

# Implicit Computational Complexity: A $\mu$ -Tutorial

Ugo Dal Lago    Simone Martini

Dipartimento di Scienze dell'Informazione  
Università di Bologna

FOLLIA meeting, January 19th 2006

# Outline

What is Implicit Computational Complexity

Model Theory

Recursion Theory

Proof Theory

From Logic to Programming Languages

From Logic to Computational Complexity

Challenges

# Logical Characterization of Complexity Classes

- ▶ Implicit Computational Complexity aims at defining machine-free characterizations of complexity classes based on Mathematical Logic.
- ▶ Various branches of Mathematical Logic have been touched by Implicit Computational Complexity:
  - ▶ **Recursion Theory**;
  - ▶ **Proof Theory** (Curry-Howard correspondence);
  - ▶ **Model Theory** (Finite Model Theory).
- ▶ We are mainly interested in proof-theoretical and recursion-theoretical characterizations.
  - ▶ They induce programming languages with bounded expressive power.
  - ▶ Resource-aware type-systems for existing programming languages can be designed from some of these systems.

# Logical Characterization of Complexity Classes

- ▶ Implicit Computational Complexity aims at defining machine-free characterizations of complexity classes based on Mathematical Logic.
- ▶ Various branches of Mathematical Logic have been touched by Implicit Computational Complexity:
  - ▶ **Recursion Theory**;
  - ▶ **Proof Theory** (Curry-Howard correspondence);
  - ▶ **Model Theory** (Finite Model Theory).
- ▶ We are mainly interested in proof-theoretical and recursion-theoretical characterizations.
  - ▶ They induce programming languages with bounded expressive power.
  - ▶ Resource-aware type-systems for existing programming languages can be designed from some of these systems.

# Logical Characterization of Complexity Classes

- ▶ Implicit Computational Complexity aims at defining machine-free characterizations of complexity classes based on Mathematical Logic.
- ▶ Various branches of Mathematical Logic have been touched by Implicit Computational Complexity:
  - ▶ **Recursion Theory**;
  - ▶ **Proof Theory** (Curry-Howard correspondence);
  - ▶ **Model Theory** (Finite Model Theory).
- ▶ We are mainly interested in proof-theoretical and recursion-theoretical characterizations.
  - ▶ They induce programming languages with bounded expressive power.
  - ▶ Resource-aware type-systems for existing programming languages can be designed from some of these systems.

# Logical Characterization of Complexity Classes

- ▶ Implicit Computational Complexity aims at defining machine-free characterizations of complexity classes based on Mathematical Logic.
- ▶ Various branches of Mathematical Logic have been touched by Implicit Computational Complexity:
  - ▶ **Recursion Theory**;
  - ▶ **Proof Theory** (Curry-Howard correspondence);
  - ▶ **Model Theory** (Finite Model Theory).
- ▶ We are mainly interested in proof-theoretical and recursion-theoretical characterizations.
  - ▶ They induce programming languages with bounded expressive power.
  - ▶ Resource-aware type-systems for existing programming languages can be designed from some of these systems.

# Logical Characterization of Complexity Classes

- ▶ Implicit Computational Complexity aims at defining machine-free characterizations of complexity classes based on Mathematical Logic.
- ▶ Various branches of Mathematical Logic have been touched by Implicit Computational Complexity:
  - ▶ **Recursion Theory**;
  - ▶ **Proof Theory** (Curry-Howard correspondence);
  - ▶ **Model Theory** (Finite Model Theory).
- ▶ We are mainly interested in proof-theoretical and recursion-theoretical characterizations.
  - ▶ They induce programming languages with bounded expressive power.
  - ▶ Resource-aware type-systems for existing programming languages can be designed from some of these systems.

## Some Words About Finite Model Theory

- ▶ The restriction to finite models is not trivial.
- ▶ Many corollaries of completeness (for example, the compactness theorem) are blatantly wrong in the world of finite models.
- ▶ A set  $\mathcal{F}$  of finite models for a first-order language over a finite vocabulary can be considered as a language  $Enc(\mathcal{F})$ .
- ▶ The set of all finite models satisfying a  $\Sigma_1^1$  formula is called a generalized spectrum.

### Theorem (Fagin)

*Let  $\mathcal{F}$  be a set of finite models for a first-order language over a finite non-empty vocabulary. Suppose  $\mathcal{F}$  is closed under isomorphism. Then  $Enc(\mathcal{F})$  is in the complexity class **NPTIME** if and only if  $\mathcal{F}$  is a generalized spectrum.*

- ▶ Many other interesting results followed.

## Limiting primitive recursion

- ▶ Usual recursion—from  $f(n)$  to  $f(n+1)$ —is *exponentially* long on the *size* of the input  $n$ .
- ▶ Move to *binary* representation for input: *Recursion on Notation*:

$$f(0, \bar{y}) = g_0(\bar{y})$$

$$f(1, \bar{y}) = g_1(\bar{y})$$

$$f(2x, \bar{y}) = h_0(x, \bar{y}, f(x, \bar{y}))$$

$$f(2x+1, \bar{y}) = h_1(x, \bar{y}, f(x, \bar{y}))$$

- ▶ Which (of course) is not enough...

### Theorem

For every PRD  $f$  of arity  $n$ , there is a polynomial  $p : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$  such that whenever  $f(u_1, \dots, u_n) \rightarrow_m e$ , we have  $|e| \leq p(m, |u_1|, \dots, |u_n|)$ .

## Limiting primitive recursion

- ▶ Usual recursion—from  $f(n)$  to  $f(n+1)$ —is *exponentially* long on the *size* of the input  $n$ .
- ▶ Move to *binary* representation for input: *Recursion on Notation*:

$$f(0, \bar{y}) = g_0(\bar{y})$$

$$f(1, \bar{y}) = g_1(\bar{y})$$

$$f(2x, \bar{y}) = h_0(x, \bar{y}, f(x, \bar{y}))$$

$$f(2x+1, \bar{y}) = h_1(x, \bar{y}, f(x, \bar{y}))$$

- ▶ Which (of course) is not enough...

### Theorem

For every PRD  $f$  of arity  $n$ , there is a polynomial  $p : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$  such that whenever  $f(u_1, \dots, u_n) \rightarrow_m e$ , we have  $|e| \leq p(m, |u_1|, \dots, |u_n|)$ .

## Bounded Recursion on Notation

- ▶ An essential breakthrough towards complexity theory is due to Bennett (1962) and Cobham (1965).
- ▶ Polynomial time computable functions are extensionally equivalent to a function algebra closed under bounded recursion on notation.
- ▶ A function  $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$  is defined by bounded recursion on notation from  $g_0, g_1 : \mathbb{N}^n \rightarrow \mathbb{N}$ ,  $h_0, h_1 : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$  and  $k : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$  if

$$f(0, \bar{y}) = g_0(\bar{y})$$

$$f(1, \bar{y}) = g_1(\bar{y})$$

$$f(2x, \bar{y}) = h_0(x, \bar{y}, f(x, \bar{y}))$$

$$f(2x + 1, \bar{y}) = h_1(x, \bar{y}, f(x, \bar{y}))$$

provided  $f(x, \bar{y}) \leq k(x, \bar{y})$ .

## Exploiting bounded recursion on notation

- ▶ Let  $x \# y = 2^{|x| \cdot |y|}$  (note:  $|x|^k = |x| \# \dots \# |x|$ ).

### Theorem (Cobham)

$$\mathcal{FPTIME} = [0, \Pi, s_0, s_1, \#; \text{COMP}, \text{BRN}]$$

- ▶ Let STRICT bounded recursion on notation (SBRN), be the same schema, but with *bound*  $f(x, \bar{y}) \leq |k(x, \bar{y})|$ .

### Theorem (Lind; Clote & Takeuti)

$$\mathcal{FLOGSPACE} = [0, \Pi, s_0, s_1, |x|, \text{BIT}, \#; \text{COMP}, \text{CRN}, \text{SBRN}]$$

where *Concatenation Recursion on Notation* (CRN) from  $g, h_0, h_1$  ( $h_i(x, \bar{y}) \leq 1$ ) is

$$\begin{aligned}f(0, \bar{y}) &= g_0(\bar{y}) \\f(1, \bar{y}) &= g_1(\bar{y}) \\f(2x, \bar{y}) &= s_{h_0(x, \bar{y})}(f(x, \bar{y})) \\f(2x + 1, \bar{y}) &= s_{h_1(x, \bar{y})}(f(x, \bar{y}))\end{aligned}$$

## Safe Recursion - I

- ▶ Unbounded recursion schemes to control the growth of functions
- ▶ Function arguments are partitioned into suitable classes.
- ▶ Function definitions are constrained to respect this partition.
- ▶ The first system pointing in this direction is due to Bellantoni and Cook.
- ▶ The arguments to a function  $f : \mathbb{N}^n \rightarrow \mathbb{N}$  are partitioned into  $m \leq n$  normal arguments and  $n - m$  safe arguments:

$$f(x_1, \dots, x_m; x_{m+1}, \dots, x_n)$$

### Theorem (Bellantoni and Cook)

*The set of polynomial time functions equals the set of functions definable by safe recursion.*

## Safe Recursion - II

- ▶ The function  $f$  is defined by **safe composition** from  $g, h_1, \dots, h_n, k_1, \dots, k_m$  if

$$f(\bar{x}; \bar{y}) = g(h_1(\bar{x}; \bar{y}), \dots, h_n(\bar{x}; \bar{y}); k_1(\bar{x}; \bar{y}), \dots, k_m(\bar{x}; \bar{y}))$$

- ▶ The function  $f$  is defined by **safe recursion on notation** from  $g_0, g_1, h_0, h_1$  if

$$f(0, \bar{x}; \bar{y}) = g_0(\bar{x}; \bar{y})$$

$$f(1, \bar{x}; \bar{y}) = g_1(\bar{x}; \bar{y})$$

$$f(2x, \bar{x}; \bar{y}) = h_0(x, \bar{x}; \bar{y}, f(x, \bar{x}; \bar{y}))$$

$$f(2x + 1, \bar{x}; \bar{y}) = h_1(x, \bar{x}; \bar{y}, f(x, \bar{x}; \bar{y}))$$

## Example: Controlling recursion by safeness

There is no safe recursion definition for the following exponential function  $e$

$$\begin{aligned}d(0; ) = d(1; ) &= 1 \\d(2x; ) = d(2x + 1; ) &= h(x; d(x)) \text{ where } h(x; y) = x * 100\end{aligned}$$

$$\begin{aligned}e(0) = e(1) &= 1 \\e(2x) = e(2x + 1) &= d(e(x))\end{aligned}$$

would need  $h(z; y) = d(y; )$ ,  
but we only obtain  $h(z; y) = d(; y)$

## Safe Affine Recursion: Logarithmic Space

- ▶ The function  $f$  is defined by **safe affine composition** from  $g, h_1, \dots, h_n, k_1, \dots, k_m$  if

$$f(\bar{x} : \bar{y}) = g(h_1(\bar{x} :), \dots, h_n(\bar{x} :)) : k_1(\bar{x}; \bar{y}_1), \dots, k_m(\bar{x}; \bar{y}_m))$$

where  $\bar{y}_1, \dots, \bar{y}_m$  is a partition of  $\bar{y}$ .

- ▶ The function  $f$  is defined by **safe affine course-of-value recursion** on notation from  $g_0, g_1, h_0, h_1$  if

$$f(0, \bar{x} : \bar{y}) = g_0(\bar{x} : \bar{y})$$

$$f(1, \bar{x} : \bar{y}) = g_1(\bar{x} : \bar{y})$$

$$f(2x, \bar{x} : \bar{y}) = h_0(x, \bar{x} : f(x', \bar{x} : \bar{y}))$$

$$f(2x + 1, \bar{x} : \bar{y}) = h_1(x, \bar{x} : f(x'', \bar{x} : \bar{y})) \text{ with } x', x'' \leq x$$

### Theorem (Mairson and Neergaard)

*The set of logarithmic space functions equals the set of functions definable by safe affine course-of-value recursion.*

# Tiering - I

- ▶ Related to safe recursion is the notion of tiering.
- ▶ Any function and argument position comes with a *tier*.
- ▶ Base functions are available at any tier.
- ▶ Composition is tier-preserving:  $f^i \circ g^i = h^i$ .
- ▶ Recursion is possible only over a variable with tier greater than that of the function:

$$f(0, y)^i = g_0(y^k)^i$$

$$f(1, y)^i = g_1(y^k)^i$$

$$f(2x, y)^i = h_0(x^l, y^k, f(x, y)^i)^i$$

$$f(2x + 1, y)^i = h_1(x^l, y^k, f(x, y)^i)^i \text{ with } l > m$$

## Tiering - II

- ▶ Tiering and safeness are equivalent
  - ▶ From a tiered  $f(x_1^{l_1}, \dots, x_n^{l_n}, y_1^i, \dots, y_m^i)^i$  where  $l_1, \dots, l_n > i$  we get  $f(x_1, \dots, x_n; y_1, \dots, y_m)$
  - ▶ From a safe definition  $f(x_1, \dots, x_n; y_1, \dots, y_m)$  for any tier  $i$ , there is a tiered definition of  $f$  in which  $f(x_1^{l_1}, \dots, x_n^{l_n}, y_1^i, \dots, y_m^i)^i$  with  $l_1, \dots, l_n > i$
- ▶ Tiering has been exploited to characterize:
  - ▶ **Polynomial Time** (Leivant)
  - ▶ **Polynomial Space** (Leivant and Marion, Oitavem)
  - ▶ **Alternating Logarithmic Time** (Leivant and Marion)

# First-order Arithmetic and Computational Complexity

- ▶ Bounded arithmetic is an arithmetical system, introduced by Buss, whose provably recursive functions are exactly the polynomial time computable functions.
- ▶ Usual quantifiers on natural numbers are replaced by their bounded versions.
- ▶ Many other systems characterizing polynomial time in the same sense, but avoiding explicit bounding conditions, have been proposed by Nelson, Leivant, Bellantoni and other authors.
- ▶ Larger computational complexity classes (e.g. elementary time computable functions) have been found to correspond to stronger arithmetical theories by Leivant and Weiner.

# Typed Lambda-Calculi: Simple Types

- ▶ What is the class of functions captured by typed lambda-calculi?
- ▶ With simple types, the class of representable functions are strongly influenced by the underlying coding scheme:
  - ▶ If we fix normal forms for  $A_0 = (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$  to be the only legal encoding of numerals, then the class of representable functions is very small (coincides with the class of extended polynomials)
  - ▶ If we relax this constraint, for example by allowing alternative encodings for numerals as normal forms for  $A_1 = (A_0 \rightarrow A_0) \rightarrow (A_0 \rightarrow A_0)$ , we increase the class of representable functions.
  - ▶ Going to the limit, we find that normalization of a generic simply-typed lambda-term is not even elementary in the size of the term being normalized.
- ▶ To fix this problem, we can either introduce **constants** (and primitive recursion) or **second order quantification**

## Typed Lambda-Calculi: Higher-Order Recursion

- ▶ Higher-order generalizations of Leivant's ramified recurrence captures elementary time computable functions (Leivant, Bellantoni et al., Dal Lago et al.)
- ▶ Polynomial time can be retrieved by constraining higher-order variables to be used in a linear way (Hofmann).
- ▶ Non-size increasing polytime computation is a calculus for polynomial time functions which uses a stricter notion of linearity, but without any ramification condition (Hofmann).
- ▶ Characterizations of major complexity classes can be obtained using syntactical (but not type-theoretical) constraints on lambda-calculi with higher-type recursion (Leivant).

# Typed Lambda-Calculi: Higher-Order Recursion II

- ▶ Uniform characterizations of complexity classes by subsystems of System **T** can be obtained by fine-tuning ramification and linearity (Dal Lago):

	<b>T</b>	<b>A</b>	<b>W</b>	$\emptyset$
<b>H</b>	PA	PR	PR	PR
<b>RH</b>	E	E	P	P

# Typed Lambda-Calculi: Second-Order Quantification

- ▶ How can we restrict the expressive power of Second-Order Lambda Calculus?
- ▶ There are characterizations of elementary time by fragments of System F (Aehlig and Johannsen).
- ▶ Isolating subsystems characterizing smaller complexity classes, on the other hand, is more difficult...

## Linear Logic

- ▶ The Curry-Howard Correspondence comes into play.
- ▶ Linear Logic can be seen as a way to decompose  $A \leftarrow B$  into  $!A \multimap B$ .
- ▶  $\multimap$  is the an arrow operator.
- ▶  $\otimes$  is the a conjunction operator.
- ▶  $!$  is a new operator governed by the following rules:

$$!A \cong !A \otimes !A$$

$$!A \otimes !B \cong !(A \otimes B)$$

$$!A \multimap !!A$$

$$!A \multimap A$$

# Linear Logic

Subsystems...

	$!A \otimes !B \cong !(A \otimes B)$	$!A \multimap !!A$	$!A \multimap A$	$!A \cong !A \otimes !A$
<b>ELL</b>	YES	NO	NO	YES
<b>LLL</b>	NO	NO	NO	YES
<b>SLL</b>	YES	NO	$!A \multimap A \otimes \dots \otimes A$	

...and their expressive power

<b>ELL</b>	$E$
<b>LLL</b>	$P$
<b>SLL</b>	$P$

# Linear Logic

Subsystems...

	$!A \otimes !B \cong !(A \otimes B)$	$!A \multimap !!A$	$!A \multimap A$	$!A \cong !A \otimes !A$
<b>ELL</b>	YES	NO	NO	YES
<b>LLL</b>	NO	NO	NO	YES
<b>SLL</b>	YES	NO	$!A \multimap A \otimes \dots \otimes A$	

...and their expressive power

<b>ELL</b>	$E$
<b>LLL</b>	$P$
<b>SLL</b>	$P$

# From Logic to Programming Languages

- ▶ How can a host machine assure the amount of resource needed to run a mobile program? A resource-aware type system or program-logic would provide implicit and verifiable certificates.
- ▶ In the realm of (first-order) term-rewriting systems, techniques like **quasi interpretations** have been shown to be useful for inferring complexity properties of programs (Bonfante et al.).
- ▶ **Type-systems** derived from non-size increasing computations have been exploited in the context of mobile resource guarantees (Hofmann et al., Beringer et al.).
- ▶ Enforcing resource-awareness in programming languages is not an easy task. The additional control provided cannot come at the price of unacceptable restrictions to programs.

# Inferring Linear Bounds on Heap Size – Hofmann & Jost

- ▶ **Language:** first-order functional programming language with recursion.
- ▶ **Type-system:** simple types, including lists, with resource annotations.
- ▶ **Example:**  $x : L(B, 2), 3 \vdash e : L(B, 4), 5$   
means
  - ▶ if we evaluate  $e$  starting with  $x$  bound to a list  $[u_1, \dots, u_m]$ ,
  - ▶ and we have a free-list of at least  $3 + 2m$  cells,
  - ▶ then the computation will not get stuck from insufficient memory availability;
  - ▶ moreover, if the result is a list  $[v_1, \dots, v_n]$ , then at the end the free-list will have at least  $5 + 4n$  cells.

- ▶ **Type-system:** Contraction can only be done splitting the corresponding resource annotations: for example, from

$$x : L(B,3), y : L(B,6) \vdash e : C, 7$$

we can derive

$$z : L(B,9) \vdash e\{z/x, z/y\} : C, 7$$

- ▶ **Decorations:** given a skeleton of a type derivation (types, but not resource annotations) for  $e$ , a set of linear inequalities  $\mathcal{L}(e)$  is derived. Solutions to  $\mathcal{L}(e)$  are in one-to-one correspondence with valid type derivations for  $e$ .

# From Logic to Computational Complexity

- ▶ Programming languages can be designed so that functions computable by acceptable programs extensionally correspond to certain computation complexity classes.
- ▶ If the underlying programming language is reasonably abstract, the system is then a machine-free characterization of a complexity class and can be used to infer properties of that same class.
- ▶ If we want to infer properties of a complexity class from properties of a certain system (which exactly characterizes it), we should keep the system as simple as possible, without emphasizing issues such as programming flexibility.

# Challenges

- ▶ The area of implicit computational complexity appears very fragmented, with many different proposals.
- ▶ It is very difficult to compare relative **intensional expressive power**.
- ▶ It is not usually the case a system can be **extended** with new features preserving its quantitative properties
- ▶ Defining just another characterization of polynomial time is not enough.
- ▶ Deep, **foundational results** are extremely needed.

Questions?