Guarded Dependent Type Theory with Coinductive Types

Aleš Bizjak[1], Hans Bugge Grathwohl[1], Ranald Clouston[1], Rasmus E. Møgelberg[2] and Lars Birkedal[1]

[1] Department of Computer Science, Aarhus University, Denmark
[2] IT University of Copenhagen, Denmark
{abizjak|hbugge|ranald.clouston|birkedal}@cs.au.dk, mogel@itu.dk

Modern implementations of intensional dependent type theories, such as Coq, Agda, and Idris, have been used successfully for programming and proving in many projects. However they offer limited support for *coinductive* types.

For programming, the challenge is to ensure that functions on coinductive types are well-defined; that is, productive with unique solutions. Syntactic guardedness checks, as used for example in Coq, ensure productivity by requiring that recursive calls be nested directly under a constructor, but such checks exclude many valid definitions, particularly in the presence of higher-order functions.

For proving, the challenge is to reconcile the extensional nature of coinductive types, which are characterised by observations, with intensional equality. For example, the natural notion of bisimulation on streams does not coincide with inhabitation of the identity type. Thus one is forced to introduce a non-standard notion of equality for coinductive types, which will typically not be a congruence [4]. It would be preferable if one could instead directly use the identity relation of the type theory.

For the programming challenge, a *type-based* approach to guarded recursion, more flexible than syntactic checks, was suggested by Nakano [6]. A new modality $\triangleright$, called 'later', allows us to distinguish between data we have access to now, and data which we have only later. $\triangleright$ must guard self-reference in type definitions, so for example *guarded streams* of natural numbers are described by the guarded recursive equation $\mathsf{Str}^g_{\mathbb{N}} \simeq \mathbb{N} \times \triangleright \mathsf{Str}^g_{\mathbb{N}}$ asserting that stream heads are available now, but tails only later.

Using $\triangleright$ alone, however, enforces a discipline more rigid than productivity. For example, all stream functions must be *causal* [3], so elements of the result must not depend on deeper elements of the argument, ruling out the 'every other' function that returns every second element of a given stream. This limitation motivated the introduction of *clock quantifiers* by Atkey and McBride [1], which permit $\triangleright$ to be eliminated in a controlled way. In earlier work [2] we recast these quantifiers as the unary type-former $\square$, called 'constant'. Moreover such type-formers give rise to types whose denotation is exactly that of standard coinductive types [5, Theorem 2]. Hence $\square$ allows us to program and prove with 'real' coinductive types, with guarded recursion providing the 'rule format' for their manipulation. For example, the coinductive type of streams of natural numbers can be defined simply as $\mathsf{Str}_{\mathbb{N}} \triangleq \square\, \mathsf{Str}^g_{\mathbb{N}}$.

In previous work [2], we introduced the *guarded λ-calculus*, $\mathsf{g}\lambda$, a simply typed lambda calculus with guarded and coinductive types, and a logic, $L\mathsf{g}\lambda$, as a separate layer, to prove properties of elements. The logic allowed us to prove properties of coinductive types, but not in the integrated fashion supported by dependent type theories. In this talk we introduce guarded dependent type theory, gDTT, which supports programming and proving with guarded and coinductive dependent types.

For the proving challenge, our approach is to use guarded recursion to prove intensional equality of terms of guarded recursive types. From this we can derive equalities of terms of coinductive types, thus circumventing the need for a non-standard equality type for coinductive types.

One of the key challenges in designing gDTT is coping with elements that are only available later, i.e., elements of types of form $\triangleright A$. With gλ we had the term formation rules

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{next}\, t : \triangleright A} \qquad \frac{\Gamma \vdash f : \triangleright(A \to B) \qquad \Gamma \vdash t : \triangleright A}{\Gamma \vdash f \circledast t : \triangleright B}$$

The first rule allows us to make later use of data that we have now. The second intuitively says that if f will later be a function mapping A to B, and we will later have an element t of A, then, later, we can apply f to t to get an element of B. Let us consider the $\circledast$ rule for dependent types, where function spaces are generalised to Π-types. Supposing that f has type $\triangleright(\Pi x : A.B)$, what type should $f \circledast t$ have? If B depends on x, it does not make sense to follow the simply-typed approach and use $\triangleright B$. But since t has type $\triangleright A$, and not A, we cannot substitute t for x. Intuitively, t will eventually reduce to some value $\mathsf{next}\, u$, and so the resulting type should be $\triangleright B[u/x]$. But if t is an open term we may not be able to perform this reduction, so cannot type our term. To address this issue we introduce a new notion, of *delayed substitution*, allowing us to express the resulting type as $\triangleright[x \leftarrow t].B$. Definitional equality rules allow us to simplify this type when t is a value, i.e., $\triangleright[x \leftarrow \mathsf{next}\, u].B \simeq \triangleright B[u/x]$ as expected.

The novel features of gDTT allow us to address both the programming and proving challenges posed by coinductive types. To illustrate, consider addition of two guarded streams:

$$\mathsf{plus}^g \triangleq \begin{array}{l} \mathsf{fix}\, \phi.\lambda(xs : \mathsf{Str}^g_{\mathbb{N}})(ys : \mathsf{Str}^g_{\mathbb{N}}). \\ \mathsf{cons}^g\, ((\mathsf{hd}^g\, xs) + (\mathsf{hd}^g\, ys))\ (\phi \circledast \mathsf{tl}^g\, xs \circledast \mathsf{tl}^g\, ys) \end{array} \ :\ \mathsf{Str}^g_{\mathbb{N}} \to \mathsf{Str}^g_{\mathbb{N}} \to \mathsf{Str}^g_{\mathbb{N}}$$

Here hd^g takes the head of a stream, and tl^g its tail (recalling that tails of guarded streams have type $\triangleright \mathsf{Str}^g_{\mathbb{N}}$). This function as whole is defined by guarded recursion, so ϕ has type $\triangleright(\mathsf{Str}^g_{\mathbb{N}} \to \mathsf{Str}^g_{\mathbb{N}} \to \mathsf{Str}^g_{\mathbb{N}})$. The $\circledast$-application $\phi \circledast \mathsf{tl}^g\, xs \circledast \mathsf{tl}^g\, ys$ has type $\triangleright \mathsf{Str}^g_{\mathbb{N}}$, so the $\mathsf{cons}^g(\cdots)$ sub-expression has type $\mathsf{Str}^g_{\mathbb{N}}$.

With gDTT we can prove the commutativity of addition on streams. Assuming that c is a proof term witnessing commutativity of $+$ on natural numbers, and that $\mathsf{p}\eta$ is a proof term yielding equality of two pairs when given equality proofs of their respective components, we can construct a proof:

$$p \triangleq \begin{array}{l} \mathsf{fix}\, \phi.\lambda\, (xs, ys : \mathsf{Str}^g_{\mathbb{N}})\, . \\ \mathsf{p}\eta\ (c\, (\mathsf{hd}^g\, xs)\, (\mathsf{hd}^g\, ys))\ (\phi \circledast \mathsf{tl}^g\, xs \circledast \mathsf{tl}^g\, ys) \end{array} \ :\ \Pi(xs, ys : \mathsf{Str}^g_{\mathbb{N}}).\mathsf{Id}_{\mathsf{Str}^g_{\mathbb{N}}}(\mathsf{plus}^g\, xs\, ys, \mathsf{plus}^g\, ys\, xs).$$

where Id is the identity type-former. Note that p is again defined by guarded recursion, and that it is quite simple: it says that to show commutativity of plus^g we proceed by using commutativity of $+$ on the heads, then continuing recursively on the tails. While p itself is simple, showing that p indeed has the right type makes use of the new features in gDTT. For example, the subterm $\phi \circledast \mathsf{tl}^g\, xs \circledast \mathsf{tl}^g\, ys$ cannot be typed without use of the generalised $\triangleright$ carrying a delayed substitution.

We can lift the function plus^g to a function on *coinductive* streams $\mathsf{plus} : \mathsf{Str}_{\mathbb{N}} \to \mathsf{Str}_{\mathbb{N}} \to \mathsf{Str}_{\mathbb{N}}$. Likewise, using the new features of gDTT, we can turn the proof p of commutativity of plus^g into one for plus, i.e., construct a term of type $\Pi(xs, ys : \mathsf{Str}_{\mathbb{N}}).\mathsf{Id}_{\mathsf{Str}_{\mathbb{N}}}(\mathsf{plus}\, xs\, ys, \mathsf{plus}\, ys\, xs)$.

Thus we can use guarded recursion to program and prove properties on guarded streams, and then convert these into programs and proofs for coinductive streams.

This abstract is based on a recently submitted paper.

References

[1] R. Atkey and C. McBride. Productive coprogramming with guarded recursion. In *Proc. of ICFP '13*, pp. 197–208. ACM, 2013.

[2] R. Clouston, A. Bizjak, H. B. Grathwohl, and L. Birkedal. Programming and reasoning with guarded recursion for coinductive types. In *Proc. of FoSSaCS 2015*, v. 9034 of *Lect. Notes in Comput. Sci.*, pp. 279–294.Springer, 2015.

[3] N. R. Krishnaswami and N. Benton. Ultrametric semantics of reactive programs. In *Proc. of LICS '11*, pp. 257–266. IEEE CS, 2011.

[4] C. McBride. Let's see how things unfold: Reconciling the infinite with the intensional. In *Proc. of CALCO '09*, v. 5728 of *Lect. Notes in Comput. Sci.*, pp. 113–126. Springer, 2009.

[5] R. E. Møgelberg. A type theory for productive coprogramming via guarded recursion. In *Proc. of CSL-LICS '14*, art. 71. ACM, 2014.

[6] H. Nakano. A modality for recursion. In *Proc. of LICS '00*, pp. 255–266. IEEE CS, 2000.