MiniAgda: Checking Termination using Sized Types

Andreas Abel

Department of Computer Science
University of Munich

Workshop on Partiality and Recursion
in Interactive Theorem Proving
FLoC 2010, Edinburgh, UK
15 July 2010

Type-based termination

- View data (natural numbers, lists, binary trees) as trees.
- Type of data is equipped with a size.
- Size = upper bound on height of tree.
- Size must decrease in each recursive call.
- Termination is ensured by type-checker.

Sized types in a nutshell

- Sizes are upper bounds.
- List^a denotes lists of length $< a$.
- List^∞ denotes list of arbitrary (but finite) length.
- Sizes induce subtyping: $\mathsf{List}^a \leq \mathsf{List}^b$ if $a \leq b$.
- Size expressions a, b.

$$
\begin{array}{rcll}
a & ::= & i & \text{variable} \\
  & \mid & a+1 & \text{successor} \\
  & \mid & \infty & \omega
\end{array}
$$

Sized Natural Numbers

Declaring a sized type.

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

Purely applicative terms.

```
let inc2 : (i : Size) -> SNat i -> SNat ($$ i)
         = \ i -> \ n -> succ ($ i) (succ i n)
```

Pattern matching.

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

Size Arguments are Parametric

- Computational behavior independent of size arguments.
- Parametric polymorphism (ML/System F) extends to sizes.

```
sized data SNat : Size -> Set
{ zero : [i : Size] -> SNat ($ i)
; succ : [i : Size] -> SNat i -> SNat ($ i)
}

let inc2 : [i : Size] -> SNat i -> SNat ($$ i)
         = \ i -> \ n -> succ ($ i) (succ i n)

fun pred : [i : Size] -> SNat ($$ i) -> SNat ($ i)
{ pred i (succ .($ i) n) = n
; pred i (zero .($ i))   = zero i
}
```

Tracking Termination and Sizes

- Subtraction does not increase the size.

```
fun minus : [i : Size] -> SNat i -> SNat # -> SNat i
{ minus i (zero (i > j))     y            = zero j
; minus i  x               (zero .#)     = x
; minus i (succ (i > j) x) (succ .# y) = minus j x y
}
```

- div $i\ x\ y$ computes $\lceil x/(y+1) \rceil$

```
fun div : [i : Size] -> SNat i -> SNat # -> SNat i
{ div i (zero (i > j))   y = zero j
; div i (succ (i > j) x) y = succ j (div j (minus j x y) y
}
```

Polymorphic Data Types

- List is a monotone type constructor.

```
data List (+ A : Set) : Set
{ nil  : List A
; cons : A -> List A -> List A
}
```

- Full types of list constructors:

$$\begin{array}{lcl} \mathsf{nil} & : & [A : \mathsf{Set}] \to \mathsf{List}\ A \\ \mathsf{cons} & : & [A : \mathsf{Set}] \to A \to \mathsf{List}\ A \to \mathsf{List}\ A \end{array}$$

- map is parametric in its type arguments.

```
fun mapList : [A:Set] -> [B:Set] -> (A->B) -> List A -> List B
{ mapList A B f (nil .A)       = nil B
; mapList A B f (cons .A a as) = cons B (f a) (mapList A B f as
}
```

Nesting (Interleaving) Inductive Types

- Rose bushes: finitely branching trees.

```
sized data Rose (+ A : Set) : Size -> Set
{ rose : [i:Size] -> A -> List (Rose A i) -> Rose A ($ i)
}
```

- map for Rose can be defined modularly from map for List.

```
fun mapRose : [A : Set] -> [B : Set] -> (A -> B) ->
              [i : Size] -> Rose A i -> Rose B i
{ mapRose A B f i (rose .A (i > j) a rs) =
    rose B j (f a) (mapList (Rose A j) (Rose B j)
                     (mapRose A B f j)
                     rs)
}
```

Coinductive Types

- Streams, unsized.

```
codata Stream (+ A : Set) : Set
{ cons : A -> Stream A -> Stream A
}
```

- Guarded corecursion.

```
cofun repeat : [A : Set] -> (a : A) -> Stream A
{ repeat A a = cons A a (repeat A a)
}
```

Sized Coinductive Types

- Size index counts the number of guards.

```
sized codata Stream (+ A : Set) : Size -> Set
{ cons : [i : Size] ->
         (head : A) ->
         (tail : Stream A i) -> Stream A ($ i)
}
```

- Just one (co)constructor: destructors are generated!

```
fun head : [A:Set] -> [i:Size] -> Stream A ($ i) -> A
{ head A i (cons .A .i a as) = a
}
fun tail : [A:Set] -> [i:Size] -> Stream A ($ i) -> Stream A i
{ tail A i (cons .A .i a as) = as
}
```

Semantics of Sized Streams

- Coinductive types are greatest fixpoints, approximated from above.

$$
\begin{array}{lcl}
\text{Stream}\ A\ 0 & = & \top \\
\text{Stream}\ A\ (i+1) & = & \{\text{cons}\ A\ i\ a\ s \mid a \in A \text{ and } s \in \text{Stream}\ A\ i\} \\
\text{Stream}\ A\ \omega & = & \bigcap_{i<\omega} \text{Stream}\ A\ i
\end{array}
$$

- Any term inhabits Stream A 0!
- When constructing a term in Stream A i we may assume $i \neq 0$.

```
cofun repeat : [A:Set] -> (a:A) -> [i:Size] -> Stream A i
{ repeat A a ($ i) = cons A i a (repeat A a i)
}
```

Preserving of Guardedness

- Many Stream functions preserve guardedness:

```
cofun map : [A : Set] -> [B : Set] -> [i : Size] ->
            (A -> B) -> Stream A i -> Stream B i
{ map A B ($ i) f (cons .A .i x xs) = cons B i (f x)
    (map A B i f xs)
}

cofun merge : [i : Size] ->
        Stream Nat i -> Stream Nat i -> Stream Nat i
{ merge ($ i) (cons .Nat .i x xs) (cons .Nat .i y ys) =
      leq x y (Stream Nat ($ i))
         (cons Nat i x (merge i xs (cons Nat i y ys)))
         (cons Nat i y (merge i (cons Nat i x xs) ys))
}
```

Challenge: The Hamming Function

Output the numbers generated by prime factors $2, 3$ in order.

```
let double : Nat -> Nat = ...
let triple : Nat -> Nat = ...

cofun ham : [i : Size] -> Stream Nat i
{ ham ($ i) = cons Nat i (succ zero)
                (merge i (map Nat Nat i double (ham i))
                         (map Nat Nat i triple (ham i)))
}
```

Challenge: The Fibonacci Stream

Produce the Fibonacci sequence (in one line of Haskell code).

```
cofun adds : [i : Size] ->
             Stream Nat i -> Stream Nat i -> Stream Nat i
{ adds ($ i) (cons .Nat .i a as) (cons .Nat .i b bs) =
    cons Nat i (add a b) (adds i as bs)
}

cofun fib : [i : Size] -> Stream Nat i
{  fib ($ i) = cons Nat i 0 (adds (cons Nat i 1 (fib i))
                                  (fib i))
}
```

$\mathsf{fib} = (0, (1, \mathsf{fib}) + \mathsf{fib})$.

Internal handling of `Size`

- $i \leq \$i \leq \infty$, extends to subtyping

  `SNat i` $\leq$ `SNat ($ i)` $\leq$ `SNat` ∞

- Strict inequality $i < \$i$ is exploited for termination checking.
- Partial reconstruction of omitted sizes in terms.
- $\$X$ unifies with ∞, yielding $X = \infty$.
- Constraint solver (using Warshall's algorithm) for inequalities
  - $x + n \leq y$ represented as $x \xrightarrow{-n} y$,
  - $x \leq y + m$ represented as $x \xrightarrow{+m} y$.

Formalization

- Sized inductive type $\mu^i X.\, A$.
- Equations and subtyping.

$$
\begin{array}{lcll}
\mu^{a+1} X.\, A & = & A[(\mu^a X.\, A)/X] & \\
\mu^{\infty} X.\, A & = & A[(\mu^{\infty} X.\, A)/X] & \\
\mu^a X.\, A & \leq & \mu^b X.\, A & \text{for } a \leq b
\end{array}
$$

- Example: lists.

$$
\begin{array}{lcl}
\mathsf{List}^i A & := & \mu^i X.\, 1 + A \times X \\
\mathsf{nil} & : & \forall A \forall i.\, \mathsf{List}^{i+1} A \\
 & := & \mathsf{inl}() \\
\mathsf{cons} & : & \forall A.\, A \to \forall i.\, \mathsf{List}^i A \to \mathsf{List}^{i+1} A \\
 & := & \lambda a \lambda as.\, \mathsf{inr}(a, as)
\end{array}
$$

Recursion

- Recursion principle (semantically):

$$\frac{\mathsf{fix}\, f \in A^0 \qquad f \in A^\alpha \to A^{\alpha+1} \qquad (\mathsf{fix}\, f \in \bigcap_{\alpha<\omega} A^\alpha) \to \mathsf{fix}\, f \in A^\omega}{\forall \beta \leq \omega.\, \mathsf{fix}\, f \in A^\beta}$$

- Step: $\mathsf{fix}\, f \in A^\alpha$ implies $f\,(\mathsf{fix}\, f) = \mathsf{fix}\, f \in A^{\alpha+1}$.
- Restrict admissible types A^α such that
  - $\mathsf{fix}\, f \in A^0$ is trivial, e.g., $A^\alpha = (\mu^\alpha X.A) \to C$, ($\mu^0 X.A$ is empty)
  - $(\bigcap_{\alpha<\omega} A^\alpha) \subseteq A^\omega$.
- Typing rule for recursion (e.g., $A^i = \mathsf{List}^i \mathsf{Int} \to \mathsf{List}^i \mathsf{Int}$):

$$\frac{f : \forall i.\, A^i \to A^{i+1}}{\mathsf{fix}\, f : A^a}\, A^i \text{ admissible}$$

Conclusions

- Toy implementation of:
  1. Dependent types and pattern matching.
  2. Sized types and termination checking.
  3. Parametric functions.
  4. η-equality for non-informative types.
- TODO:
  - Mutual data types.
  - Hidden arguments and type reconstruction.
  - Automatic size annotation for data types and functions.

References

- Parametric functions in type theory:
  - Alexandre Miquel (PhD, TLCA 2001)
  - Bruno Barras and Bruno Bernardo (FoSSaCS 2008)
- Sized types:
  - Hughes, Pareto, and Sabry (POPL 1996)
  - Barthe et. al. (MSCS 2004, LPAR 2006)
  - Blanqui (RTA 2004, CSL 2005)
  - Abel (LMCS 2008, **PAR-10**)