CSci • B607 **Problem Set #2** **Jan 16, 2001**

 Last edited Friday, January 19, 2001

On Implementation

Due — Tuesday, January 30, 2001 (9:30 a.m.)

I. Instructions

A. For general instructions see Problem Set #1 (distributed January 11).

B. Question II.C.5 refers to Haugeland's characterisation of computation,[1] which is approximately the following (he is sometimes unclear as to whether interpretability is necessary to, or only convenient for, the notion of computation):

semantic engine	⇒	computer + interpretable
computer	⇒	automatic + formal
formal	⇒	self-contained + perfectly-definite + finitely-checkable

II. Assignment

A. It is standard computational practice to **implement** one language or virtual machine on top of another. In fact one often constructs whole implementation hierarchies: Intel IA-32 machine language ⇒ C++ ⇒ Scheme ⇒ a knowledge representation language ⇒ an expert system; etc. When one does this, it seems that properties that hold at one level may or may not hold at another above or below it (e.g., Lisp is recursive, whereas IA-32 machine language is not, even though you can implement Lisp in machine language).

B. We will say that a property ϕ:

1. **Crosses implementation boundaries upwards** just in case

   a. If it holds at level **k,** in a hierarchy of abstractions of a given system (i.e., if it is true of the system, according to a description of it at level **k**), it *must also hold at all levels* **j** > **k** —i.e., at all "higher" or "more abstract" levels of description of that same system.[2]

   b. Or to say the same thing in a different way: a property ϕ *crosses implementation bounda-*

---

[1] See Haugeland's "Semantic Engines" (available for copying in Lindley 210).

[2] By "must" is meant "necessarily", or *in all cases*.

Suppose we ask whether "recursiveness crosses implementation boundaries upwards". That means: given that one level (K) in the abstraction hierarchy (i.e., Lisp), is recursive, does it follow, if we implement another virtual machine M on top of Lisp, that M must be recursive?

Note that one *can* implement another recursive language on top of Lisp. If, therefore, we were to ask the same question without the (modal) term 'must'—i.e., "Given that Lisp is recursive, *is* virtual machine M implemented on top of Lisp recursive?—the best answer one could give is: "It depends on M; we don't have enough information. Could be; could not be." The interesting point is that it *need not* be. Since we can implement non-recursive machines in Lisp, it is not true that *all* machines implemented in Lisp are recursive. Hence—according to the definition—recursiveness does not cross implementation boundaries upwards.

ries upwards just in case, if it holds of some virtual machine **M**, it must also hold of all machines **M'** implemented on top of **M**.

2. **Crosses implementation boundaries downwards** just in case
   a. If it holds at level **k**, it *must also hold at all levels* **j** < **k** (i.e., at levels that are "lower" or "more concrete").
   b. Or to say the same thing in a different way: a property ϕ *crosses implementation boundaries downwards* just in case, if it holds of some virtual machine **M**, it must also hold of all machines **M'** that **M** is implemented on top of.

B. For each of the following properties, discuss whether it "crosses implementation boundaries" both upwards and downwards, in the senses defined above. If it does, say why, and give some examples. If not, say why not (as best you can), and describe a case where it does not. As you may have noticed, in the notes for last week's class, the answer will sometimes be more complex than a simple "yes" or "no". (References are to John Haugeland's "Semantic Engines.")

1. Serial/parallel
2. Discrete/continuous
3. Executing an "algorithm" vs. executing a "heuristic" (p. 17, lines 10–15)
4. Being universal, in Turing's sense
5. Being formal, in Haugeland's sense[3]
6. Being "real-time," in the sense of being able to play rhythms, respond to interrupts within "hard temporal constraints," etc.
7. Having semantics (in the sense we talked about in lecture 2a, of representing some task domain)
8. Being *correct*
9. Having *derivative* vs. having *original* intentionality (p. 32, last ¶)
10. Being computational.

—— *end of file* ——

[3]See general note I.B (above).