From Implicit Complexity to Quantitative Resource Analysis

Overview

Ugo Dal Lago

PhD Open, University of Warsaw, June 25-27, 2015
About This Course

- Introduction to Implicit Computational Complexity.
  - Approximately 5 hours.
  - This slideset.
  - Exercise sessions along the way.

- Complexity Analysis by Program Transformation.
  - Approximately 1 hour.

- Bounded Linear Logic.
  - Approximately 1 hour.

- Website: http://www.cs.unibo.it/~dallago/FICQRA/

- Email: ugo.dallago@unibo.it
  - Please contact me whenever you want!

- Exam: there will be one one week after the end of the Tutorial
About This Course

- Introduction to Implicit Computational Complexity.
  - Approximately 5 hours.
  - This slideset.
  - Exercise sessions along the way.
- Complexity Analysis by Program Transformation.
  - Approximately 1 hour.
- Bounded Linear Logic.
  - Approximately 1 hour.
- Website: http://www.cs.unibo.it/~dallago/FICQRA/
- Email: ugo.dallago@unibo.it
  - Please contact me whenever you want!
- Exam: there will be one one week after the end of the Tutorial
About This Course

- Introduction to Implicit Computational Complexity.
  - Approximately 5 hours.
  - This slideset.
  - Exercise sessions along the way.
- Complexity Analysis by Program Transformation.
  - Approximately 1 hour.
- Bounded Linear Logic.
  - Approximately 1 hour.

- Website: http://www.cs.unibo.it/~dallago/FICQRA/
- Email: ugo.dallago@unibo.it
  - Please contact me whenever you want!
- Exam: there will be one one week after the end of the Tutorial.
About This Course

- Introduction to Implicit Computational Complexity.
  - Approximately 5 hours.
  - This slideset.
  - Exercise sessions along the way.
- Complexity Analysis by Program Transformation.
  - Approximately 1 hour.
- Bounded Linear Logic.
  - Approximately 1 hour.
- Website: http://www.cs.unibo.it/~dallago/FICQRA/
- Email: ugo.dallago@unibo.it
  - Please contact me whenever you want!
- Exam: there will be one one week after the end of the Tutorial
Goal

- Machine-free characterizations of complexity classes.
- No explicit reference to resource bounds.
- \( P, \ PSPACE, \ L, \ NC, \ldots \)

Why?

- Simple and elegant presentations of complexity classes.
- Formal methods for complexity analysis of programs.

How?

- First-order functional programs [BC92], [Leivant94], \ldots
- Model theory [Fagin73], \ldots
- Type Systems and \( \lambda \)-calculi [Hofmann97], [Girard97], \ldots
- \ldots
Implicit Computational Complexity

- **Goal**
  - Machine-free characterizations of complexity classes.
  - No explicit reference to resource bounds.
  - P, PSPACE, L, NC,...

- **Why?**
  - Simple and elegant *presentations* of complexity classes.
  - *Formal methods* for complexity analysis of programs.

- **How?**
  - First-order functional programs [BC92], [Leivant94], ...
  - Model theory [Fagin73], ...
  - Type Systems and λ-calculi [Hofmann97], [Girard97], ...
  - ...
Implicit Computational Complexity

Goal

- Machine-free characterizations of complexity classes.
- No explicit reference to resource bounds.
- $P$, $PSPACE$, $L$, $NC$, ...

Why?

- Simple and elegant presentations of complexity classes.
- Formal methods for complexity analysis of programs.

How?

- First-order functional programs [BC92], [Leivant94], ...
- Model theory [Fagin73], ...
- Type Systems and $\lambda$-calculi [Hofmann97], [Girard97], ...
- ...
Part I

Implicit Complexity at a Glance
Characterizing Complexity Classes

Programs

\mathcal{L}

Functions

\mathcal{C}
Characterizing Complexity Classes

Programs \( \mathcal{L} \) \rightarrow \text{Functions} \( \mathcal{C} \)
Characterizing Complexity Classes

Programs $\mathcal{L}$

Functions $\mathcal{C}$ $\mathcal{P}$
Characterizing Complexity Classes

Programs

\[ L \]

\[ S \]

Functions

\[ C \]

\[ P \]
Proving $[S] = \mathcal{P}$

- $\mathcal{P} \subseteq [S]$
  - For every function $f$ which *can* be computed within the bounds prescribed by $\mathcal{P}$, there is $P \in S$ such that $[P] = f$.

- $[S] \subseteq \mathcal{P}$
  - Semantically
    - For every $P \in S$, $[P] \in \mathcal{P}$ is proved by showing that some algorithms computing $[P]$ exists which works within the prescribed resource bounds.
    - $P \in \mathcal{L}$ does *not* necessarily exhibit a nice computational behavior.
    - Example: soundness by realizability [DLHofmann05].
  - Operationally
    - Sometimes, $\mathcal{L}$ can be endowed with an effective operational semantics.
    - Let $\mathcal{L}_\mathcal{P} \subseteq \mathcal{L}$ be the set of those programs which work within the bounds prescribed by $\mathcal{C}$.
    - $[S] \subseteq \mathcal{P}$ can be shown by proving $S \subseteq \mathcal{L}_\mathcal{P}$. 
Proving $[\mathcal{S}] = \mathcal{P}$

- $\mathcal{P} \subseteq [\mathcal{S}]$
  - For every function $f$ which can be computed within the bounds prescribed by $\mathcal{P}$, there is $P \in \mathcal{S}$ such that $[P] = f$.

- $[\mathcal{S}] \subseteq \mathcal{P}$
  - **Semantically**
    - For every $P \in \mathcal{S}$, $[P] \in \mathcal{P}$ is proved by showing that some algorithms computing $[P]$ exists which works within the prescribed resource bounds.
    - $P \in \mathcal{L}$ does not necessarily exhibit a nice computational behavior.
    - Example: soundness by realizability [DLHofmann05].

- **Operationally**
  - Sometimes, $\mathcal{L}$ can be endowed with an effective operational semantics.
  - Let $\mathcal{L}_\mathcal{P} \subseteq \mathcal{L}$ be the set of those programs which work within the bounds prescribed by $\mathcal{C}$.
  - $[\mathcal{S}] \subseteq \mathcal{P}$ can be shown by proving $\mathcal{S} \subseteq \mathcal{L}_\mathcal{P}$.
If Soundness is Proved Operationally...
If Soundness is Proved Operationally...
ICC Systems as Static Analyzers

$P \in \mathcal{L}$

$S$

\[\begin{cases}
\text{Yes, } P \in \mathcal{L}\mathcal{P} \\
\text{Don't know}
\end{cases}\]
ICC Systems as Static Analyzers

\[ P \in \mathcal{L} \]

\( S \)

\[ \begin{cases} 
\text{Yes, } P \in \mathcal{LP} + \text{ bounds} \\
\text{Don’t know} 
\end{cases} \]
ICC: Intensional Expressive Power
ICC: Intensional Expressive Power
ICC: Intensional Expressive Power
ICC: Intensional Expressive Power
ICC: Intensional Expressive Power
Safe Recursion [BC92]
Light Linear Logic [Girard97]
Bounded Recursion on Notation [Cobham63]
Bounded Arithmetic [Buss80]
Part II

Playing with Computational Models as Languages
Preliminaries

- **Alphabet**: a finite nonempty set, denoted with meta-variables like \( \Sigma, \Upsilon, \Phi \).

- **String** over an alphabet \( \Sigma \): a finite, ordered, possibly empty sequence of symbols from \( \Sigma \). The *Kleene’s closure* \( \Sigma^* \) of \( \Sigma \) is the set of all words over \( \Sigma \), including the empty word \( \varepsilon \).

- **Language** over the alphabet \( \Sigma \): a subset of \( \Sigma^* \).
One way of defining a language $\mathcal{L}$ is by saying that $\mathcal{L}$ is the smallest set satisfying some closure conditions, formulated as a set of productions.

**Example:** the language $\mathcal{P}$ of palindrome words over the alphabet $\{a, b\}$ can be defined as follows

$$W ::= \varepsilon \mid a \mid b \mid aWa \mid bWb.$$
Preliminaries

One way of defining a language $\mathcal{L}$ is by saying that $\mathcal{L}$ is the smallest set satisfying some closure conditions, formulated as a set of productions.

Example: the language $\mathcal{P}$ of palindrome words over the alphabet $\{a, b\}$ can be defined as follows

$$W ::= \varepsilon \mid a \mid b \mid aW a \mid bW b.$$

Example: the language $\mathcal{N}$ of all sequences in $\{0, \ldots, 9\}^*$ denoting natural numbers. It will be ranged over by meta-variables like $N, M$. Given $N$, $[N] \in \mathbb{N}$ is the natural number denoted by $N$.

Example: the language $\mathcal{B}$ of all binary strings, namely $\{0, 1\}^*$. 
Counter Programs:

\[ P ::= I \mid I, P \]
\[ I ::= \text{inc}(N) \mid \text{jmpz}(N,N) \]

where \( N \) ranges over \( \mathbb{N} \) and thus stands for any natural number in decimal notation, e.g. 12, 0, 67123.

Example:

\[ \text{jmpz}(0,4), \text{inc}(1), \text{jmpz}(2,1) \]
Counter Programs:

\[ P ::= I \mid I, P \]

\[ I ::= \text{inc}(N) \mid \text{jmpz}(N,N) \]

where \( N \) ranges over \( \mathbb{N} \) and thus stands for any natural number in decimal notation, e.g. 12, 0, 67123.

Example:

\[ \text{jmpz}(0,4), \text{inc}(1), \text{jmpz}(2,1) \]
Counter Programs

- Instructions modify the content of some registers $R_0, R_1, \ldots$, each containing a natural number.

- Formally:
  - The instruction $\text{inc}(N)$ increments by one the content of $R_n$ (where $n = \lceil N \rceil$ is the natural number denoted by $N$). The next instruction to be executed is the following one in the program.
  - The instruction $\text{jmpz}(N, M)$ determines the content $n$ of $R_{\lceil N \rceil}$ and
    - If $n$ is positive, decrements $R_{\lceil N \rceil}$ by one; the next instruction is the following one in the program.
    - If it is zero, the next instruction to be executed is the $\lceil M \rceil$-th in the program.
  - Whenever the next instruction to be executed cannot be determined following the two rules above, the execution of the program terminates.
Counter Programs

- Instructions modify the content of some registers $R_0, R_1, \ldots$, each containing a natural number.

- Formally:
  - The instruction $\text{inc}(N)$ increments by one the content of $R_n$ (where $n = \lceil N \rceil$ is the natural number denoted by $N$). The next instruction to be executed is the following one in the program.
  - The instruction $\text{jmpz}(N, M)$ determines the content $n$ of $R_{\lceil N \rceil}$ and
    - If $n$ is positive, decrements $R_{\lceil N \rceil}$ by one; the next instruction is the following one in the program.
    - If it is zero, the next instruction to be executed is the $\lceil M \rceil$-th in the program.
  - Whenever the next instruction to be executed cannot be determined following the two rules above, the execution of the program terminates.
### Counter Programs — Example

<table>
<thead>
<tr>
<th>Current Instruction</th>
<th>$R_0$</th>
<th>$R_1$</th>
<th>$R_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmpz(0,4)</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>inc(1)</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jmpz(2,1)</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>jmpz(0,4)</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>inc(1)</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>jmpz(2,1)</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>jmpz(0,4)</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>inc(1)</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>jmpz(2,1)</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>jmpz(0,4)</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>inc(1)</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>jmpz(2,1)</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>jmpz(0,4)</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
Turing Programs

- Turing Programs:

\[ P ::= I \mid I, P \]
\[ I ::= (N,c) > (N,a) \mid (N,c) > \text{stop} \]

where \( c \) ranges over the *input alphabet* \( \Sigma \) which includes a special symbol \( \square \) (called the *blank symbol*), and \( a \) ranges over the alphabet alphabet \( \Sigma \cup \{<,>\} \).
Intuitively, an instruction \((N, c) > (M, a)\) tells the machine to move to state \(M\) when the symbol under the head is \(c\) and the current state is \(N\):

- If \(a\) is in \(\Sigma\), than the symbol under the head is changed to \(a\);
- If \(a\) is \(<\), then the symbol \(c\) is left unchanged but the head moves one position leftward.
- Similarly when \(a\) is \(>\).
- The instruction \((N, c) > \text{stop}\) tells the machine to simply stop whenever in state \(N\) and faced with the symbol \(c\).

The state of the program is composed of a state and of the state of the tape.
Turing Programs — Example

- A (deterministic) Turing program on \( \{0, 1\} \), called \( R \):
  
  \[
  (0, \square) \rightarrow (1, >), \quad (1, 0) \rightarrow (2, 1), \quad (1, 1) \rightarrow (2, 0), \\
  (2, 0) \rightarrow (1, >), \quad (2, 1) \rightarrow (1, >), \quad (1, \square) \rightarrow \text{stop}
  \]

- Execution:
  
  \[
  (\varepsilon, \square, 010, 0) \xrightarrow{R} (\square, 0, 10, 1) \xrightarrow{R} (\square, 1, 10, 2) \\
  \xrightarrow{R} (\square1, 1, 0, 1) \xrightarrow{R} (\square1, 0, 0, 2) \\
  \xrightarrow{R} (\square10, 0, \varepsilon, 1) \xrightarrow{R} (\square10, 1, \varepsilon, 2) \\
  \xrightarrow{R} (\square101, \square, \varepsilon, 1).
  \]
Turing Programs — Example

- A (deterministic) Turing program on \{0, 1\}, called \( R \):

\[
(0, □) \to (1, >), \quad (1, 0) \to (2, 1), \quad (1, 1) \to (2, 0), \quad (2, 0) \to (1, >), \quad (2, 1) \to (1, >), \quad (1, □) \to \text{stop}
\]

- Execution:

\[
(ε, □, 010, 0) \xrightarrow{R} (□, 0, 10, 1) \xrightarrow{R} (□, 1, 10, 2) \\
\quad \xrightarrow{R} (□1, 1, 0, 1) \xrightarrow{R} (□1, 0, 0, 2) \\
\quad \xrightarrow{R} (□10, 0, ε, 1) \xrightarrow{R} (□10, 1, ε, 2) \\
\quad \xrightarrow{R} (□101, □, ε, 1).
\]
Another, different, way to capture computation is as follows:
- Start from a set of basic functions...
- ...and close it under some operators.

The set of recursive functions is the smallest set of functions which contains the basic functions and which is closed with respect to the operators.

Where are the programs?
- They are proofs of recursivity, which are finitary.
- They support an induction principle.
Another, different, way to capture computation is as follows:

- Start from a set of basic functions...
- ...and close it under some operators.

The set of recursive functions is the smallest set of functions which contains the basic functions and which is closed with respect to the operators.

Where are the programs?

- They are proofs of recursivity, which are finitary.
- They support an induction principle.
Function Algebras

- Another, different, way to capture computation is as follows:
  - Start from a set of basic functions...
  - ...and close it under some operators.
- The set of **recursive functions** is the smallest set of functions which contains the basic functions and which is closed with respect to the operators.
- Where are the **programs**?
  - They are *proofs of recursivity*, which are finitary.
  - They support an induction principle.
Zero. The function $z : \mathbb{N} \to \mathbb{N}$ defined as follows: $z(n) = 0$ for every $n \in \mathbb{N}$.

Successor. The function $s : \mathbb{N} \to \mathbb{N}$ defined as follows: $s(n) = n + 1$ for every $n \in \mathbb{N}$.

Projections. For every positive $n \in \mathbb{N}$ and for whenever $1 \leq m \leq n$, the function $\Pi^n_m : \mathbb{N}^n \to \mathbb{N}$ is defined as follows: $\Pi^n_m(k_1, \ldots, k_n) = k_m$. 
Composition. Suppose that $n \in \mathbb{N}$ is positive, that $f : \mathbb{N}^n \to \mathbb{N}$ and that $g_m : \mathbb{N}^k \to \mathbb{N}$ for every $1 \leq m \leq n$. Then the composition of $f$ and $g_1, \ldots, g_n$ is the function $h : \mathbb{N}^k \to \mathbb{N}$ defined as $h(\vec{i}) = f(g_1(\vec{i}), \ldots, g_n(\vec{i}))$.

Primitive Recursion. Suppose that $n \in \mathbb{N}$ is positive, that $f : \mathbb{N}^n \to \mathbb{N}$ and that $g : \mathbb{N}^{n+2} \to \mathbb{N}$. Then the function $h : \mathbb{N}^{n+1} \to \mathbb{N}$ defined as follows

\[
h(0, \vec{m}) = f(\vec{m});
\]
\[
h(k + 1, \vec{m}) = g(k, \vec{m}, h(k, \vec{m}));
\]

is said to be defined by primitive recursion form $f$ and $g$. 
Minimization. Suppose that $n \in \mathbb{N}$ is positive and that $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$. Then the function $g : \mathbb{N}^n \rightarrow \mathbb{N}$ defined as follows:

$$g(m, \vec{i}) = \begin{cases} k & \text{if } f(0, \vec{i}), \ldots, f(k, \vec{i}) \text{ are all defined and } f(k, \vec{i}) \text{ is the only one in the list being 0.} \\ \uparrow & \text{otherwise} \end{cases}$$

is said to be defined by minimization from $f$. 
**Signature.** It is a pair $\mathcal{S} = (\Sigma, \alpha)$ where:

- $\Sigma$ is an alphabet;
- $\alpha : \Sigma \rightarrow \mathbb{N}$ assigns to any symbol $c$ in $\Sigma$ a natural number $\alpha(c)$, called its *arity*.

**Examples.**

- The signature $\mathcal{S}_\mathbb{N}$ defined as $(\{0, s\}, \alpha_\mathbb{N})$ where $\alpha_\mathbb{N}(0) = 0$ and $\alpha_\mathbb{N}(s) = 1$.
- The signature $\mathcal{S}_\mathbb{B}$ defined as $(\{0, 1, e\}, \alpha_\mathbb{B})$ where $\alpha_\mathbb{B}(0) = \alpha_\mathbb{B}(1) = 1$ and $\alpha_\mathbb{B}(e) = 0$.

**Given two signatures $\mathcal{S}$ and $\mathcal{T}$ such that the underlying set of symbols are disjoint, one can naturally form the sum $\mathcal{S} + \mathcal{T}$.**
Closed Terms. Given a signature $S = (\Sigma, \alpha)$, they are the smallest set of expressions $\mathcal{C}(S)$ satisfying the following closure property: if $f \in \Sigma$ and $t_1, \ldots, t_{\alpha(f)} \in \mathcal{C}(S)$, then $f(t_1, \ldots, t_{\alpha(f)}) \in \mathcal{C}(S)$.

Examples.

$$\mathcal{C}(S_{\mathbb{N}}) = \{0, s(0), s(s(0)), \ldots\};$$
$$\mathcal{C}(S_{\mathbb{B}}) = \{e, 0(e), 1(e), 0(0(e)), \ldots\}.$$  

Open Terms. Given a set of variables $L$ distinct from $\Sigma$, the set of open terms $\mathcal{O}(S, L)$ is defined as the smallest set of words including $L$ and satisfying the closure condition above.
**Functional Programs** on $\mathcal{S} = (\Sigma, \alpha)$ and $\mathcal{T} = (\Upsilon, \beta)$:

$$
P ::= R \mid R , P
$$

$$
R ::= l \rightarrow t
$$

$$
l ::= f_1(p_1^1, \ldots , p_1^{\alpha(f_1)}) \mid \ldots \mid f_n(p_n^1, \ldots , p_n^{\alpha(f_n)})
$$

where:

- $\Sigma = \{f_1, \ldots , f_n\}$ and that $\Upsilon$ is disjoint from $\Sigma$.
- the metavariables $p_m^k$ range over $\mathcal{O}(\mathcal{T}, \mathcal{L})$,
- $t$ ranges over $\mathcal{O}(\mathcal{S} + \mathcal{T}, \mathcal{L})$. 
Example:

\[
\begin{align*}
\text{add}(0, x) & \rightarrow x \\
\text{add}(s(x), y) & \rightarrow s(\text{add}(x, y)).
\end{align*}
\]

The evaluation of \( \text{add}(s(s(0)), s(s(s(0)))) \) goes as follows:

\[
\begin{align*}
\text{add}(s(s(0)), s(s(s(0)))) & \rightarrow s(\text{add}(s(0), s(s(s(0))))) \\
& \rightarrow s(s(\text{add}(0, s(s(s(0))))) ) \\
& \rightarrow s(s(s(s(s(0)))))
\end{align*}
\]
\textbf{\Lambdan-Calculus}

\begin{itemize}
  \item **Terms:**
  \[ M ::= x \mid \lambda x. M \mid MM, \]
  \item The term obtained by \textit{substituting} a term \( M \) for a variable \( x \) into another term \( N \) is denoted as \( N\{M/x\} \).
  \item **Values:**
  \[ V ::= x \mid \lambda x. M. \]
  \item **Reduction:**
  \[
  \begin{align*}
  (\lambda x.M)V & \rightarrow_v M\{V/x\} \\
  ML & \rightarrow_v NL \\
  LM & \rightarrow_v LN
  \end{align*}
  \]
  Here \( M \) ranges over terms, while \( V \) ranges over values.
  \item **Normal Forms.** Any term \( M \) such that there is not any \( N \) with \( M \rightarrow_v N \). A term \( M \) has a normal form iff \( M \rightarrow^*_v N \). Otherwise, we write \( M \uparrow \).
\end{itemize}
Scott’s Numerals.

\[ 0^\downarrow = \lambda x.\lambda y.x; \]
\[ n + 1^\downarrow = \lambda x.\lambda y.y^\downarrow n^\downarrow. \]

Representing Functions. A \( \lambda \)-term \( M \) represents a function \( f : \mathbb{N} \rightarrow \mathbb{N} \) iff for every \( n \), if \( f(n) \) is defined and equals \( m \), then \( M^\uparrow n^\downarrow \rightarrow^* m^\downarrow \) and otherwise \( M^\uparrow n^\downarrow \uparrow \).
For the five introduced computational models, one can define the class of functions from \( \mathbb{N} \) to \( \mathbb{N} \) they compute. Unsurprisingly, they are all the same: the five models are all Turing-powerful. This means, in particular, that programs in the five formalisms can be, in general, very inefficient. They can compute whatever (computable) function one can imagine, after all!
Efficiency

- Programs consume resources (time, space, communication, energy).
- How could one formalize the concept of being efficient with respect to a given resource?

**A Measure**
- One can assign to any program $P$ a upper bound on the amount of resources (of a certain kind) $P$ needs when executed.
- Can be either precise or asymptotic.
- The more precise, the more architecture-dependent.

**A Predicate**
- The amount of resources $P$ needs when executed is bounded by a function in a “robust” class. Examples:
  - **Polynomial Functions**.
  - **Logarithmic Functions**.
  - **Elementary Functions**.
- We are mimicking complexity classes!
- If the predicate holds, extracting concrete asymptotic bounds is within reach.
Cost Models

- Let us focus our attention to *time* complexity.
- How do we *measure* the amount of time a computation takes in the five models we described?
  - **Counter Programs**: number of steps.
  - **Turing Programs**: number of steps.
  - **Functional Programs**: number of reduction steps?
  - **λ-Calculus**: number of reduction steps?
  - **Function Algebra**: ?

- **Invariance Thesis [SvEB1980]**: a cost model is *reasonable* iff the class of polytime computable functions is the same as the one defined with Turing programs.

- The most interesting cost models are, clearly, the unitary ones. They are not always invariant *by definition*, however.
Cost Models

- Let us focus our attention to *time* complexity.
- How do we *measure* the amount of time a computation takes in the five models we described?
  - **Counter Programs**: number of steps.
  - **Turing Programs**: number of steps.
  - **Functional Programs**: number of reduction steps?
  - **\(\lambda\)-Calculus**: number of reduction steps?
  - **Function Algebra**: ?

- **Invariance Thesis** [SvEB1980]: a cost model is *reasonable* iff the class of polytime computable functions is the same as the one defined with Turing programs.

- The most interesting cost models are, clearly, the unitary ones. They are not always invariant *by definition*, however.
Cost Models

- Let us focus our attention to *time* complexity.
- How do we *measure* the amount of time a computation takes in the five models we described?
  - **Counter Programs**: number of steps.
  - **Turing Programs**: number of steps.
  - **Functional Programs**: number of reduction steps?
  - **λ-Calculus**: number of reduction steps?
  - **Function Algebra**: ?

- **Invariance Thesis** [SvEB1980]: a cost model is *reasonable* iff the class of polytime computable functions is the same as the one defined with Turing programs.
- The most interesting cost models are, clearly, the unitary ones. They are not always invariant *by definition*, however.
Let us focus our attention to \textit{time} complexity.

How do we \textit{measure} the amount of time a computation takes in the five models we described?

- \textbf{Counter Programs}: number of steps.
- \textbf{Turing Programs}: number of steps.
- \textbf{Functional Programs}: number of reduction steps?
- \textbf{$\lambda$-Calculus}: number of reduction steps?
- \textbf{Function Algebra}: ?

\textbf{Invariance Thesis} [SvEB1980]: a cost model is \textit{reasonable} iff the class of polytime computable functions is the same as the one defined with Turing programs.

The most interesting cost models are, clearly, the unitary ones. They are not always invariant \textit{by definition}, however.
Let us focus our attention to *time* complexity.

How do we *measure* the amount of time a computation takes in the five models we described?

- **Counter Programs**: number of steps.
- **Turing Programs**: number of steps.
- **Functional Programs**: number of reduction steps?
- **λ-Calculus**: number of reduction steps?
- **Function Algebra**: ?

**Invariance Thesis** [SvEB1980]: a cost model is *reasonable* iff the class of polytime computable functions is the same as the one defined with Turing programs.

The most interesting cost models are, clearly, the unitary ones. They are not always invariant *by definition*, however.
Let us focus our attention to *time* complexity.

How do we *measure* the amount of time a computation takes in the five models we described?

- **Counter Programs**: number of steps.
- **Turing Programs**: number of steps.
- **Functional Programs**: number of reduction steps?
- **$\lambda$-Calculus**: number of reduction steps?
- **Function Algebra**: ?

**Invariance Thesis** [SvEB1980]: a cost model is *reasonable* iff the class of polytime computable functions is the same as the one defined with Turing programs.

The most interesting cost models are, clearly, the unitary ones. They are not always invariant *by definition*, however.
Let us focus our attention to *time* complexity.

How do we *measure* the amount of time a computation takes in the five models we described?

- **Counter Programs**: number of steps.
- **Turing Programs**: number of steps.
- **Functional Programs**: number of reduction steps?
- **λ-Calculus**: number of reduction steps?
- **Function Algebra**: ?

**Invariance Thesis** [SvEB1980]: a cost model is *reasonable* iff the class of polytime computable functions is the same as the one defined with Turing programs.

The most interesting cost models are, clearly, the unitary ones. They are not always invariant *by definition*, however.
Complexity Classes
The Complexity’s Complexity

- Checking the resource consumption of a program $P$ to be bounded by a function in a robust class is hard.
- The set of all polytime programs is invariably $\Sigma_2$-complete!
- This does not mean that there cannot be any decidable set of programs $S$ such that $[S]$ coincides with the class of polytime functions.
The Complexity’s Complexity

- Checking the resource consumption of a program $P$ to be bounded by a function in a robust class is hard.
- The set of all polytime programs is invariably $\Sigma_2$-complete!
- This does not mean that there cannot be any **decidable** set of programs $S$ such that $\llbracket S \rrbracket$ coincides with the class of polytime functions.
Exercises

- Prove that the class of functional programs over $S_N$ is Turing powerful.
- Prove that the set of functions which can be represented in the $\lambda$-calculus is Turing powerful.
- Prove that the class of polytime Turing programs is $\Sigma_2$-complete.
- Is the unitary cost model invariant in functional programs?
  - Is it possible to define a functional program that produces an exponentially long output?
- Is the unitary cost model invariant in the $\lambda$-calculus?
  - Try to play the same game as before.
Exercises

▶ Prove that the class of functional programs over $S_N$ is Turing powerful.

▶ Prove that the set of functions which can be represented in the $\lambda$-calculus is Turing powerful.

▶ Prove that the class of polytime Turing programs is $\Sigma_2$-complete.

▶ Is the unitary cost model invariant in functional programs?
  ▶ Is it possible to define a functional program that produces an exponentially long output?

▶ Is the unitary cost model invariant in the $\lambda$-calculus?
  ▶ Try to play the same game as before.
Exercises

- Prove that the class of functional programs over $S_N$ is Turing powerful.
- Prove that the set of functions which can be represented in the $\lambda$-calculus is Turing powerful.
- Prove that the class of polytime Turing programs is $\Sigma_2$-complete.
- Is the unitary cost model invariant in functional programs?
  - Is it possible to define a functional program that produces an exponentially long output?
- Is the unitary cost model invariant in the $\lambda$-calculus?
  - Try to play the same game as before.
Exercises

▶ Prove that the class of functional programs over $S_N$ is Turing powerful.
▶ Prove that the set of functions which can be represented in the $\lambda$-calculus is Turing powerful.
▶ Prove that the class of polytime Turing programs is $\Sigma_2$-complete.
▶ Is the unitary cost model invariant in functional programs?
   ▶ Is it possible to define a functional program that produces an exponentially long output?
▶ Is the unitary cost model invariant in the $\lambda$-calculus?
   ▶ Try to play the same game as before.
Part III

Implicit Complexity in Function Algebras
Recursion on Notation: since computation in unary notation is inefficient, we need to switch to binary notation.

What Should we Keep in the Algebra?
- Basic functions are innocuous.
- Polytime functions are closed by composition.
- Minimization introduces partiality, and is not needed.
- Primitive Recursion?

Certain uses of primitive recursion are dangerous, but if we do not have any form of recursion, we capture much less than polytime functions.
Recursion on Notation: since computation in unary notation is inefficient, we need to switch to binary notation.

What Should we Keep in the Algebra?

- Basic functions are innoxious.
- Polytime functions are closed by composition.
- Minimization introduces partiality, and is not needed.
- Primitive Recursion?

Certain uses of primitive recursion are dangerous, but if we do not have any form of recursion, we capture much less than polytime functions.
Recursion on Notation: since computation in unary notation is inefficient, we need to switch to binary notation.

What Should we Keep in the Algebra?
- Basic functions are innoquoous.
- Polytime functions are closed by composition.
- Minimization introduces partiality, and is not needed.
- Primitive Recursion?

Certain uses of primitive recursion are dangerous, but if we do not have any form of recursion, we capture much less than polytime functions.
Safe Functions

- **Safe Functions**: pairs in the form \((f, n)\), where \(f : \mathcal{B}^m \rightarrow \mathcal{B}\) and \(0 \leq n \leq m\).

- The number \(n\) identifies the number of *normal arguments* between those of \(f\): they are the first \(n\), while the other \(m - n\) are the *safe arguments*.

- Following [BellantoniCook93], we use semicolons to separate normal and safe arguments: if \((f, n)\) is a safe function, we write \(f(\vec{W}; \vec{V})\) to emphasize that the \(n\) words in \(\vec{W}\) are the normal arguments, while the ones in \(\vec{V}\) are the safe arguments.
Basic Safe Functions

- The safe function \((e, 0)\) where \(e : \mathcal{B} \rightarrow \mathcal{B}\) always returns the empty string \(\varepsilon\).

- The safe function \((a_0, 0)\) where \(a_0 : \mathcal{B} \rightarrow \mathcal{B}\) is defined as follows: \(a_0(W) = 0 \cdot W\).

- The safe function \((a_1, 0)\) where \(a_1 : \mathcal{B} \rightarrow \mathcal{B}\) is defined as follows: \(a_1(W) = 1 \cdot W\).

- The safe function \((t, 0)\) where \(t : \mathcal{B} \rightarrow \mathcal{B}\) is defined as follows: \(t(\varepsilon) = \varepsilon\), \(t(0W) = W\) and \(t(1W) = W\).

- The safe function \((c, 0)\) where \(c : \mathcal{B}^4 \rightarrow \mathcal{B}\) is defined as follows: \(c(\varepsilon, W, V, Y) = W\), \(c(0X, W, V, Y) = V\) and \(c(1X, W, V, Y) = Y\).

- For every positive \(n \in \mathbb{N}\) and for whenever \(1 \leq m, k \leq n\), the safe function \((\Pi^n_m, k)\), where \(\Pi^n_m\) is defined in a natural way.
Safe Composition

\[(f : \mathcal{B}^n \to \mathcal{B}, m)\]

\[(g_j : \mathcal{B}^k \to \mathcal{B}, k) \text{ for every } 1 \leq j \leq m\]

\[(h_j : \mathcal{B}^{k+i} \to \mathcal{B}, k) \text{ for every } m + 1 \leq j \leq n\]

\[\Downarrow\]

\[(p : \mathcal{B}^{k+i} \to \mathcal{B}, k) \text{ defined as follows:}\]

\[p(\vec{W}; \vec{V}) = f(g_1(\vec{W};), \ldots, g_m(\vec{W};); h_{m+1}(\vec{W}; \vec{V}), \ldots, h_n(\vec{W}; \vec{V})).\]
Safe Composition

\[(f : \mathcal{B}^n \rightarrow \mathcal{B}, m)\]

\[(g_j : \mathcal{B}^k \rightarrow \mathcal{B}, k)\text{ for every } 1 \leq j \leq m\]

\[(h_j : \mathcal{B}^{k+i} \rightarrow \mathcal{B}, k)\text{ for every } m + 1 \leq j \leq n\]

\[\downarrow\]

\[(p : \mathcal{B}^{k+i} \rightarrow \mathcal{B}, k)\text{ defined as follows:}\]

\[p(\vec{W}; \vec{V}) = f(g_1(\vec{W});), \ldots, g_m(\vec{W});); h_{m+1}(\vec{W}; \vec{V}), \ldots, h_n(\vec{W}; \vec{V}))].\]
Safe Composition

\[(f : \mathcal{B}^n \to \mathcal{B}, m)\]

\[(g_j : \mathcal{B}^k \to \mathcal{B}, k) \text{ for every } 1 \leq j \leq m\]

\[(h_j : \mathcal{B}^{k+i} \to \mathcal{B}, k) \text{ for every } m + 1 \leq j \leq n\]

\[\downarrow\]

\[(p : \mathcal{B}^{k+i} \to \mathcal{B}, k) \text{ defined as follows:}\]

\[p(\tilde{W}; \tilde{V}) = f(g_1(\tilde{W};), \ldots, g_m(\tilde{W};); h_{m+1}(\tilde{W}; \tilde{V}), \ldots, h_n(\tilde{W}; \tilde{V})).\]
Safe Composition

\[(f : \mathcal{B}^n \to \mathcal{B}, m)\]

\[(g_j : \mathcal{B}^k \to \mathcal{B}, k) \text{ for every } 1 \leq j \leq m\]

\[(h_j : \mathcal{B}^{k+i} \to \mathcal{B}, k) \text{ for every } m + 1 \leq j \leq n\]

\[\Downarrow\]

\[(p : \mathcal{B}^{k+i} \to \mathcal{B}, k) \text{ defined as follows:}\]

\[p(\vec{W}; \vec{V}) = f(g_1(\vec{W});, \ldots, g_m(\vec{W});; h_{m+1}(\vec{W}; \vec{V}), \ldots, h_n(\vec{W}; \vec{V})).\]
Safe Composition

\[
\begin{align*}
(f : \mathcal{B}^n \rightarrow \mathcal{B}, m) \\
(g_j : \mathcal{B}^k \rightarrow \mathcal{B}, k) \text{ for every } 1 \leq j \leq m \\
(h_j : \mathcal{B}^{k+i} \rightarrow \mathcal{B}, k) \text{ for every } m + 1 \leq j \leq n \\
\downarrow \\
(p : \mathcal{B}^{k+i} \rightarrow \mathcal{B}, k) \text{ defined as follows:} \\
p(\vec{W}; \vec{V}) = f(g_1(\vec{W});, \ldots, g_m(\vec{W});; h_{m+1}(\vec{W}; \vec{V}), \ldots, h_n(\vec{W}; \vec{V})).
\end{align*}
\]
(Simultaneous) Safe Recursion

\[(f^i : \mathcal{B}^n \rightarrow \mathcal{B}, m) \text{ for every } 1 \leq i \leq j\]

\[(g^i_k : \mathcal{B}^{n+j+2} \rightarrow \mathcal{B}, m+1) \text{ for every } 1 \leq i \leq j, \quad k \in \{0, 1\}\]

\[
\downarrow
\]

\[(h^i : \mathcal{B}^{n+1} \rightarrow \mathcal{B}, m+1) \text{ defined as follows:}\]

\[h^i(0, \bar{W}; \bar{V}) = f(\bar{W}; \bar{V});\]

\[h^i(0X, \bar{W}; \bar{V}) = g^i_0(X, \bar{W}; \bar{V}, h^1(X, \bar{W}; \bar{V}), \ldots, h^j(X, \bar{W}; \bar{V}));\]

\[h^i(1X, \bar{W}; \bar{V}) = g^i_1(X, \bar{W}; \bar{V}, h^1(X, \bar{W}; \bar{V}), \ldots, h^j(X, \bar{W}; \bar{V}));\]
(Simultaneous) Safe Recursion

\((f^i : \mathcal{B}^n \rightarrow \mathcal{B}, m)\) for every \(1 \leq i \leq j\)

\((g^i_k : \mathcal{B}^{n+j+2} \rightarrow \mathcal{B}, m + 1)\) for every \(1 \leq i \leq j, k \in \{0, 1\}\)

\[\Downarrow\]

\((h^i : \mathcal{B}^{n+1} \rightarrow \mathcal{B}, m + 1)\) defined as follows:

\[h^i(0, \vec{W}; \vec{V}) = f(\vec{W}; \vec{V});\]
\[h^i(0X, \vec{W}; \vec{V}) = g^i_0(X, \vec{W}; \vec{V}, h^1(X, \vec{W}; \vec{V}), \ldots, h^j(X, \vec{W}; \vec{V}));\]
\[h^i(1X, \vec{W}; \vec{V}) = g^i_1(X, \vec{W}; \vec{V}, h^1(X, \vec{W}; \vec{V}), \ldots, h^j(X, \vec{W}; \vec{V}));\]
(Simultaneous) Safe Recursion

\[(f^i : \mathcal{B}^n \rightarrow \mathcal{B}, m) \text{ for every } 1 \leq i \leq j\]

\[(g_k^i : \mathcal{B}^{n+j+2} \rightarrow \mathcal{B}, m + 1) \text{ for every } 1 \leq i \leq j, \ k \in \{0, 1\}\]

\[\Downarrow\]

\[(h^i : \mathcal{B}^{n+1} \rightarrow \mathcal{B}, m + 1) \text{ defined as follows:}\]

\[h^i(0, \vec{W}; \vec{V}) = f(\vec{W}; \vec{V});\]

\[h^i(0X, \vec{W}; \vec{V}) = g_0^i(X, \vec{W}; \vec{V}, h^1(X, \vec{W}; \vec{V}), \ldots, h^j(X, \vec{W}; \vec{V}));\]

\[h^i(1X, \vec{W}; \vec{V}) = g_1^i(X, \vec{W}; \vec{V}, h^1(X, \vec{W}; \vec{V}), \ldots, h^j(X, \vec{W}; \vec{V}));\]
BCS is the smallest class of safe functions which includes the basic safe functions above and which is closed by safe composition and safe recursion.

BC is the set of those functions $f : \mathcal{B} \rightarrow \mathcal{B}$ such that $(f, n) \in \text{BCS}$ for some $n \in \{0, 1\}$. 
Lemma (Max-Poly Lemma)

For every \((f : B^n \rightarrow B, m)\) in BCS, there is a monotonically increasing polynomial \(p_f : \mathbb{N} \rightarrow \mathbb{N}\) such that:

\[
|f(V_1, \ldots, V_n)| \leq p_f \left( \sum_{1 \leq k \leq m} |V_k| \right) + \max_{m+1 \leq k \leq n} |V_k|.
\]

Proof.

- An induction on the structure of the proof a function being in BCS.
- The restrictions safe recursion must satisfy are essential.
Lemma (Max-Poly Lemma)

For every \((f : B^n \rightarrow B, m)\) in BCS, there is a monotonically increasing polynomial \(p_f : \mathbb{N} \rightarrow \mathbb{N}\) such that:

\[
|f(V_1, \ldots, V_n)| \leq p_f \left( \sum_{1 \leq k \leq m} |V_k| \right) + \max_{m+1 \leq k \leq n} |V_k|.
\]

Proof.

- An induction on the structure of the proof a function being in BCS.
- The restrictions safe recursion must satisfy are essential.
Theorem (Polytime Soundness)

\[ \text{BC} \subseteq \text{FP}\{0,1\}. \]

Proof.

- This is an induction on the structure of the proof of a function being in BCS.
- The proof becomes much easier if we first prove that simultaneous primitive recursion can be encoded into ordinary primitive recursion.
- We make essential use of the Max-Poly Lemma.
Theorem (Polytime Soundness)

\[ BC \subseteq FP\{0,1\} \cdot \]

Proof.

- This is an induction on the structure of the proof of a function being in BCS.
- The proof becomes much easier if we first prove that simultaneous primitive recursion can be encoded into ordinary primitive recursion.
- We make essential use of the Max-Poly Lemma.
Lemma (Polynomials)

For every polynomial $p : \mathbb{N} \rightarrow \mathbb{N}$ with natural coefficients there is a safe function $(f, 1)$ where $f : \mathcal{B} \rightarrow \mathcal{B}$ such that $|f(W)| = p(|W|)$ for every $W \in \mathcal{B}$.

Theorem (Polytime Completeness)

$\mathsf{FP}_{\{0,1\}} \subseteq \mathsf{BC}$.

Theorem (BellantoniCook1992)

$\mathsf{FP}_{\{0,1\}} = \mathsf{BC}$. 
Lemma (Polynomials)

For every polynomial \( p : \mathbb{N} \rightarrow \mathbb{N} \) with natural coefficients there is a safe function \((f, 1)\) where \( f : \mathcal{B} \rightarrow \mathcal{B} \) such that \( |f(W)| = p(|W|) \) for every \( W \in \mathcal{B} \).

Theorem (Polytime Completeness)

\[ \text{FP}_{\{0, 1\}} \subseteq \text{BC}. \]

Theorem (BellantoniCook1992)

\[ \text{FP}_{\{0, 1\}} = \text{BC}. \]
Lemma (Polynomials)

For every polynomial $p : \mathbb{N} \rightarrow \mathbb{N}$ with natural coefficients there is a safe function $(f, 1)$ where $f : \mathcal{B} \rightarrow \mathcal{B}$ such that $|f(W)| = p(|W|)$ for every $W \in \mathcal{B}$.

Theorem (Polytime Completeness)

$FP\{0,1\} \subseteq BC$.

Theorem (BellantoniCook1992)

$FP\{0,1\} = BC$. 
Part IV

Implicit Complexity in Functional Programs
Which Cost Model?

- Is it sensible to take the number of reduction steps as a measure of the execution time of a functional program $P$?
  - Apparently, the answer is negative.
    - Consider

\[
\begin{align*}
  f(0) & \rightarrow \text{nil} \\
  f(s(x)) & \rightarrow \text{bin}(f(x), f(x)).
\end{align*}
\]

- We need to exploit **sharing**!
  - Can we perform rewriting on shared representations of terms?
  - Does all this introduce an unacceptable overhead?

**Theorem (DLMartini2009)**

The unitary cost model is invariant.
Is it sensible to take the number of reduction steps as a measure of the execution time of a functional program \( P \)?

Apparently, the answer is negative.

Consider

\[
\begin{align*}
  f(0) & \rightarrow \text{nil} \\
  f(s(x)) & \rightarrow \text{bin}(f(x), f(x)).
\end{align*}
\]

We need to exploit sharing!

Can we perform rewriting on shared representations of terms?

Does all this introduce an unacceptable overhead?

**Theorem (DLMartini2009)**

The unitary cost model is invariant.
Which Cost Model?

- Is it sensible to take the number of reduction steps as a measure of the execution time of a functional program \( P \)?
- Apparently, the answer is negative.
  - Consider

  \[
  f(0) \rightarrow \text{nil} \\
  f(s(x)) \rightarrow \text{bin}(f(x), f(x)).
  \]

- We need to exploit **sharing**!
  - Can we perform rewriting on shared representations of terms?
  - Does all this introduce an unacceptable overhead?

**Theorem (DLMartini2009)**

The unitary cost model is invariant.
Which Cost Model?

- Is it sensible to take the number of reduction steps as a measure of the execution time of a functional program $P$?
- Apparently, the answer is negative.
  - Consider

\[
\begin{align*}
    f(0) & \rightarrow \text{nil} \\
    f(s(x)) & \rightarrow \text{bin}(f(x), f(x)).
\end{align*}
\]

- We need to exploit **sharing**!
  - Can we perform rewriting on shared representations of terms?
  - Does all this introduce an unacceptable overhead?

**Theorem (DLMartini2009)**

*The unitary cost model is invariant.*
The Interpretation Method

- **Domain**: a well-founded partial order \((\mathcal{D}, \leq)\).
- **Assignment** \(A\) for a signature \(S = (\Sigma, \alpha)\): to every symbol \(f\) in \(\Sigma\), one puts in correspondence a function

\[
[f]_A : \mathcal{D}^{\alpha(f)} \to \mathcal{D}
\]

which is strictly increasing in any of its argument w.r.t. \(\leq\).

- Given an assignment \(A\), one can generalize it to a map on closed and open terms.

- **Interpretation** for a functional program \(P\): an assignment such that for every rule \(l \rightarrow t\), it holds that

\[
[l]_A > [t]_A
\]

**Theorem (Lankford1979)**

A functional program \(P\) is terminating iff there is one interpretation for it.
Polynomial Interpretations

- What if we choose \( \mathbb{N} \) as the underlying domain, and polynomials on the natural numbers as the functions interpreting them?
- Do we get a characterization of polynomial time computable functions?
- Not really!
  - Suppose that \( f \) is a unary function symbol, and that \( t \) is a closed term in, say, \( C(S_B) \).
  - If \( f(t) \to^n s \), then \( n \leq [f(t)]_A = [f]_A([t]_A) \)
  - But \([t]_A\) can be much bigger than \(|t|\).
  - Everything depends on the way you interpret data!
  - You need to restrict to polynomial interpretations in which data are interpreted additively, e.g.

\[
[e] = 0; \quad [0](x) = x + 1; \quad [1](x) = x + 1.
\]

Theorem (BCMT2001)

Additive polynomial interpretations characterize polynomial time computable functions.
What if we choose \( \mathbb{N} \) as the underlying domain, and polynomials on the natural numbers as the functions interpreting them?

Do we get a characterization of polynomial time computable functions?

Not really!

Suppose that \( f \) is a unary function symbol, and that \( t \) is a closed term in, say, \( \mathcal{C}(\mathcal{S}_B) \).

If \( f(t) \rightarrow^n s \), then \( n \leq [f(t)]_A = [f]_A([t]_A) \)

But \([t]_A\) can be much bigger than \(|t|\).

Everything depends on the way you interpret data!

You need to restrict to polynomial interpretations in which data are interpreted additively, e.g.

\[
[e] = 0; \quad [0](x) = x + 1; \quad [1](x) = x + 1.
\]

**Theorem (BCMT2001)**

*Additive polynomial interpretations characterize polynomial time computable functions.*
Polynomial Interpretations

- What if we choose \( \mathbb{N} \) as the underlying domain, and polynomials on the natural numbers as the functions interpreting them?
- Do we get a characterization of polynomial time computable functions?
- Not really!
  - Suppose that \( f \) is a unary function symbol, and that \( t \) is a closed term in, say, \( C(S_B) \).
  - If \( f(t) \rightarrow^n s \), then \( n \leq [f(t)]_A = [f]_A([t]_A) \)
  - But \([t]_A\) can be much bigger than \(|t|\).
  - Everything depends on the way you interpret data!
  - You need to restrict to polynomial interpretations in which data are interpreted additively, e.g.

\[
[e] = 0; \quad [0](x) = x + 1; \quad [1](x) = x + 1.
\]

Theorem (BCMT2001)

Additive polynomial interpretations characterize polynomial time computable functions.
Polynomial Interpretations

What if we choose \(\mathbb{N}\) as the underlying domain, and polynomials on the natural numbers as the functions interpreting them?

Do we get a characterization of polynomial time computable functions?

Not really!

Suppose that \(f\) is a unary function symbol, and that \(t\) is a closed term in, say, \(C(S_B)\).

If \(f(t) \rightarrow^n s\), then \(n \leq [f(t)]_A = [f]_A([t]_A)\)

But \([t]_A\) can be much bigger than \(|t|\).

Everything depends on the way you interpret data!

You need to restrict to polynomial interpretations in which data are interpreted additively, e.g.

\[
[e] = 0; \quad [0](x) = x + 1; \quad [1](x) = x + 1.
\]

Theorem (BCMT2001)

Additive polynomial interpretations characterize polynomial time computable functions.
Polynomial Interpretations

What if we choose $\mathbb{N}$ as the underlying domain, and polynomials on the natural numbers as the functions interpreting them?

Do we get a characterization of polynomial time computable functions?

Not really!

Suppose that $f$ is a unary function symbol, and that $t$ is a closed term in, say, $C(S_B)$.

If $f(t) \rightarrow^n s$, then $n \leq [f(t)]_A = [f]_A([t]_A)$

But $[t]_A$ can be much bigger than $|t|$. Everything depends on the way you interpret data!

You need to restrict to polynomial interpretations in which data are interpreted additively, e.g.

$$[e] = 0; \quad [0](x) = x + 1; \quad [1](x) = x + 1.$$

Theorem (BCMT2001)

Additive polynomial interpretations characterize polynomial time computable functions.
What if we choose \( \mathbb{N} \) as the underlying domain, and polynomials on the natural numbers as the functions interpreting them?

Do we get a characterization of polynomial time computable functions?

Not really!

Suppose that \( f \) is a unary function symbol, and that \( t \) is a closed term in, say, \( C(S_B) \).

If \( f(t) \to^n s \), then \( n \leq [f(t)]_A = [f]_A([t]_A) \)

But \([t]_A\) can be much bigger than \(|t|\).

Everything depends on the way you interpret data!

You need to restrict to polynomial interpretations in which data are interpreted additively, e.g.

\[
[e] = 0; \quad [0](x) = x + 1; \quad [1](x) = x + 1.
\]

**Theorem (BCMT2001)**

Additive polynomial interpretations characterize polynomial time computable functions.
Polynomial Interpretations

What if we choose $\mathbb{N}$ as the underlying domain, and polynomials on the natural numbers as the functions interpreting them?

Do we get a characterization of polynomial time computable functions?

Not really!

Suppose that $f$ is a unary function symbol, and that $t$ is a closed term in, say, $C(S_B)$.

If $f(t) \rightarrow^n s$, then $n \leq [f(t)]_A = [f]_A([t]_A)$

But $[t]_A$ can be much bigger than $|t|$.

Everything depends on the way you interpret data!

You need to restrict to polynomial interpretations in which data are interpreted additively, e.g.

$$[e] = 0; \quad [0](x) = x + 1; \quad [1](x) = x + 1.$$ 

Theorem (BCMT2001)

Additive polynomial interpretations characterize polynomial time computable functions.
Multiset Path Orders — Preliminaries

- **Multiset** $M$ of a set $A$: a function $M : A \rightarrow \mathbb{N}$ which associates to any element $a \in A$ its *multiplicity* $M(a)$.

- Given a sequence $a_1, \ldots, a_n$ of (not necessarily distinct) elements of $A$, the associated multiset is written as $\{a_1, \ldots, a_n\}$.

- **Multiset Extension.** Given a strict order $\prec$ on $A$, its multiset extension $\prec^m$ is a relation between multisets of $A$ defined as follows: $M \prec^m N$ iff $M \neq N$ and for every $a \in A$ if $N(a) < M(a)$ then there is $b \in A$ with $a \prec b$ and $M(b) < N(b)$.

**Lemma**

*If $\prec$ is a strict order, then so is $\prec^m$.***
Multiset Path Orders — Preliminaries

▶ **Precedence**: A strict order \( \prec_S \) on \( \Sigma \) (where \( S = (\Sigma, \alpha) \) is a signature).

▶ Given a precedence \( \prec_S \), we can define a strict ordering on terms in \( C(S) \), called \( \prec_{\text{MPO},S} \) as the smallest binary relation on \( C(S) \) satisfying the following conditions:

- \( t \prec_{\text{MPO},S} f(s_1, \ldots, s_n) \) whenever \( t \prec_{\text{MPO},S} s_m \) for some \( 1 \leq m \leq n \);
- \( t \prec_{\text{MPO},S} f(s_1, \ldots, s_n) \) whenever \( t = s_m \) for some \( 1 \leq m \leq n \);
- \( f(t_1, \ldots, t_n) \prec_{\text{MPO},S} g(s_1, \ldots, s_m) \) if:
  - either \( f \prec_S g \) and \( t_k \prec_{\text{MPO},S} g(s_1, \ldots, s_m) \) for every \( 1 \leq k \leq n \).
  - or \( f = g \) and \( \{t_1, \ldots, t_n\} \prec_{\text{MPO},S} \{s_1, \ldots, s_m\} \).
A functional program $P$ on $S$ and $T$ is said to terminate by MPO if there is a precedence $\prec_{S+T}$ such that for every rule $l \rightarrow t$ in the program $P$, it holds that $t \prec_{MPO,S+T} l$.

**Theorem (Hofbauer1992)**

The class of functions computed by functional programs terminating by MPO coincides with the primitive recursive ones.
Path Orders and Computational Complexity

- **LMPO** [Marion2003].
  - The concept of a *valency* is used to mimick normal and safe arguments.
  - This information is exploited when extending precedences to terms.
  - The existence of an LMPO does not guarantee polynomial-time complexity in the sense of the unitary cost model: a form of memoization is needed.

- **POP** [AvanziniMoser2010].
  - Similar to LMPO, but more restrictive, this way ensuring that the complexity of captured programs is indeed bounded by a polynomial.
Definition 11. Let $\preceq_\mathcal{F}$ be a precedence on $\mathcal{F}$. The light multiset path ordering is a pair $(\prec_k)_{k=0,1}$ of orderings which is recursively defined on $\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \chi)$ as follows:
1. $s \prec_k c(\ldots, t_i, \ldots)$ if $s \preceq_k t_i$ and $c \in \mathcal{C}$.
2. $s \prec_k f(\ldots, t_i, \ldots)$ if $s \preceq_k t_i$, $f \in \mathcal{F}$, and $k \preceq v(f, i)$.
3. $c(s_1, \ldots, s_n) \prec_k f(t_1, \ldots, t_m)$ if $c \in \mathcal{C}, f \in \mathcal{F}$, and $s_i \prec_k f(t_1, \ldots, t_m)$, for each $i \leq n$. Note that $c$ can be a 0-ary.
4. $g(s_1, \ldots, s_n) \prec_k f(t_1, \ldots, t_m)$ if $(g \prec_\mathcal{F} f)$ and if $s_i \prec_{\max(k, v(g, j))} f(t_1, \ldots, t_m)$ for each $i \leq n$.
5. $g(s_1, \ldots, s_n) \prec_0 f(t_1, \ldots, t_n)$ if $g \preceq_\mathcal{F} f$ and $\{s_1, \ldots, s_n\} \prec_{g,f} \{t_1, \ldots, t_n\}$.
where $\preceq_k = \prec_k \cup \simeq$. 
Definition 3.5. Let $\succ$ denote a precedence. Consider terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ such that $s = f(s_1, \ldots, s_k; s_{k+1}, \ldots, s_{k+l})$. Then $s \succ_{\text{pop}^*} t$ if one of the following alternatives holds:

1. $s_i \succeq_{\text{pop}^*} t$ for some $i \in \{1, \ldots, k + l\}$, or
2. $f \in \mathcal{D}$, $t = g(t_1, \ldots, t_m; t_{m+1}, \ldots, t_{m+n})$ where $f \succ g$ and the following conditions hold:
   - $s \succeq_{\text{pop}} t_j$ for all normal argument positions $j = 1, \ldots, m$;
   - $s \succeq_{\text{pop}^*} t_j$ for all safe argument positions $j = m + 1, \ldots, m + n$;
   - $t_j \notin \mathcal{T}(\mathcal{F}^{<f}, \mathcal{V})$ for at most one safe argument position $j \in \{m + 1, \ldots, m + n\}$;
3. $f \in \mathcal{D}$, $t = g(t_1, \ldots, t_m; t_{m+1}, \ldots, t_{m+n})$ where $f \sim g$ and the following conditions hold:
   - $\{s_1, \ldots, s_k\} \succeq_{\text{pop}^*} \{t_1, \ldots, t_m\}$;
   - $\{s_{k+1}, \ldots, s_{k+l}\} \succeq_{\text{pop}^*} \{t_{m+1}, \ldots, t_{m+n}\}$. 
ICC Systems by themselves do not suffice.
  The class of programs they can capture is very small.
One can rather use them to analyse little portions of your programs, then trying to combine the obtained results.
How could one partition a program into smaller, independent portions?
  Dependency Graphs!
This is a strategy a concrete tool, called uses:
http://colo6-c703.uibk.ac.at/tct/
ICC Systems by themselves do not suffice.

- The class of programs they can capture is very small.

One can rather use them to analyse little portions of your programs, then trying to combine the obtained results.

How could one partition a program into smaller, independent portions?

- Dependency Graphs!

This is a strategy a concrete tool, called uses: http://colo6-c703.uibk.ac.at/tct/
Towards Complexity Analysis

- ICC Systems by themselves do not suffice.
  - The class of programs they can capture is very small.
- One can rather use them to analyse little \textit{portions} of your programs, then trying to combine the obtained results.
- How could one \textit{partition} a program into smaller, independent portions?
  - Dependency Graphs!
- This is a strategy a \textit{concrete tool}, called uses:
  \url{http://colo6-c703.uibk.ac.at/tct/}
Towards Complexity Analysis

- ICC Systems **by themselves** do not suffice.
  - The class of programs they can capture is very small.
- One can rather use them to analyse little *portions* of your programs, then trying to combine the obtained results.
- How could one *partition* a program into smaller, independent portions?
  - **Dependency Graphs!**
- This is a strategy a **concrete tool**, called uses: http://colo6-c703.uibk.ac.at/tct/
Towards Complexity Analysis

- ICC Systems by themselves do not suffice.
  - The class of programs they can capture is very small.
- One can rather use them to analyse little portions of your programs, then trying to combine the obtained results.
- How could one partition a program into smaller, independent portions?
  - Dependency Graphs!
- This is a strategy a concrete tool, called uses:
  http://colo6-c703.uibk.ac.at/tct/
Exercises

- Prove the Max-Poly Bound for safe recursion on notation.
- Prove the Polytime Completeness Theorem for safe recursion on notation.
- Find a polytime (in the unitary cost model) functional program $P$ which does not admit any additive polynomial interpretation.
- Play with $\text{TCT}$.
  - Write sorting programs (Insertion, Quick, Merge, etc.).
  - Analyse their complexity.
Part V

Implicit Complexity in the λ-Calculus
The λ-Calculus by itself is simply too powerful to form an ICC system.

How should we proceed if wanting to isolate, e.g., a class of λ-terms computing polytime functions?

**Type Systems** [Hofmann1997,BNS2000,Hofmann1999].
- You endow the λ-calculus with a type system *and* with some constants for data, recursion, etc.
- This way you get something similar to Gödel’s $\text{T}$.
- Then, you impose some constraints on recursion, akin to those from [BellantoniCook1992].

**Linearity Constraints** [Girard1997,Lafont2004].
- Key observation: copying is the operation making evaluation of λ-expressions problematic from a complexity point of view.
- Let us define some constraints on duplication, then!
The $\lambda$-Calculus by itself is simply too powerful to form an ICC system.

How should we proceed if wanting to isolate, e.g., a class of $\lambda$-terms computing polytime functions?

**Type Systems** [Hofmann1997,BNS2000,Hofmann1999].
- You endow the $\lambda$-calculus with a type system and with some constants for data, recursion, etc.
- This way you get something similar to Gödel’s $T$.
- Then, you impose some constraints on recursion, akin to those from [BellantoniCook1992].

**Linearity Constraints** [Girard1997,Lafont2004].
- Key observation: copying is the operation making evaluation of $\lambda$-expressions problematic from a complexity point of view.
- Let us define some constraints on duplication, then!
The λ-Calculus by itself is simply too powerful to form an ICC system.

How should we proceed if wanting to isolate, e.g., a class of λ-terms computing polytime functions?

**Type Systems** [Hofmann1997,BNS2000,Hofmann1999].

- You endow the λ-calculus with a type system and with some constants for data, recursion, etc.
- This way you get something similar to Gödel’s T.
- Then, you impose some constraints on recursion, akin to those from [BellantoniCook1992].

**Linearity Constraints** [Girard1997,Lafont2004].

- Key observation: copying is the operation making evaluation of λ-expressions problematic from a complexity point of view.
- Let us define some constraints on duplication, then!
The λ-Calculus by itself is simply too powerful to form an ICC system.

How should we proceed if wanting to isolate, e.g., a class of λ-terms computing polytime functions?

**Type Systems** [Hofmann1997, BNS2000, Hofmann1999].
- You endow the λ-calculus with a type system *and* with some constants for data, recursion, etc.
- This way you get something similar to Gödel’s T.
- Then, you impose some constraints on recursion, akin to those from [BellantoniCook1992].

**Linearity Constraints** [Girard1997, Lafont2004].
- Key observation: copying is the operation making evaluation of λ-expressions problematic from a complexity point of view.
- Let us define some constraints on duplication, then!
Lambda calculus $\Lambda$:

$$M ::= x \mid \lambda x. M \mid MM$$

with no structural constraints.

Linear Lambda Calculus $\Lambda!$

$$M ::= x \mid \lambda x. M \mid \lambda! x. M \mid MM \mid !M$$

where $x$ appears linearly in the body of $\lambda x. M$.

Soft Lambda Calculus $\Lambda_S$

$$M ::= x \mid \lambda x. M \mid \lambda! x. M \mid MM \mid !M$$

where additional constraints are needed for $\lambda! x. M$:

- $x$ appears once in $M$, inside a single occurrence of $!$...
- ... or $x$ appears more than once in $M$, outside $!$. 
From Lambda Calculus to Soft Lambda Calculus

- Lambda calculus \( \Lambda \):

\[
M ::= x \mid \lambda x.M \mid MM
\]

with no structural constraints.

- Linear Lambda Calculus \( \Lambda! \):

\[
M ::= x \mid \lambda x.M \mid \lambda!x.M \mid MM \mid !M
\]

where \( x \) appears linearly in the body of \( \lambda x.M \).

- Soft Lambda Calculus \( \Lambda_S \):

\[
M ::= x \mid \lambda x.M \mid \lambda!x.M \mid MM \mid !M
\]

where additional constraints are needed for \( \lambda!x.M \):
- \( x \) appears once in \( M \), inside a single occurrence of \( ! \).
- ... or \( x \) appears more than once in \( M \), outside \( ! \).
Lambda calculus $\Lambda$:

\[
M ::= x | \lambda x. M | MM
\]

with no structural constraints.

Linear Lambda Calculus $\Lambda!$

\[
M ::= x | \lambda x. M | \lambda! x. M | MM | !M
\]

where $x$ appears linearly in the body of $\lambda x. M$.

Soft Lambda Calculus $\Lambda_S$

\[
M ::= x | \lambda x. M | \lambda! x. M | MM | !M
\]

where additional constraints are needed for $\lambda! x. M$:

- $x$ appears once in $M$, inside a single occurrence of $!$.
- ... or $x$ appears more than once in $M$, outside $!$. 
From Lambda Calculus to Soft Lambda Calculus

- $\Lambda \Rightarrow \Lambda!$ is a **Refinement**.
  - Whenever a term can be copied, it must be marked as such, with $!$.
  - $\Lambda$ can be embedded into $\Lambda!$
    
    $\{x\} = x$
    
    $\{\lambda x.M\} = \lambda!x.\{M\}$
    
    $\{MN\} = \{M\}!\{N\}$

  - The embedding does not make use of $\lambda x.t$.

- $\Lambda! \Rightarrow \Lambda_S$ is a **Restriction**.
  - Whenever you copy, you lose the possibility of copying.
  - Examples:
    
    $\lambda!x.yxx \checkmark$
    
    $\lambda!x.y!x \checkmark$
    
    $\lambda!x.y(!x)x \n\n$  

  - Some results:
    - Polytime soundness;
    - Polytime completeness.
Linear Logic

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

\[
!A \iff !A \otimes !A
\]
\[
!A \otimes !B \iff !(A \otimes B)
\]
\[
!A \rightarrow !!A
\]
\[
!A \rightarrow A
\]
Linear Logic

Subsystems...

<table>
<thead>
<tr>
<th></th>
<th>!A⊗!B ≅!(A ⊗ B)</th>
<th>!A →!!A</th>
<th>!A → A</th>
<th>!A ≅!A⊗!A</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELL</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>LLL</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>SLL</td>
<td>YES</td>
<td>NO</td>
<td>!A → A ⊗...⊗ A</td>
<td></td>
</tr>
</tbody>
</table>

...and their expressive power

<table>
<thead>
<tr>
<th>ELL</th>
<th>Elementary Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLL</td>
<td>Polytime Functions</td>
</tr>
<tr>
<td>SLL</td>
<td>Polytime Functions</td>
</tr>
</tbody>
</table>
Linear Logic

Subsystems...

<table>
<thead>
<tr>
<th></th>
<th>$!A \otimes !B \cong !(A \otimes B)$</th>
<th>$!A \multimap !!A$</th>
<th>$!A \multimap A$</th>
<th>$!A \cong !A \otimes !A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELL</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>LLL</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>SLL</td>
<td>YES</td>
<td>NO</td>
<td>$!A \multimap A \otimes \ldots \otimes A$</td>
<td></td>
</tr>
</tbody>
</table>

...and their expressive power

<table>
<thead>
<tr>
<th></th>
<th>Elementary Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELL</td>
<td></td>
</tr>
<tr>
<td>LLL</td>
<td>Polytime Functions</td>
</tr>
<tr>
<td>SLL</td>
<td>Polytime Functions</td>
</tr>
</tbody>
</table>
Soft Linear Logic

- It is **polynomial time sound** [Lafont2002]:
  - $B_\pi$ is the box depth of any proof $\pi$;

**Theorem**

*There is a family of polynomials $\{p_n\}$ such that the normal form of any proof $\pi$ can be computed in time $p_{B_\pi}(|\pi|)$*

- This holds for many notions of proofs: proof-nets, sequent-calculus, lambda-terms, etc.

- It is also **polynomial time complete** [Lafont02, MairsonTerui03]:
  - A function $f : \mathbb{N} \to \mathbb{N}$ can be represented in soft linear logic if a proof $\pi_f$ rewrites to an encoding of $f(n)$ when cut against an encoding of $n$.

**Theorem**

*Every polynomial time function can be represented in soft linear logic.*
From Intuitionistic Logic to Soft Linear Logic

<table>
<thead>
<tr>
<th>Logic</th>
<th>Axioms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuitionistic Logic</td>
<td>$CCC$</td>
</tr>
<tr>
<td>(Intuitionistic) Multiplicative and Exponential Linear Logic</td>
<td>$SMCC$</td>
</tr>
</tbody>
</table>
|                                            | $!A \to !A \otimes !A$
|                                            | $!A \to 1$           |
|                                            | $!A \to !!A$         |
|                                            | $!A \to A$           |
| (Intuitionistic) Soft Linear Logic         | $SMCC$               |
|                                            | $!A \to A \otimes \ldots \otimes A$ |
|                                            | $!A \to 1$           |
Thank you!

Questions?