Compositional hardware virtualization

Raphael 'kena' Poss
University of Amsterdam
January 11th, 2014

Context

- Programming is a human activity
- More and people need to program
- There are programming problems where languages can't help (well)
- Where I'm coming from: Understanding how people think to better teach them how to program

Practical problem: syntactic variance

- Observed: functional systems synthesized from *semantically equivalent* but *syntactically different* specs usually differ significantly in quality* – can we eliminate this difference by better automated tools?

- This was identified in 1994 as the "syntactic variance problem"
(D. Gajski ,Introduction to high-level synthesis. IEEE Des. Test Comput.)

Example

- `f :: [Int] -> [Int]`

- ```
  f = map sum . transpose . transpose .
      map (flip replicate 1)
  ```

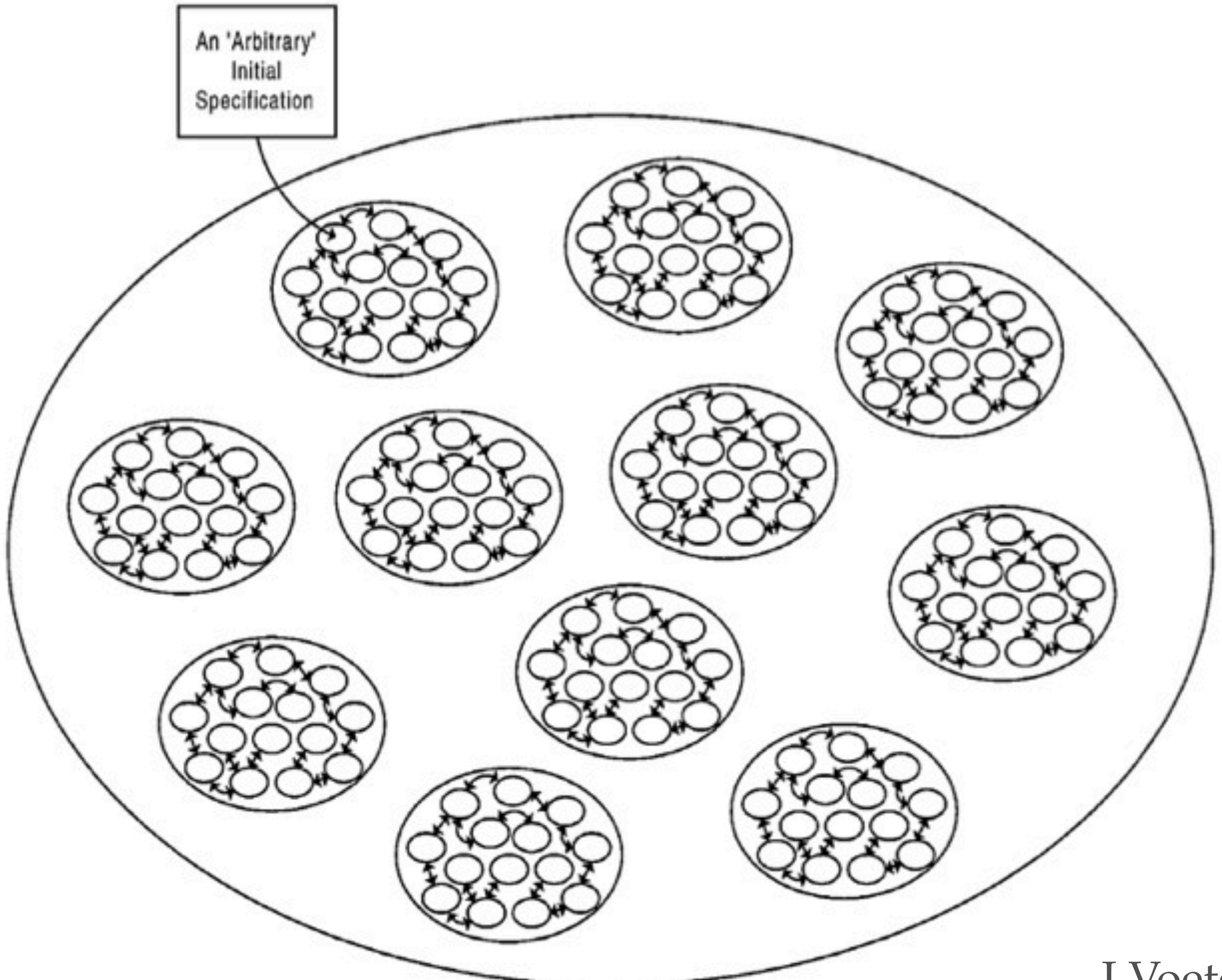
- ```
  f (p:xs) = f [ y | y <- xs, y < p ]
             ++ [p] ++
             f [ y | y <- xs, y >= p ]
  ```

- ```
  f (h:tl) = snd $ foldl g (h, []) tl
    where
     g (s, r) x | x < s = (x, s:r)
                | otherwise = (s, x:r)
  ```
  ```

NO CAN'T DO

- For any sufficiently expressive transformational system
  (e.g. set of source language(s), machine language(s) and all their *possible* compilers from one to another)
- For any initial specification
  (e.g. a program that encodes an algorithm)
- There exists some equivalent implementation specification that can never be reached by transformation
  (e.g. there exist some program that "does the same thing" but can never be produced by *any* compiler for the same input)
- (J Voeten, On the Fundamental Limitations of Transformational Design, ACM TDAES 2001)

Constellations



J Voeten (2001)

Fig. 1. Inescapable subspace of an infinite design space partitioning.

What this means in practice

- In any* language/compiler, there are equivalent programs whose extra-functional characteristics differs; but *people must choose* at some point
- How do functional programmers choose?

- Academic: legibility, clarity, simplicity, elegance
- paid programmer: also, but **extra-functional behavior** too

Extra-functional behavior (EFB)

- "Extra" = "not specifiable* in language"
- "Behavior" = "what happens at run-time"
- Examples:
    - Time to result: not specifiable because halting problem
    - Memory usage: because boring
    - Throughput/latency: because HW-dependent
    - Jitter: because user decides scenario
    - Battery life: because science not there yet
- Includes but not limited to "performance"

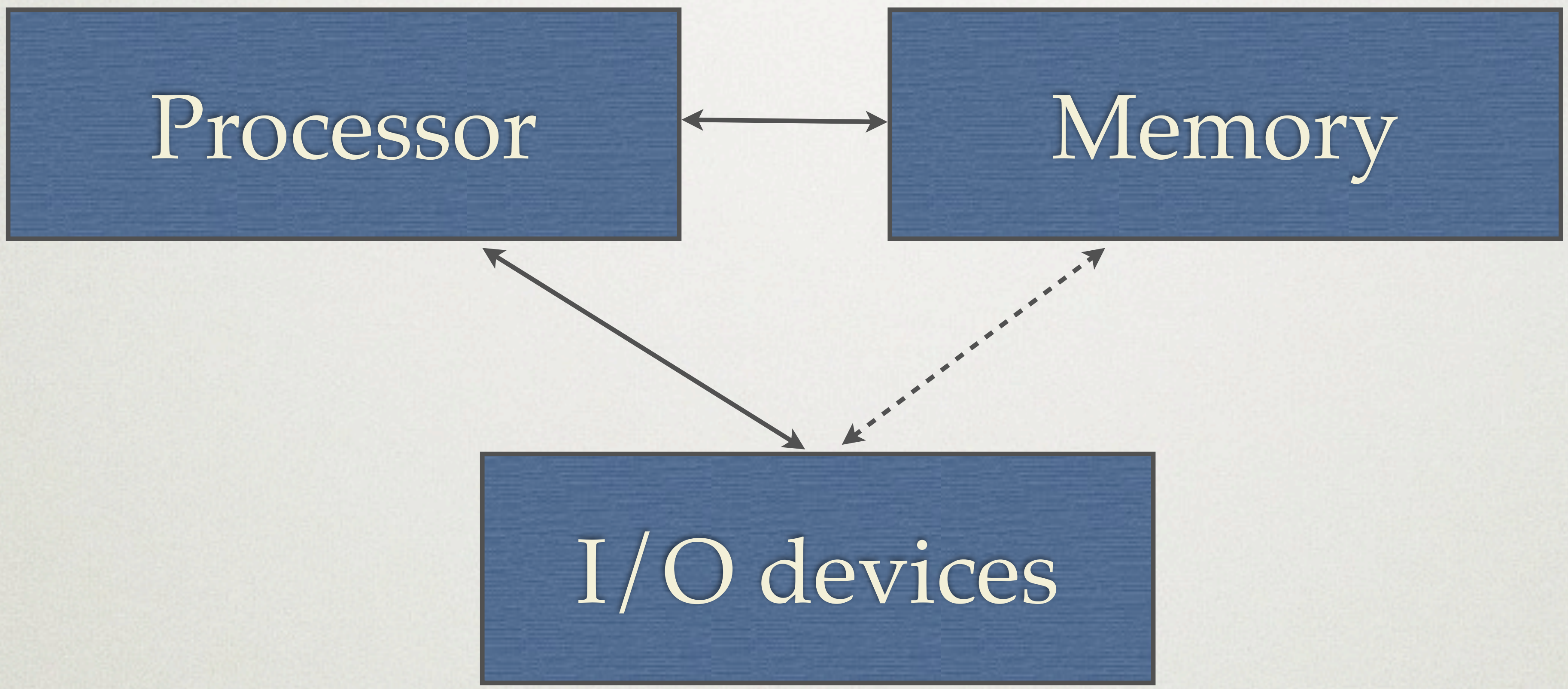
Abstract machine models (AMMs)

- Really the core topic of this talk
- Taught to newbies explicitly by some PLs
    - Usually not, though (e.g. Haskell)
        - Still, all programmers use them
- How do programmers build their own AMMs?
- Which “intuition-only” AMMs are most useful for “hard” programming tasks?
- How to capture them to later teach them?

Machines we know to build

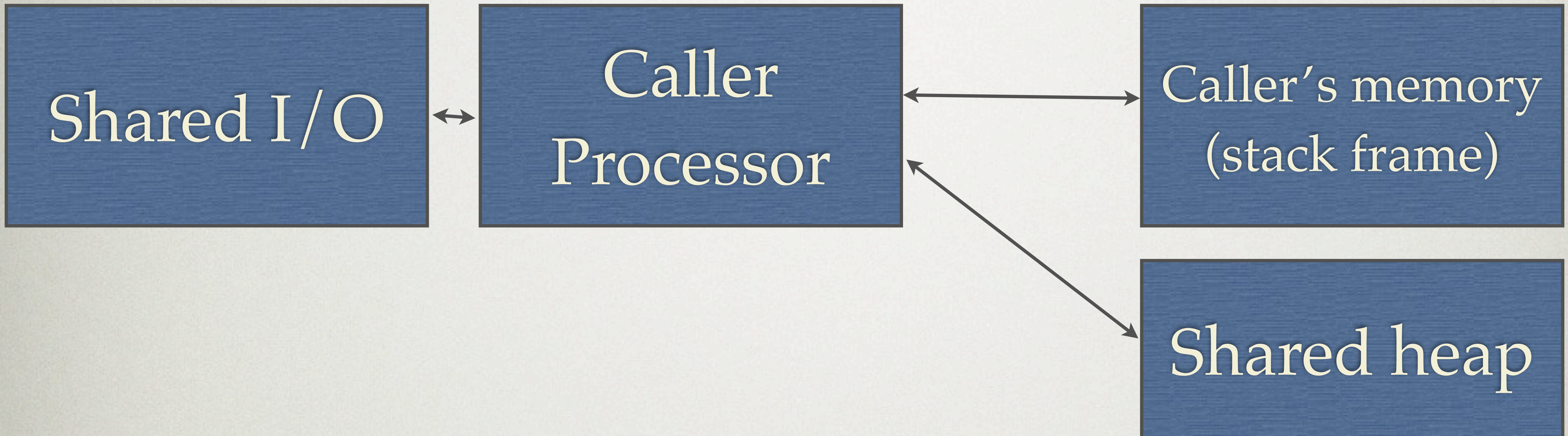
- The only hardware computers* are **register machines** and **queue machines** (and **networks** thereof)
- Everything else is simulated in software
  - *Including the stack and graph reduction machines of functional languages*
- But EFB emerges from hardware. Let’s look at what is preserved.

The essence of physical computers



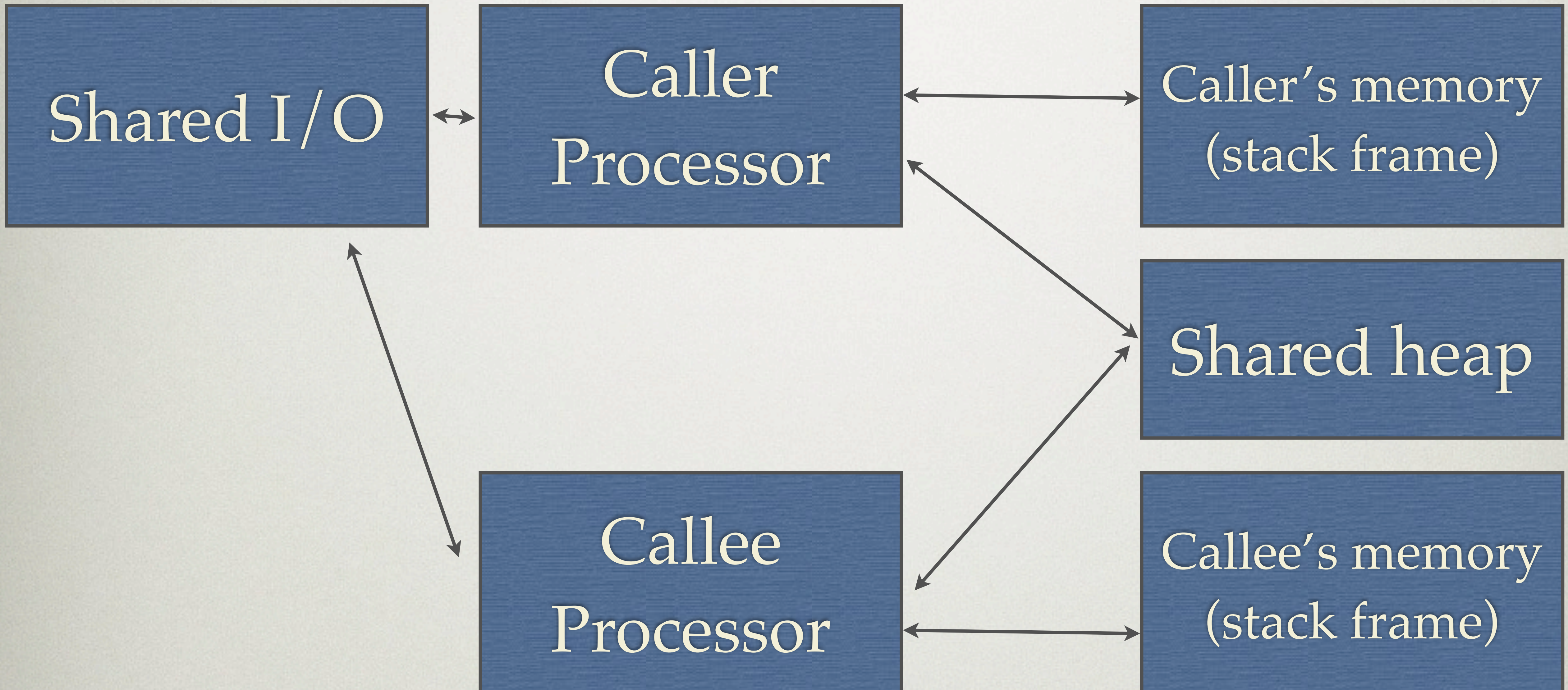
NB: This model gives productive EFB intuitions

A model of function calls



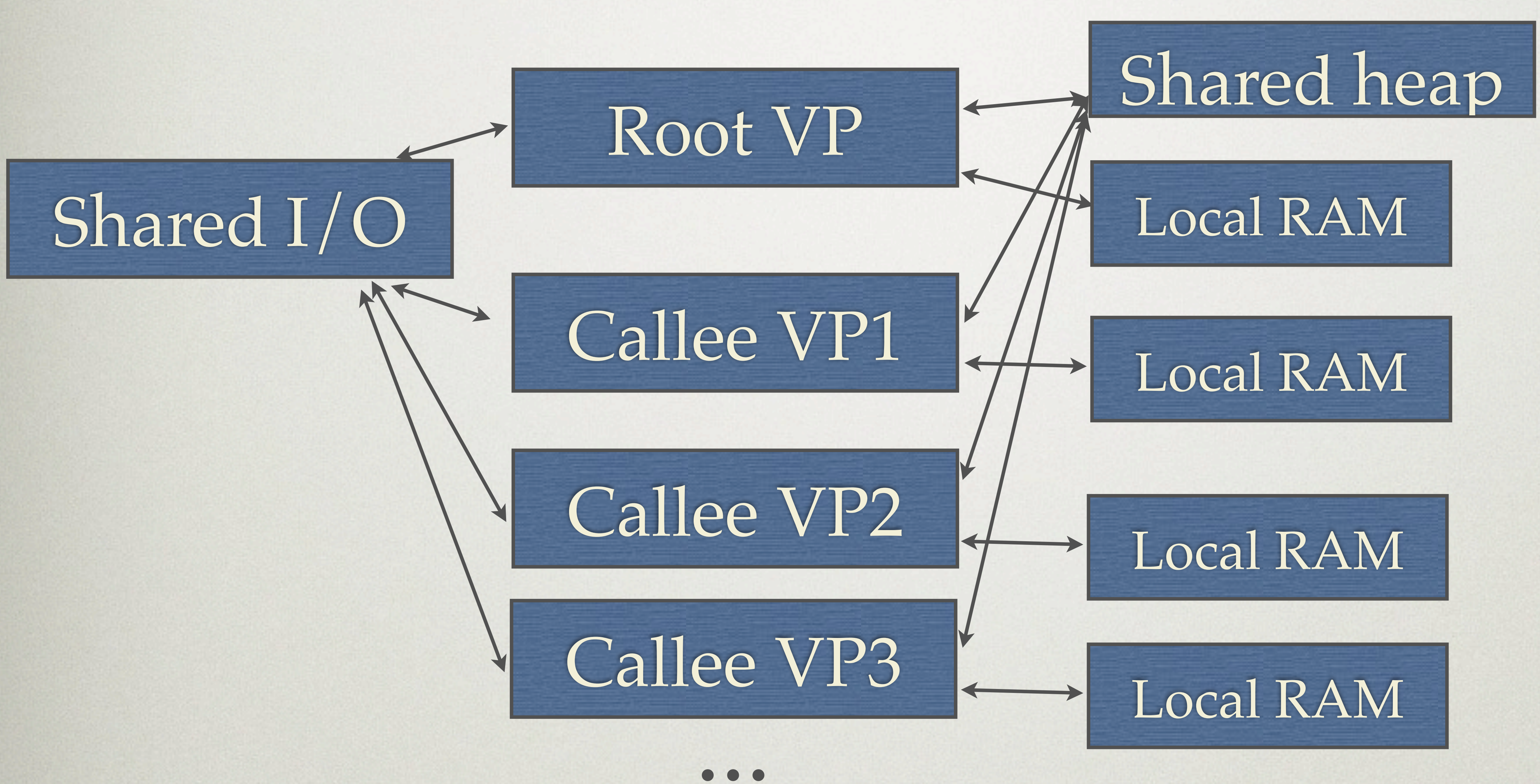
This AMM carries the EFB intuitions of the equivalent hardware machine

A model of function calls



This AMM carries the EFB intuitions of the equivalent hardware machine

A model of function calls



Recursion is modeled by compositional replication

What's in a "function call"?

- Push/jump-Pop/jump is a compositional mechanism that virtualizes the processor for each called procedure
- "function call" is an abstraction of this
  - Programmers don't think push/pop, but picture mentally a fresh *virtual context* at each call level
- Compositional virtualization carries through abstraction

Virtual hardware: a meta-functional model

- Component = VP | RAM | IO
- Operators = New | Dup | Del
  | Connect | Disconnect
  | Start | Pause | Reset
  | Wait until self-pause
- **EFB intuition = mental program in this model, not encoded in functional specs**
- **Graph structure makes the model compositional**

Other known examples

- Interrupts / async signals
  - Virtualization of powered-on-demand co-processors
- Process/thread creation
  - function call, but without stopping the caller VP
- System call interface to an OS kernel
  - Virtualization of a network link between a process' processor with own memory and an OS' processor with own memory

Predictive power

- Recursive function calls:
  (mental) stack of VPs, only one running at a time
- Recursive thread creation:
  same graph, multiple VPs running simultaneously
- Tail recursion: only one VP “alive” at a time
- Example predictable EFBs using this model:
  - space usage: additive with both threads & calls, constant with tail recursion
  - power: additive with threads, not calls & TR

Conclusion

- Programmers must mentally translate the functional evaluation strategy to an AMM to reason about EFB
  - My research: can we have both powerful abstractions and powerful AMMs?
- Open questions:
  - how many different AMMs are relevant/useful for a programmer population working on a given HW platform and programming task?
  - How reusable are teachable AMMs?
  - Should we invest in teaching AMMs or rather teaching how to build them?
- Comments, suggestions?