This course of lectures will split into three parts. The first part will show how to present a diagnostic trace of an execution of a concurrent object-oriented program as a diagram in a finite discrete geometry. The second part will give intuitive geometric proofs that the traces satisfy a collection of simple algebraic laws. These laws can be used in the implementation of a programming language, to justify optimisation of programs, perhaps by translation into a lower level language. The third part will derive from the algebra a collection of logical proof rules for reasoning about programs and their correctness.

1. Introduction

Our treatment of geometry owes everything to the ideas and inspiration of the pioneers of the subject. Euclid defined the basic concepts of plane geometry, of which we shall exploit his treatment of points, lines, figures and their edges. We also exploit his famous parallel postulate. Two thousand years later, Descartes introduced orthogonal axes and coordinates. He defined a point as the unique element at the intersection of all pairs of
orthogonal coordinates. Though I shall adapt them to our needs, these features are fundamental to our idea of geometry.

A diagram of program execution
- has one (horizontal) dimension of space
- and one (vertical) dimension of time
- It is non-metric, with no measurement of distance in space or intervals of time
- It has points representing actions inside a computer and arrows representing causation between them
- It appears as a finite and acyclic directed graph with non-termination and cycles defined as errors.

We will draw our geometric diagrams in two dimensions, as on a familiar sheet of graph paper. The horizontal axis of the diagram represents real space internal to the computer (including its memory). The vertical axis represents real time spent in program execution. But we deliberately do not entertain the concept of measurement, either of the distance between the points in space or of the duration of an interval in time. This is a major abstraction from reality. It is justified because the programmer in a high-level language willingly surrenders to the implementation the control of details of timing of execution and the positioning of resources allocated in the memory.

I will illustrate the geometric constructions by application to an example program execution, which includes many of the essential features of C++ and other modern object-oriented languages. They are listed on this slide. Geometric diagrams are good for conveying ideas and intuitions. To consolidate understanding, a formalisation of the geometry will be more appropriate. Formalisation will also be an essential preliminary to the correct design of tools that will help programmers to test their programs. These will be topics for later lectures in the series.
The points of the diagram represent actions that occurred inside the computer during a single execution of a small segment of a program under test. The segment is a syntactic component of a structured high-level program, for example, a basic command of the language, or a sequential or concurrent composition, or a loop or a method body. The diagram is enclosed in a rectangular box, whose edges define its interfaces with diagrams neighbouring segments of the program. We will see that the horizontal edges are interfaces with sequentially executed neighbours, and the vertical interfaces with concurrently executed neighbours.

This slide shows six blue vertical lines, which are parallel in the usual sense that they do not share any points. They serve as the vertical coordinates of our plane geometry. The whole coordinate records a complete history of the actions of a single separate object in computer memory. The points that lie on each coordinate record the successive actions performed by that object. The earlier actions appear above the later actions. The segment of a coordinate that lies between two adjacent points will be called an arrow.

The middle coordinate shows the allocation and disposal of an object as the top and bottom points of its coordinate. This remark can be regarded as an intuitive definition of the meaning of allocation and deletion. It states the intention, with no hint of how the intention may be implemented.

The other five vertical lines extend beyond the box, both above it and below. This shows that the coordinates continue into the horizontally neighbouring boxes, which record the behaviour of these objects before and after execution of this diagram.
A communication is a sloping segment of a vertical coordinate, shown in bold on this diagram. The slope of the segment indicates that the actions at its two ends occur at a different time as well as at a different place.

The point at the top of the communication segment represents the output of a message, and the point at the bottom represents an input. The segment often crosses the edge of a diagram, indicating that either the input or the output action occurs in a neighbouring diagram. Examples are shown on the left and right edges of this slide.

The slope in the middle of the slide represents transfer of ownership of an object from one thread to another, as will be explained later.

A horizontal coordinate (drawn in red above) denotes the execution of a single atomic command of the program. The points on each horizontal coordinate represent the simultaneous performance of actions by two or more objects, including the thread that issued the command. The coordinate satisfies the classical definition of an atomic action, that there does not exist any instant in time at which the some of its actions have been observed to occur, and others have not.

Each point in a diagram is the unique point that lies at the intersection of a horizontal with a vertical coordinate. Of course, the vast majority of such intersections are empty, simply because there is no action that actually occurred at that time and at that place. This sparsity of points is a significant deviation from Cartesian Plane Geometry, in which every intersection of two orthogonal coordinates has exactly one point.
The absence of an action at the intersection of coordinate lines is indicated simply by omitting the blob. To reduce clutter, our example diagrams have been carefully designed to contain no such empty intersections.

The segment of a coordinate stretching between any two of its adjacent points is called an arrow. An arrow denotes a causal dependency between the actions at its head and its tail. The fundamental principle of causality states that the effect at the head of the arrow cannot happen before the cause, that is given by the tail action.

All the arrows on a vertical coordinate point downward, in the direction of increase in time. The arrows on each horizontal coordinate usually point in both directions, so that all its actions are causally constrained to occur simultaneously. In future, I shall omit the arrow heads, since they can be easily restored.

A fence is an atomic event of simultaneous execution of a command from each of the threads involved. A familiar example is the synchronised communication of an unbuffered message between just two threads. A fence is represented by a horizontal coordinate that stretches across two or more boxes. Any segment of it that crosses a vertical edge of a box is drawn with dashes, as shown on the right-hand edge of this diagram.
3. LABELS

Labels

- The purpose is to correlate a diagram with the text of the segment of program whose execution it displays.
- The label on a vertical coordinate is an object name or on a horizontal coordinate it is a basic command.
- Each arrow may also be labelled by a value computed and used during program execution.

In this diagram, the blue vertical coordinates are labelled by the unique name or address of the object whose behaviour it records. Each name belongs implicitly to some object class, for example, a thread, a variable or an input or output port on a communication channel. The class defines and determines the correct behaviour of all its objects.

This diagram displays two thread objects, uniquely named as t and u. The name y denotes a simple integer variable. The name x uniquely identifies an integer variable allocated and disposed locally to this diagram. The label c? denotes the input port of a buffered communication channel c, and the label d! is similarly the output port of a channel d.

The subscripted variables at the side of the diagram are the unique names of messages that have been sent along the channels c and d. The subscript makes these names unique by giving the serial number of the message on that channel. They ensure that when diagrams are assembled, the right connections are made between local and more distant actions.
Each horizontal coordinate of the diagram is labelled by the basic command in the thread which has triggered execution of the atomic event. For example, on the left of this slide the actions of the thread \( t \) are an allocation of a new object \( x \); an input of a message on channel \( c \); and finally a release of ownership of \( x \). On the right of the slide there are the actions of the thread \( u \). They are an assignment to the variable \( y \), an acquisition of the ownership of \( x \), an output on channel \( d \), and a disposal of the object \( x \). The indices on the release and acquire commands give the serial numbers of these actions in the life of the variable \( x \).

On this slide, the additional bold numeric labels on each arrow indicate the value transmitted from the tail to the head of the arrow by performance of both these actions. On a vertical arrow, the label is the current value of the object during the interval between the occurrence of the tail action and of the head action. On a sloping arrow, it is value of the message communicated. The correctness of the values may be easily checked against the rules that define each kind of basic command in the language.

The state of the executing computer memory at the beginning of execution of a box (and at the end) is defined as the function that maps the name of each arrow crossing the top (or bottom) edge to the value which labels the arrow. For example, at the top edge of this diagram, the initial state maps the variable name \( y \) to the value 3, whereas at the bottom edge the final state maps the variable \( y \) to the value 4. A similar function describes all the interactions between neighbouring boxes that occurred during execution. For example, \( c[84] \) has the value 8.

Our model of program execution so far has been entirely event-based. We have just explained how memory can be introduced into an event-based model: it is defined in
terms of events, by exploiting the labels for objects and values that are attached to the vertical coordinates.

This diagram tells the story of part of the execution of an extremely unlikely segment of a program. The story starts at the top left with the allocation of a new object \( x \) by the thread \( t \). This is followed by input from its local input port \( c? \) of the initial value 8, which is simultaneously assigned to \( x \). The thread \( t \) then releases ownership of \( x \).

Meanwhile on the right of the diagram, the thread \( u \) has incremented the value of its own local variable \( y \). It then acquires \( x \) from the thread \( t \), and outputs on its output port \( d \) the sum of the values of \( x \) and \( y \). Finally it disposes the object \( x \).

The example program as text

\[
\begin{align*}
\{ & \text{new } x; \\
& \text{c?[8]} \text{t } x; \\
& \text{x}[3; \text{rel}]; \\
& \text{y} := y + 1; \\
& \text{y}[4; \text{acc}; \\
& \text{d}[3/2; (y + x); \\
& \text{dispose } x; \\
\}
\end{align*}
\]

where ; is sequential composition
and | is concurrent composition

This slide tells the same story as the diagram at the end of the previous section, expressed in the form of the text of the segment of program which was executed. It uses semicolon in a familiar way to denote sequential composition, and the single vertical bar denotes concurrent composition, as in process algebras like CCS. It is a great simplification to treat the bar of concurrency as a simple binary program connective, just like the semicolon which calls for sequential execution.

It is possible to reconstruct the essence of a diagram from the text that was obtained from it as described above. This is because the text contains enough information in the subscripts to reconstruct the sloping arrows.

4. SEGMENTATION OF DIAGRAMS

The definitions of two familiar composition operators follow the normal style of definitions in the natural sciences and in applied mathematics. Each technical term is defined by relating it to the real-world phenomena which it denotes. The definition describes in general terms the purpose and intended effect of the defined object without giving even
a hint of how it is implemented. Thus \((p;q)\) describes an execution which starts with a complete execution of \(p\) and finishes with a complete execution of \(q\). The start of \(q\) is preceded by the execution of \(q\). For the concurrent composition \((p;q)\), execution starts with the simultaneous start of both \(p\) and \(q\), and finishes with the simultaneous end of both \(p\) and \(q\). These definitions are illustrated in the following diagrams.

This slide shows a vertical segmentation of a rectangular box by introducing a new vertical edge into our familiar diagram. The title at the top of the slide gives a concrete syntax for the segment of program that was executed. It was formed by concurrent composition of two smaller segments, each named by the \(p\) or the \(q\) written just inside their boxes. The new splitting edge (together with all the arrows which cross it) is shared between the two component boxes. It appears as the right vertical edge of \(p\) and as the left vertical edge of \(q\). The horizontal edges are also split into disjoint segments at the point where the newly introduced line meets them.

It is immediately obvious from the horizontal edge at the top of the diagram that the two components \(p\) and \(q\) start together and finish together, as required by the definition of the purpose of concurrent composition. Furthermore, the top edge of \(p\) is wholly disjoint from the top edge of \(q\). Consequently, the vertical coordinates that cross the top edge of \(p\) are wholly distinct from those that cross the top edge of \(q\). The same is true of the bottom edges. This disjointness is the intuition behind the ownership discipline which we have incorporated into our geometric model.

An ownership discipline for threads and objects is designed to prevent a race between two threads for access to an object. The standard way of preventing them is to require that any two accesses from different threads should be related by a causal chain. We implement this policy by insisting that the vertical edge which separates the two threads
must pass through a sloping or a dashed horizontal arrow, which explicitly passes ownership from one thread to another. We regard such passing of control as events in the life of the object whose ownership is passed.

The semicolon denoting sequential composition is represented geometrically by a horizontal split in the diagram. The state of memory passed from \( p \) to \( q \) is defined by the labels on the edge which they share, in the manner that I have described earlier.

Note that it is geometrically impossible for any arrow to point upwards across a horizontal split from \( q \) to \( p \). If such a crossing were possible, the principle of causation would require the implementation to start executing the component \( q \) before finishing execution of \( p \). This would directly invalidate our original purpose, as given in the intuitive definition of semicolon.

In Euclidean plane geometry, it is impossible also for a horizontal arrow to cross a horizontal coordinate, which is intended to denote an atomic event. Thus the memory can never be in a state where some of the actions on the coordinate have already happened, and some of them have not. It is Euclid’s parallel postulate that forbids it.

This slide illustrates shows how diagrams formed by a previous split can be split again. The innermost operator of the program gives the orientation of the first split. Subsequent splits on the components may be either parallel or orthogonal to it. The process of splitting may be continued until all the diagrams contain only a single horizontal coordinate, with no dashed arrows.
That is how the previous diagram is distinguishable from the present diagram, in which the first split is horizontal, and the second and third splits are vertical. The sloping arrow in the middle of the slide now crosses a horizontal edge, whereas in the previous slide it crossed a vertical edge.

5. PROGRAM TESTING

**Purpose of testing**
- to locate errors in the program under test
  - particularly errors in the interfaces between large components
  - and those in recently changed code
- to assist in their diagnosis
  - and the search for a correction
  - or an acceptable work-around
- to record evidence for code reviews and analysis

**Faults**
- usually attributed as an error in implementation, e.g.,
  - An arrow that crosses from q to p in \((pq)\)
  - A horizontal arrow that crosses a horizontal coordinate
  - An error that should have prevented execution, e.g.,
    - a syntax error
    - a type mismatch
    - a command that is not in the programming language

Any realistic theory for program testing must be based on a study of the various kinds of error that might be detected in an execution of an arbitrary program. These may be classified in two ways: as errors in the program under test, which the programmer is capable of mending, or as errors which require mending by someone else. This slide lists includes errors which can only be due to a fault in the implementation itself. The first two points violate rules which we have quoted as part of the very definition of the programming language.
Errors in the program
usually attributed to a violation in the program
• A vertical split crossing a solid horizontal arrow
• violates ownership discipline
• An action that is explicitly forbidden
  by the language standard, e.g.
  • zero divide, null dereference, subscript overflow
• An assertion that evaluated to false
  • violates a promise made by the programmer

This slide describes generic programming errors that are explicitly ascribed to the program itself. They are the errors which program testing is aimed at detecting.

Other concurrency operators (CSP)
can be defined by forbidding different kinds of error:
• $P || q$ \quad \text{interleaving, embarrassing}
  forbids any arrow crossing between $p$ and $q$
• $p > > q$ \quad \text{chaining (or a pipe)}
  forbids any arrow crossing from $q$ to $p$
• $p \parallel q$ \quad \text{deadlock-free}
  forbids any arrows of a cyclic chain crossing between $p$ and $q$

This slide illustrates the ease of introducing new operators into our geometric model of program execution, by simply placing different restrictions on the kinds of arrows that cross between its operands. In this slide the first definition is the most restrictive, and the last is least restrictive. Thus the first two definitions also rule out deadlock.

Deadlock
• A causal cycle is physically impossible
• In program execution, it causes deadlock, attributed as a program error.
• It cannot be drawn with only downward arrows on the Euclidian plane
• but it can on a Moebius strip

The reason for forbidding deadlock is that each arrow, whether horizontal, vertical or sloping, denotes a causal connection. Thus none of the actions of the cycle can occur until they have all occurred. Except in the case of basic commands and fences, most languages do not permit the program to call for simultaneous execution of actions. The implementation is therefore permitted to stop execution on encountering a cyclic chain of dependencies.
The use of two-dimensional diagrams in software engineering has long been recommended. Our geometric diagrams are based on Message Sequence Charts, as incorporated in SDL and UML. UML offers several diagrammatic conventions for both data structure and control flow. SDL combines data and control in a single diagram, with nesting of diagrams to indicate structure. All of these diagrams are recommended for use in the design of the architecture of large-scale software products, long before the programs have been designed or coded, let alone tested.

The main suggestion of this talk is that the same geometry that has been used in architectural design should be used again by a program testing environment, to display traces of program execution, and then assist in the location, diagnosis and correction of errors that have occurred during the test.