



Functional Programming Language Constructs for Multicore

Kim P. Gostelow

Jet Propulsion Laboratory, California Institute of Technology

May 22, 2011



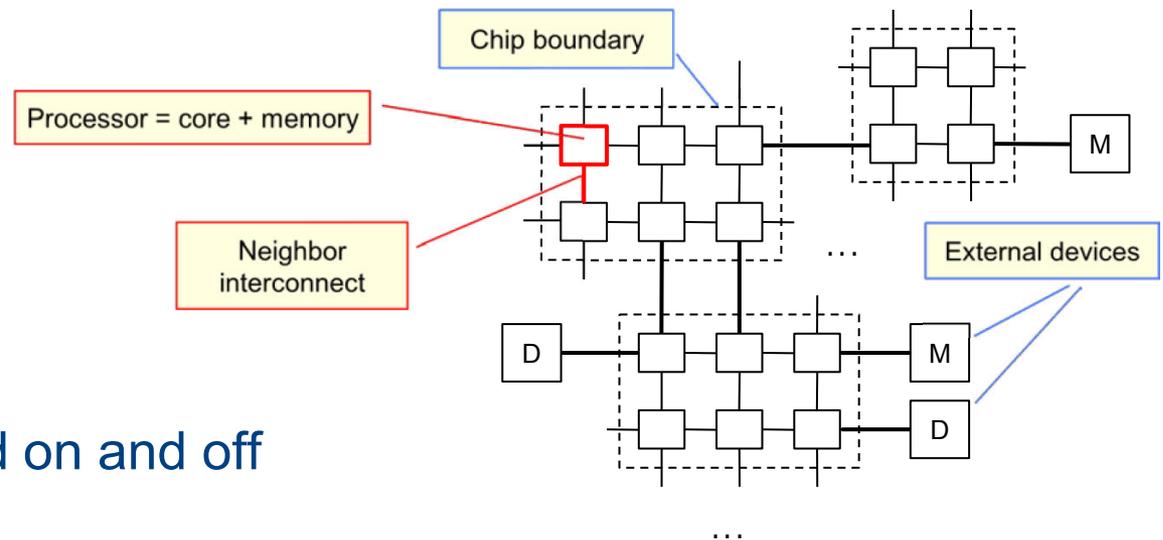
The Vision

- Look to when there are thousands of cores on a spacecraft
 - Expectation
 - Faulty core=> computations move to another core
 - Reduce power => performance slows, but does not quit
 - Policy-driven computation
 - Computations reorganize in real-time
 - Introspective
 - Little or no consideration needed by the programmer
- Programmers should not spend their time orchestrating intricate (and brittle) data arrangements and code
 - It breaks when processors fail
 - It should not be part of the job



Machine Architecture

- Large number of cores per chip
 - No shared memory visible to the programmer
 - Any shared memory is for internal purposes (e.g., message passing)
- Cores communicate with neighbors via high-speed, message-passing links



- Cores and links
 - May fail
 - May be powered on and off



What is a Functional Language?

The relation $f: A \rightarrow B$ is a **function** if:

For-all a in A there is a *unique* b in B such that $f(a) = b$

- Can define a value only once
 - Single-assignment
 - Not memory cells that can be modified/replaced/updated/...
 - Mathematical variables/values
- No side-effects



Functions (1)

The relation $f: A \rightarrow B$ is a **function** if:

For-all a in A there is a *unique* b in B such that $f(a) = b$

Program A:

```
extern int sum;

int A(int a) {
    sum = 0;
    for (int i=0; i<a; i++)
        sum += f(i);
    return sum;
}
```

Program B:

```
int B(int a) {
    int sum = 0;
    for (int i=0; i<a; i++)
        sum += f(i);
    return sum;
}
```



Functions (2)

The relation $f: A \rightarrow B$ is a **function** if:

For-all a in A there is a *unique* b in B such that $f(a) = b$

Program C:

```
int A(int a) {  
    int sum = get_clock();  
    for (int i=0; i<a; i++)  
        sum += f(i);  
    return sum;  
}
```

Program D:

```
int B(int a, int time) {  
    int sum = time;  
    for (int i=0; i<a; i++)  
        sum += f(i);  
    return sum;  
}
```



How Are Loops and I/O Functional?

- Loops

- A loop is a recurrence relation
- We make state explicit

Not yet ready for prime time

- A functional view of I/O has been a long-standing problem

- Monads in Haskell; “mutable” in Fortress; Unique type in Clean
- Different proposal coming here

But I'd like to discuss my approach with anyone interested in the topic

- Input: A file is an indexed array
 - Read => get next byte and move pointer forward
- Output: A file is an indexed array
 - Sometimes sequential dependencies, sometimes not



Example: A Simple Loop

The factorial sequence (1 1 2 6 24 120 ...) is defined by the *recurrence relation*

$$s_i = i \times s_{i-1} \text{ where } s_0 = 1.$$

```
int factorial(int n) {
    int s= 1;

    for (i=1; i<n+1; i++) {
        s = i * s;
    }
    return s;
}
```

A typical C program to compute the n^{th} factorial for $n = 0, 1, \dots$



As a State-Aware Functional Program

- Variable “s” represents a stream of state values $s[0]$, $s[1]$, ...
- The computer need keep only the **current** and the **next** $s[i]$ at any one time.

```
int factorial(int n) {  
    state int s;  
  
    return  
        initially  
            s = 1;  
        recur j = 1..n+1 {  
            next s = j * s;  
        }  
        finally last s;  
}
```

Keyword **state** defines s as a state.

The **initially** section defines $s[0]$ and the number of slots needed.

An implicit last step sets $s[i+1] = \text{next } s$

The **finally** section says what to return at loop end – that is, $s[n]$.



Example: A Slightly Bigger Loop

The Fibonacci sequence (0 1 1 2 3 5 8 13 ...) is defined by the *recurrence relation*

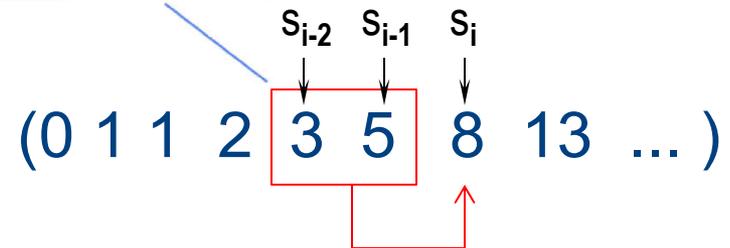
$$s_i = s_{i-2} + s_{i-1} \text{ where } s_0 = 0 \text{ and } s_1 = 1.$$

```
int fibonacci(int n) {
    int s0 = 0;
    int s1 = 1;
    int s2;

    if (n <= 0) return s0;
    if (n == 1) return s1;

    for (i=1; i<n; i++) {
        s2 = s0 + s1;
        s0 = s1;
        s1 = s2;
    }
    return s2;
}
```

The **state** in the loop.



A typical C program to compute the n^{th} Fibonacci



Fibonacci in State-Aware Notation

- Variable “s” represents a stream of state values $s[0]$, $s[1]$, ...
- The computer need keep only the most recent two ($s[i]$ and $s[i-1]$) values and the **next** $s[i]$ at any one time.

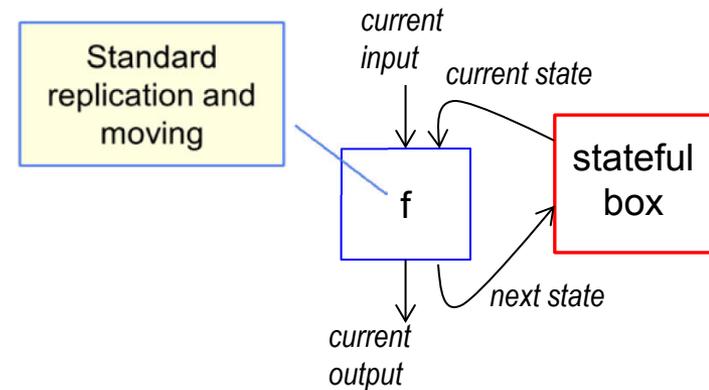
```
int fibonacci(int n) {  
    state int s;  
  
    return  
        initially s[0] = 0;  
                s[1] = 1;  
        recur (j = 2..n+1) {  
            s[i] = s[i-1] + s[i-2];  
        }  
        finally last s;  
}
```

The **initially** section defines the number of slots needed state.



Functions and State

- State is treated differently (not theoretically, but as a practical matter)
- State is explicit
 - Recognize “state” with a source language construct
 - When the program moves, the state moves with it
 - The state is often surprisingly small
- A function applied to a state value can be replicated, moved, restarted, ...





Simple Rules Work

- To write programs as functions
 - Often only small changes are needed
 - Move state changes outside the large part of the computation
 - Move calls to non-functions outside (Program C vs. D above)
 - Use the recur loop
 - Don't create state where it isn't necessary
 - Don't re-use variable names – each name has one meaning, one value
- I/O is coming



Why Restrict to Functions?

- Any two functions whose arguments are available can run concurrently
- Extra-functional properties guarantee certain behaviors without further analysis
 - Start/stop, re-execute, copy, ... at any time
- Pure functions are easier to
 - Test
 - Use in other contexts without surprises



Rule #1

- You can't change or re-assign a variable once its value has been determined.

```
int fibonacci(int n) {  
    stream state int s;  
  
    return  
        initially s[0] = 0;  
                s[1] = 1;  
        recur (i = 2..n) {  
            s[i] = s[i-1] + s[i-2];  
            s[i-1] = xxx;  
        }  
        finally s[n];  
}
```

Illegal



Rule #2

- No shared variables.

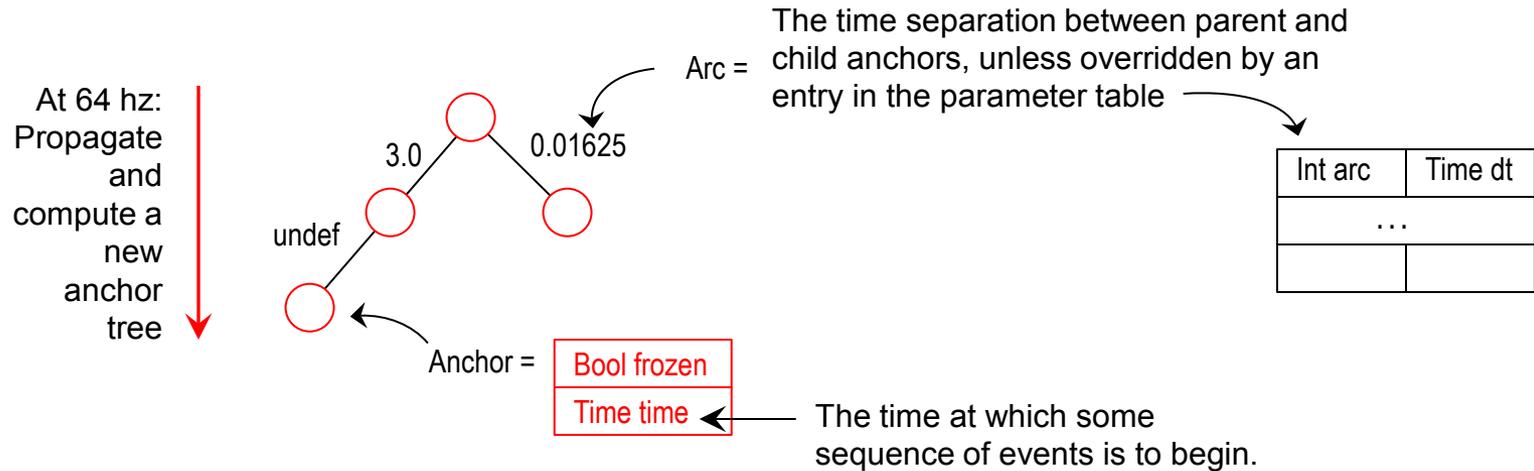
```
extern int val;  
  
int fibonacci(int n) {  
    stream state int s;  
  
    return  
        initially s[0] = 0;  
                s[1] = 1;  
        recur (i = 2..n) {  
            s[i] = s[i-1] + s[i-2] + val;  
        }  
        finally s[n];  
}
```

Illegal



A More Complicated Example (1)

- From the Mars Science Laboratory EDL (Entry, Descent, and Landing) timeline code
 - Tree of anchors
 - Anchor's time = parent anchor time + delta given by arc
 - Unless anchor's time is frozen, whereupon it can no longer change
 - When clock time passes anchor's time, the anchor freezes
 - An arc's delta can change at any time
- Time is seconds or undefined
 - Undefined value propagates





A More Complicated Example (2)

```
Anchor[] propagateAnchors(Anchor[] as, Arc[] arcs, ArcParm[] ps, Time now) {  
  initially anchors = as ← The state is an array of Anchor  
  for (arc, arcId) in enumerate(arcs) { ← Each iteration produces a new Anchor array  
    parent = arc.parent_anchor  
    child = arc.child_anchor  
    next anchors[child] = ← This parses as (next anchors)[child]  
      if anchors[child].frozen then anchors[child]  
      else {  
        let Time t = add(anchors[parent].t ,  
          if nonEmpty(parms = lookupArcParm(ps, arcId))  
            then parms[0].dt else arc.deltat);  
        in Anchor(now >= t, t) }  
      }  
  finally last anchors  
}
```



Consequences

- Thesis
 - No global state
 - State is distributed across programs in (small) pieces
 - Loops and other constructs state-explicit
 - Mini-checkpoints
 - No large, mass checkpoints are needed



References

- Ashcroft and Wadge “Structured Lucid” Univ. of Warwick, 1980
- Arvind, Gostelow, and Plouffe “Indeterminacy, Monitors, and Dataflow” Proc 6th ACM Symposium on Operating Systems Principles
- Fortress Programming Language
<http://projectfortress.java.net/>