A Few Constructions on Constructors

Conor McBride[1], Healfdene Goguen[2] and James McKinna[3]

[1] School of Computer Science and Information Technology, University of Nottingham
[2] AT&T Labs, Florham Park, New Jersey
[3] School of Computer Science, University of St Andrews

Abstract. We present four constructions for standard equipment which can be generated for every inductive datatype: case analysis, structural recursion, no confusion, acyclicity. Our constructions follow a two-level approach—they require less work than the standard techniques which inspired them [11, 8]. Moreover, given a suitably heterogeneous notion of equality, they extend without difficulty to inductive families of datatypes. These constructions are vital components of the translation from dependently typed programs in pattern matching style [7] to the equivalent programs expressed in terms of induction principles [21] and as such play a crucial behind-the-scenes rôle in Epigram [25].

1 Introduction

In this paper, we show how to equip inductive families of datatypes [10] with useful standard equipment. When you declare an inductive datatype, you expect to be able to write total programs over it by case analysis and structural recursion; you expect its constructors to be injective and disjoint; you expect cyclic equations to be refutable. We show these expectations to be well-founded by exhibiting constructions which fulfil them. These constructions may readily be mechanised, so that we may rely on these intuitive properties of datatypes implicitly as we go about our work.

In prior publications, we have indeed relied implicitly on these properties. Both the 'elimination with a motive' tactic [22] and the Epigram programming language [25] make heavy use of this equipment, but the constructions themselves have only appeared in [21]. We hope that this paper will serve as a more accessible technical reference.

Our approach to all of these constructions is more directly computational than others in the literature. In effect, our presentation of the structural recursion operator reworks Giménez's *inductive* justification of Coq's `fix` primitive, but for us the notion of 'guarded by constructors' is expressed by a *recursive* computation. Meanwhile, the 'constructors injective and disjoint' properties are captured by a single 'no confusion' theorem which computes its conclusion from its input. This sidesteps the problems of extending the 'equality respects predecessors' method of proving injectivity (automated for simple types by Cornes and Terrasse [8], and still standard [4]) to inductive families. It is, besides, rather neater. The refutation of cyclic equations is again presented as a single theorem for each datatype family. To our knowledge, it is entirely original.

2 Inductive Datatypes

Before we begin, let us be clear about the data with which we deal and also introduce our notational conventions. We mean the strictly positive inductive families of datatypes from Luo's UTT [18]. Modulo technical details, these correspond to the inductive families found in the Alf system [10], the Coq system [26] and Epigram [25]. The story we tell here is by no means specific to any one presentation.

The Tree *example.* We require that a datatype be presented by a formation rule and constructors—Tree provides the paradigm:

$$\underline{\text{data}}\ \frac{}{\mathsf{Tree} : \star}\ \underline{\text{where}}\ \frac{}{\mathsf{leaf} : \mathsf{Tree}}\quad \frac{l, r : \mathsf{Tree}}{\mathsf{node}\ l\ r : \mathsf{Tree}}$$

and equipped with an induction principle like this

$$\frac{t : \mathsf{Tree}\quad P : \mathsf{Tree} \to \star\quad \mathit{leaf}' : P\ \mathsf{leaf}\quad \dfrac{l' : P\ l\quad r' : P\ r}{\mathit{node}'\ l\ r\ l'\ r' : P\ (\mathsf{node}\ l\ r)}}{\mathbf{treeInd}\ t\ P\ \mathit{leaf}'\ \mathit{node}' : P\ t}$$

whose computational behaviour is given by reduction schemes like these

$$\begin{array}{lcl}
\mathbf{treeInd}\ \mathsf{leaf}\ \quad P\ \mathit{leaf}'\ \mathit{node}' & \leadsto & \mathit{leaf}' \\
\mathbf{treeInd}\ (\mathsf{node}\ l\ r)\ P\ \mathit{leaf}'\ \mathit{node}' & \leadsto & \mathit{node}'\ l\ r \\
 & & \quad (\mathbf{treeInd}\ l\ P\ \mathit{leaf}'\ \mathit{node}') \\
 & & \quad (\mathbf{treeInd}\ r\ P\ \mathit{leaf}'\ \mathit{node}')
\end{array}$$

Here we follow Epigram's presentational style, with declarations resembling deduction rules. Also standard practice in Epigram is to make elimination principles (inductive or otherwise) take their arguments in the order they are typically conceived. The user supplies the *target* t of the elimination; the machine infers the *motive* P for the elimination from the goal at hand [22]; the user supplies the *methods* by which the motive is pursued in each case.

The general form. For each of our constructions, we shall treat Tree as a worked example, then give the general case. We consider inductive families [10]—mutually defined collections of inductive types, *indexed* by a given telescope, Θ

$$\underline{\text{data}}\ \frac{\Theta}{\mathsf{Fam}\ \Theta}\ \underline{\text{where}}\ \cdots\ \frac{\Delta\quad \cdots\quad \dfrac{\Psi_i}{v_i\ \Psi_i : \mathsf{Fam}\ \vec{r}_i}\quad \cdots}{\mathsf{con}\ \Delta\ v_1\ \ldots\ v_n : \mathsf{Fam}\ \vec{s}}\ \cdots$$

Telescopes. We use Greek capitals to denote *telescopes* [9]—sequences of declarations $x_1 : X_1; \ldots\ x_n : X_n$ where later types may depend on earlier variables. We shall freely write iterated binders over telescopes $\lambda\Theta \Rightarrow T$, and we shall also use telescopes in argument positions, as in $\mathsf{Fam}\,\Theta$ above, to denote the sequence of variables declared, here $\mathsf{Fam}\; x_1 \;\ldots\; x_n$. For brevity, we shall sometimes also write subscripted sequences $x_1 \;\ldots\; x_n$ in vector notation $\vec{x}$.

For any $\mathsf{Fam} : \forall\Theta \Rightarrow \star$, we write $\langle\mathsf{Fam}\rangle$ to denote the telescope $\Theta; x : \mathsf{Fam}\,\Theta$, thus capturing in a uniform notation the idea of 'an arbitrary element from an arbitrary family instance'. We may declare $\vec{y} : \Theta$, the renamed telescope $y_1 : X_1; \ldots\; x_n : [\vec{y}/\vec{x}]X_n$. We may assert $\vec{t} : \Theta$, meaning the substituted sequence of typings $t_1 : X_1; \ldots\; t_n : [\vec{t}/\vec{x}]X_n$ which together assert that the sequence $\vec{t}$ is *visible through*[4] Θ.

Constructors. An inductive family may have several constructors—con, above, is typical. It has a telescope of *non-recursive* arguments Δ which, for simplicity, we take to come before the *recursive* arguments v_i. Only as the family for the return types of constructors and their recursive arguments may the symbol Fam occur—that is, our families are *strictly positive.* None the less, recursive arguments may be higher-order, parametrised by a telescope Ψ_i. Note that the indices $\vec{s}$ of the return type and $\vec{r}_i$ of each recursive argument are arbitrary sequences visible through Θ.

In addition to the above criteria, some restriction must be made on the relative position of Fam and its contents in the universe hierarchy of types if paradox is to be avoided—we refer the interested reader to [18, 6] and leave universe levels implicit [14].

For any Δ, $\vec{v}$, i and $\vec{t} : \Psi_i$, we say that $v_i\,\vec{t}$ is *one step smaller* than $\mathsf{con}\;\Delta\;\vec{v}$. The usual notion of being *guarded by constructors* is just the transitive closure of this one step relation.

Induction and Computation. The induction principle for a family is as follows:

$$\frac{\vec{x} : \langle\mathsf{Fam}\rangle \qquad P : \forall\langle\mathsf{Fam}\rangle \Rightarrow \star \qquad \cdots \qquad \dfrac{\cdots \quad \dfrac{\Psi_i}{v'_i\,\Psi_i : P\;\vec{r}_i\;(v_i\,\Psi_i)} \quad \cdots}{con'\;\Delta\;\vec{v}\;\vec{v}' : P\;\vec{s}\;(\mathsf{con}\;\Delta\;\vec{v})} \qquad \cdots}{\mathbf{famInd}\;\vec{x}\;P\;\ldots con' \ldots : P\;\vec{x}}$$

It takes a target sequence in $\langle\mathsf{Fam}\rangle$, a motive abstracting over $\langle\mathsf{Fam}\rangle$, and a method for each constructor. Each method abstracts its constructor's arguments and the inductive hypotheses associated with recursive arguments. Here are the associated reduction schemes:

$$\begin{array}{l}
\ldots \\
\mathbf{famInd}\;_\;\ldots\;_\;(\mathsf{con}\;\Delta\;\vec{v})\;P\;\ldots con' \ldots \leadsto \\
\quad con'\;\Delta\;\vec{v} \quad \cdots \quad (\lambda\Psi_i \Rightarrow \mathbf{treeInd}\;\vec{r}_i\;(v_i\,\Psi_i)\;P\;\ldots con' \ldots) \quad \cdots \\
\ldots
\end{array}$$

The notion of computation captured by these induction principles—higher-order primitive recursion—yields a system which is strongly normalizing for well

[4] Less is visible through a longer telescope, but we see it in more detail.

typed terms [13]. Moreover, they support *large elimination*—the computation of types by recursion, which we use extensively in this paper. Induction principles provide a sound and straightforward basis for programming and reasoning with inductive structures, and whilst not especially attractive, they provide a convenient 'machine code' in terms of which higher-level programming and proof notations can be elaborated. There is plenty of scope for exploiting the properties of indices to optimize the execution of induction principles and, by the same token, the representation of inductive data [5].

Proofs by Elimination. We present some of our constructions just by writing out the proof terms schematically. For the more complex constructions, this is impractical. Hence we sometimes give the higher-level proof strategy rather than the term to which it gives rise. This is also our preferred way to *implement* the constructions, via the proof tools of the host system. We shall not require any sophisticated machinery for inductive proof [22]—indeed, we are constructing components for that machinery, so we had better rely on something simpler. We require only 'undergraduate' elimination, where the motive just λ-abstracts variables already $\forall$-abstracted in the goal. Let us put induction to work.

3 Case Analysis and Structural Recursion

Coq's `case` and `fix` primitives conveniently separate induction's twin aspects of case analysis and recursion, following a suggestion from Thierry Coquand. By way of justification, in [11] Eduardo Giménez shows how to reconstruct an individual `fix`-based recursion in terms of induction by a memoization technique involving an inductively defined 'course-of-values' data structure. Here, we show how to build the analogous tools 'in software', presenting essentially the same technique in general, and defining course-of-values *computationally*.

A case analysis principle is just an induction principle with its inductive hypotheses chopped off. Correspondingly, its proof is by an induction which makes no use of the inductive hypotheses. For Tree we have

$$\frac{t : \mathsf{Tree} \quad P : \mathsf{Tree} \to \star \quad \mathit{leaf}' : P\ \mathsf{leaf} \quad \dfrac{l, r : \mathsf{Tree}}{\mathit{node}'\ l\ r : P\ (\mathsf{node}\ l\ r)}}{\mathbf{treeCase}\ t\ P\ \mathit{leaf}'\ \mathit{node}' : P\ t}$$

$$\mathbf{treeCase}\ t\ P\ \mathit{leaf}'\ \mathit{node}' \ \Rightarrow\ \mathbf{treeInd}\ t\ P\ \mathit{leaf}'\ (\lambda l; r; l'; r' \Rightarrow \mathit{node}'\ l\ r)$$

and in general,

$$\frac{\vec{x} : \langle\mathsf{Fam}\rangle \quad P : \forall\langle\mathsf{Fam}\rangle \Rightarrow \star \quad \ldots \quad \dfrac{\ldots \quad \dfrac{\Psi_i}{v_i\ \Psi_i : \mathsf{Fam}\ \vec{r}_i} \quad \ldots}{\mathit{con}'\ \Delta\ \vec{v}\ \vec{v}' : P\ \vec{s}\ (\mathsf{con}\ \Delta\ \vec{v})} \quad \ldots}{\mathbf{famCase}\ \vec{x}\ P\ \ldots \mathit{con}' \ldots : P\ \vec{x}}$$

$$\begin{array}{l}\mathbf{famCase}\ \vec{x}\ P\ \ldots \mathit{con}' \ldots \ \Rightarrow \\ \quad \mathbf{famInd}\ \vec{x}\ P\ \ldots (\lambda\Delta; \vec{v}; \vec{v}' \Rightarrow \mathit{con}'\ \Delta\ \vec{v})\ \ldots\end{array}$$

The obvious 'pattern matching equations' for **famCase** hold computationally.

$$\begin{array}{l} \ldots \\ \mathbf{famCase}\ \_ \ldots \_\ (\mathsf{con}\ \Delta\ \vec{v})\ P \ldots con' \ldots \ \Rightarrow\ con'\ \Delta\ \vec{v} \\ \ldots \end{array}$$

The recursion principle captures the notion that to compute a $P\ t$, you may assume that you have a P for every recursive subobject *guarded by constructors* in t. The computational properties of the system give us a convenient way to capture this notion: inductions proceed when fed with constructors and get stuck otherwise. With the assistance of product and unit types, we may exploit large elimination to define a predicate transformer, capturing the idea that a given predicate P holds 'below t'. Informally, we write a primitive recursive program in pattern matching style, but the translation to induction is direct:

$$\frac{P : \mathsf{Tree} \to \star \quad t : \mathsf{Tree}}{P\ \mathbf{belowTree}\ t : \star}$$

$$\begin{array}{lll} P\ \mathbf{belowTree}\ \mathsf{leaf} & \Rightarrow & \mathsf{One} \\ P\ \mathbf{belowTree}\ (\mathsf{node}\ l\ r) & \Rightarrow & \quad (P\ \mathbf{belowTree}\ l \wedge P\ l) \\ & & \wedge\ (P\ \mathbf{belowTree}\ r \wedge P\ r) \end{array}$$

We may now state the recursion principle:

$$\frac{t : \mathsf{Tree} \quad P : \mathsf{Tree} \to \star \quad \dfrac{t : \mathsf{Tree} \quad m : P\ \mathbf{belowTree}\ t}{p\ t\ m : P\ t}}{\mathbf{treeRec}\ t\ P\ p : P\ t}$$

When we apply this principle, we do not enforce any particular case analysis strategy. Rather, we install a hypothesis which will unfold computationally whenever and wherever case analyses may reveal constructors. At any point in an interactive development, this unfolding hypothesis amounts to a menu of templates for legitimate recursive calls. This is how recursion is elaborated in Epigram—examples of its use can be found in [25], including the inevitable Fibonacci function

$$\begin{array}{l} \ldots \\ \mathbf{fib}\ (\mathsf{suc}\ (\mathsf{suc}\ n))\ \Rightarrow\ \mathbf{fib}\ n + \mathbf{fib}\ (\mathsf{suc}\ n) \end{array}$$

Let us now construct **treeRec**:

$$\mathbf{treeRec}\ t\ P\ p\ \Rightarrow\ p\ t\ (\mathbf{below}\ t) \quad \underline{\text{where}}$$

$$\frac{t : \mathsf{Tree} \quad m : P\ \mathbf{belowTree}\ t}{\mathbf{step}\ t\ m : P\ \mathbf{belowTree}\ t \wedge P\ t}$$

$$\mathbf{step}\ t\ m\ \Rightarrow\ (m; p\ t\ m)$$

$$\frac{t : \mathsf{Tree}}{\mathbf{below}\ t : P\ \mathbf{belowTree}\ t}$$

$$\begin{array}{lll} \mathbf{below}\ \mathsf{leaf} & \Rightarrow & () \\ \mathbf{below}\ (\mathsf{node}\ l\ r) & \Rightarrow & (\mathbf{step}\ l\ (\mathbf{below}\ l); \mathbf{step}\ r\ (\mathbf{below}\ r)) \end{array}$$

In the general case, we construct

$$\frac{P : \forall\langle\mathsf{Fam}\rangle \Rightarrow\star \quad \vec{x} : \langle\mathsf{Fam}\rangle}{P\ \mathbf{belowFam}\ \vec{x} : \star}$$

$$\begin{array}{l} P\ \mathbf{belowFam}\ \vec{x} \ \Rightarrow \\ \quad \mathbf{famInd}\ \vec{x}\ (\lambda\langle\mathsf{Fam}\rangle \Rightarrow\star) \\ \qquad \ldots \\ \qquad (\lambda\Delta; \vec{v}; \vec{B} \Rightarrow \ldots \wedge (\forall\Psi_i \Rightarrow (B_i\ \Psi_i \wedge P\ \vec{r}_i\ (v_i\ \Psi_i))) \wedge \ldots) \\ \qquad \ldots \end{array}$$

$$\frac{\vec{x} : \langle\mathsf{Fam}\rangle \quad P : \forall\langle\mathsf{Fam}\rangle \Rightarrow\star \quad \dfrac{\vec{y} : \langle\mathsf{Fam}\rangle \quad m : P\ \mathbf{belowFam}\ \vec{y}}{p\ \vec{y}\ m : P\ \vec{y}}}{\mathbf{famRec}\ \vec{x}\ P\ p : P\ \vec{x}}$$

$$\mathbf{famRec}\ \vec{x}\ P\ p \ \Rightarrow\ p\ \vec{x}\ (\mathbf{below}\ \vec{x}) \quad \underline{\text{where}}$$

$$\frac{\vec{y} : \langle\mathsf{Fam}\rangle \quad m : P\ \mathbf{belowFam}\ \vec{y}}{\mathbf{step}\ \vec{y}\ m : P\ \mathbf{belowFam}\ \vec{y} \wedge P\ \vec{y}}$$

$$\mathbf{step}\ \vec{y}\ m \ \Rightarrow\ (m; p\ \vec{y}\ m)$$

$$\frac{\vec{y} : \langle\mathsf{Fam}\rangle}{\mathbf{below}\ \vec{y} : P\ \mathbf{belowFam}\ \vec{y}}$$

$$\begin{array}{l} \mathbf{below}\ \vec{y} \ \Rightarrow \\ \quad \mathbf{famInd}\ \vec{y}\ (P\ \mathbf{belowFam}) \\ \qquad \ldots \\ \qquad (\lambda\Delta; \vec{v}\ \vec{b} \Rightarrow (\ldots;\ \lambda\Psi_i \Rightarrow (\mathbf{step}\ \vec{r}_i\ (v_i\ \Psi_i)\ (b_i\ \Psi_i))\ ; \ldots) \\ \qquad \ldots \end{array}$$

4 Heterogeneous Equality

We shall shortly prove that the constructors of inductive families are injective and disjoint, but before we prove it, we must first figure out how to *state* it. Equality is traditionally defined inductively as follows:[5]

$$\underline{\text{data}}\ \frac{x, vy : T}{\mathsf{Eq}_T\ x\ y : \star}\ \underline{\text{where}}\ \frac{x : T}{\mathsf{reflEq}_T\ x : \mathsf{Eq}_T\ x\ x}$$

As we have often observed, this notion of equality is unsuitable once dependently typed functions and data become the object of study, rather than merely the means to study simply typed phenomena. Even such a trivium as 'a function takes equal inputs to equal outputs' ceases to be a proposition (never mind being provable) as soon as the output type depends on the input type. If $f : \forall x\!:\!S \Rightarrow T[x]$, then we may not equate $f\ x : T[x]$ with $f\ y : T[y]$. McBride's approach to this problem [21, 22] has achieved some currency—liberalize the *formation* rule to

[5] In some systems, propositions inhabit a separate universe from datatypes: this distinction is unimportant for our purposes.

admit heterogeneous aspirations to equality whilst retaining a more conservative *elimination* rule delivering equal treatment only for homogeneous equations.

$$\frac{x : X \quad y : Y}{x = y : \star} \qquad \frac{x : X}{\mathsf{refl} : x = x}$$

$$\frac{x, x' : X \quad q : x = x' \quad \dfrac{x' : X \quad q : x = x'}{P\ x'\ q : \star} \quad p : P\ x\ \mathsf{refl}}{{=}\mathbf{Elim}\ x\ x'\ q\ P\ p : P\ x'\ q}$$

$${=}\mathbf{Elim}\ \_\ \_\ \mathsf{refl}\ \_\ p \leadsto p$$

Note that $=$**Elim** is *not* the standard induction principle which we would expect for such a definition, with a motive ranging over all possible equations $\langle=\rangle$, nor is either derivable from the other. However, it is plainly the structural elimination principle for the subfamily of homogeneous equations. It is similar in character to the Altenkirch-Streicher 'K' axiom [27]

$$\frac{q : \mathsf{Eq}_T \quad P : \mathsf{Eq}_T\ x\ x \to \star \quad p : P\ (\mathsf{reflEq}_T\ x)}{\mathbf{K}\ T\ x\ q\ P\ p : P\ q}$$

which is plainly the structural elimination rule for reflexive equations. Eq-with-**K** is known to be strictly stronger than Eq-without-**K** [16]. Heterogeneous equality and Eq-with-**K** are known have the same strength [21]. We shall not repeat the whole construction here, but merely observe that one may express a heterogeneous equation as a homogeneous equation on type-term pairs $(A; a) : \exists A{:}\star \Rightarrow A$. The law $\mathsf{Eq}_{\exists A \star \Rightarrow A}\ (A; a)\ (A; a') \to \mathsf{Eq}_A\ a\ a'$ is equivalent to **K** [27].

The crucial point is this: we can now formulate *telescopic equations*. If $\vec{x} : \Delta$ and $\vec{y} : \Delta$, we say $\vec{x} = \vec{y}$ is the telescope $q_1 : x_1 = y_1; \ldots; q_n : x_n = y_n$. We add that $\vec{\mathsf{refl}} : \vec{t} = \vec{t}$. Now we can start to work with equations on inductive families.

5 No Confusion

The standard techniques for proving that constructors are injective and disjoint may be found in [8]. It is not hard to construct a predicate which is trivial for leaf and absurd for $(\mathsf{node}\ l\ r)$—given a proof that the two are equal, the substitutivity of equality gives you 'trivial implies absurd'. Meanwhile, left and right projections will respect a proof that $\mathsf{node}\ l\ r = \mathsf{node}\ l'\ r'$, yielding $l = l'$ and $r = r'$. Of course, projections are not always definable—here they are *locally* definable because l and r are available to use as 'dummy' values for the non-node cases. As observed in [4], these techniques work unproblematically for simple types but require more care, not to mention more work, in the dependent case.

Surely one must be at least a little suspicious of an approach which requires us to define predecessor projections for a whole family when we intend only to hit one constructor. Can we not build sharper tools by programming with dependent types? We propose an alternative 'two-level' approach: first, compute the relevant proposition relating each pair of values, then show that it holds when the values are equal. The former is readily computed by nested **treeCase**:

$$\frac{t, t' \;:\; \mathsf{Tree}}{\mathbf{TreeNoConfusion}\; t\; t' \;:\; \star}$$

$$\begin{array}{llll}
\mathbf{TreeNoConfusion} & \mathsf{leaf} & \mathsf{leaf} & \Rightarrow\; \forall P \Rightarrow P \to P \\
\mathbf{TreeNoConfusion} & \mathsf{leaf} & (\mathsf{node}\; l'\; r') & \Rightarrow\; \forall P \Rightarrow P \\
\mathbf{TreeNoConfusion} & (\mathsf{node}\; l\; r) & \mathsf{leaf} & \Rightarrow\; \forall P \Rightarrow P \\
\mathbf{TreeNoConfusion} & (\mathsf{node}\; l\; r) & (\mathsf{node}\; l'\; r') & \\
\multicolumn{4}{l}{\quad \Rightarrow\; \forall P \Rightarrow (l = l' \to r = r' \to P) \;\to\; P}
\end{array}$$

In each case, the statement of 'no confusion' takes the form of an elimination rule. Given a proof of $t = t'$ with constructors on both sides, **TreeNoConfusion** $t\; t'$ computes an appropriate inversion principle, ready to apply to the goal at hand. We give an 'interactive' presentation of the proof:

$$?\mathbf{treeNoConfusion} \;:\; \forall t, t' : \mathsf{Tree} \Rightarrow\; t = t' \;\to\; \mathbf{TreeNoConfusion}\; t\; t'$$

The first step is to eliminate the equation—it is homogeneous—substituting t for t' and leaving the 'diagonal' goal:

$$?\mathbf{treeNoConfDiag} \;:\; \forall t : \mathsf{Tree} \Rightarrow\; \mathbf{TreeNoConfusion}\; t\; t$$

Now apply **treeCase** t, leaving problems we can readily solve

$$\begin{array}{lll}
?\mathbf{treeNoConfLeaf} & : & \mathbf{TreeNoConfusion}\; \mathsf{leaf}\; \mathsf{leaf} \\
 & \cong & \forall P \Rightarrow\; P \to P \\
\text{proof} & & \lambda P; p \Rightarrow p \\
?\mathbf{treeNoConfNode} & : & \forall l; r \Rightarrow \mathbf{TreeNoConfusion}\; (\mathsf{node}\; l\; r)\; (\mathsf{node}\; l\; r) \\
 & \cong & \forall l; r; P \Rightarrow\; (l = l \;\to\; r = r \;\to\; P) \;\to\; P \\
\text{proof} & & \lambda l; r; P; p \Rightarrow p\; \overrightarrow{\mathsf{refl}}
\end{array}$$

It is reassuring that we have no need of specific refutations for the impossible $n^2 - n$ off-diagonal cases, nor need we construct these troublesome predecessors.

The generalization is straightforward: firstly, iterating **famCase** yields the matrix of constructor cases; for equal constructors, assert that equality of the projections can be used to solve any goal; for unlike constructors, assert that any goal holds.

$$\frac{\vec{x}, \vec{y} \;:\; \langle \mathsf{Fam} \rangle}{\mathbf{FamNoConfusion}\; \vec{x}\; \vec{y} \;:\; \star}$$

$$\begin{array}{llll}
\ldots & & & \\
\mathbf{FamNoConfusion} & (\mathsf{con}\; \vec{a}) & (\mathsf{con}\; \vec{b}) & \Rightarrow \forall P \Rightarrow\; (\vec{a} = \vec{b} \to P) \;\to\; P \\
\ldots & & & \\
\mathbf{FamNoConfusion} & (\mathsf{chalk}\; \vec{a}) & (\mathsf{cheese}\; \vec{b}) & \Rightarrow \forall P \Rightarrow\; P \\
\ldots & & &
\end{array}$$

Now, to prove

$$?\mathbf{famNoConfusion} : \forall \vec{x} : \langle \mathsf{Fam} \rangle ; \vec{y} : \langle \mathsf{Fam} \rangle \Rightarrow \\ \vec{x} = \vec{y} \rightarrow \mathbf{FamNoConfusion}\ \vec{x}\ \vec{y}$$

first eliminate the equations in left-to-right order—at each stage the leftmost equation is certain to be homogeneous, with each successive elimination unifying the types in the next equation. This leaves

$$?\mathbf{famNoConfDiag} : \forall \vec{x} : \langle \mathsf{Fam} \rangle \Rightarrow \mathbf{FamNoConfusion}\ \vec{x}\ \vec{x}$$

which reduces by $\mathbf{famCase}\ \vec{x}$ to an easy problem for each case

$$\begin{array}{ll} \ldots & \\ ?\mathbf{famNoConfCon} : & \forall \vec{a} \Rightarrow \mathbf{FamNoConfusion}\ (\mathsf{con}\ \vec{a})\ (\mathsf{con}\ \vec{a}) \\ & \cong \forall \vec{a}; P \Rightarrow (\vec{a} = \vec{a} \rightarrow P) \rightarrow P \\ \text{proof} & \lambda \vec{a}; P; p \Rightarrow p\ \vec{\mathsf{refl}} \\ \ldots & \end{array}$$

We remark that the 'pattern matching rules' for **famNoConfusion** hold computationally:

$$\begin{array}{l} \ldots \\ \mathbf{famNoConfusion}\ \vec{\mathsf{refl}}\ \mathsf{refl}_{(\mathsf{con}\ \vec{a})}\ P\ p \ \Rightarrow\ p\ \vec{\mathsf{refl}} \\ \ldots \end{array}$$

6 Acyclicity

Inductive data structures admit no cycles—this is intuitively obvious, but quite hard to establish. We must be able to disprove all equations $x = t$ where x is *constructor-guarded* in t. It is deceptively easy for natural numbers. This goal,

$$? : \forall x : \mathsf{Nat} \Rightarrow x \neq \mathsf{suc}\ (\mathsf{suc}\ (\mathsf{suc}\ x))$$

(where $x \neq y$ abbreviates $x = y \rightarrow \forall P \Rightarrow P$) is susceptible to induction on x, but only because one suc looks just like another. Watch carefully!

$$\begin{array}{l} ? : \mathsf{zero} \neq \mathsf{suc}\ (\mathsf{suc}\ (\mathsf{suc}\ \mathsf{zero})) \\ ? : \forall x : \mathsf{Nat} \Rightarrow x \neq \mathsf{suc}\ (\mathsf{suc}\ (\mathsf{suc}\ x)) \rightarrow \\ \qquad \boxed{\mathsf{suc}\ x} \neq \mathsf{suc}\ (\mathsf{suc}\ (\mathsf{suc} \boxed{(\mathsf{suc}\ x)})) \end{array}$$

The base case follows by 'constructors disjoint'. The conclusion of the step case reduces by 'constructors injective' to the hypothesis, but only because the boxed sucs introduced by the induction are indistinguishable from the sucs arising from the goal. The proof is basically a 'minimal counterexample' argument—given a minimal cyclic term, rotate the cycle to create a smaller cyclic term—but here we get away with just one third of a rotation.

Of course, we need not be afraid to rotate the cycle completely: if x is buried n constructors deep in t, we need merely do an n-step induction. Let's prove

$$? : \forall x\!:\!\mathsf{Tree} \Rightarrow x \neq \mathsf{node}\,(\mathsf{node}\,\mathsf{leaf}\,x)\,\mathsf{leaf}$$

Proceeding by **treeRec**, we get

$$\begin{aligned} ? : \forall x\!:\!\mathsf{Tree} \Rightarrow {} & (\lambda y \Rightarrow y \neq \mathsf{node}\,(\mathsf{node}\,\mathsf{leaf}\,y)\,\mathsf{leaf})\ \mathbf{belowTree}\ x \ \rightarrow \\ & x \neq \mathsf{node}\,(\mathsf{node}\,\mathsf{leaf}\,x)\,\mathsf{leaf} \end{aligned}$$

Now use **treeCase** twice to dig out the cycle. First we get

$$\begin{aligned} &? : \cdots \rightarrow \ \mathsf{leaf} \neq \mathsf{node}\,(\mathsf{node}\,\mathsf{leaf}\,\mathsf{leaf})\,\mathsf{leaf} \\ &? : \forall l, r \Rightarrow \cdots \rightarrow \ \mathsf{node}\,l\,r \neq \mathsf{node}\,(\mathsf{node}\,\mathsf{leaf}\,(\mathsf{node}\,l\,r))\,\mathsf{leaf} \end{aligned}$$

We suppress the tuples induced by **treeRec** for the moment. The first subgoal is just 'constructors disjoint'. In the second case, we can follow the 'cycle-path', doing **treeCase** l. Again, we get a trivial off-path leaf case, and an interesting node case:

$$\begin{aligned} ? : \forall ll, lr, r \Rightarrow & \\ & (\lambda y \Rightarrow y \neq \mathsf{node}\,(\mathsf{node}\,\mathsf{leaf}\,y)\,\mathsf{leaf})\ \mathbf{belowTree}\ (\mathsf{node}\,(\mathsf{node}\,ll\,lr)\,r) \ \rightarrow \\ & \quad \mathsf{node}\,(\mathsf{node}\,ll \quad\ lr \qquad\qquad\qquad\qquad\quad)\,r \\ & \neq \mathsf{node}\,(\mathsf{node}\,\mathsf{leaf}\,(\mathsf{node}\,(\mathsf{node}\,ll\,lr)\,r)\ )\,\mathsf{leaf} \end{aligned}$$

Now we have exposed the entire cycle-path, taking us to lr. The $\cdots$**belowTree**$\cdots$ computes to a tuple containing a proof of $lr \neq \mathsf{node}\,(\mathsf{node}\,\mathsf{leaf}\,lr)\,\mathsf{leaf}$. Expanding $\neq$ and applying constructors reveals equational hypotheses $ll = \mathsf{leaf}$, $r = \mathsf{leaf}$ and $lr = \mathsf{node}\,(\mathsf{node}\,ll\,lr)\,r$. Substituting the first two in the third allows us to extract a contradiction from the inductive hypothesis.

6.1 General Acyclicity

Funny how the $\cdots$**belowTree**$\cdots$ computes to a tuple containing the contradiction as soon as we expose the cycle-path! Perhaps we can exploit this behaviour more directly. If we had a proof of

$$(x \neq)\ \mathbf{belowTree}\ \mathsf{node}\,(\mathsf{node}\,\mathsf{leaf}\,x)\,\mathsf{leaf}$$

we should certainly acquire a handy proof of $x \neq x$ somewhere in the resulting tuple. For the sake of brevity, let us write

$$\begin{aligned} x \not< t \quad & \text{for} \quad (x \neq)\ \mathbf{belowTree}\ t \\ x \not\leq t \quad & \text{for} \quad (x \neq)\ \mathbf{belowTree}\ t \wedge x \neq t \end{aligned}$$

Note that $x \not< \mathsf{node}\,l\,r$ is definitionally equal to $x \not\leq l \wedge x \not\leq r$. We now have the tools we need to state and prove a *general* acyclicity theorem for Tree:

$$? : \forall x, t\!:\!\mathsf{Tree} \Rightarrow x = t \ \rightarrow \ x \not< t$$

Eliminating the equation, we get the statement 'x is unequal to any of its proper subterms':

$$? : \forall x\!:\!\mathsf{Tree} \Rightarrow x \not< x$$

Unfortunately, we cannot build this tuple structure the way our **below** did in our construction of **treeRec**—inequality to x is not a hereditary property of trees! We must do induction.

$$\begin{array}{l} ? : \mathsf{One} \\ ? : \forall s, t \Rightarrow s \not< s \;\rightarrow\; t \not< t \;\rightarrow\; (\mathsf{node}\,s\,t \not\leq s \wedge \mathsf{node}\,s\,t \not\leq t) \end{array}$$

The base case is trivial. The step case splits in two, either following the s-path or the t-path.

$$\begin{array}{l} ? : \forall s, t \Rightarrow s \not< s \;\rightarrow\; \mathsf{node}\,s\,t \not\leq s \\ ? : \forall s, t \Rightarrow t \not< t \;\rightarrow\; \mathsf{node}\,s\,t \not\leq t \end{array}$$

Each of these is an instance of

$$? : \forall \cdots \Rightarrow x \not< x \;\rightarrow\; \delta x \not\leq x$$

for some variable x (respectively s and t) and some constructor-form increment[6] of it, δx (either $(\mathsf{node}\,x\,t)$ or $(\mathsf{node}\,s\,x)$). The proof is, sad to say, a little crafty: in each case, fix x and δx, then generalize the bound:

$$? : \forall b \Rightarrow x \not< b \;\rightarrow\; \delta x \not\leq b$$

If x is not below b, its increment is certainly neither below nor equal to b. This goes by induction on b. The base case is trivial, for certainly $\delta x \neq \mathsf{leaf}$ and there is nothing below b. In the step case, we introduce hypotheses and expand definitions selectively. The boxes and arrows show the way the proof fits together.

$$\begin{array}{rcl} l, r & : & \mathsf{Tree} \\ Hl & : & \boxed{x \not< l} \rightarrow \boxed{\delta x \not\leq l} \qquad H : (\boxed{x \not< l} \wedge \boxed{x \neq l}) \qquad ? : (\boxed{\delta x \not\leq l} \\ Hr & : & \boxed{x \not< r} \rightarrow \boxed{\delta x \not\leq r} \qquad \quad \wedge\, (\boxed{x \not< r} \wedge \boxed{x \neq r}) \qquad \quad \wedge\, \boxed{\delta x \not\leq r}) \\ & & \qquad\qquad\qquad\qquad\qquad\qquad \textbf{injectivity} \qquad \wedge\, \boxed{\delta x \neq \mathsf{node}\,l\,r} \end{array}$$

For a start, we know that x is not a proper subterm of either l or r, so we may certainly deduce that δx is not a subterm of either, and hence not a proper subterm of $\mathsf{node}\,l\,r$. However, we also know that x does not equal either l or r, hence regardless of whether δx is $\mathsf{node}\,x\,t$ or $\mathsf{node}\,s\,x$, it does not equal $\mathsf{node}\,l\,r$.

This construction generalizes readily to inductive families, as soon as we construct *telescopic inequality*. If $\vec{x} : \Delta$ and $\vec{y} : \Delta$, then let $\vec{x} \neq \vec{y}$ be $\forall(\vec{x} = \vec{y}) \Rightarrow \forall P \Rightarrow P$. We may then compare *sequences in* $\langle\mathsf{Fam}\rangle$, taking $\vec{x} \not< \vec{y}$ to be $(\vec{x} \neq)$ **belowFam** $\vec{y}$ as before, with $\vec{x} \not\leq \vec{y}$ being $\vec{x} \not< \vec{y} \wedge \vec{x} \neq \vec{y}$. This is a suitably heterogeneous notion of 'not a (proper) subterm', and it is entirely compatible

[6] Readers interested in 'constructor-form increments' may find [17, 1] useful.

with the key lemmas—one for each recursive argument $v_i : \forall\Psi_i \Rightarrow \mathsf{Fam}\,\vec{r}_i$ of each constructor $\mathsf{con}\,\Delta\,\vec{v} : \mathsf{Fam}\,\vec{s}$.

$$\forall\Psi_i \Rightarrow \forall\vec{b}{:}\langle\mathsf{Fam}\rangle \Rightarrow \vec{r}_i;(v_i\,\Psi_i) \not< \vec{b} \;\rightarrow\; \vec{s};(\mathsf{con}\,\Delta\,\vec{v}) \not\leq \vec{b}$$

Note that in the higher-order case, an increment steps from a single arbitrary $x = v_i\,\Psi_i$ to the node which contains it $\delta x = \mathsf{con}\,\Delta\,\vec{v}$. The proof of this lemma is again by **famInd** $\vec{b}$, and it goes more or less as before. Once again, x is not a proper subterm of b's subnodes, so δx is not a subterm of b; moreover δx is not equal to b because either b is made with a constructor other than con, or its subnodes are all distinct from x.

Of course, in the higher-order case—infinitely branching trees—it is not generally possible to search mechanically for a cycle. However, the theorem still holds, and any cycle the user can exhibit will deliver an absurdity. In the first-order case, one need merely search the tuple $\vec{x} \not< \vec{t}$ for a proof of $\vec{x} \neq \vec{x}$.

7 Conclusions and Further Work

We have shown how to construct all the basic apparatus we need for structurally recursive programming with dependent families as proposed by Coquand [7] and implemented in Alf [19]. When pattern matching on an element from a specific branch of an inductive family, $x : \mathsf{Fam}\,\vec{t}$, one may consider only the cases where the $\vec{t}$ coincide with the indices of a constructor's return type $\mathsf{con}\,\Delta\,\vec{v} : \mathsf{Fam}\,\vec{s}$.

However, where Alf relied on a syntactic criterion for constructor-guarded recursion and an unspecified external notion of unification, we have constructed all of this technology from the standard induction principles, together with heterogeneous equality. We represent the unification constraints as equational hypotheses $\vec{s} = \vec{t}$ and reduce them where possible by the procedure given in [20], which is complete for all first-order terms composed of constructors and variables.

We have not shown here how to extend these constructions to *mutual* inductive definitions. These may always be simulated by a single inductive family indexed over a choice of mutual branch, but it is clear that a more direct treatment is desirable. We expect this to be relatively straightforward. The 'no confusion' result should extend readily to coinductive data as it relies only on case analysis. We should also seek an internal construction of guarded corecursion, underpinning the criteria in use today [12].

In the more distant future, we should like to see the computational power of our type systems working even harder for us: we have given the general form of our constructions, but they must still be rolled out once for each datatype by a metaprogram which is part of the system's implementation, manipulating the underlying data structure of terms and types. However convenient one's tools for such tasks [24], it would still be preferable to perform the constructions once, generically with respect to a universe of inductive families. Research adapting generic functional programming [15] for programs and proofs in Type Theory is showing early promise [3, 2].

Moreover, we might hope to exploit reflection to increase the size of each reasoning step. At the moment our 'no confusion' theorem computes how to simplify a single equation. This is used as a component in a unification tactic, but we might hope to reflect the entire unification process in a single theorem, reducing a system of equations to their simplest form.

The main point, however, is that our intuitive expectations of constructors have been confirmed, so we should no longer need to worry about them. The Epigram language and system [25, 23] takes these constructions for granted. We see no reason why the users of other systems should work harder than we do.

References

1. Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. ∂ for data: derivatives of data structures. *Fundamenta Informaticae*, 2005.
2. Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Generic Programming*, 2003. Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl, July 2002.
3. Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10:265–269, 2003.
4. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer-Verlag, 2004.
5. Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In Stefano Berardi, Mario Coppo, and Ferrucio Damiani, editors, *Types for Proofs and Programs, Torino, 2003*, volume 3085 of *LNCS*, pages 115–129. Springer-Verlag, 2004.
6. Thierry Coquand. An analysis of Girard's paradox. In *Proceedings of the First IEEE Symposium on Logic in Computer Science, Cambridge, Massachussetts*, pages 227–236, 1986.
7. Thierry Coquand. Pattern Matching with Dependent Types. In Bengt Nordström, Kent Petersson, and Gordon Plotkin, editors, *Electronic Proceedings of the Third Annual BRA Workshop on Logical Frameworks (Båstad, Sweden)*, 1992.
8. Cristina Cornes and Delphine Terrasse. Inverting Inductive Predicates in Coq. In *Types for Proofs and Programs, '95*, volume 1158 of *LNCS*. Springer-Verlag, 1995.
9. Nicolas G. de Bruijn. Telescopic Mappings in Typed Lambda-Calculus. *Information and Computation*, 91:189–204, 1991.
10. Peter Dybjer. Inductive Sets and Families in Martin-Löf's Type Theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*. CUP, 1991.
11. Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs, '94*, volume 996 of *LNCS*, pages 39–59. Springer-Verlag, 1994.
12. Eduardo Giménez. Structural Recursive Definitions in Type Theory. In *Proceedings of ICALP '98*, volume 1443 of *LNCS*. Springer-Verlag, 1998.
13. Healfdene Goguen. *A Typed Operational Semantics for Type Theory.* PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1994. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/94/ECS-LFCS-94-304/`.

14. Robert Harper and Randy Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991.
15. Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programmming*, 51:117–151, 2004.
16. Martin Hofmann and Thomas Streicher. A groupoid model refutes uniqueness of identity proofs. In *Proc. Ninth Annual Symposium on Logic in Computer Science (LICS) (Paris, France)*, pages 208–212. IEEE Computer Society Press, 1994.
17. Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
18. Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.
19. Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, LNCS 806. Springer-Verlag, 1994. Selected papers from the Int. Workshop TYPES '93, Nijmegen, May 1993.
20. Conor McBride. Inverting inductively defined relations in LEGO. In E. Giménez and C. Paulin-Mohring, editors, *Types for Proofs and Programs, '96*, volume 1512 of *LNCS*, pages 236–253. Springer-Verlag, 1998.
21. Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/`.
22. Conor McBride. Elimination with a Motive. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES'00)*, volume 2277 of *LNCS*. Springer-Verlag, 2002.
23. Conor McBride. Epigram: Practical programming with dependent types. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming 2004*, Lecture Notes in Computer Science. Springer-Verlag, 2005+. Revised lecture notes from the International Summer School in Tartu, Estonia.
24. Conor McBride and James McKinna. Functional Pearl: I am not a Number: I am a Free Variable. In Henrik Nilsson, editor, *Proceedings of the ACM SIGPLAN Haskell Workshop 2004, Snowbird, Utah*. ACM, 2004.
25. Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.
26. Christine Paulin-Mohring. Définitions Inductives en Théorie des Types d'Ordre Supérieur. Habilitation Thesis. Université Claude Bernard (Lyon I), 1996.
27. Thomas Streicher. Investigations into intensional type theory. Habilitation Thesis, Ludwig Maximilian Universität, 1993.