MiniAgda: Integrating Sized and Dependent Types

Andreas Abel
Department of Computer Science
Ludwig-Maximilians-University Munich, Germany [*]
andreas.abel@ifi.lmu.de

Sized types are a modular and theoretically well-understood tool for checking termination of recursive and productivity of corecursive definitions. The essential idea is to track structural descent and guardedness in the type system to make termination checking robust and suitable for strong abstractions like higher-order functions and polymorphism. To study the application of sized types to proof assistants and programming languages based on dependent type theory, we have implemented a core language, MiniAgda, with explicit handling of sizes. New considerations were necessary to soundly integrate sized types with dependencies and pattern matching, which was made possible by concepts such as inaccessible patterns and parametric function spaces. This paper provides an introduction to MiniAgda by example and informal explanations of the underlying principles.

1 Introduction

In the dependent type theories underlying the programming and proof languages of Coq [17], Agda [24], and Epigram [19], all programs need to be total to maintain logical consistency. This means that some analysis is required that ensures that all functions defined by recursion over inductive types terminate and all functions defined by corecursion into a coinductive type are productive, i. e., always yield the next piece of the output in finite time. The currently implemented termination analyses are based on the untyped structural ordering: In case of Coq, it is the guard condition [15], and in case of Agda, the foetus termination checker [4] with a recent extension to size-change termination [18, 25]. 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.

As alternative to untyped termination checking, type-based methods have been proposed [7, 26, 10, 2]. The common idea is to annotate data types with a size index which is either the precise size of elements in this slice of the data type or an upper bound on the size. For recursive calls it is checked that the sizes decrease, which by well-foundedness entails termination. Sized types provide a very principled approach to termination, yet the need to integrate sizes into the type system means that an implementation of sized types touches the very core of type theories. Most theoretical works have confined themselves to study sized types in a simply-typed or polymorphic setting, exceptions are Blanqui's work [10, 11] which adds sizes to the Calculus of Algebraic Constructions (CACSA), and $\mathsf{CIC}\widehat{\ }$ [8], which extends the Calculus of Inductive Constructions in a similar fashion. To this day, no mature implementation of sized types exists.

The language MiniAgda, which shall be presented in this article, is an implementation of a dependently typed core language with sized types. Developed by the author and Karl Mehltretter [20], it serves to explore the interaction of sized types with the other features of a dependently typed functional

[*]Part of this research was carried out as invited researcher (Oct 2009 – Mar 2010) in the PI.R2 project team of INRIA Rocquencourt at the PPS lab, Université Paris VII.

language which include pattern matching and large eliminations. A fundamental design choice of MiniAgda was that sizes are explicit: they are index arguments of data types, abstraction over sizes is ordinary lambda-abstraction, and instantiation of a size is ordinary application. One of the main lessons learned during thinking about the relationship of sized types and dependent types is that size arguments to functions are parametric, i.e., functions can never depend on sizes, sizes only serve to ensure termination and should be erased during compilation. Parametric functions are typed via a variation of Mishra-Linger and Sheard's Erasure Pure Type Systems (EPTS) [22] which rest on Miquel's [21] "implicit quantification".[1]

In the following we shall walk through the features of MiniAgda by example and explain the taken design choices. Formalization and meta-theoretical study of MiniAgda is work in progress.

2 Sized Types for Termination

Type-based termination [16, 26, 7, 10, 1] rests on two simple principles:

1. Attaching a size i to each inductive type D.
2. Checking that sizes decrease in recursive calls.

2.1 Attaching Sizes to Inductive Types

We attach a size i to each inductive type D, yielding a sized type D^i which contains only those elements of D whose height is below i. For the calculation of the height of an element d of D consider d as a tree, where the constructors of D count as nodes. In the case of Nat we have a linear tree with inner succ-nodes and a zero-leaf; the size of an element Nat is the number of constructors, which is the value of the number plus 1. In the case of List, we also have linear trees, since in cons a as only as of type List counts as subtree (even if a happens to be a list also). The height of a list is its length plus one, since nil is also counted. For the type of trees the size corresponds to the usual concept of tree height. In the case of infinitely branching trees, the height can be transfinite; this is why sizes are ordinals in general and not natural numbers. However, for the examples we present in the following, sizes will not exceed ω, so one may think about them as natural numbers for now.

Following the principle described above, sized natural numbers are given by the constructors:

$$\begin{array}{lcl} \mathsf{zero} & : & \forall i.\ \mathsf{Nat}^{i+1} \\ \mathsf{succ} & : & \forall i.\ \mathsf{Nat}^{i} \to \mathsf{Nat}^{i+1} \end{array}$$

The first design choice is how to represent size expressions i and $i+1$, sized types Nat^i and size polymorphism $\forall i$. In absence of dependent types, like in the work of Pareto et al. [16] and Barthe et al. [7], size expressions and sized types need to be special constructs, or, in a F^{ω}-like system, are modeled as type constructors [1]. In a dependently typed setting, we can model sizes as a type Size with a successor operation $\$: \mathsf{Size} \to \mathsf{Size}$, sized types just as members of $\mathsf{Size} \to \mathsf{Set}$ (where Set is the type of small types), and size quantification just as dependent function space $(i : \mathsf{Size}) \to C\ i$. This way, we can pass sizes as ordinary arguments to functions and avoid special syntactic constructs.

In MiniAgda, a sized type is declared using the `sized data` syntax.

[1]My original inspiration was Bernardo and Barras' Church-style version of implicit quantification [6] which is very similar to EPTS.

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

This declares an inductive family SNat indexed by a size with two explicitly sized constructors. The system checks that the type of SNat is a function type with domain Size, and the size index needs to be the first index of the family. The constructor declarations must start with a size quantification $(i : \mathsf{Size}) \to \ldots$. Each recursive occurrence of SNat needs to be labeled with size i and the target with size $\$i$, to comply with the concept of size as tree height as described above. The size assignment to constructors is completely mechanical and could be done automatically. Indeed, in a mature system like Agda or Coq, size assignment should be automatic, adding size information as hidden arguments. However, in MiniAgda, which lacks a reconstruction facility for omitted hidden arguments, we chose to be fully explicit. In our case, the advantage of explicit size assignment over automatic assignment is "what you see is what you get", sized constructors are used as they are declared.

As a first exercise, we write a "plus 2" function for sized natural numbers.

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

In this non-recursive definition $\lambda i \lambda n.\, \mathsf{succ}\ (\$i)\ (\mathsf{succ}\ i\ n)$, we first abstract over the size i of the argument n using an ordinary λ, and supply the size arguments to succ explicitly. To define a predecessor function on SNat, we use pattern matching.

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

To avoid non-linear left-hand sides of pattern matching equations, MiniAgda has *inaccessible patterns* [12], also called *dot patterns*. A dot pattern can be an arbitrary term, in our case it is the size expression $\$i$. A dot pattern does not bind any variables, and during evaluation dot patterns are not matched against. The dot pattern basically says "at this point, the only possible term is $\$i$".

To ensure totality and consistency, pattern matching must be complete. Currently, MiniAgda does not check completeness; the purpose of MiniAgda is to research sized types for integration into mature systems such as Agda and Coq, which already have completeness checks.

2.2 Functions are Parametric in Size Arguments

Sizes are a means to ensure termination, so they are static information for the type checker, and they should not be present at run-time. In particular, the result of a function should never be influenced by its size arguments, and we should not be allowed to pattern match on sizes. Only then it is safe to erase them during compilation. However, the *type* of a function does very well depend on size; while the first argument to pred is irrelevant, so we have $\mathsf{pred}\ i\ n = \mathsf{pred}\ j\ n$ for all sizes i, j, the types $\mathsf{SNat}\ (\$i)$ and $\mathsf{SNat}\ i$ are certainly different. The concept of "irrelevant in the term but not in the type" is nicely captured by Miquel's intersection types [21] which have been implemented by Barras [6] as a variant ICC^* of Coq and studied in Mishra-Linger's thesis [23]. Intersection types, which we write as $[x : A] \to B$ and would like to call *parametric function types*, are a generalization of polymorphism $\forall X.B$ in System F. Polymorphic functions also do not depend on their type arguments, but their types do.

Using parametric function types, we can refine our definition of SNat as follows.

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

We have now expressed that the size argument to the constructors is irrelevant, so they can be safely erased during compilation. Size arguments can *always* be irrelevant in programs. The type of predecessor is consequently refined as:

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

Note that we do bind variable i to the irrelevant size argument, but the type system of MiniAgda ensures that it only appears on the right hand side in irrelevant positions, i.e., in arguments to parametric functions.

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

Here, i is used as an argument to the parametric succ, which is fine. We would get a type error with the non-parametric definition of succ:

```
succ : (i : Size) -> SNat i -> SNat ($ i)
```

The precise typing rules for parametric functions have been formulated by Mishra-Linger and Sheard [22]. For our purposes, the informal explanation we have just given is sufficient.

2.3 Tracking Termination and Size Preservation

Consider the following implementation of subtraction. Here, we use the infinity size ∞, in concrete syntax #. An element of sized type $D\,\infty$ is one without known height bound. This means that for sized inductive types D we have the subtyping relationships $D\,i \leq D\,(\$i) \leq \cdots \leq D\,\infty$.

Also, we introduce *size patterns* $i > j$ to match a constructor $c : [j : \mathsf{Size}] \to \cdots \to D\,(\$j)$ against an argument of type $D\,i$. The size pattern $i > j$ binds size variable j and remembers that j is smaller than size variable i. The full pattern we write is $c\,(i > j)\,\vec{p}$ and the match introduces a size constraint $i > j$ into the type checking environment which is available to check the remaining patterns and the right hand side of the clause.

```
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
}
```

The last line contains one recursive call minus $j\ x\ y$ with a descent in all three arguments: the size decreases ($j < i$) and both SNat-arguments are structurally smaller ($x <$ succ $j\ x$ and $y <$ succ $\infty\ y$). Thus, termination can be certified by descent on any of the three arguments.

The type of minus expresses that minus takes an argument x of size at most i and an argument y of arbitrary size and returns a result of size at most i. Thus, sizes can be used to certify that the output is bound in size by one of the inputs. In our case, the sized type expresses that “subtracting does not increase a number". In the last line, we return minus $j\ x\ y$ which is of size j, but the type requires us to return something of size i. Since sizes are upper bounds, by the subtyping of MiniAgda we have $\mathsf{SNat}\ j \leq \mathsf{SNat}\ i$, so this example passes the type checker. To see that the first clause is well-typed, observe that zero $j : \mathsf{SNat}\ (\$j)$, and since $j < i$ entails $\$j \leq i$, we have zero $j : \mathsf{SNat}\ i$ by subtyping. Note that in MiniAgda we cannot express that the result of minus $i\ x\ (\mathsf{succ}\ \infty\ y)$ is strictly smaller in size than x.

Using the bound on the output size of minus we can certify termination of Euclidean division. A call to 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)
}
```

In the last line, since x is bounded by size j, so is minus $j\ x\ y$, hence, the recursive call to div can happen at size j which is smaller than i. That tracking a size bound comes for free with the type system is a major advantage of sized types over untyped termination approaches.

2.4 Interleaving Inductive Types

Another advantage of sized types is that they scale very well to higher-order constructions. In the following, we present an implementation of map for rose trees to demonstrate this feature.

If we do not require sizes, we can also define ordinary data types in MiniAgda. In a mature system, all data definitions would look "ordinary" but be sized behind the veil, so there would not be a need for separate syntactic constructs.

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

List is a parametric data type, and we declare that it is strictly positive in its parameter A by prefixing that parameter by two $+$ signs. This means in particular that List is a monotone type valued function, i.e., for $A \leq B$ we have $\mathsf{List}\ A \leq \mathsf{List}\ B$; this property is beneficial for subtype checking. Internally, the constructors of List are stored as follows:

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

Each parameter of the data type becomes a parametric argument of the data constructors. [2]

The map function for lists is parametric in types A and B. Observe the dot patterns in the matching against nil and cons!

```
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)
}
```

Rose trees are finitely branching node-labeled trees, where the collection of subtrees is organized as a list.

[2] This is a consequent analysis of data type parameters, compare this to Coq's handling of parameters which requires parameters in constructors to be absent in patterns but present in expressions.

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

MiniAgda checks data type definitions for strict positivity to avoid well-known paradoxes. In this case, it needs to ensure that Rose appears strictly positively in the type List (Rose A i). The test is trivially successful since we have declared A to appear only strictly positively in List A above. Also, the type argument A appears strictly positively in Rose itself since strictly positive type functions compose. So the positivity annotation of Rose is sound, and we could continue and define trees branching over roses etc.

A feature of sized types is that the map function for roses can be defined naturally using the map function for lists:

```
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)
}
```

Note that the recursive call mapRose A B f j is underapplied, the actual rose argument is missing. But since the size j of the rose argument is present, we can verify termination.[3] This is usually a severe problem for untyped termination checkers. The current solution in Coq requires Rose and List to be defined mutually, destroying important modularity in library development.

An alternative to sized types is the manual indexing of roses by their height as *natural number*. This can be done within the scope of ordinary dependent types. Yet we miss the comfortable subtyping for sized types. The height of a rose has to be computed manually using a maximum over a list, and the resulting programs will involve many casts to type-check.

3 Coinductive Types and Corecursion

Coinductive types admit potentially infinite inhabitants, the most basic and most prominent example being streams.

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

An element of Stream A is an infinite succession of elements of A knitted together by the cons coconstructor. Clearly, a function producing a stream cannot terminate, since it is supposed to produce an infinite amount of data. Instead of termination we require *productivity* [13] which means that always the next portion of the stream can be produced in finite time. A simple criterion for productivity which can be checked syntactically is guardedness [15]: In the function definition, we require the right hand side to be a coconstructor or a sequence of such, and each recursive call to be directly under, i. e., *guarded* by this coconstructor. This condition ensures that the recursive computation only continues after some initial piece of the result has been produced. For example, consider repeat A a which produces an infinite stream of as.

[3]Sized types even allow nested recursive calls [3].

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

The recursive call to repeat is directly under the coconstructor cons, so the guard condition is satisfied. The "directly under" is crucial, if we write cons $A\ a\ (f\ (\mathsf{repeat}\ A\ a))$ instead, then productivity is no longer clear, but depends on f. If f is a stream destructor like the tail function, which discards the first element of the stream and returns the rest, then we do not obtain a productive function. Indeed, for this definition of repeat, the expression tail $(\mathsf{repeat}\ A\ a)$ reduces to itself after one unfolding of the recursion, so the next element of the stream can never be computed. On the other hand, if f is a stream-constructing function or a depth-preserving function like the identity, then the new repeat is productive. Yet to be on the safe side, the syntactic guard condition as described by Coquand [13] and implemented in Coq needs to reject the definition nevertheless. In practice this is unsatisfactory, and some workarounds have been suggested, like representing streams as functions over the natural numbers [9] or defining a mixed inductive-coinductive type of streams with an additional constructor c_f which will be evaluated to f in a second step [14].

3.1 Tracking Guardedness with Sized Types

Using sized coinductive types we can keep track in the type system whether a function is stream destructing, stream constructing or depth preserving [16]. Thus, sized types can offer a systematic solution to the "guardedness-mediated-by-functions" problem.

Sized coinductive type definitions look very similar to their inductive counterpart: The rules to annotate the recursive occurrences of the coinductive type in the types of the coconstructors are identical to the rules for sized constructors.

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

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
}
```

What is counted by the size i of a Stream is a lower bound on the number of coconstructors or guards, which we call the *depth* of the stream. A fully constructed stream will always have size ∞, but during the construction of the stream we reason with approximations, i. e., streams which have depth i for some arbitrary i. This is dual to the definition of recursive functions over inductive types. Once they are fully defined, they handle trees of arbitrary height, i. e., size ∞, but to perceive their termination we assume during their construction that they can only handle trees up to size i, and derive from this that they handle also trees up to size $i+1$.

Using sized types, the definition of repeat looks as follows:

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

Assuming that repeat $A\ a\ i$ produces a well-defined stream of depth i, we have to show that repeat $A\ a\ (\$i)$ produces a stream of depth $i+1$. This is immediate since cons increases the depth by one. Technically, we have used a *successor pattern* $(\$i)$ on the left hand side. Matching on a size argument seems to violate the principle that sizes are irrelevant for computation. Also, not every size is a successor, so the matching seems incomplete. However, the match is justified by the greatest fixed-point semantics of coinductive types, as we will explain in the following.

In the semantics, we construct the approximation Stream $A\ i$ of the full type of streams over A by an induction on $i \in \{0,1,\dots,\omega\}$.

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

This is an approximation from above: we start at $\top$, the biggest set of our semantics, e. g., the set of all terms. A Stream A 0 can be arbitrary, there is no guarantee of what will happen if we try to look at its first element. To be on the safe side, we have to assume that taking the head or tail of such a “stream” will diverge. Now each constructor increases the depth of the stream by one, we obtain Stream A 1, Stream A 2 etc. An element of Stream $A\ i$ can be unrolled (at least) i times, i. e., we can take the tail i times without risking divergence. Finally, the limit Stream $A\ \omega$ is defined as the intersection of all Stream $A\ i$. Such a stream is completely defined, and we unroll it arbitrarily often.

Whenever we have to construct a Stream $A\ i$ for i a size variable, we can match size i against a successor pattern $(\$j)$. Clearly, if i stands for a successor size $n+1$, then the match succeeds with $j = n$. But also, if i stands for size $\omega = \infty$ which is the closure ordinal for all coinductive types then the match succeeds with $j = \infty$ (there is a catch, see Section 5.1). Now if i stands for 0, we have to produce something in Stream A 0, meaning that we can return anything; we have no specific obligation. In summary, if we produce something in Stream $A\ i$, matching i against a successor pattern is a complete match. Further, since there is only one case (successor), the match is irrelevant and we maintain the property that sizes do not matter computationally. Size matching on i can be generalized to the case that we have to produce something in $(\vec{x} : \vec{B}) \to \mathsf{Stream}\,A\ i$.

3.2 Depth-preserving Functions and Mediated Guardedness

Using the successor pattern, we can define map as a depth-preserving corecursive function on streams:

```
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)
}
```

By matching first on the size i (which is legal since we construct a $(A \to B) \to \mathsf{Stream}\,A\ i \to \mathsf{Stream}\,B\ i$ at this point), the last argument gets type Stream $A\ (\$i)$, so we can match on it, the only case being cons $.A\ .i\ x\ xs$. Note that unlike for inductive types we cannot match on Stream $A\ i$ since it might be a totally undefined stream. Technically, the matching on Stream $A\ i$ is prevented in MiniAgda by requiring that the size argument of a coconstructor pattern must be a dot pattern.

Similarly to map we define merge for streams. Its type expresses that the output stream has at least the same depth as both input streams have. The more precise information that its depth is the sum of the depths of the input streams is not expressible in the size language of MiniAgda. The information loss is substantiated by the use of subtyping $\mathsf{Stream}\,A\ (\$i) \leq \mathsf{Stream}\,A\ i$ in the recursive calls to merge.

```
    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))
    }
```

The code of merge uses an auxiliary function leq that compares to natural numbers and returns a Church-encoded Boolean (continuation-passing style). In other words, it is a fusion of a comparison and an if-then-else.

```
    fun leq : Nat -> Nat -> [C : Set] -> C -> C -> C
    { leq  zero     y        C t f = t
    ; leq (succ x)  zero     C t f = f
    ; leq (succ x) (succ y) C t f = leq x y C t f
    }
```

An example where corecursion goes through another function is the Hamming function which produces a stream of all the composites of two and three in order. In this case, the guarding coconstructor and the recursive calls are separated by applications of the depth-preserving map and merge.

```
    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)))
    }
```

Summing up, sized types offer an intuitive and comfortable way to track guardedness through function calls and overcome the limitations of syntactical guardedness checks. The technical solutions to integrate pattern matching with explicitly sized coconstructors are the successor pattern for sizes and the restriction of size matching to dot patterns in coconstructors.

4 Dependent Types

In this section, we give examples of proper type dependencies. First, we demonstrate that sized types harmonize with predicates, and then, with large eliminations.

4.1 Proofs in MiniAgda

In dependent type theory we represent (co)inductive predicates by (co)inductive families. For instance, stream equality, aka stream bisimilarity, can be defined by the following sized coinductive family.

```
sized codata StreamEq (A : Set) : (i : Size) -> Stream A i -> Stream A i -> Set
{
  bisim : [i : Size] -> [a : A] -> [as : Stream A i] -> [bs : Stream A i] ->
    StreamEq A i as bs ->
    StreamEq A ($ i) (cons A i a as) (cons A i a bs)
}
```

Two streams are equal if their heads are equal and their tails are equal. The latter condition leads to an infinite regression, therefore, stream equality has to be defined coinductively. The only coconstructor bisim takes an equality proof of *as* and *bs* to one of $\mathsf{cons}\,A\;i\;a\;as$ and $\mathsf{cons}\,A\;i\;a\;bs$. The other arguments of bisim are irrelevant, since they can be reconstructed from the type $\mathsf{StreamEq}\,A\;(\$i)\;(\mathsf{cons}\,A\;i\;a\;as)\;(\mathsf{cons}\,A\;i\;a\;bs)$ (Brady et al. [12]).

Note that in the type of the coconstructor bisim, the size i appears not only as index to the currently defined type StreamEq, but also to Stream. This makes sense, since depth i is sufficient for streams when comparing them up to depth i only. However, some care is necessary: uses of i need to be restricted in such a way that $\mathsf{StreamEq}\,A\;i$ is still antitone in i. In our case, this holds since $\mathsf{Stream}\,A\;i$ itself is antitone in i. If we replaced $\mathsf{Stream}\,A\;i$ by sized lists $\mathsf{List}\,A\;i$, then antitonicity of StreamEq would be lost, leading to unsoundness of subtyping. MiniAgda checks types of sized (co)constructors to ensure that monotonicity properties are retained.

Using StreamEq, we can prove properties about stream functions. For example, mapping a function f over a stream of as produces a stream of $(f\,a)$s.

```
cofun map_repeat : [A : Set] -> [B : Set] -> [i : Size] ->
  (f : A -> B) -> (a : A) ->
  StreamEq B i (repeat B (f a) i) (map A B i f (repeat A a i))
{
  map_repeat A B ($ i) f a = bisim B i (f a)
   (repeat B (f a) i) (map A B i f (repeat A a i))
   (map_repeat A B i f a)
}
```

4.2 Large Eliminations

A specific feature of dependent type theory is that one can define a type by case distinction or recursion on a value, which is called a large elimination (of the value). In this case, the shape of this type (is it a function type or a base type?) is not statically determined. Competing approaches to sized types, like $\mathsf{CIC}^{\widehat{}}$ [8] and CACSA [10], which do not treat sizes explicitly, cannot define a sized type by a large elimination, because sizes are assigned automatically and to statically visible (co)inductive types only.

In the following, we define an n-ary maximum function on natural numbers whose type expresses that the size of the output is bounded by the maximum size of all of the inputs. A binary maximum function can be defined as follows, using the built-in max function on sizes.

```
fun max2 : [i : Size] -> SNat i -> SNat i -> SNat i
{ max2 i (zero (i > j))    n                = n
; max2 i  m               (zero (i > j))    = m
; max2 i (succ (i > j) m) (succ (i > k) n) = succ (max j k) (max2 (max j k) m n)
}
```

The type of max2 expresses that both inputs and the output have the same upper bound (such a type is beyond $\mathsf{CIC}^{\frown}$). By virtue of subtyping, this type is equivalent to $\mathsf{SNat}\;i \to \mathsf{SNat}\;j \to \mathsf{SNat}\;(\mathsf{max}\;i\;j)$, however, we try to minimize the use of max since constraints like $i \leq \mathsf{max}\;j\;k$ slow down type-checking.[4]

[4]The constraint $i \leq \mathsf{max}\;j\;k$ simplifies to the disjunctive constraint $i \leq j \vee i \leq k$ which introduces a case distinction in the type checker, possibly duplicating computations.

Now, by induction on $n : \mathsf{SNat}\ \infty$ we define the type

$$\mathsf{Maxs}\ n\ i = \underbrace{\mathsf{SNat}\ i \to \cdots \to \mathsf{SNat}\ i}_{n \text{ times}} \to \mathsf{SNat}\ i$$

of the n-ary maximum function.

```
fun Maxs : SNat # -> Size -> Set
{ Maxs (zero .#  ) i = SNat i
; Maxs (succ .# n) i = SNat i -> Maxs n i
}
```

The $(n+1)$-ary maximum function maxs is now also defined by induction on n, using the binary maximum.

```
fun maxs : (n : SNat #) -> [i : Size] -> SNat i -> Maxs n i
{ maxs (zero .#)   i m = m
; maxs (succ .# n) i m = \ l -> maxs n i (max2 i m l)
}
```

5 Avoiding the Paradoxes

The theory and implementation of sized types requires some care, since there are some paradoxes lurking around (as in other areas of dependent type theory).

5.1 Non-continuous Types

A first paradox noticed by Hughes, Pareto, and Sabry [16] involves types of recursive functions which are not continuous in their size parameter. This phenomenon has been studied in detail by the author [2]. We briefly explain this issue here for the case of corecursive definitions.

To improve readability of the example to follow, let us first introduce an auxiliary function guard2 $j\ g$ which precomposes $g : \mathsf{Stream\ Nat}\ (\$j) \to \mathsf{Stream\ Nat}\ \infty$ with itself and with a guard.

```
fun guard2 : [j : Size] -> (Stream Nat ($ j) -> Stream Nat #)
                        -> (Stream Nat j     -> Stream Nat #)
{ guard2 j g xs = g (g (cons Nat j zero xs))
}
```

We now construct a corecursive function $f : [i : \mathsf{Size}] \to (\mathsf{Stream\ Nat}\ i \to \mathsf{Stream\ Nat}\ \infty) \to \mathsf{Stream\ Nat}\ i$. It expects an argument $g : \mathsf{Stream\ Nat}\ i \to \mathsf{Stream\ Nat}\ \infty$ whose type promises to add enough guards to a stream of depth i to make it infinitely guarded. For any $i < \omega$, such a function needs to actually add infinitely many guards, yet for $i = \omega$ it can be any function on streams, even a stream destructing function like tail. This non-continuous behavior gives rise to a paradox.

```
cofun f : [i : Size] -> (Stream Nat i -> Stream Nat #) -> Stream Nat i
{ f ($ j) g = guard2 j g (f j (guard2 j g))
}
```

If g is a stream constructing function or the identity, then guard2 $j\ g$ is guarding the recursive call to f, but if g is tail or another stream destructing function, then guard2 $j\ g$ is actually removing guards. The definition of f is fine as long as i is not instantiated to ∞, otherwise, we can construct a diverging term.

```
    eval let loop : Nat = head Nat # (f # (tail Nat #))
```

To exclude definitions like f, MiniAgda checks when matching a size variable i against a successor pattern $(\$ j)$ that the type of the result is *upper semi-continuous* [2] in i. The check fails since the type Stream Nat $i \to$ Stream Nat ∞ of g is neither antitonic nor inductive in i.

5.2 Size Patterns and Deep Matching

The mix of sized types with deep pattern matching leads to a new paradox which has been communicated to me by Cody Roux.

Consider the following sized inductive type which has an infinitely branching L and a binary constructor M. In the presence of infinite branching, some sizes are modeled by limit ordinals, and the closure ordinal is way above ω.

```
    sized data O : Size -> Set
    { Z : [i : Size] -> O ($ i)
    ; S : [i : Size] -> O i -> O ($ i)
    ; L : [i : Size] -> (Nat -> O i) -> O ($ i)
    ; M : [i : Size] -> O i -> O i -> O ($ i)
    }
```

By cases we define a kind of "predecessor" function on Nat $\to$ O $(\$\$i)$ which will be used later to fake a descent.

```
    let pre : [i : Size] -> (Nat -> O ($$ i)) -> Nat -> O ($ i)
      = \ i -> \ f -> \ n -> case (f (succ n))
        { (Z .($ i))   -> Z i
        ; (S .($ i) x) -> x
        ; (L .($ i) g) -> g n
        ; (M .($ i) a b) -> a
        }
```

The paradox arises when we confuse limit ordinals and successor ordinals. In a previous version of MiniAgda, it was possible to write the following deep match.

```
    let three : Nat = succ (succ (succ zero))

    fun deep : [i : Size] -> O i -> Nat
    { deep .($$$$ i) (M .($$$ i) (L .($$ i) f)  (S .($$ i) (S .($ i) (S i x))))
      = deep ($$$ i) (M   ($$ i) (L ($ i) (pre i f)) (f three))
    ; deep i x = zero
    }
```

We have managed to assign to f the type Nat $\to$ O $(\$\$i)$ on which we can apply the fake predecessor function. This makes the recursive call to deep appear with a smaller size. It is now easy to tie the loop:

```
    fun emb : Nat -> O #
    { emb zero = Z #
    ; emb (succ n) = S # (emb n)
    }
    eval let loop : Nat = deep # (M # (L # emb) (emb three))
```

In the work of Blanqui [10], deep is forbidden by a linearity condition on size variables (see his Def. 6). In MiniAgda, we avoid the paradox by introducing size patterns $(i > j)$. Now the left hand side of deep needs to be written as follows, and the right hand side no longer type-checks.

```
fun deep : [i : Size] -> O i -> Nat
{ deep i4
   (M (i4 > i3)
        (L (i3 > j2) f)
        (S (i3 > i2)
             (S (i2 > i1)
                  (S (i1 > i) x))))
  = deep _ (M _ (L _ (pre _ f)) (f three))
; deep i x = zero
}
```

Now $f : \mathsf{Nat} \to \mathsf{O}\ j_2$ is no longer a valid argument to pre since j_2 is not the double-successor of any size expression in scope; the holes _ cannot be filled with size expressions in a well-typed manner.

6 Conclusion

We have presented MiniAgda, a core dependently typed programming language with sized types. The language ensures termination of recursive functions and productivity of corecursive functions by type checking. MiniAgda is implemented in Haskell; the source code and a suite of examples are available on the author's homepage `http://www2.tcs.ifi.lmu.de/~abel/miniagda`.

The main focus of MiniAgda is the sound integration of sized types with pattern matching à la Agda [24]. Previous works [7, 1, 16] have treated sized types in a lambda-calculus with primitives for fixed points and case distinction. The exception is the work of Blanqui [10, 11] who considers type-based termination checking of rewrite rules in the Calculus of Constructions. Our work is distinguished from his in that we integrate sizes into the ordinary syntax of dependent types, so, sizes are first-class. Also, our treatment includes coinductive types and productivity.

In future work, we like to explore how to handle sized types more silently, such that they are transparent to the user. There is already a size constraint solver in MiniAgda which we did not include in our description. At the moment, one can replace size arguments on the right hand side of clauses by "_" and the system searches for the correct solution. To complete the picture, we need reconstruction of size patterns and a reconstruction of sized types in function signatures [8]. Finally, by adding Agda-style hidden arguments, sizes can disappear from the surface completely.

Acknowledgments. Big thanks to Karl Mehltretter who implemented the first version of MiniAgda [20]! Thanks to Ulf Norell with whom I implemented a preliminary version of sized types for Agda, and thanks to Nils Anders Danielsson for fruitful discussions on corecursion during the Agda Implementor's Meetings in Japan in 2008 and 2010. Thanks to Cody Roux for taking me to the CORIAS meeting in Val d'Ajol where we had many interesting discussions about sized types and their implementation. Finally, I am grateful for the comments of the anonymous referees that helped me to improve this paper significantly.

References

[1] Andreas Abel (2006): *A Polymorphic Lambda-Calculus with Sized Higher-Order Types.* Ph.D. thesis, Ludwig-Maximilians-Universität München.

[2] Andreas Abel (2008): *Semi-continuous Sized Types and Termination*. *Logical Methods in Computer Science* 4(2). CSL'06 special issue.

[3] Andreas Abel (2009): *Type-Based Termination of Generic Programs*. *Science of Computer Programming* 74(8), pp. 550–567. MPC'06 special issue.

[4] Andreas Abel & Thorsten Altenkirch (2002): *A Predicative Analysis of Structural Recursion*. *Journal of Functional Programming* 12(1), pp. 1–41.

[5] Roberto M. Amadio, editor (2008): *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, *Lecture Notes in Computer Science* 4962. Springer-Verlag.

[6] Bruno Barras & Bruno Bernardo (2008): *The Implicit Calculus of Constructions as a Programming Language with Dependent Types*. In Amadio [5], pp. 365–379.

[7] Gilles Barthe, Maria J. Frade, Eduardo Giménez, Luis Pinto & Tarmo Uustalu (2004): *Type-Based Termination of Recursive Definitions*. *Mathematical Structures in Computer Science* 14(1), pp. 97–141.

[8] Gilles Barthe, Benjamin Grégoire & Fernando Pastawski (2006): *CIC^: Type-Based Termination of Recursive Definitions in the Calculus of Inductive Constructions*. In: Miki Hermann & Andrei Voronkov, editors: *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, *Lecture Notes in Computer Science* 4246, Springer-Verlag, pp. 257–271.

[9] Yves Bertot & Ekaterina Komendantskaya (2008): *Using Structural Recursion for Corecursion*. In: Stefano Berardi, Ferruccio Damiani & Ugo de'Liguoro, editors: *Types for Proofs and Programs, International Conference, TYPES 2008, Torino, Italy, March 26-29, 2008, Revised Selected Papers*, *Lecture Notes in Computer Science* 5497, Springer-Verlag, pp. 220–236.

[10] Frédéric Blanqui (2004): *A Type-Based Termination Criterion for Dependently-Typed Higher-Order Rewrite Systems*. In: Vincent van Oostrom, editor: *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3 – 5, 2004, Proceedings*, *Lecture Notes in Computer Science* 3091, Springer-Verlag, pp. 24–39.

[11] Frédéric Blanqui (2005): *Decidability of Type-Checking in the Calculus of Algebraic Constructions with Size Annotations*. In: C.-H. Luke Ong, editor: *Computer Science Logic, 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005, Proceedings*, *Lecture Notes in Computer Science* 3634, Springer-Verlag, pp. 135–150.

[12] Edwin Brady, Conor McBride & James McKinna (2004): *Inductive Families Need Not Store Their Indices*. In: Stefano Berardi, Mario Coppo & Ferruccio Damiani, editors: *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, *Lecture Notes in Computer Science* 3085, Springer-Verlag, pp. 115–129.

[13] Thierry Coquand (1993): *Infinite Objects in Type Theory*. In: H. Barendregt & T. Nipkow, editors: *Types for Proofs and Programs (TYPES '93)*, *Lecture Notes in Computer Science* 806, Springer-Verlag, pp. 62–78.

[14] Nils Anders Danielsson (2010): *Beating the Productivity Checker Using Embedded Languages*. Workshop on Partiality And Recursion in Interactive Theorem Provers (PAR 2010), Satellite Workshop of ITP'10 at FLoC 2010.

[15] Eduardo Giménez (1995): *Codifying Guarded Definitions with Recursive Schemes*. In: Peter Dybjer, Bengt Nordström & Jan Smith, editors: *Types for Proofs and Programs, International Workshop TYPES´94, Båstad, Sweden, June 6-10, 1994, Selected Papers*, *LNCS* 996, Springer, pp. 39–59.

[16] John Hughes, Lars Pareto & Amr Sabry (1996): *Proving the Correctness of Reactive Systems Using Sized Types*. In: *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'96*, pp. 410–423.

[17] INRIA (2008): *The Coq Proof Assistant Reference Manual*. INRIA, version 8.2 edition. http://coq.inria.fr/.

[18] Chin Soon Lee, Neil D. Jones & Amir M. Ben-Amram (2001): *The Size-Change Principle for Program Termination*. In: *ACM Symposium on Principles of Programming Languages (POPL'01)*, ACM Press, London, UK, pp. 81–92.

[19] Conor McBride & James McKinna (2004): *The View from the Left*. *Journal of Functional Programming* .

[20] Karl Mehltretter (2007): *Termination Checking for a Dependently Typed Language*. Master's thesis, Department of Computer Science, Ludwigs-Maximilians-University Munich.

[21] Alexandre Miquel (2001): *The Implicit Calculus of Constructions*. In: Samson Abramsky, editor: *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings*, *Lecture Notes in Computer Science* 2044, Springer-Verlag, pp. 344–359.

[22] Nathan Mishra-Linger & Tim Sheard (2008): *Erasure and Polymorphism in Pure Type Systems*. In Amadio [5], pp. 350–364.

[23] Richard Nathan Mishra-Linger (2008): *Irrelevance, Polymorphism, and Erasure in Type Theory*. Ph.D. thesis, Portland State University.

[24] Ulf Norell (2007): *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden.

[25] David Wahlstedt (2007): *Dependent Type Theory with Parameterized First-Order Data Types and Well-Founded Recursion*. Ph.D. thesis, Chalmers University of Technology. ISBN 978-91-7291-979-2.

[26] Hongwei Xi (2002): *Dependent Types for Program Termination Verification*. *Journal of Higher-Order and Symbolic Computation* 15(1), pp. 91–131.