How my visualization tools use little memory: A tale of incrementalization and laziness

Eugene Kirpichov <ekirpichov@gmail.com>

St.-Petersburg Functional Programming User Group

Nov 7, 2012

Outline

Introduction

`http://jkff.info/software/timeplotters`

Tools for visualizing program behavior from logs, optimized for one-liners.

- **timeplot** — quantitative graphs about multiple event streams.
  - Histograms of event durations, of types of events, regular line/dot plots etc.
- **splot** — Gantt-like chart about a large number of concurrent processes.
  - Birds-eye view of thread/process/machine interaction patterns

Live demo

Live demo

Why optimize

Because they were slow and a memory hog (boxing, thunks).

- Took minutes for 100,000's of events
- Took hours or crashed for 1,000,000's of events
  - Crashing was the last straw. I couldn't do my job.

How to optimize

Rewrite in C or Java? We can do better!

- It would be a defeat :)
- Would rewrite and re-debug everything
- Would learn nothing

Turned out worth it.

splot

The easy one.

Before

1. Read input into a list
2. Traverse to calibrate axes
3. Traverse to render

After

1. Read input into a list
2. Traverse to calibrate axes
3. Read input into a list
4. Traverse to render

Lazy IO does the rest.

Cost: no output to window, no input from stdin.
Code tour.

tplot: why it can NOT be done

The hard one.

- Complex data flow: events:tracks = M:M.
- Uses Chart, which keeps data in memory and for good reasons.

Code tour: old version.

tplot: why it CAN be done

Reading too much $\neq$ drawing too much $\Rightarrow$ Chart is not a problem.

Building "data to render" for Chart in 1 pass seemed *possible*.

Code tour: PlotData, Render.hs

tplot: main idea

Push-based:

- List representation unchanged and hidden
- Push item
- Get result (at any moment)

Code tour: StreamSummary.
Live example: average + profiling.
Applicative average.

Types of stream operations

Concept	Logical type	Actual type
Summarize	`[a]` $\to$ `b`	Summary a b
Transform	`[a]` $\to$ `[b]`	Summary b r $\to$ Summary a r
Generate	`a` $\to$ `[b]`	Summary b r $\to$ (a $\to$ r)

Composition pipelines become quite funny.

Code tour: RLE etc.

tplot: new architecture

New architecture:

- Push-based builders for all plot types
- Driver loop to feed input events to output tracks
- When done, summarize and render

Code tour: driver loop, plots (vs old code).

tplot: the transition

I wanted to keep things working all the time.

1. Change interface — make the change possible.
2. Change implementation — make the change real.

tplot: the transition

Before: Completely non-incremental.

- *(interface)* Separate building and rendering
- Split into modules
- *(interface)* Explicit 2 passes, both *potentially* incremental
- Toying with incremental combinators to get a feeling for them
- *(interface)* Make all plots have an incremental interface
  (but actually use `toList` bridge)
- *(implementation)* Incrementalize plots one by one

After: Completely incremental.

Not so easy

Memory leaks due to insufficient strictness:
"push isn't pushing hard enough."

Question

What and why remains unevaluated until too late?

Profilers didn't help at all. **Debug.Trace** is insufficient: it outputs lines and I need *hierarchy*, not sequence.

Enter Debug.HTrace

`cabal install htrace`

“Code” “tour”. Live demo: average.

Why not X?

Iteratee, conduit, pipes, ...

- Scary, so I tried to get as far as I can without them
- Ended up very small, simple and nice, so I didn't look back
- Also educational

Lessons learnt

- Lazy IO is ok for this task
- Debugging laziness is hard: I wouldn't start a high-risk real-time project in Haskell
- Profilers are almost useless for debugging laziness
- Debug.Trace is better, Debug.HTrace much better
- Push-based incremental processing is fun and easy