Cyclic Proofs and Coinductive Principles

Gavin E. Mendel-Gleason
LERO http://lero.ie
ggleason@computing.dcu.ie
Dublin City University and Geoff Hamilton
LERO http://lero.ie
hamilton@computing.dcu.ie
Dublin City University

Abstract

It is possible to provide a proof for a coinductive type using a corecursive function coupled with a guardedness condition. The guardedness condition, however, is quite restrictive and many programs which are in fact productive and do not compromise soundness will be rejected. We present a system of cyclic proof for an extension of System F extended with sums, products and (co)inductive types. Using program transformation techniques we are able to take some programs whose productivity is suspected and transform them, using a suitable theory of equivalence, into programs for which guardedness is syntactically apparent. The equivalence of the proof prior and subsequent to transformation is given by a bisimulation relation.

1 Introduction

In constructive type theories habitation of a type requires a demonstration of termination or co-termination for inductive and co-inductive types respectively. A failure to do so allows unsound propositions to be proved.

In the Coq theorem prover, the guardedness condition is the condition checked to ensure admissibility of *productive* programs. The condition has the advantage of only requiring a syntactic check on the form of the program and is a sufficient condition. However, this condition suffers from being highly restrictive. It rejects many programs which *are* in fact productive.

The Curry-Howard correspondence takes as a starting point the principle that logic can be viewed as a type system, and proofs of logical statements can be viewed as type-correct programs. This equivalence has lead to very fruitful developments in both logic and computation. We use this well known correspondence to bring techniques from program transformation to bear on the problem of inductive and co-inductive proofs. In order to do so however, we need a theory of proof that is sufficiently flexible to allow program transformations which make use of *folding* which is the creation of new (co)-inductive principles.

Cyclic proof is a method of describing inductive proofs without direct reference to an induction law [4]. Instead, it allows circularities in proofs, each cycle coupled with a side condition from which soundness of the argument follows. Arguments therefore closely resemble the structure of type correctness proofs for functional programming languages [10] and hence is an appealing way to define a type theory for a functional programming language. Inhabitation of a type then follows after the side conditions can be shown to hold, otherwise the argument is only partially correct.

We describe how program transformation of functional programs naturally situates in the context of cyclic proof. Here, the Curry-Howard correspondence is explicit and program transformation can be viewed as substitution of new proofs under an equivalence which preserves behaviour, that is, all, even potentially infinite, cut-free fragments of a proof.

This paper makes a number of novel contributions. We extend Brotherston's concept of cyclic proof to include coinductive proofs. We show a method by which we can perform transformations directly on proof trees. We demonstrate a bisimulation principle for demonstrating

the equivalence of proofs. Finally, we show how these techniques can be used to transform some proofs which do not meet the guardedness condition into proofs which do.

2 Language

The language we present is a functional programming language, which we will call Λ_F with a type system based on System F with recursive types. The use of System F typing allows us to ensure that transitions can be found for any term. Our term language will follow closely on the one used by Abel [2].

$\mathbf{Var} \ni x, y$			Variables
$\mathbf{TyVar} \ni X, Y$			Type Variables
$\mathbf{Fun} \ni f, g$			Function Symbols
$\mathbf{Ty} \ni A, B, C$	$::=$	$1 \mid X \mid A \to B \mid \forall X.A \mid A + B \mid A \times B$	Types
	$\mid$	$\nu X.A \mid \mu X.A$	
$\mathbf{Tr} \ni r, s, t$	$::=$	$x \mid f \mid () \mid \lambda x : A.t \mid \Lambda X.t \mid r\ s \mid r\ A$	Terms
	$\mid$	$\mathrm{inl}(t,\ )\mid \mathrm{inr}(t,\ )\mid (t, s)$	
	$\mid$	$\mathrm{fold}_\nu(t, A) \mid \mathrm{unfold}_\nu(t, A)$	
	$\mid$	$\mathrm{fold}_\mu(t, A) \mid \mathrm{unfold}_\mu(t, A)$	
	$\mid$	$\mathrm{case}\ r\ \mathrm{of}\ \mathrm{inl}(x_1) \Rightarrow s\ ;\ \mathrm{inr}(x_2) \Rightarrow t$	
	$\mid$	$\mathrm{split}\ r\ \mathrm{as}\ x_1, x_2\ \mathrm{in}\ s$	
$\mathbf{Ctx} \ni \Gamma$	$::=$	$\cdot \mid \Gamma, x : A \mid \Gamma, X$	Contexts

We will describe *substitutions* using the map σ which will represent assignment of variables to terms and type variables to types. Extension of a substitution will be written as $\sigma \cup (x, t)$ or $\sigma \cup (X, A)$. We will use a function $FV(t)$ to obtain the free type and term variables from a term. Substitutions of a single variable will be written $[X := A]$ or $[x := t]$ for type and term variables respectively.

We will also find the need to introduce recursive terms. Though it's possible that these can be suitably encoded in System F, it simplifies the presentation to provide them explicitly.

Recursive terms will be represented using function constants. Function constants will be drawn from a set $\mathbf{F}$. We will couple our terms with a function Δ which associates a function constant f with a term e, $\Delta(f) = e$, where e may itself contain any function constants in the domain of Δ.

For a term t with type T in a context Γ we will write $\Gamma \vdash t : T$. A type derivation must be given using the rules given in Table 1.

Without further restrictions, this type system is unsound. First, the Delta rule for function constants clearly allows non-termination given $\Delta(f) = f : T$. We will deal with this potentiality later when we describe cyclic proof.

In addition, the use of iso-recursive types introduces the possibility of non-termination if the type is not given some additional restriction. We will restrict the form of types which can be used with the Fold/Unfold rule to meet an additional positivity condition.

Strictly Positive A type variable X is *strictly positive* with respect to a type T iff X does not occur on the left side of an implication in T and T is built of strictly positive types. We will then say that the type $\mu X.T$ (or $\nu X.T$) is strictly positive.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{Var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\lambda x : A.t) : A \to B}\ \text{ImpIntro}$$

$$\frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\ s : B}\ \text{ImpElim} \qquad \frac{\Gamma, X \vdash t : A}{\Gamma \vdash \Lambda X.t : \forall X.A}\ \text{AllIntro}$$

$$\frac{\Gamma \vdash t : \forall A.T}{\Gamma \vdash t\ B : A[X := B]}\ \text{AllElim} \qquad \frac{\Gamma, f : A \vdash \Delta(f) : A}{\Gamma \vdash f : A}\ \text{Delta}$$

$$\frac{}{\Gamma \vdash () : 1}\ \text{Unit} \qquad \frac{\Gamma \vdash r : A \qquad \Gamma \vdash s : B}{\Gamma \vdash (r, s) : A \times B}\ \text{PairIntro}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \text{inl}(t, T + S) : (T + S)}\ \text{OrIntroL} \qquad \frac{\Gamma \vdash t : S}{\Gamma \vdash \text{inr}(t, T + S) : (T + S)}\ \text{OrIntroR}$$

$$\frac{U = \nu X.T \qquad \Gamma \vdash t : T \qquad \alpha \in \{\mu, \nu\}}{\Gamma \vdash \text{unfold}_\alpha(t, U) : T[X := U]}\ \text{Unfold}$$

$$\frac{U = \nu X.T \qquad \Gamma \vdash t : T[X := U] \qquad \alpha \in \{\mu, \nu\}}{\Gamma \vdash \text{fold}_\alpha(t, U) : U}\ \text{Fold}$$

$$\frac{\Gamma \vdash e : T + S \qquad \Gamma, x : T \vdash t : U \qquad \Gamma, y : S \vdash s : U}{\Gamma \vdash (\text{case } e \text{ of inl}(x) \Rightarrow t\ ;\ \text{inr}(y) \Rightarrow s) : U}\ \text{OrElim}$$

$$\frac{\Gamma \vdash s : T \times S \qquad \Gamma, x : T, y : S \vdash t : U}{\Gamma \vdash (\text{split } s \text{ as } x_1, x_2 \text{ in } t) : U}\ \text{PairElim}$$

Table 1: System F Proof Rules

3 Cyclic Proof

In order to allow more fluidity in the structure of our proof trees we introduce a notion of a cyclic proof. A cyclic proof can be presented by making reference to a syntactically identical prior node. Our presentation is much the same as the one given by Brotherston [4]. We will however extend the definition to deal with coinductive types.

Rule Graph A *rule graph* is a connected graph $G = (V, s, r, p)$ where V is a set of vertices, $r : V \rightharpoonup \mathit{Rules}$ a map from vertices to some finite set of rules, $s : V \to \mathit{Seqs}$ a map from vertices to a sequent and $p : V \to [V]$ a map from vertices to lists of vertices.

For our purposes the sequents for a rule graph will be of the from $\Gamma \vdash t : A$. The map p, associates the antecedents of a vertex v which derive the consequent given by s.

Derivation Tree A *derivation tree* $\mathcal{D}$ is a rule graph $G = (V, s, r, p)$ with the following constraints:

- There is a single privileged vertex $v_0 = \mathit{root}(V)$ which is not the target of any arc called the *root* of the derivation tree. More formally: $\forall v \in V. v_0 \notin p(v)$
- The number of antecedents $|p(v)|$ of a vertex is consonant with the arity of the rule $r(v)$.

- The codomain of s is from the set Seq of well formed sequents.
- R_i is the ith rule in some list of rules $Rules$ with antecedents $\overrightarrow{D}$ and consequent Seq

Structural Proof A *structural proof* is a finite derivation tree $\mathcal{D}$ of a consequent $\Gamma \vdash t : \tau$.

Bud / Companion Node Given a derivation $\mathcal{D}$ a *bud* is a vertex $v \in V$ such that $r(b)$ is undefined.

A *companion* node of a bud node is a vertex v' such that $s(v) = s(v')$ for some vertex v and $s(v) = \Gamma \vdash t : \alpha X.T$ for $\alpha \in \mu, \nu$.

Pre-Proof A *pre-proof* is a pair $(\mathcal{D}, \mathcal{R})$ such that for each bud node v the partial function $R \rightharpoonup V$ associates a companion node.

Pre-Proof Graph Given a $\mathcal{P} = (\mathcal{D} = (V, r, s, p), \mathcal{R})$ a pre-proof, we can associate a graph $\mathcal{G}_\mathcal{P} = (V', r, s, p)$ by identifying buds with companions.

The pre-proof essentially mirrors the type checking algorithm familiar from functional programming languages such as Haskell [7] [10] [9]. Intuitively, the method is to assume the type of a recursive function, place it in the context and then unify with its subsequent occurrence.

The reason for the use of cycles rather than assumptions on the type of function constant is that more comprehensive changes to the proof tree will be possible. This will be clarified later when we look at transforming the proof.

Once a cyclic pre-proof is presented, we still have additional proof obligations in order to show that the type derivation is sound. While such a cyclic pre-proof proves *safety* [10] it can not be used for a constructive type theory since it allows inhabitation of every type.

We will not prove general soundness conditions, but rely on prior work showing that structural induction and the guardedness are sufficient conditions. Once these conditions have been satisfied, we can assume the correctness of the proof.

Path A path $< v_0, \cdots, v_n >$ is a sequence of vertices such that for each v_i, $v_{i+1} \in s(v_i)$ where $i < n$.

Sub-term A sub-term of a term t which has a derivation with root v_0, is any term which has a derivation starting from v_i which is in a path from v_0, and which contains only introductions (ImpIntro, AllIntro, OrIntroL, OrIntroR, PairIntro, Fold).

Structural Recursion A pre-proof with cycles will meet the structural recursion condition, if every bud associated with a μ type is a derivation of a sub-term of its companion.

Guardedness A pre-proof will meet the guardedness condition if every path from every companion to a bud follows a right branch of a cut (ImpElim, AllElim, OrElim, PairElim, and entirely excluding Unfold which has only one rule) and has a fold.

4 Evaluation And Equivalence

The utility which we will derive from the use of cyclic proof is in the ability to present the *same* proof in new ways by way of transformation. In order to say what we mean by the same however, we have to have a notion of equivalence. The typical evaluation relation for the lambda type lambda calculus leads us to a notion of equivalence between terms of the following form:

$(\lambda x : A.t)s = t[x := s]$	Provided that x is not free in s.
split (r, s) as x, y in $t = t[x := r, y := s]$	Provided that x, y are not free in r, s.
case $inl(r)$ of $\text{inl}(x) \Rightarrow s$; $\text{inr}(y) \Rightarrow t = s[x := r]$	Provided that x is not free in r.
case $inr(r)$ of $\text{inl}(x) \Rightarrow s$; $\text{inr}(y) \Rightarrow t = t[y := r]$	Provided that y is not free in r.
$\text{unfold}_\mu(foldmuxT, T) = x$	
$\text{unfold}_\nu(foldnuxT, T) = x$	

This equivalence can be lifted up to derivations. We show below the proof tree associated with the first equation, provided of course that we interpret replacement of the derivation to mean that every leaf with $\Gamma \vdash x : A$ is replaced with $\Gamma \vdash r : A$.

$$\begin{array}{c} \Gamma \vdots x : A \\ \vdots \qquad \vdots \\ \dfrac{\Gamma \vdash r : A \qquad \Gamma \vdash t : A \to B}{\Gamma \vdash s : B}\,\text{ImpElim} \end{array} \quad = \quad \begin{array}{c} \vdots \\ \Gamma \vdash r : A \\ \vdots \\ \Gamma \vdash s : B \end{array}$$

This is the typical β equivalence that we expect from the lambda calculus or it's equivalent of cut-elimination in proofs. We then extend this to deal with all of the previous equations.

From this equivalence of proofs we will then make a further equivalence over potentially infinite proofs, which can occur from the unfolding of function constants. This equivalence relation will use a coinductive principle to establish a bisimilarity of proof objects.

Bisimulation A proof D will be considered a bisimulation of another proof D', written as $D\ D'$ if $D = D'$ or if a decomposition of the proof tree of D and D' have $R = R'$ and $D_i\ D'_i$ under the assumption $D\ D'$, where D and D' are written as below.

$$D \quad = \quad \dfrac{D_0 \quad \cdots \quad D_n}{\Gamma \vdash t : A}\,R$$

$$D' \quad = \quad \dfrac{D'_0 \quad \cdots \quad D'_n}{\Gamma \vdash s : B}\,R$$

Intuitively, two proofs will be identical under this relation if they are equal (modulo our equivalence relation corresponding to reduction), or if they have equal rules and bisimilar subparts.

5 Example

We take an example using the *co-natural numbers* $\mathbb{N}^\infty \equiv \nu X.1 + X$ and potentially infinite lists $[A] \equiv \Lambda A.\nu X.1 + (A \times X)$. Here we take Δ to be defined as:

```
Δ(zero) := fold(inl(),ℕ^∞)
Δ(plus) := λ x y : ℕ^∞ .
                    case (unfold(x,ℕ^∞)) of
                       | inl(z) ⇒ ys
                       | inr(n) ⇒
                         fold(inr(plus n y),ℕ^∞)
Δ(sumlen) := λ xs : [ℕ^∞] .
                    case (unfold(xs,[ℕ^∞])) of
                       | inl(nil) ⇒ zero
```

```
                    | inr(p) ⇒
                      split p as (n,xs′)
                         in fold(inr(plus n (sumlen xs′)),ℕ^∞)
```

The program *sumlen* is meant to calculate the sum of elements in a potentially infinite list, plus the size of the list. It will not pass a test of the guardedness condition, despite being productive, since it involves internal cuts with the plus function. However, by producing a cyclic proof, unfolding and subsequently eliminating intermediate cuts we will be able to prove productivity.

A perhaps more interesting example is the following filter function given by Komendantskaya and Bertot in [3]. We start with the following program:

```
p Zero = Zero
p (Succ x') = x'
le Zero _ = True
le _ Zero = False
le (Succ x') (Succ y') = le x' y'
f (x:y:tl) = case (le x y) of
               True -> x:(f (y:tl))
               False -> f ((pred x):y:tl)
```

We can then create the transition system demonstrating type-correctness and reify to obtain the following functions:

```
redz Zero xs = Zero:(f (Zero:xs))
redz (Succ x) xs = redz x xs
red x y tl = case le x y of
               False -> case x of
                           Zero -> Zero:(f ((Succ y):tl))
                           Succ x' ->  red x' y tl
               True -> (Succ x):(f ((Succ y):tl))
le Zero _ = True
le _ Zero = False
le (Succ x') (Succ y') = le x' y'
f (Zero:y:tl) = Zero:(f (y:tl))
f ((Succ x):Zero:tl) = redz x tl
f ((Succ x):(Succ y):tl) = red x y tl
```

As we can see here, the *always, eventually* notion of productivity that is given in [3] is recast as a coinductive function meeting the guardedness condition, with auxiliary inductive functions representing the *eventually* criterion.

6 Conclusion and Prior Work

We have given what is to our knowledge the first account of cyclic proofs over mixed inductive and coinductive types. We demonstrate the utility of such a presentation by showing how proofs can more easily be transformed to equivalent proofs which meet syntactic restrictions ensuring soundness.

The correspondence between cyclic proof and functional programs has previously been described by Robin Cockett [5]. His work also makes a distinction between inductive and coinductive types.

Gordon developed the use of transition systems for the purpose of demonstrating bisimilarity [7]. Gordon showed that bisimilarity was a contextual equivalence relation. In future work we hope to show that the notion of equivalence given here is a contextual equivalence relation for the variant of System F given.

Various methods of dealing with inductive and coinductive types in a constructive type theory are demonstrated in [8] [6] [1]. The approach taken here differs in that we use transition systems rather than saturated sets to show consistency.

Using cyclic proofs with side conditions allows for more direct proofs of functional program correctness than do methods such as syntactic restrictions [11] [8]. As such, this direct method seems more suitable for use as an aid to proving correctness of real wold functional programs.

The authors care currently working on an implementation of a proof system using the ideas presented in this paper.

Acknowledgments. This work is supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero, the Irish Software Engineering Research Centre (www.lero.ie).

References

[1] Andreas Abel. Mixed inductive/coinductive types and strong normalization. In *APLAS*, pages 286–301, 2007.

[2] Andreas Abel. Typed applicative structures and normalization by evaluation for system f^{ω}. In Erich Grädel and Reinhard Kahle, editors, *CSL*, volume 5771 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2009.

[3] Yves Bertot and Ekaterina Komendantskaya. Inductive and coinductive components of corecursive functions in coq. *Electron. Notes Theor. Comput. Sci.*, 203(5):25–47, 2008.

[4] James Brotherston. Cyclic proofs for first-order logic with inductive definitions. In B. Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods: Proceedings of TABLEAUX 2005*, volume 3702 of *LNAI*, pages 78–92. Springer-Verlag, 2005.

[5] J. Robin B. Cockett. Deforestation, program transformation, and cut-elimination. *Electr. Notes Theor. Comput. Sci.*, 44(1), 2001.

[6] Eduardo Giménez. Structural recursive definitions in type theory. In *ICALP '98: Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, pages 397–408, London, UK, 1998. Springer-Verlag.

[7] Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Theor. Comput. Sci.*, 228(1-2):5–47, 1999.

[8] Christine Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In *TLCA '93: Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345, London, UK, 1993. Springer-Verlag.

[9] Simon L. Peyton Jones and David R. Lester. *Implementing functional languages.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[10] Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

[11] D. A. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.