text{shows } \text{msg } o \text{ shows } "': " o \text{msgf } x)) \ l <+?$
snd

lemma *check-all-list-return-iff*[*return-iff*]: *check-all-list P l msg msgf = Inr ()* \longleftrightarrow
 $(\forall x \in \text{set } l. P \ x)$
 $\langle \text{proof} \rangle$

definition *check-wf-types D* \equiv *do* {
check-all-list ($\lambda(-,t). t = \text{"object"} \vee t \in \text{fst}'\text{set } (\text{types } D)$) (*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
 $\langle \text{proof} \rangle$

definition *check-wf-domain D stg conT* \equiv *do* {
check-wf-types D;
 $(\ast \text{check } (\text{ast-domain.wf-types } D) (\text{ERRS } \text{"Types not well-formed"}) ; \ast)$
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 ($\forall (n, T) \in \text{set } (\text{consts } D). \text{ast-domain.wf-type } D \ T$) (*ERRS "Malformed*
type");
check (*distinct* (*map ast-action-schema.name* (*actions D*))) (*ERRS "Duplicate*
action name");
check-all-list (*ast-domain.wf-action-schema' D stg conT*) (*actions D*) *"Malformed*

action'' (shows o *ast-action-schema.name*)

```
(*
  check ((∀ a ∈ set (actions D). ast-domain.wf-action-schema' D stg a)
    ) (ERRS
    "Malformed action")
  *)
}
```

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

check-wf-domain D stg conT = Inr () \longleftrightarrow *ast-domain.wf-domain' D stg conT*
<proof>

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

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

```
  let D = ast-problem.domain P;
  check-wf-domain D stg conT <+? prepend-err-msg "Domain not well-formed";
  (*check (ast-domain.wf-domain' D stg) (ERRS "Domain not well-formed");*)
  check (distinct (map fst (objects P) @ map fst (consts D))) (ERRS "Duplicate
  object declaration");
  check ((∀ (n,T) ∈ set (objects P). ast-domain.wf-type D T)) (ERRS "Malformed
  type");
  check (distinct (init P)) (ERRS "Duplicate fact in initial state");
  check (∀ f ∈ 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-wf-problem P stg conT mp = Inr () \longleftrightarrow *ast-problem.wf-problem' P stg conT mp*
<proof>

definition *check-plan P πs* $\equiv \text{do } \{$

```
  let stg = ast-domain.STG (ast-problem.domain P);
  let conT = ast-domain.mp-constT (ast-problem.domain P);
  let mp = ast-problem.mp-objT P;
  check-wf-problem P stg conT mp;
  (*check (ast-problem.wf-problem' P stg mp) (ERRS "Domain/Problem not well-formed");*)
  ast-problem.valid-plan-fromE P stg mp 1 (ast-problem.I P) πs
} <+? (λ 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-plan P πs = Inr ()*

\longleftrightarrow *ast-problem.wf-problem P* \wedge *ast-problem.valid-plan P πs*

<proof>

2.4 Code Setup

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

2.4.1 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]

```

2.4.2 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

```

2.4.3 More Efficient Distinctness Check for Linorders

```

fun no-stutter :: 'a list  $\Rightarrow$  bool where
  no-stutter [] = True
| no-stutter [-] = True
| no-stutter (a#b#l) = (a#b  $\wedge$  no-stutter (b#l))

```

```

lemma sorted-no-stutter-eq-distinct: sorted l  $\Longrightarrow$  no-stutter l  $\longleftrightarrow$  distinct l
  <proof>

```

```

definition distinct-ds :: 'a::linorder list  $\Rightarrow$  bool
  where distinct-ds l  $\equiv$  no-stutter (quicksort l)

```

```

lemma [code-unfold]: distinct = distinct-ds
  <proof>

```

2.4.4 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
in SML
module-name PDDL-Checker-Exported
file code/PDDL-STRIPS-Checker-Exported.sml

```

```

end — Theory
theory Lifschitz-Consistency
imports PDDL-STRIPS-Semantics
begin

```

2.4.5 Soundness theorem for the STRIPS semantics

We prove the soundness theorem according to [Lif87].

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* \rightarrow *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 [Lif87]

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

definition *close-eq* $s = (\lambda \text{predAtm } p\ xs \Rightarrow s\ (p, xs) \mid Eq\ a\ b \Rightarrow a=b)$

lemma *close-eq-predAtm[simp]*: *close-eq* $s\ (\text{predAtm } p\ xs) \longleftrightarrow s\ (p, xs)$
 $\langle \text{proof} \rangle$

lemma *close-eq-Eq[simp]*: *close-eq* $s\ (Eq\ a\ b) \longleftrightarrow a=b$
 $\langle \text{proof} \rangle$

abbreviation *entail-eq* :: *state* \Rightarrow *object atom formula* \Rightarrow *bool* (**infix** $\models =$ 55)
where *entail-eq* $s\ f \equiv \text{close-eq } s \models f$

fun *sound-opr* :: *ground-action* \Rightarrow *action* \Rightarrow *bool* **where**
sound-opr (*Ground-Action* *pre* (*Effect* *add* *del*)) $f \longleftrightarrow$
 $(\forall s. s \models = \text{pre} \longrightarrow$

$$\begin{aligned}
& (\exists s'. f s = \text{Some } s' \wedge (\forall atm. \text{is-predAtom } atm \wedge atm \notin \text{set del} \wedge s \models_{=} \\
& atm \longrightarrow s' \models_{=} atm) \\
& \wedge (\forall atm. \text{is-predAtom } atm \wedge atm \notin \text{set add} \wedge s \models_{=} \text{Not } atm \longrightarrow s' \\
& \models_{=} \text{Not } atm) \\
& \wedge (\forall fmla. fmla \in \text{set add} \longrightarrow s' \models_{=} fmla) \\
& \wedge (\forall fmla. fmla \in \text{set del} \wedge fmla \notin \text{set add} \longrightarrow s' \models_{=} (\text{Not } fmla)) \\
&)) \\
& \wedge (\forall fmla \in \text{set add}. \text{is-predAtom } fmla)
\end{aligned}$$

lemma *sound-opr-alt*:

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

Definition B (v)–(vii) in [Lif87]

definition *sound-system*

$$\begin{aligned}
& :: \text{ground-action set} \\
& \Rightarrow \text{world-model} \\
& \Rightarrow \text{state} \\
& \Rightarrow (\text{ground-action} \Rightarrow \text{action}) \\
& \Rightarrow \text{bool}
\end{aligned}$$

where

$$\begin{aligned}
& \text{sound-system } \Sigma M_0 s_0 f \longleftrightarrow \\
& ((\forall fmla \in \text{close-world } M_0. s_0 \models_{=} fmla) \\
& \wedge (*\forall s. \forall fmla \in M_0. \neg \text{is-Atom } fmla \longrightarrow s \models fmla*) \text{wm-basic } M_0 \\
& \wedge (\forall \alpha \in \Sigma. \text{sound-opr } \alpha (f \alpha)))
\end{aligned}$$

Composing two actions

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

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

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

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

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

Soundness Theorem of [Lif87].

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 [Lif87])

obtains s'

— We have that $f(\alpha s)$ is applicable in initial state, yielding state s' (called $f_{\alpha s}(s_0)$ in [Lif87])

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*

2.5 Soundness Theorem for PDDL

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

Mapping world models to states

definition *state-to-wm* :: *state* \Rightarrow *world-model*

where *state-to-wm* $s = (\{\text{formula.Atom } (\text{predAtm } p\ xs) \mid p\ xs.\ s\ (p, xs)\})$

definition *wm-to-state* :: *world-model* \Rightarrow *state*

where *wm-to-state* $M = (\lambda(p, xs). (\text{formula.Atom } (\text{predAtm } p\ xs)) \in M)$

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

$\langle \text{proof} \rangle$

lemma *wm-to-state-inv[simp]*: *wm-to-state (state-to-wm s) = s*
 ⟨proof⟩

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)) \models (precondition g-opr) then
Some (wm-to-state (apply-effect (effect g-opr) M))
else
None)

definition *close-eq-M M =* (*M* \cap {*Atom (predAtm p xs) | p xs. True* }*)* \cup {*Atom*
(Eq a a) | a. True} \cup { \neg (*Atom (Eq a b)*) | *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)*)
 ⟨proof⟩

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*)
 ⟨proof⟩

lemma *not-dels-preserved*:
assumes *f \notin (set d) f \in M*
shows *f \in apply-effect (Effect a d) M*
 ⟨proof⟩

lemma *adds-satisfied*:
assumes *f \in (set a)*
shows *f \in apply-effect (Effect a d) M*
 ⟨proof⟩

lemma *dels-unsatisfied*:
assumes *f \in (set d) f \notin set a*
shows *f \notin apply-effect (Effect a d) M*
 ⟨proof⟩

lemma *dels-unsatisfied-2*:
assumes *f \in set (dels eff) f \notin set (adds eff)*
shows *f \notin apply-effect eff M*
 ⟨proof⟩

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-impt-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  $\pi$ 
shows sound-opr
  (resolve-instantiate  $\pi$ )
  ((pddl-opr-to-act (resolve-instantiate  $\pi$ )))
  <proof>

lemma wf-world-model-has-atoms:  $f \in M \implies \text{wf-world-model } M \implies \text{is-predAtom}$ 
 $f$ 
  <proof>

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

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

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

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

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

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

end — Context of wf-ast-problem

end

```

3 Reasoning about Invariants

```

theory invariant-verification
  imports PDDL-STRIPS-Semantics
begin
  <proof><proof><proof><proof>

```

```

context ast-problem begin

```

An invariant is preserved by all actions of the problem

definition *is-invariant* $Q \equiv \forall \pi M. Q M \wedge \text{plan-action-enabled } \pi M \longrightarrow Q (\text{execute-plan-action } \pi M)$

Q is preserved by action instance α .

definition *invariant-act* $Q \alpha =$
 $(\forall M. Q M \wedge M \models \text{precondition } \alpha \longrightarrow Q (\text{execute-ground-action } \alpha M))$

Q is preserved by executing a sequence of action instances αs .

definition *invariant* $Q \alpha s =$
 $(\forall M M'. Q M \wedge \text{ground-action-path } M \alpha s M' \longrightarrow Q M')$

If all action instances in a set preserve the invariant, also sequences of these instances preserve the invariant.

lemma *invariant-for-as*:
assumes $\bigwedge \alpha. \alpha \in A \implies \text{invariant-act } Q \alpha$
assumes $\text{set } \alpha s \subseteq A$
shows *invariant* $Q \alpha s$
 <proof>

Invariant wrt. a plan action. Note that the invariant only needs to hold for well-formed plan actions, as implicitly contained in *plan-action-enabled*.

definition *invariant-wf-plan-act* $Q \pi$
 $= (\forall M. Q M \wedge \text{plan-action-enabled } \pi M \longrightarrow Q (\text{execute-plan-action } \pi M))$

We can introduce an invariant by showing that it is invariant for any possible action instance.

lemma *invariant-action-insts-imp-invariant-plan-actions*:
assumes $\bigwedge a \text{ args}. a \in \text{set } (\text{actions } D) \wedge \text{action-params-match } a \text{ args}$
 $\implies \text{invariant-act } Q (\text{instantiate-action-schema } a \text{ args})$
shows *invariant-wf-plan-act* $Q \pi$
 <proof>

lemma *invariant-gActs-imp-invariant-pActs*:
assumes $\forall a \in \text{set } (\text{actions } D). \forall \text{args}. \text{action-params-match } a \text{ args}$
 $\longrightarrow \text{invariant-act } Q (\text{instantiate-action-schema } a \text{ args})$
shows *invariant-wf-plan-act* $Q \pi$
 <proof>

Invariant wrt. a sequence of well-formed plan actions.

definition *invariant-wf-plan-act-seq* $Q \pi s$
 $= (\forall M M'. Q M \wedge \text{plan-action-path } M \pi s M' \longrightarrow Q M')$

An invariant wrt. all plan actions is preserved by paths

lemma *invariant-for-plan-act-seq*:
assumes $\bigwedge \pi. \text{invariant-wf-plan-act } Q \pi$
shows *invariant-wf-plan-act-seq* $Q \pi s$
 $\langle \text{proof} \rangle$
end

end
theory *Paper-TeXgen*
imports *Main*
begin

fun *distinct* :: 'a list \Rightarrow bool **where**
distinct [] \longleftrightarrow True |
distinct (x # xs) \longleftrightarrow x \notin set xs \wedge *distinct* xs

end