Let's see how things unfold: reconciling the infinite with the intensional (extended abstract)

Conor McBride

University of Strathclyde

1 Introduction

Coinductive types model infinite structures unfolded on demand, like politicians' excuses: for each attack, there is a defence but no likelihood of resolution. Representing such evolving processes coinductively is often more attractive than representing them as functions from a set of permitted observations, such as projections or finite approximants, as it can be tricky to ensure that observations are meaningful and consistent. As programmers and reasoners, we need coinductive definitions in our toolbox, equipped with appropriate computational and logical machinery.

Lazy functional languages like HASKELL [18] exploit call-by-need computation to over-approximate the programming toolkit for coinductive data: in a sense, all data is coinductive and delivered on demand, or not at all if the programmer has failed to ensure the *productivity* of a program.

Tatsuya Hagino pioneered a more precise approach, separating initial data from final codata [10]. The corresponding discipline of 'coprogramming' is given expression in Cockett's work on CHARITY [5, 6] and in the work of Turner and Telford on 'Elementary Strong Functional Programming' [22, 21, 23]. Crucially, all distinguish recursion (structurally decreasing on input) from *corecursion* (structurally increasing in output). As a total programmer, I am often asked 'how do I implement a *server* as a program in your terminating language?', and I reply that I do not: a server is a *coprogram* in a language guaranteeing liveness.

To combine programming and reasoning, or just to program with greater precision, we might look to the proof assistants and functional languages based on intensional type theories, which are now the workhorses of formalized mathematics and metatheory, and the mainspring of innovation in typed programming [16, 4, 14]. But we are in for a nasty shock if we do. Coinduction in COQ is *broken*: computation does not preserve type. Coinduction in AGDA is *weak*: dependent observations are disallowed, so whilst we can unfold a process, we cannot *see* that it yields its unfolding.

At the heart of the problem is *equality*. Intensional type theories distinguish two notions of equality: the typing rules identify types and values according to an equality *judgment*, decided mechanically during typechecking; meanwhile, we can express equational *propositions* as types whose inhabitants (if we can find them) justify the substitution of like for like.

In neither Coq nor Agda is a coprogram judgmentally equal to its unfolding, hence the failure in the former. That is not just bad luck: in this presentation, I check that it is impossible for any decidable equality to admit unfolding.

Moreover, neither system admits a substitutive propositional equality which identifies bisimilar processes, without losing the basic computational necessity that closed expressions compute to canonical values [13]. That is just bad luck: in this presentation, I show how to construct such a notion of equality, following earlier joint work with Altenkirch on observational equality for functions [2].

The key technical ingredient is the notion of 'interaction structure' due to Hancock and Setzer [11] — a generic treatment of indexed coinductive datatypes, which I show here to be closed under its own notion of bisimulation. This treatment is ready to be implemented in a new version of the Epigram system.

Equipped with a substitutive propositional equality that includes bisimulation, we can rederive Coq's dependent observation for codata from Agda's simpler coalgebraic presentation, whilst ensuring that what types we have, we hold. Let's see how things unfold.

2 The Problem

Eduardo Giménez pioneered Coq's treatment of coinduction [7]. It was a great step forward in its time, giving Coq access to many new application domains. Giménez was aware of the problem with type preservation, giving a counterexample in his doctoral thesis [8]. The problem did not become particularly widely known until recently, when Nicolas Oury broke an overenthusiastic early version of coinduction in Agda, then backported his toxic program to Coq, resulting in a flurry of activity on mailing lists which has not yet entirely subsided.

Presented with a categorical flavour, Coq's treatment is essentially thus: for any given strictly positive functor $F : \mathsf{Set} \to \mathsf{Set}$, we acquire a coinductive set νF equipped with a coconstructor and a coiterator (or 'unfold', or 'anamorphism') which grows νF values on demand by successive applications of a coalgebra to a 'seed' of arbitrary type. Keeping polymorphism implicit, we obtain:

$$\begin{aligned} \nu F &: \mathsf{Set} \\ \mathsf{in}_F &: F\ (\nu F) \to \nu F \\ \mathsf{coit}_F &: (S \to F\ S) \to S \to \nu F \end{aligned}$$

Of course, Coq actually provides a much richer notation for coprograms than just coit_F, but a streamlined presentation will help to expose the problem.

For the standard example of coinductive lists, this specializes (modulo high school algebra) to the traditional pair of coconstructors and unfold.

$$\begin{aligned} \mathsf{CoList}_X &: \mathsf{Set} \\ \mathsf{nil}_X &: \mathsf{CoList}_X \\ \mathsf{cons}_X &: X \to \mathsf{CoList}_X \to \mathsf{CoList}_X \\ \mathsf{unfold}_X &: (S \to 1 + X \times S) \to S \to \mathsf{CoList}_X \end{aligned}$$

For a given seed $s : S$, the unfold_X coalgebra may deliver a value $\mathsf{inl}\,()$ from the left summand to construct nil_X, or some $\mathsf{inr}(x, s')$ to construct a cons_X with head x and a tail grown on demand from seed s'. For a standard example, construct the infinite sequence of natural numbers, starting from a given counter:

$$\mathsf{natsFrom}\; n \;\mapsto\; \mathsf{unfold}_{\mathbb{N}}\;(\lambda n \mapsto \mathsf{inr}\;(n, n+1))\; n$$

The corresponding elimination behaviour is *dependent* case analysis, not merely branching according to a coconstructor choice, but *seeing* every value as an unfolding to a coconstructor. I write $(x : S) \to T$ for dependent function spaces with explicit application. In general and particular, we acquire:

$$\begin{array}{lll}
\mathsf{case}_F & : & (P{:}\nu F \to \mathsf{Set}) \;\to\; ((t{:}F\;(\nu F)) \to P\;(\mathsf{in}_F\;t)) \;\to\; (x{:}\nu F) \to P\;x \\
\mathsf{caseCoList}_X & : & (P{:}\mathsf{CoList}_X \to \mathsf{Set}) \to \\
 & & P\;\mathsf{nil}_X \to \\
 & & ((x{:}X) \to (xs{:}\mathsf{CoList}_X) \to P\;(\mathsf{cons}_X\;x\;xs)) \to \\
 & & (xs{:}\mathsf{CoList}_X) \to P\;xs
\end{array}$$

We may readily recover the destructor — the terminal coalgebra — as a degenerate case analysis:

$$\begin{array}{l}
\mathsf{out}_F : \nu F \to F\;(\nu F) \\
\mathsf{out}_F = \mathsf{case}_F\;(\lambda\_ \mapsto \nu F)\;(\lambda t \mapsto t)
\end{array}$$

Writing $F\,-$ for the functorial action on functions, we can readily see that $\mathsf{coit}_F\;(F\;\mathsf{out}_F)$ does the same job as in_F, but the latter is not redundant, as it is needed to form the type of case_F.

Intensional type theories are computational — if we test a CoList for emptiness, we should certainly receive true or false. It is thus vital to animate case analysis with operational behaviour. Moreover, whatever we choose will play a key role in the type system. Judgmental equality, which I write $\equiv$, is given in Coq as the congruence and equivalence closure of reduction, $\leadsto$. Decidable type-checking thus rests on terminating reduction. We take care to coiterate only on demand from case analysis, hence finitely often. Coq computes as follows (but what is the type of the term in the box?)

$$\begin{array}{ll}
\mathsf{case}_F\;P\;p\;(\mathsf{in}_F\;t) & \leadsto p\;t \\
\mathsf{case}_F\;P\;p\;(\mathsf{coit}_F\;f\;s) & \leadsto \boxed{p\;(F\;(\mathsf{coit}_F\;f)\;(f\;s))}
\end{array}$$

In the special case of CoList, computing the application of coalgebra to seed delivers a value for $p\,s$ which decides whether the resulting CoList is a nil_X to be replaced by n or a cons_X to be replaced by c. (Again, typecheck the boxes.)

$$\begin{array}{ll}
\mathsf{caseCoList}_X\;P\;n\;c\;\mathsf{nil}_X & \leadsto n \\
\mathsf{caseCoList}_X\;P\;n\;c\;(\mathsf{cons}_X\;x\;xs) & \leadsto c\;x\;xs \\
\mathsf{caseCoList}_X\;P\;n\;c\;(\mathsf{unfold}_X\;p\;s) & \leadsto \begin{cases} \boxed{n} & \text{if } p\;s \leadsto^* \mathsf{inl}\;() \\ \boxed{c\;x\;(\mathsf{unfold}\;p\;s')} & \text{if } p\;s \leadsto^* \mathsf{inr}\;(x, s') \end{cases}
\end{array}$$

Here inl and inr are injections to the sum $1+X\times S$ directing coconstructor choice.

Coiteration on demand rules out spontaneous unfolding

$$\begin{aligned}
&\mathsf{coit}_F\ f\ s \not\rightsquigarrow\ \mathsf{in}_F\ (F\ (\mathsf{coit}_F\ f)\ (f\ s))\\
&\mathsf{unfold}_X\ p\ s \not\rightsquigarrow \begin{cases}\mathsf{nil}_X & \text{if } p\ s \rightsquigarrow^* \mathsf{inl}\ ()\\ \mathsf{cons}_X\ x\ (\mathsf{unfold}_X\ p\ s') & \text{if } p\ s \rightsquigarrow^* \mathsf{inr}\ (x, s')\end{cases}
\end{aligned}$$

preventing infinite regress. *Ipso facto*, the corresponding judgmental equations do not hold: the reducts shown in boxes above conspicuously fail to share the types of the redices from which they spring. Let us check:

$$\begin{array}{ccc}
\mathsf{case}_F\ P\ p\ (\mathsf{coit}_F\ f\ s) & \rightsquigarrow & p\ (F\ (\mathsf{coit}_F\ f)\ (f\ s))\\
: & & :\\
P\ (\mathsf{coit}_F\ f\ s) & \not\equiv & P\ (\mathsf{in}_F\ (F\ (\mathsf{coit}_F\ f)\ (f\ s)))
\end{array}$$

Preservation of typing fails precisely because we do not dare compute with laws which we should prefer somehow to hold.

Where does this problem bite? In fact, we can construct a bogus *proof* of exactly the missing equation, via the standard intensional propositional equality:

$$\begin{array}{ll}
-{=}- & : X \to X \to \mathsf{Prop}\\
\mathsf{refl} & : (x{:}X) \to x = x
\end{array}$$

According to the types of our components, we may check that

$$\begin{array}{l}
\mathsf{case}_F\ (\lambda x \mapsto x = \mathsf{in}_F\ (F\ (\mathsf{coit}_F\ f)\ (f\ s)))\\
\qquad\quad (\lambda t \mapsto \mathsf{refl}\ (\mathsf{in}_F\ t))\\
\qquad\quad (\mathsf{coit}_F\ f\ s)\\
: \mathsf{coit}_F\ f\ s = \mathsf{in}_F\ (F\ (\mathsf{coit}_F\ f)\ (f\ s))
\end{array}$$

but this proof computes in one step to

$$\mathsf{refl}\ (\mathsf{in}_F\ (F\ (\mathsf{coit}_F\ f)\ (f\ s)))\ \not:\ \mathsf{coit}_F\ f\ s = \mathsf{in}_F\ (F\ (\mathsf{coit}_F\ f)\ (f\ s))$$

Indeed, this propositional equality has the canonicity property, so in the empty context, all proofs must compute to some $\mathsf{refl}\ t$, making propositional equality correspond exactly to judgmental equality and thus *intensional*. Indeed, this intensionality allows us to define predicates which invalidate dependent case analysis for codata — there are predicates which hold for coconstructed codata but do not hold universally. It should not be possible to prove that a coiteration is intensionally equal to the corresponding coconstruction, just as one cannot prove intensional equality of these implementations of the identity on $\mathbb{N}$:

$$\lambda n \mapsto n \qquad\qquad \begin{array}{ll}\lambda\ 0 & \mapsto 0\\ \mid n+1 & \mapsto n+1\end{array}$$

As so often in type theory, the problem is the struggle for equality.

3 Type Equality versus Reduction

The trade in labour between the judgments for equality and type inhabitation is a crucial aspect of intensional type theories. We expect $\Gamma \vdash - : T$ to be checkable only for specific candidate terms — inhabitation requires evidence. However, we keep $\Gamma \vdash s \equiv t : T$ decidable, and we take advantage by admitting the rule

$$\frac{\Gamma \vdash s : S \quad \Gamma \vdash S \equiv T : \mathsf{Set}}{\Gamma \vdash s : T}$$

If T is a complex type which simplifies mechanically to some trivial type S, then a trivial inhabitant will suffice. As we are free to program in types, we are in a position to shift work from humans to machines, making developments like Gonthier's proof of the Four Colour Theorem tractable [9].

Within the bounds of decidability, and hopefully also good taste, we are free to question whether $\equiv$ should be exactly the equivalence and congruence closure of $\leadsto$, or whether it might be a little more generous. With care, a *type-directed* test for $\equiv$ can be made to admit rules like

$$\frac{\Gamma; x\!:\!S \vdash f\ x \equiv g\ x : T[x]}{\Gamma \vdash f \equiv g : (x\!:\!S) \to T} \qquad \frac{\Gamma \vdash u : 1 \quad \Gamma \vdash v : 1}{\Gamma \vdash u \equiv v : 1}$$

which are rather more difficult to capture by rewriting alone.

We now face a temptation. Even though coit_F must expand only on demand,

$$\mathsf{coit}_f\ f\ s \not\leadsto \mathsf{in}_F\ (F\ (\mathsf{coit}_F\ f)\ (f\ s))$$

might we not consider an equality test to be such a demand? When testing an equation, we could unfold coiteration on one side while we have coconstructors on the other, effectively adding the rule

$$\frac{\Gamma \vdash F\ (\mathsf{coit}_F\ f)\ (f\ s) \equiv t : F\ (\nu F)}{\Gamma \vdash \mathsf{coit}_F\ f\ s \equiv \mathsf{in}_F\ t : \nu F}$$

We can safely delay such unfoldings until we are comparing normal forms, as computation will unfold coiteration in redices anyway. Following this strategy, we arrive at a terminating test which is sound with respect to our extended $\equiv$ and which fixes our type error. It looks like we have won!

I used to advocate this approach, but I record it now only to spare others the embarrassment of falling for its charms. No such test can be complete if $\equiv$ is transitive. To see this, consider an automaton (a Turing machine perhaps) specified by a transition function of type $a : S \to 1 + S$ indicating whether the automaton halts or evolves to a new state. We may construct a coalgebra $\mathsf{tick}\ a$ for bitstreams in $\nu(2\times)$ with carrier $1 + S$ as follows

$$\begin{array}{lll}
\mathsf{tick}\ a & : & 1{+}S \quad \to 2 \times (1{+}S) \\
\mathsf{tick}\ a & \mapsto & \lambda\ (\mathsf{inl}\ ())\ \mapsto (\mathsf{true}, \mathsf{inl}\ ()) \\
 & & \mid\ (\mathsf{inr}\ s)\ \mapsto (\mathsf{true}, a\ s)
\end{array}$$

Now ask whether an automaton which has not yet halted ticks the same as one which has halted already:

$$\mathsf{coit}_{2\times}\ (\mathsf{tick}\ a)\ (\mathsf{inr}\ s)\ \equiv\ \mathsf{coit}_{2\times}\ (\mathsf{tick}\ a)\ (\mathsf{inl}\ ())\qquad ?$$

If the former ever halts, starting from s, then it is testably equal to a sufficiently large unfolding of form

$$\mathsf{in}_{2\times}\ (\mathsf{true}, \mathsf{in}_{2\times}\ (\mathsf{true}, \ldots \mathsf{in}_{2\times}\ (\mathsf{true}, \mathsf{coit}_{2\times}\ (\mathsf{tick}\ a)\ (\mathsf{inl}\ ()))\ldots))$$

which is clearly also an unfolding of the latter. By transitivity, a complete decision procedure for $\equiv$ must find the intermediate value for itself, effectively solving the halting problem. Hence it does not exist. We cannot hope to repair Coq's problem by augmenting the judgmental equality.

4 Weakly Final Coalgebras: a Principled Retreat

Inasmuch as dependent case analysis is not really justified by the intensional behaviour of codata — an uncomputed coiteration is not given by a coconstructor, even if there is no means to observe the difference within the theory — the principled thing to do is to throw out dependent case analysis, retaining only the nondependent observation.

We arrive at the following cut-down presentation, long advocated by Anton Setzer as closer to the destructor-based, coalgebraic intutition behind coinductive sets, and corresponding quite closely to the position implemented in Agda [15, 19, 19, 20]. For suitable strictly positive functors F, we obtain

$$\begin{aligned} \nu F &: \mathsf{Set} \\ \mathsf{out}_F &: \nu F \to F\ (\nu F) \\ \mathsf{coit}_F &: \forall S. (S \to F\ S) \to S \to \nu F \end{aligned}$$

$$\mathsf{out}_F\ (\mathsf{coit}_F\ f\ s)\ \leadsto\ F\ (\mathsf{coit}_F\ f)\ (f\ s)$$

We can restore something like the coconstructor, taking $\mathsf{in}'_F \mapsto \mathsf{coit}_F\ (F\,\mathsf{out}_F)$ for which computation and definitional expansion give us, judgmentally, that $\mathsf{out}_F\ (\mathsf{in}'_F\ t)\ \equiv\ F\,\mathsf{in}'_F\ (F\,\mathsf{out}_F\ t)$ which is perhaps less than desirable, but at least believable. Pragmatically, Agda adds in_F directly, with the more immediate decomposition behaviour

$$\mathsf{out}_F\ (\mathsf{in}_F\ t)\ \leadsto\ t$$

but this does not mitigate the main problem with this presentation: out_F is only a *weakly* final coalgebra. We are guaranteed that $\mathsf{coit}_F\ f$ is a solution of

$$\mathsf{out}_F\ (g\ s) = F\ g\ (f\ s)$$

but it is not *unique*. Consider $g \mapsto \mathsf{in}'_F \cdot F\ (\mathsf{coit}_F\ f) \cdot f$ and observe that, even with only the pointwise propositional reasoning that intensional type theories admit,

$$\begin{array}{ll}
\quad \mathsf{out}_F(g\ s) & \\
= & \{\text{definition}\} \\
\quad \mathsf{out}_F((\mathsf{in}'_F \cdot F\ (\mathsf{coit}_F\ f) \cdot f)\ s) & \\
= & \{\text{applied composition}\} \\
\quad \mathsf{out}_F\ (\mathsf{in}'_F\ (F\ (\mathsf{coit}_F\ f)\ (f\ s))) & \\
= & \{\text{computation on demand}\} \\
\quad F\ \mathsf{in}'_F\ (F\ \mathsf{out}_F\ (F\ (\mathsf{coit}_F\ f)\ (f\ s))) & \\
= & \{F\text{ functorial}\} \\
\quad F\ (\mathsf{in}'_F \cdot \mathsf{out}_F \cdot \mathsf{coit}_F\ f)\ (f\ s) & \\
= & \{\text{computation on demand}\} \\
\quad F\ (\mathsf{in}'_F \cdot F\ (\mathsf{coit}_F\ f) \cdot f)\ (f\ s) & \\
= & \{\text{definition}\} \\
\quad F\ g\ (f\ s) &
\end{array}$$

However, we cannot prove that $g = \mathsf{coit}_F\ f$, or even that they are pointwise equal. The best we can do is to define an *ad hoc* notion of bisimulation and show that both functions are pointwise bisimilar. We cannot treat bisimulation as a congruence and presume that all predicates respect it for the very good reason that they do not — the intensional equality is again a key counterexample.

This approach is in keeping with the standard properties and difficulties of intensional type theories. On the one hand, we have restored the fact that computation preserves type. On the other hand, we are forced to reason at an intensional level, just as we already are with *functions*. Functions, too, are equipped with only non-dependent elimination behaviour: application allows us to use functions but not to inspect or standardize their construction. Just as we are free to reason dependently about the result of a function application, so we may reason about the result of out_F by whatever means F supports, but the construction of elements of νF remains inscrutable.

5 O for an Observational Propositional Equality!

Earlier, we noted that an unjustified dependent case analysis principle gave rise to a bogus intensional equation between codata which were merely bisimilar. As it happens, we turn that fact on its head and derive dependent case analysis from a bisimulation.

Let us start from the safe, non-dependent presentation of codata given in the previous section. Suppose we could find a notion of equality, $\sim$, say, reflexive and substitutive at each set X

$$\begin{array}{rl}
\mathsf{refl}_X & : (x\!:\!X) \to x \sim_X x \\
\mathsf{subst}_X & : \forall x, y.\ x \sim_X y \to (P\!:\!X \to \mathsf{Set}) \to P\ x \to P\ y
\end{array}$$

for which we were able to show

$$\mathsf{io}_F\ \ :\ \ (x\!:\!\nu F) \to \mathsf{in}_F\ (\mathsf{out}_F\ x) \sim_{\nu F} x$$

We might then construct a general dependent case analysis operator from the destructor out_F.

$$\begin{array}{l} \mathsf{case}_F \ : \ (P : \nu F \to \mathsf{Set}) \ \to \ ((t : F\,(\nu F)) \to P\,(\mathsf{in}_F\,t)) \ \to \ (x : \nu F) \to P\,x \\ \mathsf{case}_F\ P\,p\,x \ \mapsto \ \mathsf{subst}_{\nu F}\,(\mathsf{in}_F\,(\mathsf{out}_F\,x))\,x\,(\mathsf{io}_F\,x)\,P\,(p\,(\mathsf{out}_F\,x)) \end{array}$$

If, moreover, we are lucky enough to have that for any $q : x \sim_X x$

$$\mathsf{subst}_X\ q\,P\,p \ \equiv \ p$$

then computing out_F gives us that the following hold judgmentally:

$$\begin{array}{ll} \mathsf{case}_F\ P\,p\,(\mathsf{in}_F\ t) & \equiv p\,t \\ \mathsf{case}_F\ P\,p\,(\mathsf{coit}_F\ f\ s) & \equiv \mathsf{subst}_{\nu F}\,(\mathsf{io}_F\,(\mathsf{coit}_F\ f\ s))\,P\,(p\,(F\,(\mathsf{coit}_F\ f)\,(f\ s))) \end{array}$$

These equations correspond almost exactly to the computation rules in the Coq presentation, but the type error in the coiteration case is repaired by explicit appeal to $\mathsf{subst}_{\nu F}$. If only we had an *observational* propositional equality, substitutive for bisimulation, then we should recover the uniqueness of $\mathsf{coit}_F\ f$ propositionally, although it cannot be unique with respect to the judgmental equality $\equiv$. But is this a tall order? I claim not.

6 Observational Equality

Let me sketch how to construct a propositional equality with the desired properties, extending earlier joint work with Altenkirch and Swierstra [2]. The machinery is quite intricate, but most of the choices are forced by the setup.

Start from a type theory equipped at least with 0, 1, dependent function types $(x : S) \to T$, and dependent pair types $(x : S) \times T$, but perhaps including sums and inductive types. Now introduce a propositional subuniverse of Set, with a 'decoder' mapping each proposition to the set of its proofs.

$$\mathsf{Prop} \ : \ \mathsf{Set} \qquad [\![-]\!] \ : \ \mathsf{Prop} \to \mathsf{Set}$$

closed under falsity, truth, conjunction, and universal quantification over sets.

$$\begin{array}{ll} [\![\bot]\!] & \rightsquigarrow 0 \\ [\![\top]\!] & \rightsquigarrow 1 \\ [\![P \wedge Q]\!] & \rightsquigarrow [\![P]\!] \times [\![Q]\!] \\ [\![\forall x : S.\ P]\!] & \rightsquigarrow (x : S) \to [\![P]\!] \end{array}$$

We may define implication, $P \Rightarrow Q$, as a degenerate universal quantification, $\forall \_ : [\![P]\!].\ Q$, recovering $[\![P \Rightarrow Q]\!] \rightsquigarrow [\![P]\!] \to [\![Q]\!]$.

Next, we may introduce equality for sets and values, not as inductive definitions, but rather computing *recursively* over the structure of types and values.

$$\frac{S, T \ : \ \mathsf{Set}}{S \leftrightarrow T \ : \ \mathsf{Prop}} \qquad \frac{s \ : \ S \quad t \ : \ T}{(s : S) \simeq (t : T) \ : \ \mathsf{Prop}}$$

Intuitively, $S \leftrightarrow T$ means that there are *structural* coercions between S and T. The heterogeneous value equality is a little more subtle: $(s:S) \simeq (t:T)$ indicates that s and t are interchangeable *if* $S \leftrightarrow T$. The intended elimination behaviour helps to motivate the construction: every equation on sets induces a *coherent coercion* between them

$$\begin{array}{l}
\mathsf{coe} : (S, T : \mathsf{Set}) \to S \leftrightarrow T \to S \to T \\
\mathsf{coh} : (S, T : \mathsf{Set}) \to (Q : S \leftrightarrow T) \to (s : S) \to (s : S) \simeq (\mathsf{coe}\ S\ T\ Q\ s : T)
\end{array}$$

We can transport values between equal sets, and the result remains equal, even though its type may have changed! Correspondingly, set equality must be constructed to ensure that coe can be computed. Meanwhile, value equality should be constructed consistently to reflect the construction of data and the observations on codata.

Set equality follows the structure of sets, yielding $\bot$ for distinct set-formers, and componentwise equations when the set-formers coincide.

$$\begin{array}{rcll}
0 & \leftrightarrow & 0 & \rightsquigarrow \top \\
1 & \leftrightarrow & 1 & \rightsquigarrow \top \\
(x:S) \to T & \leftrightarrow & (x':S') \to T' & \rightsquigarrow S' \leftrightarrow S \ \wedge\ \forall x':S'.\ \forall x:S.\ (x':S') \simeq (x:S) \Rightarrow T \leftrightarrow T' \\
(x:S) \times T & \leftrightarrow & (x':S') \times T' & \rightsquigarrow S \leftrightarrow S' \ \wedge\ \forall x:S.\ \forall x':S'.\ (x:S) \simeq (x':S') \Rightarrow T \leftrightarrow T' \\
 & \vdots & & \vdots
\end{array}$$

The orientation of the equality for the domain component of function and pair types reflects their contra- and co-variance, respectively. The definition of coe follows the same structure, exploiting 0-elimination when the types do not match, and componentwise coercion in the appropriate direction when they do.

$$\begin{array}{llllll}
\mathsf{coe} & 0 & 0 & Q\ x & \rightsquigarrow & x \\
\mathsf{coe} & 1 & 1 & Q\ x & \rightsquigarrow & x \\
\mathsf{coe} & ((x:S) \to T) & ((x':S') \to T') & Q\ f & \rightsquigarrow & \lambda x'.\ \mathrm{let}\ x \mapsto \mathsf{coe}\ S'\ S\ (\mathsf{fst}\ Q)\ x' \\
 & & & & & \qquad\quad q \mapsto \mathsf{coh}\ S'\ S\ (\mathsf{fst}\ Q)\ x' \\
 & & & & & \qquad\quad Q' \mapsto \mathsf{snd}\ Q\ x'\ x\ q \\
 & & & & & \qquad \mathrm{in}\ \ \mathsf{coe}\ T\ T'\ Q'\ (f\ x) \\
\mathsf{coe} & ((x:S) \times T) & ((x':S') \times T') & Q\ p & \rightsquigarrow & \mathrm{let}\ x \mapsto \mathsf{fst}\ p \\
 & & & & & \quad\ x' \mapsto \mathsf{coe}\ S\ S'\ (\mathsf{fst}\ Q)\ x \\
 & & & & & \quad\ q \mapsto \mathsf{coh}\ S\ S'\ (\mathsf{fst}\ Q)\ x \\
 & & & & & \quad\ Q' \mapsto \mathsf{snd}\ Q\ x\ x'\ q \\
 & & & & & \mathrm{in}\ \ (x', \mathsf{coe}\ T\ T'\ Q'\ (\mathsf{snd}\ p)) \\
\vdots & & & & \vdots &
\end{array}$$

Note how the coherence guarantee from the domain coercion is always exactly what we need to establish the equality of the codomains. To extend coercion to datatypes, exploit that fact that provably equal datatypes have the same constructors and provably equal component types to build a function recursively mapping each constructor to itself.

Crucially, coe computes *lazily* in the proof Q unless the equation is conspicuously absurd. It is this laziness which buys us our freedom to design the value equality observationally: as long as we ensure that $[\![\bot]\!]$ remains uninhabited, coe must always make progress transporting canonical values between canonical types. Moreover, we are free to add whatever propositional axioms we like, provided we retain consistency. Correspondingly, we may take coh as axiomatic, along with

$$\begin{array}{ll}\mathsf{refl} & : \forall X.\, [\![\forall x\!:\!X.\, (x\!:\!X) \simeq (x\!:\!X)]\!] \\ \mathsf{Resp} & : \forall X.\, (P\!:\!X \to \mathsf{Set}) \to [\![\forall x\!:\!X.\, \forall y\!:\!X.\, (x\!:\!X) \simeq (y\!:\!X) \Rightarrow P\,x \leftrightarrow P\,y]\!]\end{array}$$

The latter allows us to recover subst from coe.

We can take value equality to be anything we like for types with distinct constructors — it is meaningful only when the types match. For like types, we must be more careful:

$$\begin{array}{lcll}(x{:}0) & \simeq & (y{:}0) & \leadsto \top \\ (x{:}1) & \simeq & (y{:}1) & \leadsto \top \\ (f{:}(x\!:\!S) \to T) & \simeq & (f'{:}(x'\!:\!S') \to T') & \leadsto \forall x\!:\!S.\, \forall x'\!:\!S'. \\ & & & \quad (x\!:\!S) \simeq (x'\!:\!S') \Rightarrow (f\;x\!:\!T) \simeq (f'\;x'\!:\!T') \\ (p{:}(x\!:\!S) \times T) & \simeq & (p'{:}(x'\!:\!S') \times T') & \leadsto \text{let } x \mapsto \mathsf{fst}\,p \;\;;\;\; y \mapsto \mathsf{snd}\,p \\ & & & \qquad\; x' \mapsto \mathsf{fst}\,p' \;;\; y' \mapsto \mathsf{snd}\,p' \\ & & & \quad \text{in } (x\!:\!S) \simeq (x'\!:\!S') \wedge (y\!:\!T) \simeq (y'\!:\!T')\end{array}$$

As you can see, equality for pairs is componentwise and equality for functions is extensional. The coup de grace is that with some care, we can ensure that

$$\mathsf{coe}\, X\; X\; Q\; x \;\equiv\; x$$

holds, not by adding a nonlinear reduction rule, but just by extending the way in which normal forms are compared after evaluation — a detailed treatment is given in our paper [2]. This summarizes the basic machinery for observational equality. Let us now add codata, and show that we may indeed take the $x \sim_X y$ we seek to be the homogeneous special case of our value equality $(x\!:\!X) \simeq (y\!:\!X)$.

7 Interaction Structures Closed Under Bisimulation

Peter Hancock and Anton Setzer define a notion of *interaction structure* corresponding to potentially infinite trees of traces in a state-dependent command-response dialogue [11, 12]. We may introduce a new type constructor for such structures in general, parametrized by a protocol as follows:

$$\begin{array}{ll}\mathsf{IO}\;(S : \mathsf{Set}) & \text{states of the system} \\ \quad (C : S \to \mathsf{Set}) & \text{commands for each state} \\ \quad (R : (s\!:\!S) \to C\;s \to \mathsf{Set}) & \text{responses for each command} \\ \quad (n\; : (s\!:\!S) \to (c\!:\!C\;s) \to R\;s\;c \to S) & \text{new state from each response} \\ \quad\; : \; S \to \mathsf{Set} & \text{traces from each state}\end{array}$$

Interaction structures are the coinductive counterpart of Petersson-Synek trees [17]. The parameters C, R, and n characterize the strictly positive endofunctors on the category $S \to \mathsf{Set}$ of S-indexed sets with index-preserving maps. These are exactly the *indexed containers* [1, 3]

$$\begin{array}{l}[S,C,R,n] \ : \ (S \to \mathsf{Set}) \to (S \to \mathsf{Set}) \\ [S,C,R,n]\, X\ s \ \mapsto \ (c{:}C\ s) \times (r{:}R\, c\, s) \to X\ (n\ s\ c\ r)\end{array}$$

with action on morphisms defined by

$$[S,C,R,n]\, f \ \mapsto \ \lambda s.\, \lambda u.\, (\mathsf{fst}\, u, f\ s \cdot \mathsf{snd}\ u)$$

Now, $(\mathsf{IO}\ S\ C\ R\, n, \mathsf{out})$ is the terminal $[S,C,r,n]$-coalgebra. That is, we take

$$\begin{array}{ll}\mathsf{out}_{S,C,R,n} & : \ (s{:}S) \to \mathsf{IO}\ S\ C\ R\, n\ s \to [S,C,R,n]\ (\mathsf{IO}\ S\ C\ R\, n)\ s \\ \mathsf{coit}_{S,C,R,n} & : \ \forall X.\, ((s{:}S) \to X\ s \to [S,C,R,n]\ X\ s) \to \\ & \qquad ((s{:}S) \to X\ s \to \mathsf{IO}\ S\ C\ R\, n\ s) \\ \mathsf{out}_{S,C,R,n}\ (\mathsf{coit}_{S,C,R,n}\ f\ s\ x) & \rightsquigarrow \ [S,C,R,n]\ f\ s\ (f\ s\ x)\end{array}$$

and we may add

$$\begin{array}{ll}\mathsf{in}_{S,C,R,n} & : \ (s{:}S) \to [S,C,R,n]\ (\mathsf{IO}\ S\ C\ R\, n)\ s \to \mathsf{IO}\ S\ C\ R\, n\ s \\ \mathsf{out}_{S,C,R,n}\ (\mathsf{in}_{S,C,R,n}\ s\, t) & \rightsquigarrow \ t\end{array}$$

Markus Michelbrink and Anton Setzer have shown that $[S,C,R,n]$ has a final coalgebra [15]. For our purposes, however, we must now choose answers to three additional questions:

1. *When are two* IO *sets provably equal?* When their parameters are provably pointwise equal.
2. *How can we transport codata between provably equal* IO *structures?* By a 'device driver' coiteration which translates commands one way and responses the other, exploiting the equality of command and response sets.
3. *When are* IO *values provably equal?* When they are bisimilar.

Fleshing out the first two answers is largely mechanical, although some care must be taken with the precise computational details. For the third, we need to ensure that our universe Prop is capable of expressing bisimulations. Correspondingly, let us extend Prop with coinductive *predicates*, generally.

$\mathsf{HAP}\ (S\ : \mathsf{Set})$	states of the system
$(H : S \to \mathsf{Prop})$	happiness in each state
$(R : (s{:}S) \to [\![H\ s]\!] \to \mathsf{Set})$	responses to happiness
$(n\ : (s{:}S) \to (h{:}[\![H\ s]\!]) \to R\ s\ h \to S)$	new state from each response
$: \ S \to \mathsf{Prop}$	happiness from each state on

The predicate $\mathsf{HAP}\, S\, H\, R\, n$ is parametrized by a predicate H which indicates happiness with a current state drawn from S and a pair R, n indicating how states may evolve when we are happy: $\mathsf{HAP}\,S\,H\,R\,n\,s$ then holds for any state s whenceforth eternal happiness is assured. It is easy to interpret these propositions as coinductive sets of proofs:

$$[\![\mathsf{HAP}\,S\,H\,R\,n\,s]\!] \;\leadsto\; \mathsf{IO}\,S\,(\lambda s.\,[\![H\,s]\!])\,R\,n\,s$$

We may now define value equality for IO processes as a kind of happiness corresponding to bisimilarity — I precede each component with its motivation:

```
$(p{:}\mathsf{IO}\,S\,C\,R\,n\,s) \simeq (p'{:}\mathsf{IO}\,S'\,C'\,R'\,n'\,s') \;\leadsto$
  $\mathsf{HAP}$      — we consider two processes, each in its own state
      $(((s{:}S) \times \mathsf{IO}\,S\,C\,R\,n\,s) \times ((s'{:}S') \times \mathsf{IO}\,S'\,C'\,R'\,n'\,s'))$
         — we are happy *now* if they issue equal commands
      $(\lambda((s,p),(s',p')).$
         let $c \mapsto \mathsf{fst}\,(\mathsf{out}\,p)$ ;  $c' \mapsto \mathsf{fst}\,(\mathsf{out}\,p')$
         in  $(c{:}C\,s) \simeq (c'{:}C'\,s'))$
         — we need to stay happy whenever they receive equal responses
      $(\lambda((s,p),(s',p')).\,\lambda\_.$
         let $c \mapsto \mathsf{fst}\,(\mathsf{out}\,p)$ ;  $c' \mapsto \mathsf{fst}\,(\mathsf{out}\,p')$
         in  $(r{:}R\,s\,c) \times (r'{:}R'\,s'\,c') \times [\![(r{:}R\,s\,c) \simeq (r'{:}R'\,s'\,c')]\!])$
         — each process evolves according to its own protocol
      $(\lambda((s,p),(s',p')).\,\lambda\_.\,\lambda(r,r',\_).$
         let $c \mapsto \mathsf{fst}\,(\mathsf{out}\,p)$ ;  $c' \mapsto \mathsf{fst}\,(\mathsf{out}\,p')$
             $u \mapsto \mathsf{snd}\,(\mathsf{out}\,p)$ ;  $u' \mapsto \mathsf{snd}\,(\mathsf{out}\,p')$
         in  $((n\,s\,c\,r, u\,r), (n'\,s'\,c'\,r', u'\,r')))$
         — and we start from the processes at hand
      $((s,p),(s',p'))$
```

We may now establish that

$$(\mathsf{in}_{S,C,R,n}\,s\,(\mathsf{out}_{S,C,R,n}\,s\,p) : \mathsf{IO}\,S\,C\,R\,n\,s) \simeq (p : \mathsf{IO}\,S\,C\,R\,n\,s)$$

by a coiterative proof of bisimilarity. We may similarly establish that $\mathsf{coit}_{S,C,R,n}\,f$ is the unique solution of its defining equation, up to $\simeq$, again by coiteration. By generalising our treatment of codata to the indexed case, we effectively *closed* coinductive types under observational equality, re-using the same IO structures with a suitably enriched notion of state. This pleasing uniformity is rather to be expected when starting from such an expressive class of definitions.

8 Epilogue

If we want to reason about coprograms in intuitive way, we need a substitutive equality which includes bisimilarity. Equivalently, we need a dependent case analysis to see how things unfold. We have seen why the latter leads to the failure of type preservation in Coq and that there is no chance to fix this problem by strengthening the judgmental equality. We have also seen how to construct an observational propositional equality which amounts to extensionality for functions and bisimilarity for coinductive values, giving rise to a structural coercion between equal sets. By explicit appeal to this observational equality, we can restore the type safety of dependent case analysis for codata.

As proof of concept, I have constructed in AGDA a small universe of sets and propositions containing interaction structures and observational equality, equipped with coercion as described above. This development will inform a new treatment of codata in the next version of EPIGRAM, corresponding directly to the categorical notion of terminal coalgebra, and with bisimulation substutive in all contexts.

A curious consequence of this choice is that predicates of intensional stamp, like inductively defined equality, must be ruthlessly excised from the type theory. If a predicate holds for one implementation but not another, it can hardly respect observational equality. As we reason, so we must define — up to observation. It as almost as if a greater fidelity to abstract, extensional mathematics is being forced upon us, whether we like it or not. Not only *can* we express coalgebraic and algebraic structure precisely in dependent type theories, but increasingly, we *must*. 'Category theory working, for the programmer' is becoming a viable prospect and a vital prospectus.

Acknowledgements

I should like to thank Nicolas Oury for showing me this problem, Alexander Kurz for his patience with my laziness, and Peter Hancock for provocation.

References

1. Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.
2. Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In Aaron Stump and Hongwei Xi, editors, *PLPV*, pages 57–68. ACM, 2007.
3. Thorsten Altenkirch and Peter Morris. Indexed Containers. In *Proceedings of LICS*, 2009.
4. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving And Program Development: Coq'Art: the Calculus of Inductive Constructions*. Springer, 2004.
5. Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, The University of Calgary, June 1992.
6. Robin Cockett and Dwight Spencer. Strong categorical datatypes I. In R. A. G. Seely, editor, *International Meeting on Category Theory 1991*, Canadian Mathematical Society Proceedings. AMS, 1992.
7. Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan M. Smith, editors, *TYPES*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 1994.

8. Eduardo Giménez. *Un Calcul de Constructions Infinies et son application à la vérification de systèmes communicants.* PhD thesis, Ecole Normale Supérieure de Lyon, 1996.
9. Georges Gonthier. A computer-checked proof of the Four-Colour theorem. Technical report, Microsoft Research, 2005.
10. Tatsuya Hagino. *A Categorical Programming Language.* PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1987.
11. Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In Peter Clote and Helmut Schwichtenberg, editors, *CSL*, volume 1862 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2000.
12. Peter Hancock and Anton Setzer. Interactive programs and weakly final coalgebras in dependent type theory. In L. Crosilla and P. Schuster, editors, *From Sets and Types to Topology and Analysis. Towards Practicable Foundations for Constructive Mathematics*, pages 115 – 134, Oxford, 2005. Clarendon Press.
13. Martin Hofmann. *Extensional concepts in intensional type theory.* PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1995. Available from http://www.lfcs.informatics.ed.ac.uk/reports/95/ECS-LFCS-95-327/.
14. Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.
15. Markus Michelbrink and Anton Setzer. State dependent IO-monads in type theory. *Electronic Notes in Theoretical Computer Science*, 122:127 – 146, 2005.
16. Ulf Norell. *Towards a Practical Programming Language based on Dependent Type Theory.* PhD thesis, Chalmers University of Technology, 2007.
17. Kent Petersson and Dan Synek. A set constructor for inductive sets in martin-löf's type theory. In David H. Pitt, David E. Rydeheard, Peter Dybjer, Andrew M. Pitts, and Axel Poigné, editors, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 128–140. Springer, 1989.
18. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.
19. Anton Setzer. Guarded recursion in dependent type theory. Talk at Agda Implementors' Meeting 6, Göteborg, Sweden, May 2007.
20. Anton Setzer. Coalgebras in dependent type theory. Talk at Agda Intensive Meeting 9, Sendai, Japan, November 2008.
21. Alastair Telford and David Turner. Ensuring streams flow. In Michael Johnson, editor, *AMAST*, volume 1349 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1997.
22. D. A. Turner. Elementary strong functional programming. In Pieter H. Hartel and Marinus J. Plasmeijer, editors, *FPLE*, volume 1022 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1995.
23. D. A. Turner. Total functional programming. *J. UCS*, 10(7):751–768, 2004.