The view from the left

Conor McBride and James McKinna

Department of Computer Science
University of Durham
c.t.mcbride@durham.ac.uk
j.h.mckinna@durham.ac.uk

Abstract

Interactive, refinement-style proof construction in type theory has several things to offer the programmer seeking to exploit languages with dependent types.

Firstly, as is by now quite well-known, definition by pattern matching becomes a more discriminating tool for problem-solving, since it refines the explanation of types as well as values. This corresponds to the instantiation of propositions in proof by induction.

Secondly, Gentzen's sequent calculus draws attention to the rôle played by cut formulæ. Subsidiary case analyses on the results of intermediate computations, which commonly take place on the right-hand side of definitions by pattern matching, should rather be explained by 'left rules'. This subsumes the trivial case of Boolean guards in simply-typed languages.

Thirdly, pattern matching decompositions have a well-defined interface given by a dependent type. These are user-definable, and generalize Wadler's notion of 'view' (Wadler, 1987). The programmer wishing to introduce a new view of a datatype, and exploit it directly in pattern matching, may do so via a standard idiom: writing a program.

This paper introduces enough syntax and semantics to account for this high-level style of programming in dependent type theory. It culminates in the development of a typechecker for the simply-typed lambda calculus, which furnishes a view of raw terms as either being well-typed, or containing an error. The implementation of this view is a proof that typechecking is decidable.

1 Introduction

Contemporary proof development systems based on intensional type theory (with a consequent notion of explicit proof object on the one hand, and computation as a primitive notion in the theory on the other) have a highly evolved account of goal-directed problem-solving, based on the "problems-as types" principle.

Programming is a particular kind of problem-solving, and consequently we might expect to gain insight from such proof systems into the programming process, especially in the presence of languages with dependent types. Natural deduction presentations of programming, with the accompanying slogans "propositions-as-types" and "proofs-as-programs" are, of course, by now very familiar. Our motivation comes from more recent work (Pym, 1990; Herbelin, 1995; Dyckhoff & Pinto, 1998) exploring refinements of Gentzen's sequent calculus (Gentzen, 1935). Our interest lies in the light this sheds upon the *interactive* process of proof/program construction. In particular, we focus on the following features:

- *left rules*, which give rise to the left-hand sides of pattern matches;
- *cut formulæ*, which capture the types of *intermediate computations*;
- *right rules*, which more-or-less directly capture the right-hand sides of pattern matching equations, where these are in constructor form.

We intend this paper, among other things, to help push towards an appropriate syntax for *programs* presented in this style.

Such an analysis is also familiar territory to readers of Wadler's work in the early 1990's (Wadler, 1994). All of these studies, however, have largely focused on the functional fragment. Inductively-defined datatypes have received rather less attention, not least because induction interferes with cut-elimination—precisely Gentzen's motivation for introducing sequent calculus in the first place.

This paper develops an analysis of interactive program development based on these ideas from sequent calculus. We wish to identify the following new insights:

- the application of a cut rule is not restricted to the right-hand side of a match; analysing cut-formulæ on the left offers considerable notational convenience, even when programming with simple types;
- the introduction of type dependency makes pattern matching a far more powerful notion than its explanation via a term language of `case`-expressions; we explore the consequences of this in detail below;
- one of these consequences is that control structures based on combining continuations can be reified as data—this gives us 'views for free!';
- by focusing on *interactive* development, we expose the possibility of a more flexible approach to programming.

The key feature of pattern matching in simply typed languages is that the structure of an arbitrary *value* in a datatype is explained. This explanation provides a canonical control structure for the programmer, analysed by Augusstson and others in terms of decision trees, which marries a switch on the outermost constructor symbol with the exposure of subexpressions (Augustsson, 1985).

In dependently typed languages, especially those used for inductive theorem proving, pattern matching arises from the application of an elimination rule (induction principle). The key feature of induction is that the result *type* is *instantiated*, and hence further explained, by the patterns.

This has several consequences for the programmer exploiting dependent types. Firstly, we can, and should, extend the syntax and semantics of pattern matching to go beyond traditional presentations involving `case`. In particular, one should identify, and record explicitly (in the decision tree), the elimination rule giving rise to the patterns. This gives us one kind of node, which we dub 'by-nodes', in our decision trees, and correspond directly to application of left rules.

Secondly, that we can, and should, reanalyse expressions on the right-hand side of pattern matches. A common idiom is to consider subsidiary case analyses on the outcome of some subcomputation, whose type, from a logical point of view, is a *cut formula*. In the trivial case of a Boolean case split, this idiom is made concrete by the use of *guards*. The introduction of languages with type dependency, however, considerably strengthens the utility of subsidiary case analyses on such cut formulæ/subcomputations, since they can *change the types* of further subcomputations. This gives a new kind of node in the decision tree, which we dub 'with-nodes', and corresponds to the use of cut. But now there is a twist in the tail: we analyse cuts in such a way as to permit *further* pattern matching; namely that cut formulæ are analysed on the *left*, not on the right as in current practice.

Finally, the *statement* of an induction principle for a datatype corresponds to identifying an allowable set of patterns with which to match on values of the type; that is to say, it furnishes a *view* in Wadler's sense (Wadler, 1987). Now, certain views, corresponding to the simple matches on constructors, or the standard structural induction principle for the datatype (Burstall, 1969; Nordström *et al.*, 1990), are always available for free. But the programmer may also define new views, provided only that the corresponding induction principle be *admissible*.

There is a standard method for achieving this: by *writing a program*. To be able to do so, however, demands an expressive enough type system, in which types can explicitly describe patterns. Given such a type structure, one may then go even further. The programmer may introduce a new datatype, whose 'view for free' is *precisely* the sought-after view. To establish this view for the old datatype, one constructs an isomorphism or, more generally, simply a surjection from the old datatype. Such surjections embrace Coquand's notion of 'covering' (Coquand, 1992), and are also obtained by writing programs with dependent types.

We conclude our technical discussion with the development of a typechecker for the simply typed lambda calculus. This presentation furnishes a view of raw lambda terms as either being well-typed, or containing the source of a type error. The implementation of this view is a proof that typechecking is decidable, and we show, in the course of writing the program, how the type of such erroneous terms emerges from the problem decomposition.

Acknowledgements

We gratefully acknowledge the support of the EPSRC, through grants GR/N 24988 and GR/R 72259. We also thank the organisers of Dagstuhl seminar 01141, 'Semantics of Proof Search', where we presented preliminary versions of some of these

ideas. Our main debt, however, is to the programmers who have inspired us: Rod Burstall, Fred McBride and Phil Wadler.

2 Programming with Decision Trees

We present a high-level syntax for functional programming in a dependent type theory. A function is defined by giving its type signature in a natural deduction style, and writing its **decision tree**, describing the hierarchical structure of tests by which it divides its arguments into finer and finer cases. In effect, a decision tree provides a compact notation for a much larger term in the raw type theory which may be generated in full by applying proof tactics in a top-down fashion, directed by the nodes of the tree. In this paper, we focus on the notational aspects, and direct the reader interested in the underlying terms to the existing literature (McBride, 1999; McBride, 2001b).

Each node of a decision tree is labelled with a **head**—a word borrowed from the vocabulary of logic programming, here used to mean the function symbol applied to patterns. Internal nodes specify a **method** by which this head can be refined, leading to a collection of heads with more detailed patterns, addressed by decision subtrees. The leaf nodes of a decision tree, signalled by the $\mapsto$ symbol, supply a **result**—an expression over the variables occurring in the patterns, giving the function a return value for that head.

This contrasts with the usual presentation of pattern-matching (Burstall, 1969; McBride, 1970), where a function is given by a prioritised association list mapping heads to results. Our decision trees resemble the output of Augustsson's algorithm for compiling flat pattern matching into a hierarchy of 'simple' case expressions. (Augustsson, 1985).

The identity function has a one-node tree:

$$\text{let} \quad \frac{t \ : \ T}{\mathbf{iden}\, t \ : \ T} \quad \mathbf{iden}\, t \ \mapsto \ t$$

Note that we presume any free variables occurring in the type signature, e.g. T above, to be universally quantified implicitly in the type of the function. These quantifiers are inserted as far to the left as type dependency permits, and we write them longhand, when we absolutely must, with Pollack's $|$ notation (Pollack, 1992). The actual type of **iden** is $\forall T|\mathsf{Type}.T \ \to \ T$. If we want to make such arguments explicit in applications, we subscript them: $\mathbf{iden}_{\mathbb{N}}$ is the identity function for the natural numbers.

We imagine that decision trees will be constructed interactively. Indeed, this is precisely the mode of operation supported by the proof assistant ALF and its successors (Magnusson, 1994; Coquand & Coquand, 1999). An **open** decision tree contains **open** nodes which await either a method or a result. These effectively pair goal types in the underlying theory, with their corresponding heads. The type signature of a function determines the initial open tree—a single node whose head has a fresh pattern variable for each argument. We may summarise of the status of an open node in more detail, giving its **context** of pattern variables, and a type which labels the actual result type with its corresponding head. A typical development thus begins:

$$\text{let} \quad \frac{\vec{x} \ : \ \vec{S}}{\mathbf{f}\,\vec{x} \ : \ R[\vec{x}]} \qquad \mathbf{f}\,\vec{x} \ \ ?$$

$$\vec{x} : \vec{S} \ \ \vdash_? \langle \mathbf{f}\,\vec{x} \ : \ R[x] \rangle$$

We use de Bruijn's **telescope** notation to indicate a sequence of typings, such as $\vec{x} : \vec{S}$ (de Bruijn, 1991). The notation $R[\vec{x}]$ indicates that R is a term over the $\vec{x}$, and we denote instantiations of the $\vec{x}$ with terms $\vec{s}$ in R by $R[\vec{s}]$.

Our practice will be to write an incomplete program as a type signature with an open tree, then to summarise any open nodes of interest below. At any stage, an open node will have a summary resembling

$$\vec{x} : \vec{X} \ \ \vdash_? \langle \mathbf{f}\,\vec{p}[\vec{x}] \ : \ R[\vec{p}[\vec{x}]] \rangle$$

We may **close** a node by supplying a result $r[\vec{x}]$ in $R[\vec{p}[\vec{x}]]$, yielding

$$\ldots\ \mathbf{f}\,\vec{p}[\vec{x}] \ \mapsto \ r[\vec{x}]$$

We grow a decision tree by applying a method to an open node, acquiring a collection of open subnodes. In order to specify a method, we must explain how to compute the summaries of the subnodes from the summaries of the original. The implementation of a method must perform the corresponding proof step, reducing the original goal to the subgoals given by the subnodes. In effect, our approach augments traditional construction by refinement with the bookkeeping of heads via annotations in types.

The next two sections specify two such methods, both of which have ready implementations given by established proof tactics. The 'by' method supports case analysis and structural recursion; the 'with' method introduces an intermediate computation for subsequent analysis. With-nodes offer new notational convenience, even when working with simple types. However, it is in the presence of inductive families of datatypes that these two methods show their true potential.

3 By-nodes

By-nodes invoke known methods of analysis, yielding subnodes which have access to more information. Their implementation is given by McBride's `ElimUnify` tactic, details of which can be found in (McBride, 2001b). By-nodes support the application of *arbitrary* case analysis and recursion operators. Every inductive family comes equipped with 'standard' operators for constructor-case analysis and constructor-guarded recursion, hence by-nodes easily subsume Coquand's notion of pattern-matching for dependent types, in which the connection to constructors is hard-wired (Coquand, 1992). Let us consider case analysis first, then add recursion later.

3.1 Case Analysis

A simple example is found in the 'bind' function, $\lessdot$, for the Maybe monad.

$$\text{data} \quad \frac{A \;:\; \mathsf{Type}}{\mathsf{Maybe}\,A \;:\; \mathsf{Type}} \quad \text{where} \quad \frac{a \;:\; A}{\mathsf{yes}\;a \;:\; \mathsf{Maybe}\,A} \qquad \frac{}{\mathsf{no} \;:\; \mathsf{Maybe}\,A}$$

$$\text{let} \quad \frac{f \;:\; A \to \mathsf{Maybe}\,B \qquad x \;:\; \mathsf{Maybe}\,A}{f \lessdot x \;:\; \mathsf{Maybe}\,B} \qquad \begin{array}{lll} f \lessdot x & & \text{by } \mathsf{Maybe}\text{-}\mathbf{Case}\;x \\ \left[\begin{array}{l} f \lessdot \mathsf{yes}\;a \\ f \lessdot \mathsf{no} \end{array}\right. & \begin{array}{l} \\ \end{array} & \begin{array}{l} \mapsto f\;a \\ \mapsto \mathsf{no} \end{array} \end{array}$$

Note that the program is the whole tree, grouping a node with its subtrees explicitly. Later in this section, we shall examine ways to reduce this textual overhead, inferring standard by-nodes from the textual clues their offspring inherit, but for now, let us write our programs in full.

The **eliminator** e in 'by e' must have a type which abstracts a scheme of pattern matching and/or recursion. The type of $\mathsf{Maybe}\text{-}\mathbf{Case}\;x$ is

$$\forall \Phi\!:\!\mathsf{Maybe}\,A \to \mathsf{Type}.\; (\forall a\!:\!A.\; \Phi\,(\mathsf{yes}\;a)) \;\to\; (\Phi\;\mathsf{no}) \;\to\; \Phi\;x$$

This type asserts that in any setting, abstracted by Φ, x can be split into the patterns $(\mathsf{yes}\;a)$ and no. It is the type of an 'elimination rule' for Maybe instantiated to eliminate a particular 'target'. We call such types **schemes**, and we have a little syntactic sugar for them. Let

$$e \;:\; \{\Phi\;\vec{q} : \vec{I} \;\lhd\; (\vec{x}_1 : \vec{X}_1)\;\Phi\;\vec{q}_1 \mid \ldots \mid (\vec{x}_n : \vec{X}_n)\;\Phi\;\vec{q}_n\}$$

$$\text{abbreviate } e \;:\; \forall \Phi\!:\!\forall \vec{i}\!:\!\vec{I}.\,\mathsf{Type}.\; (\forall \vec{x}_1\!:\!\vec{X}_1.\;\Phi\;\vec{q}_1) \;\to\; \cdots (\forall \vec{x}_n\!:\!\vec{X}_n.\;\Phi\;\vec{q}_n) \;\to\; \Phi\;\vec{q}$$

$$\mathsf{Maybe}\text{-}\mathbf{Case}\;x \;:\; \{\Phi\;x : \mathsf{Maybe}\,A \;\lhd\; (a : A)\;\Phi\;(\mathsf{yes}\;a) \mid ()\;\Phi\;\mathsf{no}\}$$

Logically, the type of a scheme asserts that any values of form $\vec{q}$ must match at least one of the $\vec{q}_j$. A program must explain what to do for each possible match. Note that we use q's for patterns in schemes, to distinguish them from the p's in heads. A program must explain what to do for each possible match: we acquire a subnode for each $\vec{q}_j$ which *unifies* with $\vec{q}$.

At the type theory level, the implementation chooses a suitable value for Φ. McBride calls this value the **motive**, because it explains the purpose of the elimination. The motive codes up the unification problem as a set of equations. The full details can be found in (McBride, 2001b), but we recaptiulate the basic technique, a commonplace of theorem-proving with inductively defined relations. For an open node,

$$\vec{x} : \vec{X} \ \ \vdash_? \langle \mathbf{f}\ \vec{p}[\vec{x}] \ : \ R[\vec{p}[\vec{x}]]\rangle$$

we take

$$\Phi \ := \ \lambda\vec{i}{:}\vec{I}.\,\forall\vec{x}{:}\vec{X}.\,\vec{i} = \vec{q} \ \rightarrow \langle \mathbf{f}\ \vec{p}[\vec{x}] \ : \ R[\vec{p}[\vec{x}]]\rangle$$

We write $\vec{i} = \vec{q}$ for a series of equational hypotheses and $\mathsf{refl}\ \vec{t}$ for the sequence of canonical proofs that each t equals itself. We now have

$$e\ \Phi\ ?_1\ \ldots ?_n\ \vec{x}\ (\mathsf{refl}\ \vec{q}) \ : \ \langle \mathbf{f}\ \vec{p}[\vec{x}] \ : \ R[\vec{p}[\vec{x}]]\rangle$$

where each $?_j$ is a subgoal corresponding to a case of the scheme

$$\vec{x}_j : \vec{X}_j;\ \vec{x}; \vec{X};\ \vec{q}_j = \vec{q} \ \vdash_? \langle \mathbf{f}\ \vec{p}[\vec{x}] \ : \ R[\vec{p}[\vec{x}]]\rangle$$

The machine now simplifies the equations $\vec{q}_j = \vec{q}$ by first-order unification, using the substitutivity of equality and the basic properties of datatype constructors. We use the algorithm introduced in (McBride, 1998). This replicates within type theory the unification kept implicit in (Coquand, 1992). There are three possible outcomes: it may compute a most general unifier σ_j taking $\vec{x}, \vec{x}_j$ to terms over some $\vec{x}'_j : \vec{X}'_j$; it may show that the equations have no unifier, yielding a vacuous solution for $?_j$; it may get stuck on a non-constructor equation.

If unification gets stuck for any case, the machine rejects the by-method. Otherwise, it forms a by-node, with the unified subgoals as its subnodes:

$$\vec{x}'_j : \vec{X}'_j \ \vdash_? \langle \mathbf{f}\ \vec{p}[\sigma_j\vec{x}] \ : \ R[\vec{p}[\sigma_j\vec{x}]]\rangle$$

These subnodes specify the continuations which must be passed to eliminator so that it can handle each possible outcome. In our $\mathsf{Maybe}\text{-}\mathbf{Case}\ x$ example, x unifies with each pattern, so we must supply both a 'success' and a 'failure' continuation.

$$\begin{array}{llll}
\ldots & \mathbf{f}\ \ \vec{p}[\vec{x}] \quad \text{by } e & \quad \text{e.g.} \quad & f \blacktriangleleft x \quad \text{by } \mathsf{Maybe}\text{-}\mathbf{Case}\ x \\
& \left[\begin{array}{ll} \vdots & \\ \mathbf{f}\ \vec{p}[\sigma_j\vec{x}] & ? \\ \vdots & \end{array}\right. & & \left[\begin{array}{ll} f \blacktriangleleft \mathsf{yes}\ a & ? \\ f \blacktriangleleft \mathsf{no} & ? \end{array}\right.
\end{array}$$

Unification gives us the *overlap* between the patterns being split and the patterns in the cases of the scheme. When we work with dependent types, this can simplify some cases and rule out others altogether. The 'tail of a vector' has become the routine example. The family Vect defined below, refines the type of lists with an index making the lengths of vectors explicit. We choose to put the head of a vector on the right, as this suits our later examples, where they represent typing contexts.

$$\text{data} \quad \frac{A \ : \ \mathsf{Type} \qquad n \ : \ \mathbb{N}}{\mathsf{Vect}\ A\ n \ : \ \mathsf{Type}} \quad \text{where} \quad \begin{array}{c} \dfrac{}{\varepsilon \ : \ \mathsf{Vect}\ A\ 0} \\[2ex] \dfrac{xs \ : \ \mathsf{Vect}\ A\ n \qquad x \ : \ A}{xs :: x \ : \ \mathsf{Vect}\ A\ (\mathsf{s}n)} \end{array}$$

The Vect family has a single case analysis operator:

$$\begin{array}{lll}\mathsf{Vect}\text{-}\mathbf{Case}_{A;n}\;xs & : & \{\Phi\;n : \mathbb{N};\;\; xs : \mathsf{Vect}\;A\;n \lhd \\ & & \;()\;\Phi\;0\;\varepsilon \\ & & \mid (n : \mathbb{N};\;\; xs : \mathsf{Vect}\;A\;n;\;\; x : A)\;\Phi\;(\mathsf{s}n)\;(xs :: x)\}\end{array}$$

Now, consider the programming problem

$$\text{let}\quad \frac{xs\;:\;\mathsf{Vect}\;A\;(\mathsf{s}n)}{\mathbf{vtail}\;xs\;:\;\mathsf{Vect}\;A\;n}\qquad \mathbf{vtail}\;xs\;\;?$$

$$A : \mathsf{Type};\;\; n : \mathbb{N};\;\; xs : \mathsf{Vect}\;A\;(\mathsf{s}n)\;\vdash_?\;\mathbf{vtail}\;xs\;\;:\;\;\mathsf{Vect}\;A\;n$$

When we apply the method 'by $\mathsf{Vect}\text{-}\mathbf{Case}\;xs$', we are splitting a *nonempty* vector. Unification rules out the possibility of the ε constructor. The decision tree is extended with only one subnode, which we may readily close:

$$\begin{array}{lll}\;\mathbf{vtail} & xs & \text{by}\;\mathsf{Vect}\text{-}\mathbf{Case}\;xs \\ \left[\,\mathbf{vtail}\;(xs :: x)\right. & \mapsto & xs\end{array}$$

In general, a single by-node may tell us about a many pattern variables, as well as refining the result type of the function. This is as it should be: we should expect the information obtained by testing to show up in more informative types which legitimize a wider range of subsequent activity.

3.2 By-nodes for Recursion

There is nothing to prevent 'inductive hypotheses' occurring in the scheme of a by-node eliminator. For example, the traditional notion of primitive recursion for $\mathbb{N}$ is given by the scheme

$$\mathbb{N}\text{-}\mathbf{Elim}\;n\;\;:\;\;\{\Phi\;n : \mathbb{N}\;\lhd\;\;()\;\Phi\;0 \mid (n : \mathbb{N};\;\;\Phi\;n)\;\Phi\;(\mathsf{s}n)\}$$

When we build a by-node with such a scheme, the context of the s-subnode acquires an inductive hypothesis, representing a set of recursive calls. The head annotation, copied from the original node into the motive, now tell us exactly which calls are permitted. For example

$$\text{let}\quad \frac{n, m\;:\;\mathbb{N}}{n + m\;:\;\mathbb{N}}\qquad \begin{array}{ll} \;\;n + m & \text{by}\;\mathbb{N}\text{-}\mathbf{Elim}\;n \\ \left[\begin{array}{ll} 0 + m & \mapsto\; m \\ \mathsf{s}n + m & \;\;? \end{array}\right. & \end{array}$$

$$n, m : \mathbb{N};\, (m' : \mathbb{N})\,\langle n + m'\;:\;\mathbb{N}\rangle\;\vdash_?\;\mathsf{s}n + m\;\;:\;\;\mathbb{N}$$

In our example, we may call $n + m'$, for any m'. We might define $+$ tail recursively, closing the node with $n + \mathsf{s}m$; we might also return $\mathsf{s}(n + m)$. More generally, we may make any recursive call for which the machine can find an appropriate head annotation in the context.

There is no hard-wired notion of recursion. We are free to use any scheme, provided we can find an eliminator which gives it an operational semantics. We could even add *general* recursion by asserting

$$\frac{x\ :\ T}{\mathbf{general}\, x\ :\ \{\Phi\ x : T\ \lhd\ (x : T;\ \forall y\!:\!T.\ \Phi\ y)\ \Phi\ x\}}$$

Logically, this is a bare-faced lie, but it can be given the obvious 'free beer tomorrow' operational semantics.

Even without going this far, we do not have to try too hard to improve on primitive recursion. For each inductive family of datatypes, F, the machine automatically constructs an operator, $\mathsf{F}\text{-}\mathbf{Rec}$, which permits recursion to strip off more than one constructor per step. For example,

$$\mathbb{N}\text{-}\mathbf{Rec}\ n\ :\ \{\Phi\ n : \mathbb{N}\ \lhd\ (n : \mathbb{N};\ \mathbb{N}\text{-}\mathbf{Memo}\ \Phi\ n)\ \Phi\ n\}$$

$\mathbb{N}\text{-}\mathbf{Memo}\ \Phi\ n$ is the type of a data structure which holds a value in $\Phi\ n'$ for each strict subterm $n' \prec n$. Giménez defines this structure inductively (Giménez, 1994); McBride defines it by computation on n (McBride, 1999):

$$\text{let}\quad \frac{\Phi\ :\ \mathbb{N} \to \mathsf{Type}\qquad n\ :\ \mathbb{N}}{\mathbb{N}\text{-}\mathbf{Memo}\ \Phi\ n\ :\ \mathsf{Type}} \qquad \begin{array}{l} \mathbb{N}\text{-}\mathbf{Memo}\ \Phi\quad n\ \text{ by } \mathbb{N}\text{-}\mathbf{Elim}\ n \\ \left[\begin{array}{lcl} \mathbb{N}\text{-}\mathbf{Memo}\ \Phi\ \ 0 & \mapsto & \mathsf{Unit} \\ \mathbb{N}\text{-}\mathbf{Memo}\ \Phi\ (\mathsf{s}n) & \mapsto & \mathbb{N}\text{-}\mathbf{Memo}\ \Phi\ n \times \Phi\ n \end{array}\right. \end{array}$$

The more n is instantiated with constructor patterns, the more $\mathbb{N}\text{-}\mathbf{Memo}\ \Phi\ n$ expands to reveal Φ for each **guarded** subterm, the more recursive calls become available simply by projection.

The general construction for $\mathsf{F}{-}\mathbf{Memo}$ and $\mathsf{F}\text{-}\mathbf{Rec}$ is given in (McBride, 1999). Nesting $\mathsf{F}\text{-}\mathbf{Rec}$ by-nodes on a sequence of arguments delivers (at least) the strength of their lexicographic combination, where 'outer' arguments may remain fixed if 'inner' ones decrease. The equational constraints in the motive reappear in the 'inductive hypotheses' as *matching* problems which have a trivial solution exactly when the recursive call matches the original head.

The separation of recursion from case analysis gives much greater flexibility to the programmer, where primitive recursion forces an immediate case analysis on any argument to which it is applied. For example, we may write the $\leq^?$ test for $\mathbb{N}$ by recursion on its *second* argument:

$$\text{let}\quad \frac{x, y\ :\ \mathbb{N}}{x \leq^? y\ :\ \mathsf{Bool}} \qquad \begin{array}{l} \quad x \leq^? y\ \text{ by } \mathbb{N}\text{-}\mathbf{Rec}\ y \\ \left[\begin{array}{l} x \leq^? y\ \text{ by } \mathbb{N}\text{-}\mathbf{Case}\ x \\ \left[\begin{array}{l} 0 \leq^? y \ \mapsto\ \mathsf{true} \\ \mathsf{s}x \leq^? y\ \text{ by } \mathbb{N}\text{-}\mathbf{Case}\ y \\ \left[\begin{array}{lcl} \mathsf{s}x \leq^? 0 & \mapsto & \mathsf{false} \\ \mathsf{s}x \leq^? \mathsf{s}y & \mapsto & x \leq^? y \end{array}\right. \end{array}\right. \end{array}\right. \end{array}$$

This freedom becomes even more important when working with dependent types.

We may wish to write a program over some $x : F\ y$ which does its case analysis on x but is recursive on the index y, which may have a totally unrelated structure. Indeed, this is how the first-order unification algorithm is given a structurally recursive presentation in (McBride, 2001c).

3.3 Hiding Obvious By-nodes

The full decision tree for $\leq^?$ makes quite cumbersome reading, even if interactive tools help with the writing. We can reduce this burden wherever the constructor symbols in the patterns give us a hint that standard case analysis and recursion operators have been used. We should like to flatten the standard parts of decision trees as much as possible, provided the erased structure, or at least an equivalent structure, can be recovered.

Firstly, outermost **F-Rec** nodes can be removed. The space of possible lexicographic combinations of subterm orderings on a fixed number of arguments is readily searched. Indeed, Abel and Altenkirch give an elegant algorithm for discovering lexicographic constructor-guarded recursion which extends to mutually defined functions (Abel & Altenkirch, 2000).

Secondly, **F-Case** nodes can be replaced by a flat collection of their subnodes, provided there is at least one. The presence of an unexplained constructor symbol in a pattern can be used as a prompt to insert a by-node which does explain it. Cornes gives an algorithm which serves exactly this purpose in (Cornes, 1997). However, we cannot expect the machine to recover a hierarchy of case analysis which effectively proves that a type is empty—type inhabitation is undecidable. For example, given the family of finite datatypes

$$\text{data}\quad \frac{n : \mathbb{N}}{\mathsf{Fin}\ n : \mathsf{Type}} \quad \text{where} \quad \frac{}{\mathsf{f0}_n : \mathsf{Fin}\,\mathsf{s}n} \quad \frac{i : \mathsf{Fin}\ n}{\mathsf{fs}_n\ i : \mathsf{Fin}\,\mathsf{s}n}$$

the following program cannot be reduced:

$$\text{let}\quad \frac{i : \mathsf{Fin}\ 0}{\mathbf{empty}\ i : \mathsf{Unit}} \quad \mathbf{empty}\ i\ \text{by}\ \mathsf{Fin}\text{-}\mathbf{Case}\ i$$

We could choose to allow the machine to search a little for emptiness proofs, perhaps trying *one* step of case analysis on each pattern variable. This would allow the above program to be given by its type signature alone! Even without this extra work, many familiar programs are flattened entirely:

$$\text{let}\quad \frac{x, y : \mathbb{N}}{x =^? y : \mathsf{Bool}} \quad \begin{array}{lcl} 0 =^? 0 & \mapsto & \mathsf{true} \\ 0 =^? \mathsf{s}y & \mapsto & \mathsf{false} \\ \mathsf{s}x =^? 0 & \mapsto & \mathsf{false} \\ \mathsf{s}x =^? \mathsf{s}y & \mapsto & x =^? y \end{array}$$

Of course, in trying to reconstitute the full decision tree for $=^?$, we can choose recursion on either argument and case analysis in either order. We see no reason to be particular about which choice the machine makes, provided that case analysis on families is preferred to case analysis on their indices: it seems foolish to examine an index, when the same information, forced by the type, can be obtained for free by unification.

4 With-nodes in Decision Trees

Case analysis on arguments may not determine the entire control flow through a function, nor expose all the information required to compute its result. Some functions must analyse the results of *intermediate* computations. A pure pattern-matching notation forces these computations to be invoked on the right-hand side, dislocating a part of the decision process. For example, consider the function which tests if a given label is in the domain of an association list—we use vectors for the lists and numbers for the labels.

$$\text{let}\quad \frac{lxs\ :\ \mathsf{Vect}\,(\mathbb{N}\times X)\ m \qquad n\ :\ \mathbb{N}}{lxs\ \mathbf{dom}^?\ n\ :\ \mathsf{Bool}}$$

$$\begin{array}{rcl} \varepsilon\ \mathbf{dom}^?\ n & \mapsto & \mathsf{false} \\ lxs :: (l, x)\ \mathbf{dom}^?\ n & \mapsto & \text{if } l =^? n \text{ then } \mathsf{true} \text{ else } lxs\ \mathbf{dom}^?\ n \end{array}$$

Similarly, we must lurch rightwards to unpack a recursive call:

$$\text{let}\quad \frac{xys\ :\ \mathsf{Vect}\,(A\times B)\ n}{\mathbf{unzip}\ xys\ :\ \mathsf{Vect}\,A\ n\ \times\ \mathsf{Vect}\,B\ n}$$

$$\begin{array}{lcl} \mathbf{unzip}\quad \varepsilon & \mapsto & (\varepsilon, \varepsilon) \\ \mathbf{unzip}\ (xys :: (x, y)) & \mapsto & \begin{array}{rl} \text{case} & \mathbf{unzip}\ xys \\ \text{of} & (xs, ys) \mapsto (xs :: x, ys :: y) \end{array} \end{array}$$

Worse, we may be forced to make a computation on the right before we are sure to have finished decomposing the arguments on the left: Consider testing if one tree is a subtree of another (presuming the equality test has been defined):

$$\text{let}\quad \frac{s, t\ :\ \mathsf{tree}}{s\ \mathbf{sub}^?\ t\ :\ \mathsf{Bool}}$$

$$\begin{array}{lcl} s\ \mathbf{sub}^?\ t & \mapsto & \text{if } s =^? t \text{ then } \mathsf{true} \\ & & \text{else}\ \begin{array}{rcl} \text{case} & t & \\ \text{of} & \mathsf{leaf} & \mapsto \mathsf{false} \\ & t_1\ \mathsf{node}\ t_2 & \mapsto s\ \mathbf{sub}^?\ t_1\ \mathbf{or}\ s\ \mathbf{sub}^?\ t_2 \end{array} \end{array}$$

In the case of Boolean testing, the *guard* notation, to our knowledge introduced in (McBride, 1970) and now standard in Haskell, offers some help. This allows for Boolean conditions—**guards**—to be attached to a head: once the pattern variables have been bound, these must evaluate to true for the match as a whole to be successful; otherwise the machine *resumes matching* with the remaining heads. However, true guards throw us to the right, perhaps before we know all we need. Further, guards have nothing to offer the non-Boolean intermediate value.

If our decision trees find themselves in need of some critical information at any point in their analysis, we permit them to ask for it. For example, when we have reached this stage in the development of **unzip**,

$$\mathbf{unzip}\,(xys :: xy)\quad ?$$

we can trigger the recursive call which rearranges xys, by creating a **with-node**:

$$\begin{array}{llll}
 & \mathbf{unzip}\,(xys :: xy) & \text{with}\ \mathbf{unzip}\,xys & \\
\big[& \mathbf{unzip}\,(xys :: xy) & \|\quad xsys & ?
\end{array}$$

A with-node invokes an intermediate computation—a **cut-term**—and makes its result available for analysis *on the left* by adding an extra column to the head, containing a fresh pattern variable. The $\|$ symbol persists in the subtree of the with-node separating the new **cut-pattern** from the old head. This subtree may now make further analysis of both new and old data. We may now add (and then flatten) by-nodes which apply **×-Case** to extract both heads and tails from their respective tuples, then build the result. The full code for **unzip** becomes:

$$\text{let}\quad \frac{xys\ :\ \mathsf{Vect}\,(A\times B)\,n}{\mathbf{unzip}\,xys\ :\ \mathsf{Vect}\,A\,n\ \times\ \mathsf{Vect}\,B\,n}$$

$$\begin{array}{llllll}
 & \mathbf{unzip} & \varepsilon & & \mapsto & (\varepsilon, \varepsilon) \\
 & \mathbf{unzip} & (xys :: xy) & \text{with}\ \mathbf{unzip}\,xys & & \\
\big[& \mathbf{unzip} & (xys :: (x, y)) & \|\quad (xs, ys) & \mapsto & (xs :: x, ys :: y)
\end{array}$$

Similarly, the subtree test becomes much clearer if we pull the equality test to the left, just as if it were a Boolean guard:

$$\text{let}\quad \frac{s, t\ :\ \mathsf{tree}}{s\ \mathbf{sub}^?\ t\ :\ \mathsf{Bool}}$$

$$\begin{array}{l}
\quad s\ \mathbf{sub}^?\ t \qquad\qquad \text{with}\ s =^? t \\
\left[\begin{array}{llll}
s\ \mathbf{sub}^?\ t & \| & \mathsf{true} & \mapsto\ \mathsf{true} \\
s\ \mathbf{sub}^?\ \mathsf{leaf} & \| & \mathsf{false} & \mapsto\ \mathsf{false} \\
s\ \mathbf{sub}^?\ (t_1\ \mathsf{node}\ t_2) & \| & \mathsf{false} & \mapsto\ s\ \mathbf{sub}^?\ t_1\ \mathbf{or}\ s\ \mathbf{sub}^?\ t_2
\end{array}\right.
\end{array}$$

By way of focusing attention where it is needed, we permit the omission of the text left of the $\|$ where it would simply copy that of the node above:

$$\text{let}\quad \frac{lxs\ :\ \mathsf{Vect}\,(\mathbb{N}\times X)\,m \qquad n\ :\ \mathbb{N}}{lxs\ \mathbf{dom}^?\ n\ :\ \mathsf{Bool}}$$

$$\begin{array}{rlll}
\varepsilon\ \mathbf{dom}^?\ n & & \mapsto & \mathsf{false} \\
lxs :: (l,x)\ \mathbf{dom}^?\ n & \text{with } l =^? n & & \\
\left[\begin{array}{l} \\ \\ \end{array}\right. & \Vert\ \mathsf{true} & \mapsto & \mathsf{true} \\
 & \Vert\ \mathsf{false} & \mapsto & lxs\ \mathbf{dom}^?\ n
\end{array}$$

Of course, we could have exploited the Boolean nature of $\mathbf{dom}^?$ to fold the testing into an **or**, as we did with $\mathbf{sub}^?$. The same is not true for the *projection* function, **assoc**, an example suggested by Pollack, drawn from his experiences with coding *records* in type theory (Pollack, 2000). In our world of total functions, the latter presents its own problems: what are we to do if the label does not occur? One approach is to lift **assoc** to a Maybe type. Another is to make occurrence in the domain a precondition to the application of **assoc**.

The use of partial operations with *de facto* preconditions is a common idiom in simply-typed programming, functional or not, but its correct deployment is left to the programmer's conscience. Dependent types allow us a number of ways to enforce preconditions through type information. Perhaps the least radical of these comes by reflecting Boolean values as types via the following family:

$$\text{data}\quad \frac{b\ :\ \mathsf{Bool}}{\mathsf{So}\,b\ :\ \mathsf{Type}} \quad \text{where} \quad \frac{}{\mathsf{oh}\ :\ \mathsf{So}\,\mathsf{true}}$$

Of course, we could define So computationally, as the function taking true to Unit and false to Empty, but we find the family a better way to document our usage of the singleton or empty type. We may now impose a Boolean precondition b on an operation by demanding an extra argument of type $\mathsf{So}\ b$. For example,

$$\text{let}\quad \frac{lxs\ :\ \mathsf{Vect}\,(\mathbb{N}\times X)\,m \qquad n\ :\ \mathbb{N} \qquad p\ :\ \mathsf{So}\,(lxs\ \mathbf{dom}^?\ n)}{\mathbf{assoc}\ lxs\ n\ p\ :\ X}$$

We can often satisfy the precondition without computing it at run time if the label was demonstrably put in the list. We begin as we did with $\mathbf{dom}^?$:

$$\begin{array}{llll}
\mathbf{assoc} & \varepsilon & n\ p & ? \\
\mathbf{assoc} & (lxs :: (l,x)) & n\ p & ?
\end{array}$$

The ε case should be impossible, and it is. The type of p is $\mathsf{So}\,(\varepsilon\ \mathbf{dom}^?\ n)$, which reduces to $\mathsf{So}\,\mathsf{false}$, clearly empty. We write:

$\mathbf{assoc}\,\varepsilon\ n\ p$ by So-**Case** p

What of the other case? What is the type of its p, and how do we exploit it, depending on the outcome of $l =^? n$? It is here that we need a more precise account of with-nodes. Consider a programming problem

$$\vec{x} : \vec{X}\ \vdash_?\ \langle \mathbf{f}\ \vec{p}\ :\ R\rangle$$

When we attempt to apply the method 'with t', we may divide the dependency graph of the $\vec{x} : \vec{X}$ in two, with the fewest $\vec{x}_b : \vec{X}_b$ below such that t is well-typed, and the remaining $\vec{x}_a : \vec{X}_a$ above. Up to a dependency-respecting permutation, our programming problem is

$$\vec{x}_b : \vec{X}_b;\ \vec{x}_a : \vec{X}_a \vdash_? \langle \mathbf{f}\ \vec{p}\ :\ R\rangle \qquad \text{where} \qquad \vec{x}_b : \vec{X}_b\ \vdash\ t : T$$

We compute the telescope $\vec{x}_a : \vec{X}'_a$ and the type R' by syntactically replacing every occurrence of the normal form of t in the normal forms of the $\vec{X}_a$ and R by a fresh variable w and we check that this abstraction has not broken any typings where the value of t was critical. That is, we check

$$\vec{x}_b : \vec{X}_b;\ w : T;\ \vec{x}_a : \vec{X}'_a\ \vdash\ R' : \mathsf{Type}$$

If this check fails, we reject the with-node. If all is well, we pose the subproblem

$$\vec{x}_b : \vec{X}_b;\ w : T;\ \vec{x}_a : \vec{X}'_a \vdash_? \langle \mathbf{f}\ \vec{p} \parallel w\ :\ R'\rangle$$

A solution to this problem yields a solution to the original when w is instantiated with t. In fact, a with-node's immediate child is the root node for a new locally defined function, $\mathbf{f}'$ which has access to everything in the parent context (including memo structures) and the new argument w. $\mathbf{f}\ \vec{p} \parallel w$ is just a convenient display syntax for $\mathbf{f}'\ \vec{x}_a\ w\ \vec{x}_b$. In effect, the with-node is just the programming analogue of Coq's `Pattern` tactic (Coq, 2001).

Now we know the type of p in the $::$-case of **assoc**:

$$\ldots;\ p : \mathsf{So}\,(lxs\ ::\ (l,x)\ \mathbf{dom}^?\ n \parallel l =^? n) \vdash_? \langle \mathbf{assoc}\,(lxs\ ::\ (l,x))\ n\ p\ :\ X\rangle$$

The evaluation of $\mathbf{dom}^?$ has got stuck just inside its with-node, because $l =^? n$ is not a constructor. Nor is it a variable, so we cannot simply do Bool-**Case** on it, but we now have the means to turn it into a variable! We were going to test $l =^? n$ anyway, but the with-node also abstracts its term from the type of p.

$$\begin{array}{l}
\quad\mathbf{assoc}\,(lxs\ ::\ (l,x))\ n\ p\ \text{with}\ l =^? n \\
\left[\ \qquad\qquad\qquad\qquad\qquad \parallel \quad b \quad ?\right.
\end{array}$$

$$\ldots;\ p : \mathsf{So}\,(lxs\ ::\ (l,x)\ \mathbf{dom}^?\ n \parallel b) \vdash_? \langle \mathbf{assoc}\,(lxs\ ::\ (l,x))\ n\ p \parallel b\ :\ X\rangle$$

Case analysis on b now allows the type of p to reduce still further:

$$\begin{array}{l}
\quad\mathbf{assoc}\,(lxs\ ::\ (l,x))\ n\ p\ \text{with}\ l =^? n \\
\left[\begin{array}{l} \qquad\qquad\qquad\qquad\qquad \parallel\ \mathsf{true}\ \ ? \\ \qquad\qquad\qquad\qquad\qquad \parallel\ \mathsf{false}\ \ ? \end{array}\right.
\end{array}$$

$$\begin{array}{lll}
\ldots;\ p : \mathsf{So}\ \mathsf{true} & \vdash_? & \langle \mathbf{assoc}\,(lxs\ ::\ (l,x))\ n\ p \parallel \mathsf{true}\ :\ X\rangle \\
\ldots;\ p : \mathsf{So}\,(lxs\ \mathbf{dom}^?\ n) & \vdash_? & \langle \mathbf{assoc}\,(lxs\ ::\ (l,x))\ n\ p \parallel \mathsf{false}\ :\ X\rangle
\end{array}$$

We may close both nodes. Here is the finished program:

$$\text{let}\quad \frac{lxs\ :\ \mathsf{Vect}\,(\mathbb{N}\times X)\ m \qquad n\ :\ \mathbb{N}p\ \ :\ \mathsf{So}\,(lxs\ \mathbf{dom}^{?}\ n)}{\mathbf{assoc}\ lxs\ n\ p\ :\ X}$$

$$\begin{array}{llll}
\mathbf{assoc}\ \ \varepsilon\ \ n\ p & \text{by}\ \mathsf{So}\text{-}\mathbf{Case}\ p & & \\
\mathbf{assoc}\ (lxs\ ::\ (l,x))\ n\ p\ \text{with} & l =^{?} n & & \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\ \ \| & \mathsf{true} & \mapsto & x \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\ \ \| & \mathsf{false} & \mapsto & \mathbf{assoc}\ lxs\ n\ p
\end{array}$$

Pollack suggests a *heterogeneous* variant of this problem, associating each label with a dependent pair containing a type and a term with that type in the data structure $\mathsf{Vect}\,(\mathbb{N}\times\exists A\!:\!\mathsf{Type}.\ A)\ m$. We should be able to write a function $\mathbf{assoc}^{T}$ to project out the type for a label, then make **assoc** produce a term of the type given by $\mathbf{assoc}^{T}$, with $\mathbf{dom}^{?}$ a precondition to both! Our notation handles this easily, using with-nodes to synchronize all three functions.

The introduction of with-nodes helps us to tidy up previously disparate fragments of testing, collocating them on the left by allowing the extension of heads with cut-patterns corresponding to the results of intermediate computations. The treatment is uniform where guards privilege Boolean values, and the acquisition of new data does not preclude further analysis of the old data within the same decision process.

Furthermore, by bringing intermediate values—whose types are Gentzen-style cut-formulæ—into the context under scrutiny and abstracting them from types, we give a clean account of the effect their subsequent analysis has on our knowledge of the rest of the problem. In contrast, a free-floating case-expression must either re-abstract every other piece of information it affects, or else yield highly non-local consequences. As we shall shortly see, with-nodes have great impact when used in conjunction with case analysis on dependent families.

5 Views through Inductive Families

We have said that by-nodes permit the application of non-standard eliminators, but we have thus far given no examples where we exploit this potential. In this section, we shall give several such examples, and we shall show how these non-standard eliminators may be manufactured from the standard ones, yielding the functionality of Wadler's 'views', and more (Wadler, 1987). Let us begin where he did, by providing the 'cons' view of our 'snoc'-vectors, where **cons** is defined:

$$\text{let}\quad \frac{x\ :\ A \qquad xs\ :\ \mathsf{Vect}\,A\ n}{x\ \mathbf{cons}\ xs\ :\ \mathsf{Vect}\,A\ (\mathsf{s}n)} \qquad \begin{array}{lll} x\ \mathbf{cons}\ \varepsilon & \mapsto & \varepsilon\ ::\ x \\ x\ \mathbf{cons}\ (xs\ ::\ y) & \mapsto & (x\ \mathbf{cons}\ xs)\ ::\ y \end{array}$$

We may now specify the 'cons' view:

$$\begin{array}{llrcll}
\mathbf{backwards}\ xs\ : & \{ & & \Phi & n : \mathbb{N}; & xs : \mathsf{Vect}\,A\ n \\
 & \ \lhd & () & \Phi & \mathsf{0} & \varepsilon \\
 & \ \mid & (x : A;\ xs\ \mathsf{Vect}\,A\ n) & \Phi & (\mathsf{s}n) & (x\ \mathbf{cons}\ xs)\}
\end{array}$$

This view can be used to write the **vlast** function. Just as with **vtail**, unification removes the ε case:

$$\text{let}\ \ \frac{xs\ :\ \mathsf{Vect}\,A\,(\mathsf{s}n)}{\mathbf{vlast}\ xs\ :\ A} \qquad \begin{array}{l} \mathbf{vlast}\quad\ \ xs \qquad\quad \text{by}\ \mathbf{backwards}\ xs \\ \left[\ \mathbf{vlast}\ (x\ \mathbf{cons}\ xs)\ \ \mapsto\ x \right. \end{array}$$

How might we implement **backwards**? Schemes are, in effect, the types of polymorphic continuation combinators. We could write a continuation-passing program, making use of the view recursively:

$$\begin{array}{lllll} & \mathbf{backwards} & \varepsilon & \Phi\ \phi_\varepsilon\ \phi_c & \mapsto\ \phi_\varepsilon \\ & \mathbf{backwards} & (xs :: x) & \Phi\ \phi_\varepsilon\ \phi_c & \text{by}\ \ \mathbf{backwards}\ xs \\ \lceil & \mathbf{backwards} & (\varepsilon :: x) & \Phi\ \phi_\varepsilon\ \phi_c & \mapsto\ \phi_c\ x\ \varepsilon \\ \lfloor & \mathbf{backwards} & ((x\ \mathbf{cons}\ xs) :: y) & \Phi\ \phi_\varepsilon\ \phi_c & \mapsto\ \phi_c\ x\ (xs :: y) \end{array}$$

However, there is a first-order method to achieve the same effect which appears as a recurring idiom in McKinna and Pollack's work on formal metatheory (McKinna & Pollack, 1999). Whenever they need to establish an alternative induction principle for a relation R, they introduce the relation R' which natively has that induction principle, and then show that R' includes R. We may do the same for Vect: instead of showing **backwards** for every suitable Φ, we may show it for the *smallest*, turning Φ into an inductive family and the continuations $\vec{\phi}$ into its constructors:

$$\text{data}\ \ \frac{xs\ :\ \mathsf{Vect}\,A\ n}{\mathsf{Back}\ n\ xs\ :\ \mathsf{Type}} \quad \text{where} \qquad \begin{array}{c} \dfrac{}{\mathsf{back}_\varepsilon\ :\ \mathsf{Back}\,0\,\varepsilon} \\[2ex] \dfrac{x\ :\ A \qquad xs\ :\ \mathsf{Vect}\,A\ n}{\mathsf{back}_c\ x\ xs\ :\ \mathsf{Back}\,(\mathsf{s}n)\ (x\ \mathbf{cons}\ xs)} \end{array}$$

We may show that Back covers the vectors:

$$\text{let}\ \ \frac{xs\ :\ \mathsf{Vect}\,A\ n}{\mathbf{back}\ xs\ :\ \mathsf{Back}\ n\ xs}$$

$$\begin{array}{llllll} & \mathbf{back} & \varepsilon & & \mapsto\ \mathsf{back}_\varepsilon & \\ & \mathbf{back} & (xs :: x) & \text{with}\ \ \mathbf{back}\ xs & & \\ \lceil & \mathbf{back} & (\varepsilon :: x) & \ \ \| \quad \mathsf{back}_\varepsilon & \mapsto & \mathsf{back}_c\ x\ \varepsilon \\ \lfloor & \mathbf{back} & ((x\ \mathbf{cons}\ xs) :: y) & \ \ \|\ (\mathsf{back}_c\ x\ xs) & \mapsto & \mathsf{back}_c\ x\ (xs :: y) \end{array}$$

The definition of **backwards** is now trivial:

$$\begin{array}{llllll} & \mathbf{backwards} & xs & \Phi\ \phi_\varepsilon\ \phi_c\ \text{with} & \mathbf{back}\ xs & \\ \lceil & \mathbf{backwards} & \varepsilon & \Phi\ \phi_\varepsilon\ \phi_c & \|\quad \mathsf{back}_\varepsilon & \mapsto\ \phi_\varepsilon \\ \lfloor & \mathbf{backwards} & (x\ \mathbf{cons}\ xs) & \Phi\ \phi_\varepsilon\ \phi_c & \|\ (\mathsf{back}_c\ x\ xs) & \mapsto\ \phi_c\ x\ xs \end{array}$$

In effect, **backwards** delivers the effect on xs of case analysis on **back** xs. The actual composition of the 'proof' delivered by **back** is irrelevant. This is such a simple and common construction that it can and should be done on the fly. We introduce a derived form—the **with-by-node**, taking a 'proof' e whose type is an instance of an inductive family F. This has the effect of 'with e', yielding cut-pattern

x, then 'by **F-Case** x', except that we *omit* x's column from the new heads. This makes **backwards** redundant, and simplifies **back**:

$$\begin{array}{lll} \mathbf{back} & \varepsilon & \mapsto \mathsf{back}_{\varepsilon} \\ \mathbf{back} & (xs :: x) & \text{with-by } \mathbf{back}\ xs \end{array}$$
$$\left[\begin{array}{lll} \mathbf{back} & (\varepsilon :: x) & \mapsto \mathsf{back}_{c}\ x\ \varepsilon \\ \mathbf{back} & ((x\ \mathbf{cons}\ xs) :: y) & \mapsto \mathsf{back}_{c}\ x\ (xs :: y) \end{array}\right.$$

What we have done is to explain non-standard pattern-matching via the refinement of index information which naturally accompanies the standard notion of case analysis for inductive families. We have also replaced a higher-order function combining continuations with a first-order function combining constructors, inverting Church's encoding of datatypes via higher-order combinators in the λ-calculus. Turning closed fragments of function spaces into data, not merely compositional and functionally interpretable but *inductive*, will, we hope, become a powerful commonplace of dependently typed programming. It is conceivable that programs which compute such 'concrete functions' only to interpret them immediately—exactly the behaviour of a with-by-node—can be transformed automatically into a more efficient continuation-passing form by *deforestation*, a technique for which we also have Wadler to thank (Wadler, 1990).

Wadler conceived his view notation as syntactic sugar for the insertion of mutally inverse coercions between datatypes, one of which admits pattern-matching, the other potentially abstract. The idea that a signature for an abstract data structure might hide its actual representation, but nonetheless offer an *admissible* notion of pattern-matching, overcomes a genuine problem in the engineering of modular code. Programming with admissible notions of pattern-matching is exactly what our by-nodes permit, with the bonus that the interface is given by a *type* which can be required of an exported method in the usual way. Moreover, this type enforces the 'no junk' direction of the bijection: Wadler is forced by an inexpressive type system to trust the programmer.

The presentation of views through inductive families also makes it easy to state 'no confusion' as the requirement that the 'covering' function delivers the only possible proof in each case. For example, to show that our 'cons' view of vectors is unambiguous, we may prove the following **uniqueness** property of its covering function:

$$\text{goal} \quad \frac{b\ :\ \mathsf{Back}\ n\ xs}{\mathbf{back}\ xs = b}$$

5.1 Views for Testing

The essence of pattern-matching is to connect a test on data with the exposure of the information to which we become entitled, given the test's result, encapsulating selector methods within a framework which ensures that they apply.

Views allow us to extend that framework to a wider class of tests by functional programming alone. We have no need to tinker with the implementation of pattern-matching to achieve support for clearer code, nor need we accept the unbridled search by which logic programs decompose data in terms of *defined* function symbols. For example, the following view expresses the linear ordering on the natural numbers $\mathbb{N}$, incorporating both the subtraction operation with the conditions undr which it is well-defined:

$$\text{data}\quad \frac{x, y \;:\; \mathbb{N}}{\mathsf{Compare}\; x\; y} \quad \text{where} \quad \begin{array}{c} \overline{\mathsf{compLt}\; x\; y \;:\; \mathsf{Compare}\; x\; (x + \mathsf{s}y)} \\[1em] \overline{\mathsf{compEq}\; x \;:\; \mathsf{Compare}\; x\; x} \\[1em] \overline{\mathsf{compGt}\; x\; y \;:\; \mathsf{Compare}\; (y + \mathsf{s}x)\; y} \end{array}$$

$$\text{let}\quad \frac{x, y \;:\; \mathbb{N}}{\mathbf{comp}\; x\; y \;:\; \mathsf{Compare}\; x\; y}$$

$$\begin{array}{lccl}
\mathbf{comp} & 0 & 0 & \mapsto \mathsf{compEq}\, 0 \\
\mathbf{comp} & 0 & (\mathsf{s}y) & \mapsto \mathsf{compLt}\, 0\; y \\
\mathbf{comp} & (\mathsf{s}x) & 0 & \mapsto \mathsf{compGt}\; x\; 0 \\
\mathbf{comp} & (\mathsf{s}x) & (\mathsf{s}y) & \text{with-by } \mathbf{comp}\; x\; y
\end{array}$$
$$\left[\begin{array}{lccl}
\mathbf{comp} & (\mathsf{s}x) & (\mathsf{s}(x + \mathsf{s}y)) & \mapsto \mathsf{compLt}\; (\mathsf{s}x)\; y \\
\mathbf{comp} & (\mathsf{s}x) & (\mathsf{s}x) & \mapsto \mathsf{compEq}\; (\mathsf{s}x) \\
\mathbf{comp} & (\mathsf{s}(y + \mathsf{s}x)) & (\mathsf{s}y) & \mapsto \mathsf{compGt}\; x\; (\mathsf{s}y)
\end{array}\right.$$

The $\mathsf{Compare}$ family refines the enumeration $\{\mathsf{LT}, \mathsf{EQ}, \mathsf{GT}\}$, traditionally used to type the decision function of an ordering, with indices which explain the implications of the result for the data being compared. The type of **comp** tells us—and the typechecker—that its result does actually pertain to its arguments, a fact we keep to ourselves in the simply typed account. The program is not so far from the traditional coding of comparison, subtraction, maximum and minimum operators, and it does the job of *all* of them.

The same analysis applies, even more urgently, to the functions which decide *equality* for datatypes. One bit is not very much information unless you know how it is to be interpreted: how is the typechecker supposed to know that $T[x]$ and $T[y]$ are the same type, just because a particular Boolean value—that of $x =^? y$—happens to be true? In this setting, the non-linear patterns in McBride's thesis, implemented via LISP's EQUAL predicate (McBride, 1970), become more than a notational convenience.

We can achieve this effect by giving the equality test a more informative type. For example, let us define the equality test for the datatype Simp of simple type expressions, which will prove useful in our example later on:

$$\text{data}\quad \frac{}{\mathsf{Simp} \;:\; \mathsf{Type}} \quad \text{where} \quad \frac{}{\mathsf{o} \;:\; \mathsf{Simp}} \quad \frac{S, T \;:\; \mathsf{Simp}}{S \supset T \;:\; \mathsf{Simp}}$$

We define a view, $\mathsf{SimpEq}^{?}$ which, for a given $S : \mathsf{Simp}$, splits any $T : \mathsf{Simp}$ into S or 'anything else'. We shall need a type family coding up 'Simp with S removed', and an embedding from that family back into Simp:

$$\text{data} \quad \frac{S \ : \ \mathsf{Simp}}{\mathsf{Simp} - S \ : \ \mathsf{Type}} \quad \text{where} \quad \ldots$$

$$\text{let} \quad \frac{S' \ : \ \mathsf{Simp} - S}{S \backslash S' \ : \ \mathsf{Simp}} \quad \ldots$$

$$\text{data} \quad \frac{S, T \ : \ \mathsf{Simp}}{\mathsf{SimpEq}^{?}\ S\ T \ : \ \mathsf{Type}} \qquad \frac{}{\mathsf{simpSame}\, S \ : \ \mathsf{SimpEq}^{?}\ S\ S} \qquad \frac{T' \ : \ \mathsf{Simp} - T}{\mathsf{simpDiff}\ S' \ : \ \mathsf{SimpEq}^{?}\ S\ (S \backslash S')}$$

We will 'discover' the constructors of $\mathsf{Simp} - S$ and the behaviour of $S \backslash S'$ as we write the covering function, $\mathbf{simpEq}^{?}$—by case analysis, then recursive views:

$$\text{let} \quad \frac{S, T \ : \ \mathsf{Simp}}{\mathbf{simpEq}^{?}\ S\ T \ : \ \mathsf{SimpEq}^{?}\ S\ T}$$

$$\begin{array}{lllll}
\mathbf{simpEq}^{?} & \mathsf{o} & \mathsf{o} & \mapsto & \mathsf{simpSame}\,\mathsf{o} \\
\mathbf{simpEq}^{?} & \mathsf{o} & (S \supset T) & & ? \\
\mathbf{simpEq}^{?} & (S \supset T) & \mathsf{o} & & ? \\
\mathbf{simpEq}^{?} & (S_1 \supset T_1) & (S_2 \supset T_2) & \multicolumn{2}{l}{\text{with-by } \mathbf{simpEq}^{?}\ S_1\ S_2;}
\end{array}$$
$$\left[\begin{array}{lllll}
\mathbf{simpEq}^{?} & (S \supset T_1) & (S \supset T_2) & \multicolumn{2}{l}{\text{with-by } \mathbf{simpEq}^{?}\ T_1\ T_2;} \\
\multicolumn{5}{l}{\left[\begin{array}{lllll}
\mathbf{simpEq}^{?} & (S \supset T) & (S \supset T) & \mapsto & \mathsf{simpSame}\,(S \supset T) \\
\mathbf{simpEq}^{?} & (S \supset T) & (S \supset T \backslash T') & & ?
\end{array}\right.} \\
\mathbf{simpEq}^{?} & (S \supset T_1) & (S \backslash S' \supset T_2) & & ?
\end{array}\right.$$

We now give $\mathsf{Simp} - S$ constructors which just package the contexts of our four open nodes and define $S \backslash S'$ to decode them in correspondence to the nodes' patterns:

$$\begin{array}{cll}
\frac{}{\mathsf{neqo}\ S\ T \ : \ \mathsf{Simp} - \mathsf{o}} & \mathsf{o} \backslash (\mathsf{neqo}\ S\ T) & \mapsto \ S \supset T \\[1ex]
\frac{}{\mathsf{neq}{\supset} : \ \mathsf{Simp} - (S \supset T)} & (S \supset T) \backslash \mathsf{neq}{\supset} & \mapsto \ \mathsf{o} \\[1ex]
\frac{S \ : \ \mathsf{Simp} \quad T' \ : \ \mathsf{Simp} - T}{\mathsf{neqT}\ T' \ : \ \mathsf{Simp} - (S \supset T)} & (S \supset T) \backslash (\mathsf{neqT}\ T') & \mapsto \ S \supset T \backslash T' \\[1ex]
\frac{S' \ : \ \mathsf{Simp} - \ S \quad T_2 \ : \ \mathsf{Simp}}{\mathsf{neqS}\ S'\ T_2 \ : \ \mathsf{Simp} - (S \supset T_1)} & (S \supset T_1) \backslash (\mathsf{neqS}\ S'\ T_2) & \mapsto \ S \backslash S' \supset T_2
\end{array}$$

We may now complete the definition of $\mathbf{simpEq}^{?}$:

$$\begin{array}{llll}
\mathbf{simpEq}^{?} & \mathsf{o} & \mathsf{o} & \mapsto \mathsf{simpSame}\,\mathsf{o} \\
\mathbf{simpEq}^{?} & \mathsf{o} & (S \supset T) & \mapsto \mathsf{simpDiff}\,(\mathsf{neqo}\,S\ T) \\
\mathbf{simpEq}^{?} & (S \supset T) & \mathsf{o} & \mapsto \mathsf{simpDiff}\,\mathsf{neq}\supset \\
\mathbf{simpEq}^{?} & (S_1 \supset T_1) & (S_2 \supset T_2) & \text{with-by}\,\mathbf{simpEq}^{?}\ S_1\ S_2;
\end{array}$$

$$\left[\begin{array}{llll}
\mathbf{simpEq}^{?} & (S \supset T_1) & (S \supset T_2) & \text{with-by}\,\mathbf{simpEq}^{?}\ T_1\ T_2; \\
\multicolumn{4}{l}{\left[\begin{array}{llll}
\mathbf{simpEq}^{?} & (S \supset T) & (S \supset T) & \mapsto \mathsf{simpSame}\,(S \supset T) \\
\mathbf{simpEq}^{?} & (S \supset T) & (S \supset T \backslash T') & \mapsto \mathsf{simpDiff}\,(\mathsf{neqT}\ T')
\end{array}\right.} \\
\mathbf{simpEq}^{?} & (S \supset T_1) & (S \backslash S' \supset T_2) & \mapsto \mathsf{simpDiff}\,(\mathsf{neqS}\,S'\ T_2)
\end{array}\right.$$

This construction can be made entirely systematic. The $\mathsf{SimpEq}^{?}$ family can be made parametric on triples consisting of a type, its ‘subtraction’ type and the \ embedding. The second of these need not be defined inductively—it can be defined by computation for every datatype. Indeed, given a *universe construction* for a collection of inductive datatypes, this ‘system’ can be turned into a *generic program.* In (McBride, 2001a), the first author gives just such a universe construction for the **regular** datatypes—a single inductive family capturing every datatype in the language closed under polynomial type functors and least fixed point.

5.2 Views for Selection

Selector operations allow us to extract the pieces of a data structure, but sometimes, when programming with dependent types, we would like to know more—namely, that the data structure really is the thing made from the pieces. Pattern-matching delivers this information directly, and views allow us to extend its scope to more fascinating varieties of selection.

For example, we may see an element of $\mathsf{Vect}\,A\,n$ as an array of n A’s, indexed (safely) by elements of $\mathsf{Fin}\ n$, as defined above. That is, we may use $\mathsf{fs}^m\ \mathsf{f0}_n : \mathsf{Fin}\ (m + \mathsf{s}n)$ as an index into a vector of that length, chopping it into a prefix $ys : \mathsf{Vect}\,A\,m$ and a suffix $xs :: x : \mathsf{Vect}\,A\,(\mathsf{s}n)$, with x being the element so extracted. Let us first give ourselves the language with which to express this:

$$\text{let}\quad \frac{m : \mathbb{N} \quad i : \mathsf{Fin}\ n}{\mathsf{fs}^m\ i : \mathsf{Fin}\ (m + n)} \qquad \begin{array}{lll} \mathsf{fs}^{0} & i & \mapsto i \\ \mathsf{fs}^{\mathsf{s}m} & i & \mapsto \mathsf{fs}\,(\mathsf{fs}^m\ i) \end{array}$$

$$\text{let}\quad \frac{xs : \mathsf{Vect}\,A\,n \quad ys : \mathsf{Vect}\,A\,m}{xs +\!\!+\, ys : \mathsf{Vect}\,A\,(m + n)} \qquad \begin{array}{ll} xs +\!\!+\, \varepsilon & \mapsto xs \\ xs +\!\!+\, (ys :: y) & \mapsto xs +\!\!+\, ys :: y \end{array}$$

We let $+\!\!+$ bind more tightly than $::$ to minimise the number of brackets in normal forms. Observe that the arguments of $+$ are the opposite way round to those of $+\!\!+$ because we are using ‘snoc-vectors’ and ‘cons-numbers’. Let us now state our view of vectors as arrays and show that it covers:

$$\text{data}\quad \frac{xs\ :\ \mathsf{Vect}\,A\,n \qquad i\ :\ \mathsf{Fin}\,n}{\mathsf{Chop}\,xs\,i\ :\ \mathsf{Type}}$$

$$\text{where}\quad \frac{xs\ :\ \mathsf{Vect}\,A\,n \qquad x\ :\ A \qquad ys\ :\ \mathsf{Vect}\,A\,m}{\mathsf{chopGlue}\,xs\,x\,ys\ :\ \mathsf{Chop}\,(xs\ ::\ x +\!\!+\, ys)\,(\mathsf{fs}^m\ \mathsf{f0}_n)}$$

$$\text{let}\quad \frac{xs\ :\ \mathsf{Vect}\,A\,n \qquad i\ :\ \mathsf{Fin}\,n}{\mathbf{chop}\,xs\,i\ :\ \mathsf{Chop}\,xs\,i}$$

$$\begin{array}{lllll}
 & \mathbf{chop} & (xs\ ::\ x) & \mathsf{f0} & \mapsto\ \mathsf{chopGlue}\,xs\,x\,\varepsilon \\
 & \mathbf{chop} & (xs\ ::\ y) & (\mathsf{fs}\ i) & \text{with-by } \mathbf{chop}\,xs\,i \\
\lbrack & \mathbf{chop} & ((xs\ ::\ x +\!\!+\, ys)\ ::\ y) & (\mathsf{fs}\,(\mathsf{fs}^m\ \mathsf{f0}_n)) & \mapsto\ \mathsf{chopGlue}\,xs\,x\,(ys\ ::\ y)
\end{array}$$

Selection presented in this way appeals strongly to our visual sense of the structure of data—we see as we do. Many other common functions could be given a similar treatment, Pollack's 'association list' example being a prime candidate. We leave this as an exercise for the reader.

6 What do Decision Trees do?

We have presented our *notation* for programming with decision trees, and we have asserted that these programs can be rendered as (rather large) terms in type theory by the mechanisms which underpin known tactics in the domain of theorem-proving. We have said relatively little about the equations which hold of such programs, either at the level of their reduction behaviour or of the equational laws which they satisfy. To some extent, this is because there is relatively little to say: the operational effect of a by-node is given entirely by that of its eliminator applied to the continuations generated from its subtrees. The instantiation of head patterns comes from unification with scheme patterns, rather than any native understanding of datatype constructors. We bring our own semantics.

Let us first consider what we might prove about the eliminators we use in by-nodes and thus deduce about the functions we build from them. Eliminator schemes are *theorems* which assert that their cases are exhaustive for the patterns being analysed. They do not ensure that the cases are disjoint, or even that an individual pattern is unambiguous. One can easily show that the pattern $n + m$ captures all natural numbers, but this alone does not determine which n and m will be chosen at a given match. It is the eliminator itself which makes these choices. The more we know about an eliminator, the more we know about the programs which use it.

We describe an eliminator with the additional property that its patterns are disjoint and unambiguous as a **partition**. There is a standard way to show that an eliminator is a partition:

$$\text{given} \quad e \;:\; \{\Phi\,\vec{q} : \vec{I} \;\lhd\; (\vec{x}_1 : \vec{X}_1)\,\Phi\,\vec{q}_1 \mid \ldots \mid (\vec{x}_n : \vec{X}_n)\,\Phi\,\vec{q}_n\}$$

$$\text{for each } j\text{, show} \quad \frac{\vec{q} = \vec{q}_j}{e\;\Phi\;\vec{\phi} = \phi_j\;\vec{x}_j}$$

The eliminator in a with-by-node arising from a view is a partition if and only if the covering function has the uniqueness property defined above.

If every by-node in a program is a partition, then we may prove an equational law about each leaf-node, conditional on equations relating the cut-patterns $\vec{w}$ to the original cut-terms $\vec{t}$:

$$\text{for} \quad \mathbf{f}\;\vec{p} \;\|\; \vec{w} \;\mapsto\; r$$

$$\text{we have} \quad \frac{\sigma_1 t_1 = w_1 \quad \ldots \quad \sigma_n t_n = w_n}{\mathbf{f}\;\vec{p} = r}$$

where σ_j is the composition of the substitutions which have been used to instantiate pattern variables beneath the 'with t_j' node. We observe that the conditions are trivial for cut-terms applying covering functions with the uniqueness property.

Even for by-nodes with overlapping schemes, we know that one of the continuations must be applied, hence we can reason about functions which use them as if their behaviour is nondeterministic. One way to do this is to formulate inversion principles for equations of the form $\mathbf{f}\;\vec{p} = r$ which deliver a number of cases covering the 'possible behaviours' of $\mathbf{f}$, but a discussion of this technique is outside the scope of this paper. Systematic support for reasoning about decision tree programs remains an active topic of our research. Overlapping views may provide us with the means to give intuitive pattern-matching presentations of programs involving search, echoing the McBrides' experiments with ambiguous patterns in the late 1980s (McBride & McBride, 1989). We plan to investigate the implementation of search via eliminators with monadically lifted schemes, inspired by Wadler's landmark account of failure and backtracking through lists (Wadler, 1985).

Turning to the *computational* behaviour of decision trees, let us first observe that we may regard every node as a program in its own right—it is a function from its context to its result type. We may turn each node into a separate definition which λ-abstracts its context, then returns an appropriate value: for a leaf-node, this is just the supplied result; a with-node applies its sub-node—exactly a cut in the logical sense; a by-node passes its subnodes as continuations to its eliminator. A decision tree thus becomes a tree of 'lets' in type theory.

If we allow a 'native' notion of pattern-matching, as proposed in (Coquand, 1992), we can exploit the known properties of the standard **F-Rec** and **F-Case** operators to replace clusters of our 'lets' with more complex programs. In particular, the 'native' behaviour of **F-Case** is exactly

$$\textsf{F-}\mathbf{Case}\;(\mathsf{c}_j\;\vec{x}_j)\;\Phi\;\vec{\phi} \leadsto \phi_j\;\vec{x}_j$$

An eliminator which applies F-**Case** is thus trivially a partition. A tree of F-**Case** applications delivers a **covering** in Coquand's sense, and the corresponding cluster of lets may thus be replaced by a single pattern-matching definition. Moreover, the original tree structure tells us how to compile this definition, Augustsson-style.

Further, the projections from the memo-structure F-**Memo**, used to define F-**Rec**, are computationally equal to the corresponding recursive applications of F-**Rec**. We may thus merge a cluster of F-**Rec** lets, replacing the projections from memo-structures with recursive calls.

The native pattern-matching programs which arise from these simplifications have a reduction behaviour which holds at the level of conversion for the original definitions (McBride, 1999). A simple simulation argument shows that the transformation preserves strong normalization, and we conjecture that an argument by orthogonality will deliver preservation of confluence. Decision trees built only from the standard operators flatten into single 'native' programs, hence we know that for these trees, $\mapsto$ really means $\leadsto$.

Decision trees which contain with-nodes or non-standard by-nodes nonetheless reduce to a set of mutually recursive native programs. McBride's OLEG system (built from spare parts of Pollack's LEGO) has a suite of tactics for manufacturing such sets of programs, interactively supporting the construction techniques which became our 'by-nodes' and 'with-nodes'. All the examples in this paper were developed interactively using OLEG.

7 A Simple Typechecker

We now present our main example, a *typechecking view* for simply typed λ-terms in Church style. We give a first-order inductive presentation of terms which follows a long tradition in the literature, from McKinna and Pollack's treatment of '`Vclosed`' terms, through to Bellegarde and Hook's monadic definition, recently rendered in Haskell (via polymorphic recursion) by Bird and Paterson (McKinna & Pollack, 1993; Bellegarde & Hook, 1995; Bird & Paterson, 1999).

$$\text{data} \quad \frac{n \;:\; \mathbb{N}}{\mathsf{Term}\; n \;:\; \mathsf{Type}} \quad \text{where} \quad \frac{i \;:\; \mathsf{Fin}\; n}{\mathsf{var}\; i \;:\; \mathsf{Term}\; n} \quad \frac{f, s \;:\; \mathsf{Term}\; n}{\mathsf{app}\, f\; s \;:\; \mathsf{Term}\; n}$$
$$\frac{S \;:\; \mathsf{Simp} \qquad t \;:\; \mathsf{Term}\; (\mathsf{s}n)}{\mathsf{lam}\; S\; t \;:\; \mathsf{Term}\; n}$$

Following Altenkirch and Reus, we may give an inductive presentation of just the *well-typed* terms of a given type, in a given context (Altenkirch & Reus, 1999). This amounts to writing down the rules of the type system in a syntax directed form:

$$\text{data}\quad \frac{\Gamma : \mathsf{Vect\ Simp}\ n \qquad T : \mathsf{Simp}}{\mathsf{Good}\ \Gamma\ T : \mathsf{Type}}$$

$$\text{where}\quad \frac{\Gamma : \mathsf{Vect\ Simp}\ n \qquad T : \mathsf{Simp} \qquad \Delta : \mathsf{Vect\ Simp}\ m}{\mathsf{gVar}\ \Gamma\ T\ \Delta : \mathsf{Good}\ (\Gamma :: T +\!\!+ \Delta)\ T}$$

$$\frac{f : \mathsf{Good}\ \Gamma\ (S \supset T) \qquad s : \mathsf{Good}\ \Gamma\ S}{\mathsf{gApp}\ f\ s : \mathsf{Good}\ \Gamma\ T} \qquad \frac{t : \mathsf{Good}\ (\Gamma :: S)\ T}{\mathsf{gLam}\ S\ t : \mathsf{Good}\ \Gamma\ (S \supset T)}$$

There is an obvious forgetful map, **g**, from Goods to Terms. We keep the type explicit, because we would like to see the type when we use **g** in a pattern.

$$\text{let}\quad \frac{t : \mathsf{Good}_n\ \Gamma\ T}{\mathbf{g}\ T\ t : \mathsf{Term}\ n} \qquad \begin{array}{lll} \mathbf{g}\ \ T & (\mathsf{gVar}_{n,m}\ \Gamma\ T\ \Delta) & \mapsto \mathsf{var}\ (\mathsf{fs}^m\ \mathsf{f0}_n) \\ \mathbf{g}\ \ T & (\mathsf{gApp}\ f\ s) & \mapsto \mathsf{app}\ (\mathbf{g}\ f)\ (\mathbf{g}\ s) \\ \mathbf{g}\ (S \supset T) & (\mathsf{gLam}\ S\ t) & \mapsto \mathsf{lam}\ S\ (\mathbf{g}\ T) \end{array}$$

Let us now specify our typechecker as a view which tells us whether or not a given Term is Good.

$$\text{data}\quad \frac{\Gamma : \mathsf{Vect\ Simp}\ n \qquad t : \mathsf{Term}\ n}{\mathsf{TypeCheck}^?\ \Gamma\ t : \mathsf{Type}}$$

$$\text{where}\quad \frac{t : \mathsf{Good}\ \Gamma\ T}{\mathsf{good}\ T\ t : \mathsf{TypeCheck}^?\ \Gamma\ (\mathbf{g}\ T\ t)} \qquad \frac{t : \mathsf{Bad}\ \Gamma}{\mathsf{bad}\ t : \mathsf{TypeCheck}^?\ \Gamma\ (\mathbf{b}\ t)}$$

We have not yet defined the type of Bad terms, nor its forgetful map, **b**. We shall 'discover' these in due course, just as in our development of the equality view. Here are their respective formation rule and signature:

$$\text{data}\quad \frac{\Gamma : \mathsf{Vect\ Simp}\ n}{\mathsf{Bad}\ \Gamma : \mathsf{Type}} \qquad \text{let}\quad \frac{t : \mathsf{Bad}_n\ \Gamma}{\mathbf{b}\ t : \mathsf{Term}\ n}$$

The typechecker is fairly straightforward, using the **chop** view to access the context, and the $\mathbf{simpEq}^?$ view to ensure that applications are well-typed. Let us begin by taking the term apart:

$$\text{let}\quad \frac{\Gamma : \mathsf{Vect\ Simp}\ n \qquad t : \mathsf{Term}\ n}{\mathbf{typeCheck}^?\ \Gamma\ t : \mathsf{TypeCheck}^?\ \Gamma\ t} \qquad \begin{array}{ll} \mathbf{typeCheck}^?\ \Gamma\ (\mathsf{var}\ i) & ? \\ \mathbf{typeCheck}^?\ \Gamma\ (\mathsf{app}\ f\ s) & ? \\ \mathbf{typeCheck}^?\ \Gamma\ (\mathsf{lam}\ S\ t) & ? \end{array}$$

A variable is always well-typed. The **chop** view extracts its type from the context:

$$\begin{array}{lll} \mathbf{typeCheck}^?\ \Gamma\ (\mathsf{var}\ i) & \text{with-by} & \mathbf{chop}\ \Gamma\ i \\ \big[\ \mathbf{typeCheck}^?\ (\Gamma :: T\ {}_{n}\!+\!\!+_{m}\ \Delta)\ (\mathsf{var}\ (\mathsf{fs}^m\ \mathsf{f0}_n)) & \mapsto & \mathsf{good}\ T\ (\mathsf{gVar}\ \Gamma\ T\ \Delta) \end{array}$$

An abstraction is well-typed if its body is. We call the typechecker recursively. Let us leave the 'bad' case for the time being:

$$\begin{array}{ll} \mathbf{typeCheck}^?\ \Gamma\ (\mathsf{lam}\ S\ t) & \text{with-by}\ \mathbf{typeCheck}^?\ (\Gamma :: S)\ t \\ \left[\begin{array}{l} \mathbf{typeCheck}^?\ \Gamma\ (\mathsf{lam}\ S\ (\mathbf{g}\ T\ t)) \\ \mathbf{typeCheck}^?\ \Gamma\ (\mathsf{lam}\ S\ (\mathbf{b}\ t)) \end{array}\right. & \begin{array}{l} \mapsto \mathsf{good}\ (S \supset T)\ (\mathsf{gLam}\ S\ t) \\ \quad ? \end{array} \end{array}$$

To typecheck an application, we first make sure its 'function' really is functional:

$$\begin{array}{l}
\mathbf{typeCheck}^{?}\;\Gamma\;(\mathsf{app}\;f\;s) \qquad\qquad \text{with-by } \mathbf{typeCheck}^{?}\;\Gamma\;f \\
\left[\begin{array}{ll}
\mathbf{typeCheck}^{?}\;\Gamma\;(\mathsf{app}\;(\mathbf{g}\;\mathsf{o}\;f)\;s) & ? \\
\mathbf{typeCheck}^{?}\;\Gamma\;(\mathsf{app}\;(\mathbf{g}\;(S\supset T)\;f)\;s) & ? \\
\mathbf{typeCheck}^{?}\;\Gamma\;(\mathsf{app}\;(\mathbf{b}\;f)\;s) & ?
\end{array}\right.
\end{array}$$

If so, we proceed to typecheck the argument:

$$\begin{array}{l}
\mathbf{typeCheck}^{?}\;\Gamma\;(\mathsf{app}\;(\mathbf{g}\;(S\supset T)\;f)\;s) \qquad \text{with-by } \mathbf{typeCheck}^{?}\;\Gamma\;s \\
\left[\begin{array}{ll}
\mathbf{typeCheck}^{?}\;\Gamma\;(\mathsf{app}\;(\mathbf{g}\;(S\supset T)\;f)\;(\mathbf{g}\;S'\;s)) & ? \\
\mathbf{typeCheck}^{?}\;\Gamma\;(\mathsf{app}\;(\mathbf{g}\;(S\supset T)\;f)\;(\mathbf{b}\;s)) & ?
\end{array}\right.
\end{array}$$

Once we know the argument's type, we must check that it coincides with the domain of the function:

$$\begin{array}{l}
\mathbf{typeCheck}^{?}\;\Gamma\;(\mathsf{app}\;(\mathbf{g}\;(S\supset T)\;f)\;(\mathbf{g}\;S'\;s)) \qquad \text{with-by } \mathbf{simpEq}^{?}\;S\;S' \\
\left[\begin{array}{ll}
\mathbf{typeCheck}^{?}\;\Gamma\;(\mathsf{app}\;(\mathbf{g}\;(S\supset T)\;f)\;(\mathbf{g}\;S\;s)) & \mapsto\;\mathsf{good}\;T\;(\mathsf{gApp}\;f\;s) \\
\mathbf{typeCheck}^{?}\;\Gamma\;(\mathsf{app}\;(\mathbf{g}\;(S\supset T)\;f)\;(\mathbf{g}\;(S\backslash S')\;s)) & ?
\end{array}\right.
\end{array}$$

We have five open nodes remaining. These correspond to the two basic type errors—non-function application and application mismatch—together with the three cases which propagate an internal type error outwards. It is now clear how to define Bad and its forgetful map, **b**. Just as we did with $\mathbf{simpEq}^{?}$, we scoop up the contexts and patterns from the open nodes.

$$\begin{array}{cl}
\dfrac{f\;:\;\mathsf{Good}_n\;\Gamma\;\mathsf{o}\qquad s\;:\;\mathsf{Term}\;n}{\mathsf{bNonFun}\;f\;s\;:\;\mathsf{Bad}\;\Gamma} & \begin{array}{l}\mathbf{b}\;(\mathsf{bNonFun}\;f\;s)\;\mapsto \\ \;\;\mathsf{app}\;(\mathbf{g}\;\mathsf{o}\;f)\;s\end{array} \\[2ex]
\dfrac{f\;:\;\mathsf{Good}\;\Gamma\;(S\supset T)\qquad s\;:\;\mathsf{Good}\;\Gamma\;(S\backslash S')}{\mathsf{bMismatch}\;f\;s\;:\;\mathsf{Bad}\;\Gamma} & \begin{array}{l}\mathbf{b}\;(\mathsf{bMismatch}\;f\;s)\;\mapsto \\ \;\;\mathsf{app}\;(\mathbf{g}\;(S\supset T)\;f)\;(\mathbf{g}\;(S\backslash S')\;s)\end{array} \\[2ex]
\dfrac{f\;:\;\mathsf{Good}\;\Gamma\;(S\supset T)\qquad s\;:\;\mathsf{Bad}\;\Gamma}{\mathsf{bArg}\;f\;s\;:\;\mathsf{Bad}\;\Gamma} & \begin{array}{l}\mathbf{b}\;(\mathsf{bArg}\;f\;s)\;\mapsto \\ \;\;\mathsf{app}\;(\mathbf{g}\;(S\supset T)\;f)\;(\mathbf{b}\;s)\end{array} \\[2ex]
\dfrac{f\;:\;\mathsf{Bad}_n\;\Gamma\qquad s\;:\;\mathsf{Term}\;n}{\mathsf{bFun}\;f\;s\;:\;\mathsf{Bad}\;\Gamma} & \begin{array}{l}\mathbf{b}\;(\mathsf{bFun}\;f\;s)\;\mapsto \\ \;\;\mathsf{app}\;(\mathbf{b}\;f)\;s\end{array} \\[2ex]
\dfrac{t\;:\;\mathsf{Bad}\;(\Gamma :: S)}{\mathsf{bLam}\;S\;t\;:\;\mathsf{Bad}\;\Gamma} & \begin{array}{l}\mathbf{b}\;(\mathsf{bLam}\;S\;t)\;\mapsto \\ \;\;\mathsf{lam}\;S\;(\mathbf{b}\;t)\end{array}
\end{array}$$

We may thus close the five open nodes and present the completed typechecker:

```
$\mathbf{typeCheck}^{?}\ \Gamma\ (\mathsf{var}\ i)$  with-by $\mathbf{chop}\ \Gamma\ i$
  $\mathbf{typeCheck}^{?}\ (\Gamma :: T\ {}_{n}{+\!\!+}_{m}\ \Delta)\ (\mathsf{var}\ (\mathsf{fs}^{m}\ \mathsf{f0}_{n})) \mapsto \mathsf{good}\ T\ (\mathsf{gVar}\ \Gamma\ T\ \Delta)$
$\mathbf{typeCheck}^{?}\ \Gamma\ (\mathsf{app}\ f\ s)$  with-by $\mathbf{typeCheck}^{?}\ \Gamma\ f$
  $\mathbf{typeCheck}^{?}\ \Gamma\ (\mathsf{app}\ (\mathbf{g}\ \mathsf{o}\ f)\ s) \mapsto \mathsf{bad}\ (\mathsf{bNonFun}\ f\ s)$
  $\mathbf{typeCheck}^{?}\ \Gamma\ (\mathsf{app}\ (\mathbf{g}\ (S \supset T)\ f)\ s)$  with-by $\mathbf{typeCheck}^{?}\ \Gamma\ s$
    $\mathbf{typeCheck}^{?}\ \Gamma\ (\mathsf{app}\ (\mathbf{g}\ (S \supset T)\ f)\ (\mathbf{g}\ S'\ s))$  with-by $\mathbf{simpEq}^{?}\ S\ S'$
      $\mathbf{typeCheck}^{?}\ \Gamma\ (\mathsf{app}\ (\mathbf{g}\ (S \supset T)\ f)\ (\mathbf{g}\ S\ s)) \mapsto \mathsf{good}\ T\ (\mathsf{gApp}\ f\ s)$
      $\mathbf{typeCheck}^{?}\ \Gamma\ (\mathsf{app}\ (\mathbf{g}\ (S \supset T)\ f)\ (\mathbf{g}\ (S \backslash S')\ s)) \mapsto \mathsf{bad}\ (\mathsf{bMismatch}\ f\ s)$
    $\mathbf{typeCheck}^{?}\ \Gamma\ (\mathsf{app}\ (\mathbf{g}\ (S \supset T)\ f)\ (\mathbf{b}\ s)) \mapsto \mathsf{bad}\ (\mathsf{bArg}\ f\ s)$
  $\mathbf{typeCheck}^{?}\ \Gamma\ (\mathsf{app}\ (\mathbf{b}\ f)\ s) \mapsto \mathsf{bad}\ (\mathsf{bFun}\ f\ s)$
$\mathbf{typeCheck}^{?}\ \Gamma\ (\mathsf{lam}\ S\ t)$  with-by $\mathbf{typeCheck}^{?}\ (\Gamma :: S)\ t$
  $\mathbf{typeCheck}^{?}\ \Gamma\ (\mathsf{lam}\ S\ (\mathbf{g}\ T\ t)) \mapsto \mathsf{good}\ (S \supset T)\ (\mathsf{gLam}\ S\ t)$
  $\mathbf{typeCheck}^{?}\ \Gamma\ (\mathsf{lam}\ S\ (\mathbf{b}\ t)) \mapsto \mathsf{bad}\ (\mathsf{bLam}\ S\ t)$
```

This is not just a program: it is a *proof* that typechecking is decidable for the simply typed λ-calculus in Church style. It does not merely say 'yes' or 'no', but rather explains each raw term as deriving by a forgetful map either from a typed term or a broken term. Its type guarantees that the term being checked really is the term it is given. Its analysis is concisely stated and imposes the conditions for well-typedness (and its complement) just as they are expressed in the typing rules.

Moreover, as its recursive calls show, it represents these two possibilities in a pattern-matching style, visibly delivering either a well-typed term which may be passed to an exception-free interpreter in the style of Augustsson and Carlsson (Augustsson & Carlsson, 1999), or a useful error diagnostic. The latter locates the *leftmost* type error in a raw term—its 'principal gripe'. It could easily be adapted to find *every* application of a well-typed non-function or mismatched application between two well-typed terms—useful information not only for error reporting, but also for type debugging and repair, as suggested by McAdam (McAdam, 1999).

8 The Conclusion is: Further Work required

The main discovery we have made in the light of this research is how little we know about functional programming with dependent types. It is no longer credible to conceive of dependently typed programming merely as a means to recover the legitimacy of programs which were lost to us when we moved from untyped languages to the Hindley-Milner system. We take its inherent complexity as an *opportunity*, rather than a *problem*, and we hope we have given good reason to believe that a programming notation which is sensitive to the new interplay between pattern-matching, intermediate computations and result types can exploit this potential with the minimum of difficulty.

More generally, we take the explosion of power which dependent types bring to programming as a cue to re-evaluate design choices about the language with which we express programs, the tools with which we construct programs, and the programs we choose to write in the first place. This includes reassessing the interfaces and implementations of standard data structures and algorithms, no less than any other programs.

We believe that the new languages, tools and libraries will profit considerably from the experience gained in the wider domain of interactive problem-solving with dependent types. Our new analysis of the left-hand sides of functional programs stems directly from sequent calculus. By adding the cut rule (with-nodes) and permitting arbitrary left rules (by-nodes), we introduce a compositional language of analysis on the left to match the compositional language of synthesis on the right. We credit Wadler with the insight that, by constructing views, we can and should choose to adapt our perceptions of data to match our conceptions of data. We reify his views directly, by our treatment of the left. So hurrah for Wadler! Welcome to the new programming.

References

Abel, A., & Altenkirch, T. (2000). A predicative analysis of structural recursion. *J. functional programming*, March.

Altenkirch, Thorsten, & Reus, Bernhard. (1999). Monadic presentations of lambda-terms using generalized inductive types. *Computer Science Logic 1999.*

Augustsson, L., & Carlsson, M. (1999). *An exercise in dependent types: A well-typed interpreter.* www.cs.chalmers.se/~augustss/cayenne/interp.ps.

Augustsson, Lennart. (1985). Compiling Pattern Matching. *In:* (Jouannaud, 1985).

Bellegarde, Francoise, & Hook, James. (1995). Substitution: A formal methods case study using monads and transformations. *Science of Computer Programming.*

Bird, Richard, & Paterson, Ross. (1999). de Bruijn notation as a nested datatype. *Journal of Functional Programming*, **9**(1), 77–92.

Burstall, Rod. (1969). Proving properties of programs by structural induction. *Computer journal*, **12**(1), 41–48.

Coq, L'Équipe. 2001 (Apr). *The Coq Proof Assistant Reference Manual.* pauillac.inria.fr/coq/doc/main.html.

Coquand, Catarina, & Coquand, Thierry. (1999). Structured Type Theory. *Workshop on Logical Frameworks and Metalanguages.*

Coquand, Thierry. 1992 (June). Pattern Matching with Dependent Types. *Proceedings of the Logical Framework workshop at Båstad.*

Cornes, Cristina. (1997). *Conception d'un langage de haut niveau de répresenatation de preuves.* Ph.D. thesis, Université Paris VII.

de Bruijn, N.G. (1991). Telescopic Mappings in Typed Lambda-Calculus. *Information and computation*, **91**, 189–204.

Dyckhoff, R., & Pinto, L. (1998). Cut-elimination and a permutation-free sequent calculus for intuitionistic logic. *Studia logica*, **60**, 107–118.

Gentzen, G. (1935). *Investigations into logical deduction.* North-Holland. Chap. 3 of (Szabo, 1969).

Giménez, E. (1994). Codifying guarded definitions with recursive schemes. *Pages 39–59 of:* Dybjer, Peter, Nordström, Bengt, & Smith, Jan (eds), *Types for proofs and programs, '94.* LNCS, vol. 1158. Springer-Verlag.

Herbelin, H. (1995). A λ-calculus structure isomorphic to sequent calculus. *Pages 67–75 of: Proceedings of CSL'94.* LNCS, vol. 933. Springer-Verlag.

Huet, G., & Plotkin, G. D. (eds). 1990 (May). *Electronic Proceedings of the First Annual BRA Workshop on Logical Frameworks (Antibes, France).*

Jouannaud, Jean-Pierre (ed). (1985). *Functional Programming Languages and Computer Architecture.* LNCS, vol. 201. Springer-Verlag.

Magnusson, Lena. (1994). *The implementation of ALF—A Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitutiton.* Ph.D. thesis, Chalmers University of Technology, Göteborg.

McAdam, Bruce J. (1999). Generalising techniques for type explanation. *Pages 243–252 of: Scottish functional programming workshop.* Heriot-Watt Department of Computing and Electrical Engineering Technical Report RM/99/9.

McBride, Conor. (1998). Inverting inductively defined relations in LEGO. *Pages 236–253 of:* Giménez, E., & Paulin-Mohring, C. (eds), *Types for proofs and programs, '96.* LNCS, vol. 1512. Springer-Verlag.

McBride, Conor. (1999). *Dependently Typed Functional Programs and their Proofs.* Ph.D. thesis, University of Edinburgh.

McBride, Conor. (2001a). *The Derivative of a Regular Type is its Type of One-Hole Contexts.* Electronically available.

McBride, Conor. (2001b). Elimination with a Motive. Callaghan, P., Luo, Z., McKinna, J., & Pollack, R. (eds), *Types for proofs and programs (proceedings of the international workshop, types'00).* LNCS. Springer-Verlag. (in preparation).

McBride, Conor. 2001c (Feb.). *First-Order Unification by Structural Recursion.* To appear in the Journal of Functional Programming.

McBride, F., & McBride, C.T. (1989). *Craft '89.* Queen's University, Belfast. User Manual.

McBride, Fred. (1970). *Computer aided manipulation of symbols.* Ph.D. thesis, Queen's University of Belfast.

McKinna, J., & Pollack, R. (1993). Pure type systems formalized. Bezem, M., & Groote, J.F. (eds), *Int. conf. typed lambda calculi and applications.* LNCS 664. Springer-Verlag.

McKinna, J., & Pollack, R. (1999). Some lambda calculus and type theory formalized. *Journal of automated reasoning*, **23**, 373–409. (Special Issue on Formal Proof, editors Gail Pieper and Frank Pfenning).

Pollack, Robert. 1992 (May). *Implicit syntax.* An earlier version of this paper appeared in (Huet & Plotkin, 1990).

Pollack, Robert. (2000). Dependently Typed Records for Representing mathematical structure. Aagard, & Harrison (eds), *Theorem Proving in Higher Order Logics, TPHOLs 2000.* LNCS, vol. 1869. Springer-Verlag.

Nordström, B., Petersson, K., & Smith, J. (1990). *Programming in Martin-Löf's type theory: an introduction.* Oxford University Press.

Pym, D. (1990). *Proofs, search and computation in general logic.* Ph.D. thesis, University of Edinburgh. Available as CST-69-90, also published as ECS-LFCS-90-125.

Szabo, M. (1969). *The collected papers of Gerhard Gentzen.* North-Holland.

Wadler, P. (1994). *A Curry-Howard isomorphism for sequent calculus.* Talk given at the EPSRC LogFIT Final Workshop, Fairbairn House, Leeds.

Wadler, Philip. (1985). How to Replace Failure by a list of Successes. *In:* (Jouannaud, 1985).

Wadler, Philip. (1987). Views: A way for pattern matching to cohabit with data abstraction. *Popl'87.* ACM.

Wadler, Philip. (1990). Deforestation: transforming programs to eliminate trees. *Theoretical computer science*, **73**, 231–248. (Special issue of selected papers from 2'nd ESOP.).