# MiniAgda

## Terminazione e produttività con *sized-types*.

Edoardo Marangoni

# Agenda

- **Introduzione**

  Controlli di terminazione e produttività, approcci alternativi
- **Sized-types**

  *Intuizione* dei sized-types in MiniAgda
- **Tipi induttivi**

  Regole (intuitive) per il controllo della terminazione e implementazione in MiniAgda
- **Tipi co-induttivi**

  *Guardedness* non tipata (sintattica) e tipata

# 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.

Alcuni controlli implementati in Agda:

- Type checking
- Coverage checking
- Termination checking
- Productivity (via guardedness)

(dalla documentazione di Agda)

# Introduzione

Ci concentriamo su *terminazione* e *produttività*, ossia due criteri per decidere se permettere delle definizioni di funzione ricorsive e co-ricorsive in linguaggi totali.

- **Terminazione**

*"Not all recursive functions are permitted - Agda accepts only these recursive schemas that it can mechanically prove terminating."* [ ref ]

- **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 ]

# Terminazione

In Agda, il controllo di terminazione permette la definizione di funzioni ricorsive primitive e di funzioni strutturalmente ricorsive.

*"…a given argument must be exactly one constructor smaller in each recursive call…"*

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

*"…we require recursive calls to be on a (strict) subexpression of the argument; this is more general that just taking away one constructor at a time. It also means that arguments may decrease in an lexicographic order - this can be thought of as nested primitive recursion…"*

```
-- Structural recursion
ack : Nat -> Nat -> Nat
ack zero    m       = suc m
ack (suc n) zero    = ack n (suc zero)
ack (suc n) (suc m) = ack n (ack (suc n) m)
```

"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

# Sized-types

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".

A. Abel, MiniAgda

# Tipi induttivi

## Intuizione

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

# Tipi induttivi: implementazione

- **Esempio**

```
data SNat : Size -> Set
  zero : (i : Size) -> SNat ($ i);
  succ : (i : Size) -> SNat i -> SNat ($ i)
```

- **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.

```
pred : (i : Size) -> SNat ($$ i) -> SNat ($ i)
pred i (succ .($ i) n) = n
pred i (zero .($ i)) = zero i
```

# 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 $\omega$.

- **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.