MiniAgda: Integrating Sized and Dependent Types

Andreas Abel, sec 1-2

Loes Habermehl
Luuk Verkleij

Radboud University

Untyped Termination Checking

In dependent type theories underlying Coq and Agda, all programs need to be **total:**

- All functions defined by recursion over induction terminate
- All functions defined by corecursion into a coinductive type are productive

The *guard condition* in Coq is an untyped termination checker, which has some shortcomings:

- Sensitive to syntactical reformulations
- Cannot propagate size information through function calls

Typed Termination Checking

As an alternative, use **sized types**: data types with a size index, where the size index is the size of the elements or an upper bound.

Type-based termination checking:

1. Attach a size index *i* to each inductive data type *D.*

2. Check that sizes decrease in recursive calls.

Typed Termination Checking

As an alternative, use **sized types**: data types with a size index, where the size index is the size of the elements or an upper bound.

Type-based termination checking:

1. Attach a size index *i* to each inductive data type *D.*

2. Check that sizes decrease in recursive calls.

Attaching Sizes

Inductive data type *D* with size *i* → sized type D^i.

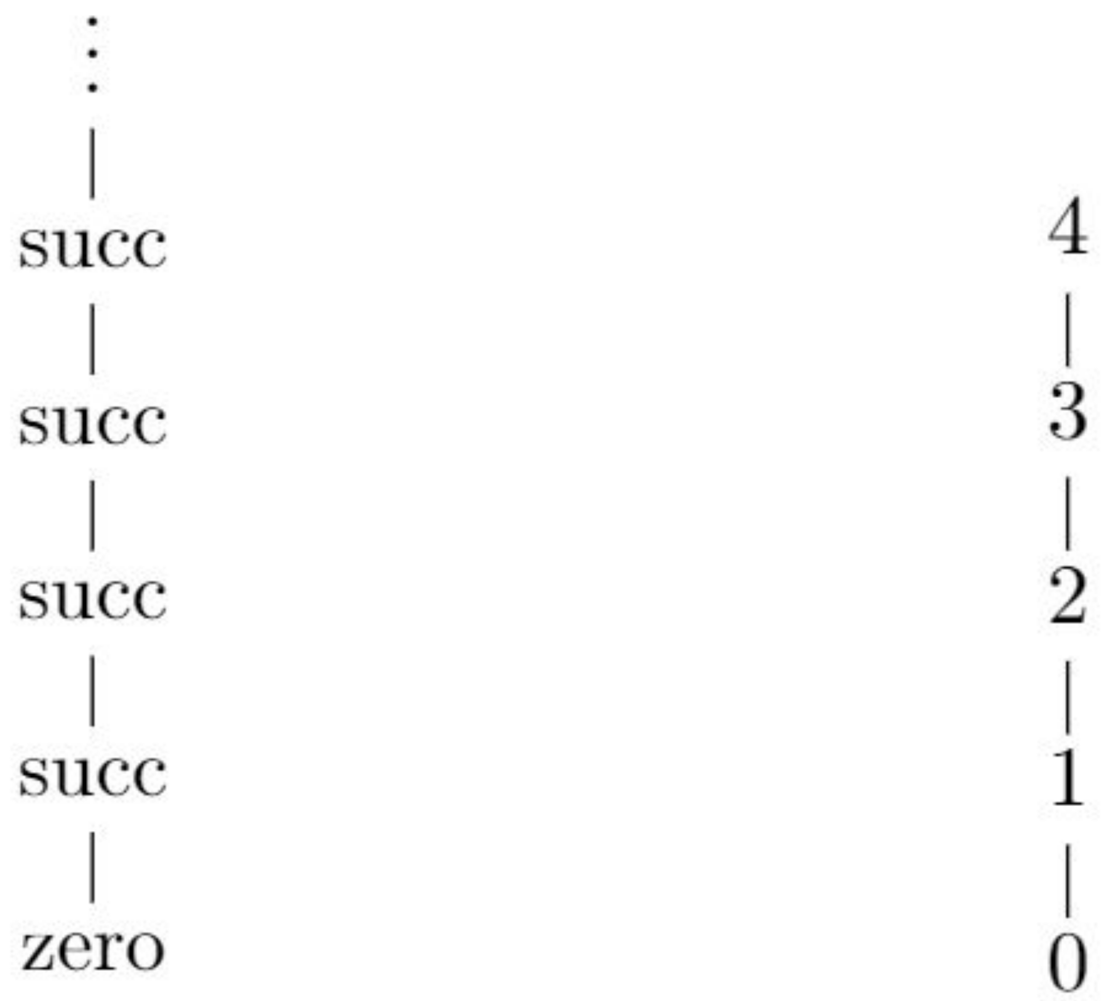
D^i contains only elements whose height is below *i*.

How do we calculate the height?

→ Represent elements as trees, where each constructor is a node.

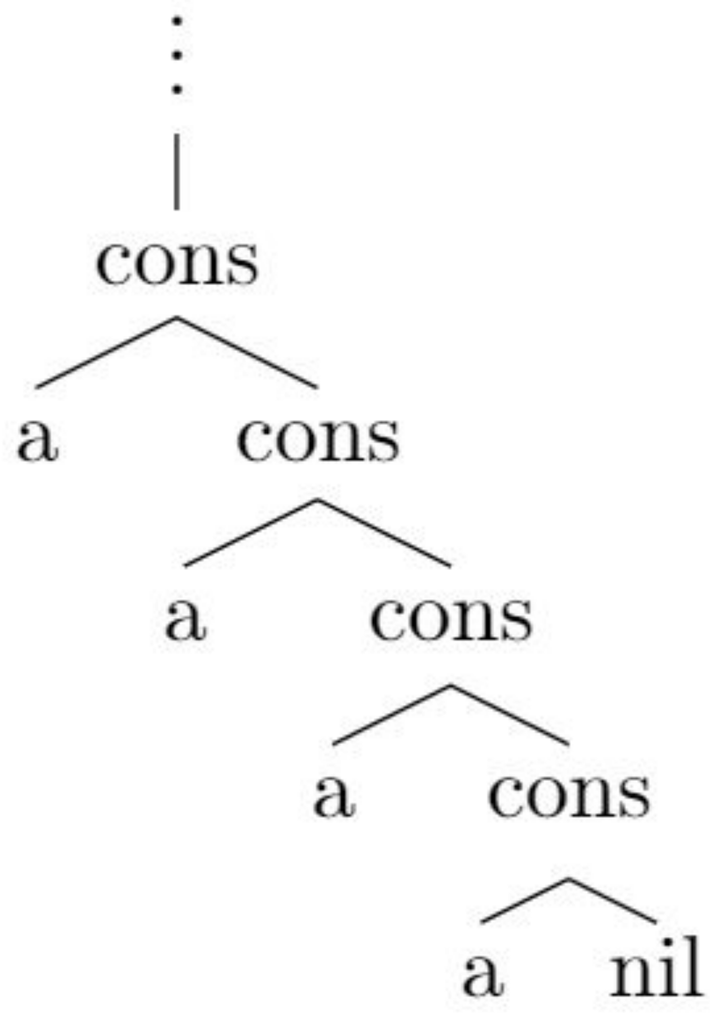
Example: Nat

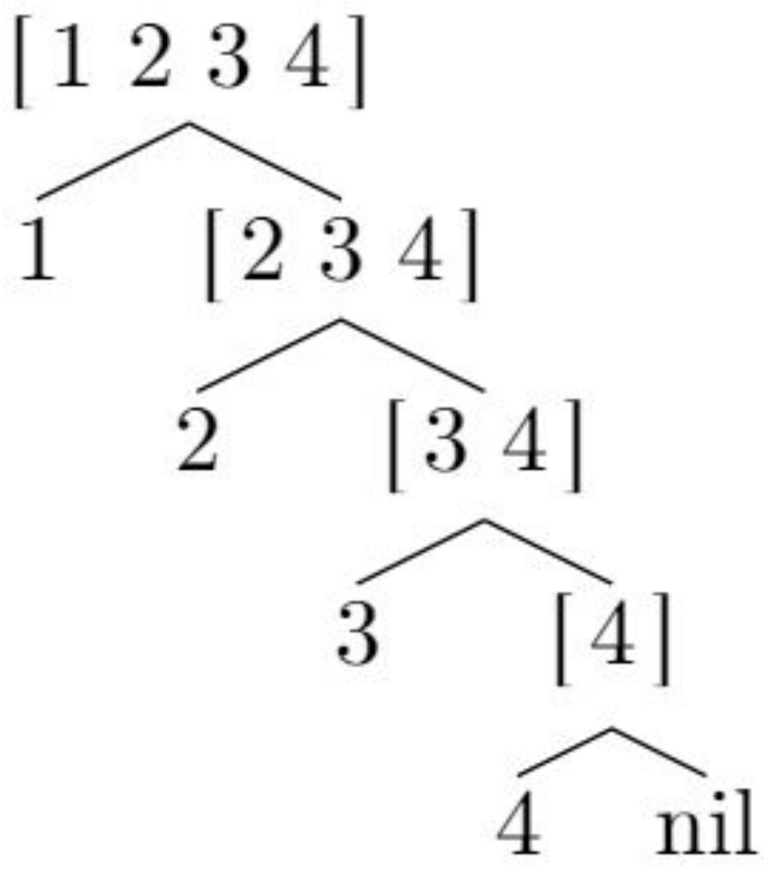
The height of an element Nat is the value plus one.



Example: List

The height of a List is its length plus one.





Implementation

Using successor operation $\uparrow$: Size $\rightarrow$ Size

```
data SNat : (i : Size) -> Set where
  zero : (i : Size) -> SNat (↑ i)
  succ : (i : Size) -> SNat (i) -> SNat (↑ i)
```

Non-recursive example:

```
inc2 : (i : Size) -> SNat (i) -> SNat (↑↑ i)
inc2 i n = succ (↑ i) (succ i n)
```

Dot Patterns

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

The **dot pattern** or *inaccessible pattern* means that there is only one possible term, in this case $\uparrow i$.

Parametric Function Types

Sizes are parametric:

- Only serve to ensure termination
- Functions can never depend on sizes
- Should be erased during compilation

However, the *type* of a function does depend on size.

For example, $pred\ i\ n = pred\ j\ n$, but the types $SNat\ (\uparrow\ i) \neq SNat\ i$.

Refined Definitions

```
data SNat : (i : Size) -> Set where
  zero : (i : Size) -> SNat (↑ i)
  succ : (i : Size) -> SNat (i) -> SNat (↑ i)
```

Refined Definitions

```
data SNat : {i : Size} -> Set where
  zero : {i : Size} -> SNat {↑ i}
  succ : {i : Size} -> SNat {i} -> SNat {↑ i}
```

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

Refined Definitions

```
data SNat : {i : Size} -> Set where
   zero : {i : Size} -> SNat {↑ i}
   succ : {i : Size} -> SNat {i} -> SNat {↑ i}
```

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

or implicit:

```
pred’ : {i : Size} -> SNat {↑↑ i} -> SNat {↑ i}
pred’ (succ n) = n
pred’ zero     = zero
```

Recall

Type-based termination checking:

1. Attach a size index *i* to each inductive data type *D.*
2. Check that sizes decrease in recursive calls.

Recall

Type-based termination checking:

1. Attach a size index *i* to each inductive data type *D.*

2. Check that sizes decrease in recursive calls.

Ensure Termination

```
minus : {i : Size} -> SNat {i} -> SNat {∞} -> SNat {i}
minus .{↑i} (zero {i})    y              = zero {i}
minus {i}    x            (zero .{∞})   = x
minus .{↑i} (succ {i} x) (succ .{∞} y) = minus {i} x y
```

∞ represents infinity, which means that the upper bound is unknown.

`{↑ i}` represents a size constraint.

The recursive call in the last line shows that all three arguments decrease, thus termination is ensured.

Ensure Termination

```
div : {i : Size} -> SNat {i} -> SNat {∞} -> SNat {i}
div .{↑ i} (zero {i})   y = zero {i}
div .{↑ i} (succ {i} x) y = succ {i} (div {i} (minus {i} x y) y)
```

Both `x` and `minus {i} x y` are bounded by size `i`, thus the recursive call is of size `i` while the arguments are of size `↑i`, so termination is ensured.

Interleaving Inductive Types

An advantages of using sized types include that they scale very well to higher order constructions.

Using sized types we have increased modularity compared to the untyped termination checkers.

We will show this by looking at the rosetree construction.

Rosetree

A rosetree is a tree with the following properties

- Nodes have values
- Nodes have a variable number of branches
- Leafs are nodes without branches

We can define a rosetree in Agda as follows:

```
data Rose (A : Set) : Set where
  rose : A -> List (Rose A) -> Rose A
```

Rosetree example

```
rose 1 []
```

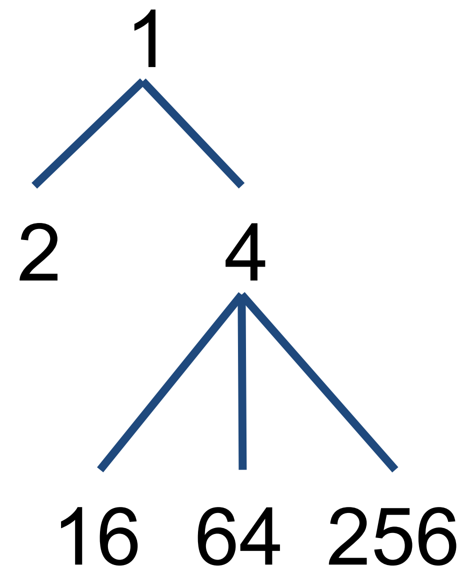
1

Rosetree example

```
rose 1 []
```

1

```
rose 1 [
   rose 2 [],
   rose 4 [
      rose 16 [], rose 64 [], rose 256 []
   ]
]
```



Recursive Functions on Rosetree

We already have list functions, for example map or filter, so if we implement function on the rosetree we want to reuse as much as possible.

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

However, both Coq and Agda fail to conclude this terminates, as the recursive call doesn't have the same type, as it is underapplied.

Recursive Functions on Rosetree

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

However termination is trivial:

All rosetrees in the recursive call have a height that is strictly smaller than the height of the previous rosetree.

This is exactly sized type rosetrees where the size is the number of constructors!

Rosetree using Sized Type

Now we expand the rosetree definition using sized type.

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

The size of the rosetree should be equal to the height of the tree. That is the case if we count the number of constructors, like we did before with SNat.

Mapping on a Sized Rosetree

Now we edit the mapping function such that it accepts SRose.

```
mapSRose : {i : Size } -> {A B : Set} ->
   (A -> B) -> SRose A {i} -> SRose B {i}
mapSRose .{↑i} f (srose {i} a l)
   = srose {i} (f a) (map (mapRose {i} f) l)
```
[1]

1. Doesn't work, replace this with "`srose {_} {i} (f a) (map (mapRose {i} f) l)`" or do not use explicit sizes