Sized Types in Agda

Andreas Abel

Department of Computer Science
Ludwig-Maximilians-University Munich

Agda Intensive Meeting
Sendai, Japan
28 November 2008

Funded by AIST and JST

Introduction

- Sized types are available now in Agda.
- They support termination checking.
- Attach sizes to inductive families.
- Currently: track non size increasing functions.
- Communicate size information through abstractions.

Simple Example: Euclidean Division

```
data Nat : {i : Size} -> Set where
  zero : {i : Size} -> Nat {↑ i}
  suc  : {i : Size} -> Nat {i} -> Nat {↑ i}

sub : {i : Size} -> Nat {i} -> Nat {∞} -> Nat {i}
sub zero n = zero
sub (suc m) zero = suc m
sub (suc m) (suc n) = sub m n

-- div' m n  computes  ceiling(m/(n+1))
div' : {i : Size} -> Nat {i} -> Nat -> Nat {i}
div' zero    n = zero
div' (suc m) n = suc (div' (sub m n) n)
```

Formalities

- To use sized types:

```
{-# OPTIONS --sized-types #-}

module SizedNat where

open import Size
```

- Switch on sized types (option `--sized-types`)
- Import standard library module `Size`

The module `Size`

```
module Size where

postulate
  Size : Set
  ↑_   : Size -> Size
  ∞    : Size

{-# BUILTIN SIZE    Size #-}
{-# BUILTIN SIZESUC ↑_   #-}
{-# BUILTIN SIZEINF ∞    #-}
```

Annotating Your Data Type

- `Size` must a first family index (not a parameter).
- Constructors quantify over size argument i.
- The target type of a constructor is annotated with size $\uparrow i$.
- The recursive occurences of the data type with size i.

```
data Nat : {i : Size} -> Set where
  zero : {i : Size} -> Nat {↑ i}
  suc  : {i : Size} -> Nat {i} -> Nat {↑ i}

data Rose (A : Set) : {_ : Size} -> Set where
  rose : {i : Size} -> A ->
         List (Rose A {i}) -> Rose A {↑ i}
```

- The usual positivity restriction applies.

Annotating Your Data Type II

- Agda resolves unconstrained sizes to ∞.
- Hide size arguments and use it as a not sized type.

```
data Ord : _ : Size -> Set where
  zero : {i : Size} -> Ord {↑ i}
  suc  : {i : Size} -> Ord {i} -> Ord {↑ i}
  lim  : {i : Size} ->
         (Nat -> Ord {i}) -> Ord {↑ i}
```

- Higher-order constructor arguments are supported.
- Semantically, sizes are ordinals.

Termination Checking with Sized Types

```
sub : {i : Size} -> Nat {i} -> Nat {∞} -> Nat {i}
sub .{↑ i} (zero {i})  n         = zero {i}
sub .{↑ i} (suc {i} m) zero      = suc {i} m
sub .{↑ i} (suc {i} m) (suc n)   = sub {i} m n

div' : {i : Size} -> Nat {i} -> Nat -> Nat {i}
div' .{↑ i} (zero {i})  n = zero {i}
div' .{↑ i} (suc {i} m) n =
  suc  {i} (div' {i} (sub {i} m n) n)
```

Note: you cannot match on sizes!

What is special about `Size`?

Option `--sized-types` enables the following features:

- $\uparrow ?$ unifies with ∞, yielding $? = \infty$.
- $i \leq \uparrow i \leq \infty$.
- A constraint solver for a set of inequalities $x + n \leq y, x \leq y + m$ (using Warshall's algorithm).
- Subtyping for sized data types:

$$\texttt{Nat}\ \{\texttt{i}\} \leq \texttt{Nat}\ \{\uparrow \texttt{i}\} \leq \texttt{Nat}\ \{\infty\}$$

- The strict inequality $i <\uparrow i$ is exploited for termination checking.

Example: Map for Rose Trees

```
data Rose (A : Set) : {_ : Size} -> Set where
  rose : {i : Size} -> A ->
         List (Rose A {i}) -> Rose A {^ i}

mapRose : {A B : Set} -> (A -> B) ->
          {i : Size} -> Rose A {i} -> Rose B {i}
mapRose f (rose a l) = rose (f a) (map (mapRose f) l)
```

Extension to Coinductive Types

- Streams

```
codata Stream (A : Set) : {_ : Size} -> Set where
  _::_ : forall {i} -> A ->
         Stream i A -> Stream (^ i) A

mapStream :
  forall {A B i} ->
  (A -> B) -> Stream A {i} -> Stream B {i}

nats : forall {i} -> Stream Nat {i}
nats = 0 :: mapStream suc nats
```

Further Issues

- Add $+$ for sizes (Barthe/Gregoire/Riba, CSL 2008).
  - Then quicksort/mergesort can be typed size-preserving.
  - Need to solve inequalities between linear combinations.
- Semantics of Agda with sizes??
  - Sizes are like types: cannot be matched on.
  - Both can be considered *erased/irrelevant*.
  - See Pfenning (LICS 2001): irrelevance as modality.
  - Miquel (Implicit CC): universal quantification as intersection.
  - Barras et al.: Implementation of Implicit Coq.

Uniform quantification

$$\frac{\Gamma, x : A \vdash B}{\Gamma \vdash \forall x : A.\, B}$$

$$\frac{\Gamma, [x : A] \vdash t : B}{\Gamma \vdash \lambda x t : \forall x : A.\, B} \qquad \frac{\Gamma \vdash r : \forall x : A.\, B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\, s : B[s/x]}$$

$$\text{no rule } \frac{}{\Gamma, [x : A], \Gamma' \vdash x : A}$$

$$\frac{\Gamma^{\oplus} \vdash B : \mathsf{Set}}{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash r = r : \forall x : A.\, B \qquad \Gamma \vdash s : A \qquad \Gamma \vdash s' : A}{\Gamma \vdash r\, s = r'\, s' : B[s/x]}$$

Code sprint suggestions

- Sized types in Agda: codata, +.
- Universe polymorphism.
- Irrelevant quantification and erasure.