Epigram: Practical Programming with Dependent Types

Conor McBride

School of Computer Science and Information Technology
University of Nottingham

1 Motivation

Find the type error in the following Haskell expression:

```
if null xs then tail xs else xs
```

You can't, of course: this program is obviously nonsense unless you're a typechecker. The trouble is that only certain computations make sense if the `null xs` test is `True`, whilst others make sense if it is `False`. However, as far as the type system is concerned, the type of the `then` branch is the type of the `else` branch is the type of the entire conditional. Statically, the test is irrelevant. Which is odd, because if the test really were irrelevant, we wouldn't do it. Of course, `tail []` doesn't go wrong—well-typed programs don't go wrong—so we'd better pick a different word for the way they do go.

Abstraction and application, tupling and projection: these provide the 'software engineering' superstructure for programs, and our familiar type systems ensure that these operations are used compatibly. However, sooner or later, most programs inspect data and make a choice—at that point our familiar type systems fall silent. They simply can't talk about specific data. All this time, we thought our *programming* was strongly typed, when it was just our *software engineering*. In order to do better, we need a static language capable of expressing the significance of particular values in legitimizing some computations rather than others. We should not give up on programming.

James McKinna and I designed Epigram [27, 26] to support a way of programming which builds more of the intended meaning of functions and data into their types. Its *style* draws heavily from the Alf system [13, 21]; its *substance* from my to Randy Pollack's Lego system [20, 23] Epigram is in its infancy and its implementation is somewhat primitive. We certainly haven't got everything right, nor have we yet implemented the whole design. We hope we've got *something* right. In these notes, I hope to demonstrate that such nonsense as we have seen above is not inevitable in real life, and that the extra articulacy which dependent types offer is both useful and *usable*. In doing so, I seek to stretch your imaginations towards what programming can be if we choose to make it so.

1.1 What are Dependent Types?

Dependent type systems, invented by Per Martin-Löf [22] generalize the usual function types $S \to T$ to dependent function types $\forall x : S \Rightarrow T$, where T may mention—hence *depend* on—x. We still write $S \to T$ when T doesn't depend on x. For example, matrix multiplication may be typed[1]

$$\mathbf{mult} : \forall i, j, k{:}\mathsf{Nat} \Rightarrow \mathbf{Matrix}\ i\ j \ \to\ \mathbf{Matrix}\ j\ k \ \to\ \mathbf{Matrix}\ i\ k$$

Datatypes like $\mathbf{Matrix}\ i\ j$ may depend on values fixing some particular property of their elements—a natural number indicating size is but one example. A function can specialize its return type to suit each argument. The typing rules for abstraction and application show how:

$$\frac{x{:}S \vdash t\ :\ T}{\lambda x \Rightarrow t\ :\ \forall x{:}S \Rightarrow T} \qquad \frac{f\ :\ \forall x{:}S \Rightarrow T \qquad s\ :\ S}{f\ s\ :\ [s/x]T}$$

Correspondingly, $\mathbf{mult}\ \mathsf{2}\ \mathsf{3}\ \mathsf{1}\ :\ \mathbf{Matrix}\ \mathsf{2}\ \mathsf{3} \ \to\ \mathbf{Matrix}\ \mathsf{3}\ \mathsf{1} \ \to\ \mathbf{Matrix}\ \mathsf{2}\ \mathsf{1}$ is the specialized multiplier for matrices of the given sizes.

We're used to universal quantification expressing *polymorphism*, but the quantification is usually over types. Now we can quantify over all values, and these include the types, which are values in $\star$. Our $\forall$ captures many forms of abstraction uniformly. We can also see $\forall x{:}S \Rightarrow T$ as a logical formula and its inhabitants as a function which computes a *proof* of $[s/x]T$ given a particular value s in S. It's this correspondence between programs and proofs, the *Curry-Howard Isomorphism*, with the slogan 'Propositions-as-Types', which makes dependent type systems particularly suitable for representing computational logics.

However, if you want to check dependent types, be careful! Look again at the application rule; watch s hopping over the copula,[2] from the term side in the argument hypothesis (eg., our specific dimensions) to the type side in the conclusion (eg., our specific matrix types). With expressions in types, we must think again about when types are equal. Good old syntactic equality won't do: $\mathbf{mult}\ (\mathsf{1}{+}\mathsf{1})$ should have the same type as $\mathbf{mult}\ \mathsf{2}$, so $\mathbf{Matrix}\ (\mathsf{1}{+}\mathsf{1})\ \mathsf{1}$ should *be* the same type as $\mathbf{Matrix}\,\mathsf{2}\,\mathsf{1}$! If we want computation to preserve types, we need at least to identify types with the same *normal forms*. Typechecking requires the *evaluation* of previously typechecked expressions—the phase distinction is still there, but it's slipperier.

What I like about dependent types is their precise language of data structures. In Haskell, we could define a sequence of types for lists of fixed lengths

```
data List0 x = Nil
data List1 x = Cons0 x (List0 x)
data List2 x = Cons1 x (List1 x)
```

but we'd have to stop sooner or later, and we'd have difficulty abstracting over either the whole collection, or specific subcollections like lists of even length. In

[1] We may write $\forall x : X; y : Y \Rightarrow T$ for $\forall x : X \Rightarrow \forall y : Y \Rightarrow T$ and $\forall x_1, x_2 : X \Rightarrow T$ for $\forall x_1 : X; x_2 : X \Rightarrow T$. We may drop the type annotation where inferrable.

[2] By *copula*, I mean the ':' which in these notes is used to link a term to its typing: Haskell uses '::', and the bold use the set-theoretic '∈'.

Epigram, we can express the lot in one go, giving us the family of *vector* types with indices from Nat representing length. Nat is just an ordinary datatype.

$$\underline{\text{data}}\ \left(\frac{}{\mathsf{Nat}\ :\ \star}\right)\ \underline{\text{where}}\ \left(\frac{}{\mathsf{zero}\ :\ \mathsf{Nat}}\right)\ ;\ \left(\frac{n\ :\ \mathsf{Nat}}{\mathsf{suc}\ n\ :\ \mathsf{Nat}}\right)$$

$$\underline{\text{data}}\ \left(\frac{n\ :\ \mathsf{Nat}\ ;\ \ X\ :\ \star}{\mathsf{Vec}\ n\ X\ :\ \star}\right)\ \underline{\text{where}}\ \left(\frac{}{\mathsf{vnil}\ :\ \mathsf{Vec}\ \mathsf{zero}\ X}\right)$$

$$\left(\frac{x\ :\ X\ ;\ \ xs\ :\ \mathsf{Vec}\ n\ X}{\mathsf{vcons}\ x\ xs\ :\ \mathsf{Vec}\ (\mathsf{suc}\ n)\ X}\right)$$

Inductive families [15], like Vec, are collections of datatypes, defined mutually and systematically, indexed by other data. Now we can use the dependent function space to give the 'tail' function a type which prevents criminal behaviour:

$$\mathbf{vtail}\ :\ \forall n\!:\!\mathsf{Nat} \Rightarrow \forall X\!:\!\star \Rightarrow \mathsf{Vec}\ (\mathsf{suc}\ n)\ X\ \rightarrow\ \mathsf{Vec}\ n\ X$$

For no n is $\mathbf{vtail}\ n\ X\ \mathsf{vnil}$ well typed. Indexed types state properties of their data which functions can rely on. They are the building blocks of Epigram programming. Our $\mathbf{Matrix}\ i\ j$ can just be defined as a vector of columns, say:

$$\underline{\text{let}}\ \left(\frac{rows, cols\ :\ \mathsf{Nat}}{\mathbf{Matrix}\ rows\ cols\ :\ \star}\right)\ \mathbf{Matrix}\ rows\ cols\ \Rightarrow\ \mathsf{Vec}\ cols\ (\mathsf{Vec}\ rows\ \mathsf{Nat})$$

Already in Haskell, there are hooks available to crooks who want more control over data. One can exploit *non-uniform* polymorphism to enforce some kinds of structural invariant [31], like this

```
data Rect col x = Columns [col] | Longer (Rect (col, x))
type Rectangular = Rect ()
```

although this type merely enforces rectangularity, rather than a specific size. One can also collect into a `Vec` type class those functors which generate vector structures [24]. Matrix multiplication then acquires a type like

```
mult :: (Vec f,Vec g,Vec h)=> f (g Int) -> g (h Int) -> f (h Int)
```

Programming with these 'fake' dependent types is an entertaining challenge, but let's be clear: these techniques are cleverly dreadful, rather than dreadfully clever. Hideously complex dependent types certainly exist, but they express basic properties like size in a straightforward way—why should the length of a list be anything less ordinary than a *number*? In Epigram, it doesn't matter whether the size of a matrix is statically determined or dynamically supplied—the size invariants are enforced, maintained and exploited, regardless of phase.

1.2 What is Epigram?

Epigram is a dependently typed functional programming language. On the surface, the system is an integrated editor-typechecker-interpreter for the language, owing a debt to the Alf [13, 21] and Agda [12] family of proof editors. Underneath, Epigram has a tactic-driven proof engine, like those of Coq [11] and Epigram's immediate ancestor, the 'Oleg' variant of Lego [20, 23]. The latter has proof tactics which mimic Alf's pattern matching style of proof in a more spartan type theory (Luo's UTT [19]); James McKinna and I designed Epigram [27] as a 'high-level programming' interface to this technology. An Epigram program is really a tree of proof tactics which drive the underlying construction in UTT.

But this doesn't answer the wider cultural question of what Epigram is. How it does it relate to functional languages like SML [29] and Haskell [32]? How does it relate to previous dependently typed languages like DML [39] and Cayenne [4]? How does it relate pragmatically to more conventional ways of working in type theory in the systems mentioned above? What's new?

I'll return to these questions at the end of these notes, when I've established more of the basis for a technical comparison, but I can say this much now: DML refines the ML type system with numerical indexing, but the programs remain the same—erase the indices and you have an ML program; Cayenne programs are LazyML programs with a more generous type system, including programs at the type level, but severely restricted support for inductive families. Epigram is not an attempt to strap a more powerful type system to standard functional programming constructs—it's rather an attempt to rethink what programming can become, given such a type system.

Dependent types can make explicit reference to programs and data. They can talk about program*ming* in a way that simple types can't. In particular, an *induction principle* is a dependent type. We learned this one as children:

$$\begin{aligned}\mathbf{NatInd} \;:\; & \forall P : \mathsf{Nat} \to \star \Rightarrow \\ & P\ \mathsf{zero} \to \quad (\forall n : \mathsf{Nat} \Rightarrow P\ n \to P\ (\mathsf{suc}\ n)) \to \\ & \forall n : \mathsf{Nat} \Rightarrow P\ n\end{aligned}$$

It gives rise to a proof technique—to give a proof of a more general proposition $P\ n$, give proofs that P holds for more specific *patterns* which n can take. Now cross out 'proof' and write 'program'. The induction principle for Nat specifies a particular strategy of case analysis and recursion, and Epigram can read it as such. Moreover, we can readily execute 'proofs' by induction, recursively applying the step program to the base value, to build a proof for any specific n:

$$\begin{aligned}&\mathbf{NatInd}\ P\ mz\ ms\ \mathsf{zero} &&\leadsto mz\\ &\mathbf{NatInd}\ P\ mz\ ms\ (\mathsf{suc}\ n) &&\leadsto ms\ n\ (\mathbf{NatInd}\ P\ mz\ ms\ n)\end{aligned}$$

Usually, functional languages have hard-wired constructs for constructor case analysis and general recursion; Epigram supports programming with *any* matching and recursion which you can specify as an induction principle and implement as a function. Epigram also supports a first-order method of implementing new induction principles—they too arise from inductive families.

It may surprise (if not comfort) functional programmers to learn that dependently typed programming seems odd to type theorists too. Type theory is usually seen either as the integration of 'ordinary' programming with a logical *superstructure*, or as a constructive logic which permits programs to be quietly extracted from proofs. Neither of these approaches really exploits dependent types in the programs and data themselves. At time of writing, neither Agda nor Coq offers substantial support for the kind of data structures and programs we shall develop in these notes, even though Alf and 'Oleg' did!

There is a tendency to see programming as a fixed notion, essentially untyped. In this view, we make sense of and organise programs by assigning types to them, the way a biologist classifies species, and in order to classify more the exotic creatures, like `printf` or the `zipWith` family, one requires more exotic types. This conception fails to engage with the full potential of types to make a positive contribution to program construction. Given what types can now express, let us open our minds afresh to the design of programming language *constructs*, and of programming *tools* and of the *programs* we choose to write anyway.

1.3 Overview of the Remaining Sections

2 Warm Up; Add Up tries to give an impression of Epigram's interactive style programming and the style of the programs via very simple examples—addition and the Fibonacci function. I expose the rôle of dependent types behind the scenes, even in simply typed programming.

3 Vectors and Finite Sets introduces some very basic datatype families and operations—I explore Vec and also the family Fin of finite enumeration types, which can be used to index vectors. I show how case analysis for dependent types can be more powerful and more subtle than its simply typed counterpart.

4 Representing Syntax illustrates the use of dependent types to enforce key invariants in expression syntax—in particular, the λ-calculus. I begin with untyped de Bruijn terms after the manner of Bird and Paterson [9] and end with simply typed de Bruijn terms in the manner of Altenkirch and Reus [2].[3] On the way, I'll examine some pragmatic issues in data structure design.

5 Is Looking Seeing? homes in on the crux of dependently typed programming—*evidence*. Programs over indexed datatypes may *enforce* invariants, but how do we *establish* them? This section explores our approach to data analysis [27], expressing inspection as a form of induction and deriving induction principles for old types by defining new families.

6 Well Typed Programs which Don't Go Wrong shows the development of two larger examples—a typechecker for simply typed λ-calculus which yields a typed version *of its input* or an informative diagnostic, and a tagless and total evaluator for the well typed terms so computed.

7 Epilogue reflects on the state of Epigram and its future in relation to what's happening more widely in type theory and functional programming.

[3] As I'm fond of pointing out, these papers were published almost simultaneously and have only one reference in common. I find that shocking!

I've dropped from these notes a more formal introduction to type theory: which introductory functional programming text explains how the typechecker works within the first forty pages? A precise understanding of type theory isn't necessary to engage with the ideas, get hold of the basics and start programming. I'll deal with technicalities as and when we encounter them. If you do feel the need to delve deeper into the background, there's plenty of useful literature out there—the next subsection gives a small selection.

1.4 Some Useful Reading

Scholars of functional programming and of type theory should rejoice that they now share much of the same ground. It would be terribly unfortunate for the two communities each to fail to appreciate the potential contribution of the other, through cultural ignorance. We must all complain less and read more!

For a formal presentation of the Epigram language, see '*The view from the left*' [27] For a deeper exploration of its underlying type theory—see '*Computation and Reasoning: A Type Theory for Computer Science*' [19] by Zhaohui Luo. User documentation, examples and solutions to exercises are available online [26].

Much of the impetus for Epigram comes from proof assistants. Proof and programming are similar activities, but the tools have a different feel. I can recommend '*Coq'Art*' [7] by Yves Bertot and Pierre Castéran as an excellent tutorial for this way of working, and for the Coq system in particular. The tactics of a theorem prover animate the rules of its underlying type theory, so this book also serves as a good practical introduction to the more formal aspects.

The seminal textbook on type theory as a programming language is '*Programming in Martin-Löf's type theory: an introduction*' [35] by Bengt Nordström, Kent Petersson and Jan Smith. It is now fifteen years old and readily available electronically, so there's no excuse to consider type theory a closed book.

Type theorists should get reading too! Modern functional programming uses richer type systems to express more of the structure of data and capture more patterns of computation. I learned a great deal from '*Algebra of Programming*' by Richard Bird and Oege de Moor [8]. It's a splendid and eye-opening introduction to a more categorical and calculational style of correct program construction. '*Purely Functional Data Structures*' by Chris Okasaki [30] is a delightful compendium of data structures and algorithms, clearly showing the advantages of fitting datatypes more closely to algorithms.

The literature on overloading, generic programming, monads, arrows, higher-order polymorphism is too rich to enumerate, but it raises important issues which type theorists must address if we want to make a useful contribution to functional programming in practice. I'd advise hungry readers to start with this very series of Advanced Functional Programming lecture notes.

1.5 For Those of you Watching in Black & White

Before we start in earnest, let's establish typographical conventions and relate the system's display with these notes. Epigram's syntax is two-dimensional: the buffer contains a *document* with a rectangular region selected—highlighted with a bright background. A *document* is a vertical sequence of lines; a *line* is a horizontal sequence of boxes; a *box* is either a character, or a bracket containing a document. A *bracket* is either a

```
       (          !
group, !   · · ·  ! , which has the usual functions of parenthesis, or a
       !          )
```

```
      [          !
shed, !   · · ·  ! , where you can tinker with text as you please. An Epigram
      !          ]
```

line may thus occupy more than one ASCII line. If a bracket is opened on a physical line, it must either be closed on that line or *suspended* with a `!`, then *resumed* on the next physical line with another `!`. I hasten to add that the Epigram editor does all of this box-drawing for you. You can fit two Epigram lines onto one physical line by separating them with `;`, and split one Epigram line into two physical lines by prefixing the second with `%`.

The Epigram document is a syntax tree in which leaves may be sheds—their contents are monochrome and belong to you. You can edit any shed without Epigram spying on you, but the rest of the document gets *elaborated*—managed, typechecked, translated to UTT, typeset, coloured in and generally abused by the system. In particular, Epigram colours recognized identifiers. There is only one namespace—this colour is just for show. If you can't see the colours in your copy of these notes, don't worry: I've adopted font and case conventions instead.

Blue	sans serif, uppercase initial	type constructor
red	sans serif, lowercase initial	data constructor
green	serif, boldface	defined variable
purple	serif, italic	abstracted variable
black	serif, underlined	reserved word

These conventions began in my handwritten slides—the colour choices are more or less an accident of the pens available, but they seem to have stuck. Epigram also has a convention for background colour, indicating the elaboration status of a block of source code.

white (light green when selected) indicates successful elaboration

yellow indicates that Epigram cannot yet see why a piece of code is good

brown indicates that Epigram can see why a piece of code is bad

Yellow backgrounds come about when typing constraints cannot yet be solved, but it's still possible for the variables they involve to become more instantiated, allowing for a solution in the future.

There are some pieces of ASCII syntax which I cannot bring myself to uglify in LaTeX. I give here the translation table for tokens and for the extensible delimiters of two-dimensional syntax:

$\star$	$\forall$	λ	$\to$	$\wedge$	$\Rightarrow$	$\Leftarrow$	$\left(\quad\right)$	$\left[\quad\right]$	$\overline{\qquad}$
`*`	`all`	`lam`	`->`	`/\`	`=>`	`<=`	`( !` `! )`	`[ !` `! ]`	`---`

Moreover, to save space here, I adopt an end-of-line style with braces {}, where the system puts them at the beginning.

At the top level, the document is a vertical sequence of **declarations** delineated by *rules*. A *rule* is a sequence of at least three `---`. The initial document has just one declaration, consisting of a shed, waiting for you to start work.

1.6 Acknowledgements

I'd like to thank the editors, Tarmo Uustalu and Varmo Vene, and the anonymous referees, for their patience and guidance. I'd also like to thank my colleagues and friends, especially James McKinna, Thorsten Altenkirch, Zhaohui Luo, Paul Callaghan, Randy Pollack, Peter Hancock, Edwin Brady, James Chapman and Peter Morris. Sebastian Hanowski and Wouter Swierstra deserve special credit for the feedback they have given me. Finally, to those who were there in Tartu, thank you for making the experience one I shall always value.

This work was supported by EPSRC grants GR/R72259 and EP/C512022.

2 Warm Up; Add Up

Let's examine the new technology in the context of a simple and familiar problem: adding natural numbers. We have seen this Epigram definition:[4]

$$\underline{\text{data}}\ \left(\frac{}{\mathsf{Nat}\ :\ \star}\right)\ \ \underline{\text{where}}\ \left(\frac{}{\mathsf{zero}\ :\ \mathsf{Nat}}\right)\ ;\ \left(\frac{n\ :\ \mathsf{Nat}}{\mathsf{suc}\ n\ :\ \mathsf{Nat}}\right)$$

We can equally well define the natural numbers in Epigram as follows:

$$\underline{\text{data}}\ \mathsf{Nat}\ :\ \star\ \ \underline{\text{where}}\ \mathsf{zero}\ :\ \mathsf{Nat}\ ;\ \ \mathsf{suc}\ :\ \mathsf{Nat} \to \mathsf{Nat}$$

In Haskell, we would write '`data Nat = Zero | Suc Nat`'.

The 'two-dimensional' version is a first-order presentation in the style of natural deduction rules[5] [34]. Above the line go hypotheses typing the arguments; below, the conclusion typing a template for a value. The declarations 'zero is a Nat; if n is a Nat, then so is $\mathsf{suc}\ n$' tell us what Nats look like. We get the actual types of zero and suc implicitly, by discharging the hypotheses.

4 Unary numbers are not an essential design feature; rest assured that primitive binary numbers will be provided eventually [10].

5 Isn't this a big step backwards for brevity? Or does Haskell's brevity depend on some implicit presumptions about what datatypes can possibly be?

We may similarly declare our function in the natural deduction style:

$$\underline{\text{let}} \quad \left(\frac{x, y \ : \ \mathsf{Nat}}{\mathbf{plus}\ x\ y \ : \ \mathsf{Nat}} \right)$$

This signals our intention to define a function called **plus** which takes two natural numbers and returns a natural number. The machine responds

$$\mathbf{plus}\ x\ y \quad [\,]$$

by way of asking 'So **plus** has two arguments, x and y. What should it do with them?'. Our type signature has become a *programming problem* to be solved interactively. The machine supplies a **left-hand side**, consisting of a function symbol applied to **patterns**, binding **pattern variables**—initially, the left-hand side looks just like the typical application of **plus** which we declared and the pattern variables have the corresponding names. But they are binding occurrences, not references to the hypotheses. The scope of a rule's hypotheses extends only to its conclusion; each problem and subproblem has its own scope.

Sheds $[\,]$ are where we develop solutions. The basic editing operation in Epigram is to expose the contents of a shed to the elaborator. We can write the whole program in one shed, then elaborate it; or we can work a little at a time. If we select a shed, Epigram will tell us what's in scope (here, x, y in Nat), and what we're supposed to be doing with it (here, explaining how to compute **plus** $x\ y$ in Nat). We may proceed by filling in a **right-hand side**, explaining how to reduce the current problem to zero or more subproblems.

I suggest that we seek to define **plus** by structural recursion on x, entering the right-hand side $\Leftarrow \underline{\text{rec}}\,x$. The '$\Leftarrow$' is pronounced 'by': it introduces right-hand sides which explain *by what means* to reduce the problem. Here we get

$$\begin{array}{l} \mathbf{plus}\ x\ y \ \Leftarrow \ \underline{\text{rec}}\ x\ \{ \\ \quad \mathbf{plus}\ x\ y \quad [\,]\ \} \end{array}$$

Apparently, nothing has changed. There is no presumption that recursion on x will be accompanied immediately (or ever) by case analysis on x. If you select the shed, you'll see that something has changed—the context of the problem has acquired an extra hypothesis, called a *memo-structure*. A precise explanation must wait, but the meaning of the problem is now 'construct **plus** $x\ y$, given x, y *and* the ability to call to **plus** on structural subterms of x'. So let us now analyse x, proceeding $\Leftarrow \underline{\text{case}}\ x$.

$$\begin{array}{l} \mathbf{plus}\ x\ y \ \Leftarrow \ \underline{\text{rec}}\ x\ \{ \\ \quad \mathbf{plus}\ x\ y \ \Leftarrow \ \underline{\text{case}}\ x\ \{ \\ \quad\quad \mathbf{plus}\ \mathsf{zero}\ y \quad [\,] \\ \quad\quad \mathbf{plus}\ (\mathsf{suc}\ x)\ y \quad [\,]\ \}\} \end{array}$$

The two subproblems are precisely those corresponding to the two ways x could have been made by constructors of Nat. We can certainly finish the first one off, entering $\Rightarrow\ y$. (The '$\Rightarrow$' is 'return'.) We can make some progress on the second by deciding to return the successor of something, $\Rightarrow\ \mathsf{suc}\ []$.

$$
\begin{array}{l}
\mathbf{plus}\ x\ y\ \Leftarrow\ \underline{\mathrm{rec}}\ x\ \{ \\
\quad \mathbf{plus}\ x\ y\ \Leftarrow\ \underline{\mathrm{case}}\ x\ \{ \\
\quad\quad \mathbf{plus}\ \mathsf{zero}\ y\ \ \Rightarrow\ y \\
\quad\quad \mathbf{plus}\ (\mathsf{suc}\ x)\ y\ \ \Rightarrow\ \mathsf{suc}\ []\ \}\}
\end{array}
$$

Select the remaining shed and you'll see that we have to fill in an element of Nat, given x, y and a subtly different memo-structure. Case analysis has instantiated the original argument, so we now have the 'ability to make recursive calls to **plus** on structural subterms of $(\mathsf{suc}\ x)$' which amounts to the more concrete and more useful 'ability to make recursive calls to **plus** on structural subterms of x, and on x itself'. Good! We can finish off as follows:

$$
\begin{array}{l}
\mathbf{plus}\ x\ y\ \Leftarrow\ \underline{\mathrm{rec}}\ x\ \{ \\
\quad \mathbf{plus}\ x\ y\ \Leftarrow\ \underline{\mathrm{case}}\ x\ \{ \\
\quad\quad \mathbf{plus}\ \mathsf{zero}\ y\ \ \Rightarrow\ y \\
\quad\quad \mathbf{plus}\ (\mathsf{suc}\ x)\ y\ \ \Rightarrow\ \mathsf{suc}\ (\mathbf{plus}\ x\ y)\ \}\}
\end{array}
$$

2.1 Who did the Work?

Two pages to add unary numbers? And that's a simple example? If it's that much like hard work, do we really want to know? Well, let's look at how much was work and how much was culture shock. We wrote the bits in the boxes:

$$
\boxed{\underline{\mathrm{let}}\quad \left(\frac{x, y\ :\ \mathsf{Nat}}{\mathbf{plus}\ x\ y\ :\ \mathsf{Nat}}\right)}\ ;\
\begin{array}{l}
\mathbf{plus}\ x\ y\ \boxed{\Leftarrow\ \underline{\mathrm{rec}}\ x}\ \{ \\
\quad \mathbf{plus}\ x\ y\ \boxed{\Leftarrow\ \underline{\mathrm{case}}\ x}\ \{ \\
\quad\quad \mathbf{plus}\ \mathsf{zero}\ y\ \boxed{\Rightarrow\ y} \\
\quad\quad \mathbf{plus}\ (\mathsf{suc}\ x)\ y\ \boxed{\Rightarrow\ \mathsf{suc}\ (\mathbf{plus}\ x\ y)}\ \}\}
\end{array}
$$

We wrote the type signature: we might have done that for virtue's sake, but virtue doesn't pay the rent. Here, we were repaid—we exchanged the usual hand-written left-hand sides for machine-generated patterns resulting from the conceptual step (usually present, seldom written) $\Leftarrow\ \underline{\mathrm{case}}\ x$. This was only possible because the machine already knew the type of x. We also wrote the $\Leftarrow\ \underline{\mathrm{rec}}\ x$, a real departure from conventional practice, but we got repaid for that too—we (humans and machines) know that **plus** is *total*.

Perhaps it's odd that the program's text is not entirely the programmer's work. It's the record of a partnership where we say what the plan is and the machine helps us carry it out. Contrast this with the 'type inference' model of programming, where we write down the details of the execution and the machine tries to guess the plan. In its pure form, this necessitates the restriction of plans to those which are blatant enough to be guessed. As we move beyond the Hindley-Milner system, we find ourselves writing down type information anyway.

'Type inference' thus has two aspects: 'top-level inference'—inferring type schemes, as with Hindley-Milner 'let'—and 'program inference given types'—inferring details when a scheme is instantiated, as with Hindley-Milner variables. Epigram rejects the former, but takes the latter further than ever. As types represent a higher-level design statement than programs, we should prefer to write types if they make programs cheaper.

Despite its interactive mode of construction, Epigram is fully compliant with the convention that a file of source code, however manufactured, contains all that's required for its recognition as a program. The bare text, without colour or other markup, is what gets elaborated. The elaboration process for a large code fragment just reconstructs a suitable interactive development offstage, cued by the program text—this is how we reload programs. You are free to negotiate your own compromise between incremental and batch-mode programming.

2.2 Where are the Dependent Types?

The type of **plus** is unremarkably simple, but if you were watching closely, you'll have noticed that the *machine* was using dependent types the whole time. Let's take a closer look. Firstly, a thought experiment—define a primitive recursion operator for `Nat` in Haskell as follows:

```
primRec{-p-} :: Nat -> p -> (Nat -> p -> p) -> p
primRec{-p-} Zero    mz ms = mz
primRec{-p-} (Suc n) mz ms = ms n (primRec{-p-} n mz ms)
```

I've made `primRec`'s type parameter explicit in a comment so we can follow what happens as we put it to work. How might we write **plus**? Try applying `primRec` to the first argument, then checking what's left to do:

```
plus :: Nat -> Nat -> Nat
plus = \ x -> primRec{-Nat -> Nat-} x mz ms where
  mz :: Nat -> Nat                          -- fill this in
  ms :: Nat -> (Nat -> Nat) -> Nat -> Nat  -- fill this in
```

Now we must fill in the *methods* `mz` and `ms`, but do their types show what rôle they play? There are seven occurrences[6] of `Nat` in their types—which is which? Perhaps *you* can tell, because you understand `primRec`, but how would a machine guess? And if we were defining a more complex function this way, we might easily get lost—try defining equality for lists using `foldr`.

However, recall our **NatInd** principle with operational behaviour just like `primRec` but a type which makes clear the relationship between the methods and the patterns for which they apply. If we're careful, we can use that extra information to light our way. Where `primRec` takes a constant type parameter, **NatInd** takes a *function* $P : \mathsf{Nat} \to \star$. If we take $P\ x \leadsto \mathsf{Nat} \to \mathsf{Nat}$, we

[6] If I had chosen a first-order recursion, there would have been as few as four, but that would presume to fix the second argument through the course of the recursion.

get the `primRec` situation. How might we use P's argument to our advantage? Internally, Epigram doesn't build **plus** : $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$ but rather a *proof*

$$\Diamond\mathbf{plus} \;:\; \forall x, y \,{:}\, \mathsf{Nat} \Rightarrow \langle\, \mathbf{plus}\; x\; y : \mathsf{Nat} \,\rangle$$

We can interpret $\langle\, \mathbf{plus}\; x\; y : \mathsf{Nat} \,\rangle$ as the property of x and y that '$\mathbf{plus}\; x\; y$ is a *computable* element of Nat'. This type is equipped with a constructor which packs up values and a function which runs computations

$$\frac{n \;:\; \mathsf{Nat}}{\mathsf{return}\langle\, \mathbf{plus}\; x\; y \,\rangle\; n \;:\; \langle\, \mathbf{plus}\; x\; y : \mathsf{Nat} \,\rangle} \qquad \frac{c \;:\; \langle\, \mathbf{plus}\; x\; y : \mathsf{Nat} \,\rangle}{\mathbf{call}\langle\, \mathbf{plus}\; x\; y \,\rangle\; c \;:\; \mathsf{Nat}}$$

such that

$$\mathbf{call}\langle\, \mathbf{plus}\; x\; y \,\rangle\; (\mathsf{return}\langle\, \mathbf{plus}\; x\; y \,\rangle\; n) \;\rightsquigarrow\; n$$

Given $\Diamond$**plus**, we may readily extract **plus**—apply and run!

$$\mathbf{plus} \rightsquigarrow \lambda x, y \Rightarrow \mathbf{call}\langle\, \mathbf{plus}\; x\; y \,\rangle\; (\Diamond\mathbf{plus}\; x\; y) \;:\; \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$$

Now, let's build $\Diamond$**plus** as a proof by **NatInd**:

$$\begin{array}{l}
\Diamond\mathbf{plus} \rightsquigarrow \mathbf{NatInd}\; (\lambda x \Rightarrow \forall y \,{:}\, \mathsf{Nat} \Rightarrow \langle\, \mathbf{plus}\; x\; y : \mathsf{Nat} \,\rangle)\; \mathbf{mz}\; \mathbf{ms} \\
\text{where } \;\mathbf{mz} \;:\; \forall y \,{:}\, \mathsf{Nat} \Rightarrow \langle\, \mathbf{plus}\; \mathsf{zero}\; y : \mathsf{Nat} \,\rangle \\
\qquad\quad\;\; \mathbf{ms} \;:\; \forall x \,{:}\, \mathsf{Nat} \Rightarrow (\forall y \,{:}\, \mathsf{Nat} \Rightarrow \langle\, \mathbf{plus}\; x\; y : \mathsf{Nat} \,\rangle) \\
\qquad\qquad\qquad\qquad\qquad \to \forall y \,{:}\, \mathsf{Nat} \Rightarrow \langle\, \mathbf{plus}\; (\mathsf{suc}\; x)\; y : \mathsf{Nat} \,\rangle
\end{array}$$

It's not hard to see how to generate the left-hand sides of the subproblems for each case—just read them off from their types! Likewise, it's not hard to see how to translate the right-hand sides we supplied into the proofs—pack them up with $\mathsf{return}\langle \cdots \rangle$ translate recursive calls via $\mathbf{call}\langle \cdots \rangle$:

$$\begin{array}{l}
\mathbf{mz} \rightsquigarrow \lambda y \Rightarrow \mathsf{return}\langle\, \mathbf{plus}\; \mathsf{zero}\; y \,\rangle\;\; y \\
\mathbf{ms} \rightsquigarrow \lambda x \Rightarrow \lambda xhyp \Rightarrow \lambda y \Rightarrow \\
\qquad\quad \mathsf{return}\langle\, \mathbf{plus}\; (\mathsf{suc}\; x)\; y \,\rangle\;\; (\mathsf{suc}\; (\mathbf{call}\langle\, \mathbf{plus}\; x\; y \,\rangle\; (xhyp\; y)))
\end{array}$$

From this proof, you can read off both the high-level program and the its low-level operational behaviour in terms of primitive recursion. And that's basically how Epigram works! Dependent types aren't just the basis of the Epigram *language*—the *system* uses them to organise even simply typed programming.

2.3 What are <u>case</u> and <u>rec</u>?

In the **plus** we actually wrote, we didn't use induction—we used <u>case</u> x and <u>rec</u> x. These separate induction into its aspects of distinguishing constructors and of justifying recursive calls. The keywords <u>case</u> and <u>rec</u> cannot stand alone, but <u>case</u> e and <u>rec</u> e are meaningful whenever e belongs to a datatype—Epigram constructs their meaning from the structure of that datatype.

In our example, $x : \mathsf{Nat}$, and Epigram give us

$$\begin{array}{l} \underline{\text{case}}\, x \;:\; \forall P : \mathsf{Nat} \to \star \;\Rightarrow \\ \qquad (P\ \mathsf{zero}) \;\to\; (\forall x' : \mathsf{Nat} \Rightarrow P\ (\mathsf{suc}\ x')) \;\to\; P\ x \end{array}$$

This is an induction principle instantiated at x with its inductive hypotheses chopped off: it just says, 'to do P with x, show how to do P with each of these patterns'. The associated computational behaviour puts proof into practice:

$$\begin{array}{ll} (\underline{\text{case}}\ \mathsf{zero}) & P\ mz\ ms \leadsto mz \\ (\underline{\text{case}}\ (\mathsf{suc}\ x)) & P\ mz\ ms \leadsto ms\ x \end{array}$$

There is nothing special about $\underline{\text{case}}\, x$. When elaborating $\Leftarrow e$, it's the *type* of e which specifies how to split a problem into subproblems. If, as above, we take

$$P \leadsto \lambda x \Rightarrow \forall y : \mathsf{Nat} \Rightarrow \langle \mathbf{plus}\ x\ y : \mathsf{Nat} \rangle$$

then the types of mz and ms give us the split we saw when we wrote the program. What about $\underline{\text{rec}}\, x$?

$$\begin{array}{l} \underline{\text{rec}}\, x \;:\; \forall P : \mathsf{Nat} \to \star \;\Rightarrow \\ \qquad (\forall x : \mathsf{Nat} \Rightarrow (\underline{\text{memo}}\ x)\ P \to P\ x) \;\to \\ \qquad P\ x \end{array}$$

This says 'if you want to do $P\ x$, show how to do it given access to P for everything structurally smaller than x'. This $(\underline{\text{memo}}\, x)$ is another gadget generated by Epigram from the structure of x's type—it uses the power of computation in types to capture the notion of 'structurally smaller':

$$\begin{array}{ll} (\underline{\text{memo}}\ \mathsf{zero}) & P \leadsto \mathsf{One} \\ (\underline{\text{memo}}\ (\mathsf{suc}\ n)) & P \leadsto (\underline{\text{memo}}\ n)\ P \;\wedge\; P\ n \end{array}$$

That is $(\underline{\text{memo}}\, x)\, P$ is the type of a big tuple which memoizes P for everything smaller than x. If we analyse x, the memo-structure computes,[7] giving us the trivial tuple for the zero case, but for $(\mathsf{suc}\ n)$, we gain access to $P\ n$. Let's watch the memo-structure unfolding in the inevitable Fibonacci example.

$$\begin{array}{lll} \underline{\text{let}} & \left(\dfrac{n : \mathsf{Nat}}{\mathbf{fib}\ n : \mathsf{Nat}}\right) ; & \mathbf{fib}\ n \;\Leftarrow\; \underline{\text{rec}}\ n\ \{ \\ & & \quad \mathbf{fib}\ n \;\Leftarrow\; \underline{\text{case}}\ n\ \{ \\ & & \qquad \mathbf{fib}\ \mathsf{zero} \;\Rightarrow\; \mathsf{zero} \\ & & \qquad \mathbf{fib}\ (\mathsf{suc}\ n) \;\Leftarrow\; \underline{\text{case}}\ n\ \{ \\ & & \qquad\quad \mathbf{fib}\ (\mathsf{suc}\ \mathsf{zero}) \;\Rightarrow\; \mathsf{suc}\ \mathsf{zero} \\ & & \qquad\quad \mathbf{fib}\ (\mathsf{suc}\ (\mathsf{suc}\ n)) \;\Rightarrow\; \boxed{[\,]}\ \}\}\} \end{array}$$

[7] Following a suggestion by Thierry Coquand, Eduardo Giménez shows how to separate induction into $\underline{\text{case}}$ and $\underline{\text{rec}}$ in [17]. He presents memo-structures inductively, to justify the *syntactic* check employed by Coq's `Fix` construct. The computational version is mine [23]; its unfolding memo-structures give you a *menu* of recursive calls.

If you select the remaining shed, you will see that the memo structure in the context has unfolded (modulo trivial algebra) to:

$$(\underline{\text{memo}}\ n)\ (\lambda x \Rightarrow \langle \mathbf{fib}\ x : \mathsf{Nat}\rangle) \quad \wedge \quad \langle \mathbf{fib}\ n : \mathsf{Nat}\rangle \quad \wedge \quad \langle \mathbf{fib}\ (\mathsf{suc}\ n) : \mathsf{Nat}\rangle$$

which is just as well, as we want to fill in $\mathbf{plus}\ (\mathbf{fib}\ n)\ (\mathbf{fib}\ (\mathsf{suc}\ n))$.

At this stage, the approach is more important than the details. The point is that programming with $\Leftarrow$ imposes *no fixed notion* of case analysis or recursion. Epigram does not have 'pattern matching'. Instead, $\Leftarrow$ admits whatever notion of problem decomposition is specified by the type of the expression (the *eliminator*) which follows it. The value of the eliminator gives the operational semantics to the program built from the solutions to the subproblems.

Of course, Epigram equips every datatype with $\underline{\text{case}}$ and $\underline{\text{rec}}$, giving us the usual notions of *constructor* case analysis *structural* recursion. But we are free to make our own eliminators, capturing more sophisticated analyses or more powerful forms of recursion. By talking about *patterns*, dependent types give us the opportunity to specify and implement new ways of programming.

2.4 Pause for Thought

Concretely, we have examined one datatype and two programs. Slightly more abstractly, we have seen the general shape of Epigram programs as **decision trees**. Each node has a left-hand side, stating a programming problem p, and a right-hand-side stating how to attack it. The leaves of the tree $p \Rightarrow t$ explain directly what value to return. The internal nodes $p \Leftarrow e$ use the type of the eliminator e as a recipe for reducing the problem statement to subproblem statements, and the value of e as a recipe for solving the whole problem, given the solutions to the subproblems. Every datatype is equipped with eliminators of form $\underline{\text{case}}\ x$ for constructor case analysis and $\underline{\text{rec}}\ x$ for structural recursion, allowing us to construct obviously total programs in a pattern matching style.

However, the $\Leftarrow$ construct is a more general tool than the `case` construct of conventional languages. We answer Wadler's question of how to combine data abstraction with notions of pattern matching [38] by making notions of pattern matching first-class values.

It's reasonable to ask 'Can't I write ordinary programs in an ordinary way? Must I build decision trees?'. I'm afraid the answer, for now, is 'yes', but it's just a matter of syntactic sugar. We're used to prioritized lists of patterns with a 'take the first match' semantics [28]. Lennart Augustsson showed us how to compile these into trees of `case`-on-variables [3]. Programming in Epigram is like being Augustsson's compiler—you choose the tree, and it shows you the patterns. The generalization lies in what may sit at the nodes. One could flatten those regions of the tree with nonempty $\Leftarrow\ \underline{\text{case}}\ x$ nodes and use Augustsson's algorithm to recover a decision tree, leaving $\Leftarrow$ explicit only at 'peculiar' nodes.

Of course, this still leaves us with explicit $\Leftarrow\ \underline{\text{rec}}\ x$ supporting structural recursion. Can we get rid of that? There are various methods of spotting safe recursive calls [17]; some even extend to tracking guardedness through mutual definitions [1]. We could use these to infer obvious appeals to rec, leaving only sophisticated recursions explicit. Again, it's a question of work. Personally, I want to write functions which are seen to be total.

Some might complain 'What's the point of a programming language that isn't Turing complete?', but I ask in return, 'Do you demand *compulsory ignorance* of totality?'. Let's guarantee totality explicitly whenever we can [37]. It's also possible, contrary to popular nonsense, to have dependent types and general recursive programs, preserving decidable typechecking: the cheapest way to do this is to work under an *assumed* eliminator with type

$$\forall P : \star \Rightarrow (P \to P) \;\to\; P$$

to which only the run time system gives a computational behaviour; a less drastic way is to treat general recursion as an impure monadic effect.

But in any case, you might be surprised how little you need general recursion. Dependent types make more programs structurally recursive, because dependent types have more structure. Inductive families with inductive indices support recursion on the data itself and recursion on the indices. For example, first-order unification [36] becomes structurally recursive when you index terms by the number of variables over which they are constructed—solving a variable may blow up the terms, but it decreases this index [25].

2.5 Some Familiar Datatypes

Just in time for the first set of exercises, let's declare some standard equipment. We shall need Bool, which can be declared like so:

$$\underline{\text{data}} \quad \mathsf{Bool} : \star \quad \underline{\text{where}} \quad \mathsf{true}, \mathsf{false} : \mathsf{Bool}$$

The standard Maybe type constructor is also useful:

$$\underline{\text{data}} \left(\frac{X : \star}{\mathsf{Maybe}\; X : \star} \right) \; \underline{\text{where}} \left(\frac{}{\mathsf{nothing} : \mathsf{Maybe}\; X} \right) ; \left(\frac{x : X}{\mathsf{just}\; x : \mathsf{Maybe}\; X} \right)$$

Note that I didn't declare X in the rules for nothing and just. The hypotheses of a rule scope only over its conclusion, so it's not coming from the Maybe rule. Rather, in each rule Epigram can tell from the way X is used that it must be a type, and it silently generalizes the constructors, just the way the Hindley-Milner system generalizes definitions.

It's the natural deduction notation which triggers this generalization. We were able to define Bool without it because there was nothing to generalize. Without the rule to 'catch' the X, plain

$$\boxed{\mathsf{nothing}} : \mathsf{Maybe}\; \boxed{X}$$

wouldn't exactly be an error. The out-of-scope X is waiting to be explained by some *prior* definition: the nothing constructor would then be specific to that X.

Rule-induced generalization is also happening here, for polymorphic lists:

$$\underline{\text{data}}\ \left(\frac{X\ :\ \star}{\mathsf{List}\ X\ :\ \star}\right)\ \ \underline{\text{where}}\ \left(\frac{}{\mathsf{nil}\ :\ \mathsf{List}\ X}\right)\ ;\ \left(\frac{x\ :\ X\ ;\ \ xs\ :\ \mathsf{List}\ X}{\mathsf{cons}\ x\ xs\ :\ \mathsf{List}\ X}\right)$$

We also need the binary trees with N-labelled nodes and L-labelled leaves.

$$\underline{\text{data}}\ \left(\frac{N, L\ :\ \star}{\mathsf{Tree}\ N\ L\ :\ \star}\right)\ \ \underline{\text{where}}\ \left(\frac{l\ :\ L}{\mathsf{leaf}\ l\ :\ \mathsf{Tree}\ N\ L}\right)\ ;\ \left(\frac{n\ :\ N\ ;\ \ s, t\ :\ \mathsf{Tree}\ N\ L}{\mathsf{node}\ n\ s\ t\ :\ \mathsf{Tree}\ N\ L}\right)$$

2.6 Exercises: Structural Merge-Sort

To get used to the system, and to programming with structural recursion, try these exercises, which only involve the simple types above

Exercise 1 (le) *Define the 'less-or-equal' test:*

$$\underline{\text{let}}\ \ \left(\frac{x, y\ :\ \mathsf{Nat}}{\mathbf{le}\ x\ y\ :\ \mathsf{Bool}}\right)$$

Does it matter which argument you do <u>rec</u> *on?*

Exercise 2 (cond) *Define the conditional expression:*

$$\underline{\text{let}}\ \ \left(\frac{b\ :\ \mathsf{Bool}\ ;\ \ then, else\ :\ T}{\mathbf{cond}\ b\ then\ else\ :\ T}\right)$$

Exercise 3 (merge) *Use the above to define the function which* **merge***s two lists, presumed already sorted into increasing order, into one sorted list containing the elements from both.*

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

Is this function structurally recursive on just one of its arguments? Nested <u>rec</u>*s combine lexicographically.*

Exercise 4 (flatten) *Use* **merge** *to implement a function* **flatten***ing a tree which may have numbers at the leaves, to produce a sorted list of those numbers. Ignore the node labels.*

$$\underline{\text{let}}\ \ \left(\frac{t\ :\ \mathsf{Tree}\ N\ (\mathsf{Maybe\ Nat})}{\mathbf{flatten}\ t\ :\ \mathsf{List\ Nat}}\right)$$

We can have a structurally recursive $\mathrm{O}(n \log n)$ sorting algorithm if we can share out the elements of a list into a *balanced* tree, then **flatten** it.

Exercise 5 (insert) *Implement the* **insert***ion of a number into a tree:*

$$\underline{\text{let}}\quad \left(\frac{n\ :\ \mathsf{Nat}\ ;\quad t\ :\ \mathsf{Tree\ Bool\ (Maybe\ Nat)}}{\mathbf{insert}\ n\ t\ :\ \mathsf{Tree\ Bool\ (Maybe\ Nat)}}\right)$$

Maintain this balancing invariant throughout: in ($\mathsf{node\ true}\ s\ t$)*,* s *and* t *contain equally many numbers, whilst in* ($\mathsf{node\ false}\ s\ t$)*,* s *contains exactly one more number than* t*.*

Exercise 6 (share, sort) *Implement*

$$\underline{\text{let}}\quad \left(\frac{ns\ :\ \mathsf{List\ Nat}}{\mathbf{share}\ ns\ :\ \mathsf{Tree\ Bool\ (Maybe\ Nat)}}\right)$$

$$\underline{\text{let}}\quad \left(\frac{ns\ :\ \mathsf{List\ Nat}}{\mathbf{sort}\ ns\ :\ \mathsf{List\ Nat}}\right)$$

so that **sort** *sorts its input in* $\mathrm{O}(n \log n)$ *time.*

3 Vectors and Finite Sets

Moving on to dependent data structures now, let's take a closer look at Vec:

$$\underline{\text{data}}\ \left(\frac{n\ :\ \mathsf{Nat}\ ;\quad X\ :\ \star}{\mathsf{Vec}\ n\ X\ :\ \star}\right)\ \underline{\text{where}}\ \left(\frac{}{\mathsf{vnil}\ :\ \mathsf{Vec\ zero}\ X}\right)$$
$$\left(\frac{x\ :\ X\ ;\quad xs\ :\ \mathsf{Vec}\ n\ X}{\mathsf{vcons}\ x\ xs\ :\ \mathsf{Vec}\ (\mathsf{suc}\ n)\ X}\right)$$

The generalization mechanism ensures that all the previously undeclared variables arising inside each deduction rule are silently quantified in the resulting type, with the *implicit* $\forall\_$ quantifier. Written out in full, we have declared

$$\begin{array}{ll}\underline{\text{data}} & \mathsf{Vec} : \mathsf{Nat} \to \star \to \star \\ \underline{\text{where}} & \mathsf{vnil} : \forall\_X : \star \Rightarrow \mathsf{Vec\ zero}\ X \\ & \mathsf{vcons} : \forall\_X : \star \Rightarrow \forall\_n : \mathsf{Nat} \Rightarrow X \to \mathsf{Vec}\ n\ X \to \mathsf{Vec}\ (\mathsf{suc}\ n)\ X\end{array}$$

On usage, Epigram tries to infer arguments for expressions with implicitly quantified types, just the way the Hindley-Milner system specializes polymorphic things—by solving the equational constraints which arise in typechecking. However, Epigram needs and supports a 'manual override': the postfix _ operator inhibits inference and makes an implicit function explicit, so

$$\begin{array}{r}\mathsf{vnil}\_ : \forall X : \star \Rightarrow \mathsf{Vec\ zero}\ X \\ \mathsf{vnil}\_\mathsf{Nat} : \mathsf{Vec\ zero\ Nat}\end{array}$$

To save space, I often write overridden arguments as subscripts—eg., $\mathsf{vnil}_{\mathsf{Nat}}$.

Given this definition, let's start to write some simple programs:

$$\underline{\mathrm{let}}\ \left(\frac{ys\ :\ \mathsf{Vec}\,(\mathsf{suc}\ m)\ Y}{\mathbf{vhead}\ ys\ :\ Y}\right)\ ;\ \ \mathbf{vhead}\ ys\ \boxed{[}\,\texttt{<= case ys}\,\boxed{]}$$

What happens when we elaborate? Well, consider which constructors can possibly have made ys. Certainly not vnil, unless $\mathsf{zero} = \mathsf{suc}\ n$. We just get a vcons case—the one case we want, for 'head' and 'tail':

$$\underline{\mathrm{let}}\ \left(\frac{ys\ :\ \mathsf{Vec}\,(\mathsf{suc}\ m)\ Y}{\mathbf{vhead}\ ys\ :\ Y}\right)\ ;\ \ \begin{array}{l}\mathbf{vhead}\ ys\ \Leftarrow\ \underline{\mathrm{case}}\ ys\ \{\\ \quad \mathbf{vhead}\ (\mathsf{vcons}\ y\ ys)\ \Rightarrow\ y\ \}\end{array}$$

$$\underline{\mathrm{let}}\ \left(\frac{ys\ :\ \mathsf{Vec}\,(\mathsf{suc}\ m)\ Y}{\mathbf{vtail}\ ys\ :\ \mathsf{Vec}\ m\ Y}\right)\ ;\ \ \begin{array}{l}\mathbf{vtail}\ ys\ \Leftarrow\ \underline{\mathrm{case}}\ ys\ \{\\ \quad \mathbf{vtail}\ (\mathsf{vcons}\ y\ ys)\ \Rightarrow\ ys\ \}\end{array}$$

In the latter, not only do we get that it's vcons as opposed to vnil: it's the particular vcons which extends vectors of the length we need. What's going on? Much as Thierry Coquand proposed in [13], Epigram is *unifying* the scrutinee of the <u>case</u> with the possible constructor patterns, in both term and type:

$ys : \mathsf{Vec}\ (\mathsf{suc}\ m)\ Y$	unifier
$\mathsf{vnil}_X : \mathsf{Vec}\ \ \mathsf{zero}\ \ X$	impossible
$\mathsf{vcons}_{X\ n}\ x\ xs' : \mathsf{Vec}\ (\mathsf{suc}\ n)\ X$	$X = Y, n = m, xs = \mathsf{vcons}_{Y\ m}\ x\ xs'$

Only the vcons case survives—Epigram then tries to choose names for the pattern variables which maintain a 'family resemblance' to the scrutinee, hence the $(\mathsf{vcons}\ y\ ys)$ in the patterns.

This unification doesn't just rule cases in or out: it can also feed information to type-level computations. Here's how to append vectors:

$$\underline{\mathrm{let}}\ \left(\frac{xs\ :\ \mathsf{Vec}\ m\ X\ ;\ \ ys\ :\ \mathsf{Vec}\ n\ X}{\mathbf{vappend}_m\ xs\ ys : \mathsf{Vec}\ (\mathbf{plus}\ m\ n)\ X}\right)$$

$$\begin{array}{l}\mathbf{vappend}_m\ xs\ ys\ \Leftarrow\ \underline{\mathrm{rec}}\ xs\ \{\\ \ \mathbf{vappend}_m\ xs\ ys\ \Leftarrow\ \underline{\mathrm{case}}\ xs\ \{\\ \ \ \mathbf{vappend}_{\mathsf{zero}}\ \mathsf{vnil}\ ys\ \Rightarrow\ ys\\ \ \ \mathbf{vappend}_{(\mathsf{suc}\ m)}\ (\mathsf{vcons}\ x\ xs)\ ys\ \Rightarrow\ \mathsf{vcons}\ x\ (\mathbf{vappend}_m\ xs\ ys)\ \}\}\end{array}$$

I've overridden the length arguments just to show what's happening—you can leave them implicit if you like. The point is that by looking at the first vector, we learn about its length. This lets **plus** compute exactly as we need for $ys\ :\ \mathsf{Vec}\ (\mathbf{plus}\ \mathsf{zero}\ n)\ X$ in the vnil case. For vcons, the return type is $\mathsf{Vec}\ (\mathbf{plus}\ (\mathsf{suc}\ m)\ n)\ X \leadsto \mathsf{Vec}\ (\mathsf{suc}\ (\mathbf{plus}\ m\ n))\ X$, which is what we supply.

3.1 Finite Sets

Let's examine the consequences of dependent case analysis for a different family:

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

What happens when we elaborate this?

$$\underline{\text{let}}\ \left(\frac{i\ :\ \mathsf{Fin\ zero}}{\mathbf{magic}\ i\ :\ X}\right)\ ;\quad \mathbf{magic}\ i\ [\texttt{<= case i}]$$

You've probably guessed, but let's just check:

$i : \mathsf{Fin}\quad \mathsf{zero}$	unifier
$\mathsf{fz}_n : \mathsf{Fin}\ (\mathsf{suc}\ n)$	impossible
$\mathsf{fs}_n\ j : \mathsf{Fin}\ (\mathsf{suc}\ n)$	impossible

So the finished product is just $\quad \mathbf{magic}\ i\ \Leftarrow\ \underline{\text{case}}\ i$

The idea is that $\mathsf{Fin}\ n$ is an enumeration type containing n values. Let's tabulate the first few members of the family, just to see what's going on. (I'll show the implicit arguments as subscripts, but write in decimal to save space.)

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

$\mathsf{Fin\ zero}$ is empty, and each $\mathsf{Fin}\ (\mathsf{suc}\ n)$ is made by embedding the n 'old' elements of $\mathsf{Fin}\ n$, using fs_n, and adding a 'new' element fz_n. $\mathsf{Fin}\ n$ provides a representation of numbers *bounded by* n, which can be used as 'array subscripts':

$$\underline{\text{let}}\ \left(\frac{xs\ :\ \mathsf{Vec}\ n\ X\ ;\ \ i\ :\ \mathsf{Fin}\ n}{\mathbf{vproj}\ xs\ i\ :\ X}\right)\ ;\ \begin{array}{l}\mathbf{vproj}\ xs\ i\ \Leftarrow\ \underline{\text{rec}}\ xs\ \{\\ \ \mathbf{vproj}\ xs\ i\ \Leftarrow\ \underline{\text{case}}\ xs\ \{\\ \ \ \mathbf{vproj}\ \mathsf{vnil}\ i\ \Leftarrow\ \underline{\text{case}}\ i\\ \ \ \mathbf{vproj}\ (\mathsf{vcons}\ x\ xs)\ i\ \Leftarrow\ \underline{\text{case}}\ i\ \{\\ \ \ \ \mathbf{vproj}\ (\mathsf{vcons}\ x\ xs)\ \mathsf{fz}\ \Rightarrow\ x\\ \ \ \ \mathbf{vproj}\ (\mathsf{vcons}\ x\ xs)\ (\mathsf{fs}\ i)\\ \ \ \ \ \Rightarrow\ \mathbf{vproj}\ xs\ i\ \}\}\}\end{array}$$

We need not fear projection from vnil, for we can dismiss $i\ :\ \mathsf{Fin\ zero}$, as a harmless fiction. Of course, we could have analysed the arguments the other way around:

$$\begin{array}{l}
\mathbf{vproj}\ xs\ i\ \Leftarrow\ \underline{\mathrm{rec}}\ xs\ \{\\
\ \ \mathbf{vproj}\ xs\ i\ \Leftarrow\ \underline{\mathrm{case}}\ i\ \{\\
\ \ \ \ \mathbf{vproj}\ xs\ \mathsf{fz}\ \Leftarrow\ \underline{\mathrm{case}}\ xs\ \{\\
\ \ \ \ \ \ \mathbf{vproj}\ (\mathsf{vcons}\ x\ xs)\ \mathsf{fz}\ \Rightarrow\ x\ \}\\
\ \ \ \ \mathbf{vproj}\ xs\ (\mathsf{fs}\ i)\ \Leftarrow\ \underline{\mathrm{case}}\ xs\ \{\\
\ \ \ \ \ \ \mathbf{vproj}\ (\mathsf{vcons}\ x\ xs)\ (\mathsf{fs}\ i)\ \Rightarrow\ \mathbf{vproj}\ xs\ i\ \}\}\}
\end{array}$$

Here, inspecting i forces n to be non-zero in each case, so xs can only be a vcons. The same result is achieved either way, but in both definitions, we rely on the impact the first case analysis has on the possibilities for the second. It may seem a tautology that dependent case analyses are not independent, but its impact is profound. We should certainly ask whether the traditional `case` expression, only expressing the patterns of its scrutinee, is as appropriate as it was in the past.

3.2 Refining Programming Problems

Our unification tables give some intuition to what is happening with case analysis. In Thierry Coquand's presentation of dependent pattern matching [13], constructor case analysis is hard-wired and unification is built into the typing rules. In Epigram, we have the more generic notion of refining a programming problem by an *eliminator*, $\Leftarrow e$. If we take a closer look at the elaboration of this construct, we'll see how unification arises and is handled *inside* the type theory. I'll maintain both the general case and the **vtail** example side by side.

As we saw with **plus**, when we say[8]

$$\underline{\mathrm{let}}\ \left(\frac{\Gamma}{\mathbf{f}\ \Gamma\ :\ R}\right)\quad\Bigg|\quad \underline{\mathrm{let}}\ \left(\frac{ys\ :\ \mathsf{Vec}\ (\mathsf{suc}\ m)\ Y}{\mathbf{vtail}\ ys\ :\ \mathsf{Vec}\ m\ Y}\right)$$

Epigram initiates the development of a proof

$$\Diamond\mathbf{f}\ :\ \forall\Gamma \Rightarrow \langle\,\mathbf{f}\ \Gamma\ :\ R\,\rangle\quad\Bigg|\quad \Diamond\mathbf{vtail}\ :\ \forall m\!:\!\mathsf{Nat};\ Y\!:\!\star;\ ys\!:\!\mathsf{Vec}\ (\mathsf{suc}\ m)\ Y \Rightarrow \langle\,\mathbf{vtail}_{m\ Y}\ ys\ :\ \mathsf{Vec}\ m\ Y\,\rangle$$

The general form of a subproblem in this development is

$$\Diamond\mathbf{fsub}\ :\ \forall\Delta \Rightarrow \langle\,\mathbf{f}\ \vec{p}\ :\ T\,\rangle\quad\Bigg|\quad \Diamond\mathbf{vtail}\ :\ \forall m\!:\!\mathsf{Nat};\ Y\!:\!\star;\ ys\!:\!\mathsf{Vec}\ (\mathsf{suc}\ m)\ Y \Rightarrow \langle\,\mathbf{vtail}_{m\ Y}\ ys\ :\ \mathsf{Vec}\ m\ Y\,\rangle$$

where $\vec{p}$ are *patterns*—expressions over the variables in Δ. In the example, I've chosen the initial patterns given by **vtails** formal parameters. Note that patterns

[8] I write Greek capitals for sequences of variables with type assignments in binders and also for their unannotated counterparts as argument sequences.

in Epigram are *not* a special subclass of expression. Now let's proceed

$$\begin{array}{l|l}
\begin{array}{l}
\mathbf{f}\;\vec{p} \Leftarrow e \\[1em]
e \;:\; \forall\;\; P\!: \forall\Theta \Rightarrow \star \\
\qquad m_1\!: \forall\Delta_1 \Rightarrow P\;\vec{s}_1 \\
\qquad\quad \vdots \\
\qquad m_n\!: \forall\Delta_n \Rightarrow P\;\vec{s}_n \\
\quad \Rightarrow P\;\vec{t}
\end{array}
&
\begin{array}{l}
\mathbf{vtail}\;\mathit{ys} \Leftarrow \underline{\mathsf{case}}\;\mathit{ys} \\[1em]
\underline{\mathsf{case}}\;\mathit{ys} \;:\; \forall\;\; P\!: \forall n; X; \mathit{xs}\!:\!\mathsf{Vec}\;n\;X \Rightarrow \star \\
\qquad m_1\!: \forall_X\!:\!\star \Rightarrow P\;\mathsf{zero}\;X\;\mathsf{vnil} \\
\qquad m_2\!: \forall_X\!:\!\star;\; _n\!:\!\mathsf{Nat} \\
\qquad\quad x\!:\!X;\; \mathit{xs}\!:\!\mathsf{Vec}\;n\;X \\
\qquad\quad \Rightarrow P\;(\mathsf{suc}\;n)\;X\;(\mathsf{vcons}\;x\;\mathit{xs}) \\
\quad \Rightarrow P\;(\mathsf{suc}\;m)\;Y\;\mathit{ys}
\end{array}
\end{array}$$

We call P the *motive*—it says what we gain from the elimination. In particular, we'll have a proof of P for the $\vec{t}$. The m_i are the *methods* by which the motive is to be achieved for each $\vec{s}_i$. James McKinna taught me to choose this motive:

$$\begin{array}{l|l}
\begin{array}{l}
\mathbf{P} \leadsto \lambda\Theta \Rightarrow \\
\qquad \forall\Delta \Rightarrow \Theta{=}\vec{t} \to \\
\qquad \langle\, \mathbf{f}\;\vec{p} \;:\; T \,\rangle
\end{array}
&
\begin{array}{l}
\mathbf{P} \leadsto \lambda n;\; X;\; \mathit{xs} \Rightarrow \\
\qquad \forall m\!:\!\mathsf{Nat};\; Y\!:\!\star;\; \mathit{ys}\!:\!\mathsf{Vec}\;(\mathsf{suc}\;m)\;Y \\
\qquad \Rightarrow n{=}(\mathsf{suc}\;m) \to X{=}Y \to \mathit{xs}{=}\mathit{ys} \to \\
\qquad \langle\, \mathbf{vtail}_{m\;Y}\;\mathit{ys} \;:\; \mathsf{Vec}\;m\;Y \,\rangle
\end{array}
\end{array}$$

This is just Henry Ford's old joke. Our motive is to produce a proof of $\forall\Delta \Rightarrow \langle\, \mathbf{f}\;\vec{p} : T \,\rangle$, for 'any Θ we like as long as it's $\vec{t}$'—the $\vec{t}$ are the only Θ we keep in stock. For our example, that means 'any vector you like as long as it's nonempty and its elements are from Y'. This $=$ is **heterogeneous equality**,[9] which allows any elements of arbitrary types to be proclaimed equal. Its one constructor, refl, says that a thing is equal to itself.

$$\frac{s \;:\; S\;;\;\; t \;:\; T}{s{=}t \;:\; \star} \qquad \frac{}{\mathsf{refl} \;:\; t{=}t}$$

Above, the types of xs and ys are different, but they will unify if we can solve the prior equations. Hypothetical equations don't change the internal rules by which the typechecker compares types—this is lucky, as hypotheses can lie.

If we can construct the methods, $\mathbf{m_i}$, then we're done:

$$\begin{array}{l|l}
\begin{array}{l}
\Diamond\mathbf{fsub} \leadsto \lambda\Delta \Rightarrow e\;\mathbf{P}\;\mathbf{m_1}\;\ldots\;\mathbf{m_n} \\
\qquad \Delta\;\mathsf{refl}\;\ldots\;\mathsf{refl} \\
\quad : \forall\Delta \Rightarrow \langle\, \mathbf{f}\;\vec{p} \;:\; T \,\rangle
\end{array}
&
\begin{array}{l}
\Diamond\mathbf{vtail} \leadsto \lambda m;\; Y;\; \mathit{ys} \Rightarrow (\underline{\mathsf{case}}\;\mathit{ys})\;\mathbf{P}\;\mathbf{m_1}\;\mathbf{m_2} \\
\qquad m\;Y\;\mathit{ys}\;\mathsf{refl}\;\mathsf{refl}\;\mathsf{refl} \\
\quad : \forall m\!:\!\mathsf{Nat};\; Y\!:\!\star;\; \mathit{ys}\!:\!\mathsf{Vec}\;(\mathsf{suc}\;m)\;Y \\
\qquad \Rightarrow \langle\, \mathbf{vtail}_{m\;Y}\;\mathit{ys} \;:\; \mathsf{Vec}\;m\;Y \,\rangle
\end{array}
\end{array}$$

But what are the methods? We must find, for each i

$$\mathbf{m_i} \;:\; \forall\Delta_i; \Delta \Rightarrow \vec{s}_i{=}\vec{t} \to \langle\, \mathbf{f}\;\vec{p} \;:\; T \,\rangle$$

[9] also known as 'John Major' equality [23]

In our example, we need

$$\begin{aligned}
\mathbf{m_1} \;:\; & \forall _X;\quad _m; _Y; ys\!:\!\mathsf{Vec}\ (\mathsf{suc}\ m)\ Y \Rightarrow \\
& \quad \mathsf{zero}{=}(\mathsf{suc}\ m) \;\to\; X{=}Y \;\to\; \mathsf{vnil}{=}ys \;\to \\
& \qquad \langle\, \mathbf{vtail}_{m\ Y}\ ys \;:\; \mathsf{Vec}\ m\ Y \,\rangle \\
\mathbf{m_2} \;:\; & \forall _X; _n; x; xs\!:\!\mathsf{Vec}\ n\ X;\quad _m; _Y; xs\!:\!\mathsf{Vec}\ (\mathsf{suc}\ m)\ Y \Rightarrow \\
& \quad (\mathsf{suc}\ n){=}(\mathsf{suc}\ m) \;\to\; X{=}Y \;\to\; (\mathsf{vcons}\ x\ xs){=}ys \;\to \\
& \qquad \langle\, \mathbf{vtail}_{m\ Y}\ ys \;:\; \mathsf{Vec}\ m\ Y \,\rangle
\end{aligned}$$

Look at the equations! They express exactly the unification problems for case analysis which we tabulated informally. Now to solve them: the rules of first-order unification for data constructors—see figure 1—are derivable in UTT. Each rule

deletion	$P \;\to$ $x{=}x \to P$	
conflict	$\mathsf{chalk}\ \vec{s}{=}\mathsf{cheese}\ \vec{t} \to P$	
injectivity	$(\vec{s}{=}\vec{t} \to P) \;\to$ $\mathsf{chalk}\ \vec{s}{=}\mathsf{chalk}\ \vec{t} \to P$	
substitution	$P\ t \;\to$ $x{=}t \;\to\; P\ x$	$x, t : T; x \notin FV(t)$
cycle	$x{=}t \to P$	x constructor-guarded in t

Fig. 1. derivable unification rule schemes

(read backwards) simplifies a problem with an equational hypothesis. We apply these simplifications to the method types. The **conflict** and **cycle** rules dispose of 'impossible case' subproblems. Meanwhile, the **substitution** rule instantiates pattern variables. In general, the equations $\vec{s}_i{=}\vec{t}$ will be reduced as far as possible by first-order unification, and either the subproblem will be dismissed, or it will yield some substitution, instantiating the patterns $\vec{p}$.

In our example, the vnil case goes by **conflict**, and the vcons case becomes:

$$\forall \_Y; \_m; x\!:\!Y; xs\!:\!\mathsf{Vec}\ Y\ m \;\Rightarrow\; \langle\, \mathbf{vtail}_{Y\ m}\ (\mathsf{vcons}\ x\ xs) \;:\; \mathsf{Vec}\ Y\ m \,\rangle$$

After 'cosmetic renaming' gives x and xs names more like the original ys, we get

$$\mathbf{vtail}\ (\mathsf{vcons}\ y\ ys)\ [\,]$$

To summarize, elaboration of $\Leftarrow e$ proceeds as follows:
(1) choose a motive with equational constraints;
(2) simplify the constraints in the methods by first-order unification;
(3) leave the residual methods as the subproblems to be solved by subprograms.

In the presence of defined functions and higher types, unification problems won't always be susceptible to first-order unification, but Epigram will make what progress it can and leave the remaining equations unsolved in the hypotheses of subproblems—later analyses may reduce them to a soluble form. Moreover, there is no reason in principle why we should not consider a constraint-solving procedure which can be customized by user-supplied rules.

3.3 Reflection on Inspection

We're not used to thinking about what functions really tell us, because simple types don't say much about values, statically. For example, we could write

$$\underline{\text{let}}\ \left(\frac{n\ :\ \mathsf{Nat}}{\mathbf{nonzero}\ n\ :\ \mathsf{Bool}}\right)\ ;\ \begin{array}{l}\mathbf{nonzero}\ n\ \Leftarrow\ \underline{\text{case}}\ n\ \{\\ \quad \mathbf{nonzero}\ \mathsf{zero}\ \Rightarrow\ \mathsf{false}\\ \quad \mathbf{nonzero}\ (\mathsf{suc}\ n)\ \Rightarrow\ \mathsf{true}\ \}\end{array}$$

but suppose we have $xs\ :\ \mathsf{Vec}\ n\ X$—what do we learn by testing $\mathbf{nonzero}\ n$? All we get is a Bool, with no direct implications for our understanding of n or xs. We are in no better position to apply $\mathbf{vtail}$ to xs after inspecting this Bool than before. Instead, if we do case analysis on n, we learn what n is *statically* as well as dynamically, and in the suc case we can apply $\mathbf{vtail}$ to xs.

Of course, we could think of writing a $\mathbf{preVtail}$ function which operates on any vector but requires a precondition, like

$$\underline{\text{let}}\ \left(\frac{xs\ :\ \mathsf{Vec}\ n\ X\ ;\ \ q\ :\ \mathbf{nonzero}\ n{=}\mathsf{true}}{\mathbf{preVtail}\ xs\ q\ :\ \mathsf{Vec}\ \boxed{[\,]}\ X}\right)$$

I'm not quite sure what to write as the length of the returned vector, so I've left a shed: perhaps it needs some kind of predecessor function with a precondition. If we have $ys\ :\ \mathsf{Vec}\,(\mathsf{suc}\ m)\ Y$, then $\mathbf{preVtail}\ ys\ \mathsf{refl}$ will be well typed. We could even use this function with a more informative conditional expression:

$$\mathbf{condInfo}\ :\ \forall P\!:\!\star;\ b\!:\!\mathsf{Bool} \Rightarrow (b{=}\mathsf{true} \to P)\ \to\ (b{=}\mathsf{false} \to P)\ \to\ P$$

However, this way of working is clearly troublesome.

Moreover, given a nonempty vector xs, there is more than just a stylistic difference between decomposing it with $\mathbf{vhead}$ and $\mathbf{vtail}$ and decomposing it with ($\underline{\text{case}}\ xs$)—the destructor functions give us an element and a shorter vector; the case analysis tells us that xs is the vcons of them, and if any types depend on xs, that might just be important. Again, we can construct a proof of $xs{=}\mathsf{vcons}\ (\mathbf{vhead}\ xs)\ (\mathbf{vtail}\ xs)$, but this is much harder to work with.

In the main, selectors-and-destructors are poor tools for working with data on which types depend. We really need forms of inspection which yield static information. This is a new issue, so there's no good reason to believe that the old design choices remain appropriate. We need to think carefully about how to reflect data's new rôle as *evidence*.

3.4 Vectorized Applicative Programming

Now that we've seen how dependent case analysis is elaborated, let's do some more work with it. The next example shows a key difference between Epigram's implicit syntax and *parametric* polymorphism. The operation

$$\underline{\text{let}}\ \left(\frac{x : X}{\mathbf{vec}\ x : \mathsf{Vec}\ n\ X}\right)$$

makes a vector of copies of its argument. For any given *usage* of **vec**, the intended type determined the length, but how are we to *define* **vec**? We shall need to work by recursion on the intended length, hence we shall need to make this explicit at definition time. The following declaration achieves this:

$$\begin{array}{l}\underline{\text{let}}\ \left(\dfrac{n : \mathsf{Nat}\ ;\ \ x : X}{\mathbf{vec}_n\ x : \mathsf{Vec}\ n\ X}\right)\ ;\ \ \mathbf{vec}_n\ x \Leftarrow \underline{\text{rec}}\ n\ \{\\ \qquad\qquad\qquad\qquad\qquad \mathbf{vec}_n\ x \Leftarrow \underline{\text{case}}\ n\ \{\\ \qquad\qquad\qquad\qquad\qquad\quad \mathbf{vec}_{\mathsf{zero}}\ x \Rightarrow \mathsf{vnil}\\ \qquad\qquad\qquad\qquad\qquad\quad \mathbf{vec}_{(\mathsf{suc}\ n)}\ x \Rightarrow \mathsf{vcons}\ x\ (\mathbf{vec}_n\ x)\ \}\}\end{array}$$

Note that in **vec**'s type signature, I explicitly declare n first, thus making it the first implicit argument: otherwise, X might happen to come first. By the way, we don't have to override the argument in the recursive call $\mathbf{vec}_n\ x$—it's got to be a $\mathsf{Vec}\ n\ X$—but it would perhaps be a little disconcerting to omit the n, especially as it's the key to **vec**'s structural recursion.

The following operation—vectorized application—turns out to be quite handy.

$$\begin{array}{l}\underline{\text{let}}\ \left(\dfrac{fs : \mathsf{Vec}\ n\ (S \to T)\ ;\ \ ss : \mathsf{Vec}\ n\ S}{\mathbf{va}\ fs\ ss : \mathsf{Vec}\ n\ T}\right)\\ \quad \mathbf{va}\ fs\ ss \Leftarrow \underline{\text{rec}}\ fs\ \{\\ \quad\ \ \mathbf{va}\ fs\ ss \Leftarrow \underline{\text{case}}\ fs\ \{\\ \quad\ \ \ \ \mathbf{va}\ \mathsf{vnil}\ ss \Leftarrow \underline{\text{case}}\ ss\ \{\\ \quad\ \ \ \ \ \ \mathbf{va}\ \mathsf{vnil}\ \mathsf{vnil} \Rightarrow \mathsf{vnil}\ \}\\ \quad\ \ \ \ \mathbf{va}\ (\mathsf{vcons}\ f\ fs)\ ss \Leftarrow \underline{\text{case}}\ ss\ \{\\ \quad\ \ \ \ \ \ \mathbf{va}\ (\mathsf{vcons}\ f\ fs)\ (\mathsf{vcons}\ s\ ss) \Rightarrow \mathsf{vcons}\ (f\ s)\ (\mathbf{va}\ fs\ ss)\ \}\}\}\end{array}$$

As it happens, the combination of **vec** and **va** equip us with 'vectorized applicative programming', with **vec** embedding the constants, and **va** providing application. Transposition is my favourite example of this:

$$\begin{array}{l}\underline{\text{let}}\ \left(\dfrac{xij : \mathsf{Vec}\ i\ (\mathsf{Vec}\ j\ X)}{\mathbf{transpose}\ xij : \mathsf{Vec}\ j\ (\mathsf{Vec}\ i\ X)}\right)\\ \quad \mathbf{transpose}\ xij \Leftarrow \underline{\text{rec}}\ xij\ \{\\ \quad\ \ \mathbf{transpose}\ xij \Leftarrow \underline{\text{case}}\ xij\ \{\\ \quad\ \ \ \ \mathbf{transpose}\ \mathsf{vnil} \Rightarrow \mathbf{vec}\ \mathsf{vnil}\\ \quad\ \ \ \ \mathbf{transpose}\ (\mathsf{vcons}\ xj\ xij) \Rightarrow \mathbf{va}\ (\mathbf{va}\ (\mathbf{vec}\ \mathsf{vcons})\ xj)\ (\mathbf{transpose}\ xij)\ \}\}\end{array}$$

3.5 Exercises: Matrix Manipulation

Exercise 7 (vmap, vZipWith) *Show how* **vec** *and* **va** *can be used to generate the vector analogues of Haskell's* `map`, `zipWith`, *and the rest of the family. (A glance at [16] may help.)*

Exercise 8 (vdot) *Implement* **vdot**, *the scalar product of two vectors of* Nat*s.*

Now, how about matrices? Recall the vector-of-columns representation:

$$\underline{\text{let}}\ \left(\frac{rows, cols\ :\ \mathsf{Nat}}{\mathbf{Matrix}\ rows\ cols\ :\ \star}\right)\quad \mathbf{Matrix}\ rows\ cols\ \Rightarrow\ \mathsf{Vec}\ cols\ (\mathsf{Vec}\ rows\ \mathsf{Nat})$$

Exercise 9 (zero, identity) *How would you compute the zero matrix of a given size? Also implement a function to compute any identity matrix.*

Exercise 10 (matrix by vector) *Implement matrix-times-vector multiplication. (ie, interpret a* **Matrix** $m\ n$ *as a linear map* $\mathsf{Vec}\ n\ \mathsf{Nat} \to \mathsf{Vec}\ m\ \mathsf{Nat}$*.)*

Exercise 11 (matrix by matrix) *Implement matrix-times-matrix multiplication. (ie, implement composition of linear maps.)*

Exercise 12 (monad) *(Mainly for Haskellers.) It turns out that for each* n, $\mathsf{Vec}\ n$ *is a monad, with* **vec** *playing the part of* `return`. *What should the corresponding notion of* `join` *do? What plays the part of* `ap`*?*

3.6 Exercises: Finite Sets

Exercise 13 (fmax, fweak) *Implement* **fmax** *(each nonempty set's maximum value) and* **fweak** *(the function preserving* fz *and* fs, *incrementing the index).*

$$\underline{\text{let}}\ \left(\frac{}{\mathbf{fmax}_n\ :\ \mathsf{Fin}\ (\mathsf{suc}\ n)}\right)\quad\Bigg|\quad \underline{\text{let}}\ \left(\frac{i\ :\ \mathsf{Fin}\ n}{\mathbf{fweak}\ i\ :\ \mathsf{Fin}\ (\mathsf{suc}\ n)}\right)$$

You should find that **fmax** and **fweak** partition the finite sets, just as fz and fs do. Imagine how we might pretend they're an alternative set of constructors...

Exercise 14 (vtab) *Implement* **vtab**, *the inverse of* **vproj**, *tabulating a function over finite sets as a vector.*

$$\underline{\text{let}}\ \left(\frac{n\ :\ \mathsf{Nat}\ ;\ \ f\ :\ \mathsf{Fin}\ n\ \to\ X}{\mathbf{vtab}_n\ f\ :\ \mathsf{Vec}\ n\ X}\right)$$

Note that **vtab** and **vproj** offer alternative definitions of matrix operations.

Exercise 15 (OPF, opf) *Devise an inductive family,* $\mathsf{OPF}\ m\ n$ *which gives a unique first-order representation of exactly the* order-preserving *functions in* $\mathsf{Fin}\ m \to \mathsf{Fin}\ n$*. Give your family a semantics by implementing*

$$\underline{\text{let}}\ \left(\frac{f\ :\ \mathsf{OPF}\ m\ n\ ;\ \ i\ :\ \mathsf{Fin}\ m}{\mathbf{opf}\ f\ i\ :\ \mathsf{Fin}\ n}\right)$$

Exercise 16 ($\mathbf{iOPF}, \mathbf{cOPF}$) *Implement identity and composition:*

$$\underline{\text{let}}\ \left(\frac{}{\mathbf{iOPF}_n\ :\ \mathsf{OPF}\ n\ n}\right) \quad\Bigg|\quad \underline{\text{let}}\ \left(\frac{f\ \mathsf{OPF}\ m\ n\ ;\ \ g\ \mathsf{OPF}\ l\ m}{\mathbf{cOPF}\ f\ g\ :\ \mathsf{OPF}\ l\ n}\right)$$

Which laws should relate **iOPF**, **cOPF** and **opf**?

4 Representing Syntax

The Fin family can represent de Bruijn indices in nameless expressions [14]. As Françoise Bellegarde and James Hook observed in [6], and Richard Bird and Ross Paterson were able to implement in [9], you can do this in Haskell, up to a point—here are the λ-terms with *free* variables given by `v`:

```
data Term v = Var v
            | App (Term v) (Term v)
            | Lda (Term (Maybe v))
```

Under a `Lda`, we use `(Maybe v)` as the variable set for the body, with `Nothing` being the new free variable and `Just` embedding the old free variables. Renaming is just `fmap`, and substitution is just the monadic 'bind' operator `>>=`.

However, `Term` is a bit *too* polymorphic. We can't see the *finiteness* of the variable context over which a term is constructed. In Epigram, we can take the number of free variables to be a number n, and choose variables from $\mathsf{Fin}\ n$.

$$\underline{\text{data}}\ \left(\frac{n\ :\ \mathsf{Nat}}{\mathsf{Tm}\ n\ :\ \star}\right)$$
$$\underline{\text{where}}\ \left(\frac{i\ :\ \mathsf{Fin}\ n}{\mathsf{var}\ i\ :\ \mathsf{Tm}\ n}\right)\ ;\ \left(\frac{f, s\ :\ \mathsf{Tm}\ n}{\mathsf{app}\ f\ s\ :\ \mathsf{Tm}\ n}\right)\ ;\ \left(\frac{t\ :\ \mathsf{Tm}\ (\mathsf{suc}\ n)}{\mathsf{lda}\ t\ :\ \mathsf{Tm}\ n}\right)$$

The n in $\mathsf{Term}\ n$ indicates the number of variables available for term formation: we can explain how to λ-lift a term, by abstracting over all the available variables:

$$\underline{\text{let}}\ \left(\frac{t\ :\ \mathsf{Tm}\ n}{\mathbf{ldaLift}_n\ t\ :\ \mathsf{Tm}\ \mathsf{zero}}\right)\ ;\ \begin{array}{l} \mathbf{ldaLift}_n\ t\ \Leftarrow\ \underline{\text{rec}}\ n\ \{ \\ \quad \mathbf{ldaLift}_n\ t\ \Leftarrow\ \underline{\text{case}}\ n\ \{ \\ \quad\quad \mathbf{ldaLift}_{\mathsf{zero}}\ t\ \Rightarrow\ t \\ \quad\quad \mathbf{ldaLift}_{(\mathsf{suc}\ n)}\ t\ \Rightarrow\ \mathbf{ldaLift}_n\ (\mathsf{lda}\ t)\ \}\} \end{array}$$

Not so long ago, we were quite excited about the power of non-uniform datatypes to capture useful structural invariants. Scoped de Bruijn terms gave a good example, but most of the others proved more awkward even than the 'fake' dependent types you can cook up using type classes [24].

Real dependent types achieve more with less fuss. This is mainly due to the flexibility of inductive families. For example, if you wanted to add 'weakening' to delay explicitly the shifting of a term as you push it under a binder—in Epigram, but not Haskell or Cayenne, you could add the constructor

$$\left(\frac{t\ :\ \mathsf{Tm}\ n}{\mathsf{weak}\ t\ :\ \mathsf{Tm}\ (\mathsf{suc}\ n)}\right)$$

4.1 Exercises: Renaming and Substitution

If $\mathsf{Fin}\ m$ is a variable set, then some $\rho : \mathsf{Fin}\ m \to \mathsf{Fin}\ n$ is a *renaming*. If we want to apply a renaming to a term, we need to be able to push it under a lda. Hence we need to *weaken* the renaming, mapping the new source variable to the new target variable, and renaming as before on the old variables.[10]

$$\underline{\text{let}}\ \left(\frac{\rho : \mathsf{Fin}\ m \to \mathsf{Fin}\ n\ ;\quad i : \mathsf{Fin}\ (\mathsf{suc}\ m)}{\mathbf{wren}\ \rho\ i : \mathsf{Fin}\ (\mathsf{suc}\ n)}\right) \quad \begin{array}{l} \mathbf{wren}\ \rho\ i \ \Leftarrow\ \underline{\text{case}}\ i\ \{ \\ \quad \mathbf{wren}\ \rho\ \mathsf{fz} \ \Rightarrow\ \mathsf{fz} \\ \quad \mathbf{wren}\ \rho\ (\mathsf{fs}\ i) \ \Rightarrow\ \mathsf{fs}\ (\rho\ i)\ \} \end{array}$$

You get to finish the development.

Exercise 17 (ren) *Use* **wren** *to help you implement the renaming traversal*

$$\underline{\text{let}}\ \left(\frac{\rho : \mathsf{Fin}\ m \to \mathsf{Fin}\ n\ ;\quad t : \mathsf{Tm}\ m}{\mathbf{ren}\ \rho\ t : \mathsf{Tm}\ n}\right)$$

Now repeat the pattern for substitutions—functions from variables to terms.

Exercise 18 (wsub, sub) *Develop weakening for substitutions, then use it to go under* lda *in the traversal:*

$$\underline{\text{let}}\ \left(\frac{\sigma : \mathsf{Fin}\ m \to \mathsf{Tm}\ n\ ;\quad i : \mathsf{Fin}\ (\mathsf{suc}\ m)}{\mathbf{wsub}\ \sigma\ i : \mathsf{Tm}\ (\mathsf{suc}\ n)}\right)$$

$$\underline{\text{let}}\ \left(\frac{\sigma : \mathsf{Fin}\ m \to \mathsf{Tm}\ n\ ;\quad t : \mathsf{Tm}\ m}{\mathbf{sub}\ \sigma\ t : \mathsf{Tm}\ n}\right)$$

Exercise 19 *(For the brave.) Refactor this development, abstracting the weakening-then-traversal pattern. If you need a hint, see chapter 7 of [23].*

4.2 Stop the World I Want to Get Off! (a first try at typed syntax)

We've seen untyped λ-calculus: let's look at how to enforce stronger invariants, by representing a *typed* λ-calculus. Recall the rules of the simply typed λ-calculus:

$$\frac{}{\Gamma; x \in \sigma; \Gamma' \vdash x \in \sigma} \qquad \frac{\Gamma; x \in \sigma \vdash t \in \tau}{\Gamma \vdash \lambda x \in \sigma.\ t \in \sigma \supset \tau} \qquad \frac{\Gamma \vdash f \in \sigma \supset \tau \quad \Gamma \vdash s \in \sigma}{\Gamma \vdash f\ s \in \tau}$$

Well-typed terms are defined with respect to a context and a type. Let's just turn the rules into data! I add a base type, to make things more concrete.

$$\underline{\text{data}}\ \left(\frac{}{\mathsf{SType} : \star}\right)\ \underline{\text{where}}\ \left(\frac{}{\mathsf{sNat} : \mathsf{SType}}\right)\ ;\ \left(\frac{\sigma, \tau : \mathsf{SType}}{\mathsf{sFun}\ \sigma\ \tau : \mathsf{SType}}\right)$$

[10] Categorists! Note **wren** makes suc a functor in the category of Fin-functions. What other structure can you sniff out here?

We could use Vec for contexts, but I prefer contexts which grow on the right.

$$\underline{\text{data}}\ \left(\frac{n : \mathsf{Nat}}{\mathsf{SCtxt}\ n : \star}\right)\ \underline{\text{where}}\ \left(\frac{}{\mathsf{empty} : \mathsf{SCtxt\ zero}}\right)\ \left(\frac{\Gamma : \mathsf{SCtxt}\ n;\ \ \sigma : \mathsf{SType}}{\mathsf{bind}\ \Gamma\ \sigma : \mathsf{SCtxt}\ (\mathsf{suc}\ n)}\right)$$

Now, assuming we have a projection function **sproj**, defined in terms of SCtxt and Fin the way we defined **vproj**, we can just turn the inference rules of the typing relation into constructors:

$$\underline{\text{data}}\ \left(\frac{\Gamma : \mathsf{SCtxt}\ n;\ \ \tau : \mathsf{SType}}{\mathsf{STm}\ \Gamma\ \tau : \star}\right)$$
$$\underline{\text{where}}\ \left(\frac{i : \mathsf{Fin}\ n}{\mathsf{svar}\ i : \mathsf{STm}\ \Gamma\ (\mathbf{sproj}\ \Gamma\ i)}\right)\ ;\ \left(\frac{t : \mathsf{STm}\ (\mathsf{bind}\ \Gamma\ \sigma)\ \tau}{\mathsf{slda}\ t : \mathsf{STm}\ \Gamma\ (\mathsf{sFun}\ \sigma\ \tau)}\right)$$
$$\left(\frac{f : \mathsf{STm}\ \Gamma\ (\mathsf{sFun}\ \sigma\ \tau);\ \ s : \mathsf{STm}\ \Gamma\ \sigma}{\mathsf{sapp}\ f\ s : \mathsf{STm}\ \Gamma\ \tau}\right)$$

This is a precise definition of the simply-typed λ-terms. But is it any good? Well, just try writing programs with it.

How would you implement *renaming*? As before, we could represent a renaming as a function $\rho : \mathsf{Fin}\ m \to \mathsf{Fin}\ n$. Can we rename a term in $\mathsf{STm}\ \Gamma\ \tau$ to get a $\mathsf{STm}\ \Delta\ \tau$, where $\Gamma : \mathsf{SCtxt}\ m$ and $\Delta : \mathsf{SCtxt}\ n$? Here comes the crunch:

$$\cdots\ \ \mathbf{ren}_{n\ \Gamma\ \Delta}\ \rho\ (\mathsf{svar}\ i)\ \Rightarrow\ \boxed{\mathsf{svar}\ (\rho\ i)}$$

The problem is that $\mathsf{svar}\ i : \mathsf{STm}\ \Gamma\ (\mathbf{sproj}\ \Gamma\ i)$, so we want a $\mathsf{STm}\ \Delta\ (\mathbf{sproj}\ \Gamma\ i)$ on the right, but we've got a $\mathsf{STm}\ \Delta\ (\mathbf{sproj}\ \Delta\ (\rho\ i))$. We need to know that ρ is type-preserving! Our choice of variable representation prevents us from building this into the type of ρ. We are forced to state an extra condition:

$$\forall i : \mathsf{Fin}\ m \Rightarrow \mathbf{sproj}\ \Gamma\ i = \mathbf{sproj}\ \Delta\ (\rho\ i)$$

We'll need to repair our program by rewriting with this proof.[11] But it's worse than that! When we move under a slda, we'll lift the renaming, so we'll need a *different* property:

$$\forall i' : \mathsf{Fin}\ (\mathsf{suc}\ m) \Rightarrow \mathbf{sproj}\ (\mathsf{bind}\ \Gamma\ \sigma)\ i' = \mathbf{sproj}\ (\mathsf{bind}\ \Delta\ \sigma)\ (\mathbf{lift}\ \rho\ i')$$

This follows from the previous property, but it takes a little effort. My program has just filled up with ghastly theorem-proving. Don't dependent types make life a nightmare? Stop the world I want to get off!

If you're not afraid of hard work, you can carry on and make this program work. I think discretion is the better part of valour—let's solve the problem instead. We're working with *typed* terms but *untyped* variables, and our function which gives types to variables does not connect the variable clearly to the context. For all we know, **sproj** always returns sNat! No wonder we need 'logical superstructure' to recover the information we've thrown away.

[11] I shan't show how to do this, as we shall shortly avoid the problem.

4.3 Dependent Types to the Rescue

Instead of using a program to assign types to variables and then reasoning about it, let's just have *typed* variables, as Thorsten Altenkirch and Bernhard Reus [2].

$$\underline{\mathrm{data}}\ \left(\frac{\Gamma : \mathsf{SCtxt}\ n\ ;\ \ \tau : \mathsf{SType}}{\mathsf{SVar}\ \Gamma\ \tau\ :\ \star}\right)$$
$$\underline{\mathrm{where}}\ \left(\frac{}{\mathsf{vz}\ :\ \mathsf{SVar}\ (\mathsf{bind}\ \Gamma\ \sigma)\ \sigma}\right)\ ;\ \left(\frac{i\ :\ \mathsf{SVar}\ \Gamma\ \tau}{\mathsf{vs}\ i\ :\ \mathsf{SVar}\ (\mathsf{bind}\ \Gamma\ \sigma)\ \tau}\right)$$

This family strongly resembles Fin. Its constructors target only nonempty contexts; it has one constructor which references the 'newest' variable; the other constructor embeds the 'older' variables. You may also recognize this family as an inductive definition of context *membership*. Being a variable *means* being a member of the context. Fin just gives a data representation for variables without their meaning. Now we can replace our awkward svar with

$$\left(\frac{i\ :\ \mathsf{SVar}\ \Gamma\ \tau}{\mathsf{svar}\ i\ \ :\ \mathsf{STm}\ \Gamma\ \tau}\right)$$

A renaming *from* Γ *to* Δ becomes an element of

$$\forall \tau : \mathsf{SType} \Rightarrow \mathsf{SVar}\ \Gamma\ \tau \to \mathsf{SVar}\ \Delta\ \tau$$

Bad design makes for hard work, whether you're making can-openers, doing mathematics or writing programs. It's often tempting to imagine that once we've made our representation of data tight enough to rule out meaningless values, our job is done and things should just work out. This experience teaches us that more is required—we should use types to give meaningful values their meaning. Fin contains the right data, but SVar actually explains it.

Exercise 20 *Construct simultaneous renaming and simultaneous substitution for this revised definition of* STm*. Just lift the pattern from the untyped version!*

4.4 Is Looking Seeing?

It's one thing to define data structures which *enforce* invariants and to write programs which *respect* invariants, but how can we *establish* invariants?

We've seen how to use a finite set to index a vector, enforcing the appropriate bounds, but what if we only have a *number*, sent to us from the outside world? We've seen how to write down the STms, but what if we've read in a program from a file? How do we compute its type-safe representation if it has one?

If we want to index a $\mathsf{Vec}\ n\ X$ by $m\ :\ \mathsf{Nat}$, it's no good testing the Boolean $m < n$. The value true or false won't explain whether m can be represented by some $i : \mathsf{Fin}\ n$. If we have a $f : \mathsf{STm}\ \Gamma\ (\mathsf{sFun}\ \sigma\ \tau)$ and some $a : \mathsf{STm}\ \Gamma\ \alpha$, we could check Boolean equality $\sigma == \alpha$, but true doesn't make α into σ, so we can't construct $\mathsf{sapp}\ f\ a$.

Similar issues show up in the 'Scrap Your Boilerplate' library of dynamically typed traversal operators by Ralf Lämmel and Simon Peyton Jones [18]. The whole thing rests on a 'type safe cast' operator, comparing types at run time:

```
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
```

This program does not, of itself, make sense. The best we can say is that *we* can make sense of it, provided `typeOf` has been correctly implemented. The machine *looks* at the types but does not *see* when they are the same, hence the `unsafeCoerce`. The significance of the test is obscure, so blind obedience is necessary. Of course, *I* trust them, but I think they could aspire for better.

The trouble is that representing the result of a computation is not enough: you need to know the *meaning* of the computation if you want to justify its consequences. A Boolean is a bit uninformative. To see when we look, we need a new way of looking. Take the vector indexing example. We can explain which number is represented by a given $i : \mathsf{Fin}\ n$ by *forgetting* its bound:

$$\underline{\text{let}}\ \left(\frac{i\ :\ \mathsf{Fin}\ n}{\mathbf{fFin}\ i\ :\ \mathsf{Nat}}\right)\ ;\ \begin{array}{l}\mathbf{fFin}\ i\ \Leftarrow\ \underline{\text{rec}}\ i\ \{\\ \quad \mathbf{fFin}\ i\ \Leftarrow\ \underline{\text{case}}\ i\ \{\\ \quad\quad \mathbf{fFin}\ \mathsf{fz}\ \Rightarrow\ \mathsf{zero}\\ \quad\quad \mathbf{fFin}\ (\mathsf{fs}\ i)\ \Rightarrow\ \mathsf{suc}\ (\mathbf{fFin}\ i)\ \}\}\end{array}$$

Now, for a given n and m, m is either

- $(\mathbf{fFin}\ i)$ for some $i : \mathsf{Fin}\ n$, or
- $(\mathbf{plus}\ n\ m')$ for some $m' : \mathsf{Nat}$

Our types can talk about values—we can *say* that!

$$\begin{array}{ll}\mathbf{checkBound}\ n\ m\ :\ & \forall P\ :\ \mathsf{Nat} \to \mathsf{Nat} \to \star\ \ \Rightarrow\\ & (\forall n{:}\mathsf{Nat};\ i{:}\mathsf{Fin}\ n \Rightarrow P\ n\ (\mathbf{fFin}\ i))\ \to\\ & (\forall n, m'{:}\mathsf{Nat} \Rightarrow P\ n\ (\mathbf{plus}\ n\ m'))\ \to\\ & P\ n\ m\end{array}$$

That's to say: 'whatever P you want to do with n and m, it's enough to explain P for n and $(\mathbf{fFin}\ i)$ and also for n and $(\mathbf{plus}\ n\ m')$'. Or 'you can match n, m against the *patterns*, $n, (\mathbf{fFin}\ i)$ and $n, (\mathbf{plus}\ n\ m')$'. I designed the above type to look like a <u>case</u> principle, so that I can program with it. Note that I don't just get an element either of $(\mathsf{Fin}\ i)$ or of Nat from an anonymous informant; it really is my very own n and m which get analysed—the type says so! If I have **checkBound**, then I can check m like this:

$$\underline{\text{let}}\ \left(\frac{xs\ :\ \mathsf{Vec}\ n\ X\ ;\quad m\ :\ \mathsf{Nat}}{\mathbf{mayProj}_n\ xs\ m\ :\ \mathsf{Maybe}\ X}\right)$$

$$\begin{array}{l}\mathbf{mayProj}_n\ xs\ m\ \Leftarrow\ \mathbf{checkBound}\ n\ m\ \{\\ \quad \mathbf{mayProj}_n\ xs\ (\mathbf{fFin}\ i)\ \Rightarrow\ \mathsf{just}\ (\mathbf{vproj}\ xs\ i)\\ \quad \mathbf{mayProj}_n\ xs\ (\mathbf{plus}\ n\ m')\ \Rightarrow\ \mathsf{nothing}\ \}\end{array}$$

In one case, we get a bounded i, so we can apply bounds-safe projection. In the other, we clearly fail. Moreover, if the return type were to depend on m, that's fine: not only do *we* see what m must be, Epigram sees it too! But **checkBound** has quite a complicated higher-order type. Do I really expect you to dump good old $m < n$ for some bizarre functional? Of course I don't: I'll now explain the straightforward first-order way to construct **checkBound**.

4.5 A Funny Way of Seeing Things

Constructor case analysis is the normal way of seeing things. Suppose I have a funny way of seeing things. We know that $\Leftarrow$ doesn't care—a 'way of seeing things' is expressed by a type and interpreted as a way of decomposing a programming problem into zero or more subproblems. But how do I establish that my funny way of seeing at things makes sense?

Given $n, m : \mathsf{Nat}$, we want to see m as either $(\mathbf{fFin}\ i)$ for some $i : \mathsf{Fin}\ n$, or else some $(\mathbf{plus}\ n\ m')$. We can write a predicate which characterizes the n and m for which this is possible—it's possible for the very patterns we want.[12]

$$\begin{array}{ll}\underline{\text{data}} & \left(\dfrac{n, m\ :\ \mathsf{Nat}}{\mathsf{BoundCheck}\ n\ m\ :\ \star}\right)\\ \underline{\text{where}} & \left(\dfrac{i\ :\ \mathsf{Fin}\ n}{\mathsf{inBound}\ i\ :\ \mathsf{BoundCheck}\ n\ (\mathbf{fFin}\ i)}\right)\\ & \left(\dfrac{m'\ :\ \mathsf{Nat}}{\mathsf{outOfBound}\ m'\ :\ \mathsf{BoundCheck}\ n\ (\mathbf{plus}\ n\ m')}\right)\end{array}$$

A value $bc : \mathsf{BoundCheck}\ n\ m$ tells us something about n and m, and it's just n and m that we care about here—bc is just a means to this end. The eliminator ($\underline{\text{case}}\ bc$) expects a motive abstracting over n, m and bc, allowing us to inspect bc also. If we restrict the motive to see only n and m, we get

$$\begin{array}{l}\lambda P : \mathsf{Nat} \to \mathsf{Nat} \to \star\ \ \Rightarrow\ \ (\underline{\text{case}}\ bc)\ (\lambda\ n'; m'; bc' \Rightarrow P\ n'\ m')\\ :\ \forall P\ :\ \mathsf{Nat} \to \mathsf{Nat} \to \star\ \ \Rightarrow\\ \quad (\forall n : \mathsf{Nat};\ i : \mathsf{Fin}\ n \Rightarrow\ P\ n\ (\mathbf{fFin}\ i))\ \to\\ \quad (\forall n, m' : \mathsf{Nat} \Rightarrow\ P\ n\ (\mathbf{plus}\ n\ m'))\ \to\\ \quad P\ n\ m\end{array}$$

[12] I forgot that I'm *programming*: of course, I mean 'datatype family'.

and that's exactly the type of $\textbf{checkBound}\, n\, m$. This construction on a predicate is sufficiently useful that Epigram gives it a special name, ($\underline{\text{view}}\ bc$). That's to say, the machine-generated eliminator which just looks at $\mathsf{BoundCheck}$'s *indices* in terms of its constructors. Logically, $\underline{\text{view}}$ gives a datatype family its **relation induction** principle. But to use this 'view', we need $bc : \mathsf{BoundCheck}\, n\, m$. That is, we must show that every n and m are checkable in this way:

$$
\begin{array}{l}
\underline{\text{let}}\ \left(\dfrac{}{\textbf{boundCheck}\ n\ m\ :\ \mathsf{BoundCheck}\ n\ m}\right) \\
\quad \textbf{boundCheck}\ n\ m\ \Leftarrow\ \underline{\text{rec}}\ n\ \{ \\
\quad\ \textbf{boundCheck}\ n\ m\ \Leftarrow\ \underline{\text{case}}\ n\ \{ \\
\quad\quad \textbf{boundCheck}\ \mathsf{zero}\ m\ \Rightarrow\ \mathsf{outOfBound}\ m \\
\quad\quad \textbf{boundCheck}\ (\mathsf{suc}\ n)\ m\ \Leftarrow\ \underline{\text{case}}\ m\ \{ \\
\quad\quad\ \textbf{boundCheck}\ (\mathsf{suc}\ n)\ \mathsf{zero}\ \Rightarrow\ \mathsf{inBound}\ \mathsf{fz} \\
\quad\quad\ \textbf{boundCheck}\ (\mathsf{suc}\ n)\ (\mathsf{suc}\ m)\ \Leftarrow\ \underline{\text{view}}\ (\textbf{boundCheck}\ n\ m)\ \{ \\
\quad\quad\quad \textbf{boundCheck}\ (\mathsf{suc}\ n)\ (\mathsf{suc}\ (\textbf{fFin}\ i))\ \Rightarrow\ \mathsf{inBound}\ (\mathsf{fs}\ i) \\
\quad\quad\quad \textbf{boundCheck}\ (\mathsf{suc}\ n)\ (\mathsf{suc}\ (\textbf{plus}\ n\ m'))\ \Rightarrow\ \mathsf{outOfBound}\ m'\ \}\}\}\}
\end{array}
$$

There's no trouble using the view we're trying to establish: the recursive call is structural, but used in an eliminator rather than a return value. This function works much the way *subtraction* works. The only difference is that it has a type which establishes a connection between the output to the function and its inputs, shown directly in the patterns! We may now take

$$\textbf{checkBound}\ n\ m\ \leadsto\ \underline{\text{view}}\ (\textbf{boundCheck}\ n\ m)$$

4.6 Patterns Forget; Matching Is Remembering

What has 'pattern matching' become? In general, a pattern is a *forgetful* operation. Constructors like zero and suc forget *themselves*—you can't tell from the type Nat, which constructor you've got. Case analysis remembers what constructors forget. And so it is with our funny patterns: the **fFin** function forgets bounded whilst ($\textbf{plus}\, n\, m'$) forgets by how much its output exceeds n. Our view remembers what these patterns forget.

The difference between Epigram views and Phil Wadler's views [38] is that Epigram views cannot lie. Epigram views talk directly about the values being inspected in terms of the forgetful operations which generate them. Wadler's views ascribe that informative significance to an independent value, whether or not it's justified. We shouldn't criticize Wadler for this—dependent types can see where simple types can only look. Of course, to work with dependent types, we *need* to be able to see. If we want to generate values in types which enforce strong invariants, we need to see that those invariants hold.

Exercise 21 *Show that* **fmax** *and* **fweak** *cover* Fin *by constructing a view.*

5 Well Typed Programs which Don't Go Wrong

Let's have a larger example of derivable pattern matching—building simply-typed terms in the STm family by typechecking 'raw' untyped terms from

$$\underline{\text{data}}\ \left(\frac{n\ :\ \mathsf{Nat}}{\mathsf{RTm}\ n\ :\ \star}\right)$$
$$\underline{\text{where}}\ \left(\frac{i\ :\ \mathsf{Fin}\ n}{\mathsf{rvar}\ i\ :\ \mathsf{RTm}\ n}\right)\ ;\ \left(\frac{f, s\ :\ \mathsf{RTm}\ n}{\mathsf{rapp}\ f\ s\ :\ \mathsf{RTm}\ n}\right)$$
$$\left(\frac{\sigma\ :\ \mathsf{SType}\ ;\quad b\ :\ \mathsf{RTm}\ (\mathsf{suc}\ n)}{\mathsf{rlda}\ \sigma\ b\ :\ \mathsf{RTm}\ n}\right)$$

Typechecking is a form of looking. It relies on two auxiliary forms of looking—looking up a variable in the context, and checking that two types are the same. Our svar constructor takes context-references expressed in terms of SVar, and our sapp constructor really needs the domain of the function to be the same as the type of the argument, so just looking is not enough. Let's see.

An SVar is a context-reference; a Fin is merely a context-pointer. We can clearly turn a reference into a pointer by forgetting what's referred to:

$$\underline{\text{let}}\ \left(\frac{\Gamma\ :\ \mathsf{SCtxt}\ n\ ;\quad i\ :\ \mathsf{SVar}\ \Gamma\ \tau}{\mathbf{fV}\ \tau\ i\ :\ \mathsf{Fin}\ n}\right)\ ;\ \begin{array}{l}\mathbf{fV}\ \tau\ i\ \Leftarrow\ \underline{\text{rec}}\ i\ \{\\ \quad \mathbf{fV}\ \tau\ i\ \Leftarrow\ \underline{\text{case}}\ i\ \{\\ \quad\quad \mathbf{fV}\ \tau\ \mathsf{vz}\ \Rightarrow\ \mathsf{fz}\\ \quad\quad \mathbf{fV}\ \tau\ (\mathsf{vs}\ i)\ \Rightarrow\ \mathsf{fs}\ (\mathbf{fV}\ \tau\ i)\ \}\}\end{array}$$

Why is τ an explicit argument? Well, the point of writing this forgetful map is to define a notion of *pattern* for finite sets which characterizes projection. We need to see the information which the pattern throws away. Let's establish the view—it's just a more informative **vproj**, telling us not only the projected thing, but that it is indeed the projection we wanted.

$$\underline{\text{data}}\ \left(\frac{\Gamma\ :\ \mathsf{SCtxt}\ n\ ;\quad i\ :\ \mathsf{Fin}\ n}{\mathsf{Find}\ \Gamma\ i\ :\ \star}\right)\ \underline{\text{where}}\ \left(\frac{i\ :\ \mathsf{SVar}\ \Gamma\ \tau}{\mathsf{found}\ \tau\ i\ :\ \mathsf{Find}\ \Gamma\ (\mathbf{fV}\ \tau\ i)}\right)$$

$$\begin{array}{l}\underline{\text{let}}\ \left(\frac{}{\mathbf{find}\ \Gamma\ i\ :\ \mathsf{Find}\ \Gamma\ i}\right)\\ \quad \mathbf{find}\ \Gamma\ i\ \Leftarrow\ \underline{\text{rec}}\ \Gamma\ \{\\ \quad\quad \mathbf{find}\ \Gamma\ i\ \Leftarrow\ \underline{\text{case}}\ i\ \{\\ \quad\quad\quad \mathbf{find}\ \Gamma\ \mathsf{fz}\ \Leftarrow\ \underline{\text{case}}\ \Gamma\ \{\\ \quad\quad\quad\quad \mathbf{find}\ (\mathsf{bind}\ \Gamma\ \sigma)\ \mathsf{fz}\ \Rightarrow\ \mathsf{found}\ \sigma\ \mathsf{vz}\ \}\\ \quad\quad\quad \mathbf{find}\ \Gamma\ (\mathsf{fs}\ i)\ \Leftarrow\ \underline{\text{case}}\ \Gamma\ \{\\ \quad\quad\quad\quad \mathbf{find}\ (\mathsf{bind}\ \Gamma\ \sigma)\ (\mathsf{fs}\ i)\ \Leftarrow\ \underline{\text{view}}\ (\mathbf{find}\ \Gamma\ i)\ \{\\ \quad\quad\quad\quad\quad \mathbf{find}\ (\mathsf{bind}\ \Gamma\ \sigma)\ (\mathsf{fs}\ (\mathbf{fV}\ \tau\ i))\ \Rightarrow\ \mathsf{found}\ \tau\ (\mathsf{vs}\ i)\ \}\}\}\}\end{array}$$

5.1 Term and Terror

We can follow the same recipe for typechecking as we did for context lookup. Help me fill in the details:

Exercise 22 (fTm) *Implement the forgetful map:*

$$\underline{\text{let}}\ \left(\frac{\Gamma\ \mathsf{SCtxt}\ n\ ;\quad t\ :\ \mathsf{STm}\ \Gamma\ \tau}{\mathbf{fTm}\ \tau\ t\ :\ \mathsf{RTm}\ n}\right)$$

But not every raw term is the forgetful image of a well typed term. We'll need

$$\underline{\text{data}}\ \left(\frac{\Gamma\ :\ \mathsf{SCtxt}\ n}{\mathsf{TError}\ \Gamma\ :\ \star}\right)\ \underline{\text{where}}\ \cdots$$

$$\underline{\text{let}}\ \left(\frac{\Gamma\ :\ \mathsf{SCtxt}\ n\ ;\quad e\ :\ \mathsf{TError}\ \Gamma}{\mathbf{fTError}\ e\ :\ \mathsf{RTm}\ n}\right)$$

Exercise 24 (TError, fTError) *Fill out the definition of* TError *and implement* **fTError**. *(This will be easy, once you've done the missing exercise. The* TErrors *will jump out as we write the typechecker—they pack up the failure cases.)*

Let's start on the typechecking view. First, the checkability relation:

$$\begin{array}{l}\underline{\text{data}}\ \left(\dfrac{\Gamma\ :\ \mathsf{SCtxt}\ n ; r\ :\ \mathsf{RTm}\ n}{\mathsf{Check}\ \Gamma\ r\ :\ \star}\right)\\ \underline{\text{where}}\ \left(\dfrac{t\ :\ \mathsf{STm}\ \Gamma\ \tau}{\mathsf{good}\ t\ :\ \mathsf{Check}\ \Gamma\ (\mathbf{fTm}\ \tau\ t)}\right)\ ;\ \left(\dfrac{e\ :\ \mathsf{TError}\ \Gamma}{\mathsf{bad}\ e\ :\ \mathsf{Check}\ \Gamma\ (\mathbf{fTError}\ e)}\right)\end{array}$$

Next, let's start on the proof of checkability—sorry, the typechecker:

$$\underline{\text{let}}\ \left(\frac{}{\mathbf{check}\ \Gamma\ r\ :\ \mathsf{Check}\ \Gamma\ r}\right)$$

$$\begin{array}{l}
\mathbf{check}\ \Gamma\ r\ \Leftarrow\ \underline{\text{rec}}\ r\ \{\\
\ \mathbf{check}\ \Gamma\ r\ \Leftarrow\ \underline{\text{case}}\ r\ \{\\
\ \ \mathbf{check}\ \Gamma\ (\mathsf{rvar}\ i)\ \Leftarrow\ \underline{\text{view}}\ (\mathbf{find}\ \Gamma\ i)\ \{\\
\ \ \ \mathbf{check}\ \Gamma\ (\mathsf{rvar}\ (\mathbf{fV}\ \tau\ i))\ \Rightarrow\ \mathsf{good}\ (\mathsf{svar}\ i)\ \}\\
\ \ \mathbf{check}\ \Gamma\ (\mathsf{rapp}\ f\ s)\ \Leftarrow\ \underline{\text{view}}\ (\mathbf{check}\ \Gamma\ f)\ \{\\
\ \ \ \mathbf{check}\ \Gamma\ (\mathsf{rapp}\ (\mathbf{fTm}\ \phi\ f)\ s)\ \Leftarrow\ \underline{\text{case}}\ \phi\ \{\\
\ \ \ \ \mathbf{check}\ \Gamma\ (\mathsf{rapp}\ (\mathbf{fTm}\ \mathsf{sNat}\ f)\ s)\ \Rightarrow\ \mathsf{bad}\ \boxed{[]}\\
\ \ \ \ \mathbf{check}\ \Gamma\ (\mathsf{rapp}\ (\mathbf{fTm}\ (\mathsf{sFun}\ \sigma\ \tau)\ f)\ s)\ \Leftarrow\ \underline{\text{view}}\ (\mathbf{check}\ \Gamma\ s)\ \{\\
\ \ \ \ \ \mathbf{check}\ \Gamma\ (\mathsf{rapp}\ (\mathbf{fTm}\ (\mathsf{sFun}\ \sigma\ \tau)\ f)\ (\mathbf{fTm}\ \alpha\ s))\ \ \boxed{[]}\\
\ \ \ \ \ \mathbf{check}\ \Gamma\ (\mathsf{rapp}\ (\mathbf{fTm}\ (\mathsf{sFun}\ \sigma\ \tau)\ f)\ (\mathbf{fTError}\ e))\ \Rightarrow\ \mathsf{bad}\ \boxed{[]}\ \}\}\\
\ \ \ \mathbf{check}\ \Gamma\ (\mathsf{rapp}\ (\mathbf{fTError}\ e)\ s)\ \Rightarrow\ \mathsf{bad}\ \boxed{[]}\ \}\\
\ \ \mathbf{check}\ \Gamma\ (\mathsf{rlda}\ \sigma\ t)\ \Leftarrow\ \underline{\text{view}}\ (\mathbf{check}\ (\mathsf{bind}\ \Gamma\ \sigma)\ t)\ \{\\
\ \ \ \mathbf{check}\ \Gamma\ (\mathsf{rlda}\ \sigma\ (\mathbf{fTm}\ \tau\ t))\ \Rightarrow\ \mathsf{good}\ (\mathsf{slda}\ t)\\
\ \ \ \mathbf{check}\ \Gamma\ (\mathsf{rlda}\ \sigma\ (\mathbf{fTError}\ e))\ \Rightarrow\ \mathsf{bad}\ \boxed{[]}\ \}\}\}
\end{array}$$

The story so far: we used **find** to check variables; we used **check** recursively to check the body of an rlda and packed up the successful outcome. Note that we don't need to write the types of the good terms—they're implicit in STm.

We also got some way with application: checking the function; checking that the function inhabits a function space; checking the argument. The only trouble is that our function expects a σ and we've got an α. We need to *see* if they're the same: that's the missing exercise.

Exercise 23 *Develop an equality* view *for* SType*:*

$$\underline{\text{data}}\ \left(\frac{\sigma, \tau\ :\ \mathsf{SType}}{\mathsf{Compare}\,\sigma\,\tau\ :\ \star}\right)$$
$$\underline{\text{where}}\ \left(\frac{}{\mathsf{same}\ :\ \mathsf{Compare}\,\tau\,\tau}\right)\ ;\ \left(\frac{\sigma'\ :\ \mathsf{Diff}\,\sigma}{\mathsf{diff}\,\sigma'\ :\ \mathsf{Compare}\,\sigma\,(\mathbf{fDiff}\,\sigma\,\sigma')}\right)$$

$$\underline{\text{let}}\ \left(\frac{}{\mathbf{compare}\,\sigma\,\tau\ :\ \mathsf{Compare}\,\sigma\,\tau\ :\ \star}\right)$$

You'll need to define a representation of STypes *which differ from a given* σ *and a forgetful map* **fDiff** *which forgets this difference.*

How to go about it? Wait and see. Let's go back to application...

$$\begin{array}{l}
\mathbf{check}\ \Gamma\ (\mathsf{rapp}\ (\mathbf{fTm}\ (\mathsf{sFun}\ \sigma\ \tau)\ f)\ (\mathbf{fTm}\ \alpha\ s))\ \Leftarrow\ \underline{\text{view}}\ (\mathbf{compare}\ \sigma\ \alpha)\ \{\\
\quad \mathbf{check}\ \Gamma\ (\mathsf{rapp}\ (\mathbf{fTm}\ (\mathsf{sFun}\ \sigma\ \tau)\ f)\ (\mathbf{fTm}\ \sigma\ s))\ \Rightarrow\ \mathsf{good}\ (\mathsf{sapp}\ f\ s)\\
\quad \mathbf{check}\ \Gamma\ (\mathsf{rapp}\ (\mathbf{fTm}\ (\mathsf{sFun}\ \sigma\ \tau)\ f)\ (\mathbf{fTm}\ (\mathbf{fDiff}\ \sigma\ \sigma')\ s))\ \Rightarrow\ \mathsf{bad}\ \boxed{[\,]}\ \}
\end{array}$$

If we use your **compare** view, we can see directly that the types match in one case and mismatch in the other. For the former, we can now return a well typed application. The latter is definitely wrong.

We've done all the good cases, and we're left with choosing inhabitants of TError Γ for the bad cases. There's no reason why you shouldn't define TError Γ to make this as easy as possible. Just pack up the information which is lying around! For the case we've just seen, you could have:[13]

$$\left(\frac{\sigma'\ :\ \mathsf{Diff}\,\sigma\ ;\ \ f\ :\ \mathsf{STm}\,\Gamma\,(\mathsf{sFun}\,\sigma\,\tau)\ ;\ \ s\ :\ \mathsf{STm}\,\Gamma\,(\mathbf{fDiff}\,\sigma\,\sigma')}{\mathsf{mismatchError}\,\sigma'\,f\,s\ :\ \mathsf{TError}\,\Gamma}\right)$$

$$\mathbf{fTError}\ (\mathsf{mismatchError}\ \sigma'\ f\ s)\ \Rightarrow\ \mathsf{rapp}\ (\mathbf{fTm}\ ?\ f)\ (\mathbf{fTm}\ ?\ s)$$

This recipe gives one constructor for each bad case, and you don't have any choice about its declaration. There are two basic type errors—the above mismatch and the application of a non-function. The remaining three bad cases just propagate failure outwards: you get a type of located errors.

[13] The ? means 'please infer'—it's often useful when writing forgetful maps. Why?

Of course, you'll need to develop **comparable** first. To define Diff, just play the same type-of-diagnostics game. Develop the equality test, much as you would with the Boolean version, but using the view recursively in order to see when the sources and targets of two sFuns are the same. If you need a hint, see [27].

What have we achieved? We've written a typechecker which not only returns *some* well typed term or error message, but, specifically, *the* well typed term or error message which corresponds to its input by **fTm** or **fTError**. That correspondance is directly expressed by a very high level derived form of pattern matching: not rvar, rapp or rlda, but 'well typed' or 'ill typed'.

5.2 A Typesafe and Total Interpreter

Once you have a well typed term, you can extract some operational benefit from its well-typedness—you can execute it without run-time checks. This example was inspired by Lennart Augustsson and Magnus Carlsson's interpreter for terms with a typing proof [5]. Epigram's inductive families allow us a more direct approach: we just write down a denotational semantics for well typed terms. Firstly, we must interpret SType:

$$\underline{\text{let}}\ \left(\frac{\tau\ :\ \mathsf{SType}}{\mathbf{Value}\,\tau\ :\ \star}\right)\ ;\ \begin{array}{l}\mathbf{Value}\,\tau\ \Leftarrow\ \underline{\text{rec}}\,\tau\ \{\\ \quad\mathbf{Value}\,\tau\ \Leftarrow\ \underline{\text{case}}\,\tau\ \{\\ \quad\quad\mathbf{Value}\ \mathsf{sNat}\ \Rightarrow\ \mathsf{Nat}\\ \quad\quad\mathbf{Value}\ (\mathsf{sFun}\ \sigma\ \tau)\ \Rightarrow\ \mathbf{Value}\,\sigma \to \mathbf{Value}\,\tau\ \}\}\end{array}$$

Now we can explain how to interpret a context by an environment of values:

$$\begin{array}{ll}\underline{\text{data}} & \left(\dfrac{\Gamma\ :\ \mathsf{SCtxt}\ n}{\mathsf{Env}\ \Gamma\ :\ \star}\right)\\ \underline{\text{where}} & \left(\dfrac{}{\mathsf{eempty}\ :\ \mathsf{Env\ empty}}\right)\ ;\ \left(\dfrac{\gamma\ :\ \mathsf{Env}\ \Gamma\ ;\ \ v\ :\ \mathbf{Value}\,\sigma}{\mathsf{ebind}\ \gamma\ \sigma\ :\ \mathsf{Env}\ (\mathsf{bind}\ \Gamma\ \sigma)}\right)\end{array}$$

Next, interpret variables by looking them up:

$$\underline{\text{let}}\ \left(\frac{\gamma\ :\ \mathsf{Env}\ \Gamma\ ;\ \ i\ :\ \mathsf{SVar}\ \Gamma\ \tau}{\mathbf{evar}\ \gamma\ i\ :\ \mathbf{Value}\,\tau}\right)\ ;\ \begin{array}{l}\mathbf{evar}\ \gamma\ i\ \Leftarrow\ \underline{\text{rec}}\ i\ \{\\ \quad\mathbf{evar}\ \gamma\ i\ \Leftarrow\ \underline{\text{case}}\ i\ \{\\ \quad\quad\mathbf{evar}\ \gamma\ \mathsf{vz}\ \Leftarrow\ \underline{\text{case}}\ \gamma\ \{\\ \quad\quad\quad\mathbf{evar}\ (\mathsf{ebind}\ \gamma\ v)\ \mathsf{vz}\ \Rightarrow\ v\ \}\\ \quad\quad\mathbf{evar}\ \gamma\ (\mathsf{vs}\ i)\ \Leftarrow\ \underline{\text{case}}\ \gamma\ \{\\ \quad\quad\quad\mathbf{evar}\ (\mathsf{ebind}\ \gamma\ v)\ (\mathsf{vs}\ i)\ \Rightarrow\ \mathbf{evar}\ \gamma\ i\ \}\}\}\end{array}$$

Finally, interpret the well typed terms:

$$\underline{\text{let}}\ \left(\frac{\gamma : \mathsf{Env}\,\Gamma\ ;\ \ t : \mathsf{STm}\,\Gamma\,\tau}{\mathbf{eval}\,\gamma\,t\ :\ \mathbf{Value}\,\tau}\right)$$

$$\begin{array}{l}
\mathbf{eval}\,\gamma\,t\ \Leftarrow\ \underline{\text{rec}}\ t\ \{\\
\ \ \mathbf{eval}\,\gamma\,t\ \Leftarrow\ \underline{\text{case}}\ t\ \{\\
\ \ \ \ \mathbf{eval}\,\gamma\,(\mathsf{svar}\ i)\ \Rightarrow\ \mathbf{evar}\,\gamma\,i\\
\ \ \ \ \mathbf{eval}\,\gamma\,(\mathsf{sapp}\,f\ s)\ \Rightarrow\ \mathbf{eval}\,\gamma\,f\,(\mathbf{eval}\,\gamma\,s)\\
\ \ \ \ \mathbf{eval}\,\gamma\,(\mathsf{slda}\ t)\ \Rightarrow\ \lambda v \Rightarrow \mathbf{eval}\,(\mathsf{ebind}\,\gamma\,v)\ t\ \}\}
\end{array}$$

Exercise 25 *Make an environment whose entries are the constructors for* Nat*, together with some kind of iterator. Add two and two.*

6 Epilogue

Well, we've learned to add two and two. It's true that Epigram is currently little more than a toy, but must it necessarily remain so? There is much work to do.

I hope I have shown that precise data structures can manipulated successfully and in a highly articulate manner. You don't have to be content with giving orders to the computer and keeping your ideas to yourself. What has become practical is a notion of program as *effective explanation*, rather than merely an effective procedure. Upon what does this practicality depend?

– *adapting the programming language to suit dependent types*
  Our conventional programming constructs are not well-suited either to cope with or to capitalize on the richness of dependent data structures. We have had to face up to the fact that inspecting one value can tell us more about types and about other values. And so it should: at long last, testing makes a difference! Moreover, the ability of types to talk about values gives us ready access to a new, more articulate way of programming with the high-level structure of values expressed directly as patterns.
– *using type information earlier in the programming process*
  With so much structure—and computation—at the type level, keeping yourself type correct is inevitably more difficult. But it isn't necessary! Machines can check types and run programs, so use them! Interactive programming shortens the feedback loop, and it makes types a positive input to the programming process, not just a means to police its output.
– *changing the programs we choose to write*
  We shouldn't expect dependently typed programming merely to extend the functional canon with new programs which could not be typed before. In order to exploit the power of dependent types to express and enforce stronger invariants, we need a new style of programming which explicitly *establishes* those invariants. We need to rework old programs, replacing uninformative types with informative ones.

6.1 Related Work

Epigram's elder siblings are DML [39] and Cayenne [4]. DML equips ML programs with types refined by linear integer constraints and equips the typechecker with a constraint-solver. Correspondingly, many basic invariants, especially those involving sizes and ranges, can be statically enforced—this significantly reduces the overhead of run time checking [40]. Epigram has no specialist constraint-solver for arithmetic, although such a thing is a possible and useful extension. Epigram's strength is in the diversity of its type-level language.

Cayenne is much more ambitious than DML and a lot closer to Epigram. It's notorious for its looping typechecker, although (contrary to popular misconception) this is not an inevitable consequence of mixing dependent types with general recursion—recursion is implemented via fixpoints, so even structurally recursive programs can loop—you can always expand a fixpoint.

Cayenne's main drawback is that it doesn't support the kind of inductive families which Epigram inherited from the Alf system [21, 13]. It rules out those in which constructors only target *parts* of a family, the way vnil makes *empty* vectors and vcons makes *nonempty* vectors. This also rules out SVar, STm and all of our 'views'. All of these examples can be given a cumbersome encoding if you are willing to work hard enough: I for one am not.

The Agda proof assistant [12], like Epigram, is very much in the spirit of Alf, but it currently imposes the same restrictions on inductive definitions as Cayenne and hence would struggle to support the programs in these notes—this unfortunate situation is unlikely to continue. Meanwhile Coq [11] certainly accepts the inductive definitions in this paper: it just has no practical support for programming with them—there is no good reason for this to remain so.

In fact, the closest programming language to Epigram at time of writing is Haskell, with `ghc`'s new 'generalised algebraic data types' [33]. These turn out to be, more or less, inductive families! Of course, in order to preserve the rigid separation of static types and dynamic terms, GADTs must be indexed by *type* expressions. It becomes quite easy to express examples like the type-safe interpreter, which exploit the invariants enforced by indexing. What is still far from obvious is how to *establish* invariants for run time data, as we did in our typechecker—this requires precisely the transfer of information from the dynamic to the static which is still excluded.

6.2 What is to be done?

We have only the very basic apparatus of dependently typed programming in place at the moment. We certainly need some way to analyse the results of *intermediate* computations in a way which reflects their significance for the existing type and value information—I have studiously avoided this issue in these notes. In [27], we propose a construct which adds the result of an intermediate computation to the collection of values being scrutinized on the left-hand side, at the same time abstracting it from types. This is not yet implemented.

We shall certainly need coinductive data in order to develop interactive systems. Inspired by the success of monads in Haskell, we shall also need to investigate the enhanced potential for 'categorical packaging' of programming in a language where the notion of category can be made abstract. And of course, there are all the 'modern conveniences': infix operators, ad-hoc polymorphism, generics, and so forth. These require design effort: the underlying expressivity is available, but we need good choices for their high-level presentation.

Work has already begun on a compiler for Epigram [10]: we have barely started to exploit our new wealth of static information for performance. We have the benefit of a large total fragment, in which evaluation strategy is unimportant and program transformation is no longer troubled by $\perp$. The fact that partial evaluation is already a fact of life for us must surely help also.

We need a library, but it's not enough to import the standard presentation of standard functionality. Our library must support the idioms of dependently typed programming, which may well be different. Standardizing too early might be a mistake: we need to explore the design space for standard equipment.

But the greatest potential for change is in the tools of program development. Here, we have barely started. Refinement-style editing is great when you have a plan, but often we don't. We need to develop *refactoring* technology for Epigram, so that we can sharpen our definitions as we learn from experiments. It's seldom straight away that we happen upon exactly the indexed data structure we need.

Moreover, we need editing facilities that reflect the idioms of programming. Many data structures have a rationale behind them—they are intended to relate to other data structures in particular ways and support particular operations. At the moment we write none of this down. The well typed terms are supposed to be a more carefully indexed version of the raw terms—we should have been able to construct them explicitly as such. If only we could express our design principles then we could follow them deliberately. Currently, we engineer coincidences, dreaming up datatypes and operations as if from thin air.

But isn't this just wishful thinking? I claim not. Dependent types, seen through the Curry-Howard lens, can characterize types and programs in a way which editing technology can exploit. We've already seen one class of logical principle reified as a programming operation—the $\Leftarrow$ construct. We've been applying reasoning to the construction of programs on paper for years. We now have what we need to do the same effectively on a computer: a high-level programming language in which reasons and programs not merely coexist but coincide.

References

1. Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 2000.
2. Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda-terms using generalized inductive types. In *Computer Science Logic 1999*, 1999.
3. 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.

4. Lennart Augustsson. Cayenne—a language with dependent types. In *ACM International Conference on Functional Programming '98*. ACM, 1998.
5. Lennart Augustsson and Magnus Carlsson. An exercise in dependent types: A well-typed interpreter. Available at `http://www.cs.chalmers.se/~augustss/cayenne/interp.ps`, 1999.
6. Françoise Bellegarde and James Hook. Substitution: A formal methods case study using monads and transformations. *Science of Computer Programming*, 1995.
7. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.
8. Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.
9. Richard Bird and Ross Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–92, 1999.
10. Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.
11. L'Équipe Coq. The Coq Proof Assistant Reference Manual. `http://pauillac.inria.fr/coq/doc/main.html`, 2001.
12. Catarina Coquand and Thierry Coquand. Structured Type Theory. In *Workshop on Logical Frameworks and Metalanguages*, 1999.
13. Thierry Coquand. Pattern Matching with Dependent Types. In Bengt Nordström, Kent Petersson, and Gordon Plotkin, editors, *Electronic Proceedings of the Third Annual BRA Workshop on Logical Frameworks (Båstad, Sweden)*, 1992.
14. Nicolas G. de Bruijn. Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation. *Indagationes Mathematicæ*, 34:381–392, 1972.
15. Peter Dybjer. Inductive Sets and Families in Martin-Löf's Type Theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*. CUP, 1991.
16. Daniel Fridlender and Mia Indrika. Do we need dependent types? *Journal of Functional Programming*, 10(4):409–415, 2000.
17. Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs, '94*, volume 996 of *LNCS*, pages 39–59. Springer-Verlag, 1994.
18. 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).
19. Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, 1994.
20. 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.
21. 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.
22. Per Martin-Löf. A theory of types. manuscript, 1971.
23. 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/`.
24. Conor McBride. Faking It (Simulating Dependent Types in Haskell). *Journal of Functional Programming*, 12(4& 5):375–392, 2002. Special Issue on Haskell.

25. Conor McBride. First-Order Unification by Structural Recursion. *Journal of Functional Programming*, 13(6), 2003.
26. Conor McBride. Epigram, 2004. `http://www.dur.ac.uk/CARG/epigram`.
27. Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.
28. Fred McBride. *Computer Aided Manipulation of Symbols*. PhD thesis, Queen's University of Belfast, 1970.
29. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML, revised edition*. MIT Press, 1997.
30. Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
31. Chris Okasaki. From Fast Exponentiation to Square Matrices: An Adventure in Types. In *ACM International Conference on Functional Programming '99*, 1999.
32. Simon Peyton Jones and John Hughes, editors. *Haskell'98: A Non-Strict Functional Language*, 1999. Available from `http://www.haskell.org/definition`.
33. Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. Unpublished, 2004.
34. Dag Prawitz. *Natural Deduction—A proof theoretical study*. Almquist and Wiksell, Stockholm, 1965.
35. Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's type theory: an introduction*. Oxford University Press, 1990.
36. Alan Robinson. A Machine-oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12:23–41, 1965.
37. David Turner. Elementary Strong Functional Programming. In *Functional Programming Languages in Education, First International Symposium*, volume 1022 of *LNCS*. Springer-Verlag, 1995.
38. Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of POPL '87*. ACM, 1987.
39. Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 1998.
40. Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*. ACM Press, 1998.