STCLang: State Thread Composition as a Foundation for Monadic Dataflow Parallelism

Sebastian Ertel*
Dresden Research Lab
Huawei Technologies
Dresden, Germany
sebastian.ertel@huawei.com

Justus Adam
Chair for Compiler Construction
Technische Universität Dresden
Dresden, Germany
justus.adam@tu-dresden.de

Norman A. Rink
Chair for Compiler Construction
Technische Universität Dresden
Dresden, Germany
norman.rink@tu-dresden.de

Andrés Goens
Chair for Compiler Construction
Technische Universität Dresden
Dresden, Germany
andres.goens@tu-dresden.de

Jeronimo Castrillon
Chair for Compiler Construction
Technische Universität Dresden
Dresden, Germany
jeronimo.castrillon@tu-dresden.de

Abstract

Dataflow execution models are used to build highly scalable parallel systems. A programming model that targets parallel dataflow execution must answer the following question: How can parallelism between two dependent nodes in a dataflow graph be exploited? This is difficult when the dataflow language or programming model is implemented by a monad, as is common in the functional community, since expressing dependence between nodes by a monadic bind suggests sequential execution. Even in monadic constructs that explicitly separate state from computation, problems arise due to the need to reason about opaquely defined state. Specifically, when abstractions of the chosen programming model do not enable adequate reasoning about state, it is difficult to detect parallelism between composed stateful computations.

In this paper, we propose a programming model that enables the composition of stateful computations and still exposes opportunities for parallelization. We also introduce `smap`, a higher-order function that can exploit parallelism in stateful computations. We present an implementation of our programming model and `smap` in Haskell and show that basic concepts from functional reactive programming can be built on top of our programming model with little effort. We compare these implementations to a state-of-the-art approach using `monad-par` and LVars to expose parallelism explicitly and reach the same level of performance, showing that our programming model successfully extracts parallelism that is present in an algorithm. Further evaluation shows that `smap` is expressive enough to implement parallel reductions and our programming model resolves short-comings of the stream-based programming model for current state-of-the-art big data processing systems.

CCS Concepts • **Software and its engineering** → *Functional languages.*

Keywords parallel programming, functional languages, partitioned state

ACM Reference Format:
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 Haskell Symposium (Haskell '19), August 22–23, 2019, Berlin, Germany.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3331545.3342600

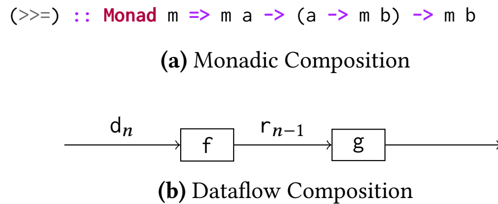
*Work done while at TU Dresden.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Haskell '19, August 22–23, 2019, Berlin, Germany
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6813-1/19/08…$15.00
https://doi.org/10.1145/3331545.3342600

1 Introduction

Dataflow graphs are an abstraction for scalable concurrent computations [12]. Although the concept has a strong functional foundation, it has received little attention in the functional programming commmunity. Specifically, to the best of our knowledge, no proposal exists that combines the explicit concurrency of a dataflow graph with the simple and elegant *monad* type class. In fact, the sequential evaluation semantics of monadic actions seem to prevent this combination.

Dataflow is the de-facto standard for systems that achieve scalability by exploiting parallelism. It is the foundation on which data management systems implement highly parallel execution engines. So-called Extract-Transform-Load (ETL) tools bring data into a data warehouse [10]. Query execution over that data is dataflow-based [18], and the engines that integrate data that continuously arrives in the form of streams build a dataflow graph pipeline [6, 44]. Similar streaming



(a) Monadic Composition

(b) Dataflow Composition

Figure 1. The sequential nature of the monadic composition and the concurrency in a pipeline of a dataflow graph.

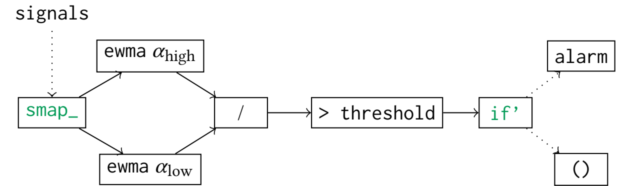


Figure 2. Dataflow Graph of the earthquake detection program.

pipelines and dataflow graphs underpin signal processing and programs for embedded systems [27, 36]. They are also used to build scalable server architectures [15, 42], for fine-grained parallelism in big data systems [13], for optimizing I/O in micro services [16], and concurrent UIs [3, 9].

Studying dataflow in the monadic context is interesting both from a theoretical and a practical perspective. On the one hand, the dataflow abstraction is so close to functional programming that many optimizations for functional languages may directly apply to dataflow [1]. For example, fusion is studied in the context of both functional languages and dataflow systems [17, 20]. On the other hand, a monadic composition of the dataflow graph offers improvement over its manual construction. Manually constructing a dataflow graph does not scale well to larger applications, which is why many established dataflow approaches provide a GUI rather than a language [22]. Implementing dataflow as a monad has the benefit of integrating nicely with the rest of the program, for example with the monad transformers framework.

Monadic composition and concurrency seem to be at odds with each other. Consider the monadic composition, i.e., bind, as defined in Figure 1a. An implementation has to apply the function passed in as a second parameter to the result of the first parameter. In Figure 1b, we show the corresponding composition of the two nodes in a dataflow graph. An arc facilitates the data dependency and denotes data flowing from node f to node g. In this pipeline, applications f d_n and g r_{n-1} are independent and, as such, concurrent.

Hughes noticed this problem as well and suggested that stream processors be implemented using *arrows*, a generalization of monads [21]. Arrows became the dominating abstraction for composing dataflow computations in the context of functional reactive programming (FRP) [8]. However, we believe monads to be simpler and better suited to composing functionality than arrows in most cases.

In this paper, we propose STCLang (State Thread Composition Language), a monad for dataflow computations that enables a parallel execution at runtime, without targeting FRP specifically. To this end, Section 2 motivates our approach on the foundation of unbounded lists/generators [2] and state threads [25, 40] rather than on the notion of streams, signals and signal functions. This allows us to study STCLang not only on data retrieved via I/O operations but also for data that resides in memory, e.g., a normal list. The main contribution of STCLang in Section 3 is a higher-order function called *smap* that is similar to the well-known *map* function. It supports stateful computations, but unlike the state monad's *mapM*, smap introduces concurrency into the computation of the result. Our smap construction introduces pipeline parallelism for stateful computations and exploits data parallelism for stateless nodes in the graph. Afterwards, we show that STCLang can support the FRP abstractions (Section 4).

In Section 5, we evaluate our STCLang implementation against other Haskell libraries for parallelism, in particular monad-par and its extension to LVars. We show that STCLang does not incur additional overheads and that its performance on standard benchmarks is on par with monad-par and LVars. We show that STCLang programs are just as easy to write as monad-par programs but enhance the programming model with implicit parallelism, pipeline parallelism and state. In combination with state threads, our smap function is even expressive enough to implement parallel reductions. In our evaluation, we also show that the transformations in the GHC harmonize with our programming model and even find data parallelism inside state threads. As a final contribution, we ported a state-of-the-art benchmark for big data processing engines to Haskell/STCLang. The resulting STCLang code removes short-comings of the current stream-based programming model used in these systems. We also report preliminary results that identify weaknesses of monad-par's scheduler and open up new research opportunities for the functional community. In Section 6 we review related work, and we conclude in Section 7.

2 Intuition and Motivation

We now motivate and introduce our programming model.

2.1 Use Case: Earthquake Detection

To motivate STCLang, we study an application for earthquake detection that served as a motivation for *Flask*, a language for programming sensor networks [29]. Figure 2 shows the STCLang dataflow graph. The application receives signals to which it applies low-pass and high-pass filters before computing a ratio. If this ratio exceeds a threshold, then an earthquake detection event is triggered. The corresponding

```haskell
detectEarthQuake :: Double
                 -> STCLang' (Source Double) ()
detectEarthQuake threshold = do
  -- prepare state threads
  ewmaHighST <- liftST initStateHigh $ ewma α_high
  ewmaLowST  <- liftST initStateLow $ ewma α_low
  alarmST    <- liftST initCnnState alarm

  let
    -- state thread composition
    detect :: STCLang' Double ()
    detect signal = do
      (low,high) <- (,) <$> ewmaHighST signal
                        <*> ewmaLowST signal
      let ratio = high/low
      let expected? = ratio > threshold
      if' expected? alarmST $ return ()

  -- apply the computation to the signal stream
  return $ \signals -> smap_ detect signals
```

Figure 3. The STCLang program for earthquake detection.

program in STCLang is implemented in Figure 3. The function detectEarthQuake[1] defines an STCLang computation that returns (), i.e., its sole purpose are the side-effects. The code maps over the infinite list of signals and applies detect to each of them. Since the ewma–nodes in the dataflow graph are stateful, they use smap/smap_ which have types:

```haskell
smap  :: STCLang m => (a -> m b) -> [a] -> m [b]
smap_ :: STCLang m => (a -> m b) -> [a] -> m ()
```

That is, smap and smap_ are semantically equivalent to mapM and mapM_ for Haskell's state monad. The generator signals receives input values via the network (I/O) and yields them one at a time. The detect function expresses the independent computations for the high and low values via the applicative functor of STCLang. The ewma function is stateful and is defined as follows:

```haskell
ewma :: Double -> Double -> State Double Double
ewma alpha current = do
  prev <- get
  let new = alpha * current + (1-alpha) * prev
  put new
  return new
```

The function ewma is lifted into the STCLang monad with liftST along with a function that creates an initial state. The computation in ewma updates its state. STCLang carries this new state over to the computation for the next signal and thereby provides the same semantics as a stateful node in a dataflow graph. This monadic state can be composed with other effects, e.g., IO, using monad transformers:

```haskell
alarm :: StateT CnnState IO ()
alarm = do
  address <- get
  liftIO $ sendAlarmMessage address
  return ()
```

[1]We define STCLang' in Section 3.3 and Source in Section 4. For now, think of STCLang' as a function `STCLang m => Dobule -> m ()` wrapped in a state monad and of Source as a generator.

Without loss of generality, we remove the IO aspect and focus solely on the composition of state threads, as well as the parallelism that smap provides. For this, our smap computations map over bounded lists that reside in memory. In Section 4, we add IO back in and show how our concepts can express *behaviors* and *discrete events* of FRP frameworks.

2.2 The Problem with State Thread Composition

Monads in functional programming are a particularly elegant way of separating state from computation. In Haskell, there are two well-established ways of doing so, using state transformers[2]. A state transformer is a function that operates either on immutable [40] or mutable [25] state. Wadler defines state transformers in the context of programming with monads as an abstraction for pure functions [40]. Launchbury and Peyton-Jones [25] use state transformers to operate on mutable references called STRefs. We list the definitions of both in Figure 4. These two approaches effectively encapsulate state inside the monad and separate it from the computational functionality. However, by encapsulating the state opaquely, they also remove the structure of the state. Consider, for example, the composition of two functions with one of the above monads via the monadic bind:

```haskell
(>>=) :: StateThread s a
      -> (a -> StateThread s b) -> StateThread s b
```

This composition of two State threads assumes that they operate on the same type of state. In order to compose two State threads

```haskell
f :: (StateThread StateTypeA a)
g :: a -> (StateThread StateTypeB b)
```

that operate on different state types, the developer needs a compound type

```haskell
CompoundState StateTypeA StateTypeB
```

The State monad does not target the scenario where threads operate on their own local state. In other words, it is unaware of the internal structure of the state and thus cannot derive a parallel execution. This is also true for ST threads. It is well-known from compilers for imperative languages that it is hard to analyze potentially aliasing references to find independent parts of common state. Both types of state threads provide a proper abstraction for encapsulation of stateful (effectful) computations but allow only a sequential execution. As a direct consequence, higher-order functions which expose parallelism naturally for pure functions, such as map, become sequential for state threads (mapM).

2.3 State Thread Composition in STCLang

To outline the intuition behind our programming model, consider the following example functions:

[2]In the Haskell community, this term is often used to refer to the type StateT s m, the monad transformer version of the State monad.

State threads

```
type StateThread s b = State s b
-- state transformers

put :: s -> State s ()
get :: State s s
-- evaluation
runState :: State s b -> s -> (b,s)
```

ST threads

```
type StateThread s b = ST s b
-- state transformers
newSTRef :: a -> ST s (STRef s a)
writeSTRef :: (STRef s a) -> a -> ST s ()
readSTRef :: (STRef s a) -> ST s a
-- evaluation
runST :: (forall s.ST s b) -> b
```

Figure 4. The two notions of state threads in Haskell. `State` threads operate on immutable state wile `ST` threads support mutable state. Both describe a pure function $(a, s) \longrightarrow (b, s)$.

```
1  f :: a -> StateThread sf b
2  g :: b -> StateThread sg c
3  h :: c -> StateThread sh d
```

These functions represent actions that could be applied to a large data set. Each function uses a local state. A key-value store is an example application that fits into this pattern, with functions `load`, `decrypt`, and `decompress`, each of which uses a local cache, i.e. local state, to optimize its execution.

To compose these functions, our programming model takes over the management of the states for the individual state threads. In order to do so, we store all states of the state threads participating in an `STCLang` computation in a single compound state list. Our model is monadic as well, such that a state thread becomes an `STCLang` computation when it is associated with a reference (index) to its private state in this global state list:

```
liftWithIndex :: STCLang m =>
  Int -> (a -> StateThread s b) -> a -> m b
```

Note at this point, we assume that there is a 1-1 correspondence of state threads and slots in the global state list and the state in the list matches the declared state type. We use the global state list and the index reference in order to clearly introduce our concept of handling the state. Later in the paper, we give a version of `liftWithIndex` that enforces type-safety for the state types and the 1-1 correspondence between states and state threads. `STCLang` computations abstract over the local states and composition works again, as for normal functions. The developer does not have to bother with the composition of the state threads with respect to their local states, our formal model of the next section provides a solid foundation. An `STCLang` computation may be seen as a state thread that is aware of its internal structure, i.e., that has clearly defined semantics for which state thread accesses which state cell in its compound state list. The composition of state threads `f`, `g` and `h` is then as follows:

```
1  composition :: STCLang m => a -> m d
2  composition input = do
3    bval <- liftWithIndex 0 f input
4    cval <- liftWithIndex 1 g bval
5    dval <- liftWithIndex 2 h cval
6    return dval
```

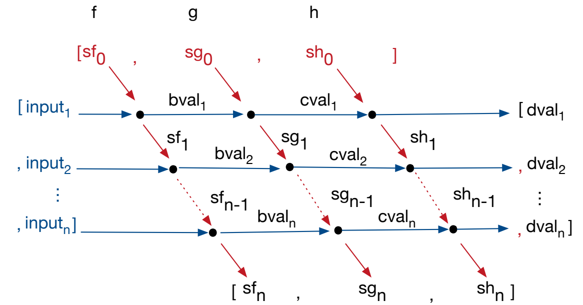


Figure 5. Flows of state and data during execution of `(smap (f >==> g >==> h) inputs)` with state threads `f`, `g`, `h` and the list `inputs` of input data.

2.4 Stateful parallel computations with smap

With composition in place, we define a new higher-order function `smap`, akin to the well-known `map`. It applies a stateful function, i.e., an `STCLang` computation, to each input value in a list. Throughout these applications, the state of the individual state threads may evolve such that the next application sees the state changes of the previous one. To provide a deterministic implementation, `smap` defines that the values are computed in the order of the input list. This is the most intuitive semantics and aligns with `mapM`.

In order to see this evolution of the state, consider the diagram depicted in Figure 5. It has two dimensions. In the horizontal dimension, we depict the composed state threads (in blue) and in the vertical dimension, we show the evolution of their states (in red). Every dot marks an invocation which requires a data input and a state and produces a data output, i.e., a result, and a new state. The result is used as input to the next state thread while the new state is passed on to the next invocation of the same state thread. A possible sequential execution proceeds either left to right and top to bottom or vice versa. But other execution orders exist. For example, when $\texttt{bval}_1$ was computed, the next step may either compute $\texttt{cval}_1$ or $\texttt{bval}_2$. That is, these steps can execute in parallel because they are independent of each other. This is generally known as pipeline parallelism.

From this discussion we gain insight into the reason for the limitations of Wadler's `State` threads and ST threads. In a sense, both can be considered as pure functions $(a, s) \rightarrow (b, s)$. In the case of `State` threads this is clear, as they declare the type of the state in their type signature and require an initial state to evaluate the computation and produce a result and a new state. ST threads, on the other hand, do not expose the state type in their type signature nor require an initial state or produce a new one. It is the rank-2 type in the signature of the evaluation function `runST` that achieves full state encapsulation via the type system. This encapsulation assures that the surrounding program remains pure and observes ST threads as pure functions $a \longrightarrow b$. In other words, ST threads execute the same pure function $(a, s) \longrightarrow (b, s)$ internally but do not expose s, i.e., the mutable state. This is part of what makes it difficult to reason about the state for exposing parallelism.

3 An STCLang Implementation in Haskell

This section is about implementing `STCLang`. The challenge is to implement *state thread composition* and `smap` in a monad so that the inherent parallelism of the dataflow graph is automatically exploited. Figure 6 shows the three types of parallelism that are explicit in the structure of the dataflow graph. We already explained how pipeline parallelism arises between nodes `f` and `g` in a dataflow graph. Two nodes that apply the same function `f` to different data items expose an opportunity for data parallelism. When the nodes that apply `f` and `g` are data-independent of each other, then we refer to this structure as task-level parallelism.

Figure 5 suggests that there exist (at least) two ways to implement the execution of the dataflow graph pipeline. An *smap* implementation based on data-dependency (DD) enforces the proper evaluation order of calls to the state threads via the (blue) data (`input`, `bval`, `cval`), and an implementation based on state-dependency (SD) enforces the evaluation order via the (red) states (`sf`, `sg`, `sh`). To overcome the limitation of the monadic bind, we used a free monad to implement the DD version. This implementation is entirely mechanical. For the rest of the paper, we focus on the implementation of the SD version for two reasons: We are unaware of any construction that follows this direction and it is considerably more concise and elegant than the DD version. The full code is publicly available at https://github.com/ohua-dev/stc-lang.

3.1 Preliminaries

Our implementation contains a core type, `STCLang`, which holds the key construction for state thread composition.

We define `STCLang` with the following type class that requires state thread composition to be implemented in terms of the monadic bind operation:

```
1  class Monad m => STCLang m where
2    liftWithIndex :: Int
3                  -> (a -> StateThread s b) -> a -> m b
4    runSTCLang    :: m b -> [S] -> b
5    smap          :: (a -> m b) -> [a] -> m [b]
6    smap_         :: (a -> m b) -> [a] -> m ()
7    if'           :: Bool -> m b -> m b -> m b
```

The `liftWithIndex` function turns a state thread into an `STCLang` computation, i.e., it associates a state thread with a particular item in the global state list.[3] `runSTCLang` takes an `STCLang` computation and the global state list as inputs and executes the `STCLang` computation. Note that we support heterogeneous types in the state list via `S`, a wrapper around the `Dynamic` type interface which provides a conversion from a concrete value into a dynamic one (`toS`) and the inverse (`fromS`). Since the state and the according state thread are only connected via the integer reference, i.e., the index, into the global state list, `STCLang` is not type-safe. `STCLang'` provides the interface presented in the previous section that adds this type-safety and follows at the end of this section. The functions `smap`, `smap_` and `if'` add control to `STCLang`. In our implementation, we abstract over the notion of a state thread:

```
1  type StateThread s b = State s b
2
3  runStateThread :: StateThread s b -> s -> (b,s)
4  runStateThread = runState
```

3.2 The state dependency-based (SD) implementation

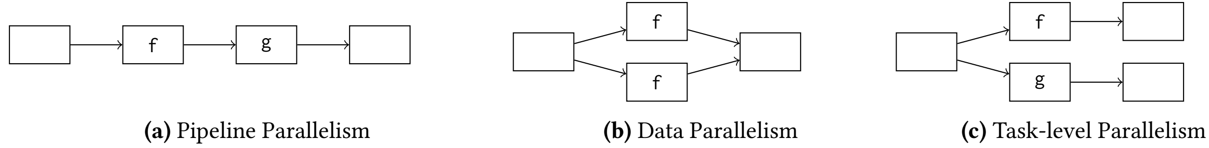
Our SD construction is essentially a version of the classic state monad that threads a (global) state list s_N through a computation composed of (fundamental) state threads. Each fundamental state thread $(a, s_n) \longrightarrow (b, s_n)$ accesses only one state s_n in the list s_N. In the SD construction, it is much more challenging to extract pipeline parallelism from *smap* than in the DD implementation that can decompose state threads. This is impossible in the state monad because its `bind` really has to apply the state threads and cannot gather the computation, as the free monad does. It is also impossible to impose parallelism via `bind` because of its strictly sequential definition. We can only introduce (pipeline) parallelism in the implementation of `smap`, similar to the classic `map` function. To provide the desired semantics, we enable the `smap` implementation to define the order in which state evolves.

3.2.1 Monadic Structure

For a lifted state thread `a -> SD b`, we define SD as follows:

```
data SD result = SD { runSD :: GlobalState -> Par result }
```

[3] We restrict our presentation to one-argument functions. Supporting multiple arguments is straightforward.



(a) Pipeline Parallelism **(b)** Data Parallelism **(c)** Task-level Parallelism

Figure 6. The types of parallelism in a dataflow graph.

Monad Par provides IVars and tasks to build dataflow graphs at runtime [30].[4] We are interested in a compile-time definition of the dataflow graph in the context of STCLang. Despite the runtime graph construction, basing our implementation on monad Par has two advantages. First, the execution benefits from Par's work-stealing scheduler, and second, STCLang computations are guaranteed to be deterministic because every Par is. This has the nice property that it immediately detects deadlocks and in such a case aborts the execution. ParIO is an extension that allows tasks to perform IO operations. This enables implementations such as the alarm state thread in our earthquake detection program. However, we focus on the state aspect and define the functor and monad instances as follows:

```
1 instance Functor SD where
2   fmap f = SD . fmap f . runSD
```

```
1 instance Monad SD where
2   return v = SD $ \sN -> return (v, sN)
3   f >>= g  = SD $ \sN -> do
4                         x <- runSD f sN
5                         runSD (g x)  sN
```

These instances are almost the same as for the state monad that we stated in the introduction. The only real difference is in the implementation of bind (>>=) which passes the same global state s_N to both of the state threads f and g. This is intuitive in our programming model because both state threads operate on different parts of the global state. The evolution of the individual states is then captured in GlobalState:

```
1 data GlobalState = GlobalState { initials :: [IVar S]
2                                , results  :: [IVar S] }
```

The global state consists of two lists of IVars (where S is again the generic state type). The first list carries the (initial) input states $[s_1, \ldots, s_n]$ and the second list stores the resulting states $[s'_1, \ldots, s'_n]$.

We instantiate our STCLang type class and lift state threads into STCLang:

```
1 liftWithIndex idx f a = SD (comp $ f a)
2   where
3     comp st (GlobalState initials results) = do
4       let ivarState  = initials !! idx
5           ivarState' = results  !! idx
6       localState <- fromS <$> Par.get ivarState
7       (r, localState') <- runStateThread st localState
8       _ <- Par.put ivarState' $ toS localState'
9       return r
```

[4] In this paper, we omit the NFData type constraints that monad Par requires to force the evaluation of its tasks.

In liftWithIndex, we use the assigned index (idx) to retrieve the IVars for this state thread from the global state (Lines 4–5). Afterwards, we get the local input state for the state thread computation from the IVar (Line 6). This call blocks until the IVar has state data. Once the call returns, fromS casts the private state from the generic state type S to the actual state type in the definition of the state thread. We apply the state thread to the local state to compute a result r and an updated local state (Line 7). To propagate the new local state, the code at Line 8 casts it back to the generic state type S and puts it into the result IVar (ivarState'). This put call is non-blocking and activates other computations that requested the content of this IVar. The last statement lifts the result into the SD monad.

3.2.2 Parallelism

So far, our construction of SD does not introduce concurrency but we prepared the setup for it on the foundation of IVars.

Pipeline Parallelism Our smap implementation orchestrates the global state to extract pipeline parallelism:

```
1 smap :: (a -> SD b) -> [a] -> SD [b]
2 smap h xs = SD $ \(GlobalState initials results) -> do
3     ysIVars <- compute h xs initials results
4     forM ysIVars Par.get
```

This implementation looks very similar to a version that implements the classic map to execute in parallel using IVars and tasks. At Line 3, the compute function spawns a task that computes each item in the input list and returns the respective IVars. At Line 4, we iterate over these IVars to get the results of the computations and return them. The vital aspect that enforces the semantics of smap is the construction of the GlobalState for the individual computations:

```
1  compute _ [] _ _ = return []
2  compute h (xi:xs) currentStates lastStates = do
3    nextStates <-
4      if null xs -- is this the last coputation?
5        then return lastStates
6        else replicateM (length currentStates) Par.new
7    yIVari <- Par.spawn
8                 $ runSD (h xi)
9                     $ GlobalState currentStates nextStates
10   ysIVars <- compute xs nextStates lastStates
11   return (yIVari : ysIVars)
```

At Line 4, we spawn a new task to compute y_i. Tasks execute concurrently, so we get an IVar that eventually contains the result. More importantly, we use nextStates as the list that captures the resulting states for computing y_i.

Next, we pass it on to the recursive call but as a parameter for the `currentStates` state list for computing y_{i+1}. Due to our implementation of `liftWithIndex`, a state thread `h` participating in the computation of y_{i+1} needs to wait until the same state thread participating in the computation of y_i has propagated the new state. At Line 3, we select the `nextStates` state list that we share across two consecutive computations. It is either an entirely new list of IVars or the `lastStates` list that was input to the whole `smap` computation (Lines 10–12). This construction preserves the order of state evolution shown in Figure 5 although computations for y_i and y_{i+1} run concurrently. Pipeline parallelism arises because lifted state threads immediately report their new local state when they complete their part of the computation. Like this, the computation for y_{i+1} can already proceed although the one for y_i has not finished yet.

Data Parallelism In order to introduce data parallelism, we provide a version of `liftWithIndex` for a `Reader` as a special type of state thread:

```
1  liftWithIndex :: Int -> (a -> Reader s b) -> a -> SD b
2  liftWithIndex idx f a = SD (comp $ f a)
3    where comp st (GlobalState initials results) = do
4          let ivarState  = initials !! idx
5              ivarState' = results  !! idx
6          localState <- Par.get ivarState
7          _          <- Par.put ivarState' localState
8          runReader st $ fromS localState
```

Note that we pass the `localState` to the `results` at Lines 6–7 even before running the state thread (Line 8). Thus, our `smap` implementation executes such state threads in a data-parallel fashion. In fact, our evaluation shows that the GHC transforms state threads that either have no state or use it only for reading into exactly this form. That is, data parallelism comes for free!

Task-level Parallelism There are two cases that expose the structure of the dataflow graph presented in Figure 6c for task-level parallelism. When the previous node splits up the input data and when the previous node splits up control, i.e., the previous node represents an `if'` such as in our earthquake detection program.

The basic form to express independent computations in Haskell is `(f, g)` for pure functions `f` and `g`. When `f` and `g` are state threads, we can express the same computation as `(,) <$> f <*> g` using the applicative functor of `SD` [31]:

```
1  instance Applicative SD where
2    pure = return
3    f <*> g = SD $ \sN -> do
4                       gResultVar <- Par.spawn $ runSD g sN
5                       fp         <- runSD f sN
6                       fp         <$> Par.get gResultVar
```

First it places the computation of the argument `g` on a new task. Then it extracts the pure function `fp` concurrently by evaluating the left side of the `<*>` expression, e.g. `(,) <$> f`. Afterwards, it waits for the result of the argument computation and applies to it the extracted pure function.

In a conditional expression not all state threads execute, only the ones of the selected expression. To allow conditionals in conjunction with `smap`, we need to transfer the states for the state threads that did not execute this time in order to avoid deadlocks. In order to do so, we extend our definition of `SD` with an `id` function:

```
1  data SD result = SD {
2    runSD :: GlobalState -> Par result,
3       id :: GlobalState -> Par () }
```

The implementation of `id` for any lifted state thread returns unit and does not alter the local state:

```
liftWithIndex idx f a = SD (comp $ f a) (comp $ return ())
```

The `comp` function remains unchanged and propagates the final state to the result IVar. The corresponding code in `>>=` applies `id` instead of `runSD` and discards the returned unit values. The implementation of `if'` calls `runSD` on the branch to be executed and `id` on the other:

```
1  if' :: Bool -> SD b -> SD b -> SD b
2  if' cond trueBranch falseBranch = SD comp idIf
3    where
4      (toExecute, toTransfer) = if cond
5        then (trueBranch, falseBranch)
6        else (falseBranch, trueBranch)
7      comp sN = do
8        _ <- id toTransfer sN
9        runSD toExecute sN
10     idIf sN = id trueBranch >> id falseBranch
```

Both calls propagate the states and fulfill the assumption of `smap`. To enable task-level parallelism when `if'` is used in conjunction with `smap`, the `comp` function propagates the state for the branch that is not executed first. This unblocks the local states such that a consecutive `smap` computation executing this branch can proceed.

This completes the construction of our implementation of the core of `STCLang` based on the state-dependency execution order. We now extend this core to implement type-safety for state threads and later to show how `STCLang` supports FRP-style computations.

3.3 Type-safety for state threads

We wish to make state threads type-safe, i.e., the type system should verify that the state used in a state thread has the proper type. In order to do so, we create a single function that takes the state and the state thread as its argument and lifts it into a `STCLang` computation:

```
1  liftST :: Typeable s
2         => s -> (a -> StateThread s b) -> STCLang' a b
```

This function provides type-safety for state threads by encapsulating the state type `s`. It can be implemented easily using a state monad that collects the local states, builds the global state list and the `STCLang` computation:

```
data CollSt = CollSt { stStates :: [S] }
type STCLang' a b =
   forall m. STCLang m => State CollSt (a -> m b)
```

The rest of the implementation is a straightforward usage of the state and STCLang monad functions:

```
liftST localState stateThread = do
  l <- state $ \s -> ( length $ stStates s
                     , stStates s ++ [toS localState])
  pure $ liftWithIndex l stateThread

runSTCLang' :: STCLang' a b -> a -> (b, [S])
runSTCLang' langComp a = do
   (comp,gs) <- runState langComp []
   runSTCLang (comp a) $ stStates gs

smap' :: STCLang' a b -> STCLang' [a] [b]
smap' comp = smap <$> comp
```

3.4 Formal Foundations

To compose state threads formally, we rely on the category-theoretic notions of *objects*, *products* and *morphisms*. The *state objects* s_n, where n denotes an index in a fixed index set N, are objects in a suitable category. Then, the global state list is taken to be the product $s_N = \prod_{n \in N} s_n$. Other state objects can be formed for any subset $I \subseteq N$, i.e. $s_I = \prod_{n \in I} s_n$. The state threads are morphisms, and associated with each state thread is a state object s_I with $I \subseteq N$. Assume that f and g are state threads with associated state objects s_I and s_J, respectively. While composing f and g in Haskell's State monad requires that the state objects s_I and s_J be the same (i.e. $I = J$), STCLang can compose state threads for which $I \neq J$. The basic idea that enables this is to lift state threads f and g to morphisms that formally operate on the global state s_N. However, only the state object s_I is truly modified by f, while the identity is applied to $s_{N \setminus I}$, and analogously for g and s_J. This equips our model with well-defined composition for state threads. The definition of smap can also be made precise in category-theoretic terms as a functor *smap*. This is done in the supplementary material [14], alongside an explanation of how different types of parallelism can be formally extracted.

4 FRP-style programming with STCLang

We motivated STCLang with applications that are currently written in a functional reactive programming (FRP) style. This section investigates how hard it is to implement FRP concepts such as signals and filters in STCLang.

Signals/IO FRP applications are centered around IO. As such, we add IO to our implementation from Section 3:

```
type StateThread s b = StateT s IO b

runStateThread :: StateThread s b -> s -> IO (b,s)
runStateThread = runStateT

type STCLang' a b =
  forall m . STCLang m => StateT CollSt IO (a -> m b)
```

FRP programs often implement a dataflow graph that is structurally similar to the one in Figure 7a. The three sources s_0, s_1, s_2 receive data via IO operations. Original FRP classified these sources into behaviors and discrete events [41]. These types were eventually collapsed into the abstraction of a signal and an associated stream thereof [33]. The signals flowing along the edges of the graph eventually get merged and the result is often emitted via an IO operation, such as for example the alarm message in our earthquake application.

Figure 7b shows the equivalent STCLang dataflow graph. We model infinite lists, i.e., streams, with generators [2]. Our generator implementation is relatively standard and therefore omitted. So is the extension of smap to take such a generator instead of a list:

```
smapGen  :: STCLang m
         => (a -> m b) -> Generator IO a -> m [b]
smapGen_ :: STCLang m
         => (a -> m b) -> Generator IO a -> m ()
```

In FRP, a signal function has type `Signal a -> Signal b`, or simply `SF a b`, cf. [33]. It essentially denotes a node in the dataflow graph where `Signal a = Time -> a` states that this function gets re-executed over time. In STCLang, we do not need this abstraction. Our smap function provides these semantics. All we need is a type for the set of events that may occur:

```
data Event events
```

We use an open union in order to provide an extensible but type-safe abstraction [28]. Individual sources are initializable generators:

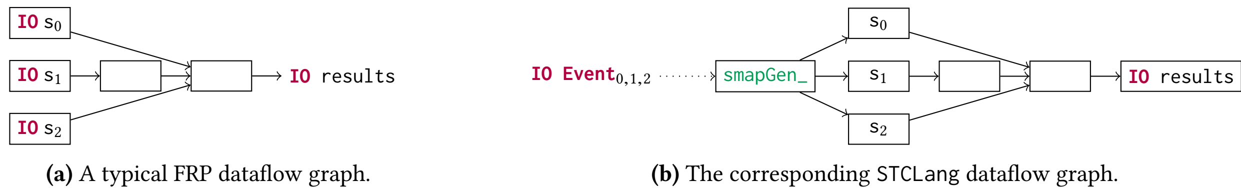
```
type Source a = IO (Generator IO a)
s0≤i≤2 :: Source ai
```

We now build some machinery to make the usage of these abstractions more convenient. In the end, the programmer can write the following code to implement the dataflow graph from Figure 7b:

```
frp = do
  s0' <- liftSource s0 init0
  s1' <- liftSource s1 init1
  s2' <- liftSource s2 init2
  return runSources $ \event -> do
    x <- s0' event
    y <- s1' event
    z <- s2' event
    ... -- computation in the graph
    return ()
```

The essential aspect is that we separate the IO operations from the state aspect in the sources. liftSource creates a state thread for the source that keeps the last event value as its state:

```
liftSource :: (Typeable a, a ∈ events)
  => Source a -> IO a -> STCLang' (Event events) a
liftSource s0 init = do
    idx <- state $ \s ->
            ( length $ sources s
            , s {sources = sources s ++ [toS <$> s0]})
    liftST init $ \(i, s) ->
```



(a) A typical FRP dataflow graph.

(b) The corresponding STCLang dataflow graph.

Figure 7. An FRP dataflow graph and its counterpart in STCLang. The STCLang graph decouples IO from state. A generator performs the IO operations for all sources while the original source nodes s_0, s_1 and s_2 only store the previous state. If s_0 receives an $\texttt{Event}_0$ from its IO source, then it updates its state accordingly and outputs its new state. Otherwise, it emits its current state, i.e., the last event it stored.

```
8              -- state thread for the source
9          if i == idx
10           then do let my = fromS s
11                   put my
12                   pure my
13           else get
```

Every event is input to all of these state threads. For $\texttt{Event}_0$, state thread s_0 stores the value of the event and emits it. The other source state threads s_1 and s_2 observe that the event was not from their source and emit the value from their state. `runSources` bundles all IO operations of the sources into a concurrent generator:

```
1   runSources :: STCLang' (Event events) a -> IO ([a], [S])
2   runSources comp = do
3       (comp', s) <- runStateT comp mempty
4       signalGen <- boundleSources $ sources s
5       runSTCLang (smapGen comp' signalGen) $ states s
```

In order to do so, STCLang' collects the sources in addition to the states:

```
1   data CollSt events = CollSt {
2       stStates :: [S]
3     , sources :: [Source' events] }
```

The `Source'` open union type allows us to store different types of sources in a single list.

Filters Filters conditionally forward data. STCLang incorporates conditionals (`if'`), which allows us to define a filter function with the following type:

```
1   filter init cond comp = do
2     g <- liftST init
3                 -- state thread for conditional emission
4                 $ maybe S.get (\i -> S.put i >> pure i)
5     -- the filter computation
6     return $ \item -> do
7       success? <- cond item
8       i <- if' success?
9                (pure Nothing)
10               $ Just <$> comp item
11      g i
```

The function creates a new STCLang computation (Lines 6–11) that applies `comp` when the check succeeds. When the check fails, it emits the last seen value. This is semantically equivalent to stopping the evaluation.

We implemented basic FRP concepts on top of STCLang with only a few simple extensions. The resulting programming framework lets us write concise code and enables parallel execution. Our FRP framework can be seen as push-based and avoids glitches naturally. An investigation of how this FRP implementation can be optimized is left for future work.

5 Evaluation

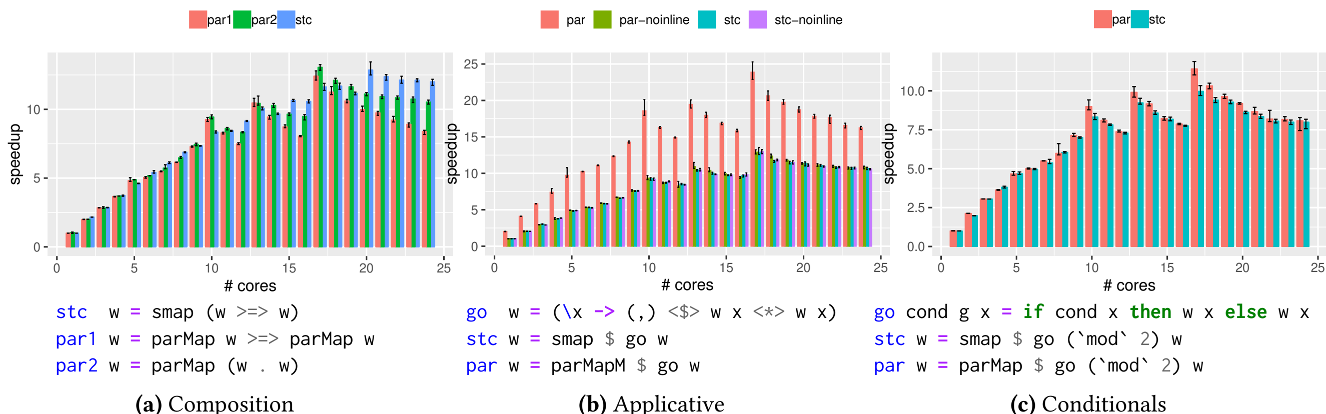
In this section, we evaluate STCLang and the simple FRP framework described in the previous section. For this, we compare against `monad-par` and *LVars* [24], an extension to `monad-par` that enables pipeline parallelism, in addition to a suite of parallel data-structures. We evaluate on microbenchmarks for the dataflow graphs from Figure 6, original `monad-par` benchmarks and study how GHC handles different types of state threads by adding state into a benchmark. Finally, we evaluate our FRP framework in the context of data streaming systems.

We ran all experiments on an Intel Xeon i7 (2.6 GHz) with 2 sockets, 6 cores per socket and support for hyperthreading enabled. Our Haskell code was compiled with GHC version 8.6.5. If not stated otherwise, we executed the benchmarks using Haskell's microbenchmarking library criterion[5] and report speedup over sequential versions.

5.1 Microbenchmarks

To study the performance overhead introduced by STCLang, we constructed three simple programs for (monadic) composition, applicative and conditionals. Figure 8 lists the versions for STCLang and `monad-par` underneath the associated results. The state threads in the STCLang versions are pure, i.e., they do not use state to compute a result. The code for the computations (w) is from the original work that introduced LVars [24]. The results show no significant overhead for STCLang compared to `monad-par`. For (monadic) composition in Figure 8a, STCLang even yields slightly better results for high thread/core counts. We attribute this to the additional pipeline parallelism that our construction adds. The aspect becomes only visible on top of the existing data parallelism. In the case of applicative (Figure 8b),

[5]http://www.serpentine.com/criterion/



(a) Composition **(b)** Applicative **(c)** Conditionals

Figure 8. Microbenchmark results comparing STCLang with monad-par.

the graph shows a super-linear speedup for monad-par, e.g., with 13 threads the performance increases by a factor of 20. This is because the computation in w is actually pure, in the case of monad-par implemented as a simple (pure . f). GHC inlines this definition and is able to apply common subexpression elimination to the resulting (f x) values. In the case of STCLang we use a syntactically more complex liftWithIndex idx f, which GHC cannot trivially optimise. To verify this analysis, we annotate w with NOINLINE to prevent this optimisation. The none-inlined version has the same performance as STCLang. STCLang trades such optimizations in favor of the enhanced parallelism of liftWithIndex. Allowing such optimizations for state threads without side-effects is certainly an interesting future research direction. The performance on conditionals (Figure 8c) shows again similar speedup results for both versions. The associated code snippets show that smap is just as simple to use as the parMap combinator from monad-par in a stateless scenario. Adding state into the computation is simple in the case of STCLang but not so in monad-par.

5.2 Benchmarks

STCLang essentially introduces only a single new combinator: smap. We demonstrate its expressiveness both for mapping and folding in a set of benchmarks that we converted from monad-par versions [30].

The first two are matrix multiplication and an implementation of the Black-Scholes algorithm that models financial contracts. Both are straightforward applications of data parallelism, i.e., parMap and smap, as shown in the previous experiments. As such, the speedup is nearly the same, as can be seen in Figure 9. Note that for matrix multiplication there is in fact a slowdown. This comes as no surprise. We compare against an essentially sequential version which is highly optimized by GHC and thus also exploits SIMD parallelism.

We demonstrate the expressiveness of smap using the next two benchmarks: the computation of the Mandelbrot set and the sum of the values of Euler's function up to a given bound. Such computations are typically implemented using a divide-and-conquer strategy. At first, the input data is split into even-sized chunks such that intermediate results can be computed in a data parallel fashion. Afterwards, the intermediate results are folded into a single one. The essence of this is captured in the MapReduce programming model that was very successfully applied to program parallel applications to crunch big data on a cluster [11]. The monad-par implementations of Mandelbrot and SumEuler are based on the combinator with the same semantics[6]:

```
1  mapReduceThresh threshold (InclusiveRange min' max')
2                  fn binop init
3   = loop min' max'
4   where
5    mapAndCombine min max =   -- mapper and combiner
6      let mapred a b = do x <- fn b;
7                          result <- a `binop` x
8                          return result
9      in foldM mapred init [min..max]
10   loop min max
11     | max - min <= threshold = mapAndCombine min max
12     | otherwise   = do
13         let mid = min + ((max - min) `quot` 2)
14         rght <- spawn $ loop (mid+1) max
15         l  <- loop min mid
16         r  <- get rght
17         l `binop` r                   -- reducer
```

The code for mapAndCombine is almost a literal translation of the execution model for the mapper and the combiner in the MapReduce programming model. The reduction is then defined at Line 10. The mapper-combiner pair and the reductions are performed in parallel because the code spawns parallel tasks until a threshold is reached. It is immediately clear how smap supports the mapper-combiner part and, as it turns out, the reduction is just as straightforward:

[6] https://github.com/simonmar/monad-par/blob/master/monad-par-extras/Control/Monad/Par/Combinator.hs

```
mapReduceThreshSTC threshold (InclusiveRange min' max')
                   fn binop init
  = do
    (_, [reduceState]) <- runSTCLang mapReduce
                            $ chunkGenerator min'
    return $ fromS reduceState
  where
    mapReduce = do
      reduceST <- liftST init reduce
      return $ smapGen
                 ((pure . mapAndCombine) >=> reduceST)
    reduce v = S.get >>= (S.put . (`binop` v))
    chunkGenerator min
      | max' - min <= threshold = yield (max, min)
                                          finish
      | otherwise = yield (newMin, min)
                          (chunkGenerator newMin)
      where newMin = min + threshold
```

Here mapAndCombine is a pure version of the code in the monad-par version. We essentially implement the fold with a state thread that folds over its inputs. Its state is the final result of the computation. Note that we effectively combine pipeline with data parallelism. The chunkGenerator pipelines chunks to mapAndCombine, which executes in a data parallel fashion because it does not use state. Its intermediate results are again pipelined to the reduce function.

The STCLang version of Mandelbrot produces speedups on par with the monad-par version in Figure 9c. In fact, we see a slight benefit again for STCLang when core counts are high due to the pipelined execution. For the SumEuler computation in Figure 10, monad-par applies parMap and then sums the results sequentially. We compare two different STCLang versions. The first version (stc) uses the mapReduce pattern to perform the addition in pipelined fashion. The second one (sum <$> stc) mimics the monad-par implementation. Here, the mapReduceSTC approach (without a range) does not provide the best results. The computation in the reduce phase does not amortize the additional cost incurred by passing along the reduction state in IVars. Most likely the sum function is already very highly optimized by the GHC.

5.3 GHC Effects

In this evaluation experiment the test program is the composition pipeline from the first microbenchmark executed with smap:

```
comp :: (Float -> State Float Float)
     -> [Float] -> [Float]
comp stateThread coll =
  runSTCLang (smap (f >=> g) coll)
             [toS (4.0 :: Float), toS (3.0 :: Float)]
  where
    f = liftWithIndex 0 stateThread
    g = liftWithIndex 1 stateThread
```

Both pipeline stages, f and g, execute the same state thread. The state thread is a parameter to the computation such that we can compare the three variants listed underneath the associated plot in Figure 12. The first state thread is in fact a pure computation. The second one uses its state only for reading. Both of these state threads actually do not perform any side-effects and as such would be applicable to a data parallel execution. The last state thread reads its state, uses it in its computation and then updates it. Function w is again the compute-intensive calculation from the LVars benchmark.

The results in Figure 12 plot the speedup over the sequential version for the three types of state threads. The state thread that has side-effects to its state has a speedup factor of 2×, i.e., it exploits the pipeline parallelism. More interestingly, the other two types of state threads experience an almost linear speedup. The stages of the pipeline execute in a data-parallel fashion. This is due to the fact that GHC transformations apply not only to state threads but to the whole code in liftWithIndex. For reasons of laziness, GHC transforms this code such that state is handled first and then the rest of the computation is performed. This has an interesting cascading effect. When the state is put into an IVar, monad-par schedules the task that waits on this IVar on the same thread to also benefit from data locality. The continuation of the computation is put as another task into the scheduling queue. As such, the computation first performs all the state plumbing and then executes the remaining computations in parallel. That is, the optimizations in the GHC, the work-stealing scheduler of monad-par and our use of it in STCLang seem to harmonize very well with each other.

5.4 Pipeline Benchmarks

Out-of-the-box, monad-par does not provide support for pipeline parallelism. LVars add this capability [24]. In essence, LVars consist of two parts: a lattice-based data structure and event handlers that fork a computation on part of the data structure. STCLang embodies pipelining implicitly via its smap construction and in terms of generators.[7]

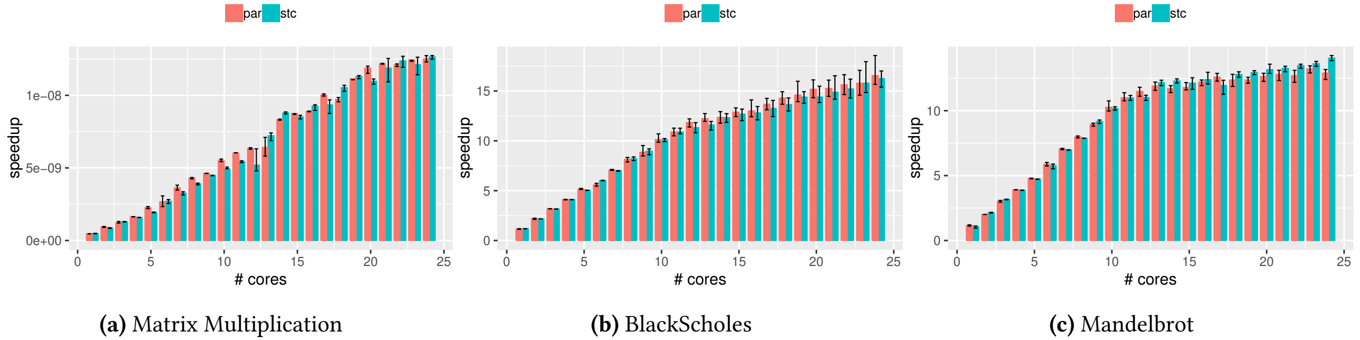
The use case in the LVars paper was a traversal of a connected component (in a larger social media graph), where a function is applied once to each component. The STCLang program is as simple as

```
smapGen work $ generator w connectedComponent
```

We isolate the pipeline parallelism effect for LVars using a lock in the (consuming) work function to synchronize the event handlers. This is not necessary for STCLang. We simply state that work is a state thread that reads and writes state. The computation performed is again the w function from the original work on LVars that we already used in the previous experiments. To create different loads in the pipeline, we also let the generator perform the w function.

In Figure 11 we study the scalability of pipeline parallism in STCLang compared to LVars, as well as the overhead and the influence of a disbalanced pipeline on the speedup. We first balance the work performed in the generator and the

[7] The work on LVars eventually found the same generator abstraction for pipelining [32].



(a) Matrix Multiplication **(b)** BlackScholes **(c)** Mandelbrot

Figure 9. Speedup results for selected benchmarks. The first two benchmarks apply data parallelism while the third uses `mapReduceThresh`.

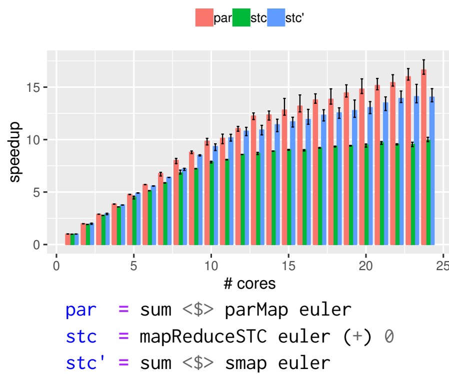


Figure 10. SumEuler with different STCLang versions.

consumer, i.e., the `w` function and consider the scalability. We find that `STCLang` code scales slightly better than the LVars code. This might also be due to the lock that we had to use in order to simulate a stateful computation and isolate pipelining. It shows that building stateful but parallel computations with `STCLang` is easier to program and provides better performance. This effect is also visible in the speedup results for the overhead. Both experiments executed a balanced pipeline, i.e., the generator and the `work` function executed the same computation per data item. In the last experiment, we study the peak performance by varying the computation in the `work` function. The corresponding plot in Figure 11c shows that maximum performance can only be achieved when the stages of the pipeline perform the same amount of computation per data item. This is the grand challenge when exploiting pipeline parallelism and an interesting area for future work.

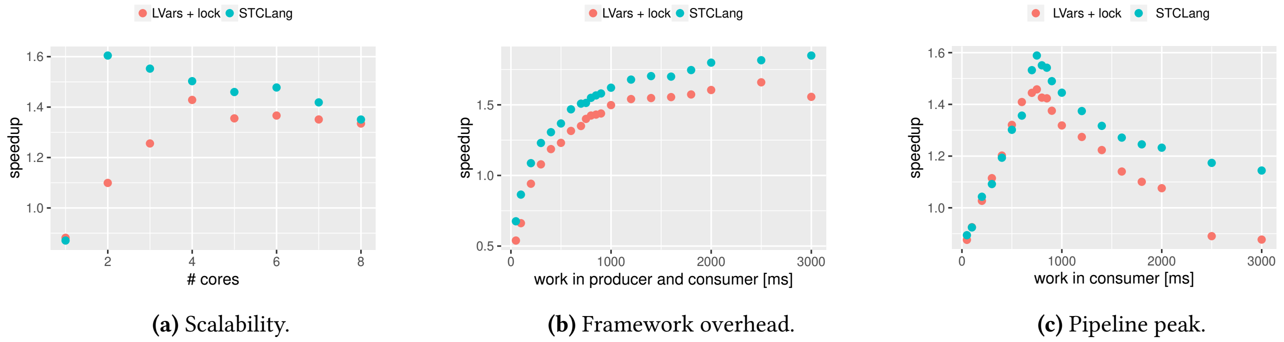
5.5 Data Streaming Benchmark

Unfortunately, we could not find a benchmark suite for FRP programs to examine typical programs that build on the dataflow abstraction, use state and expose pipeline parallelism. Instead, we investigate the applicability of our FRP abstraction to a real benchmark [7] for data streaming engines such as Flink [6], Storm [38] and Spark [44]. They represent the state of the art for big data crunching. This area has received rather little attention from the functional programming community although it borrows a lot from functional programming. The common programming abstraction across these engines is a stream of data with higher-order functions (`map`, `filter`, `groupby` etc.) applied to it. The associated compilers translate these programs into dataflows and execute them pipeline and data parallel. Special higher-order functions allow to specify stateful computations. For example, the Flink program of the benchmark writes as follows:

```
messageStream.rebalance()
             .flatMap(new DeserializeBolt())
             .filter(new EventFilterBolt())
             .<Tuple2<String, String>>project(2, 5)
             .flatMap(new RedisJoinBolt())
             .keyBy(0)
             .flatMap(new CampaignProcessor());
```

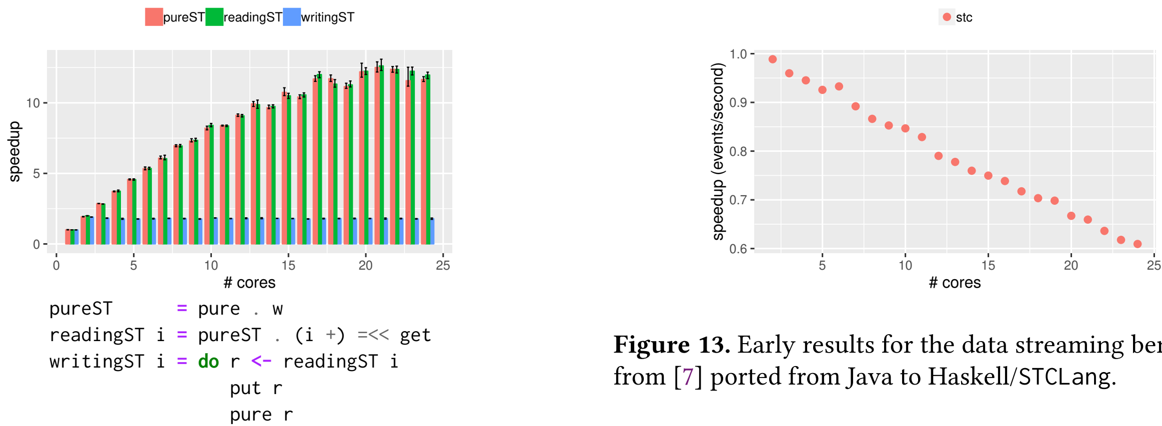
The most interesting function is the `CampaignProcessor` which accumulates the stream values and emits the aggregates to a database (Redis) every 10 seconds. The timer is actually yet another I/O source. The stream-based programming model does not allow to define more than a single I/O source. As such, the programmer has to mix the streaming model with threads and locks to implement such graphs. With `STCLang` and our FRP programming model from Section 4, we can easily implement these semantics.

```
1  streamBench = do
2    -- allocate sources and state threads
3    redisJoin <- liftST rjState redisJoinST
4    processCampaign <- liftST pcState processCampaignST
5    timerSig <- liftSource initialTime timer
6    msgSig <- liftSource kafkaState kafkaReader
7    -- compose the graph
8    filteredProcessor <-
9      filterM evFilterFunc
10             (project >=> redisJoin >=> keyBy 0)
11   return $\src ->
12     timerEv <- timerSig src
13     msgEv <- msgSig src
```



(a) Scalability. **(b)** Framework overhead. **(c)** Pipeline peak.

Figure 11. Experiments of the pipeline benchmark. We compare against LVars, which were initially introduced to support pipeline parallelism in `monad-par`. LVars event handlers do not support state and as such we used a lock to isolate the pipeline-parallel effect.




```
pureST       = pure . w
readingST i = pureST . (i +) =<< get
writingST i = do r <- readingST i
                 put r
                 pure r
```

Figure 12. Pipeline and data parallelism in a simple 2-stage pipeline for different types of state threads. Note that the theoretical maximum speedup for a 2-stage pipeline is 2.

Figure 13. Early results for the data streaming benchmark from [7] ported from Java to Haskell/STCLang.

```
14      procInput <-
15          if' (evCheck timerEv)
16              (Right <$> do
17                  msg <- pure $ deserialize msgEv
18                  join <$> filteredProcessor msg)
19              (pure $ Left timerEv)
20      processCampaign procInput
```

This makes the case that our programming model is a good fit over state-of-the-art programming models in this type of big data processing systems.

Figure 13 visualizes the results of executing the benchmark. The execution does not scale with increasing number of cores. In fact, performance degrades when adding cores/parallelism to the execution. We believe that the reason for this is the work-stealing scheduler that underpins `monad-par`. It does not have a notion of output-favored scheduling known from data streaming systems. This was not a problem in our LVars benchmark because the generator marked the first stage of the pipeline and only `smapGen` spawned tasks for the second stage of the pipeline. As such, there were only tasks from the state thread passed to the `monad-par` scheduler. In the data streaming benchmark, there are various state threads in the pipeline and some of them even perform I/O (not only inside the signal sources). We also noticed that even the sequential version is orders of magnitude slower than an execution with Storm. These preliminary results open up new research directions for the functional programming community. On the other hand, it is also a chance for the big data processing community to gather new concepts. One of them is certainly the STCLang programming model from this paper.

6 Related Work

We compare related work on state threads and arrows.

State Threads There are various approaches to introducing implicit parallel programming [4, 19, 39], but they all work on pure programs only. We are unaware of an approach that uses the notion of a state thread as the foundation for reasoning about parallelism in a program.

In their seminal paper [25], Launchbury and Peyton-Jones already noticed the appeal in using state threads as the foundation for parallelism. They also noted that in order to introduce parallelism, the strictly sequential composition of state transformers via `thenST`, i.e., `bind` in the monad implementation, must be overcome. Therefore they introduced the combinator `interleaveST` that splits the state into two disjoint parts. In their state transformer diagrams, this is visualized as a fork that creates two branches. Similar to our dataflow graphs, branches are independent and as such can be executed in parallel. They concluded: "*The only unsatisfactory feature of all this is that we see absolutely no way to guarantee that the side effects performed in the two branches of the fork are indeed independent.*" To provide this guarantee, our approach composes state threads instead of state transformers. `STCLang`, in particular the formal model defined in the supplementary material [14], generalizes over **`State`** threads and **`ST`** threads. However, the presented `STCLang` implementation so far only works for immutable **`State`** threads because it passes the state along. This is not possible with **`ST`** threads because they strongly encapsulate their state and do not expose it. This monadic state encapsulation was proven to be type-safe [26]. It is also strong enough to provide the same deterministic guarantees (contextual equivalences and refinements) as a "pure" language but for a runST-based effectful language with a global heap and in-place updates (which the authors [37] refer to as *STLang*).

In response to Launchbury and Peyton-Jones' idea of a parallel combinator, Timany et al. [37] note: "*It would be interesting to investigate whether a variation of the parallelization theorem studied for type-and-effect systems in [23] would hold for such a language.*" The theorem essentially states that two expressions can only be executed in parallel if they write to disjoint local regions and consult a shared region only for reading. Informally, this closely maps to our definition of a state thread $(a, s) \rightarrow (b, s)$, where a is located in the shared and s in the local region.

This relation of runST-based languages and type-and-effect systems was already noticed in [25]. The later formalization of Semmelroth and Sabry [35] shows that a translation exists from a language with a type-and-effect system into a runST-based one. A compiler that performs such a translation can use the formal model presented in the supplementary material of this paper [14] to extract pipeline, data and task-level parallelism.

Arrows Arrows and `STCLang` both construct a dataflow computation. The similarity between the `loop` [34] function in the `ArrowLoop` type class and state threads has recently been discovered. [43] Arrows are inarguably very elegant but the arrow functions rely on recursion, which makes it challenging for a compiler to optimize arrow computations. Much of the research on arrows essentially addresses this drawback, leading to more and more sophisticated constructions. `STCLang` builds on very simple and common concepts such as state threads and the monadic bind interface. This allows `STCLang` to directly take advantage of standard compiler optimizations and extract additional (data) parallelism.

Note also that arrow frameworks typically provide two separate operators for sequential and parallel composition. Sequential composition is monadic (>>=) while two arrows can only be composed in parallel with >>>. Here, parallel composition ensures that `IO` can execute concurrently. `STCLang` requires only the monadic bind and our `STCLang` implementation enables parallel execution beyond concurrent `IO` on the foundation of *smap*.

7 Conclusion and Future Work

In this paper we introduced `STCLang`, a programming model for composing state threads that captures enough information about the structure of the state to allow for parallelism to be exploited. This is unlike current approaches with monadic composition, and enables the implementation of a dataflow-based execution model in Haskell. Models like these are the foundation for scalable parallel systems in many domains such as databases, servers and embedded systems.

We have shown how we implemented `STCLang` as a monad in Haskell. For this, we defined the composition of state threads and we have demonstrated how the fine-grain structure of the global state can be used to exploit different forms of parallelism. The extension to support basic concepts from functional reactive programming was straightforward. Our evaluation shows that our programming model is just as easy to use as state-of-the-art parallel programming models. `STCLang` advances over these programming models by providing parallelism implicitly, exploiting pipeline parallelism and enabling stateful computations. Our evaluation shows that our implementation is simple enough such that standard GHC optimizations for lazy evaluation can increase parallelism out-of-the-box. Finally, we ported a state-of-the-art benchmark for data streaming systems and showed that `STCLang` resolves short-comings of the currently used stream-based programming model.

Future Work So far, the state thread abstraction in our implementation cannot support Haskell's **`ST`** threads that operate on mutable (instead of immutable) data. We argue that it would be safe to pass mutable state along in the form of an **`IORef`**. Linear state passing might be able to enforce this safety [5]. Linear types in `STCLang` is certainly an interesting topic for future work.

Our preliminary measurements from the data streaming benchmark indicate that the work-stealing scheduler is not a good fit for pipeline parallelism. This opens up a new research direction for big data systems in the context of functional programming languages and, more specifically to `monad-par`, for pipeline-aware schedulers.

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] 2001. *Implicit Parallel Programming in pH*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[2] Lloyd Allison. 1990. Continuations implement generators and streams. *Comput. J.* 33, 5 (1990), 460–465.

[3] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A Survey on Reactive Programming. *ACM Comput. Surv.* 45, 4, Article 52 (Aug. 2013), 34 pages. DOI: http://dx.doi.org/10.1145/2501654.2501666

[4] Adam D. Barwell and Kevin Hammond. 2017. In Search of a Map: Using Program Slicing to Discover Potential Parallelism in Recursive Functions. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing (FHPC 2017)*. ACM, New York, NY, USA, 30–41. DOI: http://dx.doi.org/10.1145/3122948.3122951

[5] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: Practical Linearity in a Higher-order Polymorphic Language. *Proc. ACM Program. Lang.* 2, POPL, Article 5 (Dec. 2017), 29 pages. DOI: http://dx.doi.org/10.1145/3158093

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.

[7] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, and others. 2016. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 1789–1792.

[8] Antony Courtney and Conal Elliott. 2001. Genuinely functional user interfaces. In *Haskell workshop*. 41–69.

[9] Evan Czaplicki and Stephen N Chong. 2013. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation-PLDI'13*. ACM Press.

[10] Umeshwar Dayal, Malu Castellanos, Alkis Simitsis, and Kevin Wilkinson. 2009. Data Integration Flows for Business Intelligence. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT '09)*. ACM, New York, NY, USA, 1–11. DOI: http://dx.doi.org/10.1145/1516360.1516362

[11] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI'04)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1251254.1251264

[12] J. B. Dennis. 1980. Data Flow Supercomputers. *Computer* 13, 11 (Nov. 1980), 48–56. DOI: http://dx.doi.org/10.1109/MC.1980.1653418

[13] Sebastian Ertel, Justus Adam, and Jeronimo Castrillon. 2018. Supporting Fine-grained Dataflow Parallelism in Big Data Systems. In *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM) (PMAM'18)*. ACM, New York, NY, USA, 41–50. DOI: http://dx.doi.org/10.1145/3178442.3178447

[14] Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. 2019. Category-Theoretic Foundations of "STCLang: State Thread Composition as a Foundation for Monadic Dataflow Parallelism". (2019). arXiv:arXiv:1906.12098

[15] Sebastian Ertel, Christof Fetzer, and Pascal Felber. 2015. Ohua: Implicit Dataflow Programming for Concurrent Systems. In *Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ '15)*. ACM, New York, NY, USA, 51–64. DOI: http://dx.doi.org/10.1145/2807426.2807431

[16] Sebastian Ertel, Andrés Goens, Justus Adam, and Jeronimo Castrillon. 2018. Compiling for Concise Code and Efficient I/O. In *Proceedings of the 27th International Conference on Compiler Construction (CC 2018)*. ACM, New York, NY, USA, 104–115. DOI: http://dx.doi.org/10.1145/3178372.3179505

[17] Andrew Gill, John Launchbury, and Simon L Peyton Jones. 1993. A short cut to deforestation. In *FPCA*, Vol. 93. Citeseer, 223–232.

[18] Goetz Graefe. 1990. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD '90)*. ACM, New York, NY, USA, 102–111. DOI: http://dx.doi.org/10.1145/93597.98720

[19] Tim Harris and Satnam Singh. 2007. Feedback Directed Implicit Parallelism. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*. ACM, New York, NY, USA, 251–264. DOI: http://dx.doi.org/10.1145/1291151.1291192

[20] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* 46, 4, Article 46 (March 2014), 34 pages. DOI: http://dx.doi.org/10.1145/2528412

[21] John Hughes. 2000. Generalising monads to arrows. *Science of Computer Programming* 37, 1 (2000), 67 – 111. DOI: http://dx.doi.org/https://doi.org/10.1016/S0167-6423(99)00023-4

[22] Wesley M Johnston, JR Hanna, and Richard J Millar. 2004. Advances in dataflow programming languages. *ACM computing surveys (CSUR)* 36, 1 (2004), 1–34.

[23] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A Relational Model of Types-and-effects in Higher-order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 218–231. DOI: http://dx.doi.org/10.1145/3009837.3009877

[24] Lindsey Kuper and Ryan R. Newton. 2013. LVars: Lattice-based Data Structures for Deterministic Parallelism. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing (FHPC '13)*. ACM, New York, NY, USA, 71–84. DOI: http://dx.doi.org/10.1145/2502323.2502326

[25] John Launchbury and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 24–35. DOI: http://dx.doi.org/10.1145/178243.178246

[26] John Launchbury and Amr Sabry. 1997. Monadic State: Axiomatization and Type Safety. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*. ACM, New York, NY, USA, 227–238. DOI: http://dx.doi.org/10.1145/258948.258970

[27] Edward Ashford Lee and David G Messerschmitt. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on computers* 100, 1 (1987), 24–35.

[28] Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 333–343. DOI: http://dx.doi.org/10.1145/199448.199528

[29] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. 2008. Flask: Staged functional programming for sensor networks. *ACM Sigplan Notices* 43, 9 (2008), 335–346.

[30] Simon Marlow, Ryan Newton, and Simon Peyton Jones. 2011. A Monad for Deterministic Parallelism. In *Proceedings of the 4th ACM Symposium on Haskell (Haskell '11)*. ACM, New York, NY, USA, 71–82. DOI: http://dx.doi.org/10.1145/2034675.2034685

[31] Conor McBride and Ross Paterson. 2008. Applicative Programming with Effects. *J. Funct. Program.* 18, 1 (Jan. 2008), 1–13. DOI: http://dx.doi.org/10.1017/S0956796807006326

[32] Ryan R. Newton, Ömer S. Ağacan, Peter Fogg, and Sam Tobin-Hochstadt. 2016. Parallel Type-checking with Haskell Using Saturating LVars and Stream Generators. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, New York, NY, USA, Article 6, 12 pages. DOI: http://dx.doi.org/10.1145/2851141.2851142

[33] Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. ACM, 51–64.

[34] Ross Paterson. 2001. A New Notation for Arrows. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*. ACM, New York, NY, USA, 229–240. DOI: http://dx.doi.org/10.1145/507635.507664

[35] Miley Semmelroth and Amr Sabry. 1999. Monadic Encapsulation in ML. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99)*. ACM, New York, NY, USA, 8–17. DOI: http://dx.doi.org/10.1145/317636.317777

[36] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Springer-Verlag, London, UK, UK, 179–196. http://dl.acm.org/citation.cfm?id=647478.727935

[37] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2017. A Logical Relation for Monadic Encapsulation of State: Proving Contextual Equivalences in the Presence of runST. *Proc. ACM Program. Lang.* 2, POPL, Article 64 (Dec. 2017), 28 pages. DOI: http://dx.doi.org/10.1145/3158152

[38] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, and others. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 147–156.

[39] José Manuel Calderón Trilla and Colin Runciman. 2015. Improving Implicit Parallelism. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*. ACM, New York, NY, USA, 153–164. DOI: http://dx.doi.org/10.1145/2804302.2804308

[40] Philip Wadler. 1992. The Essence of Functional Programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*. ACM, New York, NY, USA, 1–14. DOI: http://dx.doi.org/10.1145/143165.143169

[41] Zhanyong Wan and Paul Hudak. 2000. Functional reactive programming from first principles. In *Acm sigplan notices*, Vol. 35. ACM, 242–252.

[42] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: an architecture for well-conditioned, scalable internet services. In *ACM SIGOPS Operating Systems Review*, Vol. 35. ACM, 230–243.

[43] Jeremy Yallop and Hai Liu. 2016. Causal Commutative Arrows Revisited. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, New York, NY, USA, 21–32. DOI: http://dx.doi.org/10.1145/2976002.2976019

[44] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301