Funk2: A Frame-based Programming Language with Causally Reflective Capabilities

(2009/06/18 draft in progress)

Bo Morgan

Media Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139
bo@mit.edu

Abstract

We introduce a novel programming language called Funk2 that implements a convenient form of event tracing for process execution. Funk2 is a pure C project that implements a peer-to-peer memory system for execution large numbers of abstract threads that can be intricately monitored and controlled. Funk2 has been built as a programming language for cognitive scientists to easily work together in simulating complicated symbolic theories of mind. Much of cognitive science has recently been working with simple (ala Occam's Razor) mathematical models of mind. We are bucking this trend by writing a powerful programming language as a symblic alternative to the simple linear algebraic models that packages such as Matlab are optimized to simulate. This paper outlines two of the basic features of the Funk2 language: (1) "rewindable" memory for implementing reflective models, and (2) "imaginary" memory for experimenting with contrapositive models of planning.

1 The Reflective Model Language and Simulator

The Funk2 project includes a lisp-like programming language that has a few helpful features that most programming languages don't have. Many of these helpful features are based around a central idea that we call "causal tracing". Causal tracing is a simple idea that is not implemented in any other programming language as far as we know. The basic idea is that a memory object is passed from the causal source of a statement's evaluation through every function that is executed by that statement. We call this memory object a "cause" because it is created and used by the evaluator of the original statement. The cause object is used to implement many different features within the Funk2 programming language.

2 Why a new Programming Language?

2.1 Causally Traced Execution and Memory

Causally traced execution and memory are required for our vision of reflective perception of concurrent processes. We will discuss the causal tracing tools that we have built into the Funk2 programming language briefly in this document as well as describe how we plan to use these tools in building stages of learning to read children's stories through critical reflection. These causal tracing features are the primary reason for writing the Funk2 programming language. The causal tracing features of the Funk2 programming language are not available in any other large-scale and efficient programming language to our knowledge. That said, there are other reasons that have compelled us to write the Funk2 programming langauge, which follow.

2.2 Simple, Fast, and Free Software for the Cognitive Sciences

Free information flow in the medical scientific research community of modelling human minds is important to encourage academic research. Currently there are very few programming languages that are made to learn to write computer programs. We see the fact that Lisp has a very simple and consistent syntax as well as a run-time compiler as necessary features of any programming language with this goal. Currently, there are no free versions of Lisp that run on all operating systems and that are efficient enough to handle our task. Current free languages that run on all operating system that also contain run-time compilers, such as Python

and Ruby, are not fast enough for our task. The Funk2 language has a simple syntax, is written in pure C under the very liberal MIT license, and is optimized for learning to plan over a massive peer-to-peer distributed memory.

2.3 Peer-to-Peer Distributed Memory Layer

Our peer-to-peer distributed memory layer has been implemented for a few key reasons:

1. The simulation of very large and complicated causally traced reflective models should be able to take advantage of multiple personal computers concurrently.
2. In order to develop a cooperative environment where we can combine the expertise of all of the cognitive sciences, we need a modelling language that not only is easy to use but also allows many people to work together. So, we see our peer-to-peer memory layer as a future experiment for applying causal tracing of responsibility for human failure in the development of very large and complicated mental models of humans.
3. Peer-to-peer means no centralized, beurocratic, or hierarchical control of the project.

3 Cognitive Reflective Modelling Features

4 Basic Causal Event Tracing

The most basic of these features is the "causal event tracing" or simply the "causal tracing" feature. Following are two examples of function definitions, `a-function` and `b-function`, that report events that can be causally traced in Funk2:

```
[defunk a-function [x]
  [event 'a-function-event-type [list 'called-with-arg x]]]

[defunk b-function [y]
  [event 'b-function-event-type [list 'called-with-arg y]]
  [a-function [+ y 1]]
  [a-function [+ y 2]]
  [event 'b-function-event-type 'successfully-completed]]
```

Here, `defunk` defines a global function. So far, these event reporting functions are very similar to normal message passing architectures that simply report messages to other named processes (such as in the Erlang model); however, because our processes are causally traced, we have the ability to monitor events that only our process has *caused* to happen. We control event reception through causal tracing rather than named processes. We think of this change in the event reception control in Funk2 as similar to switching a surjective function to an injective function, thus emphasizing the collection of events from groups of processes as opposed to the distribution of events to individual processes.

Here is an example of how causal tracing can be used in a simple example, where we call the second function defined above, `b-function`, and causally collect all of the events of the type `a-function-event-type`:

```
in--> [trace 'a-function-event-type [b-function 10]]
out-> [[event a-function-event-type [called-with-arg 11]]
       [event a-function-event-type [called-with-arg 12]]]
```

Notice that the only events collected by the cause object in this case are those of the specific type `a-function-event-type`.

Here is an example of how causal tracing can be used in another simple example, where we collect all event types from executing the same function, `b-function`, with the same argument, `10`:

```
in--> [trace 'all-event-types [b-function 10]]
out-> [[event b-function-event-type [called-with-arg 10]]
       [event a-function-event-type [called-with-arg 11]]
       [event a-function-event-type [called-with-arg 12]]
       [event b-function-event-type successfully-completed]]]
```

Notice in this case that all events are collected from the hierarchical function calls. So far, these uses of causal tracing are not novel forms of process tracing, and one could see the following as an example implementation of this form of process tracing:

```
[globalize global-event-collection  nil]
[globalize global-event-type-filter nil]

[defmetro trace [event-type expression]
  '[prog [= global-event-collection nil]
         [= global-event-type-filter [quote ,event-type]]
         ,expression]]

[defunk event [event-type event-data]
  [if [eq event-type global-event-type-filter]
      [push [list event-type event-data] global-event-collection]]]
```

Here, `globalize` defines a global variable; `defmetro` defines a global macro; = sets the value of a variable; and `eq` compares two symbols for equality. Notice that this code is not thread safe and won't work for tracing concurrent programs.

For our next example, we will use the following function

that executes two causal tracing operations concurrently:

```
[defunk concurrent-tracer-function [z w]
  [parlet [[result-one [trace `a-function-event-type [b-function z]]]
           [result-two [trace `a-function-event-type [b-function w]]]]
    [list result-one result-two]]]
```

Here, `parlet` first creates two local variables, `result-one` and `result-two`. `parlet` next evaluates the two causal tracing expressions concurrently and stores their values in the newly created local variables. Lastly, `parlet` executes the `list` command with the results of the two concurrent causally traced expressions. Given this definition of `concurrent-tracer-function`, we would expect the following concurrent program to execute as follows:

```
in--> [concurrent-tracer-function 10 20]
out-> [[[event a-function-event-type [called-with-arg 11]]
        [event a-function-event-type [called-with-arg 12]]]
       [[event a-function-event-type [called-with-arg 21]]
        [event a-function-event-type [called-with-arg 22]]]]
```

If, on the other hand, we used the naive global queue implementation of process tracing for the same concurrent function call, our resulting list would be flat and out of order, similar to the following:

```
in--> [concurrent-tracer-function 10 20]
out-> [[event a-function-event-type [called-with-arg 11]]
       [event a-function-event-type [called-with-arg 21]]
       [event a-function-event-type [called-with-arg 12]]
       [event a-function-event-type [called-with-arg 22]]]
```

It should now be clear that the naive implementation of process tracing does not handle concurrently executing processes in the same highly organized manner that our causally traced event tracing does.

5 Causal Memory Event Tracing

The Funk2 memory system is fundamentally composed of arrays. Each of these arrays can either be allocated as a "traced" or "simple" array. Traced arrays have extra memory allocated for each slot in order to keep track of (1) the causal history of that slot and (2) the imaginative values of that slot.

5.1 Traced Memory

We have seen how causal tracing can be used to trace the current execution of a process. Traced memory allows for keeping track of historical mutations of array slots. By collecting linked lists of mutation events at the memory location of each array slot, we can easily "rewind" any traced memory objects to see their values at any historical point in time. For example, let us define two global variables, `sequence` and `times`, that we will mutate in order to keep track of a mutated list and the times of those mutations respectively:

```
[globalize sequence [list 0 0 0 0 0 0 0 0 0 0]]
[globalize times    [list 0 0 0 0 0 0 0 0 0 0]]

[dotimes [index 10]
  [elt-set times    index [microseconds-since-1970]]
  [elt-set sequence index index]]
```

Here, `dotimes` is a loop operator that loops ten times, mutating the elements of the `sequence` list and stores the times of those mutations in the `times` list. The values of these two lists at the current time are as follows:

```
in--> sequence
out-> [0 1 2 3 4 5 6 7 8 9]

in--> times
out-> [1398361219283811 1398361219561113 1398361219828195
       1398361220946740 1398361220361727 1398361220629388
       1398361220896560 1398361221163860 1398361221431032
       1398361221699425]
```

Now, in the case that these lists are created with traced memory objects, we can "rewind" the slot values of each memory object. We call the operation of rewinding memory objects and creating new memory objects that represent the historical states "remembering" the historical memory objects. The following is an example of remembering the `sequence` list at the time when the `dotimes` loop `index` variable was `5`:

```
in--> [remember sequence [elt times 5]]
out-> [0 1 2 3 4 0 0 0 0 0]

in--> sequence
out-> [0 1 2 3 4 5 6 7 8 9]
```

Notice here that the `sequence` list is not modified by the `remember` operation. In this section, we have focused on tracing memory mutations, but we have not discussed how causal tracing of memory mutations can help us in tracing the responsibility for these mutations. We expect to explore the tracing of the responsibility of memory creation and mutations in our reflective learning algorithms over the next year of the PhD.

5.2 Imaginative Memory

Traced arrays not only have the ability to keep track of a linked list of caused historical mutations but also the ability to keep track of frames of what we call "imaginative" memory values. For example, consider the following where we define a global variable called `value` to be the value `10` and then execute some tagged imaginary operations involving this variable:

```
in--> [globalize value 10]
out-> nil

in--> value
out-> 10

in--> [imagine 'a-imagination value]
out-> 10

in--> [imagine 'a-imagination [= value [+ value 1]]]
out-> nil

in--> value
out-> 10

in--> [imagine 'a-imagination value]
out-> 11
```

Notice that we can evaluate expressions that have mutating side effects without affecting the "real" memory system. We see our tagged imaginary memory as being a very helpful tool for building planners that perform inference in order to search for how to accomplish goals in Funk2. More advanced imaginative memory features involve recursive imaginations by using stacks of imagination tags in each cause object.

References

Acknowledgments

This work is supported in part by a fellowship from *Bank of America*.