Multi-traversal tree decoration in a functional setting: monads versus bindings

Maarten Pennings

RUU-CS-93-46
December 1993

Utrecht University
Department of Computer Science
Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : + 31 - 30 - 531454

Multi-traversal tree decoration in a functional setting: monads versus bindings

Maarten Pennings*

Department of Computer Science, Utrecht University,
P.O. Box 80089, 3508 TB Utrecht, The Netherlands.
E-mail: maarten@cs.ruu.nl

Abstract

In this paper we consider the problem of decorating trees. We examine the special case where multiple traversals over a tree are needed for full decoration. We compare three functional approaches that do not recompute any values in successive traversals: a circular program to short-circuit multiple passes, a program with bindings from one traversal to the next to explicitly pass values and finally a monadic program that records a suitable state. Given our criteria, avoiding lazy evaluation and the ability to [illegible] the traversals, bindings seem to have the most advantages.

1 Introduction

This paper discusses a special form of tree decorations that requires multiple traversals over a tree. Multiple traversals are needed if "global" information must first be gathered before it can be used: information flows from one traversal to the next.

Tree decoration in imperative programming is straightforward: attributes are stored in the tree. Therefore, no complications arise when a later traversal refers to values computed in earlier ones. On the other hand, a functional program seems appropriate too, since decoration has a functional character: a tree is passed some input values for which it computes some output values. However, in a functional setting, values can not be attached to tree nodes. Therefore, if the just mentioned *inter-traversal communications* occur, we must provide for a mechanism to deal with them.

A typical example of this class are the passes of a compiler. Compilers are in essence parse tree decorators. Type checking, type coercion, [illegible] scopes and code generation are just a view tasks carried out by a compiler. Multiple traversals of the parse tree are practically inevitable and intra-traversal communication [illegible].

We solve the intra-traversal communication problem using so called *bindings* [Pen92]. Bindings are data structures that are constructed while the tree is being traversed. They contain the values that are needed for subsequent traversals. We have also investigated two

*This research was supported by the Foundation for Computer Science (SION) of the Netherlands Organization for Scientific Research (NWO) under grant [illegible].

ISSN: 0924-3275

```
rtips :: (α → β) → Tree α → [β] → [β]
rtips n (Tip a) rs          =  i : rs where i = n a
rtips n (Fork l r) rs       =  rs''
                      where rs' = rtips n l rs
                            rs'' = rtips n r rs'
bmatch :: (α → β) → Tree α → [β] → ⟨[β], Bool⟩
match n (Tip a) (v : vs)    =  ⟨vs, v ≡ i⟩ where i = n a
match n (Fork l r) vs       =  ⟨vs'', l' ∧ r'⟩
                      where ⟨vs', l'⟩ = match n l vs
                            ⟨vs'', r'⟩ = match n r vs'
palin :: (α → β) → Tree α → Bool
palin n t                   =  t' where ⟨[], t'⟩ = match n t (rtips n t [])   .
```

Fig 3: An inefficient palindrome recognizer for termed trees

3 A circular program

The function *palin* from Fig. 3 has one major flaw. Each tip is normalized twice: once in the first pass to cons the normalized term to the tip-list and once in the second pass to compare it with an element in the tip-list. Hence, *palin* suffers from inefficiency due to recomputation.

In this section we will show a function that circumvents recomputation by giving it a circular definition [Bir84]. This solution comes close to our requirements although it has some shortcomings. Implementors of attribute grammar evaluators prefer non-lazy evaluation so that the system is easier to implement and faster in execution. On a more theoretical bases, the solution also has a shortcoming. It is not memoizable. Evaluation is essentially lazy, so we can not first compute the arguments in order to check the memo table. Lazy memoing as proposed by Hughes [Hug85] does not seem appropriate either.

There are two general methods to obtain a circular program. Firstly, one can use the rewriting technique from bird [Bir84] or, secondly, one can use a mapping to and from an attribute grammar as described in [KS87] or [Kui89, pp. 83-95]. However, since the functions for the first traversal (*rtips*) and the second (*match*) have the same pattern structure, we observe that we can bypass these elaborate techniques by simply merging the two function definitions into one

$$rtm\ n\ t\ rs\ vs = \langle rs', vs', t' \rangle \textbf{ where } rs' = rtips\ n\ t\ rs;\ \langle vs', t' \rangle = match\ n\ t\ vs \quad .$$

The function *cpalin* as defined in Fig. 4 is the result. As Bird noted in [Bir84], "one has to be careful to avoid demanding information about an argument, either through pattern matching on the left hand side or an explicit conditional on the right, when such information can be delayed or avoided altogether." We have such a case at hand, namely the pattern $(v : vs)$ which is a parameter for the "second traversal". Note that we have used an irrefutable pattern (prefix ~) which is just syntactic sugar offered by GOFER to avoid writing the less clear but lazy constructs *head* and *tail*.

Since the traversing functions now also construct respectively destruct a binding, their types have changed. This is reflected by their headers

$$
\begin{aligned}
&\mathit{rtips} :: (\alpha \to \beta) \to \text{Tree } \alpha \to [\beta] \to \langle [\beta], \text{Tree}^{\mathit{rtips}\to\mathit{match}}\ \beta\rangle \\
&\mathit{match} :: (\alpha \to \beta) \to \text{Tree } \alpha \to [\beta] \to \text{Tree}^{\mathit{rtips}\to\mathit{match}}\ \beta \to \langle [\beta], \text{Bool}\rangle \quad .
\end{aligned}
$$

After this transformation, we notice that in a **Fork** node both sons return a binding during the first traversal. This binding should be passed back to them in the second traversal. Hence, a **Fork** node must put the bindings of its sons in a binding. We conclude that a **Tip** node requires a binding of type β, whereas a **Fork** node requires a binding of two bindings, both of type $(\text{Tree}^{\mathit{rtips}\to\mathit{match}}\ \beta)$. These are disjunct instances of the same type $(\text{Tree}^{\mathit{rtips}\to\mathit{match}}\ \beta)$

$$
\begin{aligned}
\textbf{data } \text{Tree}^{\mathit{rtips}\to\mathit{match}}\ \beta \ &= \ \textbf{Tip}^{\mathit{rtips}\to\mathit{match}}\ \beta \\
&\mid \ \textbf{Fork}^{\mathit{rtips}\to\mathit{match}}\ (\text{Tree}^{\mathit{rtips}\to\mathit{match}}\ \beta)\ (\text{Tree}^{\mathit{rtips}\to\mathit{match}}\ \beta) \quad .
\end{aligned}
$$

In Fig. 5 the function *bpalin* is given. It is a palindrome recognizer for termed trees that uses bindings to circumvent recomputation of the normal forms. Function *bmatch* makes use of double pattern matching. It will never abort, since *brtips* constructs the binding in synchronization with the tree.

$$
\begin{aligned}
&\mathit{brtips} :: (\alpha \to \beta) \to \text{Tree } \alpha \to [\beta] \to \langle [\beta], \text{Tree}^{\mathit{rtips}\to\mathit{match}}\ \beta\rangle \\
&\mathit{brtips}\ n\ (\textbf{Tip}\ a)\ rs \ = \ \langle i : rs, \textbf{Tip}^{\mathit{rtips}\to\mathit{match}}\ i\rangle \ \textbf{where}\ i = n\ a \\
&\mathit{brtips}\ n\ (\textbf{Fork}\ l\ r)\ rs \ = \ \langle rs'', \textbf{Fork}^{\mathit{rtips}\to\mathit{match}}\ l_b\ r_b\rangle \\
&\qquad\qquad \textbf{where}\ \langle rs', l_b\rangle = \mathit{brtips}\ n\ l\ rs \\
&\qquad\qquad\qquad\ \ \langle rs'', r_b\rangle = \mathit{brtips}\ n\ r\ rs'
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{match} :: (\alpha \to \beta) \to \text{Tree } \alpha \to [\beta] \to \text{Tree}^{\mathit{rtips}\to\mathit{match}}\ \beta \to \langle [\beta], \text{Bool}\rangle \\
&\mathit{bmatch}\ n\ (\textbf{Tip}\ a)\ (v : vs)\ (\textbf{Tip}^{\mathit{rtips}\to\mathit{match}}\ i) \ = \ \langle vs, v \equiv i\rangle \\
&\mathit{bmatch}\ n\ (\textbf{Fork}\ l\ r)\ vs\ (\textbf{Fork}^{\mathit{rtips}\to\mathit{match}}\ l_b\ r_b) \ = \ \langle vs'', l' \wedge r'\rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where}\ \langle vs', l'\rangle = \mathit{bmatch}\ n\ l\ vs\ l_b \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \langle vs'', r'\rangle = \mathit{bmatch}\ n\ r\ vs'\ r_b
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{bpalin} :: (\alpha \to \beta) \to \text{Tree } \alpha \to \text{Bool} \\
&\mathit{bpalin}\ n\ t \ = \ t' \\
&\qquad \textbf{where}\ \langle rs, t_b\rangle = \mathit{brtips}\ n\ t\ [] \\
&\qquad\qquad\ \ \langle [], t'\rangle = \mathit{bmatch}\ n\ t\ rs\ t_b
\end{aligned}
$$

Fig 5: An palindrome recognizer with bindings

4.2 Bindings in general

In the previous paragraph we dealt with the running example. In this paragraph we will deal with the general case: mutual recursive data types with a varying number of traversals.

$$
\begin{array}{lll}
trav_N^1 & :: & \underline{X} \to \underline{ni1} \to \underline{ns1} \\
trav_N^1\ (\mathbf{c}\ X)\ ni1 & = & ns1 \\
\qquad\qquad & \textbf{where} & xi1 = f1\ ni1 \\
\qquad\qquad & & xs1 = trav_X^1\ X\ xi1 \\
\qquad\qquad & & ns1 = f2\ xs1 \quad .
\end{array}
$$

We see in Fig. 6 that N is visited three times. In the first visit to N, son X will be visited for the first time. In the second visit to N son X will be visited for the second and third time; we see the beginning of a visit border. In the third visit to N the fourth visit to X takes place. This figure also shows the intra-traversal-communications. They are flagged with a "lightning-arrow" ($\lightning$) next to the border they cross.

First, we extend the traversing functions with bindings. A tree of type N is visited three times, so there will be three bindings: $N^{1\to2}$, $N^{1\to3}$ and $N^{2\to3}$. On the other hand, X will be traversed four times, so there are six bindings: $X^{1\to2}$, $X^{1\to3}$ and $X^{1\to4}$ from traversal 1, $X^{2\to3}$ and $X^{2\to4}$ from traversal 2 and finally $X^{3\to4}$ from the third traversal. This transformation is shown in Fig. 7. The above presented function now has the following type

$$
trav_N^1 :: \underline{X} \to \underline{ni1} \to \langle \underline{ns1}, N^{1\to2}, N^{1\to3} \rangle \quad .
$$

The final transition from Fig. 7 to Fig. 8 determines the shape of the binding constructors. We will explain this in more details. First of all, some *values* are bound: $ni1$ is defined in the first traversal, but it is used in the second *as well as* in the third. Therefore, it will be bound by $N^{1\to2}$ as well as by $N^{1\to3}$. The former also bind $xs1$. Secondly, *bindings for sons* must be bound. We perform the usual define/usage analyses and note that $X^{1\to2}$ and $X^{1\to3}$ are defined in traversal 1, but they are both needed in traversal 2 since the second and third visit to X take place there. Likewise, $X^{1\to4}$ is bound by $N^{1\to3}$. We have now fully described the bindings from the first traversal. There is one binding from the second traversal ($N^{2\to3}$), it only binds the bindings for its sons. The following data types are the result

$$
\begin{array}{l}
\textbf{data}\ N^{1\to2} = \mathbf{c}^{1\to2}\ \underline{ni1}\ \underline{xs1}\ X^{1\to2}\ X^{1\to3} \mid \ldots \\
\textbf{data}\ N^{1\to3} = \mathbf{c}^{1\to3}\ \underline{ni1}\ X^{1\to4} \mid \ldots \\
\textbf{data}\ N^{2\to3} = \mathbf{c}^{2\to3}\ X^{2\to4}\ X^{3\to4} \mid \ldots
\end{array}
$$

Other binding constructors on these types exist (as the dots suggest). They are defined by other constructors on N though. There is one last thing that is worth noticing in Fig. 8. The binding for son X from visit 2 to visit 3, $X^{2\to3}$ is taken care of immediately during the second traversal to N.

4.4 Properties of bindings

Bindings closely follow the structure of the tree. Furthermore, they contain values from all over the tree, and the more traversals have take place, the more bindings have been computed. Nevertheless, bindings should not be confused with partially decorated trees that are threaded through the traversal code. When the first traversal finishes, it returns a binding for the second traversal. However, this binding only contains the values needed in the second traversal, not the values needed for the third (those are stored separately in a $1 \to 3$ binding). Of course, we

Decoration starts with a completely undecorated tree. The following function creates such a tree.

$$
\begin{array}{lcl}
\mathit{emptyA} :: \text{Tree } \alpha \to \text{Atree } \alpha\ \beta & & \\
\mathit{emptyA}\ (\mathbf{Tip}\ a) & = & \mathbf{Atip\ Undef}\ a \\
\mathit{emptyA}\ (\mathbf{Fork}\ l\ r) & = & \mathbf{Afork}\ (\mathit{emptyA}\ l)\ (\mathit{emptyA}\ r)
\end{array}
$$

We will need three operations on decoratable trees: recording a value, retrieving a value and inspecting the applied constructor. For the latter, we first introduce a data type used to return the result.

$$
\begin{array}{lcl}
\mathit{decoA} :: \beta \to \text{Atree } \alpha\ \beta \to \text{Atree } \alpha\ \beta & & \\
\mathit{decoA}\ \bar{b}\ (\mathbf{Atip}\ b\ a) & = & \mathbf{Atip}\ (\mathbf{Ok}\ \bar{b})\ a \\
\mathit{getA} :: \text{Atree } \alpha\ \beta \to \beta & & \\
\mathit{getA}\ (\mathbf{Atip}\ (\mathbf{Ok}\ b)\ a) & = & b \\
\mathbf{data}\ \text{Conses } \alpha & = & \mathbf{CTip}\ \alpha \mid \mathbf{CFork} \\
\mathit{consA} :: \text{Atree } \alpha\ \beta \to \text{Conses } \alpha & & \\
\mathit{consA}\ (\mathbf{Atip}\ b\ a) & = & \mathbf{CTip}\ a \\
\mathit{consA}\ (\mathbf{Afork}\ l\ r) & = & \mathbf{CFork}
\end{array}
$$

5.2 Walkable trees

The major problem to tackle is how to implement a "walkable" tree. This is necessary since the current location is part of the state. We will represent a walkable tree by a two tuple. The first component represents the current subtree. The second component is a list of ancestors-functions. An ancestor-function returns the tree representing the father, when passed its missing son as argument. Thus the original tree represented by the walkable tree $\langle t_n, [f_{n-1}, f_{n-2}, \ldots, f_0] \rangle$ is $(f_0 \ldots (f_{n-2}\ (f_{n-1}\ t_n)) \ldots)$.

$$
\begin{array}{lcl}
\mathbf{type}\ \text{Wtree } \alpha\ \beta & = & \langle \text{Atree } \alpha\ \beta, [\text{Atree } \alpha\ \beta] \rangle \\
\mathit{downlW}, \mathit{downrw}, \mathit{upW} :: \text{Wtree } \alpha\ \beta \to \text{Wtree } \alpha\ \beta & & \\
\mathit{downlW}\ \langle \mathbf{Afork}\ l\ r, h \rangle & = & \langle l, (\lambda l'.\mathbf{Afork}\ l'\ r) : h \rangle \\
\mathit{downrW}\ \langle \mathbf{Afork}\ l\ r, h \rangle & = & \langle r, (\lambda r'.\mathbf{Afork}\ l\ r') : h \rangle \\
\mathit{upW}\ \langle s, f : h \rangle & = & \langle f\ s, h \rangle
\end{array}
$$

Of course, the functions on Atree must be ported to Wtree.

$$
\begin{array}{lcl}
\mathit{emptyW}\ t & = & \langle \mathit{emptyA}\ t, [] \rangle \\
\mathit{decoW}\ b\ \langle t, h \rangle & = & \langle \mathit{decoA}\ b\ t, h \rangle \\
\mathit{getW}\ \langle t, h \rangle & = & \mathit{getA}\ t \\
\mathit{consW}\ \langle t, h \rangle & = & \mathit{consA}\ t
\end{array}
$$

5.3 The monad

In a pure functional language, state may be mimicked by introducing a type to represent computations that act on state. In the previous two subsections, we have defined a state that satisfies our purposes, namely (Wtree $\alpha\ \beta$).

$$
\mathbf{type}\ M\ \alpha\ \beta\ \gamma = \text{Wtree } \alpha\ \beta \to \langle \gamma, \text{Wtree } \alpha\ \beta \rangle
$$

Now, the state monad ($M\ \alpha\ \beta\ \gamma$) is a function that accepts an initial state (a partially decorated walkable tree), and returns the computed value —that may depend on the initial state— paired with the final state.

With a (state) monad we associate three functions. One that takes a value into a monad (*unitM*), one that applies a monadic function in a monad ($\star$) and one that takes a value out of a monad (*startM*). We use the quoting convention 'f' for an infix application of f.

$$
\begin{aligned}
&\mathit{unitM} :: \gamma \to M\ \alpha\ \beta\ \gamma \\
&\mathit{unitM}\ c \quad = \quad \lambda w.\langle c, w\rangle \\
&(\star) :: M\ \alpha\ \beta\ \gamma \to (\gamma \to M\ \alpha\ \beta\ \delta) \to M\ \alpha\ \beta\ \delta \\
&x \star f \quad = \quad \lambda w.\ \textbf{let}\ \langle c, w'\rangle = x\ w\ \textbf{in}\ f\ c\ w' \\
&\mathit{startM} :: M\ \alpha\ \beta\ \gamma \to \text{Wtree}\ \alpha\ \beta \to \gamma \\
&s\ \text{`}\mathit{startM}\text{`}\ w \quad = \quad c\ \textbf{where}\ \langle c, w'\rangle = s\ w
\end{aligned}
$$

What remains are the operations on the state. Two of them "inspect" the state (*consM* and *getM*) and the other four (*upM*, *downlM*, *downrM* and *decoM*) only alter the state. The value-component returned by the latter is therefore not of interest to us. In the imperative language C the function would by of type `void`, indicating that its purpose lies in a side effect. We define the type One for this.

$$
\begin{aligned}
&\textbf{data}\ \text{One} \quad = \quad \textbf{One} \\
&\mathit{downlM}, \mathit{downrm}, \mathit{upM} :: M\ \alpha\ \beta\ \textbf{One} \\
&\mathit{downlM} \quad = \quad \lambda w.\langle \textbf{One}, \mathit{downlW}\ w\rangle \\
&\mathit{downrM} \quad = \quad \lambda w.\langle \textbf{One}, \mathit{downrW}\ w\rangle \\
&\mathit{upM} \quad = \quad \lambda w.\langle \textbf{One}, \mathit{upW}\ w\rangle \\
&\mathit{decoW} :: \beta \to M\ \alpha\ \beta\ \textbf{One} \\
&\mathit{decoW}\ b \quad = \quad \lambda w.\langle \textbf{One}, \mathit{decoW}\ b\ w\rangle
\end{aligned}
$$

The other two functions do have a useful result, and no side effect.

$$
\begin{aligned}
&\mathit{getM} :: M\ \alpha\ \beta\ \beta \\
&\mathit{getM} \quad = \quad \lambda w.\langle \mathit{getW}\ w, w\rangle \\
&\mathit{consM} :: M\ \alpha\ \beta\ (\text{Conses}\ \alpha) \\
&\mathit{consM} \quad = \quad \lambda w.\langle \mathit{consW}\ w, w\rangle
\end{aligned}
$$

5.4 Using the monad

In this final section we will convert the program from Fig. 3 into monadic form. The result type must be augmented from γ to ($M\ \alpha\ \beta\ \gamma$). Furthermore, the (Tree α) parameter will be dropped. The result of this transformation is depicted in Fig. 9. The two traversals are coordinated by the following function.

$$
\begin{aligned}
&\mathit{mpalin'} :: (\alpha \to \beta) \to M\ \alpha\ \beta\ \text{Bool} \\
&\mathit{mpalin'}\ n \quad = \quad \mathit{mrtips}\ n\ [] \star \lambda rs.\mathit{mmatch}\ rs \star \lambda\langle \_, t\rangle.\mathit{unitM}\ t
\end{aligned}
$$