Transparent Synchronous Dataflow

Steven W.T. Cheung
University of Birmingham

Abstract

The various functional approaches to dataflow programming represent trade-offs between effectiveness and efficiency. On the one hand, we have lower-level approaches which are highly expressive but permit the formulation of programs with significant space-time inefficiencies. On the other hand, we have more abstract approaches which guarantee efficiency but at the cost of restricting expressiveness. As is generally the case, the more abstract idioms also make it harder for the programmer to write erroneous programs. Perhaps the most abstract style of dataflow programming is *self-adjusting computation* (SAC), which replaces the explicit notions of *event* and *signal* with implicit dependency links between memory cells, which propagate changes automatically and transparently. We call this *transparent dataflow*. It is abstract, concise, and efficient, yet it suffers from a significant restriction which limits its usefulness. It disallows dependency cycles. We introduce a new language for transparent dataflow, which generalises it to dependencies with cycles, greatly increasing its applicability. To handle circular dependencies we change the propagation mode from asynchronous to *synchronous*, which enables further applications such as filters and pipelines. The language is defined via an abstract machine which can be used to reason about soundness and (theoretical) efficiency, confirming that the language has desirable properties. We also provide a semi-naive practical implementation as an OCaml PPX extension, validating its (practical) efficiency via benchmarks.

1 Introduction

One broad class of programming languages and idioms is *dataflow programming* [7]. Its central concern is the management of streams of data. Programs are conceived as computation graphs and computation as propagation of data or changes to data. Particularly elegant is the way dataflow programming appears in functional languages, a paradigm known as *functional reactive programming* (FRP), which relies on the convenience of stream programming in a functional setting [4, 17, 6, 13].

If FRP can be seen as streams programming with no access to individual events, *self-adjusting computation* (SAC) can be seen as a further abstraction in this paradigm, obtained from hiding the streams (the "signals") altogether [1]. Instead, in SAC a special syntactic and semantic category of *cells* or *variables* is introduced, with all terms depending on these cells being automatically recomputed on-demand when the cell is changed. The changing of a cell triggers an event, which automatically propagates along a computation graph tracking the dependencies, an implicit signal, until it can be observed at the other end. Although the original motivations for SAC are different, a sophisticated memoisation technique known as "incremental computation" [15], it is not unusual to use it as a form of *transparent FRP*, bridging from FRP into object-oriented patterns such as *observer* or *model-view-controller* [16].

A commonly invoked metaphor is that of the spreadsheet, where changing data in a cell automatically triggers the recalculation of all the dependent cells, their dependants, and so on. Below we give a simple example, in a simplified syntax inspired by a popular SAC library[1].

```
let xv, yv = SAC.create 1,     SAC.create 2    in
let x, y   = SAC.watch xv,     SAC.watch yv    in
let m, mo  = SAC.map2 x y max, SAC.observe max in
SAC.stabilize (); print (SAC.value mo); SAC.set xv 3;
SAC.stabilize (); print (SAC.value mo)
```

We use the SAC module name to indicate the provenance of functions for creating variables, observing computations, and retrieving values of computations. Note the "stabilise" operation which triggers the automatic recomputations of all the dependent computations. The program above should print first 2 then 3.

Abstracting away from explicit signals allows a very efficient implementation of the computation graph and the propagation of data changes along the dependencies. Moreover, the transparent dataflow style is restrictive enough to prevent the formulation of any expensive programs. Additionally, working at a higher level of abstraction, such programs are more succinct and less error-prone. However, SAC does not allow the presence of feedback loops, thus preventing many FRP applications discussed earlier, in particular composite stateful systems.

To summarise, we propose a new programming language construct, *transparent synchronous dataflow* (TSD). The way it relates to existing programming language styles can be summarised in the following table:

	Efficiency	Expressiveness
Streams	inefficent (history)	unrestricted
Monadic FRP	inefficient (join)	unrestricted
Arrowised FRP	unknown	unrestricted
SAC	efficient	restricted no feedback
SSD	efficient	restricted feedback

[1] `https://github.com/janestreet/incremental`

To reconcile feedback with the transparent style of SAC we require the feedback to be guarded by delays, a standard solution, which in turn makes our style of programming synchronous in the sense that updates are processed in rounds consisting of the update of all cells that need updating no more than once. This allows us to opportunistically occupy another gap in the language design space, that of transparent synchronous languages. Asynchronous dataflow and FRP are found in many incarnations, whereas synchronous counterparts are rarer but do exist [2, 5]. If we think of SAC as the transparent counterpart of FRP, TSD is, for the first time, the transparent counterpart of synchronous reactive languages such as ReactiveML [11].

2 Transparent Synchronous Dataflow

To give a quick impression of the language, a state machine is created from an initial state and a transition function by the code below, where `{init}` creates an initialised cell in the dataflow graph, `link s to g` updates a dependency between the value in a cell `s` and a dataflow graph `g` and `deref c` creates dataflow graph depending on a cell `c`.

```
let state_machine init trans inp =
  let state = {init} in
  link state to (trans state inp); state
```

In particular, the `link` above creates a cyclic dependency in which the state cell depends on its previous state. Using it we can create a one-zero alternating stream as follow.

```
let alt = state_machine 1 (fun s -> 1 - deref s) 0
```

A state machine accumulating in the state the running sum of its input can be created as below, and easily composed with the alternating state machine:

```
let sum inp =
  state_machine 0 (fun s i -> i + deref s) inp
let alt_sum = sum (deref alt)
```

The command `step` runs the dataflow graph for one step, i.e. until all cells have been updated at most once. This is the same synchronous execution model encountered in clocked digital circuits, and it corresponds to one clock cycle. Inspecting the value of the state machine (`peek alt_sum`) after 3 cycles is achieved by

```
step; step; step; peek alt_sum
```

The program should return 2. This example is preloaded in the online visualiser[2].

2.1 Diagrammatic Semantics

The TSD calculus extends an applied, simply-typed lambda calculus with a notion of 'cell' and operations to structure cells (dereferencing, peeking, linking) into *computation graphs* which transparently propagate all changes upon request (stepping). The semantic model is a graph-rewriting abstract machine called *the Dynamic Geometry of Interaction* (DGOIM) which serves two purposes. First, it allows us to reason about TSD code, in particular to prove type safety. Second, it is a concrete model, giving an effective specification of the language in which it can be immediately established that the key operations of TSD involving the creation and management of the computation graph, are efficient, namely at most linear in the size of the computation graph, in space and time. The fact that DGOIM also gives an efficient model of the underlying lambda calculus (with several common reduction strategies) is proved elsewhere [12].

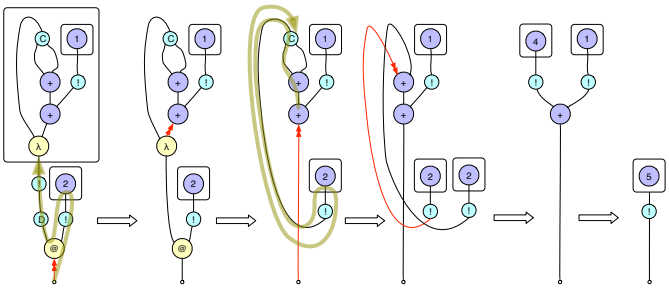

Figure 1: DGOIM evaluation of $(\lambda x.x + x + 1)\ 2$

The DGOIM represents a term as computation graphs with a selected edge ("token"). The token traverses the graph and rewrites it according to collected information. Let us consider a basic lambda term, with no dataflow. For simple expressions the computation graph resembles the syntax tree. Constants and operations are nodes and variables are edges. Abstraction is represented as a cycle in the graph. Contraction (a variable used many times) is handled by dedicated contraction nodes. All these features are illustrated by the term $(\lambda x.x + x + 1)\ 2$.[3] Other elements (blue) are linear logic artefacts for managing the bureaucracy of copying or sharing sub-graphs. Finally, some sub-graphs are identified in "boxes" which means that they are treated as a whole for the purpose of sharing or copying. The sequence of rewrites is in Fig. 1. The token is a double red arrow (some reduction steps are omitted). A highlighting yellow arrow shows the trajectory of the token when no significant rewrites occur. After visiting the argument of the first application, as right-to-left CBV evaluation requires, the first rewrite is the "opening" of the box encapsulating the lambda term. Immediately after, the λ-@ pair of nodes is eliminated, fusing the edge corresponding to the argument of the function to the argument 2. After a few propagation steps the token reaches the contraction node (C), which triggers the copying of the argument 2, thus completing the beta reduction. The final few steps reduce the arithmetic expression $2 + 2 + 1$ to the constant 5.

Dataflow features fit into this semantic model quite naturally as the DGOIM provides explicit control over what

[2] `https://anonymousgithubaccount.github.io/SSD-vis?ex=alt`

[3] `https://anonymousgithubaccount.github.io/SSD-vis/?ex=basic`

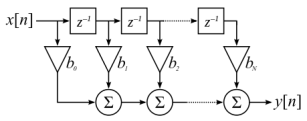

Figure 2: A discrete order-N FIR filter

to share and what to copy. In contrast to simple values which are copied by contractions, dataflow graphs are shared among the rest of diagram[4].

2.2 A natural example

A common application of synchronous reactive programming is finite-input response (FIR) filters. FIR filters do not require feedback, but are made much easier by the assumption of synchrony. These filters have a wide number of practical applications. They are stable, since the output is a sum of a finite number of finite multiples of the input values. They can be designed to be *linear phase* (i.e. the phase response of the filter is a linear function of the signal frequency), by making the coefficient sequence symmetric. This property is desirable in many applications, such as data communications, seismology, and crossover filters. FIR filters in general have numerous other applications, from signal processing to machine learning.

A FIR filter computes the sum

$$y[n] = b_0x[n]+b_1x[n-1]+\cdots+b_Nx[n-N] = \sum_{i=0}^{N} b_i \cdot x[n-i],$$

where $x[n], y[n]$ means the n-th element of the sequence. The schematic representation is in Fig. 2.

The definition of a generalisation of an order N filter in our transparent style is remarkably succinct and elegant.[5]

```
let rec fir x = function
  | [] -> x, 0
  | f :: fs -> let (inp, sum) = fir x fs in
               let s = {0} in
               link s to inp;
               s, f s + sum
```

This is a generalisation because we are using arbitrary functions at each stage rather than multiplication by constant coefficients b_i. The function `fir` returns the interface to the filter as a pair of inputs and outputs. A concrete FIR filter, such as a running average of order 3 can be instantiated from the above as

```
let avg3 x =
  let w = fun x -> x / 3 in
  let (_, sum) = fir x [w; w; w] in sum
```

3 The TSD language

Based on the calculus we implement the TSD language as a module library in OCaml together with a language extension[6]. The library serves as an embedded DSL, which is blended into the host language more seamlessly via the language extension. There are two implementations, with and without incrementalisation. The implementation is not particularly exciting, since it resorts to OCaml's imperative features and its facilities for type-casting. The "in principle" efficiency stated as a theoretical result in the preceding section is now accompanied by an practical (reasonably) efficient implementation, as indicated by several benchmarks (see appendix).

```
let state_machine init trans inp =
  let state = cell (lift init) in
  link state [%dfg (lift trans) state inp]; state

let alt = state_machine 1 (fun s _ -> 1 - s) (lift 0)
let sum inp = state_machine 0 (fun s i -> i + s) inp
let alt_sum = sum alt
let _ = step (); step (); step (); peek alt_sum
```

Figure 3: Communicating state machines in TSD

The state machine example shown in section 2, in the concrete language is given in Fig. 3. The `[%dfg t ]` ppx extension instructs the compiler to interpret a term `t` as a dataflow graph. The `lift` function maps an OCaml term into the corresponding dataflow graph so that it can be used inside a `[%dfg ... ]` context.

Comparison to the JS Incremental library As mentioned in the Introduction, incremental programming can be used as a transparent dataflow programming. We were inspired by this idiom and our main goal was to lift the restriction on feedback loops in order to support new useful applications. This of course leads to deep semantic differences. In Incremental change propagation is asynchronous and it can be run until the computation graph stabilises. In TSD change propagation is synchronous in the sense that propagation is carried out until all cells that need update are updated at most once. In feedback free dataflow graphs the behaviour of the two is equivalent. In the presence of feedback asynchronous update is illegal and synchronous stabilisation may diverge.

Comparison to Reactive ML Semantically TSD is more similar to Reactive ML, which also has a synchronous model. Therefore the two languages can target similar applications. However, the fact that Reactive ML is event-based leads to strikingly different styles of programming. In TSD the tokens serve the role of signals in Reactive ML, and are handled automatically and implicitly. Moreover, Reactive ML signals are sent into a global scope, which makes composition more cumbersome. For example the parallel composition of a process with itself could not easily discriminate which of the two instances is emitting a signal. This problem does not appear in TSD.

[4] Check out `https://anonymousgithubaccount.github.io/SSD-vis/?ex=dataflow` to see the semantics in action

[5] `https://anonymousgithubaccount.github.io/SSD-vis?ex=fir`

[6] `https://github.com/anonymousgithubaccount/SSD`

References

[1] Umut Acar. *Self-Adjusted Computation.* PhD thesis, Carnegie-Mellon University, 2005.

[2] Gérard Berry and Laurent Cosserat. The esterel synchronous programming language and its mathematical semantics. In *International Conference on Concurrency*, pages 389–448. Springer, 1984.

[3] Olivier Danvy. A rational deconstruction of landin's secd machine. In *Symposium on Implementation and Application of Functional Languages*, pages 52–71. Springer, 2004.

[4] Conal Elliott and Paul Hudak. Functional reactive animation. In *ACM SIGPLAN Notices*, volume 32, pages 263–273. ACM, 1997.

[5] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[6] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *International School on Advanced Functional Programming*, pages 159–187. Springer, 2002.

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

[8] Peter J Landin. The mechanical evaluation of expressions. *The computer journal*, 6(4):308–320, 1964.

[9] Peter J Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[10] Ian Mackie. The geometry of interaction machine. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 198–208, 1995.

[11] Louis Mandel and Marc Pouzet. Reactiveml: a reactive extension to ml. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 82–93. ACM, 2005.

[12] Koko Muroya and Dan R. Ghica. The dynamic geometry of interaction machine: A call-by-need graph rewriter. In Valentin Goranko and Mads Dam, editors, *26th EACSL Annual Conference on Computer Science Logic, CSL 2017, August 20-24, 2017, Stockholm, Sweden*, volume 82 of *LIPIcs*, pages 32:1–32:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

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

[14] Gordon D Plotkin. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60:3–15, 2004.

[15] Ganesan Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 502–510. ACM, 1993.

[16] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. Rescala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th international conference on Modularity*, pages 25–36. ACM, 2014.

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

A Benchmarks

Even though our implementation is an initial proof-of-concept without complex optimisations the performance is quite good. We made two kinds of tests, against Reactive ML and against JS Incremental, which are quite different. Reactive ML has a similar underlying propagation model which allows the programming of similar applications, but in a very different style of implementation. In contrast, JS Incremental is a similar idiom which allows the comparison of similar-looking programs. However, since JS Incremental is an implementation of (asynchronous) self-adjusting computation TSD is a strict superset in terms of what programs can be written. Both Reactive ML and JS Incremental are OCaml-based. The former is an ML-like compiler which produces OCaml code, while the latter is a library and PPX extension. All executables were ultimately produced with the OCaml compiler version 4.03.0, which makes for a reasonably fair comparison. On the TSD side we benchmark both the simple and the incrementalised version of the implementation.

To compare with Reactive ML we have used the standard examples in the distribution. For cellular automata and an n-body simulation (planets) we have tested several sizes of cells/bodies. For finite-state automata TSD implementation is so much simpler and faster as a function of number of states that we have tested as a function of run-length instead, which is more similar. Alt-sum calculates the running sum of an alternating signal, which were tested with several sizes of inputs. Finite-impulse response (FIR) filter, infinite-impulse response (IIR) filter and page-ranking algorithms are common applications that requires delay in the input and feedback. Two implementations for each of the three programs were tested, *naive* and *optimised*. Huge performance gain was observed in the Page-ranking benchmarks. The former creates plenty of intermediates thunks that multiples the inputs and the weights together while the later simply create a single thunk for dot product. Map, fold-sum and fold-max tested the efficiency of propagation with small changes to the input. Method (1) creates intermediate cells during each recursion of the fold while method (2) does not. Since the cells behave like caches in the dataflow graph therefore the incremental TSD benefits from method (1) however the TSD performance was worse due to the large number of cells. In general the TSD performance was similar. The incremental TSD had similar or worse performance than the unoptimised version since in the presence of feedback cells never stabilise.

To compare with JS Incremental we tested propagation along three kinds of (cycle-free) dataflow graphs: long paths (chains), wide shallow graphs (fields), and tree-like graphs (trees), with small changes to the input. These tests favour JS Incremental's model and serves its sophisticated implementation. Unsurprisingly in all these cases the TSD incremental implementation had a weaker or similar performance, but of the same magnitude as JS Incremental, whereas the unoptimised version was very slow on large graphs and unusable on very large graphs. Considering ours is an initial, semi-naive, implementation the small performance gap is encouraging.

	n	RML	SSD	IncSSD
Cellular (n-cell)	10^2	0.008s	0.002s	0.010s
	10^3	0.863s	0.011s	0.012s
	10^4	53.73s	1.332s	0.758s
Planets (n-body)	10^1	0.008s	0.006s	0.011s
	10^2	0.388s	0.386s	0.418s
	10^3	32.56s	36.64s	38.98s
Automata (n-step)	10^5	0.061s	0.011s	0.018s
	10^6	0.549s	0.103s	0.152s
	10^7	5.303s	0.978s	1.427s
Alt-sum (n-input)	10^5	0.026s	0.009s	0.022s
	10^6	0.233s	0.075s	0.186s
	10^7	2.149s	0.699s	1.731s
FIR (naive) (n-delay)	10^1	0.048s	0.022s	0.080s
	10^2	0.188s	0.188s	0.609s
	10^3	2.092s	2.147s	7.877s
FIR (opt) (n-delay)	10^1	0.048s	0.022s	0.073s
	10^2	0.188s	0.186s	0.615s
	10^3	2.092s	2.023s	7.433s
IIR (naive) (n-delay)	10^1	0.054s	0.113s	0.286s
	10^2	0.349s	1.159s	3.383s
	10^3	4.646s	16.36s	1m36s
IIR (opt) (n-delay)	10^1	0.054s	0.086s	0.245s
	10^2	0.349s	0.946s	2.815s
	10^3	4.646s	13.54s	59.74s
P-Rank (naive) (n-page)	10^1	0.002s	0.002s	0.002s
	10^2	0.013s	0.086s	0.182s
	10^3	2.582s	13.25s	9m53s
P-Rank (opt) (n-page)	10^1	0.002s	0.002s	0.002s
	10^2	0.013s	0.042s	0.054s
	10^3	2.582s	7.562s	20.292s
Map (n-size)	10^3	0.002s	0.002s	0.002s
	10^4	0.003s	0.014s	0.020s
	10^5	0.028s	0.251s	0.250s
Sum (1) (n-size)	10^2	0.002s	0.001s	0.001s
	10^3	0.002s	0.089s	0.002s
	10^4	0.002s	27.82s	0.054s
Sum (2) (n-size)	10^2	0.002s	0.001s	0.001s
	10^3	0.002s	0.002s	0.513s
	10^4	0.002s	0.005s	10m54s
Max (1) (n-size)	10^2	0.002s	0.002s	0.001s
	10^3	0.002s	0.085s	0.002s
	10^4	0.002s	2.688s	0.023s
Max (2) (n-size)	10^2	0.002s	0.001s	0.002s
	10^3	0.002s	0.001s	0.535s
	10^4	0.002s	0.007s	10m44s

Table 1: Comparison to Reactive ML

We can conclude that even as a slightly-optimised proof-of-concept implementation the performance of TSD is satisfactory. It is comparable to Reactive ML and is in the ball park with JS Incremental – all this while supporting a convenient and succinct, while still expressive, style of programming.[7]

[7]Benchmark code: `https://github.com/anonymousgithubaccount/SSD/tree/master/benchmarks`

	n	JS Inc	SSD	IncSSD
Chain (n-chain)	10^3	0.007s	0.059s	0.002s
	10^4	0.009s	10.79s	0.010s
	10^5	0.053s	-	0.156s
	10^6	0.613s	-	1.684s
	10^7	6.479s	-	17.11s
Field (n-cell)	10^3	0.003s	0.001s	0.001s
	10^4	0.013s	0.004s	0.006s
	10^5	0.023s	0.065s	0.017s
	10^6	0.125s	0.813s	0.079s
	10^7	1.042s	8.508s	0.641s
Tree (n-cell)	10^3	0.006s	0.002s	0.002s
	10^4	0.009s	0.008s	0.004s
	10^5	0.021s	0.053s	0.010s
	10^6	0.117s	0.799s	0.066s
	10^7	1.116s	8.093s	0.629s

Table 2: Comparison to JS Incremental