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

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 λ-calculi [Hofmann97], [Girard97], ...
  - ...

Implicit Computational Complexity

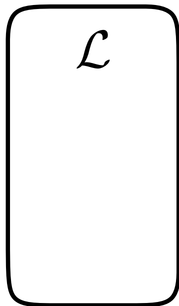
- **Goal**
  - Machine-*free* characterizations of complexity classes.
  - No *explicit* reference to resource bounds.
  - P, PSPACE, L, NC,...
- **Why?**
  - Simple and elegant *presentations* of complexity classes.
  - *Formal methods* for complexity analysis of programs.
- **How?**
  - First-order functional programs [BC92], [Leivant94], ...
  - Model theory [Fagin73], ...,
  - Type Systems and λ-calculi [Hofmann97], [Girard97], ...
  - ...

Part I

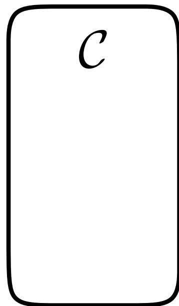
Implicit Complexity at a Glance

Characterizing Complexity Classes

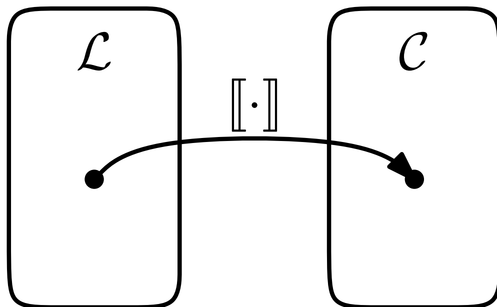
Programs



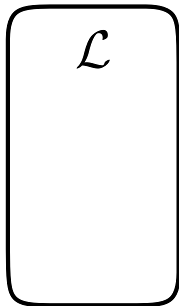
Functions

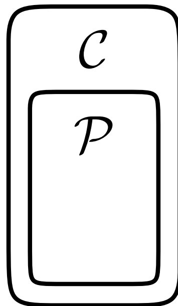


Characterizing Complexity Classes



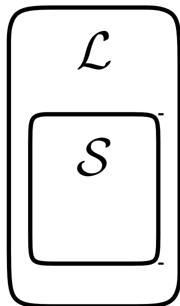
Characterizing Complexity Classes



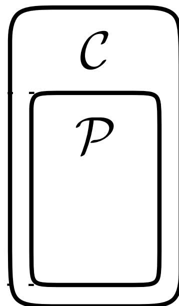


Characterizing Complexity Classes

Programs



Functions



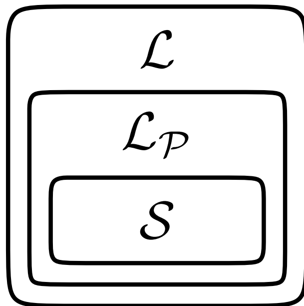
Proving $[\![\mathcal{S}]\!] = \mathcal{P}$

- $\mathcal{P} \subseteq [\![\mathcal{S}]\!]$
  - 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 $\mathcal{S} \subseteq \mathcal{L}_{\mathcal{P}}$.

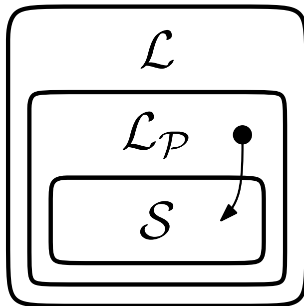
Proving $[\![\mathcal{S}]\!] = \mathcal{P}$

- $\mathcal{P} \subseteq [\![\mathcal{S}]\!]$
  - 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 $\mathcal{S} \subseteq \mathcal{L}_{\mathcal{P}}$.

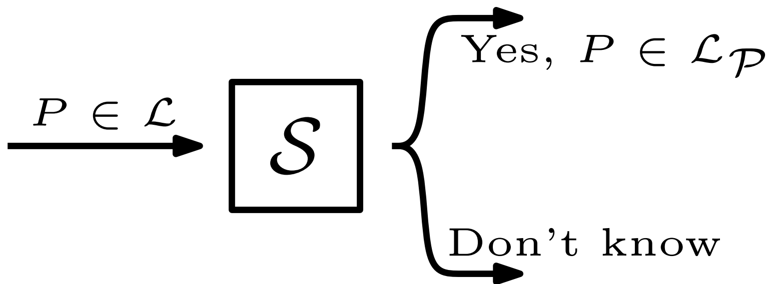
If Soundness is Proved Operationally...



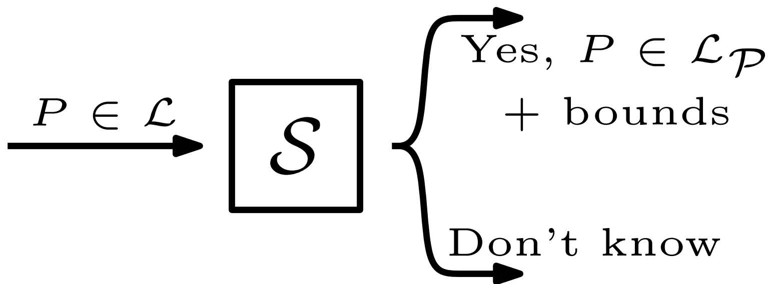
If Soundness is Proved Operationally...



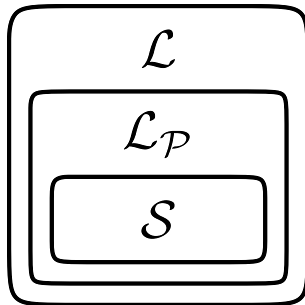
ICC Systems as Static Analyzers



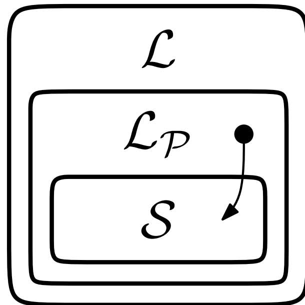
ICC Systems as Static Analyzers



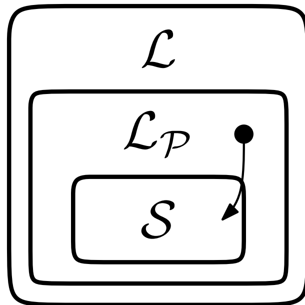
ICC: Intensional Expressive Power



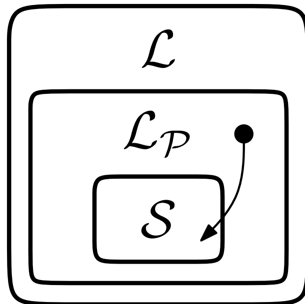
ICC: Intensional Expressive Power



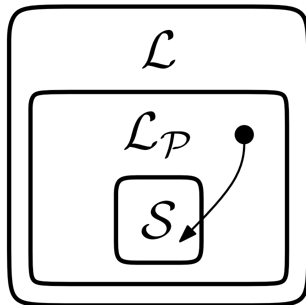
ICC: Intensional Expressive Power



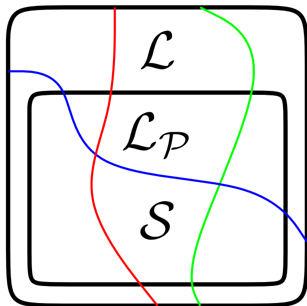
ICC: Intensional Expressive Power



ICC: Intensional Expressive Power



ICC: Intensional Expressive Power

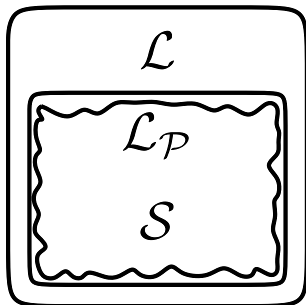


Safe Recursion [BC92]

Light Linear Logic [Girard97]

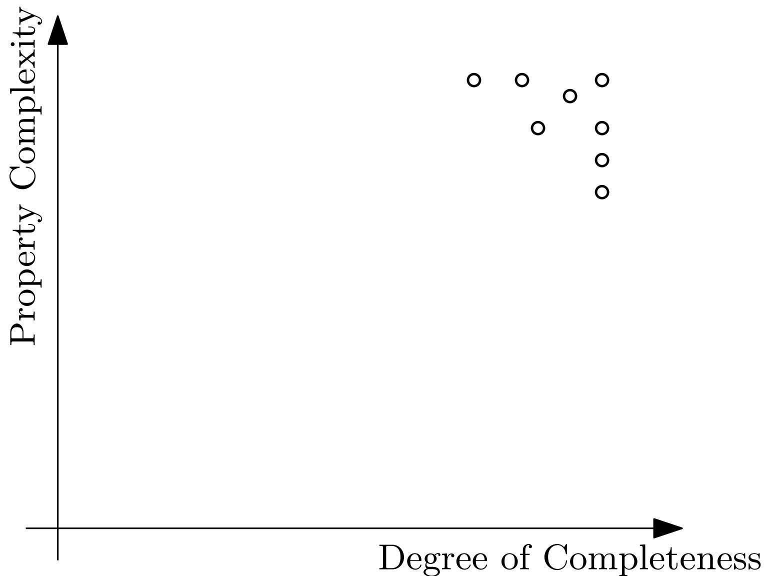
$\vdots$

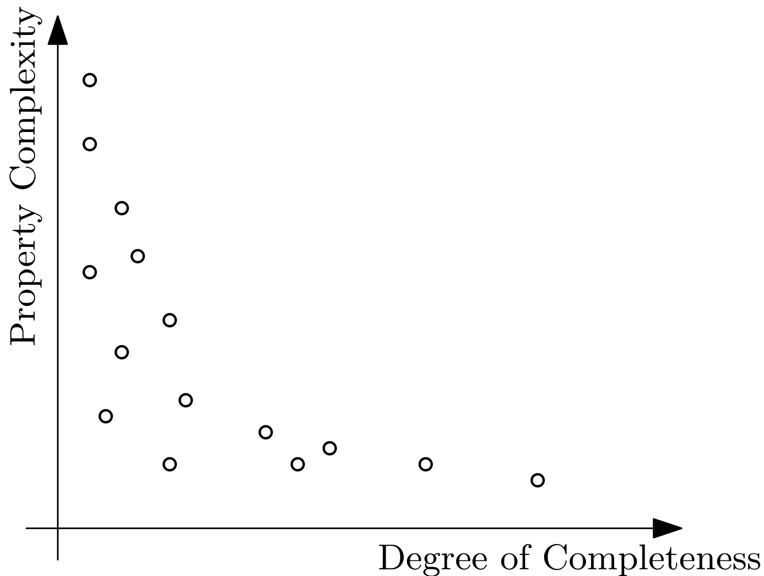
ICC: Intensional Expressive Power

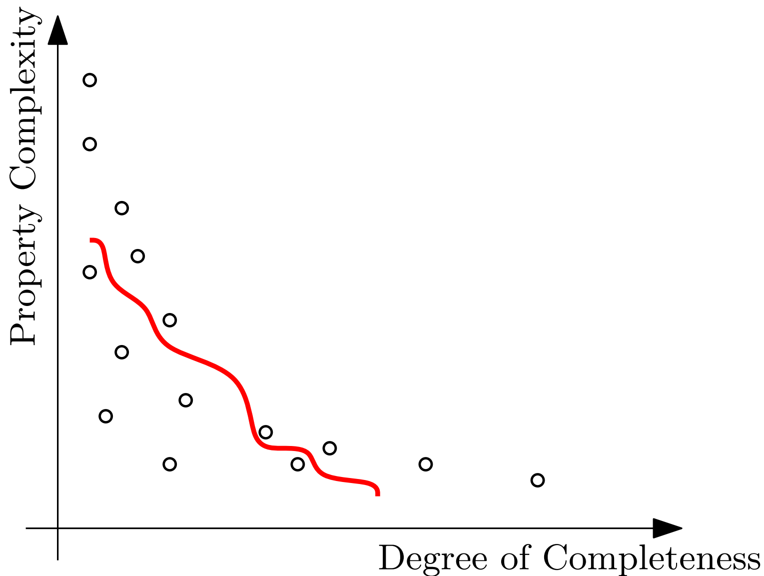


Bounded Recursion on Notation [Cobham63]
Bounded Arithmetic [Buss80]

Program Logics

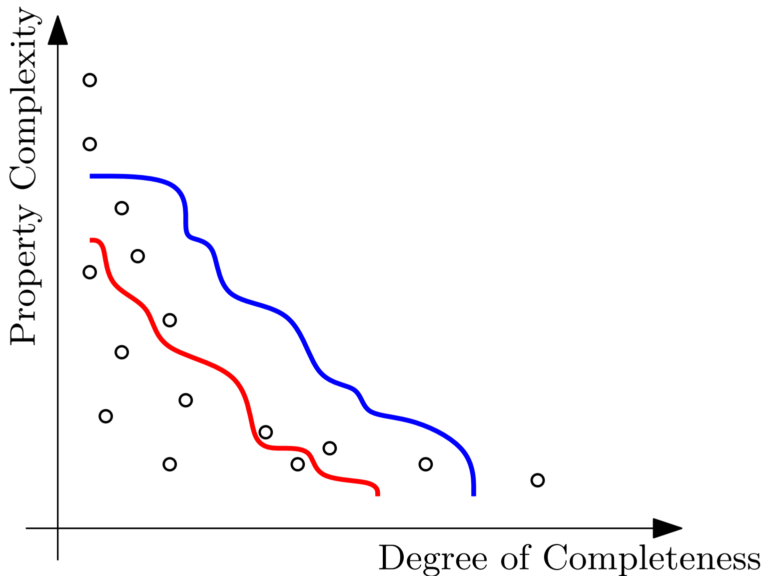



Property Complexity
Degree of Completeness



Property Complexity
Degree of Completeness

ICC Systems



Part II

Playing with Computational Models as Languages

Preliminaries

- **Alphabet**: a finite nonempty set, denoted with meta-variables like Σ, Υ, Φ.
- **String** over an alphabet Σ: a finite, ordered, possibly empty sequence of symbols from Σ. The *Kleene's closure* Σ^* of Σ is the set of all words over Σ, including the empty word ε.
- **Language** over the alphabet Σ: a subset of Σ^*.

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 $\{\mathtt{a}, \mathtt{b}\}$ can be defined as follows

$$W ::= \varepsilon \mid \mathtt{a} \mid \mathtt{b} \mid \mathtt{a}W\mathtt{a} \mid \mathtt{b}W\mathtt{b}.$$

- **Example**: the language $\mathcal{N}$ of all sequences in $\{\mathtt{0}, \ldots, \mathtt{9}\}^*$ 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 $\{\mathtt{0}, \mathtt{1}\}^*$.

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 $\{\mathtt{a}, \mathtt{b}\}$ can be defined as follows

$$W ::= \varepsilon \mid \mathtt{a} \mid \mathtt{b} \mid \mathtt{a}W\mathtt{a} \mid \mathtt{b}W\mathtt{b}.$$

- **Example**: the language $\mathcal{N}$ of all sequences in $\{\mathtt{0}, \ldots, \mathtt{9}\}^*$ 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 $\{\mathtt{0}, \mathtt{1}\}^*$.

Counter Machines

- **Counter Programs**:

$$P ::= I \mid I \, , P$$
$$I ::= \texttt{inc(}N\texttt{)} \mid \texttt{jmpz(}N\texttt{,}N\texttt{)}$$

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

- **Example**:

```
jmpz(0,4),inc(1),jmpz(2,1)
```

Counter Machines

- **Counter Programs**:

$$P ::= I \mid I\,\texttt{,}\,P$$
$$I ::= \texttt{inc(}N\texttt{)} \mid \texttt{jmpz(}N\texttt{,}N\texttt{)}$$

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

- **Example**:

```
jmpz(0,4),inc(1),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 `inc(`N`)` *increments* by one the content of R_n (where $n = [\![N]\!]$ is the natural number denoted by N). The next instruction to be executed is the following one in the program.
  - The instruction `jmpz(`N`,`M`)` *determines* the content n of $R_{[\![N]\!]}$ and
    - If n is positive, decrements $R_{[\![N]\!]}$ by one; the next instruction is the following one in the program.
    - If it is zero, the next instruction to be executed is the $[\![M]\!]$-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 `inc(`N`)` *increments* by one the content of R_n (where $n = [\![N]\!]$ is the natural number denoted by N). The next instruction to be executed is the following one in the program.
  - The instruction `jmpz(`N`,`M`)` *determines* the content n of $R_{[\![N]\!]}$ and
    - If n is positive, decrements $R_{[\![N]\!]}$ by one; the next instruction is the following one in the program.
    - If it is zero, the next instruction to be executed is the $[\![M]\!]$-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

```
jmpz(0,4),
inc(1),
jmpz(2,1)
```

Current Instruction	R_0	R_1	R_2
`jmpz(0,4)`	4	0	0
`inc(1)`	3	0	0
`jmpz(2,1)`	3	1	0
`jmpz(0,4)`	3	1	0
`inc(1)`	2	1	0
`jmpz(2,1)`	2	2	0
`jmpz(0,4)`	2	2	0
`inc(1)`	1	2	0
`jmpz(2,1)`	1	3	0
`jmpz(0,4)`	1	3	0
`inc(1)`	0	3	0
`jmpz(2,1)`	0	4	0
`jmpz(0,4)`	0	4	0

Turing Programs

- **Turing Programs**:

$$P ::= I \mid I \,\texttt{,}\, P$$

$$I ::= \texttt{(}N\texttt{,}c\texttt{)} \; \texttt{>} \; \texttt{(}N\texttt{,}a\texttt{)} \mid \texttt{(}N\texttt{,}c\texttt{)} \; \texttt{>} \; \texttt{stop}$$

where c ranges over the *input alphabet* Σ which includes a special symbol $\square$ (called the *blank symbol*), and a ranges over the alphabet alphabet $\Sigma \cup \{\texttt{<}, \texttt{>}\}$.

Turing Programs

- 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 Σ, 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) `> 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,□) > (1,>),` `(1,0) > (2,1),` `(1,1) > (2,0),`
`(2,0) > (1,>),` `(2,1) > (1,>),` `(1,□) > stop`

- Execution:

$$(\varepsilon, \square, 010, 0) \xrightarrow{R} (\square, 0, 10, 1) \xrightarrow{R} (\square, 1, 10, 2)$$
$$\xrightarrow{R} (\square 1, 1, 0, 1) \xrightarrow{R} (\square 1, 0, 0, 2)$$
$$\xrightarrow{R} (\square 10, 0, \varepsilon, 1) \xrightarrow{R} (\square 10, 1, \varepsilon, 2)$$
$$\xrightarrow{R} (\square 101, \square, \varepsilon, 1).$$

Turing Programs — Example

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

```
(0,□) > (1,>),        (1,0) > (2,1),      (1,1) > (2,0),
(2,0) > (1,>),        (2,1) > (1,>),      (1,□) > stop
```

- Execution:

$$
\begin{aligned}
(\varepsilon, \Box, 010, 0) &\xrightarrow{R} (\Box, 0, 10, 1) \xrightarrow{R} (\Box, 1, 10, 2) \\
&\xrightarrow{R} (\Box 1, 1, 0, 1) \xrightarrow{R} (\Box 1, 0, 0, 2) \\
&\xrightarrow{R} (\Box 10, 0, \varepsilon, 1) \xrightarrow{R} (\Box 10, 1, \varepsilon, 2) \\
&\xrightarrow{R} (\Box 101, \Box, \varepsilon, 1).
\end{aligned}
$$

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.

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.

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.

Function Algebras — Basic Functions

- **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: $\Pi^n_m(k_1, \ldots, k_n) = k_m$.

Function Algebras — Operations (I)

- **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.

Function Algebras — Operations (II)

- **Minimization**. Suppose that $n \in \mathbb{N}$ is positive and that $f : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$. Then the function $g : \mathbb{N}^n \rightharpoonup \mathbb{N}$ defined as follows:

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

is said to be defined by *minimization* from f.

Functional Programs — Preliminaries

- **Signature**. It is a pair $\mathcal{S} = (\Sigma, \alpha)$ where:
  - Σ is an alphabet;
  - $\alpha : \Sigma \to \mathbb{N}$ assigns to any symbol c in Σ a natural number $\alpha(c)$, called its *arity*.
- **Examples**.
  - The signature $\mathcal{S}_{\mathbb{N}}$ defined as $(\{\mathtt{0}, \mathtt{s}\}, \alpha_{\mathbb{N}})$ where $\alpha_{\mathbb{N}}(\mathtt{0}) = 0$ and $\alpha_{\mathbb{N}}(\mathtt{s}) = 1$.
  - The signature $\mathcal{S}_{\mathcal{B}}$ defined as $(\{\mathtt{0}, \mathtt{1}, \mathtt{e}\}, \alpha_{\mathcal{B}})$ where $\alpha_{\mathcal{B}}(\mathtt{0}) = \alpha_{\mathcal{B}}(\mathtt{1}) = 1$ and $\alpha_{\mathcal{B}}(\mathtt{e}) = 0$.
- Given two signatures $\mathcal{S}$ and $\mathcal{T}$ such that the underlying set of symbols are disjoint, one can naturally form the sum $\mathcal{S} + \mathcal{T}$.

Functional Programs

- **Closed Terms**. Given a signature $\mathcal{S} = (\Sigma, \alpha)$, they are the smallest set of expressions $\mathcal{C}(\mathcal{S})$ satisfying the following closure property: if $\mathbf{f} \in \Sigma$ and $t_1, \ldots, t_{\alpha(\mathbf{f})} \in \mathcal{C}(\mathcal{S})$, then $\mathbf{f}(t_1, \ldots, t_{\alpha(\mathbf{f})}) \in \mathcal{C}(\mathcal{S})$.
- **Examples**.

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

- **Open Terms**. Given a set of variables $\mathcal{L}$ distinct from Σ, the set of *open terms* $\mathcal{O}(\mathcal{S}, \mathcal{L})$ is defined as the smallest set of words including $\mathcal{L}$ and satisfying the closure condition above.

Functional Programs

- **Functional Programs** on $\mathcal{S} = (\Sigma, \alpha)$ and $\mathcal{T} = (\Upsilon, \beta)$:

$$P ::= R \mid R \texttt{,} P$$

$$R ::= l \quad \texttt{->} \quad t$$

$$l ::= \mathbf{f}_1 \texttt{(} p_1^1 \texttt{,} \ldots \texttt{,} p_1^{\alpha(\mathbf{f}_1)} \texttt{)} \mid \ldots \mid \mathbf{f}_n \texttt{(} p_n^1 \texttt{,} \ldots \texttt{,} p_n^{\alpha(\mathbf{f}_n)} \texttt{)}$$

  where:
  - $\Sigma = \{\mathbf{f}_1 \ldots, \mathbf{f}_n\}$ and that Υ is disjoint from Σ.
  - the metavariables p_m^k range over $\mathcal{O}(\mathcal{T}, \mathcal{L})$,
  - t ranges over $\mathcal{O}(\mathcal{S} + \mathcal{T}, \mathcal{L})$.

Functional Programs

- **Example**:

$$\begin{aligned}\texttt{add}(\texttt{0}, x) \quad &\texttt{->} \quad x \\ \texttt{add}(\texttt{s}(x), y) \quad &\texttt{->} \quad \texttt{s}(\texttt{add}(x, y)).\end{aligned}$$

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

$$\begin{aligned}\texttt{add}(\texttt{s}(\texttt{s}(\texttt{0})), \texttt{s}(\texttt{s}(\texttt{s}(\texttt{0})))) &\to \texttt{s}(\texttt{add}(\texttt{s}(\texttt{0}), \texttt{s}(\texttt{s}(\texttt{s}(\texttt{0}))))) \\ &\to \texttt{s}(\texttt{s}(\texttt{add}(\texttt{0}, \texttt{s}(\texttt{s}(\texttt{s}(\texttt{0})))))) \\ &\to \texttt{s}(\texttt{s}(\texttt{s}(\texttt{s}(\texttt{s}(\texttt{0}))))).\end{aligned}$$

λ-Calculus

- **Terms**:

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

- The term obtained by *substituting* a term M for a variable x into another term N is denoted as $N\{M/x\}$.
- **Values**:

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

- **Reduction**:

$$\frac{}{(\lambda x.M)V \to_v M\{V/x\}} \qquad \frac{M \to_v N}{ML \to_v NL} \qquad \frac{M \to_v N}{LM \to_v LN}$$

  Here M ranges over terms, while V ranges over values.

- **Normal Forms**. Any term M such that there is not any N with $M \to_v N$. A term M *has* a normal form iff $M \to_v^* N$. Otherwise, we write $M \uparrow$.

λ-Calculus — Computing on $\mathbb{N}$

- **Scott's Numerals**.

$$\ulcorner 0 \urcorner = \lambda x.\lambda y.x;$$
$$\ulcorner n+1 \urcorner = \lambda x.\lambda y.y\ulcorner n \urcorner.$$

- **Representing Functions**. A λ-term M represents a function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ iff for every n, if $f(n)$ is defined and equals m, then $M\ulcorner n \urcorner \rightarrow_v^* \ulcorner m \urcorner$ and otherwise $M\ulcorner n \urcorner\uparrow$

Computability

- 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 un 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?
  - **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?
  - **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?
  - **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?
  - **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.

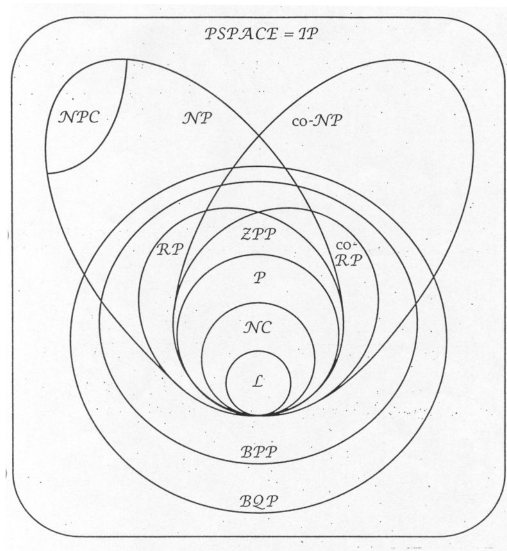
Cost Models

- 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?
  - **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.

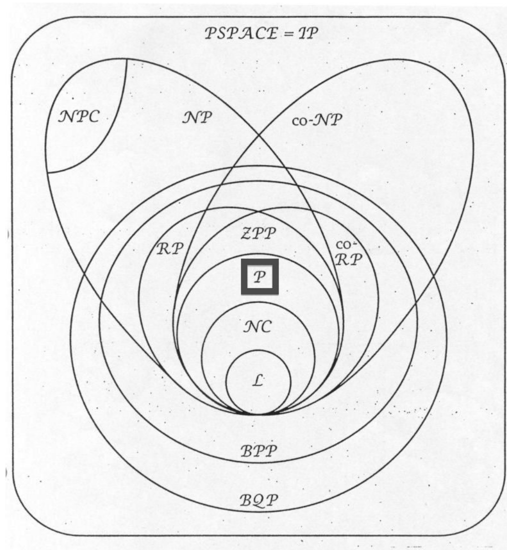
Cost Models

- 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?
  - **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



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 Σ_2-complete!
- This *does not* mean that there cannot be any **decidable** set of programs $\mathcal{S}$ such that $[\![\mathcal{S}]\!]$ 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 Σ_2-complete!
- This *does not* mean that there cannot be any **decidable** set of programs $\mathcal{S}$ such that $[\![\mathcal{S}]\!]$ coincides with the class of polytime functions.

Exercises

- Prove that the class of functional programs over $\mathcal{S}_{\mathbb{N}}$ is Turing powerful.
- Prove that the set of functions which can be respresented in the λ-calculus is Turing powerful.
- Prove that the class of polytime Turing programs is Σ_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 λ-calculus?
  - Try to play the same game as before.

Exercises

- Prove that the class of functional programs over $\mathcal{S}_{\mathbb{N}}$ is Turing powerful.
- Prove that the set of functions which can be respresented in the λ-calculus is Turing powerful.
- Prove that the class of polytime Turing programs is Σ_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 λ-calculus?
  - Try to play the same game as before.

Exercises

- Prove that the class of functional programs over $\mathcal{S}_{\mathbb{N}}$ is Turing powerful.
- Prove that the set of functions which can be respresented in the λ-calculus is Turing powerful.
- Prove that the class of polytime Turing programs is Σ_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 λ-calculus?
  - Try to play the same game as before.

Exercises

- Prove that the class of functional programs over $\mathcal{S}_{\mathbb{N}}$ is Turing powerful.
- Prove that the set of functions which can be respresented in the λ-calculus is Turing powerful.
- Prove that the class of polytime Turing programs is Σ_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 λ-calculus?
  - Try to play the same game as before.

Part III

Implicit Complexity in Function Algebras

Complexity Classes as 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 innoquous.
  - 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.

Complexity Classes as 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 innoquous.
  - 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.

Complexity Classes as 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 innoquous.
  - 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 $f : \mathcal{B}^m \to \mathcal{B}$ 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 $e : \mathcal{B} \to \mathcal{B}$ always returns the empty string ε.
- The safe function $(a_0, 0)$ where $a_0 : \mathcal{B} \to \mathcal{B}$ is defined as follows: $a_0(W) = \mathtt{0} \cdot W$.
- The safe function $(a_1, 0)$ where $a_1 : \mathcal{B} \to \mathcal{B}$ is defined as follows: $a_1(W) = \mathtt{1} \cdot W$.
- The safe function $(t, 0)$ where $t : \mathcal{B} \to \mathcal{B}$ is defined as follows: $t(\varepsilon) = \varepsilon$, $t(\mathtt{0}W) = W$ and $t(\mathtt{1}W) = W$.
- The safe function $(c, 0)$ where $c : \mathcal{B}^4 \to \mathcal{B}$ is defined as follows: $c(\varepsilon, W, V, Y) = W$, $c(\mathtt{0}X, W, V, Y) = V$ and $c(\mathtt{1}X, W, V, Y) = Y$.
- For every positive $n \in \mathbb{N}$ and for whenever $1 \leq m, k \leq n$, the safe function (Π^n_m, k), where Π^n_m is defined in a natural way.

Safe Composition

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

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

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

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

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

Safe Composition

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

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

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

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

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

Safe Composition

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

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

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

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

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

Safe Composition

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

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

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

$$\Downarrow$$

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

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

Safe Composition

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

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

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

$$\Downarrow$$

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

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

(Simultaneous) Safe Recursion

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

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

$$\Downarrow$$

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

$$\begin{aligned}
h^i(0, \vec{W}; \vec{V}) &= f(\vec{W}; \vec{V}); \\
h^i(\mathtt{0}X, \vec{W}; \vec{V}) &= g^i_0(X, \vec{W}; \vec{V}, h^1(X, \vec{W}; \vec{V}), \ldots, h^j(X, \vec{W}; \vec{V})); \\
h^i(\mathtt{1}X, \vec{W}; \vec{V}) &= g^i_1(X, \vec{W}; \vec{V}, h^1(X, \vec{W}; \vec{V}), \ldots, h^j(X, \vec{W}; \vec{V}));
\end{aligned}$$

(Simultaneous) Safe Recursion

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

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

$$\Downarrow$$

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

$$
\begin{aligned}
h^i(0, \vec{W}; \vec{V}) &= f(\vec{W}; \vec{V}); \\
h^i(\mathtt{0}X, \vec{W}; \vec{V}) &= g^i_0(X, \vec{W}; \vec{V}, h^1(X, \vec{W}; \vec{V}), \ldots, h^j(X, \vec{W}; \vec{V})); \\
h^i(\mathtt{1}X, \vec{W}; \vec{V}) &= g^i_1(X, \vec{W}; \vec{V}, h^1(X, \vec{W}; \vec{V}), \ldots, h^j(X, \vec{W}; \vec{V}));
\end{aligned}
$$

(Simultaneous) Safe Recursion

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

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

$$\Downarrow$$

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

$$\begin{aligned}
h^i(0, \vec{W}; \vec{V}) &= f(\vec{W}; \vec{V}); \\
h^i(\mathtt{0}X, \vec{W}; \vec{V}) &= g_0^i(X, \vec{W}; \vec{V}, h^1(X, \vec{W}; \vec{V}), \ldots, h^j(X, \vec{W}; \vec{V})); \\
h^i(\mathtt{1}X, \vec{W}; \vec{V}) &= g_1^i(X, \vec{W}; \vec{V}, h^1(X, \vec{W}; \vec{V}), \ldots, h^j(X, \vec{W}; \vec{V}));
\end{aligned}$$

Classes of Functions

- 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 $f : \mathcal{B} \to \mathcal{B}$ such that $(f, n) \in \mathsf{BCS}$ for some $n \in \{0, 1\}$.

Lemma (Max-Poly Lemma)

For every $(f : \mathcal{B}^n \to \mathcal{B}, m)$ *in* BCS*, there is a monotonically increasing polynomial* $p_f : \mathbb{N} \to \mathbb{N}$ *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 $(f : \mathcal{B}^n \to \mathcal{B}, m)$ *in* BCS, *there is a monotonically increasing polynomial* $p_f : \mathbb{N} \to \mathbb{N}$ *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)

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

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)

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

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.

Lemma (Polynomials)

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

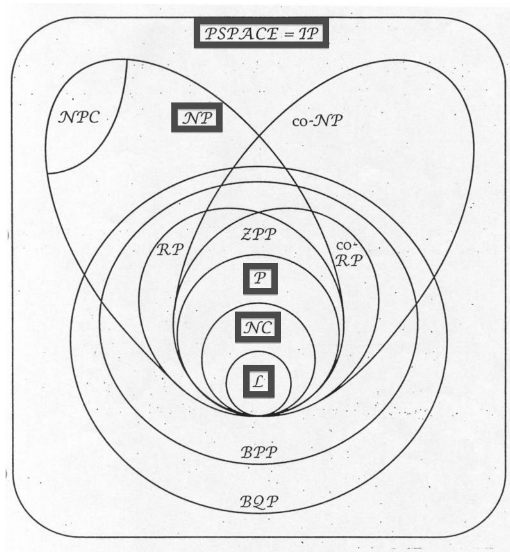
Theorem (Polytime Completeness)

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

Theorem (BellantoniCook1992)

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

Complexity Classes



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

$$\begin{aligned} \mathbf{f}(\mathtt{0}) &\ \texttt{->}\ \mathtt{nil} \\ \mathbf{f}(\mathtt{s}(x)) &\ \texttt{->}\ \mathtt{bin}(\mathbf{f}(x), \mathbf{f}(x)). \end{aligned}$$

- 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

$$\begin{aligned} \mathbf{f}(\mathtt{0}) \quad &\texttt{->} \quad \mathtt{nil} \\ \mathbf{f}(\mathtt{s}(x)) \quad &\texttt{->} \quad \mathtt{bin}(\mathbf{f}(x), \mathbf{f}(x)). \end{aligned}$$

- 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

$$
\begin{aligned}
\mathbf{f}(\mathtt{0}) &\;\;\texttt{->}\;\; \mathtt{nil} \\
\mathbf{f}(\mathtt{s}(x)) &\;\;\texttt{->}\;\; \mathtt{bin}(\mathbf{f}(x), \mathbf{f}(x)).
\end{aligned}
$$

- 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

$$\begin{aligned} \mathbf{f}(\mathtt{0}) &\quad \mathtt{->} \quad \mathtt{nil} \\ \mathbf{f}(\mathtt{s}(x)) &\quad \mathtt{->} \quad \mathtt{bin}(\mathbf{f}(x), \mathbf{f}(x)). \end{aligned}$$

- 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** $\mathcal{A}$ for a signature $\mathcal{S} = (\Sigma, \alpha)$: to every symbol $\mathbf{f}$ in Σ, one puts in correspondence a function

$$[\mathbf{f}]_{\mathcal{A}} : \mathcal{D}^{\alpha(\mathbf{f})} \to \mathcal{D}$$

  which is strictly increasing in any of its argument w.r.t. $\leq$.
- Given an assignment $\mathcal{A}$, 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 `->` t, it holds that

$$[l]_{\mathcal{A}} > [t]_{\mathcal{A}}$$

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 intepreting them?
- Do we get a characterization of polynomial time computable functions?
- Not really!
  - Suppose that $\mathbf{f}$ is a unary function symbol, and that t is a closed term in, say, $\mathcal{C}(\mathcal{S}_{\mathcal{B}})$.
  - If $\mathbf{f}(t) \rightarrow^n s$, then $n \leq [\mathbf{f}(t)]_{\mathcal{A}} = [\mathbf{f}]_{\mathcal{A}}([t]_{\mathcal{A}})$
  - But $[t]_{\mathcal{A}}$ 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.

$$[\mathsf{e}] = 0; \qquad [\mathsf{0}](x) = x + 1; \qquad [\mathsf{1}](x) = x + 1.$$

Theorem (BCMT2001)

Additive polynomial intepretations 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 intepreting them?
- Do we get a characterization of polynomial time computable functions?
- Not really!
  - Suppose that $\mathbf{f}$ is a unary function symbol, and that t is a closed term in, say, $\mathcal{C}(\mathcal{S}_\mathcal{B})$.
  - If $\mathbf{f}(t) \rightarrow^n s$, then $n \leq [\mathbf{f}(t)]_\mathcal{A} = [\mathbf{f}]_\mathcal{A}([t]_\mathcal{A})$
  - But $[t]_\mathcal{A}$ 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.

$$[\mathsf{e}] = 0; \qquad [\mathsf{0}](x) = x + 1; \qquad [\mathsf{1}](x) = x + 1.$$

Theorem (BCMT2001)

Additive polynomial intepretations 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 intepreting them?
- Do we get a characterization of polynomial time computable functions?
- Not really!
  - Suppose that $\mathbf{f}$ is a unary function symbol, and that t is a closed term in, say, $\mathcal{C}(\mathcal{S}_{\mathcal{B}})$.
  - If $\mathbf{f}(t) \to^n s$, then $n \leq [\mathbf{f}(t)]_{\mathcal{A}} = [\mathbf{f}]_{\mathcal{A}}([t]_{\mathcal{A}})$
  - But $[t]_{\mathcal{A}}$ 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.

$$[\mathsf{e}] = 0; \qquad [\mathsf{0}](x) = x + 1; \qquad [\mathsf{1}](x) = x + 1.$$

Theorem (BCMT2001)

Additive polynomial intepretations 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 intepreting them?
- Do we get a characterization of polynomial time computable functions?
- Not really!
  - Suppose that $\mathbf{f}$ is a unary function symbol, and that t is a closed term in, say, $\mathcal{C}(\mathcal{S}_{\mathcal{B}})$.
  - If $\mathbf{f}(t) \rightarrow^n s$, then $n \leq [\mathbf{f}(t)]_{\mathcal{A}} = [\mathbf{f}]_{\mathcal{A}}([t]_{\mathcal{A}})$
  - But $[t]_{\mathcal{A}}$ 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.

$$[\mathsf{e}] = 0; \qquad [\mathsf{0}](x) = x + 1; \qquad [\mathsf{1}](x) = x + 1.$$

Theorem (BCMT2001)

Additive polynomial intepretations 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 intepreting them?
- Do we get a characterization of polynomial time computable functions?
- Not really!
  - Suppose that $\mathbf{f}$ is a unary function symbol, and that t is a closed term in, say, $\mathcal{C}(\mathcal{S}_{\mathcal{B}})$.
  - If $\mathbf{f}(t) \rightarrow^n s$, then $n \leq [\mathbf{f}(t)]_{\mathcal{A}} = [\mathbf{f}]_{\mathcal{A}}([t]_{\mathcal{A}})$
  - But $[t]_{\mathcal{A}}$ 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.

$$[\mathsf{e}] = 0; \qquad [\mathsf{0}](x) = x + 1; \qquad [\mathsf{1}](x) = x + 1.$$

Theorem (BCMT2001)

Additive polynomial intepretations 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 intepreting them?
- Do we get a characterization of polynomial time computable functions?
- Not really!
  - Suppose that $\mathbf{f}$ is a unary function symbol, and that t is a closed term in, say, $\mathcal{C}(\mathcal{S}_{\mathcal{B}})$.
  - If $\mathbf{f}(t) \rightarrow^{n} s$, then $n \leq [\mathbf{f}(t)]_{\mathcal{A}} = [\mathbf{f}]_{\mathcal{A}}([t]_{\mathcal{A}})$
  - But $[t]_{\mathcal{A}}$ 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.

$$[\mathsf{e}] = 0; \qquad [\mathsf{0}](x) = x + 1; \qquad [\mathsf{1}](x) = x + 1.$$

Theorem (BCMT2001)

Additive polynomial intepretations 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 intepreting them?
- Do we get a characterization of polynomial time computable functions?
- Not really!
  - Suppose that $\mathbf{f}$ is a unary function symbol, and that t is a closed term in, say, $\mathcal{C}(\mathcal{S}_{\mathcal{B}})$.
  - If $\mathbf{f}(t) \rightarrow^n s$, then $n \leq [\mathbf{f}(t)]_{\mathcal{A}} = [\mathbf{f}]_{\mathcal{A}}([t]_{\mathcal{A}})$
  - But $[t]_{\mathcal{A}}$ 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.

$$[\mathtt{e}] = 0; \qquad [\mathtt{0}](x) = x + 1; \qquad [\mathtt{1}](x) = x + 1.$$

Theorem (BCMT2001)

Additive polynomial intepretations characterize polynomial time computable functions.

Multiset Path Orders — Preliminaries

- **Multiset** M of a set A: a function $M : A \to \mathbb{N}$ 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 $\prec_{\mathcal{S}}$ on Σ (where $\mathcal{S} = (\Sigma, \alpha)$ is a signature).
- Given a precedence $\prec_{\mathcal{S}}$, we can define a strict ordering on terms in $\mathcal{C}(\mathcal{S})$, called $\prec_{\mathsf{MPO},\mathcal{S}}$ as the smallest binary relation on $\mathcal{C}(\mathcal{S})$ satisfying the following conditions:
  - $t \prec_{\mathsf{MPO},\mathcal{S}} \mathbf{f}(s_1, \ldots, s_n)$ whenever $t \prec_{\mathsf{MPO},\mathcal{S}} s_m$ for some $1 \leq m \leq n$;
  - $t \prec_{\mathsf{MPO},\mathcal{S}} \mathbf{f}(s_1, \ldots, s_n)$ whenever $t = s_m$ for some $1 \leq m \leq n$;
  - $\mathbf{f}(t_1, \ldots, t_n) \prec_{\mathsf{MPO},\mathcal{S}} \mathbf{g}(s_1, \ldots, s_m)$ if:
    - either $\mathbf{f} \prec_{\mathcal{S}} \mathbf{g}$ and $t_k \prec_{\mathsf{MPO},\mathcal{S}} \mathbf{g}(s_1, \ldots, s_m)$ for every $1 \leq k \leq n$.
    - or $\mathbf{f} = \mathbf{g}$ and $\{\{t_1, \ldots, t_n\}\} \prec^m_{\mathsf{MPO},\mathcal{S}} \{\{s_1, \ldots, s_m\}\}$.

Multiset Path Orders

- A functional program P on $\mathcal{S}$ and $\mathcal{T}$ is said to *terminate by MPO* if there is a precedence $\prec_{\mathcal{S}+\mathcal{T}}$ such that for every rule l `->` t in the program P, it holds that $t \prec_{\mathsf{MPO},\mathcal{S}+\mathcal{T}} l$.

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.

LMPO

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. $s \prec_k \mathbf{c}(\ldots, t_i, \ldots)$ if $s \preceq_k t_i$ and $\mathbf{c} \in \mathcal{C}$.
2. $s \prec_k f(\ldots, t_i, \ldots)$ if $s \preceq_k t_i, f \in \mathcal{F}$, and $k \leqslant v(f, i)$.
3. $\mathbf{c}(s_1, \ldots, s_n) \prec_k f(t_1, \ldots, t_m)$ if $\mathbf{c} \in \mathcal{C}, f \in \mathcal{F}$, and $s_i \prec_k f(t_1, \ldots, t_m)$, for each $i \leqslant n$. Note that $\mathbf{c}$ 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 $s_i \prec_{\max(k, v(g,i))} f(t_1, \ldots, t_m)$ for each $i \leqslant n$.
5. $g(s_1, \ldots, s_n) \prec_0 f(t_1, \ldots, t_n)$ if $g \approx_{\mathcal{F}} f$ and $\{s_1, \ldots, s_n\} \prec^{v}_{g,f} \{t_1, \ldots, t_n\}$,

where $\preceq_k \; = \; \prec_k \cup \approx$.

POP*

Definition 3.5. Let $\succcurlyeq$ denote a precedence. Consider terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ such that $s = f(s_1, \dots, s_k; s_{k+1}, \dots, s_{k+l})$. Then $s >_{\mathsf{pop*}} t$ if one of the following alternatives holds:

(1) $s_i \geqslant_{\mathsf{pop*}} t$ for some $i \in \{1, \dots, k+l\}$, or

(2) $f \in \mathcal{D}$, $t = g(t_1, \dots, t_m; t_{m+1}, \dots, t_{m+n})$ where $f \succ g$ and the following conditions hold:
- $s >_{\mathsf{pop}} t_j$ for all normal argument positions $j = 1, \dots, m$;
- $s >_{\mathsf{pop*}} t_j$ for all safe argument positions $j = m+1, \dots, m+n$;
- $t_j \notin \mathcal{T}(\mathcal{F}^{\prec f}, \mathcal{V})$ for at most one safe argument position $j \in \{m+1, \dots, m+n\}$;

(3) $f \in \mathcal{D}$, $t = g(t_1, \dots, t_m; t_{m+1}, \dots, t_{m+n})$ where $f \sim g$ and the following conditions hold:
- $\{\{s_1, \dots, s_k\}\} >^{\mathsf{mul}}_{\mathsf{pop*}} \{\{t_1, \dots, t_m\}\}$;
- $\{\{s_{k+1}, \dots, s_{k+l}\}\} \geqslant^{\mathsf{mul}}_{\mathsf{pop*}} \{\{t_{m+1}, \dots, t_{m+n}\}\}$.

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/`

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/`

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 TCT.
  - Write sorting programs (Insertion, Quick, Merge, etc.).
  - Analyse their complexity.

Part V

Implicit Complexity in the λ-Calculus

λ-Calculus and Complexity

- 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!

λ-Calculus and Complexity

- 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!

λ-Calculus and Complexity

- 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!

λ-Calculus and Complexity

- 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!

From Lambda Calculus to Soft Lambda Calculus

- Lambda calculus Λ:

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

  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 Λ_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 Λ:

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

  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 Λ_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 Λ:

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

  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 Λ_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

- $\Lambda \Longrightarrow \Lambda_!$ is a **Refinement**.
  - Whenever a term can be copied, it must be marked as such, with !.
  - Λ can be embedded into $\Lambda_!$

$$
\begin{aligned}
\{x\} &= x \\
\{\lambda x.M\} &= \lambda !x.\{M\} \\
\{MN\} &= \{M\}!\{N\}
\end{aligned}
$$

  - The embedding does not make use of $\lambda x.t$.
- $\Lambda_! \Longrightarrow \Lambda_S$ is a **Restriction**.
  - Whenever you copy, you lose the possibility of copying.
  - Examples:

$$
\begin{array}{ll}
\lambda !x.yxx & \checkmark \\
\lambda !x.y!x & \checkmark \\
\lambda !x.y(!x)x & \lightning
\end{array}
$$

  - 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 $A \to B$ 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:

$$\begin{array}{rcl} !A & \cong & !A \otimes !A \\ !A \otimes !B & \cong & !(A \otimes B) \\ !A & \multimap & !!A \\ !A & \multimap & A \end{array}$$

Linear Logic

Subsystems...

	$!A \otimes !B \cong !(A \otimes B)$	$!A \multimap !!A$	$!A \multimap A$	$!A \cong !A \otimes !A$
ELL	YES	NO	NO	YES
LLL	NO	NO	NO	YES
SLL	YES	NO	$!A \multimap A \otimes \ldots \otimes A$	

...and their expressive power

ELL	Elementary Functions
LLL	Polytime Functions
SLL	Polytime Functions

Linear Logic

Subsystems...

	$!A \otimes !B \cong !(A \otimes B)$	$!A \multimap !!A$	$!A \multimap A$	$!A \cong !A \otimes !A$
ELL	YES	NO	NO	YES
LLL	NO	NO	NO	YES
SLL	YES	NO	$!A \multimap A \otimes \ldots \otimes A$	

...and their expressive power

ELL	Elementary Functions
LLL	Polytime Functions
SLL	Polytime Functions

Soft Linear Logic

- It is **polynomial time sound** [Lafont2002]:
  - $\mathbb{B}\pi$ is the box depth of any proof π;

Theorem

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

  - 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 π_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

Logic	**Axioms**
Intuitionistic Logic	$\mathcal{CCC}$
(Intuitionistic) Multiplicative and Exponential Linear Logic	$\mathcal{SMCC}$ $!A \multimap !A \otimes !A$ $!A \multimap 1$ $!A \multimap !!A$ $!A \multimap A$
(Intuitionistic) Soft Linear Logic	$\mathcal{SMCC}$ $!A \multimap A \otimes \ldots \otimes A$ $!A \multimap 1$

Thank you!

Questions?