```
theory Untyped-Arithmetic-Expressions
imports Main
begin

datatype B-term
  = BTrue
  | BFalse
  | BIf B-term B-term B-term

primrec consts-B :: B-term ⇒ B-term set where
  consts-B BTrue = {BTrue} |
  consts-B BFalse = {BFalse} |
  consts-B (BIf t1 t2 t3) = consts-B t1 ∪ consts-B t2 ∪ consts-B t3

primrec size-B :: B-term ⇒ nat where
  size-B BTrue = 1 |
  size-B BFalse = 1 |
  size-B (BIf t1 t2 t3) = 1 + size-B t1 + size-B t2 + size-B t3

primrec depth-B :: B-term ⇒ nat where
  depth-B BTrue = 1 |
  depth-B BFalse = 1 |
  depth-B (BIf t1 t2 t3) = 1 + max (depth-B t1) (max (depth-B t2) (depth-B t3))

lemma card-union-leq-sum-card: card (A ∪ B) ≤ card A + card B
  by (cases finite A ∧ finite B) (simp only: card-Un-Int, auto)

lemma card (consts-B t) ≤ size-B t
proof (induction t)
  case BTrue
  show ?case by simp
next
  case BFalse
  show ?case by simp
next
  case (BIf t1 t2 t3)
  show ?case
  proof −
    let ?t1 = consts-B t1
    let ?t2 = consts-B t2
    let ?t3 = consts-B t3
```

```
    have card (?t1 ∪ ?t2 ∪ ?t3) ≤ card ?t1 + card ?t2 + card ?t3
    by (smt card-union-leq-sum-card add-le-imp-le-right le-antisym le-trans nat-le-linear)
    also have ... ≤ size-B t1 + size-B t2 + size-B t3
      using BIf.IH by simp
    finally show ?thesis by simp
  qed
qed



inductive is-value :: B-term ⇒ bool where
  is-value-BTrue: is-value BTrue |
  is-value-BFalse: is-value BFalse

inductive-cases is-value-BIfD: is-value (BIf t1 t2 t3)

inductive eval-once :: B-term ⇒ B-term ⇒ bool where
  e-if-true: eval-once (BIf BTrue t2 t3) t2 |
  e-if-false: eval-once (BIf BFalse t2 t3) t3 |
  e-if: eval-once t1 t1' ⟹ eval-once (BIf t1 t2 t3) (BIf t1' t2 t3)

inductive-cases eval-once-BTrueD: eval-once BTrue t
inductive-cases eval-once-BFalseD: eval-once BFalse t

inductive eval :: B-term ⇒ B-term ⇒ bool where
  e-once: eval-once t t' ⟹ eval t t' |
  e-self: eval t t |
  e-transitive: eval t t' ⟹ eval t' t'' ⟹ eval t t''

inductive eval' :: B-term ⇒ B-term ⇒ bool where
  e-base': eval' t t |
  e-step': eval-once t t' ⟹ eval' t' t'' ⟹ eval' t t''

lemma e-once': eval-once t t' ⟹ eval' t t'
  by (simp add: e-base' e-step')

lemma e-transitive': eval' t t' ⟹ eval' t' t'' ⟹ eval' t t''
proof (induction t t' arbitrary: t'' rule: eval'.induct)
  case (e-base' t t'') thus ?case .
next
  case (e-step' t t' t'' t''')
  thus ?case using eval'.e-step' by blast
qed



lemma eval-eq-eval': eval = eval'
  apply (rule ext)+
  apply (rule iffI)
```

```
   apply (rename-tac t t′)
   apply (erule eval.induct)
      apply (erule e-once′)
    apply (rule e-base′)
   apply (erule e-transitive′)
   apply assumption

  apply (erule eval′.induct)
   apply (rule e-self)
  using e-once e-transitive by blast

definition is-normal-form :: B-term ⇒ bool where
  is-normal-form t ⟷ (∀ t′. ¬ eval-once t t′)




lemma
  assumes
    s: s = BIf BTrue BFalse BFalse and
    t: t = BIf s BTrue BTrue and
    u: u = BIf BFalse BTrue BTrue
  shows eval-once (BIf t BFalse BFalse) (BIf u BFalse BFalse)
proof −
  have eval-once s BFalse unfolding s by (rule e-if-true)
  hence eval-once t u unfolding t u by (rule e-if)
  thus ?thesis by (rule e-if)
qed




theorem eval-single-determinacy:
  fixes t t′ t′′ :: B-term
  shows eval-once t t′ ⟹ eval-once t t′′ ⟹ t′ = t′′
proof (induction t t′ arbitrary: t′′ rule: eval-once.induct)
  case (e-if-true t1 t2)
  thus ?case by (auto elim: eval-once.cases)
next
  case (e-if-false t1 t2)
  thus ?case by (auto elim: eval-once.cases)
next
  case (e-if t1 t1′ t2 t3)
  show ?case
    apply (rule eval-once.cases[OF e-if.prems])
    using e-if.hyps by (auto dest: eval-once-BTrueD eval-once-BFalseD e-if.IH)
qed
```

theorem *value-imp-normal-form*:
  fixes *t* :: *B-term*
  shows *is-value t* $\Longrightarrow$ *is-normal-form t*
by (*auto simp*: *is-normal-form-def elim*: *is-value.cases dest*: *eval-once-BTrueD eval-once-BFalseD*)

theorem *normal-form-imp-value*:
  fixes *t* :: *B-term*
  shows *is-normal-form t* $\Longrightarrow$ *is-value t*
proof (*rule ccontr*, *induction t rule*: *B-term.induct*)
  case *BTrue*
  thus *?case* **by** (*simp add*: *is-value-BTrue*)
next
  case *BFalse*
  thus *?case* **by** (*simp add*: *is-value-BFalse*)
next
  case (*BIf t1 t2 t3*)
  thus *?case* **by** (*metis e-if e-if-false e-if-true is-normal-form-def is-value.cases*)
qed

corollary *uniqueness-of-normal-form*:
  fixes *t u u′* :: *B-term*
  assumes
    eval t u **and**
    eval t u′ **and**
    is-normal-form u **and**
    is-normal-form u′
  shows *u* = *u′*
using *assms*
unfolding *eval-eq-eval′*
proof (*induction t u rule*: *eval′.induct*)
  case (*e-base′ t*)
  thus *?case* **by** (*metis eval′.simps is-normal-form-def*)
next
  case (*e-step′ t t′ t″*)
  thus *?case* **by** (*metis eval′.cases is-normal-form-def eval-single-determinacy*)
qed

lemma *eval-once-size-B*:

```
  assumes eval-once t t′
  shows size-B t > size-B t′
using assms
proof (induction rule: eval-once.induct)
  case e-if-true
  thus ?case by simp
next
  case e-if-false
  thus ?case by simp
next
  case e-if
  thus ?case by simp
qed
```

```
theorem eval-always-terminate:
  ∃ t′. eval t t′ ∧ is-normal-form t′
unfolding eval-eq-eval′
proof (induction rule: measure-induct-rule[of size-B])
  case (less t)
  show ?case
    apply (cases is-normal-form t)
    using e-base′ apply blast
    using e-step′ is-normal-form-def eval-once-size-B less.IH by blast
qed
```

```
datatype NBTerm
  = NBTrue
  | NBFalse
  | NBIf NBTerm NBTerm NBTerm
  | NBZero
  | NBSucc NBTerm
  | NBPred NBTerm
  | NBIs-zero NBTerm

primrec size-NB :: NBTerm ⇒ nat where
  size-NB NBTrue = 1 |
  size-NB NBFalse = 1 |
  size-NB NBZero = 1 |
  size-NB (NBSucc t) = 1 + size-NB t |
  size-NB (NBPred t) = 1 + size-NB t |
  size-NB (NBIs-zero t) = 1 + size-NB t |
  size-NB (NBIf t1 t2 t3) = 1 + size-NB t1 + size-NB t2 + size-NB t3
```

inductive *is-numeric-value-NB* :: *NBTerm* ⇒ *bool* **where**
is-numeric-value-NBZero: *is-numeric-value-NB NBZero* |
is-numeric-value-NBSucc: *is-numeric-value-NB nv* ⟹ *is-numeric-value-NB* (*NBSucc nv*)

inductive *is-value-NB* :: *NBTerm* ⇒ *bool* **where**
is-value-NBTrue: *is-value-NB NBTrue* |
is-value-NBFalse: *is-value-NB NBFalse* |
is-value-NB-numeric-value: *is-numeric-value-NB nv* ⟹ *is-value-NB nv*

inductive *eval-once-NB* :: *NBTerm* ⇒ *NBTerm* ⇒ *bool* **where**
eval-once-NBIf-NBTrue: *eval-once-NB* (*NBIf NBTrue t2 t3*) *t2* |
eval-once-NBIf-NBFalse: *eval-once-NB* (*NBIf NBFalse t2 t3*) *t3* |
eval-once-NBIf: *eval-once-NB t1 t1'* ⟹ *eval-once-NB* (*NBIf t1 t2 t3*) (*NBIf t1' t2 t3*) |
eval-once-NBSucc: *eval-once-NB t1 t1'* ⟹ *eval-once-NB* (*NBSucc t1*) (*NBSucc t1'*) |
eval-once-NBPred-NBZero: *eval-once-NB* (*NBPred NBZero*) *NBZero* |
eval-once-NBPred-NBSucc: *is-numeric-value-NB nv1* ⟹ *eval-once-NB* (*NBPred* (*NBSucc nv1*)) *nv1* |
eval-once-NBPred: *eval-once-NB t1 t1'* ⟹ *eval-once-NB* (*NBPred t1*) (*NBPred t1'*) |
eval-once-NBIs-zero-NBZero: *eval-once-NB* (*NBIs-zero NBZero*) *NBTrue* |
eval-once-NBIs-zero-NBSucc: *is-numeric-value-NB nv1* ⟹ *eval-once-NB* (*NBIs-zero* (*NBSucc nv1*)) *NBFalse* |
eval-once-NBIs-zero: *eval-once-NB t1 t1'* ⟹ *eval-once-NB* (*NBIs-zero t1*) (*NBIs-zero t1'*)

inductive *eval-NB* :: *NBTerm* ⇒ *NBTerm* ⇒ *bool* **where**
eval-NB-base: *eval-NB t t* |
eval-NB-step: *eval-once-NB t t'* ⟹ *eval-NB t' t''* ⟹ *eval-NB t t''*

definition *is-normal-form-NB* :: *NBTerm* ⇒ *bool* **where**
is-normal-form-NB t ⟷ (∀ *t'*. ¬ *eval-once-NB t t'*)

lemma *eval-once-NB-impl-eval-NB*: *eval-once-NB t t'* ⟹ *eval-NB t t'*
by (*simp add*: *eval-NB-step eval-NB-base*)

lemma *eval-NB-transitive*: *eval-NB t t'* ⟹ *eval-NB t' t''* ⟹ *eval-NB t t''*
proof (*induction t t' arbitrary*: *t'' rule*: *eval-NB.induct*)
case (*eval-NB-base t t'*)
thus *?case* **.**
next
case (*eval-NB-step t1 t2 t3*)
thus *?case* **using** *eval-NB.eval-NB-step* **by** *blast*
qed

```
inductive-cases eval-once-NBTrueD: eval-once-NB NBTrue t
inductive-cases eval-once-NBFalseD: eval-once-NB NBFalse t
inductive-cases eval-once-NBZeroD: eval-once-NB NBZero t

lemma not-eval-once-numeric-value: is-numeric-value-NB nv ⟹ eval-once-NB
nv t ⟹ P
proof (induction nv arbitrary: t rule: is-numeric-value-NB.induct)
  case is-numeric-value-NBZero
  thus ?case by (auto elim: eval-once-NB.cases)
next
  case is-numeric-value-NBSucc
  show ?case
    by (auto
      intro: is-numeric-value-NBSucc.prems[THEN eval-once-NB.cases]
      elim: is-numeric-value-NBSucc.IH)
qed


theorem eval-once-NB-right-unique:
  fixes t t′ t″ :: NBTerm
  shows eval-once-NB t t′ ⟹ eval-once-NB t t″ ⟹ t′ = t″
proof (induction t t′ arbitrary: t″ rule: eval-once-NB.induct)
  case (eval-once-NBIf-NBTrue t1 t2)
  thus ?case by (auto elim: eval-once-NB.cases)
next
  case (eval-once-NBIf-NBFalse t1 t2)
  thus ?case by (auto elim: eval-once-NB.cases)
next
  case (eval-once-NBIf t1 t1′ t2 t3)
  from eval-once-NBIf.prems eval-once-NBIf.hyps show ?case
    by (auto
      intro: eval-once-NB.cases
      dest: eval-once-NBTrueD eval-once-NBFalseD eval-once-NBIf.IH)
next
  case (eval-once-NBSucc t1 t2)
  from eval-once-NBSucc.prems eval-once-NBSucc.IH show ?case
    by (auto elim: eval-once-NB.cases)
next
  case eval-once-NBPred-NBZero
  thus ?case by (auto intro: eval-once-NB.cases dest: eval-once-NBZeroD)
next
  case (eval-once-NBPred-NBSucc nv1)
  show ?case
    apply (rule eval-once-NBPred-NBSucc.prems[THEN eval-once-NB.cases])
    using eval-once-NBPred-NBSucc.hyps
    by (auto elim: is-numeric-value-NBSucc not-eval-once-numeric-value[rotated])
next
```

```
  case (eval-once-NBPred t1 t2)
  from eval-once-NBPred.hyps eval-once-NBPred.prems show ?case
    using  is-numeric-value-NBZero
    by (auto
      intro: eval-once-NBPred.IH
      elim: eval-once-NB.cases
      dest: not-eval-once-numeric-value is-numeric-value-NBSucc)
next
  case eval-once-NBIs-zero-NBZero
  thus ?case
    by (auto intro: eval-once-NB.cases dest: eval-once-NBZeroD)
next
  case (eval-once-NBIs-zero-NBSucc nv)
  thus ?case
   by (auto intro: eval-once-NB.cases not-eval-once-numeric-value dest: is-numeric-value-NBSucc)
next
  case (eval-once-NBIs-zero t1 t2)
  show ?case
    apply (rule eval-once-NBIs-zero.prems[THEN eval-once-NB.cases])
    using eval-once-NBIs-zero.hyps
    by (auto
      intro: eval-once-NBZeroD eval-once-NBIs-zero.IH
      elim: not-eval-once-numeric-value[rotated] is-numeric-value-NBSucc)
qed
```

```
theorem value-imp-normal-form-NB:
  is-value-NB t ⟹ is-normal-form-NB t
  by (auto
    simp: is-normal-form-NB-def
    elim: is-value-NB.cases
    dest: eval-once-NBFalseD eval-once-NBTrueD not-eval-once-numeric-value)
```

```
theorem not-normal-form-imp-value-NB: ∃ t. is-normal-form-NB t ∧ ¬ is-value-NB
t (is ∃ t. ?P t)
proof
  have a: is-normal-form-NB (NBSucc NBTrue)
    by (auto elim: eval-once-NB.cases simp: is-normal-form-NB-def)
  have b: ¬ is-value-NB (NBSucc NBTrue)
    by (auto elim: is-numeric-value-NB.cases simp: is-value-NB.simps)
  from a b show ?P (NBSucc NBTrue) by simp
qed
```

corollary *uniqueness-of-normal-form-NB*:
  assumes
    eval-NB t u **and**
    eval-NB t u' **and**
    is-normal-form-NB u **and**
    is-normal-form-NB u'
  shows $u = u'$
using *assms*
proof (*induction* t u *rule*: *eval-NB.induct*)
  case (*eval-NB-base* t)
  thus *?case* **by** (*metis eval-NB.simps is-normal-form-NB-def*)
next
  case (*eval-NB-step t1 t2 t3*)
  thus *?case* **by** (*metis eval-NB.cases is-normal-form-NB-def eval-once-NB-right-unique*)
qed

lemma *eval-once-size-NB*:
  eval-once-NB t $t' \Longrightarrow$ *size-NB* $t >$ *size-NB* t'
by (*induction rule*: *eval-once-NB.induct*) *auto*

theorem *eval-NB-always-terminate*:
  $\exists t'.$ *eval-NB* t $t' \wedge$ *is-normal-form-NB* t'
proof (*induction rule*: *measure-induct-rule*[*of size-NB*])
  case (*less* t)
  show *?case*
    apply (*case-tac is-normal-form-NB* t)
    using *eval-NB-base* **apply** *blast*
    using *eval-NB-step eval-once-size-NB is-normal-form-NB-def less.IH* **by** *blast*
qed