CS 524 Lecture 9

José Meseguer

1. What is Concurrency?

Often, the simplest questions can be the hardest to answer. This is constantly demonstrated by small children, who ask their parents the deepest questions in the form:

why —···— ?

Questions that their parents, sometimes, cannot answer, or do answer in a highly unsatisfactory way.

So, what is Concurrency?

The easy copout is to answer by example:

"it is the property exhibited by Petri nets, and the π-Calculus, and GPUs, and Cloud Computing, and the Internet, and ... ?... "

Another, quite reasonable, way to deal with the question is to give an approximate pre-scientific

answer: "the property that several events can take place simultaneously and independently of each other", which is Ok as far as it goes but, as with the approach by example:

1. Does not clearly delimit the scope of the phenomena [e.g.: Physics is choke full with concurrent phenomena in this pre-scientific sense]

Furthermore, neither the approach by example nor the prescientific way to answer the question provide any help on the problem of:

2. How to model mathematically concurrent systems in general, within a general theory of computation.

To be sure, the approach by example can exhibit specific mathematical models for some concurrent systems, but fails to answer questions (1) and (2).

2. Rewriting Logic

To the best of my knowledge, before 1990 nobody had provided a full, satisfactory answer of a scientific nature to the question of

what concurrency is, squarely addressing problems (1) and (2). In 1990 rewriting logic was proposed as a mathematical answer to this question addressing problems (1) and (2) as follows:

a. Concurrent systems are systems that can be mathematically specified by rewrite theories in rewriting logic.

b. Any rewrite theory $\mathcal{R}$ has an associated rewriting logic inference system $\mathcal{L}(\mathcal{R})$, so that any finite concurrent computation of the system specified by $\mathcal{R}$ can be specified as a proof in $\mathcal{L}(\mathcal{R})$ and, conversely, any such proof specifies a concurrent computation

c. The concurrent computations of the system specified by $\mathcal{R}$ can be precisely modeled as a category $\mathbb{T}_{\mathcal{R}}$, whose objects are the system's concurrent states, and whose arrows are its (finite) concurrent computations, which are equivalence classes of proofs in $\mathcal{L}(\mathcal{R})$

2.1 Rewriting Logic: Statics

Before even starting to answer the dynamics question of what a concurrent computation *is*, we need to address the more basic question:

what is a concurrent state?

The same challenges appear also at this level, both in terms of scope (problem (1)), and in terms of having a general theory of concurrent computation.

In classical computability theory, which is the theory of sequential computation, this question is swept under the rug by means of the Turing machine encoding fallacy, and is therefore given the nonchalant answer:

A state is the state of a Turing machine

The answer is of course fallacious, because what it really means is:

A state of a sequential system is something that can be compiled into the state of a Turing machine.

For sequential computation this fallacy is something we have managed to live with since the 1940's, because, thanks to the genius of John Von Neumann, Turing machines have been successfully implemented as von Neumann machine architectures, so that any sequential computational system, and, in particular, any sequential programming language, can be compiled into a von Neumann machine and therefore, mathematically, into, actually, some variant of a Turing machine.

Conceptually, however, the ~~cost~~ cost of this fallacy has been quite high, because it has completely ~~eclipsed~~ eclipsed in the minds of most people the logical, and therefore declarative nature of computation [in spite of the fact that the lambda calculus is historically prior to the Turing model as a model of computation!]. The tragedy is that most computer science graduates and researchers can only think of computation in an imperative way: as the performing of a sequence of instructions changing the state of a [von Neumann/Turing] machine.

I claim that, for concurrent computation, this Turing machine encoding monkey business has no credible equivalent/analogue ~~alternative~~, and can only lead to utter conceptual confusion. Why so?

Because concurrent computation is a much more polymorphic [poly = many]; [morphe = form] phenomenon than sequential computation. For example, some concurrent systems are synchronous, while others are asynchronous and, furthermore, the structure of their states is essential for the kinds of concurrent computations that are possible in a given concurrent system.

The upshot of all this is that translations between models of concurrency are much more treacherous than translations of sequential systems into Turing machines, because much more will be lost in translation.

In fact, I claim that

There is no credible universal model of concurrent computation, so the translation approach is bound to fail.

In this sense, rewriting logic is completely pluralistic: it questions the very possibility of a universal model and the usefulness of translations as a means of mapping any model into such a non-existent universal models. It proposes, instead, intrinsic formal definitions of each model $\mathcal{M}$ by its own rewrite theory $\mathcal{R}_{\mathcal{M}}$ without any encoding whatsoever!

However, it does not exclude translations $H : \mathcal{R} \to \mathcal{G}$ between rewrite theories, and therefore between their models: $H : \mathcal{T}_{\mathcal{R}} \to \mathcal{T}_{\mathcal{G}}$ which, as we shall see in future lectures can be very useful. What it seriously questions as wishful thinking is the existence of a universal rewrite theory $\mathcal{U}$ so that any other rewrite theory $\mathcal{R}$ can be compiled by a translation $H : \mathcal{R} \to \mathcal{U}$.

Paradoxically, however, [but there is only an apparent paradox], rewriting logic is a reflective logic which can encode it own meta-level, and there is a universal theory $\mathcal{U}$ in exactly this reflective

way. In particular, Maude itself is a reflective language in this exact logical sense, and its META-LEVEL module efficiently implements useful computations in rewriting logic's universal theory $\mathcal{U}$. This endows Maude with very powerful meta-programming and higher-order programming capabilities [analogous to Type: Type in type theory but without its paradoxes].

Back to states! The expedient of a fallacious answer such as "a state is the state of a Turing machine" is out of the question. So what is a satisfactory general answer to characterize the states of a concurrent system?

The obvious answer is:

A concurrent state is any computable data structure, which in an actual implementation may be a distributed across several processors.

But what this obvious answer is silent about is:

What is it mathematically?

Again, in classical computability theory data structures are encoded as either strings on a finite alphabet, or as natural numbers, which are strings of the one-letter alphabet $\{1\}$ [counting with one's fingers: $3 = 111$, $5 = 11111$], but this is just part and parcel of the Turing game: the problem is swept under the rug.

Fortunately, however, since the mid 1970s, the theory of algebraic data types was developed, again as a pluralistic and intrinsic, way of defining data structures as elements of algebraic data types, where each such algebraic data type is defined as the initial algebra $T_{\Sigma/E}$ of the equational theory (Σ, E) with function symbols Σ and equations E defining the data type $T_{\Sigma/E}$ in an intrinsic way, with no encoding whatsoever!

Fortunately, also, in the 1980s Bergstra and Tucker proved the following important meta-theorem:

Theorem (Bergstra and Tucker, ICALP 1980, 76-90). Any computable data type can be formally specified as the initial algebra $T_{\Sigma/E}$ of an equational theory (Σ, E) [perhaps with some auxiliary functions] where Σ has a finite set of function symbols [typed with a finite set of sorts] and E is a finite set of equations that, when oriented as rewrite rules $\vec{E}$, are Church-Rosser [confluent] and terminating.

With other colleagues, in the 1980s I extended the theory of algebraic data types to richer equational logics: from many-sorted to order-sorted to membership equational logic [the logic of Maude's functional modules].

For simplicity I will stick to one-sorted [unsorted] equational theories in what follows [but everything can and has been extended to the membership equational logic level]. This is not so bad, since any typed algebraic data type can be viewed as untyped by disregarding types [as for the case of the λ-calculi].