Exploring the Regular Tree Types

Peter Morris, Thorsten Altenkirch and Conor McBride

School of Computer Science and Information Technology
University of Nottingham

Abstract. In this paper we use the Epigram language to define the universe of regular tree types—closed under empty, unit, sum, product and least fixpoint. We then present a generic decision procedure for Epigram's in-built equality at each type, taking a complementary approach to that of Benke, Dybjer and Jansson [7]. We also give a generic definition of map, taking our inspiration from Jansson and Jeuring [21]. Finally, we equip the regular universe with the partial derivative which can be interpreted functionally as Huet's notion of 'zipper', as suggested by McBride in [26] and implemented (without the fixpoint case) in Generic Haskell by Hinze, Jeuring and Löh [18]. We aim to show through these examples that generic programming can be ordinary programming in a dependently typed language.

1 Introduction

This paper is about generic programming [6] in the dependently typed functional language Epigram [28, 29]. Generic programming allows programmers to explain how a single algorithm can be instantiated for a variety of datatypes, by computation over each datatype's structure. In particular, we construct the universe of regular tree types—the datatypes closed under empty, unit, sum, product and least fixpoint. We define a de Bruijn indexed syntax [14] for these types, but we do not interpret this syntax via a recursive function: rather we give the elements for a given type as an *inductive family* [16]. It is Epigram's support for dependent pattern matching [13] which makes this approach practicable.

The universe of regular tree types is small compared to others we might imagine [4, 7], but it is rich in structure. We exploit some of that structure in our programs: Epigram's standard equality is decidable for every regular tree type; every regular tree type constructor has a notion of functorial 'map'; we also give the formal derivative of each type expression, including fixpoints, and acquire the related notion of one-hole context or 'zipper' [20]. In the last example McBride's observation [26], given its explanation in [3], has finally become a program.

1.1 What is a universe?

The notion of a *universe* in Type Theory was introduced by Per Martin-Löf [25, 33] as a means to abstract over specific collections of types. A universe is given by a type $U : \star$ of *codes* representing just the types in the collection, and a function $T : U \to \star$ which interprets each code as a type. A standard example is a universe of *finite* types—each type may be coded by a natural number representing its size. We can declare the natural numbers in Epigram as follows

$$\underline{\text{data}}\ \frac{}{\mathsf{Nat} : \star}\ \underline{\text{where}}\ \frac{}{\mathsf{zero} : \mathsf{Nat}}\ \frac{n : \mathsf{Nat}}{\mathsf{suc}\ n : \mathsf{Nat}}$$

One way to interpret each Nat as a finite type is to write a recursive function which calculates a type of the right size, given an empty type Zero, a unit type One and disjoint unions $S + T$

$$\begin{array}{ll} \underline{\text{let}}\ \dfrac{n : \mathsf{Nat}}{\mathbf{fin}\ n : \star} & \begin{array}{lll} \mathbf{fin}\ n & \Leftarrow & \underline{\text{rec}}\ n \\ \quad \mathbf{fin}\ \ \mathsf{zero} & \Rightarrow & \mathsf{Zero} \\ \quad \mathbf{fin}\ (\mathsf{suc}\ n) & \Rightarrow & \mathsf{One} + \mathbf{fin}\ n \end{array} \end{array}$$

Another way is to define directly an *inductive family* [16] of finite types:

$$\underline{\text{data}}\ \frac{n : \mathsf{Nat}}{\mathsf{Fin}\ n : \star}\ \underline{\text{where}}\ \frac{}{\mathsf{fz} : \mathsf{Fin}\ (\mathsf{suc}\ n)}\ \frac{i : \mathsf{Fin}\ n}{\mathsf{fs}\ i : \mathsf{Fin}\ (\mathsf{suc}\ n)}$$

$\mathsf{Fin}\ n$ gives a coding of the set $\{0, \ldots, n-1\}$. $\mathsf{Fin}\ \mathsf{zero}$ is uninhabited because no constructor targets it; $\mathsf{Fin}\ (\mathsf{suc}\ n)$ has one more element than $\mathsf{Fin}\ n$. Below we tabulate the first few types in this family: we show the 'n' arguments to fz and fs—usually left implicit—as subscripts, and write in decimal to save space.

Fin 0	Fin 1	Fin 2	Fin 3	Fin 4	$\cdots$
	fz_0	fz_1	fz_2	fz_3	$\cdots$
		$\mathsf{fs}_1\ \mathsf{fz}_0$	$\mathsf{fs}_2\ \mathsf{fz}_1$	$\mathsf{fs}_3\ \mathsf{fz}_2$	$\ddots$
			$\mathsf{fs}_2\ (\mathsf{fs}_1\ \mathsf{fz}_0)$	$\mathsf{fs}_3\ (\mathsf{fs}_2\ \mathsf{fz}_1)$	$\ddots$
				$\mathsf{fs}_3\ (\mathsf{fs}_2\ (\mathsf{fs}_1\ \mathsf{fz}_0))$	$\ddots$
					$\ddots$

In either presentation, Nat acts as a syntax for the finite types which we then equip with a semantics via $\mathbf{fin}$ or Fin. Let us emphasize that Nat is an ordinary datatype, and hence operations such as $\mathbf{plus}$ can be used to equip the finite universe with structure: $\mathsf{Fin}\ (\mathbf{plus}\ m\ n)$ is isomorphic to $\mathsf{Fin}\ m + \mathsf{Fin}\ n$. Universe constructions express generic programming for collections of datatypes [6, 18, 21] in terms of ordinary programming with their codes.

The notion of universe brings an extra degree of freedom and of precision to the business of generic programming. By their nature compiler extensions such as Generic Haskell [10] support the extension of generic operations to the whole of Haskell's type system, but we are free to construct a continuum, from large universes which support basic functionality to small, highly structured universes which support highly advanced functionality. Benke, Dybjer and Jansson provide a good introduction to this continuum in [7]. In fact every family of types, whether inductive like Fin or computational like $\mathbf{fin}$, yields a universe.

1.2 From Finite Types to Regular Tree Types

The finite types are closed under 'arithmetic' type constructors such as empty, unit, sum and product. If we also add list formation, we leave the finite universe and acquire the *regular expression types*. We can code these (with respect to an alphabet of size n) by the following syntax

$$\underline{\text{data}}\ \frac{n : \mathsf{Nat}}{\mathsf{Rex}\ n : \star}\ \underline{\text{where}}\ \frac{}{\mathsf{fail}, \mathsf{nil}, \mathsf{dot} : \mathsf{Rex}\ n}\ \frac{i : \mathsf{Fin}\ n}{\mathsf{only}\ i : \mathsf{Rex}\ n}$$
$$\frac{S, T : \mathsf{Rex}\ n}{S\ \mathsf{or}\ T,\ S\ \mathsf{then}\ T : \mathsf{Rex}\ n}\ \frac{R : \mathsf{Rex}\ n}{R\ \mathsf{star} : \mathsf{Rex}\ n}$$

So for instance we translate the regular expression

$$(A + BC)^\star \subseteq \{A, B, C\}^*$$

into the code

$$((\mathsf{only\ fz})\ \mathsf{or}\ ((\mathsf{only}\ (\mathsf{fs\ fz}))\ \mathsf{then}\ (\mathsf{only}\ (\mathsf{fs}\ (\mathsf{fs\ fz})))))\ \mathsf{star} : \mathsf{Rex}\ 3$$

From each regular expression in the syntax, we may then compute a type which represents the words which match it.

$$\underline{\text{let}}\ \frac{R : \mathsf{Rex}\ n}{\mathbf{Words}_n\ R : \star}\quad
\begin{array}{lcl}
\mathbf{Words}_n\ R & \Leftarrow & \underline{\text{rec}}\ R \\
\mathbf{Words}_n\ \mathsf{fail} & \Rightarrow & \mathsf{Zero} \\
\mathbf{Words}_n\ \mathsf{nil} & \Rightarrow & \mathsf{One} \\
\mathbf{Words}_n\ \mathsf{dot} & \Rightarrow & \mathsf{Fin}\ n \\
\mathbf{Words}_n\ (\mathsf{only}\ i) & \Rightarrow & \mathsf{Single}\ i \\
\mathbf{Words}_n\ (S\ \mathsf{or}\ T) & \Rightarrow & \mathbf{Words}_n\ S + \mathbf{Words}_n\ T \\
\mathbf{Words}_n\ (S\ \mathsf{then}\ T) & \Rightarrow & \mathbf{Words}_n\ S \times \mathbf{Words}_n\ T \\
\mathbf{Words}_n\ (R\ \mathsf{star}) & \Rightarrow & \mathsf{List}\ (\mathbf{Words}_n\ R)
\end{array}$$

Some example **Words** of the expression above would be represented thus:

$$\begin{array}{rl}
BCA \mapsto & [\mathsf{inr}\ (\mathsf{single}\ (\mathsf{fs\ fz}); \mathsf{single}\ (\mathsf{fs}\ (\mathsf{fs\ fz}))), \mathsf{inl}(\mathsf{single\ fz})] \\
A \mapsto & [\mathsf{inl}\ (\mathsf{single\ fz})] \\
\varepsilon \mapsto & []
\end{array}$$

This universe, like the finite types, has much algebraic structure to expose, and there is plenty of *ad hoc* work devoted to it, motivated by applications to document structure [19].

Moving just a little further, we can generalise from *lists* to *trees* by replacing star with a binding operator μ which indicates the least fixpoint of an algebraic type expression. Closing under μ gives us the universe of *regular tree types*. In effect, we acquire the first-order fragment of the datatypes found in Standard ML [31]. These include the string-like structures such as the natural numbers, $\mu N.\,1 + N$ and the lists of A's, $\mu L.\,1 + A \times L$, but also branching structures such as binary trees $\mu T.\,1 + T \times T$. Nesting μ yields structures like the finitely branching trees, whose nodes carry lists of subtrees, $\mu F.\, \mu L.\ 1 + F \times L$. It is this class of types together with the structures and algorithms they support, which we shall study in this paper.

Of course, there are plenty more universes to visit. Altenkirch and McBride construct the *nested datatypes*, allowing non-uniform type constructors to be defined by recursion [4]. Benke, Dybjer and Jansson construct the *indexed inductive definitions* [7, 17], their motivation being that these structures are effectively those of the Agda system [12] with which they work.

1.3 Programming in Epigram

Epigram [28, 29] is a functional programming language with an interactive editor, incrementally typechecking source code containing *sheds*, $[\cdots]$, whose contents are free text which remains unchecked. It supports programming with inductive families in a pattern matching style, as proposed by Thierry Coquand [13] and first implemented in the Alf system [24].

However, Epigram programs elaborate into a version of Luo's UTT [22]. This is a more spartan and more clearly defined theory than that of Alf, equipping inductive families only with the induction principles which directly reflect their constructors. In this respect, Epigram more closely resembles its direct ancestor, Lego [23], and also to some extent the Coq system [11]. The design criteria for a good high-level programming language and a good low-level core often pull in opposite directions, hence we separate them. At present, neither Agda nor Coq directly supports pattern matching with inductive familes—hand-coding our constructions in these systems would be possible but unnecessarily painful.

Epigram's data structures are introduced by declaring their applied formation rules and constructors in a natural deduction style. Argument declarations may be omitted where inferrable by unification from their usage—for example, in our declarations of Fin's constructors, fz and fs, there is no need to declare $n : \mathsf{Nat}$. The resemblance between constructor declarations and typing rules is no accident. We intend to encourage the view of an inductive family as a universe capturing a small type system—and that is exactly how we work in this paper.

Epigram functions are given a type signature, also in the natural deduction style, then developed in a *decision tree* structure, shown here by indentation and representing a hierarchical analysis of the task it performs. Each node in a decision tree has a left-hand side which shows the information available, in the form of the *patterns* into which the arguments have been analysed, and a right-hand side which explains how to proceed in that case. The latter may take one of three forms:

$\Rightarrow t$ the function *returns* t, an expression of the appropriate type, constructed over the pattern variables on the left;

$\Leftarrow e$ the function's analysis is refined *by* e, an *eliminator* expression, characterising some scheme of case analysis or recursion, giving rise to a bunch of subnodes with more informative left-hand sides;

$\mid w$ the subnodes' left-hand sides are then extended *with* the value of w, some intermediate computation, in an extra column: this may then be analysed in addition to the function's original arguments.

In effect, Epigram gives a programming notation to some constructions which are more familiar as tactics in proof systems: $\Rightarrow$ corresponds to Coq's `exact` and $\mid$ resembles `generalize`; $\Leftarrow$ is McBride's elimination tactic [27]. McBride and McKinna give a thorough treatment of Epigram elaboration in [29], and begin to explore the flexibility of the $\Leftarrow$ construct. In this paper, however, we shall need only the standard constructor-guarded recursion operators $\underline{\text{rec}}\,x$, which we make explicit, and the standard constructor case analysis operators $\underline{\text{case}}\,x$, which we leave implicit whenever their presence is directly inferable from the resulting constructor patterns. In general, we are only explicit about case analysis when its results are empty:

$$\underline{\text{let}}\ \frac{x\ :\ \mathsf{Fin\ zero}}{\textbf{impossible}\ x\ :\ \mathsf{Zero}}\qquad \textbf{impossible}\ x\ \Leftarrow \underline{\text{case}}\ x$$

Case analyses in Epigram, as in Alf, are constrained by the requirement in each branch that the indices of the scrutinee—zero for $x : \mathsf{Fin\ zero}$ above—coincide with those of the constructor pattern in question—above, $(\mathsf{suc}\ n)$ in both cases. When they concern constructor symbols, these constraints are automatically simplified by first-order unification: impossible cases are dismissed, as above, and the possible cases are simplified. The $\Leftarrow$ construct thus generalises Alf's dependent constructor matching 'in software'.

Before we start work in earnest, we must own up to the notational liberties we have taken in this paper which the current implementation of Epigram does not support. At present, neither the $\mid w$ notation, nor the suppression of obvious $\Leftarrow\ \underline{\text{case}}\ \ldots$ nodes has been implemented: both omissions have simple but verbose workarounds—expanding the programs here would shed more heat than light. More trivially, we work in `ASCII` rather than LaTeX and have only prefix operators thus far—the notation we use here is cosmetically more advanced.

2 The Universe of Regular Tree Types

We define the codes for the regular tree types as follows:

$$\underline{\text{data}}\ \frac{n\ :\ \mathsf{Nat}}{\mathsf{Reg}\ n\ :\ \star}$$

$$\underline{\text{where}}\ \frac{}{\text{‘Z’}\ :\ \mathsf{Reg}\ (\mathsf{suc}\ n)}\quad \frac{T\ :\ \mathsf{Reg}\ n}{\text{‘wk’}\ T\ :\ \mathsf{Reg}\ (\mathsf{suc}\ n)}\quad \frac{S\ :\ \mathsf{Reg}\ n\quad T\ :\ \mathsf{Reg}\ (\mathsf{suc}\ n)}{\text{‘let’}\ S\ T\ :\ \mathsf{Reg}\ n}$$

$$\frac{}{\text{‘0’}, \text{‘1’}\ :\ \mathsf{Reg}\ n}\quad \frac{S, T\ :\ \mathsf{Reg}\ n}{S\ \text{‘+’}\ T, S\ \text{‘}\times\text{’}\ T\ :\ \mathsf{Reg}\ n}\quad \frac{F\ :\ \mathsf{Reg}\ (\mathsf{suc}\ n)}{\text{‘}\mu\text{’}\ F\ :\ \mathsf{Reg}\ n}$$

This is syntax-with-binding in de Bruijn style—the numeric index gives the number of free type variables available in the expression. The ‘Z’ refers to the most local variable (de Bruijn index zero), where there is one; the weakening constructor 'wk', read backwards, discards the top variable, allowing access to the others. We can thus define an embedding from $\mathsf{Fin}\ n$ to the representation of variables in $\mathsf{Reg}\ n$

$$\underline{\text{let}}\ \frac{X : \mathsf{Fin}\ n}{\text{`var'}\ X : \mathsf{Reg}\ n} \quad \begin{array}{l} \text{`var'}\ X \Leftarrow \underline{\text{rec}}\ X \\ \quad \text{`var'}\ \ \mathsf{fz} \quad\ \Rightarrow \text{`Z'} \\ \quad \text{`var'}\ (\mathsf{fs}\ X) \Rightarrow \text{`wk'}\ (\text{`var'}\ X) \end{array}$$

Both 'μ' (least fixpoint) and 'let' (local definition) bind a variable. The latter clearly introduces redundancy, as does the applicability of 'wk' to expressions other than variables. We could have chosen a redundancy free representation, making '**var**' a constructor and dropping 'Z', 'wk' and 'let'. Such a syntax could be equipped with a renaming functor and a substitution monad as in [5] and we should need this equipment *and proofs of its properties* to do our work. Definition and weakening replace just enough of the behavior of substitution for us to avoid this extra effort.

A similar choice presents itself when we come to interpret this syntax. It seems natural to interpret only the *closed* type expressions—the elements of $\mathsf{Reg}\ \mathsf{zero}$—substituting whenever we go under a 'μ' or 'let' binder. Some simple operations, such as our generic equality, become even simpler if we take this choice, but other operations, such as 'map', require us to work with properties of substitution. We choose to sidestep substitution in the usual way, interpreting *open* expressions over an environment. We construct our environments carefully to support the way we shall use them: they are *telescopic* [15] in the sense that each new variable is bound to an expression over the previous variables.

$$\underline{\text{data}}\ \frac{n : \mathsf{Nat}}{\mathsf{Tel}\ n : \star}\ \underline{\text{where}}\ \frac{}{\varepsilon : \mathsf{Tel}\ \mathsf{zero}} \quad \frac{ts : \mathsf{Tel}\ n \quad t : \mathsf{Reg}\ n}{ts\!:\!t : \mathsf{Tel}\ (\mathsf{suc}\ n)}$$

We can now interpret every type expression without having to rename de Bruijn indicies at run time to account for the new context or substituting out to a closed expression. Notice that the inductive structure of $[\![-]\!]^{-}$ is *not* the inductive structure of Reg—the size of an element is not bounded by the size of its type.

$$\underline{\text{data}}\ \frac{\Gamma : \mathsf{Tel}\ n \quad T : \mathsf{Reg}\ n}{[\![T]\!]^{\Gamma} : \star}$$

$$\underline{\text{where}}\ \frac{t : [\![T]\!]^{\Gamma}}{\mathsf{top}\ t : [\![\text{`Z'}]\!]^{\Gamma:T}} \quad \frac{t : [\![T]\!]^{\Gamma}}{\mathsf{pop}\ t : [\![\text{`wk'}\ T]\!]^{\Gamma:S}} \quad \frac{t : [\![T]\!]^{\Gamma:S}}{\mathsf{def}\ t : [\![\text{`let'}\ S\ T]\!]^{\Gamma}}$$

$$\frac{s : [\![S]\!]^{\Gamma}}{\mathsf{inl}\ s : [\![S\ \text{`+'}\ T]\!]^{\Gamma}} \quad \frac{t : [\![T]\!]^{\Gamma}}{\mathsf{inr}\ t : [\![S\ \text{`+'}\ T]\!]^{\Gamma}}$$

$$\frac{}{\mathsf{void} : [\![\text{`1'}]\!]^{\Gamma}} \quad \frac{s : [\![S]\!]^{\Gamma} \quad t : [\![T]\!]^{\Gamma}}{\mathsf{pair}\ s\ t : [\![S\ \text{`}\times\text{'}\ T]\!]^{\Gamma}} \quad \frac{x : [\![F]\!]^{(\Gamma : \text{`}\mu\text{'}F)}}{\mathsf{in}\ x : [\![\text{`}\mu\text{'}\ F]\!]^{\Gamma}}$$

The telescopic environments behave as we promised. The rule for 'Z' interprets the top type T over the remaining Γ—but how did it get there? Either from a ‘let’ or a ‘μ’ extending Γ with a type which is defined over it! The rule for ‘wk’ just pops the environment. Most interesting is the definition of in, which uses the environment to expand the fixpoint—let us show how this behaves by coding the natural numbers:

$$\underline{\text{let}}\ \frac{}{\text{‘}\mathbf{Nat}\text{’} : \mathsf{Reg}\ n} \qquad \text{‘}\mathbf{Nat}\text{’} \Rightarrow \text{‘}\mu\text{’}\,(\text{‘}1\text{’}\ \text{‘}{+}\text{’}\ \text{‘}\mathsf{Z}\text{’})$$

$$\underline{\text{let}}\ \frac{}{\mathbf{ze} : [\![\text{‘}\mathbf{Nat}\text{’}]\!]^{\Gamma}} \qquad \underline{\text{let}}\ \frac{n : [\![\text{‘}\mathbf{Nat}\text{’}]\!]^{\Gamma}}{\mathbf{su}\ n : [\![\text{‘}\mathbf{Nat}\text{’}]\!]^{\Gamma}}$$

$$\mathbf{ze} \Rightarrow \mathsf{in}\ (\mathsf{inl}\ \mathsf{void}) \qquad \mathbf{su}\ n \Rightarrow \mathsf{in}\ (\mathsf{inr}\ (\mathsf{top}\ n))$$

The in constructor places a recursive copy of ‘$\mathbf{Nat}$’ on top of the telescope, which $\mathbf{su}$ invokes via top. We can program with ‘$\mathbf{Nat}$’ quite easily:

$$\underline{\text{let}}\ \frac{m, n : [\![\text{‘}\mathbf{Nat}\text{’}]\!]^{\Gamma}}{\mathbf{plus}\ m\ n : [\![\text{‘}\mathbf{Nat}\text{’}]\!]^{\Gamma}} \quad \begin{array}{llll} \mathbf{plus} & m & n & \Leftarrow \underline{\text{rec}}\ m \\ \quad\mathbf{plus} & (\mathsf{in}\ (\mathsf{inl}\ \mathsf{void})) & n & \Rightarrow n \\ \quad\mathbf{plus} & (\mathsf{in}\ (\mathsf{inr}\ (\mathsf{top}\ m))) & n & \Rightarrow \mathbf{su}\ (\mathbf{plus}\ m\ n) \end{array}$$

Note that the patterns on the left correspond to $\mathbf{ze}$ and $\mathbf{su}$. These cases are exhaustive—all the other constructors target types which conflict with the definition of ‘$\mathbf{Nat}$’, so Epigram dismisses them automatically. The recursive structure of the whole $[\![-]\!]^{-}$ family, thus specializes to that of $[\![\text{‘}\mathbf{Nat}\text{’}]\!]^{\Gamma}$.

Our recognizably inductive presentation contrasts with Benke, Dybjer and Jansson’s recursive definitions of the *functor* given by each code in a universe, whose least fixpoint is in turn the coded type. Whilst in all of their examples, it is clear to *us* that the computed functors are strictly positive and give rise to inductive types, they make no apparent attempt to explain this to Agda.

The space efficiency of $[\![-]\!]^{-}$ is a serious concern: on the face of it, each data constructor takes an environment and perhaps some type expressions as arguments. Even if sharing is preserved, this is particularly wasteful. Fortunately, as Brady has shown [8, 9], there is no need to duplicate in the data any information extractable from the type indices, so all of the Γ’s, S’s and T’s vanish, even from the open representation we need for partial evaluation in the typechecker.

Further, Brady’s work suggests that we can also remove constructor tags where these are determined by indices. In our case, this means that only elements of sums need to be tagged inl or inr, as each of the other type formers has at most one data former. Hence there is no need for an extra layer of indirection inside each top, pop, def or in. There is no reason why our explicit definition and weakening has to lead to a space penalty.

3 Deciding Equality

Every regular tree type can be given a decidable equality in a systematic way. In this section, we express that system as a program. Equality as a *Boolean test* has been a standard example of generic programming from PolyP onwards [21].

Benke, Dybjer and Jansson replay this construction in Agda [7] and, moreover, they prove generically that what is being tested really behaves like equality, in that it is reflexive and substitutive. We take a slightly different approach, given that Epigram has a built in equality type [27] which is reflexive by construction, and substitutive by case analysis:

$$\frac{a \,:\, A \quad b \,:\, B}{a{=}b \,:\, \star} \qquad \frac{}{\mathsf{refl} \,:\, a{=}a}$$

Rather than proving a Boolean test correct, we can exploit directly the type of *decisions*, which packs up either a proof or a refutation for a given proposition:

$$\underline{\text{data}}\ \frac{P \,:\, \star}{\mathsf{Decision}\ P \,:\, \star}\ \underline{\text{where}}\ \frac{y \,:\, P}{\mathsf{yes}\ y \,:\, \mathsf{Decision}\ P} \quad \frac{n \,:\, P \to \mathsf{Zero}}{\mathsf{no}\ n \,:\, \mathsf{Decision}\ P}$$

To decide the equality of x and y, and know that we have done so, we must show how to compute an inhabitant of $\mathsf{Decision}\ (x{=}y)$. We can get most of the way by analysing each element and inspecting the results of the recursive calls on corresponding subterms—see 1.

Again, dependent pattern matching ensures that we need only consider elements which have the same type. Fundamentally, all inequalities boil down to the fact that inl and inr are different: $\mathsf{inl}\ s = \mathsf{inr}\ t$ is an empty type, so case analysis leaves no branches. We have left $[\,]$s for most of the cases where we must show that recursive inequality of subterms breaks equality for the whole terms. Each of these proofs require an auxiliary definition all of which follow the same pattern. The proof for in is given below.

$$\underline{\text{let}}\ \frac{x, y \,:\, [\![F]\!]^{\Gamma :: (\text{`}\mu\text{'}\ F)} \quad n \,:\, (x{=}y) \to \mathsf{Zero} \quad q \,:\, (\mathsf{in}\ x{=}\mathsf{in}\ y)}{\mathbf{notEqIn}_{x\ y}\ n\ q \,:\, \mathsf{Zero}}$$

$$\mathbf{notEqIn}_{x\ x}\ n\ \mathsf{refl} \ \Rightarrow\ n\ \mathsf{refl}$$

4 Type Constructors and Generic Map

We can represent type constructors in our universe by type expressions with parameters, much as one does when one defined a polymorphic data structure in a functional programming language. For example, we can have

$$\underline{\text{let}}\ \frac{}{\text{`}\mathbf{List}\text{'} \,:\, \mathsf{Reg}\ (\mathsf{suc}\ n)} \quad \text{`}\mathbf{List}\text{'} \ \Rightarrow\ \text{`}\mu\text{'}\ (\text{`}\mathsf{1}\text{'}\ \text{`}{+}\text{'}\ (\text{`}\mathsf{wk}\text{'}\ \text{`}\mathsf{Z}\text{'}\ \text{`}{\times}\text{'}\ \text{`}\mathsf{Z}\text{'}))$$

We can then create specific instances of polymorphic structures by capturing the free parameter with ‘let’—the type of lists of natural numbers would then be coded by ‘let’ ‘**Nat**’ ‘**List**’. We can also develop polymorphic operations by

```
      x, y : ⟦T⟧^Γ
let ─────────────────────────────
    decEq x y : Decision (x=y)

decEq      x           y          ⇐ rec x
decEq     (def x)     (def y)    ‖ decEq x y
 decEq    (def x)     (def x)    |   yes refl     ⇒ yes refl
 decEq    (def x)     (def y)    |   no n         ⇒ no []
decEq     (top x)     (top y)    | decEq x y
 decEq    (top x)     (top x)    |   yes refl     ⇒ yes refl
 decEq    (top x)     (top y)    |   no n         ⇒ no []
decEq     (pop x)     (pop y)    | decEq x y
 decEq    (pop x)     (pop x)    |   yes refl     ⇒ yes refl
 decEq    (pop x)     (pop y)    |   no n         ⇒ no []
decEq      void        void       ⇒ yes refl
decEq     (inl sx)    (inl sy)   ‖decEq sx sy
 decEq    (inl s)     (inl s)    |   yes refl     ⇒ yes refl
 decEq    (inl sx)    (inl sy)   |   no sn        ⇒ no []
decEq     (inl sx)    (inr ty)    ⇒ no (λq ⇐ case q)
decEq     (inr tx)    (inl sy)    ⇒ no (λq ⇐ case q)
decEq     (inr tx)    (inr ty)   ‖decEq tx ty
 decEq    (inr t)     (inr t)    |   yes refl     ⇒ yes refl
 decEq    (inr tx)    (inr ty)   |   no tn        ⇒ no []
decEq   (pair sx tx) (pair sy ty)‖decEq sx sy
 decEq  (pair s tx)  (pair s ty) |   yes refl    ‖decEq tx ty
  decEq (pair s t)   (pair s t)  |   yes refl    |  yes refl    ⇒ yes refl
  decEq (pair s tx)  (pair s ty) |   yes refl    |  no tn       ⇒ no []
 decEq  (pair sx tx) (pair sy ty)|   no sn        ⇒ no []
decEq     (in x)      (in y)     ‖ decEq x y
 decEq    (in x)      (in x)     |   yes refl     ⇒ yes refl
 decEq    (in x)      (in y)     |   no n         ⇒ no (notEqIn n)
```

Fig. 1. Decidable Equality

working with open type expressions over a nonempty environment:

```
let ──────────────────          nil        ⇒ in (inl void)
    nil : ⟦‘List’⟧^Γ

    a : ⟦‘Z’⟧^Γ   as : ⟦‘List’⟧^Γ
let ──────────────────────────────  cons a as ⇒ in (inr (pair (pop a) (top as)))
       cons a as : ⟦‘List’⟧^Γ

      as, bs : ⟦‘List’⟧^Γ
let ──────────────────────────
    append as bs : ⟦‘List’⟧^Γ

    append              as                 bs ⇐ rec as
     append          (in (inl void))       bs ⇒ bs
     append (in (inr (pair (pop a) (top as)))) bs ⇒ cons a (append as bs)
```

Of course, to apply these polymorphic operations in specific cases, one must strip and apply the def constructor.

Let us now develop a generic polymorphic operation—functorial mapping. Suppose we have two environments $\mathit{\Gamma}$ and $\mathit{\Delta}$ interpreting the free type variables in an expression T (‘**List**’, for example). If we can translate between the values in the corresponding types in $\mathit{\Gamma}$ and $\mathit{\Delta}$, then we can map between $[\![T]\!]^{\mathit{\Gamma}}$ and $[\![T]\!]^{\mathit{\Delta}}$, preserving the structure due to T, but translating the data corresponding to the free type variables. Here, the fact that we represent the *syntax* of type expressions makes this task easy.

Let us define morphisms between environments and then show how to map them across polymorphic type expressions. We are careful to ensure that we can readily extend a morphism *uniformly* when we go under a binder.

$$\underline{\text{data}}\ \frac{\mathit{\Gamma}, \mathit{\Delta}\ :\ \mathsf{Tel}\ n}{\mathsf{Morph}\ \mathit{\Gamma}\ \mathit{\Delta}\ :\ \star}\ \underline{\text{where}}\ \frac{}{\mathsf{mId}\ :\ \mathsf{Morph}\ \mathit{\Gamma}\ \mathit{\Gamma}}$$

$$\frac{\phi\ :\ \mathsf{Morph}\ \mathit{\Gamma}\ \mathit{\Delta}\quad f\ :\ [\![S]\!]^{\mathit{\Gamma}} \to [\![T]\!]^{\mathit{\Delta}}}{\mathsf{mFun}\ \phi\ f\ :\ \mathsf{Morph}\ (\mathit{\Gamma}\!:\!S)\ (\mathit{\Delta}\!:\!T)}$$

$$\frac{\phi\ :\ \mathsf{Morph}\ \mathit{\Gamma}\ \mathit{\Delta}}{\mathsf{mMap}\ \phi\ :\ \mathsf{Morph}\ (\mathit{\Gamma}\!:\!T)\ (\mathit{\Delta}\!:\!T)}$$

We can now write our generic **gMap** operator by structural recursion on data. Each time we go under a binder, we extend the morphism with mMap, explaining that the type variable at that point is local. When we reach a variable, we look up the appropriate translation, using **gMap** to interpret mMap. In the case of the identity morphism, the environments are known to coincide, so no further traversal is necessary.

$$\underline{\text{let}}\ \frac{\phi\ :\ \mathsf{Morph}\ \mathit{\Gamma}\ \mathit{\Delta}\quad x\ :\ [\![T]\!]^{\mathit{\Gamma}}}{\mathbf{gMap}\ \phi\ x\ :\ [\![T]\!]^{\mathit{\Delta}}}$$

```
gMap   φ              x           ⇐ rec x
  gMap   φ            (def x)     ⇒ def (gMap (mmap φ) x)
  gMap   mId          (top x)     ⇒ top x
  gMap   (mFun φ f)   (top x)     ⇒ top (f x)
  gMap   (mMap φ)     (top x)     ⇒ top (gMap φ x)
  gMap   mId          (pop x)     ⇒ pop x
  gMap   (mFun φ f)   (pop x)     ⇒ pop (gMap φ x)
  gMap   (mMap φ)     (pop x)     ⇒ pop (gMap φ x)
  gMap   φ            (inl x)     ⇒ inl (gMap φ x)
  gMap   φ            (inr x)     ⇒ inr (gMap φ x)
  gMap   φ            void        ⇒ void
  gMap   φ            (pair x y)  ⇒ pair (gMap φ x) (gMap φ y)
  gMap   φ            (in x)      ⇒ in (gMap (mMap φ) x)
```

Instantiating **gMap** for our '**List**' example is straightforward

$$\underline{\text{let}}\ \frac{f : [\![S]\!]^{\Gamma} \to [\![T]\!]^{\Gamma} \quad as : [\![\text{'let'}\ S\ \text{'}\mathbf{List}\text{'}]\!]^{\Gamma}}{\mathbf{list}\ f\ as : [\![\text{'let'}\ T\ \text{'}\mathbf{List}\text{'}]\!]^{\Gamma}}$$
$$\mathbf{list}\ f\ (\mathsf{def}\ as) \Rightarrow \mathsf{def}\ (\mathbf{gMap}\ (\mathsf{mFun}\ \mathsf{mId}\ f)\ as)$$

Is this functorial mapping? An easy induction on x shows that

$$\mathbf{gMap}\ \mathsf{mId}\ x = x$$

but what about composition? Composition may be defined as follows

$$\underline{\text{let}}\ \frac{\phi : \mathsf{Morph}\ \Delta\ \Theta : \star \quad \psi : \mathsf{Morph}\ \Gamma\ \Delta : \star}{\phi \circ \psi : \mathsf{Morph}\ \Gamma\ \Theta}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\phi \circ \psi \Leftarrow \underline{\text{rec}}\ \phi} \\
\quad \mathsf{mId} & \circ\ \psi & \Rightarrow \psi \\
\quad \mathsf{mFun}\ \phi\ f & \circ\ \mathsf{mId} & \Rightarrow \mathsf{mFun}\ \phi\ f \\
\quad \mathsf{mFun}\ \phi\ f & \circ\ \mathsf{mFun}\ \psi\ g & \Rightarrow \mathsf{mFun}\ (\phi \circ \psi)\ (f \cdot g) \\
\quad \mathsf{mFun}\ \phi\ f & \circ\ \mathsf{mMap}\ \psi & \Rightarrow \mathsf{mFun}\ (\phi \circ \psi)\ (f \cdot \mathbf{gMap}\ \psi) \\
\quad \mathsf{mMap}\ \phi & \circ\ \mathsf{mId} & \Rightarrow \mathsf{mMap}\ \phi \\
\quad \mathsf{mMap}\ \phi & \circ\ \mathsf{mFun}\ \psi\ g & \Rightarrow \mathsf{mFun}\ (\phi \circ \psi)\ (\mathbf{gMap}\ \phi \cdot g) \\
\quad \mathsf{mMap}\ \phi & \circ\ \mathsf{mMap}\ \psi & \Rightarrow \mathsf{mMap}\ (\phi \circ \psi)
\end{array}$$

Another easy induction on x then shows that

$$\mathbf{gMap}\ (\phi \circ \psi)\ x = (\mathbf{gMap}\ \phi\ \cdot\ \mathbf{gMap}\ \psi)\ x$$

5 The Derivative and the Zipper

Formal differentiation of algebraic expressions was one of the first functional programs ever to be written in pattern matching style and executed on a computer [30]. Thirty-five years later we can run it again, but with a new meaning. As McBride observed [26], differentiating a regular tree type T with respect to a free variable X computes the type of *one-hole contexts* for a value from X in a value from T. The explanation of the derivative as coding for the linear part of a polymorphic function space between containers can be found in [3]. Here we show how this works out as code:

$$\underline{\text{let}}\ \frac{X : \mathsf{Fin}\ n \quad T : \mathsf{Reg}\ n}{\partial\ X\ T : \mathsf{Reg}\ n}$$

```
  ∂    X        T       ⇐ rec T
    ∂   fz       'Z'     ⇒ '1'
    ∂ (fs X)     'Z'     ⇒ '0'
    ∂   fz    ('wk' T)   ⇒ '0'
    ∂ (fs X)  ('wk' T)   ⇒ 'wk' (∂ X T)
    ∂    X   ('let' S T) ⇒     'let' S (∂ (fs X) T)
                           '+' 'let' S (∂ fz T) '×' ∂ X S
    ∂    X       '0'     ⇒ '0'
    ∂    X       '1'     ⇒ '0'
    ∂    X   (S '+' T)   ⇒ ∂ X S '+' ∂ X T
    ∂    X   (S '×' T)   ⇒ ∂ X S '×' T  '+'  S '×' ∂ X T
    ∂    X    ('μ' F)    ⇒     'μ' ('1' '+' 'Z' '×' 'wk' ('let' ('μ' F) (∂ fz F)))
                           '×' 'let' ('μ' F) (∂ (fs X) F)
```

Rules we learned from Leibniz take on a direct visual intuition: 'an S '+' T with a hole' is either 'an S with a hole' or 'a T with a hole'; 'an S '×' T with a hole' is either 'an S with a hole and a T' or 'an S and a T with a hole'. The *chain rule* for 'let' must account for each fs X directly in T as well as each X sitting inside an S via a fz in T—this notion of derivative is thus partial on the *free* variables and *total* on the bound variables. McBride added a new rule, inspired by Huet [20]—a one hole context inside an inductively defined container consists of a 'zipper' which wraps up the node where the hole is. Let us define a 'zipper':

$$\underline{\text{let}}\ \frac{F : \mathsf{Reg}\ (\mathsf{suc}\ n)}{\text{'}\mathbf{Zipper}\text{'}\ F : \mathsf{Reg}\ n}$$

```
'Zipper' F ⇒ 'μ' ('1' '+' 'Z' '×' 'wk' ('let' ('μ' F) (∂ fz F)))
```

A '**Zipper**' F is thus a stack of steps, each giving the context for a 'Z' inside an F, and hence a recursive subtree inside a 'μ' F. With this definition, we effectively have that $\partial\ X$ ('μ' F) is a node with a hole and a '**Zipper**' F.

Notice that it is our access to the full syntax of type expressions which enables us to differentiate types with multiple parameters, and hence local definitions and fixpoints. By contrast, programmers in Generic Haskell have no access to these syntactic details. Often this is a convenience, but here it restricts the treatment given by Hinze, Jeuring and Löh [18] to polynomials in one variable.

Let us now show how to plug a '**var**' X into a $\partial\ X\ T$, and a 'μ' F into a '**Zipper**' F. It is not hard to see that these two tasks are mutually recursive. We shall therefore need to dodge the problem that Epigram does not currently support mutually recursive functions. We do this in the obvious way, by turning the mutual definition into the definition of a *family*. First, we define the family of ‘pluggers’ for a type of contexts C with a hole type H, yielding output in O,

$$\underline{\text{data}}\ \frac{C, H, O\ :\ \mathsf{Reg}\ n}{\mathsf{Plugger}\ C\ H\ O\ :\ \star}\ \underline{\text{where}}\ \frac{X\ :\ \mathsf{Fin}\ n \quad T\ :\ \mathsf{Reg}\ n}{X \multimap T\ :\ \mathsf{Plugger}\ (\partial\ X\ T)\ (\text{‘}\mathbf{var}\text{’}\ X)\ T}$$

$$\frac{F\ :\ \mathsf{Reg}\ (\mathsf{suc}\ n)}{\circlearrowleft F\ :\ \mathsf{Plugger}\ (\text{‘}\mathbf{Zipper}\text{’}\ F)\ (\text{‘}\mu\text{’}\ F)\ (\text{‘}\mu\text{’}\ F)}$$

and then we explain how to interpret pluggers as operators, by recursion over the context—as long as we consume the context, we are free to ‘change mode’ when we need to. We start like this, by recursion on the task, then case analysis on the plugger:

$$\underline{\text{let}}\ \frac{p\ :\ \mathsf{Plugger}\ C\ H\ O \quad c\ :\ [\![C]\!]^{\Gamma} \quad h\ :\ [\![H]\!]^{\Gamma}}{c\ \langle p]\ h\ :\ [\![O]\!]^{\Gamma}} \qquad \begin{array}{ll} c\ \langle p]\ h & \Leftarrow \underline{\text{rec}}\ c \\ \quad c\ \langle X \multimap T]\ h & [\,] \\ \quad c\ \langle \circlearrowleft F]\ h & [\,] \end{array}$$

Now we can develop the two branches as if we were writing a mutual definition. We implement $\langle X \multimap T]$ as follows:

$$\begin{array}{llll}
\mathsf{void} & \langle \mathsf{fz} \multimap \text{‘}\mathsf{Z}\text{’}] & h & \Rightarrow\ h \\
c & \langle \mathsf{fs}\ X \multimap \text{‘}\mathsf{Z}\text{’}] & h & \Leftarrow \underline{\text{case}}\ c \\
c & \langle \mathsf{fz} \multimap \text{‘}\mathsf{wk}\text{’}\ T] & h & \Leftarrow \underline{\text{case}}\ c \\
\mathsf{pop}\ c & \langle \mathsf{fs}\ X \multimap \text{‘}\mathsf{wk}\text{’}\ T] & \mathsf{pop}\ h & \Rightarrow\ \mathsf{pop}\ (c\ \langle X \multimap T]\ h) \\
\mathsf{inl}\ (\mathsf{def}\ tc) & \langle X \multimap \text{‘}\mathsf{let}\text{’}\ S\ T] & h & \Rightarrow\ \mathsf{def}\ (tc\ \langle \mathsf{fs}\ X \multimap T]\ \mathsf{pop}\ h) \\
\mathsf{inr}\ (\mathsf{pair}\ (\mathsf{def}\ tc)\ sc) & \langle X \multimap \text{‘}\mathsf{let}\text{’}\ S\ T] & h & \Rightarrow\ \mathsf{def}\ (tc\ \langle \mathsf{fz} \multimap T] \\
 & & & \qquad\quad \mathsf{top}\ (sc\ \langle X \multimap S]\ h)) \\
c & \langle X \multimap \text{‘}0\text{’}] & h & \Leftarrow \underline{\text{case}}\ c \\
c & \langle X \multimap \text{‘}1\text{’}] & h & \Leftarrow \underline{\text{case}}\ c \\
\mathsf{inl}\ sc & \langle X \multimap S\ \text{‘}{+}\text{’}\ T] & h & \Rightarrow\ \mathsf{inl}\ (sc\ \langle X \multimap S]\ h) \\
\mathsf{inr}\ tc & \langle X \multimap S\ \text{‘}{+}\text{’}\ T] & h & \Rightarrow\ \mathsf{inr}\ (tc\ \langle X \multimap T]\ h) \\
\mathsf{inl}\ (\mathsf{pair}\ sc\ t) & \langle X \multimap S\ \text{‘}{\times}\text{’}\ T] & h & \Rightarrow\ \mathsf{pair}\ (sc\ \langle X \multimap S]\ h)\ t \\
\mathsf{inr}\ (\mathsf{pair}\ s\ tc) & \langle X \multimap S\ \text{‘}{\times}\text{’}\ T] & h & \Rightarrow\ \mathsf{pair}\ s\ (tc\ \langle X \multimap T]\ h) \\
\mathsf{pair}\ f\!f\ (\mathsf{def}\ fc) & \langle X \multimap \text{‘}\mu\text{’}\ F] & h & \Rightarrow\ f\!f\ \langle \circlearrowleft F] \\
 & & & \qquad \mathsf{in}\ (fc\ \langle \mathsf{fs}\ X \multimap F]\ \mathsf{pop}\ h)
\end{array}$$

Meanwhile, ‘zipping out’ iterates ‘plugging in’ tail recursively:

$$\begin{array}{llll}
\mathsf{in}\ (\mathsf{inl}\ \mathsf{void}) & \langle \circlearrowleft F] & h & \Rightarrow\ h \\
\mathsf{in}\ (\mathsf{inr}\ (\mathsf{pair}\ (\mathsf{top}\ f\!f)\ (\mathsf{pop}\ (\mathsf{def}\ fc)))) & \langle \circlearrowleft F] & h & \Rightarrow\ f\!f\ \langle \circlearrowleft F] \\
 & & & \qquad \mathsf{in}\ (fc\ \langle \text{‘}\mathsf{Z}\text{’} \multimap F]\ \mathsf{top}\ h)
\end{array}$$

This may look like a complicated definition, but we had some help to write it. The Epigram system calculates all the context types, not us: we just apply case analysis repeatedly on the contexts until the subcontexts appear. The only real choice we must make is whether $\langle \circlearrowleft F]$ should read its context as ‘hole-to-root’ or ‘root-to-hole’. Here, following Huet, we choose the former, shrinking the context and growing the subtree as we follow the path.

6 Conclusions and Further Work

In this paper, we have constructed the universe of regular tree types, closed under polynomials and least fixpoints; we equipped its syntax with an inductively defined semantics. By dependent pattern matching and structural recursion on data, we implemented a generic decision procedure for equalities on regular types and functorial mapping. We equipped the regular tree types with their differential structure, generalising 'the zipper'. We have given a tractable coding to these generic tasks without the assistance of any peculiar extensions to Epigram. Ordinary programming suffices to get us this far, and while it is inevitably harder work than *using* tools dedicated to a specific universe, it is undeniably less work than *making* those tools. A dependently typed language allows a flexible approach to programming with universes of many characters, large and small.

Perhaps we should remark on the technology which makes this approach practicable—dependent pattern matching with inductive families of datatypes. We have quietly exploited multiple layers of dependency, with equality types indexed by data from interpretations indexed by telescopes and type expressions indexed by numbers, for example, and we have not had to lift a finger to push the pattern matching through. This kind of deep dependency takes us well beyond the familiar world of inductive relations indexed by simply typed data, but it is nothing to be frightened of, given suitable tools.

There is a great deal of work yet to do. Whilst we have programmed generically in a small and *ad hoc* universe, we have not developed generic programming *for Epigram.* We should pursue the agenda set by Pfeifer and Rueß [32] to acquire generic programs and proofs for the types we *use*, not just those we *model.*

This is even more vital for dependently typed programming than it is for 'ordinary' functional programming because we tend to tailor data structures more closely to the specific properties we need for a given task. We might have sized lists, sorted lists, telescopes or transitive closures where once we just had lists—the extra detail may be just what we need for a particular problem, but it should not come at the expense of rebuilding the list library for each variation. Generic programming can potentially help us in two ways. We can seek to develop operations which work generically for all list-like types, or all concrete syntaxes, or whatever classes of structure we can characterise. We can also seek to roll out structure such as sizing or sortedness across a broad universe of datatypes.

Correspondingly, we need a representation of data structures which directly and compositionally describes inductive families in Epigram, in much the way that indexed induction-recursion [17] gives an account of data structures in Agda. A promising approach is based on the uniform representation of strictly positive structures as *containers* [1, 2]. These extend readily to dependent structures, and are closed under a fixed grammar of combinators including least and greatest fixpoints. We need the system to automate the quotation of data structures in this grammar and the maps in and out of their container form—much the way Generic Haskell [10] relates each Haskell datatype with the 'structure types' over which generic programs actually compute. Subsets of this general grammar then give us codes for smaller universes with more specific structure. If we can standardise our reflection of data structures in this way, then we really shall have reduced generic programming to ordinary programming.

References

1. Michael Abbott. *Categories of Containers*. PhD thesis, University of Leicester, 2003.
2. Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In *Proceedings of Foundations of Software Science and Computation Structures*, 2003.
3. Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. ∂ for data: derivatives of data structures. *Fundamenta Informaticae*, 65(1,2):1–128, March 2005.
4. 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.
5. Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda-terms using generalized inductive types. In *Computer Science Logic 1999*, 1999.
6. Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic Programming—An Introduction. In S. Doaitse Sweierstra, Pedro R. Henriques, and José N. Oliveira, editors, *Advanced Functional Programming, Third International Summer School (AFP '98); Braga, Portugal*, volume 1608 of *LNCS*, pages 28–115. Springer-Verlag, 1998.
7. 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.
8. Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.
9. 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.
10. Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001.
11. L'Équipe Coq. The Coq Proof Assistant Reference Manual. `http://pauillac.inria.fr/coq/doc/main.html`, 2001.
12. Catarina Coquand and Thierry Coquand. Structured Type Theory. In *Workshop on Logical Frameworks and Metalanguages*, 1999.
13. 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.
14. Nicolas G. de Bruijn. Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation. *Indagationes Mathematicæ*, 34:381–392, 1972.
15. Nicolas G. de Bruijn. Telescopic Mappings in Typed Lambda-Calculus. *Information and Computation*, 91:189–204, 1991.
16. Peter Dybjer. Inductive Sets and Families in Martin-Löf's Type Theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*. CUP, 1991.

17. Peter Dybjer and Anton Setzer. Indexed induction-recursion. In Reinhard Kahle, Peter Schroeder-Heister, and Robert F. Stärk, editors, *Proof Theory in Computer Science*, volume 2183 of *Lecture Notes in Computer Science*, pages 93–113. Springer, 2001.
18. Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programmming*, 51:117–151, 2004.
19. Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular Expression Types for XML. In *Proceedings of the International Conference on Functional Programming*, 2000.
20. Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
21. Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Proceedings of POPL '97*, pages 470–482. ACM, 1997.
22. Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press, 1994.
23. Zhaohui Luo and Robert Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-211, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.
24. 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.
25. Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis·Napoli, 1984.
26. Conor McBride. The Derivative of a Regular Type is its Type of One-Hole Contexts. Available at `http://www.dur.ac.uk/c.t.mcbride/diff.ps`, 2001.
27. 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.
28. 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.
29. Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.
30. Fred McBride. *Computer Aided Manipulation of Symbols.* PhD thesis, Queen's University of Belfast, 1970.
31. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML, revised edition.* MIT Press, 1997.
32. Holger Pfeifer and Harald Rueß. Polytypic Proof Construction. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, , and L. Théry, editors, *Proc. 12th Intl. Conf. on Theorem Proving in Higher Order Logics*, number 1690 in Lecture Notes in Computer Science, pages 55–72. Springer-Verlag, September 1999.
33. Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's type theory: an introduction.* Oxford University Press, 1990.