Taking Back Control (Flow) of Reactive Programming

Position Paper

Sean McDirmid

Microsoft Research
Beijing China
smcdirm@microsoft.com

Abstract

Event-driven programming avoids wasting user and CPU time, but is difficult to perform since program control flow is necessarily inverted and twisted. To make reactive programming easier, many advocate burying control flow within abstractions that are composed via data flow instead. This might be a mistake: data-flow has issues with expressiveness and usability that might not pan out. Instead, control flow could be re-invented to hide the adverse affects of CPU time while preserving expressiveness and directness[1].

The Rise of Data Flow

In the beginning, programs would continuously *poll* for something to happen. Polling is simple and direct; consider:

```
while true:
  clearScreen()
  if button.pushed:
    x += 1
  print(x)
```

Unfortunately, very few were happy about polling: the user was stuck waiting for the computer to finish, while the computer was doing nothing very useful in the meantime.

Event handling notifies programs when something happens so they do not need to wait and check all the time; e.g.:

```
button.onPushed(cbA)
label.content = x

def cbA():
  x += 1
  label.content = x
```

[1] This position paper motivates managed time work presented in [5].

The programmer installs a "callback" that gets called when the button is pushed (cbA); the implementation of the cbA callback must also tell label that it has new content, and should redraw itself, since what it is showing (x) has changed. In particular, much complexity is needed to ensure that changes are propagated when needed. Although events gain efficiency, the simplicity and directness of polling is lost.

Various data-flow approaches, like Rx [7] or FRP (functional reactive programming) [2] aim to tame the complexity of event programming. Rather than instruct the computer directly, programmers route data through the program; e.g.

```
print(count(button.pushed))
```

Here button.pushed is a stream of push events that is transformed by count into a stream of counts, which is then consumed by print to display the most recent count; control flow is completely buried by code that merely pushes event streams around. The resulting code is very concise, meshing with the declarative mantra of expressing **what** rather than **how**. Clearly data-flow is the way forward, right?

The Limits of Data Flow

Data flow is best understood as encoding a program as a static network of vertices and connections. For our example's graph, button's pushed flows into count, which then flows into print. The program then more resembles an electrical circuit than a procedural recipe. Pure data flow is then one giant form of indirection in that programmers can only influence control flow indirectly in how they wire up data flow relationships. Control encapsulation is great when data-flow is in focus, but is problematic if nuanced control is necessary. Consider Rx-like [7] code for a mouse drag adapter:

```
def drag(widget):
  var p = from tm in widget.mouseDown
    let pw = widget.position
    from nm in widget.mouseMove.until(widget.mouseUp):
      select pw + nm - pm
  widget.positionAt(p)
```

This example uses LINQ syntax where from initiates a map whose transformation is determined by a nested select. The drag method will route positions to its argument based on mouse down, move, and up events; But its logic is broken:

for efficiency reasons on many platforms, including the context of this example[2], mouse up and down events are only delivered to entities under the mouse: it is very easy to move the mouse off widget and no longer "hear" its mouse events. The solution to this problem is typically to "capture" the mouse on widget when the mouse goes down and release the capture when the mouse goes up, but such imperative operations require imposing on event streams with conventional event handlers, ruining most of this solution's elegance!

In a pure FRP data flow systems, state can accumulate within each node but cannot be shared via aliases, which would otherwise create implicit communication channels. This results in a very different way of programming; e.g. collections that change over time cannot be managed with "insert" and "remove" calls, but instead must "switched" in a way where their contents are defined by a function; e.g.

```
gameCore objs =
  dpSwitch route
    objs
    (arr killOrSpawn >>> notYet)
    (\sfs f -> gameCore(f sfs'))
```

In this example, taken from [1], dpSwitch takes a route function that routes signal inputs (event streams) to collection members and a killOrSpawn function that adds or removes elements from the collection. Basically, all interactions in a pure data flow system must be obvious via explicit connections; e.g. a collection's membership and external interaction must occur all at once through dpSwitch. Interactions in an imperative system can be diffused throughout various encapsulated procedures; e.g. they can populate a collection or iterate over the collection to create interactions that are encapsulated from the code calling the procedures.

The difference between data-flow and control flow systems have important modularity implications: data flow systems encapsulate all control flow but not any interactions between nodes; while control flow systems encapsulate interactions. The data-flow code is more clear but is also brittle with respect to change; e.g. adding new collection membership behavior requires a top level change to the input functions of dpSwitch. Imperative code basically boils down to "spooky actions at a distance" but is also incredibly malleable to new behavior; simply add a procedure that does it!

Another issue with data-flow programming is that while it is *incremental* in the streaming sense–new events added to a stream are immediately propagated–it is not incremental with respect to non-monotonic changes in those streams. Consider X = Y + Z, where X is a stream with values that correspond to the sum of values in the Y and Z streams. However, changes to any element of the X stream can only be dealt with by backing up the entire program and re-building the streams! In contrast, change is a first-class entity in the non-streaming imperative world: if Y or Z are assigned new values, we know that they have changed (via change propagation), which can drive the recomputation of X. The representation of changing values as pure event streams simply provides no basis to deal with non-monotonic change, though memoization can help in this regard [3].

Finally, the challenge of debugging data-flow code cannot be ignored. Given encapsulated control flow, programmers are guided to reason about and examine program execution at the level of its data flow. Debugging then becomes a matter of observing event flows as a circuit engineer would perfom with probes and oscilloscopes—as control flow is encapsulated programmers cannot easily set breakpoints or inject printfs to track down problems. Debugging without control flow is quite different and not currently well understood as evidenced by the lack of data-flow debuggers to survey.

Immediate-mode Programming

One of the primary advantages of data-flow systems is how they "abstract" over time: the dataflow network is continuously re-evaluated as new events are driven thorugh it. Could get this advantage without encapsulating control flow? Enter polling-intensive *immediate-mode UI* APIs; consider:

```
slider(GEN_ID, 15, 15, 100, ref value)
label("hello " + value, 15 * value, 30)
```

This code uses Sol [4], an immediate-mode UI toolkit for games. The value variable is updated when the slider is moved; no explicit event handler is needed because the code will be called if a value changed event is pending. The value variable then determines the horizontal placement of the "hello" label. Processing [8] uses similar abstractions to make UI programming more accessible to designers and artists. The advantage of immediate-mode programming are simplicity and directness: control flow is not inverted or encapsulated outside of simple procedure constructs.

Unfortunately, the disadvantages of immediate-mode programming are many. Not only is it inefficient, but it also does not support state or identity encapsulation. Consider the Sol slider code: the value variable both determines where the slider thumb is and holds where the user puts it, so it has to persist across render calls. For that to occur, value must be declared as a global outside of the continuously executed redraw loop–the slider method cannot encapsulate any state! Additionally, the slider needs an identity that is stable across redraws for the purpose of event routing, supplied here by a GEN_ID macro that would not work directly in a loop.

Managed Time

For immediate-mode programming to be viable, it must be efficiently executed with full support for identity and state encapsulation. State changes should also be handled *incrementally* since many reactive programs are oriented around responding to change. We meet these requirements with what we call *managed time* [5] where control flow is reinvented to hide pesky computer time. Code in this paradigm

[2] http://jasonrowe.com/2010/04/02/silverlight-drag-and-drop-with-rx

as realized by YinYang, a Python-like managed time language, looks very much like polling code:

```
object sld is Slider
object lab is Label
lab.content = "hello " ++ sld.value
lab.position = (15 * sld.value, 30)
```

Code in YinYang executes *continuously* so label lab's content and position fields are updated whenever slider sld's value field changes via user manipulation. The content, position, and value fields are not data-flow event streams, rather they are mutable fields being assigned and re-assigned as needed. Internally, YinYang statements are incrementally *replayed* whenever what they read changes, gaining efficiency over a naive polling-based interpretation. Compared to the immediate-mode UI approach, object identities are memoized across replays so that object state is encapsulated; e.g. the Slider class provides a value field.

Unlike typical imperative code, continuous execution is the default in YinYang—event handlers must be used to realize any kind of discrete execution; consider:

```
cell x = 0, y = 0
print(x + y)
on buttonA.pushed: x += 1
on buttonB.pushed: y += 1
```

Event handlers in YinYang are expressed as the bodies of if-like on statements guarded by the event being handled. In this code, x and y cells are discretely incremented when buttonA or buttonB are pushed. Unlike the event handling approach, however, outside code sees these changes automatically, updating the print statement's execution.

Behaviors that continue after an event occurs must also be considered; e.g. the mouse drag behavior mentioned previously. We handles this with an after statement; consider:

```
def drag(widget):
  on widget.mouseDown:
    var pw = widget.position
    var pm = widget.mousePosition
    after:
      widget.mouseCapture()
      widget.position = pw + (widget.mousePosition - pm)
      on widget.mouseUp:
        widget.position = widget.position # hold widget position
        break                             # stop the most inner after block
```

When the mouse goes down on widget, the initial widget and mouse positions are remembered as pw and pm to be used in an after block that (a) captures the mouse and (b) assigns the position of the widget to the dragging position. When the mouse goes up, the after block is stopped via the break statement, which stops all behavior of the block like the capture and the assignment; to prevent the widget from reverting to its pre-drag position, it must be assigned discretely to its own value when the break statement is executed. The structure of this example looks very similar to the one previously presented for Rx. However, control flow for the YinYang version is not encapsulated; e.g. the mouse is captured as long as the after block's execution is unbroken.

Unlike FRP, YinYang fully supports side effecting operations on object references (aliases) as long as these operations are both *undoable*, so they can be rolled back when no longer performed on a replay, and *commutative*, so that statements can be replayed in any order. Beyond cell-like fields, state can also be collections whose memberships are determined by normal operations; consider:

```
set monsters
trait Monster:
  cell life = 100
  if this.life > 0:
    monsters.insert(self)
  ...
  on hit: life -= 1
object m0 is Monster
object m1 is Monster
```

This code defines a Monster class-like trait and uses this trait to define two monsters (m0 and m1). As long as a monster's life cell (mutable field) greater than zero, the monster is inserted into the monsters set, which is then undone **whenever** that condition does not hold; e.g. as soon as the monster has accumulated as much damage as it has life. Aliases to elements of a set are accessible via iteration; consider:

```
if doHighlight:
  for m in monsters:
    m.highlight()
```

which highlights monsters in the monsters set whenever doHighlight is true. Such code can be diffused through out the program—monsters be iterated on as a reference, even one obtained from a field. In a pure data-flow system, such interactions must be explicitly bound in a central location.

Managed time has a cost: the overhead of dependency tracing, replay, and effect logging (needed to support rollback) is expensive. We are currently exploring ways to reduce this cost. Additionally, managed time code is not as expressive as standard imperative code: so that field (cell) updates are commutativity, fields cannot be re-assigned outside of a discrete event context. It is then impossible to express algorithms where reassignment is common, e.g. bubble sort, which in any case is a limitation of pure functional code.

We have prototyped YinYang with a programming editor that leverages managed as a library in C#. Beyond implementing the semantics shown here, this environment supports live programming along with time travel, which become more feasible when the system is managing time for the programmer. Please see [6] for an animated discussion.

Conclusion

Given the problems with event handling, we are quickly rushing to data-flow abstractions that albeit am improvement, have many expressiveness and usability challenges. We should also consider re-inventing control flow so that it is more suited to reactive programming; we have found managing time as a very fruitful way to do this.

References

[1] A. Courtney, H. Nilsson, and J. Peterson. The yampa arcade. In *Haskell*, pages 7–18, 2003.

[2] C. Elliott and P. Hudak. Functional reactive animation. In *Proc. of ICFP*, pages 263–273, 1997.

[3] M. A. Hammer, K. Y. Phang, M. Hicks, and J. S. Foster. Adapton: Composable, demand-driven incremental computation. In *Proc. of PLDI*, pages 156–166, 2014.

[4] J. Komppa. Sol on immediate mode guis (IMGUI). http://iki.fi/sol/imgui/, 2005.

[5] S. McDirmid and J. Edwards. Programming with managed time. In *Proc. of SPLASH Onward!*, Oct. 2014.

[6] S. McDirmid and J. Edwards. Programming with managed time (essay with videos). http://bit.ly/1oeuWtB, Sept. 2014.

[7] E. Meijer. Your mouse is your database. *ACM Queue*, 10(3), 2012.

[8] C. Reas and B. Fry. Processing: programming for the media arts. *AI Soc.*, 20(4):526–538, 2006.