A ternáris kondicionális nyelve

2025. szeptember 10.

Tartalomjegyzék

1. A Nyelv Rekurzív Definíciója

Ebben a részben egy egyszerű nyelvet definiálunk induktívan, amely mindössze háromféle kifejezést (termet) ismer: az igaz (`tt`), a hamis (`ff`) és a feltételes elágazást (`if_then_else`). A nyelv szintaxisát a Backus–Naur-forma (BNF) segítségével a következőképpen adhatjuk meg:

$$t ::= \texttt{tt} \mid \texttt{ff} \mid \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t$$

1.1. Definíció (Term). *A Coq rendszerben ezt az induktív definíciót az* `Inductive` *kulcsszóval hozzuk létre.*

```
Inductive Term : Set :=
  | tt : Term
  | ff : Term
  | if_then_else : Term -> Term -> Term -> Term.
```

1.2. Definíció (Típusolási Szabályok). *A nyelvünk minden kifejezése jól tipizált, egy saját* `bool` *típussal rendelkezik. A* `has_type` *predikátum definiálja, hogy egy term formailag helyes-e. Jelölés:* $\vdash t : \texttt{bool}$.

```
Inductive has_type : Term -> Prop :=
  | T_True : has_type tt
  | T_False : has_type ff
  | T_If : forall p q r,
      has_type p ->
      has_type q ->
      has_type r ->
      has_type (if_then_else p q r).

(* Egy k nyelmi jel l s a 'has_type t' kifejez sre. *)
Notation "    t [:] bool" := (has_type t) (at level 40, no associativity).
```

2. Taktikák Bemutatása Egyszerű Példákon

Ebben a szakaszban azokat a Coq taktikákat mutatjuk be, amelyeket a későbbi, bonyolultabb bizonyításokban is használni fogunk.

2.1. Taktika: `apply`

Az `apply` megpróbálja a jelenlegi bizonyítandó célt (*goal*) illeszteni egy hipotézis vagy definíció konklúziójára. Ha sikerül, a cél helyébe a hipotézis premisszái (feltételei) kerülnek.

2.1. Példa (`apply` használata). *A* `T_True` *definíció nem rendelkezik előfeltételekkel, így az* `apply` *azonnal megoldja a célt.*

```
Example apply_pelda_1 :     tt [:] bool.
Proof.
  apply T_True.
Qed.
```

2.2. Taktika: `intros` és `reflexivity`

Az `intros` taktikát arra használjuk, hogy a cél elején lévő univerzális kvantorral (`forall`) vagy implikációval lekötött változókat a feltételek (hipotézisek) közé helyezzük. A `reflexivity` taktika az $x = x$ alakú célokat oldja meg.

2.2. Példa (`intros` és `reflexivity`). *Bizonyítsuk be, hogy minden term egyenlő önmagával.*

```
Example intros_pelda : forall (t : Term), t = t.
Proof.
  intros t.      (* Bevezeti ’t’-t a hipot zisek k z   *)
  reflexivity.   (* Megoldja a ’t = t’ c lt *)
Qed.
```

2.3. Taktika: `simpl`

A `simpl` taktika kiértékeli (egyszerűsíti) a kifejezéseket a célban, például végrehajtja a rekurzív függvényhívásokat.

2.1. Definíció (Rekurzív függvény)**.** *Definiáljunk egy egyszerű rekurzív függvényt, amely "megduplázza" a feltételes kifejezéseket.*

```
Fixpoint double_if (t: Term) : Term :=
  match t with
  | tt => if_then_else tt tt tt
  | ff => if_then_else ff ff ff
  | if_then_else p q r => if_then_else (double_if p) (double_if q) (double_if r)
  end.
```

2.3. Példa (`simpl` használata)**.** *A `simpl` kiértékeli a `double_if tt` hívást, ami után a cél triviálisan igazolható.*

```
Example simpl_pelda : double_if tt = if_then_else tt tt tt.
Proof.
  simpl.
  reflexivity.
Qed.
```

2.4. Taktika: `unfold`

Az `unfold` kibont egy definíciót. Akkor hasznos, ha egy elnevezés megakadályozza a `simpl` működését.

2.4. Példa (`unfold` használata)**.** *A `simpl` nem tudja, mi az a `TRUE_TERM`, ezért előbb ki kell bontanunk a definícióját.*

```
Definition TRUE_TERM := tt.

Example unfold_pelda : TRUE_TERM = tt.
Proof.
  unfold TRUE_TERM.
  reflexivity.
Qed.
```

2.5. Taktika: `induction`

Az `induction` az egyik legfontosabb taktika, amellyel induktív adattípusokon (mint a `Term`) végzünk strukturális indukciós bizonyításokat. A rekurzív eseteknél indukciós hipotézist (IH) is kapunk a részkifejezésekre.

2.5. Példa (Indukciós bizonyítás)**.** *Bizonyítsuk be, hogy minden általunk definiált term jól tipizált.*

```
Example induction_pelda : forall t, has_type t.
Proof.
  induction t.
  - (* tt eset *) apply T_True.
  - (* ff eset *) apply T_False.
  - (* if_then_else eset *)
    apply T_If.
```

```
    + apply IHt1. (* Indukci s hipot zis az 1. r sztermre *)
    + apply IHt2. (* Indukci s hipot zis a 2. r sztermre *)
    + apply IHt3. (* Indukci s hipot zis a 3. r sztermre *)
Qed.
```

3. Operacionális Szemantika

Az operacionális szemantika azt írja le, hogyan "futnak" a programjaink, azaz hogyan értékelődnek ki a kifejezések.

3.1. Egy Program: Az OR Művelet

Definiáljunk egy logikai VAGY műveletet a nyelvünkön.

```
Definition OR (x y : Term) := if_then_else x tt y.
```

3.2. Nagy Lépéses (Big-Step) Szemantika: `beta_reduce`

A `beta_reduce` függvény egyetlen lépésben teljesen kiértékel egy kifejezést, amíg az normál formára (itt: `tt` vagy `ff`) nem egyszerűsödik.

3.1. Definíció (Nagy lépéses kiértékelés)**.** *A kiértékelés rekurzívan történik: egy* `if p q r` *kifejezés esetén először kiértékeljük* `p`*-t, és annak eredményétől függően folytatjuk* `q` *vagy* `r` *kiértékelésével.*

```
Fixpoint beta_reduce (t : Term) : Term :=
  match t with
  | tt => tt
  | ff => ff
  | if_then_else p q r =>
      match beta_reduce p with
      | tt => beta_reduce q
      | ff => beta_reduce r
      | p' => if_then_else p' q r (* Ez az  g  sosem   rhet   el, ha p j l tipiz lt *)
      end
  end.
```

Példa kiértékelésre

A `Compute` parancs segítségével láthatjuk a kiértékelés eredményét.

```
Compute beta_reduce (if_then_else ff (if_then_else ff ff tt) tt).
(* Eredm ny: = tt : Term *)
```

3.1. Lemma (Az OR művelet viselkedése)**.** *Az* `OR` *függvény helyesen működik: ha az első argumentum* `tt`*, az eredmény* `tt`*; ha* `ff`*, az eredmény a második argumentum kiértékelt alakja.*

```
Lemma Or_first_true : forall y, beta_reduce (OR tt y) = tt.
Proof. simpl. reflexivity. Qed.

Lemma Or_first_false : forall y, beta_reduce (OR ff y) = beta_reduce y.
Proof. intros y. simpl. reflexivity. Qed.
```

3.3. Normalizáció és Taktikák: `left` / `right`

Egy term akkor van normál formában, ha az vagy `tt`, vagy `ff`.

3.2. Definíció (Normál forma)**.**

```
Definition is_normal (t : Term) : Prop := t = tt \/ t = ff.
```

Ha a cél egy diszjunkció (`\/`, vagyis "vagy"), akkor a `left` vagy a `right` taktikával ki kell választanunk, hogy melyik ágat kívánjuk bizonyítani.

3.1. Példa (`left` és `right` használata).

```
Example left_pelda_1 : is_normal tt.
Proof.
  unfold is_normal. left. reflexivity.
Qed.

Example right_pelda_2 : is_normal ff.
Proof.
  unfold is_normal. right. reflexivity.
Qed.
```

3.1. Tétel (Gyenge Normalizáció). *Minden (jól tipizált) term kiértékelése egy normál formához vezet. Más szóval, a `beta_reduce` függvény mindig `tt`-vel vagy `ff`-fel tér vissza.*

```
Theorem weak_normalization : forall t : Term, is_normal (beta_reduce t).
Proof.
  induction t.
  - (* tt eset *)
    unfold is_normal; left; reflexivity.
  - (* ff eset *)
    unfold is_normal; right; reflexivity.
  - (* if_then_else p q r eset *)
    simpl.
    destruct (IHt1) as [H | H]. (* Esetv laszt s az indukci s hipot zisre *)
    + (* p -> tt *) rewrite H. exact IHt2.
    + (* p -> ff *) rewrite H. exact IHt3.
Qed.
```

4. Kis Lépéses (Small-Step) Szemantika

A kis lépéses szemantika egyszerre csak egyetlen redukciós lépést hajt végre. Ez közelebb áll a valós számítógépek működéséhez.

4.1. Definíció (Kis lépéses kiértékelés).

```
Fixpoint beta_reduce_small_step (t: Term) : Term :=
  match t with
  | tt => tt
  | ff => ff
  | if_then_else p q r =>
      match p with
      | tt => q
      | ff => r
      | _ => if_then_else (beta_reduce_small_step p) q r
      end
  end.
```

Példa a kis lépéses kiértékelésre

Látható, hogy az első lépés csak a legbelső `if` kifejezést értékeli ki.

```
Compute beta_reduce_small_step (if_then_else (if_then_else ff ff tt) ff tt).
(* Eredm ny: = if_then_else tt ff tt : Term *)

Compute beta_reduce_small_step (beta_reduce_small_step (if_then_else (if_then_else
    ff ff tt) ff tt)).
(* Eredm ny: = ff : Term *)
```

4.2. Definíció (Teljes kiértékelés kis lépésekkel). *Definiálhatunk egy "motort", ami egy adott termet n-szer léptet a `beta_reduce_small_step` függvénnyel. Ha n elég nagy (pl. a kifejezés maximális mélysége), akkor garantáltan eljutunk a normál formáig.*

```
(* Kisz molja egy kifejez s maxim lis be gyaz si  m lys g t. *)
Fixpoint depth (t: Term) : nat :=
```

```
  match t with
  | tt => 0
  | ff => 0
  | if_then_else p q r => S (max (depth p) (max (depth q) (depth r)))
  end.

(* Egy "motor", ami n-szer alkalmazza a kis l p ses ki rt kel st. *)
Fixpoint engine (n : nat) (t : Term) : Term :=
  match n with
  | 0 => t
  | S n => engine n (beta_reduce_small_step t)
  end.

(* A teljes ki rt kel s a m lys ggel korl tozott sz m  kis l p ssel. *)
Definition full_reduce t := engine (depth t) t.
```

A nyelvünkre igaz az **erős normalizáció tétele** is: bárhogyan is választjuk a redukciós lépések sorrendjét (ha több lehetőség van), véges számú lépésben mindig egyértelmű normál formához jutunk.