Szintaxisfák

September 23, 2024

1 Bevezetés

Ebben a szakaszban egy egyszerű bináris fa adatstruktúra és ahhoz kapcsolódó függvények, valamint bizonyítások kerülnek bemutatásra a `Coq` formális verifikációs rendszer segítségével. A cél egy bináris fa implementálása és az annak működésével kapcsolatos tulajdonságok igazolása, továbbá egy általánosabb, magasság-alapú fa (`HTree`) definíciója és vizsgálata.

Az induktív típusok és bizonyítások kapcsán Christine Paulin-Mohring kijelenti:

> Intuitívan, egy induktívan definiált típus a típushoz tartozó kifejezések teljes listájával van megadva. Az indukciós elv segítségével érvelünk a típus felett, és iterációval definiáljuk a típus elemein végzett függvényeket, ami elegendő erejű a primitív rekurzív funkcionálisok definiálásához."

Minden Coq definíciót követ egy helyességi állítás, ami garantálja, hogy a függvények megfelelően működnek.

2 Bináris Fa Definíciója

A bináris fa (`binTree`) egy egyszerű rekurzív adatstruktúra, amely vagy egy levél (`leaf`), vagy két alfaából álló csomópont (`node`).

```
Inductive binTree : Set :=
  | leaf : binTree
  | node : binTree -> binTree -> binTree.
```

2.1 Levélhossz számítása

A `leafLength` függvény egy bináris fa leveleinek számát adja meg. A levél (`leaf`) esetében 1-et ad vissza, míg egy csomópont esetén (`node`) rekurzívan összegzi az alfaák leveleinek számát.

```
Fixpoint leafLength (t : binTree) {struct t} : nat :=
  match t with
    | leaf => 1
    | node t1 t2 => (leafLength t1) + (leafLength t2)
  end.
```

Az alábbi lemma bizonyítja, hogy a `leafLength` függvény helyesen számolja meg a levelek számát egy levél (1. lemma) és egy csomópont (2. lemma) esetén:

```
Lemma leafLengthSound_1 : leafLength leaf = 1.
Proof.
simpl. auto.
Defined.

Lemma leafLengthSound_2 : forall t1 t2, leafLength (node t1 t2) = leafLength t1 + lea
Proof.
induction t1, t2.
all: simpl; auto.
Defined.
```

2.2 Fa megfordítása

A `revertBinTree` függvény egy bináris fa megfordítását végzi el, azaz a bal és jobb alfák helyet cserélnek minden csomópontban.

```
Fixpoint revertBinTree (t : binTree) : binTree :=
  match t with
  | leaf => leaf
  | node t1 t2 => node (revertBinTree t2) (revertBinTree t1)
  end.
```

A következő tétel igazolja, hogy egy fa kétszeres megfordítása az eredeti fát adja vissza:

```
Theorem revertBinTreeSound : forall t, revertBinTree (revertBinTree t) = t.
Proof.
induction t.
- simpl. auto.
- simpl. rewrite IHt2. rewrite IHt1. auto.
Defined.
```

2.3 Jobbra bővítés

A `mostRightAppend` függvény egy bináris fához egy másik fát fűz hozzá a jobb oldalon a legmélyebb levél helyére.

```
Fixpoint mostRightAppend (t s : binTree) {struct t} : binTree :=
  match t with
   | leaf => s
   | node t1 t2 => node t1 (mostRightAppend t2 s)
  end.
```

A `mostRightAppend_correct` definíció megadja, hogy milyen tulajdonságot kell teljesítenie a függvénynek:

```
Definition mostRightAppend_correct (t s result : binTree) : Prop :=
  forall t1 t2, t = node t1 t2 ->
  result = node t1 (mostRightAppend t2 s).
```

A következő lemma bizonyítja, hogy a `mostRightAppend` helyesen működik:

```
Lemma mostRightAppend_correct_proof :
  forall t s, mostRightAppend_correct t s (mostRightAppend t s).
Proof.
  intros t s.
  induction t.
  - (* leaf *)
    unfold mostRightAppend_correct.
    intros t1 t2 H.
    discriminate H.
  - (* node *)
    unfold mostRightAppend_correct.
    intros t1' t2' H.
    inversion H.
    rewrite <- H.
    rewrite <- H1.
    rewrite <- H2.
    simpl.
    reflexivity.
Defined.
```

Az alábbi lemma kapcsolatot teremt a `mostRightAppend` és a levélhossz között:

```
Lemma Right_leafLength : forall t s, leafLength (mostRightAppend t s) + 1 = leafLengt
Proof.
  intros t s.
  induction t.
  - simpl. lia.
  - simpl mostRightAppend.
    simpl leafLength.
    rewrite IHt2.
    auto.
Defined.
```

3 Legfeljebb H magasságú fa (`HTree`) és a Transport Hell

A `HTree` egy bináris fa, ahol minden csomópont tartalmazza a magasságot. Ez a fa egy típusparaméteres adatstruktúra, amely a magasságot is figyelembe veszi.

```
Inductive HTree : nat -> Set :=
  | Hleaf : HTree 0
  | Hnode : forall n m : nat, HTree n -> HTree m -> HTree (S (max n m)).
```

A `Height` függvény a fa magasságát számítja ki:

```
Fixpoint Height (n : nat) (t : HTree n) : nat :=
  match t with
    | Hleaf => 0
    | Hnode n m t1 t2 => (max (Height n t1) (Height m t2)) + 1
  end.
```

A következő lemma bizonyítja, hogy a `Height` függvény helyesen számolja a fa magasságát:

```
Lemma Height_lemma : forall (n : nat) (t : HTree n), Height n t = n.
Proof.
intros.
induction t.
- compute; auto.
- simpl.
  rewrite IHt1.
  rewrite IHt2.
  lia.
Defined.
```

Az ilyen típusok kezelésénél felmerülhet a "transport hell" problémája, amikor a típusok közötti transzport túlzottan bonyolulttá válik. Ez a probléma megnehezítheti az algoritmusok bizonyítását, különösen a visszafordítás (`revertBinTree`) esetében. A probléma mélyebb vizsgálata elérhető az alábbi dokumentumban: Transport Hell in Coq

4 Példa: kifejezéstípusok

Az absztrakt szintaxis fa (AST) fontos szerepet játszik a programozáselméletben és a formális verifikációban.

4.1 Kifejezés Típusok

A kifejezések típusa a következőképpen van definiálva:

```
Inductive Exp : Set :=
  | AT : nat -> Exp
  | NOT : Exp -> Exp
  | AND : Exp -> Exp -> Exp
  | OR : Exp -> Exp -> Exp.
```

A **Exp** típus a logikai kifejezéseket reprezentálja, beleértve az atomikus kifejezéseket (`AT`), a logikai negációt (`NOT`), az és (`AND`) és vagy (`OR`) műveleteket.

4.2 Egységes és Kettős Operátorok

Az egységes operátorok és a kettős operátorok típusa a következőképpen van definiálva:

```
Inductive UnOp : Set :=
  | NOT_c : UnOp.

Inductive BinOp : Set :=
  | AND_c : BinOp
  | OR_c : BinOp.
```

A **UnOp** az egységes operátorokat definiálja, míg a **BinOp** a kettős operátorokat. Az `NOT_c`, `AND_c` és `OR_c` reprezentálják a megfelelő logikai műveleteket.

4.3 Absztrakt Szintaxis Fa (AST)

Az AST definiálása a következőképpen történik:

```
Inductive AST : Set :=
  | leaf : nat -> AST
  | node1 : UnOp -> AST -> AST
  | node2 : BinOp -> AST -> AST -> AST.
```

A **AST** típus az absztrakt szintaxis fa struktúráját reprezentálja. Az `leaf` leveleket tartalmaz, míg a `node1` és `node2` csomópontok az egységes és kettős műveleteket alkalmazzák az AST csomópontra.

5 Denotációs Szemantika

A denotációs szemantika lehetővé teszi a kifejezések és AST-k kiértékelését egy adott változó értékelési függvény alapján.

5.1 Kifejezés Denotáció

A kifejezés denotációja a következőképpen van definiálva:

```
Fixpoint ExpDenote (e : Exp) (v : nat -> bool ) {struct e} :=
match e with
  | AT n => v n
  | NOT e1 => negb (ExpDenote e1 v)
  | AND e1 e2 => andb (ExpDenote e1 v) (ExpDenote e2 v)
  | OR e1 e2 => orb (ExpDenote e1 v) (ExpDenote e2 v)
end.
```

A `ExpDenote` függvény kiértékeli a kifejezést a megadott változó értékelési függvény `v` segítségével. Mintázatillesztést használunk a kifejezéstípus kiértékeléséhez.

5.2 AST Denotáció

Az AST denotációja a következőképpen van definiálva:

```
Fixpoint ASTDenote (t: AST) (v : nat -> bool ) {struct t} :=
match t with
  | leaf n => v n
  | node1 _ t1 => negb (ASTDenote t1 v)
  | node2 x t1 t2 => match x with
           | AND_c => andb (ASTDenote t1 v) (ASTDenote t2 v)
           | OR_c => orb (ASTDenote t1 v) (ASTDenote t2 v)
                   end
end.
```

A `ASTDenote` függvény hasonlóan működik, mint a `ExpDenote`, de az AST struktúra alapján értékeli ki a logikai kifejezéseket.

6 Fordítók

A következő szekciókban bemutatjuk a fordító függvényeket, amelyek átkonvertálják a kifejezéseket AST formátumba és vissza.

6.1 Kifejezés AST-ra Fordítása

A kifejezés AST-ra fordítását a következőképpen végezzük:

```
Fixpoint Translater1 (e : Exp) :=
match e with
  | AT n => leaf n
  | NOT e1 => node1 NOT_c (Translater1 e1)
  | AND e1 e2 => node2 AND_c (Translater1 e1) (Translater1 e2)
  | OR e1 e2 => node2 OR_c (Translater1 e1) (Translater1 e2)
end.
```

A `Translater1` függvény átkonvertálja a kifejezéseket AST formátumba, megfelelően térképezve a kifejezés konstruktorait.

6.2 AST Kifejezésre Fordítása

Az AST kifejezésre fordítása a következőképpen történik:

```
Fixpoint Translater2 (t : AST) :=
match t with
  | leaf n => AT n
  | node1 _ t1 => NOT (Translater2 t1)
  | node2 AND_c t1 t2 => AND (Translater2 t1) (Translater2 t2)
  | node2 OR_c t1 t2 => OR (Translater2 t1) (Translater2 t2)
end.
```

A `Translater2` visszafordítja az AST-t kifejezés formátumba, megfordítva a `Translater1` által végzett folyamatot.

7 A Helyesség Tétel

A következő tétel bizonyítja, hogy az AST és a kifejezés kiértékelési eredményei egyenlőek:

```
Theorem Soundness1 : forall t v, ASTDenote t v = ExpDenote (Translater2 t) v.
Proof.
intros.
induction t.
compute.
auto.
simpl.
rewrite IHt.
auto.
induction b.
all: simpl; rewrite IHt1; rewrite IHt2; auto.
Qed.
```