Supplementary guide to `adapton.ocaml`

Introduction

Nominal Adapton is a general purpose incremental computation engine. Users are able to write somewhat standard code (plus the inclusion of names along with data), and the system updates computation results when data is forced after inputs have been modified.

There are two ways to use this repository. One is as a library, and the other is to run internal code the way our tests run. References to modules are slightly different in each case. The library is set up for opam 1.2 with the single command `make opam-pin` (or `make opam-reload` after changes). The internal tests are set up with the single command `make test`.

See the readme file in the main directory for some examples of library use. There are currently no major examples in the repository.

We have been developing a number of list-based example programs within an extension to Adapton located in the `Source/adapton_structures` directory. This is a currently incomplete list library, where having names as first-class values within data can be very beneficial.

There are a number of internal examples set up for testing purposes. They can all be run with `experiments.native`, which has many parameters available. Calling this executable with `--help` will give a simple list of parameters, and calling it with `--experiment [anyletter]` will fail but generate a list of available experiments to run.

The scripts folder contains a number of example uses of our test program. Our workflow has been to develop code within the Adapton library and call it internally from the experiments program. We call experiments from a simple bash script for ease of use and record of trials. Results are saved as CSV files, which can be loaded into excel spreadsheets in the templates folder for generating summary information and visualization.

List Experiments

Each experiment begins by generating an input list. Then some calculation is done to get an output list. Next the input list is modified in some way, at a number of points along the list. This modification is then reversed. After each step, a sequence is demanded from the output list. We measure the time it takes to generate the output list, and the time is takes to demand data after interactions.

An explanation of the parameters to experiments.native follows:

- `--sample-num [n]` (default 0)
    - a seed value for the creation of an input list
- `--n [n]` (default 5000)
    - the number of elements in the input list
- `--num-changes [n]` (default 10)
    - the number of interactions (and reversals) with the list
- `--demand [%]` (default 100.0)
    - the percent of the output list to calculate
- `--gran [n]` (default 0)
    - the average elements per name is 2^n. At 0 all elements are named.
- `--fullinit` (default unset)
    - the initial generation time for the output list is measured at 100% demand, regardless of the value of --demand
- `--outfile [filename]` (default out/experiments.csv)

    - the file to append result data to, created if necessary
- `--experiment [name]` (no default)
    - an experiment to run, required for results
- `--0` (default r,rr,di,id,ss)
    - reset the requested interactions with the input list to none
- `--r`
    - adds 'r-replace' interaction: replace one element with another, no reversal
- `--rr`
    - adds 'rr-replace1' and 'rr-replace2': replace one element of the list with another; reverse this procedure to restore the original
- `--di`
    - adds 'di-delete' and 'di-insert': delete an item of the list; reinsert
- `--id`
    - adds 'id-insert' and 'id-delete': insert an item into the list; delete it
- `--ss`
    - adds 'ss-swap1' and 'ss-swap2': swap two adjacent items; restore
- `--mv`
    - adds 'mv-move1' and 'mv-move2': divide the list at the interaction point and move the first half to the end of the second half; restore original order
- `--perf-flags` (default set)
    - performs no additional logging, for performance
- `--test-flags` (default unset)
    - checks the output list against a trusted list algorithm, outputs to screen
- `--debug-flags` (default unset)
    - outputs list elements to screen at initial creation and after each change
- `--help`
    - outputs simplified version of the above to the screen

An explanation of the experiments available to run follows:

Most experiment titles are divided up into three sections delimited by an underscore. The final word is the articulation library used. This is some version of Adapton, or an alternative to it. Basically, it provides the methods to deal with names.

'name' is the main Adapton version being presented, using programmer supplied names to distinguish updateable sections of data. 'arg' and 'arggen' are versions that use a hash of function parameters. 'arggen' is the version used in Classic Adapton. 'sac' is an attempt to replicate the methods of self-adjusting computation. It is not fully implemented at this time and may not provide correct results for the more complicated tests.

There are two non-incremental alternatives to Adapton: 'eagernoninc' does all calculations at the time of definition. 'lazynoninc' does all calculations at the first request for results. Neither of these will return the correct result after an incremental change. They are used for performance testing only.

'lazyrecalc' is the version we've been using for most of our comparison testing. It calculates results when forced, but does not save them. It returns correct results after an incremental change, without the overhead of Adapton techniques. As an

alternative to Adapton, its limitation is that it cannot select data affected by a change, so it must rerun all calculations requested.

The first word in the experiment title currently groups experiments by data structure used. All experiments take an input list, but most of the time we find it beneficial to convert the list into some type of tree structure to get the most benefit from Adapton. At the time of this guide, we are using only the original list structure, or a rope structure, which is a tree with data in the leaves only.

Here are explanations for the various computations available for experiments:

- filter, map - common list computations. We filter even numbers and perform trivial math. The eager version updates all data when forced after a change. The non-eager version only updates forced items.
- map_paired - breaks a list into even and odd elements and them sums pairs to produce an output list. Currently the trusted computation (for --test-flags) requires lists with an even num of elements, so id and di interactions fail.
- unique - a mapping to '1' the first time an element is encountered, '0' each additional time. This makes use of a trie structure internally as a set.
- quickhull - interprets the input list as a sequence of 2d points, and returns a list of elements corresponding to the outermost points of the group.
- max_dist - similar to quickhull, but only returns a single element
- reverse - reverses a list. The List version generates a computation graph structured like a rope, the Rope version converts the list to a rope and runs a standard algorithm (producing the same computation structure)
- iter - converts list to rope and then back to list
- min, sum - common folding operations.
- center - returns the element at the halfway point of the list
- mergesort - Currently the experiment with most attention. Sorting can be incrementalized more efficiently than with Adapton's general purpose strategy, but the complexity and computation structure of mergesort is the current target for Adapton. List division is done first by conversion to a rope, then the leaves are merged together until reaching the root node
- median - composition of mergesort and center. This is our main test for maintaining uniformity of name placement when working with fewer names than list elements. It's a open area of our research that hasn't been in a paper yet.

Experiment Results

By default, results are appended to CSV files in the `out/` directory. They have a number of columns. Some of these have names that are in need of update. Here's what they mean:

- Unix Time - Timestamp when line was appended to file
- Seed - Sample (from `--sample-num`)
- Version - The name of the experiment
- Test - The interaction
- Size - Initial input list length
- Mod Pos, Mod Pos % - location in the list of the interaction; % of list size
- Demand, Demand % - The amount of the output list requested; % of list size
- Time - Seconds required to return the demanded output list
- Unit Cost - Optional statistic, often measuring computation count
- Heap - A snapshot of heap memory used. Ballpark number for comparisons
- Stack - A snapshot of stack memory used. Ballpark number for comparisons
- dirty,dirty % - count of adapton nodes flagged for recomputation
- clean,clean % - count of adapton nodes flagged for recomputation but unnecessary
- evaluate,evaluate % - count of adapton nodes recomputed
- create,create % - count of new adapton nodes
- tables - count of active memo tables
- granularity - From (`--gran`)

The CSV results data is summarized into three data points for each test: the from-scratch run time and the relative speeds of the Adapton and Nominal Adapton second runs compared to the from-scratch run time. These values are reported in the top-level table{1,2}-results.txt files (if you are using the OOPSLA artifact).

Programmer's Guide for *Incremental Computation with Names* (ICwN)

The **ICwN** paper describes incremental programming in **Nominal Adapton** in a somewhat idealized manner. This document connects that discussion to the OCaml code of our implementation, which is publicly available on github (`https://github.com/plum-umd/adapton.ocaml/`).

Noteworthy modules:

The paths below are relative to the `~/adapton.ocaml/Source` directory in the artifact:

- The ArtLib module (adapton_core/GrifolaType.ml) gives the OCaml programming API for Nominal Adapton.
- The SpreadTree module (adapton_structures/spreadTree.ml) uses the API above to define basic tree and list data structures.

(Ropes versus trees: In the implementation, we refer to binary trees as **ropes** when all their elements are restricted to leaf positions. Other binary trees that store elements at internal nodes are simply **trees**. This distinction is absent in **ICwN**.)

- The Quickhull module (test/quickhull.ml) uses tree and list definitions above to define the quickhull algorithm for convex hulls.
- The Experiments module (test/experiments.ml) defines experiments using the modules above.

ICwN Sec 2: Overview

The `list` and `map` definitions in this section of the paper correspond to the implementation's `list` type and `list_eager_map` function. They are defined in the `SpreadTree` module.

ICwN Sec 3.1: Mapping

The `SpreadTree` module defines two variants of mapping for lists:

- The `list_map` function lazily maps a list.
- The `list_eager_map` function eagerly maps a list.

ICwN Sec 3.2: Folding

- The `SpreadTree` module defines the `rope_reduce` function, which reduces a rope using an associative binary operator. This is similar to a conventional list fold, except that the operator is applied to a rope.

ICwN Sec 3.3: Unfolding

- The `SpreadTree` module defines unfolding for rope structures. The `rope_of_list` function creates a rope from a list. An in-order traversal of the rope gives the original list elements, in their original order.