From normal functors to logarithmic space queries

Nguyễn Lê Thành Dũng (LIPN, U. Paris 13) — nltd@nguyentito.eu
j.w.w. Pierre Pradic (LIP, ENS Lyon & MIMUW, U. Warsaw)

ICALP 2019 (Track B), Patras

The title refers to:

- categorical semantics (*normal functors*)
- computation over finite structures and complexity (*logarithmic space queries*)

So what's the connection?

The title refers to:

- categorical semantics (*normal functors*)
- computation over finite structures and complexity (*logarithmic space queries*)

So what's the connection?

- We use *linear logic* to do *implicit computational complexity*, i.e. machine-free characterization of complexity classes (ICC vs descriptive complexity $\approx$ functional vs declarative)
- Semantics is involved in our upper bounds proofs

Roughly, *linearity* $\approx$ no-copy / single use restriction

Linear logic as a logic

Linear logic (Girard 1987): logic with explicit duplication

Classical logic	Linear logic
for all A, $\vdash A \to A \wedge A$	in general, $\nvdash A \multimap A \otimes A$

Linear logic as a logic

Linear logic (Girard 1987): logic with explicit duplication

Classical logic	Linear logic
for all A, $\vdash A \to A \wedge A$	in general, $\not\vdash A \multimap A \otimes A$
	when $A = !B$, $\vdash !B \multimap !B \otimes !B$
	$\approx$ "as many Bs as I want"

$\longrightarrow$ *Reusable assumptions* marked by $!(-)$.

Linear logic as a logic

Linear logic (Girard 1987): logic with explicit duplication

Classical logic	Linear logic
for all A, $\vdash A \to A \wedge A$	in general, $\nvdash A \multimap A \otimes A$
	when $A = !B$, $\vdash !B \multimap !B \otimes !B$
	$\approx$ "as many Bs as I want"

$\longrightarrow$ *Reusable assumptions* marked by $!(-)$.

Why do people care about such restrictions?

- Linear logic is constructive:
  proof of $A \oplus B \cong$ proof of A or proof of B

Linear logic as a logic

Linear logic (Girard 1987): logic with explicit duplication

Classical logic	Linear logic
for all A, $\vdash A \to A \wedge A$	in general, $\nvdash A \multimap A \otimes A$
	when $A = !B$, $\vdash !B \multimap !B \otimes !B$
	$\approx$ "as many Bs as I want"

$\longrightarrow$ *Reusable assumptions* marked by $!(-)$.

Why do people care about such restrictions?

- Linear logic is constructive:
  proof of $A \oplus B \cong$ proof of A or proof of B
- $\Longrightarrow$ Can be seen as a *functional programming language*,
  via the *Curry–Howard correspondence*

Proofs as programs

The Curry–Howard correspondence

Constructive logics	vs	programming languages:
formulæ	$\leftrightsquigarrow$	types
proofs	$\leftrightsquigarrow$	programs

Proofs as programs

The Curry–Howard correspondence

Constructive logics	vs	programming languages:
formulæ	$\leftrightsquigarrow$	types
proofs	$\leftrightsquigarrow$	programs

For example:

Modus ponens

$$\frac{A \to B \qquad A}{B}$$

Proofs as programs

The Curry–Howard correspondence

Constructive logics	vs	programming languages:
formulæ	$\leftrightsquigarrow$	types
proofs	$\leftrightsquigarrow$	programs

For example:

Modus ponens $\leftrightsquigarrow$ Function application

$$\frac{A \to B \qquad A}{B} \quad \leftrightsquigarrow \quad \frac{f : A \to B \qquad x : A}{f\,x : B}$$

Proofs as programs

The Curry–Howard correspondence

Constructive logics	vs	programming languages:
formulæ	$\leftrightsquigarrow$	types
proofs	$\leftrightsquigarrow$	programs

For example (many variants, depending on the logic used):

Modus ponens	$\leftrightsquigarrow$	Function application
$\dfrac{A \to B \qquad A}{B}$	$\leftrightsquigarrow$	$\dfrac{f : A \to B \qquad x : A}{f\,x : B}$
Intuitionistic logic	$\leftrightsquigarrow$	Typed λ-calculus

theoretical basis for OCaml, Haskell, etc.

Proofs as programs

The Curry–Howard correspondence

Constructive logics	vs	programming languages:
formulæ	$\leftrightsquigarrow$	types
proofs	$\leftrightsquigarrow$	programs
normalization	$\leftrightsquigarrow$	computation

For example (many variants, depending on the logic used):

Modus ponens $\leftrightsquigarrow$ Function application

$$\frac{A \to B \qquad A}{B} \quad \leftrightsquigarrow \quad \frac{f : A \to B \qquad x : A}{f\,x : B}$$

Intuitionistic logic $\leftrightsquigarrow$ Typed λ-calculus

theoretical basis for OCaml, Haskell, etc.

Linear logic as a programming language

One motivation for inventing linear logic:
analyze the computational content (i.e. normalization) of intuitionistic proofs.

Idea

Duplication is responsible for the complexity of computation, so make it explicit using $!(-)$.

$$\begin{array}{rcl} \text{non-linear function} & \cong & \text{linear function + duplicable input} \\ A \to B & \cong & !A \multimap B \end{array}$$

Linear logic as a programming language

One motivation for inventing linear logic:
analyze the computational content (i.e. normalization) of intuitionistic proofs.

Idea

Duplication is responsible for the complexity of computation, so make it explicit using $!(-)$.

$$\begin{array}{rcl} \text{non-linear function} & \cong & \text{linear function + duplicable input} \\ A \to B & \cong & !A \multimap B \end{array}$$

Idea 2

Restricting $!(-)$ can induce complexity effects
$\longrightarrow$ Starting point for implicit complexity in linear logic

Implicit computational complexity in linear logic (1)

In linear logic, $!(-)$ satisfies not just $!A \multimap !A \otimes !A$, but also:

$$!A \multimap A \qquad !A \multimap !!A \qquad \text{(i.e. } !(-) \text{ is a comonad)}$$

In *Elementary Linear Logic* (ELL), these rules are *removed*;
ensures that !-nesting is somehow an invariant of computation.
(LL vs ELL $\approx$ modal logic S4 vs K)

Implicit computational complexity in linear logic (1)

In linear logic, $!(-)$ satisfies not just $!A \multimap !A \otimes !A$, but also:

$$!A \multimap A \qquad !A \multimap !!A \qquad \text{(i.e. } !(-) \text{ is a comonad)}$$

In *Elementary Linear Logic* (ELL), these rules are *removed*; ensures that !-nesting is somehow an invariant of computation.
(LL vs ELL $\approx$ modal logic S4 vs K)

Theorem (Girard 1998 / Danos and Joinet 2003)

The class of languages decided by programs of type $!\mathsf{Str} \multimap !^k\mathsf{Bool}$ *in second-order ELL (for varying* $k \in \mathbb{N}$*) is* ELEMENTARY.

Implicit computational complexity in linear logic (2)

Theorem (Girard 1998 / Danos and Joinet 2003)

The class of languages decided by programs of type $!\mathsf{Str} \multimap !^k\mathsf{Bool}$ *in second-order ELL (for varying* $k \in \mathbb{N}$*) is* ELEMENTARY.

This holds for any "reasonable" encoding Str of strings.

Recent work (2010s) by Baillot et al.:
characterize smaller classes (P, EXPTIME...) in variants of ELL by taking fixed values of $k \in \mathbb{N}$.

Implicit computational complexity in linear logic (2)

Theorem (Girard 1998 / Danos and Joinet 2003)

The class of languages decided by programs of type $!\mathsf{Str} \multimap !^k\mathsf{Bool}$ *in second-order ELL (for varying* $k \in \mathbb{N}$*) is* ELEMENTARY.

This holds for any "reasonable" encoding Str of strings.

Recent work (2010s) by Baillot et al.:
characterize smaller classes (P, EXPTIME...) in *variants of ELL*
by taking fixed values of $k \in \mathbb{N}$.

What happens for fixed k in ELL itself?

Implicit computational complexity in linear logic (2)

Theorem (Girard 1998 / Danos and Joinet 2003)

The class of languages decided by programs of type $!\mathsf{Str} \multimap !^k\mathsf{Bool}$ *in second-order ELL (for varying* $k \in \mathbb{N}$*) is* ELEMENTARY.

This holds for any "reasonable" encoding Str of strings.

Recent work (2010s) by Baillot et al.:
characterize smaller classes (P, EXPTIME...) in *variants of ELL* by taking fixed values of $k \in \mathbb{N}$.

What happens for fixed k in ELL itself? Some issues arise:

- Sensitivity to *input representation*
- Previous proof techniques not sufficient

ICC in LL meets finite models

Our paper: ELL-based implicit complexity
+ inspiration from Hillebrand's PhD thesis (1994),
Finite Model Theory in the Simply Typed λ-Calculus

We define a type Inp_Σ of inputs as *finite structures*, and look at queries expressed by programs of type $\mathsf{Inp}_\Sigma \multimap !!\mathsf{Bool}$.

(Σ: relational vocabulary)

ICC in LL meets finite models

Our paper: ELL-based implicit complexity
+ inspiration from Hillebrand's PhD thesis (1994),
Finite Model Theory in the Simply Typed λ-Calculus

We define a type Inp_Σ of inputs as *finite structures*, and look at queries expressed by programs of type $\mathsf{Inp}_\Sigma \multimap !!\mathsf{Bool}$.
(Σ: relational vocabulary)

- Good news: it seems that we get exactly *deterministic logarithmic space* (L) queries
  (also: much cleaner than previous characterizations of L in LL)

ICC in LL meets finite models

Our paper: ELL-based implicit complexity
+ inspiration from Hillebrand's PhD thesis (1994),
Finite Model Theory in the Simply Typed λ-Calculus

We define a type Inp_Σ of inputs as *finite structures*, and look at queries expressed by programs of type $\mathsf{Inp}_\Sigma \multimap \mathsf{!!Bool}$.
(Σ: relational vocabulary)

- Good news: it seems that we get exactly *deterministic logarithmic space* (L) queries
  (also: much cleaner than previous characterizations of L in LL)
- Bad news: we haven't managed to entirely prove that

ICC in LL meets finite models

Our paper: ELL-based implicit complexity
+ inspiration from Hillebrand's PhD thesis (1994),
Finite Model Theory in the Simply Typed λ-Calculus

We define a type Inp_Σ of inputs as *finite structures*, and look at queries expressed by programs of type $\mathsf{Inp}_\Sigma \multimap !!\mathsf{Bool}$.
(Σ: relational vocabulary)

- Good news: it seems that we get exactly *deterministic logarithmic space* (L) queries
  (also: much cleaner than previous characterizations of L in LL)
- Bad news: we haven't managed to entirely prove that
  (unless we add some ad-hoc restriction)

Our results

Theorem

The class of queries over finite structures (with linear orders) computed by 2nd-order ELL proofs/programs of type $\mathsf{Inp}_\Sigma \multimap \mathsf{!!Bool}$ *lies between* L *and* NL.

NL: non-deterministic logarithmic space

Theorem

By adding a restriction on the existential witnesses allowed in the proofs/programs, we obtain a characterization of L.

Our results

Theorem

The class of queries over finite structures (with linear orders) computed by 2nd-order ELL proofs/programs of type $\mathsf{Inp}_\Sigma \multimap \mathsf{!!Bool}$ *lies between* L *and* L^{UL} (L *with an* UL *oracle*).

NL: non-deterministic logarithmic space

UL: *unambiguous* NL (unique accepting runs)

$\mathsf{NL}^{\mathsf{NL}} = \mathsf{NL}$ (since $\mathsf{NL} = \mathsf{coNL}$) so $\mathsf{L}^{\mathsf{UL}} \subseteq \mathsf{NL}$

(It is conjectured that $\mathsf{UL} = \mathsf{NL}$, and known that $\mathsf{UL/poly} = \mathsf{NL/poly}$.)

Theorem

By adding a restriction on the existential witnesses allowed in the proofs/programs, we obtain a characterization of L.

Lower bound via descriptive complexity

Recall that over linearly ordered finite structures:

Theorem (Immerman 1983)

L = *queries in 1st-order logic* + **deterministic transitive closure**.

How to compute (deterministic) transitive closure in ELL?

Lower bound via descriptive complexity

Recall that over linearly ordered finite structures:

Theorem (Immerman 1983)

L = *queries in 1st-order logic* + **deterministic transitive closure**.

How to compute (deterministic) transitive closure in ELL?

$\Phi_R : Q \mapsto \{(x,z) \mid x = z \vee (\exists y : (x,y) \in R \wedge (y,z) \in Q)\}$

R^* = *least fixpoint* of Φ_R, obtained by iteration (Φ_R monotone).

Problem: because of $\exists y$, Φ_R uses Q once for each y,
but the $!\mathsf{np}$ type in ELL only allows us to iterate *linear* maps.

Lower bound via descriptive complexity

Recall that over linearly ordered finite structures:

Theorem (Immerman 1983)

L = *queries in 1st-order logic* + **deterministic transitive closure**.

How to compute (deterministic) transitive closure in ELL?
$\Phi_R : Q \mapsto \{(x,z) \mid x = z \vee (\exists y : (x,y) \in R \wedge (y,z) \in Q)\}$
R^* = *least fixpoint* of Φ_R, obtained by iteration (Φ_R monotone).

Problem: because of $\exists y$, Φ_R uses Q once for each y,
but the $!\mathsf{np}$ type in ELL only allows us to iterate *linear* maps.

Determinism (L vs NL) corresponds to linearity

R deterministic $\iff$ R corresponds to *partial function* f_R; then
$\Phi_R : Q \mapsto \{(x,z) \mid x = z \vee (f_R(x),z) \in Q\}$ looks at Q only once!

Upper bound via semantic evaluation

Previous work on ELL uses computation by rewriting.

Issue: not space-efficient!

Upper bound via semantic evaluation

Previous work on ELL uses computation by rewriting.

Issue: not space-efficient!

Idea

Evaluate the program in some *semantics* $[\![-]\!]$.

- type $T \rightsquigarrow$ some mathematical structure $[\![T]\!]$
- program p of type $T \rightsquigarrow [\![p]\!]$ "$\in$" $[\![T]\!]$
- $[\![-]\!]$ is *compositional*, e.g. $[\![f(x)]\!] = [\![f]\!]\,([\![x]\!])$

Upper bound via semantic evaluation

Previous work on ELL uses computation by rewriting.

Issue: not space-efficient!

Idea

Evaluate the program in some *finite effective semantics* $[\![-]\!]$.

- type $T \rightsquigarrow$ some mathematical structure $[\![T]\!]$
- program p of type $T \rightsquigarrow [\![p]\!]$ "$\in$" $[\![T]\!]$
- $[\![-]\!]$ is *compositional*, e.g. $[\![f(x)]\!] = [\![f]\!]\,([\![x]\!])$
  $\implies$ the entire computation can be done semantically!
  - apply logspace algorithmic trickery here

Upper bound via semantic evaluation

Previous work on ELL uses computation by rewriting.

Issue: not space-efficient!

Idea

Evaluate the program in the *coherence space* semantics $[\![-]\!]$.

- type $T \rightsquigarrow$ an undirected graph $[\![T]\!]$
- program p of type $T \rightsquigarrow [\![p]\!]$ clique of $[\![T]\!]$
- $[\![-]\!]$ is *compositional*, e.g. $[\![f(x)]\!] = [\![f]\!]\,([\![x]\!])$
  $\Longrightarrow$ the entire computation can be done semantically!
  - apply logspace algorithmic trickery here
    our case: compose memoryful strategies in a sort of game

Upper bound via semantic evaluation

Previous work on ELL uses computation by rewriting.

Issue: not space-efficient!

Idea

Evaluate the program in the *coherence space* semantics $[\![-]\!]$.

- type $T \rightsquigarrow$ an undirected graph $[\![T]\!]$
- program p of type $T \rightsquigarrow [\![p]\!]$ clique of $[\![T]\!]$
- $[\![-]\!]$ is *compositional*, e.g. $[\![f(x)]\!] = [\![f]\!]\,([\![x]\!])$
  $\Longrightarrow$ the entire computation can be done semantically!
  - apply logspace algorithmic trickery here
    our case: compose memoryful strategies in a sort of game

Semantic evaluation has been used for complexity in λ-calculi before (e.g. in Hillebrand's PhD) but no precedent in linear logic.

Conclusion

We brought some ideas to *implicit computational complexity* (ICC) based on (*elementary*) *linear logic* from other areas:

- descriptive complexity:
  finite models as inputs, lower bound via FO+DTC
- ICC in typed λ-calculi: upper bound via semantics
  (hence “normal functors” in the title)

Conclusion

We brought some ideas to *implicit computational complexity* (ICC) based on (*elementary*) *linear logic* from other areas:

- descriptive complexity:
  finite models as inputs, lower bound via FO+DTC
- ICC in typed λ-calculi: upper bound via semantics
  (hence "normal functors" in the title)

Concrete results: a nice conjectural characterization of L, and

- partial results: between L and $\mathsf{L}^{\mathsf{UL}} \subseteq \mathsf{NL}$
- a complete proof for a less nice version

Conclusion

We brought some ideas to *implicit computational complexity* (ICC) based on (*elementary*) *linear logic* from other areas:

- descriptive complexity:
  finite models as inputs, lower bound via FO+DTC
- ICC in typed λ-calculi: upper bound via semantics
  (hence "normal functors" in the title)

Concrete results: a nice conjectural characterization of L, and

- partial results: between L and $\mathsf{L}^{\mathsf{UL}} \subseteq \mathsf{NL}$
- a complete proof for a less nice version

Part of a larger research program; other results include a LL-based characterization of *regular transductions*.