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

By Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens and Jerónimo Castrillón-Mazo
We want...

- Performance
- Responsiveness

**Parallelism in the presence of State**

- Efficient
- Intuitive

\[
\text{mapM } (\lambda x \rightarrow f x >>= g) \ldots
\]

Monads are a powerful and familiar abstraction, but inherently sequential.

Functions depend on entire state.

Parallelism arises implicitly from model.

Granularity implicit from state scope\(^1\).

Use Case
- FRP
- Data Streaming

Functions depend on local state only.

---

State in Haskell

Lots of references strewn about

Array s b

STRef s c

ST s a

Single user defined state

get :: StateT s m s

put :: s → StateT s m ()

StateT s m a

State in Haskell

Needs Alias Analysis to disentangle

Lots of references strewn about

Array s b
STRef s c

Both antithetical to Parallelism

Can both be used to implement State Threads with local state

Theorized

Realized

put :: s → StateT s m ()

ST s a
StateT s m a

Opaque

Single user defined state

Smap (De)Construction

\[ \text{smap} \ (\lambda x \rightarrow f \ x \ >>=> g) \ [x_1, x_2] \]

State source decoupled from input (aka. \(>>=\))

Familiar monadic composition

Computations spawned in parallel

Items still sequential via \(>>=\)

Cheap write-once channels (IVar) synchronize state

Pipeline Parallel Execution arises implicitly, like in Dataflow

Smap (De)Construction

Sequential Guarantees
- Effects happen in element order
- Computation happens in composition (\( >>= \)) order

State source decoupled from input (aka. \( \gg \))

Cheap write-once channels (IVar)
synchronize state

Sequential Guarantees are sufficient to express FRP

Pipeline Parallel Execution arises implicitly, like in Dataflow

Functional Reactive Programming

Smap over one element at a time

Non-deterministic merge for ordering

Effect-only node

\[
\text{step} :: \text{Monad } m \\
\Rightarrow \text{Generator } m \text{ a } \rightarrow m (\text{Maybe } (\text{a, g a}))
\]
Functional Reactive Programming

Smap over one element at a time

- Non-deterministic merge for ordering
- All events dispatch to all source nodes
- Only matching source updates its value

- Tivial (~50 LOC)
- Using a filter avoids recompute. Returns last value instead (using state)
- Effect-only node
- Synchronizes via state

- Other sources repeat old value ⇒ No glitches

step :: Monad m
  ⇒ Generator m a → m (Maybe (a, g a))
Other Forms for Parallelism

Task Parallelism

Concisely expressed with the `<*>` operator

1. Spawned off

2. Compute this

\( x \mapsto f(x) <*> g(x) \)

3. Collect results & apply

Can be done automatically with the ApplicativeDo GHC Extension\(^1\)

Data Parallelism

Being lazy in `a` allows `s` to be sent on first

\(~(a,s) \leftarrow \text{run } f\)

If `s` unchanged, virtually free

- evaluate `s`
- send `s`

Next instance of `f` can start before this point

- evaluate `a`

Node Implementation

Works for both read-only and unused state

---

Other Sources for Parallelism

Task Parallelism

- We formalized the model using category theory
- Yields a formal explanation for the forms of extracted parallelism

Data Parallelism

Can be done automatically with the ApplicativeDo GHC Extension

Data Parallelism

Benchmark executes State Thread (f or f_write) twice, as a two stage pipeline

\[ f \ i = (i + 1) \] get

\[ f\_write \ i = \text{do} \]
\[ r \leftarrow f \ i \]
\[ \text{put } r \]
\[ \text{pure } r \]

Same as f but also writes state

\[ f\_write \text{ exploits pipeline parallelism (} \sim 2x \)\]

\[ f \text{ additionally exploits data parallelism (linear scaling) }\]
Microbenchmarks

For simplicities sake only includes the more favorable monad-par measurement (par2 in the paper)


STCLang is implemented using the monad-par library

Overhead over manual monad-par implementations is mostly negligible
MapReduce Benchmark

Regular MapReduce

map -> smap -> map -> collect -> reduce

Reduce via State and Map

smap -> map -> reduce -> reduce

Can express multiple concurrent reduces

Sum is very efficient, (+) is too cheap for the overhead

par = sum <$> parMap euler
stc = mapReduceSTC euler (+) 0
stc' = sum <$> smap euler
Streaming Data Streaming Benchmark

Work stealing scheduler does not have a notion of output favoured scheduling.

Performance degrades with parallelism.

Conclusions

\[ \text{Monad State } \checkmark \text{ Parallelism } \times \text{ Composition } \checkmark \]

\[ \text{Dataflow State } \checkmark \text{ Parallelism } \checkmark \text{ Composition } \times \]

**State Monad**
+ Dataflow Node =

**Monad Composition**
+ Dataflow Execution =

- Monad/\text{smap}
- STCLang
- State Thread
- Fine grained state structure
- Implicit Pipeline-/Task-/Data-parallelism
- Leveraging
  - Laziness
  - ApplicativeDo
- Notion of local state
- MapReduce
  - Without explicit reducer
- FRP/Stream Processing
  - Glitch free by default
  - Straightforward

Repo: [https://github.com/ohua-dev/stc-lang](https://github.com/ohua-dev/stc-lang)
Hackage: [https://hackage.haskell.org/package/stc-lang](https://hackage.haskell.org/package/stc-lang)
The STC Monad

For Sebastian: This is my backup slide to explain our monad construction. I like it, but its also dense, so I relegated it as backup.
The Sequential Monad

- Convenient, familiar
- Inherently sequential
- Only task parallelism
- No flow from \( f \) to \( g \) before the effects of \( f \)
- Effect composition independent from data
- Does anyone use Arrows? (You should though, they’re cool)
State Thread Examples

memoized :: Ord a
          => (a -> b) -> a
          -> State (LRUCache a b) b
memoized operation elem = do
  cache <- get
  case lookup elem cache of
    Just result -> return result
    Nothing -> do
      let res = operation elem
      put $ insert elem result cache
      return result

windowedAvg :: Int -> Float
              -> State [Float] Float
windowedAvg wsize i = do
  win <- get
  let win' = take wsize $ i : win
  put win'
  return $ sum win' `div` realToFrac wsize

Microbenchmarks

1. For simplicities sake only includes the more favorable monad-par measurement (par2 in the paper)
2. NOINLINE version of the benchmark, as it is representative of what we want to test. See the paper for the complete rational

```
sc w = smap (w ==> w)
pw w = parMap (w . w)

go w = (\x ->
  (,) <$> w x
  <$> w x)

sc = smap . go
paw = parMapM . go

go cond w x =
  if cond x
    then w x
    else w x

sc = smap . go even?
paw = parMap . go even?
```
Benchmarks

Matrix Multiplication

Black-Scholes

Mandelbrot

No slowdown (with respect to \texttt{par})

Near-linear scaling