Guarded Dependent Type Theory with Coinductive Types

Aleš Bizjak[1] Hans Bugge Grathwohl[1] Ranald Clouston[1]
Rasmus Ejlers Møgelberg[2] Lars Birkedal[1]

[1] Department of Computer Science, Aarhus University, Denmark
[2] IT University of Copenhagen, Denmark

21st International Conference on Types for Proofs and Programs, TYPES 2015
18 May 2015

Dependent Type Theories and Coinductive Types

Modern implementations of intensional dependent type theories are widely used for programming and theorem proving.

- Coq, Agda, Idris, ...

But support for coinductive types e.g. streams, is weak.

Key idea: dependent type theory with coinductive types defined via type-based guarded recursion.

The Problem of Productivity

Well-formed coinductive definitions must be productive: yielding e.g. stream elements 'on demand'.

Syntactic guarded recursion, as used e.g. by Coq, requires that recursive calls be nested directly under constructors.

- zeros = 0 :: zeros ✓
- vacuous = vacuous ×
- wrong = 0 :: tail wrong ×

But this does not work well with modular programming:

- nats = 0 :: map succ nats

Why is 'tail' bad but 'map succ' OK here?

Type-based Guarded recursion

[Nakano 2000] brought the syntactic side-condition into the type system via the later modality $\triangleright$.

- $\mathsf{Str}^g_{\mathbb{N}} = \mathbb{N} \times \triangleright \mathsf{Str}^g_{\mathbb{N}}$

Self-references then have $\triangleright$ added to their type:

- zeros $=$ 0 :: zeros ✓ pairing a $\mathbb{N}$ with a $\triangleright \mathsf{Str}^g_{\mathbb{N}}$
- vacuous $=$ vacuous × left is $\mathsf{Str}^g_{\mathbb{N}}$, right is $\triangleright \mathsf{Str}^g_{\mathbb{N}}$

Key rules

The guarded lambda calculus, gλ [Clouston et al 2015], and related systems have the applicative functor rules

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

Our nats example, rejected by Coq, can hence be typed:

- nats $=$ 0 :: (next(map succ)) $\circledast$ nats

But our ‘wrong’ example still cannot be typed:

- wrong $=$ 0 :: (next tail) $\circledast$ wrong $\times$

Note: 'map succ' is $\mathsf{Str}^g_{\mathbb{N}} \to \mathsf{Str}^g_{\mathbb{N}}$, but ‘tail’ is $\mathsf{Str}^g_{\mathbb{N}} \to \triangleright\mathsf{Str}^g_{\mathbb{N}}$

Applicative Functor Rules and Dependent Types

Now generalise function spaces to Π-types. What is the $\circledast$-rule?

$$\frac{\Gamma \vdash f : \triangleright(\Pi x : A.B) \qquad \Gamma \vdash t : \triangleright A}{\Gamma \vdash f \circledast t : ?}$$

We cannot substitute t for x because they do not have the same type!

If t had form $\mathsf{next}\, u$ then $f \circledast t$ should have type $\triangleright B[u/x]$.

But in general we look stuck: the type depends on data we get later; but we would like to type the term now!

Delayed Substitutions

The solution: delayed substitutions

$$\frac{\Gamma \vdash f : \triangleright(\Pi x : A.B) \qquad \Gamma \vdash t : \triangleright A}{\Gamma \vdash f \circledast t : \triangleright[x \leftarrow t].B}$$

With equation

$$\triangleright[x \leftarrow \mathsf{next}\, u].B \quad \simeq \quad B[u/x]$$

The term-former next may be similarly decorated.

The actual rule is slightly more complicated because to apply $\circledast$ repeatedly we may need delayed substitutions on multiple variables.

An Example - Stream Addition

We define stream addition $\mathsf{Str}^g_{\mathbb{N}} \to \mathsf{Str}^g_{\mathbb{N}} \to \mathsf{Str}^g_{\mathbb{N}}$ as

$$\mathsf{plus}^g\, xs\, ys \;=\; (\mathsf{hd}^g\, xs + \mathsf{hd}^g\, ys) :: (\mathsf{plus}^g \circledast \mathsf{tl}^g\, xs \circledast \mathsf{tl}^g\, ys)$$

Now suppose we wanted to prove that this is commutative, i.e. define an inhabitant p of type

$$\Pi\big(xs, ys : \mathsf{Str}^g_{\mathbb{N}}\big).\mathsf{Id}_{\mathsf{Str}^g_{\mathbb{N}}}(\mathsf{plus}^g\, xs\, ys, \mathsf{plus}^g\, ys\, xs)$$

We do this by guarded recursion!

$$\mathsf{p}\, xs\, ys \;=\; \mathsf{q}\,(\mathsf{c}(\mathsf{hd}^g\, xs)(\mathsf{hd}^g\, ys))\,(\mathsf{p} \circledast (\mathsf{tl}^g\, xs) \circledast (\mathsf{tl}^g\, ys))$$

(where c is commutativity of $+$ and q lifts pairs of proofs to a proof about pairs).

Stream Addition cont'd

The term

$$\mathsf{p}\, xs\, ys \;=\; \mathsf{q}\,(\mathsf{c}(\mathsf{hd}^g\, xs)(\mathsf{hd}^g\, ys))\,(\mathsf{p} \circledast (\mathsf{tl}^g\, xs) \circledast (\mathsf{tl}^g\, ys))$$

could hardly be simpler. But checking it is well-typed is not trivial.

In particular subterm $\mathsf{p} \circledast (\mathsf{tl}^g\, xs) \circledast (\mathsf{tl}^g\, ys)$ has type

$$\mathsf{Id}_{\triangleright \mathsf{Str}^g_C} \left(\begin{array}{l} \mathsf{next} \left[\begin{array}{c} xs' \leftarrow \mathsf{tl}^g\, xs \\ ys' \leftarrow \mathsf{tl}^g\, ys \end{array} \right] \mathsf{plus}^g\, f\, xs'\, ys', \\ \mathsf{next} \left[\begin{array}{c} xs' \leftarrow \mathsf{tl}^g\, xs \\ ys' \leftarrow \mathsf{tl}^g\, ys \end{array} \right] \mathsf{plus}^g\, f\, ys'\, xs' \end{array} \right)$$

So delayed substitutions are essential to real proofs.

Another Example - Covectors

Vectors – lists with their lengths – are the archetypal dependently typed data structure.

Covectors are colists (potentially infinite lists) with their length, which is a co-natural number:

$$\mathsf{Co}\mathbb{N} \simeq \mathbf{1} + \triangleright\,\mathsf{Co}\mathbb{N}$$

and (omitting some syntax regarding constructions with universes):

$$\begin{aligned}\mathsf{CoVec}_A\, n = \mathsf{case}\, n\, \mathsf{of}\, &\mathsf{inl}\, u \Rightarrow \mathbf{1}\\ &\mathsf{inr}\, m \Rightarrow A \times \triangleright(\mathsf{CoVec}_A \circledast m)\end{aligned}$$

Covectors cont'd

$$\begin{aligned}\mathsf{zeros}\, n \;=\; & \mathsf{case}\, n\, \mathsf{of}\, \mathsf{inl}\, u \Rightarrow \mathsf{inl}\, \langle\rangle \\ & \qquad\qquad \mathsf{inr}\, m \Rightarrow 0 :: \mathsf{zeros} \circledast m.\end{aligned}$$

Here m has type $\triangleright\, \mathsf{Co}\mathbb{N}$, but does not start with next, so $\mathsf{zeros} \circledast m$ must have type

$$\triangleright\, [n \leftarrow m]\, .\Pi(n : \mathsf{Co}\mathbb{N}).\, \mathsf{CoVec}_{\mathbb{N}}\, n$$

So even the very simplest constructions on covectors require delayed substitutions.

The Topos of Trees

The gλ-calculus can be given semantics in the topos of trees:

$$A_1 \xleftarrow{a_1} A_2 \xleftarrow{a_2} A_3 \xleftarrow{a_3} \cdots$$

e.g. guarded streams of natural numbers:

$$\mathbb{N} \xleftarrow{pr_1} \mathbb{N} \times \mathbb{N} \xleftarrow{pr_1} (\mathbb{N} \times \mathbb{N}) \times \mathbb{N} \longleftarrow \cdots$$

The $\triangleright$ modality simply adds a trivial set to the first position:

$$1 \xleftarrow{!} A_1 \xleftarrow{a_1} A_2 \xleftarrow{a_2} A_3 \xleftarrow{a_3} \cdots$$

Semantics for Guarded Dependent Type Theory

Semantics for dependent type theory often based on indexed sets.

- For coherence reasons, actual semantics are a little more complex.

[Birkedal et al 2011] shows how this approach can be lifted to the topos of trees.

- In particular there is a sensible semantics for $\triangleright$.
- We have defined a sensible semantics for $\triangleright$ generalised to carry delayed substitutions.

Soundness: All rules of our type theory are validated by our model.

Causality

Extending types with $\triangleright$ rules out many productive functions by enforcing causality:

- results cannot depend on later elements of arguments.

$$\mathsf{everyother}\, xs = \mathsf{hd}\, xs :: \mathsf{everyother}(\mathsf{tl}\ \mathsf{tl}\, xs)$$

$\mathsf{everyother}(\mathsf{tl}\ \mathsf{tl}\, xs)$ seems to have type $\triangleright\triangleright\,\mathsf{Str}^g_{\mathbb{N}}$, so we are stuck.

Solution: Clock quantifiers [Atkey-McBride 2013], or in our simplified presentation, the constant modality $\Box$.

If A is a type that may have some 'temporal' content, $\Box A$ provides that type all at once.

Constant Streams

An example from the model: recall $\mathsf{Str}^g_{\mathbb{N}}$ is

$$\mathbb{N} \xleftarrow{pr_1} \mathbb{N} \times \mathbb{N} \xleftarrow{pr_1} (\mathbb{N} \times \mathbb{N}) \times \mathbb{N} \longleftarrow \cdots$$

Then $\Box\,\mathsf{Str}^g_{\mathbb{N}}$ is

$$\mathbb{N}^\omega \xleftarrow{id} \mathbb{N}^\omega \xleftarrow{id} \mathbb{N}^\omega \longleftarrow \cdots$$

So $\Box\,\mathsf{Str}^g_{\mathbb{N}}$ is just ordinary streams $\mathsf{Str}_{\mathbb{N}}$! We have regained standard coinductive types.

everyother can be defined as a function $\mathsf{Str}_{\mathbb{N}} \to \mathsf{Str}_{\mathbb{N}}$.

Introducing the Constant Modality

If $\square$ could be introduced freely this would trivialise our guardedness guarantees.

Instead the $\square$ modality can only be introduced in constant context: context with no temporal content:

$$\frac{\Gamma' \vdash A \text{ type} \qquad \Gamma' \vdash \square \text{ ctx} \qquad \rho : \Gamma \to \Gamma'}{\Gamma \vdash \square \rho . A \text{ type}} \; \square\text{-W}$$

Here ρ is an ordinary context morphism, 'closing' a type to allow contant types to be used in larger programs with temporal content.

We now have a previous term-former prev to eliminate $\triangleright$, legal only in constant context.

- e.g. dealing with the $\triangleright\triangleright$ for the everyother function.

Ongoing Work – Our Current Biggest Headache

We need canonicity – all closed terms should be definitionally equal to some notion of value for its type.

Closed terms of identity types should be equal to reflexivity proofs.

But our current support for identity proofs for streams seems too powerful – streams defined via extensionally equal but definitionally non-equal functions from the natural numbers can be proved equal in a way that doesn't reduce to reflexivity.

On the other hand, we *want* non-trivial proofs of stream equalities!

Searching for solutions (Observational Type Theory?).

Ongoing Work – Future Challenges

We need operational results – decidable type-checking!

Infinite data poses an obvious challenge to this.

- Don't unfold forever!
- Solved for the simply typed gλ-calculus, at least.

Comparison needed with type-based 'competitor', sized types.

- More mature approach, going back to [Hughes et al 1996].
- If we can crack the problem of reasoning directly via intensional equality types, rather than indirectly via bisimulation arguments, that would be exciting.

Thank you ... any questions?