Electronic Notes in Theoretical Computer Science 44 No. 1 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume44.html 40 pages

# Deforestation, program transformation, and cut-elimination

Robin Cockett [1,2]

*Department of Computer Science*
*University of Calgary*
*Calgary, Alberta, Canada*

**Abstract**

The problem of proving that two programs, in any reasonable programming language, are equivalent is well-known to be undecidable. In a formal programming system, in which the rules for equivalence are finitely presented, the problem of provable equivalence is semi-decidable. Despite this improved situation there is a significant lack of generally accepted automated techniques for systematically searching for a proof (or disproof) of program equivalence. Techniques for searching for proofs of equivalence often stumble on the formulation of induction and, of course, coinduction (when it is present) which are often formulated in such a manner as to require inspired guesses.

There are, however, well-known program transformation techniques which do address these issues. Of particular interest to this paper are the deforestation techniques introduced by Phil Wadler and the fold/unfold program transformation techniques introduced by Burstall and Darlington. These techniques are shadows of an underlying cut-elimination procedure and, as such, should be more generally recognized as proof techniques.

In this paper we show that these techniques apply to languages which have both inductive and coinductive datatypes. The relationship between these program transformation techniques and cut-elimination requires a transformation from initial and final "algebra" proof rules into "circular" proof rules as introduced by Santocanale (and used implicitly in the model checking community). This transformation is only possible in certain proof systems. Here we show that it can be applied to cartesian closed categories with datatypes: closedness is an essential requirement. The cut-elimination theorems and attendant program transformation techniques presented here rely heavily on this alternate presentation of induction and coinduction.

[1] Email: robin@cpsc.ucalgary.ca
[2] Partially supported by NSERC, Canada.

# 1 Introduction

The problem of proving two programs equivalent in any reasonable programming language is well-known to be undecidable. In formal programming systems, in which equivalence is determined by a finite set of rules, the situation is brought under better control as one can, at least, secure the semi-decidability of the problem. Despite this improvement, even in formal systems, there is still a significant lack of generally accepted automated techniques for systematically searching for a proof (or disproof) of the equivalence of two programs. Techniques for searching for a proof that two programs are equivalent often flounder on the formulation of induction and, of course, on coinduction (when coinductive types are present). This is reinforced by two strongly entrenched traditions:

- The first is the mathematical tradition of inductive proofs in which an inspired "guess" to formulate an inductive step is often portrayed as being unavoidable. Not only is this view of developing proofs anathema to automation but it has also fueled the perception that searching for inductive (and therefore coinductive proofs) is intrinsically difficult.
- The second is the computer science habit of allowing general recursive programs which need not terminate. This means that thrown into any (inductive) proof of equivalence there is a second problem of showing the equivalence of termination conditions. This not only complicates the proof system but it also creates a psychological "black hole" into which two traditional impossibilities [3] have been concentrated!

## *1.1 Program equivalence in formal systems*

The objective of this paper is to provide a view of the proof theory of inductive and coinductive datatypes which justifies the discomfort with the traditional inductive proof techniques mentioned in the first point. In order to do this I will present a formal programming language and discuss its proof theory. I will dodge the "black hole" mentioned in the second point above by choosing a the formal system which has good termination properties. The precise meaning of "good termination properties" is a little technical. While this is not the main focus of the paper, in order to understand more precisely what program equivalence actually means in these formal programming systems it is useful to understand more precisely what the ability to evaluate programs provides *and* does not provide.

[3] It is beyond the scope of this paper, but perhaps worth mentioning, that recent advances in the equational understanding of partiality (see [6,3]) show that it is possible to formulate simple equational programming logics which give a formal account of partiality. This means that even out of this "black hole" might yet be teased a surprising amount of light!

### *1.1.1 Evaluation in formal programming systems*

The language I shall consider is a variant of **charity** [5]: it has both inductive and coinductive datatypes. This means that we cannot promise termination in its full force. Consider, for example, the program which produces the infinite list of primes (which can certainly be written in this language): clearly it is impossible to produce the whole list. Instead the language has the weaker ability to reduce in a terminating manner to a "head normal form." This allows the primes, in our example, to be produced one-by-one on a demand basis with the *guarantee* that we can always produce the next prime in a terminating manner. For further discussion of these issues see [16].

A head normal form will often contain further unevaluated material which can be unlocked by the process of coinductive destruction. This gives rise to a "lazy" evaluation strategy for coinductive types which is not simply a side-effect of evaluation order: it is quite fundamental to the whole type system. Outside the coinductive records (which, recall, are predetermined by type) one can perform evaluation *in any order*. A coinductive record in effect freezes its arguments. This means that one can in fact mix (with potential efficiency advantages) by-value evaluation and lazy evaluation in this sort of language *without changing the termination properties*.

Now evaluation only applies to closed terms and thus, strictly speaking, cannot resolve the question of equivalence of programs – which will not usually be closed. Of course, in a higher-order language we may internalize arbitrary programs as closed terms of a higher type. However, these higher types are coinductive in nature so that evaluation simply produces the unevaluated program as a head normal form and, thus, no extra information is gained concerning equality.

However, before we dismiss evaluation it is worth recalling that it can be used to *disprove* equivalence. Essentially the idea is to find ground "values" (or "points") on which the two programs perform differently. For an arbitrary program this is achieved by a combination of providing values, and destructing to ground types. These processes are both enumerable so can be undertaken in a methodical manner. Clearly this technique of distinguishing programs using their pointwise behavior should be an integral component of any automated proof system as trying to prove something which is obviously false can cause a great deal of wasted effort!

### *1.1.2 Indistinguishable and formally equivalent programs*

We note that this discussion has revealed a fundamental (and possibly rather disturbing) feature of formal programming systems: two programs whose behavior cannot be distinguished on points are not necessarily provably equal. If they were we would immediately have a decision procedure for equality - which we cannot have! Thus, the indistinguishable programs and the provably equivalent programs in such systems are never the same.

Thus, the fact that there are no points on which two given programs can be

distinguished does not imply that there is necessarily a proof that the two programs are equivalent. This, in turn, raise the unhappy specter that the proof system may be so weak that one may routinely encounter programs which are clearly indistinguishable on "points" and yet cannot be proven equivalent!

In fact, in the system I shall be considering below this is not the case. It is known that equivalent and indistinguishable programs coincide for low complexity programs and therefore for these programs the equivalence problem is decidable. However, the precise manner in which program complexity relates to decidability – that is where indistinguishable and equivalence of programs coincides is an open question of some interest.

### *1.2 On formulations of induction and coinduction*

We now turn more directly to the issue of program equivalence and, in particular the question of how to tame inductive and coinductive program equivalence proofs.

The first small step away from standard mathematical induction based on the natural numbers is the step to, so called, structural induction. This allows a convenient expression of inductive arguments for arbitrary inductive (or initial) datatypes in much the same form as mathematical induction where the form of the inductive step is determined by the structure of the datatype (see Slind [13]).

However, as a principle which could encompass coinductive as well as inductive datatypes in a programming setting it has significant shortcomings. To start with its formulation relies on the presence of a calculus of subobjects (subsets): a structural inductive argument essentially works by showing that the subobject determined by a proposition must, in fact, be the complete object. Now it is arguable whether program logics *a priori* come equipped with a calculus of subobjects strong enough to support these sorts of inductive arguments.

Putting aside our concerns over the level of logic implied by these arguments, we are immediately forced to a further level of discomfort by what this approach implies for coinduction. For the analogous theory of structural coinduction would have to use a quotient structure. Again one may, this time with more force, argue that this is absent at the level of program logic. Furthermore, this structure compared to that for subobject – if indeed we allow that *it* is present – is considerably less well-behaved. In practice one has to simulate this structure through the subobject structure and this leads one ultimately to the theory of bisimulation.

Now, I do not want to suggest that the theory of bisimulation for coinductive types is without value: in fact, I believe it is an important tool in our understanding of coinduction. However, to reach this point one cannot fail to notice that one has had to employ a series of sophisticated contortions. In particular, these steps completely obscure the symmetry between inductive

and coinductive types as one has been forced to overlay the basic program logic with further structure which heavily emphasizes the asymmetry of the underlying setting.

I think, therefore, it becomes a reasonable question to ask whether there might not be more direct approaches than this to a proof system that encompasses coinduction.

### *1.2.1 The categorical formulation*

An obvious alternative starting point, in which the symmetry between induction and coinduction is given by *duality*, uses the categorical universal property associated with inductive (initial) and coinductive (final) datatypes. This is familiar to category theorists who would claim that this the fundamental property expected of datatypes. The basic idea (see section 3) is that inductive (coinductive) datatypes are initial algebras (*resp.* final coalgebras) and as such have uniquely determined maps to all other algebras (*resp.* from all other coalgebras).

However, there is a major problem with this (categorical) abstract view: it is difficult to use as a proof principle. In particular one may need considerable inspiration to produce an algebra (or coalgebra) which secures a desired equivalence. Thus, from the standpoint of proof automation, this is discouraging and hardly an improvement on the original situation described for mathematical induction.

However, we should not loose sight of the fact that symmetry has been regained. Furthermore, these notions do seem to capture the fundamental properties of these structural types. Thus, any system we might choose to devise should, at a very minimum, satisfy the abstract categorical properties of initial and final datatypes.

### *1.2.2 Lessons from model checking*

Initial and final algebras, under the guise of least and greatest fixed points, are also used extensively in model checking and, in particular, the modal-$\mu$ calculus. These developments have found a lucrative application in hardware and protocol verification. While these applications are concerned with posetal models the abstract proof theory involved applies to programs viewed as proofs.

Strikingly the more successful of these logics abandon the initial and final fixed point rules (which are the categorical initial and final "algebra" rules but in that community they are often called "the Kozen rules" [9]) in favor of more complex proof techniques. I will refer to these as "circular proof rules" following Santocanale [11] who introduced them to me in roughly the form I will use here. The advantage of this movement away from the simple fixed point rules is significant as these systems allow one to see quite easily that derivability, that is the ability to prove one thing from another, is decidable.

While this is not the problem we wish to solve it is an indication of an

increased power that these circular proof systems provide. From a proof theoretic point of view there is also a strong imperative to adopt this modified proof system: the categorical or Kozen system does not have a very satisfactory formulation of cut elimination.

At this point in time it may be rather hard to substantiate, from the model checking community itself, the claims that I am making. Model checkers do not generally take a proof theoretic view of the world. Indeed there is no *direct* published proof of cut-elimination for the modal-$\mu$ calculus even with the "circular proofs" (as far as I know). Instead there is a game theoretic argument which translates between proofs in the modal-$\mu$ calculus and winning strategies between games and then argues that winning strategies can be composed. A direct proof is, in fact, simpler and will appear, together with a suitable formulation of the modal-$\mu$ calculus, in Aldwinckle's thesis [1].

### *1.3 The importance of cut-elimination*

A basic property of a proof system is that one should be able to compose proofs: this is the import of the cut rule in logical systems. The ability to compose proofs in this manner is clearly absolutely fundamental to the view of programs as proofs which we wish to exploit here. A crucial observation of Gentzen [14] is that logical systems often allow one to eliminate the cut by a series of proof rewritings. Basically this meant that if there was a proof from one proposition to another (which perhaps used some intermediate propositions) then there is a direct proof which does not use any intermediate propositions.

Anyone familiar with Phil Wadler's ideas on program deforestation [17] will see a parallel between the ideas there and the aim of cut-elimination. This is made more explicit in Simon Marlow's thesis [10]. The idea of program deforestation is precisely to remove from the program the building of intermediate datatypes. This removal often results in more efficient programs, as intermediate structures are no longer built only to be destroyed, and so this has been of great interest to the program transformation and compiler community.

It is tempting to think that cut elimination and deforestation are the same thing for programs. However, this is not the case: deforested terms can still have cuts present. Indeed, in some cases, there *necessarily* must be residual cuts. However, the cuts that do remain are "soft" in that they only rearrange structure - more precisely they are cuts essentially from the $\lambda$ -calculus portion of the logic. The cuts that are removed are the "deforesting" cuts: these are the ones which cause the destruction of datatypes (including coproducts and products).

In a logical system the cut-elimination process has further significance. It is a constructive process and its steps, which are usually not confluent actually determine equalities which are needed between proofs.[4] This is why

[4] It is not the case that all proof equivalences are necessarily forced by cut elimination: logical systems can also have representation rules (such as "a comma on the left is equivalent

cut-elimination is also relevant to the discussion of program equivalence: thus, one can in some sense determine the notion of program equivalence from a cut elimination process. Now the form of the equivalence rules which arise naturally from these considerations in a circular proof system bear a striking resemblance to the fold/unfold methodology of program transformation [2]: this may help to explain why those ideas, despite being only partially correct, remain the basis for most of the high-level optimizing techniques for programs.

I should mention that there is another important approach to program transformation which does use in a fundamental manner the categorical initial and final algebra presentation of the proof system. These are the fold/build-fusion techniques introduced in [**?**] and further extended in [**?**]. These laws rely on the fact that a fold applied to a datatype produced by a "producer" program can be presented as replacing the constructors of the produced datatype by the function arguments of the fold. This means that if we can abstract the (right) constructors in the producer program one can replace the composite by a simple higher-order application thereby eliminating this composition (or cut). This technique generalizes to coinductive dataypes. The difficult with the technique is, of course, to determine how to abstract the right constructors in the producer.

### *1.3.1 Transforming the proof system*

The transformation of the proof system from one based on initial and final datatypes (which I call the "algebra based" system) to a "circular proof" system was a relatively easy conceptual step for the model checking community because their models were posetal. This relieved them of the necessity of maintaining the equality of the proofs. Unfortunately, when one regards proofs as programs, proof equality becomes *the* major concern.

Also if proofs are to become programs in a programming language the manner of correctly expressing circular proofs becomes another matter of some interest. This direction has already been developed by David Turner: in fact, in [15] he argues that this style of programming, which he calls "elementary strong functional programming," is the future of functional programming. The discussion in this paper links his ideas to the **charity** system by showing that his programming system (in so far as it is the circular proof system - and I have not checked this in detail) have the same power. It is a tribute to David Turner's great programming intuition that he discovered and realized the significance of this system

It turns out that a circular proof system implies the presence of initial and final datatypes but the reverse implication is dependent on the particulars of the proof system (if it is true at all). Given the nice properties of a circular proof system it is tempting just to use circular proofs directly rather than rules which might be implied by initial and final datatypes. However, circular

to a product") which may produce further proof identifications.

proofs interact ultimately with all the rules of the system and if the whole package does not fit together the system simply will not be well-behaved. Therefore, I go to some trouble to establish that the programming logic, which I introduce below, has both a circular proof presentation and an algebra based presentation.

### *1.4 Contents*

After this rather lengthy introduction it is appropriate to show how these relationships play out for a particular programming logic. To this end we present a term calculus for cartesian closed categories with positive datatypes. We describe how it may be viewed as a proof system and illustrate some simple programming examples in the system. In this case the "circular" programs, which are very much in the functional style, provide an equivalent formulation to the one implied by initial and final datatypes. Finally we discuss the implied program equivalences, a technique for deforesting, and the possibilities for automating equivalence proofs.

## 2 Program logic for inductive and coinductive datatypes

We will set up the programming logic as a type theory as we wish to exemplify the propositions as types and programs as proofs paradigm. The semantic basis of our programming language is a cartesian closed category with positive datatypes. We shall take the view that the coproducts are primitive inductive datatypes while products and exponentiation with a fixed object $(A \Rightarrow \_)$ are primitives of the coinductive structure.

To avoid the usual issues which arise from having contravariant functors (and to keep things simple) we shall only admit as **primitive functors**:

- The constant functors $K$;
- The product functor $X \times Y$, or $\Pi_{i \in I} X_i$ for $I$ a finite set, with 1 being the empty product;
- The coproduct functor $X + Y$, or $\sum_{i \in I} Y_i$ for $I$ a finite set, with 0 being the empty coproduct;
- Exponentiation with respect to fixed objects $A$ as in $A \Rightarrow X$.

We shall refer to the functors derived from these basic functors as the **polynomial** functors of the setting. These, of course, can involve multiple free type variables. These basic polynomials are the basis for forming the **datatypes** of the setting: these include the polynomials and those types which can be obtained from the basic functors and the following two forms of type binding:

- An **inductive** binding $\mu X.T$ giving the initial datatype or least fixed point of the type $T$ with respect to the type variable $X$;
- A **coinductive** binding $\nu X.T$ giving the final datatype or greatest fixed

$$\frac{\{\Gamma_1, x_i : X_i, \Gamma_2 \vdash t_i : Y\}_{i\in I}}{\Gamma_1, z : \sum_{i\in I} X_i, \Gamma_2 \vdash \left\{ ix_i \mapsto t_i \right\}_{i\in I} z : Y}\ sum_L$$

$$\frac{\{\Gamma \vdash t_i : Y_i\}_{i\in I}}{\Gamma \vdash (i : t_i)_{i\in I} : \Pi_{i\in I} Y_i}\ prod_R \qquad \frac{\Gamma \vdash t : Y_i}{\Gamma \vdash i(t) : \sum_{i\in I} Y_i}\ sum_R$$

$$\frac{\Gamma_1, x_1 : X_1, ..., x_n : X_n, \Gamma_2 \vdash Y}{\Gamma_1, (i_1 : x_1, ..., i_n : x_n) : \Pi_{i\in I} X_i, \Gamma_2 \vdash Y}\ prod_L$$

$$\frac{\Gamma \vdash s : Y \quad \Gamma x : X \vdash t : Z}{\Gamma f : Y \Rightarrow X \vdash t[f@s/x] : Z}\ close_L \qquad \frac{x : X, \Gamma \vdash t : Y}{\Gamma \vdash \lambda x.t : X \Rightarrow Y}\ close_R$$

Fig. 1. Sum, product, and closure rules

point of the type $T$ with respect to the type variable $X$.

These constitute the **positive types** of the programming setting we describe below.

The term annotated rules of the program logic can be described in the three tables which now follow. The first figure 1 gives the inference rules governing sums, products, and exponentiation:

Here are some things to note about the terms defined in figure 1 for this logic which we should view as programs:

(i) The index sets $I$ are all finite with element denoted by $i_1, ..., i_n$.

(ii) The variables "declared" on the left of the sequents must always be distinct but can in general be product patterns. Thus the following is a valid sequent:

$$(\text{fst} : x_1, \text{snd} : x_2) : X_{\text{fst}} \times Y_{\text{snd}} \vdash x_1 : X.$$

This exemplifies the "indexed" product notation. When there is no danger of confusion we will tend to omit the indexing letting the order of components carry this information.

(iii) We follow the standard programming convention of separating the indexes of all products and coproducts so that the names of the indexes uniquely determine the type.

(iv) Products have terms which are indexed records:

$$(\text{fst} : t_1, \text{snd} : t_2) : X_{\text{fst}} \times Y_{\text{snd}}.$$

the projection from this record to the first coordinate is just the index of

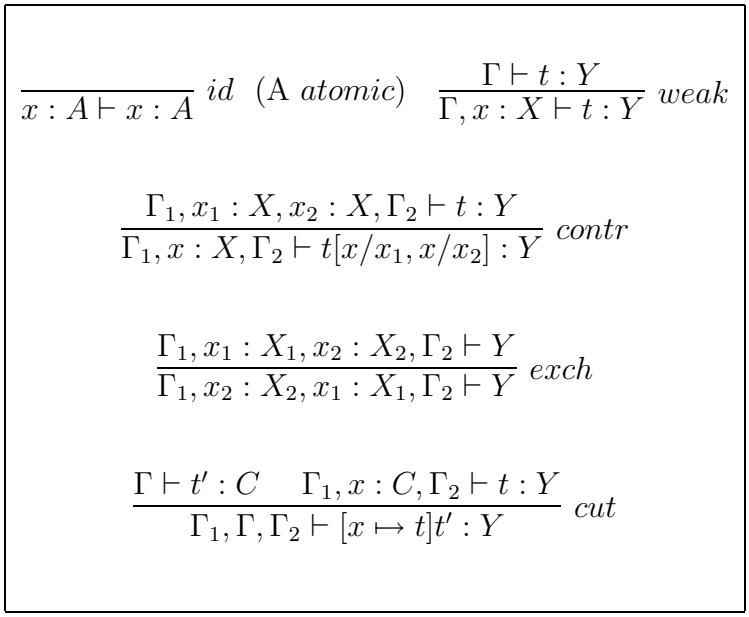


Fig. 2. Structural rules

that coordinate, fst.

(v) Maps from coproducts are expressed through "case" terms:

$$\left\{ \begin{array}{l} \mathrm{rgt}\, x \mapsto t_1 \\ \mathrm{lft}\, y \ \mapsto t_2 \end{array} \right\} : X_{\mathrm{rgt}} + Y_{\mathrm{lft}} \to Z$$

and the coproduct embedding are denoted by the index.

(vi) The logic uses $\lambda$ abstraction with application given by the infix operator @ which, as all operators do, binds more tightly that ordinary composition.

The next figure 2 gives the structural rules and, in particular, the cut rule:

Notice that the first rule introduces identity rules for *atomic types*. It is important to realize that the identity map for non-atomic types must be derived. This restriction is common in logics and greatly simplifies the proof theory. However, from a programming point of view this looks a bit crazy! While it is possible to formulate a logic without the necessity to have "expanded" identity maps it does make program equivalence checking slightly more complex as the identities need to be expanded anyway. Here we shall adopt the standard logical convention because our main interest is in program equivalence rather than program optimization.

Notice that this logic has *explicit* substitution given by the syntax

$$[x \mapsto t]t'$$

which intuitively means that $x$ should be substituted by $t'$ in $t$. Written as a "let expression" this is

$$\texttt{let}\ \ x = t'\ \ \texttt{in}\ \ t.$$

Notice also that the only rule which uses explicit substitution is the cut rule. Thus, for example, I have written the contr(action) rule using a direct substitution: this maintains the correspondence between the use of cut and the occurrences of explicit substitution in the programs.

The juxtaposition of $[x \mapsto t]$ and $t'$ should be regarded as the expression of ordinary composition. Thus, in $[x \mapsto t]f@s$, as operators bind more tightly than composition, the application is calculated before the composition. Those familiar with functional programming and the $\lambda$-calculus may find it curious that we distinguish between application for higher-order terms and application of functions. In this respect, as we want our terms to represent proofs in the logic, we are simply being blindly faithful to the logic which makes this distinction.

Clearly the expression, $[x \mapsto t]f@s$, should be formally equivalent to the substitution $t[(f@s)/x]$ and, in fact, it is. As we shall discover many of the cut-elimination steps which will force the presence of certain proof equivalences, are actually concerned with turning explicit substitution into ordinary substitution and in this sense are rather mundane. However, recall that explicit substitution does permit the "variable" which is to be substituted to be a pattern which gives the cut-elimination steps a little more content.

Now it should be clear that the semantics of the proof theory of this fragment lies in cartesian closed categories which have coproducts. That this fragment satisfies cut-elimination is well-known as it is a variant of intuitionistic logic. Furthermore this fragment has its proof equivalence decidable. In the pure logic (that is with no atomic types and no non-logical axioms) this is actually almost immediate as the initial model is just finite sets. In the general case, where arbitrary types and axioms are permitted, the problem is much harder; but even in this general case the calculus is decidable (these issues are discussed in Ghani's thesis [7] and in recent work of Altenkirch, Dybjer, Hoffman, and Scott [12]).

Finally we come to the rules of the logic which are of primary interest to this discussion. They are the rules which determine the behavior of the inductive and coinductive datatypes. The rules for the term construction are given in figure 3.

These need some explaining: we shall start with the cons(truction) and dest(ruction) rules as they are somewhat simpler. The construction rule allows one to build an inductive datatype from its component: thus, for example, a list can be built from an element and a list, via $\mu\,\mathrm{cons}.$ or (primitively) as a nil list, via $\mu\,\mathrm{nil}$. The application of a $\mu$ rule turns a coproduct term into a list term. This syntax may seem a little peculiar as usually one does not distinguish between a coproduct term and an inductive datatype term. The way we have set up this logic requires that we make the distinction. We will often ignore this requirement in order to make the terms look more familiar. The destruction rule is dual: it allows us to break up a coinductive datatype into its constituents. Notice that destruction must be substituted in the term.

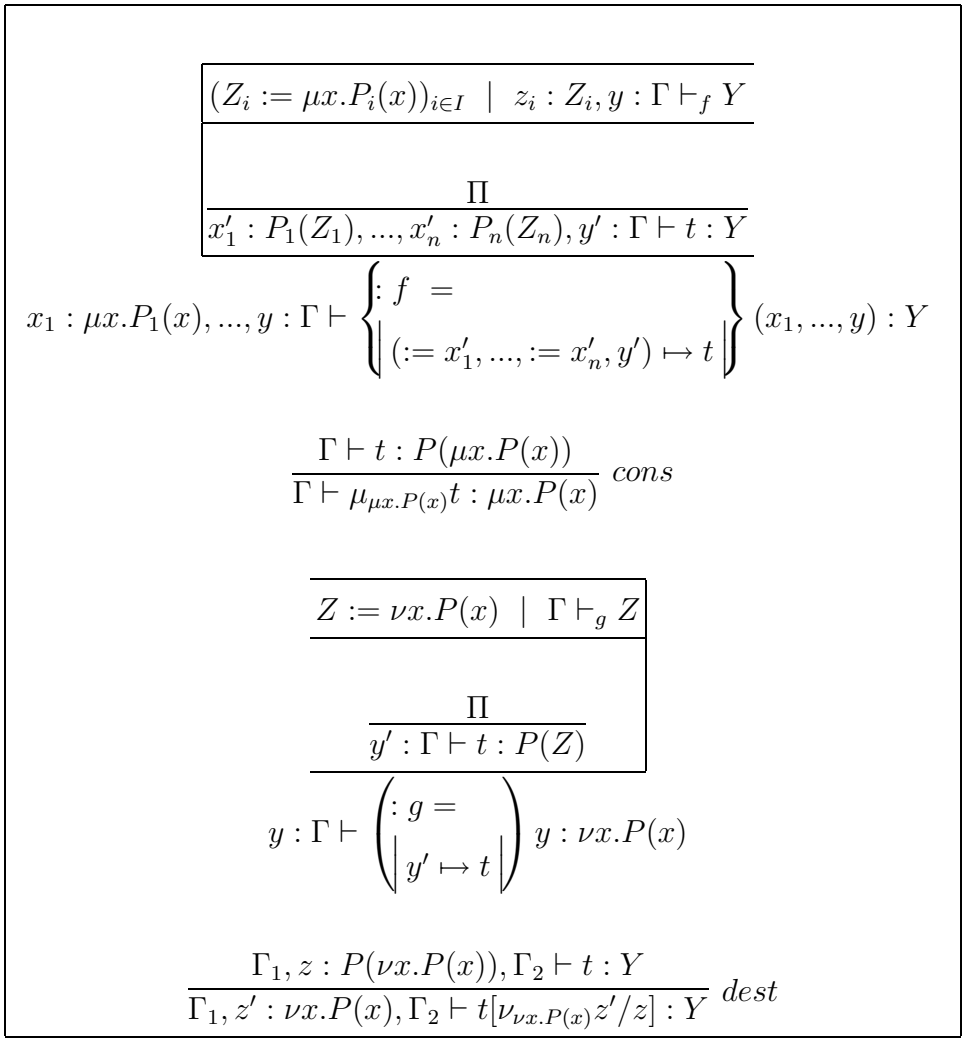


Fig. 3. Circular inductive and coinductive proof rules

It might tempting to use an explicit substitution here but this should be resisted for, as explained above, we are reserving explicit substitution to record the occurrence of the cut rule.

The remaining two rules are the inductive and coinductive **circular proof rules**. The form of these rules needs some discussion as they are not perhaps as familiar: essentially they allow the recursive specification of functions *but* in a tightly controlled manner. Thus these rules are essentially Landin's `letrec` construction, of course, we need two variants and some extra notation besides so that we can track the recursive arguments.

Please note I am following the **charity** tradition of using curly parentheses to denote inductive items and round parentheses to denote coinductive items. We shall name the datatypes: for example the inductive natural number and list datatypes are defined by:

$$\mathrm{Nat} = \mu x.1_{\mathrm{zero}} + x_{\mathrm{succ}}$$

$$x \mapsto \left\{ \begin{array}{l} : \mathrm{rev}_1(:=, y' = \mathrm{nil}) \;\; = \\ \left| \begin{array}{ll} \mathrm{nil}() & \mapsto y' \\ \mathrm{cons}(v, vs) & \mapsto \mathrm{rev}_1(vs, \mu\, \mathrm{cons}(v, y')) \end{array} \right| \end{array} \right\} x$$

Fig. 4. Fast reverse

$$x \mapsto \left\{ \begin{array}{l} : \mathrm{rev}_2 := \;\; = \\ \left| \begin{array}{l} \mathrm{nil}() \mapsto \mu\, \mathrm{nil}() \\ \mathrm{cons}(v, vs) \mapsto \mathrm{app}(\mathrm{rev}_2(vs), \mu\, \mathrm{cons}(v, \mathrm{nil})) \end{array} \right| \end{array} \right\} x.$$

Fig. 5. Naive reverse

$$\mathrm{List}(A) = \mu x.1_{\mathrm{nil}} + (A_{\mathrm{fst}} \times x_{\mathrm{snd}})_{\mathrm{cons}}.$$

The append function in this system becomes:

$$x, y \mapsto \left\{ \begin{array}{l} : \mathrm{app} \;\; = \\ \left| \begin{array}{l} (x' :=, y') \mapsto \left\{ \begin{array}{ll} \mathrm{nil}() & \mapsto y' \\ \mathrm{cons}(v, vs) & \mapsto \mu\, \mathrm{cons}(v, \mathrm{app}(vs, y')) \end{array} \right\} x' \end{array} \right| \end{array} \right\} (x, y)$$

where notice I have simplified the product notation. Notice also that we indicate the regenerated variable with the := symbol. There is another way we shall write this in order to reduce the nesting of parentheses:

$$x, y \mapsto \left\{ \begin{array}{l} : \mathrm{app}(:=, y' = y) \;\; = \\ \left| \begin{array}{ll} \mathrm{nil}() & \mapsto y' \\ \mathrm{cons}(v, vs) & \mapsto \mu\, \mathrm{cons}(v, \mathrm{app}(vs, y')) \end{array} \right| \end{array} \right\} x$$

This brings the notation more closely into line with that used by functional languages and the notation used by charity. Notice how the recursive variable is put outside while the non recursive parameter is assigned where the function is introduced.

Another example of a program which makes a non-trivial use of the the power of these circular definitions is the "fast" reverse function. This is displayed in figure 4: compare this to the higher-order definition see figure 13. The circular (or recursive) part of this function actually reverses a first list onto a fixed second list.

The naive version, see figure 5, of the reverse function uses the append function repeatedly inside an outer recursion.

Notice, in particular, that, despite the fact that I have not indicated them, there are a number of cuts in these terms. For example a circular term like app can only have variables as arguments so these must be supplied by a cut.

$$x, y \mapsto \left\{ \begin{array}{lll} : \mathrm{monus}(:=,:=) & = & \\ (\mathrm{zero}, \_) & \mapsto & \mu\, \mathrm{zero} \\ (m, \mathrm{zero}) & \mapsto & m \\ (\mathrm{succ}(m), \mathrm{succ}(n)) & \mapsto & \mu\, \mathrm{succ}\, \mathrm{monus}(m, n) \end{array} \right\} (x, y).$$

Fig. 6. Monus by simultaneous recursion

The second "naive" version of reverse, as we shall discover in section 4.3.2, is not deforested and can be automatically improved.

In general, we shall not be punctilious about recording cuts as there is a significant notational overhead instead we will feel free to use the substituted (although often not derivable) term. However, we remark that the presence of a cut is often going to be an indication that something can be simplified. Notice also that we will feel free to use names of previously defined circular functions.

The circular proof rule for inductive datatypes also allow for simultaneous recursion. Thus, one can have more than one inductive datatype being "unraveled" at the same time. However, notice that one cannot do this for the coinductive datatypes because there is a fundamental asymmetry in the logic which allows lists of propositions (types) on the left but only one proposition (type) on the right. The ability to do simultaneous recursion is of practical importance as it allows efficient implementations of certain functions. Here is a well-known example: without simultaneous recursion or higher-order function terms it is impossible to provide the efficient implementation of monus shown in figure 6.

Notice that the positions of the recursive arguments in the term logic are indicated by the := type binder so that, in a circular function $f$ which is to be recursively applied, we know precisely which arguments are active. In the proof theory the circular function involves the introduced types $Z_i$ whose scope is delimited by the box. These type variables $Z_i$ are **regenerations** of the underlying recursive types: this information is indicated by the assignment $Z_i := \mu x.P_i(x)$. The regeneration relation is, of course, transitive. A type variable $Z_i'$ which is a regeneration of another type $Z_i$ can be treated exactly as if it were the type $Z_i$. Crucially the converse is not true, thus the circular function cannot be applied to an earlier generation of its type variable.

There is another important way in which a regenerated type variable differs from the datatype from which it originated: we do not allowed the construction (or destruction) rules to be applied to a regenerated variable. This is because such an application would reverse the regeneration process. However, we do allow a regenerated variable to have the circular proof rule of the datatype applied to it. It is thus possible to have **multiple regenerations**, that is for a regenerated variable to be itself regenerated.

To illustrate the power of regeneration consider the problem of producing,

$$x \mapsto \left\{ \begin{array}{l} : \mathrm{odd}(:=) \ = \\ \left| \begin{array}{ll} \mathrm{nil}() & \mapsto \mu\,\mathrm{nil} \\ \mathrm{cons}(v, vs) & \mapsto \mu\,\mathrm{cons}(v, \left\{ \begin{array}{ll} \mathrm{nil}() & \mapsto \mu\,\mathrm{nil}) \\ \mathrm{cons}(\_, L') & \mapsto \mathrm{odd}'(L')) \end{array} \right\} vs) \end{array} \right| \end{array} \right\} x.$$

Fig. 7. The odd function

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{Z := \mathrm{List}(A) \mid Z \vdash_{\mathrm{odd}} \mathrm{List}(A)}}{Z' := Z}}{\dfrac{\dfrac{}{A, Z' \vdash A} \quad \dfrac{Z' \vdash \mathrm{List}(A)}{A, Z' \vdash \mathrm{List}(A)}\ \mathrm{odd}}{A, Z' \vdash A \times \mathrm{List}(A)}}{A, Z' \vdash 1 + A \times \mathrm{List}(A)}\ \mu\,\mathrm{cons}}{A, Z' \vdash \mathrm{List}(A)}}{A, A, Z' \vdash \mathrm{List}(A)}\ weak}{A, A \times Z' \vdash \mathrm{List}(A)}}{\dfrac{}{A, 1 \vdash \mathrm{List}(A)}\ \mu\,\mathrm{nil}} \quad \dfrac{}{A, 1 + A \times Z' \vdash \mathrm{List}(A)}}{A, Z \vdash \mathrm{List}(A)}}{A \times Z \vdash \mathrm{List}(A)} \quad \dfrac{}{1 \vdash \mathrm{List}(A)}\ \mu\,\mathrm{nil}}{1 + A \times Z \vdash \mathrm{List}(A)}}{\mathrm{List}(A) \vdash \mathrm{List}(A)}$$

Fig. 8. Circular proof of odd

from a list, the list of those elements which have odd indexes. The program is displayed in figure 7:

I have used another convention: namely that a circular proof term which never uses its circular function can be represented by simply omitting mention of the introduced function altogether. This almost reduces the syntax to that of the case combinator and, of course, this near confusion of notation is quite deliberate.

The purpose of this example was to illustrate the use of multiple regeneration. While it is not hard to see that we could implement this without a multiple regeneration (see later in figure 14 ), I would claim that the resulting program is much less natural. In figure 8 we give the derivation of this term which shows the pattern of regeneration in the proof.

Notice that first the regenerated type variable $Z$ is treated as if it were

$$IL \mapsto \left( \left| \begin{array}{l} : \mathrm{acc}' : \mathrm{InfL}(A), \mathrm{List}(A) \to \mathrm{List}(A) = \\ (L, vs) \mapsto \left( \begin{array}{l} \mathrm{hd} : vs \\ \mathrm{tl} : \mathrm{acc}'(\mathrm{tl}\,\nu L, \mu\,\mathrm{cons}(\mathrm{hd}\,\nu L, vs)) \end{array} \right) \end{array} \right| \right) (IL, \mu\,\mathrm{nil}).$$

Fig. 9. Simple accumulate

$$\dfrac{\dfrac{}{\vdash \mathrm{List}(A)}\ \mathrm{nil} \qquad \dfrac{\dfrac{}{Z := \mathrm{InfL}(A) \mid \mathrm{InfL}(A), \mathrm{List}(A) \vdash_{\mathrm{acc}'} Z} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{List}(A) \vdash \mathrm{List}(A)}}{\mathrm{InfL}(A), \mathrm{List}(A) \vdash \mathrm{List}(A)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{A, \mathrm{List}(A) \vdash A \times \mathrm{List}(A)}\ cons}{A, \mathrm{List}(A) \vdash \mathrm{List}(A)} \qquad \dfrac{}{\mathrm{InfL}(A), \mathrm{List}(A) \vdash Z}\ \mathrm{acc}'}{A, \mathrm{List}(A), \mathrm{InfL}(A) \vdash Z}\ cut}{A \times \mathrm{InfL}(A), \mathrm{List}(A) \vdash Z}}{\mathrm{InfL}(A), \mathrm{List}(A) \vdash Z}\ dest}{\mathrm{InfL}(A), \mathrm{List}(A) \vdash \mathrm{List}(A) \times Z}}{\mathrm{InfL}(A), \mathrm{List}(A) \vdash \mathrm{InfL}(\mathrm{List}(A))}}{\mathrm{InfL}(A) \vdash \mathrm{InfL}(\mathrm{List}(A))}\ cut$$

Fig. 10. Proof of the simple accumulate function

List($A$) by being the subject of a circular derivation. However, this inner circular derivation does not use it's circular function. Instead the regenerated type variable $Z'$ is treated as if it were $Z$ and has the function odd$'$ applied to it.

Now much the same considerations apply to the coinductive datatypes. We illustrate the coinductive side of this language with a function to "accumulate" an infinite list. This function given an infinite list InfL($A$) produces an infinite list InfL(List($A$)) in which the inner lists give the list of elements seen so far. Now there are two ways to accululate the values see so far: the first, see figure 9 collects the elements in reverse order, the second, see figure 11 collects the elements in order.

The infinite list datatype is defined by

$$\mathrm{InfL}(A) = \nu x.A_{\mathrm{hd}} \times x_{\mathrm{tl}}.$$

Notice that this sort of infinite list always has a "next element" by fiat. It cannot be empty or decide to stop!

This is a simple illustration of how coinductive datatypes and inductive datatypes can be used in combination to produce programs. The derivation of this term is (approximately!) displayed in figure 10.

Notice that this proof does contain a number of cuts which, in the term, I have suppressed. However, these cuts are all soft cuts so that actually we shall regard this term as being already deforested. However, the version of the accumulate function in figure 11 which collects the lists in the right order

$$IL \mapsto \left( \left| \begin{array}{l} : \mathrm{acc}' : \mathrm{InfL}(A), \mathrm{List}(A) \to \mathrm{List}(A) = \\ (L, vs) \mapsto \left( \begin{array}{l} \mathrm{hd} : vs \\ \mathrm{tl} : \mathrm{acc}(\mathrm{tl}\, L, \mathrm{app}(vs, \mathrm{cons}(\mathrm{hd}\, L, \mathrm{nil}))) \end{array} \right) \end{array} \right| \right) (IL, \mathrm{nil}).$$

Fig. 11. Accumulate in order

is not deforested we shall see how this may be deforested in section 4.3.3):

This then concludes our introduction to the formal programming language that we shall use in the sequel. At this stage the reader should be convinced – and it is most certainly the case – that one can write nontrivial programs in this language which is a basic "strong functional programming language" in the sense of David Turner.

## 3 The transformation from initial and final datatypes

These circular proof rules despite according with the recursive programming style do not seem at first sight to correspond to the initial algebra and final coalgebra interpretation of datatypes.

Recall that if $F(A, X)$ is functor then $\mu X.F(A, X)$ equipped with a map

$$\mathrm{cons} : F(A, \mu X.F(A, X)) \to \mu X.F(A, X)$$

is an **initial algebra** for $F(A, \_)$ in case for every $F(A, \_)$-algebra

$$g : F(A, Z) \to Z$$

there is a unique map $h : \mu X.F(A, X) \to Z$ such that the following commutes:

$$\begin{array}{ccc} F(A, \mu X.F(A, X)) & \xrightarrow{\mathrm{cons}} & \mu X.F(A, X) \\ {\scriptstyle F(A,h)}\downarrow & & \downarrow{\scriptstyle h} \\ F(A, Z) & \xrightarrow[g]{} & Z. \end{array}$$

This is the **universal property** of the initial algebra. Dually the couniversal property of the final coalgebra is defined as follows: $\nu X.F(A, X)$ equipped with a map

$$\mathrm{dest} : \nu X.F(A, X) \to F(A, \nu X.F(A, X))$$

is a **final algebra** for $F(A, \_)$ in case for every $F(A, \_)$-coalgebra

$$g' : Z \to F(A, Z)$$

$$\frac{\Gamma_1, z' : P(Y), \Gamma_2 \vdash t : Y}{\Gamma_1, v : \mu x.P(x), \Gamma_2 \vdash \left\{\!\!\left|\ z' \mapsto t\ \right|\!\!\right\} v : Y}\ initial$$

$$\frac{\Gamma \vdash t : P(\mu x.P(x))}{\Gamma \vdash \mathrm{cons}_{\mu x.P(x)}\, t : \mu x.P(x)}\ cons$$

$$\frac{\Gamma, x : Z \vdash t : P(Z)}{\Gamma, v : Z \vdash \left(\!\!\left|\ x \mapsto\ \ t\ \right|\!\!\right) v : \nu x.P(x)}\ final$$

$$\frac{\Gamma_1, z : P(\nu x.P(x)), \Gamma_2 \vdash t : Y}{\Gamma_1, z' : \nu x.P(x), \Gamma_2 \vdash t[\mathrm{dest}_{\nu x.P(x)}\, z'/z] : Y}\ dest$$

Fig. 12. The algebra based proof system

there is a unique map $h' : Z \to \nu X.F(A, X)$ such that the following commutes:

$$\begin{array}{ccc} Z & \xrightarrow{\quad g' \quad} & F(A, Z) \\ {\scriptstyle h'}\downarrow & & \downarrow{\scriptstyle F(A,h')} \\ \nu X.F(A, X) & \xrightarrow[\mathrm{dest}]{} & F(A, \nu X.F(A, X)). \end{array}$$

These conditions assert not only the existence of a comparison map but, importantly, the uniqueness that map. The uniqueness of the comparison map forces certain natural equalities on the programs of the system.

This algebra based approach to datatypes can also be translated into annotated inference rules see figure 12. When one writes down these rules one quickly realizes that one needs the initial property "in context" as, to maintain the style of the logic, one must allow other propositions (types) beside the initial one on the left-hand side of the sequent. This does not to relate directly to the property outlined above for the inductive datatypes, however, in the cartesian closed setting at least, the above diagrams do suffice as one can shunt the context onto the right-hand side. These matters are described in great categorical detail in [4] and in Bart Jacobs's book on categorical logic [8].

This is essentially the **charity** syntax as described in [5] and that paper provides several examples of programs.

The main purpose of this section is to show that it is possible to translate

between these two styles of proof.[5] We shall show this through the sequent proofs but it is important to realize that this works must work for the proof terms too and that this does in fact gives a method of translating between the two styles of programming.

We shall start by showing that the algebra based proof system (which in the model checking community is sometimes referred to as the Kozen system [9]) can be simulated by the circular proof system. To this end we need first to show how the initial algebra derivation can be obtained in the circular proof system. The required derivation has the following form:

$$\frac{\begin{array}{c} Z := \mu x.P(x) \mid \Gamma_1 Z \Gamma_2 \vdash_f X \\ \hline \dfrac{\dfrac{}{\Gamma_1, Z, \Gamma_2 \vdash X}\, f}{\Gamma_1, P(Z), \Gamma_2 \vdash P(X)}\, functor \quad \dfrac{\Pi}{\Gamma_1, P(X), \Gamma_2 \vdash X}\, assumption \\ \hline \Gamma_1, P(Z), \Gamma_2 \vdash X \end{array}}{\Gamma_1, \mu x.P(x), \Gamma_2 \vdash X}\, cut$$

The inference labeled "functor" has to be inductively established for all the possible functors $P$. It corresponds to a well-known categorical observation that all the functors involved are "strong" [4].

Next we need to show how the final coalgebra derivation can be simulated in the circular proof system. This is very similar to the above and again uses the strength of the functor $P$:

$$\frac{\begin{array}{c} Z := \nu x.P(x) \mid \Gamma_1, Y, \Gamma_2 \vdash_h Z \\ \hline \dfrac{\dfrac{\Pi}{\Gamma_1, Y, \Gamma_2 \vdash P(Y)}\, assumption \quad \dfrac{\dfrac{}{\Gamma_1, Y, \Gamma_2 \vdash Z}\, h}{\Gamma_1, P(Y), \Gamma_2 \vdash P(Z)}\, functor}{\Gamma_1, Y, \Gamma_2 \vdash P(Z)}\, cut \end{array}}{\Gamma_1, Y, \Gamma_2 \vdash \nu x.P(x)}$$

This then shows that the circular proof system can simulate all the derivations of the algebra based proof system.

We now have to show that we can simulate all the proofs of the circular proof system in the algebra based proof system. This is a little more difficult as we also have to handle the possibility of multiple regenerations and simultaneous recursion. We shall therefore approach this in three stages. First we

[5] Please note the ideas behind this translation are not original. In particular, I am very much in debt to Santocanale for introducing me to his proof [11] of this for the finitely bicomplete poset case with initial and final fixed points: I am unashamedly borrowing his ideas.

will show that when the recursion is sequential and there are no multiple regenerations that we can easily translate the proof into the algebra based proof system. Next we will show that a circular proof with simultaneous recursion can be reduced (within the circular proof system) to one with only sequential recursion. Finally, we show how a proof with multiple regenerations can be reduced to one which has only single regenerations.

Consider a circular proof for a program from an inductive type which has no regenerations. Letting $\pi$ stand for the derivation from the circular assumption this has the general form:

$$\dfrac{\boxed{\begin{array}{c} Z := \mu x.P(x) \mid \Gamma, Z \vdash X \\ \hline \dfrac{\Gamma, Z \vdash X}{\Gamma, P(Z) \vdash X}\ \pi(Z) \end{array}}}{\Gamma, \mu x.P(x) \vdash X}$$

Now there are actually many proofs into which we could transform this but we actually have to be somewhat careful. For example, the following might seem like a reasonable translation:

$$\dfrac{\dfrac{\dfrac{X \vdash X}{\Gamma, X, \Gamma \vdash X}\ weak}{\Gamma, P(X) \vdash X}\ \pi(X)}{\Gamma, \mu x.P(x) \vdash X}\ initial$$

where we heavily use the fact, which is not totally obvious, that the proof $\pi(Z)$ is parametric in $Z$. The reason this works is because the only rule which can involve using the implicit type of $Z$ is the circular induction rules for the type. However, a use of that rule would have meant that there was a multiple regeneration. Thus we can substitute an arbitrary $X$ for $Z$ in the proof $\pi(Z)$ to obtain a proof involving $X$.

However, consider what this transformation does to the heart of the fast reverse introduced in figure 4: it becomes

$$(x, y) \mapsto \left\{ \begin{array}{ll} \text{nil} : () & \mapsto y \\ \text{cons} : (\_, vs) & \mapsto vs \end{array} \right\} x$$

which is equivalent to $(x, y) \mapsto y$. Not at all what we want!

The problem with this approach to the translation is that the non-recursive arguments are altered prior to being used by the $\text{rev}_1$ function. By weakening we, in effect, throw away these modifications.

In order to obtain a correct translation it is necessary to make essential use of the higher-order aspects of the language. It is convenient to assume that

$$(x,y) \mapsto \left\{ \begin{array}{ll} \text{nil} : () & \mapsto \lambda x.x \\ \text{cons} : (a,f) & \mapsto \lambda x.f@(\text{cons}(a,x)) \end{array} \right\} x)@y.$$

Fig. 13. Higher-order fast reverse

$\Gamma$ is a singleton: this we can do without loss because if it is not then we may always form the product of the propositions to obtain an equivalent singleton. Here then is the desired translation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, \Gamma \Rightarrow X \vdash X}{\Gamma, P(\Gamma \Rightarrow X) \vdash X}\,\pi(\Gamma \Rightarrow X)}{P(\Gamma \Rightarrow X) \vdash \Gamma \Rightarrow X}}{\mu x.P(x) \vdash \Gamma \Rightarrow X}\,initial}{\Gamma, \mu x.P(x) \vdash X}$$

which has the following effect (see figure 13) on the fast reverse introduced in figure 4: Notice that this does have the correct effect!

Similarly consider a circular proof for a program to a coinductive type which has no regenerations (where we assume $\Gamma$ is a singleton):

$$\dfrac{\begin{array}{|c|}\hline Z := \nu x.P(x) \mid \Gamma, Y \vdash Z \\ \hline \dfrac{\Gamma, Y \vdash Z}{\Gamma, Y \vdash P(Z)}\,\pi(Z) \\ \hline \end{array}}{\Gamma, Y \vdash \nu x.P(x)}$$

this transforms to:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \Gamma \quad Y \vdash Y}{\Gamma, Y \vdash \Gamma \times Y}\,weak}{\Gamma, Y \vdash P(\Gamma \times Y)}\,\pi(Y)}{\Gamma, Y \vdash \nu x.P(x)}\,final.$$

Notice that for the coinductive transformations we do not need the any higher-order constructs.

Next we will illustrate how to remove simultaneous recursion within the circular proof system. This transformation uses the higher-order aspects of the language non-trivially: the transformation is well-known and was pointed out to me by both Eric Meijer and Simon Thompson. I shall illustrate the technique by transforming a proof which has a simultaneous recursion on two arguments. Here is a typical such proof:

$$\dfrac{\dfrac{}{Z_1 := \mu x.P_1(x), Z_2 := \mu x.P_2(x) \mid Z_1, Z_2, \Gamma \vdash C}}{\dfrac{\dfrac{Z_1, Z_2, \Gamma \vdash C}{P_1(Z_1), P_2(Z_2), \Gamma \vdash C}\ \pi}{\mu x.P_1(x), \mu x.P_2(x), \Gamma \vdash C}}$$

This gets transformed to the following proof with this simultaneous recursion removed:

$$\dfrac{\dfrac{\dfrac{}{Z_2 := \mu x.P_2(x) \mid Z_2, \Gamma \vdash \mu x.P_1(x) \Rightarrow C}}{\dfrac{\dfrac{\dfrac{\dfrac{}{Z_1 := \mu x.P_1(x) \mid Z_1, P_2(Z_2), \Gamma \vdash C}}{\dfrac{\dfrac{Z_2, \Gamma \vdash Z_1 \Rightarrow C}{Z_1, Z_2, \Gamma \vdash C}}{P_1(Z_1), P_2(Z_2), \Gamma \vdash C}\ \pi}{\mu x.P_1(x), P(Z_2), \Gamma \vdash C}}{P(Z_2), \Gamma \vdash \mu x.P_1(x) \Rightarrow C}}{\mu x.P_2(x), \Gamma \vdash \mu x.P_1(x) \Rightarrow C}}{\mu x.P_1(x), \mu x.P_2(x), \Gamma \vdash C}$$

Here notice that we use the fact that $Z_1$ is a regenerated variable so may be treated as one of its earlier incarnations namely $\mu x.P_1(x)$ so that we can discharge the inner induction using the outer premise. Notice that in this proof the inner circular induction is essentially a case combinator as its circular function is never used.

It remains to remove multiple regenerations. Our strategy is, as for simultaneous recursion, to show that multiple regenerations can be removed within the circular proof system itself. Suppose to start with that we have a multiple regeneration on an inductive type, we would then have the following form to

the proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{Z := \mu x.P(x) \mid \Gamma, Z \vdash Y}}{\dfrac{\dfrac{\dfrac{}{Z' := Z \mid \Gamma', Z' \vdash Y'}}{\dfrac{\Gamma, Z' \vdash Y \quad \Gamma', Z' \vdash Y'}{\Gamma', P(Z') \vdash Y'}\,\pi_2(Z')}}{\dfrac{\Gamma', Z \vdash Y'}{\Gamma, Z \vdash Y}\,\pi_1(Z)}}}{\Gamma, \mu x.P(x) \vdash Y}}$$

where the proofs $\pi_1$ and $\pi_2$ could both have other regenerations of $Z$ and (for $\pi_2$, $Z'$).

In order to simplify the transformed proof it is convenient to assume that $\Gamma$ and $\Gamma'$ are singletons. Please note that the transformed proof has many implicit cuts (e.g. in $\pi_1'$, $\pi_2'$, $\pi_1''$, and $\pi_2''$). Also please note that again we have made essential use of the fact that the language is higher-order.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{Z := \mu x.P(x) \mid Z \vdash (\Gamma \Rightarrow Y) \times (\Gamma' \Rightarrow Y')}}{\dfrac{\dfrac{Z \vdash (\Gamma \Rightarrow Y) \times (\Gamma' \Rightarrow Y')}{Z \vdash \Gamma' \Rightarrow Y}\,\pi_1(Z)}{\Gamma', Z \vdash Y'}\,\pi_2'}{P(Z) \vdash \Gamma \Rightarrow Y} \qquad \dfrac{\dfrac{\dfrac{Z \vdash (\Gamma \Rightarrow Y) \times (\Gamma' \Rightarrow Y')}{\Gamma', Z \vdash Y'}\,\pi_2'' \qquad \dfrac{Z \vdash (\Gamma \Rightarrow Y) \times (\Gamma' \Rightarrow Y')}{\Gamma, Z \vdash Y}\,\pi_2(Z)}{\Gamma', P(Z) \vdash Y'}}{P(Z) \vdash \Gamma' \Rightarrow Y'}}{P(Z) \vdash (\Gamma \Rightarrow Y) \times (\Gamma' \Rightarrow Y')}\,\pi_1''}{\dfrac{\mu x.P(x) \vdash (\Gamma \Rightarrow Y) \times (\Gamma' \Rightarrow Y')}{\mu x.P(x) \vdash \Gamma \Rightarrow Y}\,\pi_1'}}{\Gamma, \mu x.P(x) \vdash Y}$$

We remark that the proofs $\pi_1$ and $\pi_2$ are not parametric in the type variable $Z$ as it is possible that these proofs contain another regeneration on these variables. However, what is certainly true is that we may substitute (any regenerated variable of) the same inductive type into these proofs.

To illustrate this transformation process consider the function odd in figure 7: it has multiple regenerations, thus it is reasonable to ask what the program looks like when it is transformed into the algebra based system. In figure 14 the translated version written in **charity** using a fold is dsiplayed. Notice the rather unexpected second component of the state of the fold: it is a higher-order function. Essentially this component lags one step behind the first component, so that it holds the even list waiting for the next element to

$$x \quad \mapsto \left\{ \begin{array}{ll} \text{nil} : () & \mapsto (\text{nil}(), \lambda a.[a])) \\ \text{cons} : (v, (L, g)) & \mapsto (g@v, (\lambda a.\, \text{cons}(a, L))) \end{array} \right\} x$$
$$; (V, \_) \mapsto V.$$

Fig. 14. Higher-order odd

be added. When the next element is added it becomes (by application to that element) the odd list again.

Of course this may not be a very efficient way to compute the list of odd elements; but this really is not the point. The question is only whether this is an equivalent expression of the program (which I claim is the case). Clearly the direct approach suggested by the earlier code for odd will be more efficient.

To complete the transformation we need to also show that multiple regenerations can be removed from coinductive circular proofs. It turns out that the proof of this is much easier: we do not even need to use the closedness of the setting. Here is the general form of a coinductive regeneration in which the proofs $\pi_1$ and $\pi_2$ can themselves contain regenerations:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{Z := \nu x.P(x) \mid \Gamma \vdash Z}}{\cfrac{\cfrac{}{Z' := Z \mid \Gamma' \vdash Z'}}{\cfrac{\Gamma \vdash Z' \quad \Gamma' \vdash Z'}{\Gamma' \vdash P(Z')}\,\pi_2(Z')}}}{\cfrac{\Gamma' \vdash Z}{\Gamma \vdash P(Z)}\,\pi_1(Z)}}{\Gamma \vdash \nu x.P(x)}$$

Again to simplify the transformation it is convenient to assume that both $\Gamma$ and $\Gamma'$ are singletons. Under this assumption we can remove one level of regeneration by transforming this to the following:

$$\boxed{\begin{array}{c} Z := \nu x.P(x) \mid \Gamma + \Gamma' \vdash Z \\ \hline \\ \dfrac{\dfrac{\dfrac{\Gamma + \Gamma' \vdash Z}{\Gamma' \vdash Z}}{\Gamma \vdash P(Z)}\,\pi_1(Z) \qquad \dfrac{\dfrac{\Gamma + \Gamma' \vdash Z}{\Gamma \vdash Z} \quad \dfrac{\Gamma + \Gamma' \vdash Z}{\Gamma' \vdash Z}}{\Gamma' \vdash P(Z)}\,\pi_2(Z)}{\Gamma + \Gamma' \vdash P(Z)} \end{array}}$$

$$\frac{\Gamma + \Gamma' \vdash \nu x.P(x)}{\Gamma \vdash \nu x.P(x)}$$

We have now almost established the following:

**Theorem 3.1** *There is an isomorphism between the algebra based proof system and the circular proof based system for the program logic.*

The respect in which we have not established this theorem is actually rather important: it concerns proof equivalence. We would like it to be the case that if we translate back and forth that the resulting proof (which is different if there is recursion) is *equivalent* to the starting proof. Furthermore, we would like that, for both the translations, that equivalent proofs are translated into equivalent proofs.

Of course, all we have really established (despite suggesting otherwise) is that the two systems have the same "derivability" power. Categorically speaking we are trying to establish an isomorphism between two different presentations of a category. So far we have shown that with respect to the posetal collapse the two systems are isomorphic. However, we really do want the stronger result and I do claim it is true.

For proofs with no regenerations establishing the equivalence of the double translated proof is quite easy. Proofs with multiple regenerations or simultaneous recursion are translated internally to this special case, thus, so long as the internal translation is between equivalent proofs and the translations themselves maintain equivalence, the result will follow.

Here we do need to know that equivalence is maintained by both translations and this is has lots of details. Furthermore, to establish this we really must begin to explore the notion of proof equivalence in the circular proof systems. The algebra based proof system derives its notion of proof equivalence directly from its underlying categorical semantics: this is discussed at great length in [5]. On the other hand, it is not so clear where the natural notion of proof equivalence comes from in the circular system: the answer to this question, I claim, becomes clearer from examining how the cut-elimination procedure works for that system.

# 4 Program equivalence

A good rule of thumb when trying to understand proof equivalence in a logical system is to look at its cut-elimination procedure – if it has one. This will usually employ the more important identities in the system.

In this section we consider what cut-elimination may mean for the circular proof system. There is an obvious answer which is rather unsatisfactory as it involves infinite proof terms. However, it does suggest what the proof equivalences for the circular proof system should be.

The fact that we produce infinite term from the cut-elimination procedure is not really acceptable from a program transformation point of view. Thus, we turn to deforestation which is a process which aims at removing the cuts which remove structure. We informally introduce a technique for deforesting. This differs from that described in [10] in how it handles coinductive circular definitions. We give several examples of deforestation.

However, before, I do any of that I must first say an embarrassed word or two about the underlying proof system which corresponds to cartesian closed categories with coproducts.

## *4.1 Equivalence in intuitionistic logic*

Despite the fact that the logic of $(\wedge, \vee, \Rightarrow)$ has been heavily studied since the turn of the century the question of providing a decision procedure for proof equivalence is *still* a thorny issue. Any approach which uses rewriting is considerably complicated by the presence of fairly complex "commuting conversions" or equations which makes providing a complete proof that the system works a daunting and highly technical task (see [7]). All the evidence, however, is that this fragment is highly decidable (see as well [12] for an alternate semantic approach). Unfortunately, there is still an outstanding technical problem which I must acknowledge: there is still no feasible and fully general decision procedure in the literature for this fragment of my formal programming system.

I am not going to try to correct this situation here as this part of the system is not of primary interest in this exposition. I will therefore restrict myself to a brief discussion of some of the proof equivalences which arise.

For this fragment we have the usual cut-elimination steps which give the following (representative) rewrites.

$$\frac{\left\{\dfrac{\Gamma \vdash C}{X_i, \Gamma \vdash C}\ weak\right\}_{i\in I}}{\sum_{i\in I} X_i, \Gamma \vdash C} = \frac{\Gamma \vdash C}{\sum_{i\in I} X_i, \Gamma \vdash C}$$

$$\{i(x_i) \mapsto t\}_{i\in I}\, s = t$$

Fig. 15. Idempotence

$$\frac{\left\{\dfrac{\{X_i, Y_j, \Gamma \vdash C\}_{j\in J}}{X_i, \sum_{j\in J} Y_j, \Gamma \vdash C}\right\}_{i\in I}}{\sum_{i\in I} X_i, \sum_{j\in J} Y_j, \Gamma \vdash C} = \frac{\left\{\dfrac{\{X_i, Y_j, \Gamma \vdash C\}_{i\in I}}{\sum_{i\in I} X_i, Y_j, \Gamma \vdash C}\right\}_{i\in I}}{\sum_{i\in I} X_i, \sum_{j\in J} Y_j, \Gamma \vdash C}$$

$$\{i(x_i) \mapsto \{j(y_j) \mapsto t_{i,j}\}\, t_2\}\, t_1 = \{j(y_j) \mapsto \{i(x_i) \mapsto t_{i,j}\}\, t_1\}\, t_2$$

Fig. 16. Transposition

$$\frac{\left\{\dfrac{}{\Gamma, \sum X_i, X_j \vdash Y}\ \pi_j\right\}_{j\in I}}{\dfrac{\Gamma, \sum X_i, \sum X_i \vdash Y}{\Gamma, \sum X_i \vdash Y}\ contr} = \left\{\frac{\dfrac{X_j \vdash \sum X_i \quad \dfrac{}{\Gamma, \sum X_i, X_j \vdash Y}\ \pi_j}{\Gamma, X_j, X_j \vdash Y}\ cut}{\Gamma, X_j \vdash Y}\ contr\right\}_{j\in I}$$

$$\left\{\, i(x_i) \mapsto t_i(x) \,\right\} x = \left\{\, i(x_i) \mapsto [x \mapsto t_i(x)] i(x_i) \,\right\} x$$

Fig. 17. Repetition

$$\begin{aligned}
[z \mapsto \{i(x_i) \mapsto t_i\}_{i\in I}\, z] j(s) &\Longrightarrow [x_j \mapsto t_j] s \\
[(i_1 : x_1, ...) \mapsto t](i_1 : s_1, ...) &\Longrightarrow [x_1 \mapsto ....[x_n \mapsto t] s_n ...] s_1 \\
[z \mapsto t]\, \{i(x_i) \mapsto t_i\}_{i\in I}\, y &\Longrightarrow \{i(x_i) \mapsto [z \mapsto t] t_i\}_{i\in I}\, y \\
[z \mapsto t](i_1 : s_1, ..., i_n : s_n) &\Longrightarrow (i_1 : [z \mapsto t] s_1, ..., i_n : [z \mapsto t] s_n) \\
[f \mapsto s_1[f@s_2/x]] \lambda y.t &\Longrightarrow [x \mapsto s_1]([y \mapsto t] s_2) \\
[_ \mapsto t_1] t_2 &\Longrightarrow t_1 \\
[x \mapsto x] t &\Longrightarrow t
\end{aligned}$$

All except the first two rule we regard as "soft" cuts. Notice that they both remove structure.

There are in this system a number of additional proof equalities which arise as natural proof identifications. Below are two simple examples which are illustrated in figures 15 and 16.

However, there are other more subtle identities which arises through contraction, see for example figure 17. Notice that in this repetition identity a cut appears from nowhere. This is necessary if one wishes to push contractions *up* the proof tree. The cut reflects the fact that a choice in $x$ has been resolved

and this is recorded by adding a cut.

I worked on this fragment with Guangwu Xu (who was my student at that time) as I had a long standing conjecture that these (and some identities I have not mentioned) can be oriented to make a rewrite system modulo the two equations above. We almost completed (and may yet!) a proof of this for the first-order fragment: the termination of our system remained an outstanding issue. The details are quite daunting, however, to make significant progress in program transformation ultimately these problems must be completely resolved.

### *4.2 Circular proof equivalences*

The rules governing the datatypes in the algebra based system are determined by the underlying categorical initial algebra and final algebra semantics: they were described in [5]. Therefore, we will now focus on the rules of the circular proof system.

The first and most obvious cuts involving the datatypes are those which are concerned with the construction and destruction. Both these cuts remove structure and so must be eliminated in any deforesting transformation. They are what we shall call **deforesting cuts**. A term with residual deforesting cuts will, by definition, not be deforested.

We will deal with the inductive case first:

$$\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash P(\mu x.P(x))}\,\pi'}{\Gamma' \vdash \mu x.P(x)}\,cons \qquad \dfrac{\boxed{Z := \mu x.P(x) \mid \Gamma, Z \vdash Y}}{\dfrac{\Gamma, Z \vdash Y}{\Gamma, P(Z) \vdash Y}\,\pi}}{\Gamma, \Gamma' \vdash Y} \quad \Rightarrow \quad \dfrac{\dfrac{}{\Gamma \vdash P(\mu x.P(x))}\,\pi' \qquad \dfrac{\boxed{Z := \mu x.P(x) \vdash Y \mid \Gamma', Z \vdash Y}}{\dfrac{\Gamma, Z \vdash Y}{\Gamma, P(Z) \vdash Y}\,\pi}}{\Gamma, \Gamma' \vdash Y}\,\pi$$

in the term calculus this is the following rewrite:

$$\left[v \mapsto \left\{\!\!\begin{array}{l} f\ = \\ (z :=, y) \mapsto t \end{array}\!\!\right\} (v, u)\right] \mu(s) \quad \Rightarrow \quad \left[z \mapsto t \left[u/y, \left\{\!\!\begin{array}{l} f\ = \\ (z :=, y) \mapsto t \end{array}\!\!\right\} /f\right]\right] s$$

It may seem that no real progress has been made, but recall the the proof $\pi$ has been hauled out and this will (most likely) allow the cut to move up. Notice also how the circular proof term is substituted into its body.

The cut elimination rule for the destruction of a coinductive type is similar:

$$\dfrac{\dfrac{\boxed{Z := \nu x.P(x) \mid \Gamma \vdash Z}}{\dfrac{\Gamma \vdash Z}{\Gamma \vdash P(Z)}\,\pi}}{\Gamma \vdash \nu x.P(x)} \qquad \dfrac{\dfrac{}{\Gamma', P(\nu x.P(x)) \vdash C}\,\pi'}{\Gamma', \nu x.P(x) \vdash C}\,dest}{\Gamma', \Gamma \vdash C}\,cut \quad \Rightarrow \quad \dfrac{\dfrac{\boxed{Z := \nu x.P(x) \mid \Gamma \vdash Z}}{\dfrac{\Gamma \vdash Z}{\Gamma \vdash P(Z)}\,\pi}}{\Gamma \vdash \nu x.P(x)} \qquad \dfrac{}{\Gamma \vdash P(\nu x.P(x))}\,\pi \qquad \dfrac{}{\Gamma', P(\nu x.P(x)) \vdash C}\,\pi'}{\Gamma', \Gamma \vdash C}\,cut$$

in the term calculus this is:

$$[z \mapsto t[\nu(z)/z']](\left(\begin{array}{l}: f = \\ \mid\ y \mapsto s\ \mid\end{array}\right) u) \Rightarrow [z' \mapsto t]s[u/y, \left(\begin{array}{l}: f = \\ \mid\ y \mapsto s\ \mid\end{array}\right) /f]$$

These rules cause the circular proofs to unroll – or to "unfold" in the terminology of Burstall and Darlington. Now, in fact, if we are to remove any cut from the conclusion of a circular proof then it is necessary to unroll the recursion to expose its inner structure. To effect this unrolling it is necessary to introduce a degenerate circular proof which does not use its circular function. We may then move the cut inside such a step.

There are three ways this can happen: two inductive cases and one coinductive. For the inductive cases the cut formula can be on the left or the right. The case where the cut formula actually is the inductive type itself was handled above.

Here is the cut elimination step when the cut formula is on the left:

$$\dfrac{\dfrac{}{\Gamma' \vdash C}\ \pi' \qquad \boxed{\dfrac{}{Z := \mu x.P(x) \mid C, \Gamma, Z \vdash D}\ \dfrac{C, \Gamma, Z \vdash D}{C, \Gamma, P(Z) \vdash D}\ \pi}}{\Gamma', \Gamma, \mu x.P(x) \vdash D}\ cut \Rightarrow \boxed{\dfrac{\dfrac{}{Z := \mu x.P(x) \mid C, \Gamma, Z \vdash D}\ \dfrac{C, \Gamma, Z \vdash D}{C, \Gamma, P(Z) \vdash D}\ \pi}{C, \Gamma, \mu x.P(x) \vdash D} \qquad \dfrac{\dfrac{}{\Gamma' \vdash C}\ \pi' \qquad \dfrac{C, \Gamma, \mu x.P(x) \vdash D}{C, \Gamma, P(\mu x.P(x)) \vdash D}\ \pi}{\Gamma', \Gamma, P(\mu x.P(x)) \vdash D}\ cut}$$

where the open circular proof box indicates that the circular function is not used. Here is the corresponding term.

$$[x \mapsto \left\{\begin{array}{l}: f\ = \\ \mid\ (y :=, y') \mapsto t\ \mid\end{array}\right\} (v, x)]s \Rightarrow \left\{\!\!\left\{ y \mapsto \left[ x \mapsto t[x/y', \left\{\begin{array}{l}: f\ = \\ \mid\ (y :=, y') \mapsto t\ \mid\end{array}\right\} /f] \right] s \right\}\!\!\right\} v$$

Here is the cut step when the cut formula is on the right.

$$\dfrac{\boxed{\dfrac{}{Z := \mu x.P(x) \mid \Gamma, Z \vdash C}\ \dfrac{\Gamma, Z \vdash C}{\Gamma, P(Z) \vdash C}\ \pi}}{\Gamma, \mu x.P(x) \vdash C} \qquad \dfrac{}{C, \Gamma' \vdash D}\ \pi'}{\Gamma, \mu x.P(x), \Gamma' \vdash D}\ cut \Rightarrow \boxed{\dfrac{\dfrac{\dfrac{}{Z := \mu x.P(x) \mid \Gamma, Z \vdash C}\ \dfrac{\Gamma, Z \vdash C}{\Gamma, P(Z) \vdash C}\ \pi}{\Gamma, \mu x.P(x) \vdash C}\ \pi}{\Gamma, P(\mu x.P(x)) \vdash C} \qquad \dfrac{}{\Gamma' \vdash C}\ \pi'}{\Gamma, P(\mu x.P(x)), \Gamma' \vdash D}\ cut}$$

In the coinductive case the cut formula *must* be on the left:

$$\dfrac{\dfrac{\overline{\Gamma' \vdash C}\ \pi' \qquad \dfrac{\dfrac{\overline{Z := \nu x.P(x) \mid C, \Gamma \vdash Z}}{C, \Gamma \vdash Z}}{C, \Gamma \vdash P(Z)}\ \pi}{C, \Gamma \vdash \nu x.P(x)}}{\Gamma', \Gamma \vdash \nu x.P(x)}\ cut \Rightarrow \dfrac{\dfrac{\overline{\Gamma' \vdash C}\ \pi' \qquad \dfrac{\dfrac{\overline{Z := \nu x.P(x) \mid C, \Gamma \vdash Z}}{C, \Gamma \vdash Z}}{C, \Gamma \vdash P(Z)}\ \pi}{C, \Gamma \vdash P(\nu x.P(x))}\ \pi}{\Gamma', \Gamma \vdash P(\nu x.P(x))}\ cut}{\Gamma', \Gamma \vdash \nu x.P(x)}$$

Notice that as the cut of a proof $\pi$ with an identity proof is always equivalent to the original proof, and as we may precipitate unrolling with such a cut, we must conclude that the unrolled proof (to any depth) is equivalent to the original proof.

These are, in fact, the cut elimination steps. It is clear, however, that these steps do not terminate. In fact, in general they will precipitate an infinite unrolling of a circular proof.

In fact, it is reasonable to view cut-elimination in this system as producing infinite terms. There is an obvious "lazy" way in which one can regard this: for any demanded finite depth one can cut-eliminate to that depth. This idea was actually used, for example, by Simon Marlow [10] as the basis for a practical program transformation system. His first step was to lazily unroll the term performing cut elimination from the root upwards. As he wanted to produce a finite program this was followed up by a second stage he called **knot tying**, which was essentially Burstall and Darlington's notion of folding. In this stage he searched for recurrences within the tree and tried to tie the lazy infinite program back into a finite program.

The difficulty with the idea of knot tying was to secure the termination of the search for knots. That is a guarantee that every path in the term will either be finite or go round a knot. While some special conditions are known it seems that in general the general recursive case there can be no guarantee that one can actually complete the tying of the knot.

We may explain the idea behind knot tying proof theoretically: the behavior of a circular proof is determined by the code it produces from its innards when it is unrolled. Now if another proof can simulate the production of the same innards when we unroll it then the two proofs must be equal. If one is transforming one ties the knot by replacing the second proof by the first which is more canonical.

This scheme for inductive proofs is shown in figure 18. Notice that, in order that the proof be applicable to a regenerated variable there can be no construction used on that type in $\Pi$. The proof $\Pi$ need not start with a circular step. The term is shown in figure 19.

There is a similar inference for the coinductive datatypes whose equation

$$infer\Gamma, \mu x.P(x) \vdash Y \quad \Pi \quad = \quad \dfrac{\dfrac{\dfrac{\Pi}{\Gamma, \mu x.P(x) \vdash Y}}{\Gamma, P(\mu x.P(x)) \vdash Y}\ \pi}{\Gamma, \mu x.P(x) \vdash Y}$$

---

$$\dfrac{\Pi}{\Gamma, \mu x.P(x) \vdash Y} = \dfrac{\dfrac{Z := \mu x.P(x) \mid \Gamma, Z \vdash Y}{\dfrac{\Gamma, Z \vdash Y}{\Gamma, P(Z) \vdash Y}\ \pi}}{\Gamma, \mu x.P(x) \vdash Y}$$

Fig. 18. Inductive knot

$$t(u,w) = \left\{\!\!\left| x \mapsto s[w/y, [(u,w) \mapsto t]/f] \right|\!\!\right\} u$$

---

$$t(u,w) = \left\{ \begin{array}{l} : f \ = \\ \left| (x :=, y) \mapsto s \right| \end{array} \right\} (u,w)$$

Fig. 19. Inductive knot equation

$$t(u,w) = \left(\!\!\left| (x,y) \mapsto s[[(u,w) \mapsto t]/f] \right|\!\!\right) (u,w)$$

---

$$t(u,w) = \left( \begin{array}{l} : f = \\ \left| (x,y) \mapsto s \right| \end{array} \right) (u,w)$$

Fig. 20. Coinductive knot equation

is given in figure 20.

These circular inferences are the source of equivalences in this proof system. Once one realizes that cutting with a constructor causes unrolling, it becomes clear that these are just a reformulation of the initial algebra properties. The significance of the transformation lies in the fact that it is a much more convenient form to with which to work.

These remarks suggest in a little more detail why 3.1 is true.

### *4.3 Deforestation*

In this section I shall demonstrate some program equivalences using a technique to deforest the terms of this programming language. This technique of deforesting relies on determining where there is **demand** in the term. A term with no demand is deforested.

There are two ways that demand can occur in a term. The first source of demand is expressed at an argument of a term. This can either be at the active arguments of an inductive circular definition (including a coproduct) or at the argument of a destructor (including projection). This is one reason why we carefully labeled the active (often recursive) arguments of the inductive circular definitions using :=. However, these two expressions of demand only become *real* demand when they can potentially be put together with a supplier. For an active argument of an inductive circular definition a supplier is a constructor or something which could potentially produce a constructor. For a destructor a supplier is a coinductive circular definition or someone who can supply such a definition.

This demand will cause us do one of three things: to perform a deforesting cut if the constructor (resp. destructor) is present, to unroll the circular definition on which the demand has been placed, or, if demand is placed internally on an argument of a circular definition we will "pass the demand up": this is illustrated below (see section 4.3.3). It is important to realize that as we are dealing with finite terms whether there is demand or not can be easily detected.

Below, through examples, I am describing an algorithm which can be implemented. If this algorithm terminates it will return a deforested term. Of course, the problem concerns the termination (see also [10]); I conjecture that this procedure does terminate and so will always return a deforested (finite) term, however, at the time of writing I do not have a proof.

#### *4.3.1 Associativity of append*

We start with a classic example to illustrate the strategy of deforesting and the way it manages to discover useful identities. Recall the definition of append:

$$x, y \mapsto \left\{ \begin{array}{l} : \mathrm{app}(:=, y' = y) \;\; = \\ \mathrm{nil}() \qquad\quad \mapsto y' \\ \mathrm{cons}(v, vs) \mapsto \mu\, \mathrm{cons}(v, \mathrm{app}(vs, y')) \end{array} \right\} x$$

Notice in this definition there is no demand: or more specifically the only demand is on a free variable which we cannot satisfy until it is instantiated in some way. This means that app is already deforested.

We shall indicate the demand we are working on by a dot. This will usually be the demand nearest the root. However, the knot tying process may actually oblige us to calculate certain demands in order to secure the match it wants.

The first step below responds to the demand by unrolling the inner append:

$$\begin{aligned}
&\mathrm{app}(\cdot\,\mathrm{app}(x,y),z)\\
&=\mathrm{app}(\cdot\left\{\begin{array}{ll}\mathrm{nil}() & \mapsto y\\ \mathrm{cons}(v,vs) & \mapsto \mu\,\mathrm{cons}(v,\mathrm{app}(vs,y))\end{array}\right\}x,z)\\
&=\left\{\begin{array}{ll}\mathrm{nil}() & \mapsto \mathrm{app}(y,z)\\ \mathrm{cons}(v,vs) & \mapsto \mathrm{app}(\cdot\mu\,\mathrm{cons}(v,\mathrm{app}(vs,y)),z)\end{array}\right\}x\\
&=\left\{\begin{array}{ll}\mathrm{nil}() & \mapsto \mathrm{app}(y,z)\\ \mathrm{cons}(v,vs) & \mapsto \mu\,\mathrm{cons}(v,\mathrm{app}(\cdot\,\mathrm{app}(vs,y),z))\end{array}\right\}x
\end{aligned}$$

There is now an opportunity to tie a knot: the term $\mathrm{app}(\mathrm{app}(x,y),z)$ occurs with $vs$ substituted for the variable $x$. It is important to note that $vs$ is a "regenerate" variable (that is of regenerated type) as this tells us where the recursion lies. The trick now is to tie a tight knot! The only structure which is of relevance lies between the occurrence where the recurrence template stands and the actual recurrence. All other arguments may be treated as context as they are never altered by the recusions. Thus we may infer:

$$\mathrm{app}(\mathrm{app}(x,y),z)=\left\{\begin{array}{ll}:F(:=,u=\mathrm{app}(y,z)) & =\\ \mathrm{nil} & \mapsto u\\ \mathrm{cons}(w,ws) & \mapsto \mu\,\mathrm{cons}(w,F(ws,u))\end{array}\right\}x$$

Clearly $F=\mathrm{app}$ and so we have just deforested

$$\mathrm{app}(\mathrm{app}(x,y),z)\quad\Rightarrow\quad\mathrm{app}(x,\mathrm{app}(y,z)).$$

In this example there is also a slight efficiency improvement: the first argument is traversed twice in the first expression but only once in the second.

*4.3.2 Improving naive reverse*

Deforestation can make large efficiency improvements. Here is a classic example of an $O(n^2)$ program which can be improved to an $O(n)$ program. This transformation is interesting as it derives precisely what one might hope would be derived. Consider the following definition of reverse:

$$x\left\{\begin{array}{ll}:\mathrm{rev}_2(:=) & =\\ \mathrm{nil}() & \mapsto \mu\,\mathrm{nil}()\\ \mathrm{cons}(v,vs) & \mapsto \mathrm{app}(\mathrm{rev}_2(vs),\mu\,\mathrm{cons}(v,\mu\,\mathrm{nil}))\end{array}\right\}x$$

This definition already has internal demand: the app function creates demand on $\mathrm{rev}_2(vs)$ which causes us to unroll. We let $z = \mathrm{cons}(v, \mathrm{nil}))$ in this calculation and we will use our previous calculation in the last step (which, of course, would be recalculated automatically by a system) with a deforesting constructor cut with append:

$$\begin{aligned}
&\mathrm{app}(\cdot\, \mathrm{rev}_2(vs), z)\\
&= \mathrm{app}(\cdot \left\{\!\!\left|\begin{array}{ll} \mathrm{nil}() & \mapsto \mu\, \mathrm{nil}() \\ \mathrm{cons}(v', vs') & \mapsto \mathrm{app}(\mathrm{rev}_2(vs'), \mu\, \mathrm{cons}(v', \mu\, \mathrm{nil})) \end{array}\right|\!\!\right\} vs, z)\\
&= \left\{\!\!\left|\begin{array}{ll} \mathrm{nil}() & \mapsto \mathrm{app}(\cdot \mu\, \mathrm{nil}(), z) \\ \mathrm{cons}(v', vs') & \mapsto \mathrm{app}(\cdot\, \mathrm{app}(\mathrm{rev}_2(vs'), \mu\, \mathrm{cons}(v', \mu\, \mathrm{nil})), z) \end{array}\right|\!\!\right\} vs\\
&= \left\{\!\!\left|\begin{array}{ll} \mathrm{nil}() & \mapsto z \\ \mathrm{cons}(v', vs') & \mapsto \mathrm{app}(\cdot\, \mathrm{rev}_2(vs'), \mu\, \mathrm{cons}(v', z)) \end{array}\right|\!\!\right\} vs
\end{aligned}$$

We can now tie the knot to obtain:

$$\mathrm{app}(\mathrm{rev}_2(vs), z) = \left\{\!\!\left|\begin{array}{ll} : \mathrm{rev}_1(:=, z' = z) & = \\ \mathrm{nil}() & \mapsto z' \\ \mathrm{cons}(v', vs') & \mapsto \mathrm{rev}_1(vs', \mu\, \mathrm{cons}(v', z)) \end{array}\right|\!\!\right\} vs.$$

This has no residual demand and so is deforested. This is the fast definition of reverse which accumulates the reversed list on its second argument. Substituting this back into the original definition gives:

$$x \mapsto \left\{\!\!\left|\begin{array}{ll} \mathrm{nil}() & \mapsto \mathrm{nil}() \\ \mathrm{cons}(v, vs) & \mapsto \mathrm{rev}_1(vs, \mathrm{cons}(v, \mathrm{nil})) \end{array}\right|\!\!\right\} x$$

which has no residual demand and so is a deforested version of our original algorithm.

*4.3.3 Improving the accumulate on an infinite list*

We use the example of collecting the list of values so far in an infinite list:

$$L \mapsto \left(\!\!\left|\begin{array}{ll} : \mathrm{acc}(L', xs) = \\ (L', xs) \mapsto \left(\begin{array}{l} \mathrm{hd} : x \\ \mathrm{tl} : \mathrm{acc}(\mathrm{tl}\, \nu L', \mathrm{app}(\cdot xs, \mu\, \mathrm{cons}(\mathrm{hd}\, \nu L', \mu\, \mathrm{nil}))) \end{array}\right) \end{array}\right|\!\!\right) L$$

This function has internal demand: the demand is generated at the first argument of the app function — the destructors do not generate demand as

there is none passed down through the arguments of the inductive circular definition.

To remove this internal demand we must, this time, pass up the demand: this involves creating another argument through which we express that demand (in as tight a fashion as possible). This demand then becomes a new parameter of the coinductive circular definition. Now when we create this extra argument, rather like a time anomaly, we must reflect the change in the future circular call. In this case we abstract the append on its non-recursive argument and this gives the following equalities:

$$
\begin{aligned}
&\mathrm{acc}(L, xs)\\
&= \mathrm{acc}'(L, xs, \lambda z.\, \mathrm{app}(xs, z))\\
&= \left(\left|\begin{array}{l}\mathrm{hd} : xs\\ \mathrm{tl} : \mathrm{acc}'\ (\mathrm{tl}\,\nu L'\\ \qquad\qquad , \mathrm{app}(\cdot xs, \mu\,\mathrm{cons}(\mathrm{hd}\,\nu L', \mu\,\mathrm{nil})))\\ \qquad\qquad , \lambda z.\, \mathrm{app}(\mathrm{app}(\cdot xs, \mu\,\mathrm{cons}(\mathrm{hd}\,\nu L', \mu\,\mathrm{nil}))), z))\end{array}\right|\right)\\
&= \left(\left|\begin{array}{l}\mathrm{hd} : xs\\ \mathrm{tl} : \mathrm{acc}'\ (\mathrm{tl}\,L'\\ \qquad\qquad , \mathrm{app}(\cdot xs, \mathrm{cons}(\mathrm{hd}\,L', \mathrm{nil})))\\ \qquad\qquad , \lambda z.\, \mathrm{app}(xs, \mathrm{cons}(\mathrm{hd}\,L', z))))\end{array}\right|\right)\\
&= \left(\left|\begin{array}{l}\mathrm{hd} : xs\\ \mathrm{tl} : \mathrm{acc}'\ (\mathrm{tl}\,\nu L'\\ \qquad\qquad , \mathrm{app}(\cdot xs, \mu\,\mathrm{cons}(\mathrm{hd}\,\nu L', \mu\,\mathrm{nil})))\\ \qquad\qquad , \lambda z'.(\lambda z.\, \mathrm{app}(xs, z))@(\mu\,\mathrm{cons}(\mathrm{hd}\,\nu L', z')))\end{array}\right|\right)
\end{aligned}
$$

We can now tie the knot to obtain:

$$
\begin{aligned}
&\mathrm{acc}'(L, xs, F)\\
&= \left(\left|\begin{array}{l}: G =\\ \\ (L, y, F) \mapsto \left(\begin{array}{l}\mathrm{hd} : y\\ \mathrm{tl} : G\,(\mathrm{tl}\,\nu L\\ \qquad\qquad , \mathrm{app}(y, \mu\,\mathrm{cons}(\mathrm{hd}\,\nu L', \mu\,\mathrm{nil}))\\ \qquad\qquad , \lambda v.F@(\mu\,\mathrm{cons}(\mathrm{hd}\,\nu L, v)))\end{array}\right)\end{array}\right|\right)(L, xs, F).
\end{aligned}
$$

This is now deforested as all the demand has gone. However, we may make a further simplification: we can note that second argument can be obtained from the first at every call by $xs = F@nil$. This means we do not need the middle argument at all it can be replaced by the calculation. While this is a

nice observation it is not necessary in order to complete the deforestation and so we shall not use it here. We now have:

$$\mathrm{acc}(L, \mu\,\mathrm{nil}) = \mathrm{acc}'(L, \mu\,\mathrm{nil}, \lambda z.\,\mathrm{app}(\mu\,\mathrm{nil}, z)) = \mathrm{acc}'(L, \mu\,\mathrm{nil}, \lambda z.z)$$

which expresses the original function in deforested form.

*4.3.4 Getting values from infinite structures*

One way to think of a computation, such as calculating the $n^{\mathrm{th}}$ even number, is to create an infinite table containing the values and then use this to look up the value. Clearly this is horrendously inefficient, but, one might ask: if we allow for deforestation can this be transformed to an efficient program?

This example of a program transformation illustrates the effect of having a coinductive type "inside" an inductive type. In particular it shows how demand can be generated through a containing inductive circular definition.

We start by considering the program which gets the $n^{\mathrm{th}}$ tail of an infinite list.

$$(L, n) \mapsto \left\{\begin{array}{l| l} :\mathrm{gtt}(:=, L' = L) & = \\ \mathrm{zero} & \mapsto L \\ \mathrm{succ}(n') & \mapsto \mathrm{tl}\,\nu\,\mathrm{gtt}(n', L) \end{array}\right\} n$$

This is deforested as the demand generated by tl is never supplied. However, if we substitute a coinductive circular definition in for $L$ this will create a demand on inner occurrence of gtt. We shall resolve this as follows without reference to what caused the demand:

$$\begin{aligned} &\mathrm{tl}\,\nu \cdot \mathrm{gtt}(n, L) \\ &= \mathrm{tl}\,\nu \cdot \left\{\begin{array}{l| l} \mathrm{zero} & \mapsto L \\ \mathrm{succ}(n') & \mapsto \mathrm{tl}\,\nu \cdot \mathrm{gtt}(n', L) \end{array}\right\} n \\ &= \left\{\begin{array}{l| l} \mathrm{zero} & \mapsto \mathrm{tl}\,\nu \cdot L \\ \mathrm{succ}(n') & \mapsto \mathrm{tl}\,\nu\,\mathrm{tl}\,\nu \cdot \mathrm{gtt}(n', L) \end{array}\right\} n \end{aligned}$$

This allows us to tie the knot (tight):

$$\mathrm{tl} \cdot \mathrm{gtt}(n, L) = \left\{\begin{array}{l| l} :H(:=, L' = \mathrm{tl}\,L) & = \\ \mathrm{zero} & \mapsto L' \\ \mathrm{succ}(n') & \mapsto \mathrm{tl}\,\nu H(n', L') \end{array}\right\} n$$

Now it is clear that $H$ is gtt so we can replace $\mathrm{tl}\,\mathrm{gtt}(n', L)$ by $\mathrm{gtt}(n', \mathrm{tl}\,L)$ to

get a new version of gtt (I shall not change the name):

$$(L,n) \mapsto \left\{ \begin{array}{l} : \mathrm{gtt}(:=, L' = L) \ = \\ \mathrm{zero} \quad\quad \mapsto L \\ \mathrm{succ}(n') \mapsto \mathrm{gtt}(n', \mathrm{tl}\,\nu L) \end{array} \right\} n$$

Now let us consider the program which gets the $n^{\mathrm{th}}$ even number. first we introduce the notion of the infinite list of every other number starting at m:

$$m \mapsto \left( \begin{array}{l} : \mathrm{evn} = \\ \quad m' \mapsto \left( \begin{array}{l} \mathrm{hd} : m' \\ \mathrm{tl} : \mu\,\mathrm{succ}\,\mu\,\mathrm{succ}\,m' \end{array} \right) \end{array} \right) m.$$

Now we get the $n^{\mathrm{th}}$ element of the infinite list of evn starting at zero by applying hd to the the process of getting the $n^{\mathrm{th}}$ tail.

This means we want to deforest the following term:

$$n \mapsto \mathrm{hd}\,\nu \cdot \mathrm{gtt}(n, \mathrm{evn}(\mu\,\mathrm{zero}))$$

Where this has a demand expressed at the argument of hd and supplied at the second argument of gtt. We shall resolve this demand as follows:

$$\begin{aligned} &\mathrm{hd} \cdot \mathrm{gtt}(n, L) \\ &\quad = \mathrm{hd}\,\nu \cdot \left\{ \begin{array}{l} \mathrm{zero} \quad\quad \mapsto L \\ \mathrm{succ}(n') \mapsto \mathrm{gtt}(n', \mathrm{tl}\,\nu L) \end{array} \right\} n \\ &\quad = \left\{ \begin{array}{l} \mathrm{zero} \quad\quad \mapsto \mathrm{hd}\,\nu \cdot L \\ \mathrm{succ}(n') \mapsto \mathrm{hd}\,\nu \cdot \mathrm{gtt}(n', \mathrm{tl}\,\nu L) \end{array} \right\} n \end{aligned}$$

This allows us to tie another little knot:

$$\mathrm{hd} \cdot \mathrm{gtt}(n, L) = \left\{ \begin{array}{l} : \mathrm{gtt}'(:=, L' = L) \ = \\ \mathrm{zero} \quad\quad \mapsto \mathrm{hd}\,L' \\ \mathrm{succ}(n') \mapsto \mathrm{gtt}'(n', \mathrm{tl}\,L') \end{array} \right\} n$$

Now consider the original problem again, or rather a subproblem thereof:

$$
\begin{aligned}
&\mathrm{gtt}'(n, \mathrm{evn}(m)) \\
&= \left\{ \begin{array}{ll} \mathrm{zero} & \mapsto \mathrm{hd} \cdot \mathrm{evn}(m) \\ \mathrm{succ}(n') & \mapsto \mathrm{gtt}'(n', \mathrm{tl} \cdot \mathrm{evn}(m)) \end{array} \right\} n \\
&= \left\{ \begin{array}{ll} \mathrm{zero} & \mapsto m \\ \mathrm{succ}(n') & \mapsto \mathrm{gtt}'(n', \mathrm{evn}(\mathrm{succ}\,\mathrm{succ}\,m)) \end{array} \right\} n
\end{aligned}
$$

Notice that abstracting the zero out of the calculation helps us to spot the knot. We may now tie this knot to obtain:

$$
\mathrm{gtt}'(n, \mathrm{evn}(m)) = \left\{ \begin{array}{ll} : E(:=, m' = m) & = \\ \mathrm{zero} & \mapsto m' \\ \mathrm{succ}(n') & \mapsto E(n', \mu\,\mathrm{succ}\,\mu\,\mathrm{succ}\,m') \end{array} \right\} n.
$$

This is now deforested.

Now it is worth remarking that $E(n, \mu\,\mathrm{zero})$ may not have been quite the expected form for calculating even numbers. We may have expected:

$$
\mathrm{hd} \cdot \mathrm{gtt}(n, \mathrm{evn}(m)) = \left\{ \begin{array}{ll} : E'(:=, m) & = \\ \mathrm{zero} & \mapsto m \\ \mathrm{succ}(n') & \mapsto \mu\,\mathrm{succ}\,\mu\,\mathrm{succ}\,E'(n', m) \end{array} \right\} n.
$$

Both of these functions are deforested and, furthermore, they are equal. We shall take this as a small reminder that deforesting does not solve the equivalence problem!

### *4.4 Remarks on program equivalence*

As we have seen deforestation does not provide the solution to the program equivalence problem. However, it is very clear that we should in determining equivalence use the fact that one can deforest. If one does this then one of the remaining problems is to establish that the two programs can be expressed in such a manner as to have the same recursion pattern.

Now, in principle, this is easy to achieve. The idea is this: one unrolls the two programs in parallel. The demand to unroll arrives now from two sources. The first is because there are still deforesting cuts as usual. The second is because one wants to keep the programs unrolling in parallel. Thus, if there is demand on the one program this must be transmitted to the second. Finally knot tying must be done in parallel.

There is an advantage to this process as unrolling is equivalent to looking at distinguishability. Thus, if the two programs unroll in different ways one can actually reconstruct a pattern of destruction and application to values

which will distinguish the programs. Thus, in principle, one can either prove the programs equivalent or provide a counter-example – or never terminate, of course. If this works, it seems that this should be a standard tool should be provided with any programming language which is worth considering!

There are, unfortunately, a few outstanding problems with this idea! For example, this presumes that one has tamed the program equivalence problem for the proof theory of intuitionistic logic! It also presumes that deforestation always terminates and that such a proof technique is complete. There is, therefore, considerable work that is required before we can provide the sort of program verification tool which I feel should be standard. However, it is possible that we now have most of the technology in hand to provide such a tool.

## 5 Conclusions

I hope that these discussions have underlined that arriving at an appropriate semantic and proof theoretic formulation for inductive and coinductive datatypes is an extremely important for the field of program transformation and optimization. Furthermore, that a satisfactory semantic formulation (as given by mathematical induction or categorical initial and final datatypes) does not necessarily translate into a good manipulative system. In the case of inductive and coinductive datatypes, I would argue, the circular proof system provides a crucial insight and link between formal settings and the various techniques which have proved to be most useful in practice.

This paper has provided a walk through some of the ideas which underpin the development of good transformation and proof tools for the basic programming system provide by a cartesian closed category with datatypes. I have not attempted to provide detailed proofs and, indeed, I have introduced techniques, such as the deforestation algorithm, in a very informal way. I would be the first to admit that there remains alot of work to be done. Therefore, what I have described should be regarded as a program for future work. I hope, however, that I have provided some indications that this program might be fruitful.

## References

[1] Aldwinckle, J., *On the proof theory of model checking* (to appear).

[2] Burstall, R. and J. Darlington, *A transformation system for developing recursive programs*, Journal of the Association for Computing Machinery **24(1)** (1997), pp. 44–67.

[3] Cockett, R. and S. Lack, *Restriction categories II: Partial map classification*, Theoretical Computer Science (to appear).

[4] Cockett, R. and D. Spencer, *Strong categorical datatypes I*, in: R. A. G. Seely, editor, *International Meeting on Category Theory 1991*, Canadian Mathematical Society Proceedings (1992), pp. 141 – 169.

[5] Cockett, R. and D. Spencer, *Strong categorical datatypes II: A term logic for categorical programming*, Theoretical Computer Science **139** (1995), pp. 69–113.

[6] Fuhrmann, A. B. C. and A. Simpson, *Equational lifting monads*, Theoretical Computer Science (to appear).

[7] Ghani, N., "Adjoint Rewriting," Ph.D. thesis, University of Edinburgh (1995).

[8] Jacobs, B., "Categorical Logic and Type Theory," Elsevier, Amsterdam, 1999.

[9] Kozen, D., *Results on the propositional mu-calculus*, Theoretical Computer Science **27** (1983), pp. 113–118.

[10] Marlow, S., "Deforestation for Higher-Order Functional Programs," Ph.D. thesis, University of Glasgow (1996).

[11] Santocanale, L., "Sur les $\mu$-treillis libre," Ph.D. thesis, Univerite de Quebec a Montreal (1999).

[12] Scott, T. A. P. D. M. H. P., *Normalization by evaluation for typed lambda calculus with coproducts* (2001), extended abstract.
URL `http://www.cs.nott.ac.uk/~txa/drafts/coprod.ps`

[13] Slind, K., "Reasoning about Terminating Functional Programs," Ph.D. thesis, TU Munich (1999).
URL `http://www.cl.cam.ac.uk/users/kxs`

[14] Szabo, M., editor, "The Collected Papers of Gerhard Gentzen," Studies in Logic and the Foundations of Mathematics, North-Holland, Amsterdam, 1969.

[15] Telford, A. J. and D. A. Turner, *Ensuring termination in* esfp, in: *15th British Colloquium in Theoretical Computer Science*, 1999.

[16] Turner, D., *Elementary strong functional programming*, in: R.Plasmeijer and P.Hartel, editors, *First International Symposium on Functional Programming Languages in Education*, number 1022 in LNCS (1996).

[17] Wadler, P., *Deforestation: transforming programs to eliminate trees*, Theoretical Computer Science **73** (1990), pp. 231–248, (Special issue of selected papers from 2'nd ESOP.).