LET'S SEE HOW THINGS UNFOLD: STRONGLY FINAL COALGEBRAS IN DEPENDENT TYPE THEORY

CONOR MCBRIDE

Department of Computer and Information Sciences,University of Strathclyde
e-mail address: conor@cis.strath.ac.uk

Abstract. Coq and Agda, amongst the current crop of proof assistants based on Martin-Löf's intensional type theory, offer some support for coinductive definitions. Neither, however, gives a satisfactory account of reasoning about such definitions. Agda does not permit *dependent* case analysis of coinductive data; Coq does, at the drastic cost of losing subject reduction. At the heart of the problem is equality: in this paper, I show that dependent case analysis for coinductive data is incompatible with the inductively defined equality traditionally supported by intensional type theories, and I offer an alternative, based on the observational equality of Altenkirch et al., presented by means of a universe construction in Agda.

1. Introduction

Coinductive types model infinite structures unfolded on demand, like evasive politicians: they may never convince, but they always give an answer. Representing such evolving processes or 'survival strategies' 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 types and *coprograms* in our toolbox, equipped with appropriate computational and logical machinery. As mathematicians, we may characterize coinductive types as final coalgebras and coprograms as generated by the unique map from any particular coalgebra (a strategy for handling one demand), but mechanizing our blackboard methods brings a troubled negotiation with the bounded possibilities of computation. This paper analyses and advances the art of the possible.

Lazy functional languages like Haskell [Pey03] 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 [Hag87]. The corresponding discipline of 'coprogramming' is given expression in

1998 ACM Subject Classification: MANDATORY list of acm classifications.
Key words and phrases: MANDATORY list of keywords.
An extended abstract of this article was published in the proceedings of CALCO 2009.

LOGICAL METHODS
IN COMPUTER SCIENCE
DOI:10.2168/LMCS-???

Cockett's work on Charity [CF92, CS92] and in the work of Turner and Telford on 'Elementary Strong Functional Programming' [Tur95, TT97, Tur04]. 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 one should not: a server is a coprogram in a language guaranteeing liveness, a strategy for surviving the caprice of clients.

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 [Nor07, BC04, MM04]. 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. It is a standard but troublesome practice to ensure that these notions *coincide* for closed terms, comparing the *construction* even of infinitary objects, where substitutability is more a question of *observation*.

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 judgmental equality to admit unfolding and remain decidable.

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 of the same type [Hof95]. That is just bad luck: in this paper, I define a small dependent type theory (sufficient for exploratory purposes) and equip it with just such a notion of equality, following earlier joint work with Altenkirch and Swierstra on observational equality for functions [AMS07]. In this setting, I prove that each coprogram is equal to its unfolding.

The key technical ingredient is the notion of 'interaction structure' due to Hancock and Setzer [HS00] — a generic treatment of indexed coinductive datatypes, which I show here to be *closed* under its own notion of bisimulation. A similar treatment has been implemented in the 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. A small dependent type theory: TT

If our problem is to equip dependent type theory with a workable notion of coinductive type, let us fix a type theory with which to work. We shall need dependent functions in Π-types, $(x : S) \to T$, and dependent tuples in Σ-types, $(x : S) \times T$; we shall need the empty ($\mathsf{0}$) and unit ($\mathsf{1}$) types, modelling absence and presence, and the Boolean type ($\mathsf{2}$) to represent distinctions. I present this type theory as a system of mutually inductive judgments.

Definition 2.1 (TT Judgments)**.** *Judgments in TT take the form* $\Gamma \vdash J$*, where* Γ *is a context assigning types to variables and* J *may take one of five forms, as shown below.*

$\Gamma \vdash \textsc{valid}$	Γ *is a valid context, giving types to variables*
$\Gamma \vdash T \textsc{ type}$	T *is a type in context* Γ
$\Gamma \vdash S \equiv T \textsc{ type}$	S *and* T *are equal types in context* Γ
$\Gamma \vdash t : T$	*term* t *has type* T *in context* Γ
$\Gamma \vdash s \equiv t : T$	s *and* t *are equal at type* T *in context* Γ

The system of inference rules will be formulated to ensure that the following well-formedness conditions always hold by induction on derivations.

Definition 2.2 (Well-formed judgments)**.**

$\vdash \textsc{valid}$	*is well-formed*
$\Gamma; x\!:\!S \vdash \textsc{valid}$	*is well-formed if* $x \notin \Gamma$
$\Gamma \vdash T \textsc{ type}$	*is well-formed if* $\Gamma \vdash \textsc{valid}$
$\Gamma \vdash S \equiv T \textsc{ type}$	*is well-formed if* $\Gamma \vdash S \textsc{ type} \wedge \Gamma \vdash T \textsc{ type}$
$\Gamma \vdash t : T$	*is well-formed if* $\Gamma \vdash T \textsc{ type}$
$\Gamma \vdash s \equiv t : T$	*is well-formed if* $\Gamma \vdash s : T \wedge \Gamma \vdash t : T$
$\Gamma; x\!:\!S; \Delta \vdash J$	*is well-formed if* $\Gamma \vdash s : S \Rightarrow \Gamma; \Delta[s/x] \vdash J[s/x]$

For the sake of readability, I shall suppress premises whose only purpose is to ensure these well-formedness conditions. I presume that α-conversion happens silently, ensuring that the side-condition on freshness of variable names is always satisfied.

Context validity is entirely standard: the empty context may be extended with type assignments.

Definition 2.3 ($\Gamma \vdash \textsc{valid}$)**.**

$$\frac{}{\vdash \textsc{valid}} \qquad \frac{\Gamma \vdash S \textsc{ type}}{\Gamma; x\!:\!S \vdash \textsc{valid}}$$

TT has a fixed repertoire of types: I write judgments with multiple subjects to abbreviate multiple rules with identical premises.

Definition 2.4 ($\Gamma \vdash T \textsc{ type}$)**.**

$$\frac{}{\Gamma \vdash \mathsf{0}, \mathsf{1}, \mathsf{2} \textsc{ type}} \qquad \frac{\Gamma, x\!:\!S \vdash T \textsc{ type}}{\Gamma \vdash (x\!:\!S) \to T, (x\!:\!S) \times T \textsc{ type}} \qquad \frac{\Gamma \vdash b : \mathsf{2} \quad \Gamma \vdash T, F \textsc{ type}}{\Gamma \vdash \mathsf{Cond}(b, T, F) \textsc{ type}}$$

We have *canonical* types as follows: finite types, $\mathsf{0}, \mathsf{1}, \mathsf{2}$, Π-types $(x\!:\!S) \to T$ generalising products to dependent function types, and Σ-types $(x : S) \times T$ generalising coproducts to dependent record types. It is common practice to extend this apparatus to a hierarchy of universes, each contained by and embedded in the next, but I have carefully cut off the bottom layer, in order to construct a model of it. Let us, however, support *large elimination*: $\mathsf{Cond}(b, T, F)$ computes a type T or F depending on whether Boolean value b is $\mathtt{tt}$ or $\mathtt{ff}$, respectively. Seen through the "propositions-as-types" lens, this admits predicates distinguishing $\mathtt{tt}$ from $\mathtt{ff}$, for example

$$\mathbb{T}(b) := \mathsf{Cond}(b, \mathsf{1}, \mathsf{0})$$

inhabited only when b is $\mathtt{tt}$. To achieve this, we need a computational notion of type equality.

Definition 2.5 ($\Gamma \vdash S \equiv T$ TYPE)**.**

$$\frac{}{\Gamma \vdash \mathsf{Cond}(\mathtt{tt}, T, F) \equiv T \text{ TYPE}} \qquad \frac{}{\Gamma \vdash \mathsf{Cond}(\mathtt{ff}, T, F) \equiv F \text{ TYPE}}$$

The remaining type equality rules express structural and equivalence closure.

The following relations capture similarities and difference between canonical types.

Definition 2.6 ($S \parallel T, S \perp T$)**.** *$S \parallel T$ whenever S and T are* canonical *types formed by the same rule. $S \perp T$ whenever S and T are* canonical *types formed by* distinct *rules.*

For example,

$$\mathsf{0} \parallel \mathsf{0} \qquad \mathsf{0} \perp \mathsf{1} \qquad (\mathsf{0} \to \mathsf{1}) \parallel (\mathsf{1} \to \mathsf{0}) \qquad (\mathsf{0} \to \mathsf{1}) \perp (\mathsf{1} \times \mathsf{0})$$

Clearly, if S and T are canonical, then either $S \parallel T$ or $S \perp T$, but not both. Whenever $S \perp T$, $\Gamma \not\vdash S \equiv T$ TYPE, irrespective of any wicked lies hypothesized in Γ. Judgmental type equality is *intensional*, identifying types only as far as computation permits, and no computation can ever identify canonical constructs. This restriction reflects the idea that type checking is a job for a computer; finding type inhabitants is human work. We provide the terms whose types are assigned as follows.

Definition 2.7 ($\Gamma \vdash t : T$)**.**

$$\frac{\Gamma; x{:}S; \Delta \vdash \text{VALID}}{\Gamma; x{:}S; \Delta \vdash x : S} \qquad \frac{\Gamma \vdash s : S \quad \Gamma \vdash S \equiv T \text{ TYPE}}{\Gamma \vdash s : T}$$

$$\frac{\Gamma \vdash z : \mathsf{0}}{\Gamma \vdash z\Psi S : S} \qquad \frac{}{\Gamma \vdash \langle\rangle : \mathsf{1}}$$

$$\frac{}{\Gamma \vdash \mathtt{tt}, \mathtt{ff} : \mathsf{2}} \qquad \frac{\Gamma \vdash b : \mathsf{2} \quad \Gamma; x{:}\mathsf{2} \vdash P \text{ TYPE} \quad \Gamma \vdash t : P[\mathtt{tt}] \quad \Gamma \vdash f : P[\mathtt{ff}]}{\Gamma \vdash \mathsf{cond}(b, x.\, P, t, f) : P[b]}$$

$$\frac{\Gamma; x{:}S \vdash t : T}{\Gamma \vdash \lambda_S x.\, t : (x{:}S) \to T} \qquad \frac{\Gamma \vdash f : (x{:}S) \to T \quad \Gamma \vdash s : S}{\Gamma \vdash f\ s \vdash T[s]}$$

$$\frac{\Gamma \vdash s : S \quad \Gamma; x{:}S \vdash T \text{ TYPE} \quad \Gamma \vdash t : T[s]}{\Gamma \vdash \langle s, t\rangle_{x.T} : (x{:}S) \times T} \qquad \frac{\Gamma \vdash p : (x{:}S) \times T}{\Gamma \vdash \pi_0 p : S} \qquad \frac{\Gamma \vdash p : (x{:}S) \times T}{\Gamma \vdash \pi_1 p : T[\pi_0 p]}$$

Let us consider these terms carefully. Firstly, we have variables, typed as indicated by the context, and we have that type inhabitation is *silently* invariant with respect to *judgmental* type equality. No signal that the type changes its syntactic form is needed, as long as judgmental equality is decidable. The more powerful the judgmental equality, the fewer trivial isomorphisms we need witness.

There is no canonical way to make an element of $\mathsf{0}$, but if you have one (hypothetically), you can make anything! The element of $\mathsf{1}$ is easy to find, but useless to possess.

There are two Boolean values, $\mathtt{tt}$ and $\mathtt{ff}$, which may be distinguished computationally by a *dependent case analysis principle*. We may certainly use this to define simple conditional functions

$$\mathbf{not} := \lambda b.\, \mathsf{cond}(b, \_.\, \mathsf{2}, \mathtt{ff}, \mathtt{tt}) : \mathsf{2} \to \mathsf{2}$$

but reassuringly, unlike the conventional 'if…then…else…' construct, the two branches are not necessarily interchangeable. The return type of cond is specified with respect to a bound variable (not by a function, as this would require large function spaces), differently

instantiated in each branch, reflecting what has been *learned* by testing b. For example, we may confirm that a true value cannot have a true negation.

$$\begin{aligned}\mathbf{notTrueAndFalse} := \lambda b.\, \mathsf{cond}(b, b.\, \mathbb{T}\mathsf{f}(b) \to \mathbb{T}\mathsf{f}(\mathbf{not}\, b) \to \mathsf{0}, \lambda\_.\, \lambda z.\, z, \lambda z.\, \lambda\_.\, z)\\ : (b{:}\mathsf{2}) \to \mathbb{T}\mathsf{f}(b) \to \mathbb{T}\mathsf{f}(\mathbf{not}\, b) \to \mathsf{0}\end{aligned}$$

Once again, the typing of the above relies on equality rules, expressing conditional compilation. These follow shortly.

Functions are constructed by abstraction and eliminated by application. I write the domain annotation as a subscript λ_S to indicate that it is necessary for type synthesis but not for type checking, and I shall omit it informally wherever types are known. We do not have *dependent* elimination, so we may not observe the *construction* of a function, only its *uses*. Morally, at least, functions are to be understood extensionally. In a similar way, we may not observe the construction of a pair, only its *projections*.

Definition 2.8 ($\Gamma \vdash s \equiv t : T$)**.** *The computation rules for TT terms are as follows:*

$$\overline{\Gamma \vdash \mathsf{cond}(\mathsf{tt}, x.\, P, t, f) \equiv t : P[\mathsf{tt}]} \qquad \overline{\Gamma \vdash \mathsf{cond}(\mathsf{ff}, x.\, P, t, f) \equiv f : P[\mathsf{ff}]}$$

$$\overline{\Gamma \vdash (\lambda_S x.\, t)\; s \equiv t[s] \vdash T[s]}$$

$$\overline{\Gamma \vdash \pi_0 \langle s, t\rangle_{x.T} \equiv s : S} \qquad \overline{\Gamma \vdash \pi_1 \langle s, t\rangle_{x.T} \equiv t : T[s]}$$

The judgmental equality for this type theory closes the computation rules under equivalence and structural congruence.

We are free to consider extending the equational theory of terms still further, adding η-laws to identify f with $\lambda x.f\, x$, for example. To do so is a convenience, rather than a necessity. For the moment, let us abstain from such considerations, but at least grant ourselves the *notational* convenience of writing 'pair patterns' in binding positions, with $\langle x, y\rangle\, .t$ meaning $z.t[\pi_0 z/x, \pi_1 z/x]$.

2.1. **Shallow embeddings and deeper encodings into larger type theories.** TT is in no way peculiar, except that (for exploratory purposes) it is far weaker than the type theories typically studied in the literature or implemented in proof assistants and dependently typed programming languages. We may readily construct its canonical types in Coq or Agda. Here are the Agda versions.

```
data Zero : Set where  -- no constructors!
record One : Set where  -- no fields!
data Two : Set where tt ff : Two
Π : (S : Set) → (S → Set) → Set  -- reuse Agda's function space!
Π S T = (x : S) → T x
record Σ (S : Set)(T : S → Set) : Set where  π₀ : S;  π₁ : T π₀
```

The Coq versions are similar. We may then implement our signature of type- and term-level operations, satisfying the computation rules above in the judgmental equality of the metalanguage—the machine is perfectly capable of deciding the equational theory those rules induce. Our type theory thus has easy shallow embeddings into, and serves as a convenient microcosm of, type theories as they are typically constituted today.

Moreover, we can give an inductive characterization of the sets in this type theory. This is most readily done by *induction-recursion* [DS99] in Agda, defining a syntax U for sets simultaneously with its decoding $[\![\_]\!]$ to Agda sets, effectively constructing a least fixpoint in $\Sigma_{X:\mathsf{Set}} X \to \mathsf{Set}$. Note that Agda uses the syntax $\lambda s \to t$ for abstractions.

```
mutual
  data U : Set where                          [[_]] : U → Set
    '0 '1 '2 : U                              [['0]] = Zero
    'Π 'Σ : (S : U) → ([[S]] → U) → U         [['1]] = One
                                              [['2]] = Two
                                              [['Π S T]] = Π [[S]] (λ s → [[T s]])
                                              [['Σ S T]] = Σ [[S]] (λ s → [[T s]])
```

Coq does not support induction-recursion, but we may deploy the ugly alternative of defining the large inductive 'predicate' in $\mathsf{Set} \to \mathsf{Type}$ bearing evidence that a given Coq set is representable in our theory.

Note that our little universe of types does not encode a type of types. Inductive-recursive presentations of universe *hierarchies* have been presented in the literature: they introduce problems which are interesting, but not relevant to our struggle with codata. The problems with codata will manifest themselves quite satisfactorily in our cut-down system. The solution I propose does not depend critically on working in the small.

2.2. **Adding Inductive Families—μTT.** Before we venture in seach of coinduction, let us first consider inductive data. I propose also to add a single but rather generic notion of indexed inductive datatype—the Petersson-Synek trees [PS89]—representing terms in a multi-sorted free algebra. Morally, these are the least fixpoints of strictly positive (or 'generalized polynomial') endofunctors of slice categories, $\textsc{type}/S$. However, intensional type theory responds unpleasantly to moralisation, so we shall need to pay some attention to the details.

To tease out the structure from the bureaucracy, it will help to consider the *S-indexed type families* as (α-equivalence classes of) types binding a variable of type S, validated by an auxiliary judgment form, as follows.

Definition 2.9 (Judgment $\Gamma \vdash A\ \textsc{type}[S]$, category $\textsc{type}[S]$)**.**

$$\frac{\Gamma; x : S \vdash T\ \textsc{type}}{\Gamma \vdash x.T\ \textsc{type}[S]}$$

We may instantiate any such family $A\ \textsc{type}[S]$ to some $A[s]\ \textsc{type}$ if $s : S$. In any context, the category $\textsc{type}[S]$ has objects A such that $A\ \textsc{type}[S]$ and morphisms $f : A \to\!\!\!\!\bullet\ B$ where

$$A \to\!\!\!\!\bullet\ B := (s : S) \to A[s] \to B[s]$$

with identity and composition defined pointwise.

Again, we should not think of these indexed families as *functions*: TT does not have a type of types. Where the objects of slice categories are underlying objects with indexing morphisms, these families come ready-sliced. The requirement that morphisms respect indexing is intrinsic, checked up to the *judgmental* equality of TT.

Note that, with awareness of families, we can tidy the presentation of dependendent function and record types, writing $\Sigma\, S\, T$ for $(x : S) \times T[x]$ and $\Pi\, S\, T$ for $(x : S) \to T[x]$, whenever it is convenient to do so.

We may present inductive datatypes as fixpoints of *indexed containers* or *generalized polynomial functors* on $\textsc{type}[S]$. Let us pack up the pieces in an auxiliary judgment form $\Gamma \vdash F\ \textsc{cont}[S]$ where

$$\frac{\Gamma; s\!:\!S; c\!:\!C; r\!:\!R \vdash n : S}{\Gamma \vdash s.(c\!:\!C \triangleleft r\!:\!R.\, n)\ \textsc{cont}[S]}$$

and interpret such Fs as sum-of-products functors on S-indexed types as follows (writing † for the action on objects and ‡ for the action on morphisms, as the usual silent overloading is not easy to mechanize)

$$s.(c\!:\!C \triangleleft r\!:\!R.\, n)^\dagger s.X = s.(c\!:\!C) \times (r\!:\!R) \to X[n]$$
$$s.(c\!:\!C \triangleleft r\!:\!R.\, n)^\ddagger g = \lambda s.\, \lambda\, \langle c, f\rangle\, .\, \langle c, \lambda r.\, g\, n\, (f\, r)\rangle$$

The idea is that S is the type of *sorts* in a given multi-sorted algebra. C is the type of *constructors* for sort s, R the type of *recursive positions* within terms formed at sort s by constructor c, and n the 'next' sort, accepted at position r within such a term. A node is thus specified by a pair $\langle c, f\rangle$, being a choice of constructor c and a function f from recursive positions to substructures. To reduce clutter, whenever schematic variables like S, C, R, n scope over free variables like s, c, r, I am careful either to instantiate or to capture the latter, for example writing C rather than $C[s]$ when C effectively abstracts s already.

Let us now add the inductive types $\mu_S F\ s$ to our type theory. The family $s.\mu_S F\ s$, abbreviated $\mu_S F$ or even μF when the index set is unambiguous, is the least fixpoint of F.

Definition 2.10 (μTT)**.** *The theory μTT extends TT with new canonical types $\mu_S F\ s$, governed by the following formation, introduction, elimination, and computation rules.*

$$\frac{\Gamma \vdash F\ \textsc{cont}[S] \quad s : S}{\Gamma \vdash \mu_S F\ s\ \textsc{type}} \qquad \frac{\Gamma \vdash cf : (F^\dagger \mu_S F)[s]}{\Gamma \vdash \mathsf{in}^\mu_{sS,F}\ cf : \mu_S F\ s}$$

$$\frac{\begin{array}{l}\textit{given } F = s.(c\!:\!C \triangleleft r\!:\!R.\, n)\\ \Gamma \vdash P\ \textsc{type}[\Sigma\, S\, (\mu_S F\ )]\\ \Gamma \vdash p : (s\!:\!S) \to (\langle c, f\rangle\!:\!(F^\dagger \mu_S F)[s]) \to ((r\!:\!R) \to P[\langle n, f\, r\rangle]) \to P[\langle s, \mathsf{in}^\mu\, \langle c, f\rangle\rangle]\end{array}}{\Gamma \vdash \mathsf{ind}_{S,F}(P, p) : (sx\!:\!\Sigma\, S\, (\mu_S F\ )) \to P[sx]}$$

$$\overline{\Gamma \vdash \mathsf{ind}_{S,F}(P,p)\ \langle s, \mathsf{in}^\mu\, cf\rangle \equiv p\, s\, cf\, (\lambda r.\, \mathsf{ind}_{S,F}(P,p)\ \langle n[s, \pi_0 cf, r], \pi_1 cf\, r\rangle) : P[\langle s, \mathsf{in}^\mu\, cf\rangle]}$$

These inductive types are just the general tree types of Petersson and Synek [PS89], a sort of indexed refinement of Martin-Löf's W-types [ML84]. Hyland and Gambino have shown how to construct them from unindexed W-types in a suitably extensional setting [GH03].

Before we analyse these rules any further, let us have some examples of these types.

Example 2.11 (**Nat**—the natural numbers)**.** *The natural numbers have one sort with two constructors, the first with one recursive position, the second with none.*

$$\mathbf{Nat} := \mu_{\_}.(c\!:\!2 \triangleleft r\!:\!\mathbb{T}(c).\ \langle\rangle)\ \langle\rangle \quad \mathbf{zero} := \mathsf{in}^\mu\, \langle \mathsf{ff}, \lambda r.\, r \Psi \mathbf{Nat}\rangle \quad \mathbf{suc} := \lambda n.\, \mathsf{in}^\mu\, \langle \mathsf{tt}, \lambda r.\, n\rangle$$

Example 2.12 (**ParityList**—lists of even or odd length)**.** *The set of even-length lists of Xs can be given by indexing with* parity*.*

$$\mathbf{ParityList}(X, p) := \mu_2(p.(\langle c, \_\rangle\!:\!(c\!:\!2) \times \mathsf{Cond}(c, X, \mathbb{T}(p)) \triangleleft r\!:\!\mathbb{T}(c).\, \mathbf{not}\, p))\, p$$

The constructor takes a choice of tt *for 'cons' (in which case an element of X is also required), or* ff *for 'nil' (in which case the parity is checked to be even); recursive positions always demand the opposite parity.*

Readers familiar with the literature of dependently typed programming may be a little surprised not to find the *vectors*—lists of known length—as an example here. Vectors are a paradigmatic example of inductive families with *constrained constructors*: the 'nil' delivers only length **zero**, the 'cons' only some $\mathbf{suc}\, n$. In general, it takes some notion of propositional equality to capture such constraints, so vectors, as traditionally formulated, must wait. Alternatively, we may exploit the induction principle for (the encoding of) **Nat** to *analyse* the length and offer the appropriate structure.

The above induction principle follows the standard pattern for inductive datatypes in Martin-Löf Type Theory: we can use it to implement the projections, and so on. It computes by pattern matching and recursion. The idea, as ever, is that any P indexed by sorts and terms which is preserved by the construction of nodes must hold for all trees of every sort.

Following Jacobs and Hermida [HJ98], we may note that the function p is effectively an algebra, not for F but for $\hat{F}$ where $\hat{F}^\dagger P$ stores witnesses to predicates in the same places $F^\dagger \mu F$ keeps subobjects.

$$\hat{F} = \langle s, \mathsf{in}^\mu \langle c, f\rangle\rangle \,.(\_ : \mathsf{1} \lhd r : R.\ \langle n, f\, r\rangle)\ \textsc{cont}[\Sigma\, S\, (\mu_S F)]$$

However, it is a struggle to define the formal induction machinery in that form, as hinted at by the projection implicit in my binding of $\mathsf{in}^\mu \langle c, f\rangle$ in the above proto-definition of $\hat{F}$. It is straightforward to do the construction in the other direction. We may define both the projection and the ordinary iterator in terms of induction.

$$\begin{array}{l}
\mathbf{out}^\mu_F : \mu F \to F^\dagger \mu F \\
\mathbf{out}^\mu_F := \lambda s\, x.\, \mathsf{ind}_{_,F}(\langle s, _\rangle \,.\mu F\ s, \lambda s\, cf\ _.\, cf)\ \langle s, x\rangle \\
\mathbf{fold}_F(T, g : F^\dagger T \to T) : \mu F \to T \\
\mathbf{fold}_F(T, g) := \lambda s\, x.\, \mathsf{ind}_{_,F}(\langle s, _\rangle \,.T[s], \lambda s\ \langle c, _\rangle\ h.\, g\, s\, \langle c, h\rangle)\ \langle s, x\rangle
\end{array}$$

The computation rule then establishes that

$$\mathbf{out}^\mu_F\ s\ (\mathsf{in}^\mu\ cf) \equiv cf \qquad \mathbf{fold}_F(T, g)\ s\ (\mathsf{in}^\mu\ cf) \equiv F^\ddagger(\mathbf{fold}_F(T, g))\ s\ cf$$

3. Coinductive data in intensional type theory

Having established our basic type theory, let us now consider how we might equip it with coinductive types for. At the very least, we should express the notion that strictly positive endofunctors on $\textsc{type}[S]$ have final coalgebras. Let us add the coinductive types:

$$\frac{\Gamma \vdash F\ \textsc{cont}[S] \quad \Gamma \vdash s : S}{\Gamma \vdash \nu_S F\ s\ \textsc{type}}$$

and equip them with a coiterator, taking any $F^\dagger$-coalgebra to the final one.

$$\frac{\Gamma \vdash X\ \textsc{type}[S] \quad \Gamma \vdash m : X \to F^\dagger X}{\Gamma \vdash \mathsf{unfold}_{S,F}(X, m) : X \to \nu_S F}$$

We shall also need some sort of elimination operator, allowing us to *observe* initial segments of codata. Crucially, also, we must explain how the judgmental equality treats codata. Here we are sure to face some sort of intensional compromise, for codata are

potentially infinite. We should certainly expect to be able to implement the final coalgebra as a sort-indexed family of functions.

$$\mathbf{out}_{S,F} : \nu F \to F^\dagger(\nu F)$$

Moreover, we should expect that $\mathbf{out}_{S,F}$ provokes a step of unfolding:

$$\mathbf{out}_{S,F}\, s\, (\mathsf{unfold}(X, m)\, s\, x) \equiv F^\ddagger(\mathsf{unfold}(X, m))\, (m\, s\, x)$$

One might hope that **out** would have an inverse—a 'coconstructor'—and we might try to define one by coiteration:

$$\begin{aligned}\mathbf{in}^\nu_F &: F^\dagger \nu F \to \nu F\\ \mathbf{in}^\nu_F &:= \mathsf{unfold}(F^\dagger \nu F, F^\ddagger \mathbf{out})\end{aligned}$$

However, when we compute, we find (for $F := s.(c{:}C \triangleleft r{:}R.\, n)$) that

$$\mathbf{out}\, s\, (\mathbf{in}^\nu\, s\, \langle c, f\rangle) \equiv \langle c, \lambda r.\, \mathbf{in}^\nu\, n\, (\mathbf{out}\, n\, (f\, r))\rangle \not\equiv \langle c, f\rangle$$

This bad news should not especially surprise: when we implement the coconstructor by coiteration, we are really implementing an indirect corecursive version of the identity function—there is no reason why the latter should be recognized as the identity function, given only the equational theory for **out**.

Both Agda and Coq adopt a 'brute force' workaround, adding a distinct introduction form

$$\mathsf{in}^\nu_{s:S,F} : F^\dagger(\nu_S F)[s] \to \nu_S F\, s$$

which the implementation can distinguish from the coiterator, ensuring that

$$\mathbf{out}\, s\, (\mathsf{in}^\nu\ \langle c, f\rangle) \equiv \langle c, f\rangle$$

holds judgmentally. The coconstructor effectively presents codata which are partially unfolded already. Let us also adopt this convenience.

We have explored how we might construct and dismantle codata in type theory, but we should also hope to *reason* about the processes we may thus manipulate. Here Agda and Coq adopt distinct approaches, neither especially satisfying.

3.1. **Codata in Coq.** Eduardo Giménez pioneered Coq's treatment of coinduction [Gim94]. It was a great step forward in its time, giving Coq access to many new application domains. However, the Coq presentation is bedevilled with the problem that computation on codata does not preserve type. Giménez was aware of this difficulty, giving a counterexample in his doctoral thesis [Gim96]. 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.

I shall illustrate the issue by showing what goes wrong if we attempt to equip our type theory with the Coq presentation. The key advantage of, but also the problem with the Coq treatment is that codata have *dependent case analysis*. Every element x of a coinductive type behaves (up to observation) just like a partially unfolded element, and Coq expresses

this as a reasoning principle—like the induction principle for inductive data, but with no inductive hypotheses.

$$\frac{\begin{array}{l}\Gamma \vdash P\ \mathrm{TYPE}[(s{:}S) \times \nu F\ s] \\ \Gamma \vdash p : (s{:}S) \to (c{:}C[s]) \to (f{:}(r{:}R[s,c]) \to \nu F\ n) \to \\ \qquad P[\langle s, \mathsf{in}^\nu\ \langle c, f\rangle\rangle]\end{array}}{\Gamma \vdash \mathsf{case}_{S,F}(P,p) : (sx{:}(s{:}S) \times \nu_S F\ s) \to P[sx]}\quad F = s.(c{:}C \triangleleft r{:}R.\ n)$$

Given such a thing, one may deliver the final coalgebra by choosing P appropriately.

$$\mathbf{out}_{S,F} := \mathsf{case}_{S,F}(\langle s, x\rangle\ .\ F^\dagger(\nu F)[s], \lambda s.\ \lambda c.\ \lambda f.\ \langle c, f\rangle)$$

How does case compute? Coq supplies the following:

$$\begin{array}{l}\mathsf{case}_{S,F}(P,p)\ \langle s, \mathsf{in}^\nu\ \langle c, f\rangle\rangle \equiv p\ s\ c\ f : P[\langle s, \mathsf{in}^\nu\ \langle c, f\rangle\rangle] \\ \mathsf{case}_{S,F}(P,p)\ \langle s, \mathsf{unfold}_{S,F}(X,m)\ s\ x\rangle \equiv p\ s\ c\ f : P[\langle s, \mathsf{unfold}_{S,F}(X,m)\ s\ x\rangle] \qquad (?) \\ \quad \text{where } \langle c, f\rangle := F^\ddagger(\mathsf{unfold}_{S,F}(X,m))\ (m\ s\ x)\end{array}$$

and the latter does indeed validate the computation principle we require of **out**. There is, however, a problem with the law marked (?): the right-hand side of the equation does not typecheck—its type is

$$P[\Big\langle s, \mathsf{in}^\nu(F^\ddagger(\mathsf{unfold}_{S,F}(X,m))\ (m\ s\ x))\Big\rangle]$$

The case analysis principle unfolds coiteration at the value level, and this unfolding is reflected at the type level also. However, the judgmental equality may not identify these types. There is no trouble for non-dependent P, but in general,

$$\mathsf{unfold}_{S,F}(X,m)\ s\ x \not\equiv \mathsf{in}^\nu(F^\ddagger(\mathsf{unfold}_{S,F}(X,m))\ (m\ s\ x))$$

which is to say that the judgmental equality does not include 'spontaneous' unfolding in any context—only unfolding in the context of case.

4. Judgmental equality versus reduction

5. Weakly final coalgebras

6. Propositional equality and bisimulation?

7. Introducing observational equality

$$\frac{\Gamma \vdash S, T\ \mathrm{TYPE}}{\Gamma \vdash S \leftrightarrow T\ \mathrm{TYPE}} \qquad \frac{\Gamma \vdash s : S \quad \Gamma \vdash t : T}{\Gamma \vdash (s{:}S){=}(t{:}T)\ \mathrm{TYPE}}$$

Equality of indexed *families*, with the same index set or not, may readily be expressed by requiring coincidence given equal indices. I abbreviate

$$(s{:}S \parallel t{:}T) \to U \quad := \quad (s{:}S) \to (t{:}T) \to (s{:}S){=}(t{:}T) \to U$$

$$
\begin{array}{rl}
T_0 \leftrightarrow T_1 & \equiv \mathsf{0} \text{ if } T_0, T_1 \text{ canonical types of distinct formation rule} \\
\mathsf{0} \leftrightarrow \mathsf{0} & \equiv \mathsf{1} \\
\mathsf{1} \leftrightarrow \mathsf{1} & \equiv \mathsf{1} \\
\mathsf{2} \leftrightarrow \mathsf{2} & \equiv \mathsf{1} \\
((s_0 : S_0) \to T_0) \leftrightarrow ((s_1 : S_1) \to T_1) & \equiv S_1 \leftrightarrow S_0 \times (s_1 : S_1 \parallel s_0 : S_0) \to T_0 \leftrightarrow T_1 \\
((s_0 : S_0) \times T_0) \leftrightarrow ((s_1 : S_1) \times T_1) & \equiv S_0 \leftrightarrow S_1 \times (s_0 : S_0 \parallel s_1 : S_1) \to T_0 \leftrightarrow T_1 \\
\mu_{S_0} F_0\ s_0 \leftrightarrow \mu_{S_1} F_1\ s_1 & \equiv (S_0 \leftrightarrow S_1\ \times F_0 \rightleftharpoons F_1)\ \times (s_0 : S_0){=}(s_1 : S_1) \text{ where} \\
& \qquad s_0.(c_0 : C_0 \triangleleft r_0 : R_0.\, n_0) \rightleftharpoons s_1.(c_1 : C_1 \triangleleft r_1 : R_1.\, n_1) := \\
& \qquad (s_0 : S_0 \parallel s_1 : S_1) \to C_0 \leftrightarrow C_1 \times \\
& \qquad (c_0 : C_0 \parallel c_1 : C_1) \to R_1 \leftrightarrow R_0 \times \\
& \qquad (r_1 : R_1 \parallel r_0 : R_0) \to (n_0 : S_0){=}(n_1 : S_1)
\end{array}
$$

$$
\begin{array}{rl}
(\_ : T_0\ ){=}(\_ : T_1) & \equiv \mathsf{1} \text{ if } T_0, T_1 \text{ canonical types of distinct formation rule} \\
(\_ : \mathsf{0}){=}(\_ : \mathsf{0}) & \equiv \mathsf{1} \\
(\_ : \mathsf{1}){=}(\_ : \mathsf{1}) & \equiv \mathsf{1} \\
(\mathsf{tt} : \mathsf{2}){=}(\mathsf{tt} : \mathsf{2}) & \equiv \mathsf{1} \\
(\mathsf{tt} : \mathsf{2}){=}(\mathsf{ff} : \mathsf{2}) & \equiv \mathsf{0} \\
(\mathsf{ff} : \mathsf{2}){=}(\mathsf{tt} : \mathsf{2}) & \equiv \mathsf{0} \\
(\mathsf{ff} : \mathsf{2}){=}(\mathsf{ff} : \mathsf{2}) & \equiv \mathsf{1} \\
(f_0 : (s_0 : S_0) \to T_0){=}(f_1 : (s_1 : S_1) \to T_1) & \equiv (s_0 : S_0 \parallel s_1 : S_1) \to (f_0\ s_0 : T_0){=}(f_1\ s_1 : T_1) \\
(\langle s_0, t_0 \rangle : (s_0 : S_0) \times T_0){=}(\langle s_1, t_1 \rangle : (s_1 : S_1) \times T_1) & \equiv (s_0 : S_0){=}(s_1 : S_1)\ \times\ (t_0 : T_0){=}(t_1 : T_1) \\
(\mathsf{in}^\mu\ d_0 : \mu F_0\ s_0){=}(\mathsf{in}^\mu\ d_1 : \mu F_1\ s_1) & \equiv (d_0 : (F_0{}^\dagger \mu F_0)[s_0]){=}(d_1 : (F_1{}^\dagger \mu F_1)[s_1])
\end{array}
$$

Heterogeneous value equality, gives us two constructions to change the index set of a family in essentially administrative (and, intensionally speaking, administratively essential) ways. To understand what is going on, extensionally speaking, ignore these operations!

$$
\frac{A \text{ TYPE}[T]}{\bigcirc_{S,T} \!\!\to\!\! A \text{ TYPE}[S]} \qquad \bigcirc_{S,T} \!\!\to\!\! t.U := s.(t : T) \to (s : S){=}(t : T) \to U
$$

$$
\frac{t : T \quad q : (s : S){=}(t : T)}{\lfloor (t, q) : (\bigcirc_{S,T} \!\!\to\!\! A)[s] \to A[t]} \qquad \lfloor (t, q) := \lambda f.\, f\ t\ q
$$

$$
\frac{A \text{ TYPE}[S]}{\bigcirc_{S,T} \!\!\times\!\! A \text{ TYPE}[T]} \qquad \bigcirc_{S,T} \!\!\times\!\! s.U := t.(s : S) \times (s : S){=}(t : T) \times U
$$

$$
\frac{s : S \quad q : (s : S){=}(t : T)}{\rceil (s, q) : A[s] \to (\bigcirc_{S,T} \!\!\times\!\! A)[t]} \qquad \rceil (s, q) := \lambda x.\, \langle s, \langle q, x \rangle \rangle
$$

The former may be equipped with an unpacking operation, the latter a packer, as shown. We shall use these in the construction of algebras and coalegbras which shift data between indexed families of types.

$$\frac{\Gamma \vdash S, T \text{ TYPE} \quad \Gamma \vdash Q : S \leftrightarrow T \quad \Gamma \vdash s : S}{\Gamma \vdash \mathsf{coe}(S,T,Q) : S \to T \quad \Gamma \vdash \mathsf{coh}(S,T,Q) : (s{:}S) \to (s{:}S){=}(\mathsf{coe}(S,T,Q,s){:}T)}$$

$\mathsf{coe}(T_0, T_1, Q) \equiv \lambda\_.\, Q \Psi T_1$ if T_0, T_1 canonical types of distinct formation rule

$\mathsf{coe}(0,0,Q) \equiv \mathbf{id}_0$

$\mathsf{coe}(1,1,Q) \equiv \mathbf{id}_1$

$\mathsf{coe}(2,2,Q) \equiv \mathbf{id}_2$

$\mathsf{coe}((s_0{:}S_0) \to T_0, (s_1{:}S_1) \to T_1, \langle S_q, T_q\rangle) \equiv \lambda f_0\, s_1.\, \mathsf{coe}(T_0, T_1, T_q\, s_1\, s_0\, s_q, f_0\, s_0)$ where
$\langle s_0, S_q\rangle := \mathsf{coeh}(S_1, S_0, S_q, s_1)$

$\mathsf{coe}((s_0{:}S_0) \times T_0, (s_1{:}S_1) \times T_1, \langle S_q, T_q\rangle) \equiv \lambda\, \langle s_0, t_0\rangle\, .\, \langle s_1, \mathsf{coe}(T_0, T_1, T_q\, s_0\, s_1\, s_q, t_0)\rangle$ where
$\langle s_1, s_q\rangle := \mathsf{coeh}(S_0, S_1, S_q, s_0)$

$\mathsf{coe}(\mu F_0\, s_0, \mu F_1\, s_1, \langle\langle S_q, F_q\rangle, s_q\rangle) \equiv \lfloor (s_1, s_q) \cdot \mathbf{fold}_{F_0}(\bigcirc{\to}\mu F_1, g)\, s_0$ where
$s_0.(c_0{:}C_0 \triangleleft r_0{:}R_0.\, n_0) := F_0;\ s_1.(c_1{:}C_1 \triangleleft r_1{:}R_1.\, n_1) := F_1$
$g := \lambda s_0\ \langle c_0, f'\rangle\ s_1\, s_q.\, \mathsf{in}^{\mu}\, \langle c_1, f_1\rangle$ where
$f' : (r_0{:}R_0) \to (\bigcirc{\to}\mu F_1)[n_0]$
$\langle C_q, G_q\rangle := F_q\, s_0\, s_1\, s_q;\ \langle c_1, c_q\rangle := \mathsf{coeh}(C_0, C_1, C_q, c_0)$
$\langle R_q, h_q\rangle := G_q\, c_0\, c_1\, c_q$
$f_1 := \lambda r_1.\ \lfloor (n_1, h_q\, r_1\, r_0\, r_q)\, (f'\, r_0)$ where $\langle r_0, r_q\rangle := \mathsf{coeh}(R_1, R_0, R_q, r_1)$

8. Interaction structures, closed under bisimulation

$$\nu_{S_0} F_0\, s_0 \leftrightarrow \nu_{S_1} F_1\, s_1 \equiv (S_0 \leftrightarrow S_1\ \times F_0 \rightleftharpoons F_1)\ \times (s_0{:}S_0){=}(s_1{:}S_1)$$

$(x_0{:}\nu_{S_0} F_0\, s_0){=}(x_1{:}\nu_{S_1} F_1\, s_1) \equiv$
$\nu_{((s_0:S_0)\times\nu_{S_0}F_0\, s_0)\times((s_1:S_1)\times\nu_{S_1}F_1\, s_1)}$
$\langle\langle s_0, x_0\rangle, \langle s_1, x_1\rangle\rangle\, .(\_{:}(c_0{:}C_0){=}(c_1{:}C_1) \triangleleft$
$\langle r_0, \langle r_1, \_\rangle\rangle : (r_0{:}R_0) \times (r_1{:}R_1) \times (r_0{:}R_0){=}(r_1{:}R_1).$
$\langle\langle n_0, f_0\, r_0\rangle, \langle n_1, f_1\, r_1\rangle\rangle$
where $\langle c_0, f_0\rangle := \mathsf{out}\, x_0;\ \langle c_1, f_1\rangle := \mathsf{out}\, x_1;$
$\langle\langle s_0, x_0\rangle, \langle s_1, x_1\rangle\rangle$
where $s_0.(c_0{:}C_0 \triangleleft r_0{:}R_0.\, n_0) := F_0;\ s_1.(c_1{:}C_1 \triangleleft r_1{:}R_1.\, n_1) := F_1$

$\mathsf{coe}(\nu F_0\, s_0, \nu F_1\, s_1, \langle\langle S_q, F_q\rangle, s_q\rangle) \equiv \mathsf{unfold}_{\_,F_1}(\bigcirc{\times}\nu F_0, g)\, s_1 \cdot \rceil(s_0, s_q)$ where
$s_0.(c_0{:}C_0 \triangleleft r_0{:}R_0.\, n_0) := F_0;\ s_1.(c_1{:}C_1 \triangleleft r_1{:}R_1.\, n_1) := F_1$
$g := \lambda s_1\ \langle s_0, \langle s_q, x_0\rangle\rangle\, .\, \langle c_1, f'\rangle$ where
$\langle c_0, f_0\rangle := \mathsf{out}\, x_0$
$\langle C_q, G_q\rangle := F_q\, s_0\, s_1\, s_q;\ \langle c_1, c_q\rangle := \mathsf{coeh}(C_0, C_1, C_q, c_0)$
$\langle R_q, h_q\rangle := G_q\, c_0\, c_1\, c_q$
$f' : (r_1{:}R_1) \to (\bigcirc{\times}\nu F_1)[n_1]$
$f' := \lambda r_1.\ \rceil(n_0, h_q\, r_1\, r_0\, r_q)\, (f_0\, r_0)$ where $\langle r_0, r_q\rangle := \mathsf{coeh}(R_1, R_0, R_q, r_1)$

9. Conclusion and further work

Acknowledgement

The authors wish to acknowledge fruitful discussions with A and B.

References

[AMS07] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In Aaron Stump and Hongwei Xi, editors, *PLPV*, pages 57–68. ACM, 2007.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving And Program Development: Coq'Art: the Calculus of Inductive Constructions*. Springer, 2004.

[CF92] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, The University of Calgary, June 1992.

[CS92] 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.

[DS99] Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In Jean-Yves Girard, editor, *TLCA*, volume 1581 of *Lecture Notes in Computer Science*, pages 129–146. Springer, 1999.

[GH03] Nicola Gambino and Martin Hyland. Wellfounded trees and dependent polynomial functors. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2003.

[Gim94] 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.

[Gim96] Eduardo Giménez. *Un Calcul de Constructions Infinies et son application à la vérification de systèmes communicants*. PhD thesis, Ecole Normale Supérieure de Lyon, 1996.

[Hag87] Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1987.

[HJ98] Claudio Hermida and Bart Jacobs. Structural induction and coinduction in a fibrational setting. *Inf. Comput.*, 145(2):107–152, 1998.

[Hof95] 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/`.

[HS00] 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.

[ML84] Per Martin-Löf. *Intuitionistic type theory*. Bibliopolis, 1984.

[MM04] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[Nor07] Ulf Norell. *Towards a Practical Programming Language based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007.

[Pey03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[PS89] 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.

[TT97] 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.

[Tur95] 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.

[Tur04] D. A. Turner. Total functional programming. *J. UCS*, 10(7):751–768, 2004.