Deduktivni sistemi in meta-programiranje

I.Savnik, FAMNIT, 2012/13

Meta-programiranje

Programi, ki analizirajo, spreminjajo ter programirajo same sebe ali druge programe.

1) Statično metaprogramiranje
   - Transformacije programov.
   - Prevajalniki, razčlenjevalniki in analiza jezikov.
2) Dinamično metaprogramiranje
   - Introspekcija.
   - Generativnost.
   - Interna uporaba interpreterjev.

Statično meta-programiranje

- Makro jeziki
  - PJ omogoča pisanje generičnih makrojev.
    - Instance makrojev definirajo kodo.
  - Primeri: C in C++ pred-procesor, M4
- Orodje za konstrukcijo prevajalnikov
  - Abstraktna specifikacija jezika.
    - Slovnica, analize, prevajanje.
  - Primeri: lex, yacc, bison
- Orodja za razčlenjevanje in analizo jezikov
  - Primeri: SAX (XML), Trini (RDF)

Dinamično meta-programiranje

- Uporaba internih gradnikov PJ
  - Introspekcija.
    - Poizvedovanje po lastnostih razredov: dedovanje, metode, ...
    - Razred je objekt.
  - Dinamična kreacija, spreminjanje objektov in razredov
    - Objekte in razrede lahko spreminjamo v času izvajanja.
    - Dostop in delo z imenskimi prostori..
  - Generativnost.
    - Parametrizirani razredi: Ocaml, C++, Ruby.
    - Template metaprogramming (C++)
  - Dedovanje.
    - Ruby - povezovanje razredov, objektov z starši.

Dinamično meta-programiranje

- Interna uporaba interpreterja in okolja PJ.
  - Program sestavlja programe, ki jih sam evaluira.
  - Interpretacija izrazov med izvajanjem.
    - eval v Lisp
    - eval v Perl
    - assert v Prolog
  - Refleksivnost jezika.
    - Lahko interpretiramo jezik s samim jezikom.

Deduktivni sistemi

Deduktivno sklepanje:

- sklepanje, ki deducira (izpelje) deduktivne argumente

Deduktivni argumenti:

- poskusi prikaza zaključka kot posledico množice premis

Deduktivni argument:

- veljaven, če zaključki sledijo iz premis oz. če zaključek mora biti resničen v primeru, da so premise resnične

1. Vsi ljudje so smrtni.
2. Socrates je človek.
3. Torej, Socrates je smrten.

Twelf

Twelf je jezik s katerim lahko

- zasnujemo,
- implementiramo in
- dokazujemo lastnosti dedukcijskih sistemov

s katerimi modeliramo programske jezike.

Twelf (2)

Twelf je zasnovan na:

- logiki višjega reda (LF) in na
- principih logičnega programiranja.

Logical Frameworks (LF) je formalni sistem,ki vsebuje:

- lambda račun
- splošno rekurzijo
- LC višjega reda
- CR in SN za LF
- preverjanje tipov ni odločljivo
- tipe
- rekurzivne tipe
- induktivni meta-logični dokazovalnik izrekov

Predstavitev naravnih števil

```
z nat

X nat
-------
s(X) nat

plus(z,Y,Y)

  plus(X,Y,Z)
-------------
plus(s(X),Y,s(Z))

sub(X,z,X)

sub(z,Y,z)

   sub(X,Y,Z)
---------------
sub(s(X),s(Y),Z)
```

Predstavitev naravnih števil (2)

```
nat : type.    %name nat X.
z : nat.
s : nat -> nat.

bool: type.    %name bool B.
true: bool.
false: bool.

plus : nat -> nat -> nat -> type.  %name plus P.
p_z : plus z Y Y.
p_s : plus (s X) Y (s Z)
       <- plus X Y Z.

sub: nat -> nat -> nat -> type.    %name sub S.
%mode sub +X +Y -Z.

s_z1 : sub X z X.
s_z2 : sub z Y z.
s_ss : sub (s X) (s Y) Z
       <- sub X Y Z.
```

Predstavitev naravnih števil (3)

Največji skupni delitelj

```
gcd(X,z,X)

gcd(z,X,X)

X<Y   Y'=Y-X   gcd(X,Y',Z)
gcd(X,Y,Z)

X>=Y  X'=X-Y   gcd(X',Y,Z)
gcd(X,Y,Z)
```

Predstavitev naravnih števil (4)

```
less: nat -> nat -> bool -> type.        %name less L.
%mode less +X +Y -Z.
l_z1 : less z (s X) true.
l_z2 : less X z false.
l_ss : less (s X) (s Y) B
       <- less X Y B.

gcd: nat -> nat -> nat -> type.          %name gcd G.
%mode gcd +X +Y -Z.

gcd_z1: gcd z Y Y.
gcd_z2: gcd X z X.

gcd_s1: gcd (s X) (s Y) Z
        <- less (s X) (s Y) true
        <- sub (s Y) (s X) Y'
        <- gcd (s X) Y' Z.

gcd_s1: gcd (s X) (s Y) Z
        <- less (s X)(s Y) false
        <- sub (s X) (s Y) X'
        <- gcd X' (s Y)  Z.
```

Odločljivost programov

- Funkcija definira relacijo med vhodom in izhodom
  - Primer: plus X Y Z
  - Za vsak X:nat,Y:nat obstaja Z:nat in D: X+Y=Z
- Totalnost (cmu)::
  - Za vsak vhod zna postopek izračunati izhode.
  - Rekurzivni jezik (Chomsky)
  - Ne preverjamo funkcije: imamo lahko več različnih vrednosti Z
  - Preverjanje funkcije = enoličnost, deterministočnost

Odločljivost programov (2)

```
% declare totality assertion
%mode plus +N1 +N2 -N3.
%worlds () (plus _ _ _).

% check totality assertion
%total N1 (plus N1 _ _).
```

%mode
Deklaracija določa način kvant. družine tipov. N1 in N2 sta kvant. univerzalno (+), N3 sta kvant. eksistenčno (-).

%worlds
Mehanizem za definicijo konteksta za določen predikat. V našem primeru prazen kontekst zadošča za tipe s katerimi so predstavljena naravna števila in seštevanje.

%total
Vsaka uporaba načinov kvat. zagotavlja zaključitev evaluacije.
Twelf to preveri z indukcijo po kanoničnih N1 tipa nat.
Indeks za indukcijo je določen s pozicijo spremenljivke imenovane v %total.

Lambda račun

$T ::= unit \mid T_1 \rightarrow T_2$

```
tp : type.
arrow : tp -> tp -> tp.
unit : tp.
```

$E ::= x \mid <> \mid \lambda x{:}T.e \mid e_1\ e_2$

```
tm : type.
empty : tm.
app : tm -> tm -> tm.
lam : tp -> (tm -> tm) -> tm.
```

Statična semantika

$$\frac{}{\Gamma \vdash \langle\rangle : \mathsf{unit}}\ \textsc{of-empty}$$

$$\frac{}{\Gamma, x{:}T, \Gamma' \vdash x : T}\ \textsc{of-var}$$

$$\frac{\Gamma, x{:}T_2 \vdash e : T}{\Gamma \vdash \lambda x.e : T_2 \to T}\ \textsc{of-abs}$$

$$\frac{\Gamma \vdash e_1 : T_2 \to T \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1\ e_2 : T}\ \textsc{of-app}$$

```
of : tm -> tp -> type.
of-empty : of empty unit.
of-lam : of (lam T2 ([x] E x)) (arrow T2 T)
          <- ({x: tm} of x T2 -> of (E x) T).
of-app : of (app E1 E2) T
          <- of E1 (arrow T2 T)
          <- of E2 T2.
```

Dinamična semantika

$$\frac{}{\texttt{empty value}}\ \texttt{value-empty} \qquad \frac{}{\lambda x{:}T.e\ \texttt{value}}\ \texttt{value-lam}$$

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}\ \texttt{step-app-1} \qquad \frac{e_1\ \texttt{val} \quad e_2 \to e_2'}{e_1\ e_2 \to e_1\ e_2'}\ \texttt{step-app-2}$$

$$\frac{e_2\ \texttt{val}}{(\lambda x{:}T.e)\ e_2 \to [e_2/x]\ e}\ \texttt{step-app-beta}$$

Dinamična semantika (2)

```
value : tm -> type.
value-empty : value empty.
value-lam : value (lam T ([x] E x)).

step : tm -> tm -> type.
step-app-1 : step (app E1 E2) (app E1' E2)
               <- step E1 E1'.
step-app-2 : step (app E1 E2) (app E1 E2')
               <- value E1
               <- step E2 E2'.
step-app-beta : step (app (lam T2 ([x] E x)) E2) (E E2)
                  <- value E2.
```

Asistent za dokazovanje: COQ

Coq uporablja:

1) logiko višjega reda in
2) bogat funkcijski programski jezik.

Coq omogoča:

- definicijo funkcij in predikatov ter učinkovito implementacijo le-teh,
- izražanje matematičnih izrekov in specifikacij programske opreme,
- iteraktivni razvoj dokazov izrekov,
- strojno dokazovanje z relativno majhnim certifikacijskim jedrom, in
- ekstrakcija certifikacijskih programov za jezike kot so Ocaml, Haskell ali Scheme.

Pisanje formul

```
Check True.
True : Prop

Check False.
False : Prop

Check 3.
3 : nat
```

```
Check (3+4).
3 + 4 : nat

Check (3=5).
3=5 : Prop

Check (3,4).
(3,4) : nat * nat

Check ((3=5)/\True).
3 = 5 /\ True : Prop
```

```
Check nat -> Prop.
nat -> Prop : Type

Check (3 <= 6).
3 <= 6 : Prop

Check (fun x:nat => x = 3).
fun x : nat => x = 3 : nat -> Prop

Check (forall x:nat, x < 3 \/ (exists y:nat, x = y + 3)).
forall x : nat, x < 3 \/ (exists y : nat, x = y + 3) : Prop
```

Definicija novih konstant

```
Definition example1 := fun x : nat => x*x+2*x+1.
Definition example1 (x : nat) := x*x+2*x+1.
```

Funkcijo lahko zdaj uporabljamo v drugih ukazih.

```
Check example1.
example1 : nat -> nat

Eval compute in example1 1.
= 4 : nat
```

Računanje z naravnimi števili

```
Definition is_zero (n:nat) :=
  matach n with
    0 => true
  | S p => false
  end.
```

```
Fixpoint sum_n n :=
  match n with
    0 => 0
  | S p => p + sum_n p
  end.
```

```
Print pred.
pred = fun n : nat =>
  match n with | 0 => n | S u => u end
     : nat -> nat
```

```
Fixpoint evenb n :=
  match n with
    0 => true
  | 1 => false
  | S (S p) => evenb p
  end.
```

Delo s seznami

```
Require Import List.
```

Na voljo imamo skoraj celoten Ocaml.

```
Check 1::2::3::nil.
1 :: 2 :: 3 :: nil
     : list nat
```

```
Eval compute in map (fun x => x + 3) (1::3::2::nil).
= 4::6::5::nil : list nat
```

```
Fixpoint sum_list l :=
  match l with nil => 0 | n::tl => n + sum_list tl end.
```

Iskanje obstoječih dokazov

```
Search True.
I : True
```

Preiskovanje delovnega okolja.

```
Search le.
between_le: forall (P : nat -> Prop) (k l : nat),
    between P k l -> k <= l
exists_le_S:
  forall (Q : nat -> Prop) (k l : nat),
       exists_between Q k l -> S k <= l
...
```

```
SearchPattern (_ + _ <= _ + _).
plus_le_compat_l: forall n m p: nat, n <= m -> p + n <= p + m
plus_le_compat_r: forall n m p: nat, n <= m -> n + p <= m + p
plus_le_compat: forall n m p q: nat, n <= m -> p <= q -> n + p
                                     <= m + q
```

Konstruiranje novih dokazov

1) Uporabnik vnese stavek, ki ga želi dokazati z uporabo ukaza Theorem ali Lemma.
2) Coq izpiše formulo, ki jo je potrebno dokazati, skupaj z lokalnimi izjavami, ki se lahko uporabljajo v dokazu.
3) Uporabnik vnese ukaz, ki pove na kakšen način lahko razbijemo problem na bolj enostavne podprobleme.
4) Coq izpiše seznam formul, ki jih je še potrebno dokazati.
5) nazaj na korak 3.

Taktika == Uporabnik interaktivno usmerja dokazovanje!

Konstruiranje novih dokazov (2)

```
Lemma example2 : forall a b:Prop, a /\ b -> b /\ a.
1 subgoal
  ============================
    forall a b : Prop, a /\ b -> b /\ a
Proof.
 intros a b H.
1 subgoal
  a : Prop
  b : Prop
  H : a /\ b
  ============================
    b /\ a
split.
2 subgoals
...
intuition.
Proof completed.
Qed.
intros a b H.
split.
 destruct H as [H1 H2]
   exact H2.
 intuition.
example2 is defined
```

Dokaz z indukcijo (1)

```
Lemma sum_n_p : forall n, 2 * sum_n n + n = n * n.
induction n.
  ============================
   2 * sum_n 0 + 0 = 0 * 0
subgoal 2 is:
 2 * sum_n (S n) + S n = S n * S n

reflexivity.

n : nat
IHn : 2 * sum_n n + n = n * n
============================
 2 * sum_n (S n) + S n = S n * S n

assert (SnSn : S n * S n = n * n + 2 * n + 1).
  ============================
   S n * S n = n * n + 2 * n + 1
 ring.
  n : nat
  IHn : 2 * sum_n n + n = n * n
  SnSn : S n * S n = n * n + 2 * n + 1
  ============================
   2 * sum_n (S n) + S n = n * n + 2 * n + 1
```

Dokaz z indukcijo (2)

```
rewrite SnSn.
  n : nat
  IHn : 2 * sum_n n + n = n * n
  SnSn : S n * S n = n * n + 2 * n + 1
============================
 2 * sum_n (S n) + S n = n * n + 2 * n + 1

rewrite <- IHn.
  ============================
   2 * sum_n (S n) + S n = 2 * sum_n + n + 2 * n + 1

simpl.
  ============================
   n + sum_n n + (n + sum_n n + 0) + S n =
   sum_n + (sum_n n + 0) + (n + (n + 0)) + 1
ring.
Qed.
```

Dokazovanje lastnosti programov na seznamih

```
Fixpoint count n l :=
  match l with
    nil => 0
  | a::tl =>
    let r := count n tl in if beq_nat n a then 1+r else r
  end.
```

```
Lemma insert_incr : forall n l, count n (insert n l) =
                                1 + count n l.
intros n l; induction l.
  ============================
   count n (insert n nil) = 1 + count n nil
...
...
```

Definicija induktivnih tipov (1)

```
Inductive bin : Type :=
  L : bin
| N : bin -> bin -> bin.
```

```
Definition example7 (t : bin): bool :=
  match t with N L L => false | _ => true end.
```

```
Fixpoint flatten_aux (t1 t2:bin) : bin :=
  match t1 with
    L => N L t2
  | N t'1 t'2 => flatten_aux t'1 (flatten_aux t'2 t2)
  end.
Fixpoint flatten (t:bin) : bin :=
  match t with
    L => L | N t1 t2 => flatten_aux t1 (flatten t2)
  end.
Fixpoint size (t:bin) : nat :=
  match t with
    L => 1 | N t1 t2 => 1 + size t1 + size t2
  end.
```

Definicija induktivnih tipov (2)

```
Fixpoint flatten_aux (t1 t2:bin) : bin :=
  match t1 with
    L => N L t2
  | N t'1 t'2 => flatten_aux t'1 (flatten_aux t'2 t2)
  end.
Fixpoint flatten (t:bin) : bin :=
  match t with
    L => L | N t1 t2 => flatten_aux t1 (flatten t2)
  end.
Fixpoint size (t:bin) : nat :=
  match t with
    L => 1 | N t1 t2 => 1 + size t1 + size t2
  end.
```

```
Lemma example7_size :
   forall t, example7 t = false -> size t = 3.
intros t; destruct t.
2 subgoals
  ============================
   example7 L = false -> size L = 3
... ...
```

Dokazi z indukcijo

```
Lemma flatten_aux_size :
 forall t1 t2, size (flatten_aux t1 t2) = size t1 + size t2 + 1.
 induction t1.
  ============================
   forall t2 : bin, size (flatten_aux L t2) = size L + size t2 + 1
...
...
```

Induktivno definirane lastnosti

```
Inductive even : nat -> Prop :=
  even0 : even 0
| evenS : forall x:nat, even x -> even (S (S x)).
```

```
Lemma even_mult : forall x, even x -> exists y, x = 2*y.
intros x H; elim H.
2 subgoals
  x : nat
  H : even x
  ============================
   exists y : nat, 0 = 2 * y
...
```