

# A PHILOSOPHY OF COMPUTER SCIENCE

# Philosophy of Computer Science

Is concerned with those philosophical issues that surround and underpin the academic discipline of computer science.

# What PCS is not

- It is not theoretical computer science.
- It is not identical to the philosophy of information.
- It is not the philosophy of artificial intelligence.
- It is different to the use of computational notions to address philosophical problems.

# Meta Discipline

- Philosophy of computer science is a meta discipline.
- It stands to computer science as the philosophy of mathematics does to mathematics and the philosophy of physics does to physics.
- But much less developed. It is not even clear what, if any, the central problems are.

# Other Areas of Philosophy

- One way to see what they might be is via analogies and overlaps with other areas of philosophy e.g.
  - I. Philosophy of Mathematics
  - II. Philosophy of Science
  - III. Philosophy of Language
  - IV. Philosophy of Technology
  - V. Philosophy of Mind
  - VI. Biology etc.

# Philosophy of Mathematics

- The nature of mathematical objects.
- The nature of *proof*: the use of theorem provers and proof checkers within mainstream mathematics (e.g. 4 colour problem). Do they give a priori knowledge?
- Similar issues with proofs of program/software correctness.

# Philosophy of Language

- Languages being a central preoccupation of computer science, its philosophy also has potential connections with the philosophy of language.
- Many of the semantic questions concerning ordinary vernacular would seem to have analogues in the languages of computer science.
- Model theoretic semantics and its influence on the denotational semantics of programming languages.
- Realism/antirealism in semantics has also a potential analogue in the semantics of programming and specification languages e.g. CTT.

# Philosophy of Logic

- Loads of issues
- Logics of programs
- Termination
- Correctness
- Partial and three valued logic
- Linear logic
- *Nature of Definition*

# Philosophy of Mind

- There are also obvious connections (via artificial intelligence) with the philosophy of mind.
- Turing's analysis of computation and his argument for *mechanism*. The reply of Wittgenstein.

# Philosophy of Science

- There are strong connections with the philosophy of science.
- Specifically in the claim made by some that *computer science* is a science.
- Is it a science in the way physics is or is it only a science in the same way that mathematics and logic are claimed to be – whatever that means?

# Philosophies of Engineering and Technology

- In contrast some would claim that mainstream CS is engineering: specify, design and construct.
- In the philosophy of technology work on the nature of artefacts contains significant overlapping themes.

# New Perspectives, New Problems

- There are clearly many overlapping issues but
  - I. Are there new perspectives on existing issues to be had?
  - II. Are there genuinely new philosophical issues?

# Objectives

- To provide an overall philosophical perspective on existing work – of which there is little that is marked as PCS.
- To set an agenda for future work.

# Central Conceptual Notions of CS

- *Program*
- *Algorithm*
- *Computation*
- *Abstraction*
- *Specification*
- *Type*
- *Emergence*
- *Rule*
- *Definition*
- *Theory*
- *Artefact*

# Outline

- I. [Definition](#)
- II. [Specification and Artefact](#)
- III. [Machines](#)
- IV. [Specification and Theory](#)
- V. [Definition of Programming Languages](#)
- VI. [The Specification of Abstract Artefacts](#)
- VII. [The Ontology of Computer Science](#)
- VIII. [Knowledge of Computational Artefacts](#)
- IX. [Rules](#)
- X. [Computability](#)
- XI. [Abstraction in Computer Science](#)
- XII. [Emergence...](#)

Forthcoming *A Philosophy of Computer Science* Springer in Theory and Applications of Computation Series

# I. DEFINITION

# Philosophical Concerns

- What are definitions?
- What is their main function in computer science?
- How does it differ from their role in mathematics?
- What kinds of definition are involved?

# Forms of Definition

- Real and Nominal
- Dictionary and Ostensive
- *Stipulative*
- Descriptive
- Explicative
- Implicit -*axiomatic*

# Stipulative Definitions

- They introduce new things e.g., relations, objects, functions, properties, operating over different *types*.
- They involve no commitment that the assigned meaning agrees with prior use –if any.
- They are not *correct* or *incorrect*.

# Stipulative Definitions in Mathematics

1.  $G(x) \cong 3x^3 + 2x^2 + 7x + 9$

2.  $P[x, y] \cong \exists z. y = x + z^2$

3.  $F(0) \cong 12$

$$F(n+1) \cong 3F(n) + 8$$

# Definitions in Computer Science

- There is a very special use of definitions in computer science.
- Here stipulative definitions come into their own.
- In programming and Specification.

# Programs (usually) are Stipulative Definitions

**Flib( $n$ )**

**if  $n = 0$  return 7 else**

**if  $n = 1$  return 6 else**

**if  $n = 2$  return 3 else**

**return Flib( $n - 1$ ) + Flib( $n - 2$ ) + Flib( $n - 3$ )**

# So are Specifications -Z Notation

\_\_\_\_\_ Add Birthday \_\_\_\_\_

△ Birthday Book

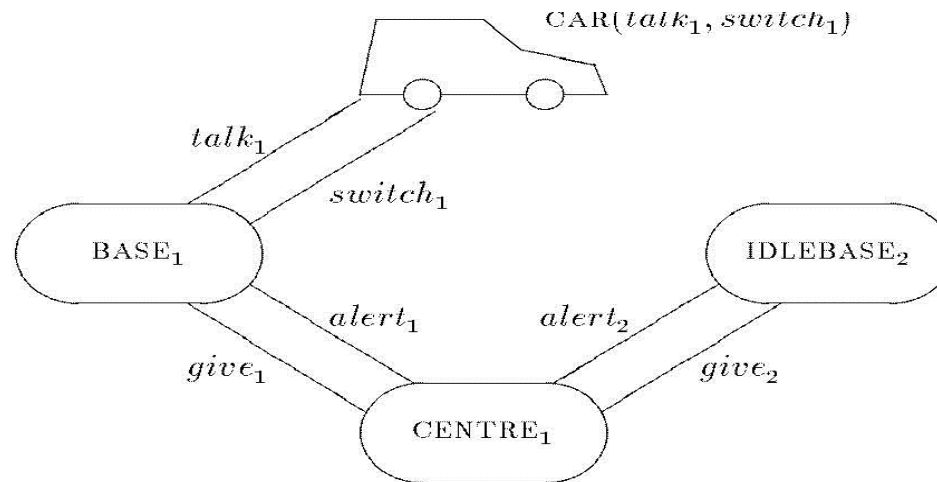
name?: NAME

date?: DATE

name?  $\notin$  known

birthday' = birthday U { name?  $\longrightarrow$  date }

# $\Pi$ -Calculus definition of the core of the handover protocol for the GSM Public Land Mobile Network

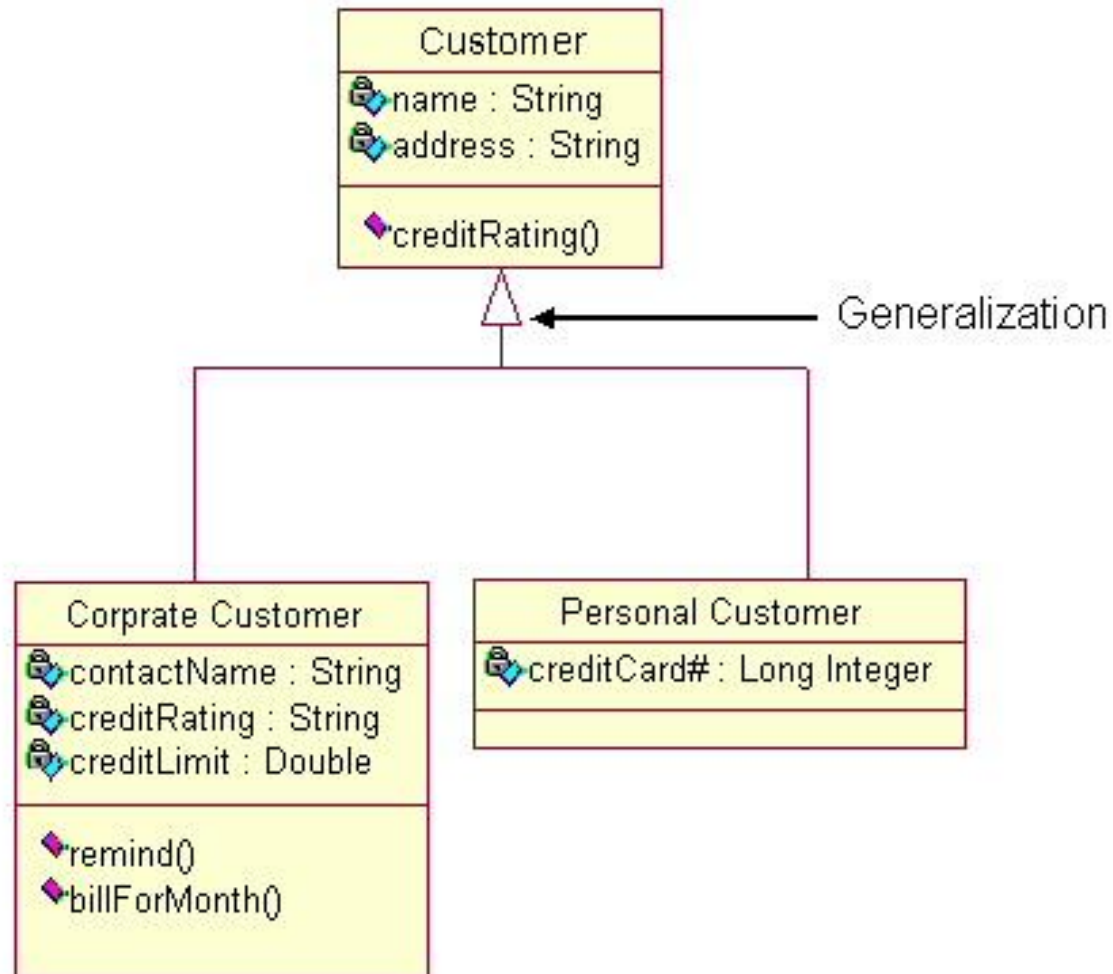


$$\begin{aligned}
 CAR(talk, switch) &\stackrel{ue1}{=} talk.CAR(talk, switch) \\
 &\quad + switch(talk' switch').CAR(talk', switch')
 \end{aligned}$$

© 1999, © 2001, © 2002, © 2003, © 2004, © 2005, © 2006, © 2007, © 2008, © 2009, © 2010, © 2011

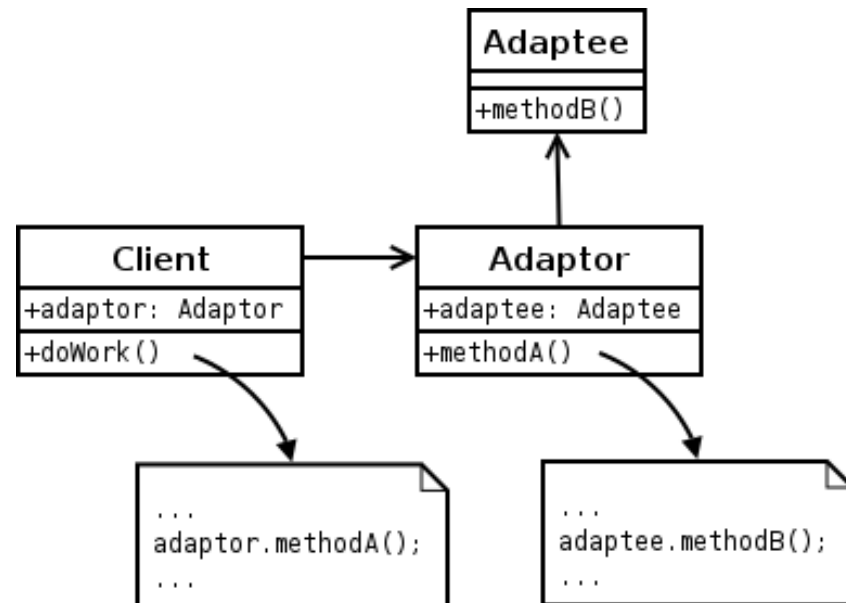
# Class Diagrams

Class diagrams typically describe the different entities of a system as classes and the relation between these.



# Architectural Patterns

- The *object adapter pattern* expressed in [UML](#). The adapter *hides* the adaptee's interface from the client.



# Definitions in Computer Science

- The act of definition forms one of the core intellectual activities of computer science.
- It brings new abstract notions into being which are then used - and sometimes analysed and studied.
- They serve to introduce new terms, programs, relations and properties over types of various kinds.
- In our examples the types of objects include numbers, sets, processes, classes etc.
- Much of this can be carried out in typed predicate logic (TPL, Turner 2011).
- Programming languages provide ways of putting definitions together.

## II. SPECIFICATION

# A Core Activity

- The specification, design, implementation and investigation of computational artefacts.
- This activity is to be found throughout the discipline.

# Philosophical Concerns

- What are specifications?
- How do they differ from definitions?
- What is the nature of the relationship between specification and artefact?
- Which things are computational artefacts?

# Beyond Definitions

- In terms of their logical form specifications take the form of stipulative definitions.
- However, they point beyond the definitions themselves:
  - i. They are aimed at the construction of artefacts.
  - ii. They tell us what to build.
  - iii. Definitions become specifications when they are given normative force over the construction of an artefact.

# Correctness and Malfunction

- The conceptual role of specification is to provide **a *criterion of correctness*** and ***malfunction*** for artefacts.
- This role is not tied to the actual process of construction. We may arrive at the artefact through various routes.
- If it says *beans* on the can, we expect it to contain beans not bananas. How they got there may explain how any mistake was made, but not why it is a mistake.

# Computational Artefacts

- Computational artefacts: the things computer scientists build.
- They may be either physical or abstract.

# Abstract Artefacts

1. Programs
2. Abstract Machines
3. Compilers
4. Type Inference Systems
5. Theorem provers
6. Natural Language systems
7. Speech recognition Systems
8. Programming Languages
9. Relational databases
- 10...etc.

# Physical Artefacts

- *A program on a chip*
- *Physical Machines (computers)*
- *Physical networks*
- *Digitised version of any software system*
- *GPU*
- *CPU...*
- *Etc.*

# Different Guises

- These two classes are quite different and generate two different notions of correctness/malfunction.
- In addition, computational artefacts often have both a *physical* and an *abstract* guise— a quite distinctive feature of computational artefacts.

# Abstract and Physical

- Abstract objects are normally taken to be subject to conceptual and mathematical investigation. They are causally inert and so, presumably, may not be investigated empirically.
- Physical ones causally interact with each other. They are subject to empirical investigation. Their properties cannot be established by mathematical means alone.
- Computer science raises some concerns over these standard characterisations.

# **III. MACHINES**

## **ABSTRACT AND PHYSICAL**

# Philosophical Concerns

- The nature of *correctness* for physical artefacts.
- The relationship to the simple mapping account of Hilary Putnam.
- Pancomputationalism.

# Abstract Machine Definition

Machine

***Update***:  $States \otimes Location \otimes Values \rightarrow States$

***Lookup***:  $States \otimes Location \rightarrow Values$

***Lookup(Update(s, x, v), y) = Lookup(s, y) where  $x \neq y$***

***Lookup(Update(s, x, v), x) = v***

# From Definition to Specification

- In itself it is just a definition of an abstract machine.
- But it may also be taken a specification of a physical one.
- It is taken to lay out the functional requirements of the required physical device.
- It is then given normative force over the construction of the physical one.

# Correctness and Malfunction

- Correctness is an empirical claim about the physical machine i.e., it meets the specification given by the abstract one.
- If when the machine is updated with input 3 into location  $z$ , it puts 7 in location  $u$ , then the physical machine has *malfunctioned*.
- But it is the **artefact, the physical machine** not the specification that is being tested. If the above happens the physical machine must be modified.

# Extensional Agreement

- Correctness is tested by testing whether the physical machine matches the abstract one.
- Here the relationship between the abstract and physical components is fixed ahead of time (e.g. what constitutes a physical state is fixed).
- This fixed relationship determines what correctness means.

# Simple Mapping Account- Putnam

- In contrast, the simple mapping account only demands that physical system can be put (via some interpretation) in extensional agreement with the abstract one.
- A physical system **P** Implements an abstract one **A** just in case there is a mapping  $I$  from the states of **P** to the states of **A** such that: for any abstract state transition  $s_1 \Rightarrow s_2$  if the system is in the physical state  $S_1$  where  $I(S_1)=s_1$ , it then goes into the physical state  $S_2$  where  $I(S_2)=s_2$

# Extension of Update

With only two locations l and r, and two possible values 0 and 1. Then we have four possible states (0,0), (0,1), (1,1) and (1,0).

Update (r,1)   Update (r,0)   Update (l,1)   Update (l,0)

(0,0)	(0,1)	(0,0)	(1,0)	(0,0)
(0,1)	(0,1)	(0,0)	(1,1)	(0,1)
(1,0)	(1,1)	(1,0)	(1,0)	(0,0)
(1,1)	(1,1)	(1,0)	(1,1)	(0,1)

# Pancomputationalism

- But this is too easy: any collection of coloured stones arranged as the update table will be taken to implement the table.
- This leads to a form of pancomputationalism where almost any physical system implements any finite abstract machine

# The Difference

- In the specification picture a fixed relationship between the abstract definition and the physical device determines what correctness means.
- In contrast, SMA only demands that physical system can be put (via some interpretation) in extensional agreement with the abstract one.
- This does not require any predetermined relationship between the physical and abstract components; it is sufficient for one to be constructed post-hoc.

# Normative Nature of Specification

- If I observe a man arranging stones in a way that is consistent with the extensional arrangement determined by my abstract machine, I may not infer that he is building a machine according to the abstract machine viewed as a specification.
- He might just be arranging them for aesthetic reasons.
- To be constructing an artefact according to the definition acting as a specification, he must be following the rules.
- This is the essence of specification and implementation.
- Implementation is not accidental.

# **IV. SPECIFICATION AND THEORY**

# Philosophical Concerns

An abstract machine may be considered as a

1. *A specification* of the physical machine (present view)
2. *A theory* of how the physical machine will behave

*The difference between 1 and 2 reflects an important conceptual difference*

# Philosophical Concerns

- What is the difference between a specification and a theory (of an artefact)?
- Do they have the same form and function?

# Theories of Artefacts

- I find an artefact in a field. It has a row of boxes with numbers in them. It has a keyboard attached with numbers on.
- Above each box is a button. I press one and a number appears. I press a number on the keyboard and press the button above a box and the number in the box is replaced.
- I postulate that it is a simple store machine. This is now a *theory* of the artefact.

# Specifications versus Theories

- While specifications have a similar logical form to theories they have an entirely different function.
- Theories are intended to be descriptive or explanatory. They are evidenced by natural artefacts. The world is not correct or incorrect.
- A specification is not something that is to be tested; it is not correct or incorrect. They fix what a correct artefact is. It is the artefact that is under test. When things go wrong, it is the artefact that is to blame.

# *SCIENTIFIC CARS*

It has been identified that there is a small risk of a brake problem in certain *Scientific* models. While customer safety has always been the primary focus for *Scientific*, we never intended the handbook to be a normative prescription of how the brakes should behave but rather as a theory or prediction of their behaviour. In keeping with this, we encourage drivers to test their brakes under the most extreme circumstances. We understand that that this might cause nightmares among the infirm, but *Scientific* would greatly appreciate receiving details of the resulting information. At no time shall we recall cars that are unsafe, but be assured that, in the fullness of time, we shall issue new handbooks that provide a better theory of how the breaks will perform.

# Restricted Mappings

- The SMA coupled with the danger of *pancomputationalism* has driven some authors (e.g. Chalmers, Copeland) to attempt to provide an account of implementation that somehow restricts the class of possible mappings.
- In particular, these (and others) authors seek to impose causal constraints on such interpretations. But this seems to be at odds with practice.
- However, there is an approach that is complementary to our specification approach which, without starting with the SMA gets at the heart of this demand for more causal constraints.

# Abstract Machines as Theories of Concrete Ones

- When so viewed we get to the heart of the claim that there must be a causal link in the statement of SMA.
- More exactly, in the statement

*For any abstract state transition  $s_1 \Rightarrow s_2$  if the system is in the physical state  $S_1$  where  $I(S_1)=s_1$ , it then goes into the physical state  $S_2$  where  $I(S_2)=s_2$*

- The material conditional is replaced by a counterfactual one. But this seems rather arbitrary – the SMA just does not fit practice.
- Treating the abstract machine as a theory of the physical one offers a less arbitrary explanation.

# **V. THE DEFINITION OF PROGRAMMING LANGUAGES**

# Philosophical Concerns

- What defines a programming language?  
The Normative nature of definition.
- Operational versus denotational semantics.
- Compositionality, full abstraction etc.

# Grammar

- Grammar only pins down what the legal strings of the language are. It does not determine their meaning
- It does not enable us to write correct programs.
- We shall illustrate some issues with the following toy programming language.

# The While Language

$P ::= x := E \mid \text{skip} \mid P; P \mid \text{if } B \text{ then } P \text{ else } P \mid$   
 $\text{while } B \text{ do } P \mid$

$E ::= x \mid 0 \mid 1 \mid E + E \mid E * E \mid$

$B ::= x \mid \text{true} \mid \text{false} \mid E < E \mid \neg B \mid B \wedge B \mid$

# Correctness

- A semantic account must reflect the intentions of the designer about the constructs of the language.
- However it is expressed or conveyed, it must provide criteria of correctness and malfunction.
- At some level, it must guide the compiler writer and the user.

# Metacircular Interpreters

- Defined via an interpretation into another programming language (or a subset of the source one)

$$L_1 \Rightarrow L_2$$

- Here the second language is intended to provide the semantics of the first.
- It is not grounded: unless the target language has a semantic interpretation it just passes the burden of correctness to another language.

# Semantics Grounded in Machines

- Programming languages get their semantic interpretation in terms of a machine.
- Maybe achieved layer by layer, one language getting its interpretation in the next, until a machine ( $\mathcal{M}$ )
- Provides the final and actual mechanism of semantic interpretation.

$$L_1 \Rightarrow L_2 \Rightarrow L_3 \Rightarrow L_4 \dots \dots \mathcal{M}$$

# What kind of Machine?

- But what kind of machine? Should the semantic account be given in terms of
  - i. A physical machine?
  - ii. An abstract machine?

# Fetzer-Colburn on Semantics?

- *...programs are supposed to possess a semantic significance that theorems seem to lack. For the sequences of lines that compose a program **are intended to stand** for operations and procedures that can be performed by a machine, whereas the sequences of lines that constitute a proof do not. (Fetzer 1988)*

- $A:=13*74$

*Physical memory location A receives the value of physically computing 13 times 74 (Colburn 2000).*

- They interpret this as physical machine semantics.

## No Semantics Without Physical Implementation

- On the physical account the semantic account is given by an implementation on a physical machine i.e., the intended meaning is to be given by the actual effect on the state of a physical machine.
- We might express the positive demand as:

*No semantics without physical implementation*

# Physical Machine Semantics

- Consider the assignment instruction

$$x := E$$

How is its semantics to be given on a physical machine?

- Presumably, the machine does what it does when the program is run - and what it does determines the meaning of assignment.

# Kripke on Machines and Meaning

*Actual machines can malfunction: through melting wires or slipping gears they may give the wrong answer. How is it determined when a malfunction occurs? By reference to the program of the machine, as intended by its designer, not simply by reference to the machine itself. Depending on the intent of the designer, any particular phenomenon may or may not count as a machine malfunction. A programmer with suitable intentions might even have intended to make use of the fact that wires melt or gears slip, so that a machine that is malfunctioning for me is behaving perfectly for him. Whether a machine ever malfunctions and, if so, when, is not a property of the machine itself as a physical object but is well defined only in terms of its program, stipulated by its designer. Given the program, once again, the physical object is superfluous for the purpose of determining what function is meant.*

# Normative Requirements

- *The fact that the expression means something implies, that there is a whole set of normative truths about my behavior with that expression: namely, that my use of it is correct in application to certain objects and not in application to others. ....*
- *The normativity of meaning turns out to be, in other words, simply a new name for the familiar fact that, regardless of whether one thinks of meaning in truth-theoretic or assertion-theoretic terms, meaningful expressions possess conditions of correct use.*
- *Kripke's insight was to realize that this observation may be converted into a condition of adequacy on theories of the determination of meaning: any proposed candidate for the property in virtue of which an expression has meaning, must be such as to ground the 'normativity' of meaning-it ought to be possible to read off from any alleged meaning constituting property of a word, what is the correct use of that word.*

*Boghossian on rule following*

# Correctness

- A semantic account, however it is expressed or conveyed, must provide criteria of correctness and malfunction.

# Correctness

- Semantic accounts are not empirical claims about the behaviour of a physical machine; they are not theories of the machines behaviour.
- They provide a notion of correctness. The semantic account of assignment must determine what it means for the physical machine to behave correctly.
- If the command  $x:=10$  places 28 in location  $y$ , this is not correct.
- On physical machines there is no notion of *malfunction* since there is no independent specification. And so there is no notion of *correctness*.

# Operational and Denotational Semantics

- What is the difference?
- Is it that operational semantics is based upon a machine while denotational semantics is not?
- Is it that operational semantics is syntactic whereas denotational semantics is somehow semantic- presumably because it is mathematical?

# Operational Semantics

We shall write

$$\langle P, s \rangle \Downarrow s'$$

to indicate that evaluating  $P$  in state  $s$  terminates in  $s'$  .

# Sequencing and Conditionals

$$\frac{\langle P, s \rangle \Downarrow s' \quad \langle Q, s' \rangle \Downarrow s''}{\langle P; Q, s \rangle \Downarrow s''}$$

$$\frac{\langle B, s \rangle \Downarrow \text{true} \quad \langle P, s \rangle \Downarrow s'}{\langle \text{If } B \text{ do } P \text{ else } Q, s \rangle \Downarrow s'}$$

$$\frac{\langle B, s \rangle \Downarrow \text{false} \quad \langle Q, s \rangle \Downarrow s''}{\langle \text{If } B \text{ do } P \text{ else } Q, s \rangle \Downarrow s''}$$

# While

$\langle B, s \rangle \Downarrow \text{true}$      $\langle P, s \rangle \Downarrow s'$      $\langle \text{while } B \text{ do } P, s' \rangle \Downarrow s''$

---

$\langle \text{while } B \text{ do } P, s \rangle \Downarrow s''$

$\langle B, s \rangle \Downarrow \text{false}$

---

$\langle \text{while } B \text{ do } P, s \rangle \Downarrow s$

# It cannot be syntax all the way down

*We can apparently get quite a long way expounding the properties of a language with purely syntactic rules and transformations.....One such language is the Lambda Calculus and, as we shall see, it can be presented solely as a formal system with syntactic conversion rules.....But we must remember that when working like this all we are doing is manipulating symbols - we have no idea at all of what we are talking about. To solve any real problem, we must give some semantic interpretation. We must say, for example, "these symbols represent the integers".*

*Stoy in his book on Denotational Semantics*

- Peter Landin ISWIM/Dana Scott OWHY.

# Mathematics and Operational Accounts?

- Operational accounts are just translations into other languages; they do not reach the *abstract world of mathematics*.
- Mathematics is about *abstract* objects and a mathematical theory must point beyond the axioms and rules to these objects.

# Denotational Semantics

- Meaning via denotation into sets and functions

$$C: \text{Program} \Rightarrow (\text{State} \Rightarrow \text{State})$$

1.  $C \ x:=n \ s = \text{Update}(x, n, s)$
2.  $C \ \mathbf{skip} \ s = s$
3.  $C \ P_1; P_2 \ s = C \ P_2 \ (C \ P_1 \ s)$
4.  $C \ \mathbf{if} \ B \ \mathbf{then} \ P_1 \ \mathbf{else} \ P_2 \ s = \text{if } C \ll B \ll s = \mathbf{true} \ \text{then } C \ll P_1 \ll s \ \text{else } C \ll P_2 \ll s$
5.  $C \ \mathbf{while} \ B \ \mathbf{do} \ P \ s = \text{if } C \ll B \ll s = \mathbf{true} \ \text{then } C \ \mathbf{while} \ B \ \mathbf{do} \ P \ (C \ll P_1 \ll s) \ \text{else } s$

# What is Different About Set Theory?

- But why are set theoretic accounts philosophically different to our operational ones? All are written in languages with rules/axioms: all end in axiomatic theories.
- If there is any substance to the above metaphysical claims about set-theory, it must go beyond the syntax of the language of set-theory. Indeed, it must go beyond its axioms. Otherwise, we will have not reached the abstract universe of sets.

# The Metaphysics of Set Theory

- Gödel gives us some insight into what might be involved

*Despite their remoteness from sense experience, we do have something like a perception also of the objects of set-theory, as is seen from the fact that the axioms force themselves upon us as being true. I don't see any reason why we should have less confidence in this kind of perception i.e., in mathematical intuition than in sense perception, which induces us to build up physical theories and to expect that future sense perceptions will agree with them, and moreover, to believe that a question not decidable now has meaning and may be decided in the future . Godel 1964*

- Gödel draws an analogy with the perception of physical objects; sets are perceived in an analogous way but what is perceived is neither the axioms and rules, nor the expressions that generate sets, but the sets themselves. It would seem that such knowledge must be taken as *knowledge of sets* rather than *knowledge that some proposition about sets is true*.

# Why are Operations and Sets Different?

- But even under Gödel's perspective, it seems hard to see how the difference between operations and sets could be made out. It is certainly not clear that Gödel would have supported such a distinction. In his Gibbs Lecture he writes:

*The greatest improvement was made possible through the precise definition of the concept of finite procedure, which plays a decisive role in these results. There are several different ways of arriving at such a definition, which, however, all lead to the same concept. The most satisfactory way, in my opinion, is that of reducing the concept of finite procedure to that of a machine with a finite number of parts, as has been done by the British mathematician Turing.*

- During this period, Gödel thought that Turing's analyses of finite procedure was definitive. In Wang's words, Gödel saw the problem of defining computability as:
- *an excellent example of a concept which did not appear sharp to us but has become so as a result of a careful reflection*
- One assumes that he would have assigned the notion of finite procedure a similar metaphysical status to sets.

# **VI. THE SPECIFICATION OF ABSTRACT ARTEFACTS**

## Philosophical Concerns

- What is correctness for abstract artefacts?
- What is the difference between a program and a specification?

# Specification and Artefact

- There are many kinds of relationship between abstract objects that can be understood as a *specification/artefact* relationship.
- One abstract relation or function implements another.
- An axiomatic definition of an abstract structure and a particular instance of that structure e.g. A group structure and a concrete example.
- The specification has jurisdiction.

# Examples

The following is a specification of a *swop* operation as a relation between the before and after states of our abstract machine.

**(S)**

*For all stores  $s$ , locations  $x$  and locations  $y$  we have:*

$$\text{Swop}(s, s', x, y)$$

$\cong$

$$\text{Lookup}(x, s') = \text{Lookup}(y, s) \leftrightarrow \text{Lookup}(y, s') = \text{Lookup}(x, s)$$

# Correctness

What does it mean for **P**,

**(P)**  $z:=x; x:=y; y:=z$

to be correct relative to the specification **S**? Given its operational semantics, we may state the correctness conditions for the program **P**, as follows. For all states  $s, s'$ ,

$\langle P(x,y),s \rangle \Downarrow s'$  *implies*  $swop(s,s',x,y)$

# Refinement

*S is a refinement of R (written  $R < S$ )*

iff

I.  $\text{Dom}(R) \subseteq \text{Dom}(S)$

II.  $\underline{S}(x, y) \rightarrow R(x, y)$

Where  $\underline{S} = S \downarrow \text{Dom}(R)$

- Where  $S$  is functional II reduces to:  $R(x, S(x))$ .  
It states that  $S$  correctly implements  $R$
- $R$  provides the correctness criteria for  $S$
- Proof of correctness is a mathematical affair.

# What is the Difference between a Specification and a Program?

- Descriptive/Imperative?
- Implemented or not? In actuality or in theory?
- What to do/how to do it?

# Programs as Specifications

- Programs are the occupants of programming languages. Programs, as semantic things, are not strings of symbols or physical devices but abstract objects.
- Given some semantic account of programs where they are taken as relations between states of the underlying abstract machine, i.e., via their semantics each generates a relation/function.
- Of course there could be other ways that the semantics might represent them.

## Programs as Specifications or Artefacts

Via their semantics, programs may be viewed as specification or artefact i.e., via their semantics each generates a mathematical object.

```
x:= 0; y:= 1;  
while x < n do (x:= x + 1; y:= x * y)
```

$$Fac(0)=1$$
$$Fac(n+1)=n+1*Fac(n)$$

# Correctness Jurisdiction

- The difference between program and specification is not attached to any characteristic of the syntax of a programming or specification language.
- Something is a specification, as opposed to a program, when deemed to have correctness jurisdiction over its artefact.
- Linguistic programs may be considered as specification of physical programs.

# The Correctness of Physical Programs

- There is a difference between the correctness of abstract programs and their physical counterparts.
- What does it mean to say that the physical program is correct ? Correctness is an empirical claim about the physical program i.e., it meets the specification given by the abstract one, i.e., correctness of the physical device is tested by empirical means.
- Much of the heated debate on correctness is enlightened by paying more careful attention to what is the specification and what is the artefact.

# **VII. THE ONTOLOGY OF COMPUTER SCIENCE**

# Some Philosophical Concerns

- What kind of *thing* is a program?
- What kind of thing is a programming language?
- What kind of *thing* is a data type?
- The foundational role of set theory in mathematics has been extended to the abstract objects of computer science.
  - i. Data types are often seen as sets.
  - ii. Denotational semantics treats programs as if they are set-theoretic functions.

Is this appropriate?

# Programming Languages

- We have suggested that programming languages are fixed by the abstract operational semantics.
- Here the relation  $\Downarrow$  is taken to be sui-generis in the proposed theory of operations. It is axiomatised by the rules.
- Under this interpretation, a programming Language constitutes an axiomatic *theory of operations*.

# Equivalence

- *Define*

$$\langle P, s \rangle \Downarrow \triangleq \exists s'. \langle P, s \rangle \Downarrow s'$$

*This provides a notion of terminating operation.*

- *Define*

$$P \simeq Q \triangleq \forall s_1. \forall s_2. \langle P, s_1 \rangle \Downarrow s_2 \leftrightarrow \langle Q, s_1 \rangle \Downarrow s_2$$

*i.e., they have the same extensional behaviour.*

# Computational Theories

- Theoretical Computer Science has developed a large number of such mathematical theories.
- Theories of *Operations and Types* (typed Lambda Calculi, polymorphic systems, partial Lambda calculi,...).
- Theories of *Processes* ( $\Pi$ -calculus, CSP).
- Theories of *Objects and Classes* ( $\zeta$ -Calculus).
- Via such theories programming languages are (*definitional extensions* of) mathematical theories of computation.

# The Dual Nature of Programs

- Many authors (Moor 1978; Rapaport 2005b; Colburn 2004) discuss the so-called *dual* nature of programs. On the face of it, a program appears to have both a textual and a physical guise.
- What is the relationship between them?
- The physical is an implementation of the linguistic? But it is unclear what this means.
- Programs, as textual objects, are said to *cause* mechanical processes. The idea seems to be that somehow the textual object physically causes the mechanical process.
- Colburn (2004) insists that software is a *concrete abstraction* that has a medium of description (the text, the *abstraction*) and a medium of execution (e.g., a *concrete* implementation in semiconductors). But this seems quite mystical.

# The Trinity of Programs

- ***Linguistic***: the written form expressed in a programming language.
- ***Abstract***: the abstract entity whose individuation is determined by the semantic equivalence relation of the theory of operations.
- ***Physical***: the correct physical representation of the linguistic item.

# The Linguistic and the Concrete

- The simple fact is that without reference to the abstract program there is no notion of correctness.
- There has to be some rules.
- The *correctness* of the implementation, a systematic way of translating linguistic entities into physical ones, is only fixed by the abstract one.
- Without the abstract account there is no way of articulating the relationship. The physical thing just does what it does.

# Types

- What kind of thing are they?
- Sets? Categories?
- Are they determined by the rules of type-checking?

$$\frac{x:A \ /-t[x]:B}{\lambda x.t:A \Rightarrow B}$$

- In terms of compiler correctness the operational rules are all we need.
- However,

# Mathematical Theories versus Theories of Computation

- The user needs more rules than the type inference rules. E.g. to reason about lists we require induction rules for lists.
- The formulation of such rules turns a programming language into a more traditional axiomatic theory.
- For example, to reason about numbers one might need the induction principle or even the whole of Peano arithmetic. But this is no longer unambiguously determined.

# **VIII. KNOWLEDGE OF COMPUTATIONAL ARTEFACTS**

# Philosophical Concerns

- What kind of knowledge can we have of computational artefacts?
- In principle abstract artefacts can be known a priori. But is this true of abstract computational artefacts?
- Because of their complexity can they only be known a posteriori?

# Complexity and Program Correctness

- Is the correctness of real software a mathematical affair?
- Is testing and verification the best we can do in regard to real software ?
- Does this make correctness an empirical question?
- Can we ever have a priori knowledge of real software? Even when the theorem prover is proven correct, the results are not knowable apriori – since the computations are carried out on a physical machine.
- Can such correctness proofs be grasped? Are they *real* proofs? Graspability and Wittgenstein on proofs.

# Tyler Burge: all mathematical Truths can be known a priori

- Provides an account where knowledge of such computer proofs can be taken as a priori knowledge.
- A priori knowledge according to Burge does not depend for its justification on any sensory experience - however, he allows that such knowledge may depend for its possibility on sensory experience. E.g. knowledge that red is a colour may be a priori, even though having this knowledge requires having sensory experience of red in order to have the concepts required to even formulate the idea.
- Assumption that all mathematical truths can be known a priori – may involve believing mathematicians without understanding the proofs.
- Do computer proofs have the same warrant?
- This is significant in that it closes the gap between pure mathematical knowledge ( e.g. Correctness proofs) and the correctness of physical programs.

# Knowledge of Programming Languages

- What is it to know a programming language? Consider the following considerations:
- A user learns the language empirically using its implementation on the given machine.
- The gleaned semantic interpretation is an empirical affair. Hence
- Programming languages together with their implementation are abstract/physical objects that are knowable only *a posteriori*.
- *Can a programming language be fixed in this way?*

# **IX. RULES**

# Axioms and Meaning

- How do formal systems get their meaning?
- In terms of another system?
- Why do we not fall into a form of formalism?

# Meaning Given by Rules

- Wittgenstein's position.
- Meaning is fixed by the axioms – understood as rules?

# Rule Following

- What is it to follow a rule?
- Kripke's paradox of rule following.

# Wittgenstein and Rule Following

- To follow a rule is to be able to reflect upon it.
- To follow a rule of any kind should one be able to answer the question : *why did you do that?*

# Semantics Revisited

- The operational semantics of a programming language provides rules for operational evaluation.
- Following them is subject to the above considerations.

# **X. COMPUTABILITY**

# Philosophical Concerns

- What is the nature of the Church-Turing thesis?
- Is it Normative /definitional?
- Is it like the notion of prime number – to be judged by its fruitfulness in mathematical practice?
- Is it somehow a claim about the world – a kind of empirical hypothesis that relates to what humans can do with pencil and paper?
- Is it a specification for a physical machine?
- Is it a theory about all possible physical devices?

# The Classical Thesis

The Church-Turing thesis concerns the notion of an *effective* or *mechanical* method in logic and mathematics.

- M is set out in terms of a finite number of exact instructions.
- M will, if carried out without error, produce the desired result in a finite number of steps;
- M can be carried out by a human being
- M demands *no insight or ingenuity* on the part of the human being carrying it out.

# The Turing Argument

- In order to provide a 'non-question begging' analysis of computation 'the smallest', or most fundamental, or least sophisticated parts must not be supposed to perform tasks or follow procedures requiring intelligence.
- *Instructions given the computer must be complete and explicit, and they must enable it to proceed step by step without requiring that it comprehend the result of any part of the operations it performs. Such a program of instructions is an algorithm. It can demand any finite number of mechanical manipulations of numbers, but it cannot ask for judgments about their meaning. ([Turing 4], p. 47)*

# An Important Epistemological Notion?

*Tarski has stressed in his lecture (and I think justly) the great importance of the concept of general recursiveness (or Turing's computability). It seems to me that this importance is largely due to the fact that with this concept one has for the first time succeeded in giving an absolute notion to an interesting epistemological notion, i.e., one not depending on the formalism chosen."(Gödel 1946 in Davis 1965: 84)*

# Wittgenstein's Response

- What has simplicity of rules got to do with it?
- Can there be a rule that requires no intelligence to follow?
- Consider *Go to step  $i$  if 0 is scanned*
- However simple this rule might be, it does indeed tell one what to do.
- According to Wittgenstein, to follow a rule of any kind one should be able to explain why you did what you did in terms of the rule.

# Turing's Move is to Reduce Normative Content

- The whole point of reducing rules to sub rules is to minimise the normative element.
- The only way to remove its normative content is to treat it not as a definition of what it does but as a theory /description of the device.
- Go to step i if 0 is scanned
- Must be seen not a definition with normative force but a description or theory of the causal events that occur in the mind/program of the computer which trigger off certain reactions.

# Definition and Theory

- Once again there is a contrast between the normative statement and a description/theory.
- On Turing's account the normativity of calculation disappears .
- Minimal sub rules function as descriptions of the computers putative hidden calculating machinery. See Shanker Discussion.

# Calculation

*You aren't calculating if, when you get now this, now that result, and cannot find a mistake, you accept this and say: this simply shows that certain circumstances which are still unknown have an influence on the result. This might be expressed: if calculation reveals a causal connection to you, then you are not calculating. . . . What I am saying comes to this, that mathematics is normative. ([49] VII, §61)*

# **XI. ABSTRACTION IN COMPUTER SCIENCE**

# Examples of CS Abstraction

- Procedures and Functions
- Abstract data types
- Recursion
- Patterns....

Are they all the same?

# Abstraction and Specification

- The specification relationship between abstract artefacts.
- The reverse of specification.
- Category theory and adjoint functors
- *S is a refinement of R : R is an abstraction of S.*
- *Information* (whatever that is) is lost between *R* and *S*.

# Abstraction in Mathematics

- Is there one kind?
- Frege
- Wright: reasons proper Study
- Kitt Fine

# Abstraction in Mathematics and Computer Science

- Is there one kind of abstraction in computer science?
- Some authors have suggested that the differences between maths and computer science is that in the latter abstraction leaves behind an implementation trace.
- But is this true? Is there always one? Or is it more like there has to be one in principle? Indeed, is this always so?
- Do specifications as abstractions need to be implementable?
- Does Bishop's constructive mathematics satisfy this criterion?

# **XII. EMERGENCY**

# Some Papers

- ([Specification](#): Minds and Machines)
- [Machines](#) (To appear [Computable Universe](#))
- [Theories and Specifications](#)
- ([Understanding Programming Languages](#))
- ([Programming Languages as Mathematical Theories](#))

