Logarithmic space queries and regular transductions in the elementary affine λ-calculus

Nguyễn Lê Thành Dũng (LIPN, Univ. Paris 13) – nltd@nguyentito.eu
partly based on j.w.w. Pierre Pradic (ENS Lyon & Univ. Warsaw)

Logic and Computational Complexity, July 8th, 2019, Patras

Implicit Computational Complexity

This is a work of *implicit computational complexity* (ICC):
we capture a complexity class via "*functional* programming"

(Whereas *descriptive* complexity $\approx$ *declarative* programming.)

Many techniques in ICC: function algebras, term rewriting, etc.

Implicit Computational Complexity

This is a work of *implicit computational complexity* (ICC):
we capture a complexity class via "*functional* programming"

(Whereas *descriptive* complexity $\approx$ *declarative* programming.)

Many techniques in ICC: function algebras, term rewriting, etc.

More precisely, we follow a *type theory* approach to ICC:

- Express functions in some λ-calculus / constructive logic
  $\longrightarrow$ same thing, via proof-as-programs correspondence!
- Indirectly enforce complexity constraint via *type system*

Type-theoretic Computational Complexity

Even in type-theoretic ICC, “there is more than 1 way to do it”!

We work at the junction between 2 distinct traditions:

1. *semantic evaluation* in the simply typed λ-calculus (STλ)
   (in particular, Hillebrand et al.’s work; also, System T, PCF...)
2. constrained variants of Girard’s *linear logic*

Type-theoretic Computational Complexity

Even in type-theoretic ICC, “there is more than 1 way to do it”!

We work at the junction between 2 distinct traditions:

1. *semantic evaluation* in the simply typed λ-calculus (STλ)
   (in particular, Hillebrand et al.’s work; also, System T, PCF...)
2. constrained variants of Girard’s *linear logic*

and look at 2 possible *input representations*:

1. Church-encoded strings $\to$ relationship with *automata*
   (see also: higher-order model checking)
2. *finite models* i.e. finite relational structures
   (as in descriptive complexity)

Some background on ICC in the simply typed λ-calculus

Expressivity of the simply typed λ-calculus

The *simply typed λ-calculus*:
a bare-bones functional programming language
no primitive data types, only *functions*

Types of "Church encodings" Str / Bool:
standard representations of strings / booleans as functions

$$\overline{abb} : (f_a, f_b) \mapsto f_a \circ f_b \circ f_b$$

What languages can we decide by programs taking Str to Bool?

Expressivity of the simply typed λ-calculus

The *simply typed λ-calculus*:
a bare-bones functional programming language
no primitive data types, only *functions*

Types of "Church encodings" Str / Bool:
standard representations of strings / booleans as functions

$$\overline{abb} : (f_a, f_b) \mapsto f_a \circ f_b \circ f_b$$

What languages can we decide by programs taking Str to Bool?

Theorem (Hillebrand & Kanellakis 1996)

The languages decided by simply typed programs of type $\mathsf{Str}[A] \to \mathsf{Bool}$ *are exactly the* regular *languages.*

($\mathsf{Str}[A]$ is Str with a type substitution)

Semantic evaluation and regular languages

Theorem (Hillebrand & Kanellakis 1996)

The languages decided by simply typed programs of type $\mathsf{Str}[A] \to \mathsf{Bool}$ *are exactly the* regular *languages.*

To prove $(\subseteq)$, we use a *semantics*:

- type $T \rightsquigarrow$ some mathematical structure $[\![T]\!]$
- program t of type $T \rightsquigarrow [\![t]\!] \in [\![T]\!]$
- if the program $p : \mathsf{Bool}$ returns $\mathtt{true}$, then $[\![p]\!] = [\![\mathtt{true}]\!]$

Semantic evaluation and regular languages

Theorem (Hillebrand & Kanellakis 1996)

The languages decided by simply typed programs of type $\mathsf{Str}[A] \to \mathsf{Bool}$ *are exactly the* regular *languages.*

To prove ($\subseteq$), we use a *semantics*:

- type $T \rightsquigarrow$ some mathematical structure $[\![T]\!]$
- program t of type $T \rightsquigarrow [\![t]\!] \in [\![T]\!]$
- if the program $p : \mathsf{Bool}$ returns `true`, then $[\![p]\!] = [\![\mathtt{true}]\!]$

Proof idea:
$w \in \Sigma^* \mapsto [\![\overline{w}]\!] \in [\![\mathsf{Str}[A]]\!]$ is a morphism to a finite monoid

So the simply typed λ-calculus doesn't seem too expressive...
Idea: *change input representation* to overcome limit.

The λ-calculus meets finite model theory

So the simply typed λ-calculus doesn't seem too expressive...
Idea: *change input representation* to overcome limit.

Theorem (Hillebrand, Kanellakis & Mairson 1993)

With the right choice of encoding, the class of queries *expressible in the simply typed λ-calculus* over finite models *is* ELEMENTARY.

The λ-calculus meets finite model theory

So the simply typed λ-calculus doesn't seem too expressive...
Idea: *change input representation* to overcome limit.

Theorem (Hillebrand, Kanellakis & Mairson 1993)

With the right choice of encoding, the class of queries *expressible in the simply typed λ-calculus* over finite models *is* ELEMENTARY.

One can be more precise:
characterizations of k-EXPTIME / k-EXPSPACE queries

- expressivity upper bounds proved via semantics as before
- lower bounds proved using descriptive complexity

Transducers in the λ-calculus

The road not taken by Hillebrand et al.:
embrace connections between Str type and automata theory

Open question

What functions $\Gamma^* \to \Sigma^*$ can be expressed in the simply typed λ-calculus by programs of type $\mathsf{Str}_\Gamma[A] \to \mathsf{Str}_\Sigma$?

Corollary (of characterization of $\mathsf{Str}_\Sigma[A] \to \mathsf{Bool}$)

For such a function $f : \Gamma^* \to \Sigma^*$,
$L \subseteq \Sigma^*$ regular $\implies f^{-1}(L)$ regular.

$\longrightarrow$ look for an answer in the theory of *transducers*.

The elementary affine λ-calculus

ICC in linear logic

Via formulæ-as-types / proofs-as-programs correspondence,
Linear logic $\approx$ type system with explicit control on *duplication*:

- *linear* function: $A \multimap B$ uses A *once* to produce B
- $A \multimap A \otimes A$ is not valid
- $!A \multimap !A \otimes !A$ is valid: '!' marks duplicable resources

ICC in linear logic

Via formulæ-as-types / proofs-as-programs correspondence,
Linear logic $\approx$ type system with explicit control on *duplication*:

- *linear* function: $A \multimap B$ uses A *once* to produce B
- $A \multimap A \otimes A$ is not valid
- $!A \multimap !A \otimes !A$ is valid: '!' marks duplicable resources

$\longrightarrow$ Restrictions on '!' have complexity effects.
Elementary Linear Logic: LL w/o $!A \multimap A$ and $!A \multimap !!A$

ICC in linear logic

Via formulæ-as-types / proofs-as-programs correspondence,
Linear logic ≈ type system with explicit control on *duplication*:

- *linear* function: $A \multimap B$ uses A *once* to produce B
- $A \multimap A \otimes A$ is not valid
- $!A \multimap !A \otimes !A$ is valid: '!' marks duplicable resources

$\longrightarrow$ Restrictions on '!' have complexity effects.
Elementary Linear Logic: LL w/o $!A \multimap A$ and $!A \multimap !!A$

Theorem (Girard 1998)

Programs of type $!\mathsf{Str} \multimap !^k\mathsf{Bool}$ *in second-order ELL (for varying* $k \in \mathbb{N}$*) correspond to elementary recursive predicates.*

ICC in a linear type system (1)

Programming language counterpart / variant of 2nd-order ELL: the *elementary affine λ-calculus* (EAλ).

Theorem (Baillot et al. 2014)

Programs of type $!\mathsf{Str} \multimap !^k\mathsf{Bool}$ *in* EAλ *(for varying* $k \in \mathbb{N}$*) correspond to elementary recursive predicates.*

ICC in a linear type system (1)

Programming language counterpart / variant of 2nd-order ELL: the *elementary affine λ-calculus* (EAλ).

Theorem (Baillot et al. 2014)

Programs of type $!\mathsf{Str} \multimap !^k\mathsf{Bool}$ *in* EAλ (*for varying* $k \in \mathbb{N}$) *correspond to elementary recursive predicates.*

Theorem (Baillot et al. 2014)

For fixed $k \in \mathbb{N}$, *the programs of type* $!\mathsf{Str} \multimap !^{k+2}\mathsf{Bool}$ *in* EAλ with type fixpoints *correspond to* k-EXPTIME *predicates.*
In particular $!\mathsf{Str} \multimap !!\mathsf{Bool}$ *corresponds to* P.

More precise, but uses type fixpoints (a.k.a. recursive types).

ICC in a linear type system (2)

Theorem (Baillot et al. 2014)

In the elementary affine λ-calculus (EAλ) with *type fixpoints, the programs of type* $!\mathsf{Str} \multimap !!\mathsf{Bool}$ *correspond to* P.

What if we remove type fixpoints?

ICC in a linear type system (2)

Theorem (Baillot et al. 2014)

In the elementary affine λ-calculus (EAλ) with *type fixpoints, the programs of type* $!\mathsf{Str} \multimap !!\mathsf{Bool}$ *correspond to* P.

What if we remove type fixpoints?

Theorem

In EAλ without *type fixpoints, the programs of type* $!\mathsf{Str} \multimap !!\mathsf{Bool}$ *correspond to* regular languages.

Proof adapted from Hillebrand and Kanellakis's theorem.
Key tool: second-order LL without '!' has a finite semantics.

(Previous work on LL-based ICC did not use semantics, unlike the simply typed λ-calculus tradition.)

Finite models as inputs

Queries in EAλ (ICALP 2019 paper with P. Pradic)

Taking inspiration from the simply typed λ-calculus,
we define a type Inp of inputs as finite models in EAλ.

Queries computed by programs of type $\mathsf{Inp} \multimap\ !!\mathsf{Bool}$ should:

- go beyond regular languages
- still be below P (the P soundness proof still holds)

Queries in EAλ (ICALP 2019 paper with P. Pradic)

Taking inspiration from the simply typed λ-calculus,
we define a type Inp of inputs as finite models in EAλ.

Queries computed by programs of type $\mathsf{Inp} \multimap !!\mathsf{Bool}$ should:

- go beyond regular languages ($\subset \mathsf{NC}^1$)
- still be below P (the P soundness proof still holds)

What we obtain – unexpectedly – is *logarithmic space*:

- lower bound: all L queries expressible (w/ linear orders)
- upper bound: NL
- encouraging partial results towards L upper bound
  - exact characterization of L with ad-hoc restriction on $\exists$-intro

Queries in EAλ (ICALP 2019 paper with P. Pradic)

Taking inspiration from the simply typed λ-calculus,
we define a type Inp of inputs as finite models in EAλ.

Queries computed by programs of type $\mathsf{Inp} \multimap\ !!\mathsf{Bool}$ should:

- go beyond regular languages ($\subset \mathsf{NC}^1$)
- still be below P (the P soundness proof still holds)

What we obtain – unexpectedly – is *logarithmic space*:

- lower bound: all L queries expressible (w/ linear orders)
- upper bound: NL (actually L^{UL}, UL = *unambiguous* NL)
- encouraging partial results towards L upper bound
    - exact characterization of L with ad-hoc restriction on $\exists$-intro

Linearity in descriptive complexity

Recall that over finite models with linear orders:

Theorem (Immerman 1983)

L = *queries in FO* + **DTC (deterministic transitive closure)**.

Linearity in descriptive complexity

Recall that over finite models with linear orders:

Theorem (Immerman 1983)

L = *queries in FO* + **DTC (deterministic transitive closure)**.

How to compute (deterministic) transitive closure in EAλ?

$\psi_R : Q \mapsto \{(x,z) \mid x = z \lor (\exists y : (x,y) \in R \land (y,z) \in Q)\}$
$R^* =$ *least fixpoint* of ψ_R, obtained by iteration (ψ_R monotone).

Problem: ψ_R is non-linear (because of $\exists y$), and the Inp type in EAλ only allows us to iterate *linear* maps.

Linearity in descriptive complexity

Recall that over finite models with linear orders:

Theorem (Immerman 1983)

L = *queries in FO* + **DTC (deterministic transitive closure)**.

How to compute (deterministic) transitive closure in EAλ?

$\psi_R : Q \mapsto \{(x, z) \mid x = z \vee (\exists y : (x, y) \in R \wedge (y, z) \in Q)\}$
R^* = *least fixpoint* of ψ_R, obtained by iteration (ψ_R monotone).

Problem: ψ_R is non-linear (because of $\exists y$), and the Inp type in EAλ only allows us to iterate *linear* maps.

For DTC, we have: $\psi_{R_\mathrm{d}} : Q \mapsto \{(x, z) \mid x = z \vee (f_R(x), z) \in Q\}$
(f_R partial function) which is linear in Q

$\longrightarrow$ *Determinism* (L vs NL) corresponds to *linearity*.

Church-encoded strings as both inputs and outputs

Transductions in EAλ (1)

Theorem

In EAλ without *type fixpoints, the programs of type* $!\mathsf{Str} \multimap !!\mathsf{Bool}$ *correspond to* regular languages.

Corollary

All functions computed by programs of type $!\mathsf{Str} \multimap !\mathsf{Str}$ *preserve regular languages by inverse image.*

Can we do better?

Transductions in EAλ (1)

Theorem

In EAλ without *type fixpoints, the programs of type* $!\mathsf{Str} \multimap !!\mathsf{Bool}$ *correspond to* regular languages.

Corollary

All functions computed by programs of type $!\mathsf{Str} \multimap !\mathsf{Str}$ *preserve regular languages by inverse image.*

Can we do better? Yes: they correspond to some class X with

$$\text{regular functions} \subsetneq X \subseteq \text{polyregular functions}$$

Regular functions: well-known class of transductions
Polyregular functions: Bojańczyk 2018 (`arXiv:1810.08760`)

Transductions in EAλ (1)

In fact, the previous result is a consequence of:

Theorem

The functions computed by EAλ *programs of type* $\mathsf{Str} \multimap \mathsf{Str}$ *are exactly the* regular functions.

Transductions in EAλ (1)

In fact, the previous result is a consequence of:

Theorem

The functions computed by EAλ *programs of type* $\mathsf{Str} \multimap \mathsf{Str}$ *are exactly the* regular functions.

Many characterizations of regular functions:

- *two-way* finite state transducers (automata with output)
- Monadic Second-Order Logic
- basic functions + combinators (at least 3 possible variants)
- register transducers with “copyless assignment” condition

Transductions in EAλ (1)

In fact, the previous result is a consequence of:

Theorem

The functions computed by EAλ *programs of type* $\mathsf{Str} \multimap \mathsf{Str}$ *are exactly the* regular functions.

Many characterizations of regular functions:

- *two-way* finite state transducers (automata with output)
- Monadic Second-Order Logic
- basic functions + combinators (at least 3 possible variants)
- register transducers with “copyless assignment” condition
  $\longrightarrow$ none other than a *linearity* condition!
  this is what we use to prove the theorem

Conclusion

We started out by drawing upon different streams of ideas in type-theoretic ICC.

This led us to connections with

- descriptive complexity,
- automata theory.

It turns out that linearity conditions occur in both of these!

New characterizations of

- hopefully, logarithmic space (still conjectural)
- regular functions on strings