bindings we can still benefit from strong typing. Not only can we still use a strongly typed tree, but we have the additional type information of the bindings themselves as well. This is especially useful when the functional program is generated, as in our case.

Finally, bindings give a better incremental behavior when memoing the traversing functions. The reason for this is that, unlike the monad, a binding only records information that is useful for the given traversal. Values for other traversals are in other bindings.

References

[Bir84] R.S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, **21**:239–250, 1984.

[HF92] Paul Hudak and Joseph H. Fasel. A gentle introduction to haskell. *ACM SIGPLAN Notices, Haskell special issue*, **27**(5), may 1992.

[HPJW+92] P. Hudak, S. Peyton-Jones, P. Wadler, et al. Report on the programming language haskell, a non-strict purely functional language (version 1.2). *ACM SIGPLAN Notices, Haskell special issue*, **27**(5), may 1992.

[Hug85] John Hughes. Lazy memo-functions. In Jean-Pierre Jouannaud, editor, *Proceedings of the IFIP Conference on Functional Programming Languages and Computer Architecture*, volume **201** of Lecture Notes in Computer Science, pages 129–146. Springer-Verlag, September 1985.

[Jon91] Mark P. Jones. *Introduction to Gofer 2.20.* Oxford Programming Research Group, November 1991.

[KS87] M.F. Kuiper and S.D. Swierstra. Using attribute grammars to derive efficient functional programs. *Computing Science in the Netherlands CSN '87*, November 1987.

[Kui89] Matthijs F. Kuiper. *Parallel Attribute Evaluation.* PhD thesis, Utrecht University, Department of Computer Science, Utrecht, The Netherlands, November 1989.

[Pen92] Maarten Pennings. *Bindings: their definition and their type.* (a personal note), April 1992.

[Vog93] Harald Vogt. *Higher order Attribute Grammars.* PhD thesis, Utrecht University, Department of Computer Science, Utrecht, The Netherlands, February 1993.

[Wad90] Philip Wadler. Comprehending monads. In *Conference on Lisp and Functional Programming*, Nice, France, June 1990. ACM.

[Wad92] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, January 1992.

[Wad93] Philip Wadler. Monads for functional programming. In Manfred Broy, editor, *Program Design Calculi*, volume **118** of Nato ASI Series F: Computer and Systems Sciences, pages 233–264. Springer-Verlag, 1993.

Since λ-abstraction has a higher priority than the monadic bind $\star$, *mpalin n* should be read as $(mrtips\ n\ [] \star (\lambda rs.mmatch\ rs \star (\lambda\langle \_, t\rangle.unitM\ t)))$. The reader may wonder where the tree is left in the definition for *mpalin'*. This function returns a *state transformer*, that must still be passed a tree. We use the top level function *mpalin* for that. This function, given below, completely hides the use of the monad. Internally it is based on *startM* that takes a value out of the monad. The computation of the initial state, a completely undecorated tree, is taken care of by *emptyW*.

$$
\begin{array}{l}
mpalin :: (\alpha \to \beta) \to \text{Tree } \alpha \to \text{Bool} \\
mpalin\ n\ t \quad = \quad mpalin'\ n\ `startM`\ (emptyW\ t) \quad .
\end{array}
$$

```
mrtips :: (α → β) → [β] → M α β [β]
mrtips n rs =
  consM ⋆ λc.
  case c of
    CTip a → unitM (n a)      ⋆ λi.
             decoM i          ⋆ λOne.
             unitM (i : rs)
    CFork  → downlM           ⋆ λOne.
             mrtips n rs      ⋆ λrs'.
             upM              ⋆ λOne.
             downrM           ⋆ λOne.
             mrtips n rs'     ⋆ λrs''.
             upM              ⋆ λOne.
             unitM rs''
```

```
mmatch :: [β] → M α β ⟨[β], Bool⟩
mmatch (v : vs) =
  consM ⋆ λc.
  case c of
    CTip a → getM                ⋆ λi.
             unitM ⟨vs, v ≡ i⟩
    CFork  → downlM              ⋆ λOne.
             mmatch (v : vs)     ⋆ λ⟨vs', l'⟩.
             upM                 ⋆ λOne.
             downrM              ⋆ λOne.
             mmatch vs'          ⋆ λ⟨vs'', r'⟩.
             upM                 ⋆ λOne.
             unitM ⟨vs'', l' ∧ r'⟩
```

Fig 9: A palindrome recognizer using a monad

As one can see from rewriting our running example into monad form, defining a suitable state requires a lot of additional data structures and functions. We will not describe the general case. However, we want to point out one detail. If the tree is constructed from mutual recursive data types, it is even harder to define a walkable tree. The reason for this lies in the type-system. The first component of the two tuple representing a walkable tree represents the current subtree. Hence, it could be any of the types occurring in the tree. Something similar holds for the second component, the list of ancestor functions. This means, that one has to "weaken" the types of the tree to one single type.

6 Conclusions

In this paper, we have described three ways to solve the intra-traversal-communication problem. The first solution, a circular program, is elegant and compact. However, being lazy, it fails to satisfy the memoizability constraint.

The other two solutions, bindings and monads, both fulfill all requirements. For our needs, bindings are preferable. The reason for this is three-fold. First of all, bindings require less additional functions and data types, and they are easier to understand. Furthermore, with

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.

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

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.

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

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

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

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.

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

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 $\alpha\ \beta$).

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

could merge those bindings so that they would mimic partially decorated trees. The number of bindings would then be linear in the number of traversals instead of quadratic. However, by not merging them we acquire better incremental behavior when we memo the traversing functions

Bindings may be empty. That is to say, the mutual recursive definitions of the bindings is such that for a particular binding, the (infinite) set of all producible terms contains no term that binds a (non-binding) value. Although this set is infinite, we can decide in polynominal time whether this situation occurs. The algorithm we have in mind is a simple transitive closure of the graph whose vertices are formed by the binding values and the (non-binding) values and whose edges are induced by the binding constructors. For example, the definition **data** $N^{1\to 2} = \mathbf{c}^{1\to 2}\ \underline{ni1}\ \underline{xs1}\ X^{1\to 2}\ X^{1\to 3}$ from above yields four edges: $(\underline{ni1} \longrightarrow N^{1\to 2})$, $(\underline{xs1} \longrightarrow N^{1\to 2})$, $(X^{1\to 2} \longrightarrow N^{1\to 2})$ and $(X^{1\to 3} \longrightarrow N^{1\to 2})$. Each binding vertex that is not reachable by a non-binding vertex is guaranteed to never bind anything useful, that is to say to never bind any (non-binding) value. Such bindings may be removed from the code.

We have nearly finished writing a compiler generator, the LRC-processor, that actually uses the binding scheme discussed in this section to by-pass side-effects of parse tree decoration. This technique allows us to memoize tree decoration. In other words, the generated compiler is incremental.

5 Monad

Tree decoration in imperative programming is straightforward: attributes are stored in the tree. We will use a monad [Wad90, Wad92, Wad93], to mimic the imperative style of decorating trees. The monad approach does not require much mental labour, but we need a large amount of additional data types and functions as will be illustrated.

Our task simply is to define an state mechanism. An appropriate state records a tree in which some attributes already have a value, and it records a current position in the tree. The rest of this section gives, in three steps, a formal description to handle states. First we define partially decorated trees. Next we define a data type that allows us to "walk" around in these trees so that we may record and retrieve attributes at correct locations. Thirdly we introduce a monad to hide this state mechanism.

5.1 Partially decorated trees

While traversing the tree, more and more attributes are computed. Therefore we speak about *partially* decorated trees. In order not to complicate matters too much, we will not record *all* attributes. Only those that are needed, i.e. the normalized tips, will be recorded.

$$
\begin{array}{lcl}
\textbf{data}\ U\ \beta & = & \textbf{Ok}\ \beta \mid \textbf{Undef} \\
\textbf{data}\ \text{Atree}\ \alpha\ \beta & = & \textbf{Atip}\ (U\ \beta)\ \alpha \\
 & \mid & \textbf{Afork}\ (\text{Atree}\ \alpha\ \beta)\ (\text{Atree}\ \alpha\ \beta)
\end{array}
$$

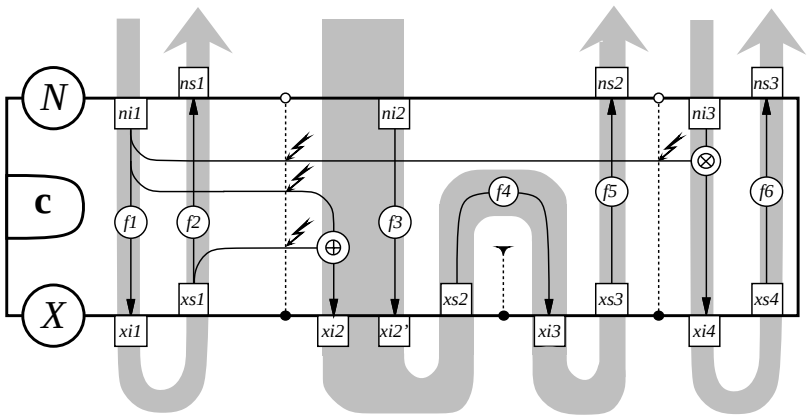

Fig 6: Dataflow and intra-traversal-communication overview

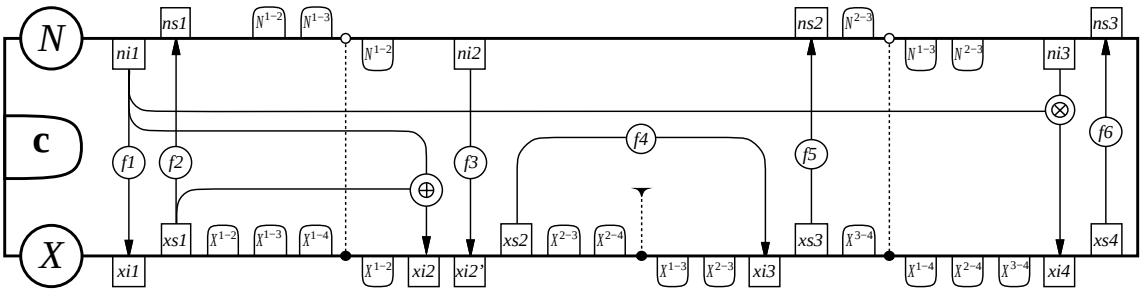

Fig 7: Dataflow extended with bindings

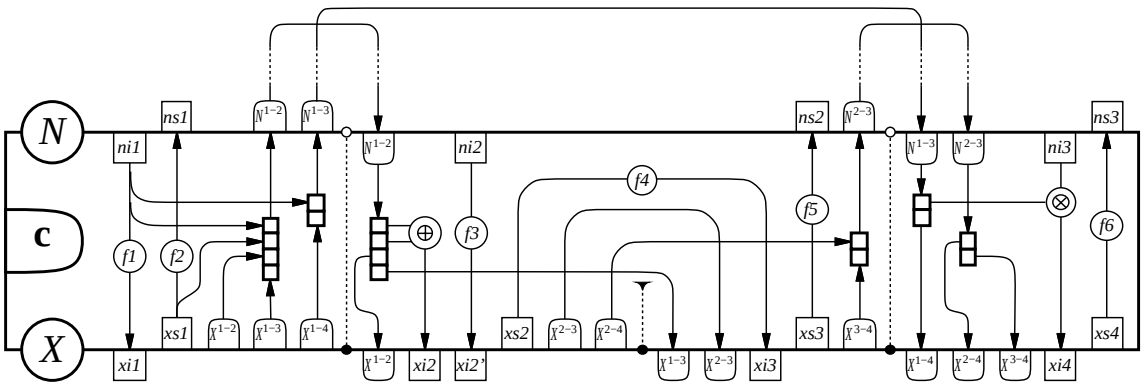

Fig 8: Final dataflow

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

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

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

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

Suppose we have n different, mutual recursive, data types T_i $(1 \leq i \leq n)$. Suppose also that we have t_i traversals on data type T_i. These traversals have the following headers, where $1 \leq i \leq n$ and $1 \leq v \leq t_i$

$$trav^v_{T_i} :: T_i \ldots inputs \ldots \rightarrow \ldots outputs \ldots \quad .$$

These functions must be augmented with bindings, a change that is reflected in their headers. The function for traversal v for data type T_i must return binding information for all traversals that follow, that is to say for traversals w with $v+1 \leq w \leq t_i$. Besides that, this function must also be passed bindings from all its predecessors, i.e. traversals w with $1 \leq w \leq v-1$. In other words, the headers are changed to

$$\begin{array}{lcl} trav^v_{T_i} & :: & T_i \ldots inputs \ldots \rightarrow T_i^{1\rightarrow v} \rightarrow T_i^{2\rightarrow v} \rightarrow \ldots \rightarrow T_i^{v-1\rightarrow v} \\ & \rightarrow & \langle \ldots outputs \ldots, T_i^{v\rightarrow v+1}, T_i^{v\rightarrow v+2}, \cdots, T_i^{v\rightarrow t_i} \rangle \quad , \end{array}$$

where $T_i^{v\rightarrow w}$ is the type of the binding computed during traversal v to T_i and used during traversal w to T_i. We must now determine the constructors for the bindings $T_i^{v\rightarrow w}$.

Suppose that there are n_i constructors $\mathbf{con}_{i,k}$ (where $1 \leq k \leq n_i$) on type T_i. With each of these constructors we associate a set of so called binding constructors $\mathbf{con}_{i,k}^{v\rightarrow w}$ on $T_i^{v\rightarrow w}$ with defining traversal v $(1 \leq v \leq t_i - 1)$ and using traversal w $(v+1 \leq w \leq t_i)$. In other words, for any type T_i, $1 \leq i \leq n$, we have $\frac{1}{2} t_i(t_i - 1)$ associated binding types $T_i^{v\rightarrow w}$, $1 \leq v < w \leq t_i$, each with n_i binding constructors:

$$\begin{array}{lcl} \mathbf{data}\ T_i^{v\rightarrow w} & = & \mathbf{con}_{i,1}^{v\rightarrow w} \ldots \\ & | & \mathbf{con}_{i,2}^{v\rightarrow w} \ldots \\ & \vdots & \\ & | & \mathbf{con}_{i,n_i}^{v\rightarrow w} \ldots \quad . \end{array}$$

Now that we have set up a framework for bindings, we are finally able to discuss the *shape* of the binding constructors. A binding constructor $\mathbf{con}_{i,k}^{v\rightarrow w}$ binds objects that are computed in traversal v of an instance of constructor $\mathbf{con}_{i,k}$ and that are used in traversal w of that same node. Binding constructors bind two kinds of objects namely *values* from the decoration, and *bindings for sons*. In our running example, a **Tip** node puts a *value* in a binding, and a **Fork** node puts the *binding* for each son in a binding.

4.3 Binding examples

Bindings have proven useful in large tree decoration settings like parse tree decoration. Since trees in these contexts are constructed from many mutual recursive types, each of them having a different number of traversals, they form an interesting example albeit much too large to present entirely. We have displayed one constructor **c** in Fig. 6 that illustrates most peculiarities of bindings.

We have picked a fairly simple constructor, namely a unairy one: $\mathbf{c} :: X \rightarrow N$. The large grey arrows sketch how the three traversals to N take place. Each small box in figures 6–8 denotes a value. Boxes pointing downwards are input values for a traversal and boxes pointing upwards are output values. The vertical dashed lines are borderlines between the successive traversals. Arrows direct the dataflow and they are labeled with function names in circle. For example, the first traversal has the following form (where we use underlining to denote the type of the object)

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

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

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 $(\mathrm{Tree}^{\mathit{rtips}\to\mathit{match}}\ \beta)$. These are disjunct instances of the same type $(\mathrm{Tree}^{\mathit{rtips}\to\mathit{match}}\ \beta)$

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

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.

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

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

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

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.

$$
\begin{array}{lcl}
rtm :: (\alpha \to \beta) \to \text{Tree } \alpha \to [\beta] \to [\beta] \to \langle [\beta], [\beta], \text{Bool} \rangle & & \\
rtm\ n\ (\mathbf{Tip}\ a)\ rs\ \tilde{}(v : vs) & = & \langle i : rs, vs, v \equiv i \rangle \ \mathbf{where}\ i = n\ a \\
rtm\ n\ (\mathbf{Fork}\ l\ r)\ rs\ vs & = & \langle rs'', vs'', l' \wedge r' \rangle \\
\quad \mathbf{where} & & \langle rs', vs', l' \rangle = rtm\ n\ l\ rs\ vs \\
& & \langle rs'', vs'', r' \rangle = rtm\ n\ r\ rs'\ vs' \\
cpalin :: (\alpha \to \beta) \to \text{Tree } \alpha \to \text{Bool} & & \\
cpalin\ n\ t & = & t' \ \mathbf{where}\ \langle rs, \_, t' \rangle = rtm\ n\ t\ []\ rs
\end{array}
$$

Fig 4: The circular palindrome recognizer for termed trees

4 Bindings

In the previous section we presented a circular program that avoided recomputation of the normal forms. Although our major criterion was met, this approach did not allow for memoization. We will present two solutions that can be memoed. Both use a two traversal strategy and record the intended side effects in an additional data structure.

The actual implementation differs though. In the first approach the first traversal computes an additional object, namely an image of the tree that records precisely those intermediate values that are needed in the second traversal, a so called a binding. In the second approach described in the next section, we use a monad to mimic a partially decorated walkable tree as it would be done in an imperative setting.

In this section, we focus on bindings [Pen92, Vog93]. Bindings contain precisely those values that should be passed from one traversal to the next. They are *constructed* in one traversal —values that should be passed are included— and *destructed* in the next so that the values are ready to be used. Bindings are terms with a structure much like the tree that is being traversed.

To get a feeling for bindings, we will first show how they are used in the running example. Next, we will discuss how bindings are constructed in general. Then we illustrate a complex case in full detail. We conclude this section with some observations.

4.1 Bindings in the running example

In the palindrome example, a tree is traversed twice: first by *rtips* and then by *match*. We regard (instances of) the *constructors* as the nodes of the tree. For each of these nodes, we must determine which values should be put in a binding. Since each node is an instance of a constructor, it suffices to determine which values should be put in a binding for each constructor. To determine this, we examine the successive traversals for a constructor and gather the values that are *computed* in one traversal, and *used* in a later one.

When we examine the program in Fig. 3, we observe that *rtips* computes a value i, the normal form of the term in the tip, that is also used by *match*. We decide to put i, which has type β, in a binding from *rtips* to *match*. The type of this binding will be denoted by $(\text{Tree}^{rtips \to match}\ \beta)$. A binding inherits the name from the data structure that is traversed. The superscript $rtips \to match$ denotes that this binding is computed in *rtips* and used in *match*.

$$
\begin{array}{lcl}
rtips :: (\alpha \to \beta) \to \text{Tree } \alpha \to [\beta] \to [\beta] & & \\
rtips\ n\ (\textbf{Tip}\ a)\ rs & = & i : rs\ \textbf{where}\ i = n\ a \\
rtips\ n\ (\textbf{Fork}\ l\ r)\ rs & = & rs'' \\
\quad \textbf{where} & & rs' = rtips\ n\ l\ rs \\
& & rs'' = rtips\ n\ r\ rs' \\
bmatch :: (\alpha \to \beta) \to \text{Tree } \alpha \to [\beta] \to \langle [\beta], \text{Bool} \rangle & & \\
match\ n\ (\textbf{Tip}\ a)\ (v : vs) & = & \langle vs, v \equiv i \rangle\ \textbf{where}\ i = n\ a \\
match\ n\ (\textbf{Fork}\ l\ r)\ vs & = & \langle vs'', l' \wedge r' \rangle \\
\quad \textbf{where} & & \langle vs', l' \rangle = match\ n\ l\ vs \\
& & \langle vs'', r' \rangle = match\ n\ r\ vs' \\
palin :: (\alpha \to \beta) \to \text{Tree } \alpha \to \text{Bool} & & \\
palin\ n\ t & = & t'\ \textbf{where}\ \langle [], t' \rangle = match\ n\ t\ (rtips\ n\ t\ [])\quad .
\end{array}
$$

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*.

Syntax and semantics are loosely based on GOFER [Jon91] a large subset of HASKELL [HPJW+92, HF92]. However, the special brackets $\langle\cdot\rangle$ will be used to denote tuples.

The first ingredient for a solution to our problem is a function that computes the reverse list of tips by folding the tree:

$$
\begin{array}{lcl}
\mathit{srtips} :: \text{Tree } \alpha \to [\alpha] \to [\alpha] & & \\
\mathit{srtips}\ (\mathbf{Tip}\ a)\ rs & = & a : rs \\
\mathit{srtips}\ (\mathbf{Fork}\ l\ r)\ rs & = & rs'' \\
\qquad\qquad \mathbf{where} & & rs' = \mathit{srtips}\ l\ rs \\
& & rs'' = \mathit{srtips}\ r\ rs' \quad .
\end{array}
$$

This reverse list of tips $[\alpha]$ must be redistributed over (Tree α), so that it may be compared with the successive tips. Since *smatch* uses compare ($\equiv$), α should be comparable.

$$
\begin{array}{lcl}
\mathit{smatch} :: \text{Tree } \alpha \to [\alpha] \to \langle [\alpha], \text{Bool} \rangle & & \\
\mathit{smatch}\ (\mathbf{Tip}\ a)\ (v : vs) & = & \langle vs, a \equiv v \rangle \\
\mathit{smatch}\ (\mathbf{Fork}\ l\ r)\ vs & = & \langle vs'', l' \wedge r' \rangle \\
\qquad\qquad \mathbf{where} & & \langle vs', l' \rangle = \mathit{smatch}\ l\ vs \\
& & \langle vs'', r' \rangle = \mathit{smatch}\ r\ vs' \quad .
\end{array}
$$

Having defined the ingredients, we are left with assembling a simple palindrome recognizer for trees.

$$
\begin{array}{lcl}
\mathit{spalin} :: \text{Tree } \alpha \to \text{Bool} & & \\
\mathit{spalin}\ t & = & t' \ \mathbf{where}\ \langle [], t' \rangle = \mathit{smatch}\ t\ (\mathit{srtips}\ t\ [\,]) \quad .
\end{array}
$$

2.2 The palindrome recognizer for term trees

In the simple recognizer we are not yet confronted with intra-traversal-communication. Therefore we will complicate matters. Up to now, the tree tips can be of any type α that can be compared. From now on, the tree tips will be "terms" which may be equal, even if they are not identical. For example the terms $3 + 5$ and 8 have a different structure, but they are considered equal nevertheless. Therefore, the tree in Fig. 2 is also a palindrome.

We formalize the idea by requiring a normalization function $\mathit{norm} :: \alpha \to \beta$ that reduces a term of type α to a normal form of type β. Instances of β can be compared by the standard equality function ($\equiv$) again.

In this example, tips will will have the following type

$$\mathbf{data}\ \text{Exp} = \mathbf{Const}\ \text{Int} \mid \mathbf{Sum}\ \text{Exp}\ \text{Exp} \quad ,$$

with a straightforward normalization function

$$
\begin{array}{lcl}
\mathit{norm} :: \text{Exp} \to \text{Int} & & \\
\mathit{norm}\ (\mathbf{Const}\ c) & = & c \\
\mathit{norm}\ (\mathbf{Sum}\ a\ b) & = & \mathit{norm}\ a + \mathit{norm}\ b \quad ,
\end{array}
$$

so that the standard equality ($\equiv$) on integers suffices to compare normalized expressions. Then *palin norm* as defined in Fig. 3 can be used to recognize a palindrome of type (Tree Exp). Note that *rtips id* = *srtips* and *match id* = *smatch* so that *palin id* = *spalin*.

other approaches, namely circular functions to short-circuit multiple traversals [Bir84, KS87, Kui89] and monads [Wad90, Wad92, Wad93] to mimic the standard imperative solution.

In judging solutions we have three criteria. The most important one is that no recomputations should take place. Once a value is computed store it for later use. Secondly, we strive for non-lazy evaluation since since this is easier to implement and faster in execution. Furthermore, non-lazy evaluation allows for memoization of the traversing functions. The ability to memoize, is essential to us. We are writing a compiler generator, and we rely on memoization to obtain incremental compilers.

Two of the solutions in this paper can handle the memoization requirement, namely the binding and the monad approach. Bindings yield shorter programs that are easier to understand than the programs that use monads. Therefore they have our preference.

2 The running example

Consider the problem of determining whether the tips of a binary tree form a palindrome. For example, the tree of Fig. 1 is a palindrome tree. A palindrome recognizer can be modeled by the following two pass algorithm: in the first pass the (reverse) list of tips is computed and in the second pass the list elements are compared with the tips. Hence, there is only one value, namely the list of tips, that is passed from the first traversal to the second. This value is passed at the root of the tree. In other words, this program does not require intra-traversal-communication.

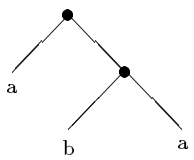

Fig 1: A simple palindrome tree

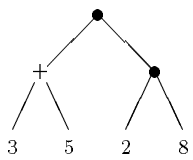

Fig 2: A termed palindrome tree

Let us now consider the same problem except that the tip values are complicated terms, like expressions as in Fig. 2. These terms are defined to be equal if their normal forms (values) are equal. In the first pass, the normal forms of the tips are collected in a list. In the second pass, the list elements are to be compared with the normal forms of the tips again. To assure that every tip is normalized only once, the normal form should be computed in the first traversal and somehow be send to the second. This requires intra-traversal-communication at every tip.

2.1 The palindrome recognizer for simple trees

First we focus on a *simple* palindrome recognizer: equality of tree elements is established using the build-in equality function ($\equiv$). Let us first define a data type for trees on α.

$$\textbf{data}\ \text{Tree}\ \alpha = \textbf{Tip}\ \alpha \mid \textbf{Fork}\ (\text{Tree}\ \alpha)\ (\text{Tree}\ \alpha) \quad .$$

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

Maarten Pennings*

Department of Computer Science, Utrecht University,
P.O. Box 80 089, 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 memoize 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 *intra-traversal-communications* occur, we must provide for a mechanism to deal with them.

A typical example of the class we study is a compiler. Compilers are in essence parse tree decorators. Type checking, type coercion, determining 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 not unlikely.

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 612-317-032.

ISSN: 0924–3275

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

Maarten Pennings

Technical Report RUU-CS-93-46
December 1993

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

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