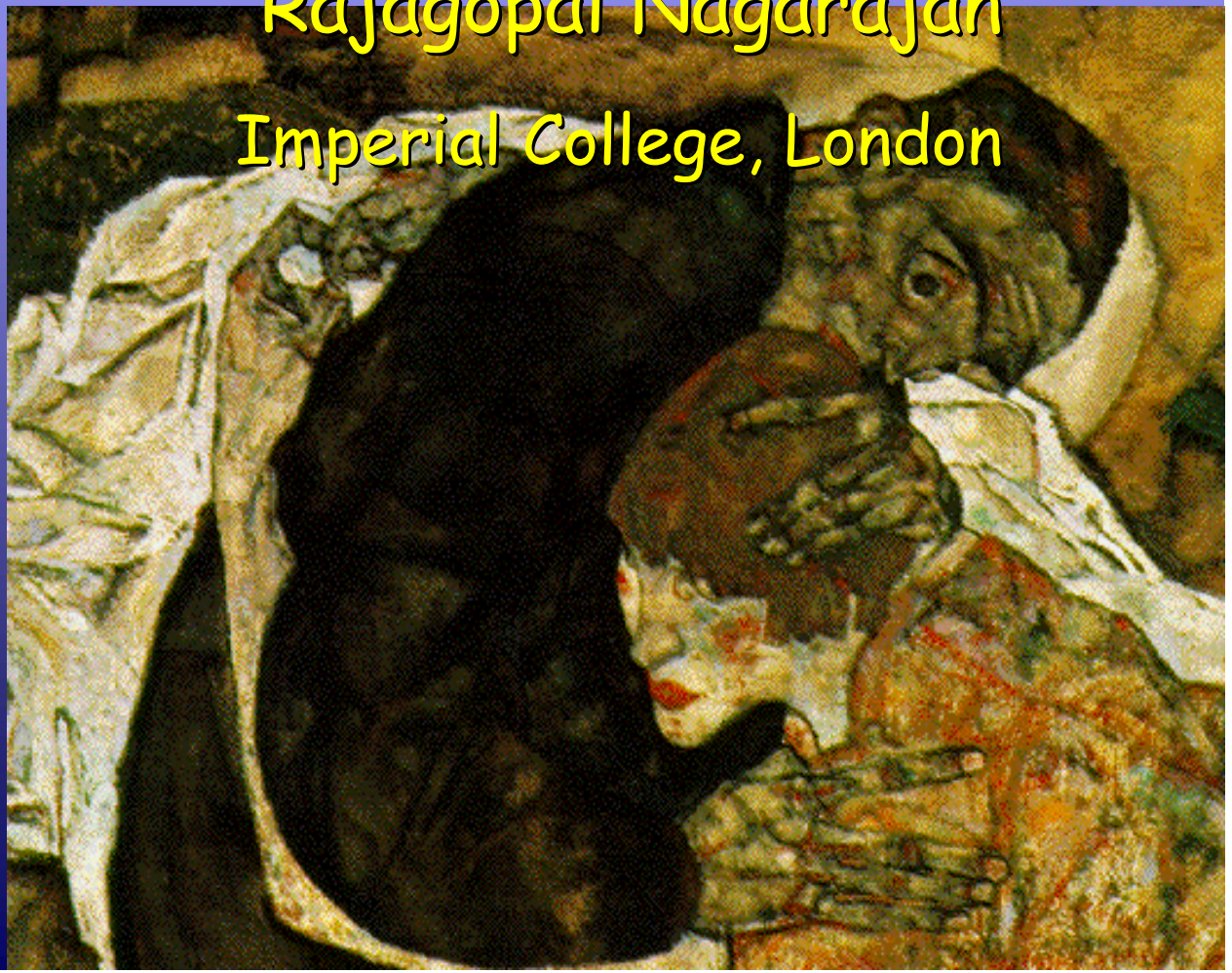
Semantics of Interaction

Rajagopal Nagarajan

Imperial College, London



26/02/99

Computation as interaction

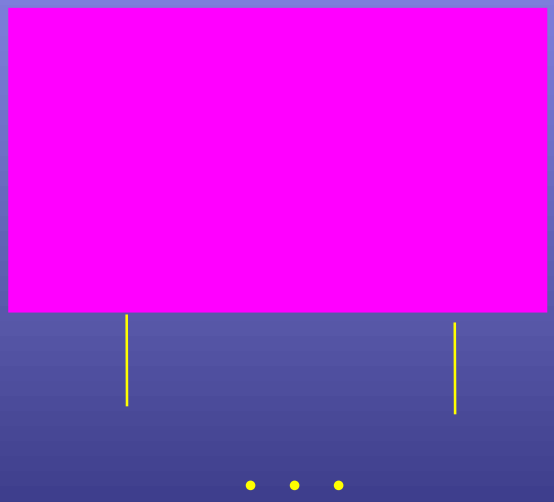
- Traditional mathematical models of computation view programs as computing functions or relations from inputs to outputs.

- This view is too restrictive and simplistic in current context of distributed, mobile, global computing.
- Processes or agents interacting with each other and information flows around the system.
- The aim is to describe such complex systems and their behaviour in a mathematical framework.

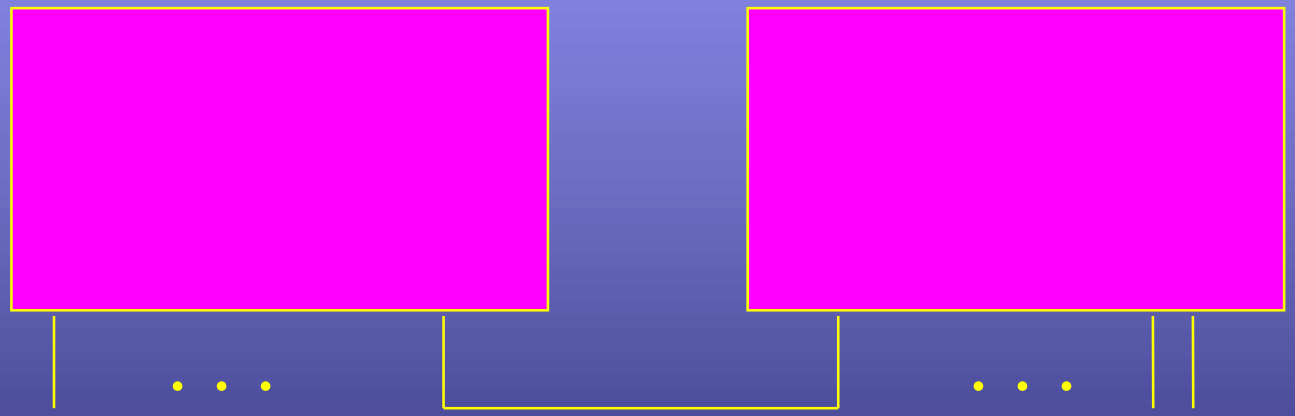
Structure of the talk

- Processes, interaction, types, semantics
- Game Semantics
- Games for Program Analysis
- Modelling Dataflow using Interaction Categories

- Processes or agents basic entities



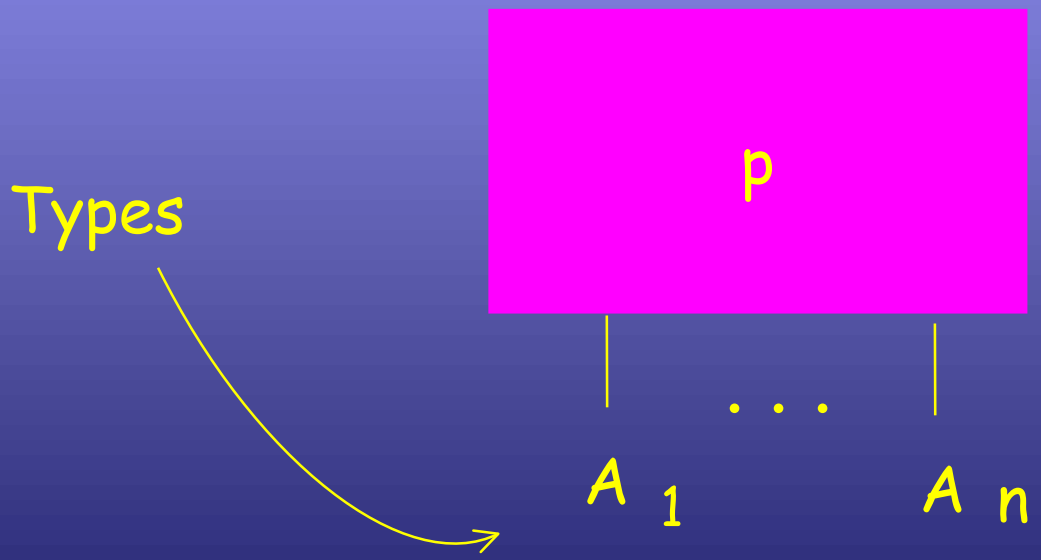
- The main operation is interaction which is achieved by connecting processes together



- Types are a valuable aid to the construction of correct programs.
- Compile-time type-checking or type-inference of particular importance.
- In sequential programming, type systems are used to express constraints on combinability of program modules.

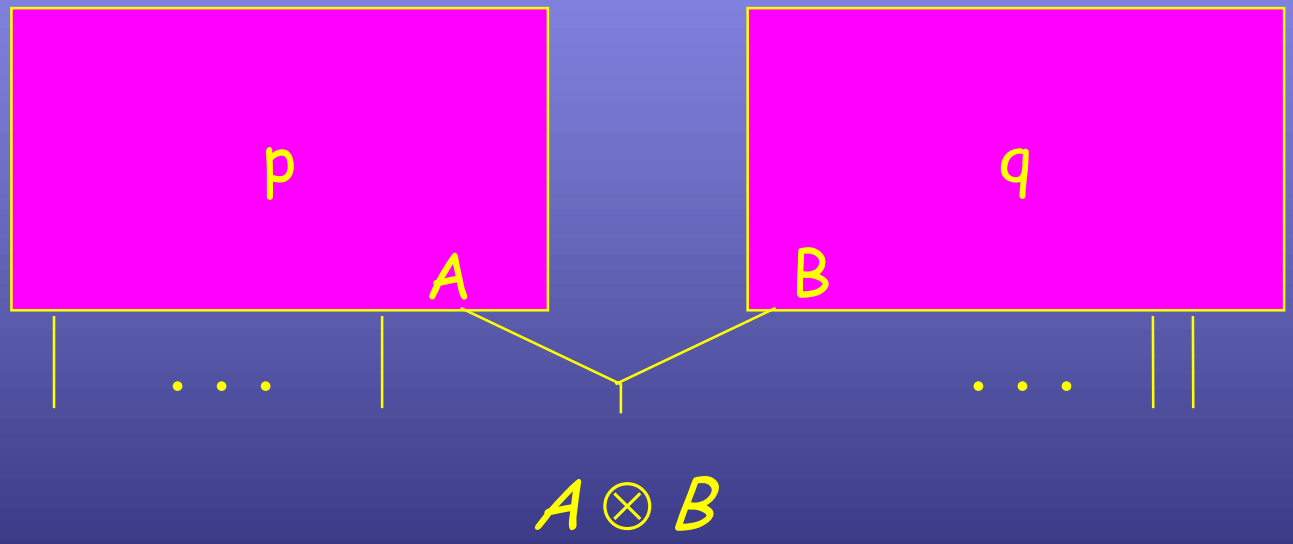
- In distributed systems, ensuring compatible combinations of subsystems is more complex.
- Interaction is in the form of prolonged pattern of communication rather than a procedure call.
- Types are harder to construct, but then benefits of the type system are much greater. Sophisticated properties of processes---for example, deadlock-freedom---can be addressed.

Types are used to describe the interface to a process

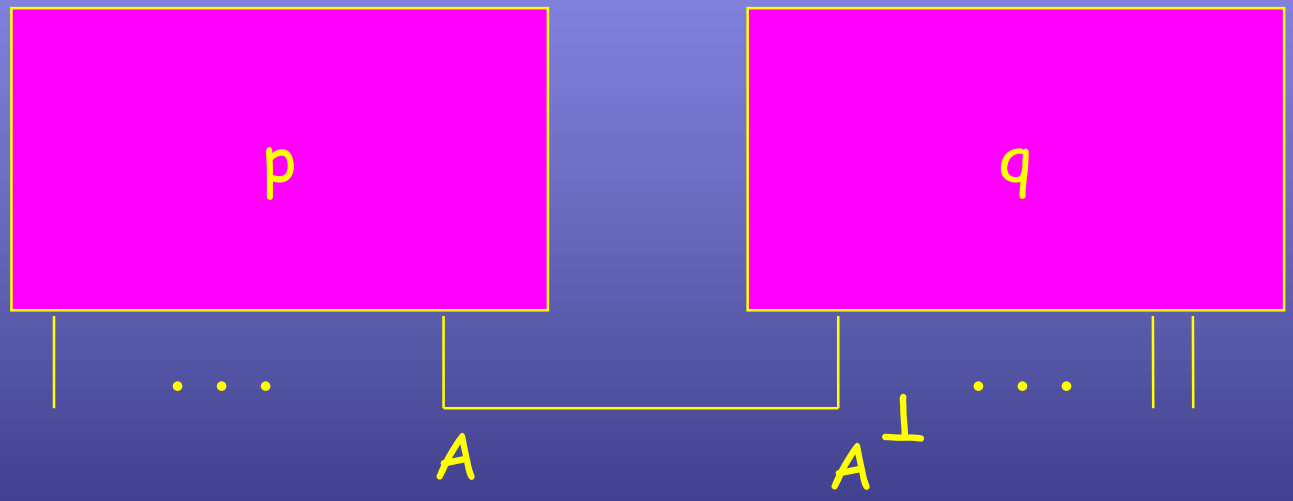


We use linear connectives to structure the interface of a process.

Tensor to combine ports of processes



We use linear negation to differentiate between input and output.



A wire or a process that acts as a buffer is as follows

- Semantics offers a sound mathematical basis for programming languages and software systems.
- A semantic framework with associated proof methods is vital for guaranteeing correctness of programs.
- Simulation or testing is not exhaustive.
- Using formal verification in addition to simulation increases the degree of confidence in software/hardware systems.

- Denotational semantics
  - "mathematical" semantics
  - models input/output behaviour
  - compositional
  - very successful in capturing functional programming

- Operational semantics.
  - "behavioural" semantics
  - describes computation steps
  - non-compositional in general
  - successful in capturing both functional and concurrent computation

Towards compositional operational semantics

- Game Semantics

  Fully abstract models and analyses of programming languages

- Interaction Categories

  Compositional type systems for concurrency

Game Semantics for Programming Languages

Abramsky, Jagadeesan, Malacaria, McCusker

Hyland, Ong, Nickau, Honda, Laird, Harmer

Games for Program Analysis

Hankin, Malacaria, Sampath

Interaction Categories

Abramsky, Gay, Nagarajan

- Game semantics models computation as playing a game.
- Two persons: Player (P) and Opponent (O).
- Player can be thought of as the system and opponent as the environment.
- In programming languages, system can be thought of as a term and environment as the context.

- Player and opponent make alternate moves.
- A type represents the kind of computation, hence types are represented by games.
- A program of type A determines how a system behaves, hence is represented by strategies for player.

A natural number game

$\mathbb{N}$

$\mathbb{Q}$ $\mathbb{O}$

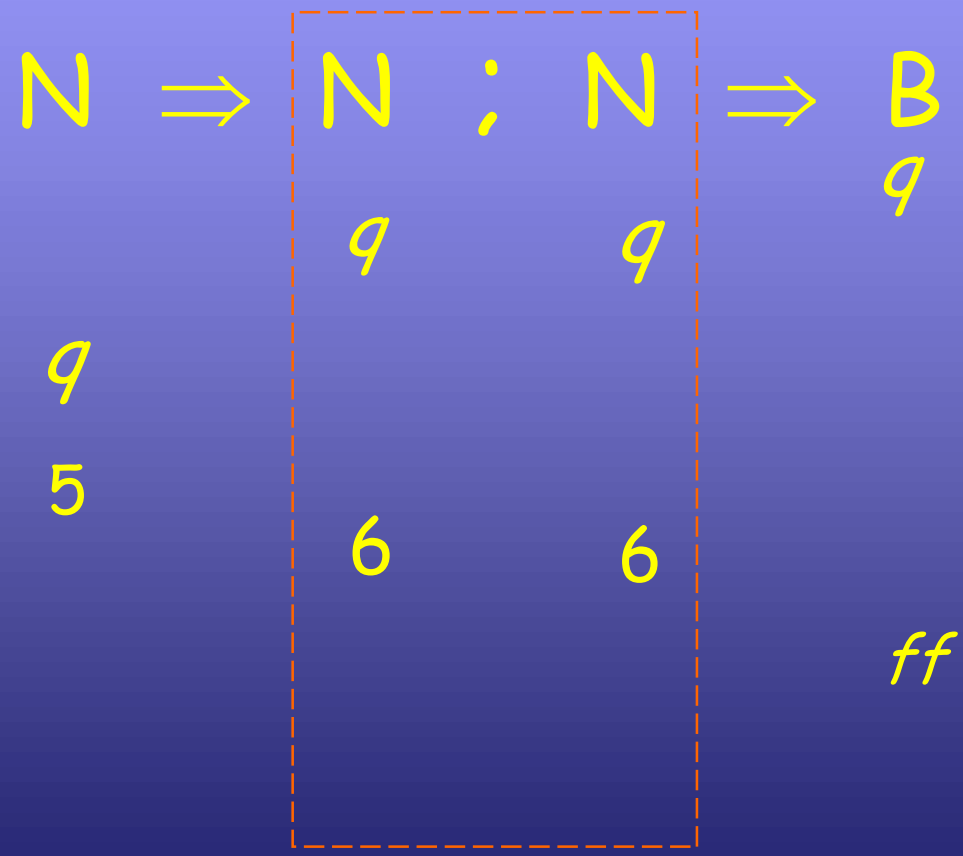
$\mathbb{Z}$ $\mathbb{P}$

Function type game

$$N \Rightarrow N$$

q	O
q	P
5	O
6	P

Composing strategies



Copycat strategy

$$N \Rightarrow N$$

q	O
q	P
n	O
n	P

- Program analysis involves compile-time analysis of run-time behaviour
- Control flow analysis for a functional language abstracts the substitutions of terms for variables that take place during the execution of a program.
- Games for control flow analysis
  - substitution is interaction
  - derived from semantics
  - analysis correct by construction

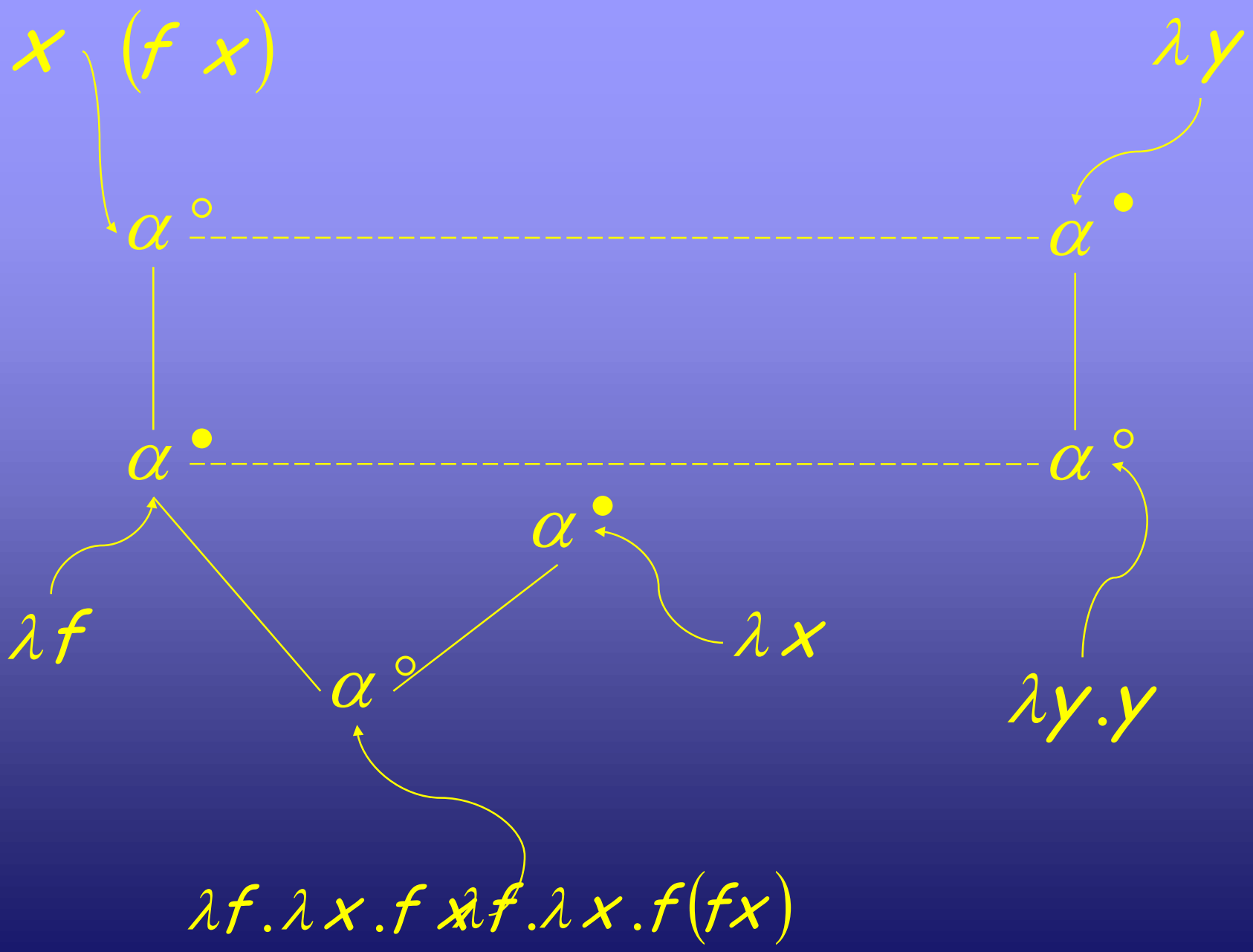
Example

$$\lambda f.\lambda x.\, f\ x:\ (\alpha \Rightarrow \alpha) \Rightarrow (\alpha \Rightarrow \alpha)$$

$$\lambda y.y : \alpha \Rightarrow \alpha$$

$$(\lambda f.\lambda x.\, f\ x)\ (\lambda y.y)$$

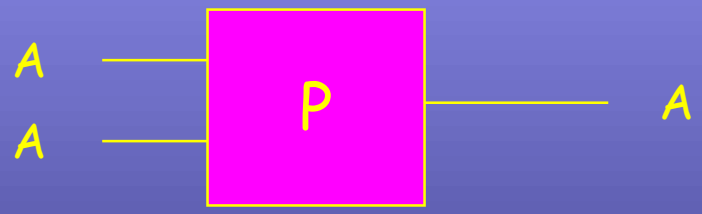
Opponent moves	λ terms
Player moves	vars of λ terms
Opponent move above a player move	arguments of a variable
Player move above an opponent move	all λs of term


x (f x)
λ y
α
α
α
α
α
λ f
λ x
λ y . y
λ f . λ x . f x
λ f . λ x . f (f x)

- Dataflow is a simple model of concurrency.
- A number of nodes connected together in a network.
- There are functions at each node performing computation.
- Operations are instantaneous and so are communications.
- There is an elegant semantic model, due to Kahn, using least fixpoints.

- Dataflow has been used as the underlying model of computation in areas such as signal processing and hardware specification.
- Synchronous dataflow is used widely in languages used for signal processing.
- Synchronous real-time languages such as SIGNAL and LUSTRE use dataflow models.
- The Ptolemy system developed at the University of California, Berkeley uses a synchronous model.

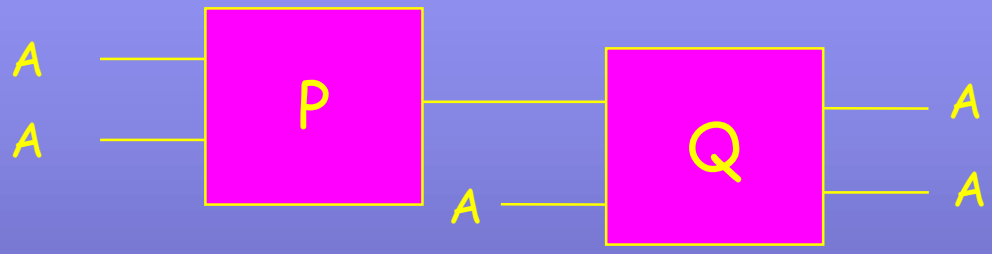
We now describe how to construct dataflow networks in a typed framework. Suppose we are working with networks in which datatypes come from some type A. A node



is modelled by a process (morphism)

$$P : A \otimes A \rightarrow A$$

Now suppose we have another node Q and connect the two together to form a simple network



The node Q is modelled by a process

$$Q : A \otimes A \rightarrow A \otimes A$$

and to model the entire network we form

$$(P \otimes id_A) ; Q : A \otimes A \otimes A \rightarrow A \otimes A$$

In our basic model, the types are tuples consisting of an alphabet of actions (labels) and a safety specification.

Combining ports produces tuples of actions. A process P of type

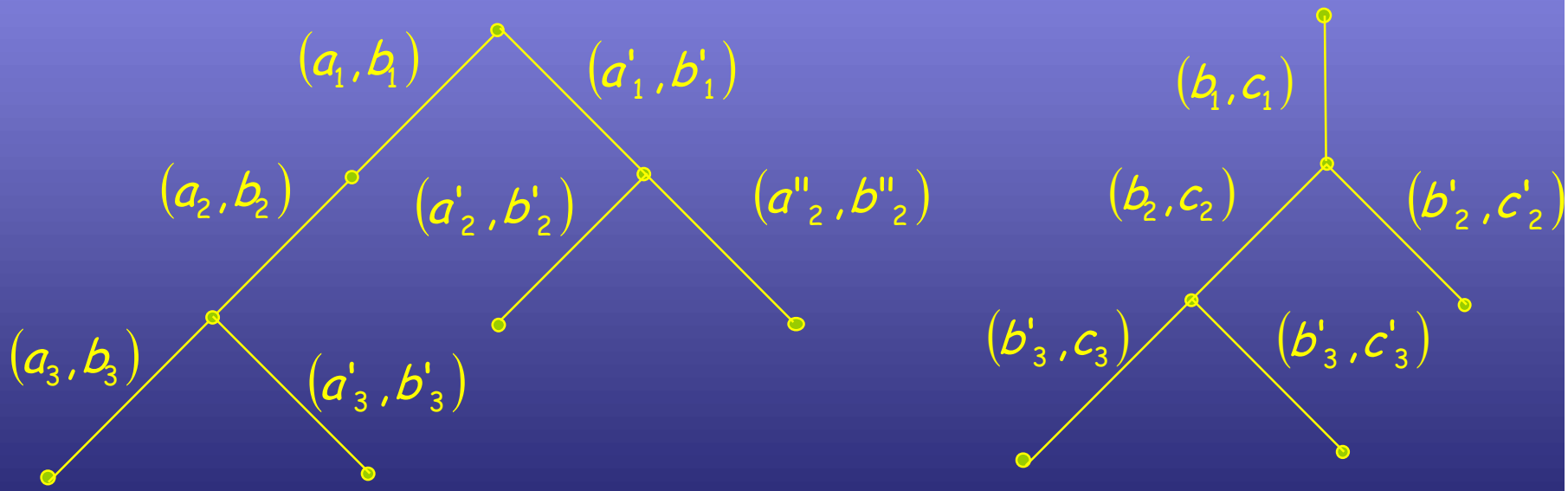
$$A_1 \otimes \cdots \otimes A_n$$

has actions of the form

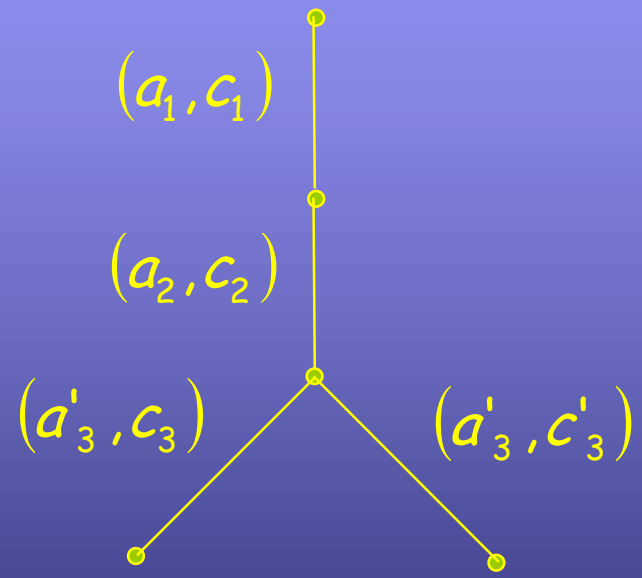
$$(a_1, \cdots, a_n)$$

- The type structure is very rudimentary, allowing us only to specify interconnection structure and safety properties.
- But it can be refined, for example, to include information about deadlock-freedom.

Processes can be represented by synchronization trees. For example, two processes P: A -> B and Q: B -> C can be as follows



- Their composition $P ; Q : A \rightarrow C$ is



- SIGNAL and LUSTRE are synchronous, real-time, dataflow languages used for signal processing and control applications.
- SIGNAL/LUSTRE operators are modelled as nodes of the dataflow network.
- Each operator given a type in the translation.
- Streams of data are translated into synchronization trees.

Conclusions

- Game semantics for modelling and analysing programming languages.
- As an example, we discussed using games to perform control flow analysis.
- Interaction Categories for providing a typed framework for concurrent computation.
- As an example, we discussed using interaction categories to model dataflow.

- Techniques from game semantics used to solve an important open problem in theoretical computer science---the full abstraction of PCF.
- Other fully abstract models include features such as imperative constructs, control, general references, and finite non-determinism.
- Games have been used to perform control flow analysis, dataflow analysis, and to discuss secure information flows.

- Interaction Categories have been used to develop type systems for concurrency.
- Benefits of this approach include type checking, verification and semantic proof methods.
- Our framework is not restricted to dataflow. We have already developed typed process calculi based on the semantics.
- We have compositional methods for the verification of deadlock-freedom.

For further information

http://theory.doc.ic.ac.uk/~raja/tcool.html