

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	8
1.5	Well-Formedness of PDDL	9
1.6	PDDL Semantics	13
1.7	Preservation of Well-Formedness	15
1.7.1	Well-Formed Action Instances	15
1.7.2	Preservation	19
2	Executable PDDL Checker	21
2.1	Generic DFS Reachability Checker	22
2.2	Implementation Refinements	24
2.2.1	Of-Type	24
2.2.2	Application of Effects	27
2.2.3	Well-Formedness	28
2.2.4	Execution of Plan Actions	29
2.2.5	Checking of Plan	31
2.3	Executable Plan Checker	33
2.4	Code Setup	35
2.4.1	Code Equations	36
2.4.2	Setup for Containers Framework	37

2.4.3	More Efficient Distinctness Check for Linorders	37
2.4.4	Code Generation	37
2.4.5	Soundness theorem for the STRIPS semantics	38
2.5	Soundness Theorem for PDDL	42
3	Reasoning about Invariants	47

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

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

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

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

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

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* \times *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* \times *name*) *list*) — (*type*, *supertype*) declarations.
 (*predicates*: *predicate-decl list*)
 (*consts*: (*object* \times *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* $M =$
 $M \cup \{\neg(\text{Atom } (\text{predAtm } p \text{ as})) \mid p \text{ as. } \text{Atom } (\text{predAtm } p \text{ as}) \notin M\}$
 $\cup \{\text{Atom } (\text{Eq } a \text{ a}) \mid a. \text{ True}\} \cup \{\neg(\text{Atom } (\text{Eq } a \text{ b})) \mid a \text{ b. } a \neq b\}$

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

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

unfolding *close-world-def close-neg-def wm-basic-def*

apply *clarsimp*

apply (*auto 0 3*)

by (*metis atom.exhaust*)

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

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

lemma

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

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

by (*auto simp: close-world-def*)

lemma *in-close-world-conv*:

$\varphi \in \text{close-world } M \longleftrightarrow ($

$\varphi \in M$

$\vee (\exists p \text{ as. } \varphi = \neg(\text{Atom } (\text{predAtm } p \text{ as})) \wedge \text{Atom } (\text{predAtm } p \text{ as}) \notin M)$

$\vee (\exists a. \varphi = \text{Atom } (\text{Eq } a \text{ a}))$

$\vee (\exists a \text{ b. } \varphi = \neg(\text{Atom } (\text{Eq } a \text{ b})) \wedge a \neq b)$

$)$

by (*auto simp: close-world-def*)

lemma *valuation-aux-1*:

fixes $M :: \text{world-model}$ **and** $\varphi :: \text{object atom formula}$

defines $C \equiv \text{close-world } M$

assumes $A: \forall \varphi \in C. \mathcal{A} \models \varphi$

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

using A **unfolding** $C\text{-def}$

apply $-$

apply (*auto simp: in-close-world-conv valuation-def Ball-def intro!: ext split: atom.split*)

apply (*metis formula-semantics.simps(1) formula-semantics.simps(3)*)

apply (*metis formula-semantics.simps(1) formula-semantics.simps(3)*)

by (*metis atom.collapse(2) formula-semantics.simps(1) is-predAtm-def*)

lemma *valuation-aux-2*:

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

lemma *val-imp-close-world: valuation M $\models \varphi \implies M \models \varphi$*
unfolding *entailment-def*
using *valuation-aux-1*
by *blast*

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

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

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

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

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

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

```

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

```

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

```

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

```

1.4 STRIPS Semantics

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

```

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

```

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

```

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

```

Execute a ground action

```

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

```

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  $\models^c$  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  $\models^c$  precondition  $\alpha$ 
   $\wedge$  (ground-action-path (apply-effect (effect  $\alpha$ ) M)  $\alpha s$  M')
by (auto simp: execute-ground-action-def)
```

end — Context of *ast-domain*

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

context *ast-domain* **begin**

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

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

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

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

```
fun subtype-edge where
```

$subtype-edge\ (ty, superty) = (superty, ty)$

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

definition $of-type :: type \Rightarrow type \Rightarrow bool$ **where**
 $of-type\ oT\ T \equiv set\ (primitives\ oT) \subseteq subtype-rel^* \text{ `` } set\ (primitives\ T)$

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**
 $is-of-type\ v\ T \longleftrightarrow ($
 $\quad case\ ty-ent\ v\ of$
 $\quad \quad Some\ vT \Rightarrow of-type\ vT\ T$
 $\quad | \quad None \Rightarrow False)$

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

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**
 $wf-atom\ (predAtm\ p\ vs) \longleftrightarrow wf-pred-atom\ (p, vs)$
 $| \quad wf-atom\ (Eq\ a\ b) \longleftrightarrow ty-ent\ a \neq None \wedge ty-ent\ b \neq None$

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**
 $wf-fmla\ (Atom\ a) \longleftrightarrow wf-atom\ a$
 $| \quad wf-fmla\ (\perp) \longleftrightarrow True$
 $| \quad wf-fmla\ (\varphi1 \wedge \varphi2) \longleftrightarrow (wf-fmla\ \varphi1 \wedge wf-fmla\ \varphi2)$
 $| \quad wf-fmla\ (\varphi1 \vee \varphi2) \longleftrightarrow (wf-fmla\ \varphi1 \wedge wf-fmla\ \varphi2)$
 $| \quad wf-fmla\ (\neg \varphi) \longleftrightarrow wf-fmla\ \varphi$
 $| \quad wf-fmla\ (\varphi1 \rightarrow \varphi2) \longleftrightarrow (wf-fmla\ \varphi1 \wedge wf-fmla\ \varphi2)$

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

Special case for a well-formed atomic predicate formula

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

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

fun *wf-effect* **where**
wf-effect (*Effect* *a* *d*) \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 $\equiv \text{map-of } (\text{consts } D)$

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*)
 $\wedge \text{wf-fmla } \text{tyt } \text{pre}$
 $\wedge \text{wf-effect } \text{tyt } \text{eff}$)

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*) $\longleftrightarrow \text{set } Ts \subseteq \text{insert "object" } (\text{fst'set } (\text{types } D))$

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

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 *map-of* (*objects P*) ++ *constT*

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

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

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

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

definition *wf-problem* **where**

```

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

```

```

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

```

```

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

```

end — locale *ast-problem*

Locale to express a well-formed domain

```

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

```

Locale to express a well-formed problem

```

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

```

begin

```

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

```

end — locale *wf-ast-problem*

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
apply (intro ext)
unfolding is-obj-of-type-def is-of-type-def by auto

```

HOL encoding of matching an action's formal parameters against an argument list. The parameters of the action are encoded as a list of *name* \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
   $\equiv$  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  $\Rightarrow$  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  $\pi$ s M' =
    (( $\forall \pi \in \text{set } \pi$ s. wf-plan-action  $\pi$ )
      $\wedge$  ground-action-path M (map resolve-instantiate  $\pi$ s) M')

```

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

```

definition valid-plan-from :: world-model  $\Rightarrow$  plan  $\Rightarrow$  bool where
  valid-plan-from M  $\pi$ s = ( $\exists M'$ . plan-action-path M  $\pi$ s M'  $\wedge$  M'  $\models$  (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  $\pi$ s  $\equiv \exists M'$ . plan-action-path I  $\pi$ s M'  $\wedge$  M'  $\models$  (goal P)
unfolding valid-plan-def valid-plan-from-def by auto

```

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
  unfolding of-type-def by auto

```

```

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

```

```

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

```

```

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

```

```

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

```


by (cases a rule: wf-fmla-atom.cases) (auto split: option.splits dest: map-leD[OF SS])
qed

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

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

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

context
fixes *Q* **and** *f* :: *variable* \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*
unfolding *is-of-type-def* **by** (auto)

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

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

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

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

```

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

```

qed

```

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

```

```

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

```

```

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

```

end

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

```

theorem wf-instantiate-action-schema:
  assumes action-params-match a args
  assumes wf-action-schema a
  shows wf-ground-action (instantiate-action-schema a args)
proof (cases a)

```

```

case [simp]: (Action-Schema name params pre eff)
have INST:
  is-of-type objT ((the ∘ map-of (zip (map fst params) args)) x) T
  if is-of-type (map-of params) x T for x T
  using that
  apply (rule is-of-type-map-ofE)
  using assms
  apply (clarsimp simp: Let-def)
  subgoal for i xT
    unfolding action-params-match-def
    apply (subst lookup-zip-idx-eq[where i=i];
      (clarsimp simp: list-all2-lengthD)?)
    apply (frule list-all2-nthD2[where p=i]; simp?)
    apply (auto
      simp: is-obj-of-type-alt is-of-type-def
      intro: of-type-trans
      split: option.splits)
    done
  done
then show ?thesis
  using assms(2) wf-inst-formula wf-inst-effect
  apply (fastforce split: term.splits simp: Let-def comp-apply[abs-def])
  done
qed
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 π M
 \longleftrightarrow wf-plan-action $\pi \wedge M \models_{\text{c}} \text{precondition } (\text{resolve-instantiate } \pi)$

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 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 $M \sqsubseteq M' \longleftrightarrow M' = M$
by (auto simp: plan-action-path-def)

```

lemma plan-action-path-Cons[simp]:
  plan-action-path  $M$  ( $\pi \# \pi s$ )  $M' \longleftrightarrow$ 
    plan-action-enabled  $\pi$   $M$ 
   $\wedge$  plan-action-path (execute-plan-action  $\pi$   $M$ )  $\pi s$   $M'$ 
by (auto)
  simp: plan-action-path-def execute-plan-action-def
    execute-ground-action-def plan-action-enabled-def)

```

end — Context of *ast-problem*

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

The initial world model is well-formed

```

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

```

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

```

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

```

Execution of plan actions preserves well-formedness

```

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

  from (plan-action-enabled  $\pi$   $s$ ) have wf-plan-action  $\pi$ 
    unfolding plan-action-enabled-def by auto
  then obtain  $a$  where
    resolve-action-schema  $name = \text{Some } a$  and
     $T$ : action-params-match  $a$   $args$ 
    by (auto split: option.splits)

```

```

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

```

```

from ⟨resolve-action-schema name = Some a⟩ have
  a ∈ set (actions D)
  unfolding resolve-action-schema-def by auto
with wf-domain have wf-action-schema a
  unfolding wf-domain-def by auto
hence wf-ground-action (resolve-instantiate π)
  using ⟨resolve-action-schema name = Some a⟩ T
  wf-instantiate-action-schema
  by auto
thus ?thesis
  apply (simp add: execute-plan-action-def execute-ground-action-def)
  apply (rule wf-apply-effect)
  apply (cases resolve-instantiate π; simp)
  by (rule ⟨wf-world-model s⟩)
qed

```

```

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

```

Execution of a plan preserves well-formedness

```

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

```

end — Context of *wf-ast-problem*

end — Theory

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*: $\text{set } (\text{succ } x) = \text{E-of-succ succ} \text{ `` } \{x\}$
unfolding *E-of-succ-def* **by** *auto*

context

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

begin

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

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

context

fixes *w₀* :: 'a list

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

begin

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

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

lemma *card-decreases*:

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

```

apply (rule psubset-card-mono)
apply (auto simp: dfs-reachable-invar-def)
done

lemma all-neq-Cons-is-Nil[simp]:
  ( $\forall y \text{ ys. } x2 \neq y \# \text{ys} \iff x2 = []$ ) by (cases x2) auto

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

end

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

lemma Some-eq-map-option [iff]:  $(\text{Some } y = \text{map-option } f \ xo) = (\exists z. \ xo = \text{Some } z \wedge f \ z = y)$ 
by (auto simp add: map-option-case split: option.split)

lemma tab-succ-correct:  $E\text{-of-succ } (\text{tab-succ } l) = \text{set } l$ 
proof –
  have  $\text{set } (\text{Mapping.lookup-default } [] \ (\text{fold } (\lambda(u, v). \text{Mapping.map-default } u \ [] \ (\text{Cons } v)) \ l \ m) \ u) = \text{set } l \text{ `` } \{u\} \cup \text{set } (\text{Mapping.lookup-default } [] \ m \ u)$ 
    for  $m \ u$ 
    apply (induction l arbitrary: m)
    by (auto
      simp: Mapping.lookup-default-def Mapping.map-default-def Mapping.default-def
      simp: lookup-map-entry' lookup-update' keys-is-none-rep Option.is-none-def
      split: if-splits
    )
  from this[where  $m = \text{Mapping.empty}$ ] show ?thesis
  by (auto simp: E-of-succ-def tab-succ-def lookup-default-empty)
qed

```

end

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

$\llbracket \text{finite } E; \text{finite } S \rrbracket \implies \text{finite } (E^* \text{“} S)$
apply (rule *finite-subset*[**where** $B=S \cup (\text{Relation.Domain } E \cup \text{Relation.Range } E)$])
apply (auto simp: intro: *finite-Domain finite-Range elim: rtranclE*)
done

lemma *dfs-reachable-tab-succ-correct*: $\text{dfs-reachable } (\text{tab-succ } l) D \text{ vs}_0 \longleftrightarrow \text{Collect } D \cap (\text{set } l)^* \text{“} \text{set vs}_0 \neq \{\}$
apply (subst *dfs-reachable-correct*)
by (simp-all add: *tab-succ-correct finite-imp-finite-dfs-reachable*)

2.2 Implementation Refinements

2.2.1 Of-Type

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

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

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

lemma *ty-term'-correct-aux*: $\text{ty-term}' \text{ varT objT } t = \text{ty-term } \text{varT } (\text{Mapping.lookup objT } t)$
by (cases t) auto

lemma *ty-term'-correct[simp]*: $\text{ty-term}' \text{ varT objT } = \text{ty-term } \text{varT } (\text{Mapping.lookup objT})$
using *ty-term'-correct-aux* **by** auto

context *ast-domain* **begin**

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

lemma *of-type-refine1*: $\text{of-type } \text{oT } T \longleftrightarrow (\forall pt \in \text{set } (\text{primitives } \text{oT}). \text{of-type1 } pt \text{ } T)$
unfolding *of-type-def of-type1-def* **by** auto

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

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

lemma *of-type1-impl*: $\text{of-type1 } pt \text{ } T \longleftrightarrow \text{dfs-reachable } (\text{tab-succ } (\text{map subtype-edge } (\text{types } D))) \text{ vs}_0$

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

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

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

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

Lifting the subtype-graph through wf-checker

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

definition is-of-type' stg v T \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
 unfolding is-of-type'-def is-of-type-def of-type-impl-correct ..

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

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

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 φ

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

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

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

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

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

end — Context fixing *ty-ent*

fun *wf-action-schema'* :: $- \Rightarrow - \Rightarrow \text{ast-action-schema} \Rightarrow \text{bool}$ **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*: $\text{wf-action-schema}' \text{ STG mp-constT } s = \text{wf-action-schema } s$
by (*cases* *s*) (*auto simp*: *wf-fmla'-correct wf-effect'-correct*)

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*: $\text{wf-domain}' \text{ STG mp-constT} = \text{wf-domain}$
unfolding *wf-domain-def wf-domain'-def*

by (auto simp: wf-action-schema'-correct)

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
    = fold ( $\lambda$ add s. Set.insert add s) a
      (fold ( $\lambda$ del s. Set.remove del s) d s)

```

lemma apply-effect-exec-refine[simp]:

```

  apply-effect-exec (Effect (a) (d)) s
    = apply-effect (Effect (a) (d)) s

```

proof(induction a arbitrary: s)

case Nil

then show ?case

proof(induction d arbitrary: s)

case Nil

then show ?case by auto

next

case (Cons a d)

then show ?case

by (auto simp add: image-def)

qed

next

case (Cons a a)

then show ?case

proof(induction d arbitrary: s)

case Nil

then show ?case by (auto; metis Set.insert-def sup-assoc insert-iff)

next

case (Cons a d)

then show ?case

by (auto simp: Un-commute minus-set-fold union-set-fold)

qed

qed

lemmas apply-effect-eq-impl-eq

```

  = apply-effect-exec-refine[symmetric, unfolded apply-effect-exec.simps]

```

end — Context of *ast-domain*

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*) *mapping*

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

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

We refine the typecheck to use the mapping

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

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

We refine the well-formedness checks to use the mapping

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

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

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

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

definition *wf-problem' stg conT mp* \equiv
wf-domain' stg conT
 \wedge *distinct (map fst (objects P) @ map fst (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*:
 $\text{wf-problem}' \text{ STG mp-constT mp-objT} = \text{wf-problem}$
unfolding *wf-problem-def wf-problem'-def wf-world-model-def*
by (*auto simp: wf-domain'-correct wf-fmla'-correct*)

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

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

end — Context of *ast-problem*

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

Resolving an action yields a well-founded action schema.

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

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

end — Context of *ast-domain*

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
  unfolding holds-def using assms valuation-iff-close-world
  by blast
```

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

```
definition en-exE :: plan-action  $\Rightarrow$  world-model  $\Rightarrow$   $\rightarrow$  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
  apply (cases a)
  using assms holds-for-wf-fmlas wf-domain
  unfolding plan-action-enabled-def execute-plan-action-def
  and execute-ground-action-def en-exE-def wf-domain-def
  by (auto
    split: option.splits
    simp: resolve-action-schemaE-def return-iff)
```

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

abbreviation *action-params-match2* *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*
apply (*cases pa*; *simp add: en-exE2-def en-exE-def Let-def*)
apply (*auto*)
simp: return-iff resolve-action-schemaE-def resolve-action-wf
simp: wf-effect-inst-weak action-params-match-def
split: error-monad-bind-split)
done

Combination of the two refinement lemmas

lemma (*in wf-ast-problem*) *en-exE2-return-iff*:
assumes *wm-basic M*
shows *en-exE2* *STG mp-objT a M* $=$ *Inr M'*
 \longleftrightarrow *plan-action-enabled a M* \wedge *M' = execute-plan-action a M*
unfolding *wf-en-exE2-eq*
apply (*subst en-exE-return-iff*)
using *assms*
by (*auto*)

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

end — Context of *ast-problem*

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  $\pi s$ 
proof(induction  $\pi s$  arbitrary: s)
  case Nil
  then show ?case by (auto simp add: plan-action-path-def valid-plan-from-def)
next
  case (Cons a  $\pi s$ )
  then show ?case
    by (auto
      simp: valid-plan-from-def plan-action-path-def plan-action-enabled-def
      simp: execute-ground-action-def execute-plan-action-def)
qed

```

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

```

fun valid-plan-fromE
  :: -  $\Rightarrow$  (object, type) mapping  $\Rightarrow$  nat  $\Rightarrow$  world-model  $\Rightarrow$  plan  $\Rightarrow$  -+unit
where
  valid-plan-fromE stg mp si s []
    = check (holds s (goal P)) (ERRS "Postcondition does not hold")
| valid-plan-fromE stg mp si s ( $\pi \# \pi s$ ) = do {
  s  $\leftarrow$  en-exE2 stg mp  $\pi$  s
   $\leftarrow$  ? ( $\lambda e \dots$  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:
  [ wm-basic s; wf-plan-action a ]
   $\implies$  wm-basic (execute-plan-action a s)
using wf-problem wf-domain
unfolding wf-problem-def wf-domain-def
apply (cases a)
apply (clarsimp
  split: option.splits
  simp: wf-fmla-atom-alt execute-plan-action-def wm-basic-def
  simp: execute-ground-action-def)
subgoal for n args schema fmla
  apply (cases effect (instantiate-action-schema schema args); simp)

```



```

    by (metis ground-action.sel(2) ast-domain.wf-effect.simps
        ast-domain.wf-fmla-atom-alt resolve-action-wf
        wf-ground-action.elims(2) wf-instantiate-action-schema)
  done

```

Refinement lemma for our plan checking algorithm

```

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

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

```

end — Context of *ast-problem*

2.3 Executable Plan Checker

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

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

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

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

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

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

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

 $(\ast$
 check $(\forall a \in \text{set } (\text{actions } D). \text{ast-domain.wf-action-schema' } D \text{ stg } a) (\text{ERRS$
 "Malformed action"})
 $\ast)$
 $\}$***

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

proof –

interpret *ast-domain* D .

show ?thesis

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

by (*auto simp: return-iff*)

qed

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

definition *check-wf-problem* $P \text{ stg conT mp} \equiv \text{do } \{$
 let $D = \text{ast-problem.domain } P$;
 check-wf-domain $D \text{ stg conT} <+ ? \text{prepend-err-msg "Domain not well-formed"};$
 $(\ast \text{check } (\text{ast-domain.wf-domain' } D \text{ stg}) (\text{ERRS "Domain not well-formed"}); \ast)$
 $\}$

```

    check (distinct (map fst (objects P) @ map fst (consts D))) (ERRS "Duplicate
object declaration");
    check (( $\forall (n, T) \in \text{set } (\text{objects } P).$  ast-domain.wf-type D T)) (ERRS "Malformed
type");
    check (distinct (init P)) (ERRS "Duplicate fact in initial state");
    check ( $\forall f \in \text{set } (\text{init } P).$  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 –

interpret *ast-problem P* .

show ?thesis

unfolding *check-wf-problem-def wf-problem'-def*

by (*auto simp: return-iff*)

qed

definition *check-plan P πs* \equiv *do* {

let stg = *ast-domain.STG (ast-problem.domain P)*;

let conT = *ast-domain.mp-constT (ast-problem.domain P)*;

let mp = *ast-problem.mp-objT P*;

check-wf-problem P stg conT mp;

(**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

} <+? ($\lambda e.$ *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-plan P πs = Inr ()*

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

proof –

interpret *ast-problem P* .

show ?thesis

unfolding *check-plan-def*

by (*auto*

simp: return-iff wf-world-model-def wf-fmla-atom-alt I-def wf-problem-def

isOK-iff

simp: wf-problem'-correct ast-problem.I-def ast-problem.valid-plan-def wm-basic-def

)

qed

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 ⇒ bool where
  no-stutter [] = True
| no-stutter [-] = True
| no-stutter (a#b#l) = (a≠b ∧ no-stutter (b#l))

```

```

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

```

```

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

```

```

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

```

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* $s\ (predAtm\ p\ xs) \longleftrightarrow s\ (p, xs)$
by (*auto simp: close-eq-def*)

lemma *close-eq-Eq[simp]*: *close-eq* $s\ (Eq\ a\ b) \longleftrightarrow a=b$
by (*auto simp: close-eq-def*)

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))  $f \longleftrightarrow$ 
    ( $\forall s. s \models pre \longrightarrow$ 
      ( $\exists s'. f\ s = \text{Some } s' \wedge (\forall atm. is-predAtom\ atm \wedge atm \notin set\ del \wedge s \models$ 
atm  $\longrightarrow s' \models atm)$ )

```

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

lemma *sound-opr-alt*:

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

Definition B (v)–(vii) in [Lif87]

definition *sound-system*

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

where

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

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]*:

$$compose-actions\ [] = Some\ \text{unfolding } compose-actions-def\ \text{by } auto$$

lemma *compose-actions-Cons*[*simp*]:
 $f\ s = \text{Some } s' \implies \text{compose-actions } (f \# fs)\ s = \text{compose-actions } fs\ s'$
proof –
interpret *monoid-add compose-action Some*
apply *unfold-locales*
unfolding *compose-action-def*
by (*auto split: option.split*)
assume $f\ s = \text{Some } s'$
then show *?thesis*
unfolding *compose-actions-def*
by (*simp add: compose-action-def fold-plus-sum-list-rev*)
qed

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 $\forall fmla \in \text{close-world } M'.\ s' \models fmla$
proof –
have (*valuation* $M' \models fmla$) **if** *wm-basic* $M'\ fmla \in M'$ **for** $fmla$
using *that* **apply** (*induction* $fmla$)
by (*auto simp: valuation-def wm-basic-def split: atom.split*)
have $\exists s'. \text{compose-actions } (\text{map } f\ \alpha s)\ s_0 = \text{Some } s' \wedge (\forall fmla \in \text{close-world } M'.\ s' \models fmla)$
using *assms*
proof(*induction* αs *arbitrary: s0 M0*)
case *Nil*
then show *?case* **by** (*auto simp add: close-world-def compose-action-def sound-system-def*)
next
case *ass: (Cons $\alpha\ \alpha s$)*
then obtain *pre add del* **where** $a: \alpha = \text{Ground-Action } pre\ (Effect\ add\ del)$
using *ground-action.exhaust ast-effect.exhaust* **by** *metis*
let $?M_1 = \text{execute-ground-action } \alpha\ M_0$
have *close-world* $M_0 \models \text{precondition } \alpha$
using *ass(4)*
by *auto*
moreover have *s0-ent-cwM0*: $\forall fmla \in (\text{close-world } M_0). \text{close-eq } s_0 \models fmla$
using *ass(2)*


```

    unfolding sound-system-def
    by auto
  ultimately have s0-ent-alpha-precond: close-eq s0  $\models$  precondition  $\alpha$ 
    unfolding entailment-def
    by auto
  then obtain s1 where s1: (f  $\alpha$ ) s0 = Some s1
    (∀ atm. is-predAtom atm  $\longrightarrow$  atm  $\notin$  set(dels (effect  $\alpha$ ))
       $\longrightarrow$  close-eq s0  $\models$  atm
       $\longrightarrow$  close-eq s1  $\models$  atm)
    (∀ fmla. fmla  $\in$  set(adds (effect  $\alpha$ ))
       $\longrightarrow$  close-eq s1  $\models$  fmla)
    (∀ atm. is-predAtom atm  $\wedge$  atm  $\notin$  set (adds (effect  $\alpha$ ))  $\wedge$  close-eq s0  $\models$  Not
    atm  $\longrightarrow$  close-eq s1  $\models$  Not atm)
    (∀ fmla. fmla  $\in$  set (dels (effect  $\alpha$ ))  $\wedge$  fmla  $\notin$  set(adds (effect  $\alpha$ ))  $\longrightarrow$  close-eq
    s1  $\models$  (Not fmla))
    (∀ a b. close-eq s0  $\models$  Atom (Eq a b)  $\longrightarrow$  close-eq s1  $\models$  Atom (Eq a b))
    (∀ a b. close-eq s0  $\models$  Not (Atom (Eq a b))  $\longrightarrow$  close-eq s1  $\models$  Not (Atom (Eq
    a b))))
    using ass(2-4)
  unfolding sound-system-def sound-opr-alt by force
  have close-eq s1  $\models$  fmla if fmla $\in$ close-world ?M1 for fmla
    using ass(2)
    using that s1 s0-ent-cwM0
  unfolding sound-system-def execute-ground-action-def wm-basic-def
  apply (auto simp: in-close-world-conv)
  subgoal
  by (metis (no-types, lifting) DiffE UnE a apply-effect.simps ground-action.sel(2)
    ast-effect.sel(1) ast-effect.sel(2) close-world-extensive subsetCE)
  subgoal
  by (metis Diff-iff Un-iff a ground-action.sel(2) ast-domain.apply-effect.simps
    ast-domain.close-eq-predAtom ast-effect.sel(1) ast-effect.sel(2) formula-semantics.simps(1)
    formula-semantics.simps(3) in-close-world-conv is-predAtom.simps(1))
  done
  moreover have (∀ atm. fmla  $\neq$  formula.Atom atm)  $\longrightarrow$  s  $\models$  fmla if fmla $\in$ ?M1
  for fmla s
  proof-
    have alpha: (∀ s. ∀ fmla $\in$ set(adds (effect  $\alpha$ )).  $\neg$  is-predAtom fmla  $\longrightarrow$  s  $\models$ 
    fmla)
      using ass(2,3)
      unfolding sound-system-def ast-domain.sound-opr-alt
      by auto
    then show ?thesis
      using that
      unfolding a execute-ground-action-def
      using ass.premis(1)[unfolded sound-system-def]
      by(cases fmla; fastforce simp: wm-basic-def)

  qed
  moreover have (∀ opr $\in$ Σ. sound-opr opr (f opr))

```

```

    using ass(2) unfolding sound-system-def
    by (auto simp add;)
  moreover have wm-basic ?M1
  using ass(2,3)
  unfolding sound-system-def execute-ground-action-def
  thm sound-opr.cases
  apply (cases ( $\alpha, f \ \alpha$ ) rule: sound-opr.cases)
  apply (auto simp: wm-basic-def)
  done
ultimately have sound-system  $\Sigma$  ?M1 s1 f
unfolding sound-system-def
by (auto simp: wm-basic-def)
from ass.IH[OF this] ass.prems obtain s' where
  compose-actions ( $\text{map } f \ \alpha s$ ) s1 = Some s'  $\wedge (\forall a \in \text{close-world } M'. s' \models a)$ 
by auto
thus ?case by (auto simp: s1(1))
qed
with that show ?thesis by blast
qed

```

More compact notation of the soundness theorem.

```

theorem STRIPS-sema-sound-compact-version:
  sound-system  $\Sigma$  M0 s0 f  $\implies \text{set } \alpha s \subseteq \Sigma$ 
 $\implies \text{ground-action-path } M_0 \ \alpha s \ M'$ 
 $\implies \exists s'. \text{compose-actions } (\text{map } f \ \alpha s) \ s_0 = \text{Some } s'$ 
 $\wedge (\forall fmla \in \text{close-world } M'. s' \models fmla)$ 
using STRIPS-sema-sound by metis

```

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)  $\longleftrightarrow \text{Atom } (\text{predAtm } p \ as)$ 
 $\in M$ 
by (auto simp: wm-to-state-def)

```

```

lemma wm-to-state-inv[simp]: wm-to-state (state-to-wm s) = s

```

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

Mapping AST action instances to actions

definition *pddl-opr-to-act g-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 =* ($M \cap \{Atom (predAtm p xs) \mid p xs. True\} \cup \{Atom$
 $(Eq a a) \mid a. True\} \cup \{\neg(Atom (Eq a b)) \mid a b. a \neq b\}$)

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

lemma *atom-in-wm-2-eq:*
close-eq (wm-to-state M) \models (formula.Atom atm)
 $\longleftrightarrow ((formula.Atom atm) \in close-eq-M M)$
by (*auto simp: wm-to-state-def*
state-to-wm-def image-def close-eq-def close-eq-M-def split:atom.splits)

lemma *not-dels-preserved:*
assumes $f \notin (set d) \quad f \in M$
shows $f \in apply-effect (Effect a d) M$
using *assms*
by *auto*

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

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

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

```

using assms
by (cases eff; auto)

lemma wf-fmla-atm-is-atom: wf-fmla-atom objT f  $\implies$  is-predAtom f
by (cases f rule: wf-fmla-atom.cases) auto

lemma wf-act-adds-are-atoms:
assumes wf-effect-inst effs ae  $\in$  set (adds effs)
shows is-predAtom ae
using assms
by (cases effs) (auto simp: wf-fmla-atom-alt)

lemma wf-act-adds-dels-atoms:
assumes wf-effect-inst effs ae  $\in$  set (dels effs)
shows is-predAtom ae
using assms
by (cases effs) (auto simp: wf-fmla-atom-alt)

lemma to-state-close-from-state-eq[simp]: wm-to-state (close-world (state-to-wm s)) = s
by (auto simp: wm-to-state-def close-world-def state-to-wm-def image-def)

lemma wf-eff-pddl-ground-act-is-sound-opr:
assumes wf-effect-inst (effect g-opr)
shows sound-opr g-opr ((pddl-opr-to-act g-opr))
unfolding sound-opr-alt
apply(cases g-opr; safe)
subgoal for pre eff s
apply (rule exI[where x=wm-to-state(apply-effect eff (state-to-wm s))])
apply (auto simp: pddl-opr-to-act-def Let-def split:if-splits)
subgoal for atm
by (cases eff; cases atm; auto simp: close-eq-def wm-to-state-def state-to-wm-def split: atom.splits)
subgoal for atm
by (cases eff; cases atm; auto simp: close-eq-def wm-to-state-def state-to-wm-def split: atom.splits)
subgoal for atm
using assms
by (cases eff; cases atm; force simp: close-eq-def wm-to-state-def state-to-wm-def split: atom.splits)
subgoal for fmla
using assms
by (cases eff; cases fmla rule: wf-fmla-atom.cases; force simp: close-eq-def wm-to-state-def state-to-wm-def split: atom.splits)
done
subgoal for pre eff fmla

```

```

using assms
by (cases eff; cases fmla rule: wf-fmla-atom.cases; force)
done

```

```

lemma wf-eff-impt-wf-eff-inst: wf-effect objT eff  $\implies$  wf-effect-inst eff
by (cases eff; auto simp add: wf-fmla-atom-alt)

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

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

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

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

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

  show wm-basic  $I$  using wf-wm-imp-basic[OF wf-I] .

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

theorem wf-plan-soundness-theorem:
  assumes plan-action-path  $I \pi s M$ 
  defines  $\alpha s \equiv \text{map } (pddl\text{-opr-to-act} \circ \text{resolve-instantiate}) \pi s$ 

```

```

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

end — Context of wf-ast-problem

end

```

3 Reasoning about Invariants

```

theory invariant-verification
  imports PDDL-STRIPS-Semantics
begin

```

```

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  $Q \ \alpha s$ 
  using assms(2)
  apply (induction  $\alpha s$ )
  using assms(1)
  by (auto simp: invariant-def invariant-act-def)

```

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* $Q \pi$
unfolding *invariant-wf-plan-act-def*
proof(safe)
fix M
assume $I0: Q M$
assume $EN: \text{plan-action-enabled } \pi M$
obtain $n \text{ args}$ **where** $[\text{simp}]: \pi = (\text{PAction } n \text{ args})$
by (*cases* π)

from EN **obtain** a **where**
 $X1: a \in \text{set } (\text{actions } D) \wedge \text{action-params-match } a \text{ args}$
and $X2: M \stackrel{c}{\models} \text{precondition } (\text{instantiate-action-schema } a \text{ args})$
and $[\text{simp}]: \text{resolve-action-schema } n = \text{Some } a$
by (*auto*)
 $\text{simp: plan-action-enabled-def resolve-action-schema-def}$
 $\text{split: option.splits}$
 $\text{dest: index-by-eq-SomeD}$
from $X1$ **assms** **have** *invariant-act* $Q (\text{instantiate-action-schema } a \text{ args})$
by *blast*
with $I0 EN X2$ **show** $Q (\text{execute-plan-action } \pi M)$
by (*auto simp:*
 $\text{invariant-act-def execute-plan-action-def}$
 $\text{execute-ground-action-def}$)
qed

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

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* $Q \pi s$
apply (*induction* πs)
using *assms*


```

    by (auto simp: invariant-wf-plan-act-seq-def invariant-wf-plan-act-def)
  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

```