Termination Checking in the Presence of 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 $X.Y.F\ X\ Y$ can be handled directly, not types of the form $Y.X.F\ X\ Y$. 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

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

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

- For every function clause $h\ p_1\ \ldots\ p_m$ and every call site $i\ e_1\ \ldots\ e_n$ 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 using *call matrices* it looks as follows (one row per caller argument, one column per callee argument):

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

- 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 $f\ \ g\ \ f$, and
  3. $(<)$, which includes the call sequence $g\ \ f\ \ g$.

- 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:

```
data Stream (A : Set) : Set where
  __ : A  (Stream A)  Stream A
```

The use of the type constructor : *Set* *Set* makes *Stream* 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:

```
  : {A : Set}  A  A
_ : {A : Set}  A  A
```

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

```
data SP (A B : Set) : Set where
  get : (A  SP A B)  SP A B
  put : B  (SP A B)  SP A B
```

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] Perhaps. 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:

$$
\begin{array}{l}
\_ : \{A\ B : \mathit{Set}\}\ \ \mathit{SP}\ A\ B\ \ \mathit{Stream}\ A\ \ \mathit{Stream}\ B \\
\mathsf{get}\ f \quad (a\ \ \mathit{as}) = f\ a\ \ (\ \mathit{as}) \\
\mathsf{put}\ b\ \mathit{sp}\ \ \mathit{as} \quad\ = b \quad\ \mathit{sp}\ \ \mathit{as}
\end{array}
$$

The first recursive call is not guarded by the coinductive constructor _, 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 _:

$$
\_\ \_ : \begin{pmatrix} = & ? & ? & ? & ? \\ ? & = & ? & ? & ? \\ ? & ? & = & ? & ? \\ ? & ? & ? & < & ? \\ ? & ? & ? & ? & ? \end{pmatrix} \qquad \_\ \_ : \begin{pmatrix} < & ? & ? & ? & ? \\ ? & = & ? & ? & ? \\ ? & ? & = & ? & ? \\ ? & ? & ? & ? & ? \\ ? & ? & ? & ? & = \end{pmatrix}
$$

Note that *sp* is not viewed as structurally smaller than put *b sp* (this measure only applies to the inductive parts of types), and that *f a* is viewed as structurally strictly smaller than 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).

We have not seen a proof of correctness for the extended termination checker described above. It is plausible that it ensures totality, at least if the rest of the language is restricted in a suitable way. However, we have not tried to prove this. The reason is that the checker makes the language somewhat strange, as described in the next section.

5 Quantifier Inversion

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

```
data Colist (A : Set) : Set where
  []  : Colist A
  __  : A (Colist A) Colist A
```

```
data Tree : Set where
  node : Colist Tree Tree
```

One might believe that the type *Tree* should be read as the nested fixpoint $X.\ Y.\ 1 + X\ \ Y$ (in the category of sets and total functions). However, the termination checker described above accepts the following definition:

```
mutual
  bad : Tree
  bad = node (node [] bads)
  bads : Colist Tree
  bads = bad ∷ bads
```

The tree *bad* could not be defined if *Tree* defined the type $X.\ Y.\ 1 + X\ \ Y$: *bad* is used in the definition of itself. The problem 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 ($Y.\ \ldots$) work as guards also for the outer fixpoint.

We conjecture that one can understand (a first-order fragment of) Agda's data type definitions—in the presence of the termination checker described above—by the following translation into a simpler core theory. For a given program we first define a type of codes for all the data types in the program (including). In the case of the example above we get the following type (where the notation $(\mathsf{c}_1 : T_1) + \ldots + (\mathsf{c}_n : T_n)$ is used for labelled sums):

```
Type : Set
Type = T. (colist : T) + (tree : 1) + (inf : T)
```

The three constructors represent *Colist*, *Tree*, and . The second step is to translate all data type definitions into a single nested fixpoint, indexed by type codes:

```
Data : Type → Set
Data = C. I. t.   ([]   : (t : Type) . t → colist t)
                + (_∷_  : (t : Type) . t → colist t → I t → I (inf (colist t)))
                + (node :                  t → tree tt → I (colist (tree tt)))
                + (_    : (t : Type) . t → inf t → C t)
```

Here we have, for instance, that *Data* (tree tt) represents *Tree* (tt is the only closed inhabitant of 1).

Note that, under the translation above, *all* data types have the form $Y.\ X.\ F\ X\ Y$. In particular, the termination checker seems to *invert* the quantifiers of *Tree* so that it behaves more like *Tree*:

```
data SnocList (A : Set) : Set where
  []  : SnocList A
  _∷_ : SnocList A → A → SnocList A
```

```
data Tree : Set where
  node : SnocList (Tree) → Tree
```

When translating *SnocList* and *Tree* we get the following types:

```
Type : Set
Type = T.
    (snocList : T)
  + (tree : 1)
  + (inf : T)
```

```
Data : Type → Set
Data = C. I. t.
    ([]   : (t : Type) . t → snocList t)
  + (_∷_  : (t : Type) . t → snocList t → I (snocList t) → I t)
  + (node :                  t → tree tt → I (snocList (inf (tree tt))))
  + (_    : (t : Type) . t → inf t → C t)
```

```
mutual
  from₁ : Tree  SnocList ( Tree)
  from₁ (node ts) = from₂ ts
  from₂ : Colist Tree  SnocList ( Tree)
  from₂ []       = []
  from₂ (t  ts) = from₁ t  node (from₂ ( ts))
from : Tree  Tree
from t = node (from₁ t)
```

```
mutual
  to₁ : Tree  Colist Tree
  to₁ (node ts) = to₂ ts
  to₂ : SnocList ( Tree)  Colist Tree
  to₂ []       = []
  to₂ (ts  t) = node (to₂ ts)  to₁ ( t)
to : Tree  Tree
to t = node (to₁ t)
```

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

It is not too hard to see that *Data* (tree tt) and *Data* (tree tt) are isomorphic (and not only because the types have the same size; the proof works also if we make *Tree* and *Tree* parametrised). As an indication that Agda actually behaves in accordance with the translation we can also prove (inside Agda) that *Tree* and *Tree* are isomorphic; for functions witnessing the isomorphism, see Figure 1.

As a final remark we note that the termination checker does seem to handle types like *Tree* correctly, i.e. like the fixpoint $Y.\ X.\ 1 + X\ \ Y$: one cannot make (direct) use of delay constructors to define infinitely long snoc-lists, because the left argument of $\_\_$ has type *SnocList A*, not (*SnocList A*).

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 $X.\ Y.\ F\ X\ Y$ 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? Note that this involves two things: rejecting definitions like *bad*, but also accepting other, currently rejected, definitions, corresponding to the recursion principles associated with types like $X.\ Y.\ F\ X\ Y$.

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.