Measurable Preorders and Complexity

Thomas Seiller[1]
Proofs, Programs, Systems – Paris 7 University

seiller@ihes.fr

TACL 2015
June 26th, 2015

[1]Partially supported by ANR-12-JS02-006-01 COQUAS.

Contents of the talk

Present a new approach to (implicit) computational complexity theory.

Slogan

There is a correspondence between complexity constraints and algebras.

Contents of the talk

Present a new approach to (implicit) computational complexity theory.

Slogan

There is a correspondence between complexity constraints and (some kinds of) algebras (modulo isomorphisms).

Contents of the talk

Present a new approach to (implicit) computational complexity theory.

Slogan

There is a correspondence between complexity constraints and (some kinds of) algebras (modulo isomorphisms).

- Two important motivations:
  - obtain a uniform mathematical approach to CT
  - gain new proof methods from mathematics

Contents of the talk

Present a new approach to (implicit) computational complexity theory.

Slogan

There is a correspondence between complexity constraints and (some kinds of) algebras (modulo isomorphisms).

- Two important motivations:
  - obtain a uniform mathematical approach to CT
  - gain new proof methods from mathematics
- At the intersection between two lines of work:
  - Implicit Computational Complexity, especially approaches using linear logic.
  - Interaction Graphs. A quantitative version of Girard's Geometry of Interaction;

Proofs as Programs – Curry-Howard – correspondence

Proof Theory	Computer Science
Proof	Program
Proof	Data
Cut rule	Application
Cut Elimination	Execution (Computation)
Formulas	Types
...	...

Proofs as Programs – Curry-Howard – correspondence

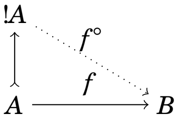
- Integers: $\text{nat} := \forall X \ \ (X \to X) \to (X \to X)$
- Functions from integers to integers: $\text{nat} \to \text{nat}$
- If $[n]$ is a (cut-free) proof of nat, and $[f]$ a proof of $\text{nat} \to \text{nat}$, we can define the proof $[f][n]$:

$$\dfrac{\begin{array}{c}[f]\\ \vdots \\ \text{nat} \vdash \text{nat}\end{array} \qquad \begin{array}{c}[n]\\ \vdots \\ \vdash \text{nat}\end{array}}{\vdash \text{nat}}\ \text{cut}$$

- The cut elimination procedure applied to $[f][n]$ corresponds (step by step) to the computation of $f(n)$. The cut-free proof it produces is equal to $[f(n)]$.

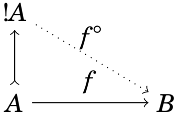
Linear Logic and Implicit Computational Complexity

- **Normal functors (Girard).** A model of programs as functors, in which functors representing programs are analytical, i.e. can be factored through a linear functor on a kind of tensor algebra $!A := \bigoplus_{k \leqslant 0} A \otimes A \otimes \cdots \otimes A$



Linear Logic and Implicit Computational Complexity

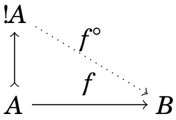
- **Normal functors (Girard).** A model of programs as functors, in which functors representing programs are analytical, i.e. can be factored through a linear functor on a kind of tensor algebra $!A := \bigoplus_{k \leqslant 0} A \otimes A \otimes \cdots \otimes A$



- **Linear logic** is obtained by just "pulling back" this decomposition into the syntax, i.e. the usual implication $A \Rightarrow B$ becomes $!A \multimap B$;

Linear Logic and Implicit Computational Complexity

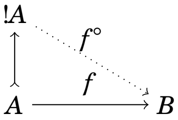
- **Normal functors (Girard).** A model of programs as functors, in which functors representing programs are analytical, i.e. can be factored through a linear functor on a kind of tensor algebra $!A := \bigoplus_{k \leqslant 0} A \otimes A \otimes \cdots \otimes A$



- **Linear logic** is obtained by just "pulling back" this decomposition into the syntax, i.e. the usual implication $A \Rightarrow B$ becomes $!A \multimap B$;
- **LL for Complexity.** The rules governing the modality ! can be modified to define sub-systems characterising complexity classes. E.g. Elementary Linear Logic (ELL), Light Linear Logic (LLL).

Linear Logic and Implicit Computational Complexity

- **Normal functors (Girard).** A model of programs as functors, in which functors representing programs are analytical, i.e. can be factored through a linear functor on a kind of tensor algebra $!A := \bigoplus_{k \leqslant 0} A \otimes A \otimes \cdots \otimes A$

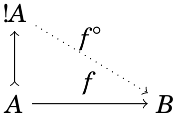


- **Linear logic** is obtained by just "pulling back" this decomposition into the syntax, i.e. the usual implication $A \Rightarrow B$ becomes $!A \multimap B$;
- **LL for Complexity.** The rules governing the modality ! can be modified to define sub-systems characterising complexity classes. E.g. Elementary Linear Logic (ELL), Light Linear Logic (LLL).

Proofs of !nat $\multimap$ nat in ELL are exactly the functions computable in elementary time.

Linear Logic and Implicit Computational Complexity

- **Normal functors (Girard).** A model of programs as functors, in which functors representing programs are analytical, i.e. can be factored through a linear functor on a kind of tensor algebra $!A := \bigoplus_{k\leqslant 0} A\otimes A\otimes\cdots\otimes A$



- **Linear logic** is obtained by just “pulling back” this decomposition into the syntax, i.e. the usual implication $A \Rightarrow B$ becomes $!A \multimap B$;
- **LL for Complexity.** The rules governing the modality ! can be modified to define sub-systems characterising complexity classes. E.g. Elementary Linear Logic (ELL), Light Linear Logic (LLL).

Proofs of $!\mathrm{nat} \multimap \mathrm{nat}$ in ELL are exactly the functions computable in elementary time.

Proofs of $!\mathrm{nat} \multimap \mathrm{nat}$ in LLL are exactly the functions computable in polynomial time.

Interaction Graphs

It is a model of **programs/proofs** and their **dynamics**.

Interaction Graphs

It is a model of **programs/proofs** and their **dynamics**.

Logic	CS	IG
Proof	Program	Graph
Proof	Data	Graph

Interaction Graphs

It is a model of **programs/proofs** and their **dynamics**.

Logic	CS	IG
Proof	Program	Graph
Proof	Data	Graph
Cut Rule	Application	Common Vertices

Interaction Graphs

It is a model of **programs/proofs** and their **dynamics**.

Logic	CS	IG
Proof	Program	Graph
Proof	Data	Graph
Cut Rule	Application	Common Vertices
Cut elim.	Computation	(Alt.) Paths

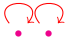
Interaction Graphs

It is a model of **programs/proofs** and their **dynamics**.

Logic	CS	IG
Proof $\pi \vdash \textbf{Nat} \Rightarrow \textbf{Nat}$	Program $f : \mathsf{nat} \to \mathsf{nat}$	Graph 
Proof	Data	Graph
Cut Rule	Application	Common Vertices
Cut elim.	Computation	(Alt.) Paths

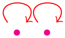
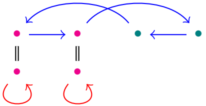
Interaction Graphs

It is a model of **programs/proofs** and their **dynamics**.

Logic	CS	IG
Proof $\pi \vdash \mathbf{Nat} \Rightarrow \mathbf{Nat}$	Program $f : \mathsf{nat} \to \mathsf{nat}$	Graph 
Proof $\rho \vdash \mathbf{Nat}$	Data $n : \mathsf{nat}$	Graph 
Cut Rule	Application	Common Vertices
Cut elim.	Computation	(Alt.) Paths

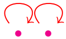
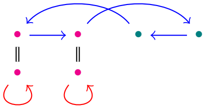
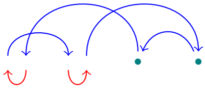
Interaction Graphs

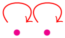
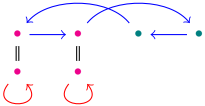
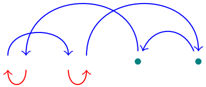
It is a model of **programs/proofs** and their **dynamics**.

Logic	CS	IG
Proof $\pi \vdash \textbf{Nat} \Rightarrow \textbf{Nat}$	Program $f : \mathsf{nat} \rightarrow \mathsf{nat}$	Graph 
Proof $\rho \vdash \textbf{Nat}$	Data $n : \mathsf{nat}$	Graph 
Cut Rule $\text{cut}(\pi, \rho) \vdash \textbf{Nat}$	Application $f(n)$	Common Vertices 
Cut elim.	Computation	(Alt.) Paths

Interaction Graphs

It is a model of **programs/proofs** and their **dynamics**.

Logic	CS	IG
Proof $\pi \vdash \mathbf{Nat} \Rightarrow \mathbf{Nat}$	Program $f : \mathsf{nat} \to \mathsf{nat}$	Graph 
Proof $\rho \vdash \mathbf{Nat}$	Data $n : \mathsf{nat}$	Graph 
Cut Rule $\mathrm{cut}(\pi, \rho) \vdash \mathbf{Nat}$	Application $f(n)$	Common Vertices 
Cut elim. $\mathrm{cut}(\pi, \rho) \rightsquigarrow \mu \vdash \mathbf{Nat}$	Computation $f(n) \rightsquigarrow m : \mathsf{nat}$	(Alt.) Paths 

Logic	CS	IG
Proof $\pi \vdash$ **Nat** $\Rightarrow$ **Nat**	Program f : nat $\to$ nat	Graphing 
Proof $\rho \vdash$ **Nat**	Data n : nat	Graphing 
Cut Rule cut$(\pi, \rho) \vdash$ **Nat**	Application $f(n)$	Common “Vertices” 
Cut elim. cut$(\pi, \rho) \rightsquigarrow \mu \vdash$ **Nat**	Computation $f(n) \rightsquigarrow m$: nat	(Alt.) Paths 

This yields models of (fragments of) Linear Logic using realizability techniques.

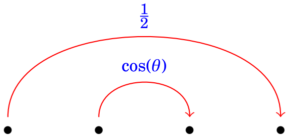
Basics

Basic hypothesis: Programs as graphings. So... what's a graphing?

Basics

Basic hypothesis: Programs as graphings. So... what's a graphing?

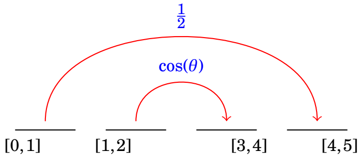
- Pick a directed graph, add weights (from a monoid Ω) on the edges.



Basics

Basic hypothesis: Programs as graphings. So... what's a graphing?

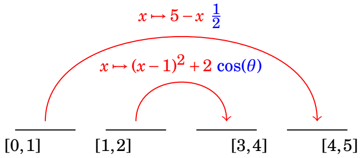
- Pick a directed graph, add weights (from a monoid Ω) on the edges.
- Consider that vertices are measurable sets, e.g. intervals.



Basics

Basic hypothesis: Programs as graphings. So... what's a graphing?

- Pick a directed graph, add weights (from a monoid Ω) on the edges.
- Consider that vertices are measurable sets, e.g. intervals.
- Decide *how* the edges map sources to targets.



Basics

Basic hypothesis: Programs as graphings. So... what's a graphing?

- Pick a directed graph, add weights (from a monoid Ω) on the edges.
- Consider that vertices are measurable sets, e.g. intervals.
- Decide *how* (i.e. which element of $\mathfrak{m}$) the edges map sources to targets.

The parameters of the interpretations:

- A measure space $(X, \mathscr{B}, \mu)$;
- A monoid Ω;
- A monoid $\mathfrak{m}$ of measurable maps $X \to X$ – called a **microcosm**;
- A **type** of graphing (e.g. deterministic, probabilistic);
- A measurable map $m : \Omega \to \mathbf{R}_{\geqslant 0} \cup \{\infty\}$.

Basics

Basic hypothesis: Programs as graphings. So... what's a graphing?

- Pick a directed graph, add weights (from a monoid Ω) on the edges.
- Consider that vertices are measurable sets, e.g. intervals.
- Decide *how* (i.e. which element of $\mathfrak{m}$) the edges map sources to targets.

The parameters of the interpretations:

- A measure space $(X, \mathscr{B}, \mu)$;
- A monoid Ω;
- A monoid $\mathfrak{m}$ of measurable maps $X \rightarrow X$ – called a **microcosm**;
- A **type** of graphing (e.g. deterministic, probabilistic);
- A measurable map $m : \Omega \rightarrow \mathbf{R}_{\geqslant 0} \cup \{\infty\}$.

Basics

Basic hypothesis: Programs as graphings. So... what's a graphing?

- Pick a directed graph, add weights (from a monoid Ω) on the edges.
- Consider that vertices are measurable sets, e.g. intervals.
- Decide *how* (i.e. which element of $\mathfrak{m}$) the edges map sources to targets.

The parameters of the interpretations:

- A measure space $(X, \mathscr{B}, \mu)$;
- A monoid Ω;
- A monoid $\mathfrak{m}$ of measurable maps $X \to X$ – called a **microcosm**;
- A **type** of graphing (e.g. deterministic, probabilistic);
- A measurable map $m : \Omega \to \mathbf{R}_{\geqslant 0} \cup \{\infty\}$.

A Complexity-through-Realizability Theory

- **Principle.** Characterise complexity classes by the type $\mathbf{Words}^{(2)}_{\{0,1\}}$ of predicates over binary words in a given model (this type always exists).

A Complexity-through-Realizability Theory

- **Principle.** Characterise complexity classes by the type $\mathbf{Words}^{(2)}_{\{0,1\}}$ of predicates over binary words in a given model (this type always exists).
- Deterministic case. Consider an element A of the type $!\mathbf{Words}^{(2)}_{\{0,1\}} \multimap \mathbf{Bool}$.
  - Then A defines a language defined as

$$\{\mathfrak{w} \mid A :: [\mathfrak{w}] = \mathtt{true}\}$$

  - The above type thus defines a complexity class.

A Complexity-through-Realizability Theory

- **Principle.** Characterise complexity classes by the type $\mathbf{Words}_{\{0,1\}}^{(2)}$ of predicates over binary words in a given model (this type always exists).
- Deterministic case. Consider an element A of the type $!\mathbf{Words}_{\{0,1\}}^{(2)} \multimap \mathbf{Bool}$.
  - Then A defines a language defined as
  $$\{\mathfrak{w} \mid A :: [\mathfrak{w}] = \mathtt{true}\}$$
  - The above type thus defines a complexity class.
- General case. Consider an element A of the type $!\mathbf{Words}_{\{0,1\}}^{(2)} \multimap \mathbf{NBool}$.
  - Define a notion of *test* T (elements of the model);
  - A defines a language w.r.t. T
  $$\{\mathfrak{w} \mid A :: [\mathfrak{w}] \perp\!\!\!\perp T\}$$
  where $\perp\!\!\!\perp$ is an "orthogonality relation" used to define types.
  - The above type then defines a complexity class w.r.t. T.

Summary of Results

We can define microcosms

$$\mathfrak{m}_1 \subset \mathfrak{m}_2 \subset \cdots \subset \mathfrak{m}_\infty \subset \mathfrak{n} \subset \mathfrak{p}$$

in order to obtain the following characterisations.

Microcosm	$\mathbb{M}_m^{\text{det}}$	$\mathbb{M}_m^{\text{ndet}}$	$\mathbb{M}_m^{\text{ndet}}$	$\mathbb{M}_m^{\text{prob}}$	Logic	Machines
$\mathfrak{m}_1$	REG	REG	REG	STOC	MALL	2-way Automata (2FA)
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\mathfrak{m}_k$	D_k	N_k	coN_k	P_k	$(\dots)$	k-heads 2FA
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\mathfrak{m}_\infty$	L	NL	coNL	PL	$(\dots)$	multihead-head 2FA (2MHFA)
$\mathfrak{n}$	P	P	P	P?	$(\dots)$	2MHFA + Pushdown Stack
$\mathfrak{p}$	P	NP	coNP	PP?	ELL	Ptime Turing Machines

Conjecture: **microcosms** correspond to **complexity constraints**.

The conjecture, formally

- We define an equivalence relation on microcosms.
- Notation: we pick a type of graphings (e.g. probabilistic) and a test, and write $\texttt{Pred}(\mathfrak{m})$ the set of languages accepted by elements of $!\mathbf{Words}_{\{0,1\}}^{(2)} \multimap \mathbf{NBool}$ w.r.t. the chosen test.

Theorem

If $\mathfrak{m} \equiv \mathfrak{n}$*, then* $\texttt{Pred}(\mathfrak{m}) = \texttt{Pred}(\mathfrak{n})$.

Conjecture

The converse holds, i.e. $\texttt{Pred}(\mathfrak{m}) = \texttt{Pred}(\mathfrak{n})$ *implies* $\mathfrak{m} \equiv \mathfrak{n}$.

If this conjecture holds, it would provide new proof techniques for separation through (co)homological invariants, e.g. $\ell^{(2)}$-Betti numbers:

$$\texttt{Pred}(\mathfrak{m}) = \texttt{Pred}(\mathfrak{n}) \Rightarrow \mathfrak{m} \equiv \mathfrak{n} \Rightarrow \mathscr{P}(\mathfrak{m}) \simeq \mathscr{P}(\mathfrak{n}) \overset{!}{\Rightarrow} \ell^{(2)}(\mathscr{P}(\mathfrak{m})) = \ell^{(2)}(\mathscr{P}(\mathfrak{m}))$$

where $\mathscr{P}(\mathfrak{m}) = \{(x,y) \mid \exists m \in \mathfrak{m}, m(x) = y\}$ is a "measurable preorder".

Conclusion

- A good "working hypothesis":
  - homogeneous approach of complexity theory (CT) (diff. computational paradigms, higher-order functions)
  - inherits the advantages of logic-based approaches to CT, e.g. machine-independent, possibility of computing complexity bounds statically
- Not a miraculous technique for separation results:
  - "Borel Equivalence Relations" are well-studied (ergodic theory, descriptive set theory), however measurable preorders are not (in particular, no $\ell^{(2)}$-Betti numbers in this case);
  - Need to characterise complexity classes in the right way;
  - **However**, it is not naturally seen as a "natural proof".

In a nutshell

The purpose is to relate two problems:

- **Mathematics.** Are two spaces X,Y homotopy equivalent?
  - Difficult to answer negatively;
- **C.S.** Are two complexity classes A,B equal?
  - Difficult to answer negatively;

In a nutshell

The purpose is to relate two problems:

- **Mathematics.** Are two spaces X, Y homotopy equivalent?
  - Difficult to answer negatively;
  - Some proof methods available (e.g. (co)homological invariants).
- **C.S.** Are two complexity classes A, B equal?
  - Difficult to answer negatively;

In a nutshell

The purpose is to relate two problems:

- **Mathematics.** Are two spaces X, Y homotopy equivalent?
  - Difficult to answer negatively;
  - Some proof methods available (e.g. (co)homological invariants).
- **C.S.** Are two complexity classes A, B equal?
  - Difficult to answer negatively;
  - No proof methods available, c.f. *Natural Proofs*.