MiniAgda

Terminazione e produttività con *sized-types*.

Edoardo Marangoni

Agenda

- **Introduzione**
- **Tipi induttivi**
  Controllare la terminazione. Regole (intuitive) per il controllo della terminazione e implementazione in MiniAgda.
- **Tipi co-induttivi**
  Controllare la produttività. Guardedness non tipata (sintattica) e tipata.

Introduzione

Nei proof assistant, la totalità è necessaria per mantenere la **consistenza** del sistema: se, infatti, ammettiamo definizioni ricorsive non terminanti, si possono dimostrari teoremi falsi.

```
Unset Guard Checking.
Fixpoint bad (x : nat) := 1 + (bad x).

Axiom bad_ax : forall x, bad x = 1 + (bad x).

Theorem n_neq_sn : forall (x:nat), (x = S x) -> False.
Proof. intros.  induction x.
  + inversion H.
  + inversion H.  apply IHx in H1. assumption.
Qed.

Theorem zero_eq_one: 0 = 1.
  assert (bad 0 = 1 + (bad 0)).
  + apply bad_ax.
  + destruct bad; simpl in H.
    * assumption.
    * apply n_neq_sn in H. exfalso. assumption.
Qed.
```

Introduzione

Oltre ai motivi di consistenza logica, in un linguaggio con tipi dipendenti la **totalità** è necessaria anche per assicurare la terminazione della fase di type-checking.

```
fooIsTrue : foo ≡ true
fooIsTrue = refl
```

Per mostrare che `refl` sia una dimostrazione del fatto che `foo ≡ true`, Agda impiega il type checker per mostrare che `foo` sia effettivamente "riscrivibile" in `true`; se `foo` non termina, allora nemmeno il type-checker termina. Ci concentriamo su **terminazione** e **produttività**, ossia due criteri per decidere se permettere delle definizioni di funzione ricorsive e co-ricorsive in linguaggi totali.

Tipi induttivi: terminazione

Esempio basilare in Coq

```
Inductive Nat : Type :=
  | zero : Nat
  | succ : Nat -> Nat

(* ricorsione primitiva *)
Fixpoint minus (x y : Nat) : Nat :=
  match x with
  | zero => zero
  | succ x' => match y with
               | zero => succ x'
               | succ y' => minus x' y'
               end
  end.

(* termina? *)
Fixpoint div (x y : Nat) :  Nat := z
match x with
| zero => zero
| succ x' => succ (div (minus x y) y)
end.
```

Tipi induttivi: terminazione

…e in Agda

```
data Nat : Set where
  zero : Nat
  succ : Nat → Nat

minus : Nat → Nat → Nat
minus zero _ = zero
minus (succ x) zero = succ x
minus (succ x) (succ y) = minus x y

div : Nat → Nat → Nat
div zero _ = zero
div (succ x) y = succ (div (minus x y) y)
```

Le definizioni di `div` ovviamente terminano: vengono accettate?

no

```
Cannot guess decreasing argument of fix.
```

```
Termination checking failed for the following functions:
  div
Problematic calls:
  div (minus x y) y
```

Coq e Agda basano il controllo della terminazione sulla sintassi e non sono pertanto in grado di catturare informazioni semantiche rilevanti.

*"The **untyped** approaches have some shortcomings, including the sensitivity of the termination checker to syntactical reformulations of the programs, and a lack of means to propagate size information through function calls."* [A. Abel, MiniAgda]

Tipi induttivi: terminazione (in agda)

Non tutte le funzioni ricorsive sono permesse: Agda, per esempio, accetta solo quegli schemi ricorsivi che riesce a dimostrare terminanti.

- **Ricorsione primitiva**
  Un argomento di una chiamata ricorsiva deve essere *esattamente* più piccola "di un costruttore".

```
plus : Nat -> Nat -> Nat
plus zero    m = m
plus (suc n) m = suc (plus n m)
```

- **Ricorsione strutturale**
  Un argomento di una chiamata ricorsiva deve essere una *sottoespressione*.

```
fib : Nat -> Nat
  fib zero          = zero
  fib (suc zero)    = suc zero
  fib (suc (suc n)) = plus (fib n) (fib (suc n))
```

[ref]

Sized-types: intuizione

Approccio **tipato**: annotiamo i tipi con una *size* (o *taglia*) che esprime un'informazione sulla dimensione dei valori di quel tipo. MiniAgda è "la prima implementazione *matura* di un sistema con sized-types". [ref]

Per quanto riguarda la terminazione, l'idea di base è la seguente:

- Associamo una *taglia* i ad ogni tipo induttivo D;
- Controlliamo che la taglia diminuisca nelle chiamate ricorsive.

Tipi induttivi sized: implementazione

- **Altezza**
  L'**altezza** di un elemento induttivo $d \in D$ è il *numero di costruttori* di d. Possiamo immaginare d come un albero in cui i nodi sono i costruttori: per esempio, l'altezza di un numero naturale n è $n + 1$ (perché c'è il costruttore $zero$ come foglia con n costruttori $succ$).

- **Taglia**
  Dato un certo tipo induttivo D associamo ad esso una taglia i, ottenendo un tipo D^i che contiene solo quei d la cui altezza **è minore** di i; in altre parole, ogni $d \in D^i$ ha un'altezza che ha come **upper bound** la taglia i.

- **Rappresentazione**
  In un linguaggio d.t. possiamo modellare la taglia come un tipo $Size$ con un successore $\$: Size \to Size$; i *sized-types* sono quindi membri di $Size \to Set$.

Tipi induttivi sized: implementazione

- **Sottotipi**
  Siccome le taglie sono upper bound dell'altezza di un elemento, viene naturale la regola di sottotipizzazione $D^i \leq D^{\$i} \leq \cdots \leq D^\omega$ dove D^ω è un elemento la cui altezza è ignota.

- **Parametricità**
  Le taglie sono utili sono durante il type checking e devono essere rimosse una volta concluso. Pertanto, i risultati di una funzione non devono essere dipendenti dalle taglie.

- **Dot patterns**
  E' necessario utilizzare dei *pattern inaccessibili* per evitare la non-linearità del lato sinistro del pattern match.

Tipi induttivi sized: esempio

- **Naturali *sized***

```
data SNat : Size → Set where
  zero : {i : Size} → SNat (↑ i)
  succ : {i : Size} → SNat (i) → SNat (↑ i)
```

- **Sottrazione *sized***

```
minus : {j : Size} → SNat j → SNat ∞ → SNat j
minus  zero y = zero
minus  x zero   = x
minus  (succ x) (succ y) = minus x y
```

- **Divisione euclidea *sized***

```
div : {i : Size} → SNat i → SNat ∞ → SNat i
div zero y = zero
div (succ x) y = succ (div (minus x y) y)
```

- Produttività
  "..Instead of termination we require productivity, which means that the next portion can always be produced in finite time. A simple criterion for productivity which can be checked syntactically is guardedness.."
  ref

Produttività

In Agda, il controllo della produttività di una funzione co-ricorsiva è basato sulla *guardedness* della definizione, ossia richiediamo che la definizione di una funzione (la parte destra) sia un (co-)costruttore e che ogni chiamata ricorsiva sia direttamente "sotto" esso.

Queste definizioni sono accettate:

```
CoFixpoint repeat (A:Type) (a:A): Stream := cons _ a (repeat _ a).
Compute hd (repeat _ 0). (* = 0 *)
Compute hd (tail (repeat _ 0)). (* = 0 *)

CoFixpoint countFrom (f :nat): Stream := cons _ f (countFrom (f + 1)).
Compute hd (countFrom  0). (* = 0*)
Compute hd (tail (countFrom 0)). (* = 1*)
Compute hd (tail (tail (countFrom 0))). (* = 2*)
```

Tipi induttivi: terminazione

Introduciamo i *size patterns* $i > j$ per legare una variabile j e "ricordarsi" che $i > j$.

```
minus : (i : Size) -> SNat i -> SNat ω -> SNat i
minus i (zero (i > j)) y = zero j
minus i x (zero .ω) = x
minus i (succ (i > j) x) (succ .ω y) = minus j x y
```

Siccome nella chiamata ricorsiva la taglia decresce in tutti e tre gli argomenti la terminazione può essere dimostrata.

Tipi co-induttivi

Questa definizione è accettata per *guardedness*:

```
repeat : (A : Set) → (a:A) → Stream A
repeat A a = cons A a (repeat A a)
```

Questa dipende da f:

```
repeatf : (A : Set) → (a:A) → Stream A
repeatf A a = cons A a (f repeat A a)
```

se f è, esempio, $tail$, la definizione si riduce a sé stessa dopo una ricorsione:

```
tail (repeat a) → tail (a :: tail repeat a) → tail repeat a → ...
```

se invece f mantiene la lunghezza dello stream o la incrementa, `repeatf` è produttiva; i controlli puramente sintattici, però, non possono catturare questo aspetto.

Tipi co-induttivi: implementazione

- ## Profondità

La **profondità** di un elemento coinduttivo $d \in D$ è il *numero di co-costruttori* di d. Uno stream interamente costruito avrà profondità ω.

- ## Taglia

Indiciamo quindi il tipo D con i ottenendo un tipo D^i che contiene solo quei d la cui altezza **è maggiore** di i; in altre parole, la taglia i di un tipo coinduttivo D^i è un **lower bound** dell'altezza degli elementi di D^i.

```
codata NatStream : Size -> Set
  cons : (i : Size) -> Nat -> NatStream i -> NatStream ($ i)

hd : (i : Size) -> NatStream $i -> Nat
hd i (cons .i a s) = a

tl : (i : Size) -> NatStream $i -> NatStream i
tl i (cons .i a s) = s
```

Tipi co-induttivi: produttività

La funzione `repeat` con i sized-types:

```
repeat: Nat -> (i:Size) -> NatStream i
repeat n ($i) = cons i n (repeat n i)
```

Assumiamo che `repeat n i` produca uno stream *ben definito* di profondità i e automaticamente mostriamo che `repeat n ($i)` produce uno stream di profondità $n + 1$.

- **Successor pattern**

Perché possiamo fare il matching su `($i)`?
Se `j = ($i) = n + 1` allora $i = n$; se `j = ω`, allora `i = ω`; se `j = 0`, allora possiamo produrre ciò che vogliamo.