Programowanie z typami zależnymi i dowodzenie twierdzeń

Refleksja

12.05.2015

Typy zależne i reguła konwersji

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \equiv A' \qquad \Gamma \vdash A' : s}{\Gamma \vdash M : A'}$$

Pomysł: wykorzystanie obliczeń w typach, tak jak dla eq:

`eq_refl(true)` jest dowodem `t = true` dla dowolnie dużego domkniętego termu `t` typu `bool`

Zalety:

- zmniejszenie rozmiaru termów dowodowych
- łatwiejsze szukanie dowodu
- większa niezawodność: mniejsza zależność od taktyk i dynamicznych błędów typowania

Typy zależne i reguła konwersji

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \equiv A' \qquad \Gamma \vdash A' : s}{\Gamma \vdash M : A'}$$

Pomysł: wykorzystanie obliczeń w typach, tak jak dla `eq`:

`eq_refl(true)` jest dowodem `t = true` dla dowolnie dużego domkniętego termu `t` typu `bool`

Zalety:

- zmniejszenie rozmiaru termów dowodowych
- łatwiejsze szukanie dowodu
- większa niezawodność: mniejsza zależność od taktyk i dynamicznych błędów typowania

Typy zależne i reguła konwersji

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \equiv A' \qquad \Gamma \vdash A' : s}{\Gamma \vdash M : A'}$$

Pomysł: wykorzystanie obliczeń w typach, tak jak dla `eq`:

`eq_refl(true)` jest dowodem `t = true` dla dowolnie dużego domkniętego termu `t` typu `bool`

Zalety:

- zmniejszenie rozmiaru termów dowodowych
- łatwiejsze szukanie dowodu
- większa niezawodność: mniejsza zależność od taktyk i dynamicznych błędów typowania

Typy zależne i reguła konwersji

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \equiv A' \qquad \Gamma \vdash A' : s}{\Gamma \vdash M : A'}$$

Pomysł: wykorzystanie obliczeń w typach, tak jak dla `eq`:

`eq_refl(true)` jest dowodem `t = true` dla dowolnie dużego domkniętego termu `t` typu `bool`

Zalety:

- zmniejszenie rozmiaru termów dowodowych
- łatwiejsze szukanie dowodu
- większa niezawodność: mniejsza zależność od taktyk i dynamicznych błędów typowania

Typy zależne i reguła konwersji

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \equiv A' \qquad \Gamma \vdash A' : s}{\Gamma \vdash M : A'}$$

Pomysł: wykorzystanie obliczeń w typach, tak jak dla `eq`:

`eq_refl(true)` jest dowodem `t = true` dla dowolnie dużego domkniętego termu `t` typu `bool`

Zalety:

- zmniejszenie rozmiaru termów dowodowych
- łatwiejsze szukanie dowodu
- większa niezawodność: mniejsza zależność od taktyk i dynamicznych błędów typowania

Schemat dowodu przez refleksję cd.

Zamiast dowodzić własność `P` używając Ltac napiszemy taktykę w Coqu.

- wprowadzam typ `A` (np. abstract syntax tree)
- definiuję `r2P : A -> Prop`
- znajduję term `p : A` taki, że `r2P p` $\equiv$ `P`
- definiuję `r2b : A -> bool`
- dowodzę lemat `rcorrect : ∀ a : A, r2b a = true -> r2P a`
- jeśli `r2b p` $\equiv$ `true` to `rcorrect p (eq_refl true)` jest typu `P`
- a więc jedyne co trzeba zrobić to obliczyć `r2b p`; te obliczenia nie zajmą miejsca w termie dowodowym.

Tak zrobione są procedury decyzyjne ring, field i omega.

Schemat dowodu przez refleksję cd.

Zamiast dowodzić własność `P` używając Ltac napiszemy taktykę w Coqu.

- wprowadzam typ `A` (np. abstract syntax tree)
- definiuję `r2P:A -> Prop`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P`
- definiuję `r2b:A -> bool`
- dowodzę lemat `rcorrect`:$\forall$ a:A, `r2b a = true -> r2P a`
- jeśli `r2b p` $\equiv$ `true` to `rcorrect p (eq_refl true)` jest typu `P`
- a więc jedyne co trzeba zrobić to obliczyć `r2b p`; te obliczenia nie zajmą miejsca w termie dowodowym.

Tak zrobione są procedury decyzyjne ring, field i omega.

Schemat dowodu przez refleksję cd.

Zamiast dowodzić własność `P` używając Ltac napiszemy taktykę w Coqu.

- wprowadzam typ `A` (np. abstract syntax tree)
- definiuję `r2P:A -> Prop`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P`
- definiuję `r2b:A -> bool`
- dowodzę lemat `rcorrect:`$\forall$ `a:A, r2b a = true -> r2P a`
- jeśli `r2b p` $\equiv$ `true` to `rcorrect p (eq_refl true)` jest typu `P`
- a więc jedyne co trzeba zrobić to obliczyć `r2b p`; te obliczenia nie zajmą miejsca w termie dowodowym.

Tak zrobione są procedury decyzyjne ring, field i omega.

Schemat dowodu przez refleksję cd.

Zamiast dowodzić własność `P` używając Ltac napiszemy taktykę w Coqu.

- wprowadzam typ `A` (np. abstract syntax tree)
- definiuję `r2P:A -> Prop`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P`
- definiuję `r2b:A -> bool`
- dowodzę lemat `rcorrect:`$\forall$ `a:A, r2b a = true -> r2P a`
- jeśli `r2b p` $\equiv$ `true` to `rcorrect p (eq_refl true)` jest typu `P`
- a więc jedyne co trzeba zrobić to obliczyć `r2b p`; te obliczenia nie zajmą miejsca w termie dowodowym.

Tak zrobione są procedury decyzyjne ring, field i omega.

Schemat dowodu przez refleksję cd.

Zamiast dowodzić własność `P` używając Ltac napiszemy taktykę w Coqu.

- wprowadzam typ `A` (np. abstract syntax tree)
- definiuję `r2P:A -> Prop`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P`
- definiuję `r2b:A -> bool`
- dowodzę lemat `rcorrect:`$\forall$ `a:A, r2b a = true -> r2P a`
- jeśli `r2b p` $\equiv$ `true` to `rcorrect p (eq_refl true)` jest typu `P`
- a więc jedyne co trzeba zrobić to obliczyć `r2b p`; te obliczenia nie zajmą miejsca w termie dowodowym.

Tak zrobione są procedury decyzyjne ring, field i omega.

Schemat dowodu przez refleksję cd.

Zamiast dowodzić własność `P` używając Ltac napiszemy taktykę w Coqu.

- wprowadzam typ `A` (np. abstract syntax tree)
- definiuję `r2P:A -> Prop`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P`
- definiuję `r2b:A -> bool`
- dowodzę lemat `rcorrect`:$\forall$ a:A, `r2b a = true -> r2P a`
- jeśli `r2b p` $\equiv$ `true` to `rcorrect p (eq_refl true)` jest typu `P`
- a więc jedyne co trzeba zrobić to obliczyć `r2b p`; te obliczenia nie zajmą miejsca w termie dowodowym.

Tak zrobione są procedury decyzyjne ring, field i omega.

Schemat dowodu przez refleksję cd.

Zamiast dowodzić własność `P` używając Ltac napiszemy taktykę w Coqu.

- wprowadzam typ `A` (np. abstract syntax tree)
- definiuję `r2P:A -> Prop`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P`
- definiuję `r2b:A -> bool`
- dowodzę lemat `rcorrect`:$\forall$ a:A, `r2b a = true -> r2P a`
- jeśli `r2b p` $\equiv$ `true` to `rcorrect p (eq_refl true)` jest typu `P`
- a więc jedyne co trzeba zrobić to obliczyć `r2b p`; te obliczenia nie zajmą miejsca w termie dowodowym.

Tak zrobione są procedury decyzyjne ring, field i omega.

Schemat dowodu przez refleksję cd.

Zamiast dowodzić własność `P` używając Ltac napiszemy taktykę w Coqu.

- wprowadzam typ `A` (np. abstract syntax tree)
- definiuję `r2P:A -> Prop`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P`
- definiuję `r2b:A -> bool`
- dowodzę lemat `rcorrect`:$\forall$ a:A, `r2b a = true -> r2P a`
- jeśli `r2b p` $\equiv$ `true` to `rcorrect p (eq_refl true)` jest typu `P`
- a więc jedyne co trzeba zrobić to obliczyć `r2b p`; te obliczenia nie zajmą miejsca w termie dowodowym.

Tak zrobione są procedury decyzyjne ring, field i omega.

Schemat dowodu przez refleksję cd.

Reifikacja (ang. reification) — znajdowanie $\texttt{p} : \texttt{A}$ takiego, że $\texttt{r2P}\ \texttt{p} \equiv \texttt{P}$:

- ad hoc taktyka lub zewnętrzny program
- p powinien być termem zamkniętym
- p powinien być mały.

Schemat dowodu przez refleksję cd.

Reifikacja (ang. reification) — znajdowanie $\mathtt{p} : \mathtt{A}$ takiego, że $\mathtt{r2P}\ \mathtt{p} \equiv \mathtt{P}$:

- ad hoc taktyka lub zewnętrzny program
- $\mathtt{p}$ powinien być termem zamkniętym
- $\mathtt{p}$ powinien być mały.

Schemat dowodu przez refleksję cd.

Reifikacja (ang. reification) — znajdowanie $\mathtt{p} : \mathtt{A}$ takiego, że $\mathtt{r2P}\ \mathtt{p} \equiv \mathtt{P}$:

- ad hoc taktyka lub zewnętrzny program
- $\mathtt{p}$ powinien być termem zamkniętym
- $\mathtt{p}$ powinien być mały.

Schemat dowodu przez refleksję cd.

Reifikacja (ang. reification) — znajdowanie $\texttt{p} : \texttt{A}$ takiego, że $\texttt{r2P}\ \ \texttt{p} \equiv \texttt{P}$:

- ad hoc taktyka lub zewnętrzny program
- $\texttt{p}$ powinien być termem zamkniętym
- $\texttt{p}$ powinien być mały.

Dowód isEven 256 — Ltac

```
Inductive isEven : nat → Prop :=
| Even_O : isEven O
| Even_SS : ∀ n, isEven n → isEven (S (S n)).

Ltac prove_even := repeat constructor.

Theorem even_256 : isEven 256.
  prove_even.
Qed.

Print even_256.

even_256 =
Even_SS
  (Even_SS
     (Even_SS
        (Even_SS
```

Ten term jest kwadratowej długości względem $n = 256$ z powodu argumentów Even_SS ukrytych przez *I*mplicit Arguments

Dowód `isEven 256` — refleksja

Pomysł: zamiast budować duży term typu `isEven n`, lepiej sprawdzić w czasie liniowym, że `n` jest parzyste i użyć lematu mówiącego, że dla każdego parzystego `n` zachodzi `isEven n`. Patrz plik EvenRefl.v

- wprowadzam typ `A`, tu `A = nat`.
- definiuję `r2P:A -> Prop`, tu `r2P n = isEven n`
- znajduję term `p:A` taki, że `r2P p ≡ P`, tu `p = n`
- definiuję `r2b:A -> bool`, tu `n2b: nat -> bool`
- dowodzę lemat `rcorrect:∀ a:A, r2b a = true -> r2P a`, tu `ecorrect: forall n:nat, en2b n = true -> en2P n`
- jeśli `n2b n ≡ true` to `ecorrect n (eq_refl true)` jest typu `isEven n`

Dowód `isEven 256` — refleksja

Pomysł: zamiast budować duży term typu `isEven n`, lepiej sprawdzić w czasie liniowym, że `n` jest parzyste i użyć lematu mówiącego, że dla każdego parzystego `n` zachodzi `isEven n`. Patrz plik EvenRefl.v

- wprowadzam typ `A`, tu `A = nat`.
- definiuję `r2P:A -> Prop`, tu `r2P n = isEven n`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P`, tu `p = n`
- definiuję `r2b:A -> bool`, tu `n2b: nat -> bool`
- dowodzę lemat `rcorrect:`$\forall$ `a:A, r2b a = true -> r2P a`, tu `ecorrect: forall n:nat, en2b n = true -> en2P n`
- jeśli `n2b n` $\equiv$ `true` to `ecorrect n (eq_refl true)` jest typu `isEven n`

Dowód `isEven 256` — refleksja

Pomysł: zamiast budować duży term typu `isEven n`, lepiej sprawdzić w czasie liniowym, że `n` jest parzyste i użyć lematu mówiącego, że dla każdego parzystego `n` zachodzi `isEven n`. Patrz plik EvenRefl.v

- wprowadzam typ `A`, tu `A = nat`.
- definiuję `r2P:A -> Prop`, tu `r2P n = isEven n`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P`, tu `p = n`
- definiuję `r2b:A -> bool`, tu `n2b: nat -> bool`
- dowodzę lemat `rcorrect:`$\forall$ `a:A, r2b a = true -> r2P a`, tu `ecorrect: forall n:nat, en2b n = true -> en2P n`
- jeśli `n2b n` $\equiv$ `true` to `ecorrect n (eq_refl true)` jest typu `isEven n`

Dowód `isEven 256` — refleksja

Pomysł: zamiast budować duży term typu `isEven n`, lepiej sprawdzić w czasie liniowym, że `n` jest parzyste i użyć lematu mówiącego, że dla każdego parzystego `n` zachodzi `isEven n`. Patrz plik EvenRefl.v

- wprowadzam typ `A`, tu `A = nat`.
- definiuję `r2P:A -> Prop`, tu `r2P n = isEven n`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P`, tu `p = n`
- definiuję `r2b:A -> bool`, tu `n2b: nat -> bool`
- dowodzę lemat `rcorrect:`$\forall$ `a:A, r2b a = true -> r2P a`, tu `ecorrect: forall n:nat, en2b n = true -> en2P n`
- jeśli `n2b n` $\equiv$ `true` to `ecorrect n (eq_refl true)` jest typu `isEven n`

Dowód `isEven 256` — refleksja

Pomysł: zamiast budować duży term typu `isEven n`, lepiej sprawdzić w czasie liniowym, że `n` jest parzyste i użyć lematu mówiącego, że dla każdego parzystego `n` zachodzi `isEven n`. Patrz plik EvenRefl.v

- wprowadzam typ `A`, tu `A = nat`.
- definiuję `r2P:A -> Prop`, tu `r2P n = isEven n`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P`, tu `p = n`
- definiuję `r2b:A -> bool`, tu `n2b: nat -> bool`
- dowodzę lemat `rcorrect:`$\forall$ `a:A, r2b a = true -> r2P a`, tu `ecorrect: forall n:nat, en2b n = true -> en2P n`
- jeśli `n2b n` $\equiv$ `true` to `ecorrect n (eq_refl true)` jest typu `isEven n`

Dowód `isEven 256` — refleksja

Pomysł: zamiast budować duży term typu `isEven n`, lepiej sprawdzić w czasie liniowym, że `n` jest parzyste i użyć lematu mówiącego, że dla każdego parzystego `n` zachodzi `isEven n`. Patrz plik EvenRefl.v

- wprowadzam typ `A`, tu `A = nat`.
- definiuję `r2P:A -> Prop`, tu `r2P n = isEven n`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P`, tu `p = n`
- definiuję `r2b:A -> bool`, tu `n2b: nat -> bool`
- dowodzę lemat `rcorrect`:$\forall$ a:A, `r2b a = true -> r2P a`, tu `ecorrect: forall n:nat, en2b n = true -> en2P n`
- jeśli `n2b n` $\equiv$ `true` to `ecorrect n (eq_refl true)` jest typu `isEven n`

Dowód `isEven 256` — refleksja

Pomysł: zamiast budować duży term typu `isEven n`, lepiej sprawdzić w czasie liniowym, że `n` jest parzyste i użyć lematu mówiącego, że dla każdego parzystego `n` zachodzi `isEven n`. Patrz plik EvenRefl.v

- wprowadzam typ `A`, tu `A = nat`.
- definiuję `r2P:A -> Prop`, tu `r2P n = isEven n`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P`, tu `p = n`
- definiuję `r2b:A -> bool`, tu `n2b: nat -> bool`
- dowodzę lemat `rcorrect`:$\forall$ a:A, `r2b a = true -> r2P a`, tu `ecorrect: forall n:nat, en2b n = true -> en2P n`
- jeśli `n2b n` $\equiv$ `true` to `ecorrect n (eq_refl true)` jest typu `isEven n`

Dowód `isEven 256` — refleksja 2

W CPDT jest inny dowód przez refleksję: patrz plik EvenReflCpdt.v

- `check_even n` sprawdza parzystość dla konkretnego `n` i buduje dowód dla tego `n`,
- `partialOut` wyjmuje ten dowód;
- jest prościej (nie trzeba dowodzić `ecorrect`) ale dowód jest duży.

Dowód `isEven 256` — refleksja 2

W CPDT jest inny dowód przez refleksję: patrz plik EvenReflCpdt.v

- `check_even n` sprawdza parzystość dla konkretnego `n` i buduje dowód dla tego `n`,
- `partialOut` wyjmuje ten dowód;
- jest prościej (nie trzeba dowodzić `ecorrect`) ale dowód jest duży.

Dowód `isEven 256` — refleksja 2

W CPDT jest inny dowód przez refleksję: patrz plik EvenReflCpdt.v

- `check_even n` sprawdza parzystość dla konkretnego `n` i buduje dowód dla tego `n`,
- `partialOut` wyjmuje ten dowód;
- jest prościej (nie trzeba dowodzić `ecorrect`) ale dowód jest duży.

Trywialne tautologie

Formuły zbudowane z `True`, koniunkcji, alternatywy i implikacji:

```
Theorem true_galore : (True ∧ True) → (True ∨ (True ∧ (True → True))).
  tauto.
Qed.

Print true_galore.
```

```
true_galore =
fun H : True ∧ True ⇒
and_ind (fun _ _ : True ⇒ or_introl (True ∧ (True → True)) I) H
     : True ∧ True → True ∨ True ∧ (True → True)
```

Trywialne tautologie

Formuły zbudowane z `True`, koniunkcji, alternatywy i implikacji:

```
Theorem true_galore : (True ∧ True) → (True ∨ (True ∧ (True → True))).
  tauto.
Qed.

Print true_galore.
```

```
true_galore =
fun H : True ∧ True ⇒
and_ind (fun _ _ : True ⇒ or_introl (True ∧ (True → True)) I) H
     : True ∧ True → True ∨ True ∧ (True → True)
```

Trywialne tautologie cd.

Typ reprezentujący składnię abstrakcyjną trywialnych formuł:

```
Inductive taut : Set :=
| TautTrue : taut
| TautAnd : taut → taut → taut
| TautOr : taut → taut → taut
| TautImp : taut → taut → taut.
```

Trywialne tautologie cd.

```
Fixpoint tautDenote (t : taut) : Prop :=
  match t with
    | TautTrue ⇒ True
    | TautAnd t1 t2 ⇒ tautDenote t1 ∧ tautDenote t2
    | TautOr t1 t2 ⇒ tautDenote t1 ∨ tautDenote t2
    | TautImp t1 t2 ⇒ tautDenote t1 → tautDenote t2
  end.
```

```
Theorem tautTrue : ∀ t, tautDenote t.
  induction t; crush.
Qed.
```

Trywialne tautologie cd.

```
Fixpoint tautDenote (t : taut) : Prop :=
  match t with
    | TautTrue ⇒ True
    | TautAnd t1 t2 ⇒ tautDenote t1 ∧ tautDenote t2
    | TautOr t1 t2 ⇒ tautDenote t1 ∨ tautDenote t2
    | TautImp t1 t2 ⇒ tautDenote t1 → tautDenote t2
  end.

Theorem tautTrue : ∀ t, tautDenote t.
  induction t; crush.
Qed.
```

Trywialne tautologie — refleksja

- wprowadzam typ `A`; tu `taut`
- definiuję `r2P:A -> Prop`; tu `tautDenote`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P` — na następnych slajdach
- definiuję `r2b:A -> bool`; nie potrzebuję bo dla wszystkich elementów `taut` to `true`
- dowodzę lemat `rcorrect`$\forall$ a:A, `r2b a = true -> r2P a`; tu `tautTrue`

Trywialne tautologie — refleksja

- wprowadzam typ `A`; tu `taut`
- definiuję `r2P:A -> Prop`; tu `tautDenote`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P` — na następnych slajdach
- definiuję `r2b:A -> bool`; nie potrzebuję bo dla wszystkich elementów `taut` to `true`
- dowodzę lemat `rcorrect`$\forall$ a:A, `r2b a = true -> r2P a`; tu `tautTrue`

Trywialne tautologie — refleksja

- wprowadzam typ `A`; tu `taut`
- definiuję `r2P:A -> Prop`; tu `tautDenote`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P` — na następnych slajdach
- definiuję `r2b:A -> bool`; nie potrzebuję bo dla wszystkich elementów `taut` to `true`
- dowodzę lemat `rcorrect`$\forall$ a:A, `r2b a = true -> r2P a`; tu `tautTrue`

Trywialne tautologie — refleksja

- wprowadzam typ `A`; tu `taut`
- definiuję `r2P:A -> Prop`; tu `tautDenote`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P` — na następnych slajdach
- definiuję `r2b:A -> bool`; nie potrzebuję bo dla wszystkich elementów `taut` to `true`
- dowodzę lemat `rcorrect`$\forall$ `a:A, r2b a = true -> r2P a`; tu `tautTrue`

Trywialne tautologie — refleksja

- wprowadzam typ `A`; tu `taut`
- definiuję `r2P:A -> Prop`; tu `tautDenote`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P` — na następnych slajdach
- definiuję `r2b:A -> bool`; nie potrzebuję bo dla wszystkich elementów `taut` to `true`
- dowodzę lemat `rcorrect`$\forall$ a:A, `r2b a = true -> r2P a`; tu `tautTrue`

Trywialne tautologie — reifikacja

```
Ltac tautReify P :=
  match P with
    | True ⇒ TautTrue
    | ?P1 ∧ ?P2 ⇒
      let t1 := tautReify P1 in
      let t2 := tautReify P2 in
        constr:(TautAnd t1 t2)
    | ?P1 ∨ ?P2 ⇒
      let t1 := tautReify P1 in
      let t2 := tautReify P2 in
        constr:(TautOr t1 t2)
    | ?P1 → ?P2 ⇒
      let t1 := tautReify P1 in
      let t2 := tautReify P2 in
        constr:(TautImp t1 t2)
  end.
```

Trywialne tautologie — procedura decyzyjna

```
Ltac obvious :=
  match goal with
    | [ ⊢ ?P ] ⇒
      let t := tautReify P in
        exact (tautTrue t)
  end.

Theorem true_galore' : (True ∧ True) → (True ∨ (True ∧ (True → True))).
  obvious.
Qed.

Print true_galore'.

true_galore' =
tautTrue
  (TautImp (TautAnd TautTrue TautTrue)
     (TautOr TautTrue (TautAnd TautTrue (TautImp TautTrue TautTrue))))
     : True ∧ True → True ∨ True ∧ (True → True)
```

Trywialne tautologie — procedura decyzyjna

```
Ltac obvious :=
  match goal with
    | [ ⊢ ?P ] ⇒
      let t := tautReify P in
        exact (tautTrue t)
  end.

Theorem true_galore' : (True ∧ True) → (True ∨ (True ∧ (True →
True))).
  obvious.
Qed.

Print true_galore'.

true_galore' =
tautTrue
  (TautImp (TautAnd TautTrue TautTrue)
     (TautOr TautTrue (TautAnd TautTrue (TautImp TautTrue
TautTrue))))
     : True ∧ True → True ∨ True ∧ (True → True)
```

Nietrywialne formuły atomowe

Patrz plik FormRefl.v

- wprowadzam typ `A`; tu `form`. Przypadek `Var` "łapie" formuły atomowe, które nie są postaci `True`, `False`
- definiuję `r2P:A -> Prop`; tu `f2P e f`. Oprócz `f:form` potrzebujemy środowiska zapisanego w liście `e`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P`; tu jest to para term typu `form` i lista tego co było na zmiennych. Obie znajdywane przez taktykę `env_form`
- definiuję `r2b:A -> bool`; to `f2b` — ona zakłada, że zmienne interpretujemy jako `true`
- dowodzę lemat `rcorrect`:$\forall$ a:A, `r2b a = true -> r2P a`; tu `fcorrect: forall (e:env) (f:form), (allTrue e) -> (f2b f=true) -> (f2P e f)`
- `allTrue e` sprawdza, że wszystkie atomowe formuły są prawdziwe.

Nietrywialne formuły atomowe

Patrz plik FormRefl.v

- wprowadzam typ `A`; tu `form`. Przypadek `Var` "łapie" formuły atomowe, które nie są postaci `True`, `False`
- definiuję `r2P:A -> Prop`; tu `f2P e f`. Oprócz `f:form` potrzebujemy środowiska zapisanego w liście `e`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P`; tu jest to para term typu `form` i lista tego co było na zmiennych. Obie znajdywane przez taktykę `env_form`
- definiuję `r2b:A -> bool`; to `f2b` — ona zakłada, że zmienne interpretujemy jako `true`
- dowodzę lemat `rcorrect:`$\forall$ `a:A, r2b a = true -> r2P a`; tu `fcorrect: forall (e:env) (f:form), (allTrue e) -> (f2b f=true) -> (f2P e f)`
- `allTrue e` sprawdza, że wszystkie atomowe formuły są prawdziwe.

Nietrywialne formuły atomowe

Patrz plik FormRefl.v

- wprowadzam typ `A`; tu `form`. Przypadek `Var` "łapie" formuły atomowe, które nie są postaci `True`, `False`
- definiuję `r2P:A -> Prop`; tu `f2P e f`. Oprócz `f:form` potrzebujemy środowiska zapisanego w liście `e`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P`; tu jest to para term typu `form` i lista tego co było na zmiennych. Obie znajdywane przez taktykę `env_form`
- definiuję `r2b:A -> bool`; to `f2b` — ona zakłada, że zmienne interpretujemy jako `true`
- dowodzę lemat `rcorrect:`$\forall$ `a:A, r2b a = true -> r2P a`; tu `fcorrect: forall (e:env) (f:form), (allTrue e) -> (f2b f=true) -> (f2P e f)`
- `allTrue e` sprawdza, że wszystkie atomowe formuły są prawdziwe.

Nietrywialne formuły atomowe

Patrz plik FormRefl.v

- wprowadzam typ `A`; tu `form`. Przypadek `Var` "łapie" formuły atomowe, które nie są postaci `True`, `False`
- definiuję `r2P:A -> Prop`; tu `f2P e f`. Oprócz `f:form` potrzebujemy środowiska zapisanego w liście `e`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P`; tu jest to para term typu `form` i lista tego co było na zmiennych. Obie znajdywane przez taktykę `env_form`
- definiuję `r2b:A -> bool`; to `f2b` — ona zakłada, że zmienne interpretujemy jako `true`
- dowodzę lemat `rcorrect:`$\forall$ `a:A, r2b a = true -> r2P a`; tu `fcorrect: forall (e:env) (f:form), (allTrue e) -> (f2b f=true) -> (f2P e f)`
- `allTrue e` sprawdza, że wszystkie atomowe formuły są prawdziwe.

Nietrywialne formuły atomowe

Patrz plik FormRefl.v

- wprowadzam typ `A`; tu `form`. Przypadek `Var` "łapie" formuły atomowe, które nie są postaci `True`, `False`
- definiuję `r2P:A -> Prop`; tu `f2P e f`. Oprócz `f:form` potrzebujemy środowiska zapisanego w liście `e`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P`; tu jest to para term typu `form` i lista tego co było na zmiennych. Obie znajdywane przez taktykę `env_form`
- definiuję `r2b:A -> bool`; to `f2b` — ona zakłada, że zmienne interpretujemy jako `true`
- dowodzę lemat `rcorrect`:$\forall$ a:A, `r2b a = true -> r2P a`; tu `fcorrect: forall (e:env) (f:form), (allTrue e) -> (f2b f=true) -> (f2P e f)`
- `allTrue e` sprawdza, że wszystkie atomowe formuły są prawdziwe.

Nietrywialne formuły atomowe

Patrz plik FormRefl.v

- wprowadzam typ `A`; tu `form`. Przypadek `Var` "łapie" formuły atomowe, które nie są postaci `True`, `False`
- definiuję `r2P:A -> Prop`; tu `f2P e f`. Oprócz `f:form` potrzebujemy środowiska zapisanego w liście `e`
- znajduję term `p:A` taki, że `r2P p` $\equiv$ `P`; tu jest to para term typu `form` i lista tego co było na zmiennych. Obie znajdywane przez taktykę `env_form`
- definiuję `r2b:A -> bool`; to `f2b` — ona zakłada, że zmienne interpretujemy jako `true`
- dowodzę lemat `rcorrect`:$\forall$ a:A, `r2b a = true -> r2P a`; tu `fcorrect: forall (e:env) (f:form), (allTrue e) -> (f2b f=true) -> (f2P e f)`
- `allTrue e` sprawdza, że wszystkie atomowe formuły są prawdziwe.

Nietrywialne formuły atomowe cd.

Dowód dowolnej formuły sprowadza się do:

```
reify.
apply fcorrect.


+ unfold allTrue.
iterateAssumptions. (*auto radzi sobie z formułami atomowymi
*)


+ reflexivity.
Qed.
```

Nietrywialne formuły atomowe – użycie quote

W Coqu jest taktyka reifikacyjna `quote`. Patrz plik FormReflQuote,v

- `Require Import Quote.`
- typ `Var` musi być `index ->form`
- tym razem taktyka `iterateAssumptions` "zostawia" użytkownikowi formuły atomowe do udowodnienia.

Nietrywialne formuły atomowe – użycie quote

W Coqu jest taktyka reifikacyjna `quote`. Patrz plik FormReflQuote,v

- `Require Import Quote.`
- typ `Var` musi być `index ->form`
- tym razem taktyka `iterateAssumptions` "zostawia" użytkownikowi formuły atomowe do udowodnienia.

Nietrywialne formuły atomowe – użycie quote

W Coqu jest taktyka reifikacyjna `quote`. Patrz plik FormReflQuote,v

- `Require Import Quote.`
- typ `Var` musi być `index ->form`
- tym razem taktyka `iterateAssumptions` "zostawia" użytkownikowi formuły atomowe do udowodnienia.

Dowodzenie równości w monoidzie

```
Section monoid.
  Variable A : Set.
  Variable e : A.
  Variable f : A → A → A.

  Infix "+" := f.

  Hypothesis assoc : ∀ a b c, (a + b) + c = a + (b + c).
  Hypothesis identl : ∀ a, e + a = a.
  Hypothesis identr : ∀ a, a + e = a.
```

Dowodzenie równości w monoidzie — pomysł

Pomysł:

- wyrażenia `w1` i `w2` w celu dowodowym `w1=w2` zostaną przetłumaczone na typ `mexp`,
- termy typu mexp można normalizować; to będzie lista zmiennych, bo dodawanie jest łączne,
- udowodnimy, że znaczenie termu przed normalizacją i po jest takie samo,
- konsekwentnie, `w1=w2` zachodzi jeśli odpowiadające im wyrażenia typu `mexp` będą miały identyczne formy normalne.

Dowodzenie równości w monoidzie — pomysł

Pomysł:

- wyrażenia `w1` i `w2` w celu dowodowym `w1=w2` zostaną przetłumaczone na typ `mexp`,
- termy typu mexp można normalizować; to będzie lista zmiennych, bo dodawanie jest łączne,
- udowodnimy, że znaczenie termu przed normalizacją i po jest takie samo,
- konsekwentnie, `w1=w2` zachodzi jeśli odpowiadające im wyrażenia typu `mexp` będą miały identyczne formy normalne.

Dowodzenie równości w monoidzie — pomysł

Pomysł:

- wyrażenia `w1` i `w2` w celu dowodowym `w1=w2` zostaną przetłumaczone na typ `mexp`,
- termy typu mexp można normalizować; to będzie lista zmiennych, bo dodawanie jest łączne,
- udowodnimy, że znaczenie termu przed normalizacją i po jest takie samo,
- konsekwentnie, `w1=w2` zachodzi jeśli odpowiadające im wyrażenia typu `mexp` będą miały identyczne formy normalne.

Dowodzenie równości w monoidzie — pomysł

Pomysł:

- wyrażenia `w1` i `w2` w celu dowodowym `w1=w2` zostaną przetłumaczone na typ `mexp`,
- termy typu mexp można normalizować; to będzie lista zmiennych, bo dodawanie jest łączne,
- udowodnimy, że znaczenie termu przed normalizacją i po jest takie samo,
- konsekwentnie, `w1=w2` zachodzi jeśli odpowiadające im wyrażenia typu `mexp` będą miały identyczne formy normalne.

Dowodzenie równości w monoidzie cd

```
Inductive mexp : Set :=
| Ident : mexp
| Var : A → mexp
| Op : mexp → mexp → mexp.
```

```
Fixpoint mdenote (me : mexp) : A :=
  match me with
    | Ident ⇒ e
    | Var v ⇒ v
    | Op me1 me2 ⇒ mdenote me1 + mdenote me2
  end.
```

Dowodzenie równości w monoidzie cd

```
Inductive mexp : Set :=
| Ident : mexp
| Var : A → mexp
| Op : mexp → mexp → mexp.
```

```
Fixpoint mdenote (me : mexp) : A :=
  match me with
    | Ident ⇒ e
    | Var v ⇒ v
    | Op me1 me2 ⇒ mdenote me1 + mdenote me2
  end.
```

Dowodzenie równości w monoidzie cd.

```
Fixpoint mldenote (ls : list A) : A :=
  match ls with
    | nil ⇒ e
    | x :: ls' ⇒ x + mldenote ls'
  end.
```

```
Fixpoint flatten (me : mexp) : list A :=
  match me with
    | Ident ⇒ nil
    | Var x ⇒ x :: nil
    | Op me1 me2 ⇒ flatten me1 ++ flatten me2
  end.
```

Dowodzenie równości w monoidzie cd.

```
Fixpoint mldenote (ls : list A) : A :=
  match ls with
    | nil ⇒ e
    | x :: ls' ⇒ x + mldenote ls'
  end.
```

```
Fixpoint flatten (me : mexp) : list A :=
  match me with
    | Ident ⇒ nil
    | Var x ⇒ x :: nil
    | Op me1 me2 ⇒ flatten me1 ++ flatten me2
  end.
```

Dowodzenie równości w monoidzie — poprawność normalizacji

```
Lemma flatten_correct' : ∀ ml2 ml1,
  mldenote ml1 + mldenote ml2 = mldenote (ml1 ++ ml2).
  induction ml1; crush.
Qed.

Theorem flatten_correct : ∀ me, mdenote me = mldenote (flatten me).
  Hint Resolve flatten_correct'.

  induction me; crush.
Qed.
```

```
Theorem monoid_reflect : ∀ me1 me2,
  mldenote (flatten me1) = mldenote (flatten me2)
  → mdenote me1 = mdenote me2.
  intros; repeat rewrite flatten_correct; assumption.
Qed.
```

Dowodzenie równości w monoidzie — poprawność normalizacji

```
Lemma flatten_correct' : ∀ ml2 ml1,
  mldenote ml1 + mldenote ml2 = mldenote (ml1 ++ ml2).
  induction ml1; crush.
Qed.

Theorem flatten_correct : ∀ me, mdenote me = mldenote (flatten me).
  Hint Resolve flatten_correct'.

  induction me; crush.
Qed.

Theorem monoid_reflect : ∀ me1 me2,
  mldenote (flatten me1) = mldenote (flatten me2)
  → mdenote me1 = mdenote me2.
  intros; repeat rewrite flatten_correct; assumption.
Qed.
```

Dowodzenie równości w monoidzie — reifikacja

```
Ltac reify me :=
  match me with
    | e ⇒ Ident
    | ?me1 + ?me2 ⇒
      let r1 := reify me1 in
      let r2 := reify me2 in
        constr:(Op r1 r2)
    | _ ⇒ constr:(Var me)
  end.

Ltac monoid :=
  match goal with
    | [ ⊢ ?me1 = ?me2 ] ⇒
      let r1 := reify me1 in
      let r2 := reify me2 in
        change (mdenote r1 = mdenote r2);
          apply monoid_reflect; simpl
  end.
```

Dowodzenie równości w monoidzie — reifikacja

```
Ltac reify me :=
  match me with
    | e ⇒ Ident
    | ?me1 + ?me2 ⇒
      let r1 := reify me1 in
      let r2 := reify me2 in
        constr:(Op r1 r2)
    | _ ⇒ constr:(Var me)
  end.

Ltac monoid :=
  match goal with
    | [ ⊢ ?me1 = ?me2 ] ⇒
      let r1 := reify me1 in
      let r2 := reify me2 in
        change (mdenote r1 = mdenote r2);
          apply monoid_reflect; simpl
  end.
```

Dowodzenie równości w monoidzie — przykład

```
Theorem t1 : ∀ a b c d, a + b + c + d = a + (b + c) + d.
  intros; monoid.

============================
 a + (b + (c + (d + e))) = a + (b + (c + (d + e)))

  reflexivity.
Qed.

Print t1.

t1 =
fun a b c d : A ⇒
monoid_reflect (Op (Op (Op (Var a) (Var b)) (Var c)) (Var d))
  (Op (Op (Var a) (Op (Var b) (Var c))) (Var d))
  (eq_refl (a + (b + (c + (d + e)))))
     : ∀ a b c d : A, a + b + c + d = a + (b + c) + d
```

Dowodzenie równości w monoidzie — przykład

```
Theorem t1 : ∀ a b c d, a + b + c + d = a + (b + c) + d.
  intros; monoid.

============================
 a + (b + (c + (d + e))) = a + (b + (c + (d + e)))

  reflexivity.
Qed.

Print t1.

t1 =
fun a b c d : A ⇒
monoid_reflect (Op (Op (Op (Var a) (Var b)) (Var c)) (Var d))
  (Op (Op (Var a) (Op (Var b) (Var c))) (Var d))
  (eq_refl (a + (b + (c + (d + e)))))
     : ∀ a b c d : A, a + b + c + d = a + (b + c) + d
```

Dowodzenie równości w monoidzie — przykład

```
Theorem t1 : ∀ a b c d, a + b + c + d = a + (b + c) + d.
  intros; monoid.

============================
 a + (b + (c + (d + e))) = a + (b + (c + (d + e)))

  reflexivity.
Qed.

Print t1.
```

```
t1 =
fun a b c d : A ⇒
monoid_reflect (Op (Op (Op (Var a) (Var b)) (Var c)) (Var d))
  (Op (Op (Var a) (Op (Var b) (Var c))) (Var d))
  (eq_refl (a + (b + (c + (d + e)))))
     : ∀ a b c d : A, a + b + c + d = a + (b + c) + d
```