Why Dependent Types Matter

Thorsten Altenkirch Conor McBride
The University of Nottingham
{txa,ctm}@cs.nott.ac.uk

James McKinna
The University of St Andrews
james.mckinna@st-andrews.ac.uk

Abstract

We exhibit the rationale behind the design of Epigram, a dependently typed programming language and interactive program development system, using refinements of a well known program—merge sort—as a running example. We discuss its relationship with other proposals to introduce aspects of dependent types into functional programming languages and sketch some topics for further work in this area.

1. Introduction

Types matter. That's what they're for—to classify data with respect to criteria which matter: how they should be stored in memory, whether they can be safely passed as inputs to a given operation, even who is allowed to see them. Dependent types are types expressed in terms of data, explicitly relating their inhabitants to that data. As such, they enable you to express more of what matters about data. While conventional type systems allow us to validate our programs with respect to a fixed set of criteria, dependent types are much more flexible, they realize a continuum of precision from the basic assertions we are used to expect from types up to a complete specification of the program's behaviour. It is the programmer's choice to what degree he wants to exploit the expressiveness of such a powerful type discipline. While the price for formally certified software may be high, it is good to know that we can pay it in installments and that we are free to decide how far we want to go. Dependent types reduce certification to type checking, hence they provide a means to convince others that the assertions we make about our programs are correct. Dependently typed programs are, by their nature, proof carrying code [NL96, HST$^+$03].

Functional programmers have started to incorporate many aspects of dependent types into novel type systems using *generalized algebraic data types* and *singleton types*. Indeed, we share Sheard's vision [She04] of closing the *semantic gap* between programs and their properties. While Sheard's language Ωmega approaches this goal by an evolutionary step from current functional languages like Haskell, we are proposing a more radical departure with Epigram, exploiting what we have learnt from proof development tools like LEGO and COQ.

Epigram is a full dependently typed programming language defined by McBride and McKinna [MM04], drawing on experience with the LEGO system. McBride has implemented a prototype which is available together with basic documentation [McB04, McB05] from the Epigram homepage.[1] The prototype implements most of the features discussed in this article, and we are continuing to develop it to close the remaining

[1] Currently `http://sneezy.cs.nott.ac.uk/epigram/`.

gaps, improve performance and add new features. Brady has implemented a compiler [Bra05, BMM04] for the Epigram language, providing important technology for producing efficient code from dependently typed programs.

$$\underline{\text{data}}\ \frac{}{\mathsf{Nat}\ :\ \star}\ \ \underline{\text{where}}\ \ \frac{}{\mathsf{0}\ :\ \mathsf{Nat}}\ \ \frac{n\ :\ \mathsf{Nat}}{\mathsf{1+}\ n\ :\ \mathsf{Nat}}$$

$$\underline{\text{data}}\ \frac{}{\mathsf{Order}\ :\ \star}\ \ \underline{\text{where}}\ \ \frac{}{\mathsf{le}, \mathsf{ge}\ :\ \mathsf{Order}}$$

$$\underline{\text{data}}\ \frac{X\ :\ \star}{\mathsf{List}\ X\ :\ \star}\ \ \underline{\text{where}}\ \ \frac{}{\mathsf{nil}\ :\ \mathsf{List}\ X}\ \ \frac{x\ :\ X\quad xs\ :\ \mathsf{List}\ X}{x :: xs\ :\ \mathsf{List}\ X}$$

$$\underline{\text{let}}\ \frac{x, y\ :\ \mathsf{Nat}}{\mathbf{order}\ x\ y\ :\ \mathsf{Order}}\qquad \begin{array}{llll} \mathbf{order} & x & y & \Leftarrow \underline{\text{rec}}\ x \\ \quad\mathbf{order} & \mathsf{0} & y & \Rightarrow \mathsf{le} \\ \quad\mathbf{order} & (\mathsf{1+}x') & \mathsf{0} & \Rightarrow \mathsf{ge} \\ \quad\mathbf{order} & (\mathsf{1+}x') & (\mathsf{1+}y') & \Rightarrow \mathbf{order}\ x'\ y' \end{array}$$

$$\underline{\text{let}}\ \frac{xs, ys\ :\ \mathsf{List}\ \mathsf{Nat}}{\mathbf{merge}\ xs\ ys\ :\ \mathsf{List}\ \mathsf{Nat}}$$

$$\begin{array}{llll} \mathbf{merge} & xs & ys & \Leftarrow \underline{\text{rec}}\ xs \\ \quad\mathbf{merge} & \mathsf{nil} & ys & \Rightarrow ys \\ \quad\mathbf{merge} & (x :: xs') & ys & \Leftarrow \underline{\text{rec}}\ ys \\ \qquad\mathbf{merge} & (x :: xs') & \mathsf{nil} & \Rightarrow xs \\ \qquad\mathbf{merge} & (x :: xs') & (y :: ys') & \left|\ \begin{array}{ll} \mathbf{order}\ x\ y & \\ \quad \mathsf{le} & \Rightarrow x :: \mathbf{merge}\ xs'\ ys \\ \quad \mathsf{ge} & \Rightarrow y :: \mathbf{merge}\ xs\ ys' \end{array}\right. \end{array}$$

$$\underline{\text{let}}\ \frac{xs\ :\ \mathsf{List}\ X}{\mathbf{deal}\ xs\ :\ \mathsf{List}\ X\ \times\ \mathsf{List}\ X}\qquad \begin{array}{lll} \mathbf{deal} & xs & \Leftarrow \underline{\text{rec}}\ x \\ \quad\mathbf{deal} & \mathsf{nil} & \Rightarrow (\mathsf{nil};\ \mathsf{nil}) \\ \quad\mathbf{deal} & (x :: \mathsf{nil}) & \Rightarrow (x :: \mathsf{nil};\ \mathsf{nil}) \\ \quad\mathbf{deal} & (y :: z :: xs) & \left|\ \begin{array}{l} \mathbf{deal}\ xs \\ (ys;\ zs)\ \Rightarrow\ (y :: ys;\ z :: zs) \end{array}\right. \end{array}$$

$$\underline{\text{let}}\ \frac{xs\ :\ \mathsf{List}\ \mathsf{Nat}}{\mathbf{sort}\ xs\ :\ \mathsf{List}\ \mathsf{Nat}}\qquad \begin{array}{l} \mathbf{sort}\ xs \Leftarrow \underline{\text{general}} \\ \quad\left|\ \begin{array}{ll} \quad\mathbf{deal}\ xs & \\ \quad(ys;\ \mathsf{nil}) & \Rightarrow ys \\ (ys;\ z :: zs) & \Rightarrow \mathbf{merge}\ (\mathbf{sort}\ ys)\ (\mathbf{sort}\ (z :: zs)) \end{array}\right. \end{array}$$

Figure 1. Merge-sort, generally

In this article we exhibit the rationale behind Epigram's design, using refinements of a well known program—merge sort—as a running example. Our starting point is the implementation shown in Figure 1: it is written in Epigram, but it could have been written in any functional language. We start by revisiting the question of totality versus partiality in section 3, showing how **sort** can be made structurally recursive. Section 4 continues by addressing the problem of how to maintain static invariants which is illustrated by implementing a sized **sort**. In section 5 we how to use dependent types to maintain static invariants about dynamic data, which is illustrated by implementing a version of **sort** which certifies that its output is in order. We look behind the curtains of the Epigram system in section 6 and discuss how dependent types support an extensible system of programming patterns which include, but are not restricted to, constructor case analysis and constructor guarded recursion; we also discuss the design of the interactive aspect of the Epigram programming environment. Finally, we describe areas of further work in 7 and summarize our conclusions in section 8.

Before embarking on this programme let's quickly describe Epigram's syntax (which may look unusual to functional programmers who have grown up with languages like ML or Haskell), taking our implementation of merge sort as an example. Epigram uses a two-dimensional syntax for declarations, based on natural deduction rules, a choice which pays off once type dependencies become more intricate. E.g. the declaration of the constructor $1+$ for the datatype[2] Nat is equivalent to writing $1+ \;:\; \mathsf{Nat} \to \mathsf{Nat}$. The conclusion of a declaration visualizes its use—this is important because parameters of any type can be implicit in Epigram. For example, when declaring nil we do not declare X—Epigram figures out from its usage that it must inhabit $\star$, the type of types and internally completes the declaration

$$\frac{X \;:\; \star}{\mathsf{nil}_X \;:\; \mathsf{List}\ X}$$

When presented with nil, Epigram uses the well known technology established by Damas and Milner [DM82] to figure out the value of this implicit parameter. We can also make an implicit parameter explicit by writing it as a subscript, eg., $\mathsf{nil}_{\mathsf{Nat}} \;:\; \mathsf{List\ Nat}$.

Epigram programs are tree-structured: each node has a 'left-hand side' indicating a *programming problem* and a 'right hand side' indicating either how to solve it outright by returning a value, $\Rightarrow t$ ($\Rightarrow$ is pronounced 'return'), or how to reduce it to subproblems by deploying a programming pattern indicated by the $\Leftarrow$ symbol ($\Leftarrow$ is pronounced 'by'). Programming patterns include structural recursion, like $\underline{\text{rec}}\ x$ in **order**, $\underline{\text{general}}$ recursion and also case analysis. We suppress nonempty case analyses for the sake of brevity—they can be recovered by the standard algorithm [Aug85]. If, as in the **merge** function, we need to analyse the result of an intermediate computation, we bring it to the left with the $|cdots$ construct ($|$ is pronounced 'with').[3] Here, **order** decides the $\leq$ relation, returning a value in the enumeration Order.

We have given a complete program, but Epigram can also typecheck and evaluate incomplete programs with unfinished sections sitting in *sheds*, $[\ \cdots\ ]$, where the typechecker is forbidden to tread. Programs can be developed interactively, with the machine showing the available context and the required type, wherever the cursor may be. Moreover, it is Epigram which generates the left-hand sides of programs from type information, each time a problem is simplified with $\Leftarrow$ on the right.

2. Related Work

Dependent types are a central feature of Martin-Löf's Type Theory [ML84, NPS90], integrating constructive logic and strongly typed functional programming. Type Theory and its impredicative extension, the Calculus of Constructions, inspired many type-based proof systems, such as NUPRL [CAB+86], LEGO [LP92] and the widely used COQ [Tea04]. Magnusson's ALF system [MN94] was not only the first system to implement inductive families [Dyb91] and pattern matching for dependent types [Coq92] and it also pioneered the interactive style of type-driven program and proof development which inspired Epigram.

Xi and Pfenning's DML (for Dependent ML) [XP99] was an impressive experiment in exploiting dependent types for a real functional programming language. DML, like other *Applied Type Systems* [Xi04], separates the world of indexing expressions and programs, thereby keeping types unaffected from potentially badly behaved programs. In contrast to DML, Augustsson's implementation of the Cayenne language [Aug98], which also inspired the AGDA proof system [CC99], uses full dependency and doesn't differentiate between static and dynamic types.

[2] While the declaration of Nat provides a convenient interface to the type of natural numbers, there is no need to implement them using a unary representation internally. Moreover, we shall also exploit the syntactic convenience of using the usual decimal notation to refer to elements of Nat.

[3] The current prototype doesn't yet support the suppression of $\underline{\text{case}}$ and it doesn't implement the $|$ notation. As a consequence its code is more verbose.

Nested types provide a poor man's approach to many indexed data structures such as square matrices or well scoped λ-terms, e.g. see [Hin01]. McBride suggested a more general approach [McB02b], exploiting that Haskell's class system provides a static logic programming language. Having realized the power of indexed data structures, Cheney and Hinze [CH03] proposed to extend the type system to introduce a rich language of index expressions leading to *generalized algebraic datatypes* (GADTs), which are basically inductive families in a programming language setting [She04, PWW04].

3. Why we are Partial to Totality

A popular myth, still to be found in learned articles and referee reports on grant applications, is that dependent types and general recursion do not mix. This is a misunderstanding, but it's an understandable one. Let us examine the facts, beginning with the typing rule for application:

$$\frac{f\ :\ \forall x : S \Rightarrow T[x] \quad s\ :\ S}{f\ s\ :\ T[s]}$$

It's clear from the premises that, as ever, to check an application we need to compare the function domain and the argument type. It's also clear from the rule's conclusion that these types may contain expressions. If computation is to preserve typings, then $f\ (2+2)$ should have the same type as $f\ 4$, so $T[2+2]$ must *be* the same type as $T[4]$. To decide typechecking, we therefore need to decide some kind of equivalence *up to computation*. There are various approaches to this problem.

The totalitarian approach. Some proof systems based on intensional type theory, including Coq and Lego, forbid general recursion. As all computations terminate, equality of types is just syntactic equality of their normal forms. Decidability of typechecking is a consequence, but it's not the primary motivation for this choice. As a proof method, general recursion is wholly bogus—its type, $\forall P \Rightarrow (P \rightarrow P) \rightarrow P$ is a blatant lie. General recursion, non-exhaustive patterns and other such non-total programming features compromise the logical soundness of a proof system. Trust is more important than termination in proof checking.

Of course, even in the absence of general recursion, it's possible to write programs which take a long time—e.g. checking all the basic configurations of four-colouring problems. That doesn't make dependent typechecking necessarily intractable: the complexity of the programs in types is entirely controlled by the programmer—the more you say, the more you pay, but the more also you can be repaid in terms of genericity, or precision, or brevity. Georges Gonthier's proof of the four colour theorem [Gon04] is made *tractable* by type-level computation, because it lets him avoid generating and checking a separate proof for each configuration—the latter approach would have involved at least as much work for the computer and a great deal more work for Georges!

The libertarian approach. It's reasonable to allow arbitrary recursion in type-level programs, provided you have some sort of cut-off mechanism which interrupts loops when they happen in practice. This is the approach taken by the Agda proof system, Cayenne and by Haskell with 'undecidable instances'—Haskell's overloading resolution amounts to executing a kind of 'compile-time Prolog'. Agda restores logical soundness by a separate termination check, performed after typechecking. The basic point is that you include recursive programs in types at your own risk: mostly they're benign and typechecking behaves sensibly.

The legendary 'loopiness' of dependent typechecking stems from the particular way the libertarian approach was implemented in Cayenne. It's perfectly reasonable to implement recursion via fixpoints in the value-only run-times of functional programming languages, but Lennart Augustsson's attempt to lift this to the open terms used in dependent typechecking had an unintended consequence—when even a structurally recursive function is stuck on a non-constructor input, you can still expand the fixpoint, potentially putting the system into a spin: this is intolerable, but it's not inevitable, as the other 'libertarian' systems have shown.

The pragmatic advantage of libertarianism is that we don't have to care why a program works in order to start playing with it—it seems a shame to ban certain programs at run-time just to protect ourselves at compile-time. However, it also seems a shame to forsake the certainties which totalitarianism supports.

The mixed economy. As both of the above have their merits, it seems sensible to support both, as long as programs are clear about where they stand. Dependent types can be used to express schemes of recursion in the form of induction principles, such as constructor-guarded recursion for a given inductive datatype. In Epigram, programs invoke recursion schemes explicitly—each recursive call must be translatable to an 'inductive hypothesis' arising from such a scheme. Any program which employs only constructor-guarded recursion, or some fancier scheme derived therefrom, is guaranteed to be total.

However, general recursion also takes the form of an induction principle (with a particularly generous inductive hypothesis). We can make general recursion available, but only by explicit appeal to this principle. If we suppress the compile-time computational behaviour of general recursion, we can preserve decidability of typechecking, leaving computation by explicitly total programs activated.

This opens a range of possibilities, as shown by the implementation of merge-sort shown in figure 1. The **order**ing of the natural numbers with respect to $\leq$ is a one-step constructor guarded recursion. To **merge** two sorted lists, we need a combination of one-step recursions on each argument—for each successive element of the first list, we step along the second list until its rightful place appears. The **deal**ing out of a list into two lists of (roughly) half the length here exploits a two-step recursion, but this still fits within the constructor-guarded scheme indicated by the keyword $\underline{\text{rec}}$. However, **sort** performs a peculiar recursion via **deal**—it's not obvious yet how to justify this, so for now we give up and use $\underline{\text{general}}$.[4]

This approach rules nothing out, but it still allows us to observe guaranteed totality where we can see the explanation. The notational overhead is not large and could be reduced still further if we were to install an Agda-style termination checker, inferring simple explanations when the user omits them. We could go even further, pushing the total-versus-general distinction into types by treating general recursion as an effect which we lock away safely in a monad. For practical purposes, we should need a better notation for monadic programming in the *functional* style. Both of these are active topics of research.

3.1 Totality is Good for more than the Soul

The warm fuzzy feeling you get when you've persuaded your program to live in a total programming language should not be underestimated. It's a strong static guarantee—you can say that you've written a *function* without having to pretend that $\perp$ is a value. But warm fuzzy feelings don't pay the rent: what are the practical benefits of virtue?

Randy Pollack has often said to us 'the point of writing a proof in a strongly normalizing calculus is that you don't need to normalize it'. When you have an expression of a given type in a total language, you can guarantee that it will compute to a value: if you don't care what that value is—as is usually the case with a proof—you have no need to perform the computation. Now we know that we can integrate proofs of logical properties into our programs at *no run-time cost*.

This is particularly important with proofs of equations. Equality is defined as follows:

$$\underline{\text{data}}\ \frac{s : S \quad t : T}{s = t : \star}\ \underline{\text{where}}\ \frac{}{\mathsf{refl} : t = t}$$

An equation between types induces a coercion from one to the other which is trivial if the proof is refl.

$$\underline{\text{let}}\ \frac{Q : S = T \quad s : S}{\{Q\}s : T} \qquad \{\mathsf{refl}\}t \;\Rightarrow\; t$$

In a partial setting, we need to run Q to check that it's refl, because trusting a false equation (like $\mathsf{Nat} = \mathsf{Nat} \to \mathsf{Nat}$) induces a run-time type error. When Q is total, the compiler can erase $\{Q\}$. Contrast this with the proposal to represent type equations in Haskell by isomorphisms, e.g. [BS02], — even though good programmers always try to ensure that these functions turn out at run-time to be functorial liftings of `id`, there is no way to guarantee this to the compiler, so the isomorphisms must actually be executed.

[4] To see how subtle the justification can be, try swapping the case analysis on **deal** xs so the patterns are $(\mathsf{nil};\ ys)$ and $(z :: zs;\ ys)$.

Moreover, an optimising compiler can exploit totality in various useful ways. The evaluation strategy of a total program is irrelevant, so they can be run as strictly or lazily as a heuristic analysis might suggest. Optimisations like replacing `foldr` with `foldl`, which only work with finite values, can be applied safely. Specialisation by partial evaluation is untroubled by $\perp$. Further the explicit marking of a program as structurally recursive is a clear invitation to apply fusion techniques.

All in all, there is no bliss to be had from ignorance of totality; there is no disadvantage to wisdom.

3.2 Defusing General Recursion

A recursive function which happens to be total will generally exploit some sort of structure, but perhaps not the 'native' structure of its inductive arguments. The totality of the function can be made clear if that structure, whatever it is, can be brought into the open and expressed inductively.

A typical 'divide and conquer' recursion often becomes structural by introducing an intermediate data structure which represents the division of the input, built by a process of insertion, and collapsed structurally by 'conquering'. This intermediate data structure thus corresponds to the control structure built by the original recursion, which can be reconstructed by fusing the building with the collapse.

As David Turner observed [Tur95], defusing quick-sort exposes the binary search tree structure. The standard example of a non-structural program is actually tree-sort—Rod Burstall's first example of a structural program [Bur69]!

We play the same game with merge-sort in figure 2. The 'divide' phase deals out the input to each sort into two inputs for sub-sorts (roughly) half the size; the 'conquer' phase merges the two sorted outputs into one (roughly) twice the size. If we build a tree representing the sorting processes, we find that each node deals its inputs fairly to its subnodes, with the leaves having none or one.

$$\underline{\text{data}}\ \frac{}{\mathsf{Parity} : \star}\ \underline{\text{where}}\ \frac{}{\mathsf{p0}, \mathsf{p1} : \mathsf{Parity}}$$

$$\underline{\text{data}}\ \frac{X : \star}{\mathsf{DealT}\ X : \star}\ \underline{\text{where}}\ \frac{}{\mathsf{empT} : \mathsf{DealT}\ X}\quad \frac{x : X}{\mathsf{leafT}\ x : \mathsf{DealT}\ X}\quad \frac{p : \mathsf{Parity}\quad l, r : \mathsf{DealT}\ X}{\mathsf{nodeT}\ p\ l\ r : \mathsf{DealT}\ X}$$

$$\underline{\text{let}}\ \frac{x : X\quad t : \mathsf{DealT}\ X}{\mathbf{insertT}\ x\ t : \mathsf{DealT}\ X}\quad \begin{array}{llll} \mathbf{insertT}\ x & t & \Leftarrow & \underline{\text{rec}}\ t \\ \quad \mathbf{insertT}\ x & \mathsf{empT} & \Rightarrow & \mathsf{leafT}\ x \\ \quad \mathbf{insertT}\ x & (\mathsf{leafT}\ y) & \Rightarrow & \mathsf{nodeT}\ \mathsf{p0}\ (\mathsf{leafT}\ y)\ (\mathsf{leafT}\ x) \\ \quad \mathbf{insertT}\ x & (\mathsf{nodeT}\ \mathsf{p0}\ l\ r) & \Rightarrow & \mathsf{nodeT}\ \mathsf{p1}\ (\mathbf{insertT}\ x\ l)\ r \\ \quad \mathbf{insertT}\ x & (\mathsf{nodeT}\ \mathsf{p1}\ l\ r) & \Rightarrow & \mathsf{nodeT}\ \mathsf{p0}\ l\ (\mathbf{insertT}\ x\ r) \end{array}$$

$$\underline{\text{let}}\ \frac{xs : \mathsf{List}\ X}{\mathbf{dealT}\ xs : \mathsf{DealT}\ X}\quad \begin{array}{llll} \mathbf{dealT} & xs & \Leftarrow & \underline{\text{rec}}\ xs \\ \quad \mathbf{dealT} & \mathsf{nil} & \Rightarrow & \mathsf{empT} \\ \quad \mathbf{dealT} & (x :: xs) & \Rightarrow & \mathbf{insertT}\ x\ (\mathbf{dealT}\ xs) \end{array}$$

$$\underline{\text{let}}\ \frac{t : \mathsf{DealT}\ \mathsf{Nat}}{\mathbf{mergeT}\ t : \mathsf{List}\ \mathsf{Nat}}\quad \begin{array}{llll} \mathbf{mergeT} & t & \Leftarrow & \underline{\text{rec}}\ t \\ \quad \mathbf{mergeT} & \mathsf{empT} & \Rightarrow & \mathsf{nil} \\ \quad \mathbf{mergeT} & (\mathsf{leafT}\ x) & \Rightarrow & x :: \mathsf{nil} \\ \quad \mathbf{mergeT} & (\mathsf{nodeT}\ p\ l\ r) & \Rightarrow & \mathbf{merge}\ (\mathbf{mergeT}\ l)\ (\mathbf{mergeT}\ r) \end{array}$$

$$\underline{\text{let}}\ \frac{xs : \mathsf{List}\ \mathsf{Nat}}{\mathbf{sort}\ xs : \mathsf{List}\ \mathsf{Nat}}\quad \mathbf{sort} \Rightarrow \mathbf{mergeT} \cdot \mathbf{dealT}$$

Figure 2. Merge-sort, structurally (with **merge** as before)

Correspondingly, a 'dealing' is a binary tree with leaves of weight zero or one, and nodes off balance by at most one—if we keep a parity bit at each node, we shall know into which subnode the next element should be dealt. In effect, we defuse the general recursion as a composition of folds, with 'divide' replacing $[\mathsf{nil}, (\mathbin{:})]$ by $[\mathsf{empT}, \mathbf{insertT}]$ and 'conquer' ($\mathbf{mergeT}$) replacing $[\mathsf{empT}, \mathsf{leafT}, \mathsf{nodeT}]$ by $[\mathsf{nil}, (\mathbin{:}\ \mathsf{nil}), \lambda p \Rightarrow \mathbf{merge}]$.

Of course, there is no panacea: there are all sorts of ways to write programs which conceal the structures by which they operate. A dependent type system provides a rich language of datatypes in which to expose these structures—other examples include evaluation for the simply typed λ-calculus with primitive recursion on Nat, which gives a denotational semantics to both types and terms, and first-order unification, which combines recursion on the number of available variables with recursion on terms over those variables.

If you care about totality, it's often easier to write a new program which works with the relevant structure than to write a proof which finds the structure which a <u>general</u>-program is hiding. The best way to tidy up the mess is not to make it in the first place, if you can possibly avoid it.

4. Maintaining Invariants by Static Indexing

An important aspect of many recent and innovative type systems is the idea of *indexing* datatypes in order to express and enforce structural invariants. There are various ways this can be achieved: in Epigram, we define *datatype families* in the style of the Alf system [Dyb91]. A good introductory example is given by the *vectors*—lists indexed with their length.

$$\underline{\text{data}}\ \frac{n : \mathsf{Nat} \quad X : \star}{\mathsf{Vec}\ n\ X : \star}\ \ \underline{\text{where}}\ \ \frac{}{\mathsf{vnil} : \mathsf{Vec}\ 0\ X}\ \ \frac{x : X \quad xs : \mathsf{Vec}\ n\ X}{\mathsf{vcons}\ x\ xs : \mathsf{Vec}\ (1{+}n)\ X}$$

Case analysis on inductive families [Coq92] involves *unifying* the type of the scrutinee with the type of each possible constructor pattern—those patterns for which constructors clash are rejected as impossible, as in this notorious example:

$$\underline{\text{let}}\ \frac{xs : \mathsf{Vec}\ (1{+}n)\ X}{\mathbf{vtail}\ xs : \mathsf{Vec}\ n\ X} \qquad \mathbf{vtail}\ (\mathsf{vcons}\ x\ xs) \ \Rightarrow\ xs$$

Vectors admit operations which enforce and maintain length invariants, such as this 'vectorized application':

$$\underline{\text{let}}\ \frac{fs : \mathsf{Vec}\ n\ (S \to T) \quad ss : \mathsf{Vec}\ n\ S}{fs \mathbin{^{@}} ss : \mathsf{Vec}\ n\ T} \qquad \begin{array}{l} fs \mathbin{^{@}} ss \ \Leftarrow\ \underline{\text{rec}}\ fs \\ \quad \begin{array}{lcl} \mathsf{vnil} \mathbin{^{@}} \mathsf{vnil} & \Rightarrow & \mathsf{vnil} \\ \mathsf{vcons}\ f\ fs' \mathbin{^{@}} \mathsf{vcons}\ s\ ss' & \Rightarrow & \mathsf{vcons}\ (f\ s)\ (fs' \mathbin{^{@}} ss') \end{array} \end{array}$$

Sometimes, we need some operations on the indices in order to express the type of an operation on indexed data. Concatenating vectors is a simple example

$$\underline{\text{let}}\ \frac{m, n : \mathsf{Nat}}{m + n : \mathsf{Nat}} \qquad \begin{array}{l} m + n \ \Leftarrow\ \underline{\text{rec}}\ m \\ \quad \begin{array}{lcl} 0 + n & \Rightarrow & n \\ (1{+}\ m') + n & \Rightarrow & 1{+}\ (m' + n) \end{array} \end{array}$$

$$\underline{\text{let}}\ \frac{xs : \mathsf{Vec}\ m\ X \quad ys : \mathsf{Vec}\ n\ X}{xs \mathbin{+\!\!+} ys : \mathsf{Vec}\ (m + n)\ X} \qquad \begin{array}{l} xs + ys \ \Leftarrow\ \underline{\text{rec}}\ xs \\ \quad \begin{array}{lcl} \mathsf{vnil} \mathbin{+\!\!+} ys & \Rightarrow & ys \\ \mathsf{vcons}\ x\ xs' \mathbin{+\!\!+} ys & \Rightarrow & \mathsf{vcons}\ x\ (xs' \mathbin{+\!\!+} ys) \end{array} \end{array}$$

Note the importance of the index unification in the above example—it's the instantiation of the first argument's length with 0 or $(1{+}\ m')$ which enables the length of the concatenation to compute down to the length of the vectors we actually return in each case.

The fact that the length is some kind of data allows us to write generic operations by computation over it. This example computes a constant vector of the required size.

$$\underline{\text{let}}\ \frac{x\ :\ X}{\mathbf{vec}_n\ x\ :\ \mathsf{Vec}\ n\ X} \qquad \begin{array}{l} \mathbf{vec}_n\ x \ \Leftarrow \underline{\text{rec}}\ n \\ \quad \mathbf{vec}_{0}\qquad x \Rightarrow \mathsf{vnil} \\ \quad \mathbf{vec}_{(1+\ n')}\ x \Rightarrow \mathsf{vcons}\ x\ (\mathbf{vec}_{n'}\ x) \end{array}$$

We write **vec**'s length argument as a subscript to indicate that it is usually to be left implicit when the **vec** function is used. For example, we can map a function f across a vector xs just by writing $\mathbf{vec} f\ ^{@}\ xs$, because the type of $^{@}$ will require the output of **vec** to have the same length as xs. The technology we have inherited from Damas and Milner is now being used to infer *value* parameters as well as types. The behaviours of **vec** and $^{@}$ combine conveniently to give us 'vectorized applicative programming' with size invariants quietly maintained: transposition shows this in action.

$$\underline{\text{let}}\ \frac{xij\ :\ \mathsf{Vec}\ i\ (\mathsf{Vec}\ j\ X)}{\mathbf{xpose}\ xij\ :\ \mathsf{Vec}\ j\ (\mathsf{Vec}\ i\ X)} \qquad \begin{array}{l} \mathbf{xpose}\ xij \ \Leftarrow \underline{\text{rec}}\ xij \\ \quad \mathbf{xpose}\qquad \mathsf{vnil} \qquad \Rightarrow \mathbf{vec}\ \mathsf{vnil} \\ \quad \mathbf{xpose}\ (\mathsf{vcons}\ xj\ xi'j) \Rightarrow \mathbf{vec}\ \mathsf{vcons}\ ^{@}\ xj\ ^{@}\ \mathbf{xpose}\ xi'j \end{array}$$

4.1 Static Indexing and Proofs

In our definition of $++$, we were lucky—the computation on the vectors was in harmony with the computation on the numbers. We are not always so lucky—if we try to reverse a vector by the usual accumulating algorithm, we kick against the computational behaviour of $+$ and the obvious program does not typecheck.

$$\underline{\text{let}}\ \frac{xs\ :\ \mathsf{Vec}\ m\ X \qquad ys\ :\ \mathsf{Vec}\ n\ X}{\mathbf{vrevacc}\ xs\ ys\ :\ \mathsf{Vec}\ (m+n)\ X} \qquad \begin{array}{l} \mathbf{vrevacc}\ xs\ ys \ \Leftarrow \underline{\text{rec}}\ xs \\ \quad \mathbf{vrevacc}\qquad \mathsf{vnil}\qquad ys \Rightarrow ys \\ \quad \mathbf{vrevacc}\ (\mathsf{vcons}\ x\ xs')\ ys \Rightarrow \boxed{\mathbf{vrevacc}\ xs'\ (\mathsf{vcons}\ x\ ys)} \end{array}$$

The trouble is that the shaded expression has length $m' + (1+\ n)$, and we require a length of $1+\ (m'+n)$, where xs' and ys have lengths m' and n respectively. The fact that these two lengths are the same does not follow directly from applying the computational behaviour of $+$, rather it's an algebraic property for which we can offer an inductive explanation.

$$\underline{\text{let}}\ \frac{}{\mathbf{plusSuc}\ m\ n\ :\ m + (1+\ n) = 1+\ (m+n)}$$

$$\begin{array}{l} \mathbf{plusSuc}\ m\ n \ \Leftarrow \underline{\text{rec}}\ m \\ \quad \mathbf{plusSuc}\quad 0 \quad n \Rightarrow \mathsf{refl} \\ \quad \mathbf{plusSuc}\ (1+\ m')\ n \Rightarrow [\mathbf{plusSuc}\ m'\ n\rangle \end{array}$$

We write $[q\rangle$ for the proof of an equation $p[s] = p[t]$ where $q\ :\ s = t$ and $\langle q]$ for the symmetric proof of $p[t] = p[s]$ Once we have this proof, we can fix our accumulating reverse:

$$\underline{\text{let}}\ \frac{xs\ :\ \mathsf{Vec}\ m\ X \qquad ys\ :\ \mathsf{Vec}\ n\ X}{\mathbf{vrevacc}_{m\ n}\ xs\ ys\ :\ \mathsf{Vec}\ (m+n)\ X}$$

$$\begin{array}{l} \mathbf{vrevacc}\ xs\ ys \ \Leftarrow \underline{\text{rec}}\ xs \\ \quad \mathbf{vrevacc}\qquad\qquad \mathsf{vnil}\qquad ys \Rightarrow ys \\ \quad \mathbf{vrevacc}_{(1+\ m')\ n}\ (\mathsf{vcons}\ x\ xs')\ ys \Rightarrow \{[\mathbf{plusSuc}\ m'\ n\rangle\}\ \mathbf{vrevacc}\ xs'\ (\mathsf{vcons}\ x\ ys) \end{array}$$

It is perhaps not surprising that to finish the job, we need another lemma (whose proof is an unremarkable induction):

$$\underline{\text{let}}\ \frac{}{\mathbf{plusZero}\ n\ :\ n + 0 = n} \qquad \cdots$$

$$\underline{\text{let}}\ \frac{xs\ :\ \mathsf{Vec}\ n\ X}{\mathbf{vrev}_n\ xs\ :\ \mathsf{Vec}\ n\ X} \qquad \mathbf{vrev}_n\ xs\ \Rightarrow\ \{[\mathbf{plusZero}\ n\rangle\}\ \mathbf{vrevacc}\ xs\ \mathsf{vnil}$$

As we have already mentioned, these proofs are erased at run-time—their only rôle is to make the typechecker aware of more than it can figure out by computation alone. Perhaps it's undesirable to 'repair' programs with proofs—our **vrev** is certainly more trouble to write than the reversal of an unsized list, but this is just the work which has to be done if you want to know that length is preserved. If we don't care, we don't have to work so hard—we could just return a vector of *some* length, rather than the *same* length:

$$\underline{\text{let}}\ \frac{xs\ :\ \mathsf{Vec}\ m\ X \qquad ys\ :\ \mathsf{Vec}\ n\ X}{\mathbf{vrevacc}'\ xs\ ys\ :\ \exists_l \Rightarrow \mathsf{Vec}\ l\ X}$$

$$\begin{array}{llll}
\mathbf{vrevacc}'\ xs\ ys \Leftarrow \underline{\text{rec}}\ xs \\
\quad \mathbf{vrevacc}' & \mathsf{vnil} & ys & \Rightarrow\ ys \\
\quad \mathbf{vrevacc}' & (\mathsf{vcons}\ x\ xs') & ys & \Rightarrow\ \mathbf{vrevacc}'\ xs'\ (\mathsf{vcons}\ x\ ys)
\end{array}$$

The notion of 'some length' is expressed via an *implicit existential* $\exists_l \Rightarrow \mathsf{Vec}\ l\ X$. The vector is packed (inferring the witness) and unpacked (discarding the witness) automatically. It allows us to recover a less dependent type by hiding an index—here we recover ordinary lists. We can write the old programs with the old precision just as easily as before.

As with totality, we have enough language to negotiate a pragmatic compromise, adopting more precise methods where the gain is worth the work. The more prudent course, perhaps, is to try to make the unremarkable proofs as cheap as possible. Xi and Pfenning's approach of equipping the typechecker with decision procedures for standard problem classes is a great help in practice: DML has no difficulty discharging the elementary properties of $+$ we required above. We should certainly emulate this functionality.

4.2 Sized Merge-Sort

We can provide a more substantial example of 'the pragmatics of precision' by rolling out size invariants across our development of merge-sort. We shall replace the lists by vectors, and we shall seek to ensure that sorting preserves the length of its input.

How will sizing affect **merge**? The output length should be the sum of the input lengths.

$$\underline{\text{let}}\ \frac{xs\ :\ \mathsf{Vec}\ m\ \mathsf{Nat} \qquad ys\ :\ \mathsf{Vec}\ n\ \mathsf{Nat}}{\mathbf{merge}_{m\ n}\ xs\ ys\ :\ \mathsf{Vec}\ (m+n)\ \mathsf{Nat}}$$

$$\begin{array}{llll}
\mathbf{merge} & xs & ys & \Leftarrow \underline{\text{rec}}\ xs \\
\ \mathbf{merge} & \mathsf{vnil} & ys & \Rightarrow\ ys \\
\ \mathbf{merge} & (\mathsf{vcons}\ x\ xs') & ys & \Leftarrow \underline{\text{rec}}\ ys \\
\quad \mathbf{merge}_{(1+\ m')\ 0} & (\mathsf{vcons}\ x\ xs') & \mathsf{vnil} & \Rightarrow\ \{\langle\mathbf{plusZero}\ (1+\ m')]\}\ xs \\
\quad \mathbf{merge}_{(1+\ m')\ (1+\ n')} & (\mathsf{vcons}\ x\ xs') & (\mathsf{vcons}\ y\ ys') & \\
\quad\quad \mid\ \mathbf{order}\ x\ y & & & \\
\quad\quad \mid\quad \mathsf{le} & \Rightarrow\ \mathsf{vcons}\ x\ (\mathbf{merge}\ xs'\ ys) & & \\
\quad\quad \mid\quad \mathsf{ge} & \Rightarrow\ \mathsf{vcons}\ y\ (\{\langle\mathbf{plusSuc}\ m'\ n]\}\ \mathbf{merge}\ xs\ ys') & &
\end{array}$$

We shall also need to add sizes to our intermediate DealT data structure. The sizes for empT and leafT are obvious enough, but what about nodeT? A useful clue is provided by the *balancing* invariant which our program preserves—the size of the left subtree is either equal to that of the right subtree or just one more, depending on the parity bit. Let's write that down (we use decimals to abbreviate numerical constants):

$$\underline{\text{let}}\ \frac{p : \mathsf{Parity}}{\hat{p} : \mathsf{Nat}} \quad \begin{array}{lcl} \hat{\mathsf{p0}} & \Rightarrow & 0 \\ \hat{\mathsf{p1}} & \Rightarrow & 1 \end{array}$$

$$\underline{\text{data}}\ \frac{n : \mathsf{Nat} \quad X : \star}{\mathsf{DealT}\ n\ X : \star} \ \underline{\text{where}} \ \frac{}{\mathsf{empT} : \mathsf{DealT}\ 0\ X} \quad \frac{x : X}{\mathsf{leafT}\ x : \mathsf{DealT}\ 1\ X}$$

$$\frac{p : \mathsf{Parity} \quad l : \mathsf{DealT}\ (\hat{p} + n)\ X \quad r : \mathsf{DealT}\ n\ X}{\mathsf{nodeT}_n\ p\ l\ r : \mathsf{DealT}\ ((\hat{p} + n) + n)\ X}$$

There is more than one way to write down the size of a nodeT.[5] The choice we make here is motivated by the **mergeT** operation, which now has a more informative type:

$$\underline{\text{let}}\ \frac{t : \mathsf{DealT}\ n\ \mathsf{Nat}}{\mathbf{mergeT}\ t : \mathsf{Vec}\ n\ \mathsf{Nat}}$$

$$\begin{array}{lll} \mathbf{mergeT}\ t & \Leftarrow & \underline{\text{rec}}\ t \\ \quad \mathbf{mergeT}\ \mathsf{empT} & \Rightarrow & \mathsf{vnil} \\ \quad \mathbf{mergeT}\ (\mathsf{leafT}\ x) & \Rightarrow & \mathsf{vcons}\ x\ \mathsf{nil} \\ \quad \mathbf{mergeT}\ (\mathsf{nodeT}\ p\ l\ r) & \Rightarrow & \mathbf{merge}\ (\mathbf{mergeT}\ l)\ (\mathbf{mergeT}\ r) \end{array}$$

We simply chose the return type for nodeT which made the old code for **mergeT** go through as it stood, given the new type of **merge**! Of course, we shall pay when we come to write **insertT**—we could shift the burden the other way by taking the size of a nodeT to be $\hat{p} + n * 2$.

$$\underline{\text{let}}\ \frac{x : X \quad t : \mathsf{DealT}\ n\ X}{\mathbf{insertT}\ x\ t : \mathsf{DealT}\ (1{+}\ n)\ X}$$

$$\begin{array}{lll} \mathbf{insertT}\ x\ t & \Leftarrow & \underline{\text{rec}}\ t \\ \quad \mathbf{insertT}\ x\ \mathsf{empT} & \Rightarrow & \mathsf{leafT}\ x \\ \quad \mathbf{insertT}\ x\ (\mathsf{leafT}\ y) & \Rightarrow & \mathsf{nodeT}\ \mathsf{p0}\ (\mathsf{leafT}\ y)\ (\mathsf{leafT}\ x) \\ \quad \mathbf{insertT}\ x\ (\mathsf{nodeT}\ \mathsf{p0}\ l\ r) & \Rightarrow & \mathsf{nodeT}\ \mathsf{p1}\ (\mathbf{insertT}\ x\ l)\ r \\ \quad \mathbf{insertT}\ x\ (\mathsf{nodeT}_n\ \mathsf{p1}\ l\ r) & \Rightarrow & \{[\mathbf{plusSuc}\ n\ n\rangle\}\ \mathsf{nodeT}_{(1+\ n)}\ \mathsf{p0}\ l\ (\mathbf{insertT}\ x\ r) \end{array}$$

The damage is not too bad—we just have to appeal to algebraic properties of $+$, to show that the constructed tree of size $(1{+}n) + n$ fits the constraint $1{+}(n + n)$. This leaves **dealT** and **sort** with new types, but basically the same code:

$$\underline{\text{let}}\ \frac{xs : \mathsf{Vec}\ n\ X}{\mathbf{dealT}\ xs : \mathsf{DealT}\ n\ X} \quad \begin{array}{lll} \mathbf{dealT}\ xs & \Leftarrow & \underline{\text{rec}}\ xs \\ \quad \mathbf{dealT}\ \mathsf{vnil} & \Rightarrow & \mathsf{empT} \\ \quad \mathbf{dealT}\ (\mathsf{vcons}\ x\ xs) & \Rightarrow & \mathbf{insertT}\ x\ (\mathbf{dealT}\ xs) \end{array}$$

$$\underline{\text{let}}\ \frac{xs : \mathsf{Vec}\ n\ \mathsf{Nat}}{\mathbf{sort}\ xs : \mathsf{Vec}\ n\ \mathsf{Nat}} \quad \mathbf{sort} \Rightarrow \mathbf{mergeT} \cdot \mathbf{dealT}$$

It seems appropriate at this point to emphasize the importance of feedback from the typechecker when doing a development like this. It's the typechecker which tells you which lemmas you need and where you need to insert them, and it's the constraints which arise during typechecking which tell you how to engineer a data structure's indices so that its operations typecheck where possible. The computational coincidences between the indices we encounter are really a matter of care, not luck.

[5] Indeed, the above does not ensure that the subtrees of a node are nonempty—this can be done by replacing n with $(1{+}\ n')$ in the type of nodeT.

4.3 A Question of Phase

In our sorting example, the natural numbers Nat are playing two separate parts—dynamically, they represent the data being sorted; statically, they give sizes to the data structures involved in the process. All the case analysis happens over the indexed datatypes, rather than the indices themselves, so there is no need for sizes at run-time. It would appear that, in this example at least, the data on which types depend is entirely static. Although we need something like the natural numbers within the language of types, should it be the natural numbers? Perhaps it would be simpler if, as well as the natural numbers, there was something else exactly like them.

It's not as ridiculous as it sounds: it enables us to keep the workaday term language out of types, allowing dependency only on the static things. This keeps the type/term distinction in alignment with the phase distinction, separating the static $\forall$ from the dynamic $\rightarrow$, each with their own distinct notions of abstraction, application and now of datatype.

Of course, static datatypes are not quite enough: our DealT datatype family relies on operations such as $+$. We need static functions over static data, together with the associated partial evaluation technology. We need to find design criteria for this emerging static programming language. Will all our static data structures be monomorphic? Do we believe that static data will never need to be indexed itself? It would be bold to imagine that 'yes' is the answer to these questions. At what point will we stop extending the static language with a replica of the dynamic language?

We're beginning to see more and more of the same phenomema showing up on either side of the phase distinction; we're even beginning to see the potential emergence of a phase *hierarchy*. Occam's razor suggests that we should understand 'data' and 'function' once, regardless of phase, since the phase distinction no longer distinguishes where these notions arise.

But Occam's razor is a subjective instrument, so we cannot presume that others will come to the same judgment as ourselves. We can, however, examine the impact of the underlying presumption that 'the data on which types depend is entirely static'.

5. Evidence is About Data; Evidence is Data

Type systems without dependency on dynamic data tend to satisfy the *replacement* property—any subexpression of a well typed expression can be replaced by an alternative subexpression of the same type in the same scope, and the whole will remain well typed. For example, in Java or Haskell, you can always swap the `then` and `else` branches of conditionals and nothing will go wrong—nothing of any static significance, anyway. The simplifying assumption is that within any given type, one value is as good as another. These type systems have no means to express the way that different data mean different things, and should be treated accordingly in different ways. That is why dependent types matter.

More specifically, we have seen how the apparatus of dependent types can be used to maintain the length invariant in our sorting algorithm, but that the length can essentially be regarded as a prior static notion which the code must *respect* dynamically. What if we wanted to guarantee the the output of our sorting algorithm is in order? The order is not a prior static notion—that is why we need a sorting algorithm—the order is *established* by run-time testing of the dynamic data. Can we observe this fact statically? As things stand, Order does not matter: we can swap around the le and ge outputs of our **order** test without affecting the well typedness of **merge**. How can we make Order matter? By making Order a dependent type!

5.1 Evidence of Ordering

We replace the uninformative type Order by an inductive family with Order x y expressing that x and y can be ordered and each possibility can be established with *evidence*.

$$\underline{\text{data}}\ \frac{x, y \ :\ \mathsf{Nat}}{x \leq y \ :\ \star}\ \underline{\text{where}}\ \cdots$$

$$\underline{\text{data}}\ \frac{x, y \ :\ \mathsf{Nat}}{\mathsf{Order}\ x\ y \ :\ \star}\ \underline{\text{where}}\ \frac{xley \ :\ x \leq y}{\mathsf{le}\ xley \ :\ \mathsf{Order}\ x\ y}\quad \frac{ylex \ :\ y \leq x}{\mathsf{ge}\ ylex \ :\ \mathsf{Order}\ x\ y}$$

We shall give $\leq$ its constructors shortly. First, let us see what happens to the **order** test—for a start, its type now tells us what its output says about its input! We provide empty sheds $[]$ which will be completed once we have decided how to represent $\leq$ proofs. The freedom to delay implementation decisions while type checking the rest of the code is an essential feature of Epigram which we further elaborate in the next section.

$$\underline{\text{let}}\ \frac{}{\mathbf{order}\ x\ y\ :\ \mathsf{Order}\ x\ y} \qquad \begin{array}{llll} \mathbf{order} & x & y & \Leftarrow \underline{\text{rec}}\ x \\ \ \mathbf{order} & 0 & y & \Rightarrow \mathsf{le}\ [] \\ \ \mathbf{order} & (1+x') & 0 & \Rightarrow \mathsf{ge}\ [] \\ \ \mathbf{order} & (1+x') & (1+y') & \mid\ \mathbf{order}\ x'\ y' \\ & & & \mid\ \quad \mathsf{le}\ x'ley' \quad \Rightarrow \mathsf{le}\ [] \\ & & & \mid\ \quad \mathsf{ge}\ y'lex' \quad \Rightarrow \mathsf{ge}\ [] \end{array}$$

This program is almost as before, except that we cannot pass the recursive call back directly—it is the ordering of x' and y', but we need to order their successors. We can treat this partial program as a clue to good constructors for $\leq$. What should we wish them to be? Well, our first $[]$ requires a proof that $0 \leq y$, so let us have

$$\frac{}{\mathsf{le0}\ :\ 0 \leq y}$$

Actually, that will also satisfy our second $[]$, which needs a proof of $0 \leq (1+x')$. Our third $[]$ requires us to establish $(1+x') \leq (1+y')$, given $x'ley'\ :\ x' \leq y'$, and the fourth $[]$ is similar, so let us have

$$\frac{xley\ :\ x \leq y}{\mathsf{leS}\ xley\ :\ (1+x') \leq (1+y')}$$

The two seem like a reasonable definition of $\leq$, and they certainly enable us to fill in our $[]$s.

$$\underline{\text{let}}\ \frac{}{\mathbf{order}\ x\ y\ :\ \mathsf{Order}\ x\ y} \qquad \begin{array}{llll} \mathbf{order} & x & y & \Leftarrow \underline{\text{rec}}\ x \\ \ \mathbf{order} & 0 & y & \Rightarrow \mathsf{le}\ \mathsf{le0} \\ \ \mathbf{order} & (1+x') & 0 & \Rightarrow \mathsf{ge}\ \mathsf{le0} \\ \ \mathbf{order} & (1+x') & (1+y') & \mid\ \mathbf{order}\ x'\ y' \\ & & & \mid\ \quad \mathsf{le}\ x'ley' \quad \Rightarrow \mathsf{le}\ (\mathsf{leS}\ x'ley') \\ & & & \mid\ \quad \mathsf{ge}\ y'lex' \quad \Rightarrow \mathsf{ge}\ (\mathsf{leS}\ y'lex') \end{array}$$

What has happened here? We started with a program which *did* the right thing but did not *say* so. It should be no surprise that when we try to make this program say what it does, we learn how to say the right thing.

However, some of you may be wondering whether it is worth saying the right thing, if it means spending heap on this data structure of evidence and losing the tail-call optimisation into the bargain. Fortunately, our $\leq$ type has the property of being *content-free* [BMM04]: just as with $=$, for any given indices x and y, $x \leq y$ contains at most one value, so the evidence can be erased at run-time and the tail-call restored. Moreover, it is no accident that $\leq$ is content-free: our method of packing up the cases arising ensured that le0 and leS covered distinct indices, and that the indices of any recursive proofs were determined in turn.

Just to be sure, let us check that $\leq$ is a total ordering—given **order**, we just need

$$\underline{\text{let}}\ \frac{}{\mathbf{leRefl}_x : x \le x} \qquad \begin{array}{lcl} \mathbf{leRefl}_x & \Leftarrow & \underline{\text{rec}}\ x \\ \quad \mathbf{leRefl}_0 & \Rightarrow & \mathsf{le0} \\ \quad \mathbf{leRefl}_{(1+\,x')} & \Rightarrow & \mathsf{leS}\ \mathbf{leRefl}_{x'} \end{array}$$

$$\underline{\text{let}}\ \frac{xley : x \le y \quad ylez : y \le z}{\mathbf{leTrans}\ xley\ ylez : x \le z} \qquad \begin{array}{llll} \mathbf{leTrans} & xley & ylez & \Leftarrow \underline{\text{rec}}\ xley \\ \quad \mathbf{leTrans} & \mathsf{le0} & ylez & \Rightarrow \mathsf{le0} \\ \quad \mathbf{leTrans} & (\mathsf{leS}\ xley') & (\mathsf{leS}\ ylez') & \Rightarrow \mathsf{leS}\ (\mathbf{leTrans}\ xley'\ ylez') \end{array}$$

$$\underline{\text{let}}\ \frac{xley : x \le y \quad ylex : y \le x}{\mathbf{leASym}\ xley\ ylex : x = y} \qquad \begin{array}{llll} \mathbf{leASym} & xley & ylex & \Leftarrow \underline{\text{rec}}\ xley \\ \quad \mathbf{leASym} & \mathsf{le0} & \mathsf{le0} & \Rightarrow \mathsf{refl} \\ \quad \mathbf{leASym} & (\mathsf{leS}\ xley') & (\mathsf{leS}\ ylex') & \Rightarrow [\mathbf{leASym}\ xley'\ ylex'\rangle \end{array}$$

As it happens, these properties of $\le$ are not necessary in order to implement sorting. To be sure that a list is sorted, you need at least to have checked that each adjacent pair is in order—that it's *locally sorted*. It's not hard to see that a list can always be locally sorted with respect to any binary relation which always holds one way around or the other (when inserting a new element, if it fits nowhere before the end, then it must fit at the end). Of course, knowing that it is also a partial order enhances what you can deduce from a locally sorted list.

5.2 Locally Sorted Lists

How shall we define locally sorted lists? We shall clearly have to index lists with some sort of interval, but there are several ways we might do it: one bound or two? open bounds or closed bounds? As with the design of the sized DealT, we should take care to ensure that the decision leads to operations which are as cleanly defined as possible. For pedagogical purposes, we shall sort lists rather than vectors—sizing and sorting are independent refinements of the list structure. We do not verify the fact that the resulting list is a permutation of the input here.

One bound or two? In order to do a 'cons', we shall certainly need to know that the head and the tail are suitably ordered, so the tail will require a lower bound. Meanwhile, **merge** makes no restrictions between the bounds of its inputs—only between the input bounds and the output bounds. That suggests that we can get away with a lower bound for this example. Of course, if we wanted to *concatenate* sorted lists (in an algorithm based on pivoting, say), we should need upper bounds too.

Open or closed? It is perhaps a little tricky to give a precise lower bound for the empty list—we could make a bound 'elements lifted with ∞' and lift $\le$ accordingly:

$$\underline{\text{data}}\ \frac{b : \mathsf{Lifted\ Nat}}{\mathsf{CList}\ b : \star}\ \underline{\text{where}}\ \frac{}{\mathsf{cnil} : \mathsf{CList}\ \infty} \quad \frac{x : \mathsf{Nat} \quad xley : x \le y \quad xs : \mathsf{CList}\ y}{\mathsf{ccons}\ x\ xley\ xs : \mathsf{CList}\ (\mathsf{lift}\ x)}$$

The lifting is not a big issue here—we only get away without $-\infty$ because Nat stops at 0. The advantage of this definition is its precision: each list is assigned its tightest bound. The disadvantage of this definition is also its precision—when we are making a CList, we must either specify its lower bound (eg., **sort** is bounded by **min**) and satisfy that specification, or say 'don't care' with an existential. But we do care! The recursive calls in **merge** can't have any old lower bound—the lower bound of the tail must be at least the head!

The fact that we can *assign* closed bounds to sorted lists does not necessarily make them a good choice for a type, because we also need to *prescribe* bounds. We can usually think of an open bound, even if it is not the best one. Let us try:

$$\underline{\text{data}}\ \frac{b : \mathsf{Nat}}{\mathsf{OList}\ b : \star}\ \underline{\text{where}}\ \frac{}{\mathsf{onil} : \mathsf{OList}\ b} \quad \frac{x : \mathsf{Nat} \quad blex : b \le x \quad xs : \mathsf{OList}\ x}{\mathsf{ocons}\ x\ blex\ xs : \mathsf{OList}\ b}$$

Here, we have that onil trivially satisfies any prescribed bound, whilst for ocons, the head must exceed the prescribed lower bound and bound the tail in turn. Any old sorted list can certainly be represented as an element of OList 0. Meanwhile, if *both* inputs to **merge** share a lower bound, then the output certainly shares it too.

$$\underline{\text{let}}\ \frac{xs, ys\ :\ \mathsf{OList}\ b}{\mathbf{merge}\ xs\ ys\ :\ \mathsf{OList}\ b}$$

$$\begin{array}{l}
\mathbf{merge}\ \ xs\ \ ys \Leftarrow \underline{\text{rec}}\ xs \\
\quad \mathbf{merge}\ \mathsf{onil}\ \ ys \Rightarrow ys \\
\quad \mathbf{merge}\ (\mathsf{ocons}\ x\ blex\ xs')\ \ ys \Leftarrow \underline{\text{rec}}\ ys \\
\quad\quad \mathbf{merge}\ (\mathsf{ocons}\ x\ blex\ xs')\ \mathsf{onil} \Rightarrow xs \\
\quad\quad \mathbf{merge}\ (\mathsf{ocons}\ x\ blex\ xs')\ (\mathsf{ocons}\ y\ bley\ ys') \\
\quad\quad\quad \mid \mathbf{order}\ x\ y \\
\quad\quad\quad\quad \mathsf{le}\ xley \Rightarrow \mathsf{ocons}\ x\ blex\ (\mathbf{merge}\ xs'\ (\mathsf{ocons}\ y\ xley\ ys')) \\
\quad\quad\quad\quad \mathsf{ge}\ ylex \Rightarrow \mathsf{ocons}\ y\ bley\ (\mathbf{merge}\ (\mathsf{ocons}\ x\ ylex\ xs')\ ys')
\end{array}$$

Each input list already satisfies the bound required for the output, so the onil cases are trivial. When we have two oconses, we know both heads satisfy the lower bound, but whichever we pick must bound the recursive **merge**. Hence, we had better pick the smaller one—the evidence we get from **order** is exactly what we need to show that the list which keeps its head satisfies the newer tighter bound.

Now we can flatten a DealT into a sorted list. This gives us a new back end which we can compose with the old **dealT** to get a sort which produces sorted output:

$$\underline{\text{let}}\ \frac{t\ :\ \mathsf{DealT\ Nat}}{\mathbf{mergeT}\ t\ :\ \mathsf{OList}\ 0} \qquad
\begin{array}{l}
\mathbf{mergeT}\ \ t \Leftarrow \underline{\text{rec}}\ t \\
\quad \mathbf{mergeT}\ \mathsf{empT} \Rightarrow \mathsf{onil} \\
\quad \mathbf{mergeT}\ (\mathsf{leafT}\ x) \Rightarrow \mathsf{ocons}\ x\ \mathsf{le0}\ \mathsf{onil} \\
\quad \mathbf{mergeT}\ (\mathsf{nodeT}\ p\ l\ r) \Rightarrow \mathbf{merge}\ (\mathbf{mergeT}\ l)\ (\mathbf{mergeT}\ r)
\end{array}$$

$$\underline{\text{let}}\ \frac{xs\ :\ \mathsf{List\ Nat}}{\mathbf{sort}\ xs\ :\ \mathsf{OList}\ 0} \qquad \mathbf{sort} \Rightarrow \mathbf{mergeT} \cdot \mathbf{dealT}$$

Remark. In the above definitions of **merge** and **mergeT**, all of the $\leq$ proofs which we supply in the lists we build are either by le0 or by direct appeal to a hypothesis in scope. The proofs which we uncover by case analysis are only used in this way. It seems reasonable to consider suppressing all of them by default from the explicit syntax of the program. Implicit hypotheses could be kept in the context and searched whenever an implicit proof is required, in much the way that Haskell handles the implicit dictionaries when unpacking and packing existential types. Of course, a 'manual override' is still necessary—not all proofs are so immediate.

5.3 Programming with Evidence

The idea that one datatype can represent evidence about the values in another is alien to mainstream functional programming languages, but its absence is beginning to cause pain. A recent experiment in 'dynamically typed' generic programming—the 'Scrap Your Boilerplate' library of traversal operators by Ralf Lämmel and Simon Peyton Jones [LP03]—is a case in point. The library relies on a 'type safe cast' operator, effectively comparing types at run time by comparing their encodings as data:

```
cast :: (Typeable a, Typeable b) => a -> Maybe b
cast x = r
       where
         r = if typeOf x == typeOf (get r)
             then Just (unsafeCoerce x)
             else Nothing
```

```
get :: Maybe a -> a
get x = undefined
```

Here, the Boolean test `typeOf x == typeOf (get r)` serves the purpose of comparing type `a` with type `b`, but the Haskell typechecker cannot interpret the value `True` as a reason to unify types. In the absence of evidence, the programmers resort to coercion. *We* trust them, of course, but this degree of trust should not be necessary. This is an example where static dependency on dynamic data is the solution, not the problem.

We now work in a setting where we expect operations which test their input in some way to have a dependent type which explains what the output reveals about the input—**order** is a simple example. Without some kind of movement between terms and types this is impossible—you can exploit valid data (eg., writing a type-preserving evaluator for a typed expression language) , but you cannot *validate* it dynamically (eg., by writing a typechecker).

Traditional dependent types achieve this movement directly—the argument of a function can appear in the type of its result, and often will, if the result is evidence of the argument's properties. An alternative, adopted for DML's numerical indices, is to push the other way with *singleton types*—types of dynamic data constrained to be equal to the corresponding static data. We would have something like this:

$$\textbf{order} \; : \; \forall x, y : \mathsf{Nat} \Rightarrow \mathsf{Only}\ x \;\rightarrow\; \mathsf{Only}\ y \;\rightarrow\; \mathsf{Order}\ x\ y$$

Here every piece of data about which we seek evidence must be abstracted twice—the type of evidence depends on the static copy, but the testing itself is performed on the dynamic copy. In the context of DML, this is a sensible separation—erasing the indices is intended to yield a well typed SML program. In a broader context, where we might need to represent properties of any kind of dynamic data, this approach seems unlikely to scale. Effectively, we may need to replace each type T by the existential pairing of its static and dynamic copies $\exists t : T \Rightarrow \mathsf{Only}\ t$ throughout our programs, in order to have data at all the levels where it is used.

Singleton types thus provide a convenient way to retro-fit a more sophisticated type system to an existing compiler architecture. With just one notion of data at all levels, dependent types provide a convenient way for programmers to exploit the potential of working with data as evidence. Our typechecker example in [MM04] not only generates enough evidence about the types being inferred and compared to feed the tagless interpreter, it generates evidence about the program being checked—it is the first typechecker in the literature which is statically guaranteed to check its input!

6. The Tools of the Trade

Programming is a complex task which can be made easier for people to do with the help of computers. The conventional cycle of programming with a text editor then compiling in 'batch mode' is a welcome shortening of the feedback loop since the days of punched cards, but it clearly under-uses the technology available today. Any typed programming language can benefit from the capacity—but not necessarily the compulsion—to invoke the typechecker incrementally on incomplete subprograms whilst they are under development. The more powerful the type system, the more pressing this need becomes—it just gets harder to do it in your head, especially when types contain *computations*, for which computers are inherently useful.

Moreover, a type acts as a partial specification of a program, and can thus be used to narrow the search space for correct programs, even if only as a key for searching a library. The choice of a program's type is inherently suggestive of the programming strategies with which that type is naturally equipped—constructors and case analysis for datatypes, abstraction and application for functions, and so on. It is a tragic waste if types play only a passive rôle in programming, providing a basis for error reporting. Our technology should enable programmers to exploit the clues which types provide.

6.1 Dependent Types Also Matter Behind the Scenes

The key innovation of Epigram is its use of types to express programming tasks and programming patterns. For example, the pattern of primitive recursion on a datatype is expressed by an *induction principle*, like

$$\begin{array}{l}\forall P : \mathsf{Nat} \to \star \;\Rightarrow \\ P\;0 \;\to\; (\forall n' : \mathsf{Nat} \Rightarrow P\;n' \to P\;(1{+}\;n')) \;\to \\ \forall n : \mathsf{Nat} \Rightarrow P\;n\end{array}$$

The conventional type of primitive recursion may be shorter, but it is less informative.

$$\forall P \Rightarrow P \;\to\; (\mathsf{Nat} \to P \to P) \;\to\; \mathsf{Nat} \;\to\; P$$

This type does not connect the arguments to the tasks they serve, whereas the induction principle is a dependent type which explains that a recursive computation for n needs a method for 0 and a method for $(1+\;n')$. Correspondingly, the system transforms a programming problem such as

$\underline{\mathrm{let}}\;\dfrac{x, y \;:\; \mathsf{Nat}}{x + y \;:\; \mathsf{Nat}}$ into a 'computability proof' goal $? : \forall x, y \;:\; \mathsf{Nat} \Rightarrow \langle x + y : \mathsf{Nat}\rangle$

We read $\langle x + y : \mathsf{Nat}\rangle$ as '$x + y$ is computable'. If we attack this goal with induction, we acquire subgoals like these:

$$\begin{array}{l}? : \forall y{:}\mathsf{Nat} \Rightarrow \langle 0 + y : \mathsf{Nat}\rangle \\ ? : \forall x'{:}\mathsf{Nat} \Rightarrow \\ \quad (\forall y{:}\mathsf{Nat} \Rightarrow \langle x' + y : \mathsf{Nat}\rangle) \;\to \\ \quad \forall y{:}\mathsf{Nat} \Rightarrow \langle (1{+}\;x') + y : \mathsf{Nat}\rangle\end{array}$$

These goals tell us exactly the 'left-hand sides' for the subprograms which the recursive strategy requires.

By using dependent types to represent programming problems and programming patterns, we have acquired an interactive programming environment for the price of an interactive proof system. The basic constructs of the Epigram language give a 'programming' presentation to the basic tactics of the system. The $\Leftarrow$ construct is just McBride's 'elimination with a motive' tactic [McB02a] which synthesizes an appropriate 'P' parameter for any induction-like rule. The $|\mathit{scrutinee}$ construct is just 'cut'. The details of the process by which Epigram code is *elaborated* into the underlying type theory—a variation of Luo's UTT [Luo94]—are given in [MM04]. Edwin Brady's compiler for UTT, which erases extraneous information at run time, is presented in [Bra05].

The point is this: UTT is not our programming language—UTT is our language for describing how programming works. Epigram has no hard-wired construct for constructor case analysis or constructor-guarded recursion—these are just programming patterns specified by type and supplied as standard with every datatype you define. However, UTT types are Epigram types, so you are free to extend our language by specifying and implementing your own patterns. Our typechecking example in [MM04] is a derived case analysis principle for expressions, exposing not their syntax, but their types or type errors. By making programming patterns first-class citizens via dependent types, we raise the level of abstraction available to programmers and provide interactive support for its deployment at a single stroke.

6.2 Programming Interactively

The interface to Epigram is inspired by the Alf proof editor [MN94], which introduced a type-directed structure editor for incomplete programs. In Epigram's concrete syntax, a *shed* [`raw text`] may stand in for any subexpression. The elaborator is not permitted inside a *shed*, so the text it contains may be edited freely. There are two basic editing moves—removing the brackets to admit the elaborator (which will process as far as any nested sheds) and placing brackets around an elaborated subexpression to 'undo' it and return it to a shed. Correspondingly, the full spectrum of interactivity is supported: an entire program can be written (or, more to the point, reloaded) inside a shed, then elaborated in 'batch mode'; or a single syntactic construct can be elaborated, with sheds for subexpressions. The advantages of structure editing are available, but the disadvantages are not compulsory.

The interactive development of a program is a kind of dialogue. The system poses the problems—the left-hand sides of programs. We supply the solutions by filling in the right-hand sides, either by directly giving the program's output $\Rightarrow\ t$, or by invoking a programming pattern which reduces the problem to subproblems which are then posed in turn. There is a direct mapping from sheds $[\ \cdots\ ]$ in source code to metavariables, $?x$ in the underlying proof state—when you elaborate a shed, your code triggers a refinement of the proof state, and the resulting refinement to the source code is read off from the types of the subgoals. Nothing is hidden—the proof state is recoverable from the source code simply by re-elaborating it in a single step.

Our approach to metavariables basically follows McBride's OLEG system [McB99]—metavariables represent not only the missing contents of sheds, but also all the unknowns arising from implicit quantification. The latter are resolved, where possible, by solving the unification constraints which arise during typechecking—we follow Dale Miller's 'mixed prefix' approach [Mil92]. Epigram will not guess the type of your program, but it will infer the bits of your program which the type determines. The Damas-Milner approach to type inference [DM82] is alive and well and working harder than ever, even though we have dispensed with the shackles on programming which allow it to be complete.

7. Further Work

We have hardly started. Exploiting the expressivity of dependent types in a practicable way involves a wide range of challenges in the development of the theory, the design of language, the engineering of tools and the pragmatics of programming. In this section, we summarize just a few of them.

First-Class Modules. No programming language can succeed without strong support for the large-scale engineering of systems. Dependent type systems already allow us to express record types which pack up data structures, operations over them—and also proofs of the properties of those operations. *Manifest* record types, which specify the values of some of their fields, can be used to express sharing between records, and between the the inputs and outputs of record-transforming operations [Pol00]. Epigram's first-class notion of programming pattern allows an abstract datatype to offer admissible notions of pattern matching which hide the actual data representation but are guaranteed to be faithful to it—we have Wadler's *views* for free [Wad87].

We now need a practical theory of subtyping to deliver a suitable form of inheritance, and a convenient high-level syntax for working with records. Our elaboration mechanism naturally lends itself to the approach of *coercive subtyping*, where subsumptions in source code elaborate to explicit coercion functions—typically projections—in the underlying theory [Luo97].

Universe Polymorphism. What is the type of types, and how do we quantify over them safely and consistently? In [MM04], we follow the *predicative* fragment of Luo's Extended Calculus of Constructions [Luo90], installing a cumulative hierarchy of universes $\star_0, \star_1, \ldots$ each of which both inhabits and embeds in the next, so that $\star_i : \star_{i+1}$ holds and $T : \star_i$ implies $T : \star_{i+1}$. As Harper and Pollack have shown [HP91], the user need never write a universe level explicitly—the machine can maintain a graph of relative level constraints and protest if any construction induces a cycle.

This much is certainly safe, and it allows every type to find its own particular level—this is called *typical ambiguity*. The trouble is that, as things stand, there is no satisfactory way to define datatype constructors which operate at *every* level—this is called *universe polymorphism*. A simple example shows up if we have a list Ts of element types, and we want to construct the corresponding list of list types

$$\mathbf{map}\ \mathsf{List}\ Ts$$

The types *in* Ts live one level below the type *of* Ts, so we need List to operate at *both* levels. There has been some very promising theoretical work in this area [CL01, Cou02] but again, a clear and convenient design has yet to emerge. In the interim, we have adopted the cheap but inconsistent fudge of taking $\star : \star$.

Generics and Reflection. Any function $T : U \rightarrow \star$ naturally *reflects* a sublanguage or *universe* of types—those given by $T\ u$. We can think of u as the 'name' of a type in the universe. If U happens to be a datatype, we can write generic programs which work for all the types named by a value u. Perhaps U is the type of regular expressions and T computes for each regular expression the type of its words, yielding *regular expression types* [HVP00]. Perhaps U represents a collection of datatypes with a decidable equality

$$\mathbf{eq} : \forall u : U;\ x, y : T\ u \Rightarrow x = y \vee x \neq y$$

Peter Morris has recently implemented exactly such a generic equality in Epigram for the universe of regular types [Mor05]. Dependently typed generic programming is a lively research area [AM03, BDJ03], inspired by the pioneering work on generics in Haskell [BJJM98, HJL04, CHJ+01].

Alternatively, perhaps our universe U represents a class of decidable propositions, equipped with a decision procedure

$$\mathbf{decide} : \forall u : U \Rightarrow T\ u \vee T\ u \rightarrow \bot$$

Such a procedure could be used to extend the elaborator's capacity to solve simple proof obligations and equational constraints automatically, following the lead of DML [XP99], but without wiring a particular constraint domain into the language design and the compiler. However, to make this work conveniently, we need language support for the declaration of universes (U, T) where U reflects a given class of types and T is invertible by construction, enabling the elaborator to infer the appropriate $u : U$ when invoking a generic operation.

Observational Type Theory. The type of equality proofs used in an intensional theory like Epigram's underlying Type Theory, is inconvenient when working with infinite objects like function types or lazy lists (codata). We plan to overcome this restriction without affecting the decidability of type checking by implementing an *Observational Type Theory* based on [Alt99, Hof95].

Monadic interfaces to the real world. A dependently typed language offers the opportunity to develop the idea of monadic IO further. Such a monadic interface comes in two guises: a static, denotational semantics which can be used to reason about the programs and an operational semantics, which is employed at runtime. This approach is not only relevant for IO, based on [Cap05] we can develop a *partiality monad* which allows us not only to implement but also reason about genuinely partial programs like interpreters or programs on the computable reals.

Refactoring. Epigram's *refinement* style of editing supports the process of working from types to programs, but it does not help with the inevitable iterations of the design cycle, as our plans evolve. Indeed, it is quite normal to build up the index structure for our in layers, as we did for our **sort** example. Our interactive editing technology should support these process also, allowing us to experiment at pushing different type refinements through our programs. We should be able to try out the options which gave rise to the 'open lower bound' choice for sorted lists.

More generally, we can seek to emulate existing tools for *refactoring* evolving the design of data structures and programs [LRT03], now in the context of a system which supports incomplete objects. There is plenty of scope to develop tools which really reflect the way most programmers work, iteratively improving program *attempts*—good ideas often come a bit at a time.

8. Conclusions

This much is clear: many programmers are already finding practical uses for the approximants to dependent types which mainstream functional languages (especially Haskell) admit, by hook or by crook. From arrows and functional reactive programming [Hug05, Nil05], through extensible records [KLS04], database programming [BH04] and dynamic web scripting [Thi02] to code generation [BS04], people are 'faking it' any way they can to great effect. Each little step along the road to dependent types makes the task a little easier, the code a little neater and the next improvement a little closer: the arrival of GADTs in Haskell is a joy and a relief [PWW04].

Now is the time to recognize the direction of these developments and pursue it by design, not drift. In the long term, there is a great deal more chaos and confusion to be feared from fumbling through today's jungle of type class Prolog, singletons and proxies than from a dependent type system whose core rules have been well studied and fit on the back of an envelope. The Epigram project is our attempt to plunder the proof systems based on Type Theory for the technology they can offer programmers. It is also a platform for radical experimentation without industrial inertia—an attempt to discover in which ways dependent types might affect the assumptions upon which mainstream functional language designs have been predicated. We are having a lot of fun, and there is plenty of fun left for many more researchers, but we do not expect to change the mainstream overnight. What we can hope to do is contribute a resource of experiments—successful or otherwise—to this design process.

More technically, what we have tried to demonstrate here is that the distinctions term/type, dynamic/static, explicit/inferred are no longer naturally aligned to each other in a type system which recognizes the relationships between values. We have decoupled these dichotomies and found a language which enables us to explore the continuum of pragmatism and precision and find new sweet spots within it. Of course this continuum also contains opportunities for remarkable ugliness and convolution—one can never legislate against bad design—but that is no reason to toss away its opportunities. Often, by bringing out the ideas which lie behind good designs, by expressing the things which matter, dependent types make data and programs fit better.

References

[Alt99] Thorsten Altenkirch. Extensional equality in intensional type theory. In *14th Symposium on Logic in Computer Science*, 1999.

[AM03] 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.

[Aug85] Lennart Augustsson. Compiling Pattern Matching. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 368–381. Springer-Verlag, 1985.

[Aug98] Lennart Augustsson. Cayenne—a language with dependent types. In *ACM International Conference on Functional Programming '98*. ACM, 1998.

[BDJ03] 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.

[BH04] Björn Bringert and Anders Höckersten. HaskellDB Improved. In Nilsson [Nil04].

[BJJM98] 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.

[BMM04] 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.

[Bra05] Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.

[BS02] Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In S. Peyton Jones, editor, *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 157–166. ACM Press, 2002.

[BS04] Arthur I. Baars and S. Doaitse Swierstra. Type safe, self inspecting code. In Nilsson [Nil04].

[Bur69] Rod Burstall. Proving properties of programs by structural induction. *Computer Journal*, 12(1):41–48, 1969.

[CAB$^+$86] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.

[Cap05] Venanzio Capretta. General recursion via coinductive types. Manuscript, available from Capretta's homepage, 2005.

[CC99] Catarina Coquand and Thierry Coquand. Structured type theory. In *Proc. Workshop on Logical Frameworks* *and Meta-languages (LFM'99)*, 1999.

[CH03] James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[CHJ+01] 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.

[CL01] Paul Callaghan and Zhaohui Luo. An implementation of LF with coercive subtyping & universes. *Journal of Automated Reasoning*, 27:3–27, 2001.

[Coq92] 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.

[Cou02] Judicaël Courant. Explicit universes for the calculus of constructions. In Victor A. Carreño, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics: 15th International Conference, TPHOLs 2002*, volume 2410 of *Lecture Notes in Computer Science*, pages 115–130, Hampton, VA, USA, August 2002. Springer-Verlag.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programming languages. In *Ninth Annual Symposium on Principles of Programming Languages (POPL) (Albuquerque, NM)*, pages 207–212. ACM, January 1982.

[Dyb91] Peter Dybjer. Inductive Sets and Families in Martin-Löf's Type Theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*. CUP, 1991.

[Gon04] Georges Gonthier. The four color theorem in Coq. Talk given at the TYPES 2004 conference, December 2004.

[Hin01] Ralf Hinze. Manufacturing datatypes. *Journal of Functional Programming, Special Issue on Algorithmic Aspects of Functional Programming Languages*, 11(5):493–524, September 2001.

[HJL04] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programmming*, 51:117–151, 2004.

[Hof95] Martin Hofmann. *Extensional concepts in intensional type theory*. PhD thesis, University of Edinburgh, 1995.

[HP91] Robert Harper and Randy Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991.

[HST+03] Nadeem A. Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof carrying-code. *Journal of Automated Reasoning (Special issue on Proof-Carrying Code)*, 31(3-4):191–229, December 2003.

[Hug05] John Hughes. Programming with arrows. In Vene and Uustalu [VU05]. Revised lecture notes from the International Summer School in Tartu, Estonia.

[HVP00] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular Expression Types for XML. In *Proceedings of the International Conference on Functional Programming*, 2000.

[KLS04] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly Typed Heterogeneous Collections. In Nilsson [Nil04].

[LP92] 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.

[LP03] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[LRT03] Huiqing Li, Claus Reinke, and Simon Thompson. Tool support for refactoring functional programs. In Johan Jeuring, editor, *ACM SIGPLAN 2003 Haskell Workshop*. Association for Computing Machinery, August 2003. ISBN 1-58113-758-3.

[Luo90] Zhaohui Luo. *ECC: An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/90/ECS-LFCS-90-118/`.

[Luo94] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.

[Luo97] Zhaohui Luo. Coercive subtyping in type theory. In *Computer Science Logic (CSL '96)*, volume 1258 of *LNCS*, pages 275–296. Springer-Verlag, 1997.

[McB99] 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/`.

[McB02a] 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.

[McB02b] Conor McBride. Faking It (Simulating Dependent Types in Haskell). *Journal of Functional Programming*, 12(4& 5):375–392, 2002. Special Issue on Haskell.

[McB04] Conor McBride. The Epigram prototype: a nod and two winks. Available from the Epigram homepage, 2004.

[McB05] Conor McBride. Epigram: Practical programming with dependent types. In Vene and Uustalu [VU05]. Revised lecture notes from the International Summer School in Tartu, Estonia.

[Mil92] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, 1992.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis·Napoli, 1984.

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

[MN94] 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.

[Mor05] Peter Morris. Generic programming in a dependently typed language. Talk given at the British Colloqium for Theoretical Computer Science, 2005.

[Nil04] Henrik Nilsson, editor. *Proceedings of the ACM SIGPLAN Haskell Workshop 2004, Snowbird, Utah*. ACM, 2004.

[Nil05] Henrik Nilsson. Dynamic Optimization in Functional Reactive Programming using Generalized Algebraic Data Types. submitted to ICFP 2005, 2005.

[NL96] George C. Necula and Peter Lee. Proof-carrying code. Technical Report CMU-CS-96-165, Computer Science Department, Carnegie Mellon University, September 1996.

[NPS90] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's type theory: an introduction*. Oxford University Press, 1990.

[Pol00] Robert Pollack. Dependently Typed Records for Representing Mathematical Structure. In Mark Aagard and John Harrison, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2000*, volume 1869 of *LNCS*. Springer-Verlag, 2000.

[PWW04] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. Manuscript, July 2004.

[She04] Tim Sheard. Languages of the future. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 116–119, October 2004.

[Tea04] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, April 2004.

[Thi02] Peter Thiemann. Programmable type systems for domain specific languages. *Electronic Notes in Theoretical Computer Science*, 76, 2002.

[Tur95] David Turner. Elementary Strong Functional Programming. In *Functional Programming Languages in Education, First International Symposium*, volume 1022 of *LNCS*. Springer-Verlag, 1995.

[VU05] 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.

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of POPL '87*. ACM, 1987.

[Xi04] Hongwei Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.