NaDeA (Natural Deduction Assistant) — Formalization in Isabelle

Jørgen Villadsen, Alexander Birch Jensen & Anders Schlichtkrull — DTU Compute — Denmark

```
theory NaDeA imports Main begin

type-synonym id = "char list"

datatype tm = Var nat | Fun id "tm list"

datatype fm = Falsity | Pre id "tm list" | Imp fm fm | Dis fm fm | Con fm fm | Exi fm | Uni fm

primrec
  semantics-term :: "(nat ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ tm ⇒ 'a"
and
  semantics-list :: "(nat ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ tm list ⇒ 'a list"
where
  "semantics-term e f (Var n) = e n" |
  "semantics-term e f (Fun i l) = f i (semantics-list e f l)" |
  "semantics-list e f [] = []" |
  "semantics-list e f (t # l) = semantics-term e f t # semantics-list e f l"

primrec
  semantics :: "(nat ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ bool) ⇒ fm ⇒ bool"
where
  "semantics e f g Falsity = False" |
  "semantics e f g (Pre i l) = g i (semantics-list e f l)" |
  "semantics e f g (Imp p q) = (if semantics e f g p then semantics e f g q else True)" |
  "semantics e f g (Dis p q) = (if semantics e f g p then True else semantics e f g q)" |
  "semantics e f g (Con p q) = (if semantics e f g p then semantics e f g q else False)" |
  "semantics e f g (Exi p) = (∃x. semantics (λn. if n = 0 then x else e (n − 1)) f g p)" |
  "semantics e f g (Uni p) = (∀x. semantics (λn. if n = 0 then x else e (n − 1)) f g p)"

primrec
  member :: "fm ⇒ fm list ⇒ bool"
where
  "member p [] = False" |
  "member p (q # z) = (if p = q then True else member p z)"

primrec
  new-term :: "id ⇒ tm ⇒ bool"
and
  new-list :: "id ⇒ tm list ⇒ bool"
where
  "new-term c (Var n) = True" |
  "new-term c (Fun i l) = (if i = c then False else new-list c l)" |
  "new-list c [] = True" |
  "new-list c (t # l) = (if new-term c t then new-list c l else False)"

primrec
  new :: "id ⇒ fm ⇒ bool"
where
  "new c Falsity = True" |
  "new c (Pre i l) = new-list c l" |
  "new c (Imp p q) = (if new c p then new c q else False)" |
```

```
  "new c (Dis p q) = (if new c p then new c q else False)" |
  "new c (Con p q) = (if new c p then new c q else False)" |
  "new c (Exi p) = new c p" |
  "new c (Uni p) = new c p"

primrec
  news :: "id ⇒ fm list ⇒ bool"
where
  "news c [] = True" |
  "news c (p # z) = (if new c p then news c z else False)"

primrec
  inc-term :: "tm ⇒ tm"
and
  inc-list :: "tm list ⇒ tm list"
where
  "inc-term (Var n) = Var (n + 1)" |
  "inc-term (Fun i l) = Fun i (inc-list l)" |
  "inc-list [] = []" |
  "inc-list (t # l) = inc-term t # inc-list l"

primrec
  sub-term :: "nat ⇒ tm ⇒ tm ⇒ tm"
and
  sub-list :: "nat ⇒ tm ⇒ tm list ⇒ tm list"
where
  "sub-term v s (Var n) = (if n < v then Var n else if n = v then s else Var (n − 1))" |
  "sub-term v s (Fun i l) = Fun i (sub-list v s l)" |
  "sub-list v s [] = []" |
  "sub-list v s (t # l) = sub-term v s t # sub-list v s l"

primrec
  sub :: "nat ⇒ tm ⇒ fm ⇒ fm"
where
  "sub v s Falsity = Falsity" |
  "sub v s (Pre i l) = Pre i (sub-list v s l)" |
  "sub v s (Imp p q) = Imp (sub v s p) (sub v s q)" |
  "sub v s (Dis p q) = Dis (sub v s p) (sub v s q)" |
  "sub v s (Con p q) = Con (sub v s p) (sub v s q)" |
  "sub v s (Exi p) = Exi (sub (v + 1) (inc-term s) p)" |
  "sub v s (Uni p) = Uni (sub (v + 1) (inc-term s) p)"

inductive
  OK :: "fm ⇒ fm list ⇒ bool"
where
Assume:
      "member p z ⟹ OK p z" |
Boole:
      "OK Falsity ((Imp p Falsity) # z) ⟹ OK p z" |
Imp-E:
      "OK (Imp p q) z ⟹ OK p z ⟹ OK q z" |
Imp-I:
      "OK q (p # z) ⟹ OK (Imp p q) z" |
Dis-E:
      "OK (Dis p q) z ⟹ OK r (p # z) ⟹ OK r (q # z) ⟹ OK r z" |
Dis-I1:
      "OK p z ⟹ OK (Dis p q) z" |
Dis-I2:
      "OK q z ⟹ OK (Dis p q) z" |
Con-E1:
```

```
    "OK (Con p q) z ⟹ OK p z" |
Con-E2:
    "OK (Con p q) z ⟹ OK q z" |
Con-I:
    "OK p z ⟹ OK q z ⟹ OK (Con p q) z" |
Exi-E:
    "OK (Exi p) z ⟹ OK q ((sub 0 (Fun c []) p) # z) ⟹ news c (p # q # z) ⟹ OK q z" |
Exi-I:
    "OK (sub 0 t p) z ⟹ OK (Exi p) z" |
Uni-E:
    "OK (Uni p) z ⟹ OK (sub 0 t p) z" |
Uni-I:
    "OK (sub 0 (Fun c []) p) z ⟹ news c (p # z) ⟹ OK (Uni p) z"

lemma "OK (Imp (Pre ''A'' []) (Pre ''A'' [])) []" proof (rule Imp-I, rule Assume, simp) qed

lemma "OK (Imp (Pre ''A'' []) (Pre ''A'' [])) []"
proof −
  have "OK (Pre ''A'' []) [(Pre ''A'' [])]" proof (rule Assume) qed simp
  then show "OK (Imp (Pre ''A'' []) (Pre ''A'' [])) []" proof (rule Imp-I) qed
qed

fun
  put :: "(nat ⇒ 'a) ⇒ nat ⇒ 'a ⇒ nat ⇒ 'a"
where
  "put e v x = (λn. if n < v then e n else if n = v then x else e (n − 1))"

lemma "put e 0 x = (λn. if n = 0 then x else e (n − 1))" proof simp qed

lemma increment:
  "semantics-term (put e 0 x) f (inc-term t) = semantics-term e f t"
  "semantics-list (put e 0 x) f (inc-list l) = semantics-list e f l"
proof (induct t and l rule: semantics-term.induct semantics-list.induct) qed simp-all

lemma commute: "put (put e v x) 0 y = put (put e 0 y) (v + 1) x" proof force qed

fun
  all :: "(fm ⇒ bool) ⇒ fm list ⇒ bool"
where
  "all b z = (∀p. if member p z then b p else True)"

lemma allhead: "all b (p # z) ⟹ b p" proof simp qed

lemma alltail: "all b (p # z) ⟹ all b z" proof simp qed

lemma allnew: "all (new c) z = news c z" proof (induct z) qed (simp, simp, metis)

lemma map':
  "new-term c t ⟹ semantics-term e (f(c := m)) t = semantics-term e f t"
  "new-list c l ⟹ semantics-list e (f(c := m)) l = semantics-list e f l"
proof (induct t and l rule: semantics-term.induct semantics-list.induct)
qed (simp, simp, metis, simp, simp, metis)

lemma map: "new c p ⟹ semantics e (f(c := m)) g p = semantics e f g p"
proof (induct p arbitrary: e)
qed (simp, simp, metis map'(2), simp, metis, simp, metis, simp, metis, simp-all)

lemma allmap: "news c z ⟹ all (semantics e (f(c := m)) g) z = all (semantics e f g) z"
proof (induct z) qed (simp, simp, metis map)
```

```
lemma substitute': 
  "semantics-term e f (sub-term v s t) = semantics-term (put e v (semantics-term e f s)) f t"
  "semantics-list e f (sub-list v s l) = semantics-list (put e v (semantics-term e f s)) f l"
proof (induct t and l rule: semantics-term.induct semantics-list.induct) qed simp-all

lemma substitute: "semantics e f g (sub v t p) = semantics (put e v (semantics-term e f t)) f g p"
proof (induct p arbitrary: e v t)
  fix i l e v t
  show "semantics e f g (sub v t (Pre i l)) =
      semantics (put e v (semantics-term e f t)) f g (Pre i l)"
  proof (simp add: substitute'(2)) qed
next
  fix p e v t assume *: "semantics e' f g (sub v' t' p) =
      semantics (put e' v' (semantics-term e' f t')) f g p" for e' v' t'
  have "semantics e f g (sub v t (Exi p)) =
      (∃x. semantics (put (put e 0 x) (v + 1) (semantics-term (put e 0 x) f (inc-term t))) f g p)"
    using * proof simp qed
  also have "... = (∃x. semantics (put (put e v (semantics-term e f t)) 0 x) f g p)"
    using commute increment(1) proof metis qed
  finally show "semantics e f g (sub v t (Exi p)) =
      semantics (put e v (semantics-term e f t)) f g (Exi p)" proof simp qed
  have "semantics e f g (sub v t (Uni p)) =
      (∀x. semantics (put (put e 0 x) (v + 1) (semantics-term (put e 0 x) f (inc-term t))) f g p)"
    using * proof simp qed
  also have "... = (∀x. semantics (put (put e v (semantics-term e f t)) 0 x) f g p)"
    using commute increment(1) proof metis qed
  finally show "semantics e f g (sub v t (Uni p)) =
      semantics (put e v (semantics-term e f t)) f g (Uni p)" proof simp qed
qed simp-all

lemma soundness': "OK p z ⟹ all (semantics e f g) z ⟹ semantics e f g p"
proof (induct arbitrary: f rule: OK.induct)
  fix f p z assume "all (semantics e f g) z"
      "all (semantics e f' g) (Imp p Falsity # z) ⟹ semantics e f' g Falsity" for f'
  then show "semantics e f g p" proof force qed
next
  fix f p q z r assume "all (semantics e f g) z"
      "all (semantics e f' g) z ⟹ semantics e f' g (Dis p q)"
      "all (semantics e f' g) (p # z) ⟹ semantics e f' g r"
      "all (semantics e f' g) (q # z) ⟹ semantics e f' g r" for f'
  then show "semantics e f g r" proof (simp, metis) qed
next
  fix f p q z assume "all (semantics e f g) z"
      "all (semantics e f' g) z ⟹ semantics e f' g (Con p q)" for f'
  then show "semantics e f g p" "semantics e f g q" proof (simp, metis, simp, metis) qed
next
  fix f p z q c assume *: "all (semantics e f g) z"
      "all (semantics e f' g) z ⟹ semantics e f' g (Exi p)"
      "all (semantics e f' g) (sub 0 (Fun c []) p # z) ⟹ semantics e f' g q"
      "news c (p # q # z)" for f'
  obtain x where "semantics (λn. if n = 0 then x else e (n − 1)) f g p"
    using *(1) *(2) proof force qed
  then have "semantics (put e 0 x) f g p" proof simp qed
  then have "semantics (put e 0 x) (f(c := λw. x)) g p"
    using *(4) allhead allnew map proof blast qed
  then have "semantics e (f(c := λw. x)) g (sub 0 (Fun c []) p)"
    proof (simp add: substitute) qed
  moreover have "all (semantics e (f(c := λw. x)) g) z"
    using *(1) *(4) alltail allnew allmap proof blast qed
  ultimately have "semantics e (f(c := λw. x)) g q" using *(3) proof simp qed
```

```
  then show "semantics e f g q" using *(4) allhead alltail allnew map proof blast qed
next
  fix f z t p assume "all (semantics e f g) z"
      "all (semantics e f' g) z ⟹ semantics e f' g (sub 0 t p)" for f'
  then have "semantics (put e 0 (semantics-term e f t)) f g p" proof (simp add: substitute) qed
  then show "semantics e f g (Exi p)" proof (simp, metis) qed
next
  fix f z t p assume "all (semantics e f g) z"
      "all (semantics e f' g) z ⟹ semantics e f' g (Uni p)" for f'
  then show "semantics e f g (sub 0 t p)" proof (simp add: substitute) qed
next
  fix f c p z assume *: "all (semantics e f g) z"
      "all (semantics e f' g) z ⟹ semantics e f' g (sub 0 (Fun c []) p)"
      "news c (p # z)" for f'
  have "semantics (λn. if n = 0 then x else e (n − 1)) f g p" for x
  proof −
    have "all (semantics e (f(c := λw. x)) g) z"
      using *(1) *(3) alltail allnew allmap proof blast qed
    then have "semantics e (f(c := λw. x)) g (sub 0 (Fun c []) p)"
      using *(2) proof simp qed
    then have "semantics (λn. if n = 0 then x else e (n − 1)) (f(c := λw. x)) g p"
      proof (simp add: substitute) qed
    then show "semantics (λn. if n = 0 then x else e (n − 1)) f g p"
      using *(3) allhead alltail allnew map proof blast qed
  qed
  then show "semantics e f g (Uni p)" proof simp qed
qed simp-all

theorem soundness: "OK p [] ⟹ semantics e f g p" proof (simp add: soundness') qed

corollary "∃p. OK p []" "∃p. ¬ OK p []"
proof −
  have "OK (Imp p p) []" for p proof (rule Imp-I, rule Assume, simp) qed
  then show "∃p. OK p []" proof iprover qed
  have "¬ semantics (e :: nat ⇒ unit) f g Falsity" for e f g proof simp qed
  then show "∃p. ¬ OK p []" using soundness proof iprover qed
qed

end
```