From linear logic to types for implicit computational complexity

Patrick Baillot

CNRS / ENS Lyon

Days in Logic 2018

Aveiro

January 25, 2018
Introduction

- Complexity classes are defined by:
  - a computational model, e.g. TM
  - a constraint on resources, e.g. time, space or size

- This does not say much about how to compute within a certain complexity class.

Complexity classes are defined “from the outside” . . .
Introduction: some questions

- **How** can we compute within a certain complexity class, for instance in FPTIME?
- Which bricks of computation can we use? data structures, primitive operations, control structures (e.g. loops) . . .
- . . . without the burden of managing explicit time annotations.
Introduction: more questions . . .

- A related question: how and when can we compose and iterate functions of a given complexity class?
- Can we define a discipline for transparent and modular FPTIME programming?
- Can we give characterizations of complexity classes not relying on explicit resource bounds?
Logic and recursion theory can help addressing some of these questions!

They have triggered
Implicit computational complexity (ICC):
characterizing complexity classes by logics / languages without explicit bounds,
but instead by restricting the constructions
Logic and recursion theory can help addressing some of these questions!

They have triggered Implicit computational complexity (ICC)):
characterizing complexity classes by logics / languages without explicit bounds,
but instead by restricting the constructions
Introduction: ICC systems

- ICC can be both foundations-oriented or certification-oriented
- ICC systems can often be expressed by
  (i) a programming language or calculus, (ii) a criterion on programs
Various approaches to ICC

- recursion theory: safe recursion (Bellantoni-Cook) / ramified recursion (Leivant)
- **linear logic** (Girard) this talk
- types controlling sizes (non-size-increasing) (Hofmann)
- interpretation methods (Marion)
- ...
The proofs-as-programs viewpoint

- our reference language here is $\lambda$-calculus
- untyped $\lambda$-calculus is Turing-complete
- type systems can guarantee termination
  ex: system F (polymorphic types)
The proofs-as-programs viewpoint (2)

- proofs-as-programs correspondence
  - proof = type derivation
  - normalization = execution

2nd order intuitionistic logic $\leftrightarrow$ system F

- some characteristics of $\lambda$-calculus:
  - higher-order types
  - no distinction between data / program
linear logic (LL):
fine-grained decomposition of intuitionistic logic
duplication is controlled with a specific connective \( ! \)
(exponential modality)

some variants of linear logic with weak rules for \( ! \) have bounded complexity: light logics
How to characterize complexity classes?

the computational engine

logical system

variant of linear logic

the specification

formula / type

formula / type

Patrick Baillot

From linear logic to types for implicit computational complexity
How to characterize complexity classes?

the computational engine
logical system

variant of linear logic

the specification
formula / type

variant of linear logic

formula / type
Outline of the course

1. λ-calculus and system F in a nutshell
2. elementary linear logic (ELL): elementary complexity
3. some finer characterizations in ELL
4. light linear logic (LLL): Ptime complexity
5. other linear logic variants
6. conclusion
We denote:

- $FDTIME(f(n))$: functions on binary words computable by a Turing machine in time $O(f(n))$
- $FPTIME = \bigcup_k FDTIME(n^k)$, feasible functions
- $FE\text{XPTIME} = \bigcup_k FDTIME(2^{n^k})$
- *Elementary*: functions computable in time $2^k_n$, for some $k$, where

\[
\begin{align*}
2^x_0 &= x \\
2^x_{k+1} &= 2^{2^k_x}
\end{align*}
\]
\textbf{λ-calculus}

- \textbf{λ-terms}:

  \[ t, u ::= x | \lambda x.t | t u \]

  notations: \( \lambda x_1 x_2.t \) for \( \lambda x_1.\lambda x_2.t \)
  \( (t u v) \) for \( ((t u) v) \)

- \textbf{β-reduction}:

  \( \xrightarrow{1} \) relation obtained by context-closure of:

  \[ ((\lambda x.t)u) \xrightarrow{1} t[u/x] \]

  → reflexive and transitive closure of \( \xrightarrow{1} \).
Typed $\lambda$-terms

system F types:

$$T, U ::= \alpha \mid T \rightarrow U \mid \forall \alpha. T$$

simple types: without $\forall$

simply typed terms, in Church-style:

$$x^T \quad (\lambda x^T . M^U)^{T \rightarrow U} \quad ((M^{T \rightarrow U})^T N^T)^U$$
Proofs-as-programs correspondence (Curry-Howard)

2nd-order intuitionistic logic proof \( \Rightarrow \) typed term

formula

proof of \( A_1, \ldots, A_n \vdash B \)

\( M^B \), with free variables \( x_i : A_i, 1 \leq i \leq n \)

normalization of proof (cut elimination)

\( \beta \)-reduction of term
Data types in F

Booleans:
\[ B^F = \forall \alpha. \alpha \to \alpha \to \alpha \]
\[ true = \lambda x. \lambda y. x \quad false = \lambda x. \lambda y. y \]

Church unary integers:
\[ N^F = \forall \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha) \]
example
\[ 2 = \lambda f^{\alpha \to \alpha}. \lambda x^{\alpha}. (f (f x)) : N^F \]

Church binary words:
\[ W^F = \forall \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha) \to (\alpha \to \alpha) \]
example
\[ \langle 1, 1, 0 \rangle = \lambda s_0^{\alpha \to \alpha}. \lambda s_1^{\alpha \to \alpha}. \lambda x^{\alpha}. (s_1 (s_1 (s_0 x))) : W^F \]
Examples of terms (1)

addition
\[ add = \lambda nmfx. (n f) (m f x) \]
\[ : N \rightarrow N \rightarrow N \]

multiplication
\[ mult = \lambda nmf. (n (m f)) \]
\[ : N \rightarrow N \rightarrow N \]

squaring
\[ square = \lambda nf. (n (n f)) \]
\[ : N \rightarrow N \rightarrow N \]
Iteration

For each inductive data type an associated iteration principle. For instance, for $N = \forall \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha)$, we can define an iterator $\text{iter}$:

$$\text{iter} = \lambda \text{fxn}. \, (n \, f \, x) : (A \to A) \to A \to N \to A,$$

for any $A$

then

$$(\text{iter} \, t \, u \, n) \to (t \, (t \ldots (t \, u) \ldots) \quad (n \, \text{times})$$

examples:

double : $N \to N$

exp = \lambda n. (\text{iter} \, \text{double} \, 1 \, n) : N \to N$
tower = \lambda n. (\text{iter} \, \text{exp} \, 1 \, n) : N \to N$$
Examples of terms (2)

concatenation

\[ \text{conc} = \lambda \mathbf{u}^W \cdot \lambda \mathbf{v}^W \cdot \lambda \mathbf{s}_0 \cdot \lambda \mathbf{s}_1 \cdot \lambda \mathbf{x}. (\mathbf{u} \mathbf{s}_0 \mathbf{s}_1) (\mathbf{v} \mathbf{s}_0 \mathbf{s}_1 \mathbf{x}) \]

: \( W \to W \to W \)

length

\[ \text{length} = \lambda \mathbf{u}^W \cdot \lambda \mathbf{f}^{\alpha \to \alpha} \cdot (\mathbf{u} \mathbf{f} \mathbf{f})^{\alpha \to \alpha} \]

: \( W \to N \)

repeated concatenation

\[ \text{rep} = \lambda \mathbf{n}^N \cdot \lambda \mathbf{v}^W \cdot \text{iter } (\text{conc } \mathbf{v})^{W \to W} \text{ nil}^W \mathbf{n}^N \]

\[ = \lambda \mathbf{n}^N \cdot \lambda \mathbf{v}^W \cdot [\mathbf{n} (\text{conc } \mathbf{v}) \text{ nil}]^W \]

: \( N \to W \to W \)
Theorem (Girard)
If a term is well typed in $F$, then it is strongly normalizable.

Thus a type derivation can be seen as a termination witness. In particular, a term $t : W \rightarrow W$ represents a function on words which terminates on all inputs.

Can we refine this system in order to guarantee feasible termination, that is to say in polynomial time?
Linear logic

- Linear logic (LL) arises from the decomposition

\[ A \Rightarrow B \equiv !A \multimap B \]

- The ! modality accounts for duplication (contraction)
- ! satisfies the following principles:

\[
\begin{align*}
!A \multimap !A \otimes !A \\
A \vdash B & \quad \Rightarrow \quad !A \vdash !B \\
!A \otimes !B \multimap !(A \otimes B) & \quad !A \multimap !!A
\end{align*}
\]
Elementary linear logic (ELL) [Girard95]

- Language of formulas:

\[ A, B := \alpha \mid A \rightarrow B \mid !A \mid \forall \alpha.A \]

Denote \(!^kA\) for \(k\) occurrences of \(!\).

- The system is designed in such a way that the following principles are not provable:

\[ !A \rightarrow A, \quad !A \rightarrow !!A \]

- Defined to characterize elementary time complexity, that is to say in time bounded by \(2^n_k\), for arbitrary \(k\).
Elementary linear logic rules

\[
\frac{x : A \vdash x : A}{\text{(Id)}}
\]

\[
\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B} \quad (\multimap i)
\]

\[
\frac{\Gamma_1 \vdash t : A \multimap B \quad \Gamma_2 \vdash u : A}{\Gamma_1, \Gamma_2 \vdash (t \ u) : B} \quad (\multimap e)
\]

\[
\frac{x_1 : !A, x_2 : !A, \Gamma \vdash t : B}{x : !A, \Gamma \vdash t[x/x_1, x/x_2] : B} \quad (\text{Cntr})
\]

\[
\frac{\Gamma \vdash t : A}{\Gamma, x : B \vdash t[x] : A} \quad (\text{Weak})
\]

\[
\frac{x_1 : B_1, \ldots, x_n : B_n \vdash t : A}{x_1 : !B_1, \ldots, x_n : !B_n \vdash t : !A} \quad (! i)
\]

\[
\frac{\Gamma_1 \vdash u : !A \quad \Gamma_2, x : !A \vdash t : B}{\Gamma_1, \Gamma_2 \vdash t[u/x] : B} \quad (! e)
\]
Elementary linear logic rules (2/2)

\[
\begin{align*}
\Gamma \vdash t : A & \quad (\forall i) (\star) \\
\Gamma \vdash t : \forall \alpha. A & \quad (\forall e)
\end{align*}
\]

where (\star) : \alpha \notin \Gamma.
• This is actually elementary **affine** logic (EAL), because of the unrestricted weakening (not only on !A formulas).

• However throughout this talk we will say linear instead of affine, so ELL will mean EAL . . .

• These rules are natural deduction style rules. There is also a sequent calculus presentation of ELL.
Consider \((\cdot)^- : ELL \rightarrow F\) defined by:

\[
(!A)^- = A^-, \quad (A \rightarrow B)^- = A^- \rightarrow B^-, \quad (\forall \alpha. A)^- = \forall \alpha. A^-, \quad \alpha^- = \alpha.
\]

**Proposition**

If \(\Gamma \vdash_{ELL} t : A\) then \(t\) is typable in \(F\) with type \(A^-\).

If \(A^- = T\), say \(A\) is a decoration of \(T\) in \(ELL\).
Data types in ELL

- **Church unary integers**

  system F: $\forall \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha)$

  ELL: $\forall \alpha. ! (\alpha \to \alpha) \to ! (\alpha \to \alpha)$

  Example: integer 2, in F:

  \[ 2 = \lambda f^{(\alpha \to \alpha)}. \lambda x^{\alpha}. (f (f x)) \, . \]

- **Church binary words**

  system F: $\forall \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha) \to (\alpha \to \alpha)$

  ELL: $\forall \alpha. ! (\alpha \to \alpha) \to ! (\alpha \to \alpha) \to ! (\alpha \to \alpha)$

  Example: $w = \langle 1, 0, 0 \rangle$, in F:

  \[ w = \lambda s_0^{(\alpha \to \alpha)}. \lambda s_1^{(\alpha \to \alpha)}. \lambda x^{\alpha}. (s_1 (s_0 (s_0 x))) \, . \]
Representation of functions

- a term $t$ of type $!^k N \to !^l N$, for some $k, l$, represents a function over unary integers
- $!^k W \to !^l W$, for some $k, l$: function over binary words
Examples of ELL terms (1)

- some examples of terms

**addition**

\[ add = \lambda nmfx. (n f) (m f x) \]
\[ : N \multimap N \multimap N \]

**multiplication**

\[ mult = \lambda nmf. (n (m f)) \]
\[ : N \multimap N \multimap N \]

**squaring**

\[ square = \lambda nf. (n (n f)) \]
\[ : !N \multimap !N \]
Iteration in ELL

recall the iterator $\textit{iter}$:

\[
\text{iter} = \lambda f \ x \ n. \ (n \ f \ x) : !(A \rightarrow A) \rightarrow !A \rightarrow N \rightarrow !A
\]

with $(\text{iter} \ t \ u \ n) \rightarrow (t \ (t \ \ldots \ (t \ u) \ldots ))$ \hspace{1cm} (n \ times)

\textbf{examples:}

$\text{double} : N \rightarrow N$

$\text{exp} = (\text{iter} \ \text{double} \ 1) : N \rightarrow !N$

remark: $\text{exp}$ cannot be iterated; $\text{tower} = (\text{iter} \ \text{exp} \ 1)$ non ELL typable.

$\text{coercion} = (\text{iter} \ \text{succ} \ 0) : N \rightarrow !N$ : an identity, but changes the type
Iteration in ELL

recall the iterator $\text{iter}$:

$$\text{iter} = \lambda f \times n. \ (n \ f \ x) : ! (A \multimap A) \multimap ! A \multimap N \multimap ! A$$

with $(\text{iter} \ t \ u \ n) \to (t \ (t \ \ldots (t \ u) \ldots)) \quad (n \ \text{times})$

examples:

$\text{double} : N \multimap N$

$exp = (\text{iter} \ \text{double} \ 1) : N \multimap ! N$

remark: $exp$ cannot be iterated; $\text{tower} = (\text{iter} \ \text{exp} \ 1)$ non ELL typable.

$\text{coercion} = (\text{iter} \ \text{succ} \ 0) : N \multimap ! N : \text{an identity, but changes the type}$
recall the iterator \( \text{iter} \):

\[
\text{iter} = \lambda f \, x \, n. \ (n \, f \, x) : ! (A \to A) \to ! A \to N \to ! A
\]

with \((\text{iter} \ t \ u \ n) \to (t \ (t \ \ldots \ (t \ u) \ldots)) \) \ (n \ times)\\

**examples:**

\( \text{double} : N \to N \)

\( exp = (\text{iter} \ \text{double} \ 1) : N \to ! N \)

remark: \( exp \) cannot be iterated; \( tower = (\text{iter} \ exp \ 1) \) non ELL typable.

\( coercion = (\text{iter} \ \text{succ} \ 0) : N \to ! N \) : an identity, but changes the type
Examples of ELL terms (2)

concatenation

\[ \text{concat} \quad \lambda u. \lambda v. \lambda s_0. \lambda s_1. \lambda x. (u \ s_0 \ s_1) \ (v \ s_0 \ s_1 \ x) \]

: \quad W \multimap W \multimap W

length

\[ \text{length} \quad \lambda u. \lambda f. (u \ f \ f) \]

: \quad W \multimap N

repeated concatenation

\[ \text{rep} \quad \lambda n. \lambda v. \left[ \text{iter} \ (\text{concat} \ v) \ \text{nil} \ n \right] \]

\[ \lambda n. \lambda v. \left[ n \ (\text{concat} \ v) \ \text{nil} \right] \]

: \quad N \multimap! W \multimap! W
From derivations to proof-nets

\[ \Pi \vdash x_1 : A_1, \ldots, x_m : A_m \vdash \text{t : B} \]

\[ \Rightarrow \quad \Pi^* = \text{proof-net} \]

\[ A_1 \quad \ldots \quad A_m \]

\(\text{derivation}\)
Elementary linear logic rules, again

\[
\begin{align*}
& \Gamma, x : A \vdash t : B \quad \Gamma \vdash \lambda x. t : A \multimap B \quad (\multimap i) \\
& \Gamma_1 \vdash t : A \multimap B \quad \Gamma_2 \vdash u : A \\
& \Gamma_1, \Gamma_2 \vdash (t \ u) : B \quad (\multimap e) \\
& \Gamma, x_1 : !A, x_2 : !A \vdash t : B \\
& \Gamma \vdash x : !A, \Gamma \vdash t[x/x_1, x/x_2] : B \quad (\text{Cntr}) \\
& \Gamma \vdash t : A \\
& \Gamma, x : B \vdash t : A \quad (\text{Weak}) \\
& \Gamma_1, \Gamma_2 \vdash t[u/x] : B \\
& \Gamma_1 \vdash u : !A \\
& \Gamma_2, x : !A \vdash t : B \\
& \Gamma_1, \Gamma_2 \vdash t[u/x] : B \quad (! e)
\end{align*}
\]
ELL Proof-Nets

\( \lambda \)-calculus and system F in a nutshell
Elementary linear logic
Some finer characterizations in ELL
Light linear logic
Other linear logic variants

\( \Pi \)-calculus and system F in a nutshell
Elementary linear logic
Some finer characterizations in ELL
Light linear logic
Other linear logic variants

ELL Proof-Nets

\[ (\text{Id}) \]

\[ (\text{\( \to \)}_c) \]

\[ (\text{\( \to \)}_e) \]

\[ (\text{Cut}) \]

\[ (\text{Weak}) \]

\[ (\text{\( \Gamma \text{ \( \to \)} \\( \Delta \}}) \]

\[ (\text{\( \Gamma \text{ \( \to \)} \\( \Delta \}}) \]

\[ (\text{\( \Gamma \text{ \( \to \)} \\( \Delta \}}) \]
ELL proof-net : example

Proof-net $R_3$ representing Church integer 3:
ELL proof-net: depth

- Depth of an edge $e$ in a proof-net $R$: number of boxes it is contained in.
- Depth $d(R)$ of proof-net $R$: maximal depth of its edges.
- Example:
  - The previous proof-net $R_3$ has depth 1.
  - Any proof-net $R_n$ representing $n$ has depth 1.
ELL proof-net reduction: cut elimination

Patrick Baillot

From linear logic to types for implicit computational complexity
ELL proof-net reduction: cut elimination
Methodology

- write programs with ELL typed \(\lambda\)-terms
- evaluate them by:
  - compiling them into proof-nets, and then performing proof-net reduction
- beware:
  - proof-net reduction does not exactly match \(\beta\)-reduction
  - ELL does not satisfy subject reduction
- but that’s all right for our present goal . . .
ELL proof-net reduction properties

- Recall: depth of an edge $e$ in a proof-net $R = \text{number of boxes it is contained in}$.
- We have

**Proposition (Stratification)**

The depth of an edge does not change during reduction.

Consequence: the depth $d$ of a proof-net does not increase during reduction.

- **Level-by-level reduction strategy:**
  $R$ proof-net of depth $d$
  perform reduction successively at depth $0, 1 \ldots, d$. 
let $R$ be an ELL proof-net of depth $d$

$|R|_i =$ number of nodes at depth $i = $ size at depth $i$

$|R| =$ total size

round $i$: reduction at depth $i$

there are $d+1$ rounds for the reduction of $R$

**what happens during round $i$?**

- $|R|_i$ decreases at each step
  - thus there are at most $|R|_i$ steps  (size bounds time)
- but $|R|_{i+1}$ can increase at each step, in fact it can double
- hence round $i$ can cause an exponential size increase

on the whole we have a $2^{|R|_d}$ size increase

this yields a $O(2^{|R|_d})$ bound on the number of steps
ELL complexity results

**Theorem (Proof-net complexity)**

If $R$ is an ELL proof-net of depth $d$, then it can be reduced to its normal form in $O(2_d^{|R|})$ steps.

**Theorem (Representable functions)**

The functions representable by a term of type $N \rightarrow^k !_N$, where $k \geq 0$, are exactly the elementary time functions.
Proof of the representability theorem

- \( \subseteq \) (soundness):
  if \( t : N \rightarrow !^k N \) for some \( k \), then \( t \) represents an elementary function \( f \).
  
  **proof**: compute \((tn)\) by proof-net reduction.

- \( \supseteq \) (completeness):
  if \( f : \mathbb{N} \rightarrow \mathbb{N} \) is an elementary function, then there exists \( k \) and \( t : N \rightarrow !^k N \) such that \( t \) represents \( f \).
  
  **proof**: simulation of \( O(2^n) \)-time bounded Turing machine, for any \( i \).
From linear logic to types for implicit computational complexity (Part 2)

Patrick Baillot

CNRS / ENS Lyon

Days in Logic 2018
Aveiro

January 26, 2018
Implicit computational complexity (ICC) : characterizing complexity classes by logics / languages without explicit bounds, but instead by restricting the constructions.

we are considering here the proofs-as-programs approach for ICC . . .

. . . illustrating the use of linear logic and its weak variants.
Language of formulas:

\[ A, B ::= \alpha \mid A \rightarrow B \mid !A \mid \forall \alpha. A \]

We denote \!^k A for \( k \) occurrences of \(!\).
Elementary linear logic rules

\[
\frac{x : A \vdash x : A}{(\text{Id})}
\]

\[
\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x. t : A \multimap B} \quad (\multimap \text{i})
\]

\[
\frac{\Gamma_1 \vdash t : A \multimap B \quad \Gamma_2 \vdash u : A}{\Gamma_1, \Gamma_2 \vdash (t \ u) : B} \quad (\multimap \text{e})
\]

\[
\frac{x_1 : !A, x_2 : !A, \Gamma \vdash t : B}{x : !A, \Gamma \vdash t[x/x_1, x/x_2] : B} \quad (\text{Cntr})
\]

\[
\frac{\Gamma \vdash t : A}{\Gamma, x : B \vdash t[x_1/x, x_2/x] : A} \quad (\text{Weak})
\]

\[
\frac{x_1 : B_1, \ldots, x_n : B_n \vdash t : A}{x_1 : !B_1, \ldots, x_n : !B_n \vdash t : !A} \quad (! \text{i})
\]

\[
\frac{\Gamma_1 \vdash u : !A \quad \Gamma_2, x : !A \vdash t : B}{\Gamma_1, \Gamma_2 \vdash t[u/x] : B} \quad (! \text{e})
\]
\[
\frac{\Gamma 
\vdash t : A}{\Gamma \vdash t : \forall \alpha. A} \quad (\forall \text{ i}) \quad (*)
\]
\[
\frac{\Gamma \vdash t : \forall \alpha. A}{\Gamma \vdash t : A[B/\alpha]} \quad (\forall \text{ e})
\]

where \((*) : \alpha \notin FV(\Gamma)\).
Data types in ELL

- **Church unary integers**

  system F:  
  \[ N^F \]
  \[ \forall \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha) \]

  \[
  \forall \alpha. \!(\alpha \multimap \alpha) \multimap \!(\alpha \multimap \alpha)
  \]

  Example: integer 2, in F:
  \[ 2 = \lambda f^{(\alpha \to \alpha)}. \lambda x^{\alpha}. (f (f x)) . \]

- **Church binary words**

  system F:  
  \[ W^F \]
  \[ \forall \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha) \to (\alpha \to \alpha) \]

  \[
  \forall \alpha. \!(\alpha \multimap \alpha) \multimap \!(\alpha \multimap \alpha) \multimap \!(\alpha \multimap \alpha)
  \]

  Example: \( w = \langle 1, 0, 0 \rangle \), in F:
  \[ w = \lambda s_0^{(\alpha \to \alpha)}. \lambda s_1^{(\alpha \to \alpha)}. \lambda x^{\alpha}. (s_1 (s_0 (s_0 x))) . \]
ELL complexity results

Theorem (Proof-net complexity)

If $R$ is an ELL proof-net of depth $d$, then it can be reduced to its normal form in $O(2^{|R|_d})$ steps.

Theorem (Representable functions)

The functions representable by a term of type $N \rightarrow^k N$, where $k \geq 0$, are exactly the elementary time functions.
The computational engine

Logical system: Elementary linear logic (ELL)

The specification

Logical system: Elementary linear logic (ELL)

Formalization: $\{N \rightarrow !^k N\}_{k \geq 0} \equiv \text{Elementary}$
Characterization of complexity classes

the computational engine
logical system

Elementary linear logic (ELL)

the specification
formula / type

\{N \rightarrow^{k} N\}_{k \geq 0} \equiv \text{Elementary}
ELL: towards finer characterizations

- We have seen a characterization of the *Elementary* class (elementary complexity) in ELL
- But can we get more fined-grained characterizations? Characterize smaller complexity classes?
The system $ELL_\mu$

- we can extend ELL by adding a new construction $\mu\alpha.A$ for formula fixpoints, with the following rules:

$$
\begin{align*}
\frac{\Gamma \vdash t : A[\mu\alpha.A/\alpha]}{\Gamma \vdash t : \mu\alpha.A} \quad (\mu f) \\
\frac{\Gamma \vdash t : \mu\alpha.A}{\Gamma \vdash t : A[\mu\alpha.A/\alpha]} \quad (\mu u)
\end{align*}
$$

We call $ELL_\mu$ this system.

- the previous results on $ELL$ also hold for $ELL_\mu$ (same bound on cut-elimination).
By the previous analysis we know that a term $t : !W \rightarrow !^2B$ can be evaluated in $O(2^n)$, so it is in 2-EXPTIME . . .

but actually it is in ... PTIME
New characterization in $ELL_\mu$
Theorem

We consider the system $ELL_\mu$. The functions representable by proofs of $!W \to !^2B$ are exactly the class PTIME, of polynomial time predicates.
Key Lemma for the soundness proof

For proving complexity soundness we use a more precise bound than before:

**Lemma (Size bound)**

Let $R$ be a proof-net with:
- only exponential cuts at depth 0,
- $k$ cuts at depth 0.

Let $R'$ be the proof-net obtained by reducing $R$ at depth 0. Then we have:

$$|R'|_1 \leq |R|_0^k |R|_1.$$
in \( ELL_\mu \) we can define new data types, eg Scott integers:

\[
N_S = \mu \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow \beta \rightarrow \beta
\]

in \( \lambda \)-calculus notation:

\[
\begin{align*}
0 &= \lambda s. \lambda x. x \\
n + 1 &= \lambda s. \lambda x. (s \ n)
\end{align*}
\]

They allows for constant time predecessor and zero-test, but . . . no iterator.

Similarly one defines \( W_S \) for Scott binary words.

We get:

\[
\text{case} : \ \forall \alpha. (W_S \rightarrow \alpha) \rightarrow (W_S \rightarrow \alpha) \rightarrow \alpha \rightarrow (W_S \rightarrow \alpha)
\]
Proof of completeness for $!W \vdash !!B$ and PTIME

- any polynomial can be represented with a proof of $!N \vdash !N$. We have $\text{length} : W \vdash N$.
- Using type fixpoints, we can define a type $\text{Config}_S$ for TM configurations, based on Scott words, with:
  - proofs
    
    $\text{init} : W \vdash \text{Config}_S$
    
    $\text{accept} ? : \text{Config}_S \vdash B$

  - For any TM $M$, a proof
    
    $\text{step} : \text{Config}_S \vdash \text{Config}_S$

    Then, by iterating $\text{step} q(|w|)$ times on input ($\text{init}(w)$) we get:
    
    $!W \vdash !^2 \text{Config}_S$.

    Composing with $\text{accept} ?$ we get: $!W \vdash !^2 B$. 

Patrick Baillot
From linear logic to types for implicit computational complexity
Why do we need type fixpoints?

Without type fixpoints we can define using second-order a type \( \text{Config} \) based on Church integers (following [Asperti-Roversi2002]). We get the same types for \textit{length}, \textit{init}, \textit{step}, and we also obtain by iteration:

\[
!W \vdash !^2 \text{Config}.
\]

However the problem is then that:

\[
\text{accept?} : \text{Config} \vdash !B
\]

So this gives:

\[
!W \vdash !^3 B,
\]

which is not the type needed . . .
Theorem

We consider the system $ELL_\mu$.

- The functions representable by proofs of $!W \to !^2 B$ are exactly the class PTIME;
- The functions representable by proofs of $!W \to !^{k+2} B$ are exactly the class $k$-EXPTIME ($k \geq 1$).

where $k$-EXPTIME $= \bigcup_{i \in \mathbb{N}} DTIME(2^{n^i}_k)$

Note that we do not use fixpoints in the types above ... but they are used in the proofs.
What about function classes?

**Theorem**

We consider the system $ELL_{\mu}$. The functions representable by proofs of $!W \multimap !^2 W_S$ are exactly the class FPTIME;

recall:

$W$: type of Church binary words

$W_S$: type of Scott binary words

However this characterization is not so satisfactory because of the I/O distinct data-types: these programs cannot be composed!
Let us define a new data-type:

\[ W_k = \text{def } !^k N \otimes !^{k+1} W_S \]

**Theorem**

We consider the system \( ELL_\mu \).

For \( k \geq 0 \), the functions representable by proofs of \( W_1 \to W_{k+1} \) are exactly the class \( k\text{-FEXPTIME} \).

For \( \text{FPTIME} \) we have the type \( W_1 \to W_1 \), and now these programs can be composed!
Characterization of complexity classes

- Elementary linear logic $ELL_{\mu}$
- The computational engine
- Logical system
- The specification

Logical system $ELL_{\mu}$

\[ \mu W \Rightarrow !^{k+2} B \equiv k-EXPTIME \]

\[ W_1 \Rightarrow W_{k+1} \equiv k-FEXPTIME \]
Characterization of complexity classes

the computational engine
logical system

Elementary linear logic $ELL_\mu$

the specification
formula / type

$!W \rightarrow !^{k+2}B \equiv k$-EXPTIME

$W_1 \rightarrow W_{k+1} \equiv k$-FEXPTIME
Characterization of complexity classes

the computational engine
logical system

Elementary linear logic $ELL_\mu$

the specification
formula / type

$!W \multimap !^{k+2}B \equiv k\text{-EXPTIME}$

$W_1 \multimap W_{k+1} \equiv k\text{-FEXPTIME}$
Comparison with previous works

- Jones 2001:
  read-only functional programs with arguments of order $\leq k$
  $\equiv k$-EXP

- Leivant 2002:
  second-order intuitionistic logic with comprehension restricted to order $\leq k$ formulas
  $\equiv k$-EXP

in these settings: restriction of a particular operation inside the proof or program
by contrast in $EAL_{\mu}$ the condition is only on the conclusion (type) of the proof.
Questioning the robustness of ELL

- Can we enrich the language by adding new primitives, and keep the properties?
- We already saw that for type fixpoints
- What about adding an FPTIME primitive $F$, with type $F : W \rightarrow W$?
Proposition

Consider an extension of ELL with a finite number of FPTIME primitives $F_i$ of type $W \rightarrow W$. Then the functions in $!W \rightarrow !B$ (resp. $!W \rightarrow !^2 B$) are in FPTIME (resp. 2-FEXPTIME).
Improve the expressivity of ELL?

- Denote \( nA \rightarrow B \) for \( A \rightarrow A \rightarrow \cdots \rightarrow A \rightarrow B \), with \( n \) occurrences of \( A \).
- There are only few functions of type \( nW \rightarrow W \) IN ELL.
- We would like a generic way of adding new primitives of type \( nW \rightarrow W \) to the language.
Consider the language $\mathit{s\ell T}$ given by:

- terms: $\lambda$-terms + constructors + iterators
- types:

  indexes \[ I, J \quad ::= \quad a \mid n \in \mathbb{N}^* \mid I + J \mid I \cdot J \]

  types \[ D, D' \quad ::= \quad N^I \mid W^I \mid D \rightarrow D' \mid D \otimes D' \]

- typing rules
Linear sized types (2)

- Examples of $\mathcal{slT}$ terms:

\[
\lambda x. s_0(s_1(x)) : W^a \to W^{a+2} \\
add = \lambda x. \text{itern}(\lambda y. \text{succ}(y), x) : N^I \to N^J \to N^{I+J}
\]

- We have:

**Proposition**

The functions representable by $\mathcal{slT}$ terms are exactly the class $\text{FPTIME}$.  

Patrick Baillot

From linear logic to types for implicit computational complexity
Consider the language of ELL$^+$ (enriched ELL) defined by:

- $s\ell T$ typing rules,
- the rules

$$
\frac{\vdash t : W^{a_1}, \ldots, W^{a_n} \rightarrow W}{\vdash t : W, \ldots, W \rightarrow W}
$$

- ELL typing rules.

This is a kind of 2-layers language.
Theorem

In $\text{ELL}^+$ we have:

- The functions representable by terms of type $!W \multimap !_B$ are exactly PTIME.
- For $k \geq 0$ the functions representable by terms of type $!W \multimap !_B^{k+1}$ are exactly $2k$-EXPTIME.
In ELL\(^+\) we can write terms for:

- **SAT**: \( N \multimap W \multimap !B \),
  where a CNF formula is given by the number of distinct variables and the encoding as a word.

- **QBF\(_k\)**: \( kN \multimap B \multimap W \multimap !B \),
  for testing satisfiability of quantified boolean formulas with \( k \) alternations of quantifiers.

- **SUBSET \_SUM**: \( W \multimap W \multimap !B \),
  where the first word represents an integer and the second one a set of integers.
Some references


From linear logic to types for implicit computational complexity (Part 3)

Patrick Baillot

CNRS / ENS Lyon

Days in Logic 2018
Aveiro

January 27, 2018
Introduction: recap

- Implicit computational complexity (ICC) : characterizing complexity classes by logics / languages without explicit bounds, but instead by restricting the constructions.

- We are considering here the proofs-as-programs approach for ICC, with linear logic.

- In the 2 first lectures we investigated Elementary linear logic (ELL).
Characterization of complexity classes

- Elementary linear logic $ELL_{\mu}$
  - $W \twoheadrightarrow !^{k+2}B \equiv k$-EXPTIME
  - $W_1 \twoheadrightarrow W_{k+1} \equiv k$-FEXPTIME
Taming the exponential blow-up in ELL?
Taming the exponential blow-up in ELL?
Light linear logic (LLL) \[\text{[Girard95]}\]

- Language of formulas:

\[A, B := \alpha | A \to B | \forall \alpha. A | !A | §A\]

intuition: § a new modality for non-duplicable boxes

- The following principles are still **not** provable

\[!A \to A, \quad !A \to !!A\]
Light linear logic rules

- rules (Id), (→ i), (→ e), (Cntr), (Weak): as in ELL.
- new rules (! i), (! e), (§ i), (§ e):

\[
\frac{x : B \vdash t : A}{x : !B \vdash t : !A} \quad (! i)
\]

\[
\frac{\Gamma_1 \vdash u : !A \quad \Gamma_2, x : !A \vdash t : B}{\Gamma_1, \Gamma_2 \vdash t[u/x] : B} \quad (! e)
\]

\[
\frac{\Gamma, \Delta \vdash t : A}{!\Gamma, \, \downarrow \Delta \vdash t : \downarrow A} \quad (\§ i)
\]

\[
\frac{\Gamma_1 \vdash u : \downarrow A \quad \Gamma_2, x : \downarrow A \vdash t : B}{\Gamma_1, \Gamma_2 \vdash t[u/x] : B} \quad (\§ e)
\]

at most one free variable in the premise judgement of (! i) rule.
Light linear logic principles

- The following formulas are provable:

\[ !A \rightarrow \underline{s}A \quad \underline{s}A \otimes \underline{s}B \rightarrow \underline{s}(A \otimes B) \]

- The following one is **not** provable in LLL, though it is in ELL:

\[ !A \otimes !B \rightarrow !{(A \otimes B)} \]
Consider $(.)^e : LLL \to ELL$ defined by:

$$(\exists A)^e = !A^e, \quad (!A)^e = !A^e$$

and other connectives unchanged.

**Proposition**

If $\Gamma \vdash_{LLL} t : A$ then $\Gamma^e \vdash_{ELL} t : A^e$. 
Data types in LLL

- **Church unary integers**
  
  system F: \[ N^F \]
  \[
  \forall \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha)
  \]
  LLL: \[ N^{LLL} \]
  \[
  \forall \alpha. (! (\alpha \to \alpha)) \to ! (\alpha \to \alpha) \to \$ (\alpha \to \alpha)
  \]

  Example: integer 2, in F:
  \[
  2 = \lambda f (\alpha \to \alpha). \lambda x \alpha. (f (f x)).
  \]

- **Church binary words**
  
  system F: \[ W^F \]
  \[
  \forall \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha) \to (\alpha \to \alpha)
  \]
  LLL: \[ W^{LLL} \]
  \[
  \forall \alpha. (! (\alpha \to \alpha)) \to ! (\alpha \to \alpha) \to ! (\alpha \to \alpha) \to \$ (\alpha \to \alpha)
  \]

  Example: \( w = \langle 1, 0, 0 \rangle \), in F:
  \[
  w = \lambda s_0 (\alpha \to \alpha). \lambda s_1 (\alpha \to \alpha). \lambda x \alpha. (s_1 (s_0 (s_0 x))).
  \]
Representation of functions

- a term $t$ of type $!^k N \multimap \$^l N$, for some $k, l$, represents a function over unary integers $!^k W \multimap \$^l W$: function over binary words.

- some examples of terms

  **addition**
  
  $$\text{add} \quad = \quad \lambda n m f x . (n f) (m f x)$$
  
  : $N \multimap N \multimap N$

  **double**
  
  $$\text{double} \quad = \quad \lambda n f x . (n f) (n f x)$$
  
  : $!N \multimap \$N$

  **concatenation**
  
  $$\text{conc} \quad : \quad W \multimap W \multimap W$$
we can type the iterator $\text{iter}$:

$$\text{iter} = \lambda f x. (n f x) : ! (A \rightarrow A) \rightarrow ! A \rightarrow N \rightarrow \$ A$$

examples:

$(\text{add3}) : N \rightarrow N$ can be iterated

$\text{double} : ! N \rightarrow \$ N$ cannot be iterated

thus some exponentially growing terms are not typable
LLL proof-nets

\[
\begin{array}{c}
\begin{array}{c}
\text{\textbf{\textit{\textbf{\textit{\textbf{\textit{\textbf{\textit{\textbf{\textit{\textbf{\textit{\textbf{\textit{-box}}}}}}}}}}}}}}}
\end{array}
\end{array}
\end{array}
\]
LLL proof-net reduction

\[
\begin{align*}
&\frac{}{\text{LLL proof-net reduction}} \\
&\frac{}{\text{LLL proof-net reduction}}
\end{align*}
\]
Level-by-level reduction of LLL proof-nets

- as in ELL we use a level-by-level strategy
- let $R$ be an LLL proof-net of depth $d$
  - round $i$: reduction at depth $i$
    - there are $d + 1$ rounds for the reduction of $R$
  - what happens during round $i$?
    - $|R|_i$ decreases at each step
      - thus there are at most $|R|_i$ steps (size bounds time)
    - yet $|R|_{i+1}$ can increase:
      - during round $i$ we can have a quadratic increase:
        $$|R'|_{i+1} \leq |R|_{i+1}^2$$
    - this repeats $d$ times, so on the whole we have a $|R|^{2d}$ size increase
    - this yields a $O(|R|^{2d})$ bound on the number of steps
Theorem (Proof-net complexity)
If $R$ is an LLL proof-net of depth $d$, then it can be reduced to its normal form in $O(|R|^{2^d})$ steps.

Thus at fixed depth $d$ we have a polynomial bound.

Theorem (Representable functions)
The functions representable by a term of type $W \rightarrow \circ^{k} W$, for $k \geq 0$, are exactly the functions of FP (polynomial time functions).
Characterization of complexity classes

Light linear logic $LLL$

$$\{ W \twoheadrightarrow \$^{k} W \}_{k \geq 0} \equiv \text{FPTIME}$$
Is LLL a good type system for lambda calculus ...?
Actually there are two problems:

- it does not satisfy subject-reduction,
- it does not ensure polynomial time complexity for \( \beta \)-reduction ...
Example:

\[ y : !(A \multimap !A \multimap !A) , \ z : !!A \vdash_{\text{LLL}} (\lambda x. yxx)^n z : \$!A \]

\[ t_n = (\lambda x. yxx)^n z, \]
\[ t_n \xrightarrow{\beta} u_n \quad \text{with} \]
\[ u_0 = z, \quad u_n = y \ u_{n-1} \ u_{n-1}. \]

we have: \( |t_n| \sim c.n, \quad |u_n| \sim 2^n. \)

**hence**: any beta-reduction of \( t_n \) to \( u_n \) costs exponential space on a Turing Machine!

**even though**: using proof-nets these reductions are done in polynomial time.

**culprit**: sharing allowed by \( !, \)

it entails that: for \( D \) type derivation for \( t \), we might have \( |t| \gg |D| \).
How to fix this problem?
Type system DLAL

To overcome the problems with typing in LLL: we restrict the use of ! to !A $\rightarrow$ B.

The DLAL (Dual Light Affine Logic) type system:

$$A, B ::= \alpha \mid A \rightarrow B \mid !A \rightarrow B \mid \$A \mid \forall \alpha. A$$

Typing judgements of the form: $\Gamma; \Delta \vdash t : A$, where
- $\Gamma$ contains duplicable variables,
- $\Delta$ contains linear variables.
DLAL satisfies the subject-reduction property.

**Theorem (Strong Ptime bound)**

If \( t \) is typable in DLAL with a derivation of depth \( d \), then any \( \beta \) reduction of \( t \) can be performed in time \( O((d + 1) \cdot |t|^{2d+1}) \).

Remarks:
- one in fact shows a bound \( O((d + 1) \cdot |t|^{2d}) \) on the number of \( \beta \)-steps and then uses the fact that the cost of each step is here bounded;
- this bound holds for any reduction strategy;
- in particular, if \( \vdash t : W \rightarrow \frac{\xi}{\xi}^k W \) then \( t \) is Ptime.
Theorem (Completeness)

For any polynomial time function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, there exists a term $t$ representing it and typable in DLAL with a type $W \rightarrow \mathcal{S}^k W$, for a certain integer $k \in \mathbb{N}$. 

Patrick Baillot
From linear logic to types for implicit computational complexity
Can we check DLAL typability?

DLAL type inference problem for system F terms:

**input:** system F term \( t \)

**problem:** does there exist a DLAL derivation for \( t \) ?

**main issue:**

- decorate the F derivation with \(! / \&\)
- for that, find out where to put boxes

...boils down to constructing a proof-net.
How can we find out the boxes needed?
At first sight there are several difficulties

1. no a priori bound on the number of boxes needed
2. even for box positions there is an exponential number of possibilities
3. furthermore: distinguish between ! and § boxes

Idea: we search for doors instead of boxes.
Example: term with doors
Example: parameterized term

\[ \mathbb{S}^{m_1}(\mathbb{S}^{b_2,m_2}\alpha \rightarrow \mathbb{S}^{m_3}\alpha), \text{ with boolean parameters } b_2 \text{ and integer parameters } m_1. \]
We express typability by a set of constraints on parameters, expressing e.g.: boxes are well-formed, a $!$-box has at most one auxiliary door etc.

- We get mixed boolean-linear constraints.
- We give a resolution procedure for deciding whether the constraints system is decidable, using linear programming.
- This resolution procedure is PTIME.
Finally we get:

**Theorem**

The DLAL type inference problem for system F terms can be decided in PTIME.
The completeness result is an extensional one, but the intensional expressivity of LLL and DLAL is limited. Indeed: rich features (higher-order, polymorphism) but ”pessimistic” account of iteration . . .
Some references

- Vincent Atassi, P. B., Kazushige Terui. Verification of Ptime Reducibility for system F Terms: Type Inference in Dual Light Affine Logic. Logical Methods in Computer Science 3(4) (2007)
A glimpse of the linear logics zoo

- for FPTIME
  - soft linear logic: \[\text{[Lafont04]}\]
    a simple system, but with more constrained programming
  - bounded linear logic: \[\text{[GSS92]}\]
    \(\forall P(\vec{x}) \rightarrow A\): more explicit, but more flexible

- for PSPACE
  - \(STA_B\) \[\text{[GMRdR08]}\]: extends soft linear logic with a craftly typed conditional

- for LOGSPACE
  - \(IntML\) \[\text{[DLS10]}\]: evaluation by computation by interaction

- for P/poly (non-uniform computation):
  - parsimonious \(\lambda\)-calculus \[\text{[MazzaTerui15]}\]
Conclusions and perspectives

- Linear logic can be used for implicit complexity
- With two ingredients:
  - Choice of the logic
  - Choice of the formulas/types
- These systems lead to type systems for $\lambda$-calculus, ensuring complexity properties
- W.r.t. other ICC approaches:
  - Handle higher-order computation
  - But limited intensional expressivity
- Relations with other ICC systems still to explore