Termination Checking Nested Inductive and Coinductive Types

Thorsten Altenkirch and Nils Anders Danielsson
University of Nottingham

Abstract

In the dependently typed functional programming language Agda one can easily mix induction and coinduction. The implementation of the termination/productivity checker is based on a simple extension of a termination checker for a language with inductive types. However, this simplicity comes at a price: only types of the form $\nu X.\mu Y.F\ X\ Y$ can be handled directly, not types of the form $\mu Y.\nu X.F\ X\ Y$. We explain the implementation of the termination checker and the ensuing problem.

1 Introduction

This short and speculative note discusses how one can—apparently—extend a termination checker which accepts structurally recursive programs so that it also accepts guarded corecursive programs (and proofs) and even mixed recursive/corecursive definitions. However, we will also point out a problem with the extended checker: the "obvious" way to represent a coinductive type nested within an inductive type does not work.

Some familiarity with total, dependently typed languages, induction, coinduction, structural recursion and guarded corecursion is assumed.

2 foetus

Originally the termination checker of the dependently typed functional programming language Agda (Norell 2007; Agda Team 2010) only supported structural recursion. The checker was based on foetus (Abel and Altenkirch 2002), which will now be explained using the following two, mutually recursive (and contrived) functions:

mutual

$$
\begin{aligned}
& f : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
& f\ m\ \mathsf{zero} \quad\ \ = m \\
& f\ m\ (\mathsf{suc}\ n) = f\ m\ n + g\ m \\
& g : \mathbb{N} \to \mathbb{N} \\
& g\ \mathsf{zero} \quad\ \ = \mathsf{zero} \\
& g\ (\mathsf{suc}\ n) = f\ n\ n
\end{aligned}
$$

The definitions of f and g are accepted by foetus, which works roughly as follows:

- For every function clause $f\ p_1\ \ldots\ p_m$ and every call site $g\ e_1\ \ldots\ e_n$ in the right-hand side of the clause, the following information is noted for every pattern-argument pair (p_i, e_j): Is e_j structurally strictly smaller than p_i, or is it equal to p_i? The former case is denoted by $<$, the latter by $=$, and otherwise the symbol ? is used.

  In the case of our example we have three calls. If we write the information as *call matrices* it looks as follows (one row per caller argument, one column per callee argument):

$$
f \to f : \quad \begin{pmatrix} = & ? \\ ? & < \end{pmatrix} \qquad\qquad f \to g : \quad \begin{pmatrix} = \\ ? \end{pmatrix} \qquad\qquad g \to f : \quad \begin{pmatrix} < & < \end{pmatrix}
$$

- This information is then combined into information about every (kind of) call path from a function to itself.

  For our example we get three kinds of call paths, denoted as vectors with one element per argument:

  1. $(=,<)$, which corresponds to f's call to itself,
  2. $(<,?)$, which includes the call sequence $f \to g \to f$, and
  3. $(<)$, which includes the call sequence $g \to f \to g$.

- Finally we need to check if, for every function, there is some lexicographic combination of arguments such that every kind of call path is strictly decreasing.

  In the case of f we need to choose the lexicographic combination (first argument, second argument), and in the case of g the only argument is strictly decreasing.

3 Coinductive Definitions in Agda

This section contains a crash course on the approach to coinduction taken in Agda. For more information, see Danielsson and Altenkirch (2010, Section 2). First consider the following Agda definition of the type of infinite streams:

$$\begin{array}{l} \textbf{data}\ \mathit{Stream}\ (A : \mathit{Set}) : \mathit{Set}\ \textbf{where} \\ \quad \_{::}\_ : A \to \infty\ (\mathit{Stream}\ A) \to \mathit{Stream}\ A \end{array}$$

The type constructor $\infty : \mathit{Set} \to \mathit{Set}$ makes its argument *coinductive*. The best way to get an intuition about ∞ may be to view it as the suspension type constructor which is sometimes used to encode non-strictness in strict languages (Wadler et al. 1998). The type constructor comes with a force function and a (tightly binding) delay constructor:

$$\begin{array}{ll} \flat & : \{A : \mathit{Set}\} \to \infty\, A \to A \\ \sharp\_ & : \{A : \mathit{Set}\} \to A \to \infty\, A \end{array}$$

Now consider the following definition of stream processors (Hancock et al. 2009):

$$\begin{array}{l} \textbf{data}\ \mathit{SP}\ (A\ B : \mathit{Set}) : \mathit{Set}\ \textbf{where} \\ \quad \mathsf{get} : (A \to \mathit{SP}\ A\ B) \to \mathit{SP}\ A\ B \\ \quad \mathsf{put} : B \to \infty\ (\mathit{SP}\ A\ B) \to \mathit{SP}\ A\ B \end{array}$$

A stream processor is either a command to read (get) another element from the input stream, and use this element to guide the rest of the computation, or a command to output (put) an element, and continue with another stream processor. The use of ∞ only for put means that a stream processor may contain an infinite number of consecutive put constructors, but only a finite number of consecutive get constructors. This is ensured by the termination checker.[1]

Agda supports structural recursion for inductive types, and guarded corecursion for coinductive types. These recursion principles can also be combined "lexicographically", as explained in the next section.

[1] We hope that it is, anyway. Neither Agda's meta-theory nor its implementation have been formally verified to be correct.

4 An Extension of foetus Which Handles Guarded Corecursion

When Agda was extended to support coinductive data types and guarded corecursion Andreas Abel just made a small change to the termination checker: an extra row and column was added to the call matrices, representing *guardedness*.

An example will illustrate the change. Consider the following definition of the semantics of a stream processor:

$$\begin{array}{l}
[\![_]\!] : \{A\ B : Set\} \rightarrow SP\ A\ B \rightarrow Stream\ A \rightarrow Stream\ B \\
[\![\ \mathsf{get}\, f \quad]\!]\ (a :: as) \;=\; [\![\, f\ a\,]\!]\ (^\flat\, as) \\
[\![\ \mathsf{put}\, b\ sp\ ]\!]\ as \qquad\;\, =\; b :: {}^\sharp\, [\![\, {}^\flat\, sp\,]\!]\ as
\end{array}$$

The first recursive call is not guarded by the coinductive constructor $^\sharp\_$, but no non-constructor function is used between the left-hand side and the call, so we say that it *preserves guardedness* $(=)$. On the other hand, in the second clause the recursive call is guarded $(<)$. We get the following call matrices, where the topmost, leftmost element represents guardedness, and the remainder of the first rows and columns do not represent anything; the rest of the matrices represent structural relations between the four arguments of $[\![\_]\!]$:

$$[\![\_]\!] \rightarrow [\![\_]\!] : \begin{pmatrix} = & ? & ? & ? & ? \\ ? & = & ? & ? & ? \\ ? & ? & = & ? & ? \\ ? & ? & ? & < & ? \\ ? & ? & ? & ? & ? \end{pmatrix} \qquad [\![\_]\!] \rightarrow [\![\_]\!] : \begin{pmatrix} < & ? & ? & ? & ? \\ ? & = & ? & ? & ? \\ ? & ? & = & ? & ? \\ ? & ? & ? & ? & ? \\ ? & ? & ? & ? & = \end{pmatrix}$$

Note that $^\flat\, sp$ is not viewed as structurally smaller than sp (this measure only applies to the inductive parts of types), and that $f\ a$ is viewed as structurally strictly smaller than $\mathsf{get}\, f$ (higher-order primitive recursion).

The call matrices above give rise to three kinds of call paths:

1. $(=,=,=,<,?)$, corresponding to the first recursive call,
2. $(<,=,=,?,=)$, corresponding to the second recursive call, and
3. $(<,=,=,?,?)$, corresponding to call paths which involve both recursive calls.

It is easy to see that one gets a strictly decreasing combination by lexicographically pairing the first component (guardedness) with the fourth (the inductive structure of the stream processor).

At this point we should note that we have not seen a proof of correctness for the extended termination checker described above. However, we would not be surprised if the general principle[2] of the checker turned out to be correct.

5 Problem

As stated in Section 4 we suspect that the termination checker may ensure totality. However, we have not tried to prove this, because there is a problem with it: quantifier inversion.

Consider the following definitions of colists and potentially infinitely branching trees:

[2]The interaction with the other features of Agda is a different question, as is the implementation itself.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Colist}\ (A : \mathit{Set}) : \mathit{Set}\ \textbf{where} \\
\quad [\,] \quad : \mathit{Colist}\ A \\
\quad \_{::}\_ : A \to \infty\ (\mathit{Colist}\ A) \to \mathit{Colist}\ A \\
\textbf{data}\ \mathit{Tree} : \mathit{Set}\ \textbf{where} \\
\quad \mathsf{node} : \mathit{Colist}\ \mathit{Tree} \to \mathit{Tree}
\end{array}
$$

One might believe that this type should be read as the nested fixpoint $\mu X.\ \nu Y.\ 1 + X \times Y$ (in the category of sets and total functions). However, the termination checker described above accepts the following definition, in which all the recursive calls are guarded:

$$
\begin{array}{l}
\textbf{mutual} \\
\quad \mathit{bad} : \mathit{Tree} \\
\quad \mathit{bad}\ =\ \mathsf{node}\ (\mathsf{node}\ [\,] :: \sharp\ \mathit{bads}) \\
\quad \mathit{bads} : \mathit{Colist}\ \mathit{Tree} \\
\quad \mathit{bads}\ =\ \mathit{bad} :: \sharp\ \mathit{bads}
\end{array}
$$

The tree *bad* could not be defined if *Tree* defined the type $\mu X.\ \nu Y.\ 1 + X \times Y$, because the latter type only admits a finite number of uses of node in a tree. The problem here seems to be that the termination checker is too untyped—it only cares about delay constructors, not about which fixpoint they "belong" to. In this case the delay constructors for the inner fixpoint work as guards also for the outer fixpoint.

In an attempt to understand what is going on we take a domain-theoretic view of the language. Assume that (simply typed) data type definitions can be understood as follows:

- First, view ∞ as lifting of (not necessarily pointed) CPOs, and the other constructions as follows: sums as separated sums, products as unlifted products, functions as continuous functions, and definitions by recursion as least fixpoints.
- Then, for every type, limit the semantic domain to a total subset of the corresponding CPO, in such a way that isomorphic CPOs are restricted in the same way. For first-order (closed) types the total subset should consist of the maximal elements.

If the explanation above is right, then *Tree* should be viewed as the maximal elements of the CPO $\mu X.\ \mu Y.\ 1 + X \times Y_\perp$. Note that this CPO is isomorphic to $\mu Y.\ \mu X.\ 1 + X \times Y_\perp$, with swapped quantifiers. This CPO arises from the type Tree':

$$
\begin{array}{l}
\textbf{data}\ \mathit{SnocList}\ (A : \mathit{Set}) : \mathit{Set}\ \textbf{where} \\
\quad [\,] \quad : \mathit{SnocList}\ A \\
\quad \_{::}\_ : \mathit{SnocList}\ A \to A \to \mathit{SnocList}\ A \\
\textbf{data}\ \mathit{Tree}' : \mathit{Set}\ \textbf{where} \\
\quad \mathsf{node} : \mathit{SnocList}\ (\infty\ \mathit{Tree}') \to \mathit{Tree}'
\end{array}
$$

If the description above is correct, then the types *Tree* and Tree' should be isomorphic, with simple isomorphisms. Due to the untyped nature of the termination checker we can actually prove that they are.[3] Figure 1 contains two functions *from* and *to*, accepted by the termination checker described in Section 4, which witness this isomorphism. By defining equality of coinductive types as bisimilarity—see, for instance, Danielsson and Altenkirch (2010, Section 2.3)—it is also easy to prove (inside the system described above) that the functions are inverses.

[3] All closed, infinite first-order types are isomorphic, but the proof still works if we make *Tree* and Tree' parametrised.

mutual

$$\begin{aligned}
&\mathit{from}_1 : \mathit{Tree} \to \mathit{SnocList}\,(\infty\,\mathit{Tree}') \\
&\mathit{from}_1\,(\mathsf{node}\,\mathit{ts}) = \mathit{from}_2\,\mathit{ts} \\
&\mathit{from}_2 : \mathit{Colist}\,\mathit{Tree} \to \mathit{SnocList}\,(\infty\,\mathit{Tree}') \\
&\mathit{from}_2\,[\,] = [\,] \\
&\mathit{from}_2\,(t :: \mathit{ts}) = \mathit{from}_1\,t :: \sharp\,\mathsf{node}\,(\mathit{from}_2\,(\flat\,\mathit{ts}))
\end{aligned}$$

$$\begin{aligned}
&\mathit{from} : \mathit{Tree} \to \mathit{Tree}' \\
&\mathit{from}\,t = \mathsf{node}\,(\mathit{from}_1\,t)
\end{aligned}$$

mutual

$$\begin{aligned}
&\mathit{to}_1 : \mathit{Tree}' \to \mathit{Colist}\,\mathit{Tree} \\
&\mathit{to}_1\,(\mathsf{node}\,\mathit{ts}) = \mathit{to}_2\,\mathit{ts} \\
&\mathit{to}_2 : \mathit{SnocList}\,(\infty\,\mathit{Tree}') \to \mathit{Colist}\,\mathit{Tree} \\
&\mathit{to}_2\,[\,] = [\,] \\
&\mathit{to}_2\,(\mathit{ts} :: t) = \mathsf{node}\,(\mathit{to}_2\,\mathit{ts}) :: \sharp\,\mathit{to}_1\,(\flat\,t)
\end{aligned}$$

$$\begin{aligned}
&\mathit{to} : \mathit{Tree}' \to \mathit{Tree} \\
&\mathit{to}\,t = \mathsf{node}\,(\mathit{to}_1\,t)
\end{aligned}$$

Figure 1: Functions witnessing the isomorphism between *Tree* and *Tree*′.

We note also that the termination checker does seem to handle types like *Tree*′ correctly, i.e. like the fixpoint $\nu Y.\ \mu X.\ 1 + X \times Y$ in the category of sets and total functions: one cannot make use of delay constructors to define infinitely long snoc-lists, because the left argument of $\_{::}\_$ has type *SnocList A*, not $\infty\,(\mathit{SnocList}\,A)$. In other words, the termination checker seems to invert the quantifiers of *Tree* so that it behaves more like *Tree*′.

6 Discussion

We have sketched a simple method, due to Andreas Abel, for extending a termination checker aimed at structural recursion so that it also handles guarded corecursion. We have also pointed out a problem with the method: it leads to "quantifier inversion", which means that nested fixpoints of the form $\mu Y.\nu X.F\,X\,Y$ cannot in general be handled directly.

Given the simplicity of the extension of the termination checker we raise a question: is it possible to make a further small modification to it so that it can handle arbitrary nested fixpoints in a nice way?

References

Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12(1):1–41, 2002.

The Agda Team. The Agda Wiki. Available at `http://wiki.portal.chalmers.se/agda/`, 2010.

Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, declaratively; an exercise in mixed induction and coinduction. To appear in the proceedings of the Tenth International Conference on Mathematics of Program Construction (MPC'10), 2010.

Peter Hancock, Dirk Pattinson, and Neil Ghani. Representations of stream processors using nested fixed points. *Logical Methods in Computer Science*, 5(3:9), 2009.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology and Göteborg University, 2007.

Philip Wadler, Walid Taha, and David MacQueen. How to add laziness to a strict language, without even being odd. In *Proceedings of the 1998 ACM SIGPLAN Workshop on ML*, 1998.