Chapter 25

Reasoning About Effects: Seeing the Wood Through the Trees

Graham Hutton[1], Diana Fulger[2]
Category: Research Paper

Abstract: Pure functional languages such as Haskell support programming with impure effects by exploiting mathematical notions such as monads, applicative functors, and arrows. However, in contrast to the wealth of research on the use of these notions to write effectful programs, there has been comparatively little progress on reasoning about the resulting programs. In this article we focus on this problem, using a simple but instructive example concerned with relabelling binary trees.

25.1 INTRODUCTION

Mathematical notions such as monads [8], applicative functors [5] and arrows [4] are now well established as mechanisms for pure programming with impure effects such as exceptions, non-determinism, state, and input/output [10]. In particular, these notions provide uniform interfaces for programming with effects, allowing programmers to focus on the essential high-level concepts, rather than the low-level implementation details.

Because these notions simplify the process of *writing* programs that utilise effects, we might similarly expect that they also simplify the process of *reasoning* about the resulting programs. Unfortunately, however, it still remains standard practice to just expand out the basic effectful operations when trying to reason about such programs, which seems rather unsatisfactory.

[1]University of Nottingham, UK; gmh@cs.nott.ac.uk
[2]University of Nottingham, UK; dqf@cs.nott.ac.uk

In this article we focus on the problem of reasoning about effectful programs, with the specific aim of avoiding such an expansion. In particular, we consider a simple but instructive example concerned with relabelling binary trees. Our thesis is that the simplicity of this example enables us to reveal a number of issues regarding reasoning about effects that may otherwise have been obscured by the complexity of larger examples. The article makes the following contributions:

- We prove the correctness of a tree relabelling function. Somewhat surprisingly, even an elementary proof of this result does not seem to have appeared in the functional programming literature before;
- We identify a number of basic properties of state in the context of applicative functors, which allow us to prove the correctness of the relabelling function using simple, point-free equational reasoning;
- We discuss a number of issues that arise from the development of this proof, in particular the role that types play, and the idea of factorising a computation into 'essential' and 'plumbing' parts, which in this example leads naturally to the idea of generalising from finite to infinite structures;
- We propose the tree relabelling problem as a useful case study for other approaches to reasoning about effectful programs.

The article is aimed at a reader who is familiar with the basics of reasoning about functional programs, say to the level of [1], but no specialist knowledge about effectful programming and reasoning is assumed. An extended version of the article that includes all proofs is available from the authors' web pages.

25.2 RELABELLING A TREE

In this section we introduce the example that will form the central focus of this article. Let us begin by defining a type *Tree a* of binary trees whose leaves contain values of some parameter type *a*, together with a function that returns a list of the leaf values or *labels* that occur in such a tree:

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Tree}\ a & = & \mathit{Leaf}\ a \mid \mathit{Node}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) \\
\mathit{labels} & :: & \mathit{Tree}\ a \to [a] \\
\mathit{labels}\ (\mathit{Leaf}\ x) & = & [x] \\
\mathit{labels}\ (\mathit{Node}\ l\ r) & = & \mathit{labels}\ l +\!\!+ \mathit{labels}\ r
\end{array}
$$

Now consider the problem of defining a function that replaces each leaf value in such a tree with a unique or *fresh* integer. This can be achieved in many different ways, but perhaps the simplest is to thread a fresh integer through a traversal of

the tree, which can be implemented as follows:

$$
\begin{array}{lcl}
\mathit{label} & :: & \mathit{Tree}\ a \to \mathit{Int} \to (\mathit{Tree}\ \mathit{Int}, \mathit{Int}) \\
\mathit{label}\ (\mathit{Leaf}\ x)\ n & = & (\mathit{Leaf}\ n, n+1) \\
\mathit{label}\ (\mathit{Node}\ l\ r)\ n & = & (\mathit{Node}\ l'\ r', n'') \\
 & & \textbf{where} \\
 & & \quad (l', n') = \mathit{label}\ l\ n \\
 & & \quad (r', n'') = \mathit{label}\ r\ n' \\
\mathit{relabel} & :: & \mathit{Tree}\ a \to \mathit{Tree}\ \mathit{Int} \\
\mathit{relabel}\ t & = & \mathit{fst}\ (\mathit{label}\ t\ 0)
\end{array}
$$

What does it mean for the relabelling function to be correct? Firstly, it must preserve the shape of the original tree. And secondly, each label in the resulting tree must be unique. For our purposes, we only consider the second property, which can be formalised and proved correct as follows.

Theorem 25.1 (correctness of *relabel***).** *For all finite trees t:*

$$\mathit{nodups}\ (\mathit{labels}\ (\mathit{relable}\ t))$$

where

$$
\begin{array}{lcl}
\mathit{nodups} & :: & \mathit{Eq}\ a \Rightarrow [a] \to \mathit{Bool} \\
\mathit{nodups}\ [\,] & = & \mathit{True} \\
\mathit{nodups}\ (x : xs) & = & \neg\ (\mathit{elem}\ x\ xs) \wedge \mathit{nodups}\ xs
\end{array}
$$

Proof: by equational reasoning, using two lemmas.

let $(t', n) = \mathit{label}\ t\ 0$ **in**

$$
\begin{array}{ll}
 & \mathit{nodups}\ (\mathit{labels}\ (\mathit{relabel}\ t)) \\
= & \quad \{\ \text{applying } \mathit{relabel}\ \} \\
 & \mathit{nodups}\ (\mathit{labels}\ t') \\
= & \quad \{\ \text{behaviour of } \mathit{labels}\ \text{(lemma 25.2)}\ \} \\
 & \mathit{nodups}\ [0\,..\,n-1] \\
= & \quad \{\ \text{intervals have no duplicates (lemma 25.3)}\ \} \\
 & \mathit{True}
\end{array}
$$

□

Lemma 25.2 (behaviour of *labels***).** *For all finite trees t and integers n:*

$$\mathit{label}\ t\ n = (t', n') \ \Rightarrow\ n < n' \ \wedge\ \mathit{labels}\ t' = [n\,..\,n'-1]$$

That is, *label* returns a new label n' that is strictly greater than the initial label n, and a new tree whose labels are precisely the non-empty interval $[n\,..\,n'-1]$.

Lemma 25.3 (intervals have no duplicates). *For all integers $a \leqslant b$:*

$$\mathit{nodups}\ [a\,..\,b]$$

The proofs of the above lemmas are straightforward, but require considerable care, and proceed using the elementary technique of expanding out definitions. The remainder of the article shows how the relabelling function, together with its proof of correctness, can be reworked in a higher-level manner.

25.3 RELABELLING USING MONADS

Our relabelling function is currently defined by explicitly threading a fresh label through the computation. We now consider how it may be defined in a more structured manner using monads [11]. Recall that in Haskell, the notion of a monad is captured by the following class declaration:

$$\begin{array}{l}\textbf{class}\ \mathit{Monad}\ m\ \textbf{where}\\ \quad \begin{array}{lcl}\mathit{return} & :: & a \to m\ a\\ (\gg\!=) & :: & m\ a \to (a \to m\ b) \to m\ b\end{array}\end{array}$$

That is, a parameterised type *m* is a member of the class *Monad* of monadic types if it is equipped with *return* and $(\gg\!=)$ functions of the specified types. A typical monadic expression built using these two functions has the following structure:

$$\begin{array}{l}\mathit{m1} \gg\!= \lambda \mathit{v1} \to\\ \mathit{m2} \gg\!= \lambda \mathit{v2} \to\\ \vdots\\ \mathit{mn} \gg\!= \lambda \mathit{vn} \to\\ \mathit{return}\ (\mathit{f}\ \mathit{v1}\ \mathit{v2}\ \cdots\ \mathit{vn})\end{array}$$

That is, apply *m1* and call its result value *v1*; then apply *m2* and call its result value *v2*; ...; then apply *mn* and call its result value *vn*; and finally, combine all the results by applying the function *f*. Haskell provides a special syntax for such monadic expressions, allowing them to be expressed in the following form:

$$\begin{array}{l}\textbf{do}\ \mathit{v1} \leftarrow \mathit{m1}\\ \quad \mathit{v2} \leftarrow \mathit{m2}\\ \quad \vdots\\ \quad \mathit{vn} \leftarrow \mathit{mn}\\ \quad \mathit{return}\ (\mathit{f}\ \mathit{v1}\ \mathit{v2}\ \cdots\ \mathit{vn})\end{array}$$

For the purposes of defining our relabelling function, we will utilise the state monad, which is based upon the notion of a *state transformer* (abbreviated by *ST*), which in turn is a function that takes a state as its argument, and returns a pair comprising a result value and a new state:

$$\textbf{type}\ \mathit{ST}\ s\ a \quad = \quad s \to (a, s)$$

For any type s of states, it is straightforward to make $ST\ s$ into a monadic type:

$$\begin{array}{lcl}
\textbf{instance}\ \mathit{Monad}\ (\mathit{ST}\ s)\ \textbf{where} & & \\
\quad \mathit{return} & :: & a \to \mathit{ST}\ s\ a \\
\quad \mathit{return}\ v & = & \lambda s \to (v, s) \\
\quad (\gg\!=) & :: & \mathit{ST}\ s\ a \to (a \to \mathit{ST}\ s\ b) \to \mathit{ST}\ s\ b \\
\quad st \gg\!= f & = & \lambda s \to \textbf{let}\ (v, s') = st\ s\ \textbf{in}\ (f\ v)\ s'
\end{array}$$

That is, *return* converts a value into a state transformer that simply returns that value without modifying the state. In turn, $\gg\!=$ provides a means of sequencing state transformers: $st \gg\!= f$ applies the state transformer st to an initial state s, then applies the function f to the resulting value v to give a second state transformer $(f\ v)$, which is then applied to the modified state s' to give the final result.

Aside: the above declaration for $ST\ s$ as a monadic type is not actually valid Haskell, as types defined using the **type** mechanism are not permitted to be made into instances of classes. The solution is to redefine ST using the **data** or **newtype** mechanisms, but this requires the introduction of a dummy constructor. For simplicity, however, we do not do this here. *End of aside.*

Now let us return to the problem of relabelling trees, which was based upon the idea of taking the next fresh integer as an additional argument, and returning the next fresh integer as an additional result. In other words, tree relabelling can be expressed using the notion of a state transformer, in which the state is simply the next fresh integer, and the result value is the relabelled tree.

In order to generate a fresh integer, we define a special state transformer that returns the current state as its result, and the next integer as the new state:

$$\begin{array}{lcl}
\mathit{fresh} & :: & \mathit{ST\ Int\ Int} \\
\mathit{fresh} & = & \lambda n \to (n, n+1)
\end{array}$$

Using this, together with the **do** notation that is supported as a result of ST being monadic, it is now straightforward to redefine the tree relabelling function

$$\mathit{label} \quad :: \quad \mathit{Trea}\ a \to \mathit{Int} \to (\mathit{Tree\ Int}, a)$$

in monadic style as a function that takes a tree and returns a state transformer that produces the same tree with each leaf labelled by a fresh integer:

$$\begin{array}{lcl}
\mathit{label} & :: & \mathit{Tree}\ a \to \mathit{ST\ Int}\ (\mathit{Tree\ Int}) \\
\mathit{label}\ (\mathit{Leaf}\ x) & = & \textbf{do}\ n \leftarrow \mathit{fresh} \\
 & & \quad \mathit{return}\ (\mathit{Leaf}\ n) \\
\mathit{label}\ (\mathit{Node}\ l\ r) & = & \textbf{do}\ l' \leftarrow \mathit{label}\ l \\
 & & \quad r' \leftarrow \mathit{label}\ r \\
 & & \quad \mathit{return}\ (\mathit{Node}\ l'\ r')
\end{array}$$

Note that when expressed in this manner, the programmer no longer has to worry about the tedious and error-prone task of plumbing of fresh labels, as this is taken care of automatically by the underlying monadic primitives.

25.4 RELABELLING USING APPLICATIVE FUNCTORS

While the definition of the *label* function is much improved using monads, there is still room for further simplification. In particular, the use of the **do** notation requires that the result of each monadic expression is given a name, even if this name is only used once, as in the definition of *label*. We now show how the need for such names can be avoided by redefining label using applicative functors [5], which in Haskell are captured by the following class declaration:

$$\begin{array}{l}\textbf{class}\ \mathit{Applicative}\ f\ \textbf{where}\\ \quad \begin{array}{lcl}\mathit{pure} & :: & a \to f\ a\\ (\circledast) & :: & f\ (a \to b) \to f\ a \to f\ b\end{array}\end{array}$$

The idea is that the function *pure* converts a value into a computation that return this value without performing any effects (and hence plays the same role as *return* for monads), while $\circledast$ provides a lifted form of function application in which the function itself, together with its argument and result, are produced by computations that may have effects. As with normal function application, $\circledast$ is assumed to associate to the left; for example, $f \circledast x \circledast y$ means $(f \circledast x) \circledast y$.

Applicative functors are more general than monads, in the sense that every monad is applicative, with the $\circledast$ operator defined by $mf \circledast mx = \textbf{do}\ \{f \leftarrow mf; x \leftarrow mx; \mathit{return}\ (f\ x)\}$. As suggested by the name, every applicative functor is also a functor, with $\mathit{map} :: (a \to b) \to f\ a \to f\ b$ defined by $\mathit{map}\ f\ x = \mathit{pure}\ f \circledast x$.

A typical applicative expression has the following structure:

$$\mathit{pure}\ f \circledast m1 \circledast m2 \circledast \cdots\ mn$$

That is, a pure function f is applied to the results of a sequence of effectful arguments *m1*, *m2*, ..., *mn*. In fact, using laws that every applicative functor must satisfy [5], any expression built using *pure* and $\circledast$ can be transformed into this form, which is abbreviated using the following special syntax:

$$[\![\ f\ m1\ m2\ \cdots\ mn\ ]\!]$$

Note that unlike with the **do** notation, there is no longer any need to name the results of the argument computations, as this is taken care of automatically by the underlying applicative functor primitives. This special syntax is not currently supported by Haskell, but we will use it in this article.

Even though state transformers are monadic and are hence applicative as described above, it is useful to consider an explicit declaration of this fact:

$$\begin{array}{l}\textbf{instance}\ \mathit{Applicative}\ (\mathit{ST}\ s)\ \textbf{where}\\ \quad \begin{array}{lcl}\mathit{pure} & :: & a \to \mathit{ST}\ s\ a\\ \mathit{pure}\ v & = & \lambda s \to (v, s)\\ (\circledast) & :: & \mathit{ST}\ s\ (a \to b) \to \mathit{ST}\ s\ a \to \mathit{ST}\ s\ b\\ mf \circledast mx & = & \lambda s \to \textbf{let}\ (f, s') = mf\ s\\ & & \qquad\quad (x, s'') = mx\ s'\\ & & \quad \textbf{in}\ (f\ x, s'')\end{array}\end{array}$$

That is, *pure v* simply returns the value *v* without modifying the state, while $mf \circledast mx$ begins by applying mf to the initial state s to produce a function f and a new state s', then applies mx to this state to produce a value x and a new state s'', and concludes by returning $f\ x$ as the final result and s'' as the final state.

Using the fact that *ST* is applicative, it is now straightforward to redefine the *label* function in applicative style, as follows:

$$\begin{array}{lcl} \mathit{label} & :: & \mathit{Tree}\ a \to \mathit{ST}\ \mathit{Int}\ (\mathit{Tree}\ \mathit{Int}) \\ \mathit{label}\ (\mathit{Leaf}\ x) & = & [\![\ \mathit{Leaf}\ \mathit{fresh}\ ]\!] \\ \mathit{label}\ (\mathit{Node}\ l\ r) & = & [\![\ \mathit{Node}\ (\mathit{label}\ l)\ (\mathit{label}\ r)\ ]\!] \end{array}$$

This definition captures in a clear and concise way our intuition about relabelling, namely that a *Leaf* is relabelled by replacing its argument value by a fresh label, and that a *Node* is relabelled by recursively relabelling its two subtrees. All the plumbing of fresh labels and argument values is handled automatically.

Just as the *label* function is now expressed in a high-level manner, so we would like to prove the correctness of this function in a similarly high-level manner. Our approach to this is based upon two ideas, namely to work at the point-free (or function) level as far as possible, and to generalise from finite to infinite lists of labels. In the next section we define a number of functions on pairs and infinite lists that will be used to formulate and prove our main correctness theorem.

25.5 INTERLUDE ON PAIRS AND LISTS

Because state transformers return pairs, when working at the point-free level it is natural to utilise a number of standard functions on pairs:

$$\begin{array}{lcl} f \times g & = & \lambda(x,y) \to (f\ x, g\ y) \\ \langle f, g \rangle & = & \lambda x \to (f\ x, g\ x) \\ \mathit{assoc}\ (x,(y,z)) & = & ((x,y),z) \\ \mathit{unassoc}\ ((x,y),z) & = & (x,(y,z)) \end{array}$$

These functions satisfy a number of familiar equational properties arising from the categorical nature of pairs [6], which will be used freely throughout the rest of this article. For example, for all functions of the appropriate types, the product $(\times)$ and split $\langle -, - \rangle$ operators satisfy the following fusion properties:

$$\begin{array}{lcl} (f \times g) \circ (h \times i) & = & (f \circ h) \times (g \circ i) \\ (f \times g) \circ \langle h, i \rangle & = & \langle f \circ h, g \circ i \rangle \\ \langle f, g \rangle \circ h & = & \langle f \circ h, g \circ h \rangle \end{array}$$

We will also use uncurried versions of the list operators cons $(:)$ and append $(+\!\!+)$, and a special variant of function composition $(\circ)$ that allows a curried func-

tion with two arguments to be composed with a function with a single argument:

$$\begin{array}{lcl} \mathit{cons} & :: & (a,[a]) \to [a] \\ \mathit{cons} & = & \mathit{uncurry}\ (:) \\ \mathit{append} & :: & ([a],[a]) \to [a] \\ \mathit{append} & = & \mathit{uncurry}\ (+\!\!+) \\ (\bullet) & :: & (c \to d) \to (a \to b \to c) \to a \to b \to d \\ f \bullet g & = & \mathit{curry}\ (f \circ \mathit{uncurry}\ g) \end{array}$$

Now let us consider infinite lists (or streams) of labels, such as those produced by the function *from*, which generates an infinitely ascending list of numbers:

$$\begin{array}{lcl} \mathit{from} & :: & \mathit{Int} \to [\mathit{Int}] \\ \mathit{from}\ n & = & n : \mathit{from}\ (n+1) \end{array}$$

For example, $\mathit{from}\ 0 = 0:1:2:3:\cdots$. In this setting, our current definition for the function *nodups* that decides if a list has no duplicate elements is no longer appropriate, as applying this function to an infinite list results in non-termination. Our solution is to utilise the following new definition:

$$\mathit{nodups}\ xs \ \Leftrightarrow\ \mathit{rmdups}\ xs = xs$$

where

$$\begin{array}{lcl} \mathit{rmdups} & :: & \mathit{Eq}\ a \Rightarrow [a] \to [a] \\ \mathit{rmdups}\ [\,] & = & [\,] \\ \mathit{rmdups}\ (x:xs) & = & x : \mathit{rmdups}\ (\mathit{filter}\ (\neq x)\ xs) \end{array}$$

That is, a list contains no duplicates if removing duplicates does not change the list. Note that the use of mathematical equality (=) rather than Haskell equality (==) in the new definition for *nodups*, which is required to avoid the issue of non-termination, means that *nodups* is now a meta-level definition rather than a Haskell definition. Moreover, we will in fact use a lifted version of *nodups* that applies to functions that generate lists, rather than lists themselves:

$$\mathit{nodups}\ f \ \Leftrightarrow\ \mathit{rmdups} \circ f = f$$

That is, a function produces no duplicates if removing duplicates after applying the function does not change the resulting list.

25.6 PROOF OF CORRECTNESS

Using the new operators from the previous section, we can now state the correctness of our *label* function in a point-free manner.

Theorem 25.4 (correctness of *label*). *For all finite trees t:*

$$\mathit{nodups}\ (\mathit{append} \circ (\mathit{labels} \times \mathit{from}) \circ \mathit{label}\ t)$$

That is, if we label a tree and then append the list of labels in the resulting tree with the infinite list of unused labels (produced by applying *from* to the first unused label) then each label in the resulting list is unique. This result is stronger than we actually need, by considering both the labels in the tree and the unused labels, but strengthening the result in this manner leads to a simple and natural proof.

Proof: by direct application of two lemmas.

$$
\begin{array}{ll}
 & \mathit{nodups}\ (\mathit{append} \circ (\mathit{labels} \times \mathit{from}) \circ \mathit{label}\ t) \\
\Leftrightarrow & \quad \{\ \text{behaviour of } \mathit{label} \text{ (lemma 25.5)}\ \} \\
 & \mathit{nodups}\ \mathit{from} \\
\Leftrightarrow & \quad \{\ \mathit{from} \text{ produces no duplicates (lemma 25.8)}\ \} \\
 & \mathit{True}
\end{array}
$$

□

Lemma 25.5 (behaviour of *label*). *For all finite trees t:*

$$\mathit{append} \circ (\mathit{labels} \times \mathit{from}) \circ \mathit{label}\ t \quad = \quad \mathit{from}$$

That is, if we label a tree and then append its list of labels with the list of unused labels, the result is the original list of fresh labels. This lemma is a natural generalisation of lemma 25.2 in our original proof, which stated that if $\mathit{label}\ t\ n = (t', n')$, then we have $n < n'$ and $\mathit{labels}\ t' = [n\,..\,n'-1]$. In order to formulate this result in a point-free manner, our first step is to consider the expression $(\mathit{labels} \times \mathit{id}) \circ \mathit{label}\ t$, which returns a pair comprising the labels of the tree, say $[n\,..\,n'-1]$, and the next fresh label, n'. But how can we specify that the output pair has this form? The approach taken above in lemma 25.5 is to append the list $[n\,..\,n'-1]$ to the infinite list from n', and assert that the resulting list is given by $\mathit{from}\ n$.

Proof: by equational reasoning, using two lemmas.

$$
\begin{array}{ll}
 & \mathit{append} \circ (\mathit{labels} \times \mathit{from}) \circ \mathit{label}\ t \\
= & \quad \{\ \text{products}\ \} \\
 & \mathit{append} \circ (\mathit{labels} \times \mathit{id}) \circ (\mathit{id} \times \mathit{from}) \circ \mathit{label}\ t \\
= & \quad \{\ \text{factorising } \mathit{label} \text{ (lemma 25.6)}\ \} \\
 & \mathit{append} \circ (\mathit{labels} \times \mathit{id}) \circ \mathit{label}'\ t \circ \mathit{from} \\
= & \quad \{\ \text{behaviour of } \mathit{label}' \text{ (lemma 25.7)}\ \} \\
 & \mathit{from}
\end{array}
$$

□

Note that the two lemmas used in this proof can be fused together to give a direct proof without the use of separate lemmas, but we find the separation of the proof into two parts in this manner to be more instructive. By analogy with program fission [3], the dual to program fusion, factorising a proof into component parts in this manner might be termed proof fission.

Lemma 25.6 (factorising *label*). *For all finite trees t:*

$$(\mathit{id} \times \mathit{from}) \circ \mathit{label}\ t \quad = \quad (\mathit{label}'\ t) \circ \mathit{from}$$

where

$$
\begin{array}{lcl}
label' & :: & Tree\ a \rightarrow ST\ [b]\ (Tree\ b) \\
label'\ (Leaf\ x) & = & [\![\ Leaf\ fetch\ ]\!] \\
label'\ (Node\ l\ r) & = & [\![\ Node\ (label'\ l)\ (label'\ r)\ ]\!] \\
fetch & :: & ST\ [a]\ a \\
fetch & = & \lambda xs \rightarrow (head\ xs, tail\ xs)
\end{array}
$$

The new function *label′* is defined in precisely the same manner as *label*, except that it utilises an infinite list of fresh labels as the internal state, rather than a single fresh label as previously, and consumes labels from this list one at a time using a new state transformer *fetch*. The lemma itself states that labelling a tree and then extending the next fresh label to an infinite list (by applying *from*) gives the same result as producing an infinite list of fresh labels and then applying *label′*.

Lemma 25.6 shows how *label* can be factorised into two parts, namely the generation of an infinite supply of fresh labels using *from* (the essential part of the computation, which is monomorphic), and the threading of this list through the tree using *label′* (the plumbing part, which is polymorphic.) Factorising *label* in this manner is both an interesting concept in its own right, and leads to a simple and natural proof of the behaviour of *label* (lemma 25.5).

Proof: by structural induction on *t*, using a number of lemmas about state transformers and basic functions, which are contained in the appendix.

Case: $t = Leaf\ x$

$$
\begin{array}{ll}
 & (id \times from) \circ label\ (Leaf\ x) \\
= & \quad \{\ \text{applying } label\ \} \\
 & (id \times from) \circ [\![\ Leaf\ fresh\ ]\!] \\
= & \quad \{\ \text{output state fusion (lemma 25.15)}\ \} \\
 & [\![\ Leaf\ ((id \times from) \circ fresh)\ ]\!] \\
= & \quad \{\ \text{relationship between } fresh \text{ and } fetch \text{ (lemma 25.18)}\ \} \\
 & [\![\ Leaf\ (fetch \circ from)\ ]\!] \\
= & \quad \{\ \text{input state fusion (lemma 25.11)}\ \} \\
 & [\![\ Leaf\ fetch\ ]\!] \circ from \\
= & \quad \{\ \text{unapplying } label'\ \} \\
 & label'\ (Leaf\ x) \circ from
\end{array}
$$

Case: $t = Node\ l\ r$

$$
\begin{array}{ll}
 & (id \times from) \circ label\ (Node\ l\ r) \\
= & \quad \{\ \text{applying } label\ \} \\
 & (id \times from) \circ [\![\ Node\ (label\ l)\ (label\ r)\ ]\!] \\
= & \quad \{\ \text{output state fusion (lemma 25.16)}\ \} \\
 & [\![\ Node\ (label\ l)\ ((id \times from) \circ label\ r)\ ]\!] \\
= & \quad \{\ \text{induction hypothesis for } r\ \} \\
 & [\![\ Node\ (label\ l)\ ((label'\ r) \circ from)\ ]\!]
\end{array}
$$

$$
\begin{array}{ll}
= & \quad \{ \text{ state shifting (lemma 25.17) } \} \\
& [\![\mathit{Node}\ ((\mathit{id} \times \mathit{from}) \circ \mathit{label}\ l)\ (\mathit{label}'\ r)]\!] \\
= & \quad \{ \text{ induction hypothesis for } l \} \\
& [\![\mathit{Node}\ (\mathit{label}'\ r \circ \mathit{from})\ (\mathit{label}'\ r)]\!] \\
= & \quad \{ \text{ input state fusion (lemma 25.12) } \} \\
& [\![\mathit{Node}\ (\mathit{label}'\ r)\ (\mathit{label}'\ r)]\!] \circ \mathit{from} \\
= & \quad \{ \text{ unapplying } \mathit{label}' \} \\
& \mathit{label}'\ (\mathit{Node}\ l\ r) \circ \mathit{from}
\end{array}
$$

□

There are two important points to note about this proof. First of all, it proceeds by simple equational reasoning, without the need to expand out the underlying definitions. And secondly, a number of intermediate steps appear to be type incorrect with respect to the $[\![\ ]\!]$ notation, by using arguments that are not state transformers. For example, in the base case the expression $(\mathit{id} \times \mathit{from}) \circ \mathit{fresh}$ has type $\mathit{Int} \to (\mathit{Int}, [\mathit{Int}])$, which is not a state transformer because the state changes from *Int* to $[\mathit{Int}]$. However, when we come to prove the state transformer lemmas themselves, we will find that such type incorrect reasoning is perfectly sound.

Lemma 25.7 (behaviour of label'). *For all finite trees t:*

$$\mathit{append} \circ (\mathit{labels}\ x\ \mathit{id}) \circ \mathit{label}'\ t \ = \ \mathit{id}$$

That is, if we apply label' to a tree and then append its labels to the unused labels, the result is the original list of fresh labels.

Proof: by structural induction on *t*, using a number of lemmas (see appendix).

Case: $t = \mathit{Leaf}\ x$

$$
\begin{array}{ll}
& \mathit{append} \circ (\mathit{labels} \times \mathit{id}) \circ \mathit{label}'\ (\mathit{Leaf}\ x) \\
= & \quad \{ \text{ applying } \mathit{label}' \} \\
& \mathit{append} \circ (\mathit{labels} \times \mathit{id}) \circ [\![\mathit{Leaf}\ \mathit{fetch}]\!] \\
= & \quad \{ \text{ bracket notation (lemma 25.9) } \} \\
& \mathit{append} \circ [\![\mathit{labels}\ [\![\mathit{Leaf}\ \mathit{fetch}]\!]]\!] \\
= & \quad \{ \text{ nested state fusion (lemma 25.13) } \} \\
& \mathit{append} \circ [\![(\mathit{labels} \circ \mathit{Leaf})\ \mathit{fetch}]\!] \\
= & \quad \{ \text{ behaviour of } \mathit{labels} \text{ on leaves (lemma 25.19) } \} \\
& \mathit{append} \circ [\![[-]\ \mathit{fetch}]\!] \\
= & \quad \{ \text{ property of } \mathit{append} \text{ (lemma 25.21) } \} \\
& \mathit{cons} \circ \mathit{fetch} \\
= & \quad \{ \text{ applying } \mathit{fetch} \} \\
& \mathit{cons} \circ \langle \mathit{head}, \mathit{tail} \rangle \\
= & \quad \{ \text{ property of } \mathit{cons} \} \\
& \mathit{id}
\end{array}
$$

Case: $t = \mathit{Node}\ l\ r$

$$
\begin{array}{ll}
 & \mathit{append} \circ (\mathit{labels} \times \mathit{id}) \circ \mathit{label}'\ (\mathit{Node}\ l\ r) \\
= & \quad \{ \text{ applying } \mathit{label}' \ \} \\
 & \mathit{append} \circ (\mathit{labels} \times \mathit{id}) \circ [\![\ \mathit{Node}\ (\mathit{label}'\ l)\ (\mathit{label}'\ r)\ ]\!] \\
= & \quad \{ \text{ bracket notation (lemma 25.10) } \} \\
 & \mathit{append} \circ [\![\ \mathit{labels}\ [\![\ \mathit{Node}\ (\mathit{label}'\ l)\ (\mathit{label}'\ r)\ ]\!]\ ]\!] \\
= & \quad \{ \text{ nested state fusion (lemma 25.14) } \} \\
 & \mathit{append} \circ [\![\ (\mathit{labels} \bullet \mathit{Node})\ (\mathit{label}'\ l)\ (\mathit{label}'\ r)\ ]\!] \\
= & \quad \{ \text{ behaviour of } \mathit{labels} \text{ on nodes (lemma 25.20) } \} \\
 & \mathit{append} \circ [\![\ (\mathit{curry}\ (\mathit{append} \circ (\mathit{labels} \times \mathit{labels})))\ (\mathit{label}'\ l)\ (\mathit{label}'\ r)\ ]\!] \\
= & \quad \{ \text{ property of } \mathit{append} \text{ (lemma 25.22) } \} \\
 & \mathit{append} \circ (\mathit{labels} \times (\mathit{append} \circ (\mathit{labels} \times \mathit{id}) \circ \mathit{labels}'\ r)) \circ \mathit{labels}'\ l \\
= & \quad \{ \text{ induction hypothesis for } r \ \} \\
 & \mathit{append} \circ (\mathit{labels} \times \mathit{id}) \circ \mathit{labels}'\ l \\
= & \quad \{ \text{ induction hypothesis for } l \ \} \\
 & \mathit{id}
\end{array}
$$

□

Lemma 25.8 (*from* produces no duplicates).

$$\mathit{nodups}\ \mathit{from}$$

This lemma is a natural generalisation of lemma 25.3 from our original proof, which stated that if $a \leqslant b$ then $\mathit{nodups}\ [a\,..\,b]$, with the finite interval $[a\,..\,b]$ generalised to an infinite list produced by the function *from*.

25.7 CONCLUSION AND FURTHER WORK

In this article we showed how a simple tree relabelling function can be proved correct using point-free equational reasoning, by identifying and exploiting basic properties of state in the context of applicative functors. Our proof relied on three ideas: factorising an effectful computation into essential and plumbing parts, generalising from finite to infinite lists of fresh labels, and relaxing the typing constraints on the applicative functor notation to accommodate internal changes in the type of the state. The first of these ideas appears to be novel, while the latter two are instances of the familiar mathematical practice of moving to a more general framework for the purposes of proofs, such as moving from two to three dimensional space in geometry, or from real to complex numbers in analysis.

There are a number of possible directions for further work. First of all, our current definition of the relabelling function uses explicit recursion, and it would be interesting to see if there is any benefit to be gained from using a more structured approach to recursion, such as that provided by applicative traversals [5] or monadic folds [7]. Secondly, we would like to propose the relabelling problems as a simple but instructive case study for other approaches to reasoning about effects, such as rigid induction [2] and Hoare type theory [9]. And finally, it is important to consider how our approach can be adapted both to other effects, and to more substantial programming examples. In this direction, we have proved in a similar

manner an analogous factorisation result for a relabelling function in which each leaf is labelled by its lexicographic path from the root of the tree, which suggests that our approach may indeed be more widely applicable.

Acknowledgements

We would like to thank Thorsten Altenkirch, Martin Escardó, Hugo Herbelin, Conor McBride, James McKinna, Nicolas Oury, Wouter Swierstra, and the other FP lab members in Nottingham for useful comments, and Galois for funding a sabbatical visit during which part of this article was written.

APPENDIX: OTHER PROPERTIES

This appendix presents the properties of state transformers and basic functions that were used to prove our main correctness result. The first two lemmas show how the bracket notation for applicative functors can be expanded out in a point-free manner for the case of functions with one or two stateful arguments.

Lemma 25.9 (bracket notation). *For all* $f :: a \to b$ *and* $x :: ST\ s\ a$*:*

$$[\![f\ x]\!] \quad = \quad (f \times id) \circ x$$

Lemma 25.10 (bracket notation). *For all* $f :: a \to b \to c$, $x :: ST\ s\ a$, *and* $y :: ST\ s\ b$*:*

$$[\![f\ x\ y]\!] \quad = \quad ((uncurry\ f) \times id) \circ assoc \circ (id \times y) \circ x$$

The next two lemmas show how a function that modifies the input state, including potentially its type, can be fused with the bracket notation.

Lemma 25.11 (input state fusion). *For all* $f :: a \to b$, $x :: ST\ s'\ a$, *and* $g :: s \to s'$*:*

$$[\![f\ x]\!] \circ g \quad = \quad [\![f\ (x \circ g)]\!]$$

Note that in the right-side of this equation, the expression $x \circ g$ has type $s \to (a, s')$, which is not a state transformer because the type of the state changes from s to s', and hence the use of the applicative functor notation is not type correct. However, when the notation is expanded out in the proof, we find that the resulting expression is indeed type correct, which formally justifies our abuse of the bracket notation. A similar comment holds for a number of other lemmas in this section.

Lemma 25.12 (input state fusion).

For all $f :: a \to b \to c$, $x :: ST\ s'\ a$, $y :: ST\ s'\ b$, *and* $g :: s \to s'$*:*

$$[\![f\ x\ y]\!] \circ g \quad = \quad [\![f\ (x \circ g)\ y]\!]$$

The next two lemmas show how two nested stateful computations can be fused together into a single stateful computation.

Lemma 25.13 (nested state fusion). *For all $f :: b \to c$, $g :: a \to b$, and $x :: ST\ s\ a$:*

$$[\![f\ [\![g\ x]\!]]\!] \quad = \quad [\![(f \circ g)\ x]\!]$$

Lemma 25.14 (nested state fusion).

For all $f :: c \to d$, $g :: a \to b \to c$, $x :: ST\ s\ a$, and $y :: ST\ s\ b$:

$$[\![f\ [\![g\ x\ y]\!]]\!] \quad = \quad [\![(f \bullet g)\ x\ y]\!]$$

The next two lemmas show how a function that modifies the output state, including potentially its type, can be fused with the bracket notation.

Lemma 25.15 (output state fusion). *For all $f :: s \to s'$, $g :: a \to b$, and $x :: ST\ s\ a$:*

$$(id \times f) \circ [\![g\ x]\!] \quad = \quad [\![g\ ((id \times f) \circ x)]\!]$$

Lemma 25.16 (output state fusion).

For all $f :: s \to s'$, $g :: a \to b \to c$, $x :: ST\ s\ a$ and $y :: ST\ s\ b$:

$$(id \times f) \circ [\![g\ x\ y]\!] \quad = \quad [\![g\ x\ ((id \times f) \circ y)]\!]$$

The next lemma shows how a function that modifies the intermediate state in a two argument computation can be shifted between the two arguments.

Lemma 25.17 (state shifting).

For all $f :: a \to b \to c$, $x :: ST\ s\ a$, $y :: ST\ s'\ b$ and $g :: s \to s'$:

$$[\![f\ ((id \times g) \circ x)\ y]\!] \quad = \quad [\![f\ x\ (y \circ g)]\!]$$

The remaining lemmas concern the behaviour of the functions *fresh*, *fetch*, *labels* and *append* that are used in the formulation of our correctness result.

Lemma 25.18 (relationship between *fresh* and *fetch*).

$$(id \times from) \circ fresh \quad = \quad fetch \circ from$$

Lemma 25.19 (behaviour of *labels* on leaves).

$$labels \circ Leaf \quad = \quad [-]$$

Lemma 25.20 (behaviour of *labels* on nodes).

$$labels \bullet Node \quad = \quad curry\,(append \circ (labels \times labels))$$

Lemma 25.21 (property of *append*). *For all* $x :: ST\ [a]\ a$*:*

$$append \circ [\![\ [-]\ x\ ]\!] \quad = \quad cons \circ x$$

Lemma 25.22 (property of *append*).

For all $f :: a \to [c]$, $g :: b \to [c]$, $h :: ST\ [c]\ a$, *and* $i :: ST\ [c]\ b$*:*

$$\begin{array}{l} \quad append \circ [\![\ (curry\ (append \circ (f \times g)))\ h\ i\ ]\!] \\ = \\ \quad append \circ (f \times (append \circ (g \times id) \circ i)) \circ h \end{array}$$

REFERENCES

[1] R. Bird. *Introduction to Functional Programming using Haskell (second edition)*. Prentice Hall, 1998.

[2] A. Filinski and K. Stovring. Inductive Reasoning About Effectful Data Types. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, Freiburg, Germany, 2007.

[3] J. Gibbons. Fission for Program Comprehension. In T. Uustalu, editor, *Mathematics of Program Construction*, volume 4014 of *Lecture Notes in Computer Science*, pages 162–179. Springer-Verlag, 2006.

[4] J. Hughes. Generalising Monads to Arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

[5] C. McBride and R. Paterson. Applicative Programming With Effects. *Journal of Functional Programming*, 18(1), 2008.

[6] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In J. Hughes, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture*, number 523 in LNCS. Springer-Verlag, 1991.

[7] E. Meijer and J. Jeuring. Merging Monads and Folds for Functional Programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LCNS*. Springer-Verlag, 1995.

[8] E. Moggi. Computation Lambda-Calculus and Monads. In *Proc. IEEE Symposium on Logic in Computer Science*, 1989.

[9] A. Nanevski, G. Morrisett, and L. Birkedal. Hoare Type Theory, Polymorphism and Separation. *Journal of Functional Programming*. To appear.

[10] P. Wadler. Monads for Functional Programming. In M. Broy, editor, *Proceedings of the Marktoberdorf Summer School on Program Design Calculi*. Springer–Verlag, Aug. 1992.

[11] P. Wadler. The Essence of Functional Programming. In *Proc. Principles of Programming Languages*, 1992.