State-based functional programming

Maarten Fokkinga, University of Twente, Dept INF

Version of July 14, 1993

The usual 'map f' is adapted so as to accept and produce a state value that is single-threaded through the computation.

This is an elaboration of Section 4 (3 pages; not including 4.1) of the paper *Type parametric programming with compile-time reflection* by Tim Sheard.

1 State-basedness. Under *state-based programming* we understand the situation that, for several functions, there is a distinguished parameter and result, called *the state*, that is passed from one function to another in such a way that, during the evaluation, the state is single-threaded through the computation. In an imperative program the state is implicit: it is the collection of all global variables. In a functional program the state has to be explicit; in this case state-based programming can quickly become very repetitious and tedious, if all manipulations with the state are expressed explicitly and "in-place". We demonstrate several ways of combining (composing) state-based functions in such a way that manipulating the state is expressed only once for each kind of composition (rather than for each use of such a composition). These ways are: St-composition, the state-based map, and the state-based catamorphism.

> *Notation.* We use the notation of *Law and Order in Algorithmics* [2], with the following exceptions: T varies over induced (= user-defined) type functors, we speak of 'types' and 'functions' rather than 'objects' and 'morphisms', and sometimes but not always we use a type name for the identity on that type. Recall that composition $;$ binds weakest of all operations.

2 Monads. As a preliminary we recall here the notion and use of monads. *This is my own view of monads, not the view expressed by Sheard.* Let F be a functor. Suppose we wish to compose functions of type $a \to Fb$, for varying types a and b. Let us call such functions F*-resultric*. The obvious way to do so, is to define the desired 'F*-composition*' $;_F$ by:

$$f \;;_F\; g \quad = \quad f \;;\; Fg \;;\; \mu \,,$$

where $\mu\colon\ FF \dot{\to} F$ is supposed to exist. This is *the* obvious definition, since it meets the typing requirement:

$$f\colon\ a \to Fb, \quad g\colon\ b \to Fc \quad \Rightarrow \quad f \,;_F\, g\colon\ a \to Fc\,.$$

Of course, one might wish the F-composition to be associative with some $\eta\colon\ I \dot{\to} F$ as unit. Given the definition above, that wish is exactly equivalent to the requirement that (F, η, μ) is a monad, that is, η and μ satisfy the following three so-called monad laws:

$$\begin{array}{lcl} \eta \,;\, \mu & = & id \quad = \quad F\eta \,;\, \mu \\ F\mu \,;\, \mu & = & \mu \,;\, \mu\,. \end{array}$$

Proof: easy; left to the reader. (The category with F-resultric functions as morphisms, η as identity, and $;_F$ as composition, is known as the *Kleisli* category.) If there exists a monad with F as functor, we say that F is *monadic*. For later use we define the lifting operation *lift* by:

$$\mathit{lift}\, f \quad = \quad f \,;\, \eta \quad : \quad a \to Fb \quad \text{for } f\colon\ a \to b\,.$$

The lift operation makes an arbitrary function into an F-resultric one in a trivial way.

To give just one example of a monad, and the usefulness of the new kind of composition, consider lists. Let $\mathit{Lst}\, a$ denote the set of lists over a, and let $\mathit{Lst}\, f$ denote the usual '$\mathit{map}\, f$'. Then $(F, \eta, \mu) := (\mathit{Lst}, [\_], \mathit{concat})$ is a monad, and list comprehension can be expressed by F-composition:

$$\begin{array}{lcl} f \,;_F\, g & = & x \mapsto [z \mid\ y \leftarrow fx \,;\, z \leftarrow gy] \\ \eta & = & x \mapsto [x]\,. \end{array}$$

Actually, reading from right to left we can take these equations as *defining* a comprehension notation in terms of F-composition, for arbitrary monad (F, η, μ). This is the proposal of Wadler in *Comprehending Monads* [3].

There are some nice algebraic properties involving $;_F$ and *lift*. For example:

$$\begin{array}{lcl} (f \,;\, g) \,;_F\, h & = & f \,;\, (g \,;_F\, h) \\ f \,;\, \mathit{lift}\, g & = & \mathit{lift}\,(f \,;\, g) \\ f \,;_F\, \mathit{lift}\, g & = & f \,;\, Fg\,, \end{array}$$

and presumably many more.

3 State-based functions. Let *state* be a type, fixed for the sequel. A function is called *state-based* if: it has type

$$a \times \mathit{state} \to b \times \mathit{state}\,,$$

for some types a, b. Defining

$$\mathit{St}\, x \quad = \quad \mathit{state} \to x \times \mathit{state}\,,$$

we find that state-based functions are St-resultric:

$$\begin{array}{l} a \times \mathit{state} \to b \times \mathit{state} \\ \cong \\ a \to (\mathit{state} \to b \times \mathit{state}) \\ = \\ a \to \mathit{St}\, b. \end{array}$$

So, state-based functions may be composed by St-composition, if St is monadic. This happens to be the case, and, moreover, by St-composition the state is single-threaded through the computation. The proof follows.

4 The St-monad. First, observe that the definition of St, namely

$$\mathit{St}\, x \quad = \quad \mathit{state} \to x \times \mathit{state}\,,$$

not only applies to types x but also applies to functions x by our conventions:

- the two-place mapping $\_ \to \_$ is defined as follows:

$$\begin{array}{lcl} a \to b & = & \{h \mid\ h\colon\ a \to b\} \\ f \to g & = & h \mapsto f \mathbin{;} h \mathbin{;} g \end{array}$$

- a type in the position of a function stands for the identity on that type;

- for functors F, G and infix-written bifunctor $\dagger$ (like $\times$ and $\to$), the functor $F \dagger G$ is defined by $(F \dagger G)x = Fx \dagger Gx$.

(Actually, $\to$ is the well-known Hom-functor, contravariant in its first parameter. Since the parameter of St only occurs in the target side of the Hom-functor, mapping St itself is a proper functor.) Written out in detail the action of St on functions reads:

$$\begin{array}{lcl} \mathit{St}\, f & = & g \mapsto \mathit{id} \mathbin{;} g \mathbin{;} f \times \mathit{id} \\ & = & g \mapsto \quad s \mapsto (fy,\ t) \text{ where } (y, t) = gs \\ & : & \mathit{St}\, a \to \mathit{St}\, b \qquad \text{for } f\colon\ a \to b\,. \end{array}$$

The reader may wish to verify that $\mathit{St}\, f \mathbin{;} \mathit{St}\, g = \mathit{St}\,(f \mathbin{;} g)$ and $\mathit{St}\, \mathit{id} = \mathit{id}$, hence St is a functor.

Second, St is monadic, by defining:

$$\begin{array}{lcl} \eta\, x & = & s \mapsto (x, s) \\ \mu\, f & = & s \mapsto g\, s' \text{ where } (g, s') = f\, s\,. \end{array}$$

We leave it to the reader to check that these are natural transformations and satisfy the monad laws. (Operations η and μ may be defined at the function level in terms of the pairing function *pair*, the application function *apply*, and currying.)

Third, by St-composition the state is single-threaded through the computation:

$$
\begin{array}{lcl}
(f \mathbin{;_{\mathit{St}}} g)\, x_0 & = & s_0 \mapsto (x_2, s_2) \\
 & & \text{where} \quad \begin{array}{lcl} (x_1, s_1) & = & f\, x_0\, s_0 \\ (x_2, s_2) & = & g\, x_1\, s_1 \,. \end{array}
\end{array}
$$

So one can avoid the tedious and error-prone explicit state manipulations of the above right-hand side by simply using St-composition.

When defining other combinations of state-based functions, single-threadedness is preserved if such functions are composed by means of St-composition only.

5 State-based map. The state-based map for lists is defined thus:

$$
\begin{array}{lcl}
\mathit{Lst}_{\mathit{St}}\, f\, [x_0, \ldots, x_{n-1}] & = & s_0 \mapsto ([y_0, \ldots, y_{n-1}], s_n) \\
 & & \text{where} \quad \begin{array}{lcl} (y_0, s_1) & = & f\, x_0\, s_0 \\ \vdots & & \\ (y_{n-1}, s_n) & = & f\, x_{n-1}\, s_{n-1} \,. \end{array}
\end{array}
$$

Notice that, indeed, the state is single-threaded through the computation. We wish to define a state-based map for arbitrary datatype T rather than the specific datatatype Lst of lists.

> Recall that with our notational conventions symbol $\dagger$ will stand for an infix-written bifunctor (the E in Tim Sheard's papers), and α will be an initial algebra: the $\triangledown$-combination of all the constructors of the datatype (for example, $\alpha = \mathit{nil} \triangledown \mathit{cons}\colon\ 1 + a \times \mathit{Lst}\, a \to \mathit{Lst}\, a$).

Let $\dagger$ be a bifunctor. Let type functor T and algebra α be induced by $\dagger$, that is, for each type a the algebra

$$\alpha \quad : \quad a \dagger Ta \to Ta$$

is the initial $(a\dagger)$-algebra, with Ta as its carrier. Mapping T is made into a functor by defining, for $f\colon\ a \to b$,

$$Tf \quad = \quad (\!|f \dagger \mathit{id} \mathbin{;} \alpha|\!) \quad : \quad Ta \to Tb \,.$$

Now we wish to define a state-based map, denoted T_{St}, that acts on state-based functions, and is similar to T except that the extra state parameter is passed through the computation as suggested in the definition of the state-based map for lists, $\mathit{Lst}_{\mathit{St}}$. So the typing requirement reads:

$$T_{\mathit{St}}\, f \quad : \quad Ta \to \mathit{St}\, Tb \qquad \text{for } f\colon\ a \to \mathit{St}\, b \,.$$

Compare this with the typing of the normal map when applied to state-based functions:

$$Tf \quad : \quad Ta \to T\,St\,b \qquad \text{for } f\colon\ a \to St\,b\,.$$

So, one might wish to define $T_{St} f = Tf \,;\, zip_T$, for some suitable natural transformation $zip_T\colon\ T\,St \dot{\to} St\,T$. It is not to be expected that such a zip_T exists for arbitrary T. In the relational framework Backhouse et al. [1, Theorem 41] construct a $zip_{F,G}\colon\ FG \dot{\to} GF$ for a class of functors (thereby generalising a theorem of Fokkinga [2, Section 4c]). In the construction of $zip_{F,G}$ they require that G belongs to a certain subclass of the regular functors; this seems not to be the case for $G := St$. A direct construction of zip_F for regular F will succeed, as we shall see shortly, but is unsatisfactory *here*, since zip_T is defined in terms of T_{St} and so cannot be used to define T_{St}. We must avoid this lurking circularity.

> *The name zip.* A natural transformation of type $FG \dot{\to} GF$ may naturally be called *comm*, since it commutes the functors. Alternatively, *distr* would also be a reasonable name, since for binary F the type should read $F(G,G) \dot{\to} GF$ and the transformation deals with the (backwards) distribution of G over the arguments of F. The name *zip* originates with the particular case where $F, G = (\times), Lst$, transforming pairs of lists into a list of pairs. We stick to the name *zip* since it is so short.

Let us try and construct T_{St} directly. Type considerations alone suggest a satisfactory definition. Let $f\colon\ a \to St\,b$ be arbitrary. Then, using φ, ψ, χ, zip as auxiliary unknows:

$$\begin{array}{ll}
 & T_{St}\, f\colon\ Ta \to St\,Tb \\
\equiv & \quad \{\text{guess } T_{St}\, f \text{ is a catamorphism on } Ta \text{ (what else?)}\,\} \\
 & \quad \textbf{put}\ \ T_{St}\, f = (\!|\varphi|\!) \\
 & (\!|\varphi|\!)\colon\ Ta \to St\,Tb \\
\Leftarrow & \quad \text{cata-}\textsc{Type} \\
 & \varphi\colon\ a \dagger St\,Tb \to St\,Tb \\
\equiv & \quad \{\text{guess } \varphi = \ldots \,;\, St\,\alpha \text{ since the result type matches}\,\} \\
 & \quad \textbf{put}\ \ \varphi = \psi \,;\, St\,\alpha \\
 & \psi \,;\, St\,\alpha\colon\ a \dagger St\,Tb \to St\,Tb \\
\Leftarrow & \quad \text{typing } \alpha \text{ and } St \\
 & \psi\colon\ a \dagger St\,Tb \to St\,(b \dagger Tb) \\
\equiv & \quad \{\text{guess } \psi = f \dagger ... \,;\, \ldots \text{ since } f \text{ has to be used on } a\ \} \\
 & \quad \textbf{put}\ \ \psi = f \dagger \chi \,;\, zip \\
 & f \dagger \chi \,;\, zip\colon\ a \dagger St\,Tb \to St\,(b \dagger Tb) \\
\Leftarrow & \quad \text{typing rules, and } f\colon\ a \to St\,b
\end{array}$$

$$
\begin{array}{ll}
 & id \dagger \chi \,;\, zip\colon\; St\, b \,\dagger\, St\, Tb \to St\,(b \dagger Tb) \\
\equiv & \qquad \{\text{guess and}\}\;\; \mathbf{put}\;\; \chi = id \\
 & zip\colon\; St\, b \,\dagger\, St\, Tb \to St\,(b \dagger Tb) \\
\Leftarrow & \qquad \{\text{guess } zip \text{ is independent of } T \text{ (and natural!)}\} \\
 & zip\colon\; St\, a \dagger St\, b \to St\,(a \dagger b) \qquad \text{for all types } a, b.
\end{array}
$$

So, provided that such a zip exists, the typing requirement is met if we define:

$$T_{St}\, f \quad = \quad \llparenthesis f \dagger id \,;\, zip \,;\, St\, \alpha \rrparenthesis \,.$$

Notice the similarity with the normal map Tf:

$$Tf \quad = \quad \llparenthesis f \dagger id \,;\, \alpha \rrparenthesis \,.$$

It remains to construct zip, to prove that the specialisation to lists gives the Lst_{St} discussed above, and to argue (or prove) that the state is single-threaded through the computation. We'll discuss these in that order.

6 Construction of zip. The task is to construct a ntrf $zip\colon St\, a \dagger St\, b \to St\,(a \dagger b)$, for arbitrary bifunctor $\dagger$. Well, again, this is impossible in general, but for *regular* mono- and bifunctors F we'll succeed in defining a whole class of transformations:

$$
\begin{array}{lcll}
zip_F & : & F\, St\, a \to St\, F\, a & \text{for 1-ary } F \\
zip_F & : & F\,(St\, a,\, St\, b) \to St\, F\,(a, b) & \text{for 2-ary } F\,,
\end{array}
$$

by induction on the formation of functor F. Notice that, in the formation of T according to the grammar below, $\dagger$ is going to be a subterm of T, and so the circularity mentioned above is avoided. The regular 1-ary and 2-ary functors are the functors produced by the following grammar:

$$F \quad ::= \quad Exl \;\mid\; Exl \;\mid\; \underline{a} \;\mid\; F + F \;\mid\; F \times F \;\mid\; T\,,$$

where a is an arbitrary type, T is a "previously defined" type functor (that is, induced by some 2-ary F from the grammar), and, according to our notational conventions [2]:

$$
\begin{array}{lcll}
Exl & = & (x, y) \mapsto x & \text{2-ary functor} \\
Exr & = & (x, y) \mapsto y & \text{2-ary functor} \\
\underline{a} & = & x \mapsto a & \text{at a function position, } a \text{ denotes } id_a \\
F + G & = & x \mapsto Fx + Gx & \\
F \times G & = & x \mapsto Fx \times Gx\,. &
\end{array}
$$

For example, the bifunctor $\dagger$ for lists, namely:

$$x \dagger y \quad = \quad 1 + x \times y\,,$$

fits the above grammar by writing it as:

$$\begin{array}{lll} \dagger & = & (x,y) \mapsto \underline{1}(x,y) + \mathit{Exl}\,(x,y) \times \mathit{Exr}\,(x,y) \\ & = & (x,y) \mapsto (\underline{1} + \mathit{Exl} \times \mathit{Exr})(x,y) \\ & = & \underline{1} + \mathit{Exl} \times \mathit{Exr}\,. \end{array}$$

As a preparation we first define for each operator $\oplus \in \{+, \times\}$ a ntrf $zip_\oplus$ with type

$$zip_\oplus \quad : \quad \mathit{St}\,a \oplus \mathit{St}\,b \to \mathit{St}\,(a \oplus b)\,.$$

Function $zip_\times$ is the $\tau_{2,1}^{\mathit{St}}$ in Sheard's paper. Here are the two definitions:

$$\begin{array}{lll} zip_\times & : & \mathit{St}\,a \times \mathit{St}\,b \to \mathit{St}(a \times b) \\ zip_\times\,(f,g) & = & f \,;\, id \times g \,;\, assoc \\ & & \text{where } \; assoc\colon\; a \times (b \times c) \to (a \times b) \times c \; \text{ is obvious} \end{array}$$

that is,

$$\begin{array}{lll} zip_\times\,(f,g) & = & s \mapsto ((x,y),r) \\ & & \text{where } \begin{array}{lll} (x,t) & = & f\,s \\ (y,r) & = & g\,t \end{array} \end{array}$$

and:

$$\begin{array}{lll} zip_+ & : & \mathit{St}\,a + \mathit{St}\,b \to \mathit{St}(a+b) \\ zip_+ & = & \mathit{St}\,\mathit{inl} \triangledown \mathit{St}\,\mathit{inr} \end{array}$$

that is,

$$\begin{array}{lll} zip_+\,(\mathit{inl}\,f) & = & id \,;\, f \,;\, \mathit{inl} \times id \\ & = & s \mapsto (\mathit{inl}\,x, t) \quad \text{where } (x,t) = f\,s \\ zip_+(\mathit{inr}\,g) & = & id \,;\, g \,;\, \mathit{inr} \times id \\ & = & s \mapsto (\mathit{inr}\,x, t) \quad \text{where } (x,t) = g\,s\,. \end{array}$$

Now we define the required functions zip_F by induction on the formation of F:

$$\begin{array}{llll} zip_{\mathit{Exl}} & = & id & \text{2-ary case} \\ zip_{\mathit{Exr}} & = & id & \text{2-ary case} \\ zip_{\underline{a}} & = & \eta & \\ zip_{F \oplus G} & = & zip_F \oplus zip_G \,;\, zip_\oplus & \\ zip_T & = & T_{\mathit{St}}\,id & \text{here } id\colon\; \mathit{St}\,a \to \mathit{St}\,a\,. \end{array}$$

This completes the construction of the functions zip_F. The required zip is now $zip_\dagger$ (where $\dagger$ is the regular bifunctor that induces T). One may notice that the definition of zip immediately generalises to an arbitrary monadic functor M instead of St, provided that the two special transformations $zip_\oplus\colon\; Ma \oplus Mb \to M(a \oplus b)$ exist. So, for any such monad M, we have shown in fact that each regular type functor T comes equiped with a map T_M for M-resultric functions.

As an example, we work out the definition of zip for lists. Recall that the bifunctor for lists may be written: $\dagger = \underline{1} + Exl \times Exr$. Hence the zip reads:

$$
\begin{aligned}
& zip_{\underline{1} + Exl \times Exr} \\
= \;& zip_{\underline{1}} + zip_{Exl \times Exr} \,;\, zip_{+} \\
= \;& \eta + (id \times id \,;\, zip_{\times}) \,;\, St\, inl \triangledown St\, inr \\
= \;& (\eta \,;\, St\, inl) \triangledown (zip_{\times} \,;\, St\, inr).
\end{aligned}
$$

(*Surprise, surprise.* The dual of zip_{F+G} occurs in the Banana Split law [2, page 88] within the right-hand side by taking $\varphi, \psi, F := zip_F, zip_G, St$:

$$
\begin{array}{lcll}
(\!|\varphi|\!)_F \vartriangle (\!|\psi|\!)_F & = & (\!|F\, exl \vartriangle F\, exr \,;\, \varphi \times \psi|\!) & \textsc{Banana Split} \\
zip_{F+G} & = & zip_F + zip_G \,;\, St\, inl \triangledown St\, inr\, . &
\end{array}
$$

Indeed, Backhouse et al. [1, Section 10] give some relationship between zips for $+$ and $\times$, and the Banana Split and other laws.

7 Specialisation to lists. Specialising the above T_{St} to lists, that is, taking T and α the type functor and algebra induced by bifunctor $x \dagger y = 1 + x \times y$, we get exactly Lst_{St} :

$$
\begin{aligned}
& T_{St}\, f \\
= \;& \qquad \text{definition state-based map} \\
& (\!|f \dagger id \,;\, zip \,;\, St\, \alpha|\!) \\
= \;& \qquad \text{definition } \dagger \text{, equation for } zip \text{ above} \\
& (\!|id_1 + f \times id \,;\, (\eta \,;\, St\, inl) \triangledown (zip_{\times} \,;\, St\, inr) \,;\, St\, \alpha|\!) \\
= \;& \qquad \text{property sum} \\
& (\!|(id \,;\, \eta \,;\, St\, inl \,;\, St\, \alpha) \triangledown (f \times id \,;\, zip_{\times} \,;\, St\, inr \,;\, St\, \alpha)|\!) \\
= \;& \qquad \text{identity, functoriality } St \text{, and } nil \triangledown cons = \alpha \\
& (\!|(\eta \,;\, St\, nil) \triangledown (f \times id \,;\, zip_{\times} \,;\, St\, cons)|\!) \\
= \;& \qquad \text{naturality } \eta\colon I \dot{\to} St \text{, definition } lift\, f = f \,;\, \eta \\
& (\!|lift\, nil \triangledown (f \times id \,;\, zip_{\times} \,;\, St\, cons)|\!).
\end{aligned}
$$

That is,

$$
\begin{array}{lcl}
(T_{St}\, f)\, nil & = & s \mapsto (nil, s) \\
(T_{St}\, f)\, cons(x, xs) & = & f\, x \,;\, id \times (T_{St}\, f)\, xs \,;\, assoc \,;\, cons \times id\, ,
\end{array}
$$

which agrees precisely with the informal definition of $Lst_{St}\, f$ given above. The term $cons \times id$, originating from $St\, \alpha$, is the *push cons* of Sheard.

8 Single-threadedness. Since I've no formal definition of single-threadedness, I cannot give a formal proof. A formal definition would require a notion of 'evaluation' or 'reduction', and this I don't want to give here: too complicated. So, let us argue informally that $T_{St}\, f$ passes the state in the right way (if f does so).

Each clause in the definition of zip seems to preserve single-threadedness, and so do the two compositions in $(\!| f \dagger id \,;\, zip \,;\, St\,\alpha |\!)$. For the latter you may notice that $zip \,;\, St\,\alpha = zip \,;_{St}\, lift\,\alpha$, as mentioned in text 2. Finally, each of the clauses in the grammar for regular bifunctors denotes a functor that only collects state-based functions, but doesn't compose them, so that single-threadedness is not invalidated. (It is the zip operation that actually composes state-based functions.)

References

[1] R. Backhouse, H. Doornbos, and P. Hoogendijk. A class of commuting relators. In *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*. Utrecht University, dept Computer Science, 1992.

[2] M.M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, dept Comp Sc, Enschede, The Netherlands, 1992.

[3] P. Wadler. Comprehending monads. In *ACM Conference on Lisp and Functional Programming*, June 1990.