Semantics of AI Planning Languages

Mohammad Abdulaziz and Peter Lammich*

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

Contents

1 Semantics of Fast-Downward's Multi-Valued Planning Tasks Language **3**
1.1 Syntax . . . 3
1.1.1 Well-Formedness . . . 3
1.2 Semantics as Transition System . . . 4
1.2.1 Preservation of well-formedness . . . 6

2 An Executable Checker for Multi-Valued Planning Problem Solutions **7**
2.1 Auxiliary Lemmas . . . 7
2.2 Well-formedness Check . . . 8
2.3 Execution . . . 8

3 PDDL and STRIPS Semantics **14**
3.1 Utility Functions . . . 15
3.2 Abstract Syntax . . . 15
3.2.1 Generic Entities . . . 15
3.2.2 Domains . . . 16
3.2.3 Problems . . . 16
3.2.4 Plans . . . 17
3.2.5 Ground Actions . . . 17
3.3 Closed-World Assumption, Equality, and Negation . . . 17
3.3.1 Proper Generalization . . . 19

*Author names are alphabetically ordered.

3.4 STRIPS Semantics . . . 20
3.5 Well-Formedness of PDDL . . . 21
3.6 PDDL Semantics . . . 25
3.7 Preservation of Well-Formedness . . . 27
3.7.1 Well-Formed Action Instances . . . 27
3.7.2 Preservation . . . 31

4 Executable PDDL Checker **33**
4.1 Generic DFS Reachability Checker . . . 34
4.2 Implementation Refinements . . . 36
4.2.1 Of-Type . . . 36
4.2.2 Application of Effects . . . 39
4.2.3 Well-Formedness . . . 39
4.2.4 Execution of Plan Actions . . . 41
4.2.5 Checking of Plan . . . 43
4.3 Executable Plan Checker . . . 45
4.4 Code Setup . . . 47
4.4.1 Code Equations . . . 47
4.4.2 Setup for Containers Framework . . . 49
4.4.3 More Efficient Distinctness Check for Linorders . . . 49
4.4.4 Code Generation . . . 49

5 Soundness theorem for the STRIPS semantics **50**
5.1 Soundness Theorem for PDDL . . . 54

theory *SASP-Semantics*
imports *Main*
begin

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

1.1 Syntax

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

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

type-synonym *plan* = *name list*

1.1.1 Well-Formedness

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

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

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

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

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

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

— Goal
$\wedge$ *wf-partial-state astG*

— Operators
$\wedge$ (*distinct* (*map fst ast*δ))
$\wedge$ ($\forall$ $\pi$$\in$*set ast*$\delta$. *wf-operator* π)

end

locale *wf-ast-problem* = *ast-problem* +
assumes *wf*: *well-formed*
begin
lemma *wf-initial*:
length astI = *numVars*
$\forall$ *x*<*numVars*. *astI*!*x* < *numVals x*
using *wf* **unfolding** *well-formed-def* **by** *auto*

lemma *wf-goal*: *wf-partial-state astG*
using *wf* **unfolding** *well-formed-def* **by** *auto*

lemma *wf-operators*:
distinct (*map fst ast*δ)
$\forall$ $\pi$$\in$*set ast*$\delta$. *wf-operator* π
using *wf* **unfolding** *well-formed-def* **by** *auto*
end

1.2 Semantics as Transition System

type-synonym *state* = *nat* $\rightharpoonup$ *nat*
type-synonym *pstate* = *nat* $\rightharpoonup$ *nat*

context *ast-problem*
begin

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

definition *range-of-var* **where** *range-of-var x* ≡ {*0*..<*numVals x*}

definition *valid-states* :: *state set* **where** *valid-states* ≡ {
s. *dom s* = *Dom* ∧ (∀ *x*∈*Dom*. *the* (*s x*) ∈ *range-of-var x*)
}

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

definition *subsuming-states* :: *pstate* ⇒ *state set*
where *subsuming-states partial* ≡ { *s*∈*valid-states*. *partial* ⊆$_m$ *s* }

definition *G* :: *state set*
where *G* ≡ *subsuming-states* (*map-of astG*)
end

definition *implicit-pres* :: *ast-effect list* ⇒ *ast-precond list* **where**
implicit-pres effs ≡
map (λ(-,*v*,*vpre*,-). (*v*,*the vpre*))
(*filter* (λ(-,-,*vpre*,-). *vpre*≠*None*) *effs*)

context *ast-problem*
begin

definition *lookup-operator* :: *name* ⇒ *ast-operator option* **where**
lookup-operator name ≡ *find* (λ(*n*,-,-,-). *n*=*name*) *astδ*

definition *enabled* :: *name* ⇒ *state* ⇒ *bool*
where *enabled name s* ≡
case lookup-operator name of
Some (-,*pres*,*effs*,-) ⇒
s∈*subsuming-states* (*map-of pres*)
∧ *s*∈*subsuming-states* (*map-of* (*implicit-pres effs*))
| *None* ⇒ *False*

definition *eff-enabled* :: *state* ⇒ *ast-effect* ⇒ *bool* **where**
eff-enabled s ≡ λ(*pres*,-,-,-). *s*∈*subsuming-states* (*map-of pres*)

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

fun *path-to* **where**
path-to s [] *s*′ ⟷ *s*′=*s*

| *path-to s* (π#πs) *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*
using *wf-initial*
unfolding *valid-states-def Dom-def I-def range-of-var-def*
by (*auto split*:*if-splits*)

lemma *lookup-operator-wf*:
assumes *lookup-operator name* = *Some* π
shows *wf-operator* π *fst* π = *name*
proof −
obtain *name' pres effs cost* **where** [*simp*]: π=(*name'*,*pres*,*effs*,*cost*) **by** (*cases* π)
hence [*simp*]: *name'*=*name* **and** *IN-AST*: (*name*,*pres*,*effs*,*cost*) $\in$ *set ast*δ
using *assms*
unfolding *lookup-operator-def*
apply −
apply (*metis* (*mono-tags*, *lifting*) *case-prodD find-Some-iff*)
by (*metis* (*mono-tags*, *lifting*) *case-prodD find-Some-iff nth-mem*)

from *IN-AST* **show** *WF*: *wf-operator* π *fst* π = *name*
unfolding *enabled-def* **using** *wf-operators* **by** *auto*
qed

lemma *execute-preserves-valid*:
assumes *s*$\in$*valid-states*
assumes *enabled name s*
shows *execute name s* $\in$ *valid-states*
proof −
from ‹*enabled name s*› **obtain** *name' pres effs cost* **where**
[*simp*]: *lookup-operator name* = *Some* (*name'*,*pres*,*effs*,*cost*)
unfolding *enabled-def* **by** (*auto split*: *option.splits*)
from *lookup-operator-wf*[*OF this*] **have** *WF*: *wf-operator* (*name*,*pres*,*effs*,*cost*) **by** *simp*

have *X1*: *s* ++ *m* $\in$ *valid-states* **if** $\forall\, x\ v.\ m\ x$ = *Some v* $\longrightarrow$ *x*<*numVars* $\wedge$ *v*<*numVals x* **for** *m*
using *that* ‹*s*$\in$*valid-states*›

by (*auto*
simp: *valid-states-def Dom-def range-of-var-def map-add-def dom-def*
split: *option.splits*)

have *X2*: *x*<*numVars v*<*numVals x*
if *map-of* (*map* (λ(-, *x*, -, *y*). (*x*, *y*)) (*filter* (*eff-enabled s*) *effs*)) *x* = *Some v*
for *x v*
proof −
from *that* **obtain** *epres vp* **where** (*epres*,*x*,*vp*,*v*) ∈ *set effs*
by (*auto dest*!: *map-of-SomeD*)
with *WF* **show** *x*<*numVars v*<*numVals x*
unfolding *wf-operator-def* **by** *auto*
qed

show *?thesis*
using *assms*
unfolding *enabled-def execute-def*
by (*auto intro*!: *X1 X2*)
qed

lemma *path-to-pres-valid*:
assumes *s*∈*valid-states*
assumes *path-to s* π*s s*′
shows *s*′∈*valid-states*
using *assms*
by (*induction s* π*s s*′ *rule*: *path-to.induct*) (*auto simp*: *execute-preserves-valid*)

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 ⋀*k v*. (*k*,*v*)∈*set l* ⟹ *m k* = *Some v*
shows *map-of l* ⊆$_m$ *m*
using *assms*
by (*metis* (*no-types*, *opaque-lifting*) *domIff map-le-def map-of-SomeD not-Some-eq*)

lemma [*simp*]: *fst* ∘ (λ(*a*, *b*, *c*, *d*). (*f a b c d*, *g a b c d*)) = (λ(*a*,*b*,*c*,*d*). *f a b c d*)
by *auto*

lemma *map-mp*: $m \subseteq_m m' \Longrightarrow m\ k = Some\ v \Longrightarrow m'\ k = Some\ v$
by (*auto simp*: *map-le-def dom-def*)

lemma *map-add-map-of-fold*:
fixes *ps* **and** $m :: 'a \rightharpoonup 'b$
assumes *distinct* (*map fst ps*)
shows m ++ *map-of ps* = *fold* (λ(*k*, *v*) *m*. *m*(*k* ↦ *v*)) *ps m*
proof −
have *X1*: (*fold* (λ(*k*, *v*) *m*. *m*(*k* ↦ *v*)) *ps m*)(*a* ↦ *b*)
= *fold* (λ(*k*, *v*) *m*. *m*(*k* ↦ *v*)) *ps* (*m*(*a* ↦ *b*))
if *a* ∉ *fst* ‘ *set ps*
for *a b ps* **and** $m :: 'a \rightharpoonup 'b$
using *that*
by (*induction ps arbitrary*: *m*) (*auto simp*: *fun-upd-twist*)

show *?thesis*
using *assms*
by (*induction ps arbitrary*: *m*) (*auto simp*: *X1*)
qed

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 ≡ λ(*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 −
have *match-pres pres s* ⟷ *map-of pres* ⊆$_m$ *s*
unfolding *match-pres-def match-pre-def*
apply (*auto split*: *prod.splits*)
using *map-le-def map-of-SomeD* **apply** *fastforce*
by (*metis* (*full-types*) *D domIff map-le-def map-of-eq-Some-iff option.simps*(*3*))

with *assms* **show** *?thesis*
unfolding *subsuming-states-def*
by *auto*
qed

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 −
    from assms show ?thesis
      unfolding subsuming-states-def
      unfolding match-implicit-pres-def implicit-pres-def
      apply (auto
          split: prod.splits option.splits
          simp: distinct-map-filter
          intro!: map-of-leI)
      apply (force simp: distinct-map-filter split: prod.split elim: map-mp)
      done
  qed

  lemma enabled-opr'-correct:
    assumes V: s∈valid-states
    assumes lookup-operator name = Some π
    shows enabled-opr' π s ⟷ enabled name s
    using lookup-operator-wf[OF assms(2)] assms
    unfolding enabled-opr'-def enabled-def wf-operator-def
    by (auto
        simp: match-pres-correct[OF - V] match-implicit-pres-correct[OF - V]
        simp: wf-partial-state-def
        split: option.split
        )

  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
    using assms
    unfolding eff-enabled'-def eff-enabled-def wf-partial-state-def
    by (auto simp: match-pres-correct)


  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 (cases π)
    case [simp]: (fields name pres effs)

    have [simp]: filter (eff-enabled' s) effs = filter (eff-enabled s) effs
      apply (rule filter-cong[OF refl])
      apply (rule eff-enabled'-correct[OF V])
      using lookup-operator-wf[OF LO]
      unfolding wf-operator-def by auto

    have X1: distinct (map fst (map (λ(-, x, -, y). (x, y)) (filter (eff-enabled s)
effs)))
```

```
    using lookup-operator-wf[OF LO]
    unfolding wf-operator-def
    by (auto simp: distinct-map-filter)

  term filter (eff-enabled s) effs

  have [simp]:
    fold (λ(-, x, -, v) s. s(x ↦ v)) l s =
     fold (λ(k, v) m. m(k ↦ v)) (map (λ(-, x, -, y). (x, y)) l) s
    for l :: ast-effect list
    by (induction l arbitrary: s) auto

  show ?thesis
    unfolding execute-opr'-def execute-def using LO
    by (auto simp: map-add-map-of-fold[OF X1])
qed


lemma lookup-operator'-correct:
  lookup-operator' problem name = lookup-operator name
  unfolding lookup-operator'-def lookup-operator-def
  unfolding astδ-def
  by (auto split: prod.split)

lemma enabled'-correct:
  assumes V: s∈valid-states
  shows enabled' problem name s = enabled name s
  unfolding enabled'-def
  using enabled-opr'-correct[OF V]
  by (auto split: option.splits simp: enabled-def lookup-operator'-correct)

lemma execute'-correct:
  assumes V: s∈valid-states
  assumes enabled name s
  shows execute' problem name s = execute name s
  unfolding execute'-def
  using execute-opr'-correct[OF V] ‹enabled name s›
  by (auto split: option.splits simp: enabled-def lookup-operator'-correct)



end

context ast-problem
begin

  fun simulate-plan :: plan ⇒ state ⇀ state where
    simulate-plan [] s = Some s
  | simulate-plan (π#πs) 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'
    by (induction s πs s' rule: path-to.induct) auto

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

  lemma check-plan-correct: check-plan πs ⟷ valid-plan πs
    unfolding check-plan-def valid-plan-def
    by (auto split: option.split simp: simulate-plan-correct[symmetric])

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
  )
  by (auto simp: enabled'-def execute'-def split: option.split)


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
    using assms
    apply (induction πs s rule: simulate-plan.induct)
    apply (auto simp: enabled'-correct execute'-correct execute-preserves-valid)
    done

  lemma simulate-plan'-correct-paper:
    assumes s∈valid-states
    shows simulate-plan' problem πs s = Some s'
        ⟷ path-to s πs s'
    using simulate-plan'-correct[OF assms] simulate-plan-correct by simp


  lemma initial-state'-correct:
    initial-state' problem = I
    unfolding initial-state'-def I-def by (auto simp: Let-def)

  lemma check-plan'-correct:
    check-plan' problem πs = check-plan πs
  proof −
    have D: distinct (map fst astG) using wf-goal unfolding wf-partial-state-def
by auto

    have S'V: s'∈valid-states if simulate-plan πs I = Some s' for s'
    using that by (auto simp: simulate-plan-correct path-to-pres-valid[OF I-valid])

    show ?thesis
      unfolding check-plan'-def check-plan-def
      by (auto
          split: option.splits
          simp: initial-state'-correct simulate-plan'-correct[OF I-valid]
          simp: match-pres-correct[OF D S'V] G-def
          )
  qed
```

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* ()
⟷ *ast-problem.well-formed problem* ∧ *ast-problem.valid-plan problem πs*
proof −
{
assume *ast-problem.well-formed problem*
then interpret *wf-ast-problem* **by** *unfold-locales*

from *check-plan′-correct check-plan-correct*
have *check-plan′ problem πs* = *valid-plan πs* **by** *simp*
}
then show *?thesis*
unfolding *verify-plan-def*
by *auto*
qed

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*)
unfolding *index-by-def*
using *assms*
by (*auto simp*: *o-def*)

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

3.2 Abstract Syntax

3.2.1 Generic Entities

type-synonym *name* = *string*

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

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

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

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

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

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

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

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

Variables are identified by their names.

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

Objects and constants are identified by their names

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

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

3.2.2 Domains

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

datatype *ast-action-schema* = *Action-Schema*
(*name*: *name*)
(*parameters*: (*variable* × *type*) *list*)
(*precondition*: *term atom formula*)
(*effect*: *term ast-effect*)

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

datatype *predicate-decl* = *PredDecl*
(*pred*: *predicate*)
(*argTs*: *type list*)

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

datatype *ast-domain* = *Domain*
(*types*: (*name* × *name*) *list*) — (*type*, *supertype*) declarations.
(*predicates*: *predicate-decl list*)
(*consts*: (*object* × *type*) *list*)
(*actions*: *ast-action-schema list*)

3.2.3 Problems

A fact is a predicate applied to objects.

type-synonym *fact* = *predicate* × *object list*

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

datatype *ast-problem* = *Problem*

(*domain*: *ast-domain*)
(*objects*: (*object* × *type*) *list*)
(*init*: *object atom formula list*)
(*goal*: *object atom formula*)

3.2.4 Plans

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

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

3.2.5 Ground Actions

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

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

3.3 Closed-World Assumption, Equality, and Negation

Discriminator for atomic predicate formulas.

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

The world model is a set of (atomic) formulas

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

It is basic, if it only contains atoms

definition *wm-basic M* ≡ ∀ *a*∈*M*. *is-predAtom a*

A valuation extracted from the atoms of the world model

definition *valuation* :: *world-model* ⇒ *object atom valuation*
where *valuation M* ≡ λ*predAtm p xs* ⇒ *Atom* (*predAtm p xs*) ∈ *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 ∪ {¬(*Atom* (*predAtm p as*)) | *p as*. *Atom* (*predAtm p as*) ∉ *M*}
∪ {*Atom* (*Eq a a*) | *a*. *True*} ∪ {¬(*Atom* (*Eq a b*)) | *a b*. *a*≠*b*}

definition *close-neg M* ≡ *M* ∪ {¬(*Atom a*) | *a*. *Atom a* ∉ *M*}

lemma *wm-basic M* $\Longrightarrow$ *close-world M = close-neg* (*M* $\cup$ {*Atom* (*Eq a a*) | *a. True*})
unfolding *close-world-def close-neg-def wm-basic-def*
apply *clarsimp*
apply (*auto 0 3*)
by (*metis atom.exhaust*)

abbreviation *cw-entailment* (**infix** ‹$^c\models_=$› *53*) **where**
M $^c\models_=$ $\varphi \equiv$ *close-world M* $\models \varphi$

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* $\longleftrightarrow$ (
$\varphi \in M$
$\vee$ ($\exists$ *p as*. φ=$\neg$(*Atom* (*predAtm p as*)) $\wedge$ *Atom* (*predAtm p as*)$\notin M$)
$\vee$ ($\exists$ *a*. φ=*Atom* (*Eq a a*))
$\vee$ ($\exists$ *a b*. φ=$\neg$(*Atom* (*Eq a b*)) $\wedge$ *a*$\neq$*b*)
)
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*
apply −
apply (*auto simp*: *in-close-world-conv valuation-def Ball-def intro*!: *ext split*: *atom.split*)
apply (*metis formula-semantics.simps*(*1*) *formula-semantics.simps*(*3*))
apply (*metis formula-semantics.simps*(*1*) *formula-semantics.simps*(*3*))
by (*metis atom.collapse*(*2*) *formula-semantics.simps*(*1*) *is-predAtm-def*)

lemma *valuation-aux-2*:
assumes *wm-basic M*
shows ($\forall$ *G*$\in$*close-world M*. *valuation M* $\models$ *G*)
using *assms* **unfolding** *wm-basic-def*
by (*force simp*: *in-close-world-conv valuation-def elim*: *is-predAtom.elims*)

lemma *val-imp-close-world*: *valuation M* $\models \varphi \Longrightarrow$ *M* $^c\models_=$ φ
unfolding *entailment-def*

using *valuation-aux-1*
by *blast*

lemma *close-world-imp-val*:
wm-basic M $\Longrightarrow$ *M* $^{c}\models_{=}$ φ $\Longrightarrow$ *valuation M* $\models$ φ
unfolding *entailment-def* **using** *valuation-aux-2* **by** *blast*

Main theorem of this section: If a world model M contains only atoms, its induced valuation satisfies a formula φ if and only if the closure of M entails φ.
Note that there are no syntactic restrictions on φ, in particular, φ may contain negation.

theorem *valuation-iff-close-world*:
assumes *wm-basic M*
shows *valuation M* $\models$ φ $\longleftrightarrow$ *M* $^{c}\models_{=}$ φ
using *assms val-imp-close-world close-world-imp-val* **by** *blast*

3.3.1 Proper Generalization

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

fun *is-STRIPS-fmla* :: *'ent atom formula* $\Rightarrow$ *bool* **where**
is-STRIPS-fmla (*Atom* (*predAtm* - -)) $\longleftrightarrow$ *True*
| *is-STRIPS-fmla* ($\bot$) $\longleftrightarrow$ *True*
| *is-STRIPS-fmla* ($\varphi_1 \wedge \varphi_2$) $\longleftrightarrow$ *is-STRIPS-fmla* φ_1 $\wedge$ *is-STRIPS-fmla* φ_2
| *is-STRIPS-fmla* ($\varphi_1 \vee \varphi_2$) $\longleftrightarrow$ *is-STRIPS-fmla* φ_1 $\wedge$ *is-STRIPS-fmla* φ_2
| *is-STRIPS-fmla* ($\neg\bot$) $\longleftrightarrow$ *True*
| *is-STRIPS-fmla* - $\longleftrightarrow$ *False*

lemma *aux1*: $[\![$*wm-basic M*; *is-STRIPS-fmla* φ; *valuation M* $\models$ φ; $\forall$ *G*$\in$*M*. $\mathcal{A}$ $\models$ *G*$]\!] \Longrightarrow \mathcal{A} \models \varphi$
apply(*induction* φ *rule*: *is-STRIPS-fmla.induct*)
by (*auto simp*: *valuation-def*)

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

lemma *valuation-iff-STRIPS*:
assumes *wm-basic M*
assumes *is-STRIPS-fmla* φ
shows *valuation M* $\models$ φ $\longleftrightarrow$ *M* $\models$ φ
proof −

have *aux1*: $\bigwedge \mathcal{A}.\ [\![valuation\ M \models \varphi;\ \forall\, G{\in}M.\ \mathcal{A} \models G]\!] \Longrightarrow \mathcal{A} \models \varphi$
using *assms* **apply**(*induction* φ *rule*: *is-STRIPS-fmla.induct*)
by (*auto simp*: *valuation-def*)
have *aux2*: $[\![\forall\, \mathcal{A}.\ (\forall\, G{\in}M.\ \mathcal{A} \models G) \longrightarrow \mathcal{A} \models \varphi]\!] \Longrightarrow valuation\ M \models \varphi$
using *assms*
apply(*induction* φ *rule*: *is-STRIPS-fmla.induct*)
apply *simp-all*
apply (*metis in-close-world-conv valuation-aux-2*)
using *in-close-world-conv valuation-aux-2* **apply** *blast*
using *in-close-world-conv valuation-aux-2* **by** *auto*
show *?thesis*
by (*auto simp*: *entailment-def intro*: *aux1 aux2*)
qed

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

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

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′* ⟷ (*M* = *M′*)
| *ground-action-path M* (*α*#*αs*) *M′* ⟷ *M* $^{c}\models_{=}$ *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′* ⟷ (*M* = *M′*)
ground-action-path M (*α*#*αs*) *M′* ⟷ *M* $^{c}\models_{=}$ *precondition α*
∧ (*ground-action-path* (*apply-effect* (*effect α*) *M*) *αs M′*)
by (*auto simp*: *execute-ground-action-def*)

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* ⊆$_m$ *varT′* ⟹ *objT* ⊆$_m$ *objT′* ⟹
ty-term varT objT ⊆$_m$ *ty-term varT′ objT′*
apply (*rule map-leI*)
subgoal for *x v*
apply (*cases x*)
apply (*auto dest*: *map-leD*)
done
done

context *ast-domain* **begin**

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

definition *sig* :: *predicate* ⇀ *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* ≡ *set* (*map subtype-edge* (*types D*))

definition *of-type* :: *type* ⇒ *type* ⇒ *bool* **where**
of-type oT T ≡ *set* (*primitives oT*) ⊆ *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* ⇀ *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*)

fun *wf-pred-atom* :: *predicate* × *'ent list* ⇒ *bool* **where**
wf-pred-atom (*p*,*vs*) ⟷ (
case sig p of
None ⇒ *False*
| *Some Ts* ⇒ *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* ⇒ *bool* **where**
wf-atom (*predAtm p vs*) ⟷ *wf-pred-atom* (*p*,*vs*)
| *wf-atom* (*Eq a b*) ⟷ *ty-ent a* ≠ *None* ∧ *ty-ent b* ≠ *None*

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* (⊥) ⟷ *True*
| *wf-fmla* (*φ1* ∧ *φ2*) ⟷ (*wf-fmla φ1* ∧ *wf-fmla φ2*)
| *wf-fmla* (*φ1* ∨ *φ2*) ⟷ (*wf-fmla φ1* ∧ *wf-fmla φ2*)
| *wf-fmla* (¬*φ*) ⟷ *wf-fmla φ*
| *wf-fmla* (*φ1* → *φ2*) ⟷ (*wf-fmla φ1* ∧ *wf-fmla φ2*)

lemma *wf-fmla φ* = (∀ *a*∈*atoms φ*. *wf-atom a*)
by (*induction φ*) *auto*

Special case for a well-formed atomic predicate formula

fun *wf-fmla-atom* **where**

```
  wf-fmla-atom (Atom (predAtm a vs)) ⟷ wf-pred-atom (a,vs)
| wf-fmla-atom - ⟷ False

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

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

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

end — Context fixing ty-ent


definition constT :: object ⇀ type where
  constT ≡ 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 ⇒ bool where
  wf-action-schema (Action-Schema n params pre eff) ⟷ (
    let
      tyt = ty-term (map-of params) constT
    in
      distinct (map fst params)
    ∧ wf-fmla tyt pre
    ∧ 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) ⟷ set Ts ⊆ 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) ⟷ (∀ T∈set Ts. wf-type T)
```

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

```
definition wf-types ≡ snd`set (types D) ⊆ 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*)
unfolding *objT-def constT-def*
apply (*clarsimp*)
done

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
  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
```

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
    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 ⇒
      action-params-match a args
    ∧ 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)
    ∧ ground-action-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' ᶜ⊨₌ (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
```

Concise definition used in paper:

```
lemma valid-plan πs ≡ ∃M'. plan-action-path I πs M' ∧ M' ᶜ⊨₌ (goal P)
  unfolding valid-plan-def valid-plan-from-def by auto
```

end — Context of *ast-problem*

3.7 Preservation of Well-Formedness

3.7.1 Well-Formed Action Instances

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

context *ast-problem* **begin**

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

```
fun wf-ground-action :: ground-action ⇒ bool where
  wf-ground-action (Ground-Action 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, opaque-lifting)
      Image-mono contra-subsetD order-refl rtrancl-image-idem)

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

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

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

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

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

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

context
fixes *Q* **and** *f* :: *variable* $\Rightarrow$ *object*
assumes *INST*: *is-of-type Q x T* $\Longrightarrow$ *is-of-type objT* (*f x*) *T*
begin

lemma *is-of-type-var-conv*: *is-of-type* (*ty-term Q objT*) (*term.VAR x*) *T* $\longleftrightarrow$ *is-of-type Q x T*
unfolding *is-of-type-def* **by** (*auto*)

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

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

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

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

lemma *wf-inst-atom*:

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

qed

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

lemma wf-inst-effect:
  assumes wf-effect (ty-term Q constT) φ
  shows wf-effect objT ((map-ast-effect o subst-term) 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 (ty-term Q constT) φ
  shows wf-fmla objT ((map-formula o map-atom o subst-term) 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-ground-action (instantiate-action-schema a args)
proof (cases a)
  case [simp]: (Action-Schema name params pre eff)
  have INST:
    is-of-type objT ((the ∘ map-of (zip (map fst params) args)) x) T
```

if *is-of-type* (*map-of params*) *x T* **for** *x T*
using *that*
apply (*rule is-of-type-map-ofE*)
using *assms*
apply (*clarsimp simp*: *Let-def*)
subgoal for *i xT*
unfolding *action-params-match-def*
apply (*subst lookup-zip-idx-eq*[**where** *i*=*i*];
(*clarsimp simp*: *list-all2-lengthD*) *?*)
apply (*frule list-all2-nthD2*[**where** *p*=*i*]; *simp?*)
apply (*auto*
simp: *is-obj-of-type-alt is-of-type-def*
intro: *of-type-trans*
split: *option.splits*)
done
done
then show *?thesis*
using *assms*(*2*) *wf-inst-formula wf-inst-effect*
by (*fastforce split*: *term.splits simp*: *Let-def comp-apply*[*abs-def*])
qed
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⊨₌ *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*#*πs*) *M′* ⟷
plan-action-enabled π M
∧ *plan-action-path* (*execute-plan-action π M*) *πs M′*

by (*auto*
simp: *plan-action-path-def execute-plan-action-def*
execute-ground-action-def plan-action-enabled-def)

end — Context of *ast-problem*

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

The initial world model is well-formed

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

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

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

Execution of plan actions preserves well-formedness

theorem *wf-execute*:
assumes *plan-action-enabled 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 ∈ *set* (*actions D*)
unfolding *resolve-action-schema-def* **by** *auto*

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

theorem *wf-execute-compact-notation*:
plan-action-enabled π *s* $\Longrightarrow$ *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*:
assumes *wf-world-model M* **and** *plan-action-path M* πs *M'*
shows *wf-world-model M'*
using *assms*
by (*induction* πs *arbitrary*: *M*) (*auto intro*: *wf-execute*)

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* ≡ { (*u*,*v*). *v*∈*set* (*succ u*) }
lemma *succ-as-E*: *set* (*succ x*) = *E-of-succ succ* `` {*x*}
unfolding *E-of-succ-def* **by** *auto*

context
fixes *succ* :: *'a* ⇒ *'a list*
begin

private abbreviation (*input*) *E* ≡ *E-of-succ succ*

definition *dfs-reachable D w* ≡
let (*V*,*w*,*brk*) = *while* (λ(*V*,*w*,*brk*). ¬*brk* ∧ *w*≠[]) (λ(*V*,*w*,-).
case w of v#*w* ⇒
if D v then (*V*,*v*#*w*,*True*)
else if v∈*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*$_0$ ≡ *set w*$_0$

definition *dfs-reachable-invar D V W brk* ⟷
W$_0$ ⊆ *W* ∪ *V*
∧ *W* ∪ *V* ⊆ *E** `` *W*$_0$
∧ *E*``*V* ⊆ *W* ∪ *V*
∧ *Collect D* ∩ *V* = {}
∧ (*brk* ⟶ *Collect D* ∩ *E** `` *W*$_0$ ≠ {})

lemma *card-decreases*:
⟦*finite V*; *y* ∉ *V*; *dfs-reachable-invar D V* (*Set.insert y W*) *brk* ⟧
⟹ *card* (*E** `` *W*$_0$ − *Set.insert y V*) < *card* (*E*$^*$ `` *W*$_0$ − *V*)
apply (*rule psubset-card-mono*)
apply (*auto simp*: *dfs-reachable-invar-def*)
done

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

```
  (∀ y ys. x2 ≠ y # ys) ⟷ x2 = [] by (cases x2) auto

lemma dfs-reachable-correct: dfs-reachable D w₀ ⟷ Collect D ∩ E* `` set w₀ ≠
{}
  unfolding dfs-reachable-def
  apply (rule while-rule[where
    P=λ(V,w,brk). dfs-reachable-invar D V (set w) brk ∧ finite V
     and r=measure (λV. card (E* `` (set w₀) − V)) <*lex*> measure length
<*lex*> measure (λTrue⇒0 | False⇒1)
    ])
  subgoal by (auto simp: dfs-reachable-invar-def)
  subgoal
    apply (auto simp: neq-Nil-conv succ-as-E[of succ] split: if-splits)
   by (auto simp: dfs-reachable-invar-def Image-iff intro: rtrancl.rtrancl-into-rtrancl)
  subgoal by (fastforce simp: dfs-reachable-invar-def dest: Image-closed-trancl)
  subgoal by blast
  subgoal by (auto simp: neq-Nil-conv card-decreases)
  done

end

definition tab-succ l ≡ 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) = (∃ z. xo = Some
z ∧ f z = y)
  by (auto simp add: map-option-case split: option.split)


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

end

lemma finite-imp-finite-dfs-reachable:
  ⟦finite E; finite S⟧ ⟹ finite (E* ``S)
```

apply (*rule finite-subset*[**where** *B*=*S* ∪ (*Relation.Domain E* ∪ *Relation.Range E*)])
apply (*auto simp*: *intro*: *finite-Domain finite-Range elim*: *rtranclE*)
done

lemma *dfs-reachable-tab-succ-correct*: *dfs-reachable* (*tab-succ l*) *D* $vs_0 \longleftrightarrow$ *Collect D* ∩ (*set l*)* ‘‘*set* $vs_0 \neq \{\}$
apply (*subst dfs-reachable-correct*)
by (*simp-all add*: *tab-succ-correct finite-imp-finite-dfs-reachable*)

4.2 Implementation Refinements

4.2.1 Of-Type

definition *of-type-impl G oT T* ≡ (∀ *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*
by (*cases t*) *auto*

lemma *ty-term′-correct*[*simp*]: *ty-term′ varT objT* = *ty-term varT* (*Mapping.lookup objT*)
using *ty-term′-correct-aux* **by** *auto*

context *ast-domain* **begin**

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

lemma *of-type-refine1*: *of-type oT T* $\longleftrightarrow$ (∀ *pt*∈*set* (*primitives oT*). *of-type1 pt T*)
unfolding *of-type-def of-type1-def* **by** *auto*

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*)))
by (*simp add*: *tab-succ-correct subtype-rel-def*)

lemma *of-type1-impl*: *of-type1 pt T* $\longleftrightarrow$ *dfs-reachable* (*tab-succ* (*map subtype-edge* (*types D*))) ((=)*pt*) (*primitives T*)
by (*simp add*: *subtype-rel-impl of-type1-def dfs-reachable-tab-succ-correct tab-succ-correct*)

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

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

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

Lifting the subtype-graph through wf-checker

context
fixes *ty-ent* :: *'ent* ⇀ *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*
unfolding *is-of-type'-def is-of-type-def of-type-impl-correct* **..**

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

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

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

lemma *wf-fmla'-correct*: *wf-fmla' STG φ* ⟷ *wf-fmla ty-ent φ*
by (*induction φ rule*: *wf-fmla.induct*) (*auto simp*: *wf-atom'-correct*)

fun *wf-fmla-atom1'* **where**

```
    wf-fmla-atom1' stg (Atom (predAtm p vs)) ⟷ wf-pred-atom' stg (p,vs)
  | wf-fmla-atom1' stg - ⟷ False

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

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

  lemma wf-effect'-correct: wf-effect' STG e = wf-effect ty-ent e
    by (cases e) (auto simp: wf-fmla-atom1'-correct)

end — Context fixing ty-ent

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

lemma wf-action-schema'-correct: wf-action-schema' STG mp-constT s = wf-action-schema
s
  by (cases s) (auto simp: wf-fmla'-correct wf-effect'-correct)

definition wf-domain' :: - ⇒ - ⇒ bool where
  wf-domain' stg conT ≡
    wf-types
  ∧ distinct (map (predicate-decl.pred) (predicates D))
  ∧ (∀ p∈set (predicates D). wf-predicate-decl p)
  ∧ distinct (map fst (consts D))
  ∧ (∀ (n,T)∈set (consts D). wf-type T)
  ∧ distinct (map ast-action-schema.name (actions D))
  ∧ (∀ a∈set (actions D). wf-action-schema' stg conT a)


lemma wf-domain'-correct: wf-domain' STG mp-constT = wf-domain
  unfolding wf-domain-def wf-domain'-def
  by (auto simp: wf-action-schema'-correct)


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 (λadd s. Set.insert add s) a
        (fold (λdel s. Set.remove del s) d s)

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

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

end — Context of *ast-domain*

4.2.3 Well-Formedness

context *ast-problem* **begin**

We start by defining a mapping from objects to types. The container frame-

work will generate efficient, red-black tree based code for that later.

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

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

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

We refine the typecheck to use the mapping

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

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

We refine the well-formedness checks to use the mapping

```
definition wf-fact' :: objT ⇒ - ⇒ fact ⇒ 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
  by (auto simp: wf-fact'-def wf-fact-def wf-pred-atom'-correct[abs-def])


definition wf-fmla-atom2' mp stg f
  = (case f of formula.Atom (predAtm p vs) ⇒ (wf-fact' mp stg (p,vs)) | - ⇒
False)

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

definition wf-problem' stg conT mp ≡
    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
unfolding *wf-problem-def wf-problem'-def wf-world-model-def*
by (*auto simp*: *wf-domain'-correct wf-fmla'-correct*)

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

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

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

```
  definition en-exE :: plan-action ⇒ world-model ⇒ -+world-model where
    en-exE ≡ λ(PAction n args) ⇒ λs. 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
    apply (cases a)
    using assms holds-for-wf-fmlas wf-domain
    unfolding plan-action-enabled-def execute-plan-action-def
      and execute-ground-action-def en-exE-def wf-domain-def
    by (auto
        split: option.splits
        simp: resolve-action-schemaE-def return-iff)
```

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

```
  abbreviation action-params-match2 stg mp a args
```

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

Combination of the two refinement lemmas

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

lemma (in wf-ast-problem) en-exE2-return-iff-compact-notation:
  ⟦wm-basic s⟧ ⟹
    en-exE2 STG 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
```

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(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-ground-action-def execute-plan-action-def)
qed
```

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

```
fun valid-plan-fromE
  :: - ⇒ (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)
  using wf-problem wf-domain
  unfolding wf-problem-def wf-domain-def
  apply (cases a)
  apply (clarsimp
    split: option.splits
    simp: wf-fmla-atom-alt execute-plan-action-def wm-basic-def
    simp: execute-ground-action-def)
  subgoal for n args schema fmla
    apply (cases effect (instantiate-action-schema schema args); simp)
    by (metis ground-action.sel(2) ast-domain.wf-effect.simps
          ast-domain.wf-fmla-atom-alt resolve-action-wf
          wf-ground-action.elims(2) wf-instantiate-action-schema)
  done
```

Refinement lemma for our plan checking algorithm

lemma (**in** *wf-ast-problem*) *valid-plan-fromE-return-iff*[*return-iff*]:
assumes *wm-basic s*
shows *valid-plan-fromE STG mp-objT k s πs = Inr* () ⟷ *valid-plan-from s πs*
using *assms* **unfolding** *valid-plan-from1-refine*
proof (*induction stg≡STG 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*

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 (λ*x*. *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*)
unfolding *check-all-list-def*
by (*induction l*) (*auto*)

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

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 −
interpret *ast-domain D* .
show *?thesis*
unfolding *check-wf-domain-def wf-domain*′*-def*
by (*auto simp*: *return-iff*)
qed

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 −
interpret *ast-problem P* .
show *?thesis*
unfolding *check-wf-problem-def wf-problem′-def*
by (*auto simp*: *return-iff*)
qed

definition *check-plan P πs* ≡ *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 −
interpret *ast-problem P* .
show *?thesis*
unfolding *check-plan-def*
by (*auto*
simp: *return-iff wf-world-model-def wf-fmla-atom-alt I-def wf-problem-def isOK-iff*
simp: *wf-problem′-correct ast-problem.I-def ast-problem.valid-plan-def wm-basic-def*
)
qed

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*
  apply (*induction l rule*: *no-stutter.induct*)
  apply (*auto simp*:)
  done

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

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

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

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

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

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

lemma *sound-opr-alt*:

```
  sound-opr opr f =
    ((∀ s. s ⊨= (precondition opr) ⟶
       (∃ s'. f s = (Some s')
            ∧ (∀ atm. is-predAtom atm ∧ atm ∉ set(dels (effect opr)) ∧ s ⊨= atm
⟶ s' ⊨= atm)
              ∧ (∀ atm. is-predAtom atm ∧ atm ∉ set (adds (effect opr)) ∧ s ⊨=
Not atm ⟶ s' ⊨= Not atm)
             ∧ (∀ atm. atm ∈ set(adds (effect opr)) ⟶ s' ⊨= atm)
               ∧ (∀ fmla. fmla ∈ set (dels (effect opr)) ∧ fmla ∉ set(adds (effect
opr)) ⟶ s' ⊨= (Not fmla))
             ∧ (∀ a b. s ⊨= Atom (Eq a b) ⟶ s' ⊨= Atom (Eq a b))
            ∧ (∀ a b. s ⊨= Not (Atom (Eq a b)) ⟶ s' ⊨= Not (Atom (Eq a b)))
             ))
      ∧ (∀ fmla∈set(adds (effect opr)). is-predAtom fmla))
  by (cases (opr,f) rule: sound-opr.cases) auto
```

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

```
definition sound-system
  :: ground-action set
    ⇒ world-model
    ⇒ state
    ⇒ (ground-action ⇒ action)
    ⇒ bool
  where
  sound-system Σ M0 s0 f ⟷
    ((∀ fmla∈close-world M0. s0  ⊨= fmla)
    ∧ wm-basic M0
    ∧ (∀ α∈Σ. sound-opr α (f α)))
```

Composing two actions

```
definition compose-action :: action ⇒ action ⇒ action where
  compose-action f1 f2 x = (case f2 x of Some y ⇒ f1 y | None ⇒ None)
```

Composing a list of actions

```
definition compose-actions :: action list ⇒ action where
  compose-actions fs ≡ 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' ⟹ 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

Soundness Theorem in Lifschitz's paper [4].

theorem *STRIPS-sema-sound*:
assumes *sound-system* $\Sigma\ M_0\ s_0\ f$
— For a sound system Σ
assumes *set* $\alpha s \subseteq \Sigma$
— And a plan αs
assumes *ground-action-path* $M_0\ \alpha s\ M'$
— Which is accepted by the system, yielding result M' (called $R(\alpha s)$ in Lifschitz's paper [4].)
obtains s'
— We have that $f(\alpha s)$ is applicable in initial state, yielding state s' (called $f_{\alpha s}(s_0)$ in Lifschitz's paper [4].)
where *compose-actions* (*map f* αs) $s_0 = Some\ s'$
— The result world model M' is satisfied in state s'
and $\forall fmla{\in}close\text{-}world\ M'.\ s' \models_= fmla$
proof −
have (*valuation* $M' \models fmla$) **if** *wm-basic* M' *fmla*$\in M'$ **for** *fmla*
using *that* **apply** (*induction fmla*)
by (*auto simp*: *valuation-def wm-basic-def split*: *atom.split*)
have $\exists s'.$ *compose-actions* (*map f* αs) $s_0 = Some\ s' \wedge (\forall fmla{\in}close\text{-}world\ M'.\ s' \models_= fmla)$
using *assms*
proof(*induction* αs *arbitrary*: $s_0\ M_0$)
case *Nil*
then show *?case* **by** (*auto simp add*: *close-world-def compose-action-def sound-system-def*)
next
case *ass*: (*Cons* $\alpha\ \alpha s$)
then obtain *pre add del* **where** *a*: α = *Ground-Action pre* (*Effect add del*)
using *ground-action.exhaust ast-effect.exhaust* **by** *metis*
let $?M_1 =$ *execute-ground-action* $\alpha\ M_0$
have *close-world* $M_0 \models$ *precondition* α
using *ass*(*4*)
by *auto*
moreover have *s0-ent-cwM0*: $\forall fmla{\in}(close\text{-}world\ M_0).$ *close-eq* $s_0 \models fmla$
using *ass*(*2*)
unfolding *sound-system-def*
by *auto*
ultimately have *s0-ent-alpha-precond*: *close-eq* $s_0 \models$ *precondition* α
unfolding *entailment-def*
by *auto*
then obtain s_1 **where** *s1*: (*f* α) $s_0 = Some\ s_1$
($\forall atm.$ *is-predAtom atm* $\longrightarrow$ *atm* $\notin$ *set*(*dels* (*effect* α))
$\longrightarrow$ *close-eq* $s_0 \models atm$

$\longrightarrow$ *close-eq* $s_1 \models$ *atm*)
(∀ *fmla*. *fmla* ∈ *set*(*adds* (*effect* α))
$\longrightarrow$ *close-eq* $s_1 \models$ *fmla*)
(∀ *atm*. *is-predAtom atm* ∧ *atm* ∉ *set* (*adds* (*effect* α)) ∧ *close-eq* $s_0 \models$ *Not atm* $\longrightarrow$ *close-eq* $s_1 \models$ *Not atm*)
(∀ *fmla*. *fmla* ∈ *set* (*dels* (*effect* α)) ∧ *fmla* ∉ *set*(*adds* (*effect* α)) $\longrightarrow$ *close-eq* $s_1 \models$ (*Not fmla*))
(∀ *a b*. *close-eq* $s_0 \models$ *Atom* (*Eq a b*) $\longrightarrow$ *close-eq* $s_1 \models$ *Atom* (*Eq a b*))
(∀ *a b*. *close-eq* $s_0 \models$ *Not* (*Atom* (*Eq a b*)) $\longrightarrow$ *close-eq* $s_1 \models$ *Not* (*Atom* (*Eq a b*)))
using *ass*(*2*−*4*)
unfolding *sound-system-def sound-opr-alt* **by** *force*
have *close-eq* $s_1 \models$ *fmla* **if** *fmla*∈*close-world* $?M_1$ **for** *fmla*
using *ass*(*2*)
using *that s1 s0-ent-cwM0*
unfolding *sound-system-def execute-ground-action-def wm-basic-def*
apply (*auto simp*: *in-close-world-conv*)
subgoal
by (*metis* (*no-types*, *lifting*) *DiffE UnE a apply-effect.simps ground-action.sel*(*2*) *ast-effect.sel*(*1*) *ast-effect.sel*(*2*) *close-world-extensive subsetCE*)
subgoal
by (*metis Diff-iff Un-iff a ground-action.sel*(*2*) *ast-domain.apply-effect.simps ast-domain.close-eq-predAtm ast-effect.sel*(*1*) *ast-effect.sel*(*2*) *formula-semantics.simps*(*1*) *formula-semantics.simps*(*3*) *in-close-world-conv is-predAtom.simps*(*1*))
done
moreover have (∀ *atm*. *fmla* ≠ *formula.Atom atm*) $\longrightarrow$ *s* $\models$ *fmla* **if** *fmla*∈$?M_1$ **for** *fmla s*
proof−
have *alpha*: (∀ *s*.∀ *fmla*∈*set*(*adds* (*effect* α)). ¬ *is-predAtom fmla* $\longrightarrow$ *s* $\models$ *fmla*)
using *ass*(*2*,*3*)
unfolding *sound-system-def ast-domain.sound-opr-alt*
by *auto*
then show *?thesis*
using *that*
unfolding *a execute-ground-action-def*
using *ass.prems*(*1*)[*unfolded sound-system-def*]
by(*cases fmla*; *fastforce simp*: *wm-basic-def*)

qed
moreover have (∀ *opr*∈Σ. *sound-opr opr* (*f opr*))
using *ass*(*2*) **unfolding** *sound-system-def*
by (*auto simp add*:)
moreover have *wm-basic* $?M_1$
using *ass*(*2*,*3*)
unfolding *sound-system-def execute-ground-action-def*
thm *sound-opr.cases*
apply (*cases* (α,*f* α) *rule*: *sound-opr.cases*)
apply (*auto simp*: *wm-basic-def*)

```
    done
  ultimately have sound-system Σ ?M1 s1 f
    unfolding sound-system-def
    by (auto simp: wm-basic-def)
  from ass.IH[OF this] ass.prems obtain s' where
    compose-actions (map f αs) s1 = Some s' ∧ (∀ a∈close-world M'. s' ⊨= a)
    by auto
  thus ?case by (auto simp: s1(1))
 qed
 with that show ?thesis by blast
qed
```

More compact notation of the soundness theorem.

```
 theorem STRIPS-sema-sound-compact-version:
  sound-system Σ M0 s0 f ⟹ set αs ⊆ Σ
  ⟹ ground-action-path M0 αs M'
  ⟹ ∃ s'. compose-actions (map f αs) s0 = Some s'
       ∧ (∀ fmla∈close-world M'. s' ⊨= fmla)
  using STRIPS-sema-sound by metis
```

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 ⇒ world-model
  where state-to-wm s = ({formula.Atom (predAtm p xs) | p xs. s (p,xs)})
 definition wm-to-state :: world-model ⇒ state
  where wm-to-state M = (λ(p,xs). (formula.Atom (predAtm p xs)) ∈ M)


 lemma wm-to-state-eq[simp]: wm-to-state M (p, as) ⟷ Atom (predAtm p as)
∈ M
  by (auto simp: wm-to-state-def)




 lemma wm-to-state-inv[simp]: wm-to-state (state-to-wm s) = s
  by (auto simp: wm-to-state-def
    state-to-wm-def image-def)
```

Mapping AST action instances to actions

```
 definition pddl-opr-to-act g-opr s = (
  let M = state-to-wm s in
  if (wm-to-state (close-world M)) ⊨= (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 ⊨= (*formula.Atom atm*)
⟷ ((*formula.Atom atm*) ∈ *close-eq-M* (*state-to-wm s*))
by (*auto simp*: *wm-to-state-def*
state-to-wm-def image-def close-eq-M-def close-eq-def split: *atom.splits*)

lemma *atom-in-wm-2-eq*:
close-eq (*wm-to-state M*) ⊨ (*formula.Atom atm*)
⟷ ((*formula.Atom atm*) ∈ *close-eq-M M*)
by (*auto simp*: *wm-to-state-def*
state-to-wm-def image-def close-eq-def close-eq-M-def split:*atom.splits*)

lemma *not-dels-preserved*:
assumes *f* ∉ (*set d*) *f* ∈ *M*
shows *f* ∈ *apply-effect* (*Effect a d*) *M*
using *assms*
by *auto*

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

lemma *dels-unsatisfied*:
assumes *f* ∈ (*set d*) *f* ∉ *set a*
shows *f* ∉ *apply-effect* (*Effect a d*) *M*
using *assms*
by *auto*

lemma *dels-unsatisfied-2*:
assumes *f* ∈ *set* (*dels eff*) *f* ∉ *set* (*adds eff*)
shows *f* ∉ *apply-effect eff M*
using *assms*
by (*cases eff*; *auto*)

lemma *wf-fmla-atm-is-atom*: *wf-fmla-atom objT f* ⟹ *is-predAtom f*
by (*cases f rule*: *wf-fmla-atom.cases*) *auto*

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

by (*cases effs*) (*auto simp*: *wf-fmla-atom-alt*)

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

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

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

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

lemma *wf-pddl-ground-act-is-sound-opr*:

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

lemma *wf-action-schema-sound-inst*:
assumes *action-params-match act args wf-action-schema act*
shows *sound-opr*
(*instantiate-action-schema act args*)
((*pddl-opr-to-act* (*instantiate-action-schema act args*)))
using
wf-pddl-ground-act-is-sound-opr[
OF 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*∈*M* ⟹ *wf-world-model M* ⟹ *is-predAtom f*
using *wf-fmla-atm-is-atom*
unfolding *wf-world-model-def*
by *auto*

lemma *wm-to-state-works-for-wf-wm-closed*:
wf-world-model M ⟹ *fmla*∈*close-world M* ⟹ *close-eq* (*wm-to-state M*) ⊨ *fmla*
apply (*cases fmla rule*: *wf-fmla-atom.cases*)
by (*auto simp*: *wf-world-model-def close-eq-def wm-to-state-def close-world-def*)

lemma *wm-to-state-works-for-wf-wm*: *wf-world-model M* ⟹ *fmla*∈*M* ⟹ *close-eq* (*wm-to-state M*) ⊨ *fmla*
apply (*cases fmla rule*: *wf-fmla-atom.cases*)
by (*auto simp*: *wf-world-model-def close-eq-def wm-to-state-def*)

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

lemma *wf-wm-imp-basic*: *wf-world-model M* $\Longrightarrow$ *wm-basic M*
by (*auto simp*: *wf-world-model-def wm-basic-def wf-fmla-atm-is-atom*)

theorem *wf-plan-sound-system*:
assumes $\forall \pi \in$ *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))
unfolding *sound-system-def*
proof(*intro conjI ballI*)
show *close-eq*(*wm-to-state I*) $\models x$ **if** $x \in$ *close-world I* **for** x
using *that*[*unfolded in-close-world-conv*]
wm-to-state-works-for-I-closed wm-to-state-works-for-wf-wm
by (*auto simp*: *wf-I*)

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

show *sound-opr* α (*pddl-opr-to-act* α) **if** $\alpha \in$ *set* (*map resolve-instantiate* πs)
for α
using *that*
using *wf-plan-act-is-sound'* *assms*
by *auto*
qed

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* αs s_0 = *Some* $s' \wedge (\forall \varphi \in$ *close-world M*. $s' \models_= \varphi)$
apply (*rule STRIPS-sema-sound*)
apply (*rule wf-plan-sound-system*)
using *assms*
unfolding *plan-action-path-def*
by (*auto simp add*: *image-def*)

end — Context of *wf-ast-problem*

end

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.