Lambda-Calculus and Type Theory

Part IV

Ugo Dal Lago

Scuola Estiva AILA, Gargnano, August 2018

Section 1

Implicit Complexity at a Glance

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], ...
  - ...

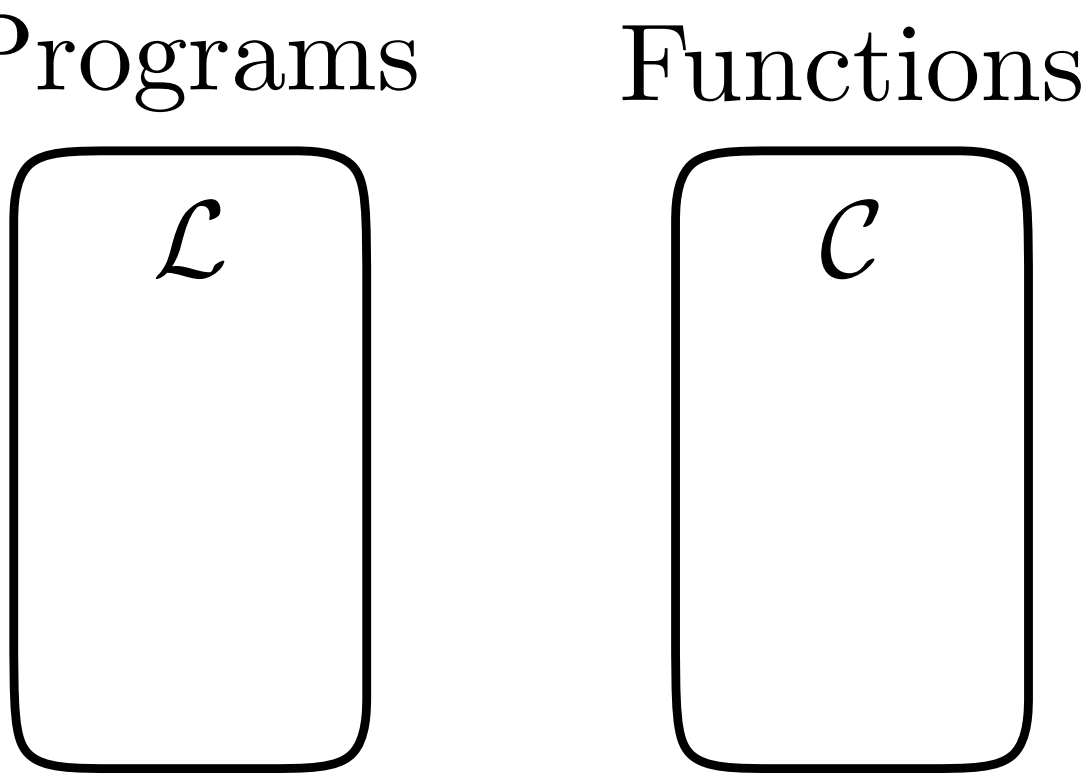
Implicit Computational Complexity

- **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], ...
  - ...

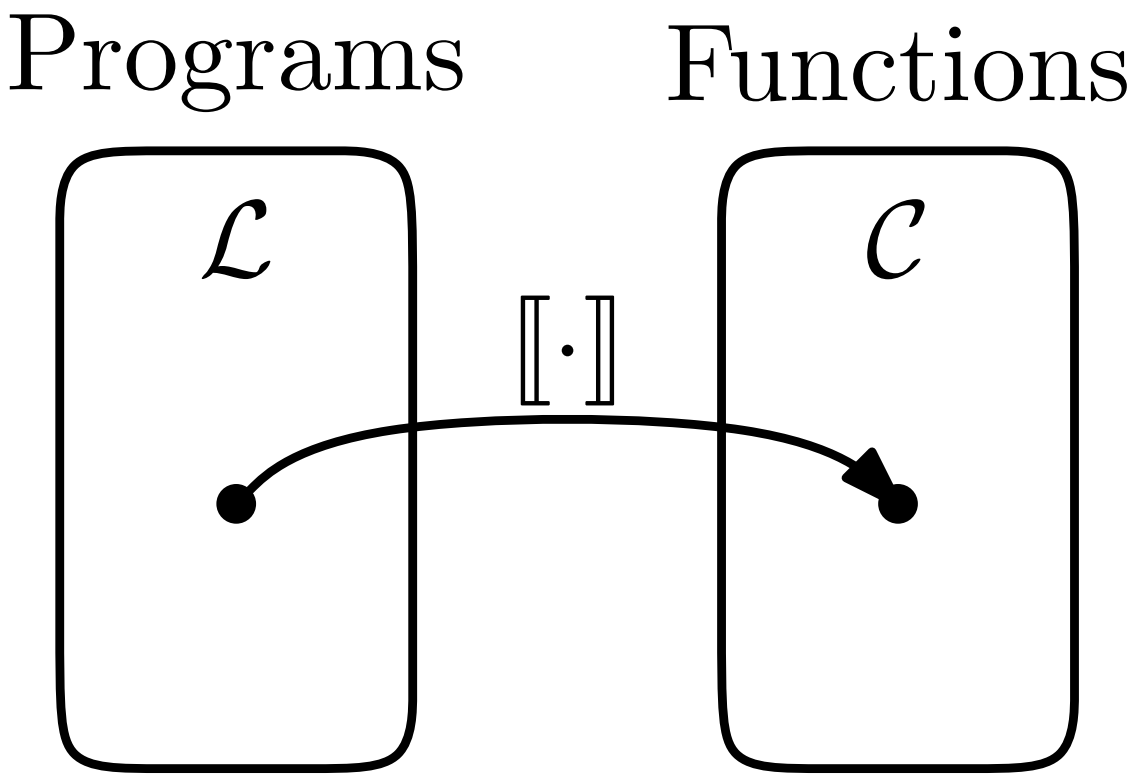
Implicit Computational Complexity

- **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], ...
  - ...

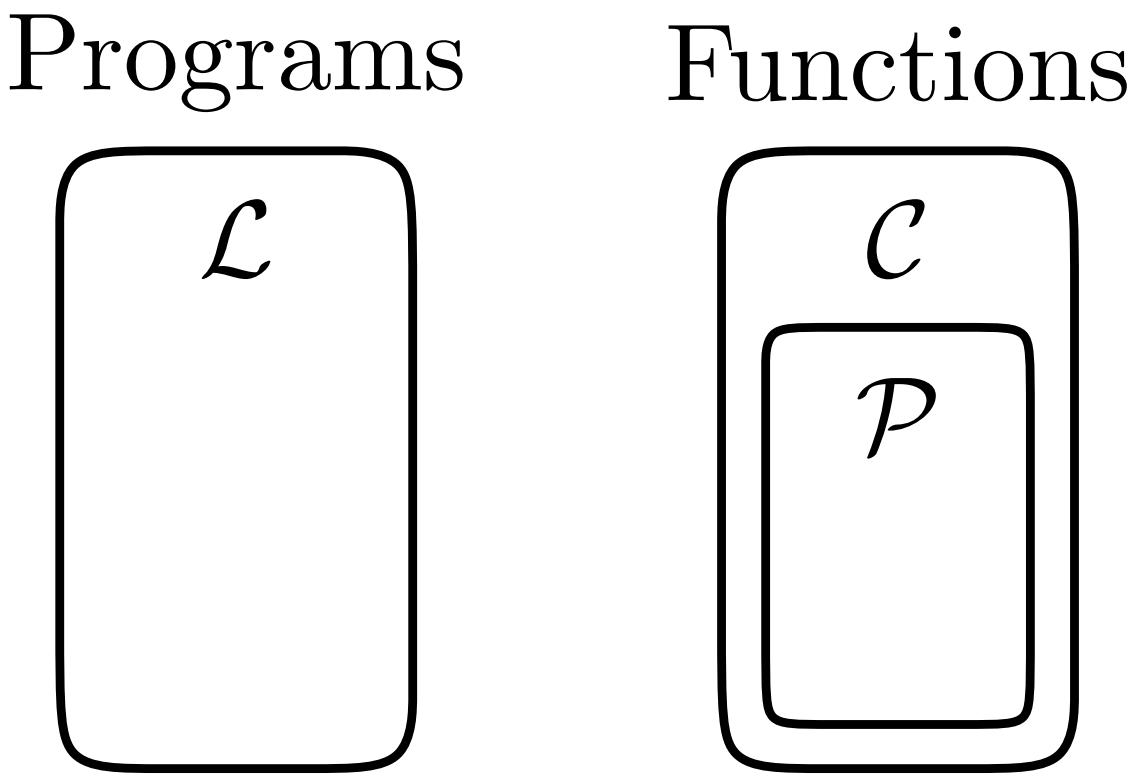
Characterizing Complexity Classes



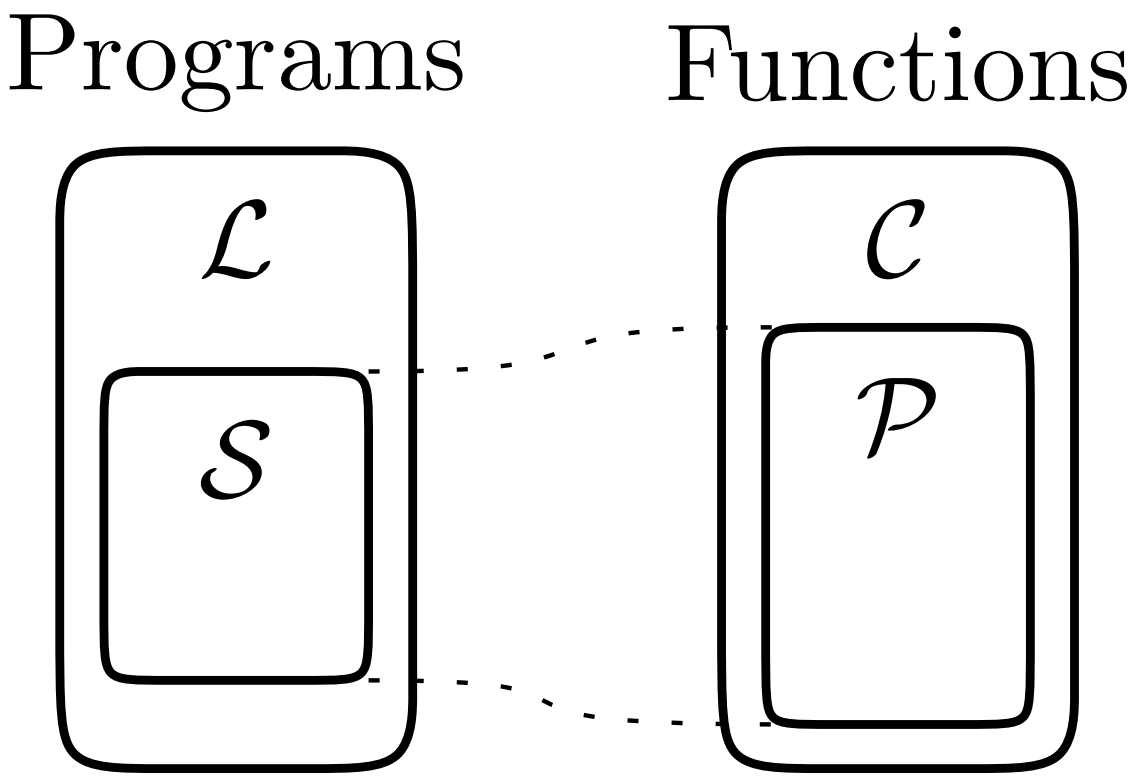
Characterizing Complexity Classes



Characterizing Complexity Classes



Characterizing Complexity Classes



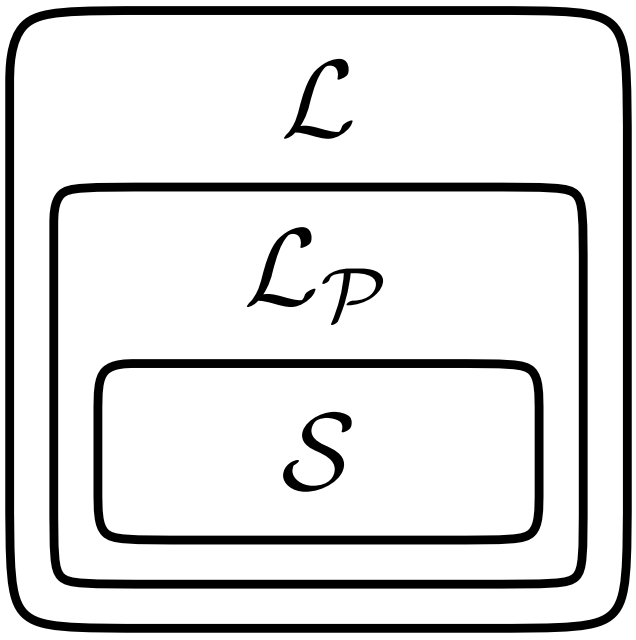
Proving $[\![\mathcal{S}]\!] = \mathcal{P}$

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

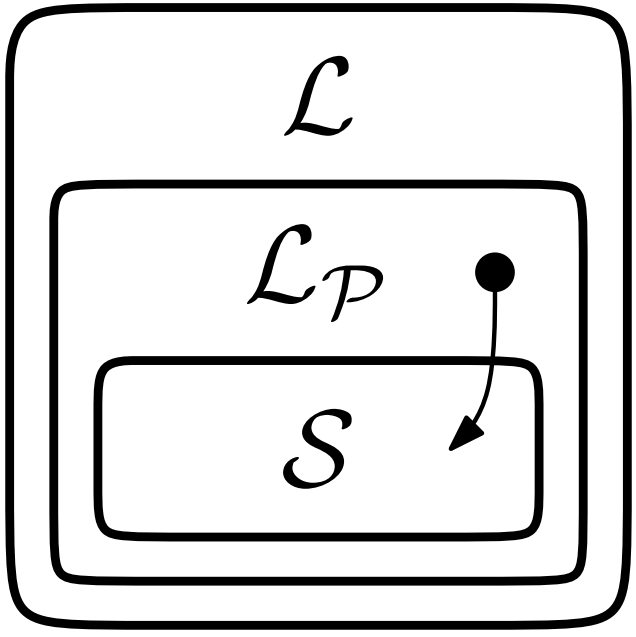
Proving $[\![\mathcal{S}]\!] = \mathcal{P}$

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

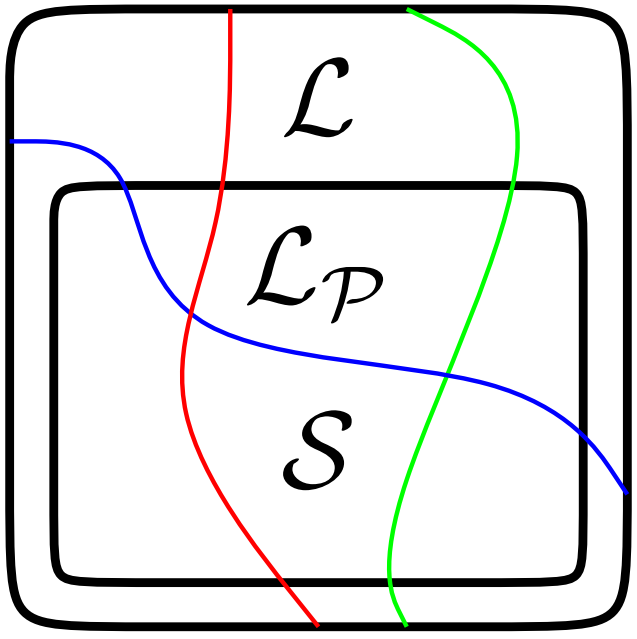
If Soundness is Proved Operationally...



If Soundness is Proved Operationally...

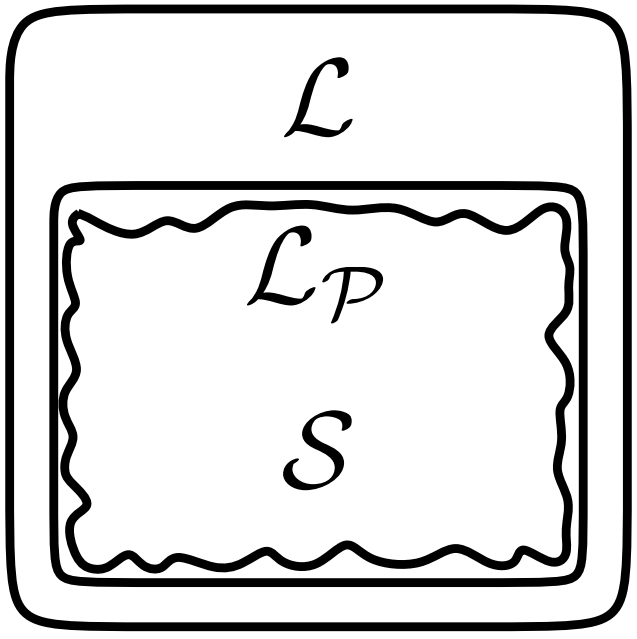


ICC: Two Styles



Safe Recursion [BC92]
Light Linear Logic [Girard97]
$\vdots$

ICC: Two Styles



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

Section 2

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

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

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

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

$$\Downarrow$$

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

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

Safe Composition

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

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

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

$$\Downarrow$$

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

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

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

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

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

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

$$\Downarrow$$

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

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

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 Recursion

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

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

$$\Downarrow$$

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

$$\begin{aligned} h(0, \vec{W}; \vec{V}) &= f(\vec{W}; \vec{V}); \\ h(0X, \vec{W}; \vec{V}) &= g_0(X, \vec{W}; \vec{V}, h(X, \vec{W}; \vec{V})); \\ h(1X, \vec{W}; \vec{V}) &= g_1(X, \vec{W}; \vec{V}, h(X, \vec{W}; \vec{V})). \end{aligned}$$

Safe Recursion

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

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

$$\Downarrow$$

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

$$h(0, \vec{W}; \vec{V}) = f(\vec{W}; \vec{V});$$

$$h(0X, \vec{W}; \vec{V}) = g_0(X, \vec{W}; \vec{V}, h(X, \vec{W}; \vec{V}));$$

$$h(1X, \vec{W}; \vec{V}) = g_1(X, \vec{W}; \vec{V}, h(X, \vec{W}; \vec{V})).$$

Safe Recursion

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

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

$$\Downarrow$$

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

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

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* $W \in \mathcal{B}$.

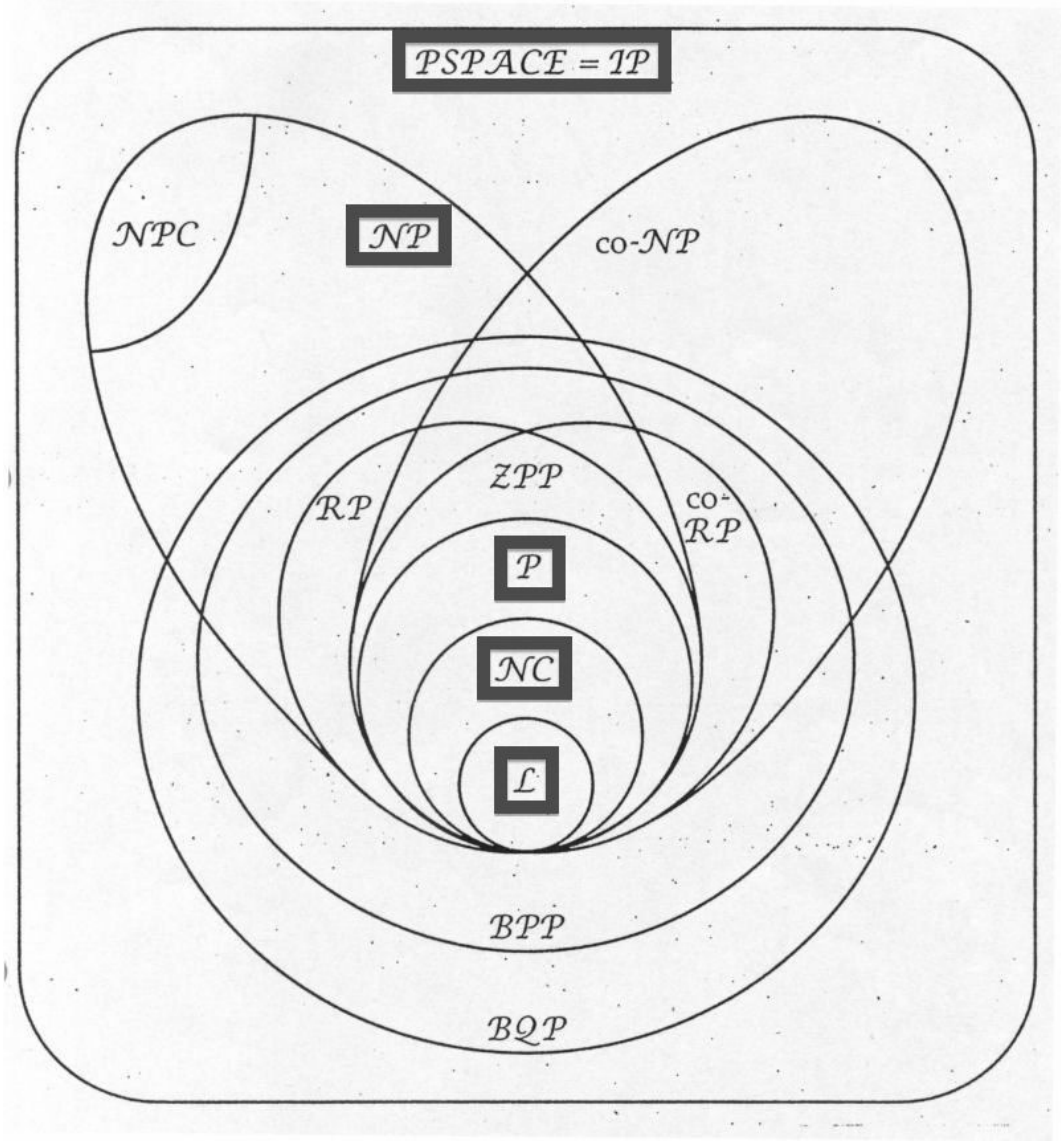
Theorem (Polytime Completeness)

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

Theorem (BellantoniCook1992)

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

Complexity Classes



Part I

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:

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

  - 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:

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

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.