```
theory Untyped-Lambda-Calculus
imports Complex-Main
begin

datatype Term
  = Var nat
  | Abs Term
  | App Term Term
```

Definition 6.1.2 — n_terms

```
inductive n-term :: nat ⇒ Term ⇒ bool where
  n-term-Var: 0 ≤ k ⟹ k < n ⟹ n-term n (Var k) |
  n-term-Abs: n-term n t ⟹ n > 0 ⟹ n-term (n − 1) (Abs t) |
  n-term-App: n-term n t1 ⟹ n-term n t2 ⟹ n-term n (App t1 t2)
```

Definition 6.2.1 — Shifting

```
primrec shift :: int ⇒ nat ⇒ Term ⇒ Term where
  shift-Var: shift d c (Var k) = Var (if k < c then k else nat (k + d)) |
  shift-Abs: shift d c (Abs t) = Abs (shift d (Suc c) t) |
  shift-App: shift d c (App t1 t2) = App (shift d c t1) (shift d c t2)
```

Exercice 6.2.2

```
lemma shift 2 0 (Abs (Abs (App (Var 1) (App (Var 0) (Var 2))))) =
                 Abs (Abs (App (Var 1) (App (Var 0) (Var 4))))
  by simp

lemma shift 2 0 (Abs (App (Var 0) (App (Var 1) (Abs (App (Var 0) (App (Var
1) (Var 2))))))) =
                 Abs (App (Var 0) (App (Var 3) (Abs (App (Var 0) (App (Var 1)
(Var 4))))))
  by simp
```

Exercice 6.2.3

```
lemma n-term n t ⟹ n-term (n + d) (shift d c t)
proof (induction n t arbitrary: d c rule: n-term.induct)
  case (n-term-Var k n)
  from n-term-Var.hyps show ?case
    using  n-term.n-term-Var by simp
next
  case (n-term-Abs n t)
  from n-term-Abs.hyps show ?case
    using n-term.n-term-Abs  by (auto intro: n-term-Abs.IH)
next
  case (n-term-App n t1 t2)
  show ?case
    by (simp add: n-term.n-term-App n-term-App.IH)
qed
```

Definition 6.2.4 — Substitution

```
primrec subst :: nat ⇒ Term ⇒ Term ⇒ Term where
  subst-Var: subst j s (Var k) = (if k = j then s else Var k) |
  subst-Abs: subst j s (Abs t) = Abs (subst (Suc j) (shift 1 0 s) t) |
  subst-App: subst j s (App t1 t2) = App (subst j s t1) (subst j s t2)
```

Exercice 6.2.5

```
lemma subst 0 (Var 1) (App (Var 0) (Abs (Abs (Var 2)))) =
                        App (Var 1) (Abs (Abs (Var 3)))
  by simp

lemma subst 0 (App (Var 1) (Abs (Var 2))) (App (Var 0) (Abs (Var 1))) =
  App (App (Var 1) (Abs (Var 2))) (Abs (App (Var 2) (Abs (Var 3))))
  by simp

lemma subst 0 (Var 1) (Abs (App (Var 0) (Var 2))) =
                        Abs (App (Var 0) (Var 2))
  by simp

lemma subst 0 (Var 1) (Abs (App (Var 1) (Var 0))) =
                        Abs (App (Var 2) (Var 0))
  by simp
```

Exercice 6.2.6

```
lemma n-term-shift: n-term n t ⟹ n-term (n + j) (shift j i t)
  by (induction n t arbitrary: j i rule: n-term.induct)
    (auto intro: n-term-Var n-term-Abs[unfolded One-nat-def] n-term-App)

lemma n-term n t ⟹ n-term n s ⟹ j ≤ n ⟹ n-term n (subst j s t)
proof (induction n t arbitrary: s j rule: n-term.induct)
  case (n-term-Var k n)
  thus ?case
    by (auto intro: n-term.n-term-Var)
next
  case (n-term-Abs n t)
  thus ?case
    using n-term.n-term-Abs n-term-shift[OF n-term-Abs.prems(1), where j = 1]
    by (auto
      intro: n-term-Abs.IH
      intro!: n-term.n-term-Abs[unfolded One-nat-def]
      simp: n-term-shift[OF n-term-Abs.prems(1), where j = 1])
next
  case (n-term-App n t1 t2)
  thus ?case
    by (simp add: n-term.n-term-App)
qed
```

Single step evaluation

```
inductive is-value :: Term ⇒ bool where
  is-value (Abs t)
```

```
inductive eval-once :: Term ⇒ Term ⇒ bool where
  eval-once-App1: eval-once t1 t1' ⟹ eval-once (App t1 t2) (App t1' t2) |
  eval-once-App2: is-value v1 ⟹ eval-once t2 t2' ⟹ eval-once (App v1 t2) (App
v1 t2') |
  eval-once-App-Abs: is-value v2 ⟹ eval-once (App (Abs t12) v2) (shift (−1) 0
(subst 0 (shift 1 0 v2) t12))
```

Theorem 3.5.4 for Untyped Lambda Calculus

```
theorem eval-once-right-unique:
  eval-once t t' ⟹ eval-once t t'' ⟹ t' = t''
proof (induction t t' arbitrary: t'' rule: eval-once.induct)
  case (eval-once-App1 t1 t1' t2)
  from eval-once-App1.hyps eval-once-App1.prems show ?case
    by (auto elim: eval-once.cases is-value.cases intro: eval-once-App1.IH)
next
  case (eval-once-App2 t1 t2 t2')
  from eval-once-App2.hyps eval-once-App2.prems show ?case
    by (auto elim: eval-once.cases is-value.cases intro: eval-once-App2.IH)
next
  case (eval-once-App-Abs v2 t12)
  from eval-once-App-Abs.prems eval-once-App-Abs.hyps show ?case
    by (auto elim: eval-once.cases simp: is-value.simps)
qed
```

Definition 3.5.6 for Untyped Lambda Calculus

```
definition is-normal-form :: Term ⇒ bool where
  is-normal-form t ⟷ (∀ t'. ¬ eval-once t t')
```

Theorem 3.5.7 for Untyped Lambda Calculus

```
theorem value-imp-normal-form: is-value t ⟹ is-normal-form t
  by (auto elim: is-value.cases eval-once.cases simp: is-normal-form-def)
```

Theorem 3.5.8 does not hold for Untyped Lambda calculus

```
theorem normal-form-does-not-imp-value:
  ∃ t. is-normal-form t ∧ ¬ is-value t (is ∃ t. ?P t)
proof
  have a: is-normal-form (Var 0)
    by (auto simp: is-normal-form-def elim: eval-once.cases)
  have b: ¬ is-value (Var 0)
    by (auto simp: is-normal-form-def dest: is-value.cases)
  from a b show ?P (Var 0) by simp
qed
```

Multistep evaluation

```
inductive eval :: Term ⇒ Term ⇒ bool where
  eval-base: eval t t |
  eval-step: eval-once t t' ⟹ eval t' t'' ⟹ eval t t''
```

Corollary 3.5.11 for Untyped Lambda Calculus

```
corollary uniqueness-of-normal-form:
  assumes
    eval t u and
    eval t u′ and
    is-normal-form u and
    is-normal-form u′
  shows u = u′
using assms
proof (induction t u rule: eval.induct)
  case (eval-base t)
  thus ?case by (metis eval.simps is-normal-form-def)
next
  case (eval-step t1 t2 t3)
  thus ?case by (metis eval.cases is-normal-form-def eval-once-right-unique)
qed
```

```
lemma eval-once-VarD: eval-once (Var x) t ⟹ P
  by (induction Var x t rule: eval-once.induct)

lemma eval-once-AbsD: eval-once (Abs x) t ⟹ P
  by (induction Abs x t rule: eval-once.induct)

theorem eval-does-not-always-terminate:
  ∃ t. ∀ t′. eval t t′ ⟶ ¬ is-normal-form t′ (is ∃ t. ∀ t′. ?P t t′)
proof
  let ?ω = Abs (App (Var 0) (Var 0))
  let ?Ω = App ?ω ?ω
  { fix t
    have eval-once ?Ω t ⟹ ?Ω = t
      by (induction ?Ω t rule: eval-once.induct) (auto elim: eval-once-AbsD)
  } note eval-once-Ω = this
  { fix t
    have eval-Ω: eval ?Ω t ⟹ ?Ω = t
      by (induction ?Ω t rule: eval.induct) (blast dest: eval-once-Ω)+
  } note eval-Ω = this
  show ∀ t′. ?P ?Ω t′
    by (auto
      simp: is-normal-form-def
      intro: eval-once-App-Abs is-value.intros
      dest!: eval-Ω)
qed

end
```