Lists in TeX's Mouth

Alan Jeffrey

1 Why lists?

Originally, I wanted lists in TeX for a paper I was writing which contained a lot of facts.

Fact i *Cows have four legs.*

Fact ii *People have two legs.*

Fact iii *Lots of facts in a row can be dull.*

These are generated with commands like

```
\begin{fact}
\Forward{Fac-yawn}
   Lots of facts in a row can be dull.
\end{fact}
```

I can then refer to these facts by saying

```
\By[Fac-yawn,Fac-cows,Fac-people]
```

to get [i, ii, iii]. And as if by magic, the facts come out sorted, rather than in the jumbled order I typed them. This is very useful, as I can reorganize my document to my heart's content, and not have to worry about getting my facts straight.

Originally I tried programming this sorting routine in TeX's list macros, from Appendix D of *The TeXbook*, but I soon ran into trouble. The problem is that all the Appendix D macros work by assigning values to macros. For example:

```
\concatenate\foo=\bar\baz
```

expands out to

```
\ta=\expandafter{\bar}
\tb=\expandafter{\baz}
\edef\foo{\the\ta\the\tb}
```

which assigns the macro `\foo` the contents of `\bar` followed by the contents of `\baz`. Programming sorting routines (which are usually recursive) in terms of these lists became rather painful, as I was constantly having to watch out for local variables, worrying about what happened if a local variable had the same name as a global one, and generally having a hard time.

Then I had one of those "flash of light" experiences — "You can do lambda-calculus in TeX," I thought, and since you can do lists directly in lambda calculus, you should be able to do lists straightforwardly in TeX. And so you can. Well, fairly straightforwardly anyway.

So I went and did a bit of mathematics, and derived the TeX macros you see here. They were formally verified, and worked first time (modulo typing errors, of which there were two).

2 TeX's mouth and TeX's stomach

TeX's programming facilities come in two forms — there are TeX's *macros* which are expanded in its mouth, and some additional *assignment* operations like `\def` which take place in the stomach. TeX can often spring surprises on you as exactly what gets evaluated where. For example, in LaTeX I can put down a label by saying `\label{Here}`. Then I can refer back to that label by saying `Section~\ref{Here}`, which produces Section 2. Unfortunately, `\ref{Here}` does *not* expand out to 2! Instead, it expands out to:

```
\edef\@tempa{\@nameuse{r@Here}}
\expandafter\@car\@tempa\@nil\null
```

This means that I can't say

```
\ifnum\ref{Here}<4 Hello\fi
```

and hope that this will expand out to Hello. Instead I get an error message. Which is rather a pity, as TeX's mouth is quite a powerful programming language (as powerful as a Turing Machine in fact).

3 Functions

A *function* is a mathematical object that takes in an argument (which could well be another function) and returns some other mathematical object. For example the function *Not* takes in a boolean and returns its complement. I'll write function application without brackets, so $\mathit{Not}\, b$ is the boolean complement of b.

Function application binds to the left, so $f\ a\ b$ is $(f\ a)\ b$ rather than $f\ (a\ b)$. For example, $\mathit{Or}\ a\ b$ is the boolean or of a and b, and *Or True* is a perfectly good function that takes in a boolean and returns *True*.

The obvious equivalents of functions in TeX are macros — if I define a function *Foo* to be:

$$\mathit{Foo}\, x \quad = \quad \mathit{True}$$

then it can be translated into TeX as:

```
\def\Foo#1{\True}
```

So where *Foo* is a function that takes in one argument, `\Foo` is a macro that takes in one parameter. Nothing has changed except the jargon and the font. TeX macros can even be partially applied, for example if we defined:

$$\mathit{Baz} \quad = \quad \mathit{Or}\ \mathit{True}$$

then the TeX equivalent would be

```
\def\Baz{\Or\True}
```

Once `\Baz` is expanded, it will expect to be given a parameter, but when we are defining things, we can go around partially applying them all we like.

Here, I'm using $=$ without formally defining it, which is rather naughty. If I say $x = y$, this means "given enough parameters, x and y will eventually expand out to the same thing." For example $\mathit{Foo} = \mathit{Baz}$, because for any x,

$$\begin{aligned} &\mathit{Foo}\,x \\ &\quad = \mathit{True} \\ &\quad = \mathit{Or}\ \mathit{True}\,x \\ &\quad = \mathit{Baz}\,x \end{aligned}$$

Normally, functions have to respect equality which means that:

- if $x = y$ then $f\,x = f\,y$, and
- if x respects equality, then $f\,x$ respects equality.

However, some TeX control sequences don't obey this. For example, `\string\Foo` and `\string\Baz` are different, even though $\mathit{Foo} = \mathit{Baz}$. Hence *string* doesn't respect equality. Unless otherwise stated, we won't assume functions respect equality, although all the functions defined here do.

All of our functions have capital letters, so that their TeX equivalents (`\Not`, `\Or` and so on) don't clash with standard TeX or LaTeX macros.

3.1 Identity

The simplest function is the *identity* function, called *Identity* funnily enough, which is defined:

$$\mathit{Identity}\,x \quad = \quad x$$

This, it must be admitted, is a pretty dull function, but it's a useful basic combinator. It can be implemented in TeX quite simply.

```
\def\Identity#1{#1}
```

The rules around this definition mean that it is actually part of `Lambda.sty` and not just another example.

3.2 Error

Whereas *Identity* does nothing in a fairly pleasant sort of way, *Error* does nothing in a particularly brutal and harsh fashion. Mathematically, *Error* is the function that destroys everything else in front of it. It is often written as $\perp$.

$$\mathit{Error}\,x \quad = \quad \mathit{Error}$$

In practice, destroying the entire document when we hit one error is a bit much, so we'll just print out an error message. The user can carry on past an error at their own risk, as the code will no longer be formally verified.

```
\def\Error
   {\errmessage{Abandon verification all
                  ye who enter here}}
```

Maybe this function ought to return a more useful error message ...

3.3 First and Second

Two other basic functions are *First* and *Second*, both of which take in two arguments, and do the obvious thing. They are defined:

$$\begin{aligned} \mathit{First}\,x\,y &= x \\ \mathit{Second}\,x\,y &= y \end{aligned}$$

We could, in fact, define *Second* in terms of *Identity* and *First*. For any x and y,

$$\begin{aligned} &\mathit{First}\ \mathit{Identity}\,x\,y \\ &\quad = \mathit{Identity}\,y \\ &\quad = y \\ &\quad = \mathit{Second}\,x\,y \end{aligned}$$

So $\mathit{First}\ \mathit{Identity} = \mathit{Second}$. This means that anywhere in our TeX code we have `\First\Identity` we could replace it by `\Second`. This is perhaps not the most astonishing TeX fact known to humanity, but this sort of proof did enable more complex bits of TeX to be verified before they were run.

The TeX definitions of `\First` and `\Second` are pretty obvious.

```
\def\First#1#2{#1}
\def\Second#1#2{#2}
```

Note that in TeX `\First\foo\bar` expands out to `\foo` *without* expanding out `\bar`. This is very useful, as we can write macros that would take forever and a day to run if they expanded all their arguments, but which actually terminate quite quickly. This is called *lazy evaluation* by the functional programming community.

3.4 Compose

Given two functions f and g we would like to be able to *compose* them to produce a function that first applies g then applies f. Normally, this is written as $f \circ g$, but unfortunately TeX doesn't have infix functions, so we'll have to write it $\mathit{Compose}\,f\,g$.

$$\mathit{Compose}\,f\,g\,x \quad = \quad f\,(g\,x)$$

¿From this definition, we can deduce that *Compose* is associative:

$$\begin{aligned} &\mathit{Compose}\,(\mathit{Compose}\,f\,g)\,h \\ &\quad = \mathit{Compose}\,f\,(\mathit{Compose}\,g\,h) \end{aligned}$$

and *Identity* is the left unit of *Compose*:

$$\mathit{Compose}\ \mathit{Identity}\,f \quad = \quad f$$

The reader may wonder why *Identity* is called a *left* unit even though it occurs on the right of the *Compose* — this is a side-effect of using prefix notations where infix is more normal. The infix version of this equation is:

$$\mathit{Identity} \circ f \quad = \quad f$$

so *Identity* is indeed on the left of the composition.

Compose can be implemented in TeX as

```
\def\Compose#1#2#3{#1{#2{#3}}}
```

3.5 Twiddle

Yet another useful little function is *Twiddle*, which takes in a function and reverses the order that function takes its (first two) arguments.

$$\mathit{Twiddle}\; f\; x\; y \quad = \quad f\; y\; x$$

Again, there aren't many immediate uses for such a function, but it'll come in handy later on. It satisfies the properties

$$\begin{aligned} \mathit{Twiddle}\ \mathit{First} &= \mathit{Second} \\ \mathit{Twiddle}\ \mathit{Second} &= \mathit{First} \\ \mathit{Compose}\ \mathit{Twiddle}\ \mathit{Twiddle} &= \mathit{Identity} \end{aligned}$$

Its TeX equivalent is

```
\def\Twiddle#1#2#3{#1{#3}{#2}}
```

This function is called "twiddle" because it is sometimes written $\widetilde{f}$ (and $\sim$ is pronounced "twiddle"). It also twiddles its arguments around, which is quite nice if your sense of humour runs to appalling puns.

4 Booleans

As we're trying to program a sorting routine, it would be nice to be able to define orderings on things, and to do this we need some representation of boolean variables. Unfortunately TeX doesn't have a type for booleans, so we'll have to invent our own. We'll implement a boolean as a function b of the form

$$b\; x\; y \quad = \quad \begin{cases} x & \text{if } b \text{ is true} \\ y & \text{otherwise} \end{cases}$$

More formally, a boolean b is a function which respects equality, such that for all f, g and z:

$$b\; f\; g\; z \quad = \quad b\; (f\; z)\; (g\; z)$$

and for all f and g which respect equality,

$$b\; (f\; b)\; (g\; b) \quad = \quad b\; (f\; \mathit{First})\; (g\; \mathit{Second})$$

All the functions in this section satisfy these properties. Surprisingly enough, so does *Error*, which is quite useful, as it allows us to reason about booleans which "go wrong".

4.1 True, False and Not

Since we are implementing booleans as functions, we already have the definitions of *True*, *False* and *Not*.

$$\begin{aligned} \mathit{True} &= \mathit{First} \\ \mathit{False} &= \mathit{Second} \\ \mathit{Not} &= \mathit{Twiddle} \end{aligned}$$

So for free we get the following results:

$$\begin{aligned} \mathit{Not}\ \mathit{True} &= \mathit{False} \\ \mathit{Not}\ \mathit{False} &= \mathit{True} \\ \mathit{Compose}\ \mathit{Not}\ \mathit{Not} &= \mathit{Identity} \end{aligned}$$

The TeX implementation is not exactly difficult:

```
\let\True=\First
\let\False=\Second
\let\Not=\Twiddle
```

4.2 And and Or

The definitions of *And* and *Or* are:

$$\mathit{And}\; a\; b \quad = \quad \begin{cases} b & \text{if } a \text{ is true} \\ \mathit{False} & \text{otherwise} \end{cases}$$

$$\mathit{Or}\; a\; b \quad = \quad \begin{cases} \mathit{True} & \text{if } a \text{ is true} \\ b & \text{otherwise} \end{cases}$$

With our definition of what a boolean is, this is just the same as

$$\begin{aligned} \mathit{And}\; a\; b &= a\; b\; \mathit{False} \\ \mathit{Or}\; a\; b &= a\; \mathit{True}\; b \end{aligned}$$

¿From these conditions, we can show that *And* is associative, and has left unit *True* and left zeros *False* and *Error*:

$$\begin{aligned} \mathit{And}\,(\mathit{And}\; a\; b)\; c &= \mathit{And}\; a\,(\mathit{And}\; b\; c) \\ \mathit{And}\ \mathit{True}\; b &= b \\ \mathit{And}\ \mathit{False}\; b &= \mathit{False} \\ \mathit{And}\ \mathit{Error}\; b &= \mathit{Error} \end{aligned}$$

Or is associative, has left unit *False* and left zeros *True* and *Error*:

$$\begin{aligned} \mathit{Or}\,(\mathit{Or}\; a\; b)\; c &= \mathit{Or}\; a\,(\mathit{Or}\; b\; c) \\ \mathit{Or}\ \mathit{False}\; b &= b \\ \mathit{Or}\ \mathit{True}\; b &= \mathit{True} \\ \mathit{Or}\ \mathit{Error}\; b &= \mathit{Error} \end{aligned}$$

De Morgan's laws hold:

$$\begin{aligned} \mathit{Not}\,(\mathit{And}\; a\; b) &= \mathit{Or}\,(\mathit{Not}\; a)\,(\mathit{Not}\; b) \\ \mathit{Not}\,(\mathit{Or}\; a\; b) &= \mathit{And}\,(\mathit{Not}\; a)\,(\mathit{Not}\; b) \end{aligned}$$

and *And* and *Or* left-distribute through one another:

$$\begin{aligned} \mathit{Or}\; a\,(\mathit{And}\; b\; c) &= \mathit{And}\,(\mathit{Or}\; a\; b)\,(\mathit{Or}\; a\; c) \\ \mathit{And}\; a\,(\mathit{Or}\; b\; c) &= \mathit{Or}\,(\mathit{And}\; a\; b)\,(\mathit{And}\; a\; c) \end{aligned}$$

And and *Or* are *not* commutative, though. For example,

$$\begin{aligned}&\mathit{Or}\ \mathit{True}\ \mathit{Error}\\ &\quad= \mathit{True}\ \mathit{True}\ \mathit{Error}\\ &\quad= \mathit{True}\end{aligned}$$

but

$$\begin{aligned}&\mathit{Or}\ \mathit{Error}\ \mathit{True}\\ &\quad= \mathit{Error}\ \mathit{True}\ \mathit{True}\\ &\quad= \mathit{Error}\end{aligned}$$

This is actually quite useful since there are some booleans that need to return an error occasionally. If a is *True* when b is safe (i.e. doesn't become *Error*) and is *False* otherwise, we can say *Or* a b and know we're not going to get an error. This is handy for things like checking for division by zero, or trying to get the first element of an empty list.

Similarly, because of the possibility of *Error*, *And* and *Or* don't right-distribute through each other, as

$$\begin{aligned}&\mathit{Or}\ (\mathit{And}\ \mathit{False}\ \mathit{Error})\ \mathit{True}\\ &\quad\neq \mathit{And}\ (\mathit{Or}\ \mathit{False}\ \mathit{True})\ (\mathit{Or}\ \mathit{Error}\ \mathit{True})\end{aligned}$$

As errors shouldn't crop up, this needn't worry us too much.

```
\def\And#1#2{#1{#2}\False}
\def\Or#1#2{#1\True{#2}}
```

4.3 Lift

Quite a lot of the time we won't be dealing with booleans, but with *predicates*, which are just functions that return a boolean. For example, the predicate *Lessthan* is defined below so that *Lessthan* i j is true whenever i ¡ j. Given a predicate p we would like to be able to *lift* it to *Lift* p, defined:

$$\mathit{Lift}\ p\ f\ g\ x \quad=\quad p\ x\ f\ g\ x$$

For example, *Lift* (*Lessthan* 0) f g takes in a number and applies f to it if it is positive and g to it otherwise. This is quite useful for defining functions.

```
\def\Lift#1#2#3#4{#1{#4}{#2}{#3}{#4}}
```

4.4 Lessthan and TEXif

Finally, we would like to be able to use TEX's built-in booleans as well as our own. For example, we would like a predicate *Lessthan* such that:

$$\mathit{Lessthan}\ i\ j \quad=\quad \begin{cases}\mathit{True} & \text{if } i \text{ ¡ } j\\ \mathit{False} & \text{if } i \geq j\\ \mathit{Error} & \text{otherwise}\end{cases}$$

The *Error* condition happens if we try applying *Lessthan* to something that isn't a number — *Lessthan True False* is *Error*[1]. This is fine as a mathematical definition, but how will we implement it? If we assume we have a macro `\TeXif`, which converts TEX if-statements into booleans, we could just define:

```
\def\Lessthan#1#2{\TeXif{\ifnum#1<#2 }}
```

So the question is just how to define `\TeXif`. Unfortunately, the "obvious" code does not work:

```
\def\TeXif#1#2#3{#1#2\else#3\fi}
```

For example, `\TeXif\iftrue\True\True` doesn't expand out to `\True`. Instead, it expands as:

```
\TeXif\iftrue\True\True
   = \iftrue\True\else\True\fi
   = \True\else\True\fi
   = \else\fi
   =
```

Another common TEXnique is to use a macro `\next` to be the expansion text:

```
\def\TeXif#1#2#3%
   {#1\def\next{#2}\else\def\next{#3}\fi
    \next}
```

However, this uses TEX's stomach to do the `\def`, and we are trying to do this using only the mouth. One (slightly tricky) solution is to use pattern-matching to gobble up the offending `\else` and/or `\fi`.

```
\def\gobblefalse\else\gobbletrue\fi#1#2%
   {\fi#1}
\def\gobbletrue\fi#1#2%
   {\fi#2}
\def\TeXif#1%
   {#1\gobblefalse\else\gobbletrue\fi}
```

So if the TEX if-statement is true, `\gobblefalse` gobbles up the false-text, otherwise `\gobbletrue` gobbles up the true-text. For example,

```
\TeXif\iftrue\True\True
   = \iftrue\gobblefalse\else
          \gobbletrue\fi\True\True
   = \gobblefalse\else
          \gobbletrue\fi\True\True
   = \fi\True
   = \True
```

Phew. And so we have booleans.

5 Lists

A list is a (possibly infinite) sequence of values. For example, the list [1; 2; 3] contains three numbers, the

---

[1] Actually, that's a little white lie — trying to persuade TEX to do run-time type checking isn't much fun. So the TEX implementation of this is actually a *refinement* where the *Error* condition has been replaced by whatever it is TEX does if you try doing `\ifnum`x `<` y when x and y aren't numbers

list [ ] contains none, and the list $[1; 2; 3; \ldots]$ contains infinitely many. A list is either *empty* (written [ ]) or is comprised of a *head* x and a *tail* xs (in which case it's written $x : xs$). For example, $1 : 2 : 3 : [\,]$ is $[1; 2; 3]$.

In a similar fashion to the implementation of booleans, a list xs is implemented as a function of the form

$$\mathit{xs}\,f\,e \;=\; \begin{cases} e & \text{if } \mathit{xs} \text{ is empty} \\ f\,y\,\mathit{ys} & \text{if } \mathit{xs} \text{ has head } y \text{ and tail } \mathit{ys} \end{cases}$$

Again, we are implementing a datatype as a function, a quite powerful trick, just not one usually seen in TEX. We will assume that whenever a list $x : xs$ is applied to f and e, $f\,x$ respects equality. This allows us to assume that if $xs = ys$ then $x : xs = x : ys$, which is handy.

5.1 Nil, Cons, Stream and Singleton

The simplest list is *Nil*, the empty list which we have been writing [ ].

$$\mathit{Nil} \;=\; \mathit{Second}$$

The other possible list is $\mathit{Cons}\,x\,\mathit{xs}$, which has head x and tail xs.

$$\mathit{Cons}\,x\,\mathit{xs}\,f\,e \;=\; f\,x\,\mathit{xs}$$

Every list can be constructed using these functions. The list $[1; 2; 3]$ is $\mathit{Cons}\,1\,(\mathit{Cons}\,2\,(\mathit{Cons}\,3\,\mathit{Nil}))$, and the list $[a; a; a; \ldots]$ is $\mathit{Stream}\,a$ where *Stream* is defined:

$$\mathit{Stream}\,a \;=\; \mathit{Cons}\,a\,(\mathit{Stream}\,a)$$

There's even at least one application for infinite lists, as we'll see in Section 7.

The singleton list $[a]$ is $\mathit{Singleton}\,a$, defined as:

$$\mathit{Singleton}\,a \;=\; \mathit{Cons}\,a\,\mathit{Nil}$$

These all have straightforward TEX definitions.

```
\let\Nil=\Second
\def\Cons#1#2#3#4{#3{#1}{#2}}
\def\Stream#1{\Cons{#1}{\Stream{#1}}}
\def\Singleton#1{\Cons{#1}\Nil}
```

5.2 Head and Tail

So, we can construct any list we like, but we still can't get any information out of it. To begin with, we'd like to be able to get the head and tail of a list.

$$\begin{aligned} \mathit{Head}\,\mathit{xs} &= \mathit{xs}\,\mathit{First}\,\mathit{Error} \\ \mathit{Tail}\,\mathit{xs} &= \mathit{xs}\,\mathit{Second}\,\mathit{Error} \end{aligned}$$

For example, the tail of $x : xs$ is

$$\begin{aligned} &\mathit{Tail}\,(\mathit{Cons}\,x\,\mathit{xs}) \\ &\quad= \mathit{Cons}\,x\,\mathit{xs}\,\mathit{Second}\,\mathit{Error} \\ &\quad= \mathit{Second}\,x\,\mathit{xs} \\ &\quad= \mathit{xs} \end{aligned}$$

The tail of [ ] is, as one would expect,

$$\begin{aligned} &\mathit{Tail}\,\mathit{Nil} \\ &\quad= \mathit{Nil}\,\mathit{Second}\,\mathit{Error} \\ &\quad= \mathit{Error} \end{aligned}$$

And the head of $\mathit{Stream}\,a$ is

$$\begin{aligned} &\mathit{Head}\,(\mathit{Stream}\,a) \\ &\quad= \mathit{Stream}\,a\,\mathit{First}\,\mathit{Error} \\ &\quad= \mathit{Cons}\,a\,(\mathit{Stream}\,a)\,\mathit{First}\,\mathit{Error} \\ &\quad= \mathit{First}\,a\,(\mathit{Stream}\,a) \\ &\quad= a \end{aligned}$$

So we can get the head of an infinite list in finite time. This is fortunate, as otherwise there wouldn't be much point in allowing infinite objects.

```
\def\Head#1{#1\First\Error}
\def\Tail#1{#1\Second\Error}
```

5.3 Foldl and Foldr

Using *Head* and *Tail* we can get at the beginning of any non-empty list, but in general we need more information than that. Rather than write a whole bunch of recursive functions on lists, I'll implement two fairly general functions, with which we can implement (almost) everything else.

Foldl and *Foldr* both take in functions and apply them recursively to a list. *Foldl* starts at the left of the list, and *Foldr* starts at the right. For example,

$$\begin{aligned} \mathit{Foldl}\,f\,e\,[1; 2; 3] &= f\,(f\,(f\,e\,1)\,2)\,3 \\ \mathit{Foldr}\,f\,e\,[1; 2; 3] &= f\,1\,(f\,2\,(f\,3\,e)) \end{aligned}$$

These functions will be used a lot later on. *Foldl* can be defined:

$$\begin{aligned} \mathit{Foldl}\,f\,e\,\mathit{xs} &= \mathit{xs}\,(\mathit{Foldl}'\,f\,e)\,e \\ \mathit{Foldl}'\,f\,e\,x\,\mathit{xs} &= \mathit{Foldl}\,f\,(f\,e\,x)\,\mathit{xs} \end{aligned}$$

So $\mathit{Foldl}\,f\,e\,[\,]$ is

$$\begin{aligned} &\mathit{Foldl}\,f\,e\,\mathit{Nil} \\ &\quad= \mathit{Nil}\,(\mathit{Foldl}'\,f\,e)\,e \\ &\quad= e \end{aligned}$$

And $\mathit{Foldl}\,f\,e\,(x : \mathit{xs})$ is

$$\begin{aligned} &\mathit{Foldl}\,f\,e\,(\mathit{Cons}\,x\,\mathit{xs}) \\ &\quad= \mathit{Cons}\,x\,\mathit{xs}\,(\mathit{Foldl}'\,f\,e)\,e \\ &\quad= \mathit{Foldl}'\,f\,e\,x\,\mathit{xs} \\ &\quad= \mathit{Foldl}\,f\,(f\,e\,x)\,\mathit{xs} \end{aligned}$$

For example, $\mathit{Foldl}\,f\,e\,[1; 2; 3]$ is

$$\begin{aligned} &\mathit{Foldl}\,f\,e\,[1; 2; 3] \\ &\quad= \mathit{Foldl}\,f\,(f\,e\,1)\,[2; 3] \\ &\quad= \mathit{Foldl}\,f\,(f\,(f\,e\,1)\,2)\,[3] \end{aligned}$$

$$\begin{aligned} &= \mathit{Foldl}\, f\, (f\, (f\, (f\, e\, 1)\, 2)\, 3)\, [\,] \\ &= f\, (f\, (f\, e\, 1)\, 2)\, 3 \end{aligned}$$

as promised. Similarly, we can define *Foldr* as

$$\begin{aligned} \mathit{Foldr}\, f\, e\, \mathit{xs} &= \mathit{xs}\, (\mathit{Foldr}'\, f\, e)\, e \\ \mathit{Foldr}'\, f\, e\, x\, \mathit{xs} &= f\, x\, (\mathit{Foldr}\, f\, e\, \mathit{xs}) \end{aligned}$$

For *Foldr f* to respect equality, *f x* should respect equality.

When we do the unfolding, we discover that

$$\begin{aligned} \mathit{Foldr}\, f\, e\, [\,] &= e \\ \mathit{Foldr}\, f\, e\, (x : \mathit{xs}) &= f\, e\, (\mathit{Foldr}\, f\, e\, \mathit{xs}) \end{aligned}$$

Foldr tends to be more efficient than *Foldl*, because *Foldl* has to run along the entire list before it can start applying *f*, whereas *Foldr* can apply *f* straight away. If *f* is a lazy function, this can make quite a difference. *Foldl* on infinite lists, anyone?

```
\def\Foldl#1#2#3%
   {#3{\Foldl@{#1}{#2}}{#2}}
\def\Foldl@#1#2#3#4%
   {\Foldl{#1}{#1{#2}{#3}}{#4}}
\def\Foldr#1#2#3%
   {#3{\Foldr@{#1}{#2}}{#2}}
\def\Foldr@#1#2#3#4%
   {#1{#3}{\Foldr{#1}{#2}{#4}}}
```

5.4 Cat

Given two lists, we would like to be able to stick them together, which is what *Cat* (short for "concatenate") does. For example, $\mathit{Cat}\,[1;2]\,[3;4]$ is $[1;2;3;4]$. It can be defined using *Foldr*:

$$\mathit{Cat}\, \mathit{xs}\, \mathit{ys} = \mathit{Foldr}\, \mathit{Cons}\, \mathit{ys}\, \mathit{xs}$$

So

$$\begin{aligned} &\mathit{Cat}\, [1;2]\, [3;4] \\ &\quad= \mathit{Foldr}\, \mathit{Cons}\, [3;4]\, [1;2] \\ &\quad= \mathit{Cons}\, 1\, (\mathit{Foldr}\, \mathit{Cons}\, [3;4]\, [2]) \\ &\quad= \mathit{Cons}\, 1\, (\mathit{Cons}\, 2\, (\mathit{Foldr}\, \mathit{Cons}\, [3;4]\, [\,])) \\ &\quad= \mathit{Cons}\, 1\, (\mathit{Cons}\, 2\, [3;4]) \\ &\quad= [1;2;3;4] \end{aligned}$$

The TeX code for `\Cat` is suspiciously similar to its mathematical definition.

```
\def\Cat#1#2{\Foldr\Cons{#2}{#1}}
```

5.5 Reverse

We can reverse any list with the function *Reverse*, defined using *Foldl*:

$$\mathit{Reverse} = \mathit{Foldl}\, (\mathit{Twiddle}\, \mathit{Cons})\, \mathit{Nil}$$

For example, $\mathit{Reverse}\,[1;2;3]$ can be calculated:

$$\begin{aligned} &\mathit{Reverse}\, [1;2;3] \\ &\quad= \mathit{Foldl}\, (\mathit{Twiddle}\, \mathit{Cons})\, \mathit{Nil}\, [1;2;3] \\ &\quad= \mathit{Twiddle}\, \mathit{Cons} \\ &\qquad\quad (\mathit{Twiddle}\, \mathit{Cons}\, (\mathit{Twiddle}\, \mathit{Cons}\, \mathit{Nil}\, 1)\, 2)\, 3 \\ &\quad= \mathit{Cons}\, 3\, (\mathit{Cons}\, 2\, (\mathit{Cons}\, 1\, \mathit{Nil})) \\ &\quad= [3;2;1] \end{aligned}$$

The TeX code for `\Reverse` doesn't even take in any parameters.

```
\def\Reverse{\Foldl{\Twiddle\Cons}\Nil}
```

5.6 All, Some and Isempty

Given a predicate p, we can find out if all the elements of a list satisfy p with *All p*. Similarly we can find if something in the list satisfies p with *Some p*. For example,

$$\begin{aligned} \mathit{All}\, (\mathit{Lessthan}\, 1)\, [1;2;3] &= \mathit{False} \\ \mathit{Some}\, (\mathit{Lessthan}\, 1)\, [1;2;3] &= \mathit{True} \end{aligned}$$

These can be defined

$$\begin{aligned} \mathit{All}\, p &= \mathit{Foldr}\, (\mathit{Compose}\, \mathit{And}\, p)\, \mathit{True} \\ \mathit{Some}\, p &= \mathit{Foldr}\, (\mathit{Compose}\, \mathit{Or}\, p)\, \mathit{False} \end{aligned}$$

For example, *Isempty* can be defined

$$\mathit{Isempty} = \mathit{All}\, (\mathit{First}\, \mathit{False})$$

This is probably not the most efficient check in the world, but we hardly ever need it — *Foldl* or *Foldr* will normally do the job.

```
\def\All#1{\Foldr{\Compose\And{#1}}\True}
\def\Some#1{\Foldr{\Compose\Or{#1}}\False}
\def\Isempty{\All{\First\False}}
```

5.7 Filter

Filter takes a predicate p and a list *xs*, and returns a list containing only those elements of *xs* that satisfy p. For example,

$$\mathit{Filter}\, (\mathit{Lessthan}\, 1)\, [1;2;3] = [2;3]$$

Filter can be defined as a *Foldr*:

$$\mathit{Filter}\, p = \mathit{Foldr}\, (\mathit{Lift}\, p\, \mathit{Cons}\, \mathit{Second})\, \mathit{Nil}$$

Another easy bit of TeX:

```
\def\Filter#1%
   {\Foldr{\Lift{#1}\Cons\Second}\Nil}
```

5.8 Map

Map takes a function f and a list xs and applies f to every element of xs. For example,

$$\mathit{Map}\, f\,[1; 2; 3] \quad = \quad [f\,1; f\,2; f\,3]$$

This is another job for *Foldr*.

$$\mathit{Map}\, f \quad = \quad \mathit{Foldr}\,(\mathit{Compose}\ \mathit{Cons}\, f)\ \mathit{Nil}$$

We shall see *Map* used later on, to convert from a list of names such as [`Fac-yawn`, `Fac-cows`], to a list of labels such as [i, iii].

```
\def\Map#1{\Foldr{\Compose\Cons{#1}}\Nil}
```

5.9 Insert

The only function we need which isn't easily defined as a reduction is *Insert*, which inserts an element into a sorted list. For example,

$$\mathit{Insert}\ \mathit{Lessthan}\ 3\,[1; 2; 4; 5] \quad = \quad [1; 2; 3; 4; 5]$$

Insert takes in an ordering as its first parameter, so we're not stuck with one particular order. It is defined directly in terms of the definition of lists.

$$\begin{aligned}
\mathit{Insert}\ o\ x\ xs &\quad = \quad xs\,(\mathit{Insert}'\ o\ x)\,(\mathit{Singleton}\ x)\\
\mathit{Insert}'\ o\ x\ y\ ys &\quad = \quad o\ x\ y\\
&\qquad\qquad (\mathit{Cons}\ x\,(\mathit{Cons}\ y\ ys))\\
&\qquad\qquad (\mathit{Cons}\ y\,(\mathit{Insert}\ o\ x\ ys))
\end{aligned}$$

We can then define the function all this has been leading up to, *Insertsort* which takes an ordering and a list, and insert-sorts the list according to the ordering. For example,

$$\mathit{Insertsort}\ \mathit{Lessthan}\,[2; 3; 1; 2] \quad = \quad [1; 2; 2; 3]$$

We can implement this as a fold:

$$\mathit{Insertsort}\ o \quad = \quad \mathit{Foldr}\,(\mathit{Insert}\ o)\ \mathit{Nil}$$

And so we've got sorted lists.

```
\def\Insert#1#2#3%
   {#3{\Insert@{#1}{#2}}{\Singleton{#2}}}
\def\Insert@#1#2#3#4%
   {#1{#2}{#3}%
      {\Cons{#2}{\Cons{#3}{#4}}}%
      {\Cons{#3}{\Insert{#1}{#2}{#4}}}}
\def\Insertsort#1{\Foldr{\Insert{#1}}\Nil}
```

Interestingly, as we have implemented unbounded lists in TeX's mouth, this means we can implement a Turing Machine. So, if you believe the Church-Turing thesis, TeX's mouth is as powerful as any computer anywhere. Isn't that good to know?

6 Sorting reference lists

So, these are the macros I've got to play with — how do we apply them to sorting lists of references? Well, I'm using LaTeX, which keeps the current reference in a macro called `\@currentlabel`, which is 6 at the moment, as this is Section 6. So I just need to store the value of `\@currentlabel` somehow.

Fortunately, I'm only ever going to be making references to facts earlier on in the document, in order to make sure I'm not proving any results in terms of themselves. So I don't need to play around with auxiliary files, and can just do everything in terms of macros.

6.1 Number and Label

Each label in the document is given a unique number, in the order the labels were put down. So the number of `Fac-cows` is `\Number{Fac-cows}`, which expands out to 1, the number of `Fac-people` is 2, and so on.

Each number has an associated label with it. For example, the first label is `\Label{1}`, which is i, the second label is ii and so on. So to find the label for `Fac-cows`, we say `\Label{\Number{Fac-cows}}` which expands out to i.

These numbers and labels are kept track of in macros. For example, the number of `Fac-cows` is kept in $\boxed{\texttt{Number-Fac-cows}}$. Similarly, the first label is kept in $\boxed{\texttt{Label-1}}$. As these macros have dashes in their names, they aren't likely to be used already.

So the TeX code for `\Number` and `\Label` is pretty simple.

```
\def\Number#1{\csname Number-#1\endcsname}
\def\Label#1{\csname Label-#1\endcsname}
```

6.2 Lastnum and Forward

The number of the most recent label is kept in `\Lastnum`.

```
\newcount\Lastnum
```

To put down a label `Foo`, I type `\Forward{Foo}`. This increments the counter `\Lastnum`, and `\xdefs` $\boxed{\texttt{Number-Foo}}$ to be the value of `\Lastnum`, which is now 4. So `\Number{Foo}` now expands to 4. Similarly, it `\xdefs` $\boxed{\texttt{Label-4}}$ to be `\@currentlabel`, which is currently 6.2. So `\Label{\Number{Foo}}` now expands to 6.2.

```
\def\Forward#1%
   {\global\advance\Lastnum by 1
    \csnameafter\xdef{Number-#1}%
       {\the\Lastnum}%
    \csnameafter\xdef{Label-\the\Lastnum}%
       {\@currentlabel}}
```

This uses `\csnameafter\foo{bar}`, which expands out to `\foo\bar`.

```
\def\csnameafter#1#2%
   {\expandafter#1\csname#2\endcsname}
```

6.3 Listize, Unlistize and Show

At the moment, lists have to be built up using `\Cons` and `\Nil`, which is rather annoying. Similarly, we can't actually do anything with a list once we've built it. We'd like some way of converting lists in the form `[a,b,c]` to and from the form $[a;\, b;\, c]$. This is done with `\Listize` and `\Unlistize`. So `\Listize[a,b,c]` expands to

```
\Cons{a}{\Cons{b}{\Cons{c}{\Nil}}}
```

Similarly, `\Unlistize` takes the list $[a;\, b;\, c]$ and expands out to `[a, b, c]`. `\Unlistize` is done with a *Foldr*.

```
\def\Unlistize#1{[#1\Unlistize@{}]}
\def\Unlistize@#1{#1\Foldr\Commaize{}}
\def\Commaize#1#2{, #1#2}
```

The macro `\Listize` is just a TeX hack with pattern matching. It would have been nice to use `\@ifnextchar` for this, but that uses `\futurelet`, which doesn't expand in the mouth. Oh well.

```
\def\Listize[#1]%
   {\Listize@[#1,\relax]}
\def\Listize@#1,#2]%
   {\TeXif{\ifx\relax#2}%
      {\Singleton{#1}}%
      {\Cons{#1}{\Listize@#2]}}
```

This only works for nonempty lists — `\Listize[]` produces the singleton list `\Singleton{}`. It also uses `\relax` as its end-of-list character, so lists with `\relax` in them have to be done by hand. You can't win them all. So

```
$\Unlistize{\Listize[a,b,c]}$
```

produces $[a;\, b;\, c]$. This is such a common construction that I've defined a macro `\Show` such that `\Show\foo[a,b,c]` expands out to

```
\Unlistize{\foo{\Listize[a,b,c]}}
```

For example, the equation

$$\mathit{Filter}\ (\mathit{Lessthan}\ 1)\ [1;2;3] \quad = \quad [2;3]$$

was generated with

```
\begin{eqnarray*}
   Filter\,(Lessthan\,1)\,[1,2,3]
      &=& \Show\Filter{\Lessthan 1}[1,2,3]
\end{eqnarray*}
```

Many of the examples in this article were typeset this way.

```
\def\Show#1[#2]%
   {\Unlistize{#1{\Listize[#2]}}}
```

6.4 By

Given these macros, we can now sort any list of references with *Bylist*, defined

$$\begin{aligned}\mathit{Bylist}\ \mathit{xs} \quad = \quad & \mathit{Map}\ \mathit{Label}\\ & \quad (\mathit{Insertsort}\ \mathit{Lessthan}\\ & \qquad (\mathit{Map}\ \mathit{Number}\ \mathit{xs}))\end{aligned}$$

This takes in a list of label names like `Fac-yawn`, converts it into a list of numbers with *Map Number*, sorts the resulting list with *Insertsort Lessthan*, and finally converts all the numbers into labels like iii with *Map Label*. For example,

$$\begin{aligned}& \mathit{Bylist}\ [\mathtt{Fac\text{-}yawn};\, \mathtt{Fac\text{-}cows}]\\ & \quad = \mathit{Map}\ \mathit{Label}\ (\mathit{Insertsort}\ \mathit{Lessthan}\\ & \qquad\quad (\mathit{Map}\ \mathit{Number}\ [\mathtt{Fac\text{-}yawn};\, \mathtt{Fac\text{-}cows}]))\\ & \quad = \mathit{Map}\ \mathit{Label}\ (\mathit{Insertsort}\ \mathit{Lessthan}\ [3;\, 1])\\ & \quad = \mathit{Map}\ \mathit{Label}\ [1;\, 3]\\ & \quad = [i;\, iii]\end{aligned}$$

The TeX code for this is

```
\def\Bylist#1%
   {\Map\Label
      {\Insertsort\Lessthan
         {\Map\Number{#1}}}}
```

So we can now stick all this together, and define the macro `\By` that prints out lists of references. It is

```
\def\By{\Show\Bylist}
```

So `\By[Fac-yawn,Fac-cows]` is [i, iii]. Which is quite nice.

7 Other applications

Is all this worth it? Well, I've managed to get my lists of facts in order, but that's not the world's most astonishing application. There are other things that these lists are useful for, though.

For example, Damian Cugley has a macro package under development for laying out magazines. MagTeX's output routine needs to be quite smart, as magazines often have gaps where illustrations or photographs are going to live. In general, each block of text needs to be output in a different fashion from every other block of text. This will be handled by keeping an infinite list of output routines. Each time a box is cut off the scroll to be output, the head of the list is chopped off and is used as the output routine for that box. That way, quite complex page shapes can be built up.

Mainly, though, these macros were written just as a challenge. I learned quite a lot about TeX and needed some TeXniques I'd never seen before. It was also quite pleasing to see that TeX code can be formally verified, albeit in a rather noddy way.

Without some sort of abstract view of lists, these TeX macros could not have been written.

8 Acknowledgements

Thanks to Jeremy Gibbons for letting me bounce ideas off him and spotting the duff ones, to Damian Cugley for saying “Do you really think TeX is meant to do this?”, and to the Problem Solving Club for hearing me out. This work was sponsored by the Science and Engineering Research Council and Hewlett Packard.

◇ Alan Jeffrey
Programming Research Group
Oxford University
11 Keble Road
Oxford OX1 3QD
`Alan.Jeffrey@uk.ac.oxford.prg`