FUNCTIONAL PEARLS

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; 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 (Möller,1997), 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

\[
\begin{align*}
\text{next} &: \ Adr \to Adr \\
\text{data} &: \ Adr \to T
\end{align*}
\]

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

\[
\begin{align*}
\star &: \ (Adr \to Adr) \to Adr \to [Adr] \\
f \star a &= \ \text{if } a = \text{Nil then } [] \text{ else } a \star (f \ a)
\end{align*}
\]

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

\[
FL(f, a) = f \ast a \text{ is a finite list} \\
ND(f, a) = f \ast a \text{ contains no duplicates} \\
DJ(f, a, b) = f \ast a \text{ and } f \ast b \text{ have no common elements}
\]

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 \(\ast\), the other basic ingredient we will need is the one-point update function defined by

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

Obvious properties of this function include:

\[
\begin{align*}
f[a := f a] &= f \\
f[a := b][a := c] &= f[a := c]
\end{align*}
\]

The key result is the following observation:

\[
a \notin f \ast x \Rightarrow f[a := b] \ast x = f \ast x
\] (1)

In words, if \(a\) doesn’t appear on the list \(f \ast x\) we can change its \(f\)-value to anything we like. Proof of (1) is a simple exercise in induction – see (Bird,1998), Chapter 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,

\[
\begin{align*}
\text{reverse} [ ] &= [ ] \\
\text{reverse} (x : xs) &= \text{reverse} xs \cdot [x]
\end{align*}
\]

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 \textit{revcat} by

\[
\text{revcat} \; xs \; ys = \text{reverse} \; xs \cdot ys
\]

and use this specification to synthesize the following alternative definition of \textit{revcat}:

\[
\begin{align*}
\text{revcat} \; [ ] \; ys &= ys \\
\text{revcat} \; (x : xs) \; ys &= \text{revcat} \; xs \; (x : ys)
\end{align*}
\]

The computation of \textit{revcat} takes linear time and, since \(\text{reverse} \; xs = \text{revcat} \; xs \; []\), we now have a linear-time algorithm for \textit{reverse}.

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

\[
\text{revcat} \; (\text{next} \ast a) \; (\text{next} \ast b) = (\text{step} \; \text{next} \; a \; b) \ast (\text{init} \; \text{next} \; a \; b)
\]
The existence of \textit{step} and \textit{init} surely depends on conditions on \textit{next}, \textit{a} and \textit{b}, so we add in a proviso \(P(\text{next}, a, b)\) and ask the supplementary question: what is the minimum \(P\)?

To answer the questions we proceed by calculation. In the case \(a = \text{nil}\) we argue:

\[
\text{revcat} (\text{next} \ast a) (\text{next} \ast b) \\
= \{\text{definition of } \ast\} \\
\text{revcat} \begin{bmatrix}\text{next} \ast a\end{bmatrix} (\text{next} \ast b) \\
= \{\text{definition of } \text{revcat}\} \\
\text{next} \ast b
\]

Hence we can take \text{step next a b} = \text{next} and \text{init next a b} = b.

In the case \(a \neq \text{nil}\) we will need to make two wishes during the course of the following calculation:

\[
\text{revcat} (\text{next} \ast a) (\text{next} \ast b) \\
= \{\text{definition of } \ast\ \text{in case } a \neq \text{nil}\} \\
\text{revcat} (a : \text{next} \ast \text{next} a) (\text{next} \ast b) \\
= \{\text{definition of } \text{revcat}\} \\
\text{revcat} (\text{next} \ast \text{next} a) (a : \text{next} \ast b) \\
= \{\text{first wish, with } f \text{ to be defined later}\} \\
\text{revcat} (f \ast \text{next} a) (f \ast a) \\
= \{\text{second wish: } P(\text{next}, a, b) \Rightarrow P(f, \text{next} a, a)\} \\
(\text{step } f (\text{next} a) a) \ast (\text{init } f (\text{next} a) a)
\]

Hence, in the case \(a \neq \text{nil}\), we can take

\[
\text{step next a b} = \text{step } f (\text{next} a) a \\
\text{init next a b} = \text{init } f (\text{next} a) a
\]

We still have to make the wishes come true, and this involves finding a function \(f\) such that when \(a \neq \text{nil}\):

\[
a : \text{next} \ast b = f \ast a \\
\text{next} \ast \text{next} a = f \ast \text{next} a \\
P(\text{next}, a, b) \Rightarrow P(f, \text{next} a, a)
\]

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

\[
f \ast a \\
= \{\text{definition of } \ast\ \text{in case } a \neq \text{nil}\} \\
a : f \ast f a \\
= \{\text{setting } f = \text{next}[a := b], \text{ so } f \ a = b\} \\
a : f \ast b \\
= \{(1), \text{ assuming } a \neq \text{next} \ast b\}
\]
Richard S. Bird

\[ a : \text{next} \ast b \]

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

\[
\begin{align*}
  f \ast \text{next} a \\
  &= \{ \text{with } f = \text{next}[a := b] \} \\
  &= \text{next}[a := b] \ast \text{next} a \\
  &= \{ (1), \text{assuming } a \not\in \text{next} \ast (\text{next} a) \} \\
  &= \text{next} \ast \text{next} a
\end{align*}
\]

The requirements on \( P \) therefore take the form

\[
P(\text{next}, a, b) \land a \neq \text{Nil} \Rightarrow \\
  a \not\in \text{next} \ast b \land a \not\in \text{next} \ast (\text{next} a) \land P(\text{next}[a := b], \text{next} a, a)
\]

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

\[
P(\text{next}, a, b) \equiv ND(\text{next}, a) \land DJ(\text{next}, a, b)
\]

In words, \( \text{next} \ast a \) has no duplicated elements and no elements in common with \( \text{next} \ast b \). Clearly, \( DJ(\text{next}, a, \text{Nil}) \) holds.

In summary, we have shown that, provided \( ND(\text{next}, a) \),

\[
\text{reverse} (\text{next} \ast a) = f \ast b \quad \text{where } (f, b) = \text{loop} \text{ next} a \text{ Nil}
\]

and

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

Here is the definition of \( \text{loop} \text{ next} a \text{ Nil} \) again, written this time in an imperative style:

\[
\begin{align*}
b &:= \text{Nil}; \\
\text{do } a \neq \text{Nil} \rightarrow \\
  \text{next, } a, b &:= \text{next}[a := b], \text{next} a, a \\
\text{od;}
\end{align*}
\]

\[
\text{return} (\text{next}, b)
\]

Replacing \( \text{next} := \text{next}[a := b] \) by \( \text{next}[a] := b \) 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 \( \text{next} \ast a \) should not contain duplicates, not that it should be a finite list. To be sure, if \( \text{next} \ast a \) were not finite the code above would not terminate, but then neither would \text{revers} so the implementation is correct. If \( \text{next} \ast a \) did contain a duplicate, so was a cyclic list, the implementation above would terminate with an incorrect result.
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 $a \neq \text{Nil}$) under the abstraction mapping $\text{map data} \cdot (\text{next } \circ)$ where

$$f \circ x = f \circ (f x)$$

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

$$\text{next } \circ a \triangleright \text{next } \circ b = (\text{step next } \circ a \circ b) \circ a \quad ?$$

Our aim is to come up with the following definition of $\text{step}$:

$$\text{step next } a \circ b = \begin{cases} \text{if next } a = \text{Nil} \text{ then } \text{next[}a := \text{next } b\text{]} \\ \text{else step next } \circ (\text{next } a) \circ b \end{cases}$$

In an imperative idiom $\text{step}$ is implemented by the loop

$$\begin{align*}
x &:= a; \\
\text{do } &\text{next}[x] \neq \text{Nil} \rightarrow x := \text{next}[x] \text{ od; } \\
\text{next}[x] &:= \text{next}[b]; \\
\text{return } &\text{next}
\end{align*}$$

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

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

$$\begin{align*}
\text{next } \circ a &\triangleright \text{next } \circ b \\
&= \quad \{\text{definition of } \circ\} \\
[\ ] &\triangleright \text{next } \circ b \\
&= \quad \{\text{definition of } \triangleright\} \\
\text{next } \circ b &
\end{align*}$$

For the claim, we reason:

$$\begin{align*}
\text{next}[a := \text{next } b] \circ a &\\
&= \quad \{\text{definition of } \circ \text{ and } \text{next}[a := \text{next } b] \circ a = \text{next } b\} \\
\end{align*}$$
Richard S. Bird

\[
\text{next}[a := \text{next } b] \ast \text{next } b
\]
\[
= \begin{cases} (1), & \text{assuming } a \notin \text{next } \circ b \end{cases}
\]
\[
\text{next } \circ b
\]

Hence we can take \text{step next } a \ b = \text{next}[a := \text{next } b], \text{ provided that }
\[
P(\text{next } a, b) \land \text{next } a = \text{Nil} \Rightarrow a \notin \text{next } \circ b
\]

In the case \text{next } a \neq \text{Nil}, we argue:
\[
\text{next } \circ a = \text{next } \circ b
\]
\[
= \begin{cases} \text{definition of } \circ \text{ and } + \end{cases}
\]
\[
\text{next } a : (\text{next } \circ \text{next } a = \text{next } \circ b)
\]
\[
= \begin{cases} \text{induction, writing } f = \text{step next } (\text{next } a) \ b, \text{ assuming } P(\text{next } a, \text{next } a, b) \end{cases}
\]
\[
\text{next } a : f \circ \text{next } a
\]
\[
= \begin{cases} \text{assume } P(\text{next } a, b) \Rightarrow \text{next } a = f a \end{cases}
\]
\[
f \circ a
\]

We can therefore take \text{step next } a \ b = f, \text{ provided that }
\[
P(\text{next } a, b) \land \text{next } a \neq \text{Nil} \Rightarrow
\]
\[
\text{next } a = \text{step next } (\text{next } a) \ b \ a \land P(\text{next } a, \text{next } a, b)
\]

This gives the definition of \text{step} described above.
To see what \text{P} entails, observe from the definition of \text{step} and the assumption that \text{next } \circ a \text{ is a finite list, that }
\[
\text{next } a \neq \text{Nil} \Rightarrow \text{step next } (\text{next } a) \ b = \text{next}[x := \text{next } b]
\]
for some \( x \in \text{next } \circ a \). Since \( a \notin \text{next } \circ a \) (otherwise \( \text{next } \circ a \) is not finite), we obtain
\[
\text{step next } (\text{next } a) \ b \ a = \text{next } a
\]
as required. The minimum solution for
\[
P(\text{next } a, b) \land \text{next } a = \text{Nil} \Rightarrow a \notin \text{next } \circ b
\]
\[
P(\text{next } a, b) \land \text{next } a \neq \text{Nil} \Rightarrow P(\text{next } a, \text{next } a, b)
\]

turns out to be
\[
P(\text{next } a, b) \equiv (\forall k : \text{next}^{k+1} a = \text{Nil} \Rightarrow \text{next}^k \notin \text{next } \circ b)
\]

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

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
given by two functions \( \ell, r : A \rightarrow A \), short for left and right. Either \( \ell a \) or \( r a \) can be \( \text{Nil} \).

Our starting point is the following standard marking algorithm:

\[
\begin{align*}
\text{mark} (\ell, r, a) &= \text{mark1} (\ell, r, \text{const} 0, [a]) \\
\text{mark1} (\ell, r, m, []) &= (\ell, r, m) \\
\text{mark1} (\ell, r, m, a : \text{as}) &= \begin{cases} 
\text{if } a \neq \text{Nil} \land m a = 0 \\
\text{then } \text{mark1} (\ell, r, m[a := 1], \ell a : \text{r a : as}) \\
\text{else } \text{mark1} (\ell, r, m, \text{as})
\end{cases}
\end{align*}
\]

The result of \( \text{mark} (\ell, r, a) \) is a triple of functions \( (\ell, r, m) \) such that \( m b = 1 \) if \( b \) is reachable from \( a \), and \( m b = 0 \) otherwise. We also return the adjacency functions \( (\ell, r) \) 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 \( \text{mark1} \) is treated as a stack.

For the first step we transform \( \text{mark1} \) into a function \( \text{mark2} \) satisfying

\[
\text{mark2} (\ell, r, m, a, \text{as}) = \text{mark1} (\ell, r, m, a : \text{map r as})
\]

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

\[
\text{mark} (\ell, r, a) = \text{mark2} (\ell, r, \text{const} 0, a, [])
\]

Synthesizing a direct recursive definition of \( \text{mark2} \) leads quite easily to the following code:

\[
\text{mark2} (\ell, r, m, a, \text{as}) =
\begin{align*}
& a \neq \text{Nil} \land m a = 0 & \rightarrow & \text{mark2} (\ell, r, m[a := 1], \ell a, a : \text{as}) \\
& \text{null as} & \rightarrow & (\ell, r, m) \\
& \text{otherwise} & \rightarrow & \text{mark2} (\ell, r, m, r (\text{head as}), \text{tail as})
\end{align*}
\]

Note that arguments \( m \) and \( \text{as} \) of \( \text{mark2} \) satisfy the property that if \( x \in \text{as} \), 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 \( n : A \rightarrow A \) (short for next) and use the abstraction

\[
\text{stack} (n, b) = n * b
\]

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

\[
\text{stack} (n, s, b) = \text{filter} (\text{marked} s) (n * b)
\]

where \( \text{marked} s a = (s a = 1) \). This representation leads to the following implemen-
tions of the stack operations:

\[
\begin{align*}
    a : (n, s, b) &= (n[a := b], s[a := 1], a) \\
    \text{head} (n, s, b) &= \text{if } s \ b = 1 \text{ then } b \text{ else } \text{head} (n, s, n \ b) \\
    \text{tail} (n, s, b) &= \text{if } s \ b = 1 \text{ then } (n, s[\ b := 0], b) \text{ else } \text{tail} (n, s, n \ b)
\end{align*}
\]

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 \( \text{mark3} \), specified by

\[
\text{mark3} (\ell, r, m, n, a, b) = \text{mark2} (\ell, r, m, a, \text{stack} (n, s, b))
\]

In particular, we have

\[
\text{mark} (\ell, r, a) = \text{mark3} (\ell, r, \text{const} \ 0, \text{const} \ 0, \bot, a, \text{Nil})
\]

since the initial values of \( s \) and \( n \) are irrelevant. We choose, however, to initialise \( s \) to \( \text{const} \ 0 \) since that will also be the final value of \( s \).

Synthesizing a direct definition of \( \text{mark3} \) leads to

\[
\text{mark3} (\ell, r, m, n, s, a, b) =
\begin{align*}
    &a \neq \text{Nil} \land m \ a = 0 \quad \rightarrow \quad \text{mark3} (\ell, r, m[a := 1], n[a := b], s[a := 1], \ell \ a, a) \\
    &b = \text{Nil} \quad \rightarrow \quad (\ell, r, m) \\
    \text{otherwise} \quad &\rightarrow \quad \text{pop} (\ell, r, m, n, s, a, b)
\end{align*}
\]

where

\[
\text{pop} (\ell, r, m, n, s, a, b) =
\begin{align*}
    &s \ b = 1 \quad \rightarrow \quad \text{mark3} (\ell, r, m, n, s[b := 0], r \ b, b) \\
    &s \ b = 0 \quad \rightarrow \quad \text{pop} (\ell, r, m, n, s, b, n \ b)
\end{align*}
\]

Since we know that if \( b \) is on the stack, then \( b \neq \text{Nil} \land m \ b = 1 \), we can eliminate calls to \( \text{pop} \) and replace \( \text{mark3} \) with the simpler though marginally less efficient version

\[
\text{mark3} (\ell, r, m, n, s, a, b) =
\begin{align*}
    &a \neq \text{Nil} \land m \ a = 0 \quad \rightarrow \quad \text{mark3} (\ell, r, m[a := 1], n[a := b], s[a := 1], \ell \ a, a) \\
    &b = \text{Nil} \quad \rightarrow \quad (\ell, r, m) \\
    &s \ b = 1 \quad \rightarrow \quad \text{mark3} (\ell, r, m, n, s[b := 0], r \ b, b) \\
    &s \ b = 0 \quad \rightarrow \quad \text{mark3} (\ell, r, m, n, s, b, n \ b)
\end{align*}
\]

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 \( \ell \) and \( r \) fields instead. More precisely, the aim is to replace \( n \) by the function \( \text{next} (\ell, r, s) \) defined by

\[
\text{next} (\ell, r, s) = \lambda x. \text{if } s \ x = 1 \text{ then } \ell \ x \text{ else } r \ x
\]

As a result, we are left with providing just one extra marking function \( s \), and since
s requires a single bit per node rather than a full address, there is a significant saving in space.

The functions \( \ell \) 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 \( n \times b \). We claim that they can be restored to their original values by the function \( \text{restore} \), defined by

\[
\text{restore}(\ell, r, s, a, b) = \\
b = \text{Nil} \quad \rightarrow \quad (\ell, r) \\
s b = 1 \quad \rightarrow \quad \text{restore}(\ell[b := a], r, s, b, n b) \\
s b = 0 \quad \rightarrow \quad \text{restore}(\ell, r[b := a], s, b, n b)
\]

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

\[
\text{restore}(\ell, r, s, a, b) = (\ell_0, r_0)
\]

it is clear that \( \ell_0 x = \ell x \) and \( r_0 x = r x \) for all \( x \) not on the list \( n \times b \).

Now introduce \( \text{mark}4 \) defined by

\[
\text{mark}4(\ell, r, m, s, a, b) = \text{mark}3(\text{restore}(\ell, r, s, a, b), m, \text{next}(\ell, r, s), s, a, b)
\]

For syntactic accuracy the first two arguments of \( \text{mark}3 \) should have been paired, so assume they were. It is easy to show that

\[
\text{mark}(\ell, r, a) = \text{mark}4(\ell, r, \text{const} 0, \text{const} 0, \bot, a, \text{Nil})
\]

Our objective is to synthesize the following recursive definition of \( \text{mark}4 \):

\[
\text{mark}4(\ell, r, m, s, a, b) = \\
\begin{align*}
& a \neq \text{Nil} \land m a = 0 \rightarrow \text{mark}4(\ell[a := b], r, m[a := 1], s[a := 1], \ell a, a) \\
& b = \text{Nil} \rightarrow (\ell, r, m) \\
& s b = 1 \rightarrow \text{mark}4(\ell[b := a], r[b := \ell b], m, s[b := 0], r b, b) \\
& s b = 0 \rightarrow \text{mark}4(\ell, r[b := a], m, s, b, r b)
\end{align*}
\]

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 \( \text{mark}4 \) into an imperative loop and we won’t give details.

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

In the case \( a \neq \text{Nil} \land m a = 0 \) we argue:

\[
\begin{align*}
\text{mark}4(\ell, r, m, s, a, b) &= \\
&\{\text{definition of mark}4\} \\
&= \text{mark}3((\ell_0, r_0), m, s, n, a, b) \\
&\{\text{case assumption}\} \\
&= \text{mark}3((\ell_0, r_0), m[a := 1], s[a := 1], n[a := b], \ell_0 a, a) \\
&\{\text{claim}\}
\end{align*}
\]
\[
mark 4 (\ell[a := b], r, m[a := 1], s[a := 1], \ell a, a)
\]
The claim relies on three facts: if \( a \neq \text{Nil} \land m a = 0 \), then
\[
\ell_0 a = \ell a \quad (6)
\]
\[
(\ell_0, n_0) = \text{restore} (\ell[a := b], r, s[a := 1], \ell a, a) \quad (7)
\]
\[
n[a := b] = \text{next} (\ell[a := b], r, s[a := 1]) \quad (8)
\]
In the case \( b = \text{Nil} \) we argue:
\[
\begin{align*}
\mark 4 (\ell, r, m, s, a, b) &= \{ \text{definition of mark 4} \} \\
\mark 3 ((\ell_0, n_0), m, s, a, b) &= \{ \text{definition of mark 3 in the case } b = \text{Nil} \} \\
(\ell_0, n_0, m) &= \{ \text{definition of restore in the case } b = \text{Nil} \} \\
(\ell, r, m)
\end{align*}
\]
Similar calculations in the case \( b \neq \text{Nil} \land s b = 1 \) yields the desired result provided, in this case, that
\[
\begin{align*}
n_0 b &= r b \quad (9) \\
(\ell_0, r_0) &= \text{restore} (\ell[b := a], r[b := \ell b], s[b := 0], r b, b) \quad (10) \\
n &= \text{next} (\ell[b := a], r[b := \ell b], s[b := 0]) \quad (11)
\end{align*}
\]
Finally, in the case \( b \neq \text{Nil} \land s b = 0 \) we require
\[
\begin{align*}
(\ell_0, n_0) &= \text{restore} (\ell, r[b := a], s, b, r b) \quad (12) \\
n b &= r b \quad (13)
\end{align*}
\]
Now we must verify that these conditions hold. Equation (6) is immediate since \( m a = 0 \) implies \( a \neq n \) \( \land b \) and so \( \ell a = \ell_0 a \) and \( r a = n_0 a \). For (7) we argue:
\[
\begin{align*}
\text{restore} (\ell[a := b], r, s[a := 1], \ell a, a) &= \{ \text{definition of restore since } s[a := 1] a = 1 \} \\
\text{restore} (\ell[a := b][a := \ell a], r, s[a := 0], a, b) &= \{ \text{simplification and } m a = 0 \Rightarrow s a = 0 \} \\
\text{restore} (\ell, r, s, a, b)
\end{align*}
\]
For (8) we argue, writing \((p \rightarrow q, r)\) as shorthand for \( \text{if } p \text{ then } q \text{ else } r \):
\[
\begin{align*}
\text{next} (\ell[a := b], r, s[a := 1]) x &= \{ \text{definition of next} \} \\
(s[a := 1] x = 1 \rightarrow \ell[a := b] x, r x) &= \{ \text{definition of update} \} \\
(x = a \rightarrow b, (s x = 1 \rightarrow \ell x, r x)) &= \{ \text{definition of } n = \text{next} (\ell, r, s) \}
\end{align*}
\]
Functionals

\[ n[a := b] \]

For (9) we argue:

\[
\text{restore} (\ell, r, s, a, b) = \{\text{case assumption } s \ b = 1\} \\
\text{restore} (\ell[b := a], r, s[b := 0], b, r b)
\]

Now, since \( b \notin n \) we have \( l_0 b = l[b := a] b = a \) and \( n b = r b \).

For (10) we argue:

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

For (11) we argue:

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

For (12) we argue:

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

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 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


Schorr, H. and Waite, W. M. (1967) An efficient machine-independent procedure for
