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

theory *PDDL-STRIPS-Semantics*
imports
Propositional-Proof-Systems.Formulas
Propositional-Proof-Systems.Sema
begin

1.1 Utility Functions

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

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

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

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

lemma *rtrancl-image-idem*[*simp*]: R^* `` $R^*$ `` *s* = R^* `` *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 *'val atom* = *predAtm* (*predicate*: *predicate*) (*arguments*: *'val list*)

| Eq (lhs: $'val$) (rhs: $'val$)

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 $'val\ ast\text{-}effect = Effect\ (Adds: ('val)\ list)\ (Dels: ('val)\ list)$

Variables are identified by their names.

datatype $variable = varname: Var\ name$

1.2.2 Domains

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

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

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

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

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

datatype $ast\text{-}domain = Domain$
($types$: $name\ list$) — Only supports flat type hierarchy
($predicates$: $predicate\text{-}decl\ list$)
($actions$: $ast\text{-}action\text{-}schema\ list$)

1.2.3 Problems

Objects are identified by their names

datatype $object = name: Obj\ name$

A fact is an atom over objects.

type-synonym $fact = object\ atom$

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

datatype *ast-problem* = *Problem*
  (*domain*: *ast-domain*)
  (*objects*: (*object* × *type*) *list*)
  (*init*: *fact formula list*)
  (*goal*: *fact formula*)

1.2.4 Plans

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

type-synonym *plan* = *plan-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 *ast-action-inst* = *Action-Inst*
  (*precondition*: (*object atom*) *formula*)
  (*effect*: (*object atom*) *formula ast-effect*)

1.3 STRIPS Semantics

Discriminator for atomic formulas.

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

The world model is a set of ground formulas

type-synonym *world-model* = *fact formula set*

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 atom formula ast-effect* ⇒ *world-model* ⇒ *world-model*
  where
    apply-effect (*Effect* (*adds*) (*dels*)) s = (s − (*set dels*)) ∪ ((*set adds*))

Execute an action instance

  definition *execute-ast-action-inst*

$:: \textit{ast-action-inst} \Rightarrow \textit{world-model} \Rightarrow \textit{world-model}$
where
$\textit{execute-ast-action-inst act-inst } M = \textit{apply-effect } (\textit{effect act-inst}) \ M$

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

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

fun *ast-action-inst-path*
$:: \textit{world-model} \Rightarrow (\textit{ast-action-inst list}) \Rightarrow \textit{world-model} \Rightarrow \textit{bool}$
where
$\textit{ast-action-inst-path } M \ [] \ M' \longleftrightarrow (M = M')$
$| \ \textit{ast-action-inst-path } M \ (\alpha\#\alpha s) \ M' \longleftrightarrow M \Vdash \textit{precondition } \alpha$
$\wedge \ (\textit{ast-action-inst-path } (\textit{execute-ast-action-inst } \alpha \ M) \ \alpha s \ M')$

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

lemma *ast-action-inst-path-in-paper*:
$\textit{ast-action-inst-path } M \ [] \ M' \longleftrightarrow (M = M')$
$\textit{ast-action-inst-path } M \ (\alpha\#\alpha s) \ M' \longleftrightarrow M \Vdash \textit{precondition } \alpha$
$\wedge \ (\textit{ast-action-inst-path } (\textit{apply-effect } (\textit{effect } \alpha) \ M) \ \alpha s \ M')$
by (*auto simp*: *execute-ast-action-inst-def*)

end — Context of *ast-domain*

1.3.1 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 $\textit{state} = \textit{object atom valuation}$

context *ast-domain* **begin**

An action is a partial function from states to states.

type-synonym $\textit{action} = \textit{state} \rightharpoonup \textit{state}$

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

Definition B (i)–(iv) in [Lif87]

fun $\textit{sound-opr} :: \textit{ast-action-inst} \Rightarrow \textit{action} \Rightarrow \textit{bool}$ **where**
$\textit{sound-opr } (\textit{Action-Inst pre } (\textit{Effect add del})) \ f \longleftrightarrow$
$(\forall s. \ s \models \textit{pre} \longrightarrow$
$(\exists s'. \ f \ s = \textit{Some } s' \ \wedge$
$(\forall \textit{atm. is-Atom atm} \wedge \textit{atm} \notin \textit{set del} \wedge s \models \textit{atm} \longrightarrow s' \models \textit{atm})$
$\wedge \ (\forall \textit{fmla. fmla} \in \textit{set add} \longrightarrow s' \models \textit{fmla})$

$\wedge$ ($\forall$ *fmla*$\in$*set add*. $\neg$*is-Atom fmla* $\longrightarrow$ ($\forall$ *s*. *s* $\models$ *fmla*))
)
)

Definition B (v)–(vii) in [Lif87]

definition *sound-system*
:: *ast-action-inst set*
$\Rightarrow$ *world-model*
$\Rightarrow$ *object atom valuation*
$\Rightarrow$ (*ast-action-inst* $\Rightarrow$ *action*)
$\Rightarrow$ *bool*
where
sound-system Σ M_0 s_0 *f* $\longleftrightarrow$
($\forall$ *fmla*$\in M_0$. s_0 $\models$ *fmla*)
$\wedge$ ($\forall$ *fmla*$\in M_0$. $\neg$*is-Atom fmla* $\longrightarrow$ ($\forall$ *s*. *s* $\models$ *fmla*))
$\wedge$ ($\forall$ $\alpha\in\Sigma$. *sound-opr* α (*f* α))

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* | *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* **unfolding** *compose-actions-def* **by** *auto*

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

lemma *sound-opr-alt*:
sound-opr opr f =
($\forall$ *s*. *s* $\models$ (*precondition opr*) $\longrightarrow$ ($\exists$ *s'*. *f s* = (*Some s'*) $\wedge$
($\forall$ *atm*. *is-Atom atm* $\wedge$ *atm* $\notin$ *set*(*Dels* (*effect opr*)) $\wedge$ *s* $\models$ *atm* $\longrightarrow$ *s'* $\models$ *atm*)
$\wedge$ ($\forall$ *atm*. *atm* $\in$ *set*(*Adds* (*effect opr*)) $\longrightarrow$ *s'* $\models$ *atm*)
$\wedge$ ($\forall$ *s*.$\forall$ *fmla*$\in$*set*(*Adds* (*effect opr*)). $\neg$*is-Atom fmla* $\longrightarrow$ *s* $\models$ *fmla*)

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

Soundness Theorem of [Lif87].

theorem *STRIPS-sema-sound*:
assumes *sound-system* Σ M_0 s_0 *f*
— For a sound system Σ
assumes *set* $\alpha s \subseteq \Sigma$
— And a plan αs
assumes *ast-action-inst-path* M_0 αs 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* (*map f* αs) s_0 = *Some* s'
— The result world model M' is satisfied in state s'
and $\forall$*fmla*$\in M'$. $s' \models$ *fmla*
proof −
have $\exists s'$. *compose-actions* (*map f* αs) s_0 = *Some* s' $\wedge$ ($\forall$*fmla*$\in M'$. $s' \models$ *fmla*)

using *assms*
proof(*induction* αs *arbitrary*: s_0 M_0)
case *Nil*
then show *?case* **by** (*auto simp add*: *compose-action-def sound-system-def*)
next
case *ass*: (*Cons* α αs)
then obtain *pre add del* **where** *a*: α = *Action-Inst pre* (*Effect add del*)
using *ast-action-inst.exhaust ast-effect.exhaust* **by** *metis*
let *?*M_1 = *execute-ast-action-inst* α M_0
obtain s_1 **where** *s1*: (f α) s_0 = *Some* s_1
($\forall$ *atm*. *is-Atom atm* $\longrightarrow$ *atm* $\notin$ *set*(*Dels* (*effect* α))
$\longrightarrow s_0 \models$ *atm*
$\longrightarrow s_1 \models$ *atm*) $\wedge$
($\forall$*fmla*. *fmla* $\in$ *set*(*Adds* (*effect* α))
$\longrightarrow s_1 \models$ *fmla*)
using *ass*(*2*−*4*)
unfolding *sound-system-def sound-opr-alt*
by (*force simp*: *entailment-def*)

have ($\forall$*fmla*$\in$*?*M_1. $s_1 \models$ *fmla*)
apply(*rule ballI*)
using *ass*(*2*) *s1 formula.exhaust*
unfolding *sound-system-def execute-ast-action-inst-def*
by (*fastforce simp add*: *a image-def*)
moreover have ($\forall s.\forall$*fmla*$\in$*?*M_1. $\neg$*is-Atom fmla* $\longrightarrow s \models$ *fmla*)
using *ass*(*2*)
unfolding *sound-system-def execute-ast-action-inst-def*
using *a ass.prems*(*2*) *ass.prems*(*3*) *entailment-def list.set-intros*(*1*)

by *fastforce*
moreover have $(\forall\, opr{\in}\Sigma.\ sound\text{-}opr\ opr\ (f\ opr))$
using *ass(2)* **unfolding** *sound-system-def*
by (*auto simp add*:)
ultimately have $sound\text{-}system\ \Sigma\ ?M_1\ s_1\ f$
unfolding *sound-system-def*
by *auto*
from *ass.IH[OF this] ass.prems* **obtain** s' **where**
$compose\text{-}actions\ (map\ f\ \alpha s)\ s_1 = Some\ s' \wedge (\forall\, a{\in}M'.\ s' \models a)$
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*:
$sound\text{-}system\ \Sigma\ M_0\ s_0\ f \Longrightarrow set\ \alpha s \subseteq \Sigma$
$\Longrightarrow ast\text{-}action\text{-}inst\text{-}path\ M_0\ \alpha s\ M'$
$\Longrightarrow \exists\, s'.\ compose\text{-}actions\ (map\ f\ \alpha s)\ s_0 = Some\ s'$
$\wedge\ (\forall\, fmla{\in}M'.\ s' \models fmla)$
using *STRIPS-sema-sound* **by** *metis*

end — Context of *ast-domain*

1.4 Well-Formedness of PDDL

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* ⇀ *type list* **where**
$sig \equiv map\text{-}of\ (map\ (\lambda PredDecl\ p\ n \Rightarrow (p,n))\ (predicates\ D))$

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.

definition $subtype\text{-}rel \equiv \{''object''\} \times UNIV$

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

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

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

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

Checks whether an entity has a given type

```
definition is-of-type :: 'ent ⇒ type ⇒ bool where
  is-of-type v T ⟷ (
    case ty-ent v of
      Some vT ⇒ of-type vT T
    | None ⇒ False)
```

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

TODO: We could check that types may actually overlap

```
fun wf-atom :: 'ent atom ⇒ bool where
  wf-atom (predAtm p vs) ⟷ (
    case sig p of
      None ⇒ False
    | Some Ts ⇒ list-all2 is-of-type vs Ts)
| wf-atom (Eq - -) = False
```

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

```
fun wf-fmla :: ('ent atom) formula ⇒ bool where
  wf-fmla (Atom a) ⟷ wf-atom a
| wf-fmla (φ1 ∧ φ2) ⟷ (wf-fmla φ1 ∧ wf-fmla φ2)
| wf-fmla (φ1 ∨ φ2) ⟷ (wf-fmla φ1 ∧ wf-fmla φ2)
| wf-fmla (¬⊥) ⟷ True
| wf-fmla - ⟷ False

lemma wf-fmla-add-simps[simp]: wf-fmla (¬φ) ⟷ φ=⊥
  by (cases φ) auto
```

Special case for a well-formed atomic formula

```
fun wf-fmla-atom where
  wf-fmla-atom (Atom a) ⟷ wf-atom a
| wf-fmla-atom - ⟷ False

lemma wf-fmla-atom-alt: wf-fmla-atom φ ⟷ is-Atom φ ∧ wf-fmla φ
  by (cases φ) auto
```

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

```
fun wf-effect where
  wf-effect (Effect adds dels) ⟷
      (∀ ae∈set adds. wf-fmla-atom ae)
    ∧ (∀ de∈set dels.  wf-fmla-atom de)
```

end — Context fixing *ty-ent*

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

fun *wf-action-schema* :: *ast-action-schema* ⇒ *bool* **where**
wf-action-schema (*Action-Schema n params pre eff*) ⟷ (
let
tyv = *map-of params*
in
distinct (*map fst params*)
∧ *wf-fmla tyv pre*
∧ *wf-effect tyv eff*)

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

fun *wf-type* **where**
wf-type (*Either Ts*) ⟷ *set Ts* ⊆ *insert ''object''* (*set* (*types D*))

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

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

A domain is well-formed if

- there are no duplicate declared primitive types,
- 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 ≡
distinct (*types D*)
∧ *distinct* (*map* (*predicate-decl.pred*) (*predicates D*))
∧ (∀ *p*∈*set* (*predicates D*). *wf-predicate-decl p*)
∧ *distinct* (*map ast-action-schema.name* (*actions D*))
∧ (∀ *a*∈*set* (*actions D*). *wf-action-schema a*)

end — locale *ast-domain*

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

locale *ast-problem* = *ast-domain domain P*

for P :: *ast-problem*
begin

We refer to the problem domain as D

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

definition *objT* :: *object* $\rightharpoonup$ *type* **where**
objT $\equiv$ *map-of* (*objects P*)

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

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

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

definition *wf-problem* **where**
wf-problem $\equiv$
wf-domain
$\wedge$ *distinct* (*map fst* (*objects P*))
$\wedge$ ($\forall (n,T) \in$ *set* (*objects P*). *wf-type T*)
$\wedge$ *distinct* (*init P*)
$\wedge$ *wf-world-model* (*set* (*init P*))
$\wedge$ *wf-fmla objT* (*goal P*)

fun *wf-effect-inst* :: *object atom formula ast-effect* $\Rightarrow$ *bool* **where**
wf-effect-inst (*Effect* (*adds*) (*dels*))
$\longleftrightarrow$ ($\forall a \in$ *set adds* $\cup$ *set dels*. *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*

1.5 PDDL Semantics

context *ast-domain* **begin**

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

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

fun *instantiate-action-schema*
:: *ast-action-schema* ⇒ *object list* ⇒ *ast-action-inst*
where
instantiate-action-schema (*Action-Schema n params pre eff*) *args* = (*let*
psubst = (*the o* (*map-of* (*zip* (*map fst params*) *args*)));
pre-inst = (*map-formula o map-atom*) *psubst pre*;
eff-inst = (*map-ast-effect o map-formula o map-atom*) *psubst eff*
in
Action-Inst pre-inst eff-inst
)

end — Context of *ast-domain*

context *ast-problem* **begin**

Initial model

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

Resolve a plan action and instantiate the referenced action schema.

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

Check whether object has specified type

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

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

lemma *is-obj-of-type-alt*: *is-obj-of-type* = *is-of-type objT*

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

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

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

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

fun *wf-plan-action* :: *plan-action* ⇒ *bool* **where**
  wf-plan-action (*PAction n args*) = (
    case resolve-action-schema n of
      None ⇒ *False*
    | *Some a* ⇒
        (∗ *Objects are valid and match parameter types* ∗)
        action-params-match a args
        (∗ *Effect is valid* ∗)
      ∧ *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* ⇒ (*plan-action list*) ⇒ *world-model* ⇒ *bool*
where
  plan-action-path M πs M′ =
      ((∀ π ∈ *set πs*. *wf-plan-action* π)
    ∧ *ast-action-inst-path M* (*map resolve-instantiate πs*) *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* ⇒ *plan* ⇒ *bool* **where**
  valid-plan-from M πs = (∃ *M′*. *plan-action-path M πs M′* ∧ *M′* ⊫ (*goal P*))

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

definition *valid-plan* :: *plan* ⇒ *bool*
  where *valid-plan* ≡ *valid-plan-from I*

end — Context of *ast-problem*

1.6 Preservation of Well-Formedness

1.6.1 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-action-inst* :: *ast-action-inst* ⇒ *bool* **where**
  wf-action-inst (*Action-Inst pre eff*) ⟷ (
    wf-fmla objT pre
   ∧ *wf-effect objT eff*
)

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

We begin with some auxiliary lemmas before the actual theorem.

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

lemma (**in** *ast-domain*) *of-type-trans*[*trans*]:
  of-type T1 T2 ⟹ *of-type T2 T3* ⟹ *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*)

context
  fixes *Q f*
  assumes *INST*: *is-of-type Q x T* ⟹ *is-of-type objT* (*f x*) *T*
begin

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

  lemma *wf-inst-atom*:
   assumes *wf-atom Q a*

```
    shows wf-atom objT (map-atom f a)
  proof -
    have X1: list-all2 (is-of-type objT) (map f xs) Ts if
      list-all2 (is-of-type Q) xs Ts for xs Ts
      using that
      apply induction
      using INST
      by auto
    then show ?thesis
      using assms wf-inst-eq-aux
      by (cases a; auto split: option.splits)
  qed

  lemma wf-inst-formula-atom:
    assumes wf-fmla-atom Q a
    shows wf-fmla-atom objT ((map-formula o map-atom) f a)
    using assms wf-inst-atom
    by (cases a; auto)

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

  lemma wf-inst-formula:
    assumes wf-fmla Q φ
    shows wf-fmla objT ((map-formula o map-atom) f φ)
    using assms
    by (induction φ) (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-action-inst (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
show *?thesis*
using *assms*(*2*) *wf-inst-formula wf-inst-effect INST*
by (*simp add*: *Let-def*; *metis comp-apply*)
qed
end — Context of *ast-problem*

1.6.2 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* ⇒ *world-model* ⇒ *bool* **where**
plan-action-enabled π *M*
⟷ *wf-plan-action* π ∧ *M* ⊫ *precondition* (*resolve-instantiate* π)

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

definition *execute-plan-action* :: *plan-action* ⇒ *world-model* ⇒ *world-model*
where *execute-plan-action* π *M*
= (*apply-effect* (*effect* (*resolve-instantiate* π)) *M*)

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

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

lemma *plan-action-path-Cons*[*simp*]:
plan-action-path M (π#π*s*) *M'* ⟷
plan-action-enabled π *M*
∧ *plan-action-path* (*execute-plan-action* π *M*) π*s M'*
by (*auto*
simp: *plan-action-path-def execute-plan-action-def*
execute-ast-action-inst-def plan-action-enabled-def)

end — Context of *ast-problem*

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

The initial world model is well-formed

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

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

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

Execution of plan actions preserves well-formedness

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

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

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

  from ‹*resolve-action-schema name* = *Some a*› **have**
    $a \in$ *set* (*actions D*)
    unfolding *resolve-action-schema-def* **by** *auto*
  with *wf-domain* **have** *wf-action-schema a*
    unfolding *wf-domain-def* **by** *auto*
  hence *wf-action-inst* (*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-ast-action-inst-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* $\Longrightarrow$ *wf-world-model s*
$\Longrightarrow$ *wf-world-model* (*execute-plan-action* π *s*)
by (*rule wf-execute*)

Execution of a plan preserves well-formedness

corollary *wf-plan-action-path*:
⟦*wf-world-model M*; *plan-action-path M* πs *M'*⟧ $\Longrightarrow$ *wf-world-model M'*
by (*induction* πs *arbitrary*: *M*) (*auto intro*: *wf-execute*)

end — Context of *wf-ast-problem*

1.7 Soundness Theorem for PDDL

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

Mapping world models to states

definition *state-to-wm s* = (*formula.Atom* ‘ {*atm. s atm*})
definition *wm-to-state M* = (%*atm.* (*formula.Atom atm*) $\in$ *M*)

Mapping AST action instances to actions

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

lemma *wm-to-state-to-wm*:
s $\models$ *f* = *wm-to-state* (*state-to-wm s*) $\models$ *f*
by (*auto simp*: *wm-to-state-def*
state-to-wm-def image-def)

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

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

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

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

lemma *wf-fmla-atm-is-atom*: *wf-fmla-atom objT f* $\Longrightarrow$ *is-Atom f*
  by (*cases f*) *auto*

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

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*)
  using *assms*
proof(*induction g-opr*)
  case *ass1*: (*Action-Inst x1a x2a*)
  then show *?case*
    unfolding *sound-opr-alt*
    apply *safe*
  proof−
    fix *s*
    assume *ass2*:
      wf-effect-inst (*effect* (*Action-Inst x1a x2a*))
      s $\models$ *precondition* (*Action-Inst x1a x2a*)
    let *?s* =
      wm-to-state(*apply-effect x2a* (*state-to-wm s*))
    have *pddl-opr-to-act* (*Action-Inst x1a x2a*) *s* = *Some ?s*
      using *ass1 ass2*
      apply (*cases x2a*; *simp*)
      apply (*cases x1a*; *simp*)
      by (*auto*

simp: *pddl-opr-to-act-def image-def Let-def*
simp: *state-to-wm-def entailment-def wf-fmla-atom-alt*)
moreover have
del-atm $\notin$ *set* (*Dels* (*effect* (*Action-Inst x1a x2a*)))
$\longrightarrow$ *is-Atom del-atm* $\longrightarrow$ *s* $\models$ *del-atm* $\longrightarrow$ *?s* $\models$ *del-atm*
for *del-atm*
using *atom-in-wm-2 not-dels-preserved atom-in-wm*
by (*metis ast-action-inst.sel*(*2*) *is-Atom.elims*(*2*)
ast-effect.collapse)
moreover have
add-atm $\in$ *set* (*Adds* (*effect* (*Action-Inst x1a x2a*)))
$\longrightarrow$ *?s* $\models$ *add-atm* **for** *add-atm*
using *wf-act-adds-are-atoms atom-in-wm-2*
and *adds-satisfied ass2*(*1*)
by (*metis ast-action-inst.sel*(*2*) *ast-effect.collapse*
is-Atom.elims(*2*))
moreover have
($\forall$ *s*. $\forall$ *fmla*$\in$*set* (*Adds* (*effect* (*Action-Inst x1a x2a*))).
$\neg$*is-Atom fmla* $\longrightarrow$ *s* $\models$ *fmla*)
using *wf-act-adds-are-atoms ass2*(*1*)
by *fastforce*
ultimately show $\exists$ *s'*. *pddl-opr-to-act* (*Action-Inst x1a x2a*) *s* = *Some s'*
$\wedge$ ($\forall$ *del-atm*.
is-Atom del-atm
$\wedge$ *del-atm* $\notin$ *set* (*Dels* (*effect* (*Action-Inst x1a x2a*)))
$\wedge$ *s* $\models$ *del-atm*
$\longrightarrow$ *s'* $\models$ *del-atm*)
$\wedge$ ($\forall$ *add-atm*.
add-atm $\in$ *set* (*Adds* (*effect* (*Action-Inst x1a x2a*)))
$\longrightarrow$ *s'* $\models$ *add-atm*)
$\wedge$ ($\forall$ *s*. $\forall$ *fmla*$\in$*set* (*Adds* (*effect* (*Action-Inst x1a x2a*))).
$\neg$*is-Atom fmla* $\longrightarrow$ *s* $\models$ *fmla*)
by *blast*
qed
qed

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

lemma *wf-pddl-ground-act-is-sound-opr*:
assumes *wf-action-inst 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 wf-instantiate-action-schema[*OF assms*]]
    by *blast*

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

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

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

lemma *wm-to-state-works-for-I*:
  assumes $x \in I$
  shows *wm-to-state* $I \models x$
  using *wf-world-model-has-atoms assms wf-problem*
  unfolding *wf-problem-def I-def*
  apply (*cases* x; *auto simp add*: *wf-problem-def*)
  using *assms atom-in-wm-2* **apply** (*auto simp*: *wm-to-state-def*)
  by *force+*

theorem *wf-plan-sound-system*:
  assumes $\forall \pi \in$ *set* πs. *wf-plan-action* π
  shows *sound-system*
    (*set* (*map resolve-instantiate* πs))
    I
    (*wm-to-state* I)
    pddl-opr-to-act
  using *wm-to-state-works-for-I wf-problem wf-world-model-has-atoms*
  unfolding *sound-system-def wf-problem-def I-def*
  apply *auto*
  using *wf-plan-act-is-sound' assms* **by** *blast*

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

1.8 Closed-World Assumption and Negation

A valuation extracted from the atoms of the world model

definition *valuation* :: *world-model* $\Rightarrow$ *object atom* $\Rightarrow$ *bool*
where *valuation M* $\equiv \lambda x$. (*Atom x* $\in$ *M*)

Augment a world model by adding negated versions of all atoms not contained in it.

definition *close-world M* = $M \cup \{\neg(\textit{Atom atm}) \mid \textit{atm. Atom atm} \notin M\}$

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

lemma *in-close-world-conv*:
$\varphi \in$ *close-world M* $\longleftrightarrow (\varphi{\in}M \vee (\exists\, atm.\ \varphi{=}\neg(\textit{Atom atm}) \wedge \textit{Atom atm}{\notin}M))$
by (*auto simp*: *close-world-def*)

lemma *valuation-aux-1*:
fixes *M* :: *world-model* **and** φ :: *object atom formula*
defines $C \equiv$ *close-world M*
assumes *A*: $\forall\, \varphi{\in}C.\ \mathcal{A} \models \varphi$
shows $\mathcal{A}$ = *valuation M*
using *A* **unfolding** *C-def*
by (*auto simp*: *in-close-world-conv valuation-def Ball-def intro*!: *ext*)

lemma *valuation-aux-2*:
assumes $\forall\, \varphi{\in}M$. *is-Atom* φ
shows ($\forall\, G{\in}$*close-world M*. *valuation M* $\models G$)
using *assms*
by (*force simp*: *in-close-world-conv valuation-def elim*: *is-Atom.elims*)

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

lemma *close-world-imp-val*:
  $\forall \varphi \in M.$ *is-Atom* $\varphi \Longrightarrow$ *close-world* $M \Vdash\!\models \varphi \Longrightarrow$ *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 $\forall \varphi \in M.$ *is-Atom* φ
  shows *valuation* $M \models \varphi \longleftrightarrow$ *close-world* $M \Vdash\!\models \varphi$
  using *assms val-imp-close-world close-world-imp-val* **by** *blast*

end — Theory

2 Executable PDDL Checker

theory *PDDL-STRIPS-Checker*
imports
  PDDL-STRIPS-Semantics

  Error-Monad-Add

  ~~/src/HOL/Library/Char-ord
  ~~/src/HOL/Library/Code-Char
  ~~/src/HOL/Library/Code-Target-Nat

  Containers.Containers
begin

2.1 Implementation Refinements

2.1.1 Of-Type

We exploit the flat type hierarchy to efficiently implement the subtype-check

context *ast-domain* **begin**

lemma *rtrancl-subtype-rel-alt*: *subtype-rel** = ({''*object*''} × *UNIV*)$^=$
  unfolding *subtype-rel-def*
  apply (*auto*)
  apply (*meson SigmaD1 converse-rtranclE singleton-iff*)

```
  done

lemma of-type-code:
  of-type oT T ⟷ (
      ''object'' ∈ set (primitives T))
    ∨ set (primitives oT) ⊆ set (primitives T)
  unfolding of-type-def rtrancl-subtype-rel-alt
  by auto

end — Context of ast-domain
```

2.1.2 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 atom formula ast-effect ⇒ world-model ⇒ world-model
where
  apply-effect-exec (Effect adds dels) s
    = fold (λadd s. Set.insert add s) adds
        (fold (λdel s. Set.remove del s) dels s)

lemma apply-effect-exec-refine[simp]:
  apply-effect-exec (Effect (adds) (dels)) s
  = apply-effect (Effect (adds) (dels)) s
proof(induction adds arbitrary: s)
  case Nil
  then show ?case
  proof(induction dels arbitrary: s)
    case Nil
    then show ?case by auto
  next
    case (Cons a dels)
    then show ?case
      by (auto simp add: image-def)
  qed
next
  case (Cons a adds)
  then show ?case
  proof(induction dels arbitrary: s)
    case Nil
    then show ?case by (auto; metis Set.insert-def sup-assoc insert-iff)
  next
    case (Cons a dels)
    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*

2.1.3 Well-Foundedness

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* (*objects P*)

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

We refine the typecheck to use the mapping

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

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

We refine the well-formedness checks to use the mapping

definition *wf-fact′* :: *objT* ⇒ *fact* ⇒ *bool*
  where
  wf-fact′ ot ≡ *wf-atom* (*Mapping.lookup ot*)

lemma *wf-fact′-correct*[*simp*]: *wf-fact′ mp-objT* = *wf-fact*
  by (*auto simp*: *wf-fact′-def wf-fact-def*)

definition *wf-fmla-atom′ mp f*
  = (*case f of formula.Atom atm* ⇒ (*wf-fact′ mp atm*) | - ⇒ *False*)

lemma *wf-problem-impl-eq*:
  wf-problem ⟷ (*let mp* = *mp-objT in*
    wf-domain
  ∧ *distinct* (*map fst* (*objects P*))
  ∧ (∀(*n*,*T*)∈*set* (*objects P*). *wf-type T*)

$\wedge$ *distinct* (*init P*)
$\wedge$ ($\forall f \in$*set* (*init P*). *wf-fmla-atom′ mp f*)
$\wedge$ *wf-fmla* (*Mapping.lookup mp*) (*goal P*))

unfolding *wf-problem-def wf-fmla-atom′-def wf-world-model-def*
and *wf-fmla-atom-alt*
apply (*simp*; *safe*)
subgoal by (*auto simp*: *wf-fact-def split*: *formula.split*)
subgoal for φ **by** (*cases* φ) *auto*
subgoal for φ
  apply (*clarsimp simp*: *wf-fact-def split*: *formula.splits*)
  by (*cases* φ) *auto*
done

Instantiating actions will yield well-founded effects. Corollary of ⟦*action-params-match ?a ?args*; *wf-action-schema ?a*⟧ ⟹ *wf-action-inst* (*instantiate-action-schema ?a ?args*).

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

end — Context of *ast-problem*

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

Resolving an action yields a well-founded action schema.

lemma *resolve-action-wf*:
  assumes *resolve-action-schema n* = *Some a*
  shows *wf-action-schema a*
proof −
  from *wf-domain* **have**
    X1: *distinct* (*map ast-action-schema.name* (*actions D*))
    and *X2*: ∀ *a*∈*set* (*actions D*). *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*

2.1.4 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* ≡
    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* ≡ (*valuation M*) ⊨ *F*

Justification of this function

  lemma *holds-for-wf-fmlas*:
    assumes ∀ *x*∈*s*. *is-Atom x wf-fmla Q F*
    shows *holds s F* ⟷ *s* ⊫ *F*
    unfolding *holds-def entailment-def valuation-def*
    using *assms*
  proof (*induction F*)
    case (*Atom x*)
    then show *?case*
      apply *auto*
      by (*metis formula-semantics.simps*(*1*) *is-Atom.elims*(*2*) *valuation-def*)
  next
    case (*Or F1 F2*)
    then show *?case*
      apply *simp* **apply** (*safe*; *clarsimp?*)
      by (*metis* (*mono-tags*) *is-Atom.elims*(*2*) *formula-semantics.simps*(*1*))
  qed *auto*

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* ⇒ *world-model* ⇒ *-+world-model* **where**
    en-exE ≡ λ(*PAction n args*) ⇒ λ*s*. *do* {
      a ← *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 ∀x∈s. is-Atom x
  shows en-exE a s = Inr s'
    ⟷ plan-action-enabled a s ∧ 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-ast-action-inst-def en-exE-def wf-domain-def
  apply (clarsimp
      split: option.splits
      simp: resolve-action-schemaE-def return-iff)
  by (metis ast-action-inst.collapse holds-for-wf-fmlas resolve-action-wf
          wf-action-inst.simps wf-instantiate-action-schema)
```

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

definition en-exE2
  :: (object, type) mapping ⇒ plan-action ⇒ world-model ⇒ -+world-model
where
  en-exE2 mp ≡ λ(PAction n args) ⇒ λs. do {
    a ← resolve-action-schemaE n;
    check (action-params-match2 mp a args) (ERRS ''Parameter mismatch'');
    let ai = instantiate-action-schema a args;
    check (holds s (precondition ai)) (ERRS ''Precondition not satisfied'');
    Error-Monad.return (apply-effect (effect ai) s)
  }
```

Justification of refinement

```
lemma (in wf-ast-problem) wf-en-exE2-eq:
  shows en-exE2 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 ∀x∈s. is-Atom x
  shows en-exE2 mp-objT a s = Inr s'
    ⟷ plan-action-enabled a s ∧ s' = execute-plan-action a s
```

```
  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:
  ⟦∀x∈s. is-Atom x⟧ ⟹
    en-exE2 mp-objT a s = Inr s'
    ⟷ plan-action-enabled a s ∧ s' = execute-plan-action a s
  using en-exE2-return-iff .
```

end — Context of *ast-problem*

2.1.5 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 ⇒ plan ⇒ bool where
  valid-plan-from1 s [] ⟷ s ⫢ (goal P)
| valid-plan-from1 s (π#πs)
    ⟷ plan-action-enabled π s
      ∧ (valid-plan-from1 (execute-plan-action π s) πs)

lemma valid-plan-from1-refine: valid-plan-from s πs = valid-plan-from1 s πs
proof(induction πs arbitrary: s)
  case Nil
  then show ?case by (auto simp add: plan-action-path-def valid-plan-from-def)
next
  case (Cons a πs)
  then show ?case
    by (auto
      simp: valid-plan-from-def plan-action-path-def plan-action-enabled-def
      simp: execute-ast-action-inst-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
  :: (object, type) mapping ⇒ nat ⇒ world-model ⇒ plan ⇒ -+unit
where
  valid-plan-fromE mp si s []
    = check (holds s (goal P)) (ERRS ''Postcondition does not hold'')
| valid-plan-fromE mp si s (π#πs) = do {
      s ← en-exE2 mp π s
        <+? (λe -. shows ''at step '' o shows si o shows '': '' o e ());
      valid-plan-fromE mp (si+1) s πs
    }
```

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

lemma (**in** *wf-ast-problem*) *wf-actions-only-add-atoms*:
  $\llbracket\ \forall\, x{\in}s.\ \textit{is-Atom}\ x;\ \textit{wf-plan-action}\ a\ \rrbracket$
    $\Longrightarrow \forall\, x{\in}\textit{execute-plan-action}\ a\ s.\ \textit{is-Atom}\ x$
  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*
    simp: *execute-ast-action-inst-def*)
  subgoal for *n args schema fmla*
    apply (*cases effect* (*instantiate-action-schema schema args*); *simp*)
    by (*metis ast-action-inst.sel*(*2*) *ast-domain.wf-effect.simps*
        ast-domain.wf-fmla-atom-alt resolve-action-wf
        wf-action-inst.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 $\forall\, x{\in}s.\ \textit{is-Atom}\ x$
  shows *valid-plan-fromE mp-objT k s* πs = *Inr* () $\longleftrightarrow$ *valid-plan-from s* πs
  using *assms* **unfolding** *valid-plan-from1-refine*
proof (*induction mp*≡*mp-objT k s* πs *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* π πs)
  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*

2.2 Executable Plan Checker

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

definition *check-plan P πs* ≡ *do* {
  check (*ast-problem.wf-problem P*) (*ERRS ''Domain/Problem not well−formed''*);
  ast-problem.valid-plan-fromE P (*ast-problem.mp-objT P*) *1* (*ast-problem.I P*) *πs*
}

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* ()
  ⟷ *ast-problem.wf-problem P* ∧ *ast-problem.valid-plan P πs*
proof −
  interpret *ast-problem P* **.**
  show *?thesis*
    unfolding *check-plan-def*
    apply (*auto*
      simp: *return-iff wf-world-model-def wf-fmla-atom-alt I-def wf-problem-def*)
    apply (*metis ast-domain.wf-fmla-atom-alt ast-problem.I-def*
              ast-problem.valid-plan-def valid-plan-fromE-return-iff'
              wf-fmla-atom.elims(*2*) *wf-problem-def wf-world-model-def*)
    by (*metis* (*full-types*) *ast-domain.wf-fmla-atom-alt ast-problem.I-def*
            ast-problem.valid-plan-def ast-problem.valid-plan-fromE-return-iff'
            wf-fmla-atom.elims(*2*) *wf-problem-def wf-world-model-def*)
qed

2.3 Code Setup

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

2.3.1 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-type.simps
  ast-domain.wf-predicate-decl.simps
  ast-domain.wf-domain-def
  ast-domain.wf-action-schema.simps
  ast-domain.wf-effect.simps
  ast-domain.wf-fmla.simps
  ast-domain.wf-atom.simps
  ast-domain.is-of-type-def

ast-domain.of-type-code

declare *wf-domain-code*[*code*]

lemmas *wf-problem-code* =
ast-problem.wf-problem-impl-eq
ast-problem.wf-fact′-def

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

ast-problem.wf-fact-def
ast-problem.wf-plan-action.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-domain.wf-fmla-atom.simps
ast-problem.wf-fmla-atom′-def
valuation-def
declare *check-code*[*code*]

2.3.2 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*

2.3.3 More Efficient Distinctness Check for Linorders

fun *no-stutter* :: ′*a list* ⇒ *bool* **where**

$\quad$*no-stutter* [] = *True*
| *no-stutter* [-] = *True*
| *no-stutter* $(a\#b\#l)$ = $(a{\neq}b \wedge$ *no-stutter* $(b\#l))$

lemma *sorted-no-stutter-eq-distinct*: *sorted* $l \Longrightarrow$ *no-stutter* $l \longleftrightarrow$ *distinct* l
  apply (*induction l rule*: *no-stutter.induct*)
  apply (*auto simp*: *sorted-Cons*)
  done

definition *distinct-ds* :: *'a::linorder list* $\Rightarrow$ *bool*
  where *distinct-ds* $l \equiv$ *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

2.3.4 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
  in *SML*
  module-name *PDDL-Checker-Exported*
  file *code/PDDL-STRIPS-Checker-Exported.sml*

end — Theory

3 Reasoning about Invariants

theory *invariant-verification*
  imports *PDDL-STRIPS-Semantics*
begin

context *ast-domain* **begin**
  definition *is-invariant-inst* $Q\ \alpha \longleftrightarrow$
    $(\forall M.\ Q\ M \wedge M \models$ *precondition* α
    $\longrightarrow Q$ (*execute-ast-action-inst* $\alpha\ M$))

end

context *ast-problem* **begin**

An invariant is a predicate preserved under execution of plan actions

definition *is-invariant-P* $Q \longleftrightarrow$
$(\forall M\ \pi.\ Q\ M \wedge$ *plan-action-enabled* $\pi\ M$
$\longrightarrow Q$ (*execute-plan-action* $\pi\ M$))

This also implies invariance under plans.

lemma *invarP-imp-plan-invar*:
assumes *I*: *is-invariant-P* Q
assumes $Q\ M$ *plan-action-path* $M\ \pi s\ M'$
shows $Q\ M'$
using *assms*(*2*,*3*)
apply (*induction* πs *arbitrary*: M)
using *I* **unfolding** *is-invariant-P-def* **by** *auto*

To prove that Q is invariant, we can show that it is preserved by every possible instantiation of the action schemas declared by the domain.

lemma *is-invariant-PI*:
assumes $\bigwedge a$ *args*.
$[\![a \in$ *set* (*actions* D); *action-params-match* a *args* $]\!]$
$\Longrightarrow$ *is-invariant-inst* Q (*instantiate-action-schema* a *args*)
shows *is-invariant-P* Q
unfolding *is-invariant-P-def*
proof *safe*
fix $M\ \pi$
assume *I0*: $Q\ M$ **and** *EN*: *plan-action-enabled* $\pi\ M$

obtain n *args* **where** [*simp*]: $\pi =$ (*PAction* n *args*)
by (*cases* π)

from *EN* **obtain** a **where**
X1: $a \in$ *set* (*actions* D) *action-params-match* a *args*
and *X2*: $M \models$ *precondition* (*instantiate-action-schema* a *args*)
and [*simp*]: *resolve-action-schema* $n =$ *Some* a
by (*auto*
simp: *plan-action-enabled-def resolve-action-schema-def*
split: *option.splits*
dest: *index-by-eq-SomeD*)

show Q (*execute-plan-action* $\pi\ M$)
using *I0 X1 X2 assms execute-ast-action-inst-def*
execute-plan-action-def is-invariant-inst-def
by *auto*
qed

end

context *ast-domain* **begin**

In the context of a domain, an invariant must be preserved by any action of any well-formed problem in this domain.

definition *is-invariant Q* $\longleftrightarrow$
($\forall$ *P*. *ast-problem.wf-problem P*
$\longrightarrow$ *ast-problem.is-invariant-P P Q*)

An invariant can be introduced by showing that it preserves all possible action instances of all possible problems.

lemma *is-invariant-I*:
assumes $\bigwedge$*a args P*.
$[\![$ *ast-problem.wf-problem P*; *a* $\in$ *set* (*actions* (*domain P*));
ast-problem.action-params-match P a args $]\!]$
$\Longrightarrow$ *is-invariant-inst Q* (*instantiate-action-schema a args*)
shows *is-invariant Q*
unfolding *is-invariant-def*
apply *safe*
apply (*rule ast-problem.is-invariant-PI*)
by (*rule assms*)
end

An invariant is preserved by any path in any well-formed problem

lemma (**in** *wf-ast-problem*) *invar-imp-plan-invar*:
assumes *is-invariant Q*
assumes *Q M plan-action-path M* π*s M'*
shows *Q M'*
by (*metis assms ast-domain.is-invariant-def invarP-imp-plan-invar wf-problem*)

end