Introduction
Implicit Computational Complexity (ICC) aims at providing formal methods for asserting computational properties of programs given in a natural programming language, e.g:

- That a given program is feasible (PTIME)
- That a given program runs with a bounded amount of memory usage (LOGSPACE, PSPACE)
- That a given program can be efficiently parallelized (NC)
- ...

Implicit Complexity: complexity bounds are derived from abstract properties of the formal methods at hand, not explicitly given.
Undecidability of the Problem addressed

ICC Problem:
Input: a given program $P$ written in any Turing-complete language (C, Caml, ...)
Question: does $P$ run in time polynomial in the size of its input?
Undecidability of the Problem addressed

ICC Problem:
Input: a given program $P$ written in any Turing-complete language (C, Caml, ...)
Question: does $P$ run in time polynomial in the size of its input?

ICC Problem is undecidable ($\Sigma_2$-complete - harder than the halting problem) (Marion/Moyen 2006)
Undecidability of the Problem addressed

ICC Problem:
Input: a given program $P$ written in any Turing-complete language (C, Caml, ...)
Question: does $P$ run in time polynomial in the size of its input?

ICC Problem is **undecidable** ($\Sigma_2$-complete - harder than the halting problem) (Marion/Moyen 2006)

Workaround: ICC systems recognize only some - not all - PTIME programs.

Expressivity vs Effectiveness:
The more PTIME programs an ICC system recognizes, the more complex it is.
Intensionality Vs Extensionality

Functions Vs Programs:
Function $f$ PTIME $\equiv$ at least one PTIME program that computes $f$.
ICC systems assert program properties, not function properties.
Intensionality Vs Extensionality

Functions Vs Programs:
Function $f$ PTIME $\equiv$ at least one PTIME program that computes $f$.

ICC systems assert program properties, not function properties.

Properties of ICC systems:
- PTIME soundness: Every program in the system is PTIME.
Intensionality Vs Extensionality

Functions Vs Programs:
Function $f$ PTIME $\equiv$ at least one PTIME program that computes $f$.

ICC systems assert program properties, not function properties.

Properties of ICC systems:

- PTIME soundness: Every program in the system is PTIME.
- Extensional PTIME completeness: Every PTIME function has at least one program in the system.
Intensionality Vs Extensionality

Functions Vs Programs:
Function \( f \) PTIME \( \equiv \) at least one PTIME program that computes \( f \).

ICC systems assert program properties, not function properties.

Properties of ICC systems:

- PTIME soundness: Every program in the system is PTIME.
- Extensional PTIME completeness: Every PTIME function has at least one program in the system.
- Intensional PTIME completeness: Every PTIME program is in the system.
Intensionality VS Extensionality (2)

PTIME soundness + PTIME Intensional completeness → undecidable system

Main goal of ICC approaches:

- PTIME Soundness
- PTIME Extensional Completeness
- As much Intensionality as possible
- Tractability of the system

+ Balance between Intensionality and Tractability
A Short History of ICC

- 1964: Cobham (Recursion theory, PTIME)
- 1992: Bellantoni/Cook (Recursion theory, PTIME)
- 1992: Girard/Scott/Scedrov (Bounded linear logic, PTIME)
- 1993: Leivant/Marion (Ramified typed $\lambda$-calculus, PTIME)
- 1994: Leivant (Ramified recursion, PTIME)
- 1994: Leivant/Marion (Recursion theory, PSPACE)
- 1995: Girard (Light linear logic, PTIME)
- 1999: Jones (Read-only languages, PTIME and LOGSPACE)
- 1999: Hofmann (NSI and linear types, PTIME)
- 2000: Bonfante, Marion, Moyen (Polynomial QI, PTIME)
- 2000: Lafont (Soft linear logic, PTIME)
Outline of the Course

- Preliminaries and Definitions: Models of computation, Complexity classes, Church/Cook thesis.
- Part I: Recursion. Cobham, Bellantoni/Cook, Leivant/Marion approaches
- Part II: Quasi Interpretations and 1st order functional languages
- Part III: Linear Logic and typed $\lambda$-calculus
Preliminaries
A machine $M$ is given by

- A finite alphabet $\Sigma$ (e.g. $\Sigma = \{0, 1\}$),
- a bi-infinite tape over $\Sigma \cup \#$, equipped with a scanning head,
- a finite set $Q$ of states, among which $\text{INIT, ACCEPT, REJECT}$,
- a finite set of transitions $Q \times \Sigma \rightarrow Q \times \Sigma \times \{\leftarrow, \downarrow, \rightarrow\}$

$M$ decides a language $L \subseteq \Sigma^*$, or computes a function $\Sigma^* \rightarrow \Sigma^*$. 
Complexity Classes

Complexity measure: amount of some resource used by $M$ for computing over inputs of a given size. ($\mathbb{N} \to \mathbb{N}$). Examples: time, space...

Complexity classes:

- **PTIME**: Set of languages $L \subseteq \Sigma^*$ decided in polynomial time by a machine $M$ over $\Sigma$.
- **LOGSPACE**: Set of languages $L \subseteq \Sigma^*$ decided in logarithmic space by a machine $M$ over $\Sigma$.
- **PSPACE**: Set of languages $L \subseteq \Sigma^*$ decided in polynomial space by a machine $M$ over $\Sigma$.

$\text{LOGSPACE} \subseteq \text{PTIME} \subseteq \text{PSPACE}$. 
Church-Cook Thesis

Other models of computation, with time complexity:

- boolean circuits – size of the circuits
- partial/primitive recursive functions – size of recursion tree
- rewriting systems – rewriting steps
- $\lambda$-calculus – $\beta$-reduction steps
- ...

Church-Cook Thesis: all "reasonable" definitions of PTIME are equivalent.
Different ICC Systems rely on different models of computation.
Part I: Recursion
Partial/Primitive Recursion (Kleene 1936)

Datatype = binary words \( w \in \{0, 1\}^* \).

Basic functions:
Datatype= binary words $\overline{w} \in \{0, 1\}^*$.
Basic functions:

\[
\begin{align*}
\text{hd}(a.\overline{w}) &= a & \text{tl}(a.\overline{w}) &= \overline{w} & \text{cons}(a.\overline{w_1}, \overline{w_2}) &= a.\overline{w_2} \\
\text{hd}(\varepsilon) &= \varepsilon & \text{tl}(\varepsilon) &= \varepsilon & \text{cons}(\varepsilon, \overline{w_2}) &= \overline{w_2}.
\end{align*}
\]
Datatype = binary words $\overline{w} \in \{0, 1\}^*$.  
Basic functions:

- $\text{hd}(a.\overline{w}) = a$  
  $\text{tl}(a.\overline{w}) = \overline{w}$  
  $\text{cons}(a.\overline{w_1}, \overline{w_2}) = a.\overline{w_2}$  

- $\text{hd}(\epsilon) = \epsilon$  
  $\text{tl}(\epsilon) = \epsilon$  
  $\text{cons}(\epsilon, \overline{w_2}) = \overline{w_2}$.  

- Projections: $Pr_i(\overline{w_1}, \ldots, \overline{w_k}) = \overline{w_i}, 1 \leq i \leq k$
Partial/Primitive Recursion (Kleene 1936)

Datatype= binary words $\overline{w} \in \{0, 1\}^*$. Basic functions:

- $\text{hd}(a.\overline{w}) = a \quad \text{tl}(a.\overline{w}) = \overline{w} \quad \text{cons}(a.\overline{w_1}, \overline{w_2}) = a.\overline{w_2}$
- $\text{hd}(\epsilon) = \epsilon \quad \text{tl}(\epsilon) = \epsilon \quad \text{cons}(\epsilon, \overline{w_2}) = \overline{w_2}$.

- Projections: $\text{Pr}_i(\overline{w_1}, \ldots, \overline{w_k}) = \overline{w_i}, 1 \leq i \leq k$

- Test: $\text{Select}(\overline{x}, \overline{y}, \overline{z}) = \begin{cases} \overline{y} & \text{if } \text{hd}(\overline{x}) = 1 \\ \overline{z} & \text{otherwise} \end{cases}$
Datatype = binary words $\overline{w} \in \{0, 1\}^*$. 

Basic functions:

- $\text{hd}(a.\overline{w}) = a$  
  $\text{tl}(a.\overline{w}) = \overline{w}$  
  $\text{cons}(a.\overline{w_1}, \overline{w_2}) = a.\overline{w_2}$
- $\text{hd}(\epsilon) = \epsilon$  
  $\text{tl}(\epsilon) = \epsilon$  
  $\text{cons}(\epsilon, \overline{w_2}) = \overline{w_2}$.

Projections: $P_{r_i}(\overline{w_1}, \ldots, \overline{w_k}) = \overline{w_i}, 1 \leq i \leq k$

Test: $\text{Select}(\overline{x}, \overline{y}, \overline{z}) = \begin{cases} 
\overline{y} & \text{if } \text{hd}(\overline{x}) = 1 \\
\overline{z} & \text{otherwise}
\end{cases}$

Composition:

$$f(\overline{x}) = g(h_1(\overline{x}), \ldots, h_k(\overline{x})).$$
Partial/Primitive Recursion

- **Primitive Recursion:**
  \[
  f(\epsilon, \overline{x}) = h(\overline{x})
  \]
  \[
  f(a\overline{y}, \overline{x}) = \begin{cases} 
    g(a, \overline{y}, f(\overline{y}, \overline{x}), \overline{x}) & \text{if } f(\overline{y}, \overline{x}) \neq \bot \\
    \bot & \text{otherwise.}
  \end{cases}
  \]
Partial/Primitive Recursion -2

- Primitive Recursion:
  \[ f(\epsilon, \overline{x}) = h(\overline{x}) \]
  \[ f(a.\overline{y}, \overline{x}) = \begin{cases} 
  g(a, \overline{y}, f(\overline{y}, \overline{x}), \overline{x}) & \text{if } f(\overline{y}, \overline{x}) \neq \bot \\
  \bot & \text{otherwise.} 
\end{cases} \]

- Minimization:
  \[ \mu \overline{x} g(\overline{x}, \overline{y}) = \begin{cases} 
  \bot & \text{if } \forall t \in \mathbb{N} \text{ } \text{hd}(g(0^t, \overline{y})) \neq 1 \\
  1^k : \quad k = \min \{ t \in \mathbb{N} : \text{hd}(g(0^t, \overline{y})) = 1 \} 
\end{cases} \]
Partial/Primitive Recursion -2

- **Primitive Recursion:**
  \[ f(\epsilon, x) = h(x) \]
  \[ f(a.y, x) = \begin{cases} 
    g(a, y, f(y, x), x) & \text{if } f(y, x) \neq \bot \\
    \bot & \text{otherwise.}
  \end{cases} \]

- **Minimization:**
  \[ \mu x g(x, y) = \begin{cases} 
    \bot & \text{if } \forall t \in \mathbb{N} \\text{hd}(g(0^t, y)) \neq 1 \\
    1^k : \ k = \min \{ t \in \mathbb{N} : \text{hd}(g(0^t, y)) = 1 \} 
  \end{cases} \]

Partial Recursive Functions: Closure of Basic functions under Composition, Primitive Recursion and minimization.
Partial/Primitive Recursion -2

- **Primitive Recursion:**
  \[ f(\epsilon, \overline{x}) = h(\overline{x}) \]
  \[ f(a.\overline{y}, \overline{x}) = \begin{cases} 
  g(a, \overline{y}, f(\overline{y}, \overline{x}), \overline{x}) & \text{if } f(\overline{y}, \overline{x}) \neq \bot \\
  \bot & \text{otherwise.} 
\end{cases} \]

- **Minimization:**
  \[ \mu \overline{x} g(\overline{x}, \overline{y}) = \begin{cases} 
  \bot & \text{if } \forall t \in \mathbb{N} \text{ hd}(g(0^t, \overline{y})) \neq 1 \\
  1^k : k = \min \{ t \in \mathbb{N} : \text{hd}(g(0^t, \overline{y})) = 1 \} 
\end{cases} \]

Partial Recursive Functions: Closure of Basic functions under Composition, Primitive Recursion and minimization.

**Partial/Primitive Recursion:**

Primitive Recursive Functions: Closure of Basic functions under Composition and Primitive Recursion.
Theorem 1  Partial Recursive Functions ≡ Partial Turing computable functions.
Theorem 1  \textit{Partial Recursive Functions} $\equiv$ \textit{Partial Turing computable functions}.  

Proof:  
- Soundness: any reasonable semantics  
- Completeness: simulation of the transitions of a Turing Machine by Primitive Recursive functions + minimization scheme for the time bound.
Primitive Recursion: Church/Cook Thesis

**Theorem 1**  \( \text{Partial Recursive Functions} \equiv \text{Partial Turing computable functions} \).

Proof:

- **Soundness**: any reasonable semantics
- **Completeness**: simulation of the transitions of a Turing Machine by Primitive Recursive functions + minimization scheme for the time bound.

Time complexity measure for Primitive Recursive functions: size of recursion tree.

**Theorem 2**  \( \text{PTIME Primitive Recursive Functions} \equiv \text{PTIME Turing functions} \).

Proof: as for Theorem 1.
An Example: Colson’s $C$ function

$$C(\overline{x}, \overline{y}) = 2^{\min\{|\overline{x}|, |\overline{y}|\}} - 1,$$
given by the following program:

\[
\begin{align*}
C(a.\overline{x}, \epsilon) &= 0 \\
C(\epsilon, b.\overline{y}) &= 0 \\
C(a.\overline{x}, b.\overline{y}) &= 1.C(\overline{x}, \overline{y})
\end{align*}
\]

The program above is \textit{not} Primitive Recursive, yet Colson’s $C$ function is Primitive Recursive.
An Example: Colson’s $C$ function

$$C(\overline{x}, \overline{y}) = 2^{\min\{|\overline{x}|, |\overline{y}|\}} - 1,$$
given by the following program:

\begin{align*}
C(a.\overline{x}, \epsilon) &= 0 \\
C(\epsilon, b.\overline{y}) &= 0 \\
C(a.\overline{x}, b.\overline{y}) &= 1.C(\overline{x}, \overline{y})
\end{align*}

The program above is \textit{not} Primitive Recursive, yet Colson’s $C$ function is Primitive Recursive.

**Theorem 3 (Colson, 1989)** Any Primitive Recursive program computing $C$ has time complexity strictly greater than $O(\min\{|\overline{x}|, |\overline{y}|\})$. 
Smash function: $\overline{x} \# \overline{y} = 2^{|x| \cdot |y|}$.

Limited Recursion on Notation (LRn):

$$f(\epsilon, \overline{x}) = h(\overline{x})$$
$$f(a \overline{y}, \overline{x}) = g(a, \overline{y}, f(\overline{y}, \overline{x}), \overline{x})$$
$$f(\overline{y}, \overline{x}) \leq k(\overline{y}, \overline{x}).$$

$L = \text{closure of basic functions and smash under composition and LRn.}$
Cobham (1964)

Smash function: \( \overline{x} \# \overline{y} = 2^{|x| \cdot |y|} \).

Limited Recursion on Notation (LRn):

\[
\begin{align*}
  f(\epsilon, \overline{x}) &= h(\overline{x}) \\
  f(a \cdot \overline{y}, \overline{x}) &= g(a, \overline{y}, f(\overline{y}, \overline{x}), \overline{x}) \\
  f(\overline{y}, \overline{x}) &\leq k(\overline{y}, \overline{x}).
\end{align*}
\]

\( \mathcal{L} = \) closure of basic functions and smash under composition and LRn.

**Theorem 4 (Cobham 1964)** \( \mathcal{L} \) is sound and extensionally complete for PTIME.
Cobham (1964)

Smash function: $\bar{x} \# \bar{y} = 2^{|x| \cdot |y|}$.

Limited Recursion on Notation (LRn):

$$f(\epsilon, \bar{x}) = h(\bar{x})$$
$$f(a.\bar{y}, \bar{x}) = g(a, \bar{y}, f(\bar{y}, \bar{x}), \bar{x})$$
$$f(\bar{y}, \bar{x}) \leq k(\bar{y}, \bar{x}).$$

$L = \text{closure of basic functions and smash under composition and LRn}.$

Theorem 4 (Cobham 1964) $L$ is sound and extensionally complete for $\text{PTIME}$.

Intensional Incompleteness: the (natural) program given above for Colson’s $C$ function is not in $L$. 
1. Soundness: Easy induction on the number of LRn schemes.
1. Soundness: Easy induction on the number of LRn schemes.

2. Completeness:
   - Smash + LRn allow to compute any polynomial P
Cobham - Proof of Theorem 4

1. Soundness: Easy induction on the number of LRn schemes.
2. Completeness:
   - Smash + LRn allow to compute any polynomial P
   - Encoding of any polynomial PR function in $L$ by explicit polynomial bounds.
1. Soundness: Easy induction on the number of LRn schemes.

2. Completeness:
   - Smash + LRn allow to compute any polynomial P
   - Encoding of any polynomial PR function in $\mathcal{L}$ by explicit polynomial bounds.

$\mathcal{L}$ is historically the first ICC system. Drawbacks:
   - Explicit bounds on the recursion schemes
   - Poor intensionality
Two types of arguments: "Normal; Safe". Safe arguments cannot appear in normal position. Basic safe functions:
Two types of arguments: "Normal;Safe". Safe arguments cannot appear in normal position.

Basic safe functions:

\[
\begin{align*}
\text{hd}(a.w) &= a \\
\text{tl}(a.w) &= w \\
\text{cons}(a.w_1, w_2) &= a.w_2 \\
\text{hd}(\epsilon) &= \epsilon \\
\text{tl}(\epsilon) &= \epsilon \\
\text{cons}(\epsilon, w_2) &= w_2.
\end{align*}
\]
Two types of arguments: "Normal; Safe". Safe arguments cannot appear in normal position.

Basic safe functions:

- \( \text{hd}(\; a.w) = a \)
- \( \text{tl}(\; a.w) = w \)
- \( \text{cons}(\; a.w_1, w_2) = a.w_2 \)

- \( \text{hd}(\; \epsilon) = \epsilon \)
- \( \text{tl}(\; \epsilon) = \epsilon \)
- \( \text{cons}(\; \epsilon, w_2) = w_2. \)

- **Projections:** \( Pr_i(\; w_1, \ldots, w_k) = w_i, \; 1 \leq i \leq k \)
Two types of arguments: "Normal; Safe". Safe arguments cannot appear in normal position. Basic safe functions:

- \[
    \text{hd}(a.w) = a \quad \text{tl}(a.w) = w \quad \text{cons}(a.w_1, w_2) = a.w_2
    \]
- \[
    \text{hd}(\epsilon) = \epsilon \quad \text{tl}(\epsilon) = \epsilon \quad \text{cons}(\epsilon, w_2) = w_2.
    \]
- **Projections:** \( Pr_i(w_1, \ldots, w_k) = w_i, 1 \leq i \leq k \)
- **Test:** \( Select(x, y, z) = \begin{cases} y & \text{if } \text{hd}(x) = 1 \\ z & \text{otherwise} \end{cases} \)
Two types of arguments: "Normal;Safe". Safe arguments cannot appear in normal position.

Basic safe functions:

\[
\begin{align*}
\text{hd}(; a.\overline{w}) &= a \\
\text{tl}(; a.\overline{w}) &= \overline{w} \\
\text{cons}(; a.\overline{w}_1, \overline{w}_2) &= a.\overline{w}_2 \\
\text{hd}(; \epsilon) &= \epsilon \\
\text{tl}(; \epsilon) &= \epsilon \\
\text{cons}(; \epsilon, \overline{w}_2) &= \overline{w}_2.
\end{align*}
\]

- Projections: \( Pr_i(; \overline{w}_1, \ldots, \overline{w}_k) = \overline{w}_i, 1 \leq i \leq k \)

- Test: \( \text{Select}(; x, y, z) = \begin{cases} 
\overline{y} & \text{if } \text{hd}(; \overline{x}) = 1 \\
\overline{z} & \text{otherwise}
\end{cases} \)

Safe Composition:

\[
f(\overline{x}; \overline{y}) = g(h_1(\overline{x}; ), \ldots, h_m(\overline{x}; ); h_{m+1}(\overline{x}; \overline{y}), \ldots, h_k(\overline{x}; \overline{y})).
\]
Safe Recursion on Notation:

\[ f(\epsilon, \overline{x}; \overline{y}) = h(\overline{x}; \overline{y}) \]
\[ f(a.\overline{z}, \overline{x}; \overline{y}) = g(a, \overline{z}, \overline{x}; f(\overline{z}, \overline{x}; \overline{y}), \overline{y}). \]

BC = closure of basic safe functions under safe composition and safe recursion.

**Theorem 5 (Bellantoni/Cook 1992)**  
BC is sound and extensionally complete for PTIME.
An Example: Unary Exponential

Consider the function $\exp(x) = 2^{2^{\lvert x \rvert}}$ given by the following program:

\[
\begin{align*}
\text{Append}(\epsilon, \overline{y}) &= \overline{y} \\
\text{Append}(a.\overline{x}, \overline{y}) &= a.\text{Append}(\overline{x}, \overline{y}) \\
\text{Double}(\overline{x}) &= \text{Append}(\overline{x}, \overline{x}) \\
\exp(\epsilon) &= 1 \\
\exp(a.\overline{x}) &= \text{Double}(\exp(\overline{x}))
\end{align*}
\]

This program is Primitive Recursive, has time complexity $\exp(\lvert x \rvert)$, and cannot be turned into a Safe Recursive one.
An Example: Unary Exponential

Consider the function $\exp(x) = 2^{2^{|x|}}$ given by the following program:

$$\begin{align*}
\text{Append}(\epsilon; y) &= y \\
\text{Append}(a.\overline{x}, y) &= a.\text{Append}(\overline{x}, y) \\
\text{Double}(\overline{x}) &= \text{Append}(\overline{x}, \overline{x}) \\
\exp(\epsilon) &= 1 \\
\exp(a.\overline{x}) &= \text{Double}(\exp(\overline{x}))
\end{align*}$$

This program is Primitive Recursive, has time complexity $\exp(|\overline{x}|)$, and cannot be turned into a Safe Recursive one.
An Example: Unary Exponential

Consider the function \( \exp(\overline{x}) = 2^{2^{\overline{x}}} \) given by the following program:

\[
\begin{align*}
\text{Append}(\epsilon; \overline{y}) &= \overline{y} \\
\text{Append}(a.\overline{x}; \overline{y}) &= a.\text{Append}(\overline{x}; \overline{y}) \\
\text{Double}(\overline{x}) &= \text{Append}(\overline{x}, \overline{x}) \\
\exp(\epsilon) &= 1 \\
\exp(a.\overline{x}) &= \text{Double}(\exp(\overline{x}))
\end{align*}
\]

This program is Primitive Recursive, has time complexity \( \exp(|\overline{x}|) \), and cannot be turned into a Safe Recursive one.
An Example: Unary Exponential

Consider the function \( \exp(\overline{x}) = 2^{2|\overline{x}|} \) given by the following program:

\[
\begin{align*}
\text{Append}(\epsilon; \overline{y}) &= \overline{y} \\
\text{Append}(a.\overline{x}; \overline{y}) &= a.\text{Append}(\overline{x}; \overline{y}) \\
\text{Double}(\overline{x}; \overline{x}) &= \text{Append}(\overline{x}; \overline{x}) \\
\exp(\epsilon) &= 1 \\
\exp(a.\overline{x}) &= \text{Double}(\exp(\overline{x});)
\end{align*}
\]

This program is Primitive Recursive, has time complexity \( \exp(|\overline{x}|) \), and cannot be turned into a Safe Recursive one.
Consider the function \( \exp(\overline{x}) = 2^{2^{|\overline{x}|}} \) given by the following program:

\[
\begin{align*}
\text{Append}(\epsilon; \overline{y}) & = \overline{y} \\
\text{Append}(a.\overline{x}; \overline{y}) & = a.\text{Append}(\overline{x}; \overline{y}) \\
\text{Double}(\overline{x}; \overline{x}) & = \text{Append}(\overline{x}; \overline{x}) \\
\exp(\epsilon; \overline{x}) & = 1 \\
\exp(a.\overline{x}; \overline{x}) & = \text{Double}(\overline{x}; \exp(\overline{x});)
\end{align*}
\]

This program is Primitive Recursive, has time complexity \( \exp(|\overline{x}|) \), and cannot be turned into a Safe Recursive one.
Soundness: The execution time is bounded by the length of the normal arguments.

Proof by induction on the number of safe recursion schemes: polynomial bound on the length of normal arguments at any step in the recursion tree -> polynomial bound on the size of the tree.
Soundness: The execution time is bounded by the length of the normal arguments.
Proof by induction on the number of safe recursion schemes: polynomial bound on the length of normal arguments at any step in the recursion tree -> polynomial bound on the size of the tree.

Completeness: reduction from Cobham. Encoding of smash function and of limited recursion on notation by safe recursion on notation.
Safe Recursion with substitutions:

\[ f(\epsilon, x; \overline{u}, \overline{y}) = h(\overline{x}; \overline{u}, \overline{y}) \]

\[ f(a.z, x; \overline{u}, \overline{y}) = g(a, z, x; f(z, x; \sigma_1(\overline{u}), \overline{y}), \ldots, f(z, x; \sigma_k(\overline{u}), \overline{y}), \overline{y}) \]

LM = closure of basic safe functions under safe composition and safe recursion with substitutions.
Safe Recursion with substitutions:

\[ f(\epsilon, x; u, y) = h(x; u, y) \]
\[ f(a, z, x; u, y) = g(a, z, x; f(z, x; \sigma_1(; u), y), \ldots, f(z, x; \sigma_k(; u), y), y). \]

LM = closure of basic safe functions under safe composition and safe recursion with substitutions.

**Theorem 6 (Leivant/Marion 1999)** \( LM \) is sound and extensionally complete for \( \text{PSPACE}. \)
Preliminary: PSPACE=PAR (class of languages decided by P-uniform families of boolean circuits of polynomial depth)
Preliminary: \( \text{PSPACE} = \text{PAR} \) (class of languages decided by \( P \)-uniform families of boolean circuits of polynomial depth)

PAR Soundness: As for BC, polynomial bound on the length of normal arguments at any step in the recursion tree $\rightarrow$ polynomial bound on the depth of the tree. Parallel evaluation of each recursive call $\rightarrow$ PAR algorithm.
Leivant/Marion -2: Proof of Theorem 6

- Preliminary: PSPACE=PAR (class of languages decided by P-uniform families of boolean circuits of polynomial depth)

- PAR Soundness: As for BC, polynomial bound on the length of normal arguments at any step in the recursion tree -> polynomial bound on the depth of the tree. Parallel evaluation of each recursive call -> PAR algorithm.

- PAR Completeness:
  - Theorem 5 yields BC functions for the P-uniformity
  - Substitutions yield left-right numbering of the nodes in the circuits (path from output node to any inner node)
  - Safe recursion with substitutions yields recursive evaluation of circuit nodes (from input nodes to output nodes).
Part II: Quasi-Interpretations
We consider here first order functional programs.

We want to apply Cobham’s idea: bounding the size of values and restricting recurrence gives time bounds.

We want to have more intensional power (capture more “real” algorithms).
First order functional programs

Given three disjoints sets of variables \((x \in V)\), functions \((f \in F)\) and constructors \((c \in C)\), we define the sets of terms and the equations:

\[
\begin{align*}
T(C) &\ni v ::= c(v_1, \ldots, v_n) \\
T(C, F, V) &\ni t ::= x \mid c(t_1, \ldots, t_n) \mid f(t_1, \ldots, t_n) \\
P &\ni p ::= x \mid c(p_1, \ldots, p_n) \\
E &\ni e ::= f(p_1, \ldots, p_n) \to t
\end{align*}
\]
Example: Lists concatenation

List concatenation: $\mathcal{C} = \{[], \text{cons}\}$, $\mathcal{F} = \{\text{concat}\}$, $\mathcal{V} = \{l_1, l_2, x\}$ (all can be inferred from the program).

\[
\text{concat}([], l_2) \rightarrow l_2
\]
\[
\text{concat}(\text{cons}(x, l_1), l_2) \rightarrow \text{cons}(x, \text{concat}(l_1, l_2))
\]
Example: Lists concatenation

List concatenation: $\mathcal{C} = \{[], \text{cons}\}, \mathcal{F} = \{\text{concat}\}, \mathcal{V} = \{l_1, l_2, x\}$ (all can be inferred from the program).

$$\text{concat}([], l_2) \rightarrow l_2$$

$$\text{concat}(\text{cons}(x, l_1), l_2) \rightarrow \text{cons}(x, \text{concat}(l_1, l_2))$$

Same thing as unary addition:

$$\text{add}(Z, y) \rightarrow y$$

$$\text{add}(S(x), y) \rightarrow S(\text{add}(x, y))$$
Example: Lists concatenation

List concatenation: \( C = \{ [], \text{cons} \}, \ F = \{ \text{concat} \}, \ V = \{ l_1, l_2, x \} \) (all can be inferred from the program).

\[
\text{concat}([], l_2) \rightarrow l_2
\]

\[
\text{concat}(\text{cons}(x, l_1), l_2) \rightarrow \text{cons}(x, \text{concat}(l_1, l_2))
\]

Same thing as unary addition:

\[
\text{add}(Z, y) \rightarrow y
\]

\[
\text{add}(S(x), y) \rightarrow S(\text{add}(x, y))
\]

⇒ we work with unary integers, but they’re actually length of lists!

Part II: Quasi-Interpretations
Polynomial bounds

We want to bound the values computed (unary values $\sim$ length of lists). Obviously, if $x = S^n(Z)$ and $y = S^m(Z)$ then $\text{add}(x, y) = S^{n+m}(Z)$.
Polynomial bounds

We want to bound the values computed (unary values $\sim$ length of lists). Obviously, if $x = S^n(Z)$ and $y = S^m(Z)$ then $\text{add}(x, y) = S^{n+m}(Z)$.

How to express this for $\text{concat}$? We need to convert lists to integers using a size function $|\cdot|:

$$\text{If } |l_1| = n \text{ and } |l_2| = m \text{ then } |\text{concat}(l_1, l_2)| = n + m$$
We want to bound the values computed (unary values ~ length of lists).

Obviously, if \( x = S^n(Z) \) and \( y = S^m(Z) \) then \( \text{add}(x, y) = S^{n+m}(Z) \)

How to express this for \( \text{concat} \)? We need to convert lists to integers using a size function \( \cdot \):

If \( |l_1| = n \) and \( |l_2| = m \) then \( |\text{concat}(l_1, l_2)| = n + m \)

But... this is true only for normal forms! (What is the size of a variable or function, anyway?)
Polynomial interpretations

To each term $t$, we associate the size of its normal form, $(t)$. We say that this is an interpretation of the terms:

$$(\text{add}(S(S(Z)), S(S(S(Z))))) = 5$$
Polynomial interpretations

To each term \( t \), we associate the size of its normal form, \( (t) \). We say that this is an interpretation of the terms:

\[
(\text{add}(\text{S}(\text{S}(\text{Z})), \text{S}(\text{S}(\text{S}(\text{Z})))))) = 5
\]

To make this compositional, we interpret each symbol individually:

\[
(\text{Z}) = 0
\]
Polynomial interpretations

To each term $t$, we associate the size of its normal form, $(t)$. We say that this is an interpretation of the terms:

$$\left|\text{add}(S(S(Z)), S(S(S(Z))))\right| = 5$$

To make this compositional, we interpret each symbol individually:

$$(Z) = 0 \quad (S) = \quad =$$
Polynomial interpretations

To each term $t$, we associate the size of its normal form, $(t)$. We say that this is an interpretation of the terms:

$$(\text{add}(S(S(Z)), S(S(S(Z)))))) = 5$$

To make this compositional, we interpret each symbol individually:

$$(Z) = 0 \quad (S)(X) =$$
Polynomial interpretations

To each term $t$, we associate the size of its normal form, $(t)$. We say that this is an interpretation of the terms:

$$\left| \text{add}(S(S(Z)), S(S(S(Z)))) \right| = 5$$

To make this compositional, we interpret each symbol individually:

$$(Z) = 0 \quad (S)(X) = X + 1$$
Polynomial interpretations

To each term \( t \), we associate the size of its normal form, \( (t) \). We say that this is an interpretation of the terms:

\[
(\text{add}(\text{S}(\text{S}(\text{Z})), \text{S}(\text{S}(\text{S}(\text{Z})))))) = 5
\]

To make this compositional, we interpret each symbol individually:

\[
(\text{Z}) = 0 \quad (\text{S})(X) = X + 1
\]

(\text{S}(\text{S}(\text{Z}))) =
Polynomial interpretations

To each term $t$, we associate the size of its normal form, $(t)$. We say that this is an interpretation of the terms:

$$(\text{add}(\text{S}(\text{S}(\text{Z})), \text{S}(\text{S}(\text{S}(\text{Z})))))) = 5$$

To make this compositional, we interpret each symbol individually and compose in the obvious way:

$$(\text{Z}) = 0 \quad (\text{S})(X) = X + 1$$

$$(\text{S}(\text{S}(\text{Z})))) =$$
Polynomial interpretations

To each term \( t \), we associate the size of its normal form, \( (t) \). We say that this is an interpretation of the terms:

\[
(|\text{add}(S(S(Z)), S(S(S(Z))))|) = 5
\]

To make this compositional, we interpret each symbol individually and compose in the obvious way:

\[
(Z) = 0 \quad (S)(X) = X + 1
\]

\[
(|S(S(Z))|) = (|S|)(\quad )
\]
Polynomial interpretations

To each term $t$, we associate the size of its normal form, $(t)$. We say that this is an interpretation of the terms:

$$(\text{add}(\text{S}(\text{S}(\text{Z})), \text{S}(\text{S}(\text{S}(\text{Z})))))) = 5$$

To make this compositional, we interpret each symbol individually and compose in the obvious way:

$$(\text{Z}) = 0 \quad (\text{S})(X) = X + 1$$

$$(\text{S}(\text{S}(\text{Z})))) = (\text{S})( (\text{S}(\text{Z}))))$$
Polynomial interpretations

To each term \( t \), we associate the size of its normal form, \( \langle t \rangle \). We say that this is an interpretation of the terms:

\[
\langle \text{add}(S(S(Z)), S(S(S(Z)))) \rangle = 5
\]

To make this compositional, we interpret each symbol individually and compose in the obvious way:

\[
\langle Z \rangle = 0 \quad \langle S \rangle(X) = X + 1
\]

\[
\langle S(S(Z)) \rangle = \langle S \rangle(\langle S(Z) \rangle) = (\langle S(Z) \rangle) + 1
\]
Polynomial interpretations

To each term $t$, we associate the size of its normal form, $(t)$. We say that this is an interpretation of the terms:

$$(\text{add}(\text{S}(\text{S}(\text{Z})), \text{S}(\text{S}(\text{S}(\text{Z})))))) = 5$$

To make this compositional, we interpret each symbol individually and compose in the obvious way:

$$(\text{Z}) = 0 \quad (\text{S})(X) = X + 1$$

$$((\text{S}(\text{S}(\text{Z})))) = (\text{S})((\text{S}(\text{Z})))) = ((\text{S}(\text{Z})))) + 1 = (\text{S})((\text{Z}))) + 1$$
Polynomial interpretations

To each term $t$, we associate the size of its normal form, $(t)$. We say that this is an interpretation of the terms:

$$(\text{add}(\text{S}(\text{S}(\text{Z})), \text{S}(\text{S}(\text{S}(\text{Z})))))) = 5$$

To make this compositional, we interpret each symbol individually and compose in the obvious way:

$$(\text{Z}) = 0 \quad (\text{S})(X) = X + 1$$

$$(\text{S}(\text{S}(\text{Z})))) = (\text{S})(((\text{S}(\text{Z})))) = ((\text{S}(\text{Z})))) + 1 = ((\text{S})(\text{Z})))) + 1 = \ldots = 2$$
Polynomial interpretations

To each term $t$, we associate the size of its normal form, $(t)$. We say that this is an *interpretation* of the terms:

$$\left(\text{add}(S(S(Z)), S(S(S(Z))))\right) = 5$$

To make this compositional, we interpret each symbol individually and compose in the obvious way:

$$\begin{align*}
(Z) &= 0 & (S)(X) &= X + 1 \\
(S(S(Z))) &= (S)((S(Z))) = ((S(Z))) + 1 = (S)((Z))) + 1 = \ldots = 2
\end{align*}$$

$$\text{add}(X, Y) = X + Y$$
Polynomial interpretations

To each term $t$, we associate the size of its normal form, $(t)$. We say that this is an interpretation of the terms:

$$|\text{add}(S(S(Z)), S(S(S(Z))))| = 5$$

To make this compositional, we interpret each symbol individually and compose in the obvious way:

$$(Z) = 0 \quad (S)(X) = X + 1$$

$$(S(S(Z))) = (S)((S(Z))) = ((S(Z))) + 1 = ((S)(Z))) + 1 = \ldots = 2$$

$$(\text{add})(X, Y) = X + Y$$

$$|\text{add}(S(S(Z)), S(S(S(Z))))| = |\text{add}|(, )$$
Polynomial interpretations

To each term $t$, we associate the size of its normal form, $(t)$. We say that this is an interpretation of the terms:

$$(\text{add}(\text{S}(\text{S}(\text{Z})), \text{S}(\text{S}(\text{S}(\text{Z})))))) = 5$$

To make this compositional, we interpret each symbol individually and compose in the obvious way:

$$(\text{Z}) = 0 \quad (\text{S})(X) = X + 1$$

$$(\text{S}(\text{S}(\text{Z}))) = (\text{S})(\text{S}(\text{Z})) = (\text{S}(\text{Z})) + 1 = (\text{S})(\text{S}(\text{Z})) + 1 = \ldots = 2$$

$$(\text{add})(X, Y) = X + Y$$

$$(\text{add}(\text{S}(\text{S}(\text{Z})), \text{S}(\text{S}(\text{S}(\text{Z})))))) = (\text{add})(\text{S}(\text{S}(\text{Z})), \text{S}(\text{S}(\text{S}(\text{Z}))))))$$
Polynomial interpretations

To each term $t$, we associate the size of its normal form, $(t)$. We say that this is an interpretation of the terms:

$$\langle \text{add}(S(S(Z)), S(S(S(Z)))) \rangle = 5$$

To make this compositional, we interpret each symbol individually and compose in the obvious way:

$$(Z) = 0 \quad (S)(X) = X + 1$$

$$(S(S(Z))) = (S)((S(Z))) = ((S(Z))) + 1 = ((S)(S(Z))) + 1 = \ldots = 2$$

$$(\text{add})(X, Y) = X + Y$$

$$\langle \text{add}(S(S(Z)), S(S(S(Z)))) \rangle = \langle S(S(Z)) \rangle + \langle S(S(S(Z))) \rangle$$
Polynomial interpretations

To each term $t$, we associate the size of its normal form, $(t)$. We say that this is an interpretation of the terms:

$$(\text{add}(S(S(Z)), S(S(S(Z))))\rangle = 5$$

To make this compositional, we interpret each symbol individually and compose in the obvious way:

$$(Z) = 0 \quad (S)(X) = X + 1$$

$$(S(S(Z))) = (S)((S(Z))) = ((S(Z))) + 1 = (S)((Z))) + 1 = \ldots = 2$$

$$(\text{add})(X, Y) = X + Y$$

$$(\text{add}(S(S(Z)), S(S(S(Z))))\rangle = (S(S(Z))) + (S(S(S(Z))))\rangle = \ldots = 5$$
Interpretation of programs

A program admits an interpretation if each rule $l \rightarrow r$ verifies: $(l) \geq (r)$. 
A program admits an interpretation if each rule $l \rightarrow r$ verifies: $(l) \geq (r)$.

$$\text{add}(S(x), y) \rightarrow S(\text{add}(x, y))$$

$$\langle S \rangle (X) = X + 1 \quad \langle \text{add} \rangle (X, Y) = X + Y$$

$$\langle \text{add}(S(x), y) \rangle =$$
A program admits an interpretation if each rule \( l \rightarrow r \) verifies: \( (l) \geq (r) \).

\[
\text{add}(\text{S}(x), y) \rightarrow \text{S}(\text{add}(x, y))
\]

\[
(\text{S})(X) = X + 1 \quad (\text{add})(X, Y) = X + Y
\]

\[
(\text{add}(\text{S}(x), y)) = (\text{add})(\langle \text{S}(x) \rangle, \langle y \rangle)
\]
A program admits an interpretation if each rule \( l \rightarrow r \) verifies: \( (l) \geq (r) \).

\[
\text{add}(\mathbf{S}(x), y) \rightarrow \mathbf{S}(\text{add}(x, y))
\]

\[
(\mathbf{S})(X) = X + 1 \quad (\text{add})(X, Y) = X + Y
\]

\[
(\text{add}(\mathbf{S}(x), y)) = (\text{add})(\mathbf{S}(x), y)
\]

\[
= (\mathbf{S}(x)) + y
\]
Interpretation of programs

A program admits an interpretation if each rule \( l \rightarrow r \) verifies: \( (l) \geq (r) \).

\[
\text{add}(S(x), y) \rightarrow S(\text{add}(x, y))
\]

\[
(S)(X) = X + 1 \quad (\text{add})(X, Y) = X + Y
\]

\[
(\text{add}(S(x), y)) = (\text{add})(S(x), y) = S(x) + y
\]

\[
(S(x)) = (x) + 1
\]
A program admits an interpretation if each rule $l \rightarrow r$ verifies: $(l) \geq (r)$.

$$\text{add}(S(x), y) \rightarrow S(\text{add}(x, y))$$

$$(S)(X) = X + 1 \quad (\text{add})(X, Y) = X + Y$$

$$(\text{add}(S(x), y)) = (\text{add})(S(x), (y)) = S(x) + (y)$$

$$(S(x)) = (x) + 1 = X + 1$$
A program admits an interpretation if each rule \( l \rightarrow r \) verifies: \( (l) \geq (r) \).

\[
\text{add}(\text{S}(x), y) \rightarrow \text{S}(\text{add}(x, y))
\]

\[
\text{(S)}(X) = X + 1 \quad \text{(add)}(X, Y) = X + Y
\]

\[
\text{(add(S(x), y))} = \text{(add)}(\text{(S(x))}, \text{(y)})
\]

\[
= \text{(S(x))} + \text{(y)}
\]

\[
= (X + 1) + Y
\]

\[
\text{(S(x))} = (x) + 1
\]

\[
= X + 1
\]
Interpretation of programs

A program admits an interpretation if each rule \( l \rightarrow r \) verifies: \( \langle l \rangle \geq \langle r \rangle \).

\[
\text{add}(\text{S}(x), y) \rightarrow \text{S}(\text{add}(x, y))
\]

\[
\langle \text{S} \rangle(X) = X + 1 \quad \langle \text{add} \rangle(X, Y) = X + Y
\]

\[
\langle \text{add}(\text{S}(x), y) \rangle = \langle \text{add} \rangle(\langle \text{S}(x) \rangle, \langle y \rangle)
\]

\[
= \langle \text{S}(x) \rangle + \langle y \rangle
\]

\[
= (X + 1) + Y
\]

\[
= (X + Y) + 1
\]
Interpretation of programs

A program admits an interpretation if each rule $l \rightarrow r$ verifies: $(l) \geq (r)$.

$$
\text{add}(S(x), y) \rightarrow S(\text{add}(x, y))
$$

$$
(S)(X) = X + 1 \quad \langle\text{add}\rangle(X, Y) = X + Y
$$

$$
\langle\text{add}(S(x), y)\rangle = \langle\text{add}\rangle(\langle S(x) \rangle, \langle y \rangle)
$$

$$
= \langle S(x) \rangle + \langle y \rangle
$$

$$
= (X + 1) + Y
$$

$$
= (X + Y) + 1
$$

$$
= \langle \text{add}(x, y) \rangle + 1
$$
A program admits an interpretation if each rule $l \rightarrow r$ verifies: $(l) \geq (r)$.

$$\text{add}(S(x), y) \rightarrow S(\text{add}(x, y))$$

$$(S)(X) = X + 1 \quad \text{add}(X, Y) = X + Y$$

$$(\text{add}(S(x), y)) = (\text{add})(S(x), y) = (S(x)) + y = (X + 1) + Y = (X + Y) + 1 = (\text{add}(x, y)) + 1 = (S(\text{add}(x, y)))$$
Other examples

\[ \text{mult}(\mathbb{Z}, y) \rightarrow \mathbb{Z} \]

\[ \text{mult}(\mathbb{S}(x), y) \rightarrow \text{add}(y, \text{mult}(x, y)) \]
Other examples

\[
\text{mult}(Z, y) \to Z
\]

\[
\text{mult}(S(x), y) \to \text{add}(y, \text{mult}(x, y))
\]

\[
\text{concat}([], l_2) \to l_2
\]

\[
\text{concat(\text{cons}(x, l_1), l_2) \to \text{cons}(x, \text{concat}(l_1, l_2))}
\]
Other examples

\[
\text{mult}(Z, y) \rightarrow Z
\]

\[
\text{mult}(S(x), y) \rightarrow \text{add}(y, \text{mult}(x, y))
\]

\[
(\text{mult})(X, Y) = X \times Y
\]
Other examples

$\text{concat}([], l_2) \rightarrow l_2$

$\text{concat}(\text{cons}(x, l_1), l_2) \rightarrow \text{cons}(x, \text{concat}(l_1, l_2))$

$\langle [] \rangle = 0 \quad \langle \text{cons} \rangle (A, L) = A + L \quad \langle \text{concat} \rangle (L, L') = L + L'$
When polynomial interpretations don’t exist

\[\text{dbl}(\mathbb{Z}) \rightarrow \mathbb{Z}\]

\[\text{dbl}(S(x)) \rightarrow S(S(\text{dbl}(x)))\]
When polynomial interpretations don’t exist

\[
\begin{align*}
\text{dbl}(Z) & \rightarrow Z \\
\text{dbl}(S(x)) & \rightarrow S(S(\text{dbl}(x))) \\
\exp(Z) & \rightarrow S(Z) \\
\exp(S(x)) & \rightarrow \text{dbl}(\exp(x))
\end{align*}
\]
When polynomial interpretations don’t exist

\[
\text{dbl}(Z) \rightarrow Z \quad \text{exp}(Z) \rightarrow S(Z) \\
\text{dbl}(S(x)) \rightarrow S(S(\text{dbl}(x))) \quad \text{exp}(S(x)) \rightarrow \text{dbl}(\text{exp}(x))
\]

\[
(Z) = 0 \quad (S)(X) = X + 1
\]
When polynomial interpretations don’t exist

\[
\begin{align*}
    \text{dbl}(Z) & \rightarrow Z \\
    \text{dbl}(S(x)) & \rightarrow S(S(\text{dbl}(x))) \\
    \exp(Z) & \rightarrow S(Z) \\
    \exp(S(x)) & \rightarrow \text{dbl}(\exp(x))
\end{align*}
\]

\( (Z) = 0 \quad (S)(X) = X + 1 \)

\( (\text{dbl}(S(x))) \geq (S(S(\text{dbl}(x)))) \)
When polynomial interpretations don’t exist

\[
\begin{align*}
\text{dbl}(Z) & \rightarrow Z & \text{exp}(Z) & \rightarrow S(Z) \\
\text{dbl}(S(x)) & \rightarrow S(S(\text{dbl}(x))) & \text{exp}(S(x)) & \rightarrow \text{dbl}(\text{exp}(x))
\end{align*}
\]

\[
(Z) = 0 \quad (S)(X) = X + 1
\]

\[
\begin{align*}
(\text{dbl}(S(x))) & \geq (S(S(\text{dbl}(x)))) \\
(\text{dbl})(X + 1) & \geq (\text{dbl})(X) + 2
\end{align*}
\]
When polynomial interpretations don’t exist

\[
\begin{align*}
\text{dbl}(Z) & \rightarrow Z & \text{exp}(Z) & \rightarrow S(Z) \\
\text{dbl}(S(x)) & \rightarrow S(S(\text{dbl}(x))) & \text{exp}(S(x)) & \rightarrow \text{dbl}(\text{exp}(x)) \\
\end{align*}
\]

\[
(Z) = 0 \quad (S)(X) = X + 1
\]

\[
\begin{align*}
|\text{dbl}(S(x))| & \geq |S(S(\text{dbl}(x)))| \\
|\text{dbl}(X + 1)| & \geq |\text{dbl}(X) + 2 \\
\vdots \\
|\text{dbl}(X)| & \geq 2X
\end{align*}
\]
When polynomial interpretations don’t exist

\[
\begin{align*}
\text{dbl}(Z) & \rightarrow Z & \text{exp}(Z) & \rightarrow S(Z) \\
\text{dbl}(S(x)) & \rightarrow S(S(\text{dbl}(x))) & \text{exp}(S(x)) & \rightarrow \text{dbl}(\text{exp}(x)) \\
\end{align*}
\]

\[
(Z) = 0 \quad (S)(X) = X + 1
\]

\[
\begin{align*}
(|\text{dbl}(S(x))|) & \geq (|S(S(\text{dbl}(x)))|) & (|\text{exp}(S(x))|) & \geq (|\text{dbl}(\text{exp}(x))|) \\
(|\text{dbl}|)(X + 1) & \geq (|\text{dbl}|)(X) + 2 & \vdots & \\
(|\text{dbl}|)(X) & \geq 2X
\end{align*}
\]
When polynomial interpretations don’t exist

\[
\begin{align*}
\text{dbl}(Z) & \rightarrow Z & \text{exp}(Z) & \rightarrow S(Z) \\
\text{dbl}(S(x)) & \rightarrow S(S(\text{dbl}(x))) & \text{exp}(S(x)) & \rightarrow \text{dbl}(\text{exp}(x))
\end{align*}
\]

\[
(Z) = 0 \quad (S)(X) = X + 1
\]

\[
\begin{align*}
(\text{dbl}(S(x))) & \geq (S(S(\text{dbl}(x)))) \\
(\text{dbl})(X + 1) & \geq (\text{dbl})(X) + 2 \\
(\text{exp}(S(x))) & \geq (\text{dbl}(\text{exp}(x))) \\
(\text{exp})(X + 1) & \geq 2 \cdot (\text{exp})(X) \\
\vdots
\end{align*}
\]

\[
(\text{dbl})(X) \geq 2X
\]
When polynomial interpretations don’t exist

\[
\begin{align*}
\text{dbl}(\mathbb{Z}) & \rightarrow \mathbb{Z} & \text{exp}(\mathbb{Z}) & \rightarrow S(\mathbb{Z}) \\
\text{dbl}(S(x)) & \rightarrow S(S(\text{dbl}(x))) & \text{exp}(S(x)) & \rightarrow \text{dbl}(\text{exp}(x))
\end{align*}
\]

\[
(Z) = 0 \quad (S)(X) = X + 1
\]

\[
\begin{align*}
(\text{dbl}(S(x))) & \geq (S(S(\text{dbl}(x)))) & (\text{exp}(S(x))) & \geq (\text{dbl}(\text{exp}(x))) \\
(\text{dbl})(X + 1) & \geq (\text{dbl})(X) + 2 & (\text{exp})(X + 1) & \geq 2 \cdot (\text{exp})(X) \\
& \vdots & & \vdots \\
(\text{dbl})(X) & \geq 2X & (\text{exp})(X) & \geq 2^X
\end{align*}
\]
Properties of interpretations

- If the program admits an interpretation, then the size of the normal form is bounded by the interpretation.
- If the interpretation is polynomial and “additive” (cf next slide), then the program only computes polynomially bounded values.
- Interpretation is not sufficient for termination: $f(x) \rightarrow f(x)$
- Interpretation + Termination $\Rightarrow$ complexity bound ($\text{P} \text{TIME}$ or $\text{P} \text{SPACE}$, depending on the termination proof).
Cheating with constructors

(Bonfante, Cichon, Marion, Touzet, 1998)

\[ \text{dbl}(Z) \rightarrow Z \quad \quad \quad \text{exp}(Z) \rightarrow S(Z) \]
\[ \text{dbl}(S(x)) \rightarrow S(S(\text{dbl}(x))) \quad \text{exp}(S'(x)) \rightarrow \text{dbl}(\exp(x)) \]
Cheating with constructors

(Bonfante, Cichon, Marion, Touzet, 1998)

\[
\begin{align*}
\text{dbl}(Z) & \rightarrow Z & \text{exp}(Z) & \rightarrow S(Z) \\
\text{dbl}(S(x)) & \rightarrow S(S(\text{dbl}(x))) & \text{exp}(S'(x)) & \rightarrow \text{dbl}(\text{exp}(x))
\end{align*}
\]

\[
\langle Z \rangle = 0 \quad \langle S \rangle(X) = X + 1
\]
Cheating with constructors

(Bonfante, Cichon, Marion, Touzet, 1998)

\[
\begin{align*}
\text{dbl}(Z) & \rightarrow Z & \text{exp}(Z) & \rightarrow S(Z) \\
\text{dbl}(S(x)) & \rightarrow S(S(\text{dbl}(x))) & \text{exp}(S'(x)) & \rightarrow \text{dbl}(\text{exp}(x))
\end{align*}
\]

\[
(\text{Z}) = 0 \quad (\text{S})(X) = X + 1 \quad (\text{S}')(X) = 2X + 1
\]
Cheating with constructors

(Bonfante, Cichon, Marion, Touzet, 1998)

\[
\begin{align*}
\text{dbl}(Z) & \rightarrow Z & \text{exp}(Z) & \rightarrow S(Z) \\
\text{dbl}(S(x)) & \rightarrow S(S(\text{dbl}(x))) & \text{exp}(S'(x)) & \rightarrow \text{dbl}(\text{exp}(x))
\end{align*}
\]

\[
\begin{align*}
(Z) & = 0 & (S)(X) & = X + 1 & (S')(X) & = 2X + 1 \\
(\text{dbl})(X) & = 2X
\end{align*}
\]
Cheating with constructors

(Bonfante, Cichon, Marion, Touzet, 1998)

\begin{align*}
\text{dbl}(\mathbb{Z}) & \rightarrow \mathbb{Z} & \text{exp}(\mathbb{Z}) & \rightarrow \mathbb{S}(\mathbb{Z}) \\
\text{dbl}(\mathbb{S}(x)) & \rightarrow \mathbb{S}(\mathbb{S}(\text{dbl}(x))) & \text{exp}(\mathbb{S}')(x) & \rightarrow \text{dbl}(\text{exp}(x))
\end{align*}

\begin{align*}
\langle \mathbb{Z} \rangle &= 0 & \langle \mathbb{S} \rangle(X) &= X + 1 & \langle \mathbb{S}' \rangle(X) &= 2X + 1 \\
\langle \text{dbl} \rangle(X) &= 2X & \langle \text{exp} \rangle(X) &= X + 1
\end{align*}
Cheating with constructors

(Bonfante, Cichon, Marion, Touzet, 1998)

\[
\begin{align*}
\text{dbl}(Z) & \rightarrow Z \quad \text{exp}(Z) \rightarrow S(Z) \\
\text{dbl}(S(x)) & \rightarrow S(S(\text{dbl}(x))) \quad \text{exp}(S'(x)) \rightarrow \text{dbl}(\text{exp}(x))
\end{align*}
\]

\[
\begin{align*}
\langle Z \rangle &= 0 & \langle S \rangle(X) &= X + 1 & \langle S' \rangle(X) &= 2X + 1 \\
\langle \text{dbl} \rangle(X) &= 2X & \langle \text{exp} \rangle(X) &= X + 1 \\
\langle \text{exp}(S'(x)) \rangle &= \langle \text{exp} \rangle(\langle S' \rangle(x)) &= \langle \text{exp} \rangle(2X + 1) = 2X + 2 \geq 2(X + 1)
\end{align*}
\]
Cheating with constructors

(Bonfante, Cichon, Marion, Touzet, 1998)

\[
\begin{align*}
\text{dbl}(Z) & \rightarrow Z & \exp(Z) & \rightarrow S(Z) \\
\text{dbl}(S(x)) & \rightarrow S(S(\text{dbl}(x))) & \exp(S'(x)) & \rightarrow \text{dbl}(\exp(x))
\end{align*}
\]

\[
\begin{align*}
(Z) &= 0 & (S)(X) &= X + 1 & (S')(X) &= 2X + 1 \\
(\text{dbl})(X) &= 2X & (\exp)(X) &= X + 1 \\
(\exp(S'(x))) &= (\exp)((S'(x))) = (\exp)(2X + 1) = 2X + 2 \geq 2(X + 1) \\
\exp(S'^n(Z)) &\rightarrow S^{2^n}(Z)
\end{align*}
\]
Cheating with constructors

(Bonfante, Cichon, Marion, Touzet, 1998)

\[
\begin{align*}
\text{dbl}(Z) & \rightarrow Z \\
\text{dbl}(S(x)) & \rightarrow S(S(\text{dbl}(x))) \\
\text{exp}(Z) & \rightarrow S(Z) \\
\text{exp}(S'(x)) & \rightarrow \text{dbl}(\text{exp}(x))
\end{align*}
\]

\[
\begin{align*}
\langle Z \rangle &= 0 \\
\langle S \rangle(X) &= X + 1 \\
\langle S' \rangle(X) &= 2X + 1
\end{align*}
\]

\[
\begin{align*}
\langle \text{dbl} \rangle(X) &= 2X \\
\langle \text{exp} \rangle(X) &= X + 1
\end{align*}
\]

\[
\langle \text{exp}(S'(x)) \rangle = \langle \text{exp}(\langle S' \rangle(x)) \rangle = \langle \text{exp} \rangle(2X + 1) = 2X + 2 \geq 2(X + 1)
\]

\[
\text{exp}(S'^n(Z)) \overset{!}{\rightarrow} S^{2^n}(Z)
\]

\[
\langle S^n(Z) \rangle \geq 2^n \Rightarrow \text{exponential “potential” stored in a value.}
\]
Cheating with constructors

(Bonfante, Cichon, Marion, Touzet, 1998)

\[
\begin{align*}
\text{dbl}(Z) & \rightarrow Z & \text{exp}(Z) & \rightarrow S(Z) \\
\text{dbl}(S(x)) & \rightarrow S(S(\text{dbl}(x))) & \text{exp}(S'(x)) & \rightarrow \text{dbl}(\text{exp}(x)) \\
\langle Z \rangle & = 0 & \langle S \rangle (X) & = X + 1 & \langle S' \rangle (X) & = 2X + 1 \\
\langle \text{dbl} \rangle (X) & = 2X & \langle \text{exp} \rangle (X) & = X + 1 \\
\langle \text{exp}(S'(x)) \rangle & = \langle \text{exp}(S'(x)) \rangle & = \langle \text{exp} \rangle (2X + 1) & = 2X + 2 \geq 2(X + 1) \\
\text{exp}(S'^n(Z)) & \rightarrow S^{2^n}(Z) \\
\langle S'^n(Z) \rangle & \geq 2^n \Rightarrow \text{exponential “potential” stored in a value.}
\end{align*}
\]

Additive Interpretation: \(\langle c \rangle (X_1, \cdots, X_n) = \sum X_i + \alpha\)
Polynomials are not enough

Computing the Longest Common Subsequence:

\[ \text{AABBA} \]

\[ \text{BABAABA} \]
Polynomials are not enough

Computing the Longest Common Subsequence:

\[ AABBA \]

\[ BABAABBA \]
Polynomials are not enough

Computing the Longest Common Subsequence:

\[ AABBA \]

\[ BABAAABBA \]
Polynomials are not enough

Computing the Longest Common Subsequence:

\[
\begin{align*}
AABBA \\
BABAABBA
\end{align*}
\]

\[
\begin{align*}
lcs(x, \epsilon) & \rightarrow \mathbb{Z} \\
lcs(\epsilon, y) & \rightarrow \mathbb{Z} \\
lcs(i(x), i(y)) & \rightarrow S(lcs(x, y)) \\
lcs(i(x), j(y)) & \rightarrow \max(lcs(x, j(y)), lcs(i(x), y))
\end{align*}
\]
Polynomials are not enough

\[
\text{max}(\mathbb{Z}, n) \rightarrow n \\
\text{max}(m, \mathbb{Z}) \rightarrow m \\
\text{max}(S(m), S(n)) \rightarrow S(\text{max}(m, n)) \\
\text{lcs}(x, \epsilon) \rightarrow \mathbb{Z} \\
\text{lcs}(\epsilon, y) \rightarrow \mathbb{Z} \\
\text{lcs}(i(x), i(y)) \rightarrow S(\text{lcs}(x, y)) \\
\text{lcs}(i(x), j(y)) \rightarrow \text{max}(\text{lcs}(x, j(y)), \text{lcs}(i(x), y))
\]
Polynomials are not enough

\[ \max(Z, n) \rightarrow n \]
\[ \max(m, Z) \rightarrow m \]
\[ \max(S(m), S(n)) \rightarrow S(\max(m, n)) \]

\[ \text{lcs}(x, \epsilon) \rightarrow Z \]
\[ \text{lcs}(\epsilon, y) \rightarrow Z \]
\[ \text{lcs}(i(x), i(y)) \rightarrow S(\text{lcs}(x, y)) \]
\[ \text{lcs}(i(x), j(y)) \rightarrow \max(\text{lcs}(x, j(y)), \text{lcs}(i(x), y)) \]

\[ (\max)(X, Y) = X + Y \]
Polynomials are not enough

\[
\max(Z, n) \rightarrow n
\]
\[
\max(m, Z) \rightarrow m
\]
\[
\max(S(m), S(n)) \rightarrow S(\max(m, n))
\]
\[
lcs(x, \epsilon) \rightarrow Z
\]
\[
lcs(\epsilon, y) \rightarrow Z
\]
\[
lcs(i(x), i(y)) \rightarrow S(lcs(x, y))
\]
\[
lcs(i(x), j(y)) \rightarrow \max(lcs(x, j(y)), lcs(i(x), y))
\]
\[
(\max)(X, Y) = X + Y \quad (\lcs)(X, Y) =
\]
Polynomials are not enough

\[
\max(Z, n) \rightarrow n \\
\max(m, Z) \rightarrow m \\
\max(S(m), S(n)) \rightarrow S(\max(m, n))
\]

\[
lcs(x, \epsilon) \rightarrow Z \\
lcs(\epsilon, y) \rightarrow Z \\
lcs(i(x), i(y)) \rightarrow S(lcs(x, y)) \\
lcs(i(x), j(y)) \rightarrow \max(lcs(x, j(y)), lcs(i(x), y))
\]

\[
(\max)(X, Y) = X + Y \quad (\mathsf{lcs})(X, Y) = \ldots
\]
Polynomials are not enough

\[
\begin{align*}
\max(Z, n) & \rightarrow n \\
\max(m, Z) & \rightarrow m \\
\max(S(m), S(n)) & \rightarrow S(\max(m, n)) \\
lcs(x, \epsilon) & \rightarrow Z \\
lcs(\epsilon, y) & \rightarrow Z \\
lcs(i(x), i(y)) & \rightarrow S(lcs(x, y)) \\
lcs(i(x), j(y)) & \rightarrow \max(lcs(x, j(y)), lcs(i(x), y)) \\
\langle\max\rangle(X, Y) & = X + Y \quad \langle lcs\rangle(X, Y) = \ldots \\
\langle\max\rangle(X, Y) & = \max(X, Y)
\end{align*}
\]
Polynomials are not enough

\[ \max(Z, n) \rightarrow n \]
\[ \max(m, Z) \rightarrow m \]
\[ \max(S(m), S(n)) \rightarrow S(\max(m, n)) \]

\[ \text{lcs}(x, \epsilon) \rightarrow Z \]
\[ \text{lcs}(\epsilon, y) \rightarrow Z \]
\[ \text{lcs}(i(x), i(y)) \rightarrow S(\text{lcs}(x, y)) \]
\[ \text{lcs}(i(x), j(y)) \rightarrow \max(\text{lcs}(x, j(y)), \text{lcs}(i(x), y)) \]

\[ (\text{max})(X, Y) = X + Y \quad (\text{lcs})(X, Y) = \ldots \]
\[ (\text{max})(X, Y) = \max(X, Y) \quad (\text{lcs})(X, Y) = \max(X, Y) \]
Polynomial Quasi-interpretations

(Bonfante, Marion, Moyen, 2000)

A polynomial quasi-interpretation of a symbol $a$ is a function $(a)$ such that:
A polynomial quasi-interpretation of a symbol $a$ is a function $(a)$ such that:

- $(a)$ is bounded by a polynomial.
A polynomial quasi-interpretation of a symbol $a$ is a function $\langle a \rangle$ such that:

- $\langle a \rangle$ is bounded by a polynomial.
- $\langle c \rangle(X_1, \ldots, X_n) = \sum X_i + \alpha_c$
Polynomial Quasi-interpretations

(Bonfante, Marion, Moyen, 2000)

A polynomial quasi-interpretation of a symbol \( a \) is a function \( (a) \) such that:

- \( (a) \) is bounded by a polynomial.

- \( (c)(X_1, \ldots, X_n) = \sum X_i + \alpha_c \)

- \( (a)(X_1, \ldots, X_n) \geq X_i \) for all \( i \).

- \( (a) \) is (non-strictly) increasing.
Polynomial Quasi-interpreations

(Bonfante, Marion, Moyen, 2000)

A polynomial quasi-interpretation of a symbol $a$ is a function $(a)$ such that:

- $(a)$ is bounded by a polynomial.
- $(c)(X_1, \cdots, X_n) = \sum X_i + \alpha_c$
- $(a)(X_1, \cdots, X_n) \geq X_i$ for all $i$.
- $(a)$ is (non-strictly) increasing.

\[
(a(t_1, \cdots, t_n)) = (a)((t_1), \cdots, (t_n))
\]
A polynomial quasi-interpretation of a symbol $a$ is a function $(a)$ such that:

- $(a)$ is bounded by a polynomial.
- $(c)(X_1, \ldots, X_n) = \sum X_i + \alpha_c$
- $(a)(X_1, \ldots, X_n) \geq X_i$ for all $i$.
- $(a)$ is (non-strictly) increasing.

$$(a(t_1, \ldots, t_n)) = (a)((t_1), \ldots, (t_n))$$

A system has a Quasi-Interpretation if each rule $l \rightarrow r$ verifies $(r) \leq (l)$. 
A polynomial quasi-interpretation of a symbol $a$ is a function $\langle a \rangle$ such that:

- $\langle a \rangle$ is bounded by a polynomial.
- $\langle c \rangle(X_1, \cdots, X_n) = \sum X_i + \alpha_c$
- $\langle a \rangle(X_1, \cdots, X_n) \geq X_i$ for all $i$.
- $\langle a \rangle$ is (non-strictly) increasing.

$$\langle a(t_1, \cdots, t_n) \rangle = \langle a \rangle(\langle t_1 \rangle, \cdots, \langle t_n \rangle)$$

A system has a Quasi-Interpretation if each rule $l \rightarrow r$ verifies $\langle r \rangle \leq \langle l \rangle$. Termination is not ensured ($f(x) \rightarrow f(x)$).
For constructors terms: \( (v) \approx |v| \)
Quasi-interpretations and size

- For constructors terms: \((v) \approx |v|\)
- During reductions: \(t \rightarrow s \Rightarrow (t) \geq (s)\).
Quasi-interpretations and size

- For constructors terms: $\langle v \rangle \approx |v|$

- During reductions: $t \xrightarrow{+} s \Rightarrow \langle t \rangle \geq \langle s \rangle$.

“Plug and play” Lemma:

If $f(v_1, \ldots, v_n) \xrightarrow{!} v$, then $|v| \leq P(|v_1|, \ldots, |v_n|)$
For constructors terms: \( (v) \approx |v| \)

During reductions: \( t \xrightarrow{+} s \Rightarrow (|t|) \geq (|s|) \).

"Plug and play" Lemma:
If \( f(v_1, \ldots, v_n) \overset{!}{\rightarrow} v \), then \(|v| \leq P(|v_1|, \ldots, |v_n|)\)

\[ |v| \approx (v) \leq (f(v_1, \ldots, v_n)) \leq P(|v_1|, \ldots, |v_n|) \approx P(|v_1|, \ldots, |v_n|) \]
Termination orderings

(Dershowitz, Mana & Ness, Kamin & Levy, Jouannaud)

Monotonous and well-founded ordering $\prec$ on (closed) terms.
Termination orderings

(Dershowitz, Mana & Ness, Kamin & Levy, Jouannaud)

Monotonous and well-founded ordering $\prec$ on (closed) terms.

A program has a termination ordering if each rule $l \rightarrow r$, verifies $r \prec l$. Then, the program terminates on all inputs.
Termination orderings

(Dershowitz, Mana & Ness, Kamin & Levy, Jouannaud)

Monotonous and well-founded ordering $\prec$ on (closed) terms.

A program has a termination ordering if each rule $l \rightarrow r$, verifies $r \prec l$. Then, the program terminates on all inputs.

Each reduction “decreases” a subterm (rules are ordered), hence the whole term (the order is monotonous). So, the number of reduction steps is finite (well-foundness).
Recursive Path Ordering

\[ t = f(t_1, \cdots, t_n) \prec_{rpo} g(s_1, \cdots, s_m) = s \]
Recursive Path Ordering

(Dershowitz)

\[ t = f(t_1, \ldots, t_n) \prec_{rpo} g(s_1, \ldots, s_m) = s \]

\[ \exists i, t \preceq_{rpo} s_i \]

\[ t \preceq_{rpo} s \]
Recursive Path Ordering

(Dershowitz)

\[ t = f(t_1, \ldots, t_n) \prec_{rpo} g(s_1, \ldots, s_m) = s \]

\( \prec_F \) is an ordering of \( F \cup C \).

\[ \exists i, t \preceq_{rpo} s_i \]

\[ t \preceq_{rpo} s \]

\[ f \prec_F g \]
Recursive Path Ordering

(Dershowitz)

\[ t = f(t_1, \cdots, t_n) \prec_{rpo} g(s_1, \cdots, s_m) = s \]

\(<_F\) is an ordering of \(F \cup C\).

\[
\exists i, t \preceq_{rpo} s_i \\
\hline
\therefore t \prec_{rpo} s
\]

\[
\forall i, t_i \prec_{rpo} g(s_1, \cdots, s_m) \quad f <_F g
\]
Recursive Path Ordering
(Dershowitz)

\[ t = f(t_1, \cdots, t_n) \prec_{rpo} g(s_1, \cdots, s_m) = s \]

\( <_F \) is an ordering of \( F \cup C \).

\[ \exists i, t \preceq_{rpo} s_i \]
\[ \frac{}{t \preceq_{rpo} s} \]

\[ \forall i, t_i \preceq_{rpo} g(s_1, \cdots, s_m) \]
\[ f <_F g \]
\[ \frac{}{t \preceq_{rpo} s} \]
Recursive Path Ordering

\[ t = f(t_1, \cdots, t_n) \prec_{rpo} g(s_1, \cdots, s_m) = s \]

\( \prec_{\mathcal{F}} \) is an ordering of \( \mathcal{F} \cup \mathcal{C} \).

\[ \exists i, t \preceq_{rpo} s_i \quad \frac{t \preceq_{rpo} s}{t \prec_{rpo} s} \]

\[ \forall i, t_i \preceq_{rpo} g(s_1, \cdots, s_m) \quad f \prec_{\mathcal{F}} g \quad \frac{t \preceq_{rpo} s}{t \prec_{rpo} s} \]

\[ f \approx_{\mathcal{F}} g \]
Recursive Path Ordering

(Dershowitz)

\[ t = f(t_1, \cdots, t_n) \prec_{rpo} g(s_1, \cdots, s_m) = s \]

\(<_F\) is an ordering of \(\mathcal{F} \cup \mathcal{C}\).

\[ \exists i, t \preceq_{rpo} s_i \quad \frac{t \preceq_{rpo} s}{t <_{rpo} s} \]

\[ \forall i, t_i <_{rpo} g(s_1, \cdots, s_m) \quad f <_F g \quad \frac{t <_{rpo} s}{t <_{rpo} s} \]

\[ \forall i, t_i <_{rpo} s \quad \{t_1, \cdots, t_n\} <_{rpo} \{s_1, \cdots, s_n\} \quad f \approx_{_F} g \]
Recursive Path Ordering

(Dershowitz)

\[ t = f(t_1, \ldots , t_n) \prec_{rpo} g(s_1, \ldots , s_m) = s \]

\(<_\mathcal{F}\) is an ordering of \(\mathcal{F} \cup \mathcal{C}\).

\[ \exists i, t \preceq_{rpo} s_i \quad \frac{t \preceq_{rpo} s}{t \preceq_{rpo} s} \]

\[ \forall i, t_i \preceq_{rpo} g(s_1, \ldots , s_m) \quad f <_\mathcal{F} g \quad t \preceq_{rpo} s \]

\[ \forall i, t_i \preceq_{rpo} s \quad \{ t_1, \ldots , t_n \} \preceq_{rpo} \{ s_1, \ldots , s_n \} \quad f \approx_\mathcal{F} g \quad t \preceq_{rpo} s \]
MPO, LPO, PPO

- MPO: Multiset ordering.
MPO, LPO, PPO

- **MPO**: Multiset ordering.

- **LPO**: Lexicographic ordering.
  - $\forall i < j, t_i \preceq_{lpo} s_i$.
  - $t_j \prec_{lpo} s_j$. 

Part II: Quasi-Interpretations
MPO, LPO, PPO

- **MPO**: Multiset ordering.

- **LPO**: Lexicographic ordering.
  - $\forall i < j, t_i \preceq_{lpo} s_i$.
  - $t_j \prec_{lpo} s_j$.

- **PPO**: Product ordering.
  - $\forall i, t_i \preceq_{ppo} s_{\pi(i)}$.
  - $\exists j, t_j \prec_{ppo} s_{\pi(j)}$. 

Part II: Quasi-Interpretations
Theorem 7 (Hofbauer, 1992, Marion-Moyen, 2000)

\[ PPO \equiv MPO \equiv PRIMREC \]

The set of \textit{functions} computed by MPO/PPO programs is exactly the set of primitive recursive functions.
Extensional characterisations

Theorem 7 (Hofbauer, 1992, Marion-Moyen, 2000)

\[ PPO \equiv MPO \equiv \text{PRIMREC} \]

The set of functions computed by MPO/PPO programs is exactly the set of primitive recursive functions.

Theorem 8 (Weiermann 1995)

\[ LPO \equiv \text{MULTREC} \]
Theorem 7 (Hofbauer, 1992, Marion-Moyen, 2000)

\[ PPO \equiv MPO \equiv \text{PRIMREC} \]

The set of functions computed by MPO/PPO programs is exactly the set of primitive recursive functions.

Theorem 8 (Weiermann 1995) \[ LPO \equiv \text{MULTREC} \]

- Better intensionality than primitive recursion (Colson’s \( C \)).
- No intensional completeness (quicksort).
Theorem 9 (Marion and Moyen, 2000) \( MPO/PPO + QI \equiv \text{PTIME} \)
Theorem 9 (Marion and Moyen, 2000) \( \text{MPO/PPO + QI} \equiv \text{P\hspace{-1em}TIME} \)

Similarities: MPO/PPO implies primitive recursive. Primitive recursive + polynomial bound implies \( \text{P\hspace{-1em}TIME} \) (Cobham).
Theorem 9 (Marion and Moyen, 2000) $MPO/PPO + QI \equiv PTIME$

Similarities: MPO/PPO implies primitive recursive. Primitive recursive + polynomial bound implies PTIME (Cobham).

- Better intensionality than Cobham’s or BC systems.
- No intensional completeness (quicksort).
- Is this an implicit or explicit characterisation?
- QI (over reals) can be semi-automatically constructed (Tarski’s theorem, ICAR and CROCUS softwares).
Other complexity classes

Theorem 10 (Bonfante, Marion and Moyen, 2001)

\[ \text{LPO} + \text{QI} \equiv \text{PSPACE} \]

The space bound can be extracted from the termination proof.

(Amadio and al. 2004).
Other complexity classes

Theorem 10 (Bonfante, Marion and Moyen, 2001)

\[ LPO + QI \equiv \text{PSPACE} \]

The space bound can be extracted from the termination proof.

(Amadio and al. 2004).

Theorem 11 (Bonfante, Marion and Moyen, 2005)

- \( LPO + \text{linearity} + QI \equiv \text{PTIME} \)
- \( LPO + \text{MPO} + QI \equiv \text{PSPACE} \)
- \( \text{MPO} + \text{NonDeterminism} + QI \equiv \text{PSPACE} \)
- \( LPO + QI_{\alpha \neq f} \equiv \text{LINSPACE} \)
Intrinsic Complexity

\[ u = abaaab \quad v = aabb \]

\[ \text{lcs}(\epsilon, y) \rightarrow Z \]
\[ \text{lcs}(x, \epsilon) \rightarrow Z \]
\[ \text{lcs}(i(x), i(y)) \rightarrow S(\text{lcs}(x, y)) \]
\[ \text{lcs}(i(x), j(y)) \rightarrow \text{max}(\text{lcs}(i(x), y), \text{lcs}(x, j(y))) \]
Intrinsic Complexity

\[ u = abaaab \quad v = aabb \]

\[ \text{lcs}(\epsilon, y) \rightarrow \mathbb{Z} \]
\[ \text{lcs}(x, \epsilon) \rightarrow \mathbb{Z} \]
\[ \text{lcs}(i(x), i(y)) \rightarrow S(\text{lcs}(x, y)) \]
\[ \text{lcs}(i(x), j(y)) \rightarrow \max(\text{lcs}(i(x), y), \text{lcs}(x, j(y))) \]

*Extrinsic* complexity : exponential.
**Intrinsic Complexity**

\[ u = abaaab \quad v = aabb \]

\[ \text{lcs}(\epsilon, y) \to Z \]
\[ \text{lcs}(x, \epsilon) \to Z \]
\[ \text{lcs}(i(x), i(y)) \to S(\text{lcs}(x, y)) \]
\[ \text{lcs}(i(x), j(y)) \to \max(\text{lcs}(i(x), y), \text{lcs}(x, j(y))) \]

**Extrinsic complexity**: exponential.
Terminates by PPO, admits a QI.

**Intrinsic complexity**: polynomial (dynamic programming).
Memoisation

Exponential complexity comes from computation done several times.
Memoisation

Exponential complexity comes from computation done several times.

\[ lcs(aa, bb) \]
Memoisation

Exponential complexity comes from computation done several times.

\[ lcs(a, bb) \]

\[ lcs(aa, bb) \]

\[ lcs(aa, b) \]
Memoisation

Exponential complexity comes from computation done several times.
Memoisation

Exponential complexity comes from computation done several times.

\[
lcs(\varepsilon, bb) \\
lcs(a, bb) \\
lcs(a, b) \\
lcs(aa, bb) \\
lcs(aa, b) \\
lcs(a, b) \\
lcs(aa, \varepsilon)
\]
Memoisation

Exponential complexity comes from computation done several times.
Memoisation

Exponential complexity comes from computation done several times. To reach the polynomial bound, one needs to store results of intermediate computations. If the same computation is done another time, it is sufficient to fetch it from the cache.

\[
\begin{align*}
\text{lcs}(\epsilon, bb) & \quad \text{lcs}(a, bb) \\
0 & \quad \text{lcs}(\epsilon, b) \\
0 & 0
\end{align*}
\]

\[
\begin{align*}
\text{lcs}(a, bb) & \quad \text{lcs}(a, b) \\
\text{lcs}(\epsilon, b) & \quad \text{lcs}(a, \epsilon) \\
0 & 0
\end{align*}
\]

\[
\begin{align*}
\text{lcs}(aa, bb) & \quad \text{lcs}(aa, b) \\
\text{lcs}(a, b) & \quad \text{lcs}(a, \epsilon) \\
\text{lcs}(\epsilon, b) & \quad \text{lcs}(a, \epsilon) \\
0 & 0
\end{align*}
\]

\[
\begin{align*}
\text{lcs}(aa, bb) & \quad \text{lcs}(aa, b) \\
\text{lcs}(a, b) & \quad \text{lcs}(aa, \epsilon) \\
\text{lcs}(\epsilon, b) & \quad \text{lcs}(a, \epsilon) \\
0 & 0
\end{align*}
\]
Memoisation

Exponential complexity comes from computation done several times. To reach the polynomial bound, one needs to store results of intermediate computations. If the same computation is done another time, it is sufficient to fetch it from the cache.
Memoisation

Exponential complexity comes from computation done several times. To reach the polynomial bound, one needs to store results of intermediate computations. If the same computation is done another time, it is sufficient to fetch it from the cache.

Cache can be minimised thanks to the information gathered by the termination proof/ordering (Marion, 2000).
An interpreter with cache

\[ \mathcal{E}, \sigma \vdash \langle C, x \rangle \]
An interpreter with cache

\[ \sigma(x) = v \]
\[ E, \sigma \vdash \langle C, x \rangle \]
An interpreter with cache

\[ \sigma(x) = v \]

\[ \mathcal{E}, \sigma \vdash \langle C, x \rangle \rightarrow \langle C, v \rangle \]
An interpreter with cache

\[
\frac{\sigma(x) = v}{\mathcal{E}, \sigma \vdash \langle C, x \rangle \rightarrow \langle C, v \rangle}
\]

\[
\frac{c \in \mathcal{C}}{\mathcal{E}, \sigma \vdash \langle C_0, c(t) \rangle}
\]
An interpreter with cache

$\sigma(x) = v$

$\mathcal{E}, \sigma \vdash \langle C, x \rangle \rightarrow \langle C, v \rangle$

$c \in C$

$\mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle$

$c(t)$
An interpreter with cache

\[
\sigma(x) = v \\
\frac{E, \sigma \vdash \langle C, x \rangle \rightarrow \langle C, v \rangle}{E, \sigma \vdash \langle C, x \rangle \rightarrow \langle C, v \rangle}
\]

\[
\begin{array}{c}
c \in C \\
E, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle
\end{array}
\]

\[
E, \sigma \vdash \langle C_0, c(\vec{t}) \rangle \rightarrow \langle C_n, c(\vec{v}) \rangle
\]
An interpreter with cache

\[
\begin{align*}
\sigma(x) &= v \\
& \quad \Rightarrow \quad \mathcal{E}, \sigma \vdash \langle C, x \rangle \rightarrow \langle C, v \rangle \\
\mathcal{C} \in \mathcal{C} & \quad \Rightarrow \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle \\
& \quad \Rightarrow \quad \mathcal{E}, \sigma \vdash \langle C_0, \mathcal{C}(\vec{t}) \rangle \rightarrow \langle C_n, \mathcal{C}(\vec{v}) \rangle \\
\mathcal{F} \in \mathcal{F} & \quad \Rightarrow \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle \\
& \quad \Rightarrow \quad \mathcal{E}, \sigma \vdash \langle C_0, \mathcal{F}(\vec{t}) \rangle
\end{align*}
\]
An interpreter with cache

\[ \sigma(x) = v \]
\[ \mathcal{E}, \sigma \vdash \langle C, x \rangle \rightarrow \langle C, v \rangle \]
\[ \mathcal{C} \in \mathcal{C} \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle \]
\[ \mathcal{E}, \sigma \vdash \langle C_0, \mathcal{C}(\vec{t}) \rangle \rightarrow \langle C_n, \mathcal{C}(\vec{v}) \rangle \]

\[ f \in \mathcal{F} \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle \quad (f(\vec{v}), v) \in C_n \]
\[ \mathcal{E}, \sigma \vdash \langle C_0, f(\vec{t}) \rangle \]
An interpreter with cache

\[ \sigma(x) = v \]
\[ \mathcal{E}, \sigma \vdash \langle C, x \rangle \rightarrow \langle C, v \rangle \]

\[ \mathcal{C} \in \mathcal{C} \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle \]

\[ \mathcal{E}, \sigma \vdash \langle C_0, \mathcal{C}(\vec{t}) \rangle \rightarrow \langle C_n, \mathcal{C}(\vec{v}) \rangle \]

\[ f \in \mathcal{F} \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle \quad (f(\vec{v}), v) \in C_n \]

\[ \mathcal{E}, \sigma \vdash \langle C_0, f(\vec{t}) \rangle \rightarrow \langle C_n, v \rangle \]
An interpreter with cache

\[ \sigma(x) = v \]

\[
\begin{align*}
\mathcal{E}, \sigma & \vdash \langle C, x \rangle \rightarrow \langle C, v \rangle \\
\mathcal{C} \in \mathcal{C} & \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle \\
\mathcal{E} \in \mathcal{E} & \quad \mathcal{C}(\vec{t}) \rightarrow \langle C_n, \mathcal{C}(\vec{v}) \rangle \\
\mathcal{F} \in \mathcal{F} & \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle \\
\mathcal{E} \in \mathcal{E} & \quad \mathcal{F}(\vec{t}) \rightarrow \langle C_n, v \rangle \\
\mathcal{F} \in \mathcal{F} & \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle \\
\mathcal{E} \in \mathcal{E} & \quad \mathcal{F}(\vec{t}) \rightarrow \langle C_n, \mathcal{F}(\vec{t}) \rangle
\end{align*}
\]
An interpreter with cache

\[
\sigma(x) = v \quad \frac{c \in C \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle}{\mathcal{E}, \sigma \vdash \langle C, x \rangle \rightarrow \langle C, v \rangle}
\]

\[
\mathcal{E}, \sigma \vdash \langle C_0, c(\vec{t}) \rangle \rightarrow \langle C_n, c(\vec{v}) \rangle
\]

\[
f \in \mathcal{F} \quad \frac{\mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle}{\mathcal{E}, \sigma \vdash \langle C_0, f(\vec{t}) \rangle \rightarrow \langle C_n, v \rangle}
\]

\[
f \in \mathcal{F} \quad \frac{\mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle}{f(\vec{p}) \rightarrow r \in \mathcal{E} \quad p_i\sigma' = v_i}
\]

\[
\mathcal{E}, \sigma \vdash \langle C_0, f(\vec{t}) \rangle
\]
An interpreter with cache

\[ \sigma(x) = v \]
\[ \mathcal{E}, \sigma \vdash \langle C, x \rangle \rightarrow \langle C, v \rangle \]
\[ \mathcal{E}, \sigma \vdash \langle C, \sigma \rangle \rightarrow \langle C, v \rangle \]
\[ \mathcal{C} \in \mathcal{C} \]
\[ \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle \]
\[ \mathcal{E}, \sigma \vdash \langle C_0, \mathcal{C}(\vec{t}) \rangle \rightarrow \langle C_n, \mathcal{C}(\vec{v}) \rangle \]

\[ f \in \mathcal{F} \]
\[ \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle \]
\[ \mathcal{E}, \sigma \vdash \langle C_0, f(\vec{t}) \rangle \rightarrow \langle C_n, v \rangle \]

\[ f \in \mathcal{F} \]
\[ \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle \]
\[ f(\vec{p}) \rightarrow r \in \mathcal{E} \]
\[ p_i \sigma' = v_i \]
\[ \mathcal{E}, \sigma' \vdash \langle C_n, r \rangle \rightarrow \langle C, v \rangle \]
\[ \mathcal{E}, \sigma \vdash \langle C_0, f(\vec{t}) \rangle \]
An interpreter with cache

\[ \sigma(x) = v \]
\[ \mathcal{E}, \sigma \vdash \langle C, x \rangle \rightarrow \langle C, v \rangle \]

\[ \mathcal{C} \in \mathcal{C}, \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle \]

\[ \mathcal{E}, \sigma \vdash \langle C_0, \mathcal{C}(\vec{t}) \rangle \rightarrow \langle C_n, \mathcal{C}(\vec{v}) \rangle \]

\[ \mathcal{F} \in \mathcal{F}, \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle \quad (\mathcal{F}(\vec{v}), v) \in C_n \]

\[ \mathcal{E}, \sigma \vdash \langle C_0, \mathcal{F}(\vec{t}) \rangle \rightarrow \langle C_n, v \rangle \]

\[ \mathcal{F} \in \mathcal{F}, \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i \rangle \rightarrow \langle C_i, v_i \rangle \]

\[ \mathcal{F}(\vec{p}) \rightarrow r \in \mathcal{E}, \quad p_i\sigma' = v_i \]

\[ \mathcal{E}, \sigma' \vdash \langle C_n, r \rangle \rightarrow \langle C, v \rangle \]

\[ \mathcal{E}, \sigma \vdash \langle C_0, \mathcal{F}(\vec{t}) \rangle \rightarrow \langle C \cup \mathcal{F}(\vec{v}), v \rangle \]
Any BC system terminates by PPO and admits a QI. For example: \( f(a, \overline{z}, \overline{x}; \overline{y}) \rightarrow g(a, \overline{z}, \overline{x}; f(\overline{z}, \overline{x}; \overline{y}), \overline{y}) \).
Any BC system terminates by PPO and admits a QI. For example: $f(a.z, x; y) \rightarrow g(a, z, x; f(z, x; y), y)$.

- It terminates by PPO with $g \prec f$ because $z \prec_{ppo} a.z$. 
Any BC system terminates by PPO and admits a QI. For example: \( f(a, z, x; y) \rightarrow g(a, z, x; f(z, x; y), y) \).

- It terminates by PPO with \( g \prec f \) because \( z \prec_{ppo} a.z \).

- It admits a QI of the shape:

\[
(h)(X_1, \ldots, X_n; Y_1, \ldots, Y_m) = P_h(X_1, \ldots, X_n) + \max(Y_1, \ldots, Y_m)
\]

Here \( P_f \) is the polynomial \( P_f(X, Y) = X \cdot P_g(X, Y) \)

Part II: Quasi-Interpretations
Any BC system terminates by PPO and admits a QI. For example: $f(a., z, x; y) \rightarrow g(a, z, x; f(z, x; y), y)$.

- It terminates by PPO with $g <_{\mathcal{F}} f$ because $z <_{\text{ppo}} a.z$.
- It admits a QI of the shape:

$$\langle h \rangle (X_1, \cdots, X_n; Y_1, \cdots, Y_m) = P_h(X_1, \cdots, X_n) + \max(Y_1, \cdots, Y_m)$$

Here $P_f$ is the polynomial $P_f(X, Y) = X \cdot P_g(X, Y)$.

$$\langle f(a., z, x; y) \rangle \geq \langle g(a, z, x; f(z, x; y), y) \rangle$$
Any BC system terminates by PPO and admits a QI. For example:
\[ f(a, \overline{z}, x; y) \rightarrow g(a, \overline{z}, x; f(z, x; y), y). \]

- It terminates by PPO with \( g <_\mathcal{F} f \) because \( \overline{z} <_{ppo} a.\overline{z} \).
- It admits a QI of the shape:
\[
(h)(X_1, \cdots, X_n; Y_1, \cdots, Y_m) = P_h(X_1, \cdots, X_n) + \max(Y_1, \cdots, Y_m)
\]

Here \( P_f \) is the polynomial
\[
P_f(X, Y) = X \cdot P_g(X, Y)
\]

\[
(f)(Z + 1, X; Y) \geq (g(a, \overline{z}, x; f(z, x; y), y))
\]

\[
(f)(Z + 1, X; Y) \geq
\]
Bellantoni-Cook

Any BC system terminates by PPO and admits a QI. For example: \( f(a.\overline{z}, \overline{x}; \overline{y}) \rightarrow g(a, \overline{z}, \overline{x}; f(\overline{z}, \overline{x}; \overline{y}), \overline{y}) \).

- It terminates by PPO with \( g \prec_f f \) because \( \overline{z} \prec_{ppo} a.\overline{z} \).
- It admits a QI of the shape:

\[
(h)(X_1, \cdots, X_n; Y_1, \cdots, Y_m) = P_h(X_1, \cdots, X_n) + \max(Y_1, \cdots, Y_m)
\]

Here \( P_f \) is the polynomial \( P_f(X, Y) = X \cdot P_g(X, Y) \)

\[
(f)(a.\overline{z}, \overline{x}; \overline{y}) \geq (g(a, \overline{z}, \overline{x}; f(\overline{z}, \overline{x}; \overline{y}), \overline{y}))
\]

\[
(f)(Z + 1, X; Y) \geq (g)(Z, X; (f)(Z, X; Y), Y)
\]
Any BC system terminates by PPO and admits a QI. For example: \( f(a, z, x; y) \rightarrow g(a, z, x; f(z, x; y), \overline{y}) \).

- It terminates by PPO with \( g \prec f \) because \( z \prec_{ppo} a.\overline{z} \).
- It admits a QI of the shape:

\[
(h)(X_1, \cdots, X_n; Y_1, \cdots, Y_m) = P_h(X_1, \cdots, X_n) + \max(Y_1, \cdots, Y_m)
\]

Here \( P_f \) is the polynomial \( P_f(X, Y) = X \cdot P_g(X, Y) \)

\[
(f)(Z + 1, X; Y) \geq (g)(Z, X; (f)(Z, X; Y), Y)
\]

\[
P_f(Z + 1, X) + Y \geq P_g(Z, X) + \max(P_f(Z, X) + Y, Y)
\]
Any BC system terminates by PPO and admits a QI. For example: \( f(a.z, x; y) \rightarrow g(a, z, x; f(z, x; y), y) \).

- It terminates by PPO with \( g <_F f \) because \( z <_{ppo} a.z \).

- It admits a QI of the shape:

\[
(h)(X_1, \cdots, X_n; Y_1, \cdots, Y_m) = P_n(X_1, \cdots, X_n) + \max(Y_1, \cdots, Y_m)
\]

Here \( P_f \) is the polynomial \( P_f(X, Y) = X \cdot P_g(X, Y) \)

\[
(f(a.z, x; y)) \geq (g(a, z, x; f(z, x; y), y))
\]

\[
(f)(Z + 1, X; Y) \geq (g)(Z, X; (f)(Z, X; Y), Y)
\]

\[
P_f(Z + 1, X) + Y \geq P_g(Z, X) + \max(P_f(Z, X) + Y, Y)
\]

\[
P_f(Z + 1, X) + Y \geq P_g(Z, X) + P_f(Z, X) + Y
\]
Any BC system terminates by PPO and admits a QI. For example: $f(a,z,x;y) \rightarrow g(a,z,x;f(z,x;y),y)$.

- It terminates by PPO with $g \prec_f f$ because $z \prec_{ppo} a.z$.
- It admits a QI of the shape:

  $$(\langle h \rangle)(X_1, \cdots, X_n; Y_1, \cdots, Y_m) = P_h(X_1, \cdots, X_n) + \max(Y_1, \cdots, Y_m)$$

Here $P_f$ is the polynomial $P_f(X,Y) = X \cdot P_g(X,Y)$

$$(\langle f \rangle(a,z,x;y)) \geq (\langle g \rangle(a,z,x;f(z,x;y),y))$$

$$(\langle f \rangle)(Z + 1, X; Y) \geq (\langle g \rangle)(Z, X; (\langle f \rangle)(Z, X; Y), Y)$$

$$P_f(Z + 1, X) + Y \geq P_g(Z, X) + \max(P_f(Z, X) + Y, Y)$$

$$P_f(Z + 1, X) + Y \geq P_g(Z, X) + P_f(Z, X) + Y$$

$$(Z + 1) \cdot P_g(Z + 1, X) \geq P_g(Z, X) + Z \cdot P_g(Z, X)$$
Conclusion of Part II

- Separating termination and bounds gives good results.
- For more intensional power:
  - Handling functions with results smaller than the inputs: Marion and Péchoux (sup-interpretations).
  - Handling non monotonic interpretations: Shkaravska, van Eekelen, and van Kesteren.
- Similar works on imperative programs:
  - Niggl and Wunderlich: \( \sim \) polynomial interpretations.
  - \( mwP \)-polynomials (Kristiansen and Jones): \( \sim \) QI.
  - “Set of polynomials”/“Set of paths” (Ben-Amram and Kristiansen): intensionally complete on a restricted non-deterministic language.
Part III: Linear logic and typed $\lambda$-calculus
Our reference language here is λ-calculus
untyped λ-calculus is Turing-complete

Type systems can guarantee termination
ex: system F (polymorphic types)

Curry-Howard correspondence

\text{proof} = \text{type derivation}

\text{normalization (cut elimination)} = \text{execution}

\text{intuitionistic logic} \leftrightarrow \text{system F}

Some characteristics of λ-calculus:
higher-order types
no distinction between data / program
Linear logic (LL): fine-grained decomposition of intuitionistic logic duplication is controlled with a specific connective '!' (exponential)

Variants of linear logic with different rules for ! have bounded complexity: light logics, i.e. proof normalization can be performed with a bounded complexity (e.g. PTIME)

These logics (or subsystems) can then be used as type systems for λ-calculus, in order to guarantee complexity bounds
Plan of Part III

- Background on \( \lambda \)-calculus and system F
- Light linear logic and the type system DLAL
- Relating the Bellantoni-Cook algebra and Light linear logic
- Conclusion
\[ t, u ::= x \mid \lambda x.t \mid (t u) \]

notations: \( \lambda x_1 x_2.t \) for \( \lambda x_1.\lambda x_2.t \)

\( (t u v) \) for \( ((t u) v) \)

substitution: \( t[u/x] \)

\[ \beta \text{-reduction:} \]

\( \frac{}{\rightarrow} \) relation obtained by context-closure of:

\( (((\lambda x.t)u) \frac{}{\rightarrow} t[u/x] \)

\( \rightarrow \) reflexive and transitive closure of \( \frac{}{\rightarrow} \).
Typed Lambda-terms

System F types:

\[ T, U ::= \alpha \mid T \rightarrow U \mid \forall \alpha . T \]

Simple types: without \( \forall \)

Simply typed terms, \( \text{\`a la Church} \) :

\[ x^T \quad (\lambda x^T . M^U)^{T \rightarrow U} \quad ((M^{T \rightarrow U} N^T)^T)^U \]
System F typing rules (à la Curry)

\[
\frac{x : A \vdash x : A}{\text{Id}}
\]

\[
\frac{\Gamma_1, x : A \vdash t : B}{\Gamma_1 \vdash \lambda x.t : A \to B}
\]

(\rightarrow i)

\[
\frac{\Gamma_1 \vdash t : B}{\Gamma_1, x : A \vdash t : B}
\]

(Weak)

\[
\frac{\Gamma_1 \vdash t : A}{\Gamma_1 \vdash t : \forall \alpha. A}
\]

(\forall i) (*)

\[
\frac{\Gamma_1 \vdash t : A \to B \quad \Gamma_2 \vdash u : A}{\Gamma_1, \Gamma_2 \vdash (t \ u) : B}
\]

(\rightarrow e)

\[
\frac{\Gamma_1 \vdash t : A \to B}{\Gamma_1, \Gamma_2 \vdash (t \ u) : B}
\]

(Cntr)

\[
\frac{x_1 : A, x_2 : A, \Gamma_1 \vdash t : B}{x : A, \Gamma_1 \vdash t[x/x_1, x/x_2] : B}
\]

\[
\frac{\Gamma_1 \vdash t : \forall \alpha. A}{\Gamma_1 \vdash t : A[B/\alpha]}
\]

(\forall e)

\[(*) \alpha \not\in \Gamma_1.\]
Proofs-programs correspondence (Curry-Howard)

| typed term | ⇒ | 2nd-order intuitionistic logic proof |
| type | | formula |
| $M^B$, with | proof of $A_1, \ldots, A_n \vdash B$ |
| free variables $x_i : A_i$, $1 \leq i \leq n$ | | |

$\beta$-reduction of term

normalization of proof

(cut elimination)
Examples of F types

Polymorphic identity:
\[ \lambda x^\alpha . x : \forall \alpha . (\alpha \to \alpha) \]

Tally integers:
\[ N = \forall \alpha . (\alpha \to \alpha) \to (\alpha \to \alpha) \]

example
\[ 2 = \lambda f^\alpha \to \alpha . \lambda x^\alpha . (f (f x))^{\alpha} : N \]

Binary words:
\[ W = \forall \alpha . (\alpha \to \alpha) \to (\alpha \to \alpha) \to (\alpha \to \alpha) \]

example
\[ 110 = \lambda s_0^\alpha \to \alpha . \lambda s_1^\alpha \to \alpha . \lambda x^\alpha . (s_1 (s_1 (s_0 x)))^{\alpha} : W \]
For each inductive data type, there is an associated iteration principle. For instance, for $N = \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$, we can define for any $A$ an iterator $\text{iter}_A$:

$$\text{iter}_A = \lambda f \, x. \, (n \, f \, x) : (A \rightarrow A) \rightarrow A \rightarrow N \rightarrow A$$

then $(\text{iter}_A \, F \, t \, n) \rightarrow (F \, (F \ldots (F \, t) \ldots) \ldots)$  ($n$ times)

example:

$double : N \rightarrow N$

$exp = \lambda n. (\text{iter}_N \, double \, 1 \, n) : N \rightarrow N$

Iteration on binary lists:

$$\text{iter}^W_A : (A \rightarrow A) \rightarrow (A \rightarrow A) \rightarrow A \rightarrow W \rightarrow A$$
Examples of terms

concatenation

\[
\text{append} = \lambda u^W . \lambda v^W . \lambda s_0 . \lambda s_1 . \lambda x . (u \ s_0 \ s_1 \ (v \ s_0 \ s_1 \ x))
\]

: \quad W \to W \to W

length

\[
\text{length} = \lambda u^W . \lambda f^{\alpha \to \alpha} . (u \ f \ f)^{\alpha \to \alpha}
\]

: \quad W \to N

multiplication

\[
\text{mult} = \lambda n^N . \lambda m^N . (m \ \lambda k . \lambda f . \lambda x . (n \ f \ (k \ f \ x))) \ 0
\]

: \quad N \to N \to N
Theorem 12 (Girard 1971)  \textit{If a term is well typed in } F, \textit{ then it is strongly normalizable.}

Thus a type derivation can be seen as a termination certificate. In particular, a term $t : W \rightarrow W$ represents a function on words which terminates on all inputs.

Can we refine this system in order to guarantee \textit{feasible} termination, that is to say in polynomial time?
How can exponential blow up arise? (Terui)

2 easy ways to cause exponential blow-up:

- basic functions: $0 : \mathbb{N}, \quad s : \mathbb{N} \to \mathbb{N}, \quad + : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$.

- exponential blow-up can be caused by 2 schemes:
  1. iteration-iteration

    $$
    \begin{align*}
    \text{dbl}(0) &= 0 & \text{exp}(0) &= 1 \\
    \text{dbl}(s(x)) &= s(s(\text{dbl}(x))) & \text{exp}(s(x)) &= \text{dbl}(\text{exp}(x))
    \end{align*}
    $$

  2. contraction-iteration

    $$
    \begin{align*}
    \text{dbl}(x) &= x + x & \text{exp}(0) &= 1 \\
    \text{exp}(s(x)) &= \text{dbl}(\text{exp}(x))
    \end{align*}
    $$

- to keep contraction and iteration, we need to forbid bad combinations of these.
Linear logic

Linear logic (LL) arises from the decomposition

\[ A \Rightarrow B = !A \rightarrow B \]

The `!` modality (exponential) accounts for duplication (contraction)

(Intuitionistic) Linear logic formulas:

\[ A, B ::= \alpha | A \rightarrow B | A \otimes B | !A | \forall \alpha.A \]

Resource-based intuition:

\( (1 \text{ euro}) \rightarrow (1 \text{ coffee}) \)
\( (1 \text{ euro}) \rightarrow (1 \text{ croissant}) \)

but \( (1 \text{ euro}) \rightarrow (1 \text{ coffee}) \otimes (1 \text{ croissant}) \) doesn’t hold.
Linear logic: exponential principles

Principles:

- **contraction**
  \[ !A \multimap \bot \Rightarrow !A \otimes !A \]

- **functoriality of \( ! \)**
  \[
  \frac{A \vdash B}{!A \vdash !B}
  \]

- **monoidalness**
  \[ !A \otimes !B \multimap !\left( A \otimes B \right) \]

- **dereliction**
  \[ !A \multimap A \]

- **digging**
  \[ !A \multimap \,!A \]

By removing **dereliction** and **digging** one defines ELL, corresponding to elementary complexity (bound \( 2^{2^{\cdots 2^n}} \) of fixed height).
Light linear logic, LLL (Girard 98)

\[ \frac{A \vdash B}{\vdash A \otimes B} \]

new modality \( \& \), with: \( !A \rightarrow \&A \)

\( \& \) is a functor and \( \&A \otimes \&B \rightarrow \&(A \otimes B) \)

\( \rightarrow \) manages to avoid both exponentiation schemes

Light affine logic (LAL) is the variant with full weakening.

Proofs can be represented as proof-nets (graphs). Normalization of proof-nets corresponds to program execution.

**Theorem 13 (PTIME Soundness. Girard)** Light linear logic proof-nets admit a polynomial time normalization (at fixed depth).

**Theorem 14 (Completeness. Girard/Asperti-Roversi)** All PTIME functions can be represented in LLL (or LAL).
Can we use LLL or LAL directly as type systems for $\lambda$-calculus? Goal: if a program is well-typed, then it is PTIME

However there are two pitfalls:

- they do not give subject-reduction,
- no polynomial bound on the number of $\beta$-reduction steps for typed terms (even if there is one on proof-net normalization).
essentially one drops types \( A \to !B, \)
\( \$ A \)
To overcome the problems with typing in LAL: we can restrict in Light affine logic the use of ! to !A \rightarrow B, denoted $A \Rightarrow B$.

The DLAL (Dual Light Affine Logic) type system [BT04]:

$$A, B ::= \alpha \mid A \rightarrow B \mid A \Rightarrow B \mid \sharp A \mid \forall \alpha. A$$

typing judgements of the form: $\Gamma; \Delta \vdash t : A$, where

- $\Gamma$ contains duplicable variables,
- $\Delta$ contains linear variables.
DLAL typing rules

\[
\frac{x : A \vdash x : A}{\Gamma; \Delta \vdash x : A} \quad \text{(Id)}
\]

\[
\frac{\Gamma_1; \Delta_1, x : A \vdash t : B}{\Gamma_1; \Delta_1 \vdash \lambda x.t : A \to B} \quad \text{($\to$ i)}
\]

\[
\frac{\Gamma_1, x : A; \Delta_1 \vdash t : B}{\Gamma_1; \Delta_1 \vdash \lambda x.t : A \Rightarrow B} \quad \text{($\Rightarrow$ i)}
\]

\[
\frac{\Gamma_1; \Delta_1 \vdash t : A}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash t : A} \quad \text{(Weak)}
\]

\[
\frac{\Gamma; x_1 : B_1, \ldots, x_n : B_n \vdash t : A}{\Gamma; x_1 : \#B_1, \ldots, x_n : \#B_n \vdash t : \#A} \quad \text{($\#$ i)}
\]

\[
\frac{\Gamma_1; \Delta_1 \vdash u : \#A \quad \Gamma_2; x : \#A, \Delta_2 \vdash t : B}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash t[u/x] : B} \quad \text{($\#$ e)}
\]

\[
\frac{\Gamma_1; \Delta_1 \vdash t : A \to B \quad \Gamma_2; \Delta_2 \vdash u : A}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash (t\ u) : B} \quad \text{($\to$ e)}
\]

\[
\frac{\Gamma_1; \Delta_1 \vdash t : A \Rightarrow B \quad z : C \vdash u : A}{\Gamma_1, z : C; \Delta_1 \vdash (t\ u) : B} \quad \text{($\Rightarrow$ e)}
\]

\[
\frac{x_1 : A, x_2 : A, \Gamma_1; \Delta_1 \vdash t : B}{x : A, \Gamma_1; \Delta_1 \vdash t[x/x_1, x/x_2] : B} \quad \text{(Cntr)}
\]
### DLAL typing rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>; x : A ⊢ x : A</td>
<td>(Id)</td>
</tr>
<tr>
<td>Γ₁; Δ₁, x : A ⊢ t : B</td>
<td>Γ₁; Δ₁ ⊢ λx.t : A → B</td>
</tr>
<tr>
<td>Γ₁, x : A; Δ₁ ⊢ t : B</td>
<td>Γ₁, Δ₁ ⊢ λx.t : A ⇒ B</td>
</tr>
<tr>
<td>Γ₁; Δ₁ ⊢ t : A</td>
<td>Γ₁; Δ₁ ⊢ t : A ⊸ B, Δ₂ ⊢ u : A</td>
</tr>
<tr>
<td>; Γ, x₁ : B₁, ..., xₙ : Bₙ ⊢ t : A</td>
<td>Γ; x₁ : B₁, ..., xₙ : Bₙ ⊢ t : A</td>
</tr>
<tr>
<td>Γ₁; Δ₁ ⊢ u : §A</td>
<td>Γ₂; x : §A, Δ₂ ⊢ t : B</td>
</tr>
</tbody>
</table>

#### Weak typing rule

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Γ₁, Δ₁ ⊢ t : A, Δ₂ ⊢ (t u) : B</td>
<td>Γ₁; Δ₁ ⊢ t : A ⇒ B</td>
</tr>
<tr>
<td>x₁ : A, x₂ : A, Δ₁ ⊢ t : B</td>
<td>x : A, Γ₁; Δ₁ ⊢ t[x/x₁, x/x₂] : B</td>
</tr>
</tbody>
</table>

**We have:**

*If* Γ; Δ ⊢_{DLAL} t : A *and* x ∈ Δ *then* x *has at most one occurrence in* t.
DLAL typing rules

\begin{align*}
\Gamma_1 ; \Delta_1, x : A \vdash x : A & \quad \text{(Id)} \\
\Gamma_1 ; \Delta_1, x : A ; t : B \vdash \lambda x.t : A \to B & \quad \text{(-o i)} \\
\Gamma_1, x : A ; \Delta_1 \vdash t : B \quad \Gamma_1 ; \Delta_1 \vdash \lambda x.t : A \Rightarrow B & \quad \text{(⇒ i)} \\
\Gamma_1, \Delta_1 \vdash t : A & \quad \text{(Weak)} \\
\Gamma_1 ; \Gamma_2, \Delta_1, \Delta_2 \vdash t : A & \quad \text{(Weak)} \\
\Gamma_1, \Delta_1 \vdash t : A & \quad \text{(⇒ e)} \\
\Gamma_1, \Delta_1 \vdash t : A \Rightarrow B ; z : C \vdash u : A \Rightarrow B & \quad \text{(⇒ e)} \\
\Gamma_1, \Delta_1 \vdash t : A \Rightarrow B & \quad \text{(⇒ e)} \\
\Gamma_1, \Delta_1 \vdash t[u/x] : B & \quad \text{(⇒ e)} \\
\Gamma_1, x_1 : B_1, \ldots, x_n : B_n \vdash t : A & \quad \text{(§ i)} \\
\Gamma ; x_1 : \text{§} B_1, \ldots, x_n : \text{§} B_n \vdash t : \text{§} A & \quad \text{(§ e)} \\
\Gamma_1, \Delta_1 \vdash u : \text{§} A \quad \Gamma_2, x : \text{§} A, \Delta_2 \vdash t : B & \quad \text{(§ e)} \\
\Gamma_1, \Gamma_2, \Delta_1, \Delta_2 \vdash t[u/x] : B & \quad \text{(§ e)}
\end{align*}

(-o i) (resp. (⇒ i)) corresponds to abstraction on a linear (resp. non-linear) variable,
an argument $u$ of a term $t$ of type $A \Rightarrow B$ must have at most one free variable $z$, which is moreover linear.

$\rightarrow$ prevents 2nd exponentiation scheme (contraction-iteration).
DLAL typing rules

\[
\frac{\Gamma; x : A \vdash x : A}{\text{(Id)}} \quad \frac{\Gamma_1; \Delta_1, x : A \vdash t : B}{\Gamma_1; \Delta_1 \vdash \lambda x.t : A \to B} \quad \frac{\Gamma_1; \Delta_1 \vdash \lambda x.t : A \Rightarrow B}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash t : A} \quad \frac{\Gamma_1; \Delta_1 \vdash t : A \to t : A \to B \quad \Gamma_2; \Delta_2 \vdash u : A}{\Gamma_1, \Gamma_2, \Delta_1, \Delta_2 \vdash (t u) : B} \quad \frac{\Gamma_1; \Delta_1 \vdash t : A \Rightarrow B \quad \Gamma_1, \Delta_1 \vdash z : C \vdash u : A}{\Gamma_1, \Gamma_2, \Delta_1, \Delta_2 \vdash (t u) : B} \quad \frac{x_1 : A, x_2 : A, \Gamma_1; \Delta_1 \vdash t : B}{x : A, \Gamma_1; \Delta_1 \vdash t[x/x_1, x/x_2] : B} \quad \frac{\Gamma_1; \Delta_1 \vdash u : \pre B \quad \Gamma_2; x : \pre B, \Delta_2 \vdash t : B}{\Gamma_1, \Gamma_2, \Delta_1, \Delta_2 \vdash t[u/x] : B}
\]

the rule (§ i) allows to turn linear variables (in \( \Gamma \)) into non-linear ones.

\( \rightarrow \text{prevents 1st exponentiation scheme (iteration-iteration).} \)
DLAL typing rules

\[ \vdash x : A \quad \text{(Id)} \]
\[ \Gamma_1; \Delta_1, x : A \vdash t : B \quad (\rightarrow i) \]
\[ \Gamma_1; \Delta_1 \vdash \lambda x.t : A \rightarrow B \quad (\rightarrow e) \]
\[ \Gamma_1, x : A; \Delta_1 \vdash t : B \quad (\Rightarrow i) \]
\[ \Gamma_1; \Delta_1 \vdash \lambda x.t : A \Rightarrow B \quad (\Rightarrow e) \]
\[ \Gamma_1; \Delta_1 \vdash t : A \quad (\text{Weak}) \]
\[ \Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash t : A \quad (\text{Cntr}) \]
\[ \vdash ; \Gamma, x_1 : B_1, \ldots, x_n : B_n \vdash t : A \quad (§ i) \]
\[ \Gamma ; x_1 : §B_1, \ldots, x_n : §B_n \vdash t : §A \quad (§ e) \]

Part III: Linear Logic ad typed \(\lambda\)-calculus
DLAL typing rules

\[
\frac{x : A ⊢ x : A}{x : A ⊢ x : A} \quad \text{(Id)}
\]

\[
\frac{\Gamma_1; \Delta_1, x : A ⊢ t : B}{\Gamma_1; \Delta_1 ⊢ \lambda x.t : A \to B} \quad \text{(→ i)}
\]

\[
\frac{\Gamma_1, x : A; \Delta_1 ⊢ t : B}{\Gamma_1; \Delta_1 ⊢ \lambda x.t : A \Rightarrow B} \quad \text{(⇒ i)}
\]

\[
\frac{\Gamma_1; \Delta_1 ⊢ t : A}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 ⊢ t : A} \quad \text{(Weak)}
\]

\[
\frac{\Gamma, x_1 : B_1, \ldots, x_n : B_n ⊢ t : A}{\Gamma; x_1 : §B_1, \ldots, x_n : §B_n ⊢ t : §A} \quad \text{(§ i)}
\]

\[
\frac{\Gamma_1; \Delta_1 ⊢ u : §A \quad \Gamma_2; x : §A, \Delta_2 ⊢ t : B}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 ⊢ t[u/x] : B} \quad \text{(§ e)}
\]

\[
\frac{\Gamma_1; \Delta_1 ⊢ t : A \to B \quad \Gamma_2; \Delta_2 ⊢ u : A}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 ⊢ (t \ u) : B} \quad \text{(→ e)}
\]

\[
\frac{\Gamma_1; \Delta_1 ⊢ t : A \Rightarrow B \quad z : C ⊢ u : A}{\Gamma_1, z : C; \Delta_1 ⊢ (t \ u) : B} \quad \text{(⇒ e)}
\]

\[
\frac{x_1 : A, x_2 : A, \Gamma_1; \Delta_1 ⊢ t : B}{x : A, \Gamma_1; \Delta_1 ⊢ t[x/x_1, x/x_2] : B} \quad \text{(Cntr)}
\]

**Depth** \(d\) of a derivation \(\mathcal{D}\): maximal number of \((§ i)\) and r.h.s. premises of \((⇒ e)\) in a branch of \(\mathcal{D}\).
DLAL typing rules

\begin{align*}
\frac{x : A \vdash x : A}{\Gamma_1; \Delta_1, x : A \vdash t : B} & \quad (\text{Id}) \\
\frac{\Gamma_1; \Delta_1 \vdash \lambda x. t : A \to B}{\Gamma_1; \Delta_1, x : A \vdash t : B} & \quad (\to \text{ i}) \\
\frac{\Gamma_1, x : A; \Delta_1 \vdash t : B}{\Gamma_1; \Delta_1 \vdash \lambda x. t : A \Rightarrow B} & \quad (\Rightarrow \text{ i}) \\
\frac{\Gamma_1; \Delta_1 \vdash t : A}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash t : A} & \quad (\text{Weak}) \\
\frac{\Gamma, x_1 : B_1, \ldots, x_n : B_n \vdash t : A}{\Gamma; x_1 : \mathbb{B} B_1, \ldots, x_n : \mathbb{B} B_n \vdash t : \mathbb{B} A} & \quad (\mathbb{B} \text{ i}) \\
\frac{\Gamma_1; \Delta_1 \vdash t : A}{\Gamma_1; \Delta_1 \vdash t : \forall \alpha. A} & \quad (\forall \text{ i}) \\
\frac{\Gamma_1; \Delta_1 \vdash t : \forall \alpha. A}{\Gamma_1; \Delta_1 \vdash t : A[B/\alpha]} & \quad (\forall \text{ e})
\end{align*}

\(\forall \alpha \notin \Gamma_1, \Delta_1.\)
Example of type derivation in DLAL

\[ M = (\lambda f. (f \ (f \ x))) ((\lambda h. h) \ g) \]

\[
\vdots \\
; f_2 : \alpha \to \alpha, x : \alpha \vdash (f_2 \ x) : \alpha \\
; f_1 : \alpha \to \alpha \vdash f_1 : \alpha \to \alpha
\]

(§ i)

\[
; f_1 : \beta, f_2 : \beta, x : \alpha \vdash (f_1 \ (f_2 \ x)) : \alpha \\
; h : \beta \vdash h : \beta \\
; \vdash \lambda h. h : \beta \to \beta \\
; g : \beta \vdash g : \beta
\]

(⇒ e)

\[
; x : \xi \alpha \vdash \lambda f. (f \ (f \ x)) : \beta \Rightarrow \xi \alpha \\
g : \beta \\
; g : \beta \vdash (\lambda h. h) \ g : \beta \\
\]

where \( \beta = \alpha \to \alpha \).
DLAL and system F

- Forgetful map \((.)^{-}\) from DLAL to F:
  - remove occurrences of \(\$\),
  - replace \(\Rightarrow\) and \(\Rightarrow\) with \(\rightarrow\).

\((.)^{-}\) gives a map from DLAL derivations to system F derivations.

- Data types in DLAL:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>DLAL</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Church integers</td>
<td>(\forall \alpha . (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha))</td>
<td>(\forall \alpha . (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha))</td>
</tr>
<tr>
<td>Binary lists</td>
<td>(\forall \alpha . (\alpha \rightarrow \alpha) \Rightarrow $ (\alpha \rightarrow \alpha))</td>
<td>(\forall \alpha . (\alpha \rightarrow \alpha) \Rightarrow (\alpha \rightarrow \alpha) \Rightarrow $ (\alpha \rightarrow \alpha))</td>
</tr>
</tbody>
</table>

Example for binary lists: \(w = 100\):

\[
w = \lambda s_0^{(\alpha \rightarrow \alpha)} . \lambda s_1^{(\alpha \rightarrow \alpha)} . \lambda x^{\alpha} . (s_1 (s_0 (s_0 x))).
\]
we have in DLAL as type for the iterator:

\[ \text{iter}_A = \lambda f x n. \ (n \ f \ x) : (A \to A) \Rightarrow \mathbb{S}A \to N \to \mathbb{S}A \]

\( \text{(iter}_A \ F \ t \ n \) \to (F (F \ldots (F \ t) \ldots)) \quad (n \text{ times}) \)

**example:**
add : \( N \to N \to N, \quad (\text{add} \ 2) : N \to N \)
Double = \( \lambda n.(\text{iter}_N (\text{add} \ 2) \ 0) \ n : N \to \mathbb{S}N \)
Double cannot be iterated : 1st exponentiation scheme is ruled out.
Some examples of types in DLAL

⊢ append : \( W \rightarrow W \rightarrow W \)

⊢ length : \( W \rightarrow N \)

⊢ mult : \( N \Rightarrow N \rightarrow \&N \)

Functions on binary lists are represented by types \( W \rightarrow \&^k W \), for \( k \in \mathbb{N} \).

With a type \( L(A) \) for lists over \( A \), we get for the insertion sort:

⊢ insert : \( \&A \rightarrow L(A) \rightarrow L(A) \)

⊢ sort : \( L(\&A) \rightarrow \&L(A) \)
Intuitionistic Proof-Nets for DLAL terms

\((A \Rightarrow B)^* = !A^* \land B^*\)

*depth* of an edge: number of boxes it is contained in.
*depth* of proof-net: maximal depth of its edges.
Proof-net reduction

During normalization: the depth of an edge is unchanged. This is the \textit{stratification property}.
Example

\[ M = (\lambda f.(f (f \, x)))((\lambda h.h) \, g) \]
\[ M = (\lambda f^{\alpha \to \alpha}.(f \, (f \, x^{\alpha}))((\lambda h^{\alpha \to \alpha}.h) \, g^{\alpha \to \alpha}) : \alpha \]
Example of type derivation in DLAL

\[ M = (\lambda f. (f (f \ x)))((\lambda h. h) \ g) \]

\[
\vdash ; f_2 : \alpha \multimap \alpha, x : \alpha \vdash (f_2 \ x) : \alpha \quad ; f_1 : \alpha \multimap \alpha \vdash f_1 : \alpha \multimap \alpha
\]

\[
(\S \ i) \frac{f_1 : \beta, f_2 : \beta, x : \alpha \vdash (f_1 (f_2 \ x)) : \alpha}{f_1 : \beta, f_2 : \beta ; x : \S \alpha \vdash (f_1 (f_2 \ x)) : \S \alpha}
\]

\[
(\Rightarrow \ e) \frac{x : \S \alpha \vdash \lambda f. (f (f \ x)) : \beta \Rightarrow \S \alpha}{g : \beta ; x : \S \alpha \vdash (\lambda f. (f (f \ x)))(\lambda h. h) \ g : \S \alpha}
\]

where \( \beta = \alpha \multimap \alpha \).
Example

\[ M = (\lambda f. (f\ (f\ x)))((\lambda h. h)\ g) \]
Boxes

Boxes in linear logic proof-nets have 2 characteristics:

1. *logically*:
   they correspond to a sequentiality information on the graph.

2. *dynamically*:
   they delimitate portions of the graph that can be duplicated.

Note that in LAL, (2) is not relevant for § boxes, since they cannot be duplicated.

In Light logics, the crucial point of boxes is that they allow to define some *invariants* of the dynamics of the graphs:

- depth of edges,
- quantitative invariant: the number of inputs of duplicable boxes (! boxes) is at most one.
DLAL: complexity bounds

DLAL satisfies subject-reduction: if $\Gamma; \Delta \vdash t : A$ and $t \rightarrow t'$, then $\Gamma; \Delta \vdash t' : A$.

**Theorem 15 (Strong PTIME bound)**  *If $t$ is typable in DLAL with a derivation of depth $d$, then any $\beta$ reduction of $t$ can be performed in time $O((d + 1) \cdot |t|^{2d+1})$.*

Remarks:

- one in fact shows a bound $O((d + 1) \cdot |t|^{2d})$ on the number of $\beta$-steps and then uses the fact that the cost of each step is here bounded;
- this bound holds for any reduction strategy;
- in particular, if $\vdash t : W \rightarrow \text{ } \overline{\delta}^k W$ then $t$ is PTIME.
In practice we often need to change the type of arguments, for instance from $\mathcal{N}N$ to $N$, or from non-linear to linear position.

This is possible for data arguments (e.g. $N$ or $W$), thanks to coercions.

In DLAL one can define terms with the following types:

- $\text{coer}_1 : N \rightarrow \mathcal{N}N$, s.t. $(\text{coer}_1 n) \rightarrow n$
- $\text{coer}_2 : \mathcal{N}(N \Rightarrow A) \rightarrow (N \rightarrow \mathcal{N}A)$, s.t. $(\text{coer}_2 t n) \rightarrow (t n)$

Similarly, there exists a term contracting arguments by:

- $\text{cont} : \mathcal{N}(N \Rightarrow N \rightarrow A) \rightarrow (N \rightarrow \mathcal{N}A)$ with:

  $(\text{cont} t n) \rightarrow (t n n)$
Proposition 16  If $P \in \mathbb{N}[X]$, then there exists a term $t_P$ representing $P$ and an integer $k$ such that: $\vdash_{DLAL} t_P : N \multimap \mathcal{S}^k N$.

Using this Proposition, one can simulate PTIME Turing machines by DLAL typed terms and obtain:

**Theorem 17 (PTIME Completeness)**  For any polynomial time function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, there exists a term $t$ representing it and typable in DLAL with a type $W \multimap \mathcal{S}^k W$, for a certain integer $k \in \mathbb{N}$.

However DLAL (or LAL) is not very expressive from an intensional point of view: some simple PTIME lambda terms might not be typable.
We show how to embed a variant of BC into DLAL, following Murawski/Ong [MO04] (see also Tranquilli’s Master thesis, 2005)

\[ f(\vec{x}; \vec{y}) \] will be represented by

\[ ; \vec{x} : W, \vec{y} : \frac{\gamma^k}{\gamma} W \vdash_{DLAL} t : \frac{\gamma^k}{\gamma} W, \text{ for some } k \]

Note that:

- in \( f(\vec{x}; \vec{y}) \), \( \vec{x} \) stands for \( x_1, \ldots, x_n \),
- in the DLAL judgement, \( \vec{x} : W \) stands for \( x_1 : W, \ldots, x_n : W \).

**Lemma 18** \( i \geq k \), then there exists \( u \) representing the same function as \( t \) and such that

\[ ; \vec{x} : W, \vec{y} : \frac{\gamma^i}{\gamma} W \vdash_{DLAL} u : \frac{\gamma^i}{\gamma} W. \]
The algebra BC$^-$

In the DLAL judgements given, observe that normal variables (of type $\mathcal{W}$) can be contracted, but that safe ones (of type $\mathcal{S}^k\mathcal{W}$) cannot. Fragment of BC with non-contractible safe variables: BC$^-$. 

BC$^-$ is defined by:

recursion: \[ f(\epsilon, \vec{x}; \vec{y}) = h(\vec{x}; \vec{y}) \]
\[ f(a.z, \vec{x}; \vec{y}) = g(a, z, \vec{x}; f(z, \vec{x}; \vec{y})) \]

composition: \[ f(\vec{x}; \vec{y}) = g(h_1(\vec{x};), \ldots, h_m(\vec{x};); h_{m+1}(\vec{x}; \vec{y}_1), \ldots, h_k(\vec{x}; \vec{y}_i)) \]
where $\vec{y}_1, \ldots, \vec{y}_i$ is a partition of $\vec{y}$.

basic functions: as in BC.
BC$^-$ inside DLAL

**Proposition 19** [MO04] The algebra BC$^-$ can be embedded in DLAL. However BC$^-$ is (presumably) not complete for PTIME.

But one can consider extra constructs to enrich the algebra:

- a form of definition by cases

  $$\text{case}_K(;u)[p_1 : f_1 | \ldots | p_m : f_m | \text{else} : f_{m+1}]$$

- a restricted recursion on safe arguments: $\sigma\text{rec}$

  $$f(\vec{x}; \epsilon, \vec{y}) = h(\vec{x}; \vec{y})$$
  $$f(\vec{x}; a.z, \vec{y}) = \text{step}_i( ; f(\vec{x}; z, \vec{y}))$$

  with $\text{step}_i$ of a specific form (permutation)

See [MO04] for complete definitions.
Let $BC^\pm$ be the algebra defined as $BC^-$ but with these 2 new constructs ($\text{case}_K$ and $\sigma\text{rec}$).

**Theorem 20** [MO04]

1. $BC^\pm$ is sound and extensionally complete for PTIME,
2. $BC^\pm$ can be embedded in DLAL.

The second point comes from the fact that the new constructs can be represented in DLAL.
Other directions in linear logic for ICC

- Various light logics:
  - for PTIME: BLL [GSS92], SLL [Laf04]
  - for Elementary time: ELL [Gir98]

- realizability and denotational semantics: [DL05], [MO04] (games), [Bai04], [LTdF08]

- geometry of interaction techniques: [DL06], [BCDL07]
Conclusion of Part III

- Other type systems derived from linear logic characterize different complexity classes: LOGSPACE (Schöpp 07), PSPACE [GMR08]
- The relations between the different light logics and possible ways to unify them are still to explore
- Type inference can be performed by constraints solving. In particular type inference for DLAL can be done in PTIME [ABT07] (for a system F term). The system is thus tractable.
- However this approach has a modest intensional expressivity. Could it be combined to other ICC systems for more flexibility?
Some other directions in ICC

- Higher-order type systems for safe recursion (Hofmann 00, Bellantoni-Niggl-Schwichtenberg 00)
- Non-size-increasing computation: linear type system with a basic type $\diamond$ (Hofmann 03)
- Functional languages with restricted operations (no `cons`): characterizations of PTIME, LOGSPACE (Jones 99)
Conclusion on ICC

ICC has provided:

1. some machine-independent characterizations of complexity classes of functions,

2. some criteria for verifying statically that a program, admits a certain complexity bound.

For (1) it would be useful to relate together the various approaches.

For (2): some recent works try to take advantage of techniques coming from termination analysis (e.g. size-change-principle) and possibly from abstraction techniques.
<table>
<thead>
<tr>
<th>Reference</th>
<th>Details</th>
</tr>
</thead>
</table>


### Bibliography

<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
</table>


<table>
<thead>
<tr>
<th>Reference</th>
<th>Citation</th>
</tr>
</thead>
</table>


