Well-Founded Sized Types in the Calculus of 
(Co)Inductive Constructions *

Jorge Luis Sacchini

Qatar Campus, Carnegie Mellon University, Qatar
sacchini@qatar.cmu.edu

Type-based termination is a mechanism for ensuring termination and productivity of (co)recursive definitions [4]. Its main feature is the use of sized types (i.e. types annotated with size information) to track the size of arguments in (co)recursive calls. Termination of recursive function (and productivity of corecursive functions) is ensured by restricting recursive calls to smaller arguments (as evidenced by their types). Type-based approaches to termination and productivity have several advantages over the syntactic-based approaches currently implemented in Coq and Agda [5,1,2]. In particular, they are more expressive and easier to understand.

In previous work [6,5], we studied the metatheory of extensions of the Calculus of (Co)Inductive Constructions (CIC) with a simple notion of sized types. While these systems are more expressive than the guard predicates of Coq, there are two main shortcomings in the case of coinductive types. First, corecursive definitions that use `tail` are not expressible, e.g.:

\[
corec \text{zeroes} := \text{cons}(0, \text{cons}(0, \text{tail zeroes})).
\]

Second, Subject Reduction (SR) does not hold (a well-known problem in Coq that was observed by Gimnez [3]). While in practice, the lack of SR does not seem to be a major issue, it is theoretically unsatisfying.

In this work, we consider an extension of CIC with a notion of well-founded sized types, inspired by $F_{\omega}^{\text{cop}}$ [2], that solves both issues. As in previous work, (co)inductive types are annotated with size expressions taken from the size algebra given by $s ::= \mathbf{1} | s + 1 | \infty$, where $\mathbf{1}$ is a size variable. In this approach, the (simplified) typing rule for streams looks like:

\[
\frac{f : \mathbf{| j < s |} \text{stream}^{\mathbf{s}} A \vdash t : \text{stream}^{\mathbf{j}} A \quad \mathbf{s fresh}}{\vdash (\text{corec } f := t) : \forall \mathbf{s} . \text{stream}^{\mathbf{s}} A}
\]

The type $\text{stream}^{\mathbf{s}} A$ is the type of streams (whose elements have type $A$) where at least $\mathbf{s}$ elements can be produced. The definition above ensures that $t$ can only make (co)recursive calls on smaller (i.e. already produced) streams. Since $\mathbf{j} < \mathbf{s}$ we ensure that at least one more element is produced by $t$. We require that each corecursive call to $f$ be explicitly applied to a size $\mathbf{s}$ satisfying the constraint $\mathbf{s} < \mathbf{j}$.

The type of the stream constructor, $\text{cons}$, is (informally) the following:

\[
\text{cons} : (\forall j < s . A \times \text{stream}^{\mathbf{j}} A) \to \text{stream}^{\mathbf{s}} A
\]

To construct an element of type $\text{stream}^{\mathbf{s}} A$ we need to provide the head of type $A$ and the tail of type $\text{stream}^{\mathbf{j}}$ for an arbitrary $j < s$. To destruct $\text{cons}$ above, we must apply it to a size $\mathbf{r}$ satisfying $\mathbf{r} < \mathbf{s}$.

\*This publication was made possible by a JSREP grant (JSREP 4:004:1-001) from the Qatar National Research Fund (a member of The Qatar Foundation). The statements made herein are solely the responsibility of the author.
Let us see how well-founded sized types overcome the first limitation of our previous approach mentioned above. Consider the following definition of the stream of Fibonacci numbers:

\[
\text{FIB} \equiv \text{corec} \; \text{fib} := \text{cons}(j, 0, \text{cons}(\kappa. 1, \text{sum}[\kappa] \text{fib}[\kappa] (\text{tail}[\kappa] \text{fib}[j])))
\]

where, given \( t : \text{stream}^s A \), \( \text{tail}[r](t) : \text{stream}^s A \) for any \( r < s \), and \( \text{sum} : \forall \kappa. \text{stream}^s \text{nat} \rightarrow \text{stream}^s \text{nat} \) computes the point-wise addition of two streams of natural numbers.

This definition is well typed. The size variables introduced \((j, \kappa)\) satisfy \( \kappa < j \) and \( j < 1 \).

The recursive call \( \text{fib}[j] \) has type \( \text{stream}^s \text{nat} \), and \( \text{tail}[\kappa] \text{fib}[j] \) has type \( \text{stream}^s \text{nat} \).

Let us consider the second issue mentioned above: lack of SR. In CIC (and Coq), unfolding of (co)recursive definitions is restricted in order to have a strongly normalizing evaluation. In the case of Coq, unfolding of corecursive definitions is only allowed within case analysis (which leads to the loss of SR).

With well-founded sized types, we use the size annotations to restrict unfolding. Corecursive functions have types of the form \( \forall \kappa. \text{stream}^s A \) (in general, \( \forall \kappa. \Pi \Delta \text{stream}^s A \)). Like corecursive calls, we require that corecursive functions be applied to a size annotation (size application is denoted \( t[s] \)). We restrict unfolding to the case where the size annotation is \( \infty \):

\[
(\text{corec} \; f := t)[s] \rightarrow t[f/\text{corec} \; f := t] \quad \text{if} \quad s = \infty
\]

This reduction strategy satisfies SR (the proof is standard) [7]. Furthermore, we have proved that this reduction relation is strongly normalizing. For instance, it prevents infinite unfolding of \( \text{tail} \text{FIB} \):

\[
\begin{align*}
\text{tail}[\infty] \text{FIB}[\infty] & \rightarrow \text{tail}[\infty] (\text{cons}(j, 0, \text{cons}(\kappa. 1, \text{sum}[\kappa] \text{fib}[\kappa] (\text{tail}[\kappa] \text{fib}[j]))) \\
& \rightarrow \text{cons}(\kappa. 1, \text{sum}[\kappa] \text{fib}[\kappa] (\text{tail}[\kappa] \text{fib}[\infty])) \\
& \rightarrow \text{cons}(\kappa. 1, \text{sum}[\kappa] \text{fib}[\kappa] (\text{cons}(\kappa'. 1, \text{sum}[\kappa'] \text{fib}[\kappa'] (\text{tail}[\kappa'] \text{fib}[\kappa]))))
\end{align*}
\]

Unfolding cannot proceed until the stream is further observed (e.g. by applying \( \text{tail} \) again).

This work is still in progress; the current version is available at [7]. We have proved most of the metatheory, including strong normalization. We have also proved that size-inference is decidable, i.e. size annotation can be inferred from minimal user-provided annotations, and we implemented a small prototype (available at [7]). Our prototype performs size inference; however, the algorithm used is very inefficient. We are currently investigating more efficient alternatives.

References


