Semantics of AI Planning Languages

Mohammad Abdulaziz and Peter Lammich*

This is an Isabelle/HOL formalisation of the semantics of the multi-valued planning tasks language that is used by the planning system Fast-Downward [3], the STRIPS [2] fragment of the Planning Domain Definition Language [5] (PDDL), and the STRIPS soundness meta-theory developed by Lifschitz [4]. It also contains formally verified checkers for checking the well-formedness of problems specified in either language as well the correctness of potential solutions. The formalisation in this entry was described in an earlier publication [1].

Contents

*Author names are alphabetically ordered.

theory *SASP-Semantics*
imports *Main*
begin

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

1.1 Syntax

type-synonym *name* = *string*
type-synonym *ast-variable* = *name* × *nat option* × *name list*
type-synonym *ast-variable-section* = *ast-variable list*
type-synonym *ast-initial-state* = *nat list*
type-synonym *ast-goal* = (*nat* × *nat*) *list*
type-synonym *ast-precond* = (*nat* × *nat*)
type-synonym *ast-effect* = *ast-precond list* × *nat* × *nat option* × *nat*
type-synonym *ast-operator* = *name* × *ast-precond list* × *ast-effect list* × *nat*
type-synonym *ast-operator-section* = *ast-operator list*

type-synonym *ast-problem* =
ast-variable-section × *ast-initial-state* × *ast-goal* × *ast-operator-section*

type-synonym *plan* = *name list*

1.1.1 Well-Formedness

locale *ast-problem* =
fixes *problem* :: *ast-problem*
begin
definition *astDom* :: *ast-variable-section*
where *astDom* ≡ *case problem of* (*D*,*I*,*G*,*δ*) ⇒ *D*
definition *astI* :: *ast-initial-state*
where *astI* ≡ *case problem of* (*D*,*I*,*G*,*δ*) ⇒ *I*
definition *astG* :: *ast-goal*
where *astG* ≡ *case problem of* (*D*,*I*,*G*,*δ*) ⇒ *G*
definition *astδ* :: *ast-operator-section*
where *astδ* ≡ *case problem of* (*D*,*I*,*G*,*δ*) ⇒ *δ*

definition *numVars* ≡ *length astDom*
definition *numVals x* ≡ *length* (*snd* (*snd* (*astDom*!*x*)))

definition *wf-partial-state ps* ≡
distinct (*map fst ps*)
∧ (∀ (*x*,*v*) ∈ *set ps*. *x* < *numVars* ∧ *v* < *numVals x*)

definition *wf-operator* :: *ast-operator* ⇒ *bool*
where *wf-operator* ≡ λ(*name*, *pres*, *effs*, *cost*).
wf-partial-state pres
∧ *distinct* (*map* (λ(-, *v*, -, -). *v*) *effs*) — This may be too restrictive

```
  ∧ (∀(epres,x,vp,v)∈set effs.
      wf-partial-state epres
    ∧ x < numVars ∧ v < numVals x
    ∧ (case vp of None ⇒ True | Some v ⇒ v<numVals x)
    )


definition well-formed ≡
  — Initial state
  length astI = numVars
∧ (∀x<numVars. astI!x < numVals x)

  — Goal
∧ wf-partial-state astG

— Operators
∧ (distinct (map fst astδ))
∧ (∀π∈set astδ. wf-operator π)


end

locale wf-ast-problem = ast-problem +
  assumes wf: well-formed
begin
  lemma wf-initial:
    length astI = numVars
    ∀x<numVars. astI!x < numVals x
    ⟨proof⟩

  lemma wf-goal: wf-partial-state astG
    ⟨proof⟩

  lemma wf-operators:
    distinct (map fst astδ)
    ∀π∈set astδ. wf-operator π
    ⟨proof⟩
end
```

1.2 Semantics as Transition System

```
type-synonym state = nat ⇀ nat
type-synonym pstate = nat ⇀ nat


context ast-problem
begin

  definition Dom :: nat set where Dom = {0..<numVars}
```

definition *range-of-var* **where** *range-of-var x* $\equiv$ {*0..<numVals x*}

definition *valid-states* :: *state set* **where** *valid-states* $\equiv$ {
  s. dom s = Dom $\wedge$ ($\forall$ *x*$\in$*Dom. the* (*s x*) $\in$ *range-of-var x*)
}

definition *I* :: *state*
  where *I v* $\equiv$ *if v<length astI then Some* (*astI!v*) *else None*

definition *subsuming-states* :: *pstate* $\Rightarrow$ *state set*
  where *subsuming-states partial* $\equiv$ { *s*$\in$*valid-states. partial* $\subseteq_m$ *s* }

definition *G* :: *state set*
  where *G* $\equiv$ *subsuming-states* (*map-of astG*)

end

definition *implicit-pres* :: *ast-effect list* $\Rightarrow$ *ast-precond list* **where**
  implicit-pres effs $\equiv$
  map (λ(-,*v*,*vpre*,-). (*v*,*the vpre*))
    (*filter* (λ(-,-,*vpre*,-). *vpre*$\neq$*None*) *effs*)

context *ast-problem*
begin

definition *lookup-operator* :: *name* $\Rightarrow$ *ast-operator option* **where**
  lookup-operator name $\equiv$ *find* (λ(*n*,-,-,-). *n*=*name*) *ast*δ

definition *enabled* :: *name* $\Rightarrow$ *state* $\Rightarrow$ *bool*
  where *enabled name s* $\equiv$
    case lookup-operator name of
      Some (-,*pres*,*effs*,-) $\Rightarrow$
          s$\in$*subsuming-states* (*map-of pres*)
        $\wedge$ *s*$\in$*subsuming-states* (*map-of* (*implicit-pres effs*))
    | *None* $\Rightarrow$ *False*

definition *eff-enabled* :: *state* $\Rightarrow$ *ast-effect* $\Rightarrow$ *bool* **where**
  eff-enabled s $\equiv$ λ(*pres*,-,-,-). *s*$\in$*subsuming-states* (*map-of pres*)

definition *execute* :: *name* $\Rightarrow$ *state* $\Rightarrow$ *state* **where**
  execute name s $\equiv$
    case lookup-operator name of
      Some (-,-,*effs*,-) $\Rightarrow$
        s ++ *map-of* (*map* (λ(-,*x*,-,*v*). (*x*,*v*)) (*filter* (*eff-enabled s*) *effs*))
    | *None* $\Rightarrow$ *undefined*

fun *path-to* **where**
  path-to s [] *s'* $\longleftrightarrow$ *s'*=*s*

| *path-to* s $(\pi\#\pi s)$ s' $\longleftrightarrow$ *enabled* π s $\wedge$ *path-to* (*execute* π s) πs s'

definition *valid-plan* :: *plan* $\Rightarrow$ *bool*
where *valid-plan* $\pi s \equiv \exists\, s' \in G$. *path-to* I πs s'

end

1.2.1 Preservation of well-formedness

context *wf-ast-problem*
begin
lemma *I-valid*: $I \in$ *valid-states*
⟨*proof*⟩

lemma *lookup-operator-wf*:
assumes *lookup-operator name* = *Some* π
shows *wf-operator* π *fst* π = *name*
⟨*proof*⟩

lemma *execute-preserves-valid*:
assumes $s \in$ *valid-states*
assumes *enabled name* s
shows *execute name* $s \in$ *valid-states*
⟨*proof*⟩

lemma *path-to-pres-valid*:
assumes $s \in$ *valid-states*
assumes *path-to* s πs s'
shows $s' \in$ *valid-states*
⟨*proof*⟩

end

end
theory *SASP-Checker*
imports *SASP-Semantics*
HOL−Library.Code-Target-Nat
begin

2 An Executable Checker for Multi-Valued Planning Problem Solutions

2.1 Auxiliary Lemmas

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

shows *map-of* $l \subseteq_m m$
⟨*proof*⟩

lemma [*simp*]: $fst \circ (\lambda(a, b, c, d). (f\ a\ b\ c\ d, g\ a\ b\ c\ d)) = (\lambda(a,b,c,d). f\ a\ b\ c\ d)$
⟨*proof*⟩

lemma *map-mp*: $m \subseteq_m m' \Longrightarrow m\ k = Some\ v \Longrightarrow m'\ k = Some\ v$
⟨*proof*⟩

lemma *map-add-map-of-fold*:
fixes *ps* **and** $m :: 'a \rightharpoonup 'b$
assumes *distinct* (*map fst ps*)
shows m ++ *map-of ps* = *fold* $(\lambda(k, v)\ m.\ m(k \mapsto v))$ *ps m*
⟨*proof*⟩

2.2 Well-formedness Check

lemmas *wf-code-thms* =
ast-problem.astDom-def ast-problem.astI-def ast-problem.astG-def ast-problem.astδ*-def*
ast-problem.numVars-def ast-problem.numVals-def
ast-problem.wf-partial-state-def ast-problem.wf-operator-def ast-problem.well-formed-def

declare *wf-code-thms*[*code*]

export-code *ast-problem.well-formed* **in** *SML*

2.3 Execution

definition *match-pre* :: *ast-precond* ⇒ *state* ⇒ *bool* **where**
match-pre ≡ $\lambda(x,v)\ s.\ s\ x = Some\ v$

definition *match-pres* :: *ast-precond list* ⇒ *state* ⇒ *bool* **where**
match-pres pres s ≡ ∀ *pre*∈*set pres. match-pre pre s*

definition *match-implicit-pres* :: *ast-effect list* ⇒ *state* ⇒ *bool* **where**
match-implicit-pres effs s ≡ ∀ (-,*x*,*vp*,-)∈*set effs.*
(*case vp of None* ⇒ *True* | *Some v* ⇒ *s x* = *Some v*)

definition *enabled-opr'* :: *ast-operator* ⇒ *state* ⇒ *bool* **where**
enabled-opr' ≡ λ(*name*,*pres*,*effs*,*cost*) *s. match-pres pres s* ∧ *match-implicit-pres effs s*

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

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

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

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

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

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

  lemma *match-pres-correct*:
    assumes *D*: *distinct* (*map fst pres*)
    assumes *s*∈*valid-states*
    shows *match-pres pres s* ⟷ *s*∈*subsuming-states* (*map-of pres*)
  ⟨*proof*⟩

  lemma *match-implicit-pres-correct*:
    assumes *D*: *distinct* (*map* (λ(-, *v*, -, -). *v*) *effs*)
    assumes *s*∈*valid-states*
  shows *match-implicit-pres effs s* ⟷ *s*∈*subsuming-states* (*map-of* (*implicit-pres effs*))
  ⟨*proof*⟩

  lemma *enabled-opr′-correct*:
    assumes *V*: *s*∈*valid-states*
    assumes *lookup-operator name* = *Some* π
    shows *enabled-opr′* π *s* ⟷ *enabled name s*
    ⟨*proof*⟩

  lemma *eff-enabled′-correct*:
    assumes *V*: *s*∈*valid-states*
    assumes *case eff of* (*pres*,-,-,-) ⇒ *wf-partial-state pres*
    shows *eff-enabled′ s eff* ⟷ *eff-enabled s eff*
    ⟨*proof*⟩

  lemma *execute-opr′-correct*:
    assumes *V*: *s*∈*valid-states*
    assumes *LO*: *lookup-operator name* = *Some* π

shows *execute-opr'* π s = *execute name s*
⟨*proof*⟩

lemma *lookup-operator'-correct*:
lookup-operator' problem name = *lookup-operator name*
⟨*proof*⟩

lemma *enabled'-correct*:
assumes V: $s \in$ *valid-states*
shows *enabled' problem name s* = *enabled name s*
⟨*proof*⟩

lemma *execute'-correct*:
assumes V: $s \in$ *valid-states*
assumes *enabled name s*
shows *execute' problem name s* = *execute name s*
⟨*proof*⟩

end

context *ast-problem*
begin

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

lemma *simulate-plan-correct*: *simulate-plan* πs s = *Some* s' ⟷ *path-to* s πs s'
⟨*proof*⟩

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

lemma *check-plan-correct*: *check-plan* πs ⟷ *valid-plan* πs
⟨*proof*⟩

```
end

fun simulate-plan' :: ast-problem ⇒ plan ⇒ state ⇀ state where
  simulate-plan' problem [] s = Some s
| simulate-plan' problem (π#πs) s = (
    if enabled' problem π s then
      let s = execute' problem π s in
      simulate-plan' problem πs s
    else
      None
  )
```

Avoiding duplicate lookup.

```
lemma simulate-plan'-code[code]:
  simulate-plan' problem [] s = Some s
  simulate-plan' problem (π#πs) s = (
    case lookup-operator' problem π of
      None ⇒ None
    | Some π ⇒
        if enabled-opr' π s then
          simulate-plan' problem πs (execute-opr' π s)
        else None
  )
  ⟨proof⟩


definition initial-state' :: ast-problem ⇒ state where
  initial-state' problem ≡ let astI = ast-problem.astI problem in (
     λv. if v<length astI then Some (astI!v) else None
   )

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


context wf-ast-problem
begin

  lemma simulate-plan'-correct:
    assumes s∈valid-states
    shows simulate-plan' problem πs s = simulate-plan πs s
    ⟨proof⟩

  lemma simulate-plan'-correct-paper:
    assumes s∈valid-states
    shows simulate-plan' problem πs s = Some s'
```

$\longleftrightarrow$ *path-to s πs s'*
⟨*proof*⟩

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

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

end

definition *verify-plan* :: *ast-problem* ⇒ *plan* ⇒ *String.literal* + *unit* **where**
verify-plan problem πs = (
if ast-problem.well-formed problem then
if check-plan' problem πs then Inr () *else Inl* (*STR ''Invalid plan''*)
else Inl (*STR ''Problem not well formed''*)
)

lemma *verify-plan-correct*:
verify-plan problem πs = *Inr* ()
$\longleftrightarrow$ *ast-problem.well-formed problem* ∧ *ast-problem.valid-plan problem πs*
⟨*proof*⟩

definition *nat-opt-of-integer* :: *integer* ⇒ *nat option* **where**
nat-opt-of-integer i = (*if* (*i* ≥ *0*) *then Some* (*nat-of-integer i*) *else None*)

export-code *verify-plan nat-of-integer integer-of-nat nat-opt-of-integer Inl Inr String.explode String.implode*
in *SML*
module-name *SASP-Checker-Exported*

end

3 PDDL and STRIPS Semantics

theory *PDDL-STRIPS-Semantics*
imports
Propositional-Proof-Systems.Formulas
Propositional-Proof-Systems.Sema
Propositional-Proof-Systems.Consistency
Automatic-Refinement.Misc
Automatic-Refinement.Refine-Util

begin
no-notation *insert* (- ▷ - [*56*,*55*] *55*)

3.1 Utility Functions

definition *index-by f l* ≡ *map-of* (*map* (λ*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* ⟷ (*x*∈*set l* ∧ *f x* = *n*)
 ⟨*proof*⟩

lemma *index-by-eq-SomeD*:
 shows *index-by f l n* = *Some x* ⟹ (*x*∈*set l* ∧ *f x* = *n*)
 ⟨*proof*⟩

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*)
 ⟨*proof*⟩

lemma *rtrancl-image-idem*[*simp*]: R^* ‘‘ R^* ‘‘ *s* = R^* ‘‘ *s*
 ⟨*proof*⟩

3.2 Abstract Syntax

3.2.1 Generic Entities

type-synonym *name* = *string*

datatype *predicate* = *Pred* (*name*: *name*)

Some of the AST entities are defined over a polymorphic *′val* type, which gets either instantiated by variables (for domains) or objects (for problems).

An atom is either a predicate with arguments, or an equality statement.

datatype *′ent atom* = *predAtm* (*predicate*: *predicate*) (*arguments*: *′ent list*)
 | *Eq* (*lhs*: *′ent*) (*rhs*: *′ent*)

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

datatype *type* = *Either* (*primitives*: *name list*)

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

datatype $'ent$ *ast-effect* = *Effect* (*adds*: ($'ent$ *atom formula*) *list*) (*dels*: ($'ent$ *atom formula*) *list*)

Variables are identified by their names.

datatype *variable* = *varname*: *Var name*

Objects and constants are identified by their names

datatype *object* = *name*: *Obj name*

datatype *term* = *VAR variable* | *CONST object*
hide-const (**open**) *VAR CONST* — Refer to constructors by qualified names only

3.2.2 Domains

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

datatype *ast-action-schema* = *Action-Schema*
(*name*: *name*)
(*parameters*: (*variable* $\times$ *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*)

3.2.3 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-problem* = *Problem*
(*domain*: *ast-domain*)
(*objects*: (*object* $\times$ *type*) *list*)
(*init*: *object atom formula list*)
(*goal*: *object atom formula*)

3.2.4 Plans

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

type-synonym *plan* = *plan-action list*

3.2.5 Ground Actions

The following datatype represents an action scheme that has been instantiated by replacing the arguments with concrete objects, also called ground action.

datatype *ground-action* = *Ground-Action*
(*precondition*: (*object atom*) *formula*)
(*effect*: *object ast-effect*)

3.3 Closed-World Assumption, Equality, and Negation

Discriminator for atomic predicate formulas.

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

The world model is a set of (atomic) formulas

type-synonym *world-model* = *object atom formula set*

It is basic, if it only contains atoms

definition *wm-basic* $M \equiv \forall a \in M.$ *is-predAtom* a

A valuation extracted from the atoms of the world model

definition *valuation* :: *world-model* ⇒ *object atom valuation*
where *valuation* $M \equiv \lambda$*predAtm* p xs ⇒ *Atom* (*predAtm* p xs) $\in M$ | *Eq* a b ⇒ $a=b$

Augment a world model by adding negated versions of all atoms not contained in it, as well as interpretations of equality.

definition *close-world* :: *world-model* ⇒ *world-model* **where** *close-world* M =
$M \cup \{\neg$(*Atom* (*predAtm* p as)) | p as. *Atom* (*predAtm* p as) $\notin M\}$
$\cup$ {*Atom* (*Eq* a a) | a. *True*} $\cup$ {¬(*Atom* (*Eq* a b)) | a b. $a \neq b$}

definition *close-neg* $M \equiv M \cup \{\neg$(*Atom* a) | a. *Atom* $a \notin M\}$
lemma *wm-basic* $M \Longrightarrow$ *close-world* M = *close-neg* ($M \cup$ {*Atom* (*Eq* a a) | a. *True*})
⟨*proof*⟩

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

$M\ {}^{c}\Vdash_{=}\ \varphi \equiv$ *close-world* $M \Vdash \varphi$

lemma
close-world-extensive: $M \subseteq$ *close-world* M **and**
close-world-idem[*simp*]: *close-world* (*close-world* M) = *close-world* M
⟨*proof*⟩

lemma *in-close-world-conv*:
$\varphi \in$ *close-world* $M \longleftrightarrow$ (
  $\varphi{\in}M$
  $\vee$ ($\exists\, p\ as.\ \varphi{=}\neg(Atom\ (predAtm\ p\ as)) \wedge Atom\ (predAtm\ p\ as){\notin}M$)
  $\vee$ ($\exists\, a.\ \varphi{=}Atom\ (Eq\ a\ a)$)
  $\vee$ ($\exists\, a\ b.\ \varphi{=}\neg(Atom\ (Eq\ a\ b)) \wedge a{\neq}b$)
)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *valuation-aux-2*:
assumes *wm-basic* M
shows ($\forall\, G{\in}$*close-world* M. *valuation* $M \models G$)
⟨*proof*⟩

lemma *val-imp-close-world*: *valuation* $M \models \varphi \Longrightarrow M\ {}^{c}\Vdash_{=}\ \varphi$
⟨*proof*⟩

lemma *close-world-imp-val*:
wm-basic $M \Longrightarrow M\ {}^{c}\Vdash_{=}\ \varphi \Longrightarrow$ *valuation* $M \models \varphi$
⟨*proof*⟩

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

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

theorem *valuation-iff-close-world*:
assumes *wm-basic* M
shows *valuation* $M \models \varphi \longleftrightarrow M\ {}^{c}\Vdash_{=}\ \varphi$
⟨*proof*⟩

3.3.1 Proper Generalization

Adding negation and equality is a proper generalization of the case without negation and equality

fun *is-STRIPS-fmla* :: *'ent atom formula* ⇒ *bool* **where**
  is-STRIPS-fmla (*Atom* (*predAtm* - -)) ⟷ *True*
| *is-STRIPS-fmla* (⊥) ⟷ *True*
| *is-STRIPS-fmla* (φ_1 ∧ φ_2) ⟷ *is-STRIPS-fmla* φ_1 ∧ *is-STRIPS-fmla* φ_2
| *is-STRIPS-fmla* (φ_1 ∨ φ_2) ⟷ *is-STRIPS-fmla* φ_1 ∧ *is-STRIPS-fmla* φ_2
| *is-STRIPS-fmla* (¬⊥) ⟷ *True*
| *is-STRIPS-fmla* - ⟷ *False*

lemma *aux1*: ⟦*wm-basic M*; *is-STRIPS-fmla* φ; *valuation M* ⊨ φ; ∀ G∈M. $\mathcal{A}$ ⊨ G⟧ ⟹ $\mathcal{A}$ ⊨ φ
  ⟨*proof*⟩

lemma *aux2*: ⟦*wm-basic M*; *is-STRIPS-fmla* φ; ∀ $\mathcal{A}$. (∀ G∈M. $\mathcal{A}$ ⊨ G) ⟶ $\mathcal{A}$ ⊨ φ⟧ ⟹ *valuation M* ⊨ φ
  ⟨*proof*⟩

lemma *valuation-iff-STRIPS*:
  assumes *wm-basic M*
  assumes *is-STRIPS-fmla* φ
  shows *valuation M* ⊨ φ ⟷ M ⊫ φ
⟨*proof*⟩

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

theorem *proper-STRIPS-generalization*:
  ⟦*wm-basic M*; *is-STRIPS-fmla* φ⟧ ⟹ M ^{c}⊫$_=$ φ ⟷ M ⊫ φ
  ⟨*proof*⟩

3.4 STRIPS Semantics

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

locale *ast-domain* =
  fixes D :: *ast-domain*
begin

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

  fun *apply-effect* :: *object ast-effect* ⇒ *world-model* ⇒ *world-model*
  where
    apply-effect (*Effect a d*) *s* = (*s* − *set d*) ∪ (*set a*)

Execute a ground action

definition *execute-ground-action* :: *ground-action* ⇒ *world-model* ⇒ *world-model*
where
execute-ground-action a M = *apply-effect* (*effect a*) *M*

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

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

fun *ground-action-path*
:: *world-model* ⇒ *ground-action list* ⇒ *world-model* ⇒ *bool*
where
ground-action-path M [] $M' \longleftrightarrow (M = M')$
| *ground-action-path* M $(\alpha\#\alpha s)$ $M' \longleftrightarrow M$ $^c\Vdash_=$ *precondition* α
∧ *ground-action-path* (*execute-ground-action* α M) αs M'

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

lemma *ground-action-path-in-paper*:
ground-action-path M [] $M' \longleftrightarrow (M = M')$
ground-action-path M $(\alpha\#\alpha s)$ $M' \longleftrightarrow M$ $^c\Vdash_=$ *precondition* α
∧ (*ground-action-path* (*apply-effect* (*effect* α) M) αs M')
⟨*proof*⟩

end — Context of *ast-domain*

3.5 Well-Formedness of PDDL

fun *ty-term* **where**
ty-term varT objT (*term.VAR v*) = *varT v*
| *ty-term varT objT* (*term.CONST c*) = *objT c*

lemma *ty-term-mono*: *varT* $\subseteq_m$ *varT'* $\Longrightarrow$ *objT* $\subseteq_m$ *objT'* $\Longrightarrow$
ty-term varT objT $\subseteq_m$ *ty-term varT' objT'*
⟨*proof*⟩

context *ast-domain* **begin**

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

definition *sig* :: *predicate* ⇀ *type list* **where**
sig ≡ *map-of* (*map* (λ*PredDecl p n* ⇒ (*p*,*n*)) (*predicates D*))

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

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

fun *subtype-edge* **where**
subtype-edge $(ty,superty) = (superty,ty)$

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

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

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

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

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

Checks whether an entity has a given type

definition *is-of-type* :: $'ent \Rightarrow type \Rightarrow bool$ **where**
is-of-type v T $\longleftrightarrow$ (
case ty-ent v of
Some vT $\Rightarrow$ *of-type vT T*
| *None* $\Rightarrow$ *False*)

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

Predicate-atoms are well-formed if their arguments match the signature, equalities are well-formed if the arguments are valid objects (have a type).
TODO: We could check that types may actually overlap

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

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

fun *wf-fmla* :: ($'ent$ *atom*) *formula* $\Rightarrow$ *bool* **where**
wf-fmla (*Atom a*) $\longleftrightarrow$ *wf-atom a*
| *wf-fmla* $(\bot)$ $\longleftrightarrow$ *True*
| *wf-fmla* $(\varphi 1 \wedge \varphi 2)$ $\longleftrightarrow$ (*wf-fmla* $\varphi 1$ $\wedge$ *wf-fmla* $\varphi 2$)
| *wf-fmla* $(\varphi 1 \vee \varphi 2)$ $\longleftrightarrow$ (*wf-fmla* $\varphi 1$ $\wedge$ *wf-fmla* $\varphi 2$)
| *wf-fmla* $(\neg\varphi)$ $\longleftrightarrow$ *wf-fmla* φ

| *wf-fmla* ($\varphi 1 \rightarrow \varphi 2$) $\longleftrightarrow$ (*wf-fmla* $\varphi 1$ $\wedge$ *wf-fmla* $\varphi 2$)

lemma *wf-fmla* φ = ($\forall$ $a \in$ *atoms* φ. *wf-atom a*)
⟨*proof*⟩

Special case for a well-formed atomic predicate formula

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

lemma *wf-fmla-atom-alt*: *wf-fmla-atom* φ $\longleftrightarrow$ *is-predAtom* φ $\wedge$ *wf-fmla* φ
⟨*proof*⟩

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

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

end — Context fixing *ty-ent*

definition *constT* :: *object* $\rightharpoonup$ *type* **where**
constT $\equiv$ *map-of* (*consts D*)

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

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

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

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

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

fun *wf-predicate-decl* **where**
wf-predicate-decl (*PredDecl p Ts*) $\longleftrightarrow$ ($\forall$ *T*$\in$*set Ts*. *wf-type T*)

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

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

A domain is well-formed if

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

definition *wf-domain* :: *bool* **where**
wf-domain ≡
wf-types
∧ *distinct* (*map* (*predicate-decl.pred*) (*predicates* D))
∧ (∀ *p*∈*set* (*predicates* D). *wf-predicate-decl* *p*)
∧ *distinct* (*map* *fst* (*consts* D))
∧ (∀ (*n*, *T*)∈*set* (*consts* D). *wf-type* *T*)
∧ *distinct* (*map* *ast-action-schema.name* (*actions* D))
∧ (∀ *a*∈*set* (*actions* D). *wf-action-schema* *a*)

end — locale *ast-domain*

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

locale *ast-problem* = *ast-domain* *domain* *P*
for *P* :: *ast-problem*
begin

We refer to the problem domain as D

abbreviation D ≡ *ast-problem.domain* *P*

definition *objT* :: *object* ⇀ *type* **where**
objT ≡ *map-of* (*objects* *P*) ++ *constT*

lemma *objT-alt*: *objT* = *map-of* (*consts* D @ *objects* *P*)
⟨*proof*⟩

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

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

definition *wf-world-model* **where**
wf-world-model *M* = (∀ *f*∈*M*. *wf-fmla-atom* *objT* *f*)

definition *wf-problem* **where**

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

fun *wf-effect-inst* :: *object ast-effect* ⇒ *bool* **where**
  wf-effect-inst (*Effect* (*a*) (*d*))
    ⟷ (∀ *a*∈*set a* ∪ *set d*. *wf-fmla-atom objT a*)

lemma *wf-effect-inst-alt*: *wf-effect-inst eff* = *wf-effect objT eff*
  ⟨*proof*⟩

end — locale *ast-problem*

Locale to express a well-formed domain

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

Locale to express a well-formed problem

locale *wf-ast-problem* = *ast-problem P* **for** *P* +
  assumes *wf-problem*: *wf-problem*
begin
  sublocale *wf-ast-domain domain P*
    ⟨*proof*⟩

end — locale *wf-ast-problem*

3.6 PDDL Semantics

context *ast-domain* **begin**

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

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

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

fun *instantiate-action-schema*
  :: *ast-action-schema* ⇒ *object list* ⇒ *ground-action*

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

end — Context of *ast-domain*

context *ast-problem* **begin**

Initial model

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

Resolve a plan action and instantiate the referenced action schema.

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

Check whether object has specified type

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

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

lemma *is-obj-of-type-alt*: *is-obj-of-type* = *is-of-type objT*
  ⟨*proof*⟩

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

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

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

fun *wf-plan-action* :: *plan-action* ⇒ *bool* **where**
  wf-plan-action (*PAction n args*) = (

$case\ resolve\text{-}action\text{-}schema\ n\ of$
$\quad None \Rightarrow False$
$\mid Some\ a \Rightarrow$
$\qquad action\text{-}params\text{-}match\ a\ args$
$\quad \wedge\ wf\text{-}effect\text{-}inst\ (effect\ (instantiate\text{-}action\text{-}schema\ a\ args))$
$\quad)$

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*
$:: world\text{-}model \Rightarrow plan\text{-}action\ list \Rightarrow world\text{-}model \Rightarrow bool$
where
$plan\text{-}action\text{-}path\ M\ \pi s\ M' =$
$\quad ((\forall \pi \in set\ \pi s.\ wf\text{-}plan\text{-}action\ \pi)$
$\quad \wedge\ ground\text{-}action\text{-}path\ M\ (map\ resolve\text{-}instantiate\ \pi s)\ M')$

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

definition $valid\text{-}plan\text{-}from :: world\text{-}model \Rightarrow plan \Rightarrow bool$ **where**
$valid\text{-}plan\text{-}from\ M\ \pi s = (\exists M'.\ plan\text{-}action\text{-}path\ M\ \pi s\ M' \wedge M'\ {}^{c}\Vdash_{=}\ (goal\ P))$

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

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

Concise definition used in paper:

lemma $valid\text{-}plan\ \pi s \equiv \exists M'.\ plan\text{-}action\text{-}path\ I\ \pi s\ M' \wedge M'\ {}^{c}\Vdash_{=}\ (goal\ P)$
$\langle proof \rangle$

end — Context of *ast-problem*

3.7 Preservation of Well-Formedness

3.7.1 Well-Formed Action Instances

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

context *ast-problem* **begin**

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

fun $wf\text{-}ground\text{-}action :: ground\text{-}action \Rightarrow bool$ **where**
$wf\text{-}ground\text{-}action\ (Ground\text{-}Action\ pre\ eff) \longleftrightarrow ($

wf-fmla objT pre
∧ *wf-effect objT eff*
)

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

We begin with some auxiliary lemmas before the actual theorem.

lemma (**in** *ast-domain*) *of-type-refl*[*simp*, *intro*!]: *of-type T T*
⟨*proof*⟩

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

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

lemma *wf-atom-mono*:
assumes *SS*: *tys* $\subseteq_m$ *tys′*
assumes *WF*: *wf-atom tys a*
shows *wf-atom tys′ a*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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

end

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

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

⟨*proof*⟩
end — Context of *ast-problem*

3.7.2 Preservation

context *ast-problem* **begin**

We start by defining two shorthands for enabledness and execution of a plan action.

Shorthand for enabled plan action: It is well-formed, and the precondition holds for its instance.

definition *plan-action-enabled* :: *plan-action* ⇒ *world-model* ⇒ *bool* **where**
plan-action-enabled π *M*
⟷ *wf-plan-action* π ∧ *M* $^c\Vdash_=$ *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*
⟨*proof*⟩

lemma *plan-action-path-Cons*[*simp*]:
plan-action-path M (π#πs) *M′* ⟷
plan-action-enabled π *M*
∧ *plan-action-path* (*execute-plan-action* π *M*) πs *M′*
⟨*proof*⟩

end — Context of *ast-problem*

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

The initial world model is well-formed

lemma *wf-I*: *wf-world-model I*
⟨*proof*⟩

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

lemma *wf-apply-effect*:
assumes *wf-effect objT e*
assumes *wf-world-model s*
shows *wf-world-model* (*apply-effect e s*)
⟨*proof*⟩

Execution of plan actions preserves well-formedness

theorem *wf-execute*:
assumes *plan-action-enabled* π *s*
assumes *wf-world-model s*
shows *wf-world-model* (*execute-plan-action* π *s*)
⟨*proof*⟩

theorem *wf-execute-compact-notation*:
plan-action-enabled π *s* $\Longrightarrow$ *wf-world-model s*
$\Longrightarrow$ *wf-world-model* (*execute-plan-action* π *s*)
⟨*proof*⟩

Execution of a plan preserves well-formedness

corollary *wf-plan-action-path*:
assumes *wf-world-model M* **and** *plan-action-path M* πs M'
shows *wf-world-model* M'
⟨*proof*⟩

end — Context of *wf-ast-problem*

end — Theory

4 Executable PDDL Checker

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

Error-Monad-Add
HOL.String

HOL−Library.Code-Target-Nat

HOL−Library.While-Combinator

Containers.Containers
begin

4.1 Generic DFS Reachability Checker

Used for subtype checks

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

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

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

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

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

context
  fixes $w_0 :: 'a\ list$
  assumes *finite-dfs-reachable*[*simp*, *intro*!]: $finite\ (E^*\ ``\ set\ w_0)$
begin

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

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

lemma *card-decreases*:
  $[\![finite\ V;\ y \notin V;\ dfs\text{-}reachable\text{-}invar\ D\ V\ (Set.insert\ y\ W)\ brk\ ]\!]$
  $\Longrightarrow card\ (E^*\ ``\ W_0 - Set.insert\ y\ V) < card\ (E^*\ ``\ W_0 - V)$
  ⟨*proof*⟩

lemma *all-neq-Cons-is-Nil*[*simp*]:
  $(\forall y\ ys.\ x2 \neq y\ \#\ ys) \longleftrightarrow x2 = []$ ⟨*proof*⟩

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

end

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

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

lemma *tab-succ-correct*: *E-of-succ* (*tab-succ l*) = *set l*
⟨*proof*⟩

end

lemma *finite-imp-finite-dfs-reachable*:
⟦*finite E*; *finite S*⟧ $\Longrightarrow$ *finite* (E^* ‘‘*S*)
⟨*proof*⟩

lemma *dfs-reachable-tab-succ-correct*: *dfs-reachable* (*tab-succ l*) *D* vs_0 $\longleftrightarrow$ *Collect* *D* $\cap$ (*set l*)* ‘‘*set* vs_0 $\neq$ {}
⟨*proof*⟩

4.2 Implementation Refinements

4.2.1 Of-Type

definition *of-type-impl G oT T* ≡ ($\forall$ *pt*∈*set* (*primitives oT*). *dfs-reachable G* ((=) *pt*) (*primitives T*))

fun *ty-term′* **where**
ty-term′ varT objT (*term.VAR v*) = *varT v*
| *ty-term′ varT objT* (*term.CONST c*) = *Mapping.lookup objT c*

lemma *ty-term′-correct-aux*: *ty-term′ varT objT t* = *ty-term varT* (*Mapping.lookup* *objT*) *t*
⟨*proof*⟩

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

context *ast-domain* **begin**

definition *of-type1 pt T* $\longleftrightarrow$ *pt* $\in$ *subtype-rel** ‘‘ *set* (*primitives T*)

lemma *of-type-refine1*: *of-type oT T* $\longleftrightarrow$ ($\forall$ *pt*∈*set* (*primitives oT*). *of-type1 pt* *T*)

⟨*proof*⟩

definition *STG* ≡ (*tab-succ* (*map subtype-edge* (*types D*)))

lemma *subtype-rel-impl*: *subtype-rel* = *E-of-succ* (*tab-succ* (*map subtype-edge* (*types D*)))
⟨*proof*⟩

lemma *of-type1-impl*: *of-type1 pt T* ⟷ *dfs-reachable* (*tab-succ* (*map subtype-edge* (*types D*))) ((=)*pt*) (*primitives T*)
⟨*proof*⟩

lemma *of-type-impl-correct*: *of-type-impl STG oT T* ⟷ *of-type oT T*
⟨*proof*⟩

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

lemma *mp-objT-correct*[*simp*]: *Mapping.lookup mp-constT* = *constT*
⟨*proof*⟩

Lifting the subtype-graph through wf-checker

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

definition *is-of-type' stg v T* ⟷ (
case ty-ent v of
Some vT ⇒ *of-type-impl stg vT T*
| *None* ⇒ *False*)

lemma *is-of-type'-correct*: *is-of-type' STG v T* = *is-of-type ty-ent v T*
⟨*proof*⟩

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

lemma *wf-pred-atom'-correct*: *wf-pred-atom' STG pvs* = *wf-pred-atom ty-ent pvs*
⟨*proof*⟩

fun *wf-atom'* :: - ⇒ *'ent atom* ⇒ *bool* **where**
wf-atom' stg (*atom.predAtm p vs*) ⟷ *wf-pred-atom' stg* (*p*,*vs*)
| *wf-atom' stg* (*atom.Eq a b*) = (*ty-ent a* ≠ *None* ∧ *ty-ent b* ≠ *None*)

lemma *wf-atom'-correct*: *wf-atom' STG a* = *wf-atom ty-ent a*
⟨*proof*⟩

fun $\textit{wf-fmla}'$ :: - $\Rightarrow$ ($'\textit{ent atom}$) *formula* $\Rightarrow$ *bool* **where**
$\textit{wf-fmla}'\ stg\ (\textit{Atom}\ a) \longleftrightarrow \textit{wf-atom}'\ stg\ a$
$|\ \textit{wf-fmla}'\ stg\ \bot \longleftrightarrow \textit{True}$
$|\ \textit{wf-fmla}'\ stg\ (\varphi 1 \wedge \varphi 2) \longleftrightarrow (\textit{wf-fmla}'\ stg\ \varphi 1 \wedge \textit{wf-fmla}'\ stg\ \varphi 2)$
$|\ \textit{wf-fmla}'\ stg\ (\varphi 1 \vee \varphi 2) \longleftrightarrow (\textit{wf-fmla}'\ stg\ \varphi 1 \wedge \textit{wf-fmla}'\ stg\ \varphi 2)$
$|\ \textit{wf-fmla}'\ stg\ (\varphi 1 \rightarrow \varphi 2) \longleftrightarrow (\textit{wf-fmla}'\ stg\ \varphi 1 \wedge \textit{wf-fmla}'\ stg\ \varphi 2)$
$|\ \textit{wf-fmla}'\ stg\ (\neg\varphi) \longleftrightarrow \textit{wf-fmla}'\ stg\ \varphi$

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

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

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

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

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

end — Context fixing *ty-ent*

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

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

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

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

lemma *wf-domain′-correct*: *wf-domain′ STG mp-constT* = *wf-domain*
⟨*proof*⟩

end — Context of *ast-domain*

4.2.2 Application of Effects

context *ast-domain* **begin**

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

fun *apply-effect-exec*
:: *object ast-effect* ⇒ *world-model* ⇒ *world-model*
where
apply-effect-exec (*Effect a d*) *s*
= *fold* (λ*add s. Set.insert add s*) *a*
(*fold* (λ*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*⟩

lemmas *apply-effect-eq-impl-eq*
= *apply-effect-exec-refine*[*symmetric*, *unfolded apply-effect-exec.simps*]

end — Context of *ast-domain*

4.2.3 Well-Formedness

context *ast-problem* **begin**

We start by defining a mapping from objects to types. The container framework will generate efficient, red-black tree based code for that later.

type-synonym *objT* = (*object*, *type*) *mapping*

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

lemma *mp-objT-correct*[*simp*]: *Mapping.lookup mp-objT* = *objT*
⟨*proof*⟩

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*
⟨*proof*⟩

We refine the well-formedness checks to use the mapping

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

lemma *wf-fact′-correct*[*simp*]: *wf-fact′ mp-objT STG* = *wf-fact*
⟨*proof*⟩

definition *wf-fmla-atom2′ mp stg f*
= (*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
⟨*proof*⟩

definition *wf-problem′ stg conT mp* ≡
wf-domain′ stg conT
∧ *distinct* (*map fst* (*objects P*) @ *map fst* (*consts D*))
∧ (∀ (*n*, *T*)∈*set* (*objects P*). *wf-type T*)
∧ *distinct* (*init P*)
∧ (∀ *f*∈*set* (*init P*). *wf-fmla-atom2′ mp stg f*)
∧ *wf-fmla′* (*Mapping.lookup mp*) *stg* (*goal P*)

lemma *wf-problem′-correct*:
wf-problem′ STG mp-constT mp-objT = *wf-problem*
⟨*proof*⟩

Instantiating actions will yield well-founded effects. Corollary of ⟦*action-params-match ?a ?args*; *wf-action-schema ?a*⟧ $\Longrightarrow$ *wf-ground-action* (*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*)
⟨*proof*⟩

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*⟩

end — Context of *ast-domain*

4.2.4 Execution of Plan Actions

We will perform two refinement steps, to summarize redundant operations

We first lift action schema lookup into the error monad.

context *ast-domain* **begin**
  definition *resolve-action-schemaE n* ≡
    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 *wm-basic s*
  shows *holds s F* ⟷ *close-world s* ⊫ *F*
  ⟨*proof*⟩

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 *wm-basic s*
shows *en-exE a s* = *Inr s′*
⟷ *plan-action-enabled a s* ∧ *s′* = *execute-plan-action a s*
⟨*proof*⟩

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

abbreviation *action-params-match2 stg mp a args*
≡ *list-all2* (*is-obj-of-type-impl stg mp*)
args (*map snd* (*ast-action-schema.parameters a*))

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

Justification of refinement

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

Combination of the two refinement lemmas

lemma (**in** *wf-ast-problem*) *en-exE2-return-iff*:
assumes *wm-basic M*
shows *en-exE2 STG mp-objT a M* = *Inr M′*
⟷ *plan-action-enabled a M* ∧ *M′* = *execute-plan-action a M*
⟨*proof*⟩

lemma (**in** *wf-ast-problem*) *en-exE2-return-iff-compact-notation*:
⟦*wm-basic s*⟧ ⟹
en-exE2 STG mp-objT a s = *Inr s′*
⟷ *plan-action-enabled a s* ∧ *s′* = *execute-plan-action a s*
⟨*proof*⟩

end — Context of *ast-problem*

4.2.5 Checking of Plan

context *ast-problem* **begin**

First, we combine the well-formedness check of the plan actions and their execution into a single iteration.

fun *valid-plan-from1* :: *world-model* ⇒ *plan* ⇒ *bool* **where**
valid-plan-from1 s [] ⟷ *close-world 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*⟩

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 stg mp si s []
= *check* (*holds s* (*goal P*)) (*ERRS ''Postcondition does not hold''*)
| *valid-plan-fromE stg mp si s* (*π*#*πs*) = *do* {
s ← *en-exE2 stg mp π s*
<+? (*λe* -. *shows ''at step '' o shows si o shows '': '' o e* ());
valid-plan-fromE stg 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*:
⟦ *wm-basic s*; *wf-plan-action a* ⟧
⟹ *wm-basic* (*execute-plan-action a s*)
⟨*proof*⟩

Refinement lemma for our plan checking algorithm

lemma (**in** *wf-ast-problem*) *valid-plan-fromE-return-iff*[*return-iff*]:
assumes *wm-basic s*
shows *valid-plan-fromE STG mp-objT k s πs* = *Inr* () ⟷ *valid-plan-from s πs*
⟨*proof*⟩

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

end — Context of *ast-problem*

4.3 Executable Plan Checker

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

definition *check-all-list P l msg msgf* ≡
forallM (*λx. check* (*P x*) (λ-::*unit. shows msg o shows* ′′: ′′ *o msgf x*)) *l* <+? *snd*

lemma *check-all-list-return-iff*[*return-iff*]: *check-all-list P l msg msgf* = *Inr* () ⟷ (∀ *x*∈*set l. P x*)
⟨*proof*⟩

definition *check-wf-types D* ≡ *do* {
check-all-list (λ(-,*t*). *t*=′′*object*′′ ∨ *t*∈*fst*‘*set* (*types D*)) (*types D*) ′′*Undeclared supertype*′′ (*shows o snd*)
}

lemma *check-wf-types-return-iff*[*return-iff*]: *check-wf-types D* = *Inr* () ⟷ *ast-domain.wf-types D*
⟨*proof*⟩

definition *check-wf-domain D stg conT* ≡ *do* {
check-wf-types D;
check (*distinct* (*map* (*predicate-decl.pred*) (*predicates D*))) (*ERRS* ′′*Duplicate predicate declaration*′′);
check-all-list (*ast-domain.wf-predicate-decl D*) (*predicates D*) ′′*Malformed predicate declaration*′′ (*shows o predicate.name o predicate-decl.pred*);
check (*distinct* (*map fst* (*consts D*))) (*ERRS* ′′*Duplicate constant declaration*′′);
check (∀ (*n*,*T*)∈*set* (*consts D*). *ast-domain.wf-type D T*) (*ERRS* ′′*Malformed type*′′);
check (*distinct* (*map ast-action-schema.name* (*actions D*))) (*ERRS* ′′*Duplicate action name*′′);
check-all-list (*ast-domain.wf-action-schema*′ *D stg conT*) (*actions D*) ′′*Malformed action*′′ (*shows o ast-action-schema.name*)

}

lemma *check-wf-domain-return-iff*[*return-iff*]:
check-wf-domain D stg conT = *Inr* () ⟷ *ast-domain.wf-domain*′ *D stg conT*
⟨*proof*⟩

definition *prepend-err-msg msg e* ≡ λ-::*unit. shows msg o shows* ′′: ′′ *o e* ()

definition *check-wf-problem P stg conT mp* ≡ *do* {
let D = *ast-problem.domain P*;
check-wf-domain D stg conT <+? *prepend-err-msg ''Domain not well−formed''*;
check (*distinct* (*map fst* (*objects P*) @ *map fst* (*consts D*))) (*ERRS ''Duplicate object declaration''*);
check ((∀ (*n*,*T*)∈*set* (*objects P*). *ast-domain.wf-type D T*)) (*ERRS ''Malformed type''*);
check (*distinct* (*init P*)) (*ERRS ''Duplicate fact in initial state''*);
check (∀ *f*∈*set* (*init P*). *ast-problem.wf-fmla-atom2′ P mp stg f*) (*ERRS ''Malformed formula in initial state''*);
check (*ast-domain.wf-fmla′ D* (*Mapping.lookup mp*) *stg* (*goal P*)) (*ERRS ''Malformed goal formula''*)
}

lemma *check-wf-problem-return-iff*[*return-iff*]:
check-wf-problem P stg conT mp = *Inr* () ⟷ *ast-problem.wf-problem′ P stg conT mp*
⟨*proof*⟩

definition *check-plan P πs* ≡ *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;
ast-problem.valid-plan-fromE P stg mp 1 (*ast-problem.I P*) *πs*
} <+? (λ*e*. *String.implode* (*e* () *''''*))

Correctness theorem of the plan checker: It returns *Inr* () if and only if the problem is well-formed and the plan is valid.

theorem *check-plan-return-iff*[*return-iff*]: *check-plan P πs* = *Inr* ()
⟷ *ast-problem.wf-problem P* ∧ *ast-problem.valid-plan P πs*
⟨*proof*⟩

4.4 Code Setup

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

4.4.1 Code Equations

We first register the code equations for the functions of the checker. Note that we not necessarily register the original code equations, but also optimized ones.

lemmas *wf-domain-code* =
ast-domain.sig-def
ast-domain.wf-types-def
ast-domain.wf-type.simps

ast-domain.wf-predicate-decl.simps
ast-domain.STG-def
ast-domain.is-of-type'-def
ast-domain.wf-atom'.simps
ast-domain.wf-pred-atom'.simps
ast-domain.wf-fmla'.simps
ast-domain.wf-fmla-atom1'.simps
ast-domain.wf-effect'.simps
ast-domain.wf-action-schema'.simps
ast-domain.wf-domain'-def
ast-domain.subst-term.simps
ast-domain.mp-constT-def

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

lemmas *wf-problem-code* =
ast-problem.wf-problem'-def
ast-problem.wf-fact'-def

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

ast-problem.wf-fact-def
ast-problem.wf-plan-action.simps

ast-domain.subtype-edge.simps
declare *wf-problem-code*[*code*]

lemmas *check-code* =
ast-problem.valid-plan-def
ast-problem.valid-plan-fromE.simps
ast-problem.en-exE2-def
ast-problem.resolve-instantiate.simps
ast-domain.resolve-action-schema-def
ast-domain.resolve-action-schemaE-def
ast-problem.I-def
ast-domain.instantiate-action-schema.simps
ast-domain.apply-effect-exec.simps

ast-domain.apply-effect-eq-impl-eq

ast-problem.holds-def
ast-problem.mp-objT-def
ast-problem.is-obj-of-type-impl-def
ast-problem.wf-fmla-atom2'-def
valuation-def
declare *check-code*[*code*]

4.4.2 Setup for Containers Framework

derive *ceq predicate atom object formula*
derive *ccompare predicate atom object formula*
derive (*rbt*) *set-impl atom formula*

derive (*rbt*) *mapping-impl object*

derive *linorder predicate object atom object atom formula*

4.4.3 More Efficient Distinctness Check for Linorders

fun *no-stutter* :: *'a list* ⇒ *bool* **where**
  no-stutter [] = *True*
| *no-stutter* [-] = *True*
| *no-stutter* (*a*#*b*#*l*) = (*a*≠*b* ∧ *no-stutter* (*b*#*l*))

lemma *sorted-no-stutter-eq-distinct*: *sorted l* ⟹ *no-stutter l* ⟷ *distinct l*
  ⟨*proof*⟩

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

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

4.4.4 Code Generation

export-code
  check-plan
  nat-of-integer integer-of-nat Inl Inr
  predAtm Eq predicate Pred Either Var Obj PredDecl BigAnd BigOr
  formula.Not formula.Bot Effect ast-action-schema.Action-Schema
  map-atom Domain Problem PAction
  term.CONST term.VAR
  String.explode String.implode
  in *SML*
  module-name *PDDL-Checker-Exported*
  file *PDDL-STRIPS-Checker-Exported.sml*

export-code *ast-domain.apply-effect-exec* **in** *SML* **module-name** *ast-domain*

end — Theory

5 Soundness theorem for the STRIPS semantics

We prove the soundness theorem according to [4].

theory *Lifschitz-Consistency*
imports *PDDL-STRIPS-Semantics*
begin

States are modeled as valuations of our underlying predicate logic.

type-synonym *state* = (*predicate*×*object list*) *valuation*

context *ast-domain* **begin**

An action is a partial function from states to states.

type-synonym *action* = *state* ⇀ *state*

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

Definition B (i)–(iv) in Lifschitz's paper [4]

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

definition *close-eq s* = (λ*predAtm p xs* ⇒ *s* (*p*,*xs*) | *Eq a b* ⇒ *a*=*b*)

lemma *close-eq-predAtm*[*simp*]: *close-eq s* (*predAtm p xs*) ⟷ *s* (*p*,*xs*)
⟨*proof*⟩

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

abbreviation *entail-eq* :: *state* ⇒ *object atom formula* ⇒ *bool* (**infix** $\models_=$ *55*)
where *entail-eq s f* ≡ *close-eq s* ⊨ *f*

fun *sound-opr* :: *ground-action* ⇒ *action* ⇒ *bool* **where**
sound-opr (*Ground-Action pre* (*Effect add del*)) *f* ⟷
(∀ *s*. *s* $\models_=$ *pre* ⟶
(∃ *s'*. *f s* = *Some s'* ∧ (∀ *atm*. *is-predAtom atm* ∧ *atm* ∉ *set del* ∧ *s* $\models_=$ *atm*
⟶ *s'* $\models_=$ *atm*)
∧ (∀ *atm*. *is-predAtom atm* ∧ *atm* ∉ *set add* ∧ *s* $\models_=$ *Not atm* ⟶ *s'*
$\models_=$ *Not atm*)
∧ (∀ *fmla*. *fmla* ∈ *set add* ⟶ *s'* $\models_=$ *fmla*)
∧ (∀ *fmla*. *fmla* ∈ *set del* ∧ *fmla* ∉ *set add* ⟶ *s'* $\models_=$ (*Not fmla*))
))
∧ (∀ *fmla*∈*set add*. *is-predAtom fmla*)

lemma *sound-opr-alt*:
sound-opr opr f =
((∀ *s*. *s* $\models_=$ (*precondition opr*) ⟶
(∃ *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*))
⟨*proof*⟩

Definition B (v)–(vii) in Lifschitz's paper [4]

definition *sound-system*
:: *ground-action set*
$\Rightarrow$ *world-model*
$\Rightarrow$ *state*
$\Rightarrow$ (*ground-action* $\Rightarrow$ *action*)
$\Rightarrow$ *bool*
where
sound-system Σ M_0 s_0 *f* $\longleftrightarrow$
(($\forall$ *fmla*$\in$*close-world* M_0. s_0 $\models_=$ *fmla*)
$\wedge$ *wm-basic* M_0
$\wedge$ ($\forall$ $\alpha\in\Sigma$. *sound-opr* 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* ⟨*proof*⟩

lemma *compose-actions-Cons*[*simp*]:
f s = *Some s'* $\Longrightarrow$ *compose-actions* (*f*#*fs*) *s* = *compose-actions fs s'*
⟨*proof*⟩

Soundness Theorem in Lifschitz's paper [4].

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 *ground-action-path* M_0 αs M'
— Which is accepted by the system, yielding result M' (called $R(\alpha s)$ in Lifschitz's paper [4].)
obtains s'
— We have that $f(\alpha s)$ is applicable in initial state, yielding state s' (called $f_{\alpha s}(s_0)$ in Lifschitz's paper [4].)
where *compose-actions* (*map f* αs) s_0 = *Some* s'
— The result world model M' is satisfied in state s'
and $\forall$ *fmla*$\in$*close-world* M'. $s' \models_=$ *fmla*
⟨*proof*⟩

More compact notation of the soundness theorem.

theorem *STRIPS-sema-sound-compact-version*:
sound-system Σ M_0 s_0 $f \Longrightarrow$ *set* $\alpha s \subseteq \Sigma$
$\Longrightarrow$ *ground-action-path* M_0 αs M'
$\Longrightarrow \exists s'$. *compose-actions* (*map f* αs) s_0 = *Some* s'
$\wedge$ ($\forall$ *fmla*$\in$*close-world* M'. $s' \models_=$ *fmla*)
⟨*proof*⟩

end — Context of *ast-domain*

5.1 Soundness Theorem for PDDL

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

Mapping world models to states

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

lemma *wm-to-state-eq*[*simp*]: *wm-to-state M* (*p*, *as*) $\longleftrightarrow$ *Atom* (*predAtm p as*) $\in$ *M*
⟨*proof*⟩

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

Mapping AST action instances to actions

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

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

lemma *atom-in-wm-eq*:
s $\models_=$ (*formula.Atom atm*)
⟷ ((*formula.Atom atm*) ∈ *close-eq-M* (*state-to-wm s*))
⟨*proof*⟩

lemma *atom-in-wm-2-eq*:
close-eq (*wm-to-state M*) ⊨ (*formula.Atom atm*)
⟷ ((*formula.Atom atm*) ∈ *close-eq-M M*)
⟨*proof*⟩

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

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

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

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

lemma *wf-fmla-atm-is-atom*: *wf-fmla-atom objT f* ⟹ *is-predAtom f*
⟨*proof*⟩

lemma *wf-act-adds-are-atoms*:
assumes *wf-effect-inst effs ae* ∈ *set* (*adds effs*)
shows *is-predAtom ae*
⟨*proof*⟩

lemma *wf-act-adds-dels-atoms*:
assumes *wf-effect-inst effs ae* ∈ *set* (*dels effs*)
shows *is-predAtom ae*
⟨*proof*⟩

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

⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *wf-eff-impt-wf-eff-inst*: *wf-effect objT eff* $\Longrightarrow$ *wf-effect-inst eff*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *wf-plan-act-is-sound'*:
assumes *wf-plan-action* π
shows *sound-opr*
(*resolve-instantiate* π)
((*pddl-opr-to-act* (*resolve-instantiate* π)))
⟨*proof*⟩

lemma *wf-world-model-has-atoms*: $f \in M$ $\Longrightarrow$ *wf-world-model M* $\Longrightarrow$ *is-predAtom f*
⟨*proof*⟩

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

lemma *wm-to-state-works-for-wf-wm*: *wf-world-model M* $\Longrightarrow$ *fmla*$\in$*M* $\Longrightarrow$ *close-eq* (*wm-to-state M*) $\models$ *fmla*
⟨*proof*⟩

lemma *wm-to-state-works-for-I-closed*:
assumes $x \in$ *close-world I*
shows *close-eq* (*wm-to-state I*) $\models x$
⟨*proof*⟩

lemma *wf-wm-imp-basic*: *wf-world-model M* $\Longrightarrow$ *wm-basic M*
⟨*proof*⟩

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*)
(($\lambda\alpha$. *pddl-opr-to-act* α))
⟨*proof*⟩

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* $s' \wedge (\forall \varphi\in$*close-world M*. $s' \models_= \varphi)$
⟨*proof*⟩

end — Context of *wf-ast-problem*

end

References

[1] M. Abdulaziz and P. Lammich. A Formally Verified Validator for Classical Planning Problems and Solutions. In *International Conference on Tools in Artificial Intelligence (ICTAI)*. IEEE, 2018.

[2] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[3] M. Helmert. The fast downward planning system. *J. Artif. Intell. Res.(JAIR)*, 26:191–246, 2006.

[4] V. Lifschitz. On the semantics of strips. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pages 1–9, 1987.

[5] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL: The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.