Don't forget the hardware!

Raphael 'kena' Poss
Universiteit van Amsterdam
7 dec. 2012

Two parts

- The syntactic variance problem
- Compositional hardware virtualization

Syntactic variance

An "annoying" question

- Since the 1970's: "pure functional languages are ideal to program parallel computers - immutability, referential transparency, etc."
- 40 years later, still can't seem to get it right
- What's wrong?

Another "annoying" question

- On your desktop machine:
  - A "good" sort is algorithm X
- On my research platform:
  - Algorithm X is difficult to parallelize
  - Algorithm Y parallelizes well, and gets large speedups compared to mostly-sequential X
  - Maybe there is yet another Z better than both, but *yet unknown*

Another "annoying" question

- Is there a language and compiler in which a sort specification Z can translate to either a good X, Y or Z depending on the target machine?
  - Haven't found one in 60 years
  - Why?

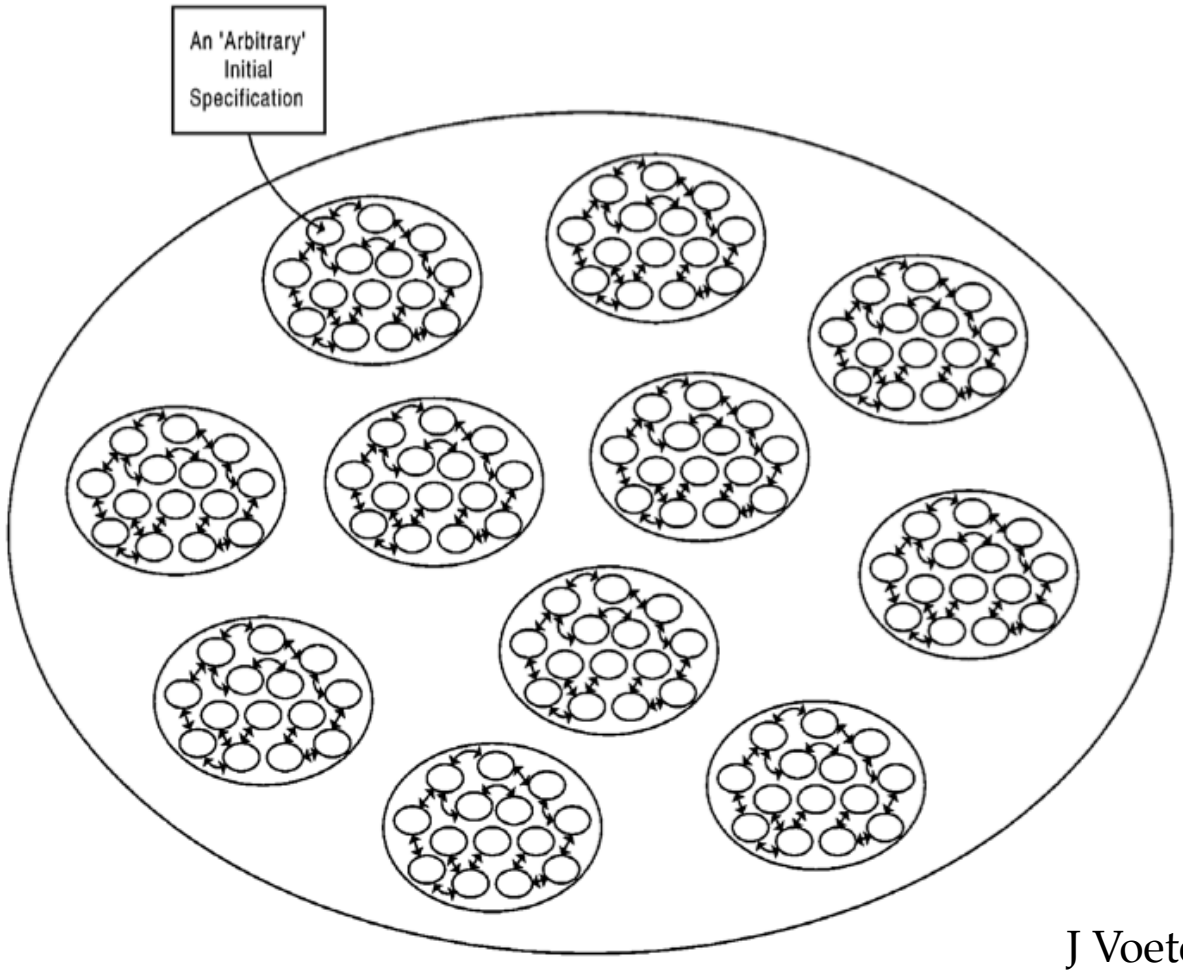
Background: syntactic variance

- Chip design is “hard”: define components, connect them, lay them out on the chip, route the wires
- This is why Verilog and VHDL exist
- Observed: Two designs synthesized from two semantically equivalent but syntactically different descriptions in VHDL 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.)

NO CAN'T DO

- For any sufficiently expressive transformational system
  (e.g. a language and all its *possible* compilers)
- 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 machine code that "does the same thing" but can never produced by *any* compiler for the same language)
- (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

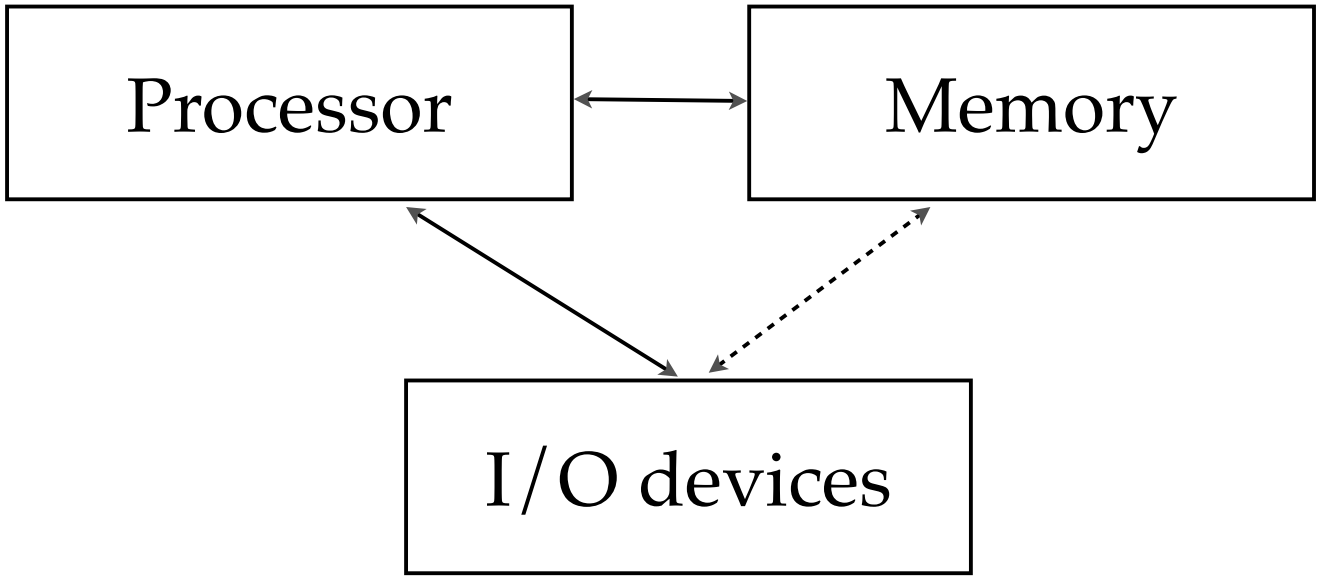
- In any *possible* language/compiler, there are equivalent programs whose implementation quality differ
  - Example: Haskell's 20+ different sorts
- How to choose a good one?
- Human, know thy implementation

Compositional hardware virtualization

Machines we know to build

- There are many Turing-equivalent models
- In 70 years, the only computing machines we know to build are **register machines** and **stack machines** (and **networks** thereof)
- Everything else is simulated
  - Including all the graph reduction machines of functional languages

The essence of physical computers



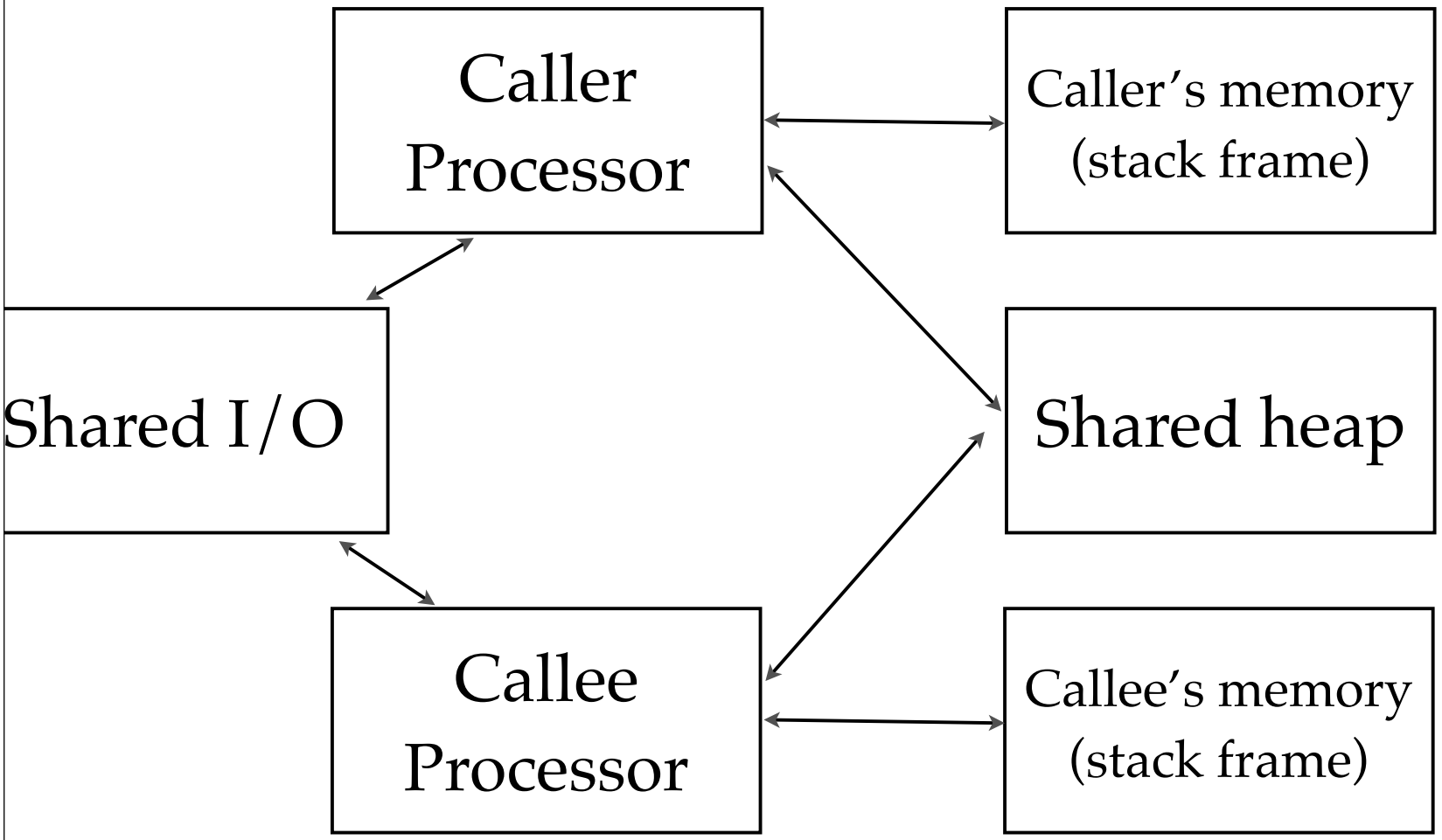
What's in a "function call"?

- To call:
  - Write PC to [SP--]
  - Write registers to [SP--]
  - Jump to new PC to call
- To return:
  - Read registers from [SP++]
  - Read old PC from [SP++]
  - Jump to old PC to return

What's in a "function call"?

- Procedure: reusable sequence of instructions
- The **author** of a procedure **assumes** that the procedure has complete control over the processor
- Push/jump-Pop/jump is a compositional mechanism that virtualizes the processor for each called procedure
- The concept of "function call" in C is a useful abstraction of this mechanism
- But C also has a virtual machine, and the author of a C function also assumes complete control of the VM during the function's execution
  - compositional virtualization carries through abstraction

A model of function calls



A model of function calls

- A “function call” is a virtualization mechanism:
  - That creates a new virtual processor
  - With its own stack memory
  - Connects it to the shared heap and I/O
  - Stops the caller virtual processor until the callee halts

The basic components of a virtualization mechanism

- VP creation / deletion
- Memory creation/deletion
- Connecting / disconnecting components
- Stop a VP, start a VP
- Synchronize: “wait until VP halts”
- All these are basic computer architecture “operators”

What else from there?

- A fruitful exercise: what virtualization mechanisms appear by using a different set of building operators?
- Some known mechanisms:
  - function call, but without stopping the caller processor = thread creation
  - new memory without creating a new processor = malloc

Other known examples

- Interrupts (signals in C)
  - Virtualization of powered-on-demand co-processors
- System calls interface to an OS kernel
  - Virtualization of a network link between a process' processor with own memory and an OS' processor with own memory

Recursive virtualization

- (Recursive definition: using the name of F in the definition body of F)
- Recursive call: using a recursive definition in a function-call syntax
- Recursive data structure: using a recursive definition in a data-use syntax
- Recursive thread creation: using a recursive definition in a thread-creation syntax

A possible research statement

- Conjecture:
  - The set of hardware virtualization mechanisms abstracted in a language is exactly the machine intuition used by programmers to write good programs for a given platform

Why this is useful

- May help tackle the problem of parallel programming: “What abstractions are useful?”
  - I’d rather ask:
    “What virtualization combinators can be designed that help programmers productively exploit parallel hardware?”
- By thinking hardware virtualization combinators, *instead of functional abstraction*, one can discover new, useful abstract mechanisms
  - For example, recursive placement of threads

This is where I got so far

- Comments, suggestions?
- Thank you!