Category-Theoretic Foundations of "STCLang: State Thread Composition as a Foundation for Monadic Dataflow Parallelism"

SEBASTIAN ERTEL*, Huawei Technologies, Germany
JUSTUS ADAM, Technische Universität Dresden, Germany
NORMAN A. RINK, Technische Universität Dresden, Germany
ANDRÉS GOENS, Technische Universität Dresden, Germany
JERONIMO CASTRILLON, Technische Universität Dresden, Germany

This manuscript gives a category-theoretic foundation to the composition of State Threads as a Foundation for Monadic Dataflow Parallelism. It serves as a supplementary formalization of the concepts introduced in the Article "STCLang: State Thread Composition as a Foundation for Monadic Dataflow Parallelism", as published in the Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell (Haskell'19) [1].

1 CATEGORY-THEORETICAL FOUNDATION FOR STATE THREADS

This manuscripts develops in some detail a formalization of state threads in `STCLang` [1]. Our development relies on the formalism of category theory. This manuscript aims to serve as supplementary material for [1], and presumes familarity with concepts presented therein.

The two key ideas underlying `STCLang` are that (1) each state thread operates on its own private state, and (2) the composition of state threads retains enough information to extract parallelism from composed state threads. Once these ideas have been made precise, they naturally lead to the introduction of the `smap` functor, which generalizes `map` to situations where state must be kept track of. The `smap` functor introduces enough structure into our state threads to let us extract (pipeline) parallelism. We also identify other structures in state threads that are inherently parallel.

1.1 Foundations

`STCLang` is a typed λ-calculus extended with state threads. The details of the λ-calculus are not important, and almost any typed λ-calculus can be augmented with state threads to yield an implementation of `STCLang`. For our formal model of state threads presented in this section it is only relevant that the semantics of the λ-calculus can be interpreted in category-theoretic terms.

Let $\mathscr{H}$ be the category whose objects $obj(\mathscr{H})$ are the types in the λ-calculus and whose morphisms $morph(\mathscr{H})$ are the functions of the λ-calculus. The category $\mathscr{H}$ is required to be *cartesian closed*, which essentially means that for any types $a, b \in obj(\mathscr{H})$, the product type $a \times b$ and the function type $a \to b$ exist, i.e. $a \times b \in obj(\mathscr{H})$ and $a \to b \in obj(\mathscr{H})$. Examples of cartesian closed categories are the categories of domains typically encountered in denotational semantics.

In more concrete terms, since most functional programming languages are fancy λ-calculi, `STCLang` can be built on top of almost any functional language. In the case of Haskell, for example, the category $\mathscr{H}$ is known as *Hask*.[1]

*Work done while at TU Dresden.

[1] See https://wiki.haskell.org/Hask, although full *Hask* is not cartesian closed, and may in fact not even be a category (cf. http://math.andrej.com/2016/08/06/hask-is-not-a-category/).

1.2 State threads

In STCLang, every state thread has its own private state that it operates on. Hence, state threads and their respective states are both indexed by the same index set, henceforth denoted as N. In practice, N is typically finite, but it is generally sufficient to assume that N is countable, i.e. $N \cong \mathbb{N}$.

For the formal development of STCLang in the present section, it is convenient to require not only that each state thread has its own state, but also that every state is of a distinct type. Types are objects in the category $\mathscr{H}$, leading to the following definition.

Definition 1.1 (State objects, global state). Let N be a countable index set. For $n \in N$, let $s_n \in obj(\mathscr{H})$ be pairwise distinct (i.e. $s_n = s_m \Rightarrow n = m$).

(1) For $I \subseteq N$, define $s_I = \prod_{n \in I} s_n$. The s_I are called state objects.
(2) The state objects s_n, for $n \in N$, are called fundamental.
(3) The state object $s_N = \prod_{n \in N} s_n$ is called the global state.

Note that $s_{\{n\}} = \prod_{m \in \{n\}} s_m = s_n$, $n \in N$, i.e. the fundamental state objects are precisely the state objects s_I for which $I \subseteq N$ has cardinality 1. We also use the convention $s_\emptyset = ()$, i.e. the unit type.

The requirement that the s_n be pairwise distinct is not a restriction of STCLang's programming model. In Haskell, one can use the `newtype` keyword to generate new and distinct types. Typically, λ-calculi with less advanced type systems also offer ways of constructing new types in similar ways, e.g. by suitably tagging types.

Having introduced state objects, we can now define STCLang's state threads. It is then readily seen that state threads form a subcategory of $\mathscr{H}$, which we refer to as the *category of sate threads*.

Definition 1.2 (State thread). Let $\{s_n\}_{n \in N}$ be the set of fundamental state objects. A state thread is a morphism $f \in morph(\mathscr{H})$ such that

$$f : (a \times s_I) \to (b \times s_I)\,, \tag{1}$$

where $I \subseteq N$. A fundamental state thread is a state thread $f : (a \times s_n) \to (b \times s_n)$, i.e. a state thread for which $I = \{n\}$, $n \in N$, in Equation (1).

Lemma 1.3. *The following define the objects and morphisms of a subcategory $\mathscr{S}$ of $\mathscr{H}$,*

$$obj(\mathscr{S}) = \{a \times s_I \mid a \in obj(\mathscr{H}), I \subseteq N\}\,, \tag{2}$$

$$morph(\mathscr{S}) = \{f : (a \times s_I) \to (b \times s_I) \mid f \in morph(\mathscr{H}), I \subseteq N\}\,. \tag{3}$$

Proof. Clearly, $\mathrm{id}_{a \times s_I} \in morph(\mathscr{S})$. $\mathscr{S}$ inherits composition of morphisms from $\mathscr{H}$. Now, let $f, g \in morph(\mathscr{S})$. Whenever $g \circ f$ is defined in $\mathscr{H}$, then $g \circ f \in morph(\mathscr{S})$ follows directly by inspecting the signatures of f, g, and $g \circ f$. □

Definition 1.4 (Category of state threads). The category $\mathscr{S}$ from Lemma 1.3 is called the category of state threads.

The intuition is that the function that corresponds to the state thread $f : (a \times s_I) \to (b \times s_I)$ in the underlying λ-calculus only manipulates the part s_I of the global state s_N, $I \subseteq N$. The proof of Lemma 1.3 relies on the fact that state threads $f : (a \times s_I) \to (b \times s_I)$ and $g : (b \times s_J) \to (c \times s_J)$ can be composed (in $\mathscr{H}$ or $\mathscr{S}$) if and only if $I = J$. (This observation relies on the pairwise distinctness of the $\{s_n\}_{n \in N}$.) In the intuition just given, this means that f and g operate on the exact same part of the global state. Without additional information about the structure of f and g, an implementation of STCLang is then forced to evaluate the composition $g \circ f$ sequentially. However, an implementation can potentially exploit parallelism if $I \cap J = \emptyset$, i.e. when f and g operate on disjoint parts of the global state. The next section explains how STCLang facilitates the composition of state threads $f : (a \times s_I) \to (b \times s_I)$ and $g : (b \times s_J) \to (c \times s_J)$ with arbitrary $I, J \subseteq N$.

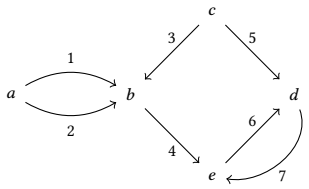

Fig. 1. Example of a multi-graph $\Delta_{\mathscr{M}}$ of fundamental state threads for $N = \{1, \ldots, 7\}$.

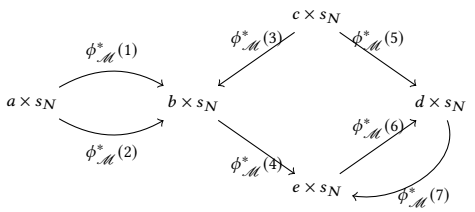

Fig. 2. The corresponding subcategory $\mathscr{C}_{\mathscr{M}}$ in $\mathscr{S}$.

1.3 Composition of state threads

At a high level, STCLang programs are composed of state threads, and compositions can ultimately be broken down into fundamental state threads. From now on, we assume that the fundamental state threads that occur in a given STCLang program are in 1-1 correspondence with the index set N. The following definition introduces the symbol $\mathscr{M}$ to refer to the set of fundamental state threads in a program, i.e. the *state threads of interest*.

Definition 1.5 (Fundamental state threads of interest). Let N be an index set and let $\{s_n\}_{n \in N}$ be the (pairwise distinct) fundamental state objects, as in the previous section. Let $\mathscr{M} \subseteq morph(\mathscr{S})$, and assume there is a bijective map $\phi_{\mathscr{M}} : N \to \mathscr{M}$ (i.e. a 1-1 correspondence) such that

$$\phi_{\mathscr{M}}(n) : (a_n \times s_n) \to (b_n \times s_n)\,, \tag{4}$$

where $a_n, b_n \in obj(\mathscr{H})$. Then the elements of $\mathscr{M}$ are the fundamental state threads of interest.

STCLang handles state implicitly. This motivates the organization of the fundamental state threads in $\mathscr{M}$ into a graph that hides the state objects but makes the possibility of composition explicit.

Definition 1.6 (Multi-graph of fundamental state threads). Let $\mathscr{M}$ and $\phi_{\mathscr{M}}$ as in Definition 1.5. The directed (multi-)graph $\Delta_{\mathscr{M}}$ has the following vertices (V) and edges (E),

$$V(\Delta_{\mathscr{M}}) = \{a_n, b_n \mid \phi_{\mathscr{M}}(n) : (a_n \times s_n) \to (b_n \times s_n), n \in N\}\,, \tag{5}$$

$$E(\Delta_{\mathscr{M}}) = N\,, \tag{6}$$

and the maps $src, tgt : E(\Delta_{\mathscr{M}}) \to V(\Delta_{\mathscr{M}})$ are defined as follows,

$$src(n) = a_n, \text{ if } \phi_{\mathscr{M}}(n) : (a_n \times s_n) \to (b_n \times s_n)\,, \tag{7}$$

$$tgt(n) = b_n, \text{ if } \phi_{\mathscr{M}}(n) : (a_n \times s_n) \to (b_n \times s_n)\,. \tag{8}$$

Note that directed multi-graphs are also referred to as *quivers* in the literature. Also note that in the light of Equations (7) and (8), the signature of $\phi_{\mathscr{M}}(n)$ can be written without referring to the (arbitrary) objects a_n, b_n:

$$\phi_{\mathscr{M}}(n) : (src(n) \times s_n) \to (tgt(n) \times s_n)\,. \tag{9}$$

Figure 1 gives an example of a multi-graph $\Delta_{\mathscr{M}}$ for seven fundamental state threads. Note how the state objects do not appear explicitly; they are, however, implicit in the naming of the edges. By contrast, composition of the state threads $\phi_{\mathscr{M}}(m)$ and $\phi_{\mathscr{M}}(n)$ is explicitly suggested whenever either $tgt(n) = src(m)$ or $tgt(m) = src(n)$.

The composition of state threads is natural in $\mathscr{S}$, and we would like to use this composition also for the state threads in $\mathscr{M}$. To facilitate this, we now construct a suitable embedding of the graph $\Delta_{\mathscr{M}}$ into the category $\mathscr{S}$. Our language is deliberately imprecise here to convey the right intuition. More correctly, we should speak of embedding $\Delta_{\mathscr{M}}$ into the graph underlying $\mathscr{S}$. Said yet another way, we are looking for a way to identify the free category over $\Delta_{\mathscr{M}}$ inside $\mathscr{S}$; and this is precisely what is achieved by the remaining definitions and lemma in the present section.

Definition 1.7 (Free category over a graph). The free category $\mathscr{F}(\Delta_{\mathscr{M}})$ over $\Delta_{\mathscr{M}}$ is the category whose objects are the vertices of $\Delta_{\mathscr{M}}$ and whose morphisms are precisely the paths in $\Delta_{\mathscr{M}}$, i.e.

$$obj(\mathscr{F}(\Delta_{\mathscr{M}})) = V(\Delta_{\mathscr{M}}), \tag{10}$$

$$morph(\mathscr{F}(\Delta_{\mathscr{M}})) = \{n_k n_{k-1} \dots n_2 n_1 \mid k \in \mathbb{N}, n_i \in N, tgt(n_i) = src(n_{i+1}) \text{ for } 1 \leq i \leq k-1\} \cup \{\epsilon_v \mid v \in V(\Delta_{\mathscr{M}})\}\,. \tag{11}$$

This definition of the free category over a graph is completely standard. Note that we take a separate copy of the empty path ϵ for each vertex v of $\Delta_{\mathscr{M}}$. In categorical terms, $\epsilon_v : v \to v$ is the identity morphism at the object v. The morphisms $morph(\mathscr{F}(\Delta_{\mathscr{M}}))$ can be thought of as words over the alphabet N. In the following, we adopt this point of view. Note that there is then a separate copy of the empty word for each vertex v of $\Delta_{\mathscr{M}}$.

By the universal property of the product, any state thread $f : (a \times s_I) \to (b \times s_I)$, with $I \subseteq N$, has a natural (and unique) extension to a state thread that operates on the global state s_N.

Definition 1.8 (Extension of state threads). Let $f : (a \times s_I) \to (b \times s_I)$ be a state thread. The state thread $f^* : (a \times s_N) \to (b \times s_N)$ is obtained from f by extending f with the identity on $s_{N \setminus I}$.

Using this extension of state threads to the global state s_N, we can finally define the functor that identifies the graph $\Delta_{\mathscr{M}}$ inside the category of state threads.

Definition 1.9. The functor $\Phi_{\mathscr{M}} : \mathscr{F}(\Delta_{\mathscr{M}}) \to \mathscr{S}$ is defined by $\Phi_{\mathscr{M}}(v) = v \times s_N$ for objects $v \in obj(\mathscr{F}(\Delta_{\mathscr{M}}))$ and by

$$\Phi_{\mathscr{M}}(\epsilon_v) = \mathrm{id}_{v \times s_N}\,, \tag{12}$$

$$\Phi_{\mathscr{M}}(n_k n_{k-1} \dots n_2 n_1) = \phi^*_{\mathscr{M}}(n_k) \circ \phi^*_{\mathscr{M}}(n_{k-1}) \circ \dots \circ \phi^*_{\mathscr{M}}(n_2) \circ \phi^*_{\mathscr{M}}(n_1) \tag{13}$$

for morphisms in $morph(\mathscr{F}(\Delta_{\mathscr{M}}))$. The composition on the right-hand side of Equation (13) is the composition in $\mathscr{S}$ (which is the same as in $\mathscr{H}$).

Based on Equations (12) and (13), the functor properties are readily verified for $\Phi_{\mathscr{M}}$. More interestingly, $\Phi_{\mathscr{M}}$ picks out a subcategory in $\mathscr{S}$.

Lemma 1.10. *The image of $\Phi_{\mathscr{M}}$ forms a subcategory of $\mathscr{S}$.*

Proof. Straightforward. The only subtle aspect is that for two words $w_1, w_2 \in morph(\mathscr{F}(\Delta_{\mathscr{M}}))$ such that the composition $\Phi_{\mathscr{M}}(w_2) \circ \Phi_{\mathscr{M}}(w_1)$ is in $\mathscr{S}$, one must show that $\Phi_{\mathscr{M}}(w_2) \circ \Phi_{\mathscr{M}}(w_1)$ is in the image of $\Phi_{\mathscr{M}}$. Now, if $\Phi_{\mathscr{M}}(w_2)$ and $\Phi_{\mathscr{M}}(w_1)$ can be composed in $\mathscr{S}$, then $tgt(w_1) = src(w_2)$, with natural extensions of src, tgt from letters in N to words in $morph(\mathscr{F}(\Delta_{\mathscr{M}}))$. But then, $w_2 w_1 \in morph(\mathscr{F}(\Delta_{\mathscr{M}}))$, and hence $\Phi_{\mathscr{M}}(w_2) \circ \Phi_{\mathscr{M}}(w_1) = \Phi_{\mathscr{M}}(w_2 w_1)$ is in the image of $\Phi_{\mathscr{M}}$. □

Definition 1.11 (Image of $\Phi_{\mathscr{M}}$). The subcategory of $\mathscr{S}$ that is the image of $\Phi_{\mathscr{M}}$ is denoted as $\mathscr{C}_{\mathscr{M}}$.

In summary, by extending the state threads of interest to operate on the global state s_N, it has become possible to compose state threads $f^* : (a \times s_N) \to (b \times s_N)$ and $g^* : (b \times s_N) \to (c \times s_N)$ even if the original state threads f, g operate on disjoint parts s_I and s_J of the global state. At the same time, the information that the extended state thread f^* leaves the state $s_{N \setminus I}$ unchanged is retained by the fact $f^* = \Phi_{\mathscr{M}}(w)$, for some $w \in morph(\mathscr{F}(\Delta_{\mathscr{M}}))$. In fact, the letters from N that occur in w are precisely the elements of the subset $I \subseteq N$. An analogous statement holds for g^*.

Moreover, we have identified the subcategory $\mathscr{C}_{\mathscr{M}}$ of $\mathscr{S}$ that is generated by the state threads of interest in $\mathscr{M}$. Figure 2 visualizes how $\mathscr{C}_{\mathscr{M}}$ is related to the multi-graph $\Delta_{\mathscr{M}}$ from Figure 1.

1.4 The *smap* functor

The functor $\Phi_{\mathscr{M}}$ from Definition 1.9 is not the only way of identifying $\mathscr{F}(\Delta_{\mathscr{M}})$ as a subcategory in $\mathscr{S}$. Recall that the objects of $\mathscr{F}(\Delta_{\mathscr{M}})$ are the vertices of the multi-graph $\Delta_{\mathscr{M}}$, which in turn are objects of $\mathscr{H}$, i.e. types in the λ-calculus on that STCLang is based. An alternative way of identifying $\mathscr{F}(\Delta_{\mathscr{M}})$ in $\mathscr{S}$ is obtained by mapping the objects of $\mathscr{F}(\Delta_{\mathscr{M}})$ to list types. By making this precise we will naturally be led to the *smap* functor, i.e. the functor that generalizes *map* to state threads.

Definition 1.12. Let $\mathscr{M}$ be the set of state threads of interest, and let $\phi_{\mathscr{M}} : N \to \mathscr{M}$ be the corresponding bijective map. For each $n \in N$, recursively define a state thread $\psi_{\mathscr{M}}(n)$ as follows,

$$\psi_{\mathscr{M}}(n) : ([src(n)] \times s_n) \to ([tgt(n)] \times s_n) \tag{14}$$

$$\psi_{\mathscr{M}}(n)\,([], \sigma) = ([], \sigma) \tag{15}$$

$$\begin{aligned}\psi_{\mathscr{M}}(n)\,(x : xs, \sigma) = \texttt{let}\ \ & (y, \sigma') = \phi_{\mathscr{M}}(n)(x, \sigma) \\ & (ys, \sigma'') = \psi_{\mathscr{M}}(n)\,(xs, \sigma') \\ \texttt{in}\ \ & (y : ys, \sigma'')\,,\end{aligned} \tag{16}$$

where *src* and *tgt* are the maps defining the multi-graph $\Delta_{\mathscr{M}}$ from Definition 1.6.

Definition 1.13. The functor $\Psi_{\mathscr{M}} : \mathscr{F}(\Delta_{\mathscr{M}}) \to \mathscr{S}$ is defined by $\Psi_{\mathscr{M}}(v) = [v] \times s_N$ for objects $v \in obj(\mathscr{F}(\Delta_{\mathscr{M}}))$ and by

$$\Psi_{\mathscr{M}}(\epsilon_v) = \mathrm{id}_{[v]\times s_N}\,, \tag{17}$$

$$\Psi_{\mathscr{M}}(n_k n_{k-1} \ldots n_2 n_1) = \psi^*_{\mathscr{M}}(n_k) \circ \psi^*_{\mathscr{M}}(n_{k-1}) \circ \cdots \circ \psi^*_{\mathscr{M}}(n_2) \circ \psi^*_{\mathscr{M}}(n_1) \tag{18}$$

for morphisms in $morph(\mathscr{F}(\Delta_{\mathscr{M}}))$.

Exactly as in Lemma 1.10 one verifies that the image of $\Psi_{\mathscr{M}}$ is a subcategory of $\mathscr{S}$.

Definition 1.14 (Image of $\Psi_{\mathscr{M}}$). The subcategory of $\mathscr{S}$ that is the image of $\Psi_{\mathscr{M}}$ is denoted as $\mathscr{C}^{[]}_{\mathscr{M}}$.

The *smap* functor will be defined to mediate between the categories $\mathscr{C}_{\mathscr{M}}$ and $\mathscr{C}^{[]}_{\mathscr{M}}$. This means that, analogously to the *map* functor, *smap* takes a state thread with signature $(a \times s_N) \to (b \times s_N)$ and returns a state thread with signature $([a]\times s_N) \to ([b]\times s_N)$. Additionally, if the argument of *smap* is composed of multiple fundamental state threads, *smap* implements the appropriate plumbing of state in the resulting state thread $([a] \times s_N) \to ([b] \times s_N)$.

Before we can define *smap*, we need a lemma that states that, under certain conditions, the value of the functor $\Phi_{\mathscr{M}}$ fully determines $\Psi_{\mathscr{M}}$.

Lemma 1.15. *Let $w_1, w_2 \in morph(\mathscr{F}(\Delta_{\mathscr{M}}))$ be such that no letter of N occurs more than once in either w_1 or w_2. Then,*

$$\Phi_{\mathscr{M}}(w_1) = \Phi_{\mathscr{M}}(w_2) \;\Rightarrow\; \Psi_{\mathscr{M}}(w_1) = \Psi_{\mathscr{M}}(w_2)\,. \tag{19}$$

Proof. The proof appears in Section 1.6. It relies on an algebraic manipulation that is known as *let floating* in the context of functional language compilers [2]. □

Theorem 1.16 (and definition of *smap*). *If the multi-graph $\Delta_{\mathscr{M}}$ has no cycles, then the following define a functor smap : $\mathscr{C}_{\mathscr{M}} \to \mathscr{C}^{[]}_{\mathscr{M}}$,*

$$smap\,(v \times s_N) = [v] \times s_N\,,\ for\ v \in obj(\mathscr{F}(\Delta_{\mathscr{M}})) \tag{20}$$

$$smap\,(\Phi_{\mathscr{M}}(w)) = \Psi_{\mathscr{M}}(w)\,,\ for\ w \in morph(\mathscr{F}(\Delta_{\mathscr{M}}))\,. \tag{21}$$

Proof. Since $\Delta_{\mathscr{M}}$ has no cycles, no letter from N can occur more than once in any $w \in morph(\mathscr{F}(\Delta_{\mathscr{M}}))$. Hence, Lemma 1.15 guarantees that *smap* is well-defined.

Verifying the functor properties is mechanical. Let $w_1, w_2 \in morph(\mathscr{F}(\Delta_{\mathscr{M}}))$, and assume $w_1 = n_k \dots n_1$, $w_2 = m_l \dots m_1$, with $m_l \dots m_1, n_k \dots n_1 \in N$. Then,

$$
\begin{aligned}
smap\,(\Phi_{\mathscr{M}}(w_2) \circ \Phi_{\mathscr{M}}(w_1)) &= smap\left(\phi^*_{\mathscr{M}}(m_l) \circ \dots \circ \phi^*_{\mathscr{M}}(m_1) \circ \phi^*_{\mathscr{M}}(n_k) \circ \dots \circ \phi^*_{\mathscr{M}}(n_1)\right) && (22)\\
&= smap\,(\Phi_{\mathscr{M}}(w_2 w_1)) && (23)\\
&= \Psi_{\mathscr{M}}(w_2 w_1) && (24)\\
&= \Psi_{\mathscr{M}}(m_l \dots m_1 n_k \dots n_1) && (25)\\
&= \psi^*_{\mathscr{M}}(m_l) \circ \dots \circ \psi^*_{\mathscr{M}}(m_1) \circ \psi^*_{\mathscr{M}}(n_k) \circ \dots \circ \psi^*_{\mathscr{M}}(n_1) && (26)\\
&= \Psi_{\mathscr{M}}(w_2) \circ \Psi_{\mathscr{M}}(w_1) && (27)\\
&= smap\,(\Phi_{\mathscr{M}}(w_2)) \circ smap\,(\Phi_{\mathscr{M}}(w_1))\ . && (28)
\end{aligned}
$$

□

1.5 Extracting parallelism from the structure of state threads

Having defined state threads in STCLang and the *smap* functor, we now investigate opportunities for extracting parallelism based on the structure of state threads. We show that pipeline parallelism arises naturally from *smap*, and we identify structures that exhibit data and task-level parallelism.

1.5.1 Pipeline parallelism. The *smap* functor is defined in terms of $\Psi_{\mathscr{M}}$, for which Equation (18) suggests a very sequential implementation: to evaluate $\Psi_{\mathscr{M}}(n_k \dots n_1)$ on an input $(xs, \sigma) \in [a] \times s_N$, one should first apply $\psi^*_{\mathscr{M}}(n_1)$, then $\psi^*_{\mathscr{M}}(n_2)$, and so on. By Definition 1.12, this means that $\phi_{\mathscr{M}}(n_1)$ is first applied to every element of the list *xs* before $\phi_{\mathscr{M}}(n_2)$ is applied etc. To obtain pipeline parallelism, this order must be relaxed.

How this can be done is illustrated in Figure 3 for $k = 2$. The top diagram in Figure 3 is a graphical representation of Equation (18) applied to the argument $([x_1, \dots, x_l], (\sigma_{n_1}, \sigma_{n_2}, \widetilde{\sigma})) \in [a] \times s_N$. Red and blue arrows indicate which components of this argument are modified by applications of $\phi_{\mathscr{M}}(n_1)$ and $\phi_{\mathscr{M}}(n_2)$ respectively. Note that each application of $\phi_{\mathscr{M}}(n_1)$ and $\phi_{\mathscr{M}}(n_2)$ modifies two components, and hence there are two arrows in every column of the top diagram. The bottom diagram in Figure 3 can be thought of as a squeezed version of the top diagram. In all but the first and the last column there are now four arrows: one pair of red arrows and one pair of blue arrows. This indicates that $\phi_{\mathscr{M}}(n_1)$ and $\phi_{\mathscr{M}}(n_2)$ can be evaluated in parallel, yielding pipeline parallelism. Note that while the top diagram has $2l$ columns, the bottom one only has $l+1$. The data flowing through the pipeline are the elements of the lists $[x_1, \dots, x_l]$, $[y_1, \dots, y_l]$, and $[z_1, \dots, z_l]$.

Squeezing the top diagram of Figure 3 into the bottom diagram is possible since $\phi_{\mathscr{M}}(n_1)$ and $\phi_{\mathscr{M}}(n_2)$ operate on different fundamental state objects, i.e. $n_1 \neq n_2$. That $n_1 \neq n_2$ follows from the fact that the multi-graph $\Delta_{\mathscr{M}}$ is acyclic, which was required to ensure that *smap* is well-defined by Equation (21). When $\Delta_{\mathscr{M}}$ has cycles, pipeline parallelism can still be exploited in evaluating $\Psi_{\mathscr{M}}(n_k \dots n_1)$ provided the $n_1, \dots, n_k \in N$ are pairwise distinct. More generally, for $w_1, w_2, w_2 \in morph(\mathscr{F}(\Delta_{\mathscr{M}}))$ such that only w_2 contains multiple occurrences of the same letter in N, the functor property, i.e. $\Psi_{\mathscr{M}}(w_3 w_2 w_1) = \Psi_{\mathscr{M}}(w_3) \circ \Psi_{\mathscr{M}}(w_2) \circ \Psi_{\mathscr{M}}(w_1)$, can be used to still exploit the parallelism in $\Psi_{\mathscr{M}}(w_1)$ and $\Psi_{\mathscr{M}}(w_3)$.

1.5.2 Data parallelism. When fundamental state threads have certain additional structure, *smap* reduces to *map*, enabling the exploitation of data parallelism. In the following, two structures for which this is possible are presented.

First, consider a morphism in $\mathscr{H}$ of the form $f : a \times s_n \to b$, which uses the state object s_n in a read-only fashion (similar to Haskell's `Reader` type). By the universal property of the product, we can extend f to a state

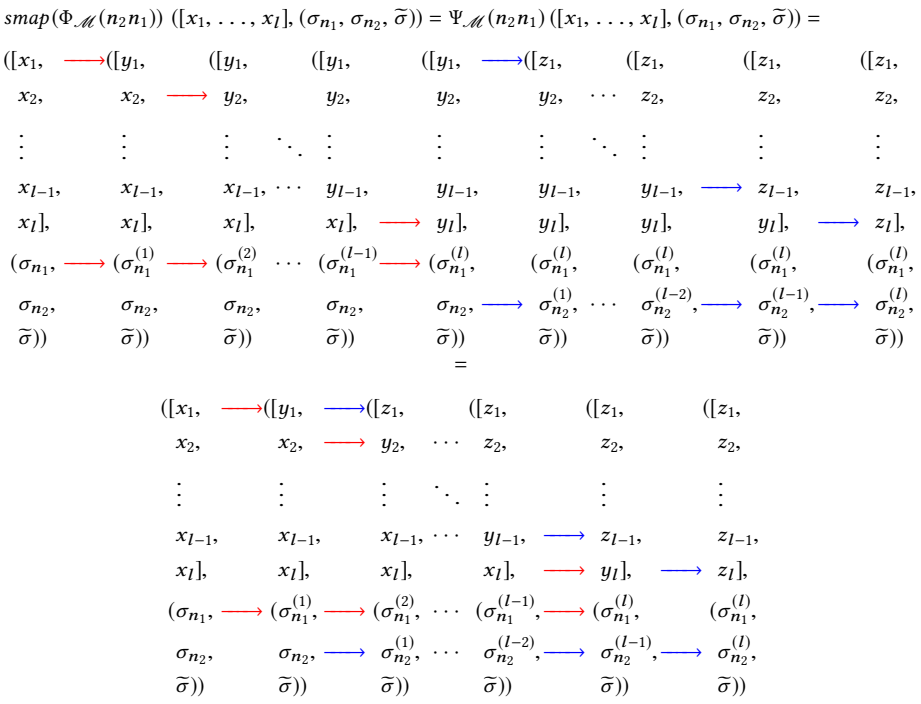

Fig. 3. Graphical representation of the *smap* functor. Red arrows indicate applications of $\phi_{\mathscr{M}}(n_1)$, blue arrows indicate applications of $\phi_{\mathscr{M}}(n_2)$. The top diagram is a direct representation based on the definition of $\Psi_{\mathscr{M}}$ in Equation (18). The equivalent diagram on the bottom exhibits the inherent pipeline parallelism of *smap*.

Fig. 4. Universal diagrams for the product $b \times s_n$ with the natural projections π_1 and π_2.

thread, i.e. to a morphism $\tilde{f}$ in $\mathscr{S}$ by setting $\tilde{f}(x,\sigma) = (f(x,\sigma),\sigma)$ for $x \in a$ and $\sigma \in s_n$. The left pane of Figure 4 gives the corresponding universal diagram. If, in the notation introduced in Section 1.2, $\tilde{f} \in \mathscr{M}$, then $\tilde{f} = \phi_{\mathscr{M}}(n)$, and hence $\tilde{f}^* = \Phi_{\mathscr{M}}(n)$. Evaluating $smap(\tilde{f}^*)$ requires $\psi_{\mathscr{M}}(n)$, whose defining Equation (16) reduces to

$$\begin{aligned} \psi_{\mathscr{M}}(n)\,(xs,\sigma) = \texttt{let}\ \ ys &= map\,(x \mapsto f(x,\sigma))\,xs \\ \texttt{in}\ \ &(ys,\sigma)\,, \end{aligned} \tag{29}$$

and data parallelism can be exploited in evaluating *map*.

The second instance of data parallelism arises if a fundamental state thread $(a \times s_n) \longrightarrow (b \times s_n)$ operates independently on a and s_n. To see this, let $g : a \longrightarrow b$ and $h : s_n \longrightarrow s_n$ be morphisms in $\mathscr{H}$. Again, the universal property of the product can be used to construct a fundametal state thread $g \times h = \phi_{\mathscr{M}}(n)$, as in the right pane of Figure 4. Alternatively, $g \times h$ is characterized by $(g \times h)(x,\sigma) = (g(x), h(\sigma))$. Now, Equation (16) for the

corresponding $\psi_{\mathscr{M}}(n)$ reduces to

$$\begin{aligned}\psi_{\mathscr{M}}(n)\,(xs,\sigma) = \texttt{let}\ \ ys &= map\,g\,xs \\ \sigma' &= (\underbrace{h \circ \cdots \circ h}_{length(xs)\text{ times}})\,\sigma \\ \texttt{in}\ \ (ys,\sigma')\,.&\end{aligned} \tag{30}$$

Again, data parallelism can be exploited in evaluating *map*.

Observe that while $map\,g\,xs$ in Equation (30) is data-parallel, the values of ys and σ' can be computed in parallel too, which is an instance of task-level parallelism.

1.5.3 Task-level parallelism. The simplest case of task-level parallelism occurs if a state thread $h : (a \times b \times s_I \times s_J) \to (c \times d \times s_I \times s_J)$ with $I, J \subseteq N$ and $I \cap J = \emptyset$ decomposes into $f : (a \times s_I) \to (c \times s_I)$ and $g : (b \times s_J) \to (d \times s_J)$, i.e. $h = f \times g$ using the same construction and notation as in the right diagram in Figure 4. Then, h can be evaluated by executing f and g in parallel. Here *smap* is not required to arrive at parallelism.

A more interesting case occurs when the underlying category $\mathscr{H}$ has coproducts, i.e., if for any $a, b \in obj(\mathscr{H})$, there exists an object $a + b \in obj(\mathscr{H})$ and natural injections $inl : a \to a + b$, $inr : b \to a + b$. Then, consider the following fundamental state threads, together with their extensions to s_N,

$$\begin{aligned} &f_1 : a \times s_{n_1} \to (b + c) \times s_{n_1}\,, && f_1^* : a \times s_N \to (b + c) \times s_N\,, \\ &f_2 : b \times s_{n_2} \to b' \times s_{n_2}\,, && f_2^* : b \times s_N \to b' \times s_N\,, \\ &f_3 : c \times s_{n_3} \to c' \times s_{n_3}\,, && f_3^* : c \times s_N \to c' \times s_N\,, \\ &f_4 : (b' + c') \times s_{n_4} \to d \times s_{n_4}\,, && f_4^* : (b' + c') \times s_N \to d \times s_N\,. \end{aligned} \tag{31}$$

From the universal property of the coproduct, we obtain the state thread

$$[f_2^*, f_3^*] : (b + c) \times s_N \to (b' + c') \times s_N \tag{32}$$

$$\begin{aligned}[f_2^*, f_3^*](inl\,x_b, (\sigma_{n_2}, \sigma_{n_3}, \sigma_{N\setminus\{n_1,n_2\}})) = \text{let } (x_b', \sigma_{n_2}') &= f_2(x_b, \sigma_{n_2}) \\ \text{in } (inl'\,x_b', (\sigma_{n_2}', \sigma_{n_3}, \sigma_{N\setminus\{n_1,n_2\}}))&\end{aligned} \tag{33}$$

$$\begin{aligned}[f_2^*, f_3^*](inr\,x_c, (\sigma_{n_2}, \sigma_{n_3}, \sigma_{N\setminus\{n_1,n_2\}})) = \text{let } (x_c', \sigma_{n_3}') &= f_3(x_c, \sigma_{n_3}) \\ \text{in } (inr'\,x_c', (\sigma_{n_2}, \sigma_{n_3}', \sigma_{N\setminus\{n_1,n_2\}}))\,.&\end{aligned} \tag{34}$$

We can then form the composed state thread

$$f_4^* \circ [f_2^*, f_3^*] \circ f_1^* : a \times s_N \to d \times s_N\,. \tag{35}$$

To define how *smap* acts on this state thread, we need two helper morphisms, *split* and *join*, that use the boolean type $\mathbb{B}$ with values **T** and **F**. The morphism *split* decomposes a list of coproduct values, i.e. $[b + c]$, into two lists of types $[b]$ and $[c]$ respectively. This decomposition is completely natural. However, in defining the inverse operation of *split*, one faces a choice: The elements in the lists $[b]$ and $[c]$ can be arranged in different orders to form a list of coproduct values, $[b + c]$. This choice introduces a source of non-determinism, which must be avoided since STCLang is meant to be deterministic. Therefore, *split* and *join* operate on an additional data structure, namely a list of booleans, that encodes the order in which *join* must form a list $[b + c]$ from the

two lists $[b]$, $[c]$.

$$split : [b + c] \to [b] \times [c] \times [\mathbb{B}] \tag{36}$$
$$split\,([]) = ([], [], []) \tag{37}$$
$$split\,((inl\,x_b) : xs) = \text{let } (bs, cs, flags) = split\,xs \text{ in } (x_b : bs,\ cs,\ \mathbf{T} : flags) \tag{38}$$
$$split\,((inr\,x_c) : xs) = \text{let } (bs, cs, flags) = split\,xs \text{ in } (bs,\ x_c : cs,\ \mathbf{F} : flags) \tag{39}$$

$$join : [b] \times [c] \times [\mathbb{B}] \to [b + c] \tag{40}$$
$$join\,([], [], []) = [] \tag{41}$$
$$join\,(x_b : bs,\ cs,\ \mathbf{T} : flags) = (inl\,x_b) : join\,(bs, cs, flags) \tag{42}$$
$$join\,(bs,\ x_c : cs,\ \mathbf{F} : flags) = (inr\,x_c) : join\,(bs, cs, flags) \tag{43}$$

The action of $smap$ on $f_4^* \circ [f_2^*, f_3^*] \circ f_1^*$ is then defined as follows,

$$smap\left(f_4^* \circ [f_2^*, f_3^*] \circ f_1^*\right) : [a] \times s_N \to [d] \times s_N \tag{44}$$

$$\begin{aligned}
&smap\left(f_4^* \circ [f_2^*, f_3^*] \circ f_1^*\right)\ (as, (\sigma_{n_1}, \sigma_{n_2}, \sigma_{n_3}, \sigma_{n_4}, \widetilde{\sigma})) = \\
&\quad \text{let } (as', (\sigma'_{n_1}, \sigma_{n_2}, \sigma_{n_3}, \sigma_{n_4}, \widetilde{\sigma})) = smap\left(f_1^*\right)\ (as, (\sigma_{n_1}, \sigma_{n_2}, \sigma_{n_3}, \sigma_{n_4}, \widetilde{\sigma})) \\
&\qquad (bs, cs, flags) = split\,as' \\
&\qquad (bs', (\sigma'_{n_1}, \sigma'_{n_2}, \sigma_{n_3}, \sigma_{n_4}, \widetilde{\sigma})) = smap\left(f_2^*\right)\ (bs, (\sigma'_{n_1}, \sigma_{n_2}, \sigma_{n_3}, \sigma_{n_4}, \widetilde{\sigma})) \\
&\qquad (cs', (\sigma'_{n_1}, \sigma_{n_2}, \sigma'_{n_3}, \sigma_{n_4}, \widetilde{\sigma})) = smap\left(f_3^*\right)\ (cs, (\sigma'_{n_1}, \sigma_{n_2}, \sigma_{n_3}, \sigma_{n_4}, \widetilde{\sigma})) \\
&\qquad ds = join\,(bs', cs', flags) \\
&\quad \text{in } smap\left(f_4^*\right)\ (ds, (\sigma'_{n_1}, \sigma'_{n_2}, \sigma'_{n_3}, \sigma_{n_4}, \widetilde{\sigma}))\,,
\end{aligned} \tag{45}$$

where $\widetilde{\sigma} \in s_{N \setminus \{n_1, n_2, n_3, n_4\}}$. Note how *flags* is used to ensure determinism by communicating the order of list elements between *split* and *join*. Task-level parallelism can be utilized in Equation (45) by concurrently executing $smap(f_2^*)$ and $smap(f_3^*)$, which is possible since there are no dependencies between the data and state components operated on by f_2^* and f_3^*.

A special case of the previous construction is obtained for $b' = d$, $c' = d$, and

$$f_4 : (d + d) \times s_{n_4} \to d \times s_{n_4} \tag{46}$$
$$f_4(inl\,x_d, \sigma) = (x_d, \sigma) \tag{47}$$
$$f_4(inr\,x_d, \sigma) = (x_d, \sigma)\,. \tag{48}$$

With this f_4, $smap(f_4^* \circ [f_2^*, f_3^*] \circ f_1^*)$ yields a task-parallel version of an if-expression. Conditionals with more than two options are obtained by repeatedly applying the construction from this section.

1.6 Proof of Lemma 1.15

Recall that Lemma 1.15 states that the functor $\Phi_{\mathscr{M}}$ fully determines $\Psi_{\mathscr{M}}$. As a preliminary step towards establishing this, we derive a recursive formula for $\Psi_{\mathscr{M}}$.

Lemma 1.17. *Let $w \in morph(\mathscr{F}(\Delta_{\mathscr{M}}))$ be such that no letter of N occurs more than once in w. Let W be the set of letters in w, and let $\sigma_W \in s_W$, $\sigma_{N\setminus W} \in s_{N\setminus W}$. Then,*

$$\begin{aligned}\Psi_{\mathscr{M}}(w)\,(x : xs, (\sigma_W, \sigma_{N\setminus W})) = \texttt{let}\ \ & (y, (\sigma'_W, \sigma_{N\setminus W})) = \Phi_{\mathscr{M}}(w)\,(x, (\sigma_W, \sigma_{N\setminus W})) \\ & (ys, (\sigma''_W, \sigma_{N\setminus W})) = \Psi_{\mathscr{M}}(w)\,(xs, (\sigma'_W, \sigma_{N\setminus W})) \\ \texttt{in}\ \ & (y : ys, (\sigma''_W, \sigma_{N\setminus W}))\,. \end{aligned} \tag{49}$$

Proof. By induction on the length of w. For $w = \epsilon_v$, $v \in obj(\mathscr{F}(\Delta_{\mathscr{M}}))$, Equation (49) holds trivially since $\Psi_{\mathscr{M}}(\epsilon_v) = \mathrm{id}$ and $\Phi_{\mathscr{M}}(\epsilon_v) = \mathrm{id}$. For the induction step, let $w = nw'$ with $n \in N$ and $w' \in morph(\mathscr{F}(\Delta_{\mathscr{M}}))$. Let W' be the set of letters in w', and let $\sigma_W = (\sigma_n, \sigma_{W'})$ with $\sigma_n \in s_n$, $\sigma_{W'} \in s_{W'}$. Then,

$$\Psi_{\mathscr{M}}(nw')\,(x : xs, (\sigma_n, \sigma_{W'}, \sigma_{N\setminus W})) = \psi^*_{\mathscr{M}}(n) \circ \Psi_{\mathscr{M}}(w')\,(x : xs, (\sigma_n, \sigma_{W'}, \sigma_{N\setminus W})) \tag{50}$$

$$\begin{aligned} &= \texttt{let}\ (y : ys, (\sigma_n, \sigma''_{W'}, \sigma_{N\setminus W})) = \Psi_{\mathscr{M}}(w')\,(x : xs, (\sigma_n, \sigma_{W'}, \sigma_{N\setminus W})) \\ &\quad \texttt{in}\ \psi^*_{\mathscr{M}}(n)\,(y : ys, (\sigma_n, \sigma''_{W'}, \sigma_{N\setminus W})) \end{aligned} \tag{51}$$

$$\begin{aligned} &= \texttt{let}\ (y, (\sigma_n, \sigma'_{W'}, \sigma_{N\setminus W})) = \Phi_{\mathscr{M}}(w')\,(x, (\sigma_n, \sigma_{W'}, \sigma_{N\setminus W})) \\ &\qquad (ys, (\sigma_n, \sigma''_{W'}, \sigma_{N\setminus W})) = \Psi_{\mathscr{M}}(w')\,(xs, (\sigma_n, \sigma'_{W'}, \sigma_{N\setminus W})) \\ &\quad \texttt{in let}\ (z, (\sigma'_n, \sigma''_{W'}, \sigma_{N\setminus W})) = \phi^*_{\mathscr{M}}(n)(y, (\sigma_n, \sigma''_{W'}, \sigma_{N\setminus W})) \\ &\qquad\quad (zs, (\sigma''_n, \sigma''_{W'}, \sigma_{N\setminus W})) = \psi^*_{\mathscr{M}}(n)\,(ys, (\sigma'_n, \sigma''_{W'}, \sigma_{N\setminus W})) \\ &\qquad \texttt{in}\ (z : zs, (\sigma''_n, \sigma''_{W'}, \sigma_{N\setminus W})) \end{aligned} \tag{52}$$

$$\begin{aligned} &= \texttt{let}\ (z, (\sigma'_n, \sigma'_{W'}, \sigma_{N\setminus W})) = \Phi_{\mathscr{M}}(nw')\,(x, (\sigma_n, \sigma_{W'}, \sigma_{N\setminus W})) \\ &\qquad (zs, (\sigma''_n, \sigma''_{W'}, \sigma_{N\setminus W})) = \Psi_{\mathscr{M}}(nw')\,(xs, (\sigma'_n, \sigma'_{W'}, \sigma_{N\setminus W})) \\ &\quad \texttt{in}\ (z : zs, (\sigma''_n, \sigma''_{W'}, \sigma_{N\setminus W}))\,, \end{aligned} \tag{53}$$

where the induction hypothesis was used in going from Equation (51) to Equation (52). The manipulation required to go from Equation (52) to Equation (53) is known as *let floating* in the context of functional language compilers [2]. The assumption that no letter occurs more than once in $w = nw'$ is used whenever elements of state objects are decomposed into components and to determine on which of these components $\Phi_{\mathscr{M}}$ and $\Psi_{\mathscr{M}}$ act as the identity. □

Proof of Lemma 1.15. Let $a = src(w_1) = src(w_2)$. Let $xs \in [a]$ and let $\sigma \in s_N$. The proof proceeds by induction on the length of xs. For $xs = []$, one finds immediately that $\Psi_{\mathscr{M}}(w_1)\,([], \sigma) = ([], \sigma) = \Psi_{\mathscr{M}}(w_2)\,([], \sigma)$. Now, let $xs = x : xs'$, with $x \in a$, $xs' \in [a]$. From Lemma 1.17,

$$\begin{aligned} \Psi_{\mathscr{M}}(w_1)\,(x : xs', \sigma) = \texttt{let}\ \ & (y, \sigma') = \Phi_{\mathscr{M}}(w_1)\,(x, \sigma) \\ & (ys, \sigma'') = \Psi_{\mathscr{M}}(w_1)\,(xs, \sigma') \\ \texttt{in}\ \ & (y : ys, \sigma'') \end{aligned} \tag{54}$$

$$\begin{aligned} = \texttt{let}\ \ & (y, \sigma') = \Phi_{\mathscr{M}}(w_2)\,(x, \sigma) \\ & (ys, \sigma'') = \Psi_{\mathscr{M}}(w_2)\,(xs, \sigma') \\ \texttt{in}\ \ & (y : ys, \sigma'') \end{aligned} \tag{55}$$

$$= \Psi_{\mathscr{M}}(w_2)\,(x : xs', \sigma)\,. \tag{56}$$

Going from Equation (54) to Equation (55) uses both the assumption $\Phi_{\mathscr{M}}(w_1) = \Phi_{\mathscr{M}}(w_2)$ and the induction hypothesis. Equation (56) is arrived at by applying Lemma 1.17 again. □

ACKNOWLEDGMENTS

This work was supported in part by the German Research Foundation (DFG) within the Collaborative Research Center HAEC and the Center for Advancing Electronics Dresden (cfaed).

REFERENCES

[1] Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. 2019. STCLang: State Thread Composition As a Foundation for Monadic Dataflow Parallelism. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019)*. ACM, New York, NY, USA, 146–161. DOI: http://dx.doi.org/10.1145/3331545.3342600

[2] Simon Peyton Jones, Will Partain, and André Santos. 1996. Let-floating: Moving Bindings to Give Faster Programs. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*. ACM, New York, NY, USA, 1–12. DOI: http://dx.doi.org/10.1145/232627.232630