J. Functional Programming **11** (3): 347–358, May 2001. Printed in the United Kingdom

FUNCTIONAL PEARL
Unfolding pointer algorithms

RICHARD S. BIRD
Programming Research Group, Oxford University
Wolfson Building, Parks Road, Oxford OX1 3QD, UK

1 Introduction

A fair amount has been written on the subject of reasoning about pointer algorithms. There was a peak about 1980 when everyone seemed to be tackling the formal verification of the Schorr–Waite marking algorithm, including Gries (1979, Morris (1982) and Topor (1979). Bornat (2000) writes: "The Schorr–Waite algorithm is the first mountain that any formalism for pointer aliasing should climb". Then it went more or less quiet for a while, but in the last few years there has been a resurgence of interest, driven by new ideas in relational algebras (Möeller, 1993), in data refinement Butler (1999), in type theory (Hofmann, 2000; Walker and Morrisett, 2000), in novel kinds of assertion (Reynolds, 2000), and by the demands of mechanised reasoning (Bornat, 2000). Most approaches end up being based in the Floyd–Dijkstra–Hoare tradition with loops and invariant assertions. To be sure, when dealing with any recursively-defined linked structure some declarative notation has to be brought in to specify the problem, but no one to my knowledge has advocated a purely functional approach throughout. Mason (1988) comes close, but his Lisp expressions can be very impure. Möller (1999) also exploits an algebraic approach, and the structure of his paper has much in common with what follows.

This pearl explores the possibility of a simple functional approach to pointer manipulation algorithms.

2 A little theory

Suppose *Adr* is some set of 'addresses', containing a distinguished element *Nil*. A list of type $[T]$ can be represented by an address a and two functions

$$\begin{array}{lcl} \mathit{next} & :: & \mathit{Adr} \to \mathit{Adr} \\ \mathit{data} & :: & \mathit{Adr} \to T \end{array}$$

The abstraction function is $\mathit{map}\ \mathit{data} \cdot (\mathit{next}\,\star\,)$, where

$$\begin{array}{lcl} (\star) & :: & (\mathit{Adr} \to \mathit{Adr}) \to \mathit{Adr} \to [\mathit{Adr}] \\ f \star a & = & \mathbf{if}\ a = \mathit{Nil}\ \mathbf{then}\ [\,]\ \mathbf{else}\ a : f \star (f\ a) \end{array}$$

The operator $\star$ is a cut-down version of a more general function *unfold*; see Gibbons and Jones (1998) for a discussion of the use of *unfold* in functional programming.

https://doi.org/10.1017/S0956796801003914 Published online by Cambridge University Press

Since all the algorithms considered below are polymorphic, the *data* function plays no essential part in the calculations, so we will quietly ignore it.

For later use, define the predicates

$$\begin{array}{lcl} FL\,(f,a) & = & f \star a \text{ is a finite list} \\ ND\,(f,a) & = & f \star a \text{ contains no duplicates} \\ DJ\,(f,a,b) & = & f \star a \text{ and } f \star b \text{ have no common elements} \end{array}$$

It is clear that $FL \Rightarrow ND$ because the presence of a duplicate element produces a cycle. And $ND \Rightarrow FL$ if the set Adr is finite.

Apart from $\star$, the other basic ingredient we will need is the one-point update function defined by

$$f[a := b] \quad = \quad \lambda x.\mathbf{if}\ x = a\ \mathbf{then}\ b\ \mathbf{else}\ f\ x$$

Obvious properties of this function include:

$$\begin{array}{lcl} f[a := f\ a] & = & f \\ f[a := b][a := c] & = & f[a := c] \end{array}$$

The key result is the following observation:

$$a \notin f \star x \quad \Rightarrow \quad f[a := b] \star x = f \star x \qquad (1)$$

In words, if a doesn't appear on the list $f \star x$ we can change its f-value to anything we like. Proof of (1) is a simple exercise in induction (see Bird, 1998, Ch. 9), and we omit details.

3 Reversal

Let us begin with something that every functional programmer knows: efficient list reversal. Everyone knows that the naive definition of *reverse*, namely,

$$\begin{array}{lcl} reverse\,[\,] & = & [\,] \\ reverse\,(x : xs) & = & reverse\ xs +\!\!+ [x] \end{array}$$

takes quadratic time in the length of the list. And everyone knows that the way to improve efficiency is to introduce an accumulating parameter. More precisely, define *revcat* by

$$revcat\ xs\ ys \quad = \quad reverse\ xs +\!\!+ ys$$

and use this specification to synthesize the following alternative definition of *revcat*:

$$\begin{array}{lcl} revcat\ [\,]\ ys & = & ys \\ revcat\ (x : xs)\ ys & = & revcat\ xs\ (x : ys) \end{array}$$

The computation of *revcat* takes linear time and, since $reverse\ xs = revcat\ xs\ [\,]$, we now have a linear-time algorithm for *reverse*.

For the next step, suppose that the lists are presented to us as linked lists through the function *next* of the previous section. We can pose the question: for what functions *step* and *init*, if any, do we have

$$revcat\ (next \star a)\,(next \star b) \quad = \quad (step\ next\ a\ b) \star (init\ next\ a\ b) \qquad ?$$

https://doi.org/10.1017/S0956796801003914 Published online by Cambridge University Press

The existence of *step* and *init* surely depends on conditions on *next*, a and b, so we add in a proviso $P(next, a, b)$ and ask the supplementary question: what is the minimum P?

To answer the questions we proceed by calculation. In the case $a = Nil$ we argue:

$$\begin{array}{ll} & revcat\ (next \star a)\ (next \star b) \\ = & \quad \{\text{definition of } \star\} \\ & revcat\ [\,]\ (next \star b) \\ = & \quad \{\text{definition of } revcat\} \\ & next \star b \end{array}$$

Hence we can take $step\ next\ a\ b = next$ and $init\ next\ a\ b = b$.

In the case $a \neq Nil$ we will need to make two wishes during the course of the following calculation:

$$\begin{array}{ll} & revcat\ (next \star a)\ (next \star b) \\ = & \quad \{\text{definition of } \star \text{ in case } a \neq Nil\} \\ & revcat\ (a : next \star next\ a)\ (next \star b) \\ = & \quad \{\text{definition of } revcat\} \\ & revcat\ (next \star next\ a)\ (a : next \star b) \\ = & \quad \{\text{first wish, with } f \text{ to be defined later}\} \\ & revcat\ (f \star next\ a)\ (f \star a) \\ = & \quad \{\text{second wish: } P(next, a, b) \Rightarrow P(f, next\ a, a)\} \\ & (step\ f\ (next\ a)\ a) \star (init\ f\ (next\ a)\ a) \end{array}$$

Hence, in the case $a \neq Nil$, we can take

$$\begin{array}{lcl} step\ next\ a\ b & = & step\ f\ (next\ a)\ a \\ init\ next\ a\ b & = & init\ f\ (next\ a)\ a \end{array}$$

We still have to make the wishes come true, and this involves finding a function f such that when $a \neq Nil$:

$$a : next \star b \quad = \quad f \star a \tag{2}$$

$$next \star next\ a \quad = \quad f \star next\ a \tag{3}$$

$$P(next, a, b) \quad \Rightarrow \quad P(f, next\ a, a) \tag{4}$$

Implication (1) can be used to establish (2). To see this, we argue:

$$\begin{array}{ll} & f \star a \\ = & \quad \{\text{definition of } \star \text{ in case } a \neq Nil\} \\ & a : f \star f\ a \\ = & \quad \{\text{setting } f = next[a := b]\text{, so } f\ a = b\} \\ & a : f \star b \\ = & \quad \{(1)\text{, assuming } a \notin next \star b\} \\ & a : next \star b \end{array}$$

https://doi.org/10.1017/S0956796801003914 Published online by Cambridge University Press

Implication (1) can also be used to establish (3):

$$\begin{array}{ll} & f \star next\ a \\ = & \quad \{\text{with } f = next\,[a := b]\} \\ & next\,[a := b] \star next\ a \\ = & \quad \{(1), \text{ assuming } a \notin next \star (next\ a)\} \\ & next \star next\ a \end{array}$$

The requirements on P therefore take the form

$$\begin{array}{l} P(next, a, b) \wedge a \neq Nil \Rightarrow \\ \quad a \notin next \star b \wedge a \notin next \star (next\ a) \wedge P(next\,[a := b], next\ a, a) \end{array}$$

The weakest solution for P of this implication can be computed, with some effort, and turns out to be

$$P(next, a, b) \quad \equiv \quad ND(next, a) \wedge DJ(next, a, b)$$

In words, $next \star a$ has no duplicated elements and no elements in common with $next \star b$. Clearly, $DJ(next, a, Nil)$ holds

In summary, we have shown that, provided $ND(next, a)$,

$$reverse\,(next \star a) \quad = \quad f \star b \quad \textbf{where } (f, b) = loop\ next\ a\ Nil$$

and

$$loop\ next\ a\ b \quad = \quad \textbf{if}\, a = Nil \textbf{ then } (next, b) \textbf{ else } loop\,(next\,[a := b])\,(next\ a)\ a$$

Here is the definition of $loop\ next\ a\ Nil$ again, written this time in an imperative style:

```
$b := Nil$;
do $a \neq Nil \rightarrow$
    $next, a, b := next\,[a := b], next\ a, a$
od ;
return $(next, b)$
```

Replacing $next := next\,[a := b]$ by $next\,[a] := b$ gives essentially the code for the in-place reversal of a linked list. Bornat (2000) writes: “the in-place list-reversal algorithm is the lowest hurdle that a pointer-aliasing formalism ought to be able to jump”. We have made the hurdle a little higher than it might have been by not stating a reasonable precondition at the outset. But then, we didn’t give the details of how to compute the minimum precondition P from its specification. Note carefully that the precondition is that $next \star a$ should not contain duplicates, not that it should be a finite list. To be sure, if $next \star a$ were not finite the code above would not terminate, but then neither would *revcat* so the implemention is correct. If $next \star a$ did contain a duplicate, so was a cyclic list, the implementation above would terminate with an incorrect result.

https://doi.org/10.1017/S0956796801003914 Published online by Cambridge University Press

4 Concatenation

Before proceeding to the looming mountain of Schorr–Waite, let us dally in the foothills of a simpler problem, namely an in-place pointer algorithm for list concatenation.

Many operations on linked lists are simpler to implement when the lists are represented using so-called *header cells*. In a header-cell implementation, a list xs is represented by the address a of a special cell (so $a \neq Nil$) under the abstraction mapping $map\ data \cdot (next\ \diamond\ )$ where

$$f \diamond x \quad = \quad f \star (f\ x)$$

The use of header cells explains why we pose the question for list concatenation in the following form: for what function $step$, and under what proviso P, do we have

$$next \diamond a \mathbin{+\!\!+} next \diamond b \quad = \quad (step\ next\ a\ b) \diamond a \qquad ?$$

Our aim is to come up with the following definition of $step$:

$$\begin{array}{lll} step\ next\ a\ b & = & \textbf{if}\ next\ a = Nil\ \textbf{then}\ next\,[a := next\ b] \\ & & \textbf{else}\ step\ next\ (next\ a)\ b \end{array}$$

In an imperative idiom $step$ is implemented by the loop

```
$x := a$;
do $next[x] \neq Nil \rightarrow x := next[x]$ od ;
$next[x] := next[b]$;
return $next$
```

If $next \diamond a$ is not a finite list, then the value of $step$ is $\perp$. But in functional programming $xs \mathbin{+\!\!+} ys = xs$ if xs is an infinite list. To implement $\mathbin{+\!\!+}$ faithfully the algorithm above would not suffice; instead we would have to detect whether $next \star a$ is cyclic and do nothing if it was. To avoid this complexity we will assume at the outset that $next \diamond a$ is a finite list.

To justify the implementation, we again proceed by calculation. In the case $next\ a = Nil$, we argue:

$$\begin{array}{ll} & next \diamond a \mathbin{+\!\!+} next \diamond b \\ = & \quad \{\text{definition of } \diamond\} \\ & [\,] \mathbin{+\!\!+} next \diamond b \\ = & \quad \{\text{definition of } \mathbin{+\!\!+}\} \\ & next \diamond b \\ = & \quad \{\text{claim, assuming } a \notin next \diamond b\} \\ & next\,[a := next\ b] \diamond a \end{array}$$

For the claim, we reason:

$$\begin{array}{ll} & next\,[a := next\ b] \diamond a \\ = & \quad \{\text{definition of } \diamond \text{ and } next\,[a := next\ b]\ a = next\ b\} \end{array}$$

https://doi.org/10.1017/S0956796801003914 Published online by Cambridge University Press

$$\begin{array}{ll} & \mathit{next}\,[a := \mathit{next}\ b] \star \mathit{next}\ b \\ = & \quad \{(1), \text{ assuming } a \notin \mathit{next} \diamond b\} \\ & \mathit{next} \diamond b \end{array}$$

Hence we can take $\mathit{step}\ \mathit{next}\ a\ b = \mathit{next}\,[a := \mathit{next}\ b]$, provided that

$$P(\mathit{next}, a, b) \wedge \mathit{next}\ a = \mathit{Nil} \quad \Rightarrow \quad a \notin \mathit{next} \diamond b$$

In the case $\mathit{next}\ a \neq \mathit{Nil}$, we argue:

$$\begin{array}{ll} & \mathit{next} \diamond a \mathbin{+\!\!+} \mathit{next} \diamond b \\ = & \quad \{\text{definition of } \diamond \text{ and } \mathbin{+\!\!+}\} \\ & \mathit{next}\ a : (\mathit{next} \diamond \mathit{next}\ a \mathbin{+\!\!+} \mathit{next} \diamond b) \\ = & \quad \{\text{induction, writing } f = \mathit{step}\ \mathit{next}\ (\mathit{next}\ a)\, b, \text{ assuming } P(\mathit{next}, \mathit{next}\ a, b)\} \\ & \mathit{next}\ a : f \diamond \mathit{next}\ a \\ = & \quad \{\text{assume } P(\mathit{next}, a, b) \Rightarrow \mathit{next}\ a = f\ a\} \\ & f \diamond a \end{array}$$

We can therefore take $\mathit{step}\ \mathit{next}\ a\ b = f$, provided that

$$\begin{array}{l} P(\mathit{next}, a, b) \wedge \mathit{next}\ a \neq \mathit{Nil} \Rightarrow \\ \qquad \mathit{next}\ a = \mathit{step}\ \mathit{next}\ (\mathit{next}\ a)\, b\ a \wedge P(\mathit{next}, \mathit{next}\ a, b) \end{array}$$

This gives the definition of *step* described above.

To see what P entails, observe from the definition of *step* and the assumption that $\mathit{next} \diamond a$ is a finite list, that

$$\mathit{next}\ a \neq \mathit{Nil} \quad \Rightarrow \quad \mathit{step}\ \mathit{next}\ (\mathit{next}\ a)\, b = \mathit{next}\,[x := \mathit{next}\ b]$$

for some $x \in \mathit{next} \diamond a$. Since $a \notin \mathit{next} \diamond a$ (otherwise $\mathit{next} \diamond a$ is not finite), we obtain

$$\mathit{step}\ \mathit{next}\ (\mathit{next}\ a)\, b\ a \quad = \quad \mathit{next}\ a$$

as required. The minimum solution for

$$\begin{array}{lll} P(\mathit{next}, a, b) \wedge \mathit{next}\ a = \mathit{Nil} & \Rightarrow & a \notin \mathit{next} \diamond b \\ P(\mathit{next}, a, b) \wedge \mathit{next}\ a \neq \mathit{Nil} & \Rightarrow & P(\mathit{next}, \mathit{next}\ a, b) \end{array}$$

turns out to be

$$P(\mathit{next}, a, b) \quad \equiv \quad (\forall k :: \mathit{next}^{k+1}\ a = \mathit{Nil} \Rightarrow \mathit{next}^{k} \notin \mathit{next} \diamond b)$$

One can show that $\mathit{DJ}(\mathit{next}, a, b) \Rightarrow P(\mathit{next}, a, b)$, so it is sufficient to assume that the finite list $\mathit{next} \star a$ has no elements in common with $\mathit{next} \star b$.

5 Schorr–Waite

The Schorr–Waite marking algorithm takes as inputs a directed graph with outdegree at most two and an initial node a, and returns a function m such that $m\,b = 1$ if node b is reachable from a and $m\,b = 0$ otherwise. The adjacency information is

https://doi.org/10.1017/S0956796801003914 Published online by Cambridge University Press

given by two functions $\ell, r :: Adr \to Adr$, short for left and right. Either $\ell\ a$ or $r\ a$ can be Nil.

Our starting point is the following standard marking algorithm:

$$\begin{array}{lcl} mark\ (\ell, r, a) & = & mark1\ (\ell, r, const\ 0, [a]) \\ mark1\ (\ell, r, m, [\,]) & = & (\ell, r, m) \\ mark1\ (\ell, r, m, a : as) & = & \mathbf{if}\ a \neq Nil \wedge m\ a = 0 \\ & & \mathbf{then}\ mark1\ (\ell, r, m[a := 1], \ell\ a : r\ a : as) \\ & & \mathbf{else}\ mark1\ (\ell, r, m, as) \end{array}$$

The result of $mark\ (\ell, r, a)$ is a triple of functions (ℓ, r, m) such that $m\ b = 1$ if b is reachable from a, and $m\ b = 0$ otherwise. We also return the adjacency functions (ℓ, r) because during the course of the Schorr–Waite algorithm they are modified, and we wish to ensure that they end up restored to their original values. Note, finally, that the list argument of $mark1$ is treated as a stack.

For the first step we transform $mark1$ into a function $mark2$ satisfying

$$mark2\ (\ell, r, m, a, as) \quad = \quad mark1\ (\ell, r, m, a : map\ r\ as)$$

The idea is to use the stack as only as a repository for marked nodes whose right subtrees have not yet been explored. In particular,

$$mark\ (\ell, r, a) = mark2\ (\ell, r, const\ 0, a, [\,])$$

Synthesizing a direct recursive definition of $mark2$ leads quite easily to the following code:

$$\begin{array}{l} mark2\ (\ell, r, m, a, as) = \\ \quad \left|\ \begin{array}{lcl} a \neq Nil \wedge m\ a = 0 & \to & mark2\ (\ell, r, m[a := 1], \ell\ a, a : as) \\ null\ as & \to & (\ell, r, m) \\ otherwise & \to & mark2\ (\ell, r, m, r\ (head\ as), tail\ as) \end{array}\right. \end{array}$$

Note that arguments m and as of $mark2$ satisfy the property that if $x \in as$, then $m\ x = 1$.

The next step is to represent the stack by a linked list. The way this is done is the central idea of the Schorr–Waite algorithm. We will tackle this rock face by first considering two simpler representations.

The most obvious representation is to introduce an additional function $n :: Adr \to Adr$ (short for $next$) and use the abstraction

$$stack\ (n, b) \quad = \quad n \star b$$

As a somewhat more complicated representation, we can represent the stack by a triple (s, n, b), where n and b are as above and s is a new marking function. The abstraction function is

$$stack\ (n, s, b) \quad = \quad filter\ (marked\ s)\ (n \star b)$$

where $marked\ s\ a = (s\ a = 1)$. This representation leads to the following implementations of the stack operations:

$$\begin{array}{lcl} a : (n,s,b) & = & (n[a := b], s[a := 1], a) \\ \mathit{head}\,(n,s,b) & = & \textbf{if}\ s\,b = 1\ \textbf{then}\ b\ \textbf{else}\ \mathit{head}\,(n,s,n\,b) \\ \mathit{tail}\,(n,s,b) & = & \textbf{if}\ s\,b = 1\ \textbf{then}\ (n, s[b := 0], b)\ \textbf{else}\ \mathit{tail}\,(n,s,n\,b) \end{array}$$

The marking function s is used to delay removing elements from the stack. When an element a is added to the stack, $s\,a$ is set to 1. When this element is popped it is not removed immediately but instead $s\,a$ is set to 0. It is removed only when access to successors on the stack is required.

This representation of the stack leads to the introduction of *mark*3, specified by

$$\mathit{mark}3\,(\ell,r,m,s,n,a,b) \quad = \quad \mathit{mark}2\,(\ell,r,m,a,\mathit{stack}\,(n,s,b))$$

In particular, we have

$$\mathit{mark}\,(\ell,r,a) \quad = \quad \mathit{mark}3\,(\ell,r,\mathit{const}\ 0,\mathit{const}\ 0,\bot,a,\mathit{Nil})$$

since the initial values of s and n are irrelevant. We choose, however, to initialise s to *const* 0 since that will also be the final value of s.

Synthesizing a direct definition of *mark*3 leads to

$$\begin{array}{l} \mathit{mark}3\,(\ell,r,m,n,s,a,b) = \\ \quad \left| \begin{array}{lcl} a \neq \mathit{Nil} \wedge m\,a = 0 & \rightarrow & \mathit{mark}3\,(\ell,r,m[a := 1], n[a := b], s[a := 1], \ell\,a, a) \\ b = \mathit{Nil} & \rightarrow & (\ell,r,m) \\ \mathit{otherwise} & \rightarrow & \mathit{pop}\,(\ell,r,m,n,s,a,b) \end{array} \right. \end{array}$$

where

$$\begin{array}{l} \mathit{pop}\,(\ell,r,m,n,s,a,b) = \\ \quad \left| \begin{array}{lcl} s\,b = 1 & \rightarrow & \mathit{mark}3\,(\ell,r,m,n,s[b := 0], r\,b, b) \\ s\,b = 0 & \rightarrow & \mathit{pop}\,(\ell,r,m,n,s,b,n\,b) \end{array} \right. \end{array}$$

Since we know that if b is on the stack, then $b \neq \mathit{Nil} \wedge m\,b = 1$, we can eliminate calls to *pop* and replace *mark*3 with the simpler though marginally less efficient version

$$\begin{array}{l} \mathit{mark}3\,(\ell,r,m,n,s,a,b) = \\ \quad \left| \begin{array}{lcl} a \neq \mathit{Nil} \wedge m\,a = 0 & \rightarrow & \mathit{mark}3\,(\ell,r,m[a := 1], n[a := b], s[a := 1], \ell\,a, a) \\ b = \mathit{Nil} & \rightarrow & (\ell,r,m) \\ s\,b = 1 & \rightarrow & \mathit{mark}3\,(\ell,r,m,n,s[b := 0], r\,b, b) \\ s\,b = 0 & \rightarrow & \mathit{mark}3\,(\ell,r,m,n,s,b,n\,b) \end{array} \right. \end{array}$$

We are now ready for the third representation of the stack. The cunning idea of Schorr and Waite is to eliminate the function n and to store its values in the ℓ and r fields instead. More precisely, the aim is to replace n by the function $\mathit{next}\,(\ell,r,s)$ defined by

$$\mathit{next}\,(\ell,r,s) \quad = \quad \lambda x.\textbf{if}\ s\,x = 1\ \textbf{then}\ \ell\,x\ \textbf{else}\ r\,x \qquad (5)$$

As a result, we are left with providing just one extra marking function s, and since s

https://doi.org/10.1017/S0956796801003914 Published online by Cambridge University Press

requires a single bit per node rather than a full address, there is a significant saving in space.

The functions ℓ and r are modified during the algorithm, in fact at any point $\ell\,x$ and $r\,x$ are guaranteed to have their initial values only if x is not on the list $n \star b$. We claim that they can be restored to their original values by the function *restore*, defined by

$$
\begin{array}{l}
\mathit{restore}\,(\ell, r, s, a, b) = \\
\quad\left|\begin{array}{lcl}
b = \mathit{Nil} & \rightarrow & (\ell, r) \\
s\,b = 1 & \rightarrow & \mathit{restore}\,(\ell[b := a], r, s, b, n\,b) \\
s\,b = 0 & \rightarrow & \mathit{restore}\,(\ell, r[b := a], s, b, n\,b)
\end{array}\right.
\end{array}
$$

where $n = \mathit{next}\,(\ell, r, s)$. Informally, the stack is traversed and the values of ℓ and r are restored by appropriate updating. By definition of *next* we can replace $n\,b$ by $\ell\,b$ in the first recursive call of *restore* and by $r\,b$ in the second. Setting

$$\mathit{restore}\,(\ell, r, s, a, b) \quad = \quad (\ell_0, r_0)$$

it is clear that $\ell_0\,x = \ell\,x$ and $r_0\,x = r\,x$ for all x not on the list $n \star b$.

Now introduce *mark*4 defined by

$$\mathit{mark}4\,(\ell, r, m, s, a, b) \quad = \quad \mathit{mark}3\,(\mathit{restore}\,(\ell, r, s, a, b), m, \mathit{next}\,(\ell, r, s), s, a, b)$$

For syntactic accuracy the first two arguments of *mark*3 should have been paired, so assume they were. It is easy to show that

$$\mathit{mark}\,(\ell, r, a) \quad = \quad \mathit{mark}4\,(\ell, r, \mathit{const}\ 0, \mathit{const}\ 0, \bot, a, \mathit{Nil})$$

Our objective is to synthesize the following recursive definition of *mark*4:

$$
\begin{array}{l}
\mathit{mark}4\,(\ell, r, m, s, a, b) = \\
\quad\left|\begin{array}{lcl}
a \neq \mathit{Nil} \wedge m\,a = 0 & \rightarrow & \mathit{mark}4\,(\ell[a := b], r, m[a := 1], s[a := 1], \ell\,a, a) \\
b = \mathit{Nil} & \rightarrow & (\ell, r, m) \\
s\,b = 1 & \rightarrow & \mathit{mark}4\,(\ell[b := a], r[b := \ell\,b], m, s[b := 0], r\,b, b) \\
s\,b = 0 & \rightarrow & \mathit{mark}4\,(\ell, r[b := a], m, s, b, r\,b)
\end{array}\right.
\end{array}
$$

This is the Schorr–Waite marking algorithm. The functions m and s are implemented as additional fields in each node. One can easily translate the tail recursive *mark*4 into an imperative loop and we do not give details.

For convenience in the synthesis, let $(\ell_0, r_0) = \mathit{restore}\,(\ell, r, s, a, b)$ and $n = \mathit{next}\,(\ell, r, s)$.

In the case $a \neq \mathit{Nil} \wedge m\,a = 0$ we argue:

$$
\begin{array}{cl}
 & \mathit{mark}4\,(\ell, r, m, s, a, b) \\
= & \quad \{\text{definition of } \mathit{mark}4\} \\
 & \mathit{mark}3\,((\ell_0, r_0), m, s, n, a, b) \\
= & \quad \{\text{case assumption}\} \\
 & \mathit{mark}3\,((\ell_0, r_0), m[a := 1], s[a := 1], n[a := b], \ell_0\,a, a) \\
= & \quad \{\text{claim}\} \\
 & \mathit{mark}4\,(\ell[a := b], r, m[a := 1], s[a := 1], \ell\,a, a)
\end{array}
$$

The claim relies on three facts: if $a \neq Nil \wedge m\,a = 0$, then

$$\begin{array}{rcll} \ell_0\,a & = & \ell\,a & (6)\\ (\ell_0, r_0) & = & restore\,(\ell[a := b], r, s[a := 1], \ell\,a, a) & (7)\\ n[a := b] & = & next\,(\ell[a := b], r, s[a := 1]) & (8) \end{array}$$

In the case $b = Nil$ we argue:

$$\begin{array}{ll} & mark4\,(\ell, r, m, s, a, b)\\ = & \quad \{\text{definition of } mark4\}\\ & mark3\,((\ell_0, r_0), m, s, a, b)\\ = & \quad \{\text{definition of } mark3 \text{ in the case } b = Nil\}\\ & (\ell_0, r_0, m)\\ = & \quad \{\text{definition of } restore \text{ in the case } b = Nil\}\\ & (\ell, r, m) \end{array}$$

Similar calculations in the case $b \neq Nil \wedge s\,b = 1$ yields the desired result provided, in this case, that

$$\begin{array}{rcll} r_0\,b & = & r\,b & (9)\\ (\ell_0, r_0) & = & restore\,(\ell[b := a], r[b := \ell\,b], s[b := 0], r\,b, b) & (10)\\ n & = & next\,(\ell[b := a], r[b := \ell\,b], s[b := 0]) & (11) \end{array}$$

Finally, in the case $b \neq Nil \wedge s\,b = 0$ we require

$$\begin{array}{rcll} (\ell_0, r_0) & = & restore\,(\ell, r[b := a], s, b, r\,b) & (12)\\ n\,b & = & r\,b & (13) \end{array}$$

Now we must verify that these conditions hold. Equation (6) is immediate since $m\,a = 0$ implies $a \notin n \star b$ and so $\ell\,a = \ell_0\,a$ and $r\,a = r_0\,a$. For (7) we argue:

$$\begin{array}{ll} & restore\,(\ell[a := b], r, s[a := 1], \ell\,a, a)\\ = & \quad \{\text{definition of } restore \text{ since } s[a := 1]\,a = 1\}\\ & restore\,(\ell[a := b][a := \ell\,a], r, s[a := 0], a, b)\\ = & \quad \{\text{simplification and } m\,a = 0 \Rightarrow s\,a = 0\}\\ & restore\,(\ell, r, s, a, b) \end{array}$$

For (8) we argue, writing $(p \rightarrow q, r)$ as shorthand for **if** p **then** q **else** r:

$$\begin{array}{ll} & next\,(\ell[a := b], r, s[a := 1])\,x\\ = & \quad \{\text{definition of } next\}\\ & (s[a := 1]\,x = 1 \rightarrow \ell[a := b]\,x, r\,x)\\ = & \quad \{\text{definition of } update\}\\ & (x = a \rightarrow b, (s\,x = 1 \rightarrow \ell\,x, r\,x))\\ = & \quad \{\text{definition of } n = next\,(\ell, r, s)\}\\ & n[a := b] \end{array}$$

https://doi.org/10.1017/S0956796801003914 Published online by Cambridge University Press

For (9) we argue:

$$\begin{array}{ll} & restore\,(\ell, r, s, a, b) \\ = & \quad \{\text{case assumption } s\,b = 1\} \\ & restore\,(\ell[b := a], r, s[b := 0], b, \ell\,b) \end{array}$$

Now, since $b \notin n \star \ell\, b$ we have $\ell_0\, b = \ell[b := a]\, b = a$ and $r_0\, b = r\, b$.

For (10) we argue:

$$\begin{array}{ll} & restore\,(\ell[b := a], r[b := \ell\,b], s[b := 0], r\,b, b) \\ = & \quad \{\text{definition of } restore \text{ and } s[b := 0]\,b = 0\} \\ & restore\,(\ell[b := a], r[b := \ell\,b][b := r\,b], s[b := 0], b, r[b := \ell\,b]\,b) \\ = & \quad \{\text{simplification}\} \\ & restore\,(\ell[b := a], r, s[b := 0], b, \ell\,b) \\ = & \quad \{\text{definition of } restore \text{ and case assumption } s\,b = 1\} \\ & restore\,(\ell, r, s, a, b) \end{array}$$

For (11) we argue:

$$\begin{array}{ll} & next\,(\ell[b := a], r[b := \ell\,b], s[b := 0])\,x \\ = & \quad \{\text{definition of } next\} \\ & (s[b := 0]\,x = 1 \rightarrow \ell[b := a]\,x, r[b := \ell\,b]\,x) \\ = & \quad \{\text{definition of update}\} \\ & (x = b \rightarrow \ell\,b, (s\,x = 1 \rightarrow \ell\,x, r\,x)) \\ = & \quad \{\text{case assumption } s\,b = 1\} \\ & n\,x \end{array}$$

For (12) we argue:

$$\begin{array}{ll} & restore\,(\ell, r[b := a], s, b, r\,b) \\ = & \quad \{\text{definition of } restore \text{ and case assumption } s\,b = 0\} \\ & restore\,(\ell, r, s, a, b) \end{array}$$

Finally, (13) is immediate from the case assumption $s\,b = 0$ and the definition of n.

6 Conclusions

I guess the main conclusion is that one can do most things functionally if one puts one's mind to it. One reason it seems to work with pointer algorithms is that, as functional programmers, we already have access to a large body of useful notations and ideas (accumulating parameters, tupling, and so on), ideas that have to be explained from first principles in other work. The development of the Schorr–Waite algorithm turned out to be basically one of program transformation using straightforward techniques. We started with a marking algorithm for directed graphs, but we could have begun earlier with the preorder traversal of a binary tree, and

https://doi.org/10.1017/S0956796801003914 Published online by Cambridge University Press

developed the starting point from that. Most of the subsequent treatment consisted of transformations to introduce a slightly curious implementation of stacks, followed by a data refinement to get rid of the *next* field.

While most of the reasoning consists of the manipulation of functional expressions, one also needs the occasional invariant between the arguments of functions. I have lectured to second-year students about pointer algorithms, using a refinement calculus of pre- and postconditions. None of the developments were as short as the ones above. To be sure, any treatment of the Schorr–Waite algorithm is bound to be fairly detailed, and none of the examples involved the creation of fresh addresses pointing to new cells. For that one would have to carry around a free list as an extra argument to functions that produce new cells. No doubt a suitable state monad would prove useful in hiding detail. From now on I will teach pointers using a functional approach.

References

Bijlsma, A. (1989) Calculating with pointers. *Science of Computer Programming*, 12, 191–205.

Bird, R. (1998) *Introduction to Functional Programming using Haskell.* Prentice Hall International.

Bornat, R. (2000) Proving pointer programs in Hoare Logic. *Mathematics of Program Construction Conference*, Punto de Lima.

Butler, M. (1999) Calculational derivation of pointer algorithms from tree operations. *Science of Computer Programming*, 33(3), 221–260.

Gibbons, J. and Jones, G. (1998) The underappreciated unfold. *ACM/SIGPLAN Conference on Functional Programming*, Baltimore, MD.

Gries, D. (1979) The Schorr–Waite graph marking algorithm. *Acta Informatica*, 11, 223–232.

Hofmann, M. (2000) A type system for bounded space and functional in-place update.

Luckham, D. C. and Suzuki, N. (1979) Verification of array, record, and pointer operations in Pascal. *ACM Trans. Programming Lang. and Syst.*, 1(2), 227–243.

Mason, I. A. (1988) Verification of programs that destructively manipulate data. *Sci. of Comput. Programming*, 10(2), 177–210.

Möller, B. (1997) Calculating with pointer structures. In: R. Bird and L. Meertens (editors), *Algorithmic Languages and Calculi*, pp 24–48. IFIP TC2/WG2.1 Working Conference. Chapman & Hall.

Möller, B. (1999) Calculating with acyclic and cyclic lists. *Infor. Sci.*, 119, 135–154.

Morris, J. M. (1982) A proof of the Schorr–Waite algorithm. In: M. Broy and G. Schmidt (editors), *Proceedings 1981 Marktoberdorf Summer School*, pp. 25–51. Reidel.

Reynolds, J. C. (2000) Reasoning about shared mutable data structure. *Proceedings of Hoare's Retirement Symposium*, Oxford.

Schorr, H. and Waite, W. M. (1967) An efficient machine-independent procedure for garbage collection in various list structures. *Comm. ACM*, 10, 501–506.

Topor, R. W. (1979) The correctness of the Schorr–Waite marking algorithm. *Acta Informatica*, 11, 211–221.

Walker, D. and Morrisett, G. (2000) Alias types for recursive data structures. *ACM Workshop on Types in Compilation*, Montreal, Canada (to appear).

https://doi.org/10.1017/S0956796801003914 Published online by Cambridge University Press